Wednesday, December 30, 2015
1:20 AM

NEURAL NETWORKS

by Christos Stergiou and Dimitrios Siganos

Abstract

This report is an introduction to Artificial Neural Networks. The various types of neural networks are explained and demonstrated, applications of neural networks like ANNs in medicine are described, and a detailed historical background is provided. The connection between the artificial and the real thing is al investigated and explained. Finally, the mathematical models involved are presented and demonstrated.

Contents:

1. Introduction to neural networks

1.1 What is a Neural Network?

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, proces information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly

specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well.

1.2 Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least major setback and several eras.

Many importand advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few reserchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

For a more detailed description of the history click here

The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits. But the technology available at tha time did not allow them to do too much.

1.3 Why use neural networks?

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that ar too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyse. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which tak advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

1.4 Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. B computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements(neurones) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific tas The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is tha because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understa These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmi approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

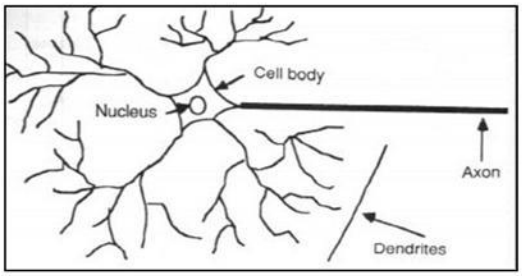
Neural networks do not perform miracles. But if used sensibly they can produce some amazing results.

Back to Contents

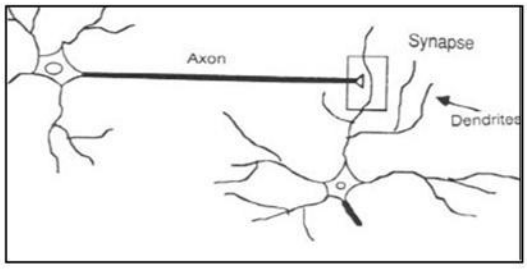
2. Human and Artificial Neurones - investigating the similarities

2.1 How the Human Brain Learns?

Much is still unknown about how the brain trains itself to process information, so theories abound. In the human brain, a typical neuron collects signals from others through a host of fine structures called *dendrites*. The neuron sends out spikes of electrical activity through a long, thin stand known as an *axon*, which splits into thousands of branches. At the end of each branch, a structure called a *synapse* converts the activity from the axon into electrical effects that inhibit excite activity from the axon into electrical effects that inhibit or excite activity in the connected neurones. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.



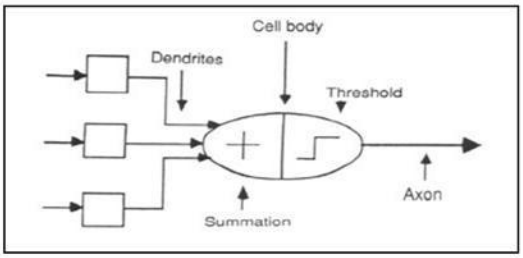
Components of a neuron



The synapse

2.2 From Human Neurones to Artificial Neurones

We conduct these neural networks by first trying to deduce the essential features of neurones and their interconnections. We then typically program a comput to simulate these features. However because our knowledge of neurones is incomplete and our computing power is limited, our models are necessarily gross idealisations of real networks of neurones.



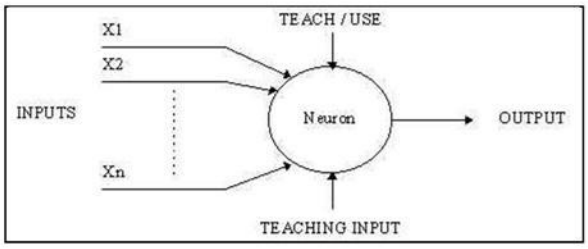
The neuron model

Back to Contents

3. An engineering approach

3.1 A simple neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the traini mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associ output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire o not.



A simple neuron

3.2 Firing rules

The firing rule is an important concept in neural networks and accounts for their high flexibility. A firing rule determines how one calculates whether a neuro should fire for any input pattern. It relates to all the input patterns, not only the ones on which the node was trained.

A simple firing rule can be implemented by using Hamming distance technique. The rule goes as follows:

> Take a collection of training patterns for a node, some of which cause it to fire (the 1-taught set of patterns) and others which prevent it from doing so (the 0-taught set). Then the patterns not in the collection cause the node to fire if, on comparison , they have more input elements in common with the 'nearest' pattern in the 1-taught set than with the 'nearest' pattern in the 0-taught set. If there is a tie, then the pattern remains in the undefined state.

For example, a 3-input neuron is taught to output 1 when the input (X1,X2 and X3) is 111 or 101 and to output 0 when the input is 000 or 001. Then, before

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0/1	0/1	0/1	1	0/1	1

As an example of the way the firing rule is applied, take the pattern 010. It differs from 000 in 1 element, from 001 in 2 elements, from 101 in 3 elements and from 111 in 2 elements. Therefore, the 'nearest' pattern is 000 which belongs in the 0-taught set. Thus the firing rule requires that the neuron should not fire w the input is 001. On the other hand, 011 is equally distant from two taught patterns that have different outputs and thus the output stays undefined (0/1).

By applying the firing in every column the following truth table is obtained;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0	0/1	0/1	1	1	1

The difference between the two truth tables is called the *generalisation of the neuron.* Therefore the firing rule gives the neuron a sense of similarity and enal it to respond 'sensibly' to patterns not seen during training.

3.3 Pattern Recognition - an example

An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward (figure 1) neural networ that has been trained accordingly. During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the i pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is gi as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

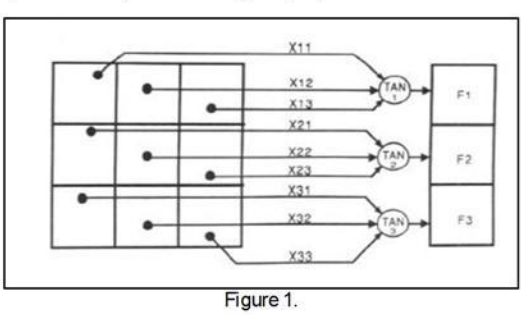


Figure 1.

For example:
The network of figure 1 is trained to recognise the patterns T and H. The associated patterns are all black and all white respectively as shown below.

If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurones after generalisation are;

X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Top neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

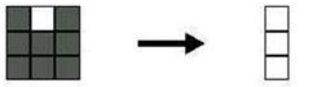
Middle neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1

Bottom neuron

From the tables it can be seen the following associasions can be extracted:

In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern.



Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.

Here, the top row is 2 errors away from the a T and 3 from an H. So the top output is black. The middle row is 1 error away from both T and H so the output random. The bottom row is 1 error away from T and 2 away from H. Therefore the output is black. The total output of the network is still in favour of the T shape.

3.4 A more complicated neuron

The previous neuron doesn't do anything that conventional conventional computers don't do already. A more sophisticated neuron (figure 2) is the McCulloc and Pitts model (MCP). The difference from the previous model is that the inputs are 'weighted', the effect that each input has at decision making is depende the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs then added together and if they exceed a pre-set threshold value, the neuron fires. In any other case the neuron does not fire.

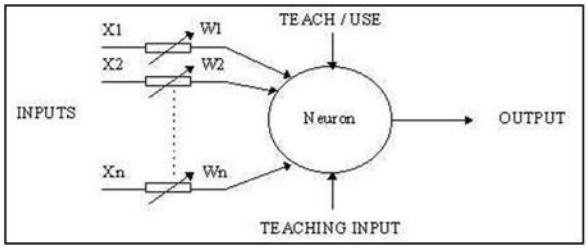


Figure 2. An MCP neuron

In mathematical terms, the neuron fires if and only if;

$$X_1W_1 + X_2W_2 + X_3W_3 + \ldots > T$$

The addition of input weights and of the threshold makes this neuron a very flexible and powerful one. The MCP neuron has the ability to adapt to a particul situation by changing its weights and/or threshold. Various algorithms exist that cause the neuron to 'adapt'; the most used ones are the Delta rule and the bac error propagation. The former is used in feed-forward networks and the latter in feedback networks.

Back to Contents

4 Architecture of neural networks

4.1 Feed-forward networks

Feed-forward ANNs (figure 1) allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not af that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition This type of organisation is also referred to as bottom-up or top-down.

4.2 Feedback networks

Feedback networks (figure 1) can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and ca get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations.

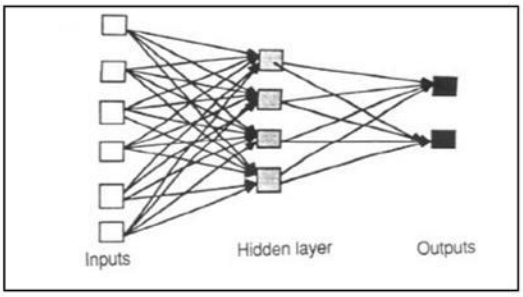


Figure 4.1 An example of a simple feedforward network

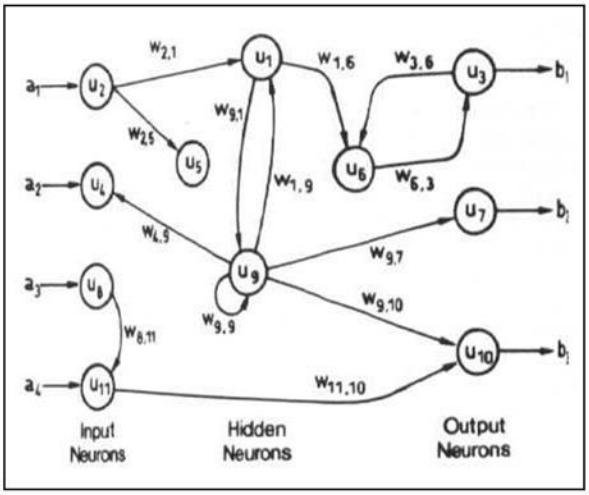


Figure 4.2 An example of a complicated network

4.3 Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units: a layer of "**input**" units is connected to a layer of "**hidden**" units which is connected to a layer of "**output**" units. (see Figure 4.1)

- The activity of the input units represents the raw information that is fed into the network.
- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish single-layer and multi-layer architectures. The single-layer organisation, in which all units are connected to one another, constitutes the r general case and is of more potential computational power than hierarchically structured multi-layer organisations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

4.4 Perceptrons

The most influential work on neural nets in the 60's went under the heading of 'perceptrons' a term coined by Frank Rosenblatt. The perceptron (figure 4.4) tu out to be an MCP model (neuron with weighted inputs) with some additional, fixed, pre--processing. Units labelled A1, A2, Aj , Ap are called association u and their task is to extract specific, localised featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They we mainly used in pattern recognition even though their capabilities extended a lot more.

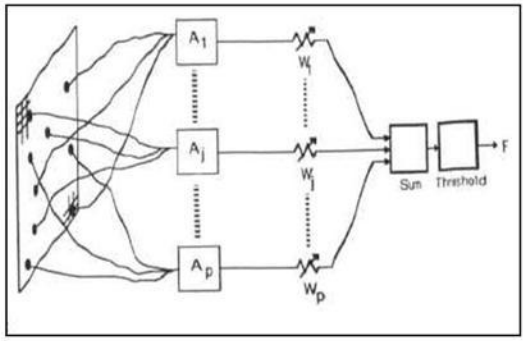


Figure 4.4

In 1969 Minsky and Papert wrote a book in which they described the limitations of single layer Perceptrons. The impact that the book had was tremendous a caused a lot of neural network researchers to loose their interest. The book was very well written and showed mathematically that *single layer* perceptrons co

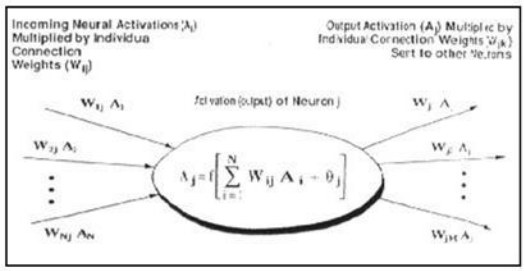
Back to Contents

5. The Learning Process

The memorisation of patterns and the subsequent response of the network can be categorised into two general paradigms:

- **associative mapping** in which the network learns to produce a particular pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associtive mapping can generally be broken down into two mechanisms:
  - *auto-association*: an input pattern is associated with itself and the states of input and output units coincide. This is used to provide pattern completition, ie to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network actually stores pairs of patterns building an association between two sets of patterns.
  - *hetero-association*: is related to two recall mechanisms:
    - *nearest-neighbour* recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and
    - *interpolative* recall, where the output pattern is a similarity dependent interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, ie when there is a fixed set of categories into which the input patterns are to be classified.
- **regularity detection** in which units learn to respond to particular properties of the input patterns. Whereas in asssociative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network posseses knowledge which is contained in the values of the connections weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights.



Information is stored in the weight matrix W of a neural network. Learning is the determination of the weights. Following the way learning is performed, we distinguish two major categories of neural networks:

- **fixed networks** in which the weights cannot be changed, ie dW/dt=0. In such networks, the weights are fixed a priori according to the problem to solve.
- **adaptive networks** which are able to change their weights, ie dW/dt not= 0.

All learning methods used for adaptive neural networks can be classified into two major categories:

- **Supervised learning** which incorporates an external teacher, so that each output unit is told what its desired response to input signals ought to be. During the learning process global information may be required. Paradigms of supervised learning include error-correction learning, reinforcement learning and stochastic learning.
  An important issue conserning supervised learning is the problem of error convergence, ie the minimisation of error between the desired and computed unit values. The aim is to determine a set of weights which minimises the error. One well-known method, which is common to many learning paradigms is the least mean square (LMS) convergence.
- **Unsupervised learning** uses no external teacher and is based upon only local information. It is also referred to as self-organisation, in the sense that it self-organises data presented to the network and detects their emergent collective properties. Paradigms of unsupervised learning are Hebbian lerning and competitive learning.
  Ano2.2 From Human Neurones to Artificial Neuronesther aspect of learning concerns the distinction or not of a seperate phase, during which the network is trained, and a subsequent operation phase. We say that a neural network learns off-line if the learning phase and the operation phase are distinct. A neural network learns on-line if it learns and operates at the same time. Usually, supervised learning is performed off-line, whereas usupervised learning is performed on-line.

5.1 Transfer Function

The behaviour of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- linear (or ramp)
- threshold
- sigmoid

For **linear units**, the output activity is proportional to the total weighted output.

For **threshold units**, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.

For **sigmoid units**, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurones than do lin or threshold units, but all three must be considered rough approximations.

To make a neural network that performs some specific task, we must choose how the units are connected to one another (see figure 4.1), and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strengt the influence.

We can teach a three-layer network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

5.2 An Example to illustrate the above teaching procedure:

Assume that we want a network to recognise hand-written digits. We might use an array of, say, 256 sensors, each recording the presence or absence of ink i small area of a single digit. The network would therefore need 256 input units (one for each sensor), 10 output units (one for each kind of digit) and a numbe hidden units.

For each kind of digit recorded by the sensors, the network should produce high activity in the appropriate output unit and low activity in the other output un

To train the network, we present an image of a digit and compare the actual activity of the 10 output units with the desired activity. We then calculate the err which is defined as the square of the difference between the actual and the desired activities. Next we change the weight of each connection so as to reduce t error.We repeat this training process for many different images of each different images of each kind of digit until the network classifies every image correct

To implement this procedure we need to calculate the error derivative for the weight (EW) in order to change the weight by an amount that is proportional to rate at which the error changes as the weight is changed. One way to calculate the EW is to perturb a weight slightly and observe how the error changes. But method is inefficient because it requires a separate perturbation for each of the many weights.

Another way to calculate the EW is to use the Back-propagation algorithm which is described below, and has become nowadays one of the most important to for training neural networks. It was developed independently by two teams, one (Fogelman-Soulie, Gallinari and Le Cun) in France, the other (Rumelhart, Hinton and Williams) in U.S.

5.3 The Back-Propagation Algorithm

In order to train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and t actual output is reduced. This process requires that the neural network compute the error derivative of the weights (**EW**). In other words, it must calculate ho the error changes as each weight is increased or decreased slightly. The back propagation algorithm is the most widely used method for determining the **EW**.

The back-propagation algorithm is easiest to understand if all the units in the network are linear. The algorithm computes each **EW** by first computing the **EA** the rate at which the error changes as the activity level of a unit is changed. For output units, the **EA** is simply the difference between the actual and the desir output. To compute the **EA** for a hidden unit in the layer just before the output layer, we first identify all the weights between that hidden unit and the output units to which it is connected. We then multiply those weights by the **EA**s of those output units and add the products. This sum equals the **EA** for the chosen hidden unit. After calculating all the **EA**s in the hidden layer just before the output layer, we can compute in like fashion the **EA**s for other layers, moving fr layer to layer in a direction opposite to the way activities propagate through the network. This is what gives back propagation its name. Once the **EA** has bee computed for a unit, it is straight forward to compute the **EW** for each incoming connection of the unit. The **EW** is the product of the EA and the activity through the incoming connection.

Note that for non-linear units, (see Appendix C) the back-propagation algorithm includes an extra step. Before back-propagating, the **EA** must be converted i the **EI**, the rate at which the error changes as the total input received by a unit is changed.

Back to Contents

6. Applications of neural networks

6.1 Neural Networks in Practice

Given this description of neural networks and how they work, what real world applications are they suited for? Neural networks have broad applicability to r world business problems. In fact, they have already been successfully applied in many industries.

Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs including:

- sales forecasting
- industrial process control
- customer research
- data validation
- risk management
- target marketing

But to give you some more specific examples; ANN are also used in the following specific paradigms: recognition of speakers in communications; diagnosis hepatitis; recovery of telecommunications from faulty software; interpretation of multimeaning Chinese words; undersea mine detection; texture analysis; thr dimensional object recognition; hand-written word recognition; and facial recognition.

6.2 Neural networks in medicine

Artificial Neural Networks (ANN) are currently a 'hot' research area in medicine and it is believed that they will receive extensive application to biomedical systems in the next few years. At the moment, the research is mostly on modelling parts of the human body and recognising diseases from various scans (e.g cardiograms, CAT scans, ultrasonic scans, etc.).

Neural networks are ideal in recognising diseases using scans since there is no need to provide a specific algorithm on how to identify the disease. Neural networks learn by example so the details of how to recognise the disease are not needed. What is needed is a set of examples that are representative of all the variations of the disease. The quantity of examples is not as important as the 'quantity'. The examples need to be selected very carefully if the system is to perform reliably and efficiently.

6.2.1 Modelling and Diagnosing the Cardiovascular System

Neural Networks are used experimentally to model the human cardiovascular system. Diagnosis can be achieved by building a model of the cardiovascular system of an individual and comparing it with the real time physiological measurements taken from the patient. If this routine is carried out regularly, potenti harmful medical conditions can be detected at an early stage and thus make the process of combating the disease much easier.

A model of an individual's cardiovascular system must mimic the relationship among physiological variables (i.e., heart rate, systolic and diastolic blood pressures, and breathing rate) at different physical activity levels. If a model is adapted to an individual, then it becomes a model of the physical condition of individual. The simulator will have to be able to adapt to the features of any individual without the supervision of an expert. This calls for a neural network.

Another reason that justifies the use of ANN technology, is the ability of ANNs to provide sensor fusion which is the combining of values from several diffe sensors. Sensor fusion enables the ANNs to learn complex relationships among the individual sensor values, which would otherwise be lost if the values wer individually analysed. In medical modelling and diagnosis, this implies that even though each sensor in a set may be sensitive only to a specific physiologica variable, ANNs are capable of detecting complex medical conditions by fusing the data from the individual biomedical sensors.

6.2.2 Electronic noses

ANNs are used experimentally to implement electronic noses. Electronic noses have several potential applications in telemedicine. Telemedicine is the practi of medicine over long distances via a communication link. The electronic nose would identify odours in the remote surgical environment. These identified odours would then be electronically transmitted to another site where an door generation system would recreate them. Because the sense of smell can be an important sense to the surgeon, telesmell would enhance telepresent surgery.
For more information on telemedicine and telepresent surgery click here.

6.2.3 Instant Physician

An application developed in the mid-1980s called the "instant physician" trained an autoassociative memory neural network to store a large number of medic records, each of which includes information on symptoms, diagnosis, and treatment for a particular case. After training, the net can be presented with input consisting of a set of symptoms; it will then find the full stored pattern that represents the "best" diagnosis and treatment.

6.3 Neural Networks in business

Business is a diverted field with several general areas of specialisation such as accounting or financial analysis. Almost any neural network application would into one business area or financial analysis.
There is some potential for using neural networks for business purposes, including resource allocation and scheduling. There is also a strong potential for usi neural networks for database mining, that is, searching for patterns implicit within the explicitly stored information in databases. Most of the funded work in area is classified as proprietary. Thus, it is not possible to report on the full extent of the work going on. Most work is applying neural networks, such as the Hopfield-Tank network for optimization and scheduling.

6.3.1 Marketing

There is a marketing application which has been integrated with a neural network system. The Airline Marketing Tactician (a trademark abbreviated as AMT a computer system made of various intelligent technologies including expert systems. A feedforward neural network is integrated with the AMT and was trai using back-propagation to assist the marketing control of airline seat allocations. The adaptive neural approach was amenable to rule expression. Additionaly application's environment changed rapidly and constantly, which required a continuously adaptive solution. The system is used to monitor and recommend

While it is significant that neural networks have been applied to this problem, it is also important to see that this intelligent technology can be integrated with expert systems and other approaches to make a functional system. Neural networks were used to discover the influence of undefined interactions by the vario variables. While these interactions were not defined, they were used by the neural system to develop useful conclusions. It is also noteworthy to see that neur networks can influence the bottom line.

6.3.2 Credit Evaluation

The HNC company, founded by Robert Hecht-Nielsen, has developed several neural network applications. One of them is the Credit Scoring system which increase the profitability of the existing model up to 27%. The HNC neural systems were also applied to mortgage screening. A neural network automated mortgage insurance underwritting system was developed by the Nestor Company. This system was trained with 5048 applications of which 2597 were certifi The data related to property and borrower qualifications. In a conservative mode the system agreed on the underwritters on 97% of the cases. In the liberal m the system agreed 84% of the cases. This is system run on an Apollo DN3000 and used 250K memory while processing a case file in approximately 1 sec.

Back to Contents

7. Conclusion

The computing world has a lot to gain fron neural networks. Their ability to learn by example makes them very flexible and powerful. Furthermore there is n need to devise an algorithm in order to perform a specific task; i.e. there is no need to understand the internal mechanisms of that task. They are also very we suited for real time systems because of their fast responseand computational times which are due to their parallel architecture.

Neural networks also contribute to other areas of research such as neurology and psychology. They are regularly used to model parts of living organisms and investigate the internal mechanisms of the brain.

Perhaps the most exciting aspect of neural networks is the possibility that some day 'consious' networks might be produced. There is a number of scientists arguing that conciousness is a 'mechanical' property and that 'consious' neural networks are a realistic possibility.

Finally, I would like to state that even though neural networks have a huge potential we will only get the best of them when they are intergrated with comput AI, fuzzy logic and related subjects.

Back to Contents

References:

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html
3. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
4. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html
5. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht
6. Neural Networks by Eric Davalo and Patrick Naim
7. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
8. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
9. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab. Neural Networks, Eric Davalo and Patrick Naim
10. Assimov, I (1984, 1950), Robot, Ballatine, New York.
11. Electronic Noses for Telemedicine
    http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html
12. Pattern Recognition of Pathology Images
    http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html

Back to Contents

Appendix A - Historical background in detail

The history of neural networks that was described above can be divided into several periods:

1. **First Attempts**: There were some initial simulations using formal logic. McCulloch and Pitts (1943) developed models of neural networks based on th understanding of neurology. These models made several assumptions about how neurons worked. Their networks were based on simple neurons which were considered to be binary devices with fixed thresholds. The results of their model were simple logic functions such as "a or b" and "a and b". Anot attempt was by using computer simulations. Two groups (Farley and Clark, 1954; Rochester, Holland, Haibit and Duda, 1956). The first group (IBM reserchers) maintained closed contact with neuroscientists at McGill University. So whenever their models did not work, they consulted the neuroscientists. This interaction established a multidiscilinary trend which continues to the present day.
2. **Promising & Emerging Technology:** Not only was neroscience influential in the development of neural networks, but psychologists and engineers al contributed to the progress of neural network simulations. Rosenblatt (1958) stirred considerable interest and activity in the field when he designed and developed the Perceptron. The Perceptron had three layers with the middle layer known as the association layer.This system could learn to connect or

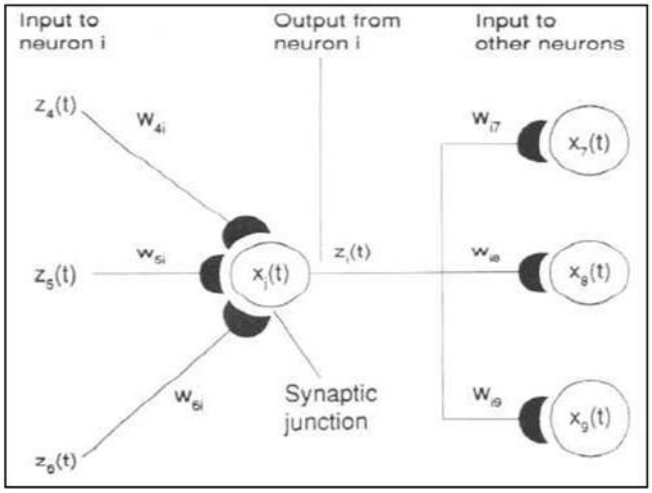
Another system was the ADALINE (ADAptive LInear Element) which was developed in 1960 by Widrow and Hoff (of Stanford University). The ADALINE was an analogue electronic device made from simple components. The method used for learning was different to that of the Perceptron, it employed the Least-Mean-Squares (LMS) learning rule.

3. **Period of Frustration & Disrepute:** In 1969 Minsky and Papert wrote a book in which they generalised the limitations of single layer Perceptrons to multilayered systems. In the book they said: "...our intuitive judgment that the extension (to multilayer systems) is sterile". The significant result of the book was to eliminate funding for research with neural network simulations. The conclusions supported the disenhantment of reserchers in the field. A result, considerable prejudice against this field was activated.
4. **Innovation:** Although public interest and available funding were minimal, several researchers continued working to develop neuromorphically based computaional methods for problems such as pattern recognition.
   During this period several paradigms were generated which modern work continues to enhance.Grossberg's (Steve Grossberg and Gail Carpenter in 19 influence founded a school of thought which explores resonating algorithms. They developed the ART (Adaptive Resonance Theory) networks based o biologically plausible models. Anderson and Kohonen developed associative techniques independent of each other. Klopf (A. Henry Klopf) in 1972, developed a basis for learning in artificial neurons based on a biological principle for neuronal learning called heterostasis.
   Werbos (Paul Werbos 1974) developed and used the back-propagation learning method, however several years passed before this approach was popularized. Back-propagation nets are probably the most well known and widely applied of the neural networks today. In essence, the back-propagati net. is a Perceptron with multiple layers, a different thershold function in the artificial neuron, and a more robust and capable learning rule.
   Amari (A. Shun-Ichi 1967) was involved with theoretical developments: he published a paper which established a mathematical theory for a learning b (error-correction method) dealing with adaptive patern classification. While Fukushima (F. Kunihiko) developed a step wise trained multilayered neura network for interpretation of handwritten characters. The original network was published in 1975 and was called the Cognitron.
5. **Re-Emergence:** Progress during the late 1970s and early 1980s was important to the re-emergence on interest in the neural network field. Several fact influenced this movement. For example, comprehensive books and conferences provided a forum for people in diverse fields with specialized technica languages, and the response to conferences and publications was quite positive. The news media picked up on the increased activity and tutorials helpe disseminate the technology. Academic programs appeared and courses were inroduced at most major Universities (in US and Europe). Attention is nov focused on funding levels throughout Europe, Japan and the US and as this funding becomes available, several new commercial with applications in industry and finacial institutions are emerging.
6. **Today:** Significant progress has been made in the field of neural networks-enough to attract a great deal of attention and fund further research. Advancement beyond current commercial applications appears to be possible, and research is advancing the field on many fronts. Neurally based chips emerging and applications to complex problems developing. Clearly, today is a period of transition for neural network technology.

Back to Contents

Appendix B - The back-propagation Algorithm - a mathematical approach

Units are connected to one another. Connections correspond to the edges of the underlying directed graph. There is a real number associated with each connection, which is called the weight of the connection. We denote by W_{ij} the weight of the connection from unit u_i to unit u_j. It is then convenient to repres the pattern of connectivity in the network by a weight matrix W whose elements are the weights W_{ij}. Two types of connection are usually distinguished: excitatory and inhibitory. A positive weight represents an excitatory connection whereas a negative weight represents an inhibitory connection. The pattern o connectivity characterises the architecture of the network.



A unit in the output layer determines its activity by following a two step procedure.

- First, it computes the total weighted input x_j, using the formula:

$$X_j = \sum_i y_i W_{ij}$$

  where y_i is the activity level of the jth unit in the previous layer and W_{ij} is the weight of the connection between the ith and the jth unit.

- Next, the unit calculates the activity y_j using some function of the total weighted input. Typically we use the sigmoid function:

$$y_j = \frac{1}{1 + e^{-x_j}}$$

Once the activities of all output units have been determined, the network computes the error E, which is defined by the expression:

$$E = \frac{1}{2}\sum_i (y_i - d_i)^2$$

where y_j is the activity level of the jth unit in the top layer and d_j is the desired output of the jth unit.

The back-propagation algorithm consists of four steps:

1. Compute how fast the error changes as the activity of an output unit is changed. This error derivative (EA) is the difference between the actual and the des activity.

$$EA_j = \frac{\partial E}{\partial y_j} = y_j - d_j$$

2. Compute how fast the error changes as the total input received by an output unit is changed. This quantity (EI) is the answer from step 1 multiplied by the at which the output of a unit changes as its total input is changed.

$$EI_j = \frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \times \frac{dy_j}{dx_j} = EA_j y_j (1 - y_j)$$

3. Compute how fast the error changes as a weight on the connection into an output unit is changed. This quantity (EW) is the answer from step 2 multiplied the activity level of the unit from which the connection emanates.

$$EW_{ij} = \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial W_{ij}} = EI_j y_i$$

4. Compute how fast the error changes as the activity of a unit in the previous layer is changed. This crucial step allows back propagation to be applied to multilayer networks. When the activity of a unit in the previous layer changes, it affects the activites of all the output units to which it is connected. So to compute the overall effect on the error, we add together all these seperate effects on output units. But each effect is simple to calculate. It is the answer in ste multiplied by the weight on the connection to that output unit.

$$EA_i = \frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial y_i} = \sum_j EI_j W_{ij}$$

By using steps 2 and 4, we can convert the EAs of one layer of units into EAs for the previous layer. This procedure can be repeated to get the EAs for as ma previous layers as desired. Once we know the EA of a unit, we can use steps 2 and 3 to compute the EWs on its incoming connections.

Back to Contents

Appendix C - References used throughout the review

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html
3. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht
4. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
5. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html
6. Electronic Noses for Telemedicine
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html
7. An Introduction to Computing with Neural Nets (Richard P. Lipmann, IEEE ASSP Magazine, April 1987)
8. Pattern Recognition of Pathology Images
   http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html
9. Developments in autonomous vehicle navigation. Stefan Neuber, Jos Nijhuis, Lambert Spaanenburg. Institut fur Mikroelektronik Stuttgart, Allmandrin 30A, 7000 Stuttgart-80
10. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
11. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab.
12. Neural Networks, Eric Davalo and Patrick Naim.
13. Assimov, I (1984, 1950), Robot, Ballatine, New York.
14. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
15. Alkon, D.L 1989, Memory Storage and Neural Systems, Scientific American, July, 42-50
16. Minsky and Papert (1969) Perceptrons, An introduction to computational geometry, MIT press, expanded edition.
17. Neural computers, NATO ASI series, Editors: Rolf Eckmiller Christoph v. d. Malsburg

Back to Contents

Wednesday, December 30, 2015
1:26 AM

Artificial Neural Networks: A Tutorial

Anil K. Jain
Michigan State University

Jianchang Mao
K.M. Mohiuddin
IBM Almaden Research Center

These massively parallel systems with large numbers of interconnected simple processors may solve a variety of challenging computational problems. This tutorial provides the background and the basics.

Numerous advances have been made in developing intelligent systems, some inspired by biological neural networks. Researchers from many scientific disciplines are designing artificial neural networks (ANNs) to solve a variety of problems in pattern recognition, prediction, optimization, associative memory, and control (see the "Challenging problems" sidebar).

Conventional approaches have been proposed for solving these problems. Although successful applications can be found in certain well-constrained environments, none is flexible enough to perform well outside its domain. ANNs provide exciting alternatives, and many applications could benefit from using them.[1-3]

This article is for those readers with little or no knowledge of ANNs to help them understand the other articles in this issue of *Computer*. We discuss the motivations behind the development of ANNs, describe the basic biological neuron and the artificial computational model, outline network architectures and learning processes, and present some of the most commonly used ANN models. We conclude with character recognition, a successful ANN application.

WHY ARTIFICIAL NEURAL NETWORKS?

The long course of evolution has given the human brain many desirable characteristics not present in von Neumann or modern parallel computers. These include

- massive parallelism,
- distributed representation and computation,
- learning ability,
- generalization ability,
- adaptivity,
- inherent contextual information processing,
- fault tolerance, and
- low energy consumption.

It is hoped that devices based on biological neural networks will possess some of these desirable characteristics.

Modern digital computers outperform humans in the domain of numeric computation and related symbol manipulation. However, humans can effortlessly solve complex perceptual problems (like recognizing a man in a crowd from a mere glimpse of his face) at such a high speed and extent as to dwarf the world's fastest computer. Why is there such a remarkable difference in their performance? The biological neural system architecture is completely different from the von Neumann architecture (see Table 1). This difference significantly affects the type of functions each computational model can best perform.

Numerous efforts to develop "intelligent" programs based on von Neumann's centralized architecture have not resulted in general-purpose intelligent programs. Inspired by biological neural networks, ANNs are massively parallel computing systems consisting of an exremely large number of simple processors with many interconnections. ANN models attempt to use some "organizational" principles believed to be used in the human

0018-9162/96/$5.00 © 1996 IEEE

Challenging problems

Let us consider the following problems of interest to computer scientists and engineers.

Pattern classification

The task of pattern classification is to assign an input pattern (like a speech waveform or handwritten symbol) represented by a feature vector to one of many prespecified classes (see Figure A1). Well-known applications include character recognition, speech recognition, EEG waveform classification, blood cell classification, and printed circuit board inspection.

Clustering/categorization

In clustering, also known as unsupervised pattern classification, there are no training data with known class labels. A clustering algorithm explores the similarity between the patterns and places similar patterns in a cluster (see Figure A2). Well-known clustering applications include data mining, data compression, and exploratory data analysis.

Function approximation

Suppose a set of n labeled training patterns (input-output pairs), $\{(\mathbf{x}_1, y_1),(\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, have been generated from an unknown function $\mu(\mathbf{x})$ (subject to noise). The task of function approximation is to find an estimate, say $\hat{\mu}$, of the unknown function μ (Figure A3). Various engineering and scientific modeling problems require function approximation.

Prediction/forecasting

Given a set of n samples $\{y(t_1), y(t_2), \ldots, y(t_n)\}$ in a time sequence, $t_1, t_2, \ldots, t_n$, the task is to predict the sample $y(t_{n+1})$ at some future time t_{n+1}. Prediction/forecasting has a significant impact on decision-making in business, science, and engineering. Stock market prediction and weather forecasting are typical applications of prediction/forecasting techniques (see Figure A4).

Optimization

A wide variety of problems in mathematics, statistics, engineering, science, medicine, and economics can be posed as optimization problems. The goal of an optimization algorithm is to find a solution satisfying a set of constraints such that an objective function is maximized or minimized. The Traveling Salesman Problem (TSP), an *NP-complete* problem, is a classic example (see Figure A5).

Content-addressable memory

In the von Neumann model of computation, an entry in memory is accessed only through its address, which is independent of the content in the memory. Moreover, if a small error is made in calculating the address, a completely different item can be retrieved. Associative memory or content-addressable memory, as the name implies, can be accessed by their content. The content in the memory can be recalled even by a partial input or distorted content (see Figure A6). Associative memory is extremely desirable in building multimedia information databases.

Control

Consider a dynamic system defined by a tuple $\{u(t), y(t)\}$, where $u(t)$ is the control input and $y(t)$ is the resulting output of the system at time t. In model-reference adaptive control, the goal is to generate a control input $u(t)$ such that the system follows a desired trajectory determined by the reference model. An example is engine idle-speed control (Figure A7).

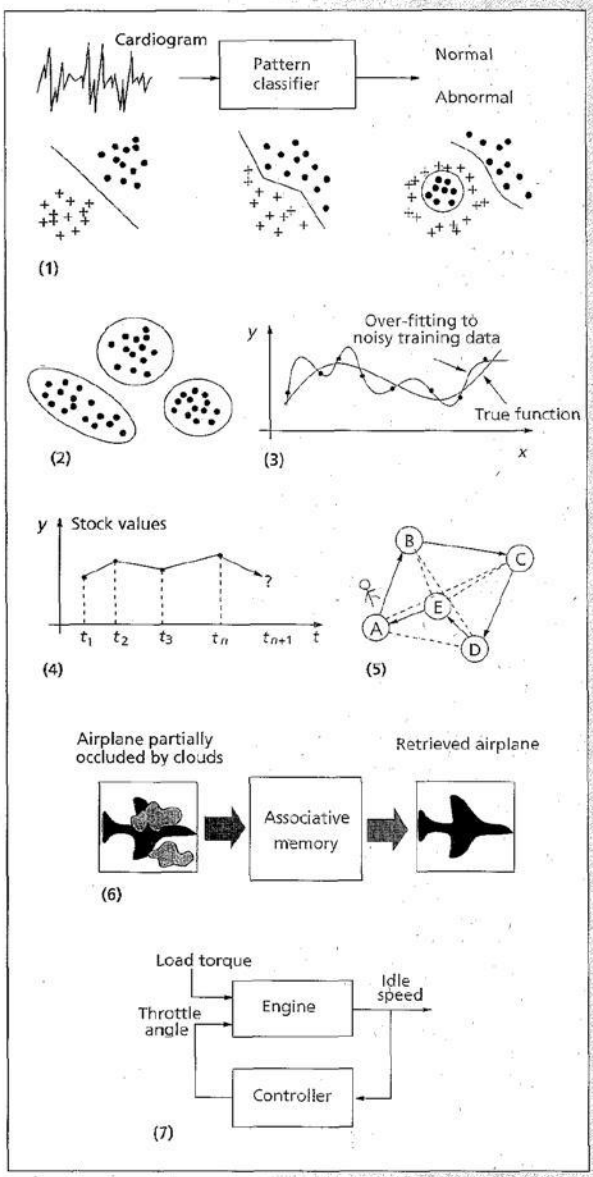


Figure A. Tasks that neural networks can perform: (1) pattern classification; (2) clustering/categorization; (3) function approximation; (4) prediction/forecasting; (5) optimization (a TSP problem example); (6) retrieval by content; and (7) control (engine idle speed). (Adapted from *DARPA Neural Network Study*[1])

brain. Modeling a biological nervous system using ANNs can also increase our understanding of biological functions. State-of-the-art computer hardware technology (such as *VLSI* and optical) has made this modeling feasible.

A thorough study of ANNs requires knowledge of neurophysiology, cognitive science/psychology, physics (statistical mechanics), control theory, computer science, artificial intelligence, statistics/mathematics, pattern recognition, computer vision, parallel processing, and hardware (digital/analog/VLSI/optical). New developments in these disciplines continuously nourish the field. On the other hand, ANNs also provide an impetus to these disciplines in the form of new tools and representations. This symbiosis is necessary for the vitality of neural network research. Communications among these disciplines ought to be encouraged.

Brief historical review

ANN research has experienced three periods of extensive activity. The first peak in the 1940s was due to McCulloch and Pitts' pioneering work.[4] The second occurred in the 1960s with Rosenblatt's perceptron convergence theorem[5] and Minsky and Papert's work showing the limitations of a simple perceptron.[6] Minsky and Papert's results dampened the enthusiasm of most researchers, especially those in the computer science community. The resulting lull in neural network research lasted almost 20 years. Since the early 1980s, ANNs have received considerable renewed interest. The major developments behind this resurgence include Hopfield's energy approach[7] in 1982 and the back-propagation learning algorithm for multilayer perceptrons (multilayer feedforward networks) first proposed by Werbos,[8] reinvented several times, and then popularized by Rumelhart et al.[9] in 1986. Anderson and Rosenfeld[10] provide a detailed historical account of ANN developments.

Biological neural networks

A *neuron* (or nerve cell) is a special biological cell that processes information (see Figure 1). It is composed of a cell body, or *soma*, and two types of out-reaching tree-like branches: the *axon* and the *dendrites*. The cell body has a nucleus that contains information about hereditary traits and a plasma that holds the molecular equipment for producing material needed by the neuron. A neuron receives signals (impulses) from other neurons through its dendrites (receivers) and transmits signals generated by its cell body along the axon (transmitter), which eventually branches into strands and substrands. At the terminals of these strands are the *synapses*. A synapse is an elementary structure and functional unit between two neurons (an axon strand of one neuron and a dendrite of another). When the impulse reaches the synapse's terminal, certain chemicals called neurotransmitters are released. The neurotransmitters diffuse across the synaptic gap, to enhance or inhibit, depending on the type of the synapse, the receptor neuron's own tendency to emit electrical impulses. The synapse's effectiveness can be adjusted by the signals passing through it so that the synapses can *learn* from the activities in which they participate. This dependence on history acts as a memory, which is possibly responsible for human memory.

The cerebral cortex in humans is a large flat sheet of neurons about 2 to 3 millimeters thick with a surface area of about 2,200 cm^2, about twice the area of a standard computer keyboard. The cerebral cortex contains about 10^{11} neurons, which is approximately the number of stars in the Milky Way.[11] Neurons are massively connected, much more complex and dense than telephone networks. Each neuron is connected to 10^3 to 10^4 other neurons. In total, the human brain contains approximately 10^{14} to 10^{15} interconnections.

Neurons communicate through a very short train of pulses, typically milliseconds in duration. The *message* is modulated on the pulse-transmission frequency. This frequency can vary from a few to several hundred hertz, which is a million times slower than the fastest switching speed in electronic circuits. However, complex perceptual decisions such as face recognition are typically made by humans within a few hundred milliseconds. These decisions are made by a network of neurons whose operational speed is only a few milliseconds. This implies that the computations cannot take more than about 100 serial stages. In other

Table 1. Von Neumann computer versus biological neural system.

	Von Neumann computer	Biological neural system
Processor	Complex High speed One or a few	Simple Low speed A large number
Memory	Separate from a processor Localized Noncontent addressable	Integrated into processor Distributed Content addressable
Computing	Centralized Sequential Stored programs	Distributed Parallel Self-learning
Reliability	Very vulnerable	Robust
Expertise	Numerical and symbolic manipulations	Perceptual problems
Operating environment	Well-defined, well-constrained	Poorly defined, unconstrained

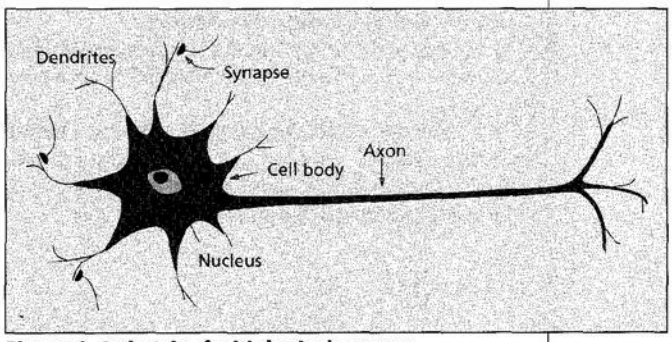


Figure 1. A sketch of a biological neuron.

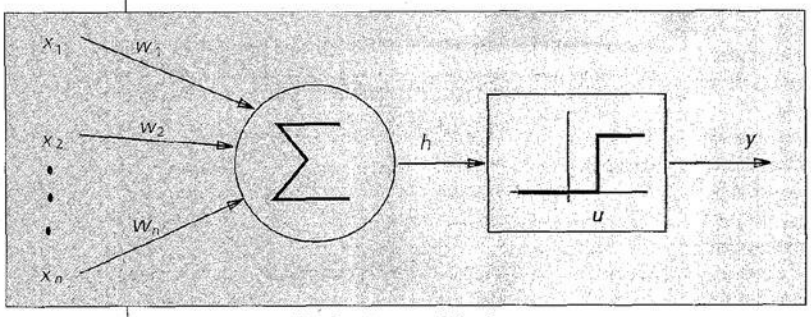


Figure 2. McCulloch-Pitts model of a neuron.

words, the brain runs parallel programs that are about 100 steps long for such perceptual tasks. This is known as the *hundred step rule.*[12] The same timing considerations show that the amount of information sent from one neuron to another must be very small (a few bits). This implies that critical information is not transmitted directly, but captured and distributed in the interconnections—hence the name, *connectionist* model, used to describe ANNs.

Interested readers can find more introductory and easily comprehensible material on biological neurons and neural networks in Brunak and Lautrup.[11]

ANN OVERVIEW

Computational models of neurons

McCulloch and Pitts[4] proposed a binary threshold unit as a computational model for an artificial neuron (see Figure 2).

This mathematical neuron computes a weighted sum of its n input signals, $x_j, j = 1, 2, \ldots, n$, and generates an output of 1 if this sum is above a certain threshold u. Otherwise, an output of 0 results. Mathematically,

$$y = \theta\left(\sum_{j=1}^{n} w_j x_j - u\right),$$

where $\theta(\cdot)$ is a unit step function at 0, and w_j is the synapse weight associated with the jth input. For simplicity of notation, we often consider the threshold u as another weight $w_0 = -u$ attached to the neuron with a constant input $x_0 = 1$. Positive weights correspond to *excitatory* synapses, while negative weights model *inhibitory* ones. McCulloch and Pitts proved that, in principle, suitably chosen weights let a synchronous arrangement of such neurons perform universal computations. There is a crude analogy here to a biological neuron: wires and interconnections model axons and dendrites, connection weights represent synapses, and the threshold function approximates the activity in a soma. The McCulloch and Pitts model, however, contains a number of simplifying assumptions that do not reflect the true behavior of biological neurons.

The McCulloch-Pitts neuron has been generalized in many ways. An obvious one is to use activation functions other than the threshold function, such as piecewise linear, sigmoid, or Gaussian, as shown in Figure 3. The sigmoid function is by far the most frequently used in ANNs. It is a strictly increasing function that exhibits smoothness and has the desired asymptotic properties. The standard sigmoid function is the *logistic* function, defined by

$$g(x) = 1/(1 + \exp\{-\beta x\}),$$

where β is the slope parameter.

Network architectures

ANNs can be viewed as weighted directed graphs in which artificial neurons are nodes and directed edges (with weights) are connections between neuron outputs and neuron inputs.

Based on the connection pattern (architecture), ANNs can be grouped into two categories (see Figure 4):

- *feed-forward* networks, in which graphs have no loops, and
- *recurrent* (or *feedback*) networks, in which loops occur because of feedback connections.

In the most common family of feed-forward networks, called multilayer perceptron, neurons are organized into layers that have unidirectional connections between them. Figure 4 also shows typical networks for each category.

Different connectivities yield different network behaviors. Generally speaking, feed-forward networks are *static*, that is, they produce only one set of output values rather than a sequence of values from a given input. Feed-forward networks are memory-less in the sense that their response to an input is independent of the previous network state. Recurrent, or feedback, networks, on the other hand, are dynamic systems. When a new input pattern is presented, the neuron outputs are computed. Because of the feedback paths, the inputs to each neuron are then modified, which leads the network to enter a new state.

Different network architectures require appropriate learning algorithms. The next section provides an overview of learning processes.

Learning

The ability to learn is a fundamental trait of intelligence. Although a precise definition of learning is difficult to formulate, a learning process in the ANN context can be viewed as the problem of updating network architecture and connection weights so that a network can efficiently perform a specific task. The network usually must learn the connection weights from available training patterns. Performance is improved over time by iteratively updating the weights in the network. ANNs' ability to automatically *learn from examples* makes them attractive and exciting. Instead of following a set of *rules* specified by human experts, ANNs appear to learn underlying rules (like input-output relationships) from the given collection of representative examples. This is one of the major advantages of neural networks over traditional expert systems.

To understand or design a learning process, you must first have a model of the environment in which a neural network operates, that is, you must know what information is available to the network. We refer to this model as

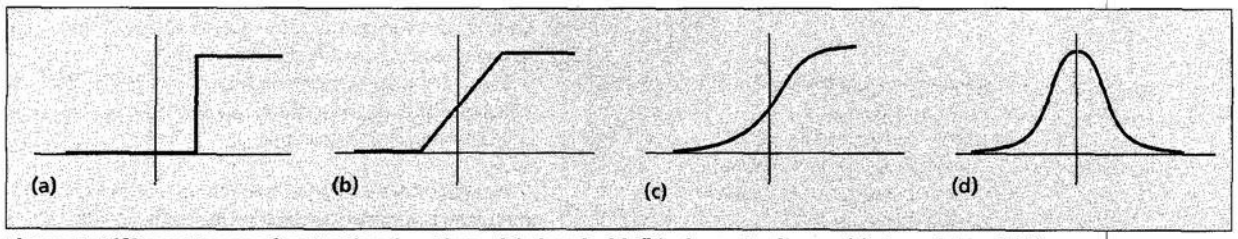


Figure 3. Different types of activation functions: (a) threshold, (b) piecewise linear, (c) sigmoid, and (d) Gaussian.

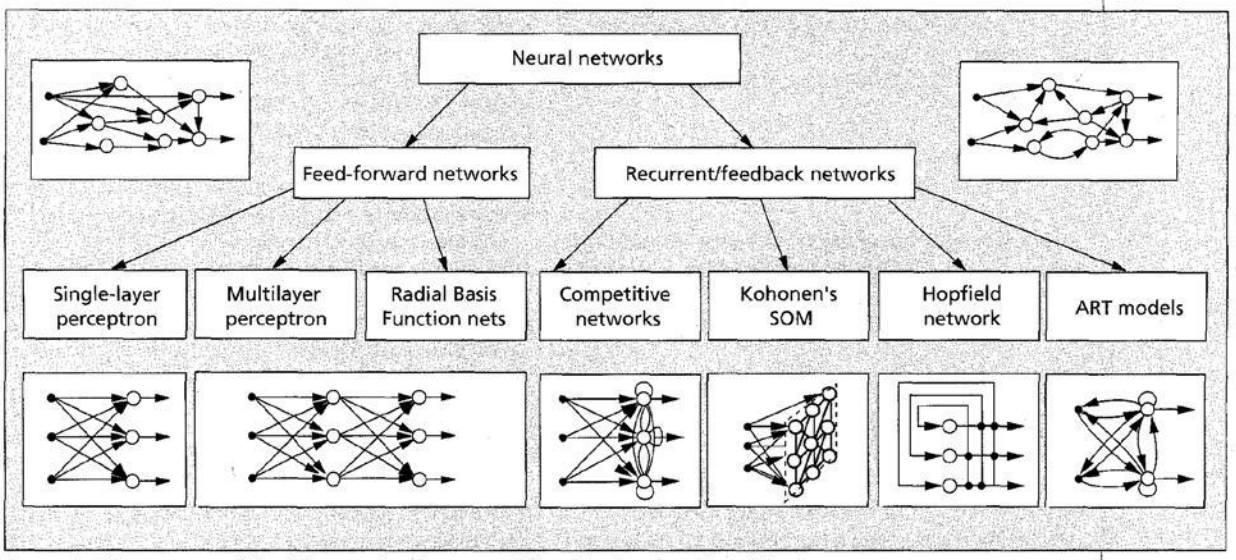


Figure 4. A taxonomy of feed-forward and recurrent/feedback network architectures.

a learning paradigm.[3] Second, you must understand how network weights are updated, that is, which *learning rules* govern the updating process. A *learning algorithm* refers to a procedure in which learning rules are used for adjusting the weights.

There are three main learning paradigms: supervised, unsupervised, and hybrid. In supervised learning, or learning with a "teacher," the network is provided with a correct answer (output) for every input pattern. Weights are determined to allow the network to produce answers as close as possible to the known correct answers. Reinforcement learning is a variant of supervised learning in which the network is provided with only a critique on the correctness of network outputs, not the correct answers themselves. In contrast, unsupervised learning, or learning without a teacher, does not require a correct answer associated with each input pattern in the training data set. It explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories from these correlations. Hybrid learning combines supervised and unsupervised learning. Part of the weights are usually determined through supervised learning, while the others are obtained through unsupervised learning.

Learning theory must address three fundamental and practical issues associated with learning from samples: capacity, sample complexity, and computational complexity. Capacity concerns how many patterns can be stored, and what functions and decision boundaries a network can form.

Sample complexity determines the number of training patterns needed to train the network to guarantee a valid generalization. Too few patterns may cause "over-fitting" (wherein the network performs well on the training data set, but poorly on independent test patterns drawn from the same distribution as the training patterns, as in Figure A3).

Computational complexity refers to the time required for a learning algorithm to estimate a solution from training patterns. Many existing learning algorithms have high computational complexity. Designing efficient algorithms for neural network learning is a very active research topic.

There are four basic types of learning rules: error-correction, Boltzmann, Hebbian, and competitive learning.

ERROR-CORRECTION RULES. In the supervised learning paradigm, the network is given a desired output for each input pattern. During the learning process, the actual output y generated by the network may not equal the desired output d. The basic principle of error-correction learning rules is to use the error signal $(d - y)$ to modify the connection weights to gradually reduce this error.

The perceptron learning rule is based on this error-correction principle. A perceptron consists of a single neuron with adjustable weights, $w_j, j = 1, 2, \ldots, n$, and threshold u, as shown in Figure 2. Given an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^t$, the net input to the neuron is

Perceptron learning algorithm

1. Initialize the weights and threshold to small random numbers.
2. Present a pattern vector $(x_1, x_2, \ldots, x_n)^t$ and evaluate the output of the neuron.
3. Update the weights according to

$$w_j(t+1) = w_j(t) + \eta\,(d - y)\,x_j,$$

where d is the desired output, t is the iteration number, and η $(0.0 < \eta < 1.0)$ is the gain (step size).

$$v = \sum_{j=1}^{n} w_j x_j - u$$

The output y of the perceptron is $+1$ if $v > 0$, and 0 otherwise. In a two-class classification problem, the perceptron assigns an input pattern to one class if $y = 1$, and to the other class if $y = 0$. The linear equation

$$\sum_{j=1}^{n} w_j x_j - u = 0$$

defines the decision boundary (a hyperplane in the n-dimensional input space) that halves the space.

Rosenblatt[5] developed a learning procedure to determine the weights and threshold in a perceptron, given a set of training patterns (see the "Perceptron learning algorithm" sidebar).

Note that learning occurs only when the perceptron makes an error. Rosenblatt proved that when training patterns are drawn from two linearly separable classes, the perceptron learning procedure converges after a finite number of iterations. This is the *perceptron convergence theorem*. In practice, you do not know whether the patterns are linearly separable. Many variations of this learning algorithm have been proposed in the literature.[2] Other activation functions that lead to different learning characteristics can also be used. However, *a single-layer perceptron can only separate linearly separable patterns as long as a monotonic activation function is used.*

The back-propagation learning algorithm (see the "Back-propagation algorithm sidebar") is also based on the error-correction principle.

BOLTZMANN LEARNING. Boltzmann machines are symmetric recurrent networks consisting of binary units (+1 for "on" and −1 for "off"). By symmetric, we mean that the weight on the connection from unit i to unit j is equal to the weight on the connection from unit j to unit i ($w_{ij} = w_{ji}$). A subset of the neurons, called *visible*, interact with the environment; the rest, called *hidden*, do not. Each neuron is a stochastic unit that generates an output (or state) according to the Boltzmann distribution of statistical mechanics. Boltzmann machines operate in two modes: *clamped*, in which visible neurons are clamped onto specific states determined by the environment; and *free-running*, in which both visible and hidden neurons are allowed to operate freely.

Boltzmann learning is a stochastic learning rule derived from information-theoretic and thermodynamic principles.[10] The objective of Boltzmann learning is to adjust the connection weights so that the states of visible units satisfy a particular desired probability distribution. According to the Boltzmann learning rule, the change in the connection weight w_{ij} is given by

$$\Delta w_{ij} = \eta(\bar{\rho}_{ij} - \rho_{ij}),$$

where η is the learning rate, and $\bar{\rho}_{ij}$ and ρ_{ij} are the correlations between the states of units i and j when the network operates in the clamped mode and free-running mode, respectively. The values of $\bar{\rho}_{ij}$ and ρ_{ij} are usually estimated from Monte Carlo experiments, which are extremely slow.

Boltzmann learning can be viewed as a special case of error-correction learning in which error is measured not as the direct difference between desired and actual outputs, but as the difference between the correlations among the outputs of two neurons under clamped and free-running operating conditions.

HEBBIAN RULE. The oldest learning rule is *Hebb's postulate of learning*.[13] Hebb based it on the following observation from neurobiological experiments: If neurons on both sides of a synapse are activated synchronously and repeatedly, the synapse's strength is selectively increased.

Mathematically, the Hebbian rule can be described as

$$w_{ij}(t+1) = w_{ij}(t) + \eta\, y_j(t)\, x_i(t),$$

where x_i and y_j are the output values of neurons i and j, respectively, which are connected by the synapse w_{ij}, and η is the learning rate. Note that x_i is the input to the synapse.

An important property of this rule is that learning is done locally, that is, the change in synapse weight depends only on the activities of the two neurons connected by it. This significantly simplifies the complexity of the learning circuit in a VLSI implementation.

A single neuron trained using the Hebbian rule exhibits an orientation selectivity. Figure 5 demonstrates this property. The points depicted are drawn from a two-dimen-

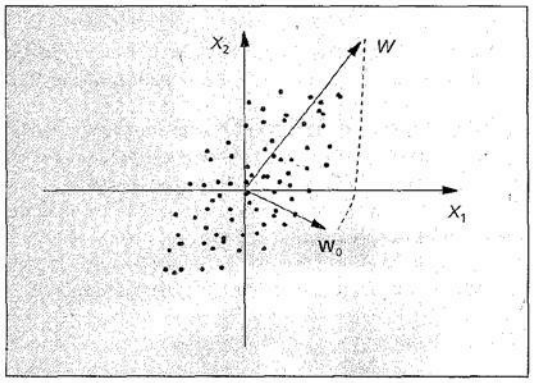


Figure 5. Orientation selectivity of a single neuron trained using the Hebbian rule.

sional Gaussian distribution and used for training a neuron. The weight vector of the neuron is initialized to $\mathbf{w}_0$ as shown in the figure. As the learning proceeds, the weight vector moves progressively closer to the direction $\mathbf{w}$ of maximal variance in the data. In fact, $\mathbf{w}$ is the eigenvector of the covariance matrix of the data corresponding to the largest eigenvalue.

COMPETITIVE LEARNING RULES. Unlike Hebbian learning (in which multiple output units can be fired simultaneously), competitive-learning output units compete among themselves for activation. As a result, only one output unit is active at any given time. This phenomenon is known as *winner-take-all*. Competitive learning has been found to exist in biological neural networks.[3]

Competitive learning often clusters or categorizes the input data. Similar patterns are grouped by the network and represented by a single unit. This grouping is done automatically based on data correlations.

The simplest competitive learning network consists of a single layer of output units as shown in Figure 4. Each output unit i in the network connects to all the input units (x_j's) via weights, $w_{ij}, j=1, 2, \ldots, n$. Each output unit also connects to all other output units via inhibitory weights but has a self-feedback with an excitatory weight. As a result of competition, only the unit i^* with the largest (or the smallest) net input becomes the winner, that is, $\mathbf{w}_{i^*} \cdot \mathbf{x} \geq \mathbf{w}_i \cdot \mathbf{x}$, $\forall i$, or $\| \mathbf{w}_{i^*} - \mathbf{x} \| \leq \| \mathbf{w}_i - \mathbf{x} \|$, $\forall i$. When all the weight vectors are normalized, these two inequalities are equivalent.

A simple competitive learning rule can be stated as

$$\Delta w_{ij} = \begin{cases} \eta(x_j^u - w_{i^*j}), & i = i^*, \\ 0, & i \neq i^*. \end{cases} \quad (1)$$

Note that only the weights of the winner unit get updated. The effect of this learning rule is to move the stored pattern in the winner unit (weights) a little bit closer to the input pattern. Figure 6 demonstrates a geometric interpretation of competitive learning. In this example, we assume that all input vectors have been normalized to have unit length. They are depicted as black dots in Figure 6. The weight vectors of the three units are randomly initialized. Their initial and final positions on the sphere after competitive learning are marked as Xs in Figures 6a and 6b, respectively. In Figure 6, each of the three natural groups (clusters) of patterns has been discovered by an output unit whose weight vector points to the center of gravity of the discovered group.

You can see from the competitive learning rule that the network will not stop learning (updating weights) unless the learning rate η is 0. A particular input pattern can fire different output units at different iterations during learning. This brings up the stability issue of a learning system. The system is said to be *stable* if no pattern in the training data changes its category after a finite number of learning iterations. One way to achieve stability is to force the learning rate to decrease gradually as the learning process proceeds towards 0. However, this artificial freezing of learning causes another problem termed *plasticity*, which is the ability to adapt to new data. This is known as Grossberg's *stability-plasticity* dilemma in competitive learning.

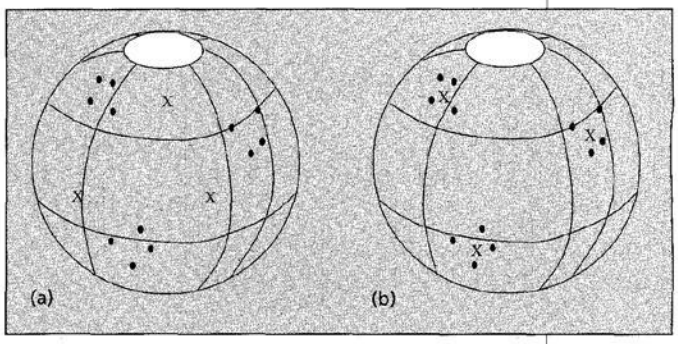


Figure 6. An example of competitive learning: (a) before learning; (b) after learning.

The most well-known example of competitive learning is *vector quantization* for data compression. It has been widely used in speech and image processing for efficient storage, transmission, and modeling. Its goal is to represent a set or distribution of input vectors with a relatively small number of prototype vectors (weight vectors), or a codebook. Once a codebook has been constructed and agreed upon by both the transmitter and the receiver, you need only transmit or store the index of the corresponding prototype to the input vector. Given an input vector, its corresponding prototype can be found by searching for the nearest prototype in the codebook.

SUMMARY. Table 2 summaries various learning algorithms and their associated network architectures (this is not an exhaustive list). Both supervised and unsupervised learning paradigms employ learning rules based

Back-propagation algorithm

1. Initialize the weights to small random values.
2. Randomly choose an input pattern $\mathbf{x}^{(u)}$.
3. Propagate the signal forward through the network.
4. Compute δ_i^L in the output layer $(o_i = y_i^L)$

   $$\delta_i^L = g'(h_i^L)\left[d_i^u - y_i^L\right],$$

   where h_i^l represents the net input to the ith unit in the lth layer, and g' is the derivative of the activation function g.
5. Compute the deltas for the preceding layers by propagating the errors backwards;

   $$\delta_i^l = g'(h_i^l)\sum_j w_{ij}^{l+1}\delta_j^{l+1},$$

   for $l = (L-1), \ldots, 1$.
6. Update weights using

   $$\Delta w_{ji}^l = \eta \delta_j^l y_j^{l-1}$$

7. Go to step 2 and repeat for the next pattern until the error in the output layer is below a prespecified threshold or a maximum number of iterations is reached.

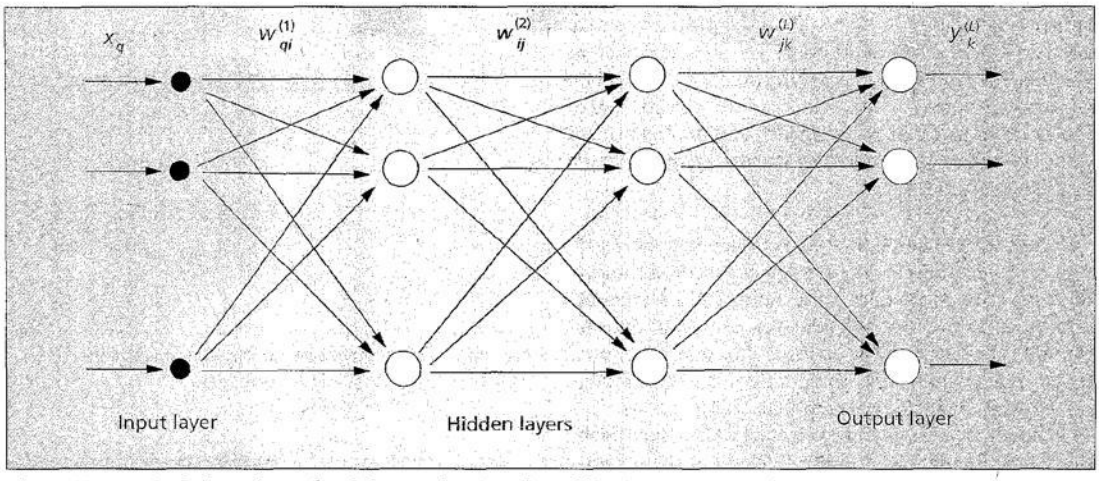


Figure 7. A typical three-layer feed-forward network architecture.

on error-correction, Hebbian, and competitive learning. Learning rules based on error-correction can be used for training feed-forward networks, while Hebbian learning rules have been used for all types of network architectures. However, each learning algorithm is designed for training a specific architecture. Therefore, when we discuss a learning algorithm, a particular network architecture association is implied. Each algorithm can

Table 2. Well-known learning algorithms.

Paradigm	Learning rule	Architecture	Learning algorithm	Task
Supervised	Error-correction	Single- or multilayer perceptron	Perceptron learning algorithms Back-propagation Adaline and Madaline	Pattern classification Function approximation Prediction, control
	Boltzmann	Recurrent	Boltzmann learning algorithm	Pattern classification
	Hebbian	Multilayer feed-forward	Linear discriminant analysis	Data analysis Pattern classification
	Competitive	Competitive	Learning vector quantization	Within-class categorization Data compression
		ART network	ARTMap	Pattern classification Within-class categorization
Unsupervised	Error-correction	Multilayer feed-forward	Sammon's projection	Data analysis
	Hebbian	Feed-forward or competitive	Principal component analysis	Data analysis Data compression
		Hopfield Network	Associative memory learning	Associative memory
	Competitive	Competitive	Vector quantization	Categorization Data compression
		Kohonen's SOM	Kohonen's SOM	Categorization Data analysis
		ART networks	ART1, ART2	Categorization
Hybrid	Error-correction and competitive	RBF network	RBF learning algorithm	Pattern classification Function approximation Prediction, control

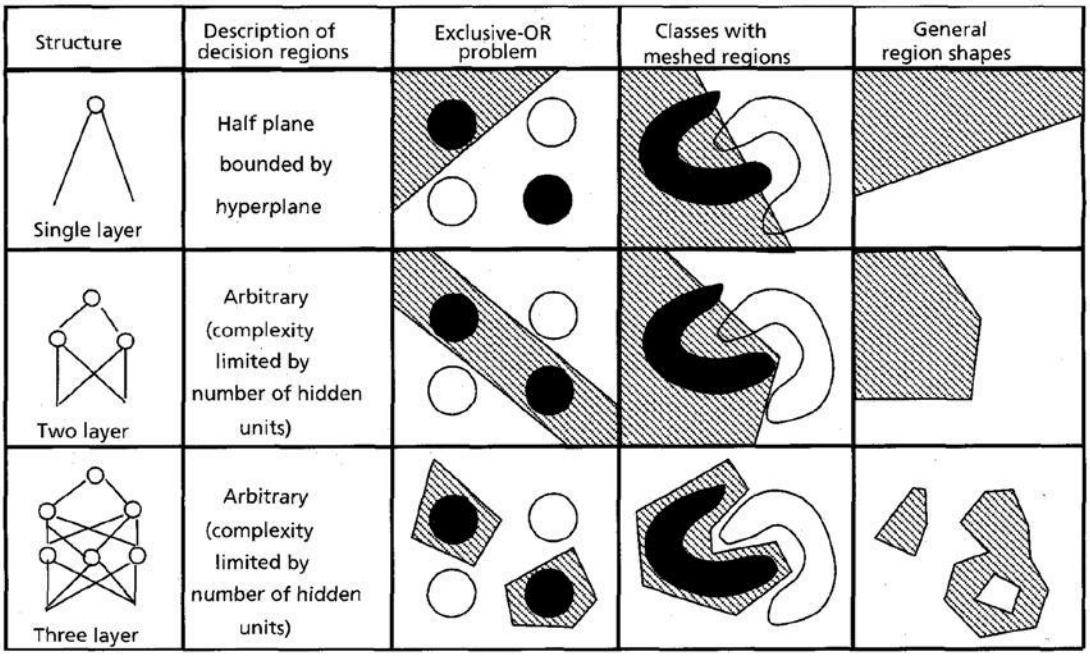


Figure 8. A geometric interpretation of the role of hidden unit in a two-dimensional input space.

perform only a few tasks well. The last column of Table 2 lists the tasks that each algorithm can perform. Due to space limitations, we do not discuss some other algorithms, including Adaline, Madaline,[14] linear discriminant analysis,[15] Sammon's projection,[15] and principal component analysis.[2] Interested readers can consult the corresponding references (this article does not always cite the first paper proposing the particular algorithms).

MULTILAYER FEED-FORWARD NETWORKS

Figure 7 shows a typical three-layer perceptron. In general, a standard L-layer feed-forward network (we adopt the convention that the input nodes are not counted as a layer) consists of an input stage, $(L-1)$ hidden layers, and an output layer of units successively connected (fully or locally) in a feed-forward fashion with no connections between units in the same layer and no feedback connections between layers.

Multilayer perceptron

The most popular class of multilayer feed-forward networks is *multilayer perceptrons* in which each computational unit employs either the thresholding function or the sigmoid function. Multilayer perceptrons can form arbitrarily complex decision boundaries and represent any Boolean function.[6] The development of the *back-propagation* learning algorithm for determining weights in a multilayer perceptron has made these networks the most popular among researchers and users of neural networks.

We denote $w_{ij}^{(l)}$ as the weight on the connection between the ith unit in layer $(l-1)$ to jth unit in layer l.

Let $\{(\mathbf{x}^{(1)}, \mathbf{d}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{d}^{(2)}), \ldots, (\mathbf{x}^{(p)}, \mathbf{d}^{(p)})\}$ be a set of p training patterns (input-output pairs), where $\mathbf{x}^{(i)} \in R^n$ is the input vector in the n-dimensional pattern space, and $\mathbf{d}^{(i)} \in [0, 1]^m$, an m-dimensional hypercube. For classification purposes, m is the number of classes. The squared-error cost function most frequently used in the ANN literature is defined as

$$E = \frac{1}{2}\sum_{i=1}^{p} \left\| \mathbf{y}^{(i)} - \mathbf{d}^{(i)} \right\|^2 \qquad (2)$$

The back-propagation algorithm[9] is a gradient-descent method to minimize the squared-error cost function in Equation 2 (see "Back-propagation algorithm" sidebar).

A geometric interpretation (adopted and modified from Lippmann[14]) shown in Figure 8 can help explicate the role of hidden units (with the threshold activation function).

Each unit in the first hidden layer forms a hyperplane in the pattern space; boundaries between pattern classes can be approximated by hyperplanes. A unit in the second hidden layer forms a hyperregion from the outputs of the first-layer units; a decision region is obtained by performing an AND operation on the hyperplanes. The output-layer units combine the decision regions made by the units in the second hidden layer by performing logical OR operations. Remember that this scenario is depicted only to explain the role of hidden units. Their actual behavior, after the network is trained, could differ.

A two-layer network can form more complex decision boundaries than those shown in Figure 8. Moreover, multilayer perceptrons with sigmoid activation functions can form smooth decision boundaries rather than piecewise linear boundaries.

Radial Basis Function network

The Radial Basis Function (RBF) network,[3] which has two layers, is a special class of multilayer feed-forward net-

works. Each unit in the hidden layer employs a radial basis function, such as a Gaussian kernel, as the activation function. The radial basis function (or kernel function) is centered at the point specified by the weight vector associated with the unit. Both the positions and the widths of these kernels must be learned from training patterns. There are usually many fewer kernels in the RBF network than there are training patterns. Each output unit implements a linear combination of these radial basis functions. From the point of view of function approximation, the hidden units provide a set of functions that constitute a basis set for representing input patterns in the space spanned by the hidden units.

There are a variety of learning algorithms for the RBF network.[3] The basic one employs a two-step learning strategy, or hybrid learning. It estimates kernel positions and kernel widths using an unsupervised clustering algorithm, followed by a supervised least mean square (LMS) algorithm to determine the connection weights between the hidden layer and the output layer. Because the output units are linear, a noniterative algorithm can be used. After this initial solution is obtained, a supervised gradient-based algorithm can be used to refine the network parameters.

This hybrid learning algorithm for training the RBF network converges much faster than the back-propagation algorithm for training multilayer perceptrons. However, for many problems, the RBF network often involves a larger number of hidden units. This implies that the runtime (after training) speed of the RBF network is often slower than the runtime speed of a multilayer perceptron. The efficiencies (error versus network size) of the RBF network and the multilayer perceptron are, however, problem-dependent. It has been shown that the RBF network has the same asymptotic approximation power as a multilayer perceptron.

SOM learning algorithm

1. Initialize weights to small random numbers; set initial learning rate and neighborhood.
2. Present a pattern $\mathbf{x}$, and evaluate the network outputs.
3. Select the unit (c_i, c_j) with the minimum output:

$$\|\mathbf{x} - \mathbf{w}_{c_i c_j}\| = \min_{ij} \|\mathbf{x} - \mathbf{w}_{ij}\|$$

4. Update all weights according to the following learning rule:

$$\mathbf{w}_{ij}(t+1) = \begin{cases} \mathbf{w}_{ij}(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{w}_{ij}(t)], & \text{if } (i,j) \in N_{c_i c_j}(t), \\ \mathbf{w}_{ij}(t), & \text{otherwise,} \end{cases}$$

where $N_{c_i c_j}(t)$ is the neighborhood of the unit (c_i, c_j) at time t, and $\alpha(t)$ is the learning rate.

5. Decrease the value of $\alpha(t)$ and shrink the neighborhood $N_{c_i c_j}(t)$.
6. Repeat steps 2 through 5 until the change in weight values is less than a prespecified threshold or a maximum number of iterations is reached.

Issues

There are many issues in designing feed-forward networks, including

- how many layers are needed for a given task,
- how many units are needed per layer,
- how will the network perform on data not included in the training set (generalization ability), and
- how large the training set should be for "good" generalization.

Although multilayer feed-forward networks using back-propagation have been widely employed for classification and function approximation,[2] many design parameters still must be determined by trial and error. Existing theoretical results provide only very loose guidelines for selecting these parameters in practice.

KOHONEN'S SELF-ORGANIZING MAPS

The self-organizing map (SOM)[16] has the desirable property of topology preservation, which captures an important aspect of the feature maps in the cortex of highly developed animal brains. In a topology-preserving mapping, nearby input patterns should activate nearby output units on the map. Figure 4 shows the basic network architecture of Kohonen's SOM. It basically consists of a two-dimensional array of units, each connected to all n input nodes. Let $\mathbf{w}_{ij}$ denote the n-dimensional vector associated with the unit at location (i, j) of the 2D array. Each neuron computes the Euclidean distance between the input vector $\mathbf{x}$ and the stored weight vector $\mathbf{w}_{ij}$.

This SOM is a special type of competitive learning network that defines a spatial neighborhood for each output unit. The shape of the local neighborhood can be square, rectangular, or circular. Initial neighborhood size is often set to one half to two thirds of the network size and shrinks over time according to a schedule (for example, an exponentially decreasing function). During competitive learning, all the weight vectors associated with the winner and its neighboring units are updated (see the "SOM learning algorithm" sidebar).

Kohonen's SOM can be used for projection of multivariate data, density approximation, and clustering. It has been successfully applied in the areas of speech recognition, image processing, robotics, and process control.[2] The design parameters include the dimensionality of the neuron array, the number of neurons in each dimension, the shape of the neighborhood, the shrinking schedule of the neighborhood, and the learning rate.

ADAPTIVE RESONANCE THEORY MODELS

Recall that the *stability-plasticity* dilemma is an important issue in competitive learning. How do we learn new things (plasticity) and yet retain the stability to ensure that existing knowledge is not erased or corrupted? Carpenter and Grossberg's Adaptive Resonance Theory models (ART1, ART2, and ARTMap) were developed in an attempt to overcome this dilemma.[17] The network has a sufficient supply of output units, but they are not used until deemed necessary. A unit is said to be *committed* (*uncommitted*) if it is (is not) being used. The learning algorithm updates

the stored prototypes of a category only if the input vector is sufficiently similar to them. An input vector and a stored prototype are said to resonate when they are sufficiently similar. The extent of similarity is controlled by a *vigilance parameter*, ρ, with $0 < \rho < 1$, which also determines the number of categories. When the input vector is not sufficiently similar to any existing prototype in the network, a new category is created, and an uncommitted unit is assigned to it with the input vector as the initial prototype. If no such uncommitted unit exists, a novel input generates no response.

We present only ART1, which takes binary (0/1) input to illustrate the model. Figure 9 shows a simplified diagram of the ART1 architecture.[2] It consists of two layers of fully connected units. A top-down weight vector $\mathbf{w}_j$ is associated with unit j in the input layer, and a bottom-up weight vector $\overline{\mathbf{w}}_i$ is associated with output unit i; $\overline{\mathbf{w}}_i$ is the normalized version of $\mathbf{w}_i$.

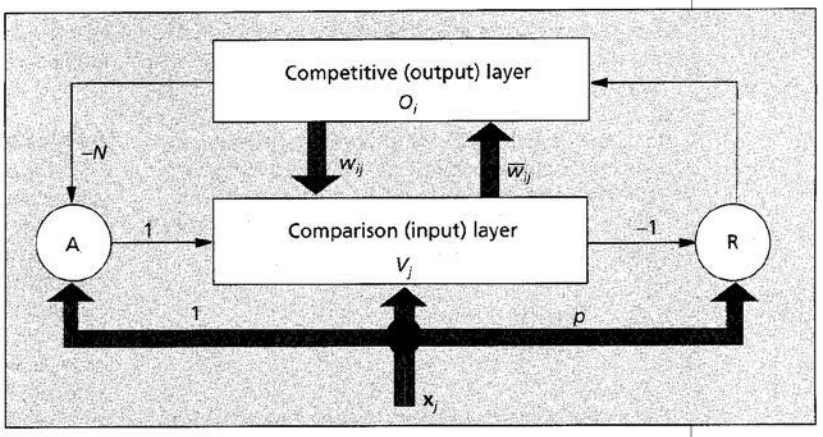


Figure 9. ART1 network.

$$\overline{\mathbf{w}}_i = \frac{\mathbf{w}_i}{\varepsilon + \sum_j w_{ji}}, \tag{3}$$

where ε is a small number used to break the ties in selecting the winner. The top-down weight vectors $\mathbf{w}_j$'s store cluster prototypes. The role of normalization is to prevent prototypes with a long vector length from dominating prototypes with a short one. Given an n-bit input vector $\mathbf{x}$, the output of the auxiliary unit A is given by

$$A = Sgn_{0/1}\left(\sum_j x_j - n\sum_i O_i - 0.5\right),$$

where $Sgn_{0/1}(x)$ is the *signum* function that produces +1 if $x \geq 0$ and 0 otherwise, and the output of an input unit is given by

$$V_j = Sgn_{0/1}\left(x_j + \sum_i w_{ji} O_i + A - 1.5\right)$$
$$= \begin{cases} x_j, & \text{if no output } O_j \text{ is "on",} \\ x_j \wedge \Sigma_i w_{ji} O_i, & \text{otherwise.} \end{cases}$$

A reset signal R is generated only when the similarity is less than the vigilance level. (See the "ART1 learning algorithm" sidebar.)

The ART1 model can create new categories and reject an input pattern when the network reaches its capacity. However, the number of categories discovered in the input data by ART1 is sensitive to the vigilance parameter.

HOPFIELD NETWORK

Hopfield used a network *energy* function as a tool for designing recurrent networks and for understanding their dynamic behavior.[7] Hopfield's formulation made explicit the principle of storing information as dynamically stable attractors and popularized the use of recurrent networks for associative memory and for solving combinatorial optimization problems.

A Hopfield network with n units has two versions: binary and continuously valued. Let v_i be the state or output of the ith unit. For binary networks, v_i is either +1 or −1, but for continuous networks, v_i can be any value between 0 and 1. Let w_{ij} be the synapse weight on the connection from units i to j. In Hopfield networks, $w_{ij} = w_{ji}$, $\forall i, j$ (symmetric networks), and $w_{ii} = 0$, $\forall i$ (no self-feedback connections). The network dynamics for the binary Hopfield network are

$$v_i = Sgn\left(\sum_j w_{ij} v_j - \theta_i\right) \tag{4}$$

ART1 learning algorithm

1. Initialize $w_{ij} = 1$, for all i, j. Enable all the output units.
2. Present a new pattern $\mathbf{x}$.
3. Find the winner unit i^* among the enabled output units

   $\overline{\mathbf{w}}_{i^*} \cdot \mathbf{x} \geq \overline{\mathbf{w}}_i \cdot \mathbf{x}, \forall i$

4. Perform vigilance test

   $r = \dfrac{\mathbf{w}_{i^*} \cdot \mathbf{x}}{\Sigma_j x_j}$

   If $r \geq \rho$ (resonance), go to step 5. Otherwise, disable unit i^* and go to step 3 (until all the output units are disabled).
5. Update the winning weight vector $\mathbf{w}_{i^*}$, enable all the output units, and go to step 2

   $\Delta w_{ji^*} = \eta\,(V_j - w_{ji^*})$

6. If all output units are disabled, select one of the uncommitted output units and set its weight vector to x. If there is no uncommitted output unit (capacity is reached), the network rejects the input pattern.

The dynamic update of network states in Equation 4 can be carried out in at least two ways: *synchronously* and *asynchronously*. In a synchronous updating scheme, all units are updated simultaneously at each time step. A central clock must synchronize the process. An asynchronous updating scheme selects one unit at a time and updates its state. The unit for updating can be randomly chosen.

The energy function of the binary Hopfield network in a state $\mathbf{v} = (v_1, v_2, \ldots, v_n)^T$ is given by

$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} v_i v_j \quad (5)$$

The central property of the energy function is that as network state evolves according to the network dynamics (Equation 4), the network energy always decreases and eventually reaches a local minimum point (attractor) where the network stays with a constant energy.

Associative memory

When a set of patterns is stored in these network attractors, it can be used as an *associative memory*. Any pattern present in the basin of attraction of a stored pattern can be used as an index to retrieve it.

An associative memory usually operates in two phases: storage and retrieval. In the storage phase, the weights in the network are determined so that the attractors of the network memorize a set of p n-dimensional patterns $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^p\}$ to be stored. A generalization of the Hebbian learning rule can be used for setting connection weights w_{ij}. In the retrieval phase, the input pattern is used as the initial state of the network, and the network evolves according to its dynamics. A pattern is produced (or retrieved) when the network reaches equilibrium.

How many patterns can be stored in a network with n binary units? In other words, what is the memory capacity of a network? It is finite because a network with n binary units has a maximum of 2^n distinct states, and not all of them are attractors. Moreover, not all attractors (stable states) can store useful patterns. Spurious attractors can also store patterns different from those in the training set.[2]

It has been shown that the maximum number of random patterns that a Hopfield network can store is $P_{max} \approx 0.15n$. When the number of stored patterns $p < 0.15n$, a nearly perfect recall can be achieved. When memory patterns are orthogonal vectors instead of random patterns, more patterns can be stored. But the number of spurious attractors increases as p reaches capacity. Several learning rules have been proposed for increasing the memory capacity of Hopfield networks.[2] Note that we require n^2 connections in the network to store p n-bit patterns.

Energy minimization

Hopfield networks always evolve in the direction that leads to lower network energy. This implies that if a combinatorial optimization problem can be formulated as minimizing this energy, the Hopfield network can be used to find the optimal (or suboptimal) solution by letting the network evolve freely. In fact, any quadratic objective function can be rewritten in the form of Hopfield network energy. For example, the classic Traveling Salesman Problem can be formulated as such a problem.

APPLICATIONS

We have discussed a number of important ANN models and learning algorithms proposed in the literature. They have been widely used for solving the seven classes of problems described in the beginning of this article. Table 2 showed typical suitable tasks for ANN models and learning algorithms. Remember that to successfully work with real-world problems, you must deal with numerous design issues, including network model, network size, activation function, learning parameters, and number of training samples. We next discuss an optical character recognition (OCR) application to illustrate how multilayer feed-forward networks are successfully used in practice.

OCR deals with the problem of processing a scanned image of text and transcribing it into machine-readable form. We outline the basic components of OCR and explain how ANNs are used for character classification.

An OCR system

An OCR system usually consists of modules for preprocessing, segmentation, feature extraction, classification, and contextual processing. A paper document is scanned to produce a gray-level or binary (black-and-white) image. In the preprocessing stage, filtering is applied to remove noise, and text areas are located and converted to a binary image using a global or local adaptive thresholding method. In the segmentation step, the text image is separated into individual characters. This is a particularly difficult task with handwritten text, which contains a proliferation of touching characters. One effective technique is to break the composite pattern into smaller patterns (over-segmentation) and find the correct character segmentation points using the output of a pattern classifier.

Because of various degrees of slant, skew, and noise level, and various writing styles, recognizing segmented characters is not easy. This is evident from Figure 10, which shows the size-normalized character bitmaps of a sample set from the NIST (National Institute of Standards and Technology) hand-print character database.[18]

Schemes

Figure 11 shows the two main schemes for using ANNs in an OCR system. The first one employs an explicit feature extractor (not necessarily a neural network). For instance, contour direction features are used in Figure 11. The extracted features are passed to the input stage of a multilayer feed-forward network.[19] This scheme is very flexible in incorporating a large variety of features. The other scheme does not explicitly extract features from the raw data. The feature extraction implicitly takes place within the intermediate stages (hidden layers) of the ANN. A nice property of this scheme is that feature extraction and classification are integrated and trained simultaneously to produce optimal classification results. It is not clear whether the types of features that can be extracted by this integrated architecture are the most effective for character recognition. Moreover, this scheme requires a much larger network than the first one.

A typical example of this integrated feature extraction-classification scheme is the network developed by Le Cun et al.[20] for zip code recognition. A 16 × 16 normalized gray-level image is presented to a feed-forward network with three hidden layers. The units in the first layer are locally connected to the units in the input layer, forming a set of local feature maps. The second hidden layer is constructed in a similar way. Each unit in the second layer also combines local information coming from feature maps in the first layer.

The activation level of an output unit can be interpreted as an approximation of the a posteriori probability of the input pattern's belonging to a particular class. The output categories are ordered according to activation levels and passed to the post-processing stage. In this stage, contextual information is exploited to update the classifier's output. This could, for example, involve looking up a dictionary of admissible words, or utilizing syntactic constraints present, for example, in phone or social security numbers.

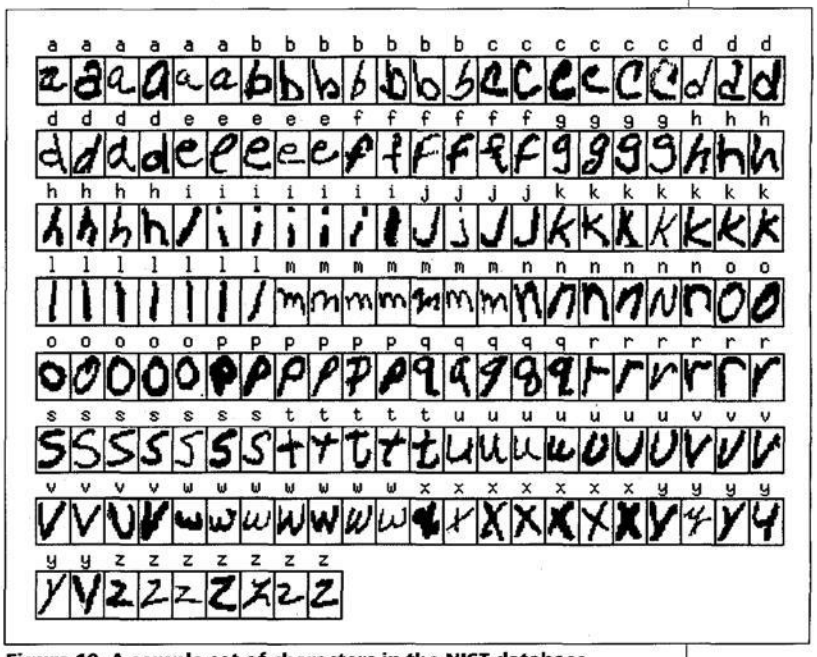


Figure 10. A sample set of characters in the NIST database.

Results

ANNs work very well in the OCR application. However, there is no conclusive evidence about their superiority over conventional statistical pattern classifiers. At the First Census Optical Character Recognition System Conference held in 1992,[18] more than 40 different handwritten character recognition systems were evaluated based on their performance on a common database. The top 10 performers used either some type of multilayer feed-forward network or a nearest neighbor-based classifier. ANNs tend to be superior in terms of speed and memory requirements compared to nearest neighbor methods. Unlike the nearest neighbor methods, classification speed using ANNs is independent of the size of the training set. The recognition accuracies of the top OCR systems on the NIST isolated (presegmented) character data were above 98 percent for digits, 96 percent for uppercase characters, and 87 percent for lowercase characters. (Low recognition accuracy for lowercase characters was largely due to the fact that the test data differed significantly from the training data, as well as being due to "ground-truth" errors.) One conclusion drawn from the test is that OCR system performance on isolated characters compares well with human performance. However, humans still outperform OCR systems on unconstrained and cursive handwritten documents.

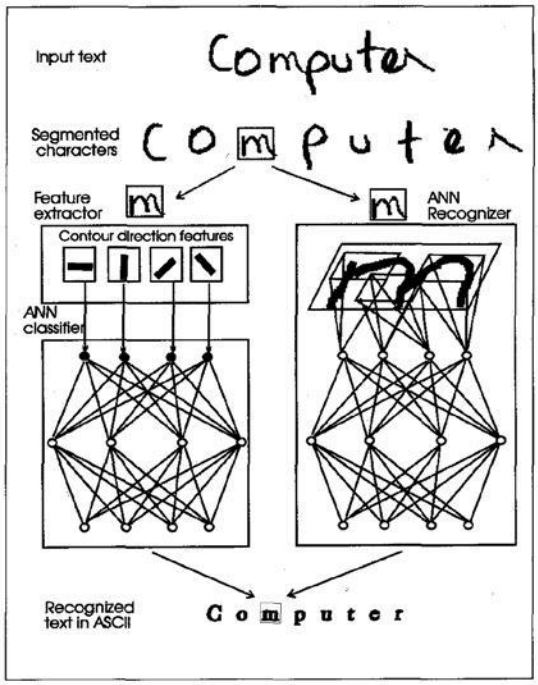


Figure 11. Two schemes for using ANNs in an OCR system.

DEVELOPMENTS IN ANNS HAVE STIMULATED a lot of enthusiasm and criticism. Some comparative studies are optimistic, some offer pessimism. For many tasks, such as pattern recognition, no one approach dominates the others. The choice of the best technique should be driven by the given application's nature. We should try to understand the capacities, assumptions, and applicability of various approaches and maximally exploit their complementary advantages to develop better intelligent systems. Such an effort may lead to a synergistic approach that combines the strengths of ANNs with other technologies to achieve significantly better performance for challenging problems. As Minsky[21] recently observed, the time has come to build systems out of diverse components. Individual modules are important, but we also need a good methodology for integration. It is clear that communication and cooperative work between

researchers working in ANNs and other disciplines will not only avoid repetitious work but (and more important) will stimulate and benefit individual disciplines. ■

Acknowledgments

We thank Richard Casey (IBM Almaden); Pat Flynn (Washington State University); William Punch, Chitra Dorai, and Kalle Karu (Michigan State University); Ali Khotanzad (Southern Methodist University); and Ishwar Sethi (Wayne State University) for their many useful suggestions.

References

1. *DARPA Neural Network Study*, AFCEA Int'l Press, Fairfax, Va., 1988.
2. J. Hertz, A. Krogh, and R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Reading, Mass., 1991.
3. S. Haykin, *Neural Networks: A Comprehensive Foundation*, MacMillan College Publishing Co., New York, 1994.
4. W.S. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bull. Mathematical Biophysics*, Vol. 5, 1943, pp. 115-133.
5. R. Rosenblatt, *Principles of Neurodynamics*, Spartan Books, New York, 1962.
6. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, Mass., 1969.
7. J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," in *Proc. Nat'l Academy of Sciences*, USA 79, 1982, pp. 2,554-2,558.
8. P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," PhD thesis, Dept. of Applied Mathematics, Harvard University, Cambridge, Mass., 1974.
9. D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, MIT Press, Cambridge, Mass., 1986.
10. J.A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, Mass., 1988.
11. S. Brunak and B. Lautrup, *Neural Networks, Computers with Intuition*, World Scientific, Singapore, 1990.
12. J. Feldman, M.A. Fanty, and N.H. Goddard, "Computing with Structured Neural Networks," *Computer*, Vol. 21, No. 3, Mar. 1988, pp. 91-103.
13. D.O. Hebb, *The Organization of Behavior*, John Wiley & Sons, New York, 1949.
14. R.P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, Vol. 4, No. 2, Apr. 1987, pp. 4-22.
15. A.K. Jain and J. Mao, "Neural Networks and Pattern Recognition," in *Computational Intelligence: Imitating Life*, J.M. Zurada, R. J. Marks II, and C.J. Robinson, eds., IEEE Press, Piscataway, N.J., 1994, pp. 194-212.
16. T. Kohonen, *Self Organization and Associative Memory*, Third Edition, Springer-Verlag, New York, 1989.
17. G.A. Carpenter and S. Grossberg, *Pattern Recognition by Self-Organizing Neural Networks*, MIT Press, Cambridge, Mass., 1991.
18. "The First Census Optical Character Recognition System Conference," R.A. Wilkinson et al., eds., . Tech. Report, NISTIR 4912, US Dept. Commerce, NIST, Gaithersburg, Md., 1992.
19. K. Mohiuddin and J. Mao, "A Comparative Study of Different Classifiers for Handprinted Character Recognition," in *Pattern Recognition in Practice IV*, E.S. Gelsema and L.N. Kanal, eds., Elsevier Science, The Netherlands, 1994, pp. 437-448.
20. Y. Le Cun et al., "Back-Propagation Applied to Handwritten Zip Code Recognition, *Neural Computation*, Vol. 1, 1989, pp. 541–551.
21. M. Minsky, "Logical Versus Analogical or Symbolic Versus Connectionist or Neat Versus Scruffy," *AI Magazine*, Vol. 65, No. 2, 1991, pp. 34-51.

Anil K. Jain *is a University Distinguished Professor and the chair of the Department of Computer Science at Michigan State University. His interests include statistical pattern recognition, exploratory pattern analysis, neural networks, Markov random fields, texture analysis, remote sensing, interpretation of range images, and 3D object recognition.*

Jain served as editor-in-chief of IEEE Transactions on Pattern Analysis and Machine Intelligence *from 1991 to 1994, and currently serves on the editorial boards of* Pattern Recognition, Pattern Recognition Letters, Journal of Mathematical Imaging, Journal of Applied Intelligence, *and* IEEE Transactions on Neural Networks. *He has coauthored, edited, and coedited numerous books in the field. Jain is a fellow of the IEEE and a speaker in the IEEE Computer Society's Distinguished Visitors Program for the Asia-Pacific region. He is a member of the IEEE Computer Society.*

Jianchang Mao *is a research staff member at the IBM Almaden Research Center. His interests include pattern recognition, neural networks, document image analysis, image processing, computer vision, and parallel computing.*

Mao received the BS degree in physics in 1983 and the MS degree in electrical engineering in 1986 from East China Normal University in Shanghai. He received the PhD in computer science from Michigan State University in 1994. Mao is the abstracts editor of IEEE Transactions on Neural Networks. *He is a member of the IEEE and the IEEE Computer Society.*

K.M. Mohiuddin *is the manager of the Document Image Analysis and Recognition project in the Computer Science Department at the IBM Almaden Research Center. He has led IBM projects on high-speed reconfigurable machines for industrial machine vision, parallel processing for scientific computing, and document imaging systems. His interests include document image analysis, handwriting recognition/OCR, data compression, and computer architecture.*

Mohiuddin received the MS and PhD degrees in electrical engineering from Stanford University in 1977 and 1982, respectively. He is an associate editor of IEEE Transactions on Pattern Analysis and Machine Intelligence. *He served on Computer's editorial board from 1984 to 1989, and is a senior member of the IEEE and a member of the IEEE Computer Society.*

Readers can contact Anil Jain at the Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, MI 48824; jain@cps.msu.edu.

Wednesday, December 30, 2015
1:46 AM

AI : Neural Network for beginners (Part 2 of 3)

Sacha Barber, 29 Jan 2007

AI : An Introduction into Neural Networks (Multi-layer networks / Back Propagation)

Download demo project (includes source code) - 812 Kb

Introduction

This article is part 2 of a series of 3 articles that I am going to post. The proposed article content will be as follows:

1. Part 1 : Is an introduction into Perceptron networks (single layer neural networks).
2. Part 2 : This one, is about multi layer neural networks, and the back propagation training method to solve a non linear classification problem such as the logic of an XOR logic gate. This is something that a Perceptron can't do. This is explained further within this article.
3. Part 3 : Will be about how to use a genetic algorithm (GA) to train a multi layer neural network to solve some logic problem.

Summary

This article will show how to use a multi-layer neural network to solve the XOR logic problem.

A Brief Recap (From part 1 of 3)

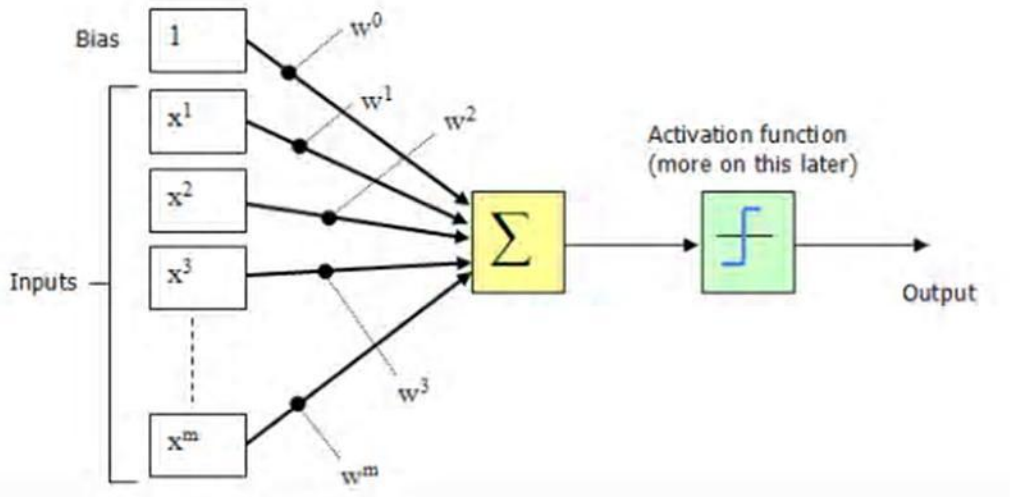
Before we commence with the nitty gritty of this new article which deals with multi layer Neural Networks, let just revisit a few key concepts. If you haven't read Part 1, perhaps you should start there.

Perceptron Configuration (Single layer network)

The inputs ☺ϸ ╕╖ ╕ϸ╛╛╒≠ ☻and connection weights ☺∑o ╘∑п ╘∑ϸ╛╛∑≠ ☻ shown below are typically real values, both positive and negative (-).

The perceptron itself, consists of weights, the summation processor, an activation function, and an adjustable threshold processor

For convenience, the normal practice is to treat the bias as just another input. The following diagram illustrates the revised configuration.



The bias can be thought of as the propensity (a tendency towards a particular way of behaving) of the perceptron to fire irrespective it's inputs. The perceptron configuration network shown above fires if the weighted sum > 0, or if you have into maths type explanations

$$\sum_{i=1}^{m} bias + (w^i x^i)$$

So that's the basic operation of a perceptron. But we now want to build more layers of these, so let's carry on to the new stuff.

So Now The New Stuff (More layers)

From this point on, anything that is being discussed relates directly to this article's code.

In the summary at the top, the problem we are trying to solve was how to use a multi-layer neural network to solve the XOR logic problem. So how is this done. Well it's really an incremental build on what Part 1 already discussed. So let's march on.

What does the XOR logic problem look like? Well, it looks like the following truth table:

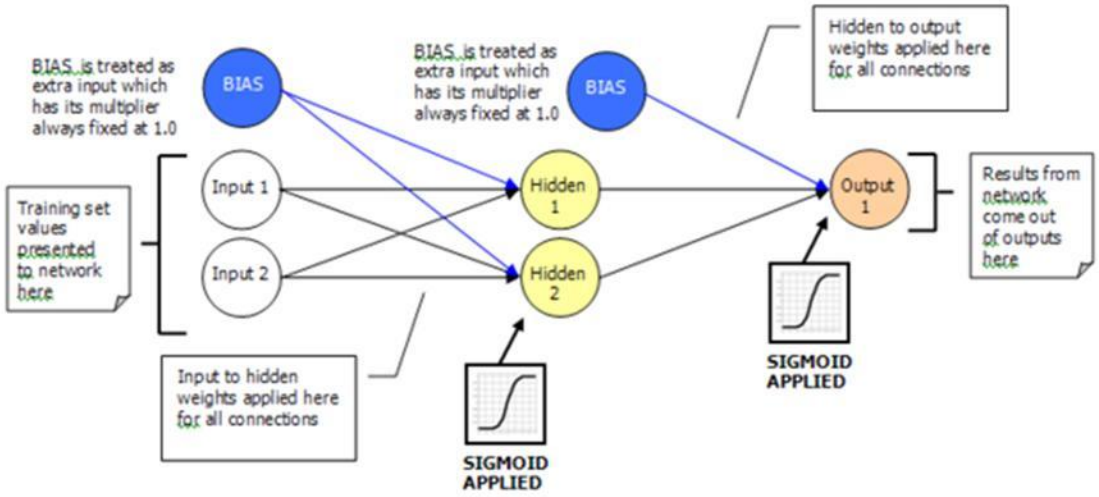
I1	I2	Output
0	0	0
0	1	1
1	0	1
1	1	0

XOR Logic Gate

Remember with a single layer (perceptron) we can't actually achieve the XOR functionality, as it is not linearly separable. But with a multi-layer network, this is achievable.

What Does The New Network Look Like

The new network that will solve the XOR problem will look similar to a single layer network. We are still dealing with inputs / weights outputs. What is new is the addition of the hidden layer.



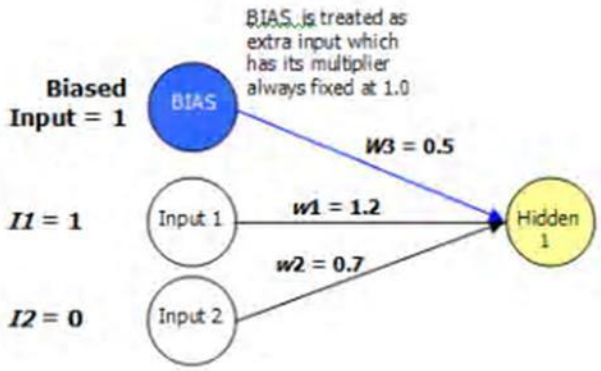
As already explained above, there is one input layer, one hidden layer and one output layer.

It is by using the inputs and weights that we are able to work out the activation for a given node. This is easily achieved for the hidd layer as it has direct links to the actual input layer.

The output layer, however, knows nothing about the input layer as it is not directly connected to it. So to work out the activation for output node we need to make use of the output from the hidden layer nodes, which are used as inputs to the output layer nodes.

This entire process described above can be thought of as a pass forward from one layer to the next.

This still works like it did with a single layer network; the activation for any given node is still worked out as follows:



$$A = \sum_{n=1}^{N+1} wi * Ii$$

NOTE : N + 1 is bias and the value of the input for the bias is 1.

Where (wi is the weight(i), and Ii is the input(i) value)

You see it the same old stuff, no demons, smoke or magic here. It's stuff we've already covered.

So that's how the network looks/works. So now I guess you want to know how to go about training it.

Types Of Learning

There are essentially 2 types of learning that may be applied, to a Neural Network, which is "Reinforcement" and "Supervised"

Reinforcement

In Reinforcement learning, during training, a set of inputs is presented to the Neural Network, the Output is 0.75, when the target w expecting 1.0.

The error (1.0 - 0.75) is used for training ('wrong by 0.25').

What if there are 2 outputs, then the total error is summed to give a single number (typically sum of squared errors). Eg "your total error on all outputs is 1.76"

Note that this just tells you how wrong you were, not in which direction you were wrong.

Using this method we may never get a result, or it could be a case of 'Hunt the needle'.

NOTE : Part 3 of this series will be using a GA to train a Neural Network, which is Reinforcement learning. The GA simply does what a GA does, and all the normal GA phases to select weights for the Neural Network. There is no back propagation of values. The Neura Network is just good or just bad. As one can imagine, this process takes a lot more steps to get to the same result.

Supervised

In Supervised Learning the Neural Network is given more information.
Not just 'how wrong' it was, but 'in what direction it was wrong' like 'Hunt the needle' but where you are told 'North a bit', 'West a b

So you get, and use, far more information in Supervised Learning, and this is the normal form of Neural Network learning algorithm Back Propagation (what this article uses, is Supervised Learning)

Learning Algorithm

In brief, to train a multi-layer Neural Network, the following steps are carried out:

- Start off with random weights (and biases) in the Neural Network
- Try one or more members of the training set, see how badly the output(s) are compared to what they should be (compared the target output(s))
- Jiggle weights a bit, aimed at getting improvement on outputs
- Now try with a new lot of the training set, or repeat again, jiggling weights each time
- Keep repeating until you get quite accurate outputs

This is what this article submission uses to solve the XOR problem. This is also called "Back Propagation" (normally called BP or BackProp)

Backprop allows you to use this error at output, to adjust the weights arriving at the output layer, but then also allows you to calcula the effective error 1 layer back, and use this to adjust the weights arriving there, and so on, back-propagating errors through any number of layers.

The trick is the use of a sigmoid as the non-linear transfer function (which was covered in Part 1. The sigmoid is used as it offers the ability to apply differentiation techniques.

$$y = g(x) = \frac{1}{1 + e^{-x}}$$

Because this is nicely differentiable —it so happens that

$$\frac{dg}{dx} = g'(x) = g(x)(1 - g(x))$$

Which in context of the article can be written as

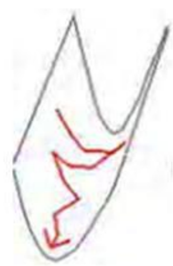
delta_outputs[i] = outputs[i] * (1.0 - outputs[i]) * (targets[i] - outputs[i])

It is by using this calculation that the weight changes can be applied back through the network.

Things To Watch Out For

Valleys: Using the rolled ball metaphor, there may well be valleys like this, with steep sides and a gently sloping floor. Gradient desc tends to waste time swooshing up and down each side of the valley (think ball!)

So what can we do about this. Well we add a momentum term, that tends to cancel out the back and forth movements and emphasizes any consistent direction, then this will go down such valleys with gentle bottom-slopes much more successfully (faster)

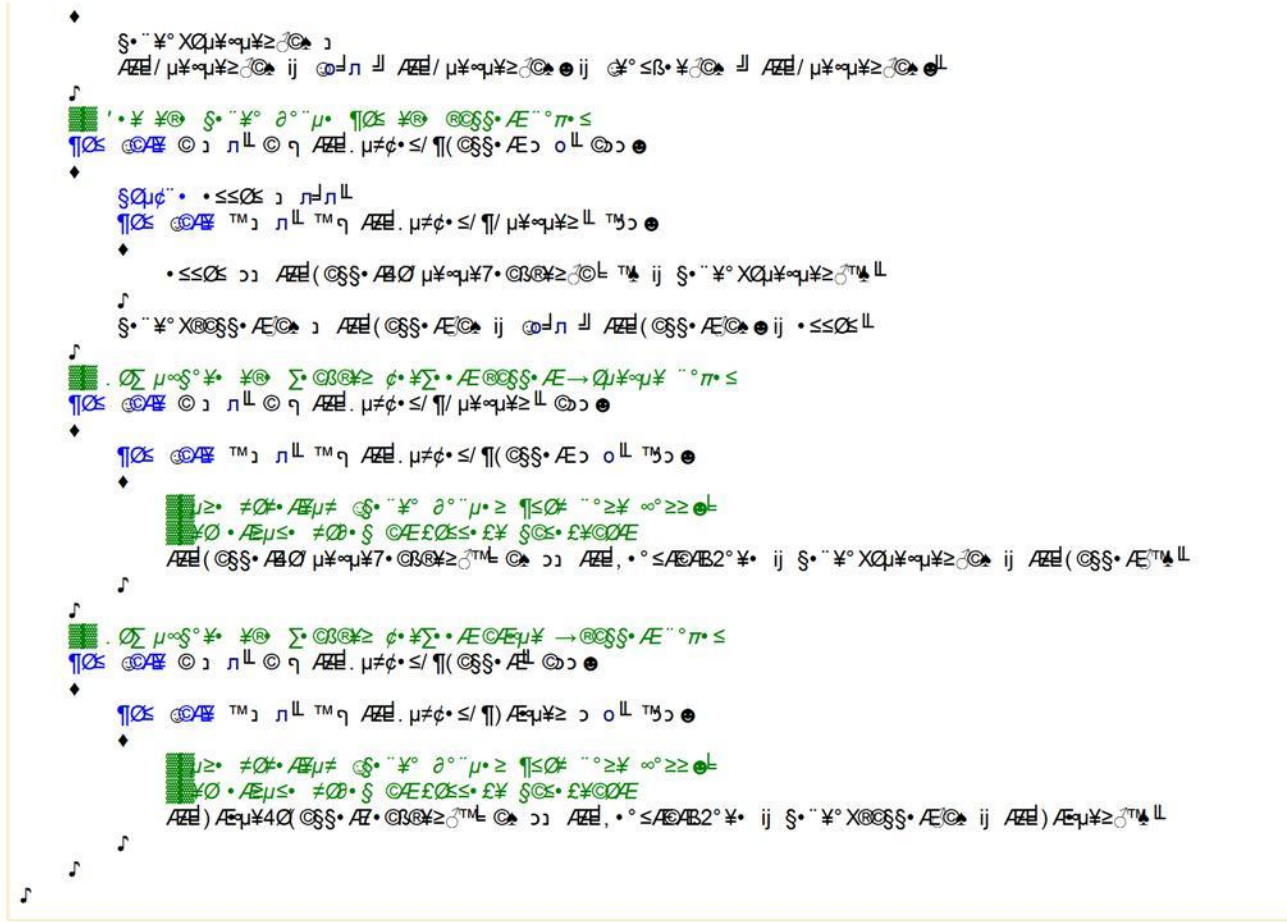


Starting The Training

This is probably best demonstrated with a code snippet from the article's actual code:

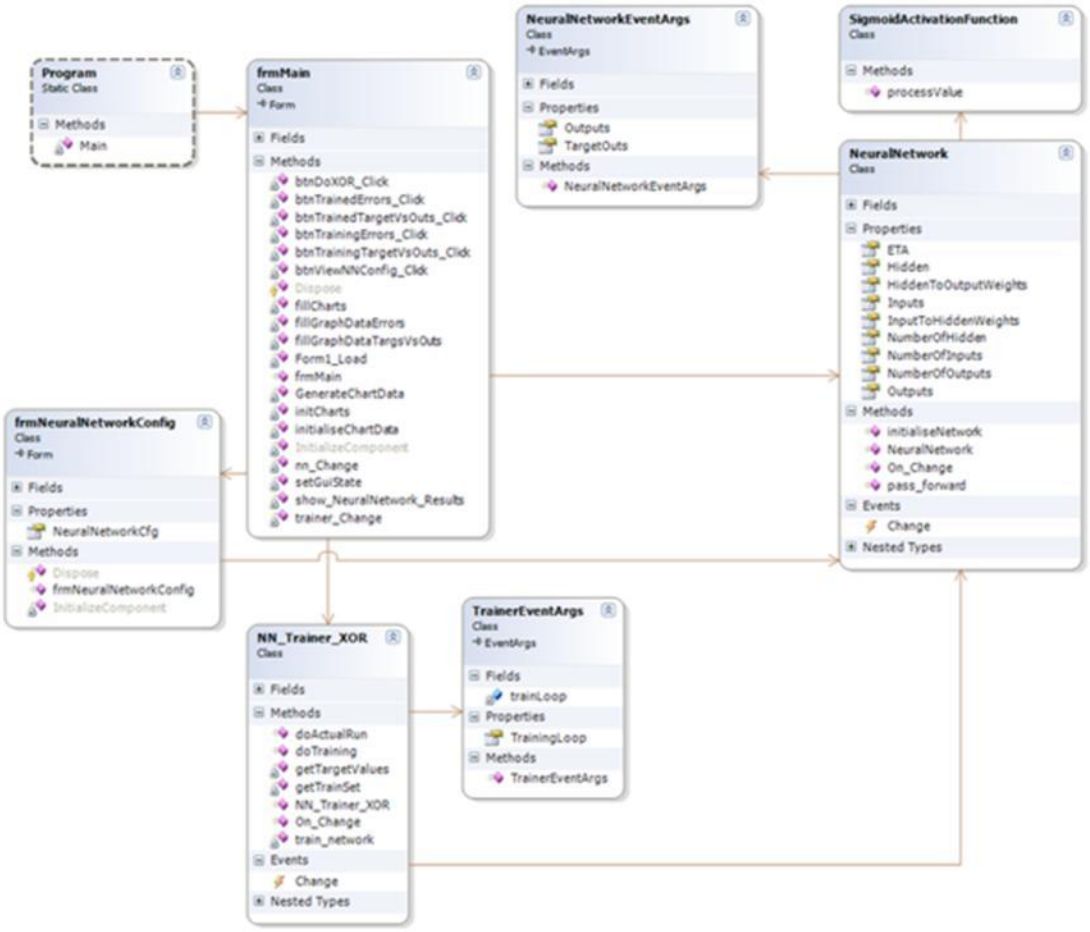
Hide ¡ Shrink ▲ ¡ Copy C

```
[illegible]
```



So Finally The Code

Well, the code for this article looks like the following class diagram (It's Visual Studio 2005 C#, .NET v2.0)



The main classes that people should take the time to look at would be :

- . . X4≤° ©Æ≤X8/ 2 : Trains a Neural Network to solve the XOR problem
- 4≤° ©Æ≤%∂• Æ¥! ≤ß≥ : Training event args, for use with a GUI
- . • µ≤° ¨ . • ¥∑Ø≤´ : A configurable Neural Network
- . • µ≤° ¨ . • ¥∑Ø≤´ %∂• Æ¥! ≤ß≥ : Training event args, for use with a GUI
- 3©ß≠Ø©§! £¥©∂° ¥©Ø®µÆ¥©Ø® A static method to provide the sigmoid activation function

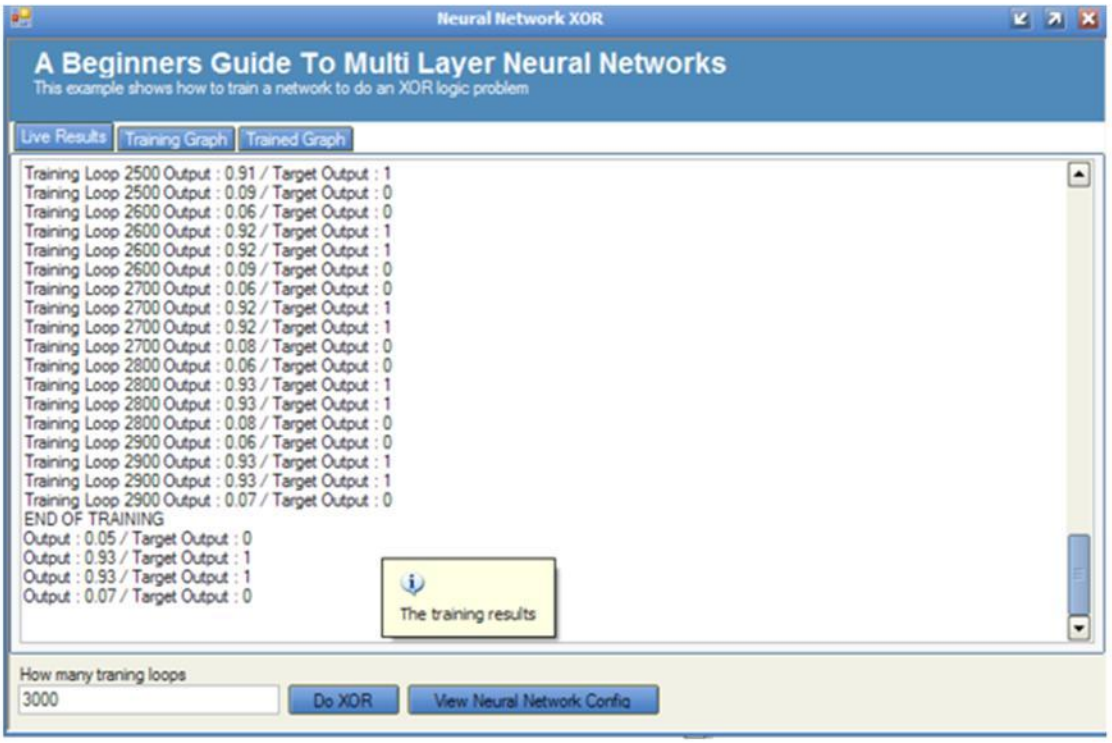
The rest are a GUI I constructed simply to show how it all fits together.

NOTE : the demo project contains all code, so I won't list it here.

Code Demos

The DEMO application attached has 3 main areas which are described below:

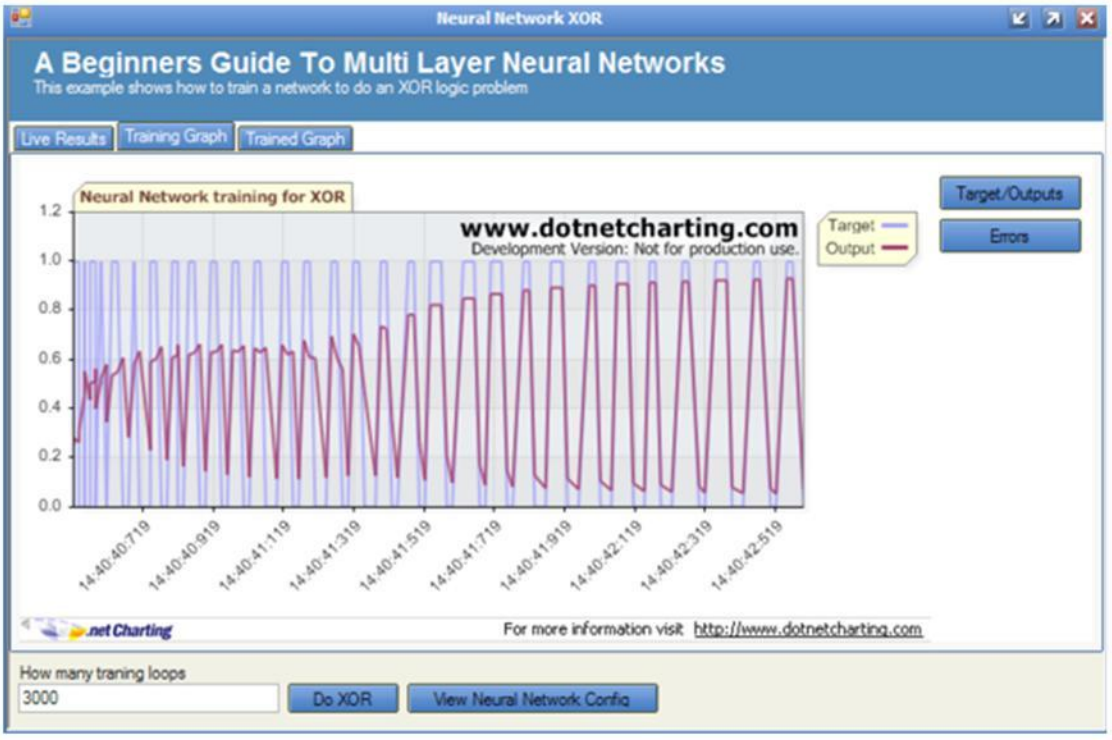
LIVE RESULTS Tab



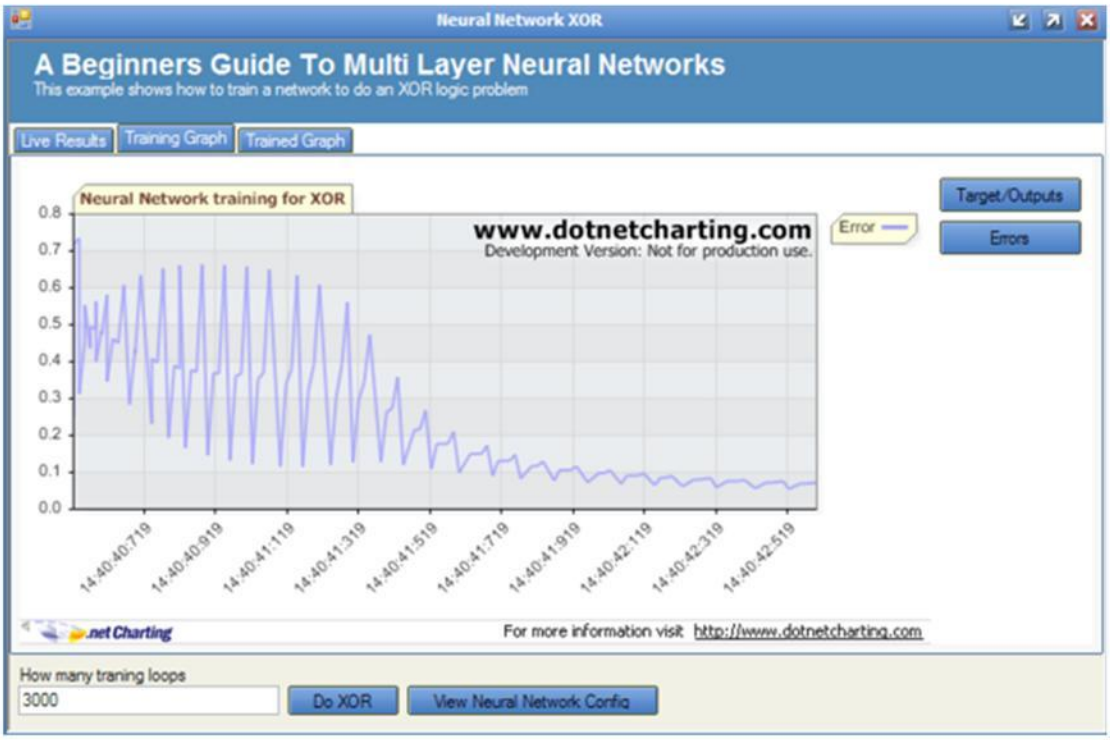
It can be seen that this has very nearly solved the XOR problem (You will probably never get it 100% accurate)

TRAINING RESULTS Tab

Viewing the training phase target/outputs together

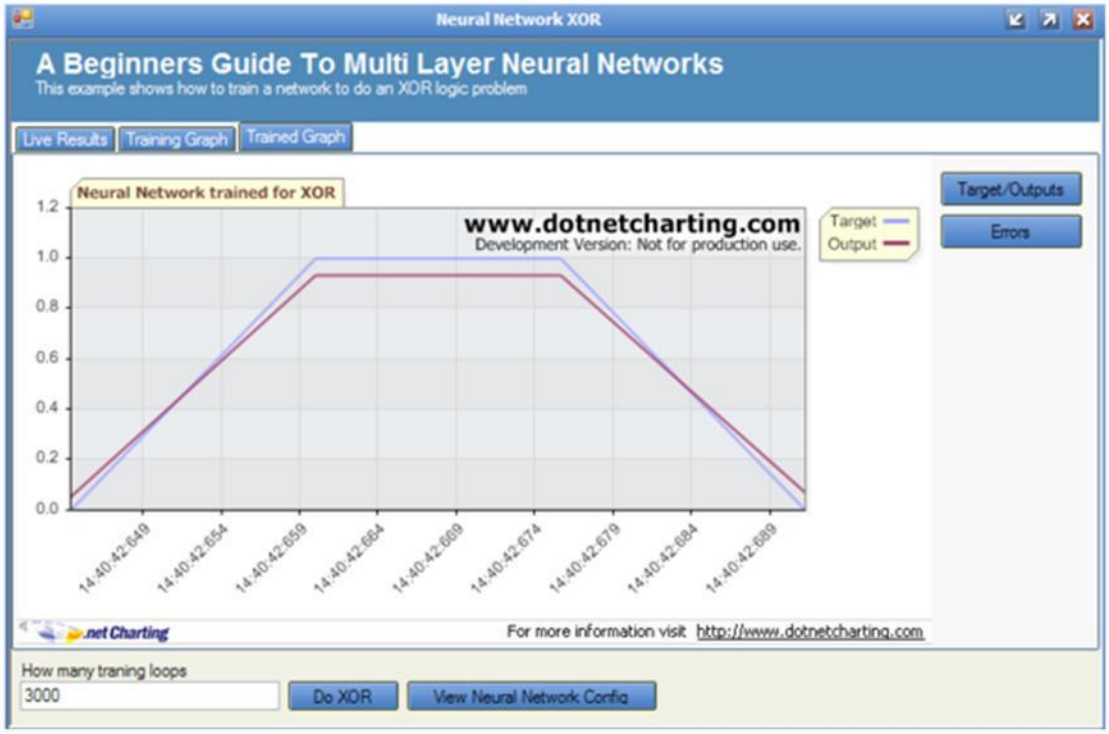


Viewing the training phase errors

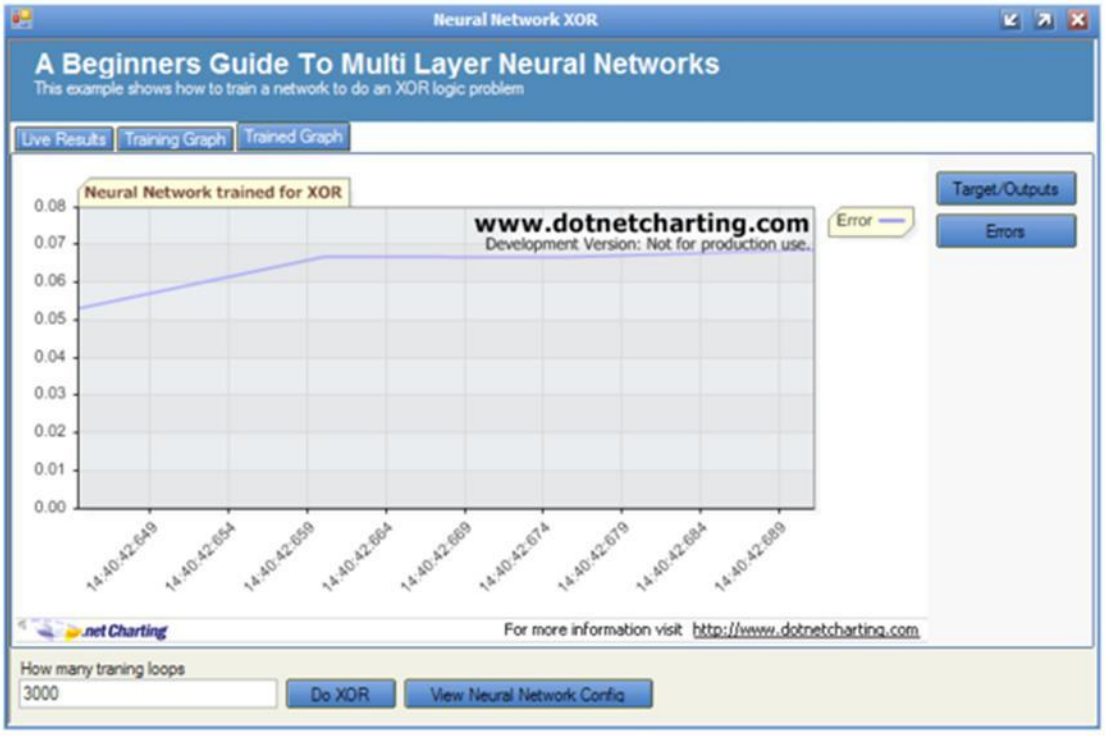


TRAINED RESULTS Tab

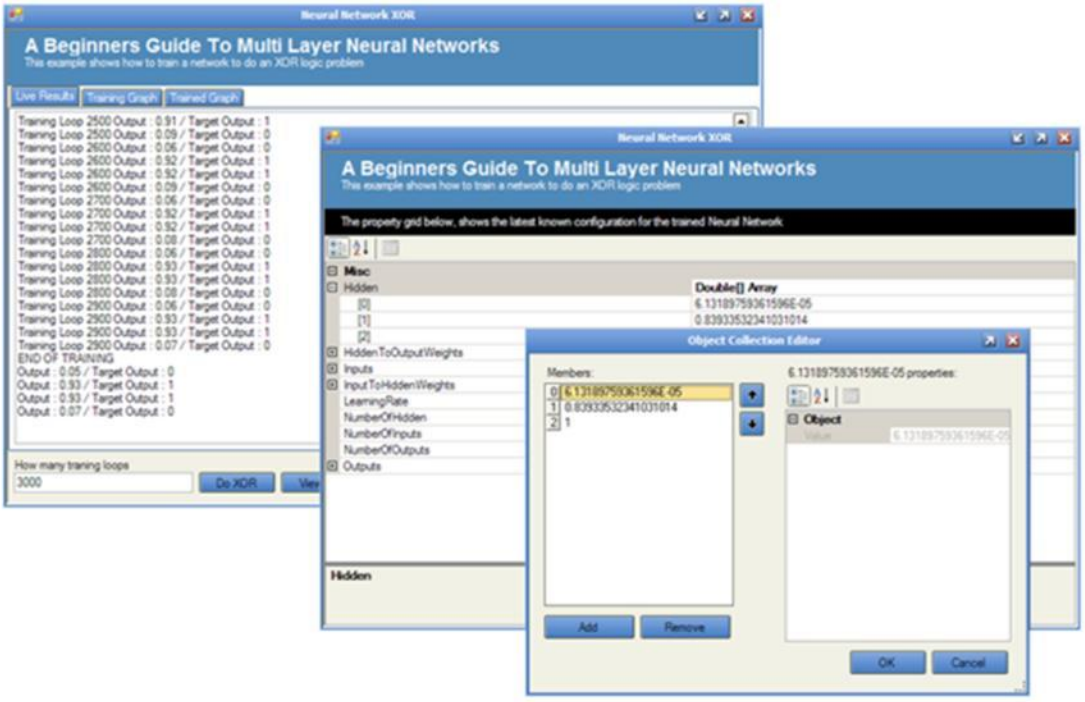
Viewing the trained target/outputs together



Viewing the trained errors



It is also possible to view the Neural Networks final configuration using the "View Neural Network Config" button. If people are interested in what weights the Neural Network ended up with, this is the place to look.



What Do You Think ?

That's it. I would just like to ask, if you liked the article, please vote for it.

I think AI is fairly interesting, that's why I am taking the time to publish these articles. So I hope someone else finds it interesting, an that it might help further someone's knowledge, as it has my own.

Anyone that wants to look further into AI type stuff, that finds the content of this article a bit basic should check out Andrew Krillov articles, at Andrew Krillov CP articles as his are more advanced, and very good. In fact anything Andrew seems to do, is very good.

History

- v1.0 24/11/06

Bibliography

- Artificial Intelligence 2nd edition, Elaine Rich / Kevin Knight. McGraw Hill Inc.
- Artificial Intelligence, A Modern Approach, Stuart Russell / Peter Norvig. Prentice Hall.

License

Share

About the Author

Sacha Barber

Software Developer (Senior)
United Kingdom

I currently hold the following qualifications (amongst others, I also studied Music Technology and Electronics, for my sins)

- MSc (Passed with distinctions), in Information Technology for E-Commerce

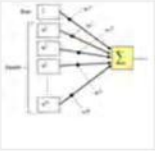
Both of these at Sussex University UK.

Award(s)

I am lucky enough to have won a few awards for Zany Crazy code articles over the years

- Microsoft C# MVP 2015
- Codeproject MVP 2015
- Microsoft C# MVP 2014
- Codeproject MVP 2014
- Microsoft C# MVP 2013
- Codeproject MVP 2013
- Microsoft C# MVP 2012
- Codeproject MVP 2012
- Microsoft C# MVP 2011
- Codeproject MVP 2011
- Microsoft C# MVP 2010
- Codeproject MVP 2010
- Microsoft C# MVP 2009
- Codeproject MVP 2009
- Microsoft C# MVP 2008
- Codeproject MVP 2008
- And numerous codeproject awards which you can see over at my blog

You may also be interested in...



AI: Neural Network for Beginners (Part 3 of 3)

Speed Up Your Git Repository: Introducing Git-Over-FASP

AI : Neural Network for beginners (Part 1 of 3)

Taking COBOL mobile

AI Life

COBOL programmers: Skill up and save timeÀ

Comments and Discussions

You must Sign In to use this message board.

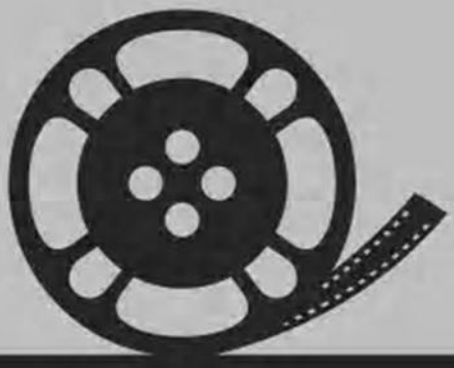
Search Comments

Profile popups Spacing Relaxed Layout Normal Per page 25 Upda

First Prev Nex

help	**Member 11265993**	**26-Nov-14 22:**
Sigmoid function	**Member 10973839**	**26-Jul-14 23:**
Sigmoid function	**Member 10973839**	**26-Jul-14 23:**
Takes a long time to converge	**Member 9401821**	**2-Mar-14 0:**
Re: Takes a long time to converge	LudemeGames	2-Mar-14 4:
I have a question,thank you for telling me .	**fengyelan**	**16-Apr-13 22:**
My vote of 5	**Nickydo**	**10-Sep-12 2:**
Part 3?	**Mauro Leggieri**	**5-Apr-09 7:**
Re: Part 3?	Sacha Barber	5-Apr-09 9:
Re: Part 3?	Mauro Leggieri	6-Apr-09 3:
About parameterizing the 'momentum' factor	**mahabir**	**23-Sep-08 20:**
Re: About parameterizing the 'momentum' factor	Sacha Barber	23-Sep-08 22:
Re: About parameterizing the 'momentum' factor	Sacha Barber	23-Sep-08 22:
Re: About parameterizing the 'momentum' factor	ramesh0285	26-Nov-12 18:
part 1	**gholamabbas Sayyad**	**18-Sep-08 21:**
Re: part 1	Sacha Barber	18-Sep-08 22:
Solution for getTrainSet(int idx)	**DKHVC**	**16-Apr-08 21:**
[Message Deleted]	**Danny Rodriguez**	**27-Jan-08 10:**
Hello	**MohamadJaber**	**11-Dec-07 0:**
Erratic Bahaviour?	**rampantandroid**	**15-Oct-07 18:**
Re: Erratic Bahaviour?	rampantandroid	15-Oct-07 19:
Small Suggestion	**dfhgesart**	**28-Jul-07 16:**
Re: Small Suggestion	Sacha Barber	29-Jul-07 0:
Excellent!	**merlin981**	**17-May-07 5:**
license?	**famousj.dejazzd.com**	**17-Jan-07 9:**

Wednesday, December 30, 2015
2:46 AM



Visual Studio 2013

Succinctly

by Alessandro Del Sole

Syncfusion | Technology Resource Portal

Visual Studio 2013 Succinctly

By
Alessandro Del Sole

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Jeff Boenig
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Alessandro Del Sole has been a Microsoft Most Valuable Professional (MVP) since 2008. Awarded MVP of The Year in 2009, 2010, 2011, and 2012, he is internationally considered a Visual Studio expert. Alessandro has authored six printed books and three e-books on programming with Visual Studio, including *Visual Basic 2012 Unleashed*, *Visual Studio LightSwitch Unleashed*, *Hidden Visual Studio LightSwitch*, *Hidden WPF*, and *Visual Basic 2010 Unleashed*. He has written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and in English for many developer portals, including the Visual Basic Developer Center from Microsoft. He is a frequent speaker at Italian conferences, and he has released a number of apps for Windows Phone and Windows 8. He has also produced a number of instructional videos both in English and Italian. You can follow him on Twitter at @progalex.

Introduction

Microsoft Visual Studio 2013 is the new version of the popular integrated development environment for building modern, high-quality applications for a number of platforms such as Windows, the web, Microsoft cloud, tablets running Windows 8, and Windows Phone devices.

The key word in Visual Studio 2013 is "productivity." Microsoft well knows that developers spend most of their time writing code, so the new version adds many tools to increase productivity and to help developers be faster and more efficient. The .NET Framework 4.5.1 does not introduce any new features to managed languages such as Visual Basic and Visual C#; on the other side, lots of enhancements have been made to the integrated development environment.

In this book you will learn what's new in Visual Studio 2013 for the code editor, for the debugger, for Windows 8.1, for the web and the cloud (including the new integrated support for Windows Azure subscriptions), and much more. There are so many improvements to support new and updated technologies that you will easily understand why a new release of Visual Studio was important after only one year.

Visual Studio 2013 ships with the following editions: Ultimate, Premium, Professional, and Test Professional, plus the free Express editions. Most features described in this book require Visual Studio 2013 Professional, but some of them require Visual Studio 2013 Ultimate, which is the most complete edition available. I will specify where the Ultimate edition is required. For a full comparison, you can look at this page in the Visual Studio portal from Microsoft. You can also download a fully-functional, 90-day trial of Visual Studio 2013 Ultimate (and other editions) from the Visual Studio Downloads page. The Express Editions are lightweight, free of charge editions of specific development environments for non-professional developers, hobbyists, and students that can be used even for commercial purposes.

Available products are Visual Studio 2013 Express for Windows Desktop (which you use to build WPF, Windows Forms, and Console apps), Visual Studio 2013 Express for Windows (which you use to build Windows Store apps for Windows 8.x), and Visual Studio 2013 Express for Web (which you use to build apps and sites for the web and the cloud). You can download the Express Editions from the same download page as above.

It is worth mentioning that Visual Studio 2013 allows building apps for Windows Phone 8, but not for Windows Phone 7.x. If you still need to build apps for Windows Phone 7.x, you will need to use Visual Studio 2012 and the Windows Phone 7.1 SDK. Visual Studio 2013 can be safely installed side-by-side with Visual Studio 2012. Also, Visual Studio 2013 allows opening most Visual Studio 2012 projects without modifying files for a perfect backward compatibility. A full list of conversion scenarios is provided in the MSDN documentation.

Chapter 1 Synchronized Settings and Notifications

Most developers work on different computers, such as desktop workstations, laptops, and servers. In most situations, developers install Visual Studio onto each computer they work with. As you know, the IDE (integrated development environment) is customizable and allows adjusting a number of settings, such as adding buttons to toolbars, changing colors, using different fonts, and so on. Before Visual Studio 2013, you had to adjust settings manually on every installation of Visual Studio, which requires more time and the risk of forgetting to change some settings. Visual Studio 2013 introduces *synchronized settings*, so that every time you make customizations in the environment, these will be automatically applied to the other installations of Visual Studio on different computers. This chapter explains how this new feature works and how you can customize your work environment just once.

Sign in to Visual Studio

The first time you start Visual Studio 2013, you will be asked to specify a default profile, such as web programming, Visual Basic programming, general development, and other profiles from previous versions of the IDE. This is a step you've already taken many times, so I will not spend much time here. After selecting the profile, Visual Studio 2013 will ask you to sign in with a Microsoft Account (formerly known as Windows Live ID). A Microsoft Account is an email address based on one of the Microsoft providers such as Hotmail, Live, or Outlook. Figure 1 demonstrates this.

Figure 1: Visual Studio 2013 asks you to sign in with a Microsoft Account.

Signing in with a Microsoft Account is not mandatory; you will certainly be able to use Visual Studio without an email address. However, signing in is advantageous for the following reasons:

- You can take advantage of Synchronized Settings, as described later in this chapter.
- Signing in with a Microsoft Account permanently unlocks any Visual Studio Express you have installed.
- You will be automatically logged in to the Team Foundation Service account associated with your email address if you subscribed to this service.
- You can use a trial version of Visual Studio for 90 days instead of 30.
- Signing in will unlock Visual Studio if your Microsoft Account is associated with an MSDN subscription.

Assuming you already have a Microsoft Account, click **Sign in**. At this point you will be asked to enter your email address and password, as represented in Figure 2.

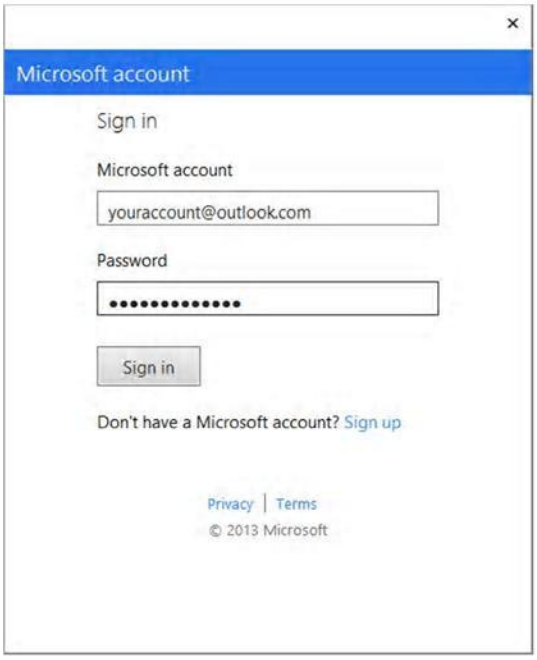


Figure 2: Enter your credentials to get started.

Click **Sign in**. At this point Visual Studio will recognize your profile and will show some information while preparing the environment for the first use (see Figure 3).

Tip: If you are installing Visual Studio 2013 for the first time on a computer but you already installed it onto a different machine, this is also the moment in which settings are synchronized. Information on how settings are synchronized is coming shortly.

Figure 3: Enter your credentials to get started.

Visual Studio will also ask you to select one of the available development settings. If you have a previous version installed, such as Visual Studio 2012, the new IDE provides an option to import customizations from the previous version. You can choose from among General, JavaScript, SQL Server, Visual Basic, Visual C#, Visual C++, Visual F#, Web Development, and Web Development (Code Only). Choose the one that is closest to your interest. If you do not know what the best choice for you is, simply choose the General settings. Also, you will be able to select one of the available graphic themes (Light, Dark, Blue). Once signed in, Visual Studio shows your profile name and picture at the upper right corner of the IDE, including shortcuts to access your profiled detailed information and to connect to Team Foundation Service.

Note: Team Foundation Service is a cloud-based version of Team Foundation Server, the popular Microsoft platform for team collaboration. With Team Foundation Service you can host team projects and take advantage of source control and other team development tools wherever you are. Another important reason for signing into Visual Studio with a Microsoft Account is that the IDE automatically connects your account to the associated Team Foundation Service account. To create your Team Foundation Service account, visit http://tfs.visualstudio.com.

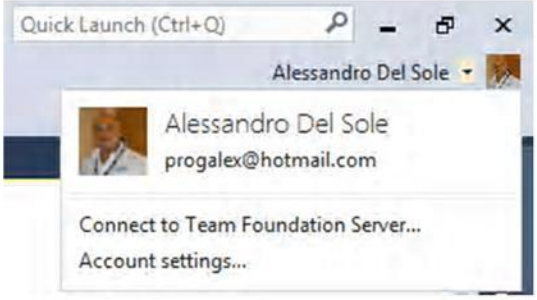


Figure 4: The IDE recognizes the user and provides useful shortcuts.

Once you have signed in, your settings are ready to be synchronized and shared across multiple computers.

Synchronized settings

By default, Visual Studio 2013 can synchronize the following settings:

- Development settings. These are related to the development profile selected the first time you ran Visual Studio and can be changed by selecting Tools, Import and Export Settings, Reset All Settings.
- Theme settings, available in Options, Environment page, General tab
- Startup settings available in Options, StartUp
- All settings for the text editor available in Options, Text Editor
- All settings for fonts and colors available in Options, Environment, Fonts and Colors
- All default and custom keyboard shortcuts, available in Options, Environment, Keyboard
- Customized command aliases

Tip: Command aliases are a way to enter commands inside the Command window and allow opening dialogs or launch other tasks in Visual Studio, instead of using menus and menu items. The full list of built-in aliases is available at: http://msdn.microsoft.com/en-us/library/c3a0kd3x.aspx

When you make changes or customizations, Visual Studio stores the aforementioned settings on the cloud, that is, on Microsoft servers, and associates those settings to the Microsoft Account you used to sign in. When you sign into Visual Studio with the same account on a different computer, the IDE downloads settings associated to your account and applies them to the active environment. Developers have been requesting this feature for a long time and finally Visual Studio 2013 solves the problem. This is just another example of how the cloud can make your life easier as a developer. Remember that synchronization works even if you have different editions of Visual Studio 2013, such as Ultimate, Premium, and Professional. Synchronization also applies to Express Editions, but it does not work if you have Express and non-Express editions on the same machine.

Selective synchronization

You can disable settings synchronization or choose what you want to synchronize among the settings listed in the previous paragraph. To accomplish this, go to the **Tools** menu, then select the **Options** submenu, then the **Synchronized Settings** command. Figure 5 shows how the Options dialog appears at this point.

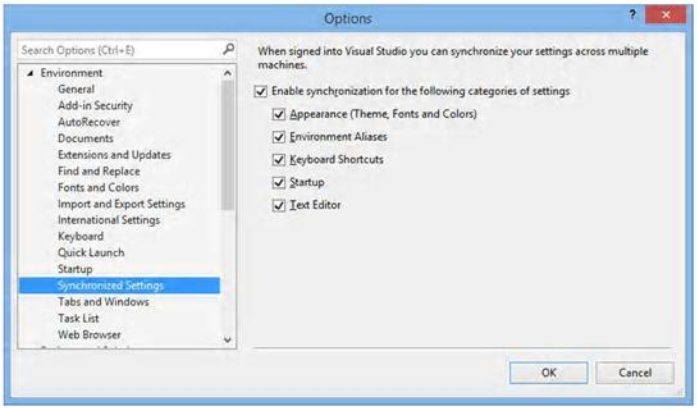


Figure 5: Enabling Synchronization and Settings Selection

The **Enable synchronization for the following categories of settings** check box is selected by default. If you unselect it, synchronization will stop until you explicitly re-enable it. You can also select one or more specific settings you want to synchronize, excluding settings you do not use. Click **OK** to apply your changes.

Synchronization conflicts

For various reasons, synchronization across multiple machines can occasionally fail. If this happens, Visual Studio shows a message in the Notification Hub (see the next topic of this chapter). The MSDN documentation describes three possible solutions with some manual work. As a personal suggestion, turn synchronization off on the computer where it was not successful, and then sign out. Next, sign in again and turn synchronization on again.

Notifications Hub

Visual Studio 2013 introduces a new concept of notifications. The goal is keeping the developer informed about product updates, extension updates, documentation updates, license issues, problems with the Microsoft Account, unresolved conflicts, and other errors. The IDE presents notifications to you via the Notifications Hub. The Notification Hub consists of a new tool window called Notifications and of a small flag icon (the Notifications button) placed near the Quick Launch bar, indicating the number of available notifications. To open the Notifications window:

1. Click the **View** menu.
2. Select the **Notifications** entry.

You can also click the **Notifications** button on the Quick Launch bar for faster opening. Figure 6 shows the Notification Hub.

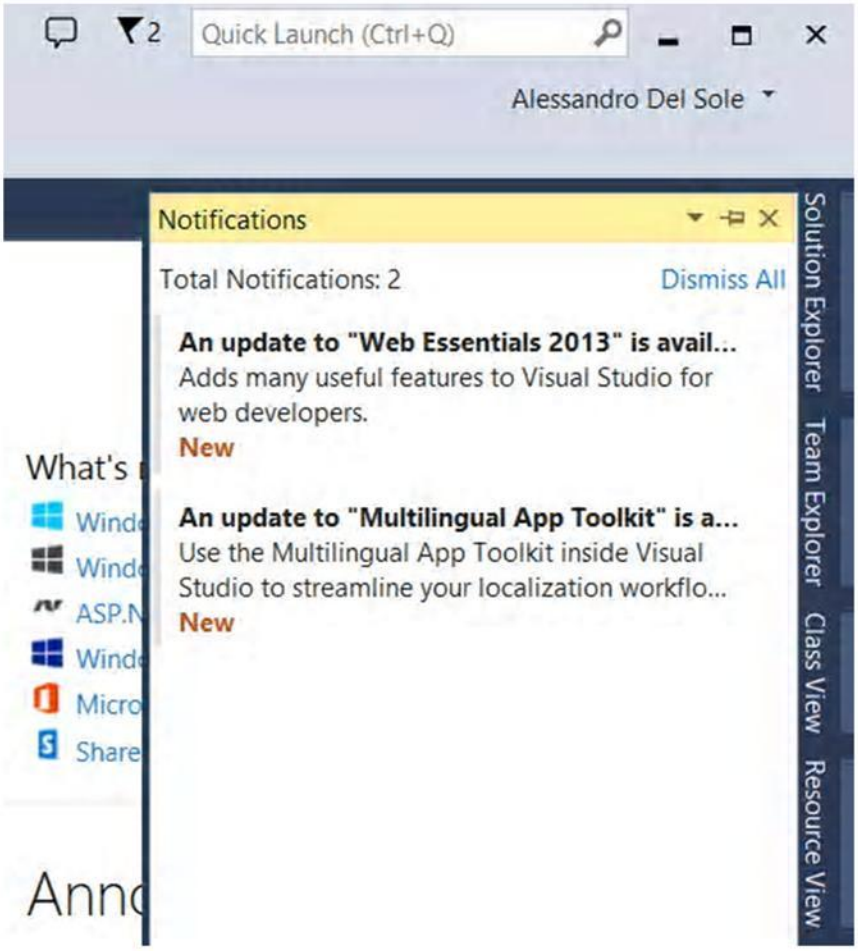


Figure 6: Enabling Synchronization and Settings Selection

If you click the button, the Notifications tool window opens and shows a full list of notifications. You can expand the notification description and, in case the notification is about an update, you will be able to click a hyperlink that will redirect you to the download page. You can also ignore all notifications by clicking **Dismiss All**.

Chapter summary

Among the new features in the IDE, Visual Studio 2013 makes it easy to share settings across multiple computers with Synchronized Settings; with this feature, most settings are saved to the cloud and applied to all of your other installations of Visual Studio. The Notifications Hub provides an easy way to download updates and to present information about other issues related to Visual Studio. Both features require you to sign into Visual Studio with your Microsoft Account, which also allows connecting to other Microsoft services without additional effort.

Chapter 2 The Start Page Revisited

The Start Page has been an important place in Visual Studio since the early days. In the first versions of Visual Studio .NET, it was a static page containing shortcuts for creating new or opening existing projects, and a place to get the latest announcements from Microsoft. In Visual Studio 2010, the Start Page was completely redesigned; it was built upon Windows Presentation Foundation (WPF), providing not only a better integration with the IDE but also offering an opportunity to build completely customized entry points. In Visual Studio 2013 the Start Page has evolved even more, becoming the place where you start your work as well as learn about new and updated technologies.

A new Start experience

An important concept behind the development experience in Visual Studio 2013 is that programmers should have everything they need inside the active page. The Start Page in Visual Studio 2013 has been reorganized based on this concept and includes not only shortcuts for working with projects, but also updated links to learning resources and announcements, all in one place. The Start Page has a dynamic layout, meaning that items inside the page are automatically rearranged when you resize the Visual Studio's window. Figure 7 shows how the Start Page appears when you run Visual Studio 2013.

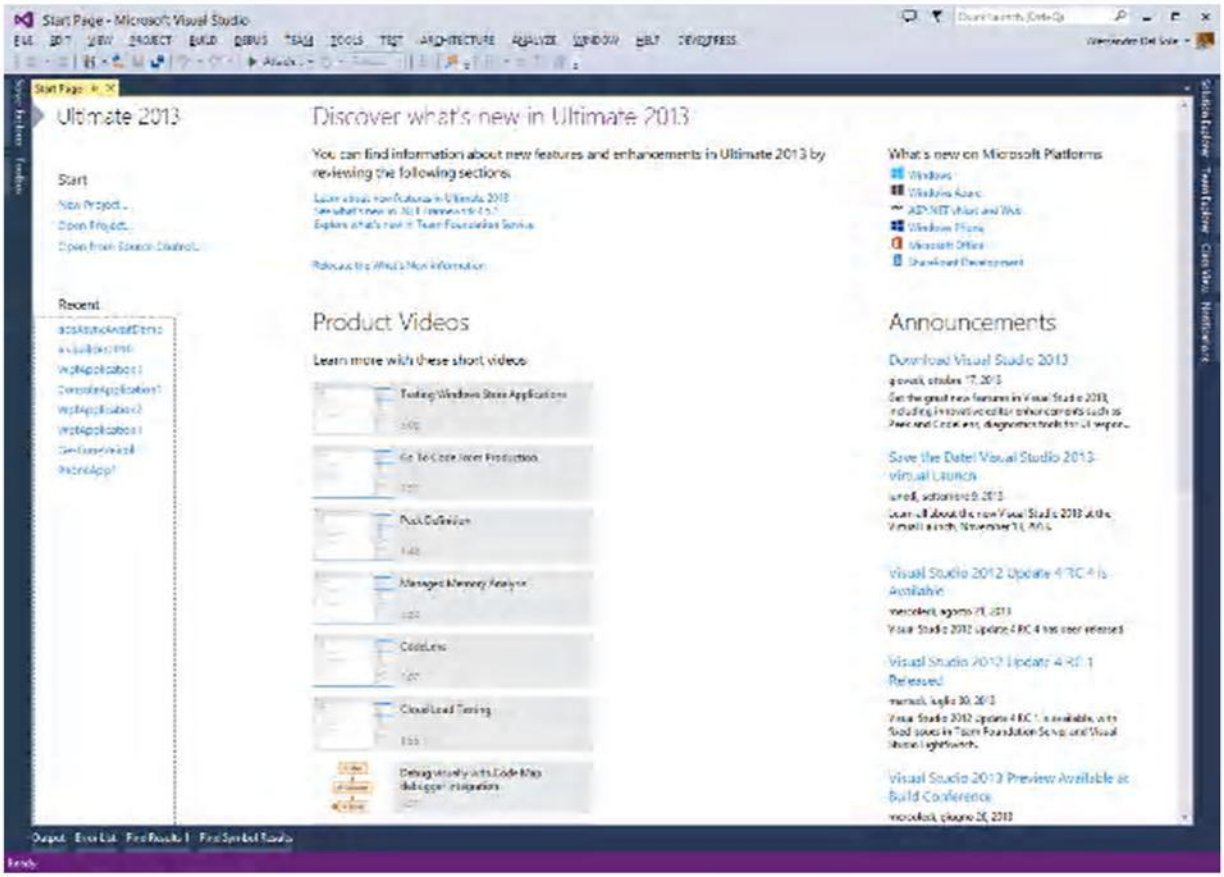


Figure 7: The Start Page in Visual Studio 2013

The Start Page is made of several areas, each described in the next sections of this chapter. Of course, you can still create and use custom start pages based on WPF (as you could do in Visual Studio 2010 and 2012) or you can disable the Start Page and choose a different entry point by using Tools, Options, Startup.

Note: This chapter does not cover how to build custom start pages. If you wish to create your own start page, read Start Pages in the MSDN documentation.

What you find here is a description of what the Start Page contains and how it can make your life easier.

Work with projects

Tip: In this and the next subsections, use Figure 7 as a reference to locate items in the Start Page.

The first thing you probably do when you launch Visual Studio is open a project. On the left side of the Start Page you will find two areas related to working with projects, Start and Recent. Start contains shortcuts for creating new projects or opening existing ones, including from source controls platforms such as Team Foundation Server, Team Foundation Services, and GIT. Recent shows a list of recent projects; if you right-click the name of a project in the recent list, you will be able to open the project, open the containing folder, or remove the project from the list.

Staying up to date: Announcements

The Announcements area shows news about product updates, new releases, events/conferences, and technical content from the various teams in Redmond working on Visual Studio. This is not new in the Start Page, but the behavior is different. First, you can no longer customize the source of the announcements; in the previous versions of Visual Studio you could specify a different RSS feed to show contents, but now the news channel cannot be changed. However, the news channel is now filtered with information that you actually need to stay up to date with new releases and with events focused on Visual Studio 2013.

Learning

The Start Page now has more content for getting started with Microsoft technologies and with specific product features, as described in this section.

What's new on Microsoft platforms

The **What's new on Microsoft platforms** area has shortcuts that make it easier to access the MSDN documentation for each of the most recent development platforms, operating systems, and collaboration platforms, such as Windows 8, Windows Azure, the web and ASP.NET, Windows Phone, Office, and SharePoint.

Product Videos

The **Product Videos** area allows watching short instructional videos about specific features in the Visual Studio IDE. This is very useful for a better understanding of most of the new features, because the videos show them in action with practical examples. You might see the following text:

```
We have a lot of great content to show you, but we need your permission to get it and
keep it updated.
```

If you see this message, you need to click **Tools**, then click **Options**, and select **Startup** under the **Environment** node in the **Options** dialog; finally, check the **Download content every** check box. The default time interval is 60 minutes but you can increase or decrease the value. The reason for this is that Visual Studio uses your Internet connection to retrieve the list of available contents, so it needs your permission first.

Discover what's new

At the top of the Start Page you can find an area that offers shortcuts to learn what new features are available in Visual Studio 2013, the .NET Framework 4.5.1, and Team Foundation Services. Such shortcuts will direct you to the appropriate page of the MSDN documentation.

Chapter summary

With its revisited and dynamic layout, the Start Page in Visual Studio 2013 is more than a simple place where you create new projects or pick up existing ones; the Start Page is now the place where you can easily find all the learning resources and product releases you need to start building applications for the most recent Microsoft platforms.

Chapter 3 Code Editor Improvements

The code editor in Visual Studio 2013 is one of the areas of the IDE where Microsoft made many investments. The goal is to make developers stay focused on the code they are writing, helping them perform common tasks more quickly and save time. This chapter describes new features in the code editor that will help you be more productive when writing code.

Peek Definition

Peek Definition is a new feature that you can use to see and edit the definition of a class or class member inside a popup shown within the active code editor window. This helps you avoid the need to leave the active window in order to open the code file that contains the code block you need to edit. To understand how it works, create a new Console application then add a **`Person`** class like the following.

Visual C#

```
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime DateOfBirth { get; set; }

        public override string ToString()
        {
            return string.Format("{0} {1}, born {2}",
                this.FirstName, this.LastName,
                this.DateOfBirth.ToShortDateString());
        }
    }
```

Visual Basic

```
Public Class Person
    Public Property FirstName As String
    Public Property LastName As String
    Public Property DateOfBirth As Date

    Public Overrides Function ToString() As String
        Return String.Format("{0} {1}, born {2}",
            Me.FirstName, Me.LastName,
            Me.DateOfBirth.ToShortDateString())
    End Function
```

```
End Class
```

The **Main** method of the sample application simply creates a new instance of the **Person** class and assigns some values as in the following code.

Visual C#

```
    class Program
    {
        static void Main(string[] args)
        {
            Person person = new Person();

            person.FirstName = "Alessandro";
            person.LastName = "Del Sole";
            person.DateOfBirth = new DateTime(1977, 5, 10);

            Console.WriteLine(person.ToString());
            Console.ReadLine();
        }
    }
```

Visual Basic

```
Module Module1
    Sub Main()
        Dim person As New Person

        person.FirstName = "Alessandro"
        person.LastName = "Del Sole"
        person.DateOfBirth = New DateTime(1977, 5, 10)

        Console.WriteLine(person.ToString())
        Console.ReadLine()
    End Sub
End Module
```

Now suppose you want to make some edits to the **Person** class, such as renaming members or adding new ones. Right-click the type name (**Person** in our example) and select **Peek Definition**. As you can see from Figure 8, a pop-up appears showing the code of the **Person** class and the name of the code file where it is defined.

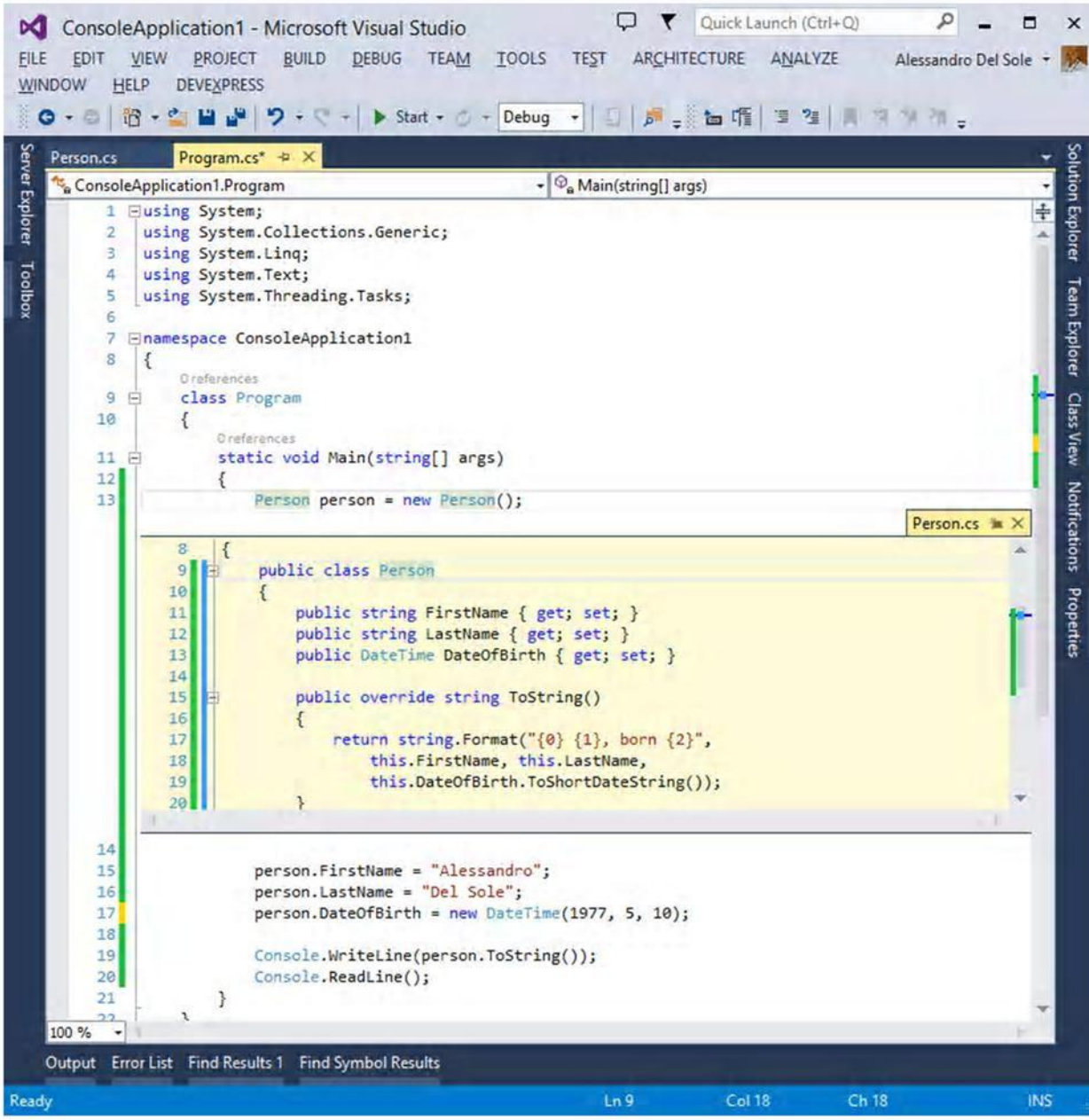


Figure 8: Editing Code with Peek Definition

Peek Definition offers a fully functional editor, so you can change your class (or member) definition according to your needs without leaving the active window. If you make any changes, these are immediately visible in the code that uses the class. When done, you can simply click the usual Close button to hide the Peek Definition content. This tool is a very useful addition, because it not only allows you to stay in the active editor window while making changes to a type, but it also makes it easier to find type definitions among millions of lines of code and thousands of code files.

CodeLens

Note: This feature is available only in the Ultimate edition.

In many situations, you might need to know how many times an object has been used in your code and where. The previous (and current) versions of Visual Studio provide a tool called **Find All References**, which shows a list of references to an object inside a tool window called Find Symbols Results that you can invoke by right-clicking an object's name in the code editor and then selecting **Find All References**. Visual Studio 2013 makes a step forward, offering an additional integrated view of code references called **CodeLens**. Take a look at Figure 9, which shows the **Person** class definition inside the code editor.

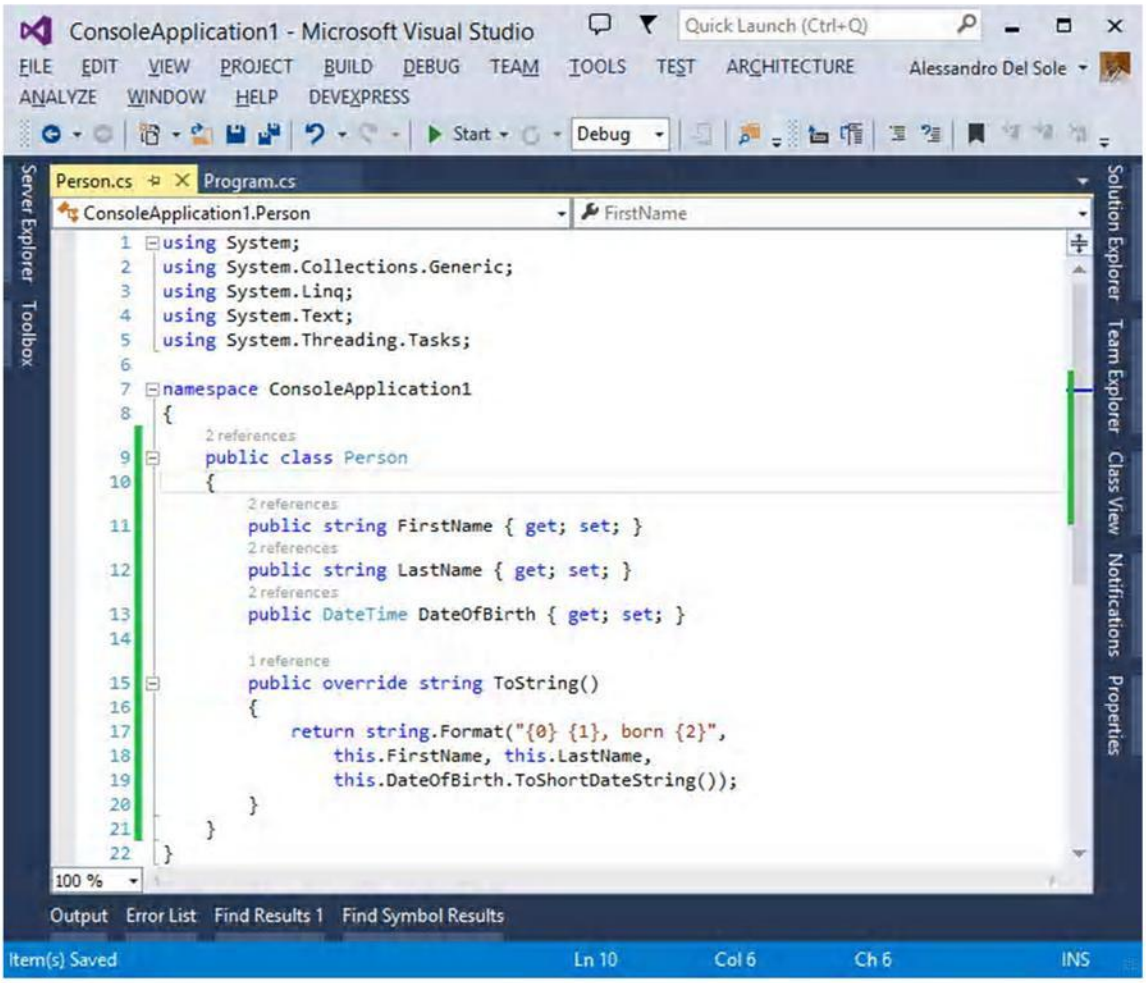


Figure 9: Visual Studio 2013 shows the number of references for each type and member.

As you can see, above each type and member name Visual Studio shows the number of references. If you click that number, a tooltip will show where the object is used; if you pass the mouse pointer over the line of code that contains the reference, another tooltip will show the full code block containing the reference (see Figure 10).

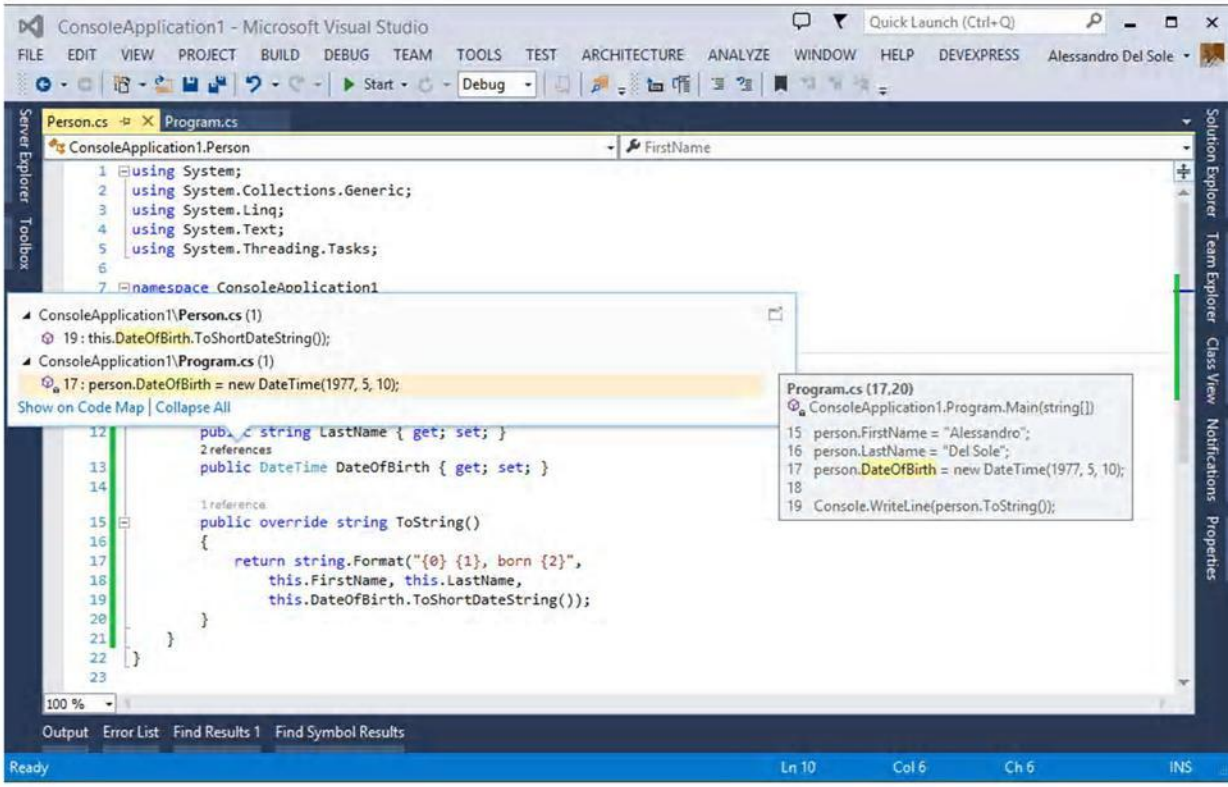


Figure 10: Finding Object References from within the Code Editor

CodeLens also shows the containing code file for each reference and the line number where the object is used, and allows fast navigation to the reference by double-clicking the line of code in the tooltip. Actually CodeLens does also an amazing job when your solution is under the source control of Team Foundation Server; in fact, it can show information about unit tests, passed tests, failed tests, and edits made by other team members.

Enhanced Scroll Bar

In most of real world projects, code files are made of hundreds of lines of code, so finding specific code blocks inside a file can become difficult. In order to make it easier to browse very long code files, Visual Studio 2013 provides an improved scroll bar in the code editor window, known as enhanced scroll bar. Basically the scroll bar can show a map of the code (map mode) so that when you move the mouse pointer up and down, a magnifier shows a preview of the code block. This is very useful with long code files, if you want to see some code definition without jumping from one position to another in the code file.

To enable the map mode, right-click the scroll bar, then select **Scroll Bar Options**. In the **Options** dialog, locate the **Behavior** group and then select the **Use map mode for vertical scroll bar** option, as shown in Figure 11.

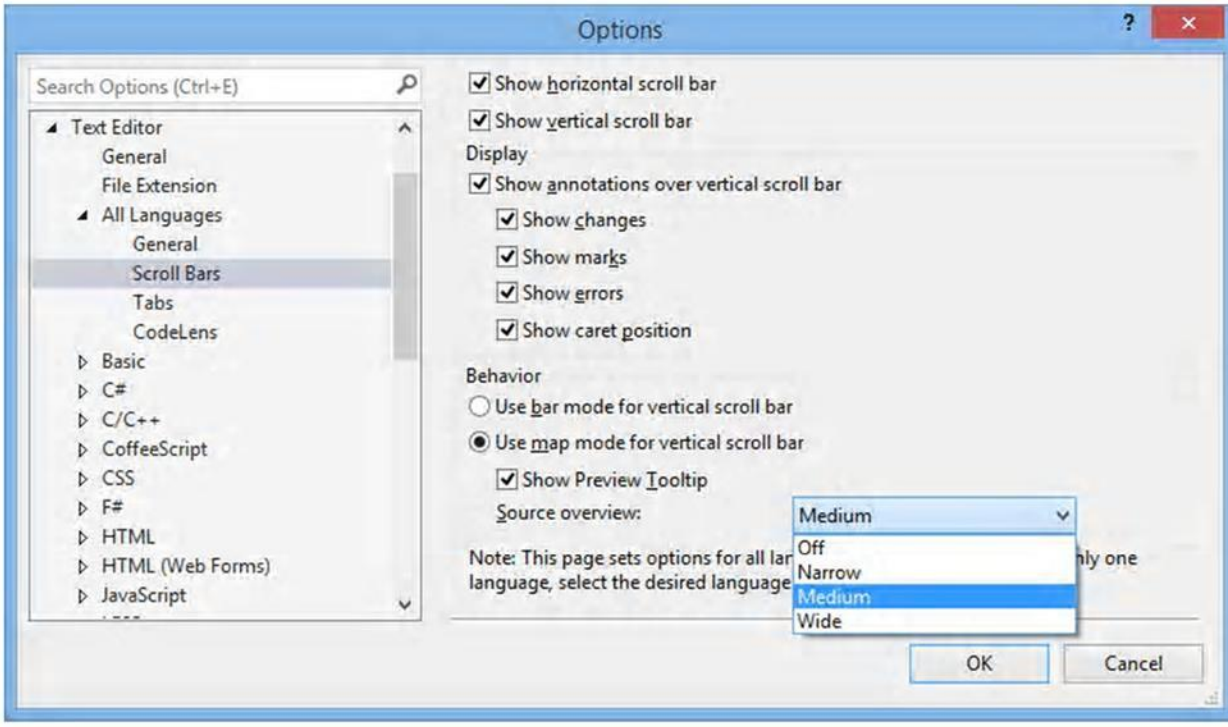


Figure 11: Enabling the Map Mode for the Scroll Bar

Note: Enabling the code preview is optional, but I encourage you to leave it selected. After all, it's the real benefit of this tool.

You can also choose the size of the map by changing the value of the **Source overview** box. The default value is Medium, which is a good choice for most situations. Click **OK** to enable the map mode. When you go back to the code editor, you can see the scroll bar's new look. Figure 12 shows an example.

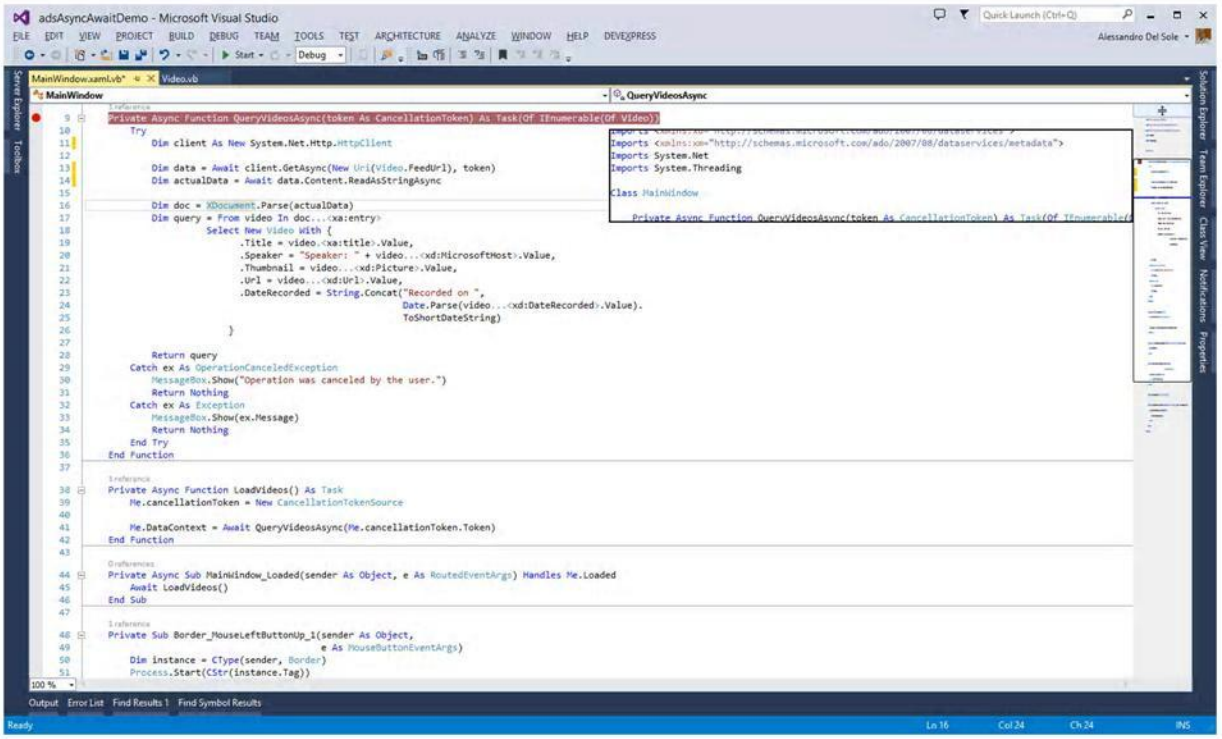


Figure 12: Browsing Code with the Scroll Bar in Map Mode

When you move the mouse pointer over the scroll bar, a tooltip shows a preview of the code for the current position on the map. A blue line indicates the cursor position, yellow dots indicate edited lines of code, and red dots represent breakpoints. You can simply revert to the classic scroll bar by going back to the scroll bar options and selecting the **Use bar mode for vertical scroll bar** option.

Tip: The enhanced scroll bar works with all languages supported by Visual Studio 2013. This means that when you enable the map mode, the scroll bar will show the map for every code file in any language until you disable it again.

Navigate To

Another key feature of Visual Studio 2013 in the code editor is called **Navigate To**. With this feature, you can easily find the definition of a type or member by placing the cursor on the type or member and then pressing **CTRL + ,** . Figure 13 demonstrates how Visual Studio 2013 shows types and members that contain the name selected in the code editor.

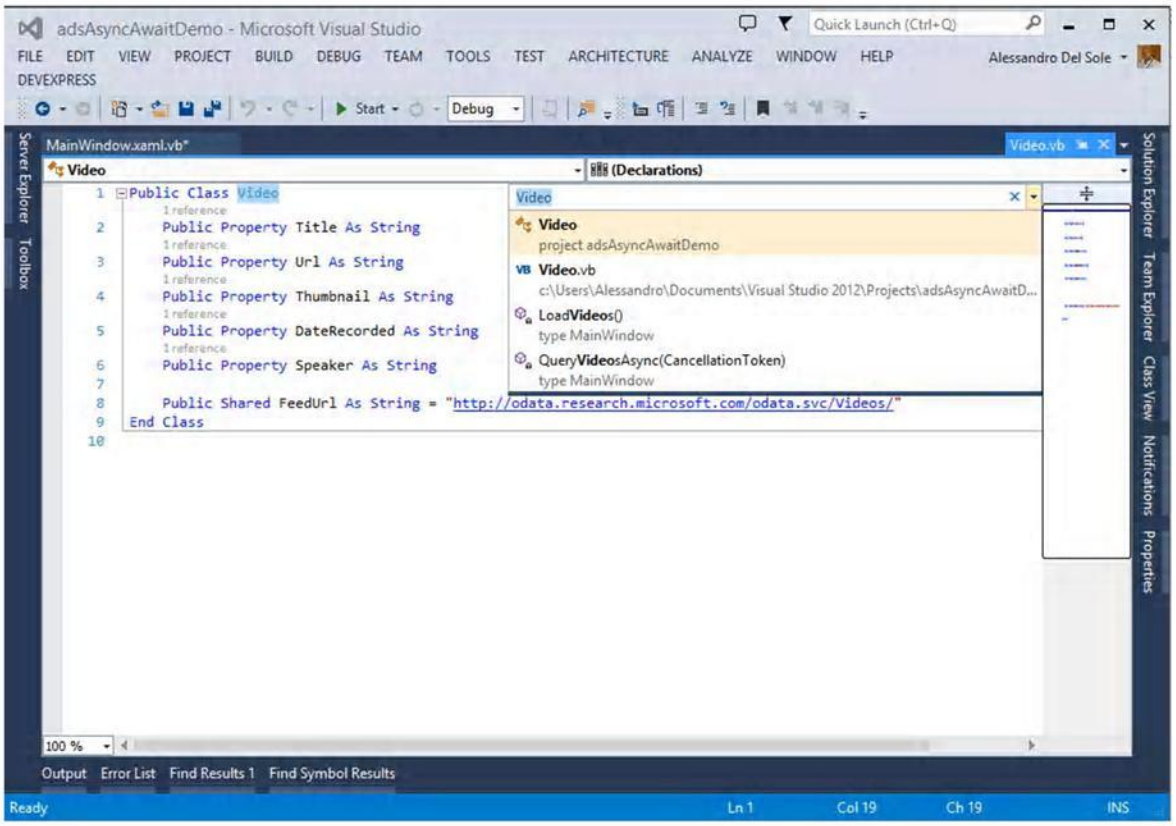


Figure 13: Using Navigate To

As you can see from Figure 13, a list of objects matching the type or member name is shown. You can press the up and down arrows on your keyboard to see every definition in the code editor without actually activating a window. As you can easily understand, Navigate To becomes particularly useful if you have multiple definitions of the same type in your solution and you want to go to its definition quickly.

Chapter summary

The code editor in Visual Studio 2013 has been dramatically enhanced with new features that increase the developer's productivity by making it easier to complete common tasks. Peek Definition, integrated references, the enhanced scroll bar, and Navigate To give their contribution to make the new IDE a great place for writing code.

Chapter 4 XAML IntelliSense Improvements

XAML (*eXtensible Application Markup Language*) is a markup language used to design the user interface in technologies such as Windows Presentation Foundation (WPF), Silverlight, Windows Phone, and Windows Store Apps for Windows 8. If you have experience with at least one of these technologies, which I assume you have, you know that Visual Studio, especially in the latest versions, offers a pretty good designer. In addition, Microsoft offers another tool called Expression Blend, which is dedicated to professional designers and allows working on the user interface with professional tools without interfering with the managed code behind. As a developer, most of the time you will work with Visual Studio. Although the designer has reached a very good level of productivity, a lot of times you will need to write XAML code manually; this is a very common practice, for example when you need to resize elements in the UI with exact proportions or when you need to assign styles or set data-bindings to controls. When you edit the XAML manually, you use the XAML code editor, which implements the IntelliSense technology, allowing you to write code faster. However, IntelliSense for XAML has always lacked some important points, such as recognizing available data sources and resources when using data-binding or assigning styles. Finally, Visual Studio 2013 addresses this issue and introduces a lot of new goodies into the XAML code editor.All of these new features are available to all technologies based on XAML.

Note: In the first preview of Visual Studio 2013, the new features in the XAML code editor were only available to Windows Store Apps. This limitation has been removed in the Visual Studio 2013 Release Candidate.

XAML IntelliSense for data-binding and resources

In Visual Studio 2013 you can now take advantage of IntelliSense when assigning a data source to a binding expression or when you assign a resource such as a style.

Note: This feature only works with data sources and resources that you declare in XAML. If you create an instance of a collection in managed code (at runtime), this cannot be recognized by the IntelliSense. It also works with design-time information that you declare through the d: XML namespace.

Let's use an example to see how this feature works.

The goal of the example is declaring a collection of objects and binding the collection to the user interface using the new IntelliSense features. Data will be shown inside a **`ListBox`** control. Create a new WPF Application project called *Chapter 4*, for the sake of consistency. Add a new folder to the project and call it Model. Add a new class to the folder, called **`Person`**. The code for the new class looks like the following.

Visual C#

```
namespace Chapter4.Model
{
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
    }
}
```

Visual Basic

```
Namespace Model
    Public Class Person
        Public Property FirstName As String
        Public Property LastName As String
        Public Property Age As Integer
    End Class
End Namespace
```

Note: While Visual C# automatically adds a namespace definition for each subfolder you create in the project, Visual Basic doesn't; it just recognizes objects under the root namespace. For the sake of consistency in both languages, we are adding a namespace declaration in Visual Basic so that we can use the same XAML with both.

This is a very simple class with only three properties, but we are focusing on the new tools now rather than on writing complex code. The next step is adding to the Model folder a new collection of **`Person`** objects, called **`People`**. The code is the following.

Visual C#

```
using System.Collections.Generic;
using System.Collections.ObjectModel;

namespace Chapter4.Model
{
    //add a using System.Collections.ObjectModel; directive
    public class People: ObservableCollection<Person>
    {
        public People()
        {
            Person one = new Person {LastName="Del Sole",
                                     FirstName="Alessandro", Age=36};
            Person two = new Person { LastName = "White",
                                      FirstName = "Robert", Age = 39};
```

```
            Person three = new Person { LastName = "Red",
                                        FirstName = "Stephen", Age = 42 };

            this.Add(one);
            this.Add(two);
            this.Add(three);
        }
    }
}
```

Visual Basic

```
Imports System.Collections.ObjectModel
Namespace Model
    Public Class People

        Inherits ObservableCollection(Of Person)
        Public Sub New()
            Dim one As New Person() With {.LastName = "Del Sole",
                                          .FirstName = "Alessandro",
                                          .Age = 36}
            Dim two As New Person() With {.LastName = "White",
                                          .FirstName = "Robert", .Age = 39}
            Dim three As New Person() With {.LastName = "Red",
                                            .FirstName = "Stephen",
                                            .Age = 42}

            Me.Add(one)
            Me.Add(two)
            Me.Add(three)
        End Sub
    End Class
End Namespace
```

The **People** class inherits from **ObservableCollection<Person>**. The constructor of the **People** collection creates three instances of the **Person** class, and populates them with sample data. The reason why we are creating a collection this way is that IntelliSense for XAML does not support collections created at runtime. Instead, with this approach we can declare the collection in the application's resources; every time a class is declared in the XAML resources, the constructor of the class is invoked, so in our case an instance of the collection is automatically created and populated when added to the XAML resources. Such an instance can then be data-bound to controls in the user interface. To do so, double-click the **MainWindow.xaml** file in Solution Explorer. When the designer and the XAML editor appear, first add the following namespace declaration within the **Window** tag.

```
<Window x:Class="Chapter4.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Chapter4.Model"
        xmlns:controls="clr-namespace:Chapter4"
        Title="MainWindow" Height="350" Width="525">
```

This is necessary in order to reference the **People** and **Person** classes. The next step is declaring the data source as a resource, as shown in the following code.

```
    <Window.Resources>
        <local:People x:Key="PeopleData"/>
    </Window.Resources>
```

This is the point in which an instance of the **People** collection is declared, so we are ready to bind data to a **ListBox** control. As you know, in order to present information coming from a collection, the so-called item controls (like the **ListBox**) need to implement a **DataTemplate**. Let's add a **ListBox** and its **DataTemplate** without pointing to any data source, by writing the following code within the **Window** tag.

```
    <Grid>
        <ListBox Name="PeopleBox" >
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="Black"
                            BorderThickness="2">
                        <StackPanel Orientation="Vertical">
                            <TextBlock />
                            <TextBlock />
                            <TextBlock />
                        </StackPanel>
                    </Border>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
```

The data template simply presents the value of each property of the **Person** class with a **TextBlock** control, arranged inside a **StackPanel** container. The **Border** adorner is used for a better highlighting inside the designer, but it is optional. For a full demonstration of the new IntelliSense features, we can also add a new style for **TextBlock** controls. In Solution Explorer, double-click the **App.xaml** file. Within **the Application.Resources** tag, add the following style, which allows presenting text in red and with different size and weight for the current font.

```
<Application.Resources>
    <Style x:Key="MyTextBlockStyle" TargetType="TextBlock">
        <Setter Property="Foreground" Value="Red"/>
        <Setter Property="FontSize" Value="16"/>
        <Setter Property="FontWeight" Value="SemiBold"/>
    </Style>
</Application.Resources>
```

Now you are ready to test the new amazing IntelliSense for XAML.

IntelliSense for data-binding

Switch back to the MainWindow.xaml file locate the **ListBox** control. As you know, item controls are populated by assigning their **ItemsSource** property with an instance of a collection, either at design-time (with XAML code) or at runtime (in managed code). We previously declared a data source as a resource, so a Source binding expression is needed to assign it as the **ItemsSource** property for the **ListBox**. To understand the benefit of XAML IntelliSense at this point, type the following code (not just copy/paste).

```
<ListBox Name="PeopleBox"
         ItemsSource="{Binding Source={StaticResource PeopleData}}">
```

After you type **StaticResource**, you will see how the IntelliSense will show a list of available objects that can be used for data-binding, as demonstrated in Figure 14.

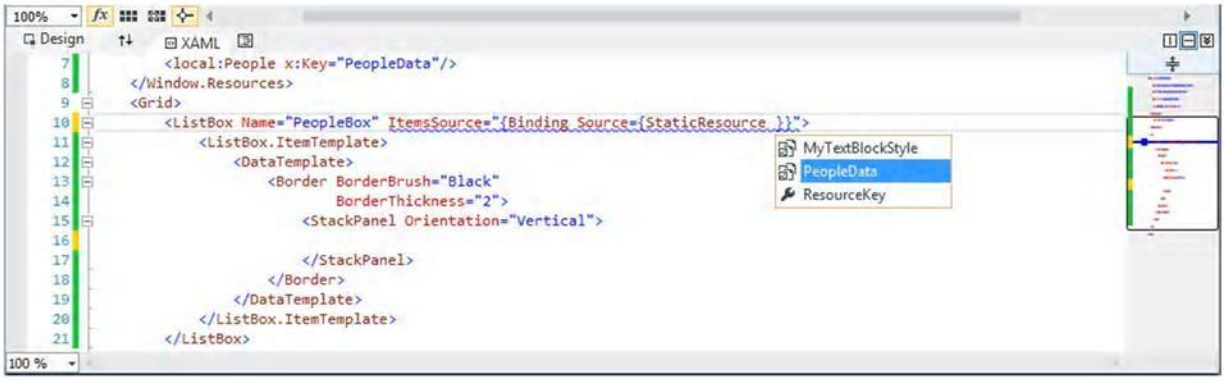


Figure 14: IntelliSense for Data-Binding in Action

Select the **PeopleData** object to finalize data-binding. As in every other scenario where you write code, IntelliSense will help you complete the expression while you type. You can also select the data source with the arrows on your keyboard and then press **Tab**.

Tip: If you have multiple data-bound controls in the Window, you might want to bind the parent container's DataContext property instead (in this case the Grid) and then assign the ItemsSource property with the {Binding} expression.

Similarly, you can bind **TextBlock** controls to properties of the collection with the help of IntelliSense, as demonstrated in Figure 15.

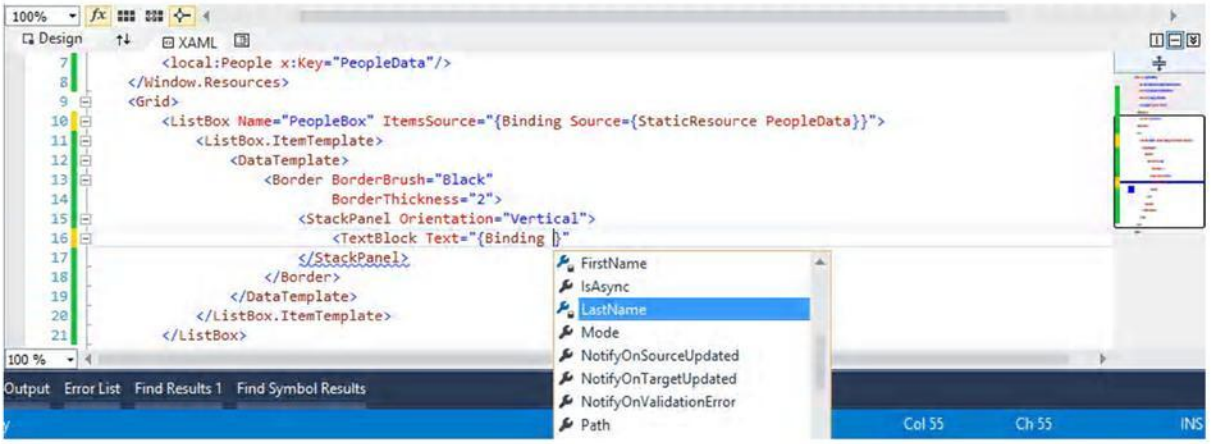


Figure 15: IntelliSense shows properties that can be data-bound.

This is a tremendous benefit for several reasons. First, you can write code faster. Secondly, you do not need to remember the name or the casing of properties, thus minimizing the risk of typos. The complete code of the **ListBox**'s data template is the following.

```
        <DataTemplate>
           <Border BorderBrush="Black"
                   BorderThickness="2">
               <StackPanel Orientation="Vertical">
                   <TextBlock Text="{Binding LastName}"/>
                   <TextBlock Text="{Binding FirstName}"/>
                   <TextBlock Text="{Binding Age}"/>
               </StackPanel>
            </Border>
        </DataTemplate>
```

IntelliSense for resources

The next step is using IntelliSense to assign resources. We previously defined a style that must be now assigned to each **TextBlock** control in the Window. The code for the first **TextBlock** looks like the following. You might want to type the style assignment manually in order to see the XAML IntelliSense feature in action.

```
<TextBlock Text="{Binding LastName}"
           Style="{StaticResource MyTextBlockStyle}"/>
```

As it happened for data-binding, when you are assigning the **StaticResource** expression the IntelliSense will show available resources, as shown in Figure 16.

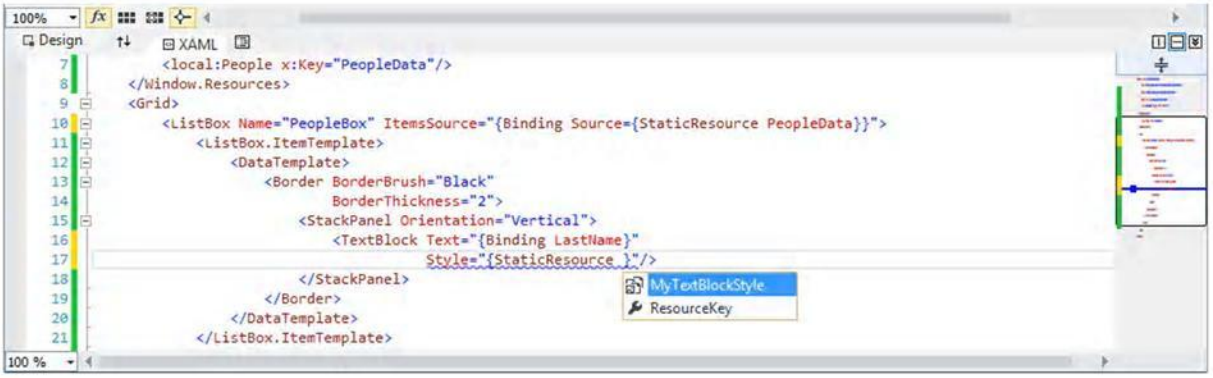


Figure 16: IntelliSense shows available resources for the specified control.

It is worth mentioning that, in the case of styles, IntelliSense will only show styles valid for the control that you are working on, either defined in the application or defined in the .NET Framework or SDK extensions. This is an additional benefit, since you not only will avoid errors and will write code faster, but you will be also be picking up only resources that are specific for the selected element of the user interface. For the sake of completeness, Figure 17 shows how the designer looks at this point.

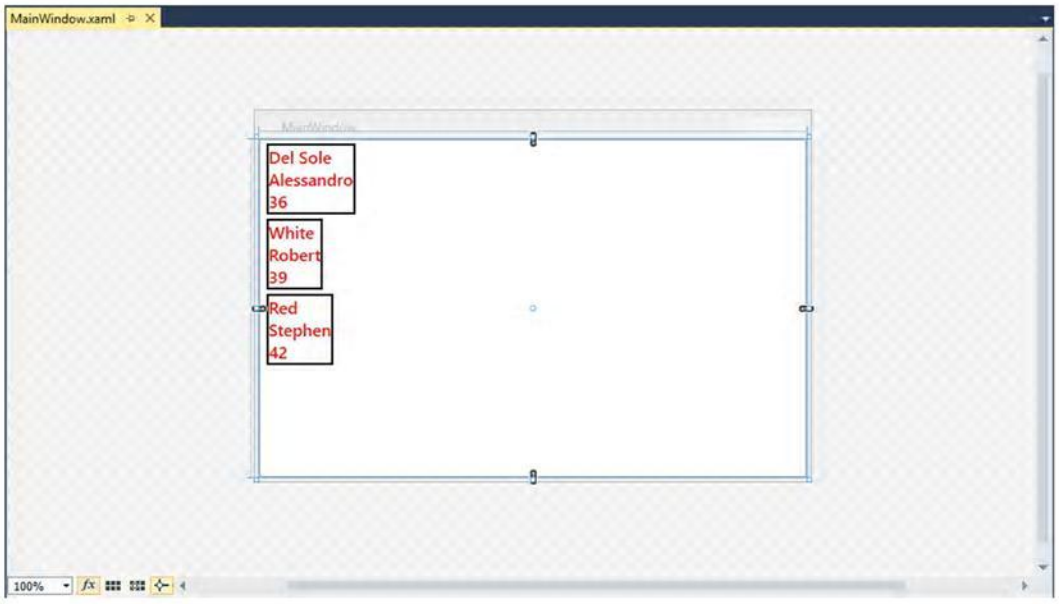


Figure 17: The Designer after the Data and Style Assignments

You can run the application by pressing **F5** to see how it displays data.

Go To Definition

Go To Definition is a feature that you already know from the managed code editor. With this feature, you can right-click an object's name, select **Go To Definition** from the context menu, and see how the object is defined in the Object Browser window, if it is a built-in object from the .NET Framework, or in the appropriate code file if it is an object you wrote. This feature is now available to the XAML editor too, as demonstrated in Figure 18.

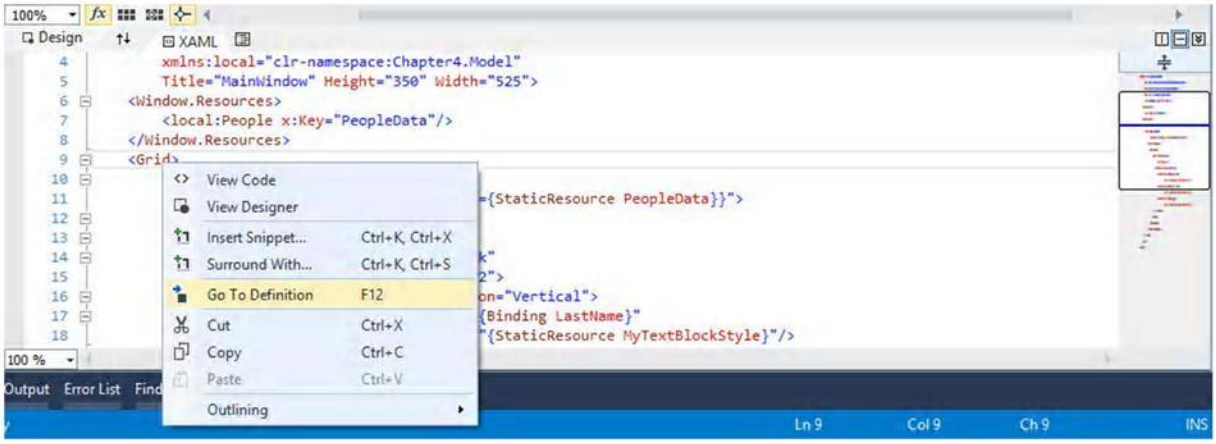


Figure 18: Go To Definition is now available in the XAML code editor.

Tip: The keyboard shortcut for Go To Definition is also F12 in the XAML code editor.

Go To Definition is available for the following objects:

- Resources
- System types
- Local types (custom controls)
- Binding expressions

Let's walk through every object to see the different behavior of Go To Definition.

Resources

In an XAML-based application, resources can be of two types: resources defined in an assembly (from .NET, from the SDK, or from a 3rd party library) and resources defined in the current application. In the first case, Go To Definition will open the Object Browser window pointing to the specified resource definition. For example, in the sample application created previously, place the cursor on the **`MyTextBlockStyle`** assignment in any of the **`TextBlock`** controls, then right-click and select **Go To Definition** (see Figure 19).

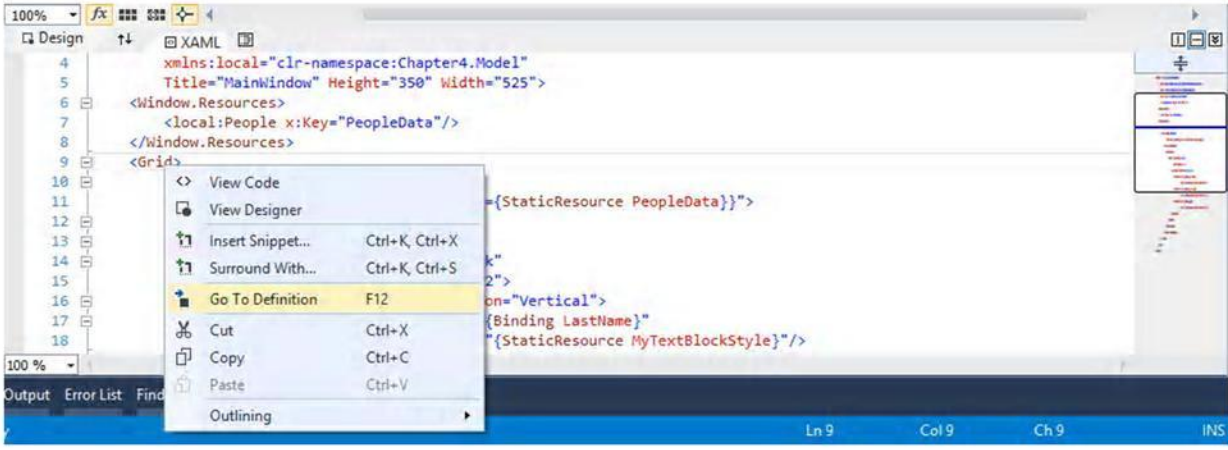


Figure 19: Go To Definition Over the Style Defined in the Sample Application

At this point, the code editor will open the definition of the resource at the exact position in the appropriate code file; in our case, the style definition inside the App.xaml file. As you can see (Figure 20), the cursor is placed at the beginning of the definition and the **`Style`** tag is selected.

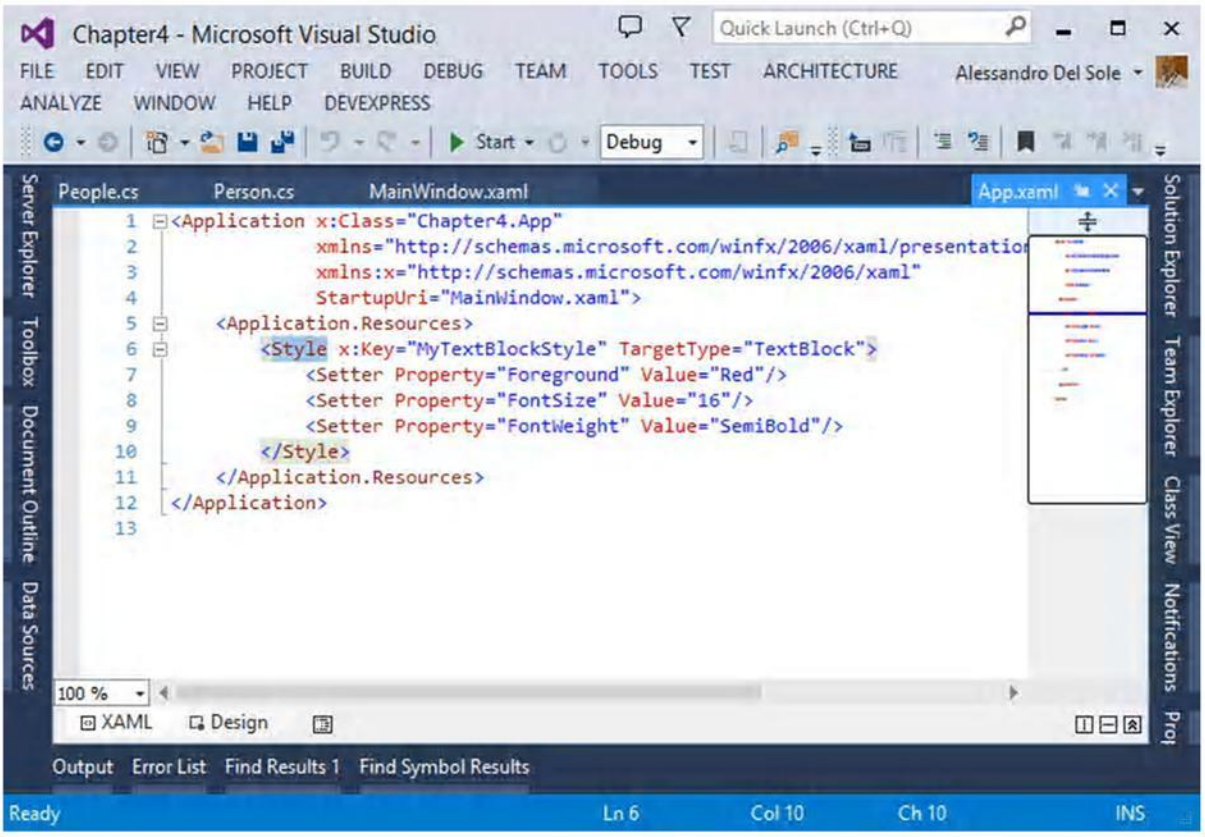


Figure 20: Go To Definition opens the resource definition at the exact position.

System Types

Go To Definition works with types defined in the .NET Framework and SDK extensions. Since the source code of these types is not available, Visual Studio shows the definition inside the Object Browser window. For example, if you select Go To Definition on the **`Grid`** control in the MainWindow.xaml file of the sample application, the Object Browser will be opened, showing the control's definition (see Figure 21).

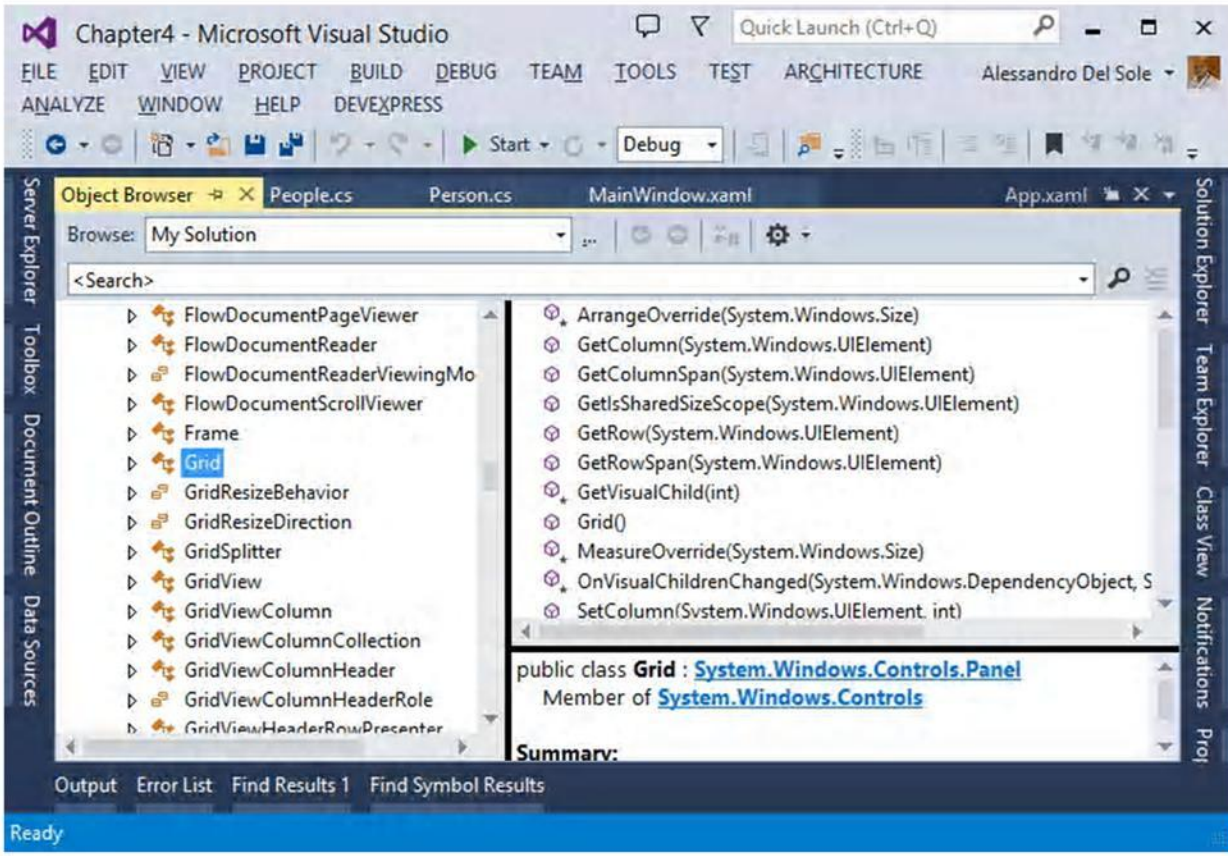


Figure 21: Seeing a Control's Definition with the Object Browser

Local Types

Go To Definition has a particular behavior with local types; these are user controls and custom controls created by developers.

Note: This is not a book about WPF and other XAML-based technologies, so I will not cover the difference between user controls and custom controls in detail. As a hint, user controls are the result of the composition of existing controls; custom controls extend existing built-in controls with additional functionalities in code, and provide templating, theming, and styling entry points. For further information, read the Control Authoring Overview in the MSDN Library.

To understand how it works, let's make some edits to the sample application. In **Solution Explorer**, right-click the project name, select **Add New Item**. Then, in the **Add New Item** dialog, select the **User Control (WPF)** template and name the new control **CustomBoxControl.xaml** (see Figure 22).

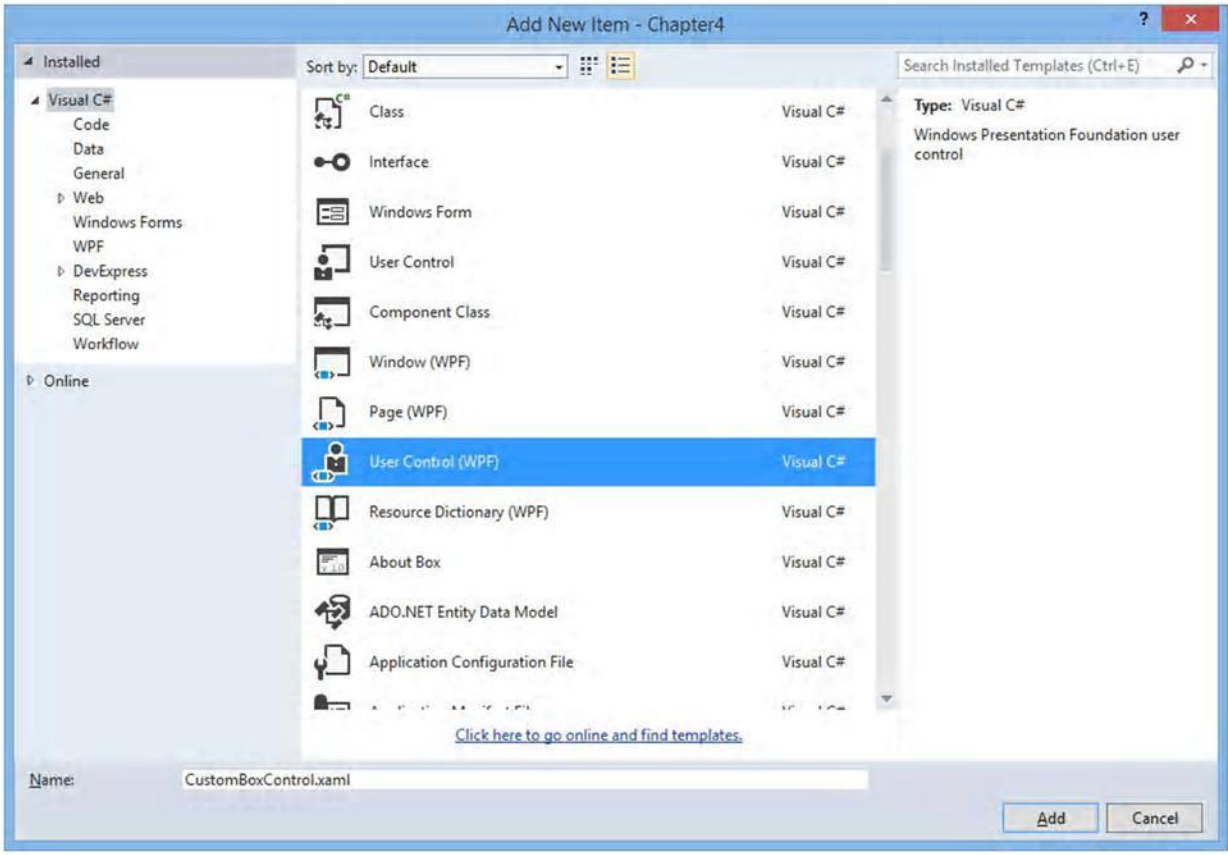


Figure 22: Adding a New User Control

Cut and paste the **`ListBox`** definition from MainWindow.xaml to the new control, and repeat this step for the **`local`** XML namespace declaration. Finally, add a local resource that points to the **`People`** collection as you did in MainWindow.xaml. The full code of the user control looks like the following.

```
<UserControl x:Class="Chapter4.CustomBoxControl"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d" xmlns:local="clr-namespace:Chapter4.Model"
             d:DesignHeight="300" d:DesignWidth="300">
    <UserControl.Resources>
        <local:People x:Key="PeopleData"/>
    </UserControl.Resources>
```

```
    <Grid>
        <ListBox Name="PeopleBox"
                 ItemsSource="{Binding
                 Source={StaticResource PeopleData}}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="Black"
                            BorderThickness="2">
                        <StackPanel Orientation="Vertical">
                            <TextBlock Text="{Binding LastName}"
                                       Style="{StaticResource
                                       MyTextBlockStyle}"/>
                            <TextBlock Text="{Binding FirstName}"
                                       Style="{StaticResource
                                       MyTextBlockStyle}"/>
                            <TextBlock Text="{Binding Age}"
                                       Style="{StaticResource
                                       MyTextBlockStyle}"/>
                        </StackPanel>
                    </Border>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</UserControl>
```

In MainWindow.xaml, add the following XML namespace to include the user control.

```
xmlns:controls="clr-namespace:Chapter4"
```

Then, add the user control as follows.

```
    <Grid>
        <controls:CustomBoxControl/>
    </Grid>
```

If you did everything correctly, the designer now should still look like in Figure 17. Now, right-click **`CustomBoxControl`** inside the **`Grid`** and select **Go To Definition**. As you can see (Figure 23), Visual Studio 2013 opens the Find Symbol Results window and shows two results, one for XAML and one for managed code.

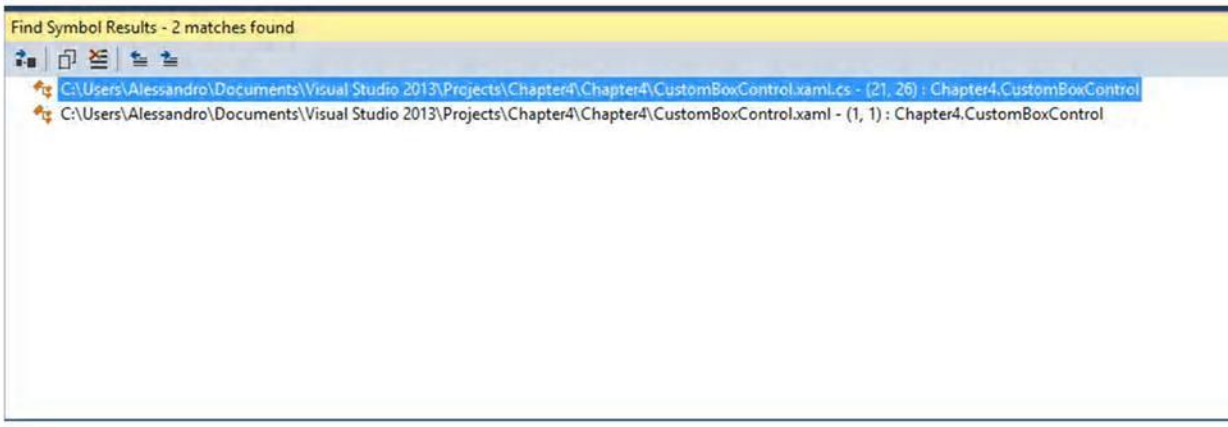


Figure 23: Adding a New User Control

The reason is that user controls (and custom controls as well) are made of two components, the XAML file defining the user interface and a code-behind file. You can then double-click the file you need and see the appropriate definition. The Find Symbol Result is a good choice, because you can also see the line number where the definition begins.

Binding expressions

Go To Definition also works with binding expressions. For instance, you can right-click the name of the data source or of the bound property inside a **`Binding`** expression and select Go To Definition. With data sources (collections) defined as static resources, Go To Definition moves you to the definition of the resource; with data sources defined in managed code, Go To Definition attempts to make a full symbol search in the code, showing search results in the Find Symbol Results window. In the case of data-bound properties, Go To Definition moves you to the code of the class that exposes such a property.

Automatic closing tag

When you add an item in XAML, the code editor automatically adds the closing tag. For instance, when you add a **`<Button>`** tag, Visual Studio adds the matching **`</Button>`** tag. This is not new, since it is the normal behavior in the earlier versions. The new feature is that if you add the slash before the **`>`** symbol in the first tag, the closing tag is automatically removed. In other words, in this code:

```
<Button Width="100" Height="50" Click="Button_Click" Name="Button1" Content="Click
me!"></Button>
```

If you type the slash before the **`>`** symbol, it automatically turns into the following code.

```
<Button Width="100" Height="50" Click="Button_Click" Name="Button1" Content="Click
me!"/>
```

This is another way the editor can help you write code faster.

IntelliSense matching

Continuing its purpose to make your coding experience better, Visual Studio 2013 adds another feature to the XAML code editor, known as IntelliSense matching. Basically, when you start typing the name of a control or resource, the IntelliSense will help you find the appropriate control as you type, even if you enter the wrong characters. For example, Figure 24 shows how IntelliSense understands you need a **`StackPanel`** even if you are typing it incorrectly.

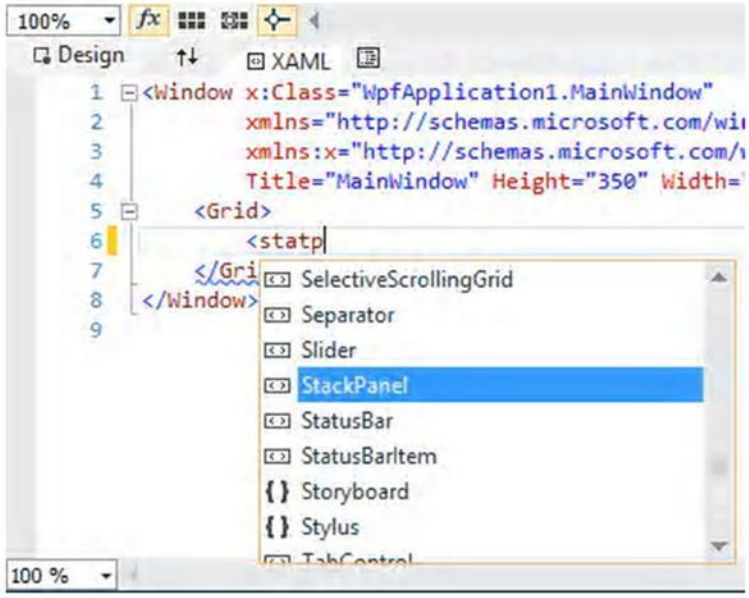


Figure 24: IntelliSense Matching makes it easy to select controls as you type.

IntelliSense does an excellent job, depending on how many identifiers can match what you are typing. For instance, in a Windows Store App, if you type **`Abbb`** it will suggest the **`AppBar`** control, which is probably your choice.

Better support for comments

A common issue in previous versions of Visual Studio is that when you add a comment to a code block containing another comment, it causes the code editor to show an error message. Figure 25 shows how Visual Studio 2012 handles this kind of situation.

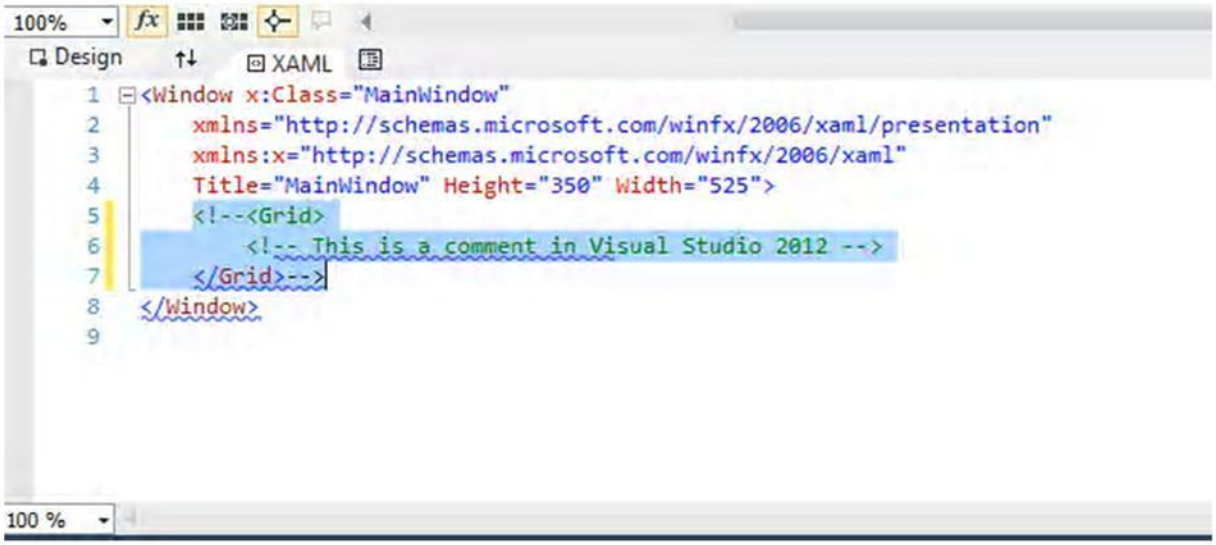


Figure 25: Visual Studio 2012 does not recognize nested comments correctly.

The problem was that the code editor did not recognize comment closing tags correctly. Visual Studio 2013 addresses this issue, so when you add a comment to a code block containing another comment, the entire code block gets commented, as demonstrated in Figure 26.

Figure 26: Visual Studio 2013 correctly recognizes nested comments.

Reusable XAML code snippets

Reusable code snippets for IntelliSense have been a very popular feature since Visual Studio 2005, but have always been limited to managed languages, XML, and HTML/JavaScript in version 2012. With code snippets you can take advantage of a huge code library offered by Visual Studio or create your own code snippets, so that it is easier to reuse your code with the support of IntelliSense. The need of code snippets for XAML has always been very strong, so many developers used different techniques to store their reusable code. I wrote myself an extension for Visual Studio 2010 to support XAML code snippets. Visual Studio 2013 makes another step forward, introducing built-in support for code snippets in the XAML code editor.

To use code snippets, simply right-click in the code editor and select **Insert Snippet** or **Surround With**, as shown in Figure 27.

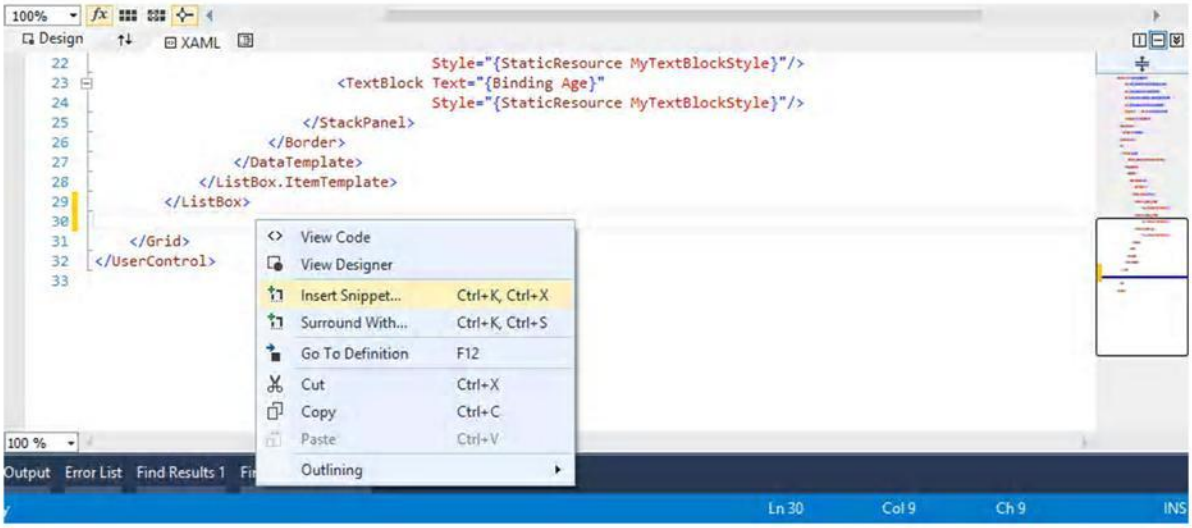


Figure 27: Visual Studio 2013 provides support for XAML code snippets.

At this point, a list of available code snippets appears. You can select the code snippet you need from the list and the related code will be placed there (see Figure 28).

Figure 28: Selecting a Code Snippet from the List

Note: This chapter is based on Visual Studio 2013 RTM released to the MSDN subscribers in October, 2013. In this release only one XAML snippet is supplied. At the time of this writing, we cannot predict if additional snippets will be provided, or if you will only be able to import your own, custom code snippets. This chapter does not explain how to build custom code snippet files, which is out of its scope. However, you can read this interesting article from Tim Heuer, which explains XAML code snippets from creation to deployment.

You can manage code snippet files with the Code Snippets Manager tool (available from the Tools menu), as you already did for other languages. This is the place where you can import, remove, and view detailed information about code snippet files. By adding XAML code snippets, Visual Studio 2013 bridges the gap with code editors for other languages.

Chapter summary

Without a doubt, adjusting the user interface and supplying data-bindings manually in XAML code is a very common task in any XAML-based technology, including WPF, Silverlight, Windows Phone, and Windows Store Apps, despite the existence of specialized tools for designing. Visual Studio 2013 finally provides important improvements to the XAML code editor, such as IntelliSense for recognizing data sources and styles; Go To Definition for system types, local types, custom, and user controls; commenting nested code blocks; better handling of closing tags, and IntelliSense matching to help you select controls quickly; and reusable code snippets, finally added to XAML completing the availability of this tool to all of the supported languages.

Chapter 5 Visual Studio 2013 for the web and Windows Azure

Programming for the web is the core business for many companies and developers. Creating websites means making an application available to potential customers worldwide through the Internet or providing internal portals or departmental applications through a local Intranet. Because of the importance of the web in a programmer's life, cloud computing platforms are becoming more and more important every day. With Windows Azure, Microsoft has released one of the most powerful and complete cloud infrastructures ever. With a platform like Windows Azure you no longer need an in-house data-center, removing the need of purchasing physical servers and paying for their maintenance; with Windows Azure, you only pay for services you actually use. This chapter does not explain what ASP.NET and Windows Azure are, nor does it explain how to create applications for both platforms; that's the goal of other resources. Instead, here we focus on how the new tooling available in Visual Studio 2013 makes programming for the web and the cloud an even more amazing experience with a deeper integration with the IDE.

Note: The .NET Framework 4.5.1 introduces some new features to ASP.NET. If you want to learn about what's new, you can visit the appropriate page in the MSDN Library. Here you learn about new features in the IDE for ASP.NET, not about the runtime.

What's new in the IDE for ASP.NET

Visual Studio 2013 introduces new tools and updates the existing environment for web development with ASP.NET. This chapter focuses on the most important features that you must know, as they will change the way you create web applications.

One ASP.NET: A new, unified experience

In the past, Microsoft released several technologies for creating web applications, like Web Forms, ASP.NET Dynamic Data, and MVC. Similarly, a number of frameworks and libraries were released, such as jQuery, jQuery Mobile, Web API, and Windows Identity Foundation. You could choose among several kinds of project templates in order to build web applications and add references to your desired frameworks later. Visual Studio 2013 dramatically simplifies this process by introducing the so-called **One ASP.NET**, which provides a unified development experience and makes it easy to use any of the available platforms, as well as making libraries interchangeable. But what does One ASP.NET mean in practice? To understand what it is, open Visual Studio 2013 and select **File**, **New Project**. Select the **Web** templates folder. As you can see from Figure 29, now there is only one project template.

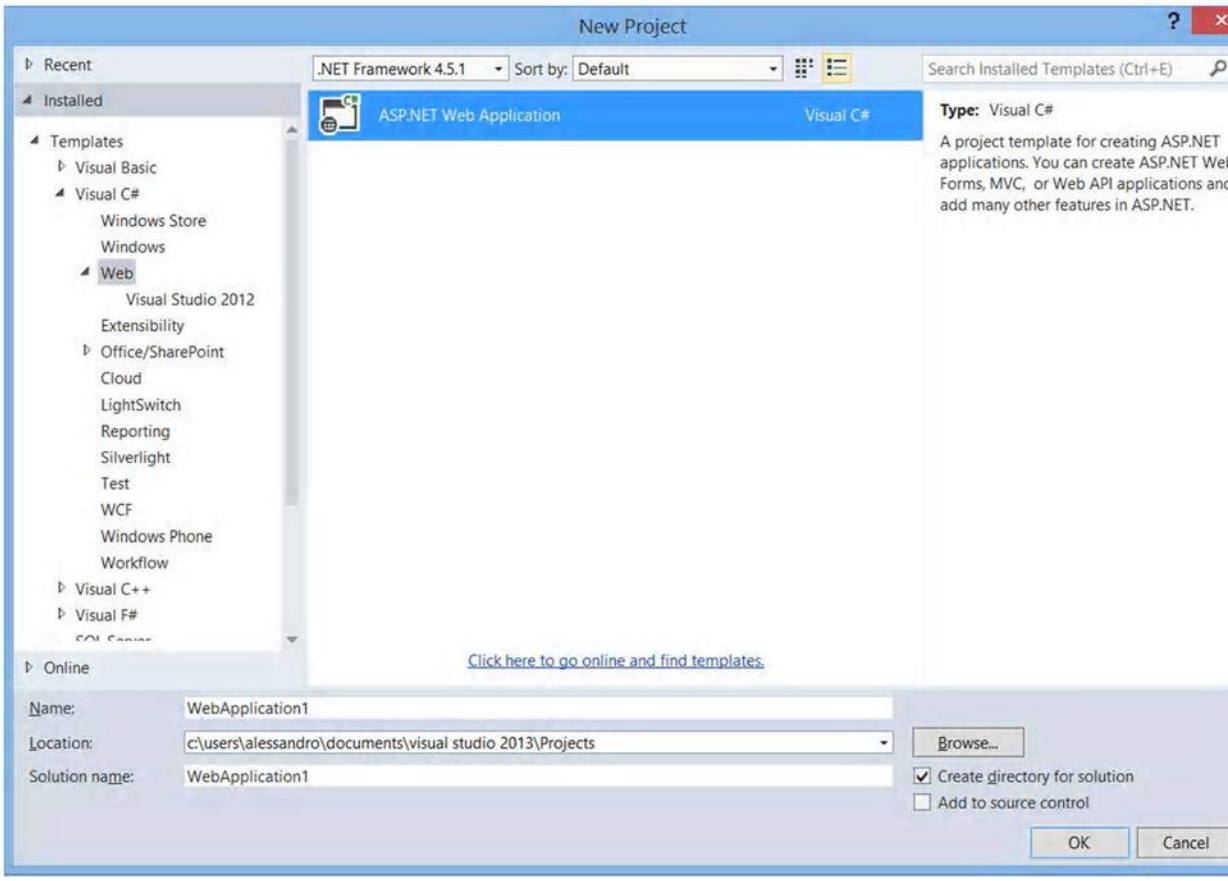


Figure 29: One ASP.NET also means simplifying a project's creation.

Unlike in the past, where you had to choose among a number of several project templates, now you have only one template. Don't be scared of this; in the next steps you will learn the reasons behind this feature and how to take advantage of it. For backward compatibility, you can still create web applications using templates inherited from Visual Studio 2012. You can simply expand the **Web** template folder and select the **Visual Studio 2012** subfolder (which is visible in Figure 29). You will see the classic list of available project templates based on Web Forms, MVC, and Ajax. Let's focus on One ASP.NET and double-click the single project template. At this point Visual Studio 2013 will show a new dialog, represented in Figure 30.

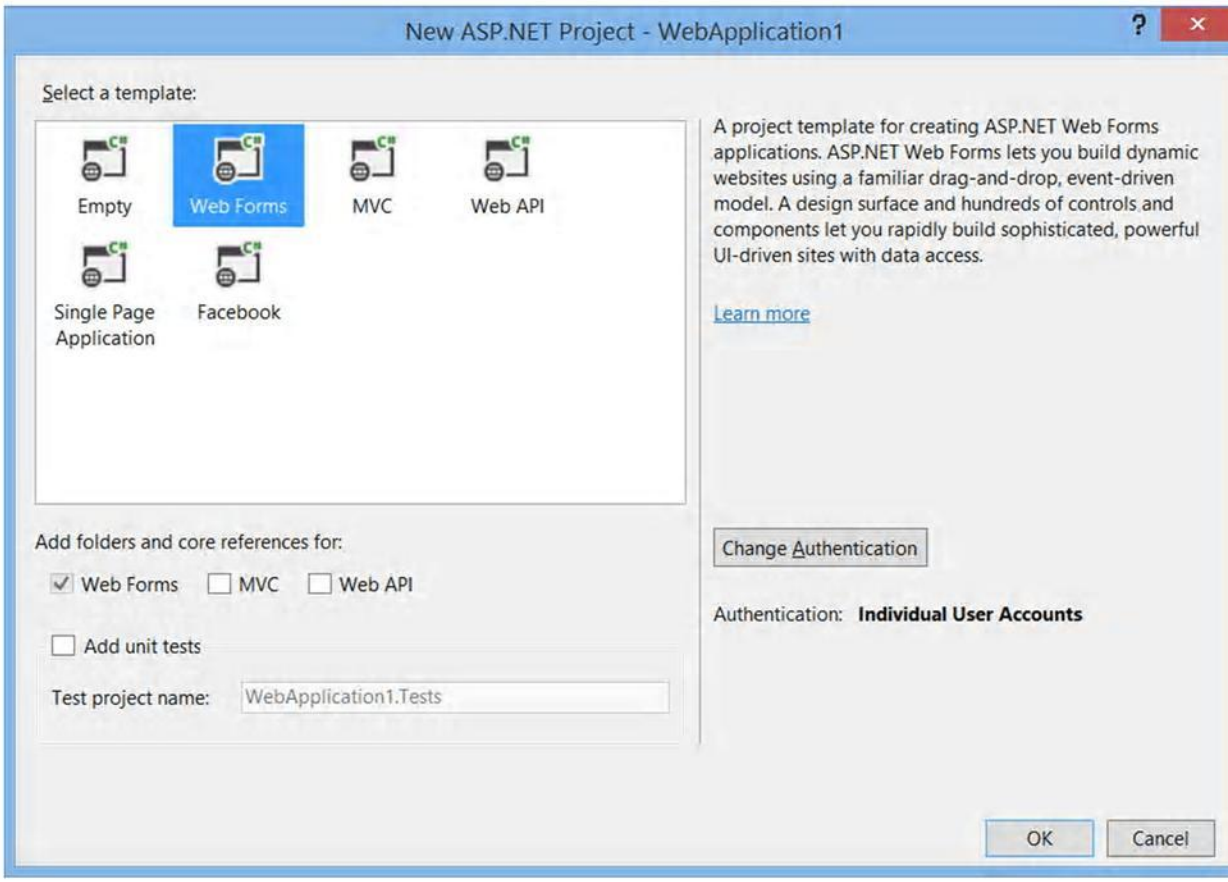


Figure 30: Selecting the Presentation Framework, Authentication, References

It's becoming clearer why the new approach is called One ASP.NET. In one place, you can:

- Select the presentation framework, such as Web Forms or MVC.
- Select a ready-to-use application stub, such as the Single Page Application or the Facebook application templates.
- Choose the authentication model: anonymous, Individual (ASP.NET), Windows (for Windows domains), Organizational (based on Active Directory, Office 365, and Windows Azure Active Directory).
- Add unit tests.
- Add folders and references to libraries that are specific to other frameworks; for instance, in a Web Forms application you can use MVC libraries and vice versa.

This new way of creating web applications gives you an opportunity to work with multiple kinds of libraries, taking full advantage of all the libraries from ASP.NET 4.5.1. Also, One ASP.NET simplifies the process by adding references for you. Experimenting with Web Forms, MVC, and other project templates is left to you as an exercise. Let's now take a closer look at new features from the IDE.

Scaffolding for Web Forms

Scaffolding is all about data. Basically with scaffolding the IDE can generate view models and views based on a modeled data source (such as the ADO.NET Entity Framework); with scaffolding, Visual Studio generates for you pages that can read, insert, delete, or update data without you writing a single line of code. Technically speaking, Visual Studio generates a **controller**, which is a class containing the necessary code to perform C.R.U.D. (Create, Read, Update, Delete) operations against data, and pages to work with data (**Views**), one for each of the C.R.U.D. operations. Scaffolding is not a new concept in the ASP.NET development; it was first introduced with ASP.NET MVC. The good news is that Visual Studio 2013 brings scaffolding to Web Forms as well. This is possible because of the One ASP.NET experience; in fact, Visual Studio injects the appropriate MVC dependencies into a Web Form project and then does most of the work for you. If you already used this technique in MVC projects, you will be familiar with most of the concepts.

Note: With Visual Studio 2008 and .NET Framework 3.5 Service Pack 1, Microsoft introduced ASP.NET Dynamic Data, an innovative way of creating modern, data-centric applications for the web. Dynamic Data is still available in later versions. The concept of scaffolding was the base of ASP.NET Dynamic Data, but what we mean today by scaffolding is pretty different, and is based on new frameworks, libraries, and implements different code-behind. For this reason, be sure you have clear that in this chapter we refer to scaffolding by indicating the MVC (and now Web Forms) implementations.

To understand how scaffolding works, we will now create a sample ASP.NET application based on Web Forms, then we will add a reference to a database. Then we will use the new tooling in Visual Studio 2013 to generate data-bound pages without writing a single line of code. Before you continue, ensure you have downloaded and installed the following prerequisites:

- Microsoft SQL Server 2012 Express Edition, as the database engine required for data access. I recommend you download either the "With Tools" or the "With Advanced Services" editions that will also install SQL Server Management Studio Express for database management.
- The Adventure Works sample database from Microsoft. It can be downloaded for free from this page of the CodePlex website.

We assume that you already know how to install a database like Adventure Works to SQL Server, so let's go ahead.

Create a sample project

First you will create a sample project. This is also the first time for you to see the One ASP.NET tooling in action. To create the project, follow these steps:

1. Select **File**, **New Project.**
2. Select the **Web** templates folder (see Figure 1).
3. Select the one **ASP.NET Web Application** and call the new project **ScaffoldingDemo.**
4. Click **OK.**
5. In the **New ASP.NET Project** dialog (take Figure 30 as a reference), select **Web Forms** as the presentation framework, then select the **MVC** checkbox.

6. For the sake of simplicity, change the authentication mode to anonymous. This is just for testing purposes; in a real world scenario you must select the appropriate authentication type according to your needs. To change the authentication, click **Change Authentication**, then in the **Change Authentication** dialog, select **No Authentication** (see Figure 31).
7. Click **OK** to create the project.

Figure 31: Changing the Authentication Type

Tip: For your curiosity or for real needs, while you are in the Change Authentication dialog, try to click on each option to see how you can implement different authentication types and how each type satisfies specific platform requirements.

Adding Data Connection and Entity Data Model

When the project is ready, in **Solution Explorer** right-click the project name, select **Add New Item**, and select the **Data** template folder. This is the point in which a data connection will be added, as demonstrated in Figure 32. Select the **ADO.NET Entity Data Model** item template; call the new model **AdventureWorks.edmx** and click **Add**.

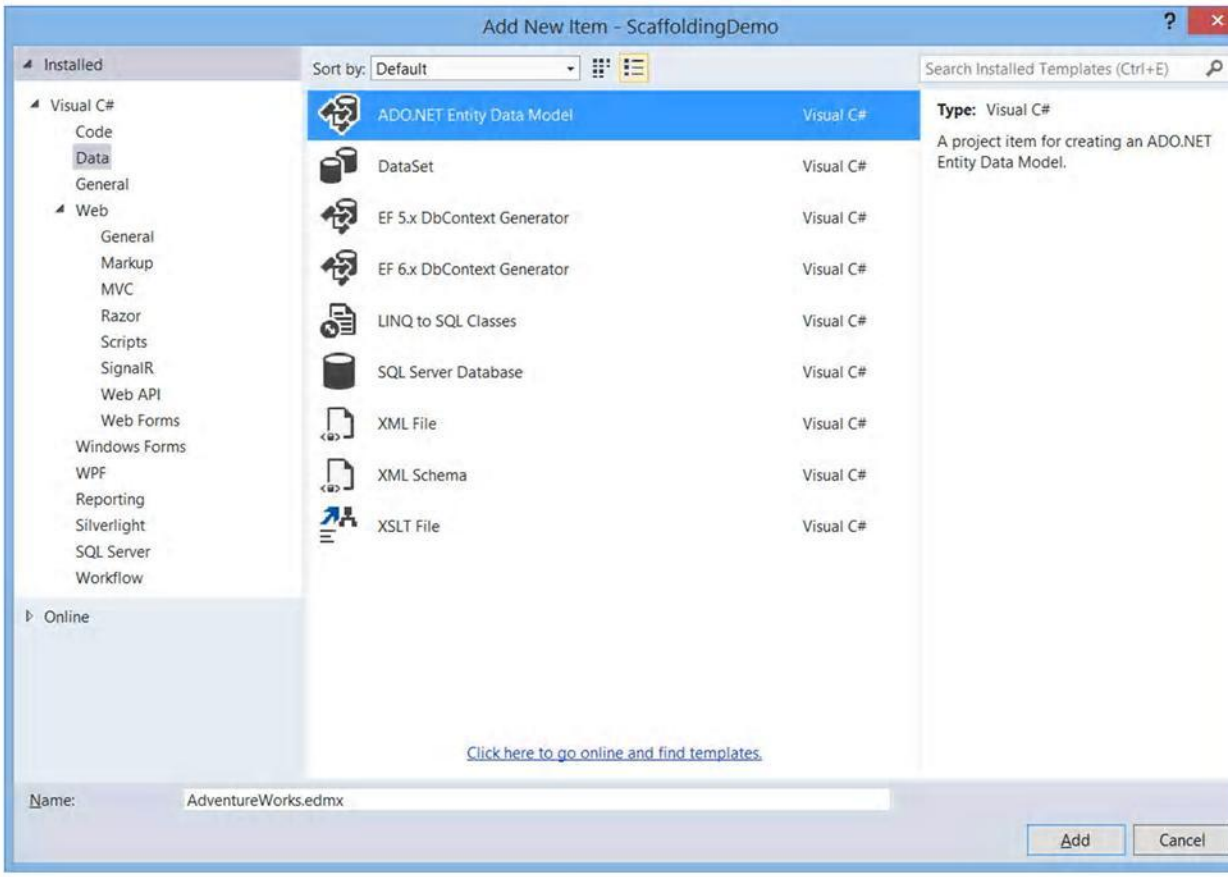


Figure 32: Adding a New Entity Data Model

As you know, the Entity Data Model is based on the ADO.NET Entity Framework. In the **Entity Data Model** wizard, select the **Generate From Database** option first, then click **Next**. Click the **New Connection** button, then add a new connection that points to the **AdventureWorks** database. Figure 33 shows how the Connection Properties window will look like at this point; of course, you can have a different server name on your machine.

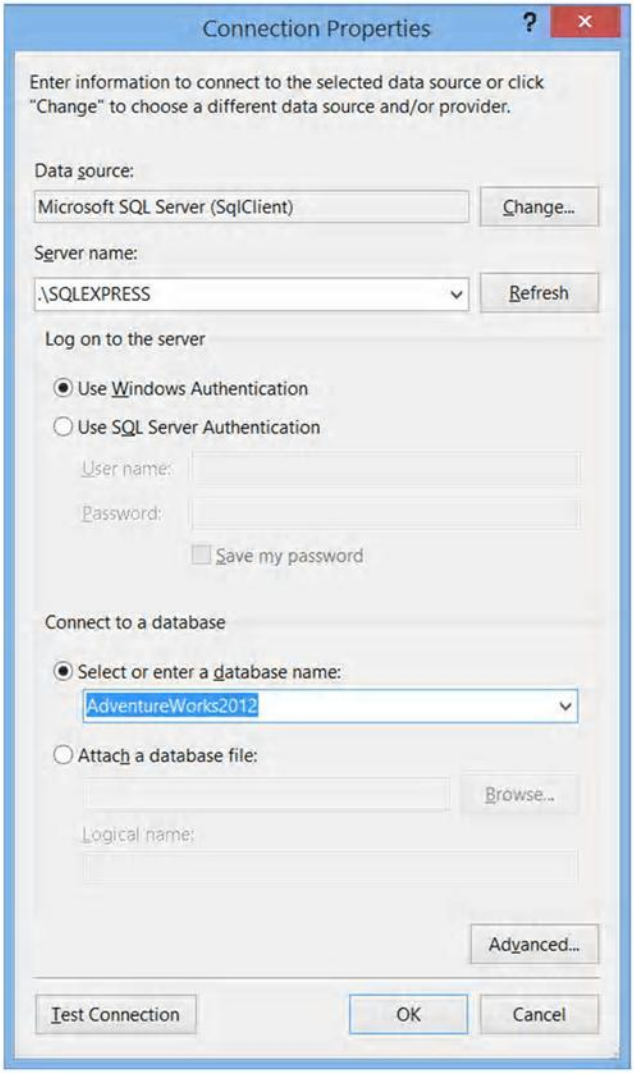


Figure 33: Adding a New Entity Data Model

At this point the Entity Data Model Wizard will show summary information for the newly created connection (see Figure 6). You can definitely leave unchanged the identifier for the connection settings or provide a different one (at the bottom of Figure 34).

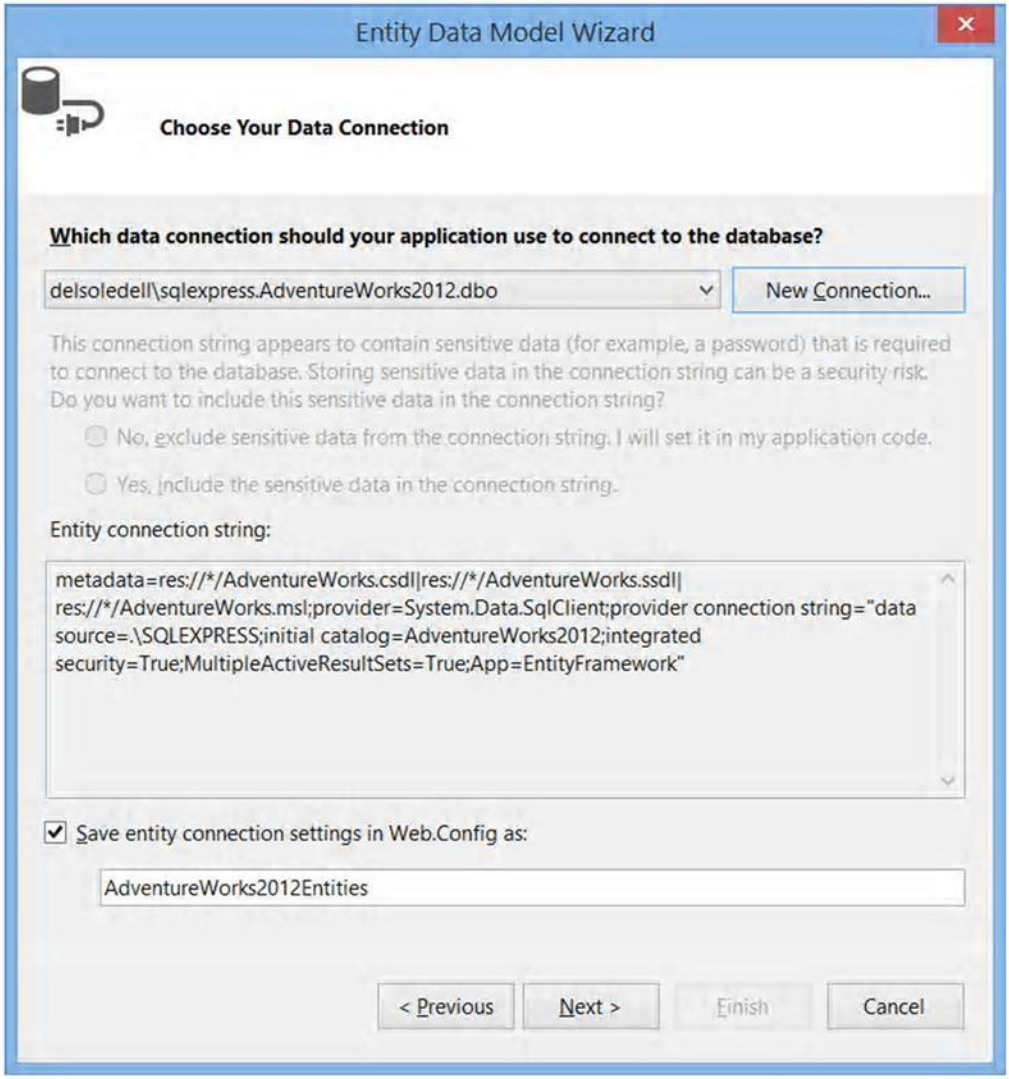


Figure 34: Summary Information for the New Entity Data Model

When you click **Next**, Visual Studio will ask you to specify the version of the Entity Framework you want to use, between 5.0 and 6.0 (default option). Leave unchanged the selection on version 6.0 and click **Next**. At this point you will need to specify the tables or views you want to add to the model. Just select the **Person** table, as we want to demonstrate concepts easily without having a complex data structure. Figure 35 shows how to make such a selection.

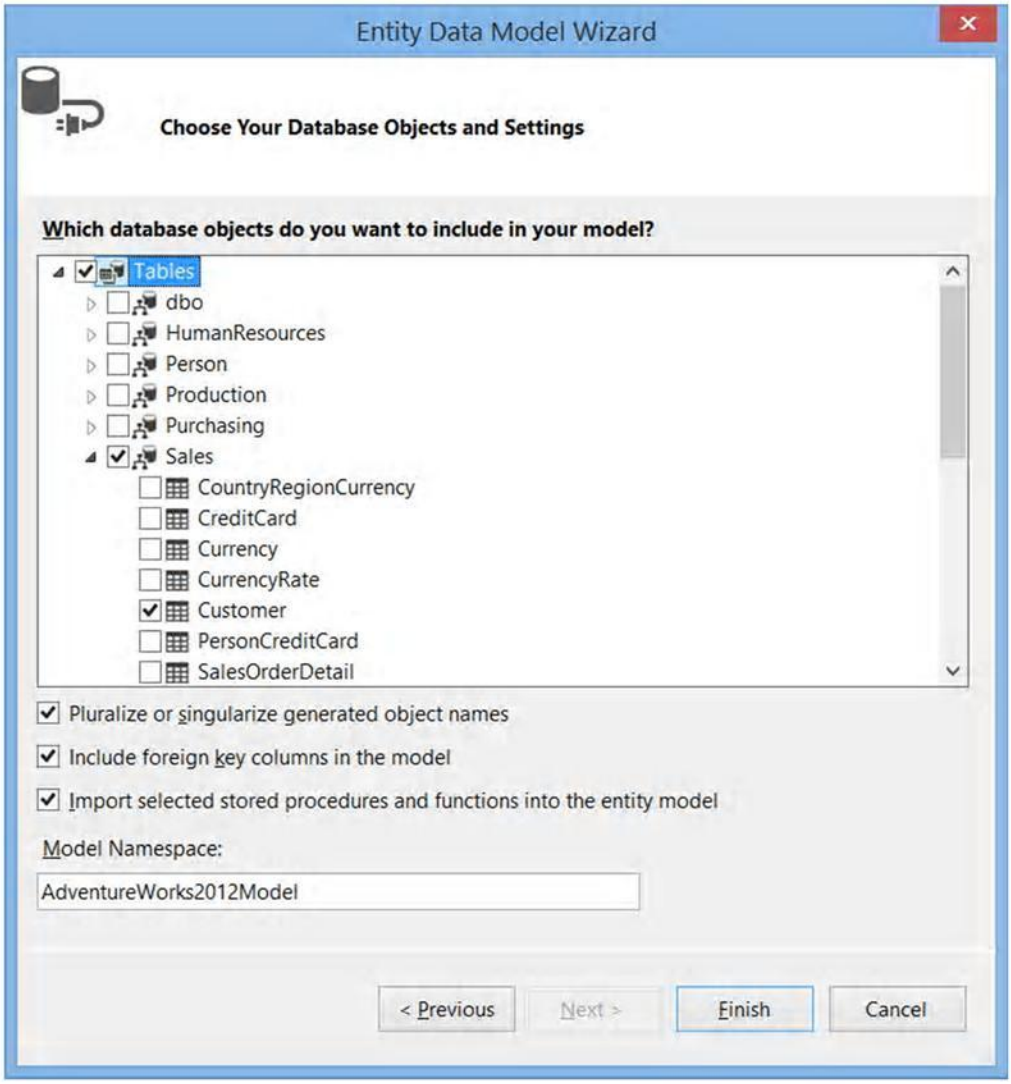


Figure 35: Selecting Database Objects

Click **Finish**. After a few seconds the Entity Data Model designer shows the .NET representation of the selected table (see Figure 36).

Tip: If you see a message that says "Running this template can potentially harm your computer," ignore it. Visual Studio always analyzes code snippets that execute actions against local resources, such as a database, including auto-generated snippets, but of course executing such actions is safe at this point.

Before doing anything else, rebuild the project (**CTRL** + **Shift** + **B**) so that all references are updated.

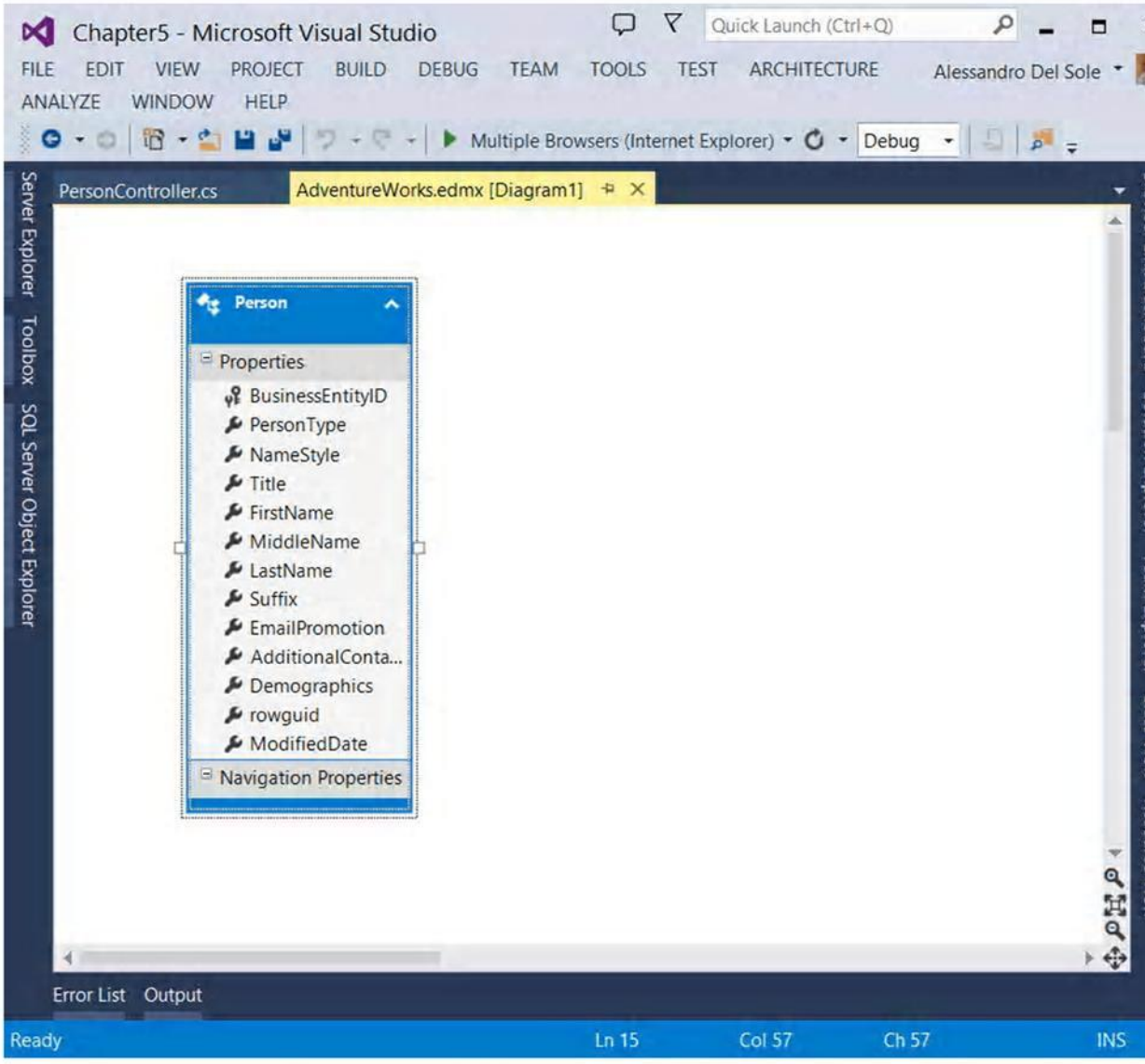


Figure 36: The Entity Data Model designer shows the table representation.

Depending on the database version you have installed, you might see additional entities in your designer. In fact, there are some differences between versions 2008 and 2012. You can ignore those additional entities and focus on entities you see in Figure 36. In the designer, right-click the **`Demographics`** property and select **Delete from Model**. The reason for this is that the **`Demographics`** property contains long XML markup that would make it difficult to provide readable figures for this e-book. Now that you have a connection to your data source, you need to present data and give users an opportunity of editing data. So, let's dive into scaffolding.

Generate data-bound pages with scaffolding

The benefit of scaffolding is the ability to generate data-bound views without writing a single line of code. Visual Studio 2013 generates a controller for each entity set and one page per action, which means one page for listing a collection of items, one for adding an item, one for editing, and one for deleting. Since scaffolding generates views, in **Solution Explorer** right-click the **Views** folder, then select **Add**, **New Scaffolded Item**. The **Add Scaffold** dialog appears and allows specifying the item you need, as you can see in Figure 37.

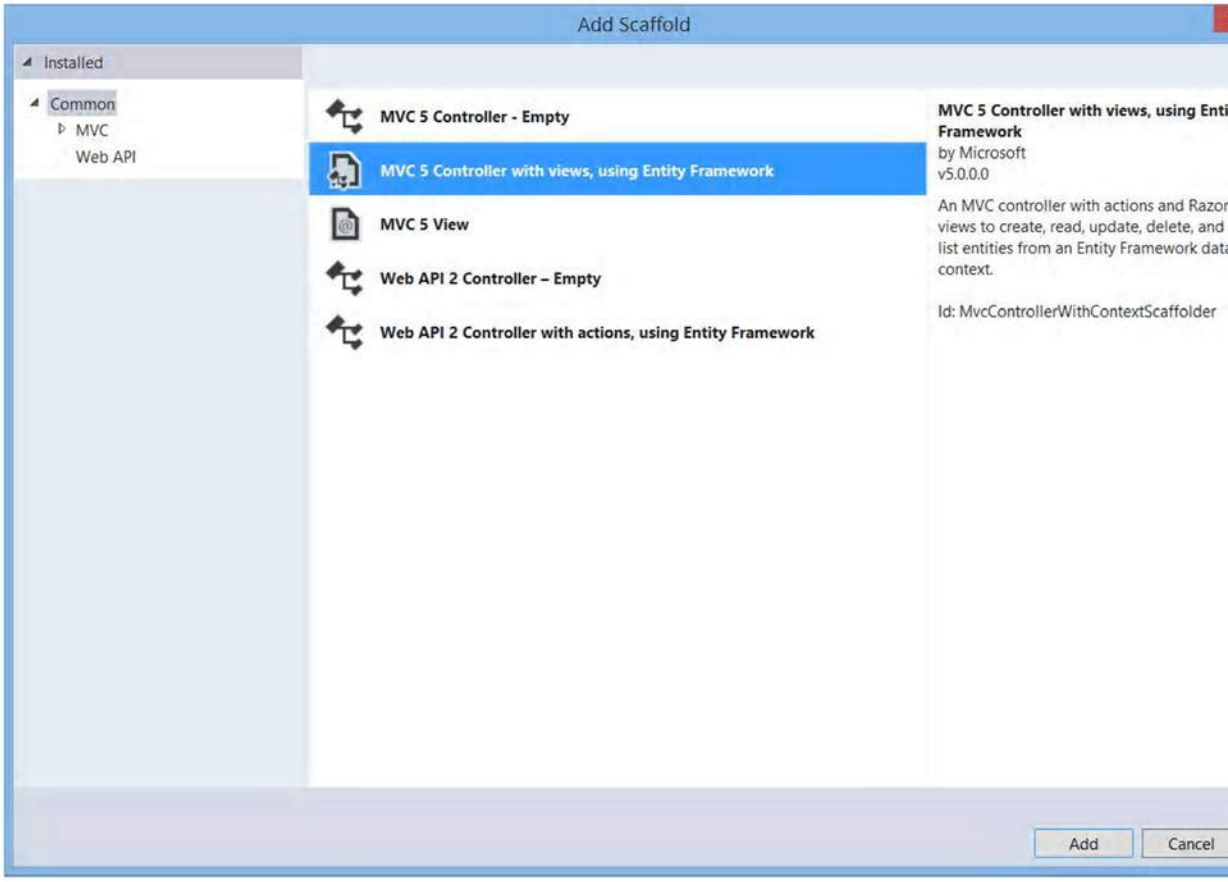


Figure 37: Selecting an Appropriate Controller for Scaffolding

As you can see, you can choose among different kinds of controllers. In this case you are working with the Entity Framework, so the appropriate controller is MVC 5 Controller with views, using Entity Framework. You can also choose an empty controller or a view. Also notice how you can take advantage of the Web API framework to create controllers that can be exposed to other consumers. When you click Add, Visual Studio shows the Add Controller dialog (see Figure 38).

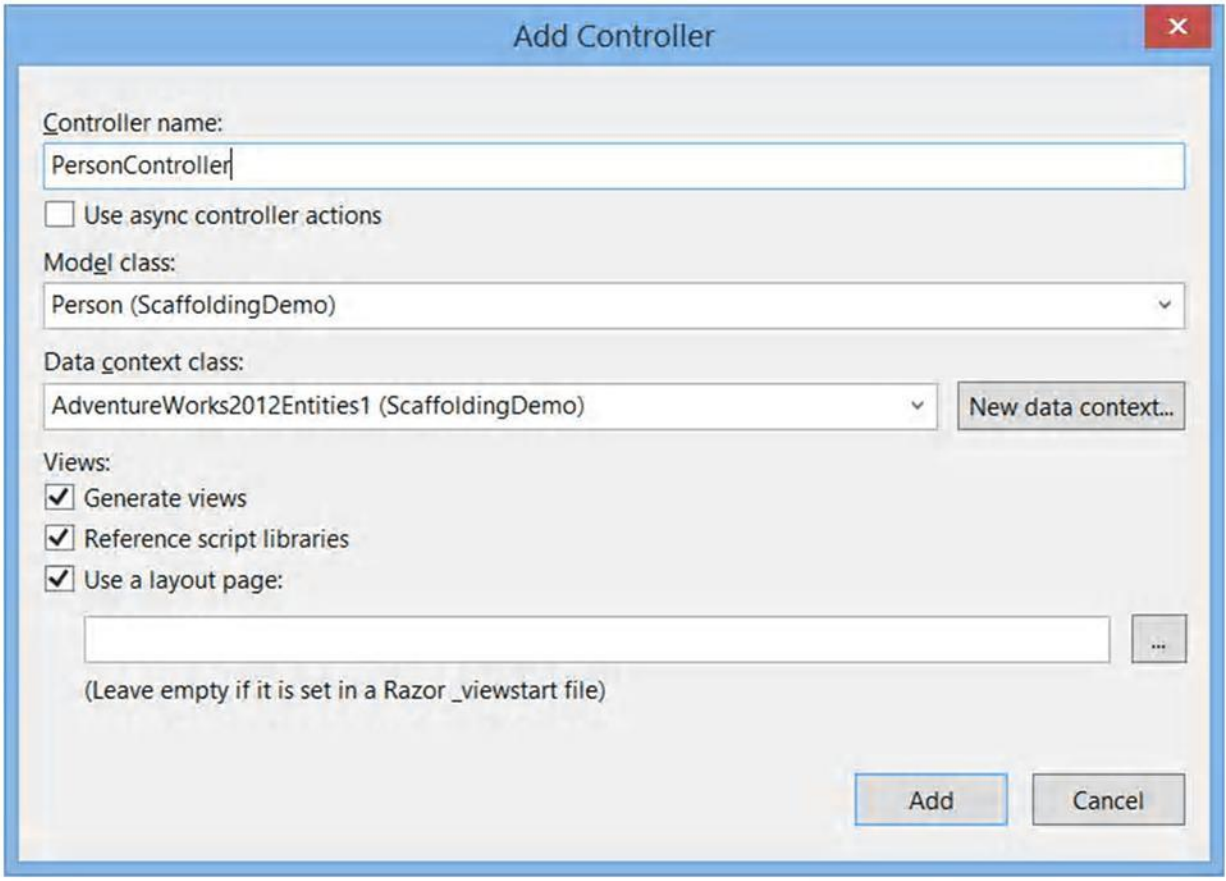


Figure 38: Specifying Properties and Settings for the New Controller

Here you can specify a number of settings for the new controller. The following table describes these settings and indicates how to rename items.

Table 1: Members and settings for the Add Controller dialog

Item name	Description	Value
Controller name	The name of the controller class that will be generated to interact with data	PersonController
Model class	The entity class that will be managed by the controller	Person

Item name	Description	Value
Data Context class	The context class generated by the Entity Framework to represent the database in a modeled way	AdventureWorks2012Entities
Generate views	Select this checkbox to make Visual Studio generate views (pages) for you	True
Reference script libraries	Select this checkbox to import scripting libraries	True
Use a layout page	Select this checkbox if you want to use a custom page for the layout; no need for this example	False

Click **Add**. In a few seconds, Visual Studio 2013 generates the **`PersonController`** class and a number of .cshtml files (under the Views\Person subfolder); each file is a page whose name is self-explanatory, such as Create.cshtml or Delete.cshtml. If you double-click the **`PersonController`** class in Solution Explorer, you will see the code that interacts with the data model. Figure 39 shows the result of scaffolding that is the controller and generated files in Solution Explorer.

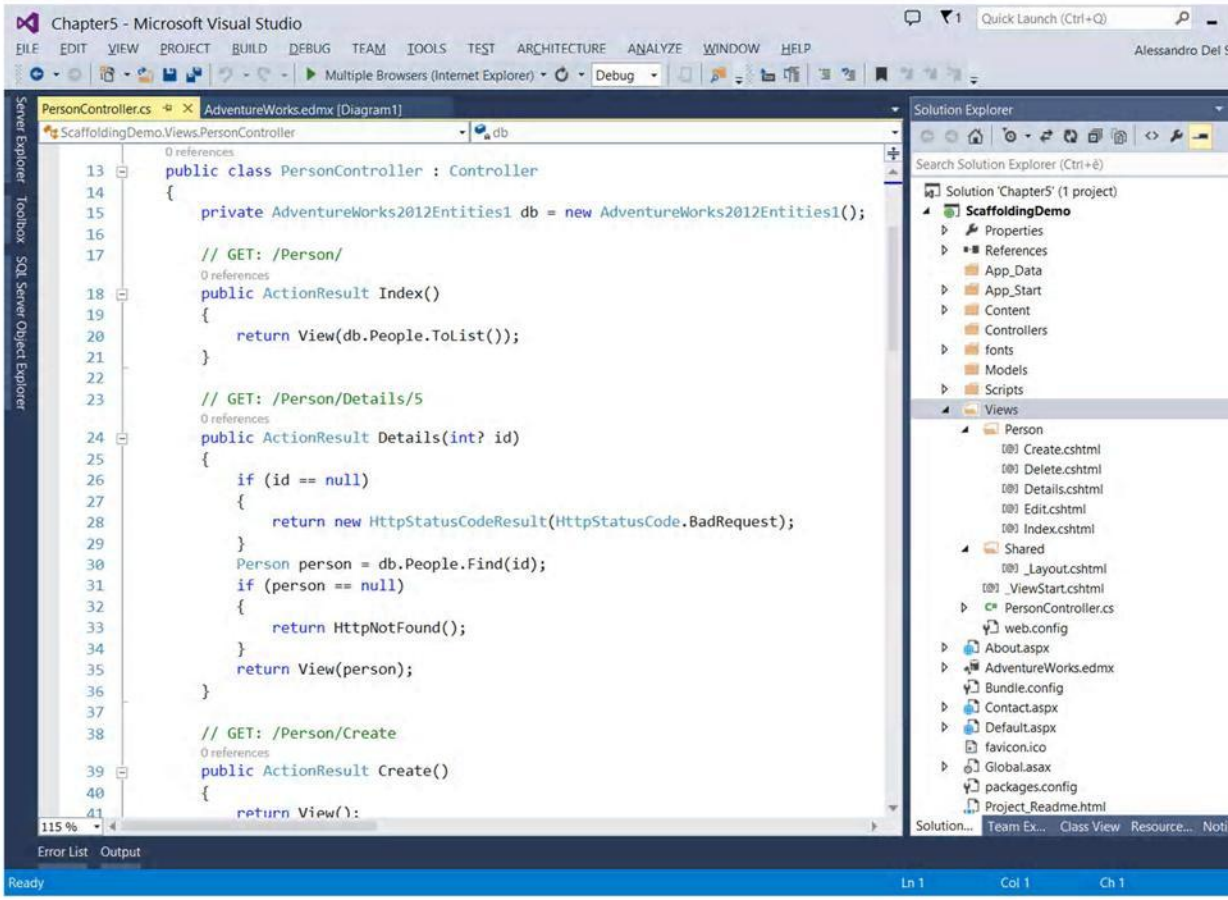


Figure 39: The Result of Scaffolding in Solution Explorer

The full code for the **PersonController** class follows.

Visual C#

```
    public class PersonController : Controller
    {
        private AdventureWorks2012Entities1 db = new
AdventureWorks2012Entities1();

        // GET: /Person/
        public ActionResult Index()
        {
            return View(db.People.ToList());
        }

        // GET: /Person/Details/5
        public ActionResult Details(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
```

```
            }
            Person person = db.People.Find(id);
            if (person == null)
            {
                return HttpNotFound();
            }
            return View(person);
        }

        // GET: /Person/Create
        public ActionResult Create()
        {
            return View();
        }

        // POST: /Person/Create
        // To protect from overposting attacks, please enable the specific
properties you want to bind to, for
        // more details see http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult
Create([Bind(Include="BusinessEntityID,PersonType,NameStyle,Title,FirstName
,MiddleName,LastName,Suffix,EmailPromotion,AdditionalContactInfo,rowguid,Mo
difiedDate")] Person person)
        {
            if (ModelState.IsValid)
            {
                db.People.Add(person);
                db.SaveChanges();
                return RedirectToAction("Index");
            }

            return View(person);
        }

        // GET: /Person/Edit/5
        public ActionResult Edit(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Person person = db.People.Find(id);
            if (person == null)
            {
                return HttpNotFound();
            }
            return View(person);
        }
```

```
        // POST: /Person/Edit/5
        // To protect from overposting attacks, please enable the specific
properties you want to bind to, for
        // more details see http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult
Edit([Bind(Include="BusinessEntityID,PersonType,NameStyle,Title,FirstName,M
iddleName,LastName,Suffix,EmailPromotion,AdditionalContactInfo,rowguid,Modi
fiedDate")] Person person)
        {
            if (ModelState.IsValid)
            {
                db.Entry(person).State = EntityState.Modified;
                db.SaveChanges();
                return RedirectToAction("Index");
            }
            return View(person);
        }

        // GET: /Person/Delete/5
        public ActionResult Delete(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Person person = db.People.Find(id);
            if (person == null)
            {
                return HttpNotFound();
            }
            return View(person);
        }

        // POST: /Person/Delete/5
        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public ActionResult DeleteConfirmed(int id)
        {
            Person person = db.People.Find(id);
            db.People.Remove(person);
            db.SaveChanges();
            return RedirectToAction("Index");
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
```

```
            {
                db.Dispose();
            }
            base.Dispose(disposing);
        }
    }
```

Visual Basic

```
Imports System
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.Entity
Imports System.Linq
Imports System.Net
Imports System.Web
Imports System.Web.Mvc

Namespace ScaffoldingDemo
    Public Class PersonController
        Inherits System.Web.Mvc.Controller

        Private db As New AdventureWorks2012Entities

        ' GET: /Person/
        Function Index() As ActionResult
            Return View(db.People.ToList())
        End Function

        ' GET: /Person/Details/5
        Function Details(ByVal id As Integer?) As ActionResult
            If IsNothing(id) Then
                Return New HttpStatusCodeResult(HttpStatusCode.BadRequest)
            End If
            Dim person As Person = db.People.Find(id)
            If IsNothing(person) Then
                Return HttpNotFound()
            End If
            Return View(person)
        End Function

        ' GET: /Person/Create
        Function Create() As ActionResult
            Return View()
        End Function

        ' POST: /Person/Create
```

```
        'To protect from overposting attacks, please enable the specific
properties you want to bind to, for
        'more details see http://go.microsoft.com/fwlink/?LinkId=317598.
        <HttpPost()>
        <ValidateAntiForgeryToken()>
        Function Create(<Bind(Include :=
"BusinessEntityID,PersonType,NameStyle,Title,FirstName,MiddleName,LastName,
Suffix,EmailPromotion,AdditionalContactInfo,rowguid,ModifiedDate")> ByVal
person As Person) As ActionResult
            If ModelState.IsValid Then
                db.People.Add(person)
                db.SaveChanges()
                Return RedirectToAction("Index")
            End If
            Return View(person)
        End Function

        ' GET: /Person/Edit/5
        Function Edit(ByVal id As Integer?) As ActionResult
            If IsNothing(id) Then
                Return New HttpStatusCodeResult(HttpStatusCode.BadRequest)
            End If
            Dim person As Person = db.People.Find(id)
            If IsNothing(person) Then
                Return HttpNotFound()
            End If
            Return View(person)
        End Function

        ' POST: /Person/Edit/5
        'To protect from overposting attacks, please enable the specific
properties you want to bind to, for
        'more details see http://go.microsoft.com/fwlink/?LinkId=317598.
        <HttpPost()>
        <ValidateAntiForgeryToken()>
        Function Edit(<Bind(Include :=
"BusinessEntityID,PersonType,NameStyle,Title,FirstName,MiddleName,LastName,
Suffix,EmailPromotion,AdditionalContactInfo,rowguid,ModifiedDate")> ByVal
person As Person) As ActionResult
            If ModelState.IsValid Then
                db.Entry(person).State = EntityState.Modified
                db.SaveChanges()
                Return RedirectToAction("Index")
            End If
            Return View(person)
        End Function

        ' GET: /Person/Delete/5
        Function Delete(ByVal id As Integer?) As ActionResult
            If IsNothing(id) Then
```

```
                Return New HttpStatusCodeResult(HttpStatusCode.BadRequest)
            End If
            Dim person As Person = db.People.Find(id)
            If IsNothing(person) Then
                Return HttpNotFound()
            End If
            Return View(person)
        End Function

        ' POST: /Person/Delete/5
        <HttpPost()>
        <ActionName("Delete")>
        <ValidateAntiForgeryToken()>
        Function DeleteConfirmed(ByVal id As Integer) As ActionResult
            Dim person As Person = db.People.Find(id)
            db.People.Remove(person)
            db.SaveChanges()
            Return RedirectToAction("Index")
        End Function

        Protected Overrides Sub Dispose(ByVal disposing As Boolean)
            If (disposing) Then
                db.Dispose()
            End If
            MyBase.Dispose(disposing)
        End Sub
    End Class
End Namespace
```

Although lengthy, the code is not difficult; you have a number of methods responsible for C.R.U.D. operations, such as **Create**, **Edit**, **Details**, **Delete**, and **Index**. The latter returns the full list of records in the table mapped by the model class. Every method returns an object of type **ActionResult**, which is exposed by the MVC framework (remember you are using Web Forms here). It encapsulates the result of the aforementioned methods and is used to perform framework-level operations on behalf of the method. Of course, you can make additional edits to the controller class if you wish. In this particular example, it is a good idea to restrict the number of people returned by the **Index** method, in order to speed up the process. Imagine you want to retrieve only the first ten people in the table. You can edit the **Index** method as follows:

Visual C#

```
        // GET: /Person/
        public ActionResult Index()
        {
            return View(db.People.Take(10).ToList());
        }
```

Visual Basic

```
        ' GET: /Person/
        Function Index() As ActionResult
            Return View(db.People.Take(10).ToList())
        End Function
```

You can use any LINQ operator to edit the query in a way that best fits your needs. In this case, the code uses **Take** to retrieve the first 10 records. It is worth mentioning that for each method in the code you will find some comments that explain how to invoke the related page in the browser. For instance, if you want to view the list of records in the table, you will use the /Person relative URL (see the code in the previous listing). You will get a demonstration shortly.

Test the application

Press **F5** to start debugging the application. Your default web browser will open and show the default page of the application. In the address bar, type the /Person relative URL at the end (see the highlight in Figure 40). The application will show the first ten people in the table at this point, as you can see in Figure 40.

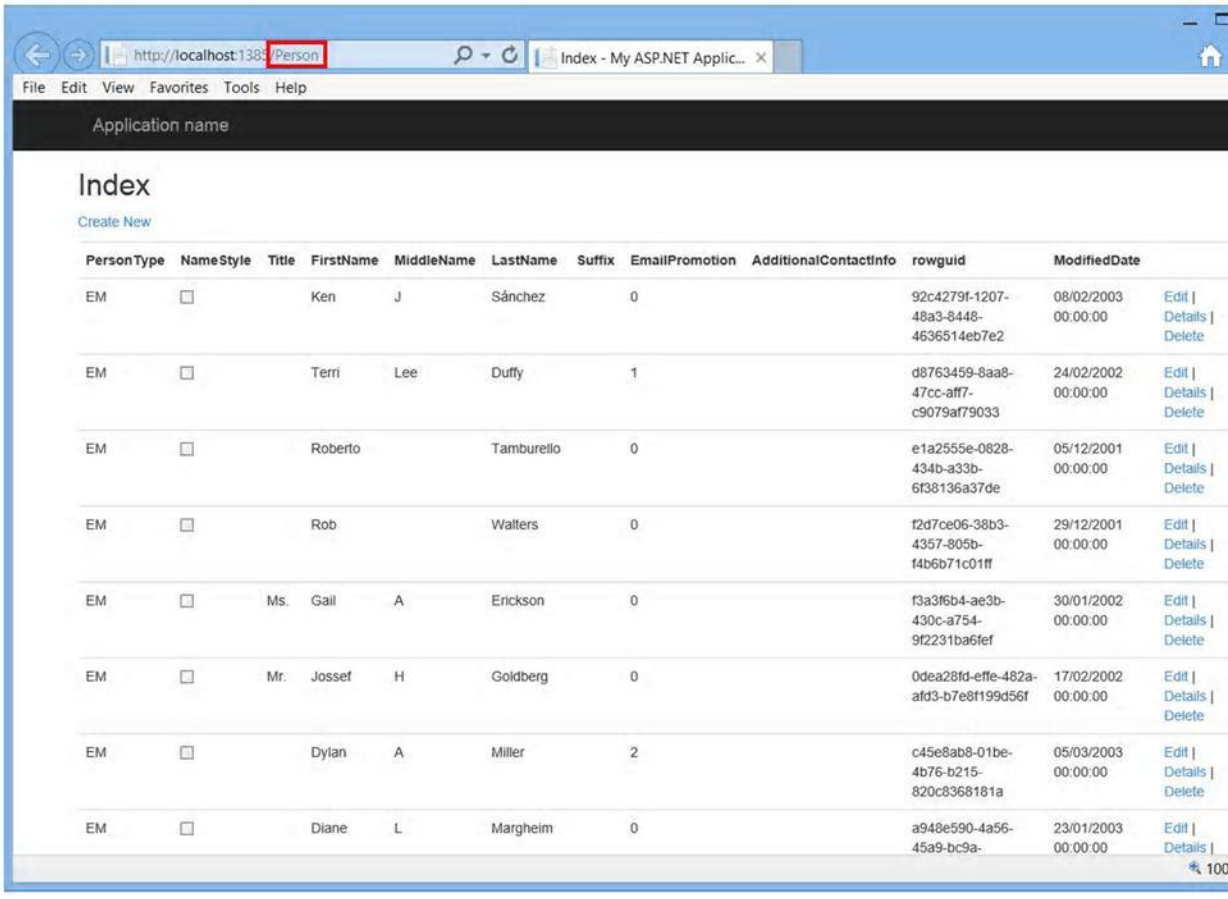


Figure 40: The application shows a list of items.

Notice the shortcuts to pages for data operations. For example, you can click **Edit** to see and change details of an item. This is also useful to understand how the application invokes the appropriate methods in the controller. Take a look at Figure 41, which shows how to edit an existing item.

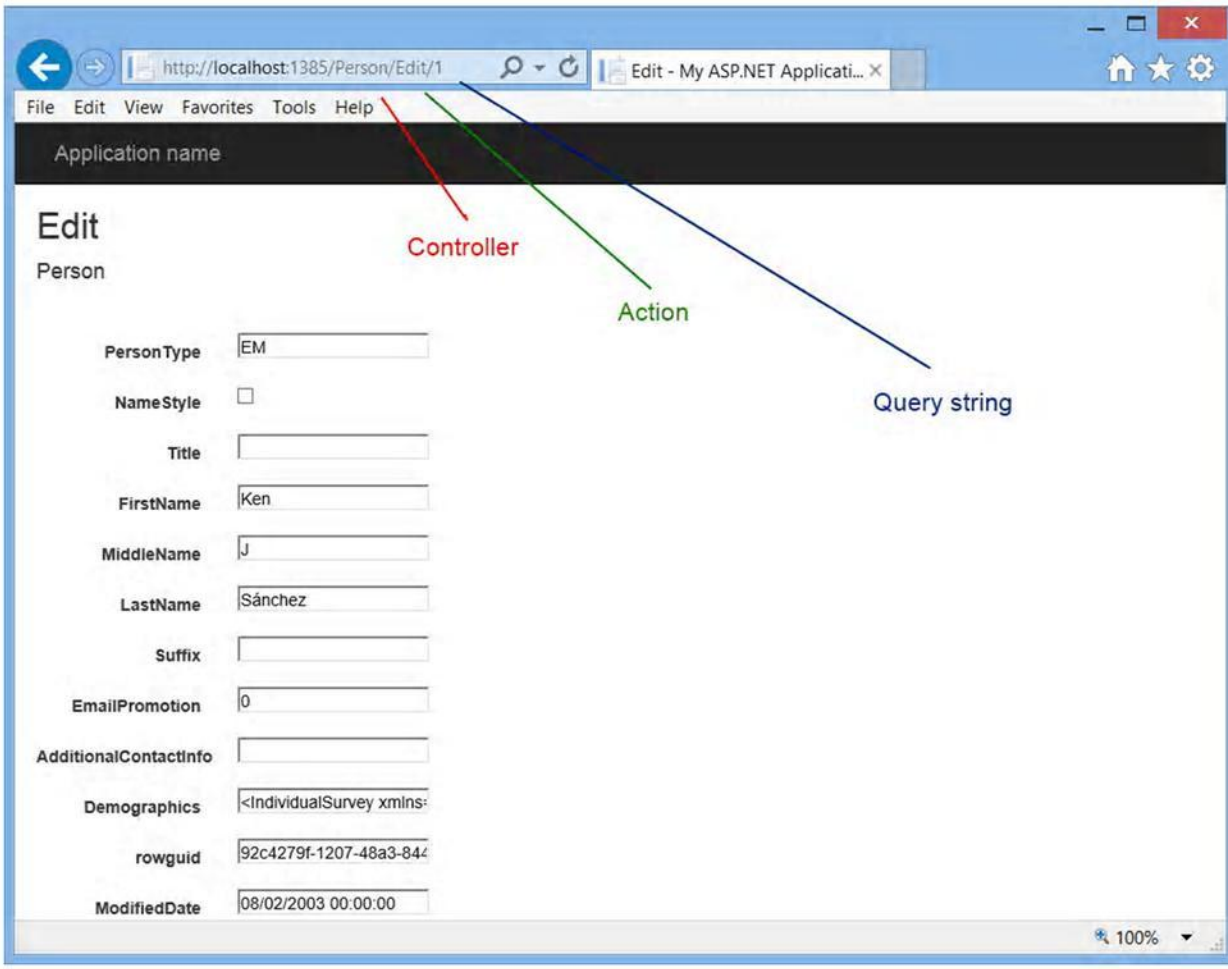


Figure 41: Editing an Item and the Anatomy of the Address

Apart from seeing how easy it is to edit an item without writing any code, notice look at the address bar. The address is made of the following elements:

- The web address for the application
- The name of the controller (in this case Person)
- The name of the method in the controller (in this case Edit)
- The value for the query string that will be used by the invoked method to retrieve a specific item

You can go back to the previous page and select **Create New** to see how easy it is to add a new person to the database (see Figure 42).

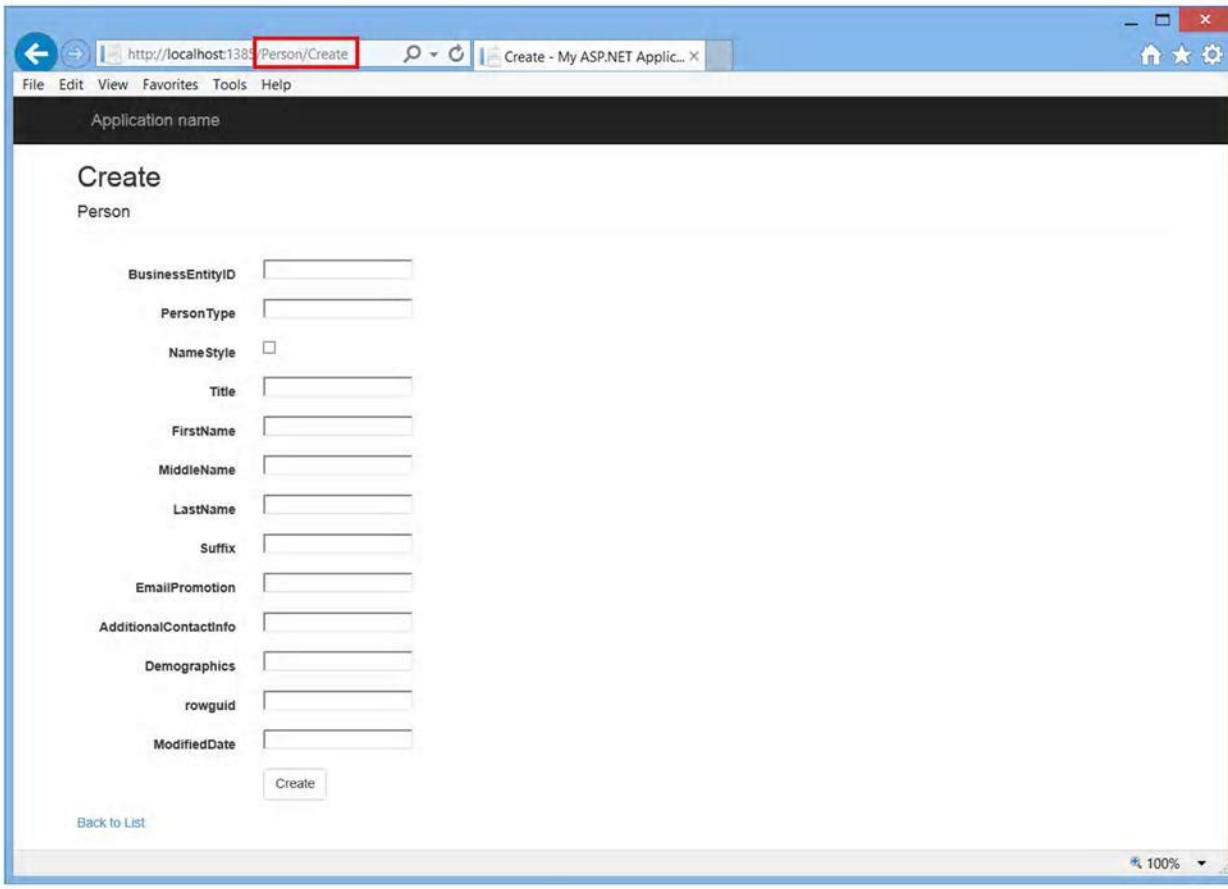


Figure 42: Adding a New Item

Figure 42 also highlights how both the controller name and the method name are used in the address. So you have seen how to take advantage of scaffolding in a Web Forms application due to the new One ASP.NET experience. Pages generated by Visual Studio can be edited to provide a different layout, but it is evident how with a minimal effort you can create powerful data-centric applications.

Browsers Link Dashboard

You already know that with Visual Studio you can test your web applications with different browsers. In Visual Studio 2013, if you have your application running in different browsers and you make changes in the IDE, such changes can be refreshed to every browser with a single click. This is possible because Visual Studio 2013 and the .NET Framework 4.5.1 use SignalR 2.0, the popular library that allows sending real-time notifications.

To understand how it works, first create a new ASP.NET project called BrowserLinkDemo. By following the lesson learned in the previous section, select the **Web Forms** template and the anonymous authentication. Of course, this feature works not only with Web Forms, but also with ASP.NET MVC.

Note: Do not delete the new project until you complete this chapter. It will be used again later when discussing the new tools for Windows Azure.

The next step is telling Visual Studio to use multiple browsers to run the application, so if you did not do this before, select the arrow near the Start button on the toolbar (see Figure 43), then **Browse With**.

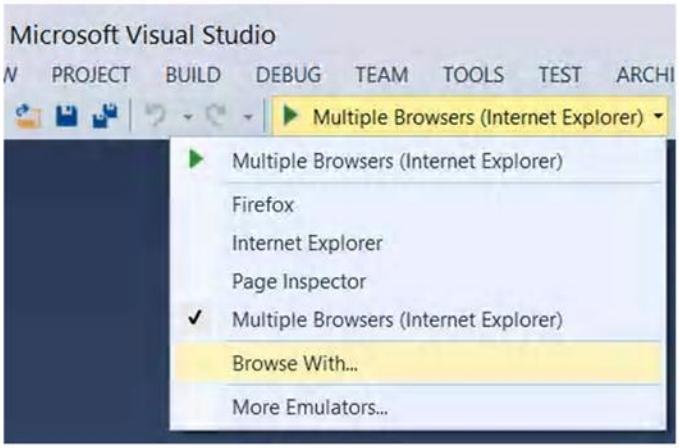


Figure 43: Accessing the Command to Change the Default Browser(s) for Testing

In the Browse With dialog, select two or more web browsers. On my machine, I have Internet Explorer and FireFox installed, so my selection includes both (see Figure 44).

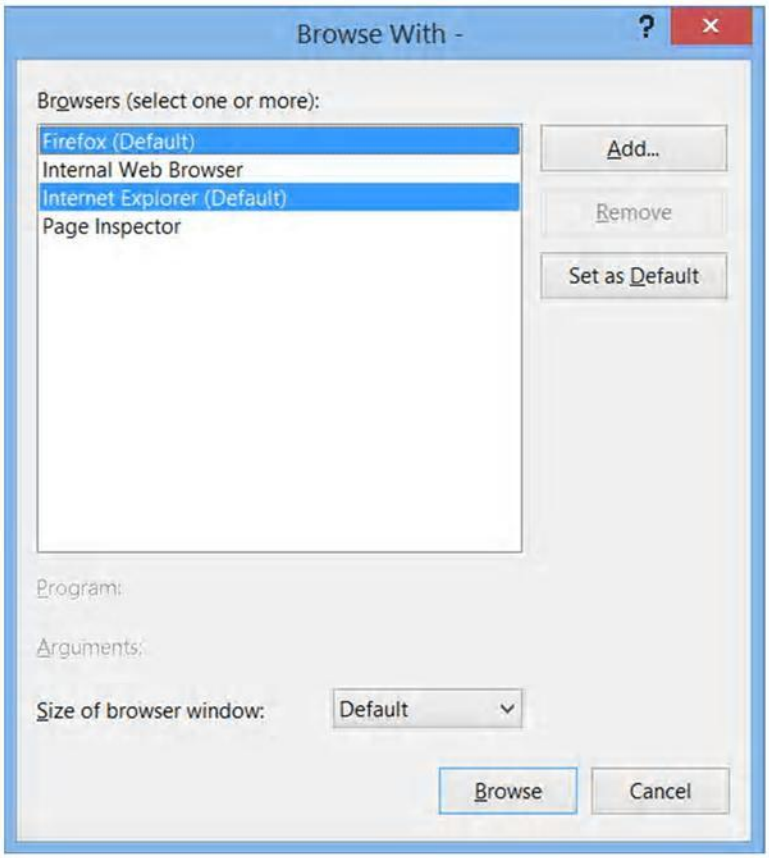


Figure 44: Selecting Two or More Web Browsers for Testing

Press **CTRL** and click on each browser you want to use, then select **Set as Default** and close the dialog. Now, instead of debugging the usual way with F5, press **CTRL + F5** to start without debugging. This way, the application will be launched in all of the browsers you selected. Figure 45 shows the application running inside Internet Explorer.

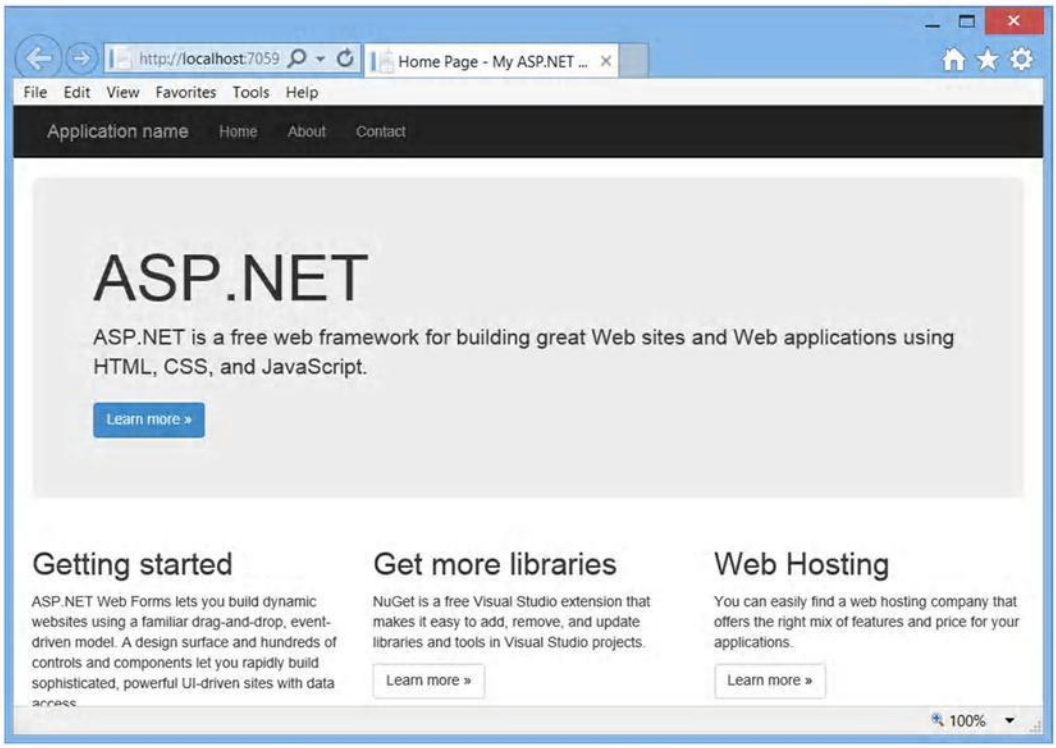


Figure 45: The Sample Application Running

Now let's make a very simple edit in Visual Studio. Open the Default.aspx page in the designer and replace the ASP.NET string with **Visual Studio 2013 Succinctly**. On the **Standard** toolbar, click the **Refresh** button, which you can see highlighted in Figure 46, or press **CTRL+ALT+Enter**.

Figure 46: The Refresh Button for Linked Browsers

Now switch back to both browsers; you will see how they show the updated string. With this technique, you do not need to stop the application, make your edits, and restart debugging. By clicking the arrow near the Refresh button, you can access additional shortcuts, including the one to enable the Browser Link Dashboard tool window. With the Browser Link Dashboard you can see connections for each application in the solution and you can take specific action for each linked browser, such as refreshing only one instead. Figure 47 shows what the Browser Link Dashboard looks like against a solution containing both sample projects described in this chapter.

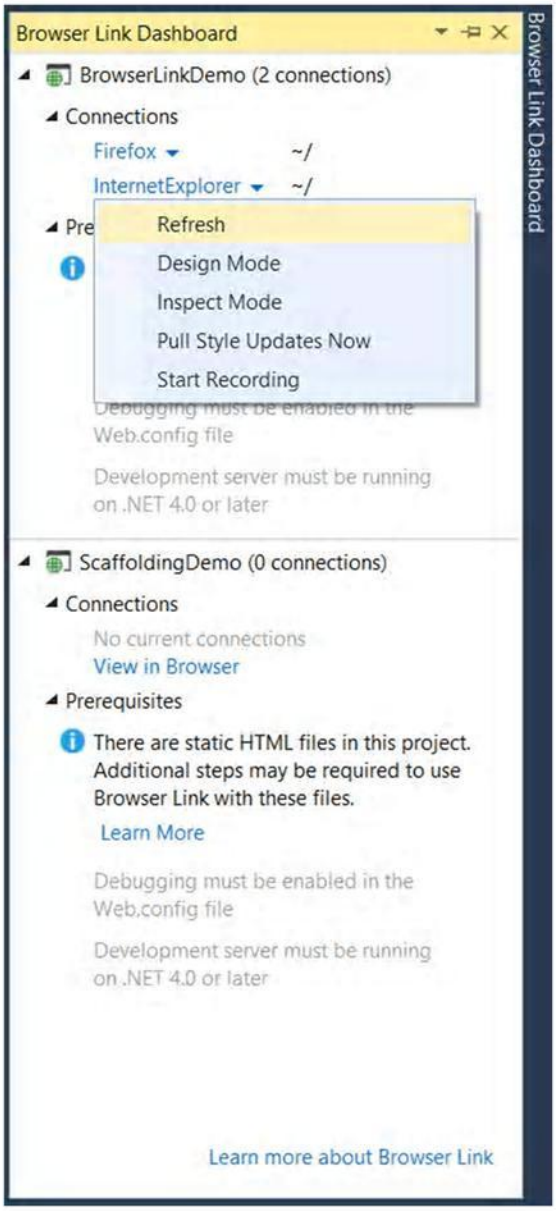


Figure 47: The Browser Link Dashboard allows managing actions for single connections.

With Scaffolding and Browsers Link, you have seen two relevant features in Visual Studio 2013. But this new release has a great focus on the cloud. This is discussed in the next section.

What's new in Windows Azure

Windows Azure is the popular cloud computing platform from Microsoft, first unveiled at the Professional Developer Conference (PDC) in 2008. Over the years, Windows Azure has dramatically evolved by introducing tons of services, and the cloud has become an important part of our daily lives. Many websites, mobile devices, and applications use Windows Azure's services. If you've ever developed applications for Windows Azure before, you know that you had to do most things outside of Visual Studio, using the Windows Azure Developer Portal on the web or special client applications; in fact, Visual Studio lacked a good integration with the platform. As for other platforms, Visual Studio 2013 solves the problem and provides deep integration with Windows Azure making it easy to manage many services from within the IDE. This chapter provides guidance on how Visual Studio 2013 integrates with Azure and on how you can leverage this integration to build applications faster.

What you need before reading this section

Because we focus on the new tooling in Visual Studio 2013 for Windows Azure, we assume you already have at least a basic knowledge of the platform, including information about paid services and pricing. In fact, this chapter can neither summarize all services offered by Windows Azure nor it can explain programming for Azure, since this would require an entire book. If you need an overview of programming for Windows Azure before reading this chapter, refer to the official documentation available at http://www.windowsazure.com/en-us/documentation/ . In order to complete the steps described in this chapter, you must install the Windows Azure SDK 2.2 or later.

Note: This chapter describes Windows Azure from the developer's perspective and will describe services that you can access only if you have a current subscription. But Windows Azure is a paid service. Fortunately, you can enable a 30-day trial at http://www.windowsazure.com/en-us/pricing/free-trial/. If you want to fully understand the rest of this chapter, you are encouraged to subscribe the trial.

Server Explorer window

You've probably already used the Server Explorer tool window in Visual Studio many times, for different purposes such as managing connections to databases, or web servers. In Visual Studio 2013, Server Explorer offers a new node called Windows Azure, which allows connecting to your subscription and managing services from within the IDE. Figure 48 shows how the Windows Azure node looks.

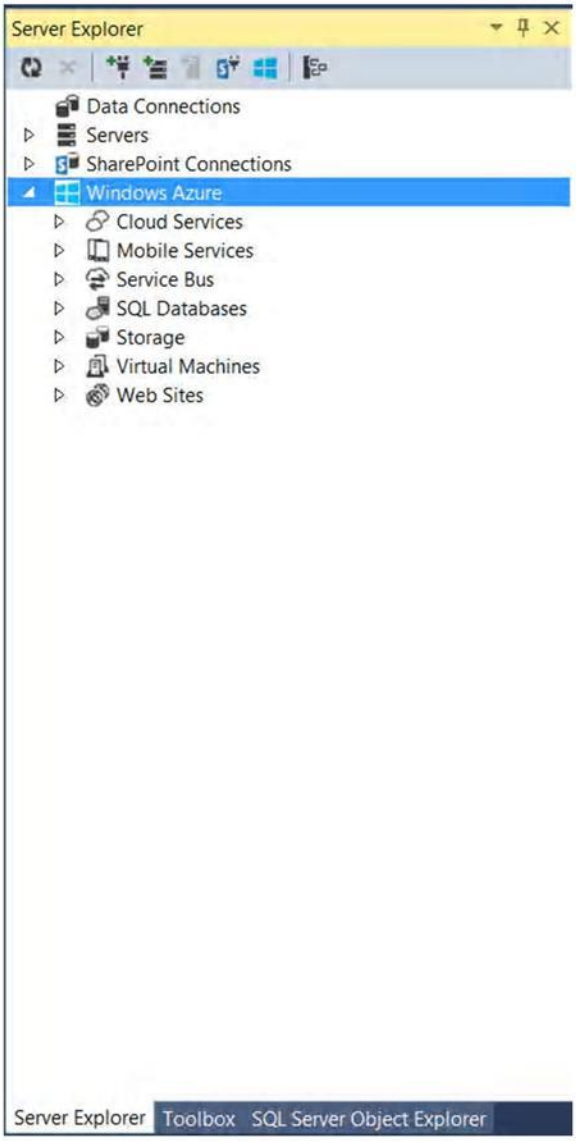


Figure 48: The New Windows Azure Tooling in the Server Explorer Window

As you can see, you can manage most services without leaving Visual Studio. The first thing you need to do is associate a valid Windows Azure subscription to Visual Studio 2013. To do so, right-click **Windows Azure** and select **Connect to Windows Azure**. You will be asked to enter the Microsoft Account of your current subscription. If you have multiple subscriptions associated with your Microsoft Account, you will be able to manage subscriptions. Simply right-click again **Windows Azure** and select **Manage Subscriptions**. At this point, the Manage Windows Azure Subscriptions appears, as shown in Figure 49.

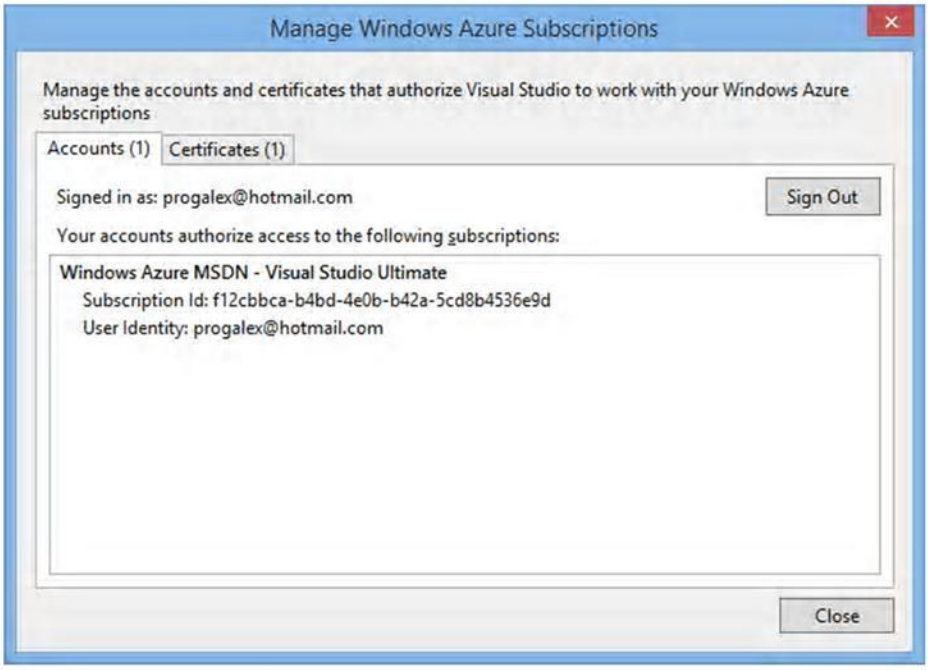


Figure 49: Managing Windows Azure Subscriptions

You will need to import your subscription settings in Visual Studio 2013 at this point. Click **Certificates**, then **Import**. In the appearing Import Windows Azure Subscriptions dialog (see Figure 50), click the **Download subscription file** hyperlink.

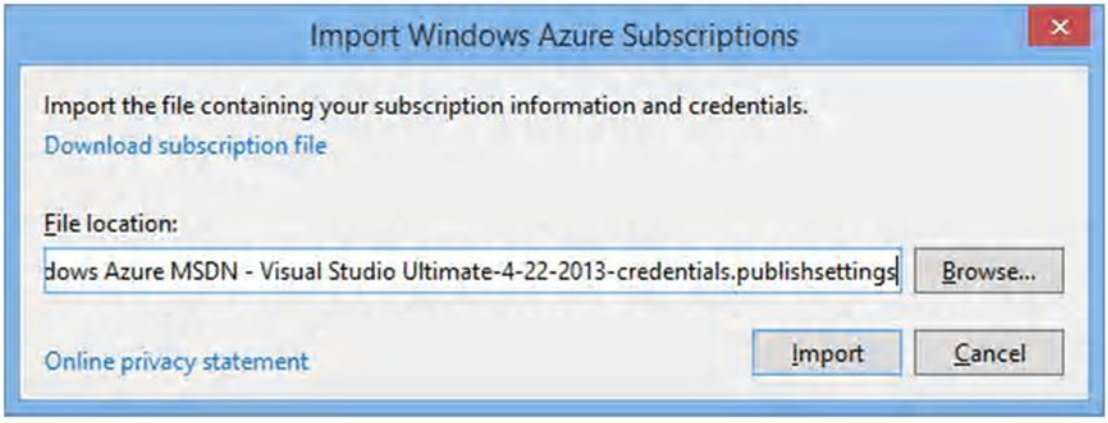


Figure 50: Managing Windows Azure Subscriptions

At this point, a page in the Windows Azure's website will be opened and a file with your subscription's settings will be automatically available for downloading. Once you've downloaded the settings file, click **Browse** and in the dialog search for the downloaded settings file, select it, and finally click **Import**. By following these steps, your subscription will be added to the list of subscriptions in Visual Studio. If you have added multiple subscriptions, in the Manage Windows Azure Subscriptions dialog you will be able to select the subscription you want to work with. Click **Close** when you have made your selection. Let's now see in more detail what you can do with Server Explorer.

Integration with mobile services

Mobile Services can be used as a backend for mobile applications. A very common use of Mobile Services is storing data inside tables. In Visual Studio 2013 you can now create a mobile service directly, add tables, and see logs for the service. To create a new service, right-click **Mobile Services**, and then select **Create Service**. In the Create Mobile Service dialog you will have to specify the new service's details, as shown in Figure 51.

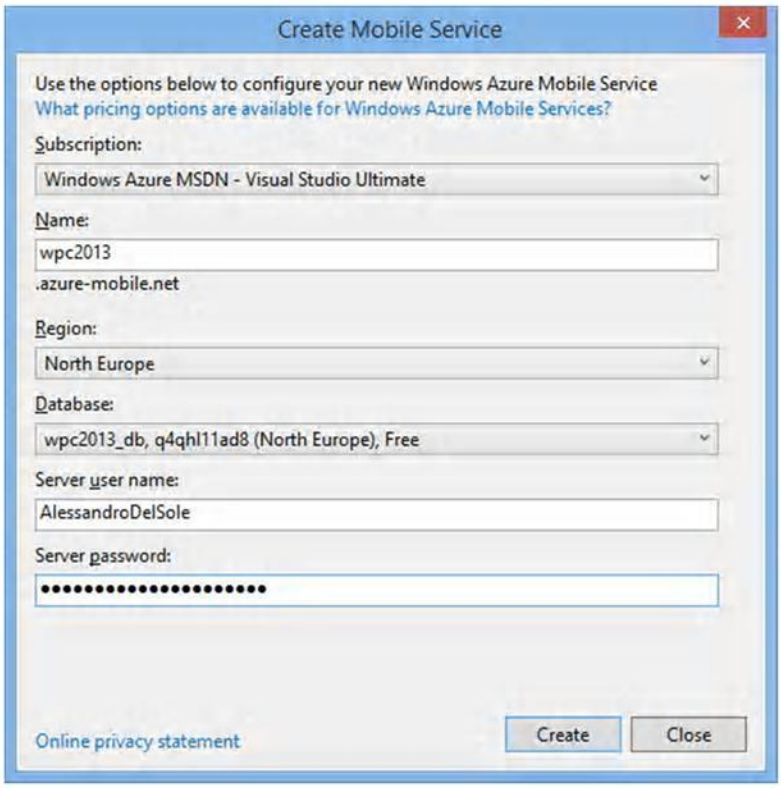


Figure 51: Creating a New Mobile Service

You will need to specify the following information:

- The subscription where the service will be created. You can leave the selection unchanged if you want the service to be created in the current subscription.
- The service name. Visual Studio 2013 will check the availability of the service name as you type.
- The Region. Remember to select the region that is nearest to your location.
- A new or existing SQL Azure database to be associated with the new service (optional). If you did not create a SQL Azure database in the Management Portal before, you can use the **Create a free SQL database** option or the **Create a billed SQL database**. Of course, I strongly recommend to create a free SQL Azure database (up to 20 MB) rather than a billed one.
- User name and password for the SQL Azure server in order to access the database.

Note: I'm assuming you have already configured a SQL Azure server, since I'm talking about server user name and password required to access a database on the cloud. If you

did not configure your SQL Azure server, you can get started by reading the official documentation.

When you're finished, click **Create**. The new service will appear under the Mobile Services node of Server Explorer. You can also add tables to the service directly. Right-click the service's name and select **Create Table**. In the Create Table dialog (see Figure 52) you can specify the table name and permission for each of the C.R.U.D. operations.

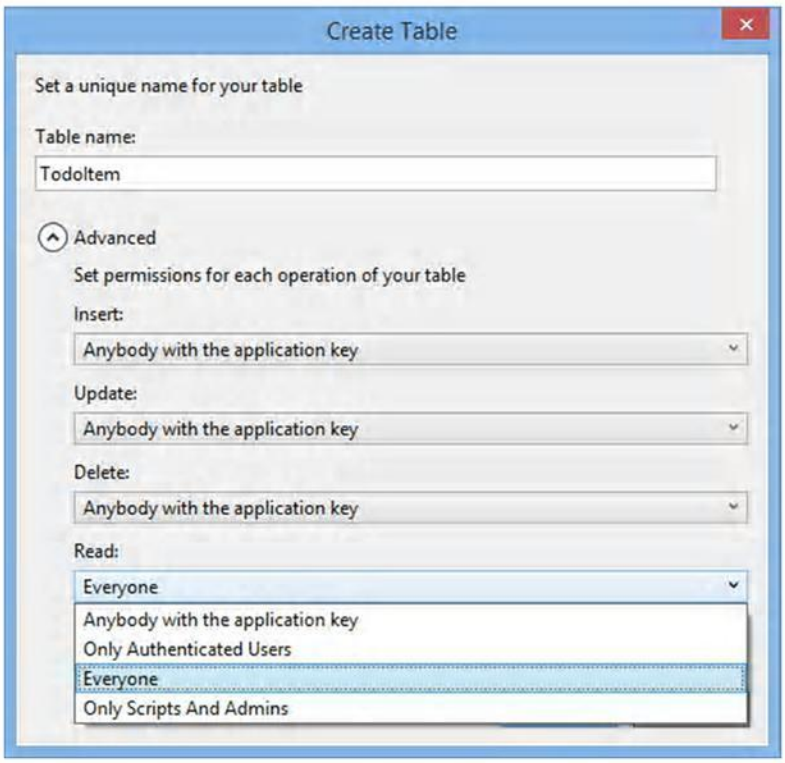


Figure 52: Creating a New Mobile Service

Now create a new table called **TodoItem**. We will use this table in the next chapter when demonstrating Windows 8 support for mobile services. Notice how you have deep control over permissions; you can allow everyone, authenticated users, administrators, or users having the application key (managing the application key is only available in the management portal). Click **Create** when you're ready. You will be able to see the new table as a node under the mobile service's name. Also, if you expand the table name, you will see a number of JavaScript code files, which are responsible of performing operations against data, such as read, insert, update, and delete. Figure 53 shows how the Server Explorer window appears at this point.

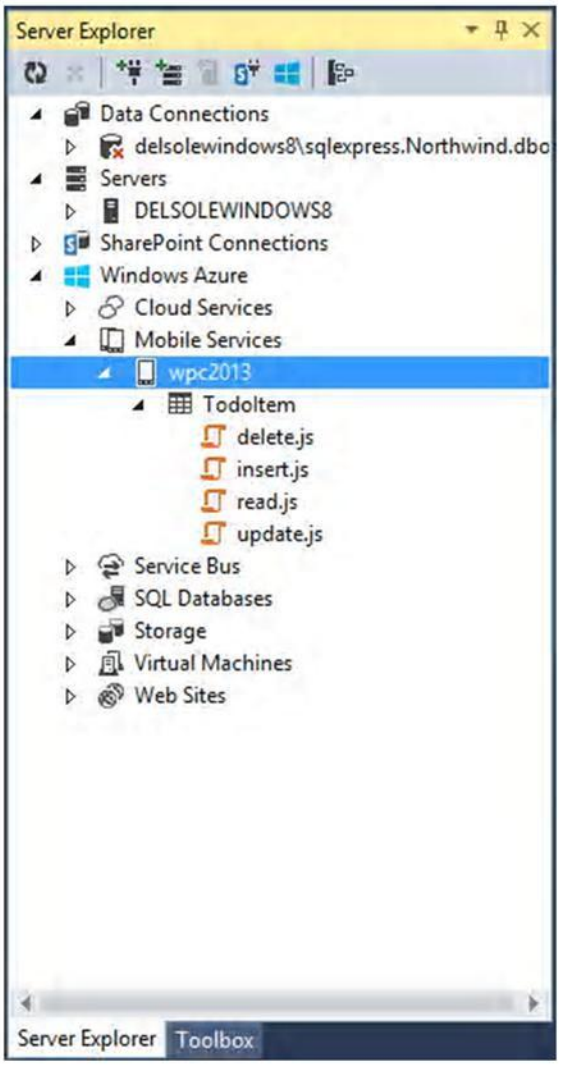


Figure 53: Mobile service, table, and JavaScript files as they appear in Server Explorer.

With only a few steps, you have created a mobile service that can be used as a backend in your mobile applications. You can use mobile services in the following applications:

- Windows Store apps for Windows 8.x
- Windows Phone apps
- ASP.NET web applications

You will need to right-click the project name in Solution Explorer and select **Add Connected Service**. This is discussed further in Chapter 7.

Integration with Azure websites and cloud services

In Windows Azure, you publish your web applications through websites or cloud services. A website is a simplified, pre-configured environment for easy deployment. A cloud service, instead, is a highly customizable environment that requires you to make some configurations before deployment, and where you can take a lot of actions over the system. You can find the full description of both environments in this page of the Windows Azure documentation (I recommend that you read it). Whatever your choice is, Visual Studio 2013 supports publishing applications to both web sites and cloud services.

Note: You can connect to Virtual Machines from Server Explorer, but you can only publish to websites and cloud services directly.

For instance, imagine you want to deploy the sample application created earlier in this chapter to demonstrate the Browser Link feature. In Server Explorer, right-click the **Web Sites** node and then select **Add New Site**. A new dialog called Create site on Windows Azure appears. Here you will have to specify details for the new website, as shown in Figure 54.

Figure 54: Creating a New Website

The following information is required:

- The site name, an identifier that will be used to construct the website's URL. Visual Studio will check for the availability of the name as you type.
- The location: Microsoft has several data centers across the world, so ensure you select the location nearest to you.
- Database server, database username, and database password. These are optional fields; you only need to supply them if your application will use an SQL Azure database.

Tip: The sample application does not use any database. For the sake of completeness, Figu 54 shows how to fill database-related fields.

Once you have entered all the required information, click **Create**. Once the website has been created, double-click its name in Server Explorer so that Visual Studio shows a configuration window (see Figure 55).

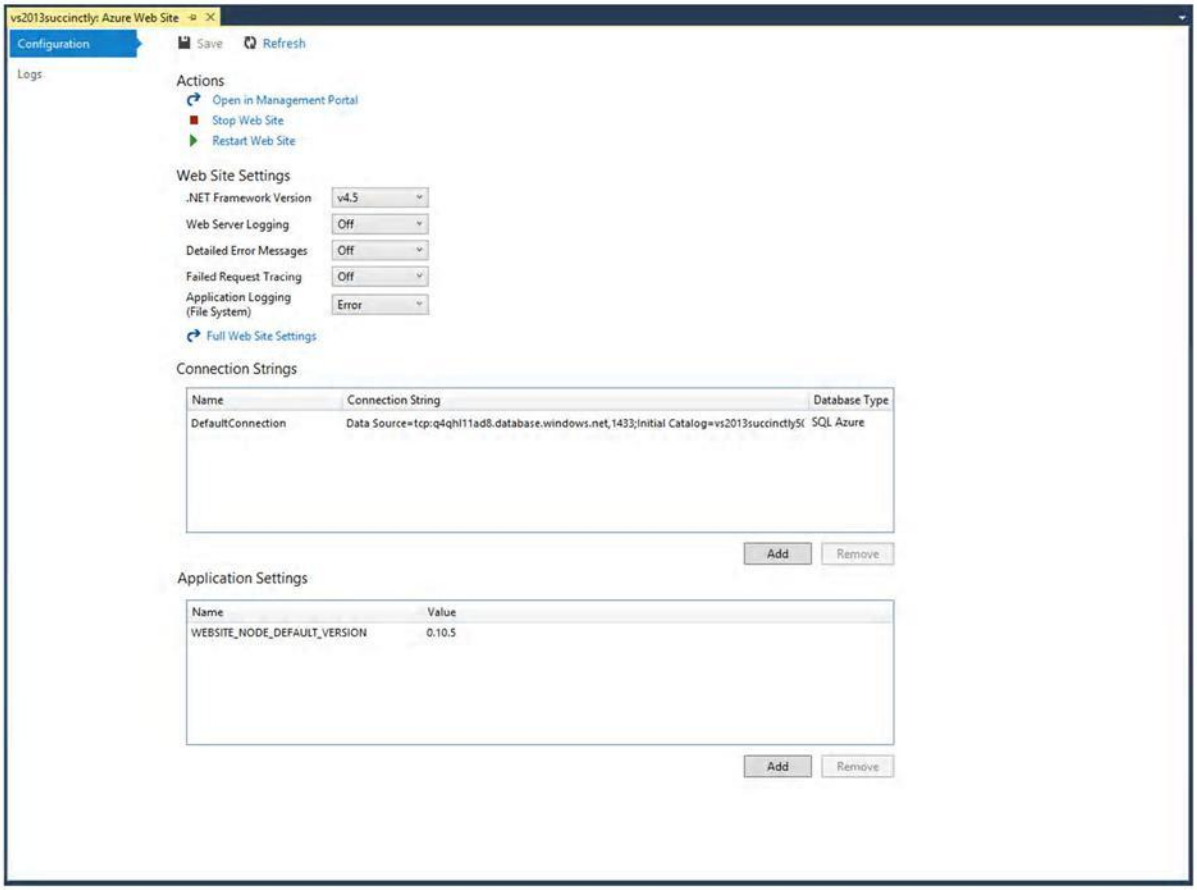


Figure 55: Configuration Window for the New Website

The window is divided into four main areas:

- Actions: here you find shortcuts to common tasks such as opening the site in the Azure's management portal, stopping or restarting the site.
- Web Site Settings: here you can fine-tune the configuration for your website, including error management and tracing.
- Connection Strings: here you can manage connection strings that your application uses to connect to data sources.
- Application Settings: this allows listing and adding environment variables for your websites. You should never change or remove settings added by Visual Studio.

Now you want to publish the BrowserLinkDemo sample application to the newly created website. First, right-click the project name in Solution Explorer, then select Convert, Convert to

Windows Azure Cloud Service Project. This action will add a new project with a Windows Azure role to the solution. By default, the new project's name has the same name of the original project plus the .Azure suffix, in our example it is BrowserLinkDemo.Azure. Now, in Solution Explorer right-click the original project (BrowserLinkDemo) and then click Publish.

Tip: Use the Publish command if you want to deploy your application to a website. Because this is our current case, we are using this option. If you want to deploy to a Cloud Service instead, right-click the project name and select Publish to Windows Azure. A specific dialog will allow creating a new Cloud Service and easily deploy the application.

The Publish Web dialog appears and has no preconfigured settings, so you need to click the **Import** button to specify the target website. Visual Studio shows the Import Publish Settings dialog at this point, which is visible in Figure 55.

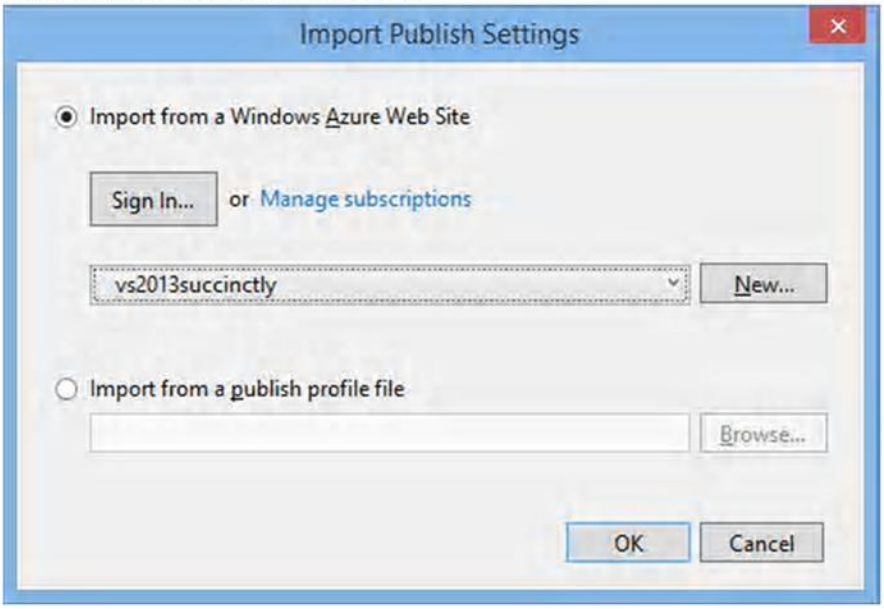


Figure 56: Selecting the Website for Publishing

Since the IDE is still connected to your Azure subscription, you can simply select the target website from the first combo box. You can also select a different subscription by clicking **Sign In** or by importing an existing publish profile file.

Tip: Visual Studio 2013 makes it easy to download and publish profiles from your Windows Azure subscription. Just right-click a website in Server Explorer and then click Download Publish Profile.

When you're ready, click **OK**. The Profile form in the Publish Web dialog will be filled with information coming from the publish profile you just selected. Go ahead to the **Connection** form (see Figure 57).

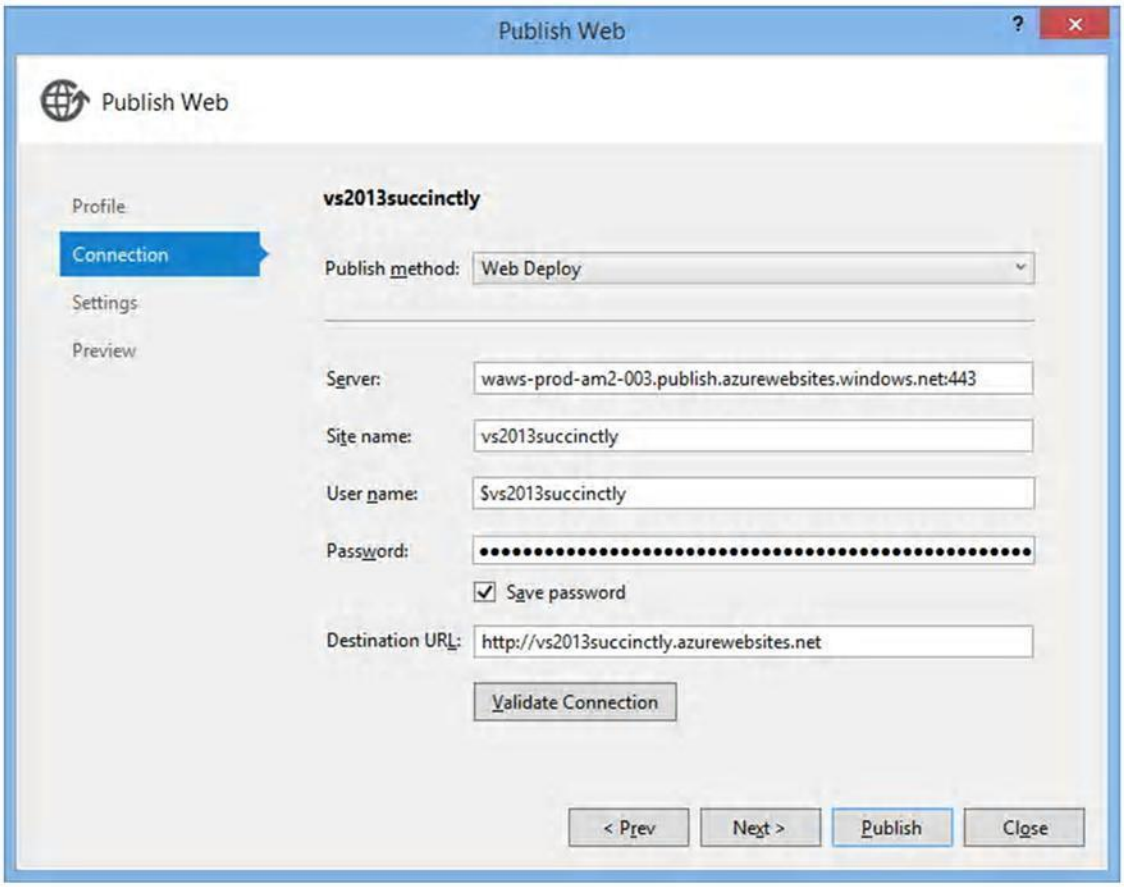


Figure 57: Connection Settings for the Application

Visual Studio 2013 automatically provides information for this form so you do not need to change any field. Notice the content of the Destination URL field, which contains the web address for your application once published. Click **Next** to access the Settings form (see Figure 58).

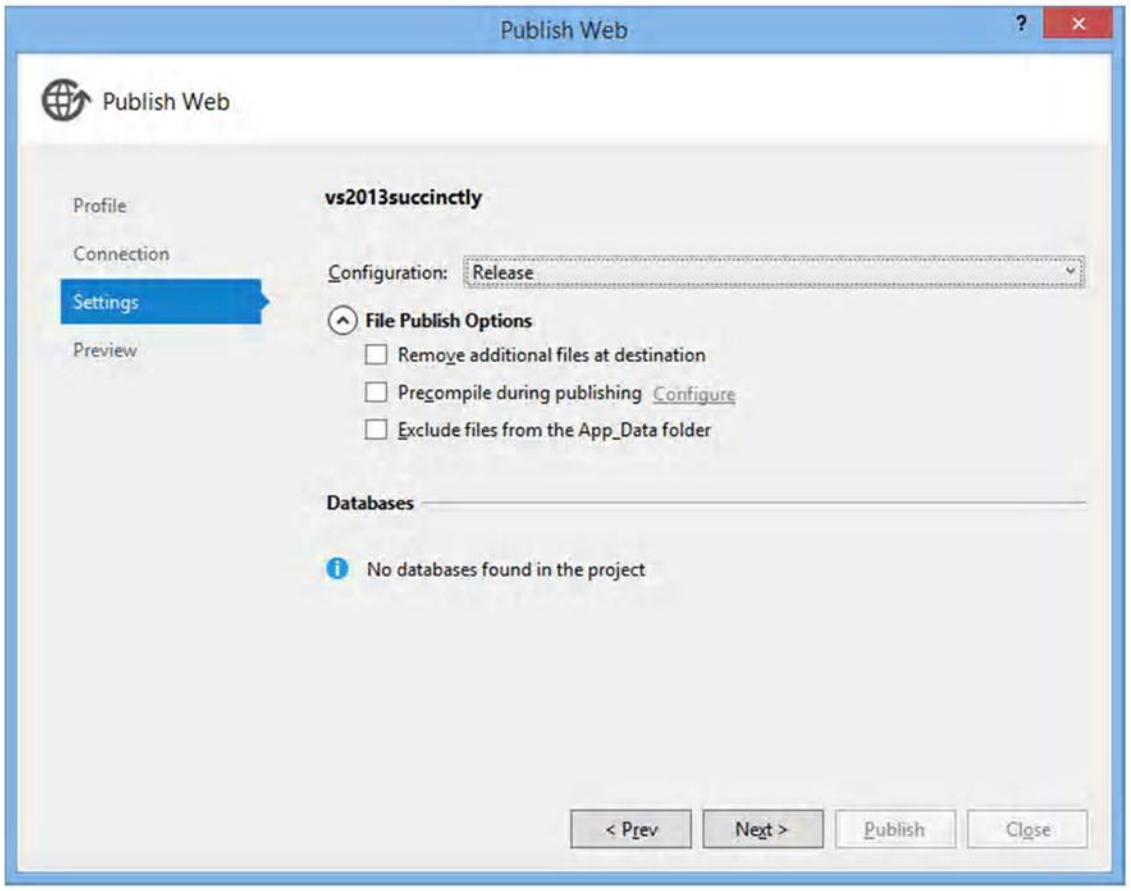


Figure 58: Publish Settings for the Application

In this form of the Publish Web dialog you can change the configuration between **Release** (default) and **Debug**, or decide if you want to remove additional files at destination, precompile managed code during publishing, or exclude files from the App_Data folder. If you are also using a database, here you will be able to set a default connection string. When you click **Next**, you access the Preview form where you can optionally view the full list of files that will be published to the websites, by clicking **Start Preview**. The result of pressing this button is shown in Figure 59.

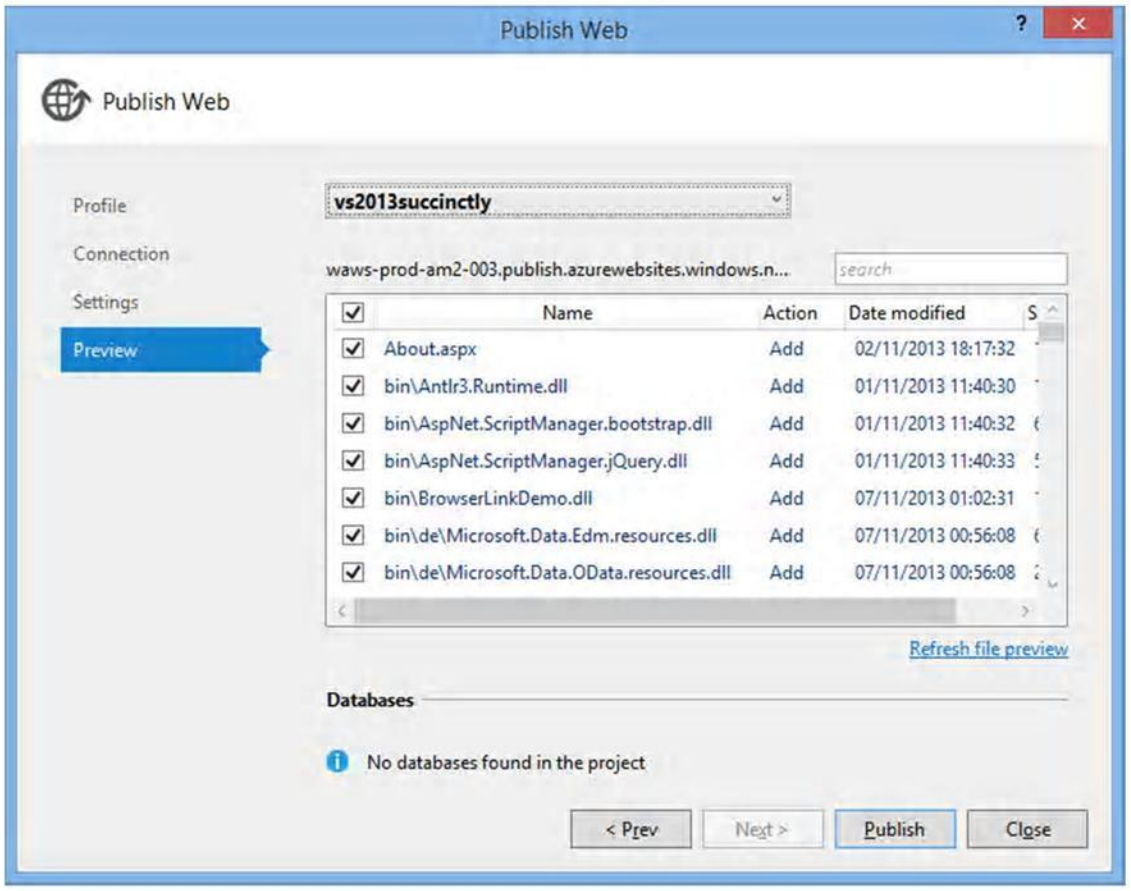


Figure 59: The List of Files that Will Be Published to the Website

You can finally click **Publish**. The progress of publishing will be shown in the Output window and the speed may vary depending on your Internet connection. When the application has been published and is online, you will see a message in the Output window and you will be able to start the application by using the appropriate web address. Figure 60 shows the sample application running on the website created previously.

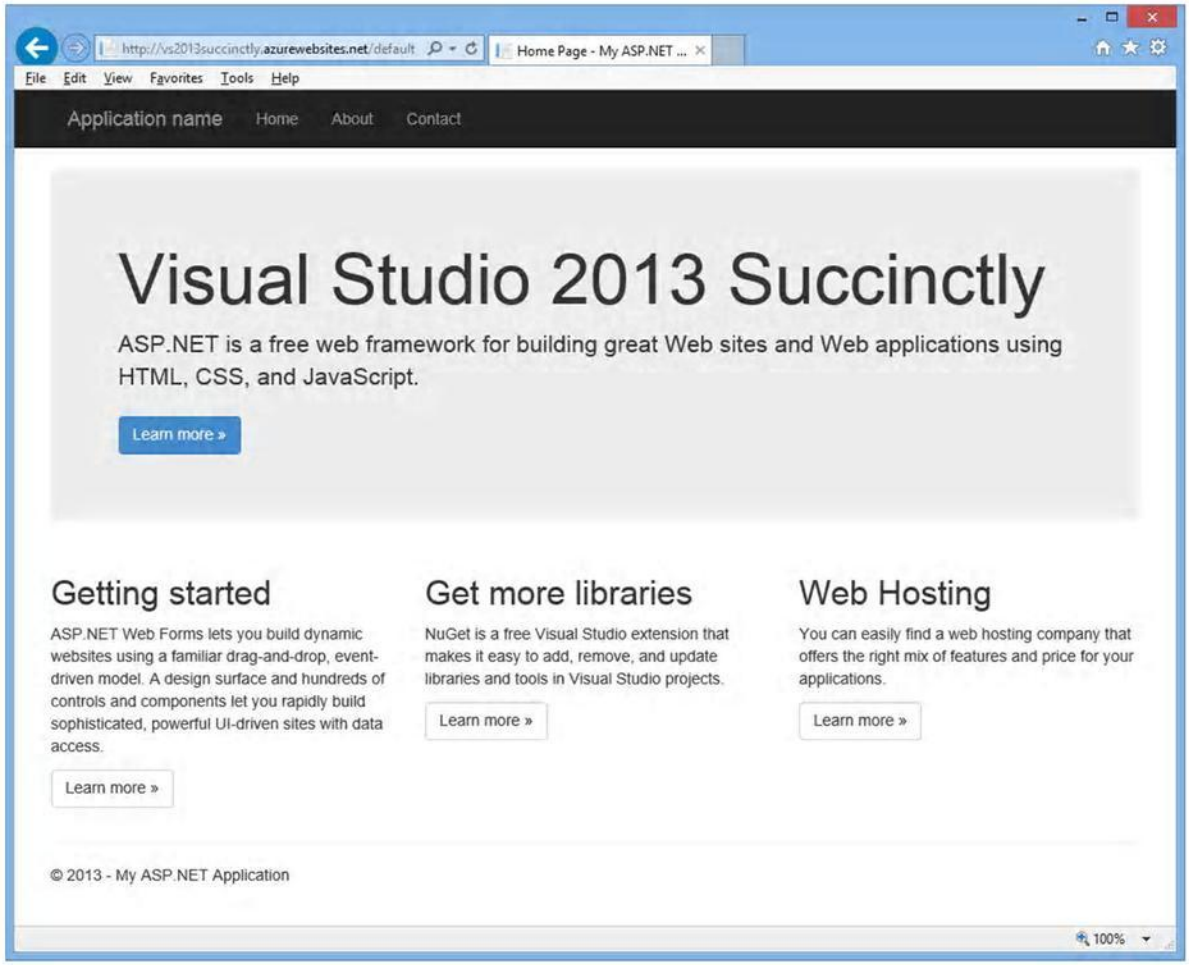


Figure 60: The Sample Application Running on the Website

As you can see, with a very few steps you have been able to publish an ASP.NET application to Windows Azure, without the need of having an in-house server and hosting infrastructure.

Note: If you create websites for testing purposes like in the current scenario, remember to delete the site when finished, even if you have the spending limit enabled. Websites consume resources even if they are offline, so the only way to ensure they do not consume any unneeded resources to delete them.

Integration with Azure Storage

Note: For the rest of Azure services described in this chapter, we focus on the new tools in the IDE. If you need guidance on how to access resources in the Azure Storage from your applications, make sure you read the appropriate documentation.

The Windows Azure Storage is the place where you can store data and files for your cloud-based applications. For instance, if your application needs to show some pictures or allows downloading files, these will be first saved onto the Storage and then accessed via their web address (HTTP). The Windows Azure Storage provides the following services:

- Blob Storage: here you can upload unstructured binary data, such as files.

- Queue Storage: this is used to save and retrieve messages for workflows and communications.
- Table Storage: here you can store non-relational and unschematized data with support for queries.

The Blob Storage can also recognize VHD files and allows mounting these files as virtual hard disks on the cloud. Visual Studio 2013 finally provides integration with the Windows Azure Storage from within the IDE. This is a tremendous benefit and a significant step forward, because before Visual Studio 2013, the only way to upload to or manage information on the Storage was by using 3rd party client applications. Now you can definitely use Visual Studio to perform operations such as uploading files to the Blob Storage. Any Windows Azure subscription supports multiple storages. For each storage, you need to create a storage account. Since any transaction (download, upload, login) has a cost, Microsoft offers the Development Account, which consists of the Windows Azure Emulator and other components that allow simulating a production environment on your development machine. This way, you can freely test transactions locally, and move to the cloud only when you are ready to go to production. Remember to check the MSDN documentation for the storage programmability and for understanding how to access information on the storage from your applications.

Creating a storage account

In Server Explorer, expand the **Storage** node under Windows Azure. The local development storage will be shown by default. If it's not already running, Visual Studio will start the Windows Azure Storage Emulator. The development storage works exactly like an online storage; however, it is a good idea to see how to work with an online account. Unfortunately, you cannot create new storage accounts from Visual Studio; instead, you can connect to existing accounts. So, open the Windows Azure Management Portal in your favorite web browser. Once logged in, click **Storage** in the dashboard on the left. Figure 61 shows how the management portal appears at this point, with no accounts available yet.

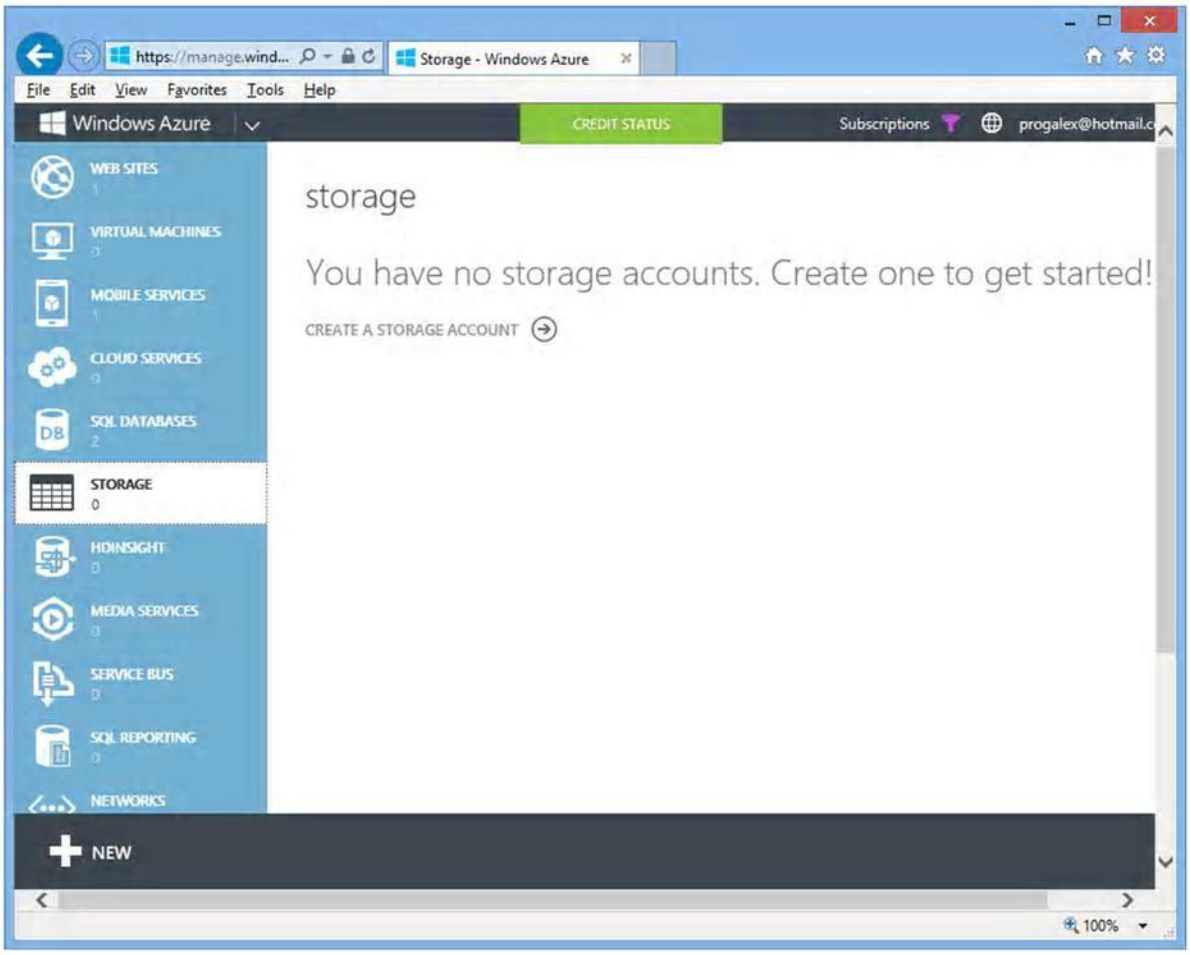


Figure 61: The Windows Azure subscription has no storages at the moment.

Click **CREATE A STORAGE ACCOUNT**. The Management Portal will open a new page where you can easily create a new storage account by supplying the account name and the location. Figure 62 shows how the management portal appears at this point.

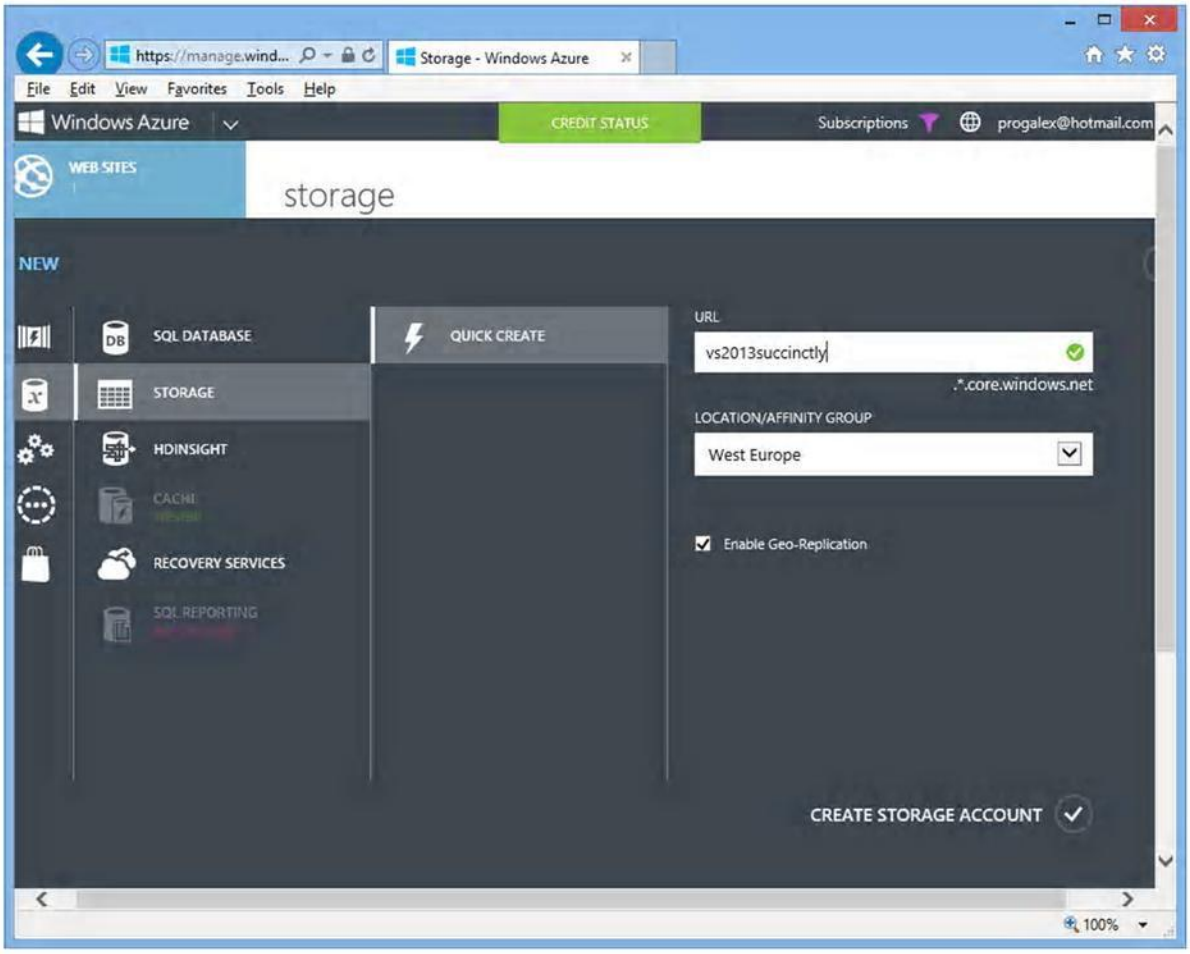


Figure 62: Creating a New Storage Account

Always remember to select the nearest location to where you live. As you might know, the account name becomes the prefix of the URL for services exposed by the storage. Table 2 shows URLs for each service (where storagename is the name you entered when creating the storage account).

Table 2: Members and settings for the Add Controller dialog

Storage	URL
Blob Storage	http://storagename.blob.core.windows.net
Table Storage	http://storagename.table.core.windows.net
Queue Storage	http://storagename.queue.core.windows.net

Addresses shown in Table 2 are very important, since they are the way you access each storage. Click **CREATE STORAGE ACCOUNT**. After about one minute, you will be able to see the storage account online in the Management Portal. Now close the management portal and go back to Visual Studio 2013. In Server Explorer, right-click the **Storage** node and then click **Refresh**. You will now see the newly created storage account (see Figure 63).

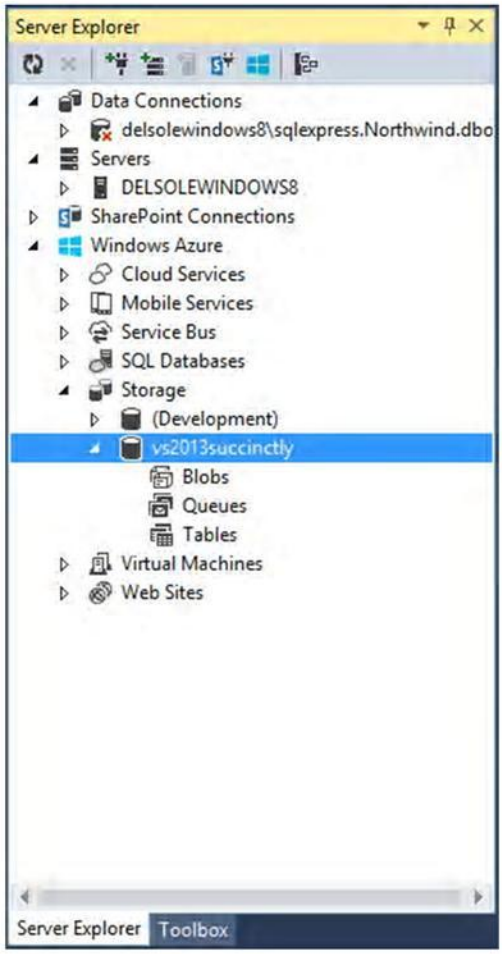


Figure 63: The new Storage Account is visible inside Server Explorer.

Now you will see how to interact with the storage.

Managing the Blob Storage

Say you want to upload an image file to the Blob Storage. Blobs are organized within folders called containers, so you first need to create a container. Follow these steps:

1. Click the Storage Account of your interest (in this case, vs2013succinctly) and expand it in order to make the Blobs node visible.
2. Select **Create Blob Container**.
3. In the Create Blob Container dialog (see Figure 64), specify the name of your new container all lower case. For example, enter **pictures**.
4. The new container will be visible in Server Explorer. Right-click it and select **View Blob Container**. You will see a new window called Container whose purpose is showing and

filtering the list of blobs in a container. It also allows uploading and/or deleting blobs (see Figure 65).

5. Click the **Upload Blob** button (the one with the black arrow). In the file selection dialog, select an image file (possibly of type Jpeg, in order to save space and time due to the lower size of this format). Visual Studio will start uploading your file to the storage, showing the progress of the operation. As you can see from Figure 66, the window shows metadata information for the file and, most importantly, the URL to access it.

Figure 64: Creating a New Blob Container

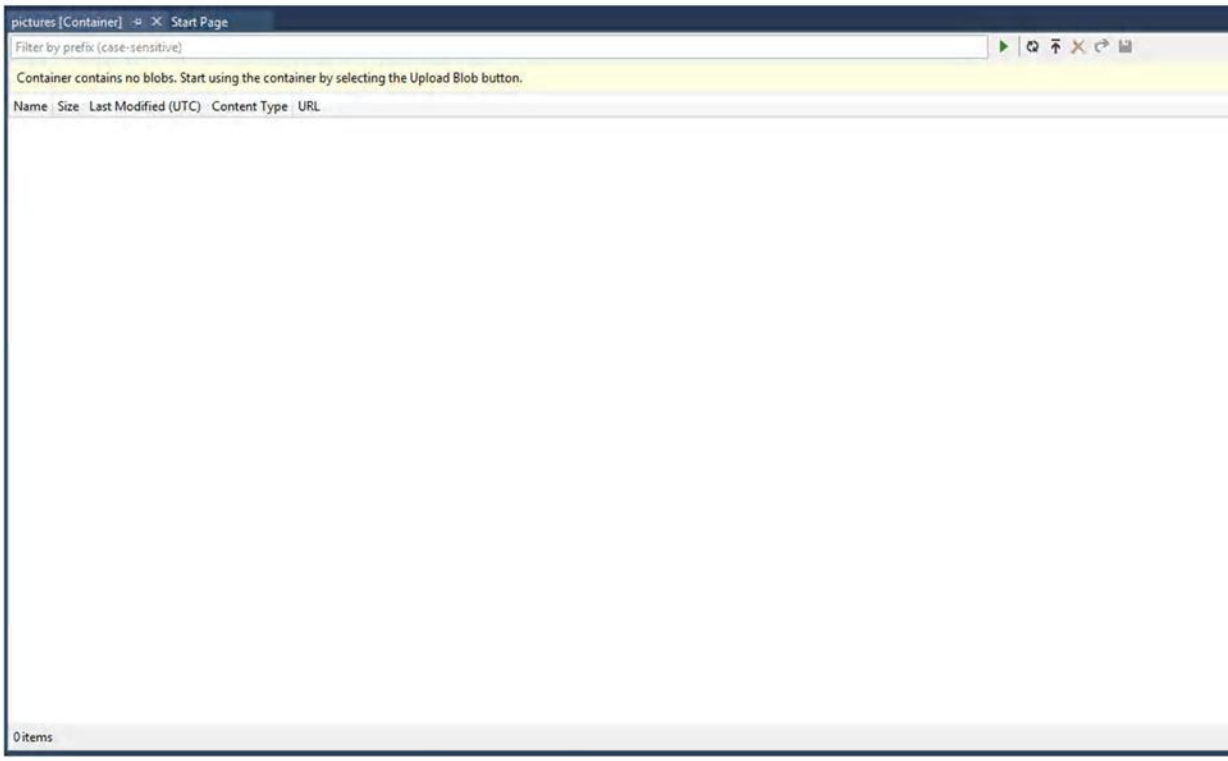


Figure 65: The list of blobs for the current container is empty.

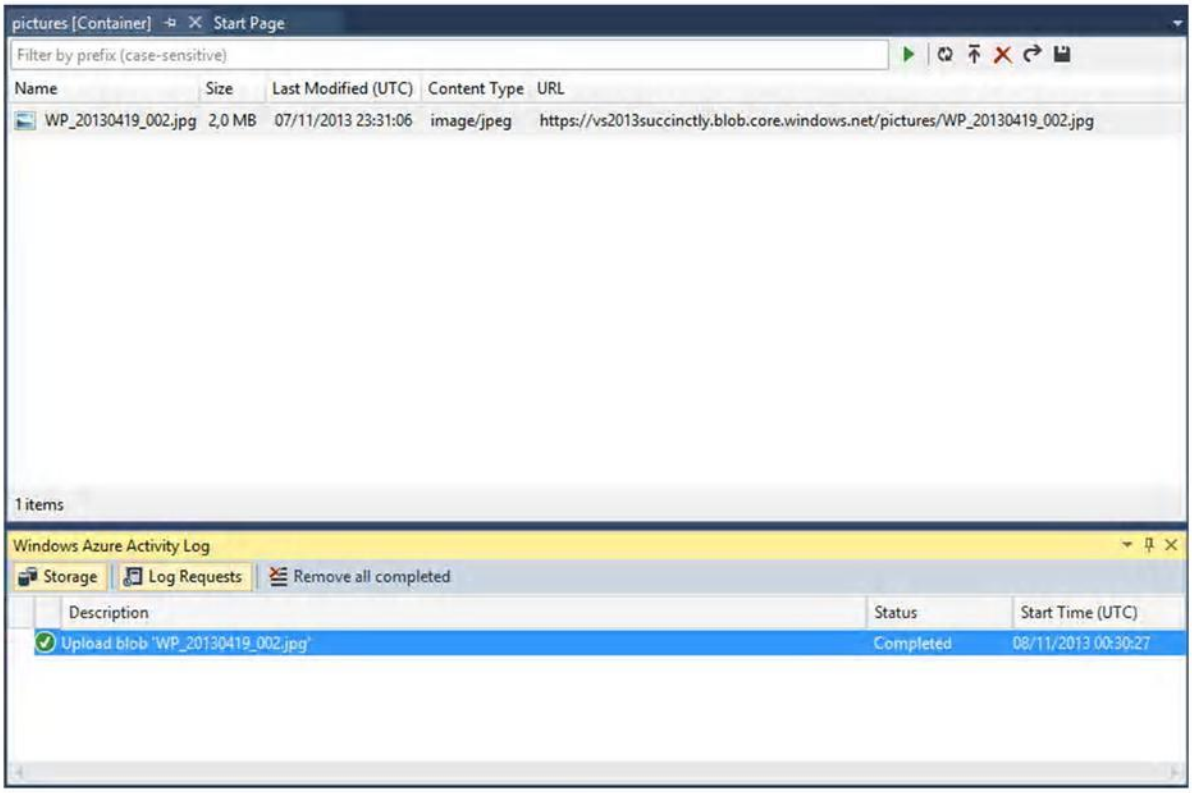


Figure 66: The blob has been uploaded and you can see its metadata and URL.

If you right-click the blob, you will get a popup menu showing a list of interesting commands. You can download the blob to disk (Save As), open the blob (Open), delete the blob (Delete), view more detailed properties in the Properties window (Properties), and copy the blob's URL to the clipboard (Copy URL). In order to access the blob in code, you will need to supply a connection string and credentials for your Windows Azure subscription. Writing code to accomplish this is beyond the scope of this chapter, so read this page in the Windows Azure documentation for this purpose. What you really need to understand is how you can upload the blob (and how you can manage blobs and containers) with the new tooling in Visual Studio 2013, without the need of 3rd party client applications.

Create and query tables

The Windows Azure storage supports creating tables to store nonrelational data. Right-click **Tables** in your storage account and select **Create Table**. In the **Create Table** dialog (see Figure 67), enter the name of your table, for example **mydata**.

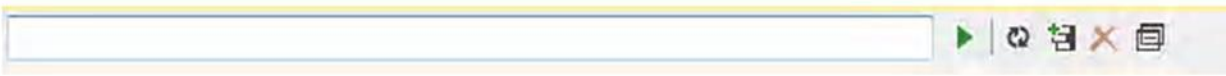


Figure 67: Creating a New Table

When the new table is created, double-click it: you will see a window similar to the one you saw for containers. The window's toolbar (see Figure 68) has a button called Add Entity, represented by three drawers and a green addition symbol.

Figure 68: The Table's Designer Toolbar

Click **Add Entity** to enter new data. Figure 69 shows how you can enter data for the new entity.

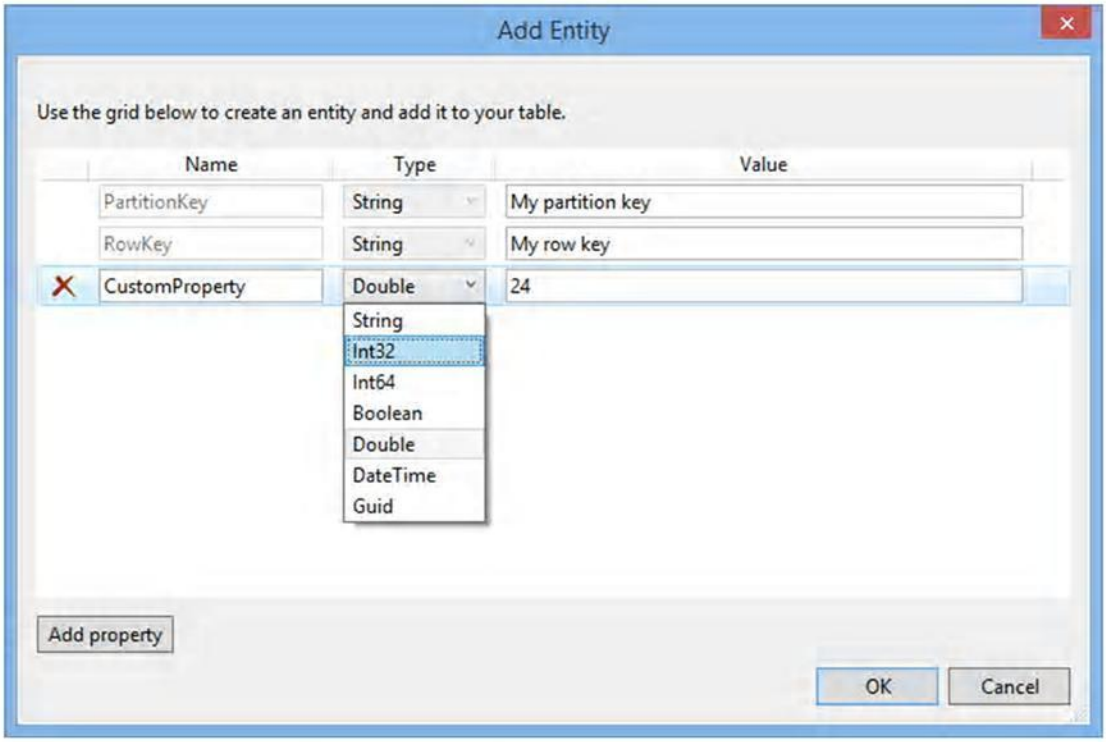


Figure 69: Entering Data for the New Entity and Property Customization

Every entity has some predefined properties, such as PartitionKey and RowKey, both of type String, that can be assigned with your values. Also, you can add custom properties of different types (see Figure 69). Click **OK** when ready. The new entity will be now visible inside the window, as shown in Figure 70.

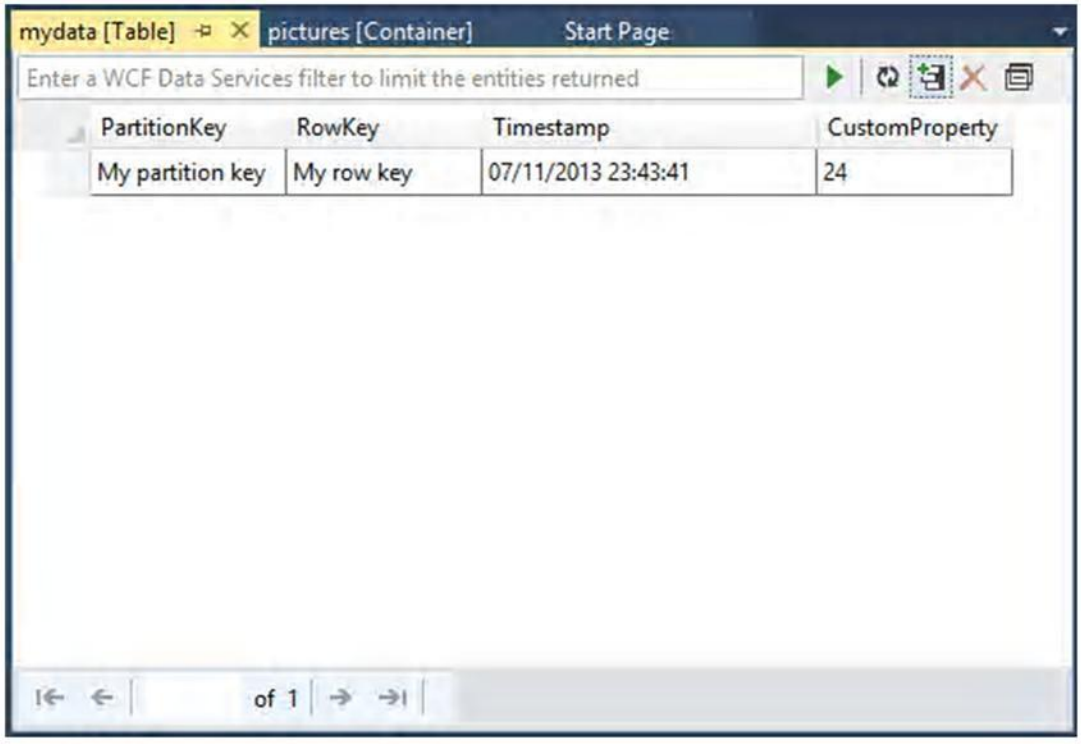


Figure 70: Listing Tables in your Storage Account

The toolbar has another button called Query Builder (the one on the right represented by two overlaid squares as shown in Figure 68), which allows executing queries against tables. You can specify one or more filters against different properties, as shown in Figure 71.

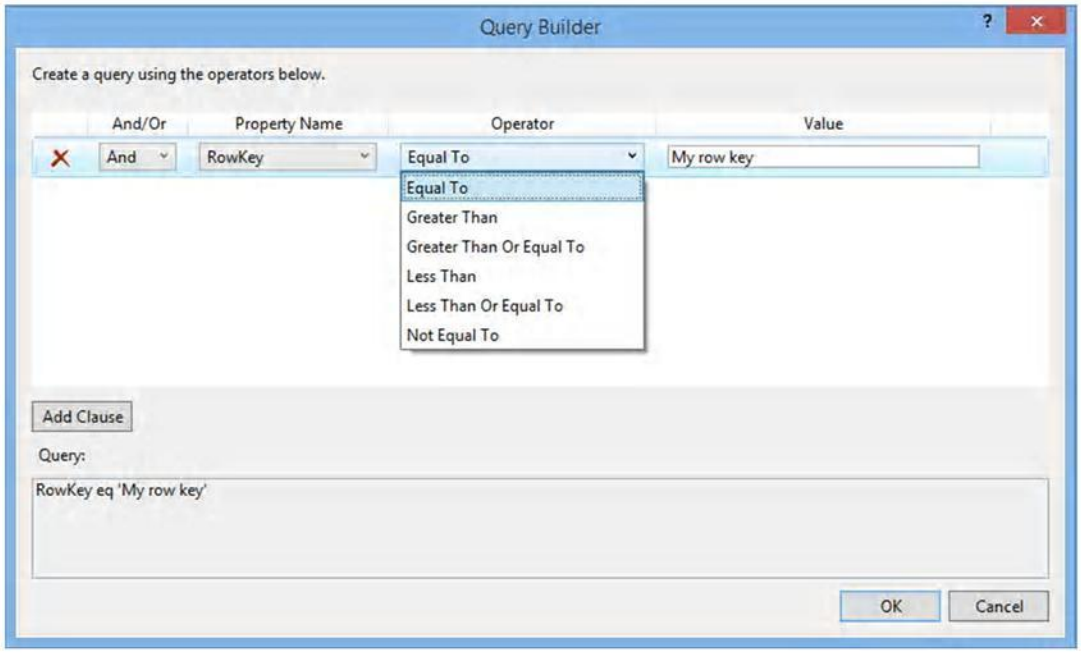


Figure 71: Defining Queries with the Query Builder

Notice that custom properties are not available in the Property Name dropdown. However, you can type queries and add filters by custom properties in the textbox inside the designer's toolbar (see Figure 68). When you have defined all the required filters, click **OK**. You will see the query syntax in the text box; if you click **Execute**, the query will be executed and only entities matching the specified criteria will be shown.

Create message queues

Visual Studio 2013 supports creating queues in the storage. With queues, cloud-based applications can easily share messages. To create a queue, in Server Explorer expand **Storage**, then expand the storage account of your choice (you can use **Development**), then right-click **Queues**. In the **Create Queue** dialog, enter the name for the new queue, again lower case (see Figure 72).

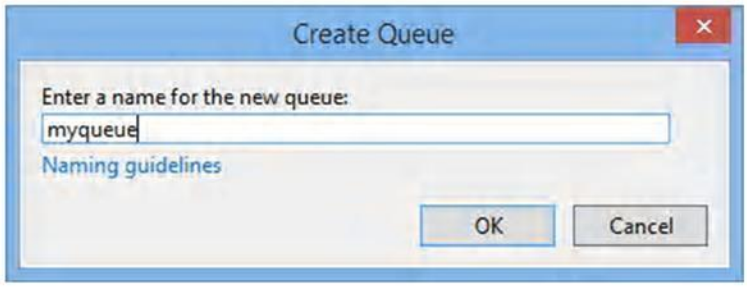


Figure 72: Creating a New Queue

When the new queue is visible in Server Explorer, double-click it to open the Queue window. Here is the place where you add and manage messages. To add one, click **Add Message** on the toolbar (the button with the icon of a letter and the green + symbol). In the appearing **Add Message** dialog, enter a text message and define when the message will expire (see Figure 73), and then click **OK**.

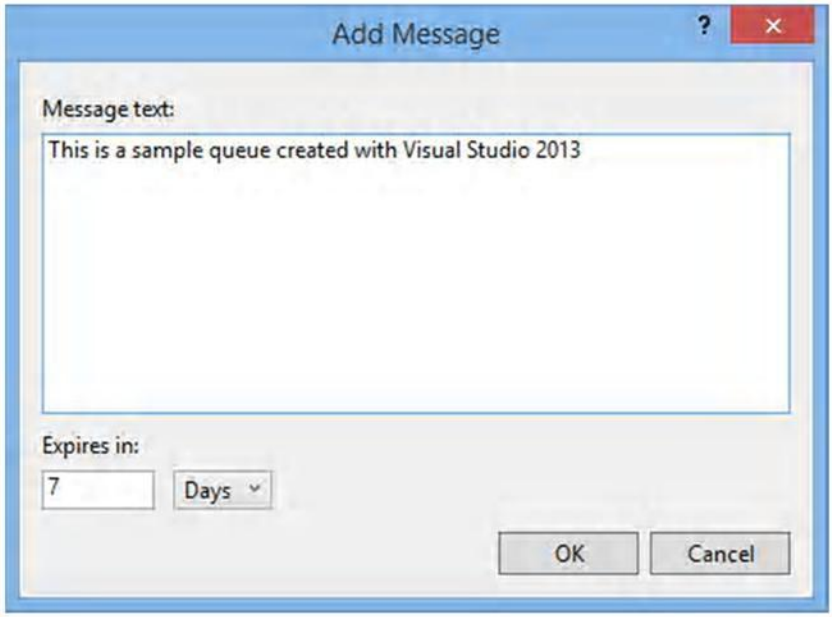


Figure 73: Creating a New Message

The new message will be added to the queue and will be visible in the Queue window, as you can see in Figure 74.

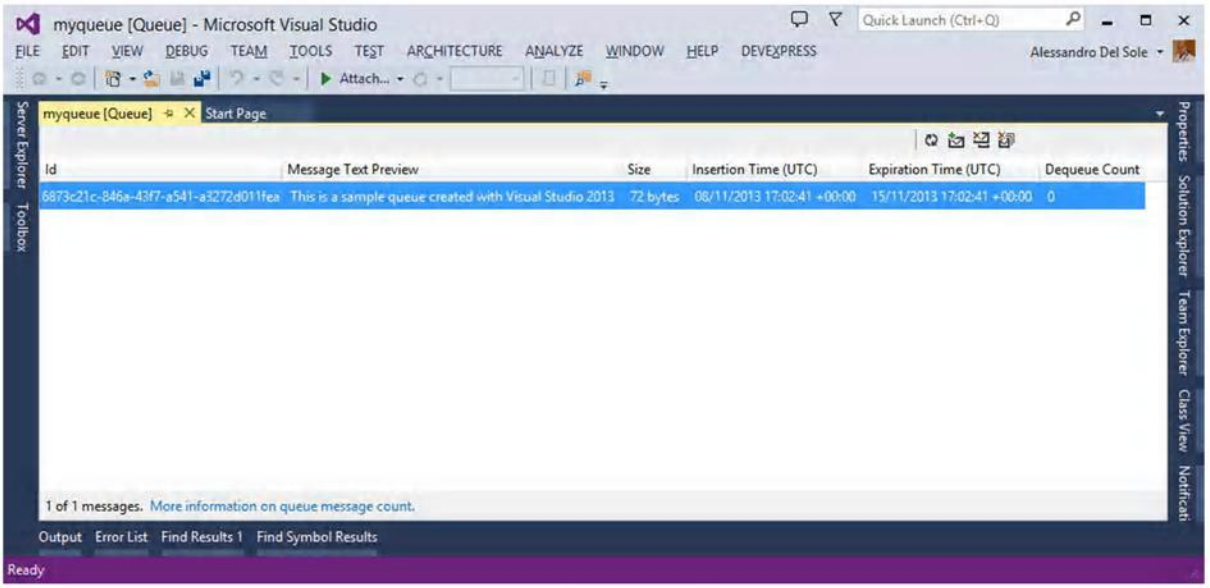


Figure 74: Creating a New Queue

Interesting information is shown, such as the insertion time, expiration time, and ID. By using the appropriate buttons on the toolbar, you can also de-queue or remove messages. If you are not familiar with using queues in your applications, check out the Windows Azure documentation about programming with queues. As for other storage types, Visual Studio 2013 makes it very easy to create and manage queues avoiding the need of using 3rd party applications.

Chapter summary

With the idea of offering the most productive environment ever, Visual Studio 2013 introduces many new tools for web development, including building applications for Windows Azure. On the ASP.NET side, Visual Studio 2013 introduces the One ASP.NET experience, which provides a unified approach to web development making it easy to use libraries from different frameworks into one application. This includes the use of scaffolding (formerly available only for MVC applications) in Web Forms applications. With scaffolding you can easily build data-centric applications and take advantage of auto-generated code for data access and data-bound, ready-to-use pages. Also, Visual Studio 2013 makes it easier to test applications in different web browsers with the new Browser Link feature, which allows refreshing all browsers with one click. For Windows Azure, Visual Studio 2013 enhances the Server Explorer tool window, which now offers all you need to work with most services exposed by the platform from within the IDE.

Chapter 6 New and Enhanced Tools for Debugging

As a developer, you probably spend a lot of time testing and debugging your code. Visual Studio 2013 introduces new debugging tools and updates some existing ones, continuing in its purpose of offering the most productive environment ever.

64-bit Edit and Continue

Visual Studio 2013 finally introduces Edit and Continue for 64-bit applications. As you know, with Edit and Continue, you can break the application's execution, edit your code, and then restart. So far, this has been available only for 32-bit applications. It is very easy to demonstrate how this feature works. Consider a very simple Console application, whose goal is retrieving the list of running processes and displaying the name of the first process in the list; the code is the following.

Visual C#

```
    class Program
    {
        static void Main(string[] args)
        {
            var runningProcesses = System.Diagnostics.
                                   Process.GetProcesses();
            Console.WriteLine(runningProcesses.First().ProcessName);
            Console.ReadLine();
        }
    }
```

Visual Basic

```
Module Module1

    Sub Main()
        'Add a breakpoint here and make your edits at 64-bits!
        Dim runningProcesses = System.Diagnostics.Process.GetProcesses()
        Console.WriteLine(runningProcesses.First().ProcessName)
        Console.ReadLine()
    End Sub

End Module
```

Before running the application, open the project's properties, select the **Build** tab, and change the platform target to **x64**, as shown in Figure 75.

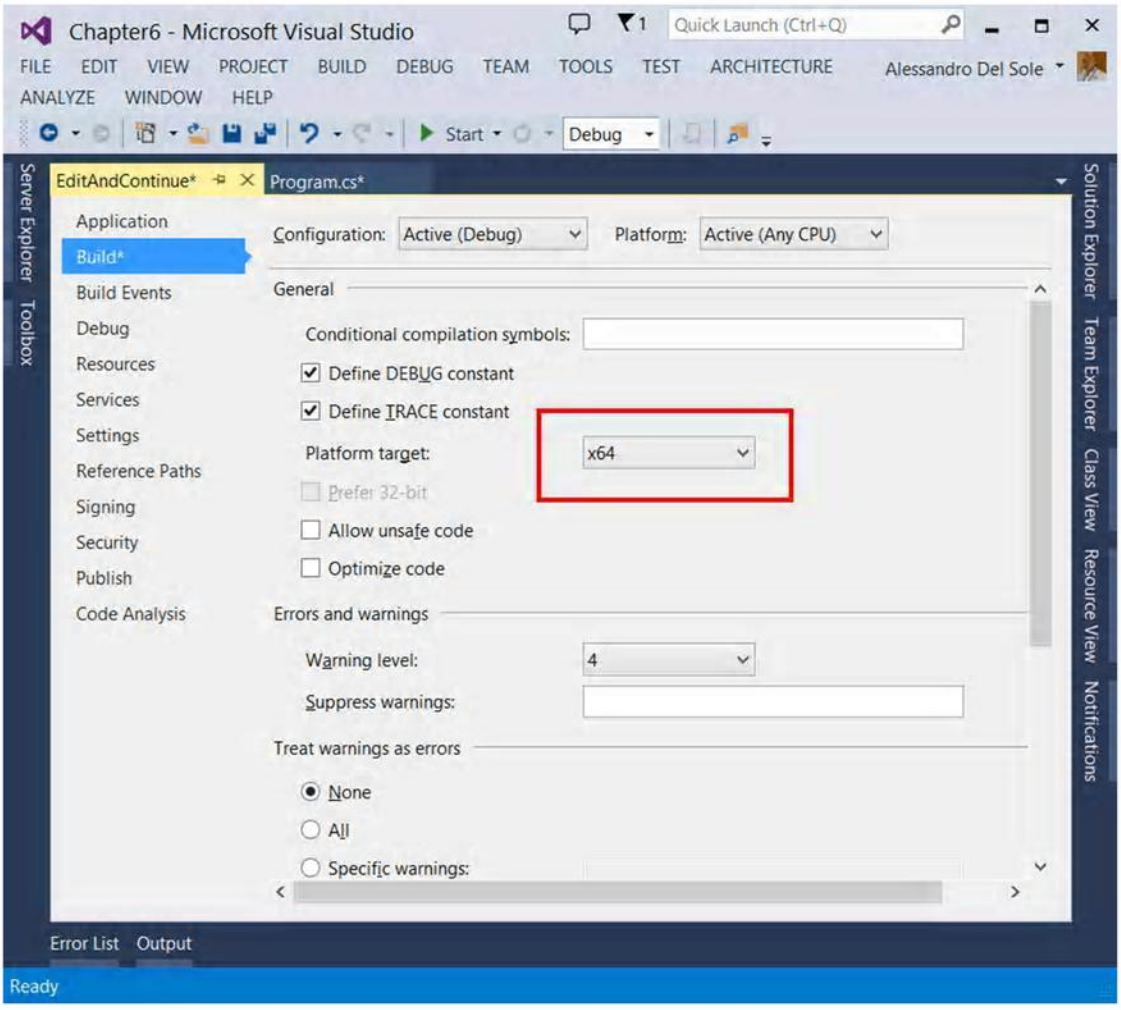


Figure 75: Selecting 64-bit Target Architectures

Now go back to the code, and place a breakpoint on the line containing the declaration of the **`runningProcesses`** variable by pressing **F9**. Finally, press **F5** to run the application. When the debugger encounters the breakpoint, the code editor is shown. You can simply rename the **`runningProcesses`** variable into **`currentProcesses`** (see Figure 76); this is enough to demonstrate how Edit and Continue is now working. Before Visual Studio 2013, if you tried to edit your code, at this point you would receive an error saying that Edit and Continue is only supported in 32-bit applications.

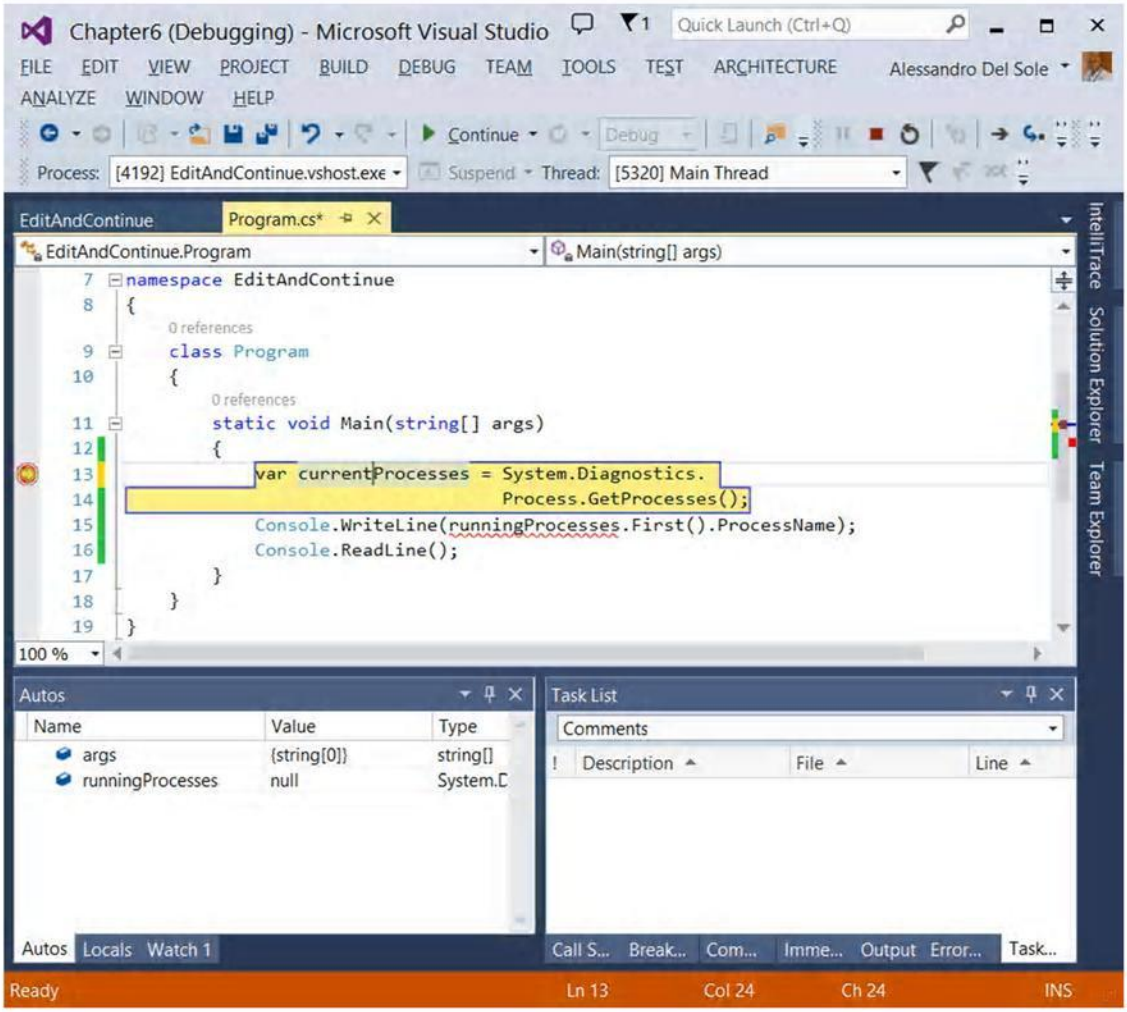


Figure 76: You can edit your code before resuming the execution.

Asynchronous debugging

Visual Studio 2012 and the .NET Framework 4.5 introduced a new pattern for coding asynchronous operations, known as the Async/Await pattern based on the new **async** and **await** keywords in the managed languages. The goal of this pattern is making the UI thread always responsive; the compiler can generate appropriate instances of the `Task` class and execute an operation asynchronously even in the same thread. You will see shortly a code example that will make your understanding easier, however there is very much more to say about Async/Await, so you are strongly encouraged to read the MSDN documentation if you've never used it. I

f you are already familiar with this pattern, you know that it is pretty difficult to get information about the progress and the state of an asynchronous operation at debugging time. For this reason, Visual Studio 2013 introduces a new tool window called Tasks. The purpose of this new tool window is to show the list of running tasks and provide information on active and pending tasks, time of execution, and executing code. The Tasks window has been very much publicized as a new addition to Windows Store apps development, but it is actually available to a number of other technologies, such as WPF. This is the reason why this feature is discussed in this chapter rather than in the next one about Windows 8.1.

Create a sample project

To understand how this feature works, let's create a new WPF Application project called *AsyncDebugging*. This application will create a new text file when the user clicks a button. The XAML code for the user interface is very simple, as represented in the following listing.

```
<Window x:Class="AsyncDebugging.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Width="100" Height="30" Name="FileButton" Content="Create
file" Click="FileButton_Click"/>
    </Grid>
</Window>
```

The code-behind file for the main window will contain the following code (see comments inside).

Visual C#

```
using System.IO;

        //Asynchronous method that passes some variables to
        //the other async method that will write the file
        //You wait for the async operation to be completed by using
        //the await operator. This method cannot be awaited itself
        //because it returns void.
        private async void WriteFile()
        {
            string filePath = @"C:\temp\testFile.txt";
            string text = "Visual Studio 2013 Succinctly\r\n";

            await WriteTextAsync(filePath, text);
        }


        //Asynchronous method that writes some text into a file
        //Marked with "async"
```

```
        private async Task WriteTextAsync(string filePath, string text)
        {
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            using (FileStream sourceStream = new FileStream(filePath,
                FileMode.Append, FileAccess.Write, FileShare.None,
                bufferSize: 4096, useAsync: true))
            {
                //new APIs since .NET 4.5 offer async methods to read
                //and write files
                //you use "await" to wait for the async operation to be
                //completed and to get the result
                await sourceStream.WriteAsync(encodedText, 0,
                      encodedText.Length);
            };
        }

        private void FileButton_Click(object sender, RoutedEventArgs e)
        {
            //Place a breakpoint here...
            WriteFile();
        }
```

Visual Basic

```
    Imports System.IO

    'Asynchronous method that passes some variables to
    'the other async method that will write the file
    'You wait for the async operation to be completed by using
    'the await operator. This method cannot be awaited itself
    'because it returns void.
    Private Async Sub WriteFile()
        Dim filePath As String = "C:\temp\testFile.txt"
        Dim text As String = "Visual Studio 2013 Succinctly"

        Await WriteTextAsync(filePath, text)
    End Sub

    'Requires Imports System.IO
    'Asynchronous method that writes some text into a file
    'Marked with "async"
    Private Async Function WriteTextAsync(filePath As String,
            text As String) As Task
        Dim encodedText As Byte() = Encoding.Unicode.GetBytes(text)

        Using sourceStream As New FileStream(filePath, FileMode.Append,
                                             FileAccess.Write,
```

```
                                                FileShare.None,
                                                bufferSize:=4096,
                                                useAsync:=True)
            'new APIs since .NET 4.5: async methods to read and write files
            'you use "await" to wait for the async operation
            'to be completed and to get the result
            Await sourceStream.WriteAsync(encodedText, 0,
                  encodedText.Length)
        End Using
    End Function

    Private Sub FileButton_Click(sender As Object, e As RoutedEventArgs)
        WriteFile()
    End Sub
```

In order to run the code without any errors, ensure you have a C:\Temp folder; if not, create one or edit the code to point to a different folder. If you start the application normally, after a few seconds you will see that the text file has been created correctly into the C:\Temp folder. If you already have used the Async/Await pattern in the past, you know that the debugging tools available until Visual Studio 2012 could not show the lifecycle of tasks; you could not know what task was active and which one was waiting. Let's see how Visual Studio 2013 changes things at this point.

Understanding the Tasks lifecycle with the Tasks window

Place a breakpoint on the **`WriteFile`** method invocation inside the button's click event handler (see the comment in the previous listing). Start the application and, when ready, click the button. When Visual Studio breaks the execution on the breakpoint, go to **Debug**, **Windows**, and select **Tasks**. The Tasks tool window will be opened and docked inside the IDE. Start debugging with Step Into by pressing **F11**. While asynchronous methods are invoked, the Tasks window shows their status, as demonstrated in Figure 77.

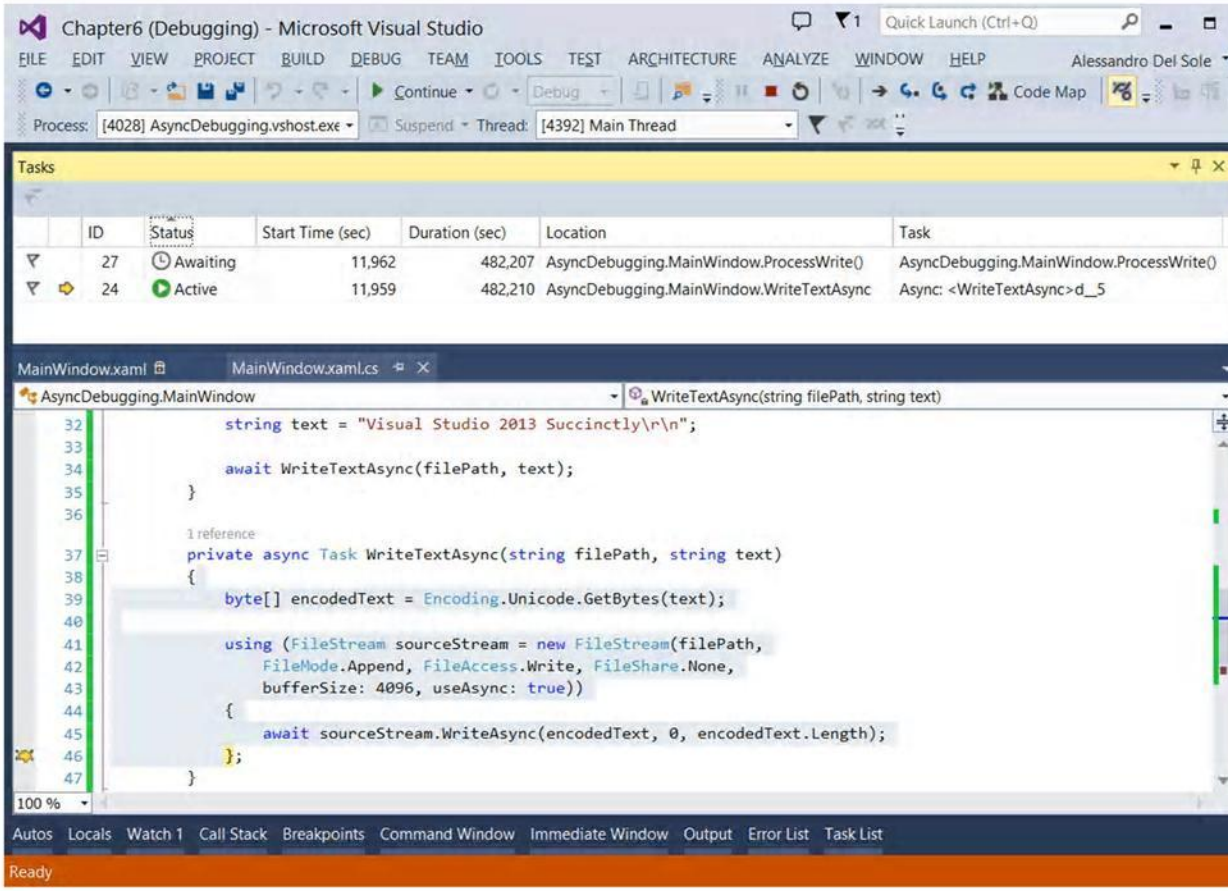


Figure 77: The Tasks window shows the status of asynchronous tasks.

By default, the Tasks window shows the following columns and related information:

- ID, which represents the task identifier.
- Status, which indicates whether the task is active or awaiting.
- Start Time (sec), which indicates the start time in seconds for the tasks.
- Location, which shows the name of the method where the task has been invoked.
- Task, which summarizes the operation in progress.

You can customize the Task window by adding or removing columns. If you right-click any column and then select **Columns**, a popup menu will show the full list of available columns; for instance, you might be interested in the Thread Assignment column to see what thread contains the selected task. The Tasks window is definitely useful when you need a better understanding of asynchronous operations' lifecycle, including when you need to analyze a task's performance. If the Tasks window does not display information as you step through lines of code using F11, and you are working with a desktop application, restart debugging and retry. This is a known issue. If you are working with a Windows Store app instead, you will not encounter this problem.

Performance and Diagnostics Hub

Analyzing performance and the behavior of an application is crucial. If your application is fast, fluid, and does not consume a lot of system resources (including battery for mobile apps), users will love it. Visual Studio has been offering analysis tools for many years, focusing on different areas such as memory usage, CPU usage, unit tests, and code analysis. With the big growth of mobile apps, Visual Studio has also been offering analysis tools specific to mobile platforms. In Visual Studio 2013, Microsoft has made another step forward, introducing a new unique place where you find such analysis tools. This place is called Performance and Diagnostics Hub. You can reach it by selecting **Debug**, **Performance** and **Diagnostics** or by pressing **ALT+F2**. Figure 78 shows how the Performance and Diagnostics Hub appears with a Windows Store app project.

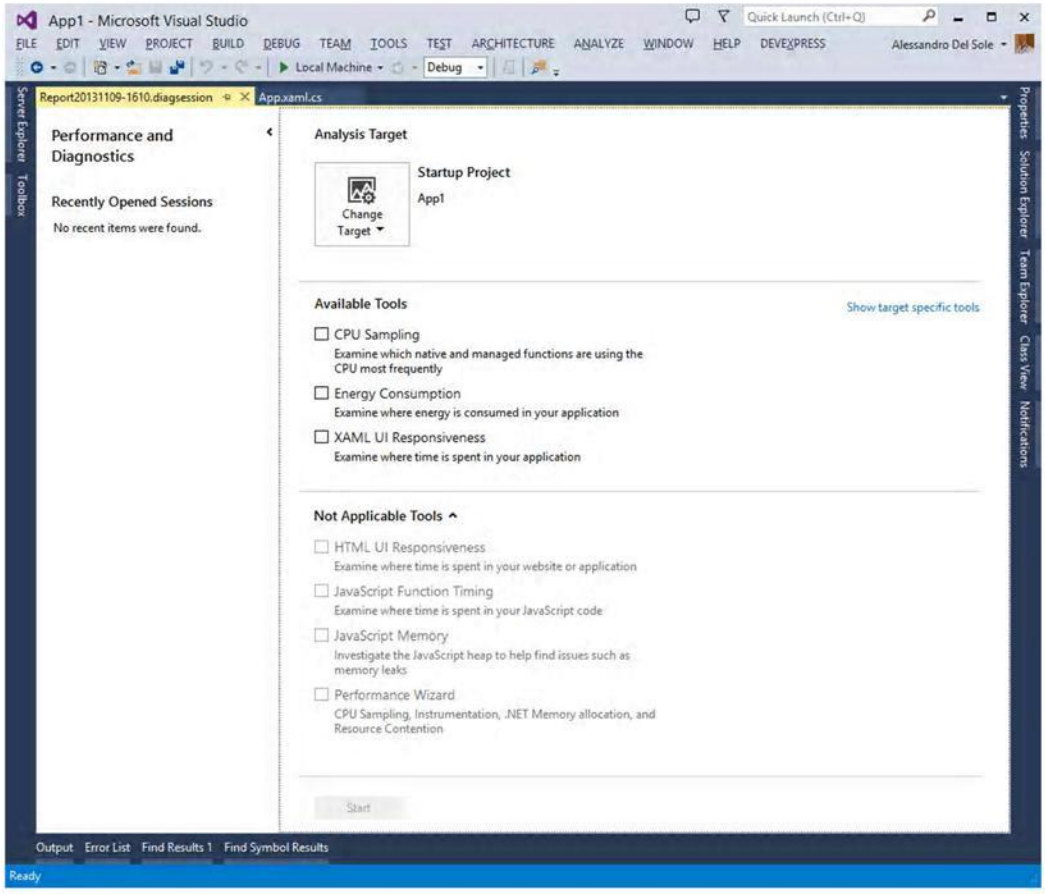


Figure 78: The Performance and Diagnostics Hub

Visual Studio 2013 will enable only target-specific tools. Figure 78 refers to a XAML Windows Store app, so all the other tools that target HTML Windows Store apps are disabled. For ASP.NET and desktop applications, only the CPU Sampling is available. Table 3 shows the list of available analysis tools per project type.

Table 3: Analysis Tools per Project Type

Analysis Tool	Purpose	Project Type(s)
Performance Wizard (includes CPU Sampling)	Analyze CPU usage, managed memory allocation, runtime diagnostics of the application state	All project types
Energy Consumption	Analyze potential battery usage through the Windows simulator	Windows Store apps
XAML UI Responsiveness	Analyze how time is spent in rendering layout	XAML Windows Store apps
HTML UI Responsiveness	Analyze how time is spent in rendering layout	HTML Windows Store apps
JavaScript Memory	Analyze the JavaScript heap to help find issues such as memory leaks	HTML Windows Store apps
JavaScript Function Timing	Analyze how time is spent in executing JavaScript code	HTML Windows Store apps

The CPU Sampling analysis tool invokes the Profiler that ships with Visual Studio, which you already know from previous versions. To start a diagnostics session you just select the tool you need and then click **Start** at the bottom of the page. When you close the application or break the diagnostic session manually, Visual Studio will generate a report based on the analysis type you selected. In the next chapter, when we discuss new features for Windows 8.1, you will get a more detailed demonstration of this tool. Remember that you can still access analysis tools via the Analyze menu as you did with previous versions of the IDE.

Code Map debugging

Note: Code Map is available only in Visual Studio 2013 Ultimate.

Another interesting addition to Visual Studio 2013 is Code Map. Actually, Code Map is available in Visual Studio 2012 with Update 1, but now the tool is integrated in the IDE. With Code Map, you can get an incremental visualization of your application and dependencies. In simpler words, you can get a visual representation of method calls, references, and fields while debugging, inside an interactive window where you can also add comments, flag an item for follow up, and export graphics to an image file.

To understand how Code Map works, let's consider the WPF sample application we created to demonstrate asynchronous debugging earlier in this chapter. Ensure a breakpoint is still inside the button's click event handler, then start the application with **F5**. Click the button in the application, then when the debugger encounters the breakpoint and breaks, click the **Code Map** on the toolbar (see Figure 79).

Figure 79: The Code Map Button

Visual Studio will start generating a map at this point. After a few seconds, you will see the method call in the Code Map, as represented in Figure 80.

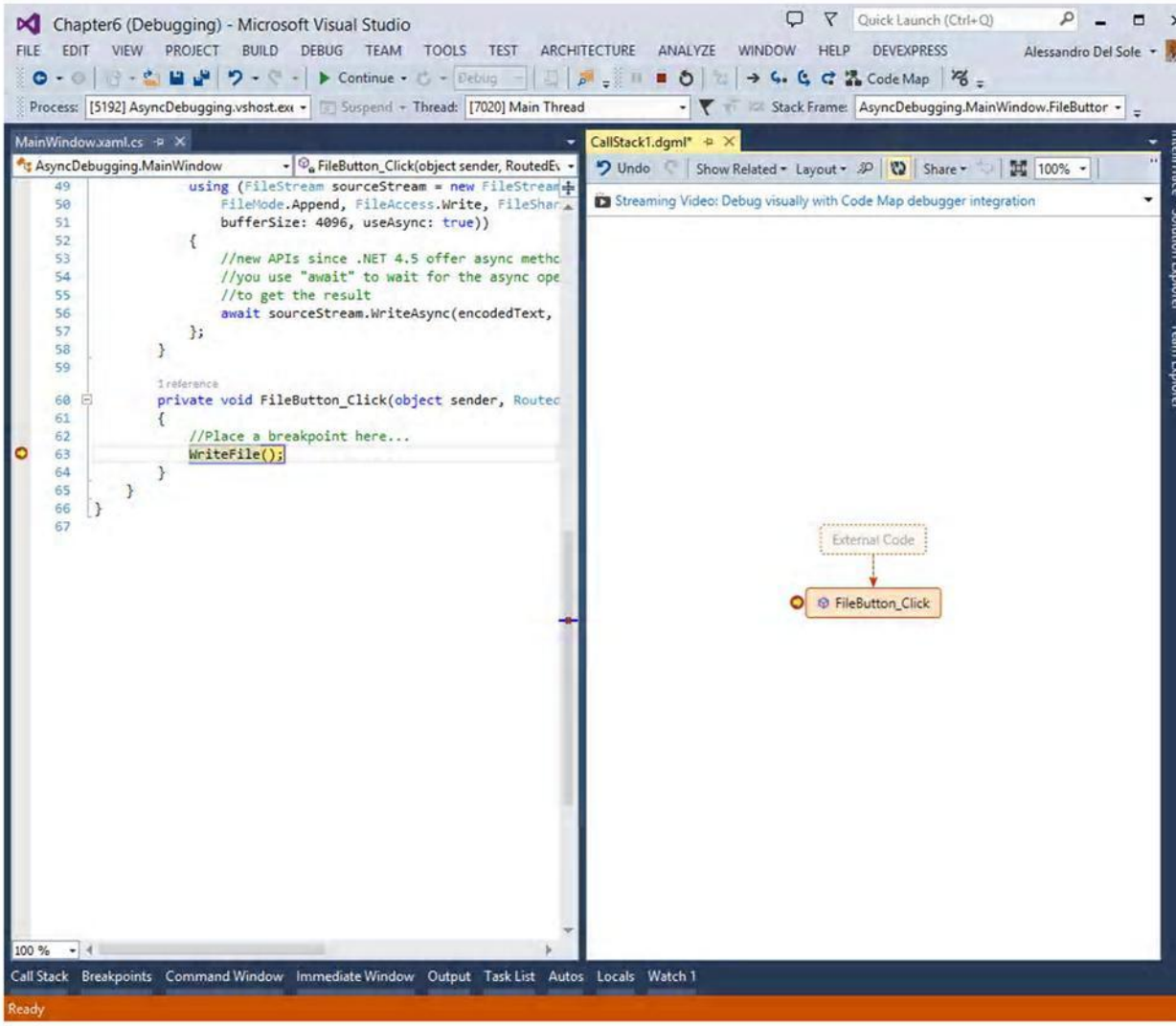


Figure 80: A New Code Map

Before continuing, you can play with the various buttons on the window's toolbar. For instance, if you check the Share button, you will see how you can easily export or email the diagram as an image file or as a portable XPS file. The Layout button offers an option to show the code map in different ways, whereas Show Related allows finding references to methods and types for the selected item in the map. Now press **F11** to execute the next line of code. The Code Map is immediately updated with the call to the **`WriteFile`** method, as shown in Figure 81.

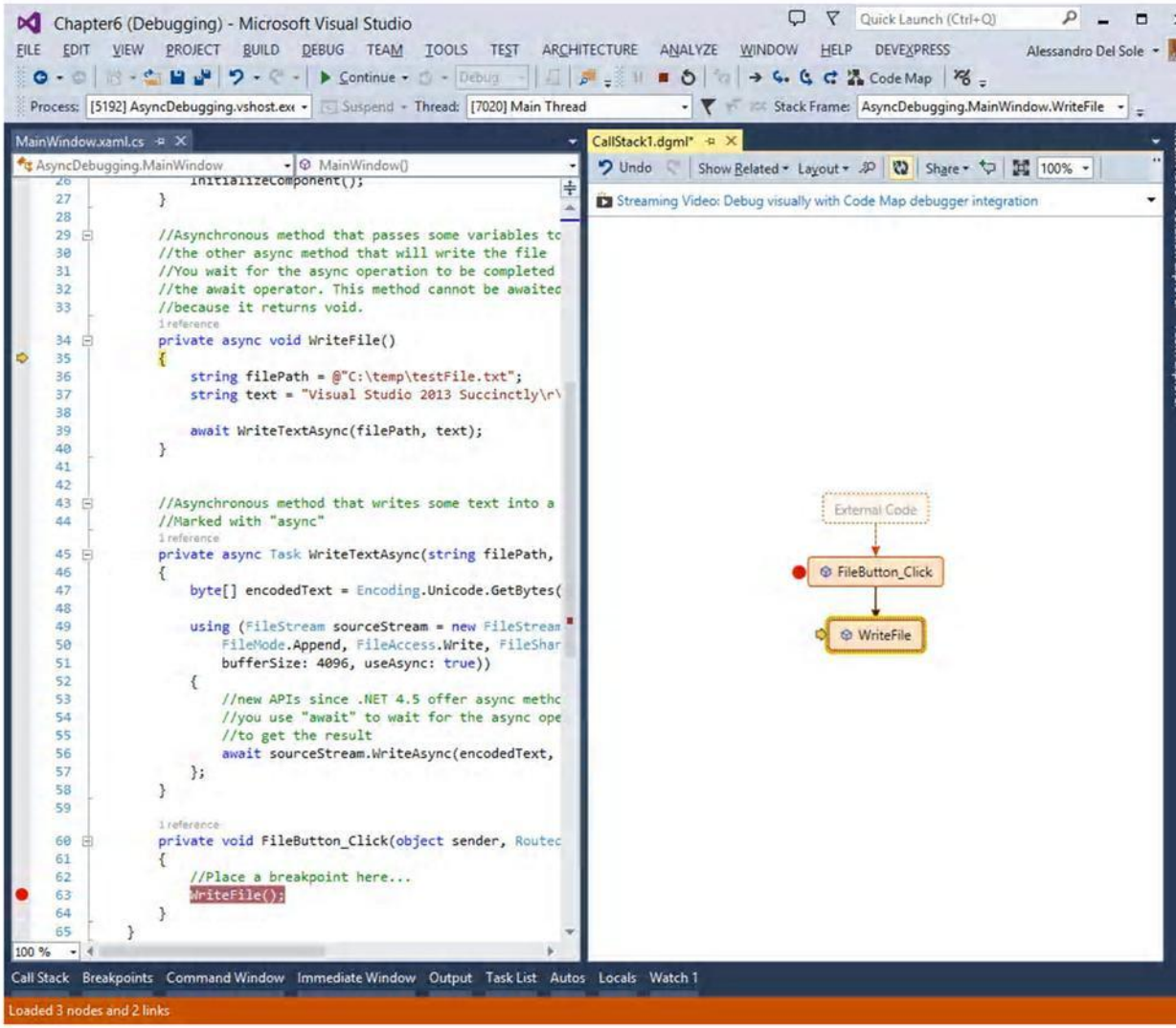


Figure 81: The code map is updated while debugging.

Objects are also represented on the Code Map. For example, while you are debugging the **`WriteFile`** method, right-click the **`text`** variable and then click **Show On Code Map**. The map will be updated (see Figure 82) with the referenced variable, shown inside its containing object.

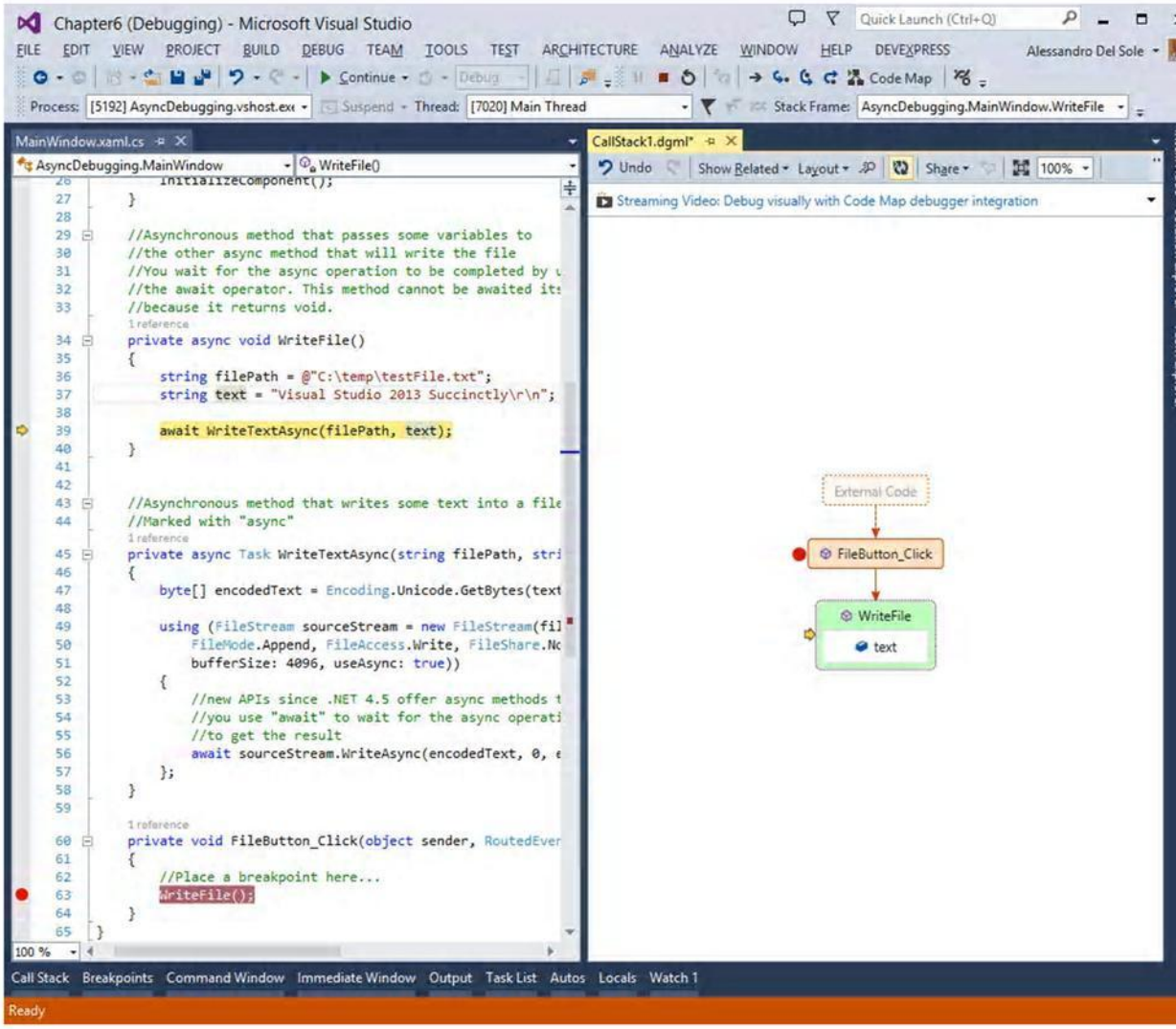


Figure 82: The code map is updated while debugging.

If you right-click a method in the map, you will be able to display a number of data points such as calls to other methods, fields the method references, and the containing type. For example, right-click the **`WriteFile`** method and then select **Show Methods This Call**. Visual Studio will show calls to other methods made by **`WriteFile`**, as shown in Figure 83.

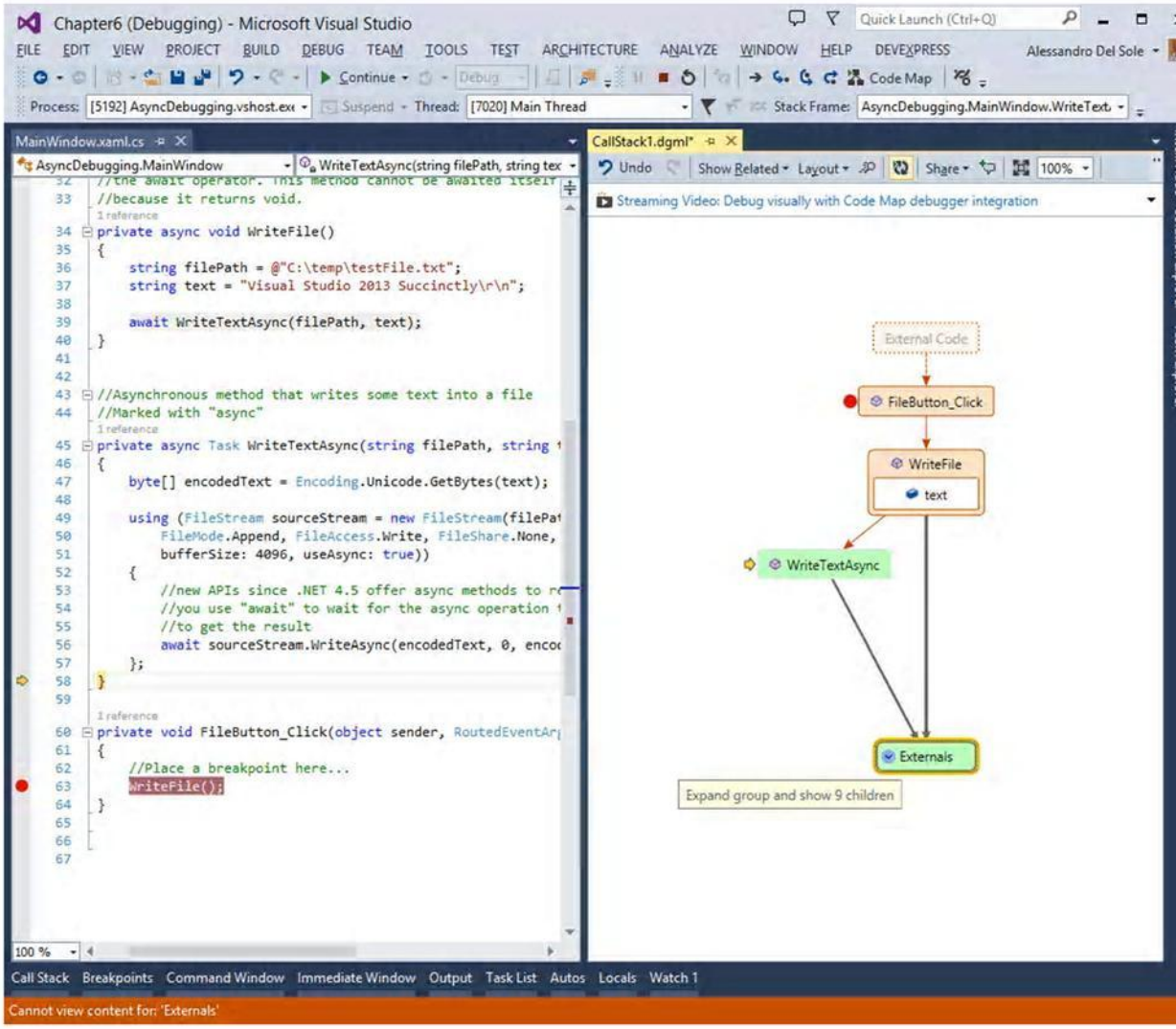


Figure 83: Showing method calls from the selected method.

The method calls **`WriteTextAsync`**, which invokes external code. Such an external code is how the runtime translates the Async/Await pattern into the backing .NET methods. This can be easily demonstrated by expanding the **Externals** node by clicking the expansion button inside. As a tooltip suggests, if you expand the Externals node you will be able to see nine children objects, as represented in Figure 84.

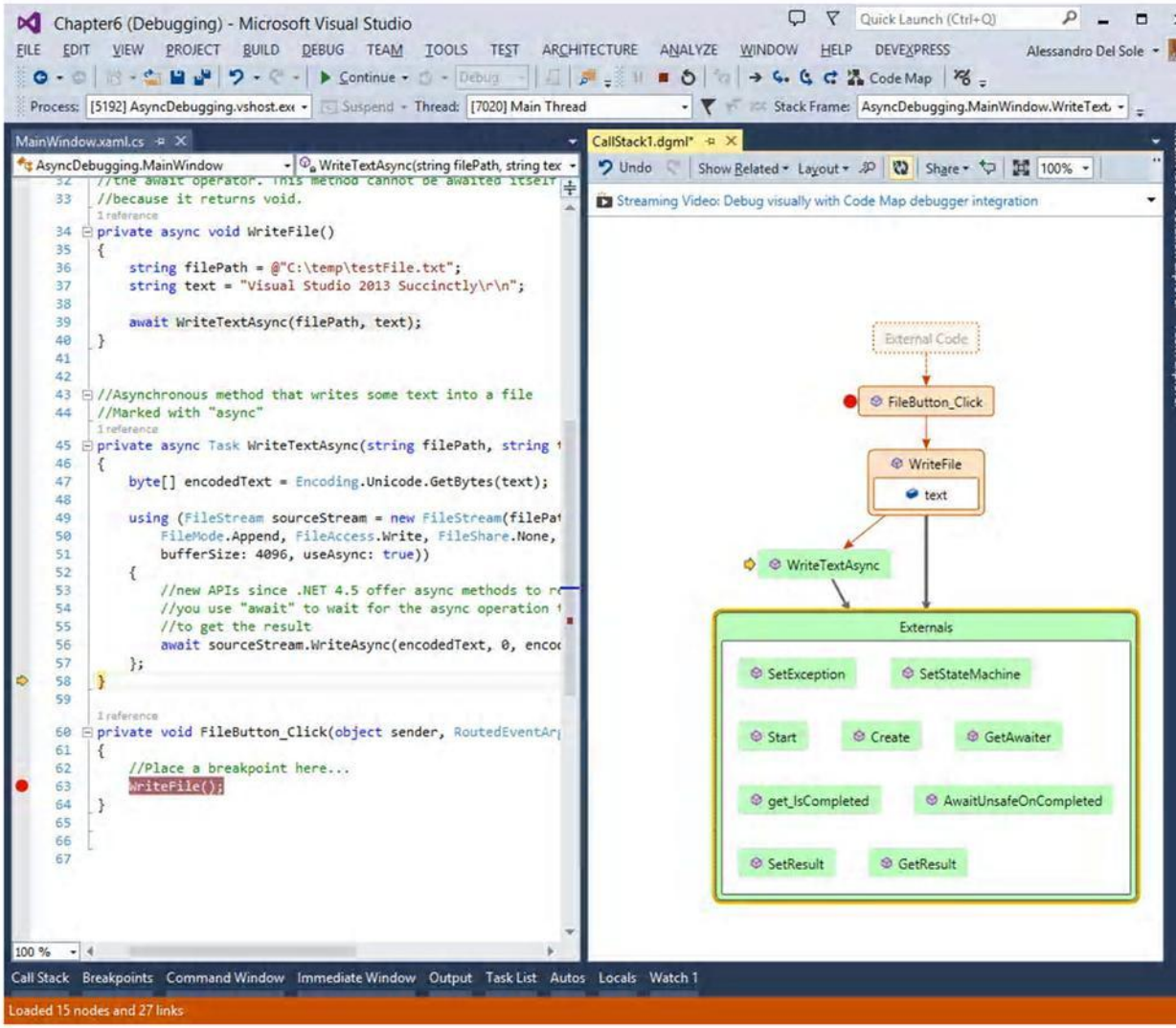


Figure 84: Investigating External Calls

All the method calls you see in the map are handled by the runtime to manage asynchronous operations on your behalf. It is worth mentioning that every time you pass the mouse pointer over a method, a tooltip shows the method definition in code. You can also add comments and flag items for follow up. To add a comment, right-click an item and then select **New Comment**. You will be able to enter your comment inside a text box. To flag an item for follow up, right-click it and then select **Flag for Follow Up**. In Figure 85, you can see a comment and the **`WriteTextAsync`** method flagged for follow up.

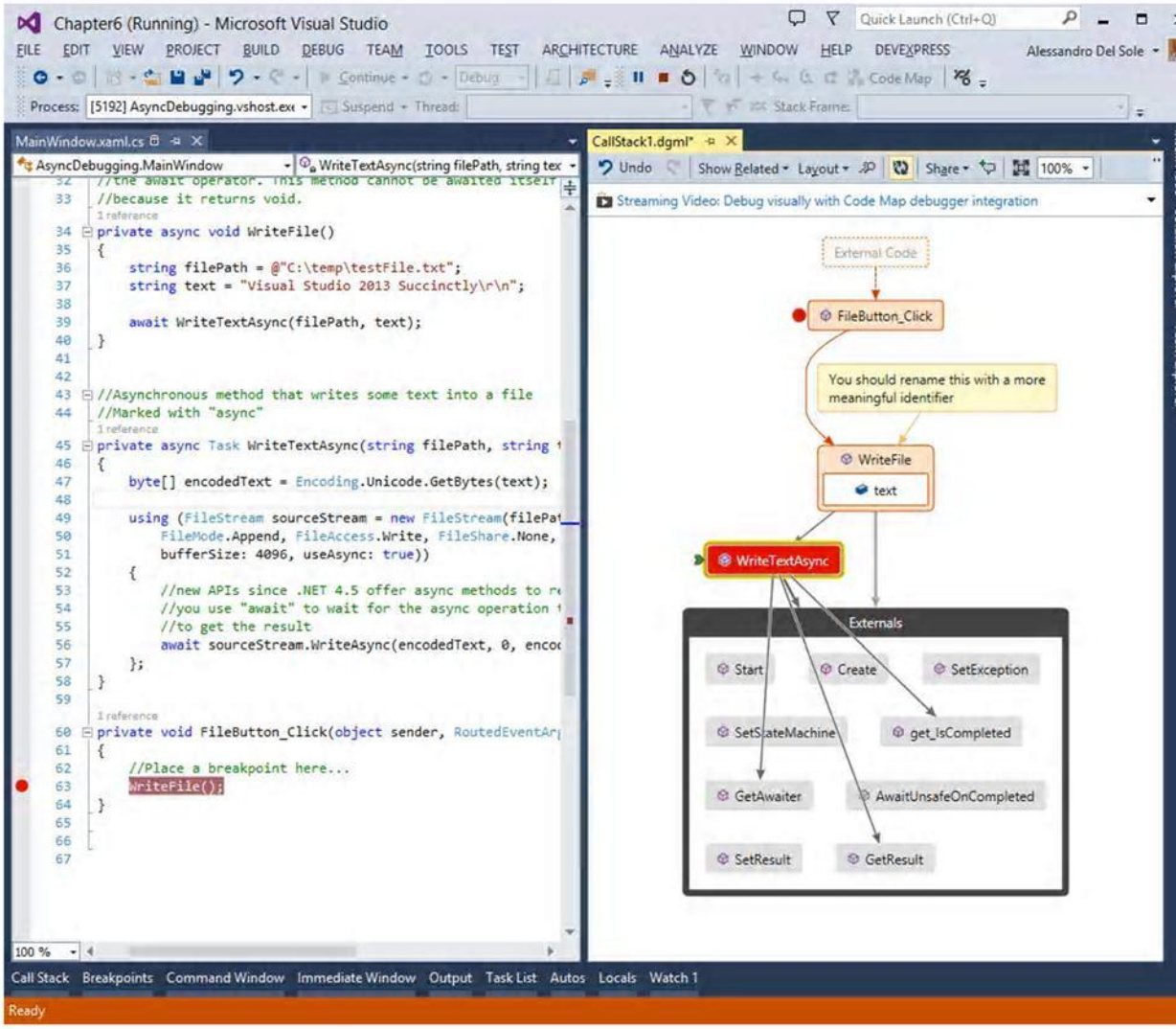


Figure 85: Adding Comments and Flags

You can finally right-click an item and see advanced properties by selecting the **Advanced** group in the context menu. Figure 86 shows the result of the command **Show Containing Type**, **Namespace**, and **Assembly**.

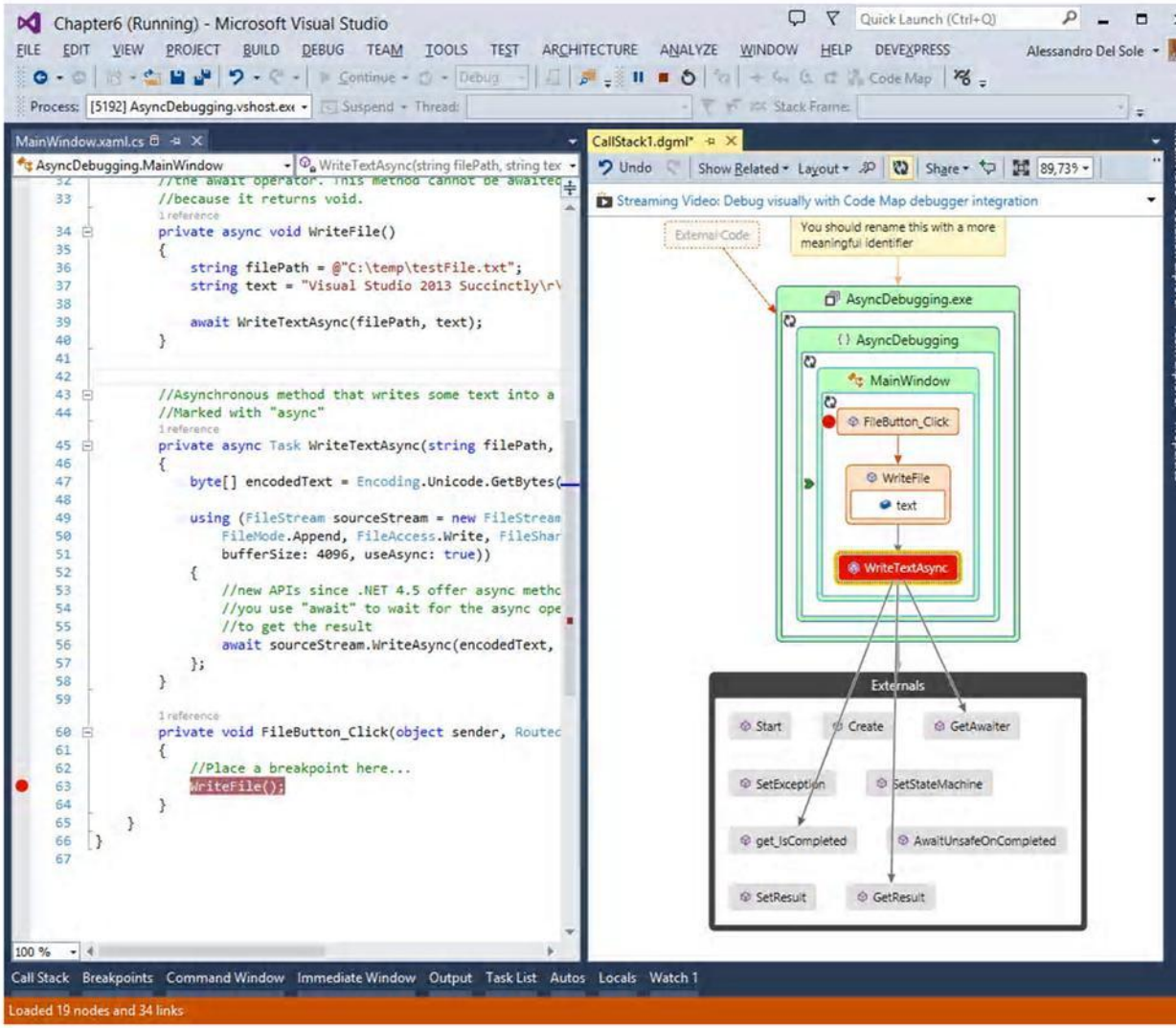


Figure 86: Visualizing Advanced Properties

It is worth mentioning that the context menu you see when you right-click any items will show the Go To Definition command, which will redirect you to the object definition in either the code editor or the Object Browser window. As you can easily understand, Code Map provides a great benefit because it allows debugging while literally seeing what is happening; this makes it easier to discover the most subtle bugs.

Method Return Value

Visual Studio 2013 brings to Visual C# and Visual Basic a feature that was already available to C++, which is the ability to view a method's return value inside the Autos window without the need to step into the code. To understand how this feature works, create a new Console application. Now consider the following code.

Visual C#

```
    class Program
    {
        static void Main(string[] args)
        {
            //Step Over (F10)
            int result = Multiply(Five(), Six());
        }

        private static int Multiply(int num1, int num2)
        {
            return (num1 * num2);
        }

        private static int Five()
        {
            return (5);
        }

        private static int Six()
        {
            return (6);
        }
    }
```

Visual Basic

```
Module Module1

    Sub Main()
        'Step over (F10)
        Dim result As Integer = Multiply(Five(), Six())
    End Sub

    Private Function Multiply(num1 As Integer, num2 As Integer) As Integer
        Return (num1 * num2)
    End Function

    Private Function Five() As Integer
        Return (5)
    End Function

    Private Function Six() As Integer
        Return (6)
    End Function
End Module
```

As you can see, this simplified code returns the result of a multiplication by invoking two methods, each returning an integer value. As suggested in the code, place a breakpoint on the only line of code in the **`Main`** method and start the application by pressing **F5**. You can step over (**F10**) to execute the method without executing the other methods line by line. At this point, you will be able to see the value returned by every intermediate method call in the Autos window, as shown in Figure 87.

Tip: If the Autos window is not displayed automatically, go to Debug, then select Windows, then Autos.

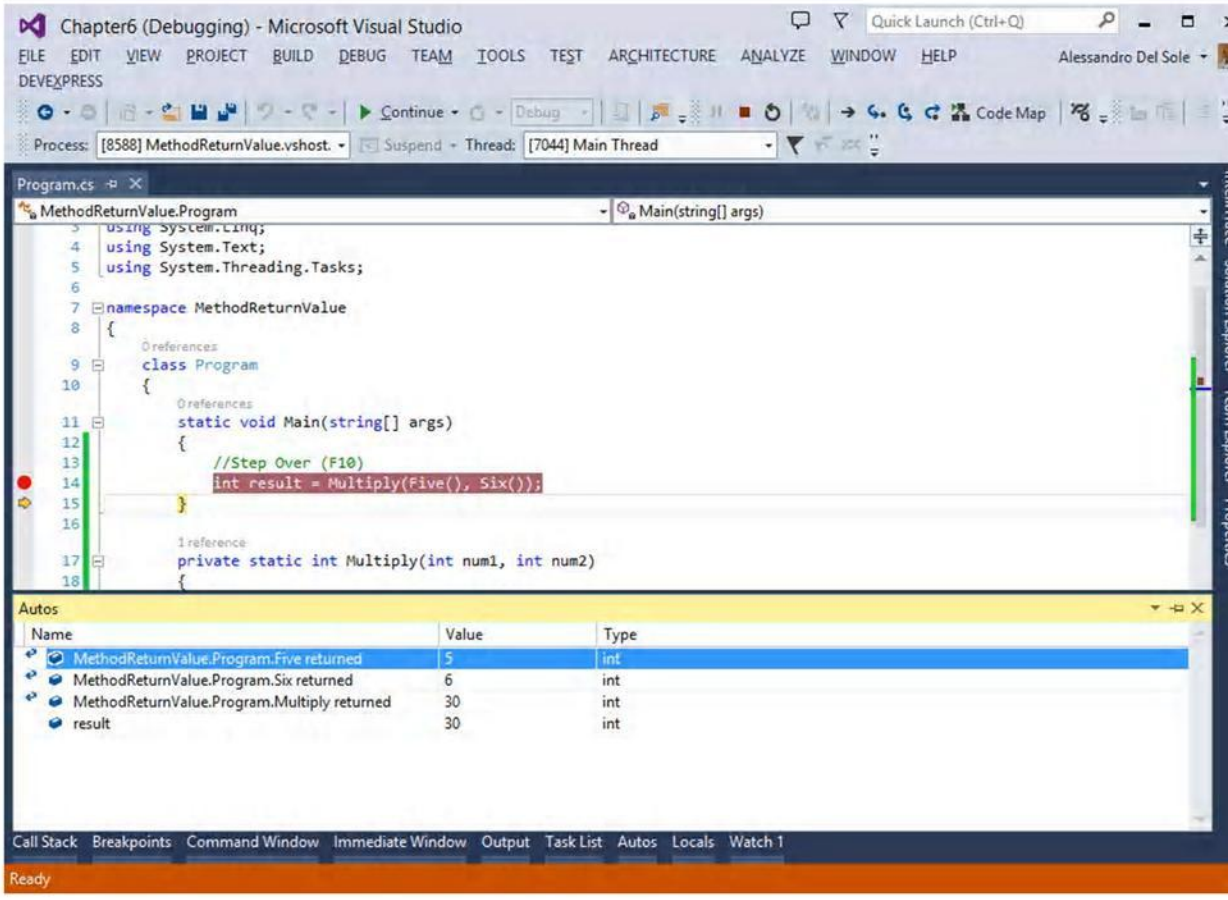


Figure 87: Method Return Values Shown in the Autos Window Without Executing Line by Line

This feature is useful when you need to focus on one piece of code and you do not want to step into every single line, but you still want to see the result of every method call.

Chapter summary

Because debugging is one of the most important activities in application development, Microsoft has made a significant investment to make the debugging experience in Visual Studio 2013 even more productive. Now you can finally use the popular Edit and Continue feature against 64-bit applications. You can take advantage of asynchronous debugging to understand the lifecycle of asynchronous operations based on the Async/Await pattern. You now have a unified place to analyze your applications' performances and behavior with the new Performance and Diagnostics Hub. You can get a graphical representation of your code execution while debugging with Code Map. Finally, you can now get method return values without stepping into every single line of code, just by stepping over the caller method. All these new features will save you time and help you write high-quality code.

Chapter 7 Visual Studio 2013 for Windows 8.1

One reason for the release of a new version of Visual Studio after only one year is that several technologies other have been updated. Probably the most important update has been Windows 8.1., which introduces many new APIs and changes in the existing infrastructure. Because of this, developers need an updated version of the .NET Framework (the 4.5.1) to support Windows 8.1 and a new version of Visual Studio based on .NET 4.5.1. This chapter covers new features in the IDE related to Windows Store app development. If you instead wish to learn about the new APIs in Windows 8.1, you can refer to the MSDN documentation.

Tip: XAML IntelliSense improvements discussed in Chapter 4 are certainly valid for Windows 8.1 app development. Since I already talked about such improvements in detail before, I will not repeat them here.

New project templates

Windows 8.1 introduces a new control called **Hub**, which provides the ability to create a central hub in your Windows Store apps. Basically the concept of a hub is providing users with a landing page that gives an overview of different parts of the app in one place. Before Windows 8.1, developers had to do a bit of work to manually create a hub. To highlight the importance of this control, Visual Studio 2013 introduces the **Hub** control and a specific project template called Hub App, which is available for both XAML and HTML modes, and only if you run Visual Studio 2013 on Windows 8.1. Figure 88 shows the New Project dialog with the new template selected.

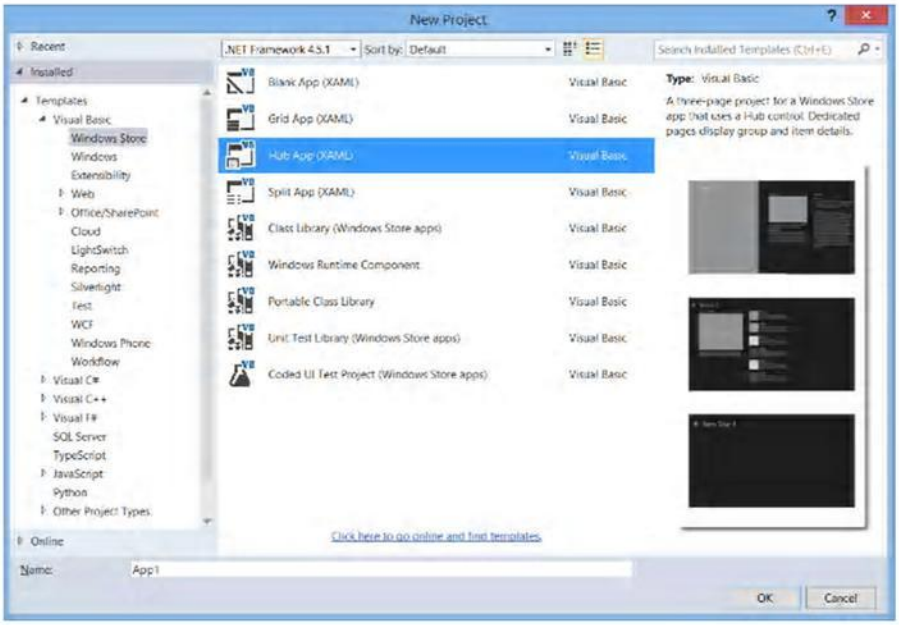


Figure 88: The New Hub App Project Template

Understanding how the **Hub** control works is very easy. You can just create a new project based on the Hub App template. When the project is ready, start the application before looking at the code. By either using the mouse or your finger, you can scroll the main page horizontally to see how the Hub allows creating sections of contents or shortcuts to additional pages. Figure 89 shows the sample app running.

Figure 89: The Hub Control allows organizing content and shortcuts.

Now look at the XAML code. The built-in project template provides a very rich and powerful example, but what you need to know at the higher-level is represented in the following code.

```
        <Hub SectionHeaderClick="Hub_SectionHeaderClick">
            <Hub.Header>
                <Grid>
                <!-- Controls here... -->
                </Grid>
            </Hub.Header>
            <HubSection Width="780" Margin="0,0,80,0">
                <HubSection.Background>
                <!-- Your brush here... -->
                </HubSection.Background>
                <Grid>
                <!-- Controls here... -->
                </Grid>
            </HubSection>
            <HubSection Width="500" Header="Section 1">
                <DataTemplate>
                <!-- Your data-bound items here... -->
                </DataTemplate>
            </HubSection>
```

```
        </Hub>
```

Among the others, the **Hub** control exposes the **Header** property that shows content summarizing the topic of the section. Because of the XAML hierarchical nature, **Header** can be not only text, but also a set of nested controls. The **Hub** control contains **HubSection** controls for as many topics as you need to summarize. The **HubSection** control is very versatile, since it can store any kind of content. As you can see from the code snippets in the previous listing, you can put text or panels, set the background, and even place data-bound controls via **DataTemplate** elements. The MSDN Code Gallery contains a very good example of the **Hub** control for both XAML and HTML that you can download for additional testing. Of course, the MSDN documentation provides everything you need to know for building apps with the **Hub** control. Since explaining how to program this control in detail is beyond the scope of this book, you can check out the related page on MSDN.

Improved Device tool window

When you work with the designer on a Windows Store app, you can take advantage of a useful tool window called Device (also known as Device panel). It allows changing some properties of your application so that you can get a preview of your edits at design time, avoiding the need to launch the application every time. The Device panel is not new in Visual Studio 2013; it was already available in Visual Studio 2012, but has now been reorganized and updated with new features. Figure 90 shows the Device panel.

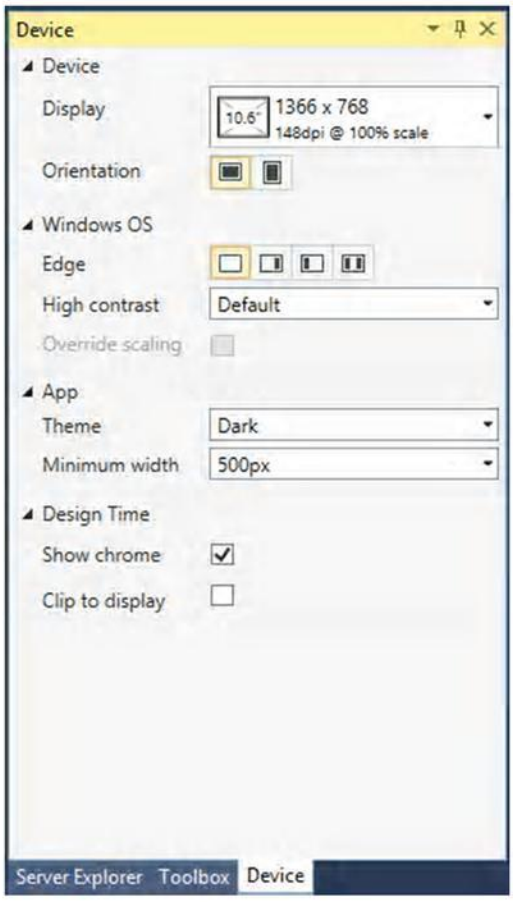


Figure 90: The Device Tool Window

Properties on the window are self-explanatory, and it's easy to see the result on the designer when you change them. In summary you can:

- Change the app resolution in the designer with the Display property.
- Switch between horizontal and vertical orientation with the Orientation property.
- Test how the app will appear on screen with the Edge property.
- Select the screen contrast with the High contrast property.
- Test the app on a different scaling with the Override scaling property. Scaling is increased by 40%.
- Change the theme to see how the app responds to system settings with the Theme property.
- Establish a minimum app width with the Minimum width property
- Show or remove the device chrome in the designer with the Chrome property.
- Clip the entire document or show the document display with the Clip to display property.

The Device panel is a good companion to get a preview of the behavior of the app directly in the designer, so that you can change the app layout and appearance and see if the result you get is what you (and your users) expect.

Connect to Windows Azure mobile services

In Chapter 5, you discovered new features in the Server Explorer window to provide integration with Windows Azure services from within the IDE. You learned what a mobile service is and how to create one from Server Explorer. Continuing the integration with the cloud platform, Visual Studio 2013 allows connecting easily to a mobile service in Windows 8.1 applications.

In a Store app, first save the project—otherwise the tooling will not work without giving you any warning. Then right-click the project name in Solution Explorer, then select **Add**, **Connected Service**. At this point the Services Manager dialog appears. Select the **Windows Azure** node to see a list of available services, as shown in Figure 91.

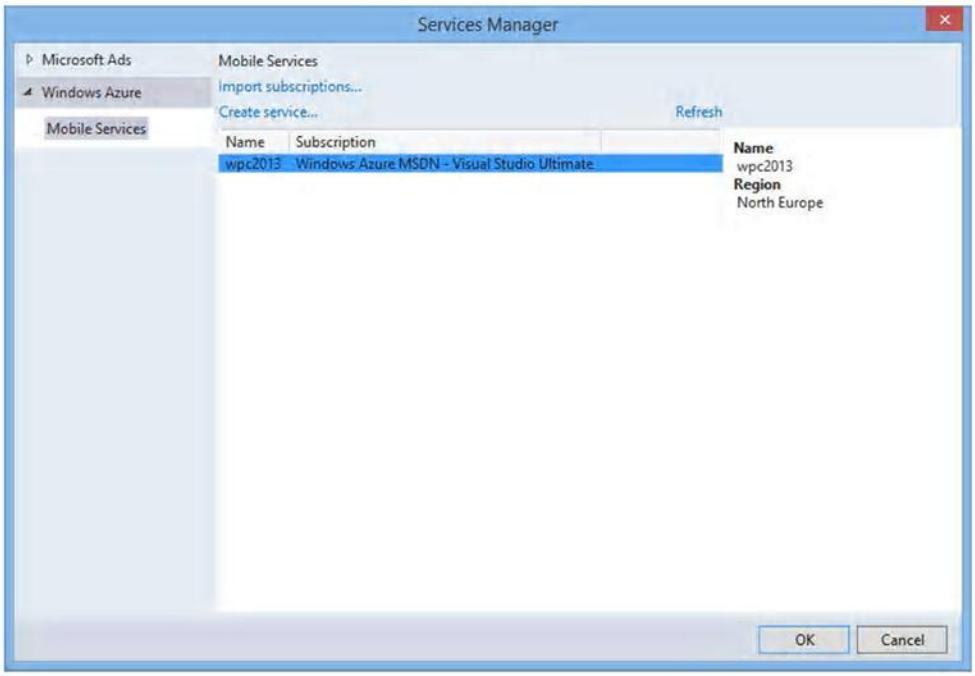


Figure 91: The Services Manager dialog shows available Mobile Services.

Once you click **OK**, a service reference is added. Visual Studio 2013 will automatically add references to assemblies that are required in order to connect to the service in code and to manage data stored inside the service. You can expand the **References** node in Solution Explorer to see what libraries have been referenced; Figure 92 demonstrates this.

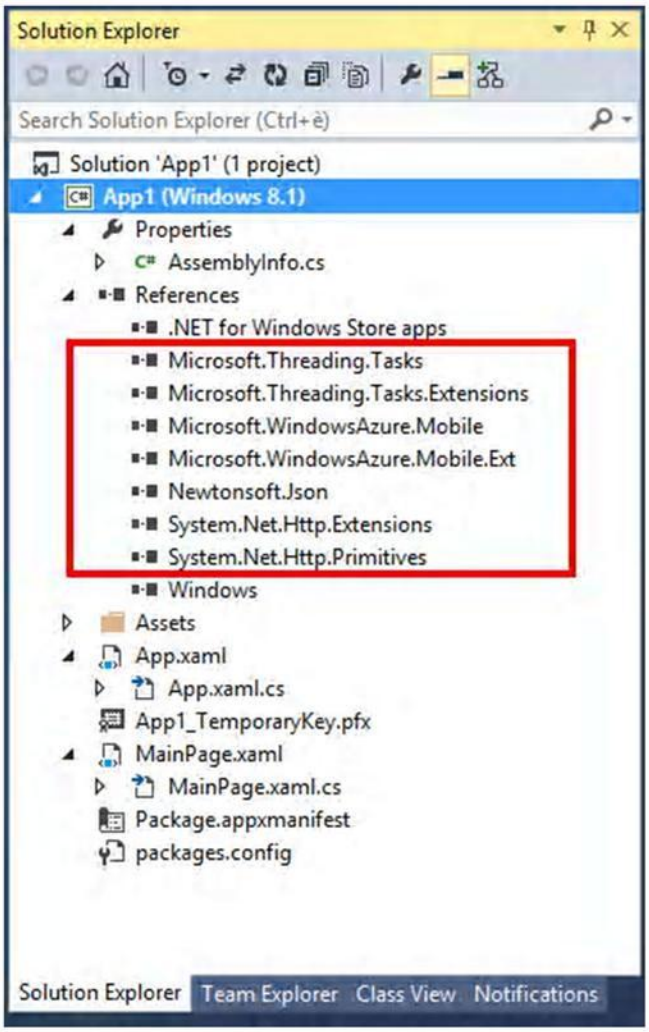


Figure 92: References Added to Support Coding against Mobile Services

The new assemblies required to interact with a Mobile Service are Microsoft.WindowsAzure.Mobile.dll and Microsoft.WindowsAzure.Mobile.Ext.dll. Other assemblies are part of the .NET runtime and are required for serializing and deserializing data through the JSON format over the network. Not limited to this, Visual Studio 2013 will also add to the **App** class code the following line (**yourmobileservice** stands for the name of your service and **YOURSECRETKEY** stands for the client secret key, both added appropriately):

Visual C#

```
public static Microsoft.WindowsAzure.MobileServices.MobileServiceClient
yourmobileserviceClient = new
Microsoft.WindowsAzure.MobileServices.MobileServiceClient(
        "https://yourmobileservice.azure-mobile.net/",
        "YOURSECRETKEY");
```

Visual Basic

```
Public Shared yourmobileserviceClient As New
Microsoft.WindowsAzure.MobileServices.MobileServiceClient(
        "https://yourmobileservice.azure-mobile.net/",
        "YOURSECRETKEY");
```

By creating an instance of the **`Microsoft.WindowsAzure.MobileServices.MobileServiceClient`** class, your app will be able to connect to the specified mobile service. For a deeper understanding of the code you need to write to manage data and C.R.U.D. operations from your app, you can follow the example shown in the Getting started with Mobile Services page in the Windows Azure documentation, which also provides guidance to use these services in other platforms.

Asynchronous debugging

Visual Studio 2013 introduces a new tool window called Tasks, which helps developers debug asynchronous operations written according to the Async/Await pattern introduced with the previous version. This feature was discussed in detail in the previous chapter, so you should be able to use it successfully against Windows Store apps.

Analyze performance with the XAML UI Responsiveness Tool

Visual Studio 2013 brings to XAML Store apps the UI Responsiveness Tool that was already available for HTML/JavaScript Store applications. As the name implies, this tool analyzes the user interface performance and generates a detailed report about the app behavior. In order to use this tool, you need to open the Performance and Diagnostics Hub that you discovered in the previous chapter. When visible, you have to select the **XAML UI Responsiveness** check box, as shown in Figure 93.

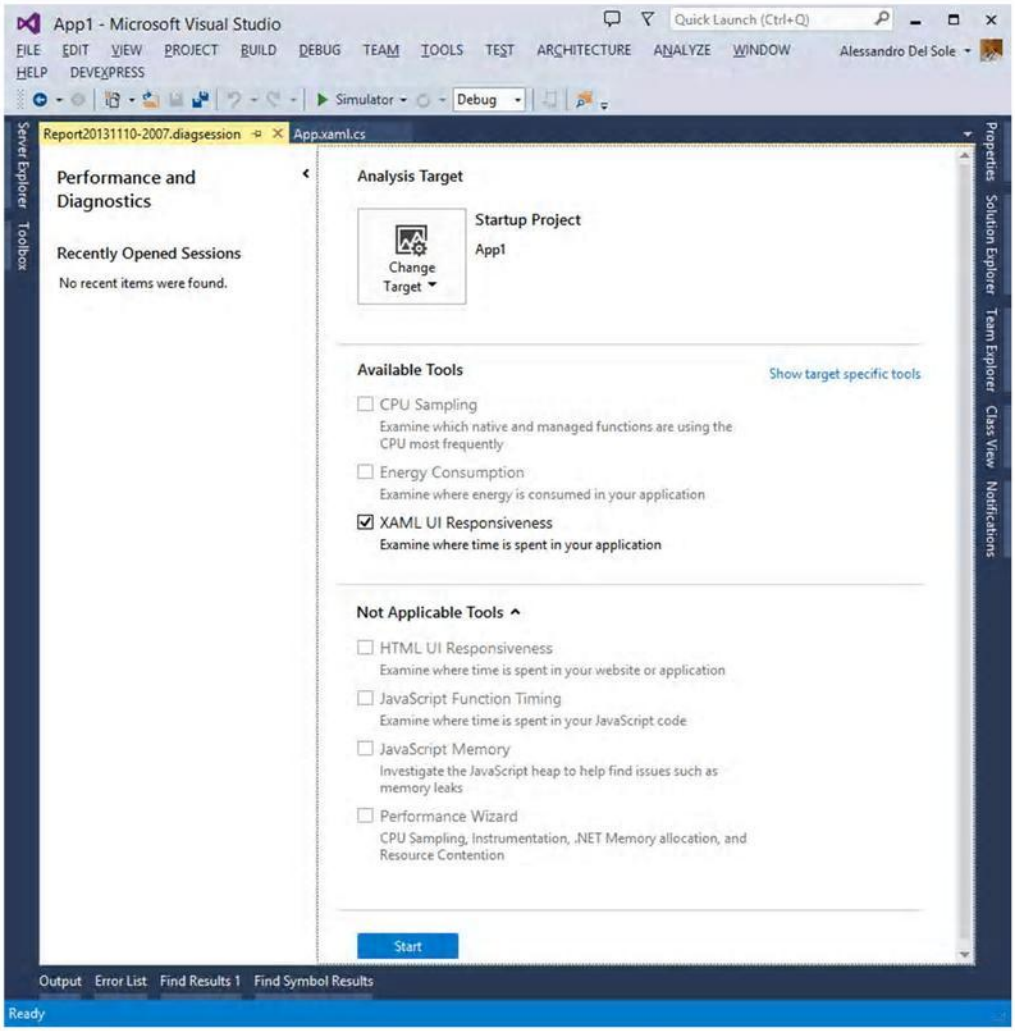


Figure 93: Selecting the XAML UI Responsiveness Tool

When ready, click the **Start** button. Use your app for a while, then close it or go back to Visual Studio and click the **Stop Collection** hyperlink. After a few seconds, Visual Studio shows a detailed report about the collected information about performances; Figure 94 shows an example.

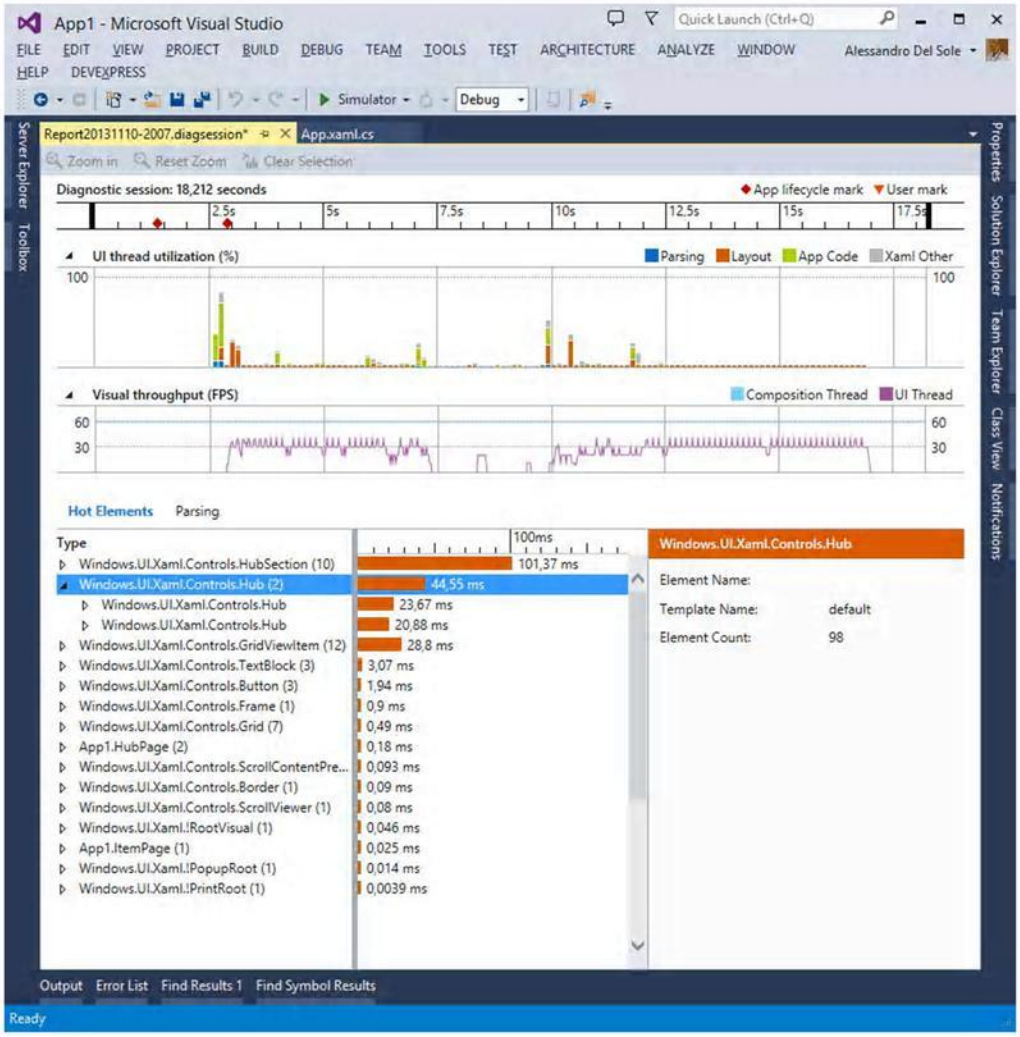


Figure 94: Selecting the XAML UI Responsiveness Tool

The report can be divided into four main parts. Let's discuss them in more detail.

Diagnostic Session

The Diagnostic Session report shows information about the application lifecycle and user interaction. It displays the test duration and it shows how much time the app required for activation. This section can be useful to discover important performance problems that you must avoid for a successful app submission.

UI Thread Utilization

The UI Thread Utilization section shows how the UI thread has been exploited in percentage by different tasks managed by the runtime. You can understand how many resources have been consumed by the XAML parser (blue), how many in rendering the user interface (dark orange), in executing the app code (light green), and in other tasks related to XAML (not parsing). This can be very useful to understand what areas of your code have the most negative impact on the overall performance.

Visual Throughput (FPS)

This section shows how many frames per second (FPS) have been rendered during the application lifecycle; for timing, you can take the Diagnostic Session as a reference. This tool is very straightforward, and can show frames in both the UI thread and the composition thread. If you pass the mouse pointer over the graphic, you will see a tooltip showing frames per second for both threads at the given time.

Tip: If you are new to Windows Store application development, you might not be familiar wi the concept of composition thread. The composition thread was introduced with Windows Phone and is a companion thread for the UI thread, in that it does some work that the UI thread would normally do. The composition thread is normally responsible for combining graphics texture and for sending them to the GPU for rendering. This is all managed by the runtime; by invoking the composition thread, the runtime can make an app stay much more responsive and you, as the developer, will not need to do any additional work.

Hot Elements and Parsing

At the bottom of the report, you will find two tabs, **Hot Elements** and **Parsing**. Hot Elements shows the list of UI elements (.NET objects with their fully qualified name) and the time in milliseconds they have been busy with the application execution. Objects in the list can be expanded to show nested controls and types. When you click an object, you will also be able to see additional information such as nested elements count, XAML code file (if not a system object), and the control template, on the right side of the window. The Parsing tab shows the list of XAML files in the application package and how much time in milliseconds has been required for parsing. This tool is not limited to XAML files you see in Solution Explorer, but also works against built-in XAML files prepackaged in the application.

Chapter summary

Windows 8.1 is the new major upgrade for the Windows 8 operating system, which introduces tons of new APIs and features that developers can leverage to build amazing apps. Visual Studio 2013 is the environment you need to build apps for Windows 8.1. This new version introduces a new **Hub** control, which Visual Studio 2013 supports with the Hub App project template. At design time, you can get previews of your app's look and feel via the Device panel. You can now easily connect to Windows Azure Mobile Services and Visual Studio will do most of the work for you. Great apps are fluid and fast apps, so with Visual Studio 2013 you have a new analysis tool called XAML UI Responsiveness, which helps you understand how and where your app spends more time and consumes more resources.

Wednesday, December 30, 2015
2:42 PM

AI : Neural Network for beginners (Part 2 of 3)


Sacha Barber, 29 Jan 2007

AI : An Introduction into Neural Networks (Multi-layer networks / Back Propagation)

Download demo project (includes source code) - 812 Kb

Introduction

This article is part 2 of a series of 3 articles that I am going to post. The proposed article content will be as follows:

1. Part 1 : Is an introduction into Perceptron networks (single layer neural networks).
2. Part 2 : This one, is about multi layer neural networks, and the back propagation training method to solve a non linear classification problem such as the logic of an XOR logic gate. This is something that a Perceptron can't do. This is explained further within this article.
3. Part 3 : Will be about how to use a genetic algorithm (GA) to train a multi layer neural network to solve some logic problem.

Summary

This article will show how to use a multi-layer neural network to solve the XOR logic problem.

A Brief Recap (From part 1 of 3)

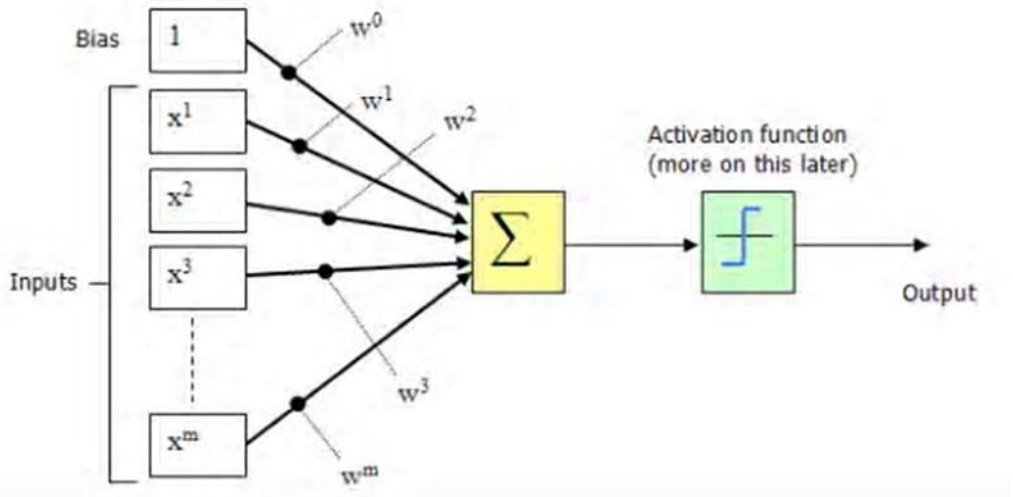
Before we commence with the nitty gritty of this new article which deals with multi layer Neural Networks, let just revisit a few key concepts. If you haven't read Part 1, perhaps you should start there.

Perceptron Configuration (Single layer network)

The inputs ☺╤ϸ ╡╤ɳ ╡╤ϸ┘┘╤≠ ☻and connection weights☺∑o ╘∑ɳ ╘∑ϸ┘┘∑≠ ☻ shown below are typically real values, both positive and negative (-).

The perceptron itself, consists of weights, the summation processor, an activation function, and an adjustable threshold processor

For convenience, the normal practice is to treat the bias as just another input. The following diagram illustrates the revised configuration.



The bias can be thought of as the propensity (a tendency towards a particular way of behaving) of the perceptron to fire irrespective it's inputs. The perceptron configuration network shown above fires if the weighted sum > 0, or if you have into maths type explanations

$$\sum_{i=1}^{m} bias + (w^i x^i)$$

So that's the basic operation of a perceptron. But we now want to build more layers of these, so let's carry on to the new stuff.

So Now The New Stuff (More layers)

From this point on, anything that is being discussed relates directly to this article's code.

In the summary at the top, the problem we are trying to solve was how to use a multi-layer neural network to solve the XOR logic problem. So how is this done. Well it's really an incremental build on what Part 1 already discussed. So let's march on.

What does the XOR logic problem look like? Well, it looks like the following truth table:

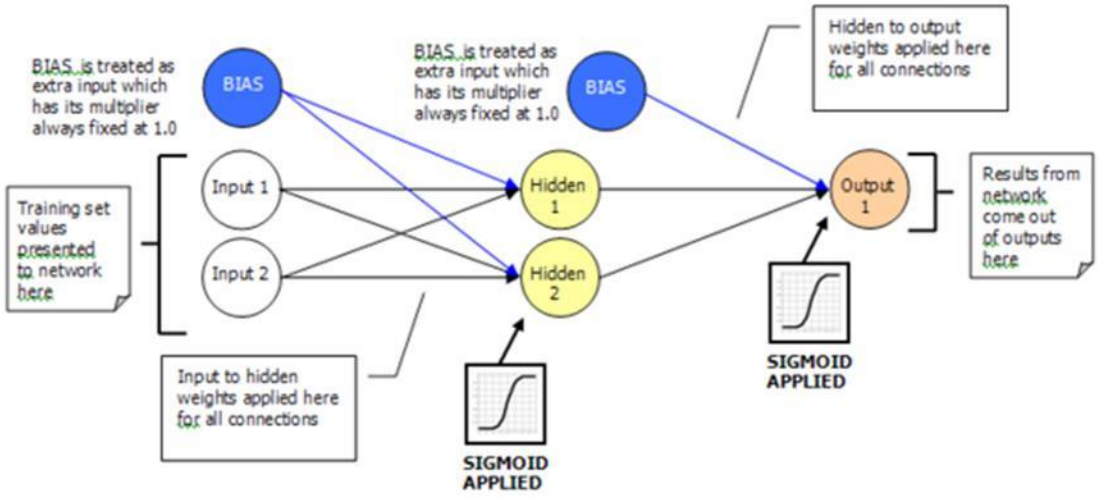
I1	I2	Output
0	0	0
0	1	1
1	0	1
1	1	0

XOR Logic Gate

Remember with a single layer (perceptron) we can't actually achieve the XOR functionality, as it is not linearly separable. But with a multi-layer network, this is achievable.

What Does The New Network Look Like

The new network that will solve the XOR problem will look similar to a single layer network. We are still dealing with inputs / weights outputs. What is new is the addition of the hidden layer.



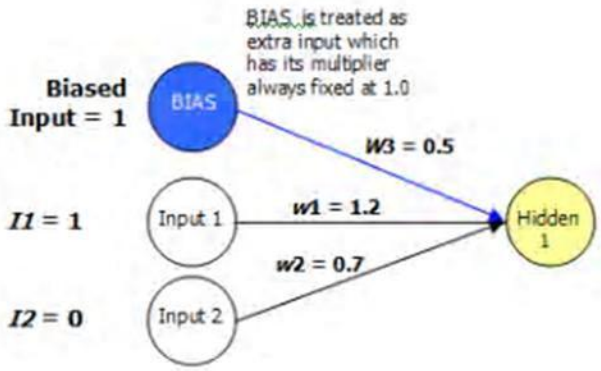
As already explained above, there is one input layer, one hidden layer and one output layer.

It is by using the inputs and weights that we are able to work out the activation for a given node. This is easily achieved for the hidd layer as it has direct links to the actual input layer.

The output layer, however, knows nothing about the input layer as it is not directly connected to it. So to work out the activation for output node we need to make use of the output from the hidden layer nodes, which are used as inputs to the output layer nodes.

This entire process described above can be thought of as a pass forward from one layer to the next.

This still works like it did with a single layer network; the activation for any given node is still worked out as follows:



$$A = \sum_{n=1}^{N+1} wi * Ii$$

NOTE : N + 1 is bias and the value of the input for the bias is 1.

Where (wi is the weight(i), and Ii is the input(i) value)

You see it the same old stuff, no demons, smoke or magic here. It's stuff we've already covered.

So that's how the network looks/works. So now I guess you want to know how to go about training it.

Types Of Learning

There are essentially 2 types of learning that may be applied, to a Neural Network, which is "Reinforcement" and "Supervised"

Reinforcement

In Reinforcement learning, during training, a set of inputs is presented to the Neural Network, the Output is 0.75, when the target w expecting 1.0.

The error (1.0 - 0.75) is used for training ('wrong by 0.25').

What if there are 2 outputs, then the total error is summed to give a single number (typically sum of squared errors). Eg "your total error on all outputs is 1.76"

Note that this just tells you how wrong you were, not in which direction you were wrong.

Using this method we may never get a result, or it could be a case of 'Hunt the needle'.

NOTE : Part 3 of this series will be using a GA to train a Neural Network, which is Reinforcement learning. The GA simply does what a GA does, and all the normal GA phases to select weights for the Neural Network. There is no back propagation of values. The Neura Network is just good or just bad. As one can imagine, this process takes a lot more steps to get to the same result.

Supervised

In Supervised Learning the Neural Network is given more information.
Not just 'how wrong' it was, but 'in what direction it was wrong' like 'Hunt the needle' but where you are told 'North a bit', 'West a b

So you get, and use, far more information in Supervised Learning, and this is the normal form of Neural Network learning algorithm Back Propagation (what this article uses, is Supervised Learning)

Learning Algorithm

In brief, to train a multi-layer Neural Network, the following steps are carried out:

- Start off with random weights (and biases) in the Neural Network
- Try one or more members of the training set, see how badly the output(s) are compared to what they should be (compared the target output(s))
- Jiggle weights a bit, aimed at getting improvement on outputs
- Now try with a new lot of the training set, or repeat again, jiggling weights each time
- Keep repeating until you get quite accurate outputs

This is what this article submission uses to solve the XOR problem. This is also called "Back Propagation" (normally called BP or BackProp)

Backprop allows you to use this error at output, to adjust the weights arriving at the output layer, but then also allows you to calcula the effective error 1 layer back, and use this to adjust the weights arriving there, and so on, back-propagating errors through any number of layers.

The trick is the use of a sigmoid as the non-linear transfer function (which was covered in Part 1. The sigmoid is used as it offers the ability to apply differentiation techniques.

$$y = g(x) = \frac{1}{1 + e^{-x}}$$

Because this is nicely differentiable —it so happens that

$$\frac{dg}{dx} = g'(x) = g(x)(1 - g(x))$$

Which in context of the article can be written as

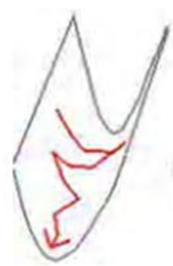
delta_outputs[i] = outputs[i] * (1.0 - outputs[i]) * (targets[i] - outputs[i])

It is by using this calculation that the weight changes can be applied back through the network.

Things To Watch Out For

Valleys: Using the rolled ball metaphor, there may well be valleys like this, with steep sides and a gently sloping floor. Gradient desc tends to waste time swooshing up and down each side of the valley (think ball!)

So what can we do about this. Well we add a momentum term, that tends to cancel out the back and forth movements and emphasizes any consistent direction, then this will go down such valleys with gentle bottom-slopes much more successfully (faster)

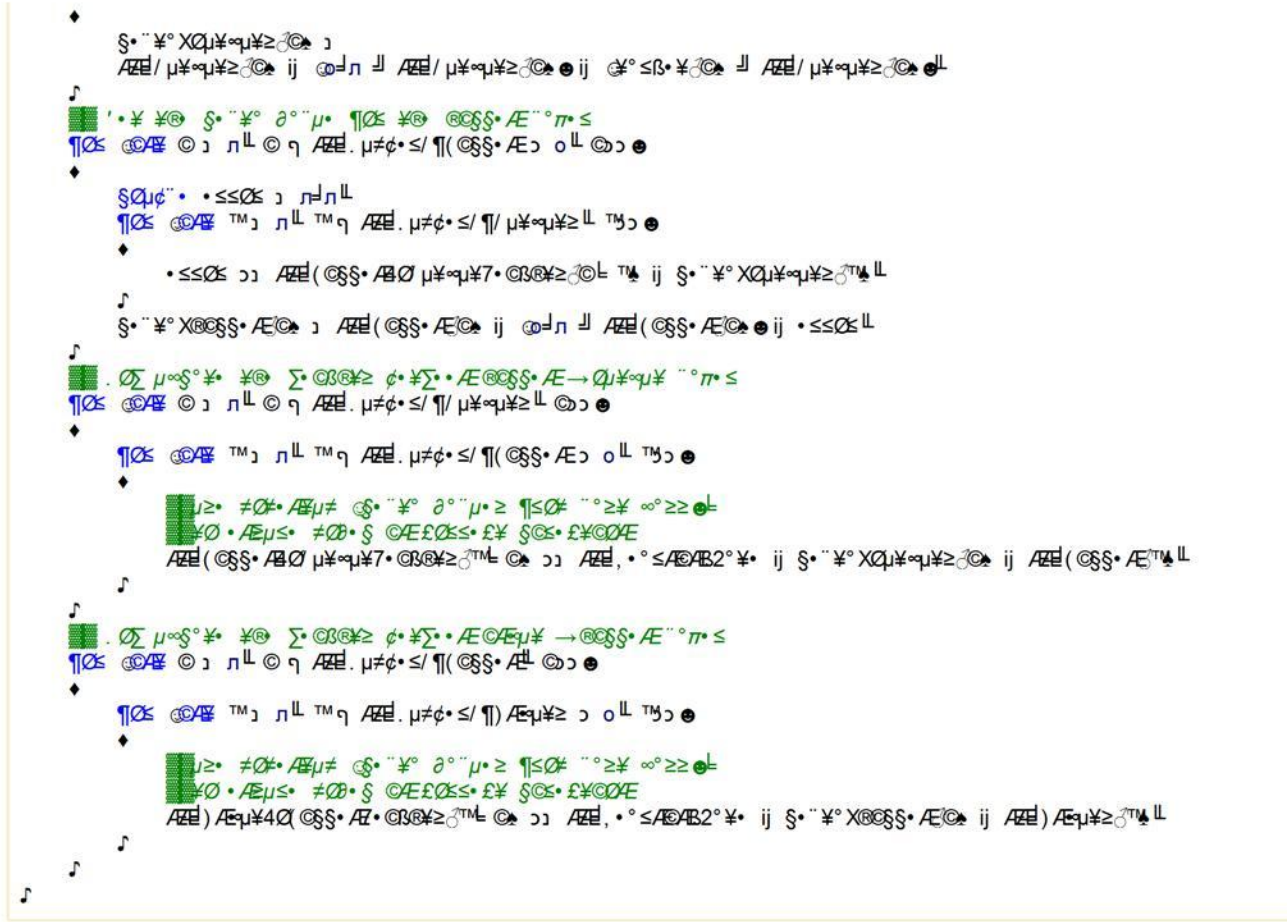


Starting The Training

This is probably best demonstrated with a code snippet from the article's actual code:

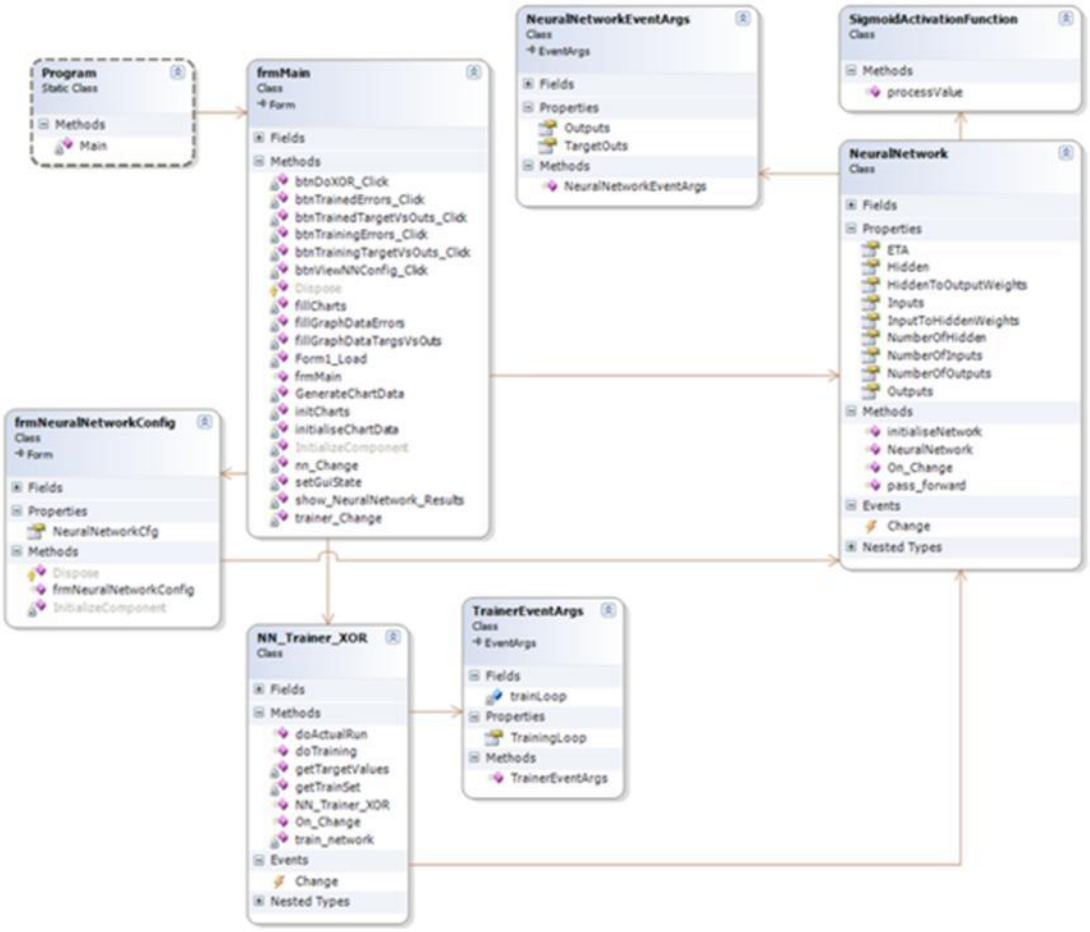
Hide ¡ Shrink ▲ ¡ Copy C

```
ʋ≥µ≠≠°≤ππ
4® ≠°©Æ¥≤°©Æ©ÆB 4® •∏• £¥•§ ¥°≤ß•¥ ∂°¨µ•≥ °≤• ∞°≥≥•§ ©Æ¥Ø ¥®©≥
≠•¥®Ø§ °≥ ∞°≤°≠•¥•≤≥ °Æ§ ¥® ʋ≥•• £≤•¶ɣЭ.•µ≤°¨.•¥∑Ø≤´Эπ.•µ≤°¨.•¥∑Ø≤´ʋ≥••π
©≥ ¥® Ƶ∞§°¥•§ ∑©¥® ≥≠°¨¨ ∑•©ß®¥ £®°ÆB•≥ ¶Ø≤ ¥®©≥ ¥≤°©Æ©ÆB ©¥•≤°¥©ØÆ
4®©≥ ≠•¥®Ø§ °¨≥Ø °∞∞¨© § ≠Ø≠•Æ¥µ≠ ¥Ø •Æµ≤• ¥®°¥ ¥® .•µ≤°¨.•¥∑Ø≤´ ©≥
Æ≤≤¥µ≤•§ ©ÆØ ∞≤ØŒ••§©ÆB ©Æ¥® £Ø≤≤•£¥ §©≤•£¥©ØÆ 7• °≤• ¥≤π©ÆB ¥Ø °∂Ø©§ ∂°¨¨•π≥
)¶ πØµ §ØÆ¥ ´ÆØ∑ ∑®°¥ ∂°¨¨•π≥ ≠•°Æ ≤•°§ ¥® °≤¥©£¨•≥ °≥≥Ø£©°¥•§ ¥•∏¥
ʋ≥µ≠≠°≤ππ
ʋ∞°≤°≠ Æ≠•ɣЭ¥°≤ß•¥Эπ! §Øµ¢¨•♂♠ °≤≤°π £ØÆ¥°©Æ©ÆB ¥® ¥°≤ß•¥ ∂°¨µ• ≥ʋ∞°≤°≠π
∞≤©∂°¥• ∂Ø©§ ¥≤°©ÆXÆ¥∑Ø≤´ §Øµ¢¨•♂♠ ¥°≤ß•¥
♦
    ß•¥ ≠Ø≠•Æ¥µ≠ ∂°¨µ•≥ §•¨¥° ∂°¨µ•≥ ¶≤Ø≠ ¨°≥¥ ∞°≥≥
    §Øµ¢¨•♂♠ §•¨¥°X®©§§•Æɔ Æ∑ §Øµ¢¨•♂ÆÆ. µ≠¢•≤/¶(©§§•Æɔ o♠╙
    §Øµ¢¨•♂♠ §•¨¥°XØµ¥∞µ¥≥ ɔ Æ∑ §Øµ¢¨•♂ÆÆ. µ≠¢•≤/¶/ µ¥∞µ¥≥♠╙
```



So Finally The Code

Well, the code for this article looks like the following class diagram (It's Visual Studio 2005 C#, .NET v2.0)



The main classes that people should take the time to look at would be :

- . . X4≤° ©Æ≤X8/ 2 : Trains a Neural Network to solve the XOR problem
- 4≤° ©Æ≤%∂• Æ¥! ≤ß≥ : Training event args, for use with a GUI
- . • µ≤° ¨ . • ¥∑Ø≤´ : A configurable Neural Network
- . • µ≤° ¨ . • ¥∑Ø≤´ %∂• Æ¥! ≤ß≥ : Training event args, for use with a GUI
- 3©ß≠Ø©§! £¥©∂° ¥©Ø®µÆ¥©Ø® A static method to provide the sigmoid activation function

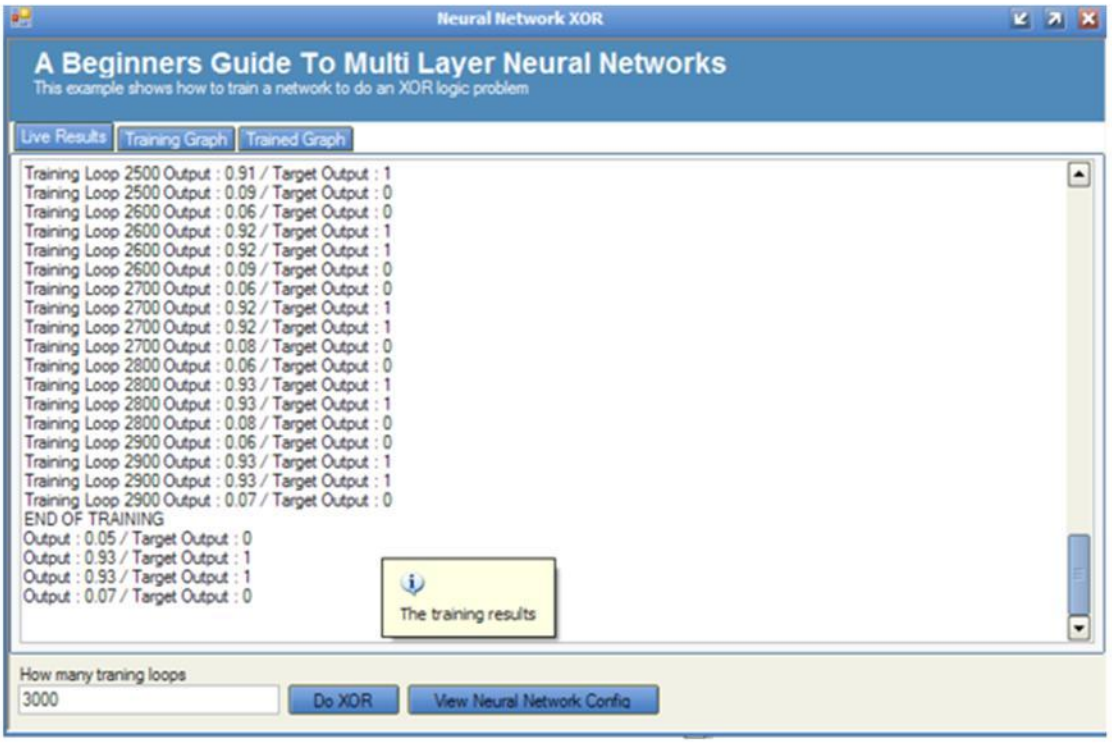
The rest are a GUI I constructed simply to show how it all fits together.

NOTE : the demo project contains all code, so I won't list it here.

Code Demos

The DEMO application attached has 3 main areas which are described below:

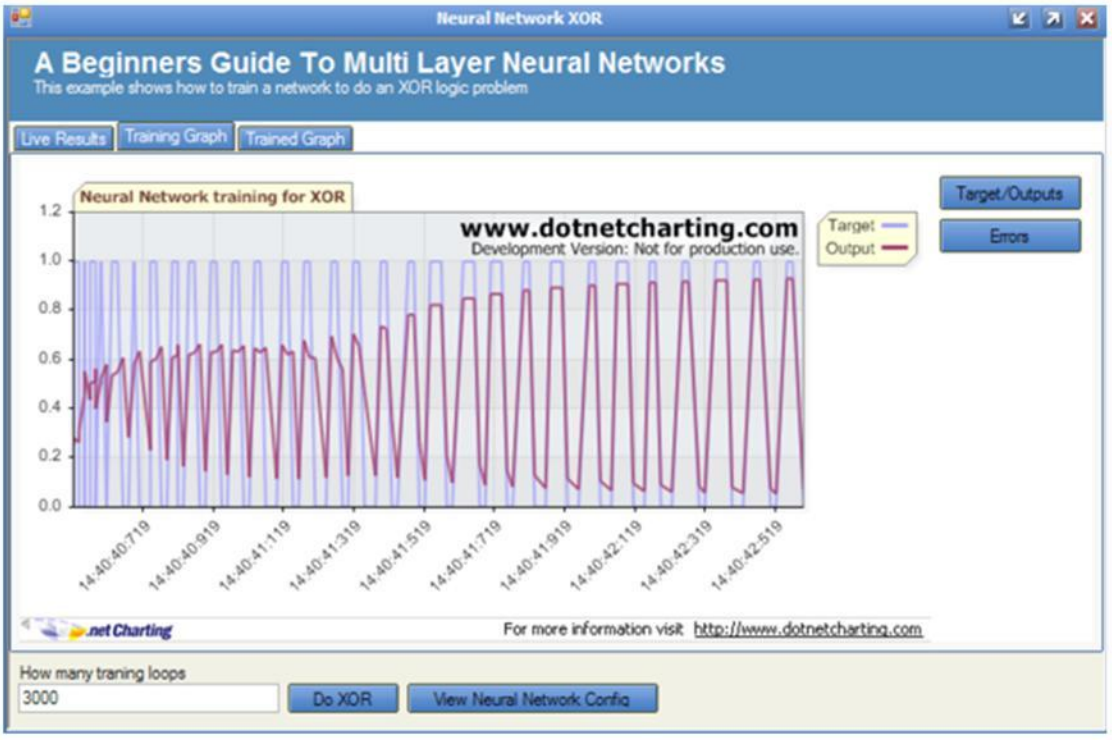
LIVE RESULTS Tab



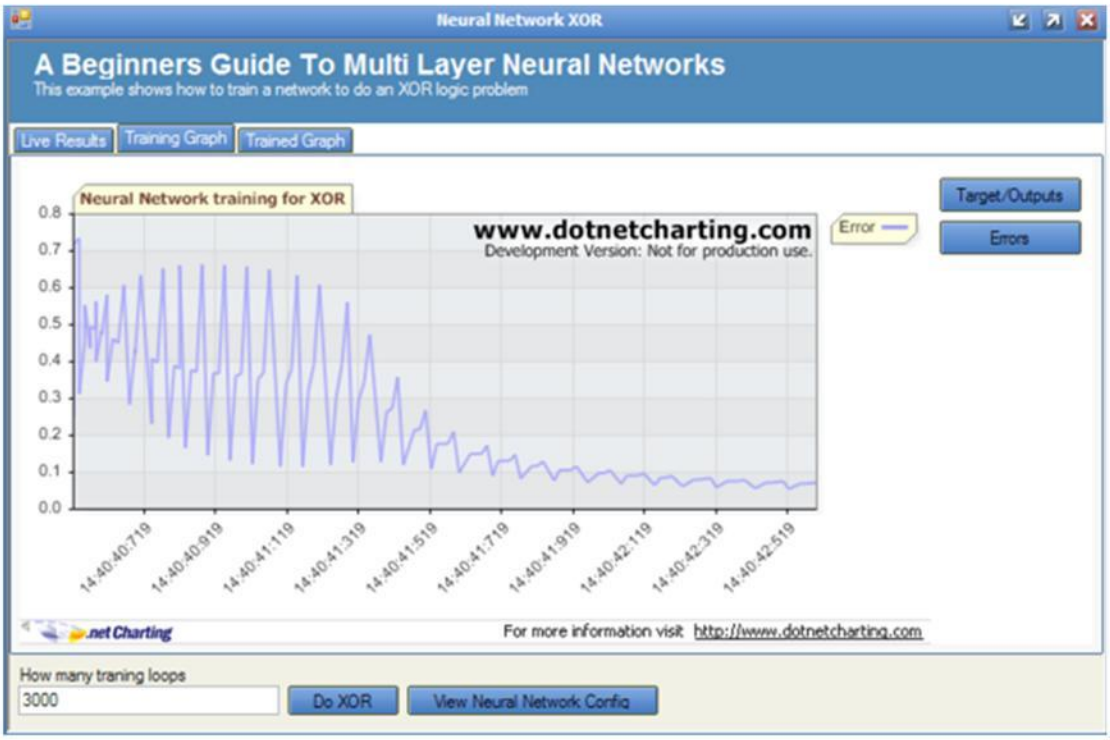
It can be seen that this has very nearly solved the XOR problem (You will probably never get it 100% accurate)

TRAINING RESULTS Tab

Viewing the training phase target/outputs together

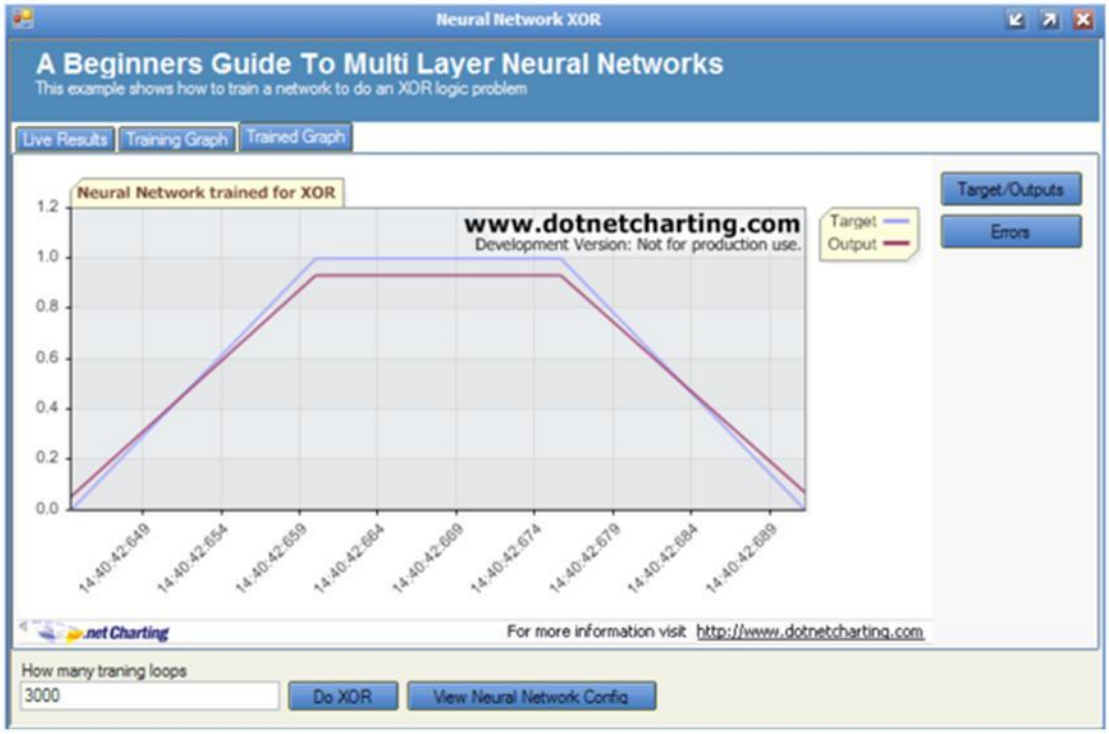


Viewing the training phase errors

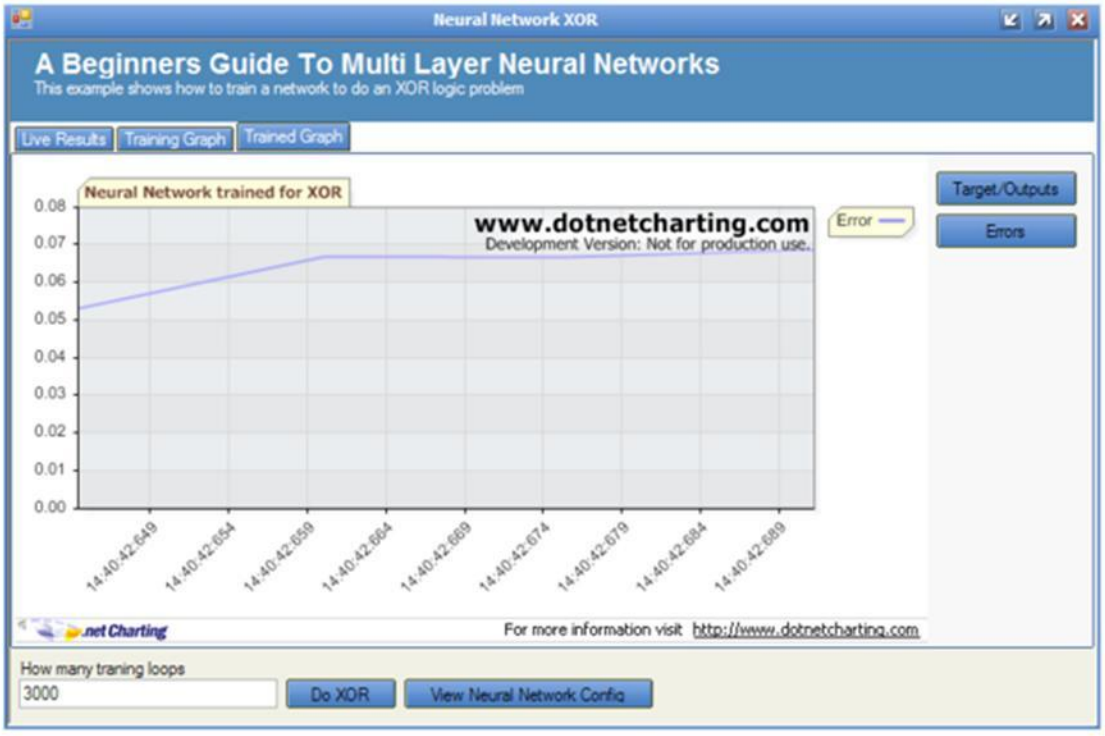


TRAINED RESULTS Tab

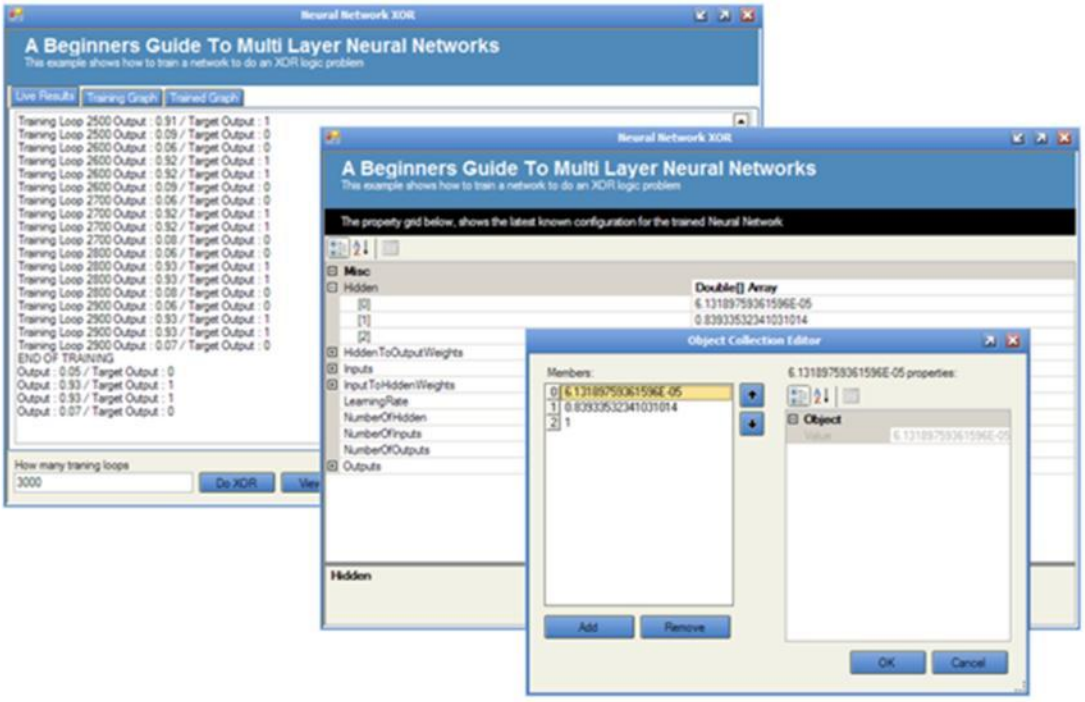
Viewing the trained target/outputs together



Viewing the trained errors



It is also possible to view the Neural Networks final configuration using the "View Neural Network Config" button. If people are interested in what weights the Neural Network ended up with, this is the place to look.



What Do You Think ?

That's it. I would just like to ask, if you liked the article, please vote for it.

I think AI is fairly interesting, that's why I am taking the time to publish these articles. So I hope someone else finds it interesting, an that it might help further someone's knowledge, as it has my own.

Anyone that wants to look further into AI type stuff, that finds the content of this article a bit basic should check out Andrew Krillov articles, at Andrew Krillov CP articles as his are more advanced, and very good. In fact anything Andrew seems to do, is very good.

History

- v1.0 24/11/06

Bibliography

- Artificial Intelligence 2nd edition, Elaine Rich / Kevin Knight. McGraw Hill Inc.
- Artificial Intelligence, A Modern Approach, Stuart Russell / Peter Norvig. Prentice Hall.

License

Share

About the Author

Sacha Barber

Software Developer (Senior)
United Kingdom

I currently hold the following qualifications (amongst others, I also studied Music Technology and Electronics, for my sins)

- MSc (Passed with distinctions), in Information Technology for E-Commerce

Both of these at Sussex University UK.

Award(s)

I am lucky enough to have won a few awards for Zany Crazy code articles over the years

- Microsoft C# MVP 2015
- Codeproject MVP 2015
- Microsoft C# MVP 2014
- Codeproject MVP 2014
- Microsoft C# MVP 2013
- Codeproject MVP 2013
- Microsoft C# MVP 2012
- Codeproject MVP 2012
- Microsoft C# MVP 2011
- Codeproject MVP 2011
- Microsoft C# MVP 2010
- Codeproject MVP 2010
- Microsoft C# MVP 2009
- Codeproject MVP 2009
- Microsoft C# MVP 2008
- Codeproject MVP 2008
- And numerous codeproject awards which you can see over at my blog

Comments and Discussions

help	**Member 11265993**	**26-Nov-14 22:**
Sigmoid function	**Member 10973839**	**26-Jul-14 23:**
Sigmoid function	**Member 10973839**	**26-Jul-14 23:**
Takes a long time to converge	**Member 9401821**	**2-Mar-14 0:**
Re: Takes a long time to converge	LudemeGames	2-Mar-14 4:
I have a question,thank you for telling me .	**fengyelan**	**16-Apr-13 22:**
My vote of 5	**Nickydo**	**10-Sep-12 2:**
Part 3?	**Mauro Leggieri**	**5-Apr-09 7:**
Re: Part 3?	Sacha Barber	5-Apr-09 9:
Re: Part 3?	Mauro Leggieri	6-Apr-09 3:
About parameterizing the 'momentum' factor	**mahabir**	**23-Sep-08 20:**
Re: About parameterizing the 'momentum' factor	Sacha Barber	23-Sep-08 22:
Re: About parameterizing the 'momentum' factor	Sacha Barber	23-Sep-08 22:
Re: About parameterizing the 'momentum' factor	ramesh0285	26-Nov-12 18:
part 1	**gholamabbas Sayyad**	**18-Sep-08 21:**
Re: part 1	Sacha Barber	18-Sep-08 22:
Solution for getTrainSet(int idx)	**DKHVC**	**16-Apr-08 21:**
[Message Deleted]	**Danny Rodriguez**	**27-Jan-08 10:**
Hello	**MohamadJaber**	**11-Dec-07 0:**
Erratic Bahaviour?	**rampantandroid**	**15-Oct-07 18:**
Re: Erratic Bahaviour?	rampantandroid	15-Oct-07 19:
Small Suggestion	**dfhgesart**	**28-Jul-07 16:**
Re: Small Suggestion	Sacha Barber	29-Jul-07 0:
Excellent!	**merlin981**	**17-May-07 5:**
license?	**famousj.dejazzd.com**	**17-Jan-07 9:**

Wednesday, December 30, 2015
3:26 PM

NEURAL NETWORKS

by Christos Stergiou and Dimitrios Siganos

Abstract

This report is an introduction to Artificial Neural Networks. The various types of neural networks are explained and demonstrated, applications of neural networks like ANNs in medicine are described, and a detailed historical background is provided. The connection between the artificial and the real thing is al investigated and explained. Finally, the mathematical models involved are presented and demonstrated.

Contents:

1. Introduction to neural networks

1.1 What is a Neural Network?

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, proces information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly

specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well.

1.2 Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least major setback and several eras.

Many importand advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few reserchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

For a more detailed description of the history click here

The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits. But the technology available at tha time did not allow them to do too much.

1.3 Why use neural networks?

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that ar too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyse. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which tak advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

1.4 Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. B computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements(neurones) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific tas The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is tha because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understa These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmi approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

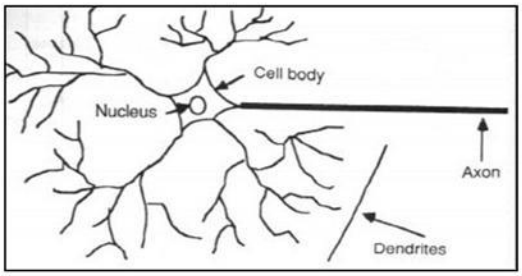
Neural networks do not perform miracles. But if used sensibly they can produce some amazing results.

Back to Contents

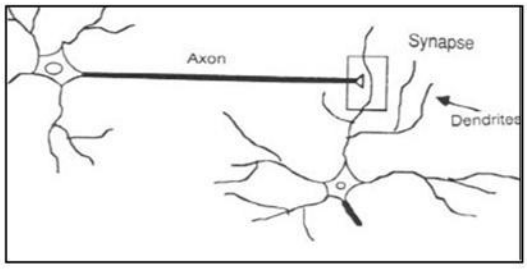
2. Human and Artificial Neurones - investigating the similarities

2.1 How the Human Brain Learns?

Much is still unknown about how the brain trains itself to process information, so theories abound. In the human brain, a typical neuron collects signals from others through a host of fine structures called *dendrites*. The neuron sends out spikes of electrical activity through a long, thin stand known as an *axon*, which splits into thousands of branches. At the end of each branch, a structure called a *synapse* converts the activity from the axon into electrical effects that inhibit excite activity from the axon into electrical effects that inhibit or excite activity in the connected neurones. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.



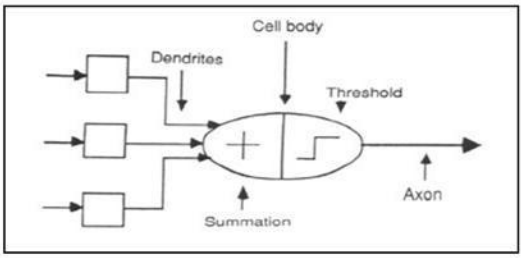
Components of a neuron



The synapse

2.2 From Human Neurones to Artificial Neurones

We conduct these neural networks by first trying to deduce the essential features of neurones and their interconnections. We then typically program a comput to simulate these features. However because our knowledge of neurones is incomplete and our computing power is limited, our models are necessarily gross idealisations of real networks of neurones.



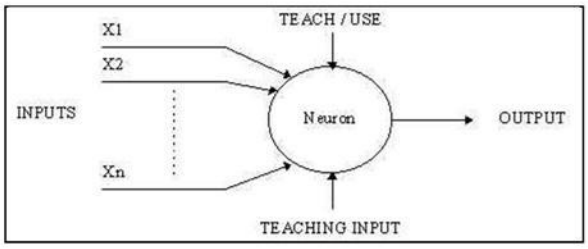
The neuron model

Back to Contents

3. An engineering approach

3.1 A simple neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the traini mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associ output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire o not.



A simple neuron

3.2 Firing rules

The firing rule is an important concept in neural networks and accounts for their high flexibility. A firing rule determines how one calculates whether a neuro should fire for any input pattern. It relates to all the input patterns, not only the ones on which the node was trained.

A simple firing rule can be implemented by using Hamming distance technique. The rule goes as follows:

> Take a collection of training patterns for a node, some of which cause it to fire (the 1-taught set of patterns) and others which prevent it from doing so (the 0-taught set). Then the patterns not in the collection cause the node to fire if, on comparison , they have more input elements in common with the 'nearest' pattern in the 1-taught set than with the 'nearest' pattern in the 0-taught set. If there is a tie, then the pattern remains in the undefined state.

For example, a 3-input neuron is taught to output 1 when the input (X1,X2 and X3) is 111 or 101 and to output 0 when the input is 000 or 001. Then, before

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0/1	0/1	0/1	1	0/1	1

As an example of the way the firing rule is applied, take the pattern 010. It differs from 000 in 1 element, from 001 in 2 elements, from 101 in 3 elements and from 111 in 2 elements. Therefore, the 'nearest' pattern is 000 which belongs in the 0-taught set. Thus the firing rule requires that the neuron should not fire w the input is 001. On the other hand, 011 is equally distant from two taught patterns that have different outputs and thus the output stays undefined (0/1).

By applying the firing in every column the following truth table is obtained;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0	0/1	0/1	1	1	1

The difference between the two truth tables is called the *generalisation of the neuron.* Therefore the firing rule gives the neuron a sense of similarity and enal it to respond 'sensibly' to patterns not seen during training.

3.3 Pattern Recognition - an example

An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward (figure 1) neural networ that has been trained accordingly. During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the i pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is gi as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

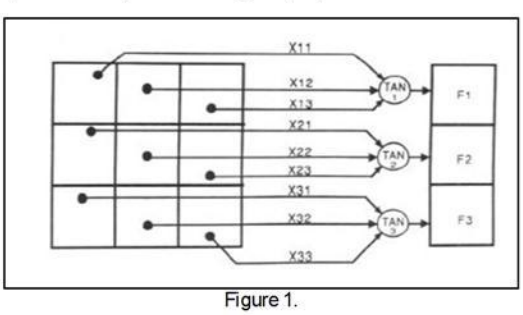


Figure 1.

For example:
The network of figure 1 is trained to recognise the patterns T and H. The associated patterns are all black and all white respectively as shown below.

If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurones after generalisation are;

X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Top neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

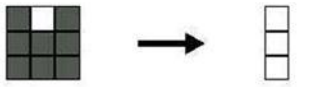
Middle neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1

Bottom neuron

From the tables it can be seen the following associasions can be extracted:

In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern.



Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.

Here, the top row is 2 errors away from the a T and 3 from an H. So the top output is black. The middle row is 1 error away from both T and H so the output random. The bottom row is 1 error away from T and 2 away from H. Therefore the output is black. The total output of the network is still in favour of the T shape.

3.4 A more complicated neuron

The previous neuron doesn't do anything that conventional conventional computers don't do already. A more sophisticated neuron (figure 2) is the McCulloc and Pitts model (MCP). The difference from the previous model is that the inputs are 'weighted', the effect that each input has at decision making is depende the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs then added together and if they exceed a pre-set threshold value, the neuron fires. In any other case the neuron does not fire.

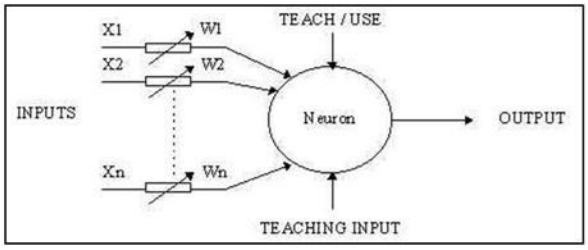


Figure 2. An MCP neuron

In mathematical terms, the neuron fires if and only if;

$$X_1W_1 + X_2W_2 + X_3W_3 + \ldots > T$$

The addition of input weights and of the threshold makes this neuron a very flexible and powerful one. The MCP neuron has the ability to adapt to a particul situation by changing its weights and/or threshold. Various algorithms exist that cause the neuron to 'adapt'; the most used ones are the Delta rule and the bac error propagation. The former is used in feed-forward networks and the latter in feedback networks.

Back to Contents

4 Architecture of neural networks

4.1 Feed-forward networks

Feed-forward ANNs (figure 1) allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not af that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition This type of organisation is also referred to as bottom-up or top-down.

4.2 Feedback networks

Feedback networks (figure 1) can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and ca get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations.

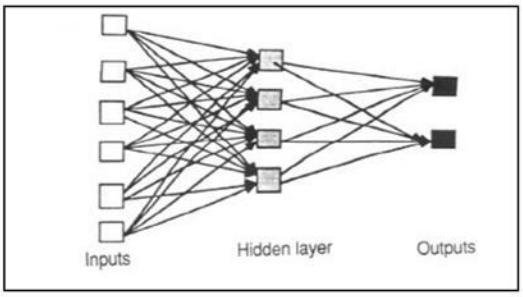


Figure 4.1 An example of a simple feedforward network

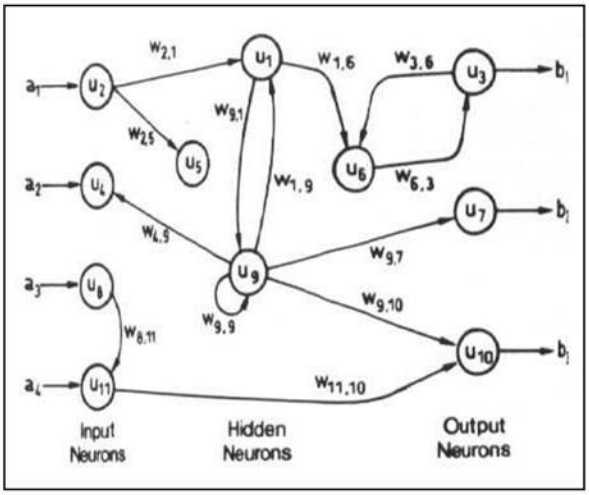


Figure 4.2 An example of a complicated network

4.3 Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units: a layer of "**input**" units is connected to a layer of "**hidden**" units which is connected to a layer of "**output**" units. (see Figure 4.1)

- The activity of the input units represents the raw information that is fed into the network.
- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish single-layer and multi-layer architectures. The single-layer organisation, in which all units are connected to one another, constitutes the r general case and is of more potential computational power than hierarchically structured multi-layer organisations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

4.4 Perceptrons

The most influential work on neural nets in the 60's went under the heading of 'perceptrons' a term coined by Frank Rosenblatt. The perceptron (figure 4.4) tu out to be an MCP model (neuron with weighted inputs) with some additional, fixed, pre--processing. Units labelled A1, A2, Aj , Ap are called association u and their task is to extract specific, localised featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They we mainly used in pattern recognition even though their capabilities extended a lot more.

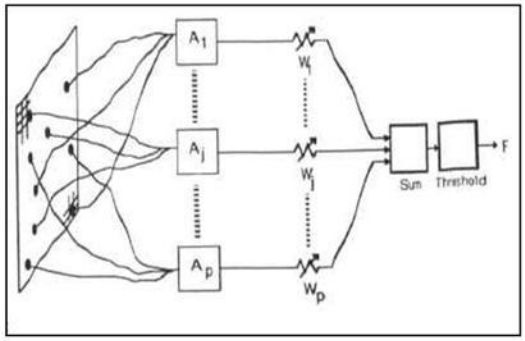


Figure 4.4

In 1969 Minsky and Papert wrote a book in which they described the limitations of single layer Perceptrons. The impact that the book had was tremendous a caused a lot of neural network researchers to loose their interest. The book was very well written and showed mathematically that *single layer* perceptrons co

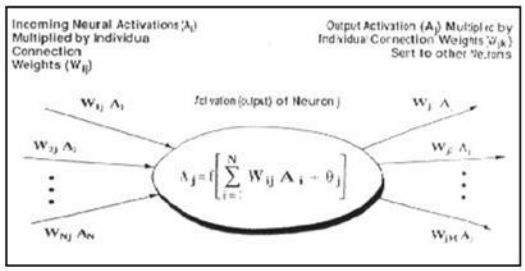
Back to Contents

5. The Learning Process

The memorisation of patterns and the subsequent response of the network can be categorised into two general paradigms:

- **associative mapping** in which the network learns to produce a particular pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associtive mapping can generally be broken down into two mechanisms:
  - *auto-association*: an input pattern is associated with itself and the states of input and output units coincide. This is used to provide pattern completition, ie to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network actually stores pairs of patterns building an association between two sets of patterns.
  - *hetero-association*: is related to two recall mechanisms:
    - *nearest-neighbour* recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and
    - *interpolative* recall, where the output pattern is a similarity dependent interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, ie when there is a fixed set of categories into which the input patterns are to be classified.
- **regularity detection** in which units learn to respond to particular properties of the input patterns. Whereas in asssociative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network posseses knowledge which is contained in the values of the connections weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights.



Information is stored in the weight matrix W of a neural network. Learning is the determination of the weights. Following the way learning is performed, we distinguish two major categories of neural networks:

- **fixed networks** in which the weights cannot be changed, ie dW/dt=0. In such networks, the weights are fixed a priori according to the problem to solve.
- **adaptive networks** which are able to change their weights, ie dW/dt not= 0.

All learning methods used for adaptive neural networks can be classified into two major categories:

- **Supervised learning** which incorporates an external teacher, so that each output unit is told what its desired response to input signals ought to be. During the learning process global information may be required. Paradigms of supervised learning include error-correction learning, reinforcement learning and stochastic learning.
  An important issue conserning supervised learning is the problem of error convergence, ie the minimisation of error between the desired and computed unit values. The aim is to determine a set of weights which minimises the error. One well-known method, which is common to many learning paradigms is the least mean square (LMS) convergence.
- **Unsupervised learning** uses no external teacher and is based upon only local information. It is also referred to as self-organisation, in the sense that it self-organises data presented to the network and detects their emergent collective properties. Paradigms of unsupervised learning are Hebbian lerning and competitive learning.
  Ano2.2 From Human Neurones to Artificial Neuronesther aspect of learning concerns the distinction or not of a seperate phase, during which the network is trained, and a subsequent operation phase. We say that a neural network learns off-line if the learning phase and the operation phase are distinct. A neural network learns on-line if it learns and operates at the same time. Usually, supervised learning is performed off-line, whereas usupervised learning is performed on-line.

5.1 Transfer Function

The behaviour of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- linear (or ramp)
- threshold
- sigmoid

For **linear units**, the output activity is proportional to the total weighted output.

For **threshold units**, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.

For **sigmoid units**, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurones than do lin or threshold units, but all three must be considered rough approximations.

To make a neural network that performs some specific task, we must choose how the units are connected to one another (see figure 4.1), and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strengt the influence.

We can teach a three-layer network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

5.2 An Example to illustrate the above teaching procedure:

Assume that we want a network to recognise hand-written digits. We might use an array of, say, 256 sensors, each recording the presence or absence of ink i small area of a single digit. The network would therefore need 256 input units (one for each sensor), 10 output units (one for each kind of digit) and a numbe hidden units.

For each kind of digit recorded by the sensors, the network should produce high activity in the appropriate output unit and low activity in the other output un

To train the network, we present an image of a digit and compare the actual activity of the 10 output units with the desired activity. We then calculate the err which is defined as the square of the difference between the actual and the desired activities. Next we change the weight of each connection so as to reduce t error.We repeat this training process for many different images of each different images of each kind of digit until the network classifies every image correct

To implement this procedure we need to calculate the error derivative for the weight (EW) in order to change the weight by an amount that is proportional to rate at which the error changes as the weight is changed. One way to calculate the EW is to perturb a weight slightly and observe how the error changes. But method is inefficient because it requires a separate perturbation for each of the many weights.

Another way to calculate the EW is to use the Back-propagation algorithm which is described below, and has become nowadays one of the most important to for training neural networks. It was developed independently by two teams, one (Fogelman-Soulie, Gallinari and Le Cun) in France, the other (Rumelhart, Hinton and Williams) in U.S.

5.3 The Back-Propagation Algorithm

In order to train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and t actual output is reduced. This process requires that the neural network compute the error derivative of the weights (**EW**). In other words, it must calculate ho the error changes as each weight is increased or decreased slightly. The back propagation algorithm is the most widely used method for determining the **EW**.

The back-propagation algorithm is easiest to understand if all the units in the network are linear. The algorithm computes each **EW** by first computing the **EA** the rate at which the error changes as the activity level of a unit is changed. For output units, the **EA** is simply the difference between the actual and the desir output. To compute the **EA** for a hidden unit in the layer just before the output layer, we first identify all the weights between that hidden unit and the output units to which it is connected. We then multiply those weights by the **EA**s of those output units and add the products. This sum equals the **EA** for the chosen hidden unit. After calculating all the **EA**s in the hidden layer just before the output layer, we can compute in like fashion the **EA**s for other layers, moving fr layer to layer in a direction opposite to the way activities propagate through the network. This is what gives back propagation its name. Once the **EA** has bee computed for a unit, it is straight forward to compute the **EW** for each incoming connection of the unit. The **EW** is the product of the EA and the activity through the incoming connection.

Note that for non-linear units, (see Appendix C) the back-propagation algorithm includes an extra step. Before back-propagating, the **EA** must be converted i the **EI**, the rate at which the error changes as the total input received by a unit is changed.

Back to Contents

6. Applications of neural networks

6.1 Neural Networks in Practice

Given this description of neural networks and how they work, what real world applications are they suited for? Neural networks have broad applicability to r world business problems. In fact, they have already been successfully applied in many industries.

Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs including:

- sales forecasting
- industrial process control
- customer research
- data validation
- risk management
- target marketing

But to give you some more specific examples; ANN are also used in the following specific paradigms: recognition of speakers in communications; diagnosis hepatitis; recovery of telecommunications from faulty software; interpretation of multimeaning Chinese words; undersea mine detection; texture analysis; thr dimensional object recognition; hand-written word recognition; and facial recognition.

6.2 Neural networks in medicine

Artificial Neural Networks (ANN) are currently a 'hot' research area in medicine and it is believed that they will receive extensive application to biomedical systems in the next few years. At the moment, the research is mostly on modelling parts of the human body and recognising diseases from various scans (e.g cardiograms, CAT scans, ultrasonic scans, etc.).

Neural networks are ideal in recognising diseases using scans since there is no need to provide a specific algorithm on how to identify the disease. Neural networks learn by example so the details of how to recognise the disease are not needed. What is needed is a set of examples that are representative of all the variations of the disease. The quantity of examples is not as important as the 'quantity'. The examples need to be selected very carefully if the system is to perform reliably and efficiently.

6.2.1 Modelling and Diagnosing the Cardiovascular System

Neural Networks are used experimentally to model the human cardiovascular system. Diagnosis can be achieved by building a model of the cardiovascular system of an individual and comparing it with the real time physiological measurements taken from the patient. If this routine is carried out regularly, potenti harmful medical conditions can be detected at an early stage and thus make the process of combating the disease much easier.

A model of an individual's cardiovascular system must mimic the relationship among physiological variables (i.e., heart rate, systolic and diastolic blood pressures, and breathing rate) at different physical activity levels. If a model is adapted to an individual, then it becomes a model of the physical condition of individual. The simulator will have to be able to adapt to the features of any individual without the supervision of an expert. This calls for a neural network.

Another reason that justifies the use of ANN technology, is the ability of ANNs to provide sensor fusion which is the combining of values from several diffe sensors. Sensor fusion enables the ANNs to learn complex relationships among the individual sensor values, which would otherwise be lost if the values wer individually analysed. In medical modelling and diagnosis, this implies that even though each sensor in a set may be sensitive only to a specific physiologica variable, ANNs are capable of detecting complex medical conditions by fusing the data from the individual biomedical sensors.

6.2.2 Electronic noses

ANNs are used experimentally to implement electronic noses. Electronic noses have several potential applications in telemedicine. Telemedicine is the practi of medicine over long distances via a communication link. The electronic nose would identify odours in the remote surgical environment. These identified odours would then be electronically transmitted to another site where an door generation system would recreate them. Because the sense of smell can be an important sense to the surgeon, telesmell would enhance telepresent surgery.
For more information on telemedicine and telepresent surgery click here.

6.2.3 Instant Physician

An application developed in the mid-1980s called the "instant physician" trained an autoassociative memory neural network to store a large number of medic records, each of which includes information on symptoms, diagnosis, and treatment for a particular case. After training, the net can be presented with input consisting of a set of symptoms; it will then find the full stored pattern that represents the "best" diagnosis and treatment.

6.3 Neural Networks in business

Business is a diverted field with several general areas of specialisation such as accounting or financial analysis. Almost any neural network application would into one business area or financial analysis.
There is some potential for using neural networks for business purposes, including resource allocation and scheduling. There is also a strong potential for usi neural networks for database mining, that is, searching for patterns implicit within the explicitly stored information in databases. Most of the funded work in area is classified as proprietary. Thus, it is not possible to report on the full extent of the work going on. Most work is applying neural networks, such as the Hopfield-Tank network for optimization and scheduling.

6.3.1 Marketing

There is a marketing application which has been integrated with a neural network system. The Airline Marketing Tactician (a trademark abbreviated as AMT a computer system made of various intelligent technologies including expert systems. A feedforward neural network is integrated with the AMT and was trai using back-propagation to assist the marketing control of airline seat allocations. The adaptive neural approach was amenable to rule expression. Additionaly application's environment changed rapidly and constantly, which required a continuously adaptive solution. The system is used to monitor and recommend

While it is significant that neural networks have been applied to this problem, it is also important to see that this intelligent technology can be integrated with expert systems and other approaches to make a functional system. Neural networks were used to discover the influence of undefined interactions by the vario variables. While these interactions were not defined, they were used by the neural system to develop useful conclusions. It is also noteworthy to see that neur networks can influence the bottom line.

6.3.2 Credit Evaluation

The HNC company, founded by Robert Hecht-Nielsen, has developed several neural network applications. One of them is the Credit Scoring system which increase the profitability of the existing model up to 27%. The HNC neural systems were also applied to mortgage screening. A neural network automated mortgage insurance underwritting system was developed by the Nestor Company. This system was trained with 5048 applications of which 2597 were certifi The data related to property and borrower qualifications. In a conservative mode the system agreed on the underwritters on 97% of the cases. In the liberal m the system agreed 84% of the cases. This is system run on an Apollo DN3000 and used 250K memory while processing a case file in approximately 1 sec.

Back to Contents

7. Conclusion

The computing world has a lot to gain fron neural networks. Their ability to learn by example makes them very flexible and powerful. Furthermore there is n need to devise an algorithm in order to perform a specific task; i.e. there is no need to understand the internal mechanisms of that task. They are also very we suited for real time systems because of their fast responseand computational times which are due to their parallel architecture.

Neural networks also contribute to other areas of research such as neurology and psychology. They are regularly used to model parts of living organisms and investigate the internal mechanisms of the brain.

Perhaps the most exciting aspect of neural networks is the possibility that some day 'consious' networks might be produced. There is a number of scientists arguing that conciousness is a 'mechanical' property and that 'consious' neural networks are a realistic possibility.

Finally, I would like to state that even though neural networks have a huge potential we will only get the best of them when they are intergrated with comput AI, fuzzy logic and related subjects.

Back to Contents

References:

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html
3. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
4. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html
5. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht
6. Neural Networks by Eric Davalo and Patrick Naim
7. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
8. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
9. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab. Neural Networks, Eric Davalo and Patrick Naim
10. Assimov, I (1984, 1950), Robot, Ballatine, New York.
11. Electronic Noses for Telemedicine
    http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html
12. Pattern Recognition of Pathology Images
    http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html

Back to Contents

Appendix A - Historical background in detail

The history of neural networks that was described above can be divided into several periods:

1. **First Attempts**: There were some initial simulations using formal logic. McCulloch and Pitts (1943) developed models of neural networks based on th understanding of neurology. These models made several assumptions about how neurons worked. Their networks were based on simple neurons which were considered to be binary devices with fixed thresholds. The results of their model were simple logic functions such as "a or b" and "a and b". Anot attempt was by using computer simulations. Two groups (Farley and Clark, 1954; Rochester, Holland, Haibit and Duda, 1956). The first group (IBM reserchers) maintained closed contact with neuroscientists at McGill University. So whenever their models did not work, they consulted the neuroscientists. This interaction established a multidiscilinary trend which continues to the present day.
2. **Promising & Emerging Technology:** Not only was neroscience influential in the development of neural networks, but psychologists and engineers al contributed to the progress of neural network simulations. Rosenblatt (1958) stirred considerable interest and activity in the field when he designed and developed the Perceptron. The Perceptron had three layers with the middle layer known as the association layer.This system could learn to connect or

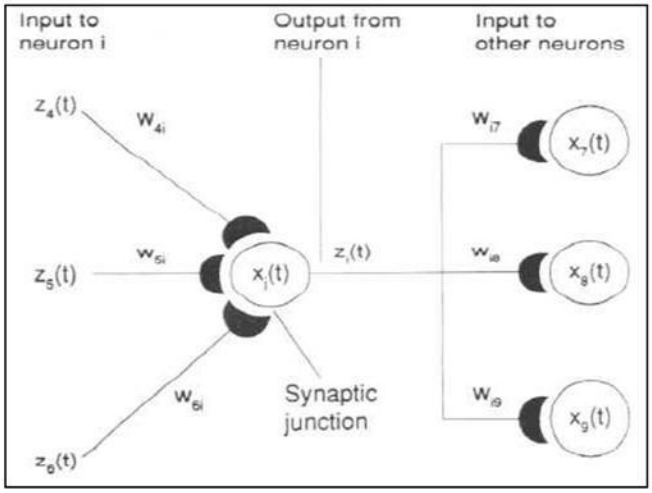
Another system was the ADALINE (ADAptive LInear Element) which was developed in 1960 by Widrow and Hoff (of Stanford University). The ADALINE was an analogue electronic device made from simple components. The method used for learning was different to that of the Perceptron, it employed the Least-Mean-Squares (LMS) learning rule.

3. **Period of Frustration & Disrepute:** In 1969 Minsky and Papert wrote a book in which they generalised the limitations of single layer Perceptrons to multilayered systems. In the book they said: "...our intuitive judgment that the extension (to multilayer systems) is sterile". The significant result of the book was to eliminate funding for research with neural network simulations. The conclusions supported the disenhantment of reserchers in the field. A result, considerable prejudice against this field was activated.
4. **Innovation:** Although public interest and available funding were minimal, several researchers continued working to develop neuromorphically based computaional methods for problems such as pattern recognition.
   During this period several paradigms were generated which modern work continues to enhance.Grossberg's (Steve Grossberg and Gail Carpenter in 19 influence founded a school of thought which explores resonating algorithms. They developed the ART (Adaptive Resonance Theory) networks based o biologically plausible models. Anderson and Kohonen developed associative techniques independent of each other. Klopf (A. Henry Klopf) in 1972, developed a basis for learning in artificial neurons based on a biological principle for neuronal learning called heterostasis.
   Werbos (Paul Werbos 1974) developed and used the back-propagation learning method, however several years passed before this approach was popularized. Back-propagation nets are probably the most well known and widely applied of the neural networks today. In essence, the back-propagati net. is a Perceptron with multiple layers, a different thershold function in the artificial neuron, and a more robust and capable learning rule.
   Amari (A. Shun-Ichi 1967) was involved with theoretical developments: he published a paper which established a mathematical theory for a learning b (error-correction method) dealing with adaptive patern classification. While Fukushima (F. Kunihiko) developed a step wise trained multilayered neura network for interpretation of handwritten characters. The original network was published in 1975 and was called the Cognitron.
5. **Re-Emergence:** Progress during the late 1970s and early 1980s was important to the re-emergence on interest in the neural network field. Several fact influenced this movement. For example, comprehensive books and conferences provided a forum for people in diverse fields with specialized technica languages, and the response to conferences and publications was quite positive. The news media picked up on the increased activity and tutorials helpe disseminate the technology. Academic programs appeared and courses were inroduced at most major Universities (in US and Europe). Attention is nov focused on funding levels throughout Europe, Japan and the US and as this funding becomes available, several new commercial with applications in industry and finacial institutions are emerging.
6. **Today:** Significant progress has been made in the field of neural networks-enough to attract a great deal of attention and fund further research. Advancement beyond current commercial applications appears to be possible, and research is advancing the field on many fronts. Neurally based chips emerging and applications to complex problems developing. Clearly, today is a period of transition for neural network technology.

Back to Contents

Appendix B - The back-propagation Algorithm - a mathematical approach

Units are connected to one another. Connections correspond to the edges of the underlying directed graph. There is a real number associated with each connection, which is called the weight of the connection. We denote by W_{ij} the weight of the connection from unit u_i to unit u_j. It is then convenient to repres the pattern of connectivity in the network by a weight matrix W whose elements are the weights W_{ij}. Two types of connection are usually distinguished: excitatory and inhibitory. A positive weight represents an excitatory connection whereas a negative weight represents an inhibitory connection. The pattern o connectivity characterises the architecture of the network.



A unit in the output layer determines its activity by following a two step procedure.

- First, it computes the total weighted input x_j, using the formula:

$$X_j = \sum_i y_i W_{ij}$$

  where y_i is the activity level of the jth unit in the previous layer and W_{ij} is the weight of the connection between the ith and the jth unit.

- Next, the unit calculates the activity y_j using some function of the total weighted input. Typically we use the sigmoid function:

$$y_j = \frac{1}{1 + e^{-x_j}}$$

Once the activities of all output units have been determined, the network computes the error E, which is defined by the expression:

$$E = \frac{1}{2}\sum_i (y_i - d_i)^2$$

where y_j is the activity level of the jth unit in the top layer and d_j is the desired output of the jth unit.

The back-propagation algorithm consists of four steps:

1. Compute how fast the error changes as the activity of an output unit is changed. This error derivative (EA) is the difference between the actual and the des activity.

$$EA_j = \frac{\partial E}{\partial y_j} = y_j - d_j$$

2. Compute how fast the error changes as the total input received by an output unit is changed. This quantity (EI) is the answer from step 1 multiplied by the at which the output of a unit changes as its total input is changed.

$$EI_j = \frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \times \frac{dy_j}{dx_j} = EA_j y_j (1 - y_j)$$

3. Compute how fast the error changes as a weight on the connection into an output unit is changed. This quantity (EW) is the answer from step 2 multiplied the activity level of the unit from which the connection emanates.

$$EW_{ij} = \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial W_{ij}} = EI_j y_i$$

4. Compute how fast the error changes as the activity of a unit in the previous layer is changed. This crucial step allows back propagation to be applied to multilayer networks. When the activity of a unit in the previous layer changes, it affects the activites of all the output units to which it is connected. So to compute the overall effect on the error, we add together all these seperate effects on output units. But each effect is simple to calculate. It is the answer in ste multiplied by the weight on the connection to that output unit.

$$EA_i = \frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial y_i} = \sum_j EI_j W_{ij}$$

By using steps 2 and 4, we can convert the EAs of one layer of units into EAs for the previous layer. This procedure can be repeated to get the EAs for as ma previous layers as desired. Once we know the EA of a unit, we can use steps 2 and 3 to compute the EWs on its incoming connections.

Back to Contents

Appendix C - References used throughout the review

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html
3. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht
4. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
5. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html
6. Electronic Noses for Telemedicine
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html
7. An Introduction to Computing with Neural Nets (Richard P. Lipmann, IEEE ASSP Magazine, April 1987)
8. Pattern Recognition of Pathology Images
   http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html
9. Developments in autonomous vehicle navigation. Stefan Neuber, Jos Nijhuis, Lambert Spaanenburg. Institut fur Mikroelektronik Stuttgart, Allmandrin 30A, 7000 Stuttgart-80
10. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
11. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab.
12. Neural Networks, Eric Davalo and Patrick Naim.
13. Assimov, I (1984, 1950), Robot, Ballatine, New York.
14. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
15. Alkon, D.L 1989, Memory Storage and Neural Systems, Scientific American, July, 42-50
16. Minsky and Papert (1969) Perceptrons, An introduction to computational geometry, MIT press, expanded edition.
17. Neural computers, NATO ASI series, Editors: Rolf Eckmiller Christoph v. d. Malsburg

Back to Contents

Friday, January 01, 2016
1:28 AM

C#

Succinctly

by Joe Mayo

Syncfusion | Technology Resource Portal

C# Succinctly

By
Joe Mayo

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Stephen Haunts
Copy Editor: Ben Ball
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Joe Mayo is an author, a consultant at Mayo Software, LLC, and an instructor who specializes in Microsoft .NET technology. Joe has written several books, including *C# Unleashed* (Sams) and *LINQ Programming* (McGraw-Hill), and coauthored *ASP.NET 2.0 MVP Hacks and Tips* (Wrox). His articles have been published in CODE Magazine and the online publications Inform IT and C# Station.

Joe is a regular presenter on .NET topics and has received multiple Microsoft Visual C# MVP awards. His open source project, LINQ to Twitter, is hosted on GitHub, and you can read his blog at Geeks with Blogs. You can find Joe on Twitter as @JoeMayo.

Chapter 1 Introducing C# and .NET

Welcome to *C# Succinctly*. True to the *Succinctly* series concept, this book is very focused on a single topic: the C# programming language. I might briefly mention some technologies that you can write with C# or explain how a feature fits into those technologies, but the whole of this book is about helping you become familiar with C# syntax.

In this chapter, I'll start with some introductory information and then jump straight into a simple C# program.

What can I do with C#?

C# is a general purpose, object-oriented, component-based programming language. As a general purpose language, you have a number of ways to apply C# to accomplish many different tasks. You can build web applications with ASP.NET, desktop applications with Windows Presentation Foundation (WPF), or build mobile applications for Windows Phone. Other applications include code that runs in the cloud via Windows Azure, and iOS, Android, and Windows Phone support with the Xamarin platform. There might be times when you need a different language, like C or C++, to communicate with hardware or real-time systems. However, from a general programming perspective, you can do a lot with C#.

What is .NET?

.NET is a platform that includes languages, a runtime, and framework libraries, allowing developers to create many types of applications. C# is one of the .NET languages, which also includes Visual Basic, F#, C++, and more.

The runtime is more formally named the Common Language Runtime (CLR). Programming languages that target the CLR compile to an Intermediate Language (IL). The CLR itself is a virtual machine that runs IL and provides many services such as memory management, garbage collection, exception management, security, and more.

The Framework Class Library (FCL) is a set of reusable code that provides both general services and technology-specific platforms. The general services include essential types such as collections, cryptography, networking, and more. In addition to general classes, the FCL includes technology-specific platforms like ASP.NET, WPF, web services, and more. The value the FCL offers is to have common components available for reuse, saving time and money without needing to write that code yourself.

There's a huge ecosystem of open-source and commercial software that relies on and supports .NET. If you visit CodePlex, GitHub, or any other open-source code repository site, you'll see a multitude of projects written in C#. Commercial offerings include tools and services that help you build code, manage systems, and offer applications. Syncfusion is part of this ecosystem, offering reusable components for many of the .NET technologies I have mentioned.

Writing, Running, and Deploying a C# Program

The previous section described plenty of great things you can do with C#, but most of them are so detailed that they require their own book. To stay focused on the C# programming language, the code in this book will be for the console application. A console application runs on the command line, which you'll learn about in this section. You can write your code with any editor, but this book uses Visual Studio.

Note: The code samples in this book can be downloaded at https://bitbucket.org/syncfusiontech/c-succinctly.

Starting a New Program

You'll need an editor or Integrated Development Environment (IDE) to write code. Microsoft offers Visual Studio (VS), which is available via Community Edition as a free download for training and individual purposes (https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx). There are other development tools, but you can also use any editor, including Notepad. Notepad++ is another editor that does syntax highlighting, but there are many more available. Essentially, you just need the ability to type a text document. Pick your editor or IDE of choice and it will work for all programs in this book.

Note: You need to use Visual Studio 2015 to compile the samples in this book.

To get started, we need a program to run. In VS, select **File** > **New** > **Project**, then select **Installed** > **Templates** > **Visual C#** in the tree on the left, and finally select the **Console Application** project type. Name the solution **Chapter01**, name the project **Greetings**, set the location to your preference, and click **OK**. This will create a new solution for you. Delete the **Program.cs** file and add a **Greetings.cs** file. In any text editor, just create a file named **Greetings.cs**. The following is a C# program that prints a greeting to the command line.

```
using System;

class Greetings
{
    static void Main()
    {
        Console.WriteLine("Greetings!");
    }
}
```

Code Listing 1

The **class** is a container for code, defining a type, named **Greetings**. A **class** has members and this example shows a method member named **Main**. A method is similar to functions and

procedures in other programming languages. For desktop application types, like console or WPF, naming a method **`Main`** tells the C# compiler where the program begins executing. Both the **`Greetings`** class and **`Main`** method have curly braces, referred to as a block, indicating beginning and ending scope.

The **`void`** keyword isn't a type; it indicates that a method does not return a value. For **`Main`**, you can replace **`void`** with **`int`**, meaning that the program has a return code. This number can be used by command-line shell tools to evaluate the conditions under which the program ended. It is unique to each program and specified by you. Later, you'll learn more about methods and return values.

The **`static`** modifier indicates that there is only ever one instance of a **`Greetings`** class that has that **`Main`** method—it is the **`static`** instance. **`Main`** must be **`static`**, but other methods can omit **`static`**, which makes them instance members. This means that you can have many copies of a class or instance with their own method.

Since a program only needs a single **`Main`** method, **`static`** makes sense. A program that manages customers might have a **`Customer`** class and you would need multiple instances to represent each **`Customer`**. You'll see examples of instantiating classes in later chapters of this book.

Inside the **`Main`** method is a statement that prints words to the command line. The words, enclosed in double quotes, are a string. That string is passed to the **`WriteLine`** method, which writes that string to the command line and causes it to move to the next line. **`WriteLine`** is a method that belongs to a class named **`Console`**. You see in this example, just like the **`Greetings`** class, the **`Console`** is a class too. This **`Console`** class belongs to the **`System`** namespace, which is why the **`using`** clause appears at the top of the file, allowing us to use that **`Console`** class.

The code begins with a **`using`** clause for the **`System`** namespace. The FCL is grouped into namespaces to keep code organized and avoid clashes between identically named types. This **`using`** clause allows us to use the code in the **`System`** namespace, which we're doing with the **`Console`** class. Without that, the compiler doesn't know what **`Console`** means or how to find it, but now C# knows that we're using the **`System.Console`** class.

Namespaces and Code Organization

There are various ways to organize code and the choice should be based on the standards of your team and the nature of the project you're building. One of the common ways to organize code is with the C# namespace feature. Here's a hierarchical description of where namespaces fit into the overall structure of a program:

- Namespace
  - Type
    - Type Members

Out of this hierarchy, the namespace is optional, as demonstrated in the previous program where the **`Greetings`** class was not contained in a namespace. This means **`Greetings`** is a

member of the **global** namespace. You should avoid this practice as it opens the possibility for other developers working with your code to write their own **Greetings** class in the same namespace, which will cause errors because the C# compiler can't figure out which **Greetings** class to use. While **Greetings** might seem unique and unlikely, think of common names, such as **File**, **Math**, or **Window**, that would cause problems. The following program uses namespaces appropriately.

```
using static System.Math;

namespace Syncfusion
{
    public class Calc
    {
        public static double Pythagorean(double a, double b)
        {
            double cSquared = Pow(a, 2) + Pow(b, 2);
            return Sqrt(cSquared);
        }
    }
}
```

Code Listing 2

The **Calc** class is a member of the **Syncfusion** namespace. The **Pythagorean** method is a member of the **Calc** class. A method is a block of code with a name, parameters, and return value that you can call from other code. This follows the namespace, class, member organization.

System is a namespace in the FCL and **Math** is a class in the **System** namespace. The **using static** clause allows the code to use static members of the **Math** class without full qualification. Instead of writing **Math.Pow(a, 2)**, which squares the value of **a**, you can use the shorthand syntax in the **Pythagorean** method. The **Pythagorean** method uses **Math.Sqrt**, which provides square root, similarly. The following sample shows how you can use this code.

```
using Syncfusion;
using System;

using Crypto = System.Security.Cryptography;

namespace NamespaceDemo
{
    class Program
    {
        static void Main()
        {
            double hypotenuse = Calc.Pythagorean(2, 3);
            Console.WriteLine("Hypotenuse: " + hypotenuse);

            Crypto.AesManaged aes = new Crypto.AesManaged();

            Console.ReadKey();
```

```
        }
    }
}
```

Code Listing 3

The **Main** method calls the **Pythagorean** method of the **Calc** class, passing arguments **2** and **3** and receiving a result in **hypotenuse**. Since **Calc** is in the **Syncfusion** namespace, the code adds a **using** clause for **Syncfusion** to the top of the file. Had the code not included that **using** clause, **Main** would have been required to use the fully qualified name, **Syncfusion.Calc.Pythagorean**.

Another feature of the previous program is the namespace alias, **Crypto**. This syntax allows you to use a shorthand syntax when you need to fully qualify a namespace, but want to reduce syntax in your code. If there had been another **Cryptography** namespace used in the same code, though not in this listing, full qualification would have been necessary. **Crypto** is the alias for **System.Security.Cryptography** and **Main** uses that alias in **Crypto.AesManaged** to make the code more readable.

Running the Program

The rest of this chapter returns to the previous Greetings program in this chapter.

Now the program is written and you want to continue by compiling the program and running it. You'll want to save this file with the name **Greetings.cs**. The name isn't necessarily important, but by convention should be meaningful and is often the same name as a class it contains. You're allowed to put multiple classes in the same file, but it's easier to find a class later if it is alone in its own file of the same name. C# files have a .cs extension.

In VS, click the green **Start** arrow on the toolbar and it will build and run the program. The program runs and stops so quickly that you won't see the command-line output, so you can press **Ctrl + F5** to make the command line stay open. This book uses Visual Studio 2015, but Syncfusion has published *Visual Studio 2013 Succinctly*, which explains many features that are still valid in Visual Studio 2015. In the meantime, I'm going to show you how to use the C# compiler directly—the benefit being that you see what the IDE is doing for you.

Tip: Adding `Console.ReadKey();` as the last line in `Main` makes the command line stop and wait for a key press.

Minimally, you need the .NET Framework installed on your machine, which is free for commercial as well as non-commercial use. If you installed VS, you already have the .NET Framework. Otherwise, download it from http://www.microsoft.com/en-us/download/details.aspx?id=30653 and install it. This link is for .NET Framework 4.5, but any future version should work fine.

Once .NET is installed, open Windows Explorer and do a search for the C# compiler, **csc.exe**. Since I'm using Visual Studio 2015 for the examples in this book, the C# 6 compiler on my machine is located at **C:\Program Files (x86)\MSBuild\14.0\Bin**, but yours may differ.

Next, make sure the C# compiler is in your path. Open your **System Properties** window. As of this writing, I'm on Windows 8.1 and found it via selecting **Control Panel** > **System and Security** > **System**, and then clicking **Advanced System Settings**. Select the **Advanced** tab and click the **Environment Variables** button. In the **System variables** list, select **Path**, and click **Edit**. You should see several paths separated by semicolons. At the end of that path, add your C# compiler's path that you found with the Windows Explorer search and make sure it's separated from the previous paths with a semicolon. Close out of all these windows when you're done setting the path.

Now that you have the .NET Framework installed and have the path to the C# compiler set, you can build the program that you typed in the previous example. First, open a command prompt window. On my system, I can do this by pressing the **Windows logo key + R**, typing **cmd.exe** in the **Run** dialog, and then clicking **OK**. If you've never used a command line, it's a good idea to open your favorite search engine and look for a tutorial. Alternatively, it might be good to learn PowerShell; Syncfusion has a book on it titled *PowerShell Succinctly*. In the meantime, navigate to the directory where you saved **Greetings.cs**. You can type **cd\your\path\there** and press **Enter** to get there. You can verify you're in the right location by typing **dir** to see what files reside in the current directory.

To compile the program, type **csc Greetings.cs**. If you see compiler errors, go back to Code Listing 1 and make sure you've typed the code exactly as it is there and then recompile.

Tip: Use a space separated list to compile multiple files; e.g., *csc.exe file1.cs file2.cs****. For C# compiler help, type*** *csc.exe /help****.***

Now type **dir** and you'll see a new file named **Greetings.exe**. This is an executable assembly. In .NET, an assembly is a unit of identity, execution, and deployment, which is why it's not just called a file. For the purposes of this book, you won't be involved with all the nuances of assemblies, but it's an encompassing term that includes both executable (.exe) and library (.dll) files.

Now type **Greetings.exe** and press **Enter**. The program will print **`Greetings!`** on the command line. Then you'll see a new command-line prompt, meaning that the program ended. This came from the **`Console.WriteLine`** statement in the **`Main`** method. When the **`Main`** method finishes executing, the program finishes too.

Deploying the Program

.NET uses XCopy deployment, which means that you only need to copy the assembly to anywhere you want it to go. The one caveat is that whatever machine you run the program on must also have the .NET CLR installed. Installing VS or the .NET Framework automatically installs the CLR. Also, you can only install the .NET Framework Runtime, which doesn't include development tools, to a machine where you only want to run a C# program but not perform any

development tasks. In practical terms, most Windows systems already have .NET installed from the original installation and it is kept up-to-date via Windows Update.

Whenever you run the program, Windows looks at the executable, determines that it's a .NET assembly, loads the CLR, and then gives that assembly to the CLR to run. From the users' perspective, the CLR behavior is behind the scenes; the program appears like any other program when they run the executable.

Summary

This chapter included a couple broader takeaways regarding how C# fits into the .NET Framework ecosystem and how to create a C# program. Remember that C# is a programming language, but it builds programs that use the FCL to run applications managed by the CLR. What this gives you is the ability to compile programs into assemblies that can be deployed and run on any machine that supports the CLR. The program entry point is the **`Main`** method. You can use any editor or an IDE like Visual Studio to write your code. To run a program, press **F5** in VS or compile with **`csc.exe`** on the command line. To deploy, copy the program to a machine with the CLR installed. In the next chapter, you'll learn more about how to code logic in C# using expressions and statements.

Chapter 2 Coding Expressions and Statements

In Chapter 1, you saw how to write, compile, and execute a C# program. The example program had a single statement in the **Main** method. In this chapter, you'll learn how to write more statements and add logic to your program. For efficiency, many of the examples in the rest of the book are snippets, but you can still add these statements inside of a **Main** method to compile and get a better feel for C# syntax. There will be plenty of complete programs too.

Writing Simple Statements

By combining language operators and syntax, you can build expressions and statements in C#. Here are a few examples of simple C# statements.

```
int count = 7;
char keyPressed = 'Q';
string title = "Weekly Report";
```

Code Listing 4

Each of the examples in the previous code listing have common syntactical elements: type, variable identifier, assignment operator, value, and statement completion. The types are **int**, **char**, and **string**, which represent a number, a character, and a sequence of characters respectively. These are a few of the several built-in types that C# offers. Variables are a name that can be used in later code. The = operator assigns the right-hand side of the expression to the left-hand side. Each statement ends with a semicolon.

The previous example showed how to declare a variable and perform assignment at the same time, but that isn't necessarily required. As long as you declare a variable before trying to use it, you'll be okay. Here's a separate declaration.

```
string title;
```

Code Listing 5

And the variable's later assignment.

```
title = "Weekly Report";
```

Code Listing 6

Note: C# is case sensitive, so "title" and "Title" are two separate variable names.

Overview of C# Types and Operators

C# is a strongly typed language, meaning that the compiler won't implicitly convert between incompatible types. For example, you can't assign a **`string`** to an **`int`** or an **`int`** to a **`string`**—at least, not implicitly. The following code will not compile.

```
int total = "359";
string message = 7;
```

Code Listing 7

The "**`359`**" with double quotes is a string, and the **`7`** without quotes is an **`int`**. While you can't perform conversions implicitly, there are ways to do this explicitly. For example, you'll often receive text input from a user that should be an **`int`** or another type. The following code listing shows a couple examples of how to perform such tasks explicitly.

```
int total = int.Parse("359");
string message = 7.ToString();
```

Code Listing 8

In the previous listing, **`Parse`** will convert the string to an **`int`** if the string represents a valid **`int`**. Calling **`ToString`** on any value will always produce a **`string`** that will compile.

In addition to the previous conversion examples, C# has a cast operator that lets you convert between types that allow explicit conversions. Let's say you have a **`double`**, which is a 64-bit floating point type, and want to assign that to an **`int`**, which is a 32-bit whole number. You could cast it like this:

```
double preciseLength = 5.61;
int roundedLength = (int)preciseLength;
```

Code Listing 9

Without the cast operator, you would receive a compiler error because a **`double`** is not an **`int`**. Essentially, the C# compiler is protecting you from shooting yourself in the foot because assigning a **`double`** to an **`int`** means that you lose precision. In the previous example, **`roundedLength`** becomes **`5`**. Using the cast operator allows you to tell the C# compiler that you know this operation could be dangerous in the wrong circumstances, but makes sense for your particular situation.

The following table lists the built-in types so you can see what is available:

Table 1: Built-In Types

Type (Literal Suffix)	Description	Values/Range
byte	8-bit unsigned integer	0 to 255
sbyte	8-bit signed integer	-128 to 127
short	16-bit signed integer	-32,768 to 32,767
ushort	16-bit unsigned integer	0 to 65,535
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647
uint	32-bit unsigned integer	0 to 4,294,967,295
long (l)	64-bit signed integer	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong (ul)	64-bit unsigned integer	0 to 18,446,744,073,709,551,615
float (f)	32-bit floating point	-3.4×10^{38} to $+3.4 \times 10^{38}$
double (d)	64-bit floating point	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
decimal (m)	128-bit, 28 or 29 digits of precision (ideal for financial)	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / (10^{0 \text{ to } 28})$
bool	Boolean	true or false
char	16-bit Unicode character (use single quotes)	U+0000 to U+FFFF
string	Sequence of Unicode characters (use double quotes)	E.g., “abc”

You should add a suffix to a number when the meaning would be ambiguous. In the following example, the **`m`** suffix ensures the **`9.95`** literal is treated as a **`decimal`** number:

```
decimal price = 9.95m;
```

Code Listing 10

You can assign Unicode values directly to a char. The following example shows how to assign a carriage return.

```
char cr = '\u0013';
```

Code Listing 11

You can also obtain the Unicode value of a character with a cast operator as shown here.

```
int crUnicode = (int)cr;
```

Code Listing 12

So far, you've only seen statements with the assignment operator, but C# has many other operators that allow you to perform all of the logical operations you would expect of any general purpose programming language. The following table lists some of the available operators.

Table 2: C# Operators

Category	Description
Primary	`x.y  x?.y  f(x)  a[x]  x++  x--  new  typeof  default  checked  unchecked  nameof`
Unary	`+  -  !  ~  ++x  --x  (T)x  await x`
Multiplicative	`*  /  %`
Additive	`+  -`
Shift	`<<  >>`
Relational and Type Testing	`<  >  <=  >=  is  as`
Equality	`==  !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Null Coalescing	`??`
Conditional	`?:`
Assignment	`=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=  =>`

Prefix operators change the value of the variable before assignment, and postfix operators change a variable after assignment, as demonstrated in the following sample.

```
int val1 = 5;
int val2 = ++val1;
int val3 = 2;
int val4 = val3--;
```

Code Listing 13

In the previous code listing, both **val1** and **val2** are **6**. The **val3** variable is **1**, but **val4** is **2** because the postfix operator evaluates after assignment.

The ternary operator offers simple syntax for if-then-else logic. Here's an example:

```
decimal priceGain = 2.5m;
string action = priceGain > 2m ? "Buy" : "Sell";
```

Code Listing 14

On the left side of **?** is a Boolean expression, **priceGain > 2m**. If that is true, which it is in this example, the ternary operator returns the first value between **?** and **:**, which is **"Buy"**. Otherwise, the ternary operator would return the value after the **:**, which is **"Sell"**. This statement assigns the result of the ternary operator, **"Buy"**, to the string variable, **action**.

In addition to the built-in types, the FCL has many types you will use on a daily basis. One of these is **DateTime**, which represents a date and time. Here's a quick demo showing a couple things you can do with a **DateTime**.

```
DateTime currentTime = DateTime.Now;
string shortDateString = currentTime.ToShortDateString();
string longDateString = currentTime.ToLongDateString();
string defaultDateString = currentTime.ToString();
DateTime tomorrow = currentTime.AddDays(1);
```

Code Listing 15

The previous code shows how to get the current **DateTime**, a short representation of a date (e.g., 12/8/2014), a long representation of the date and time (everything spelled out), the default numeric representation, and how to use **DateTime** methods for calculations.

Tip: Search the FCL before creating your own library of types. Many of the common types you use every day, like `DateTime`, will already exist.

Operator Precedence and Associativity

The C# operators listed in Table 2 outlines operators in their general order of precedence. The precedence defines which operators evaluate first. Operators of higher precedence evaluate before operators of lower precedence.

Assignment and conditional operators are right-associative and all other operators are left-associative. You can change the normal order of operations by using parentheses as shown in the following code listing.

```
int result1 = 2 + 3 * 5;
int result2 = (2 + 3) * 5;
```

Code Listing 16

In the previous code, **result1** is **17**, but **result2** is **25**.

Formatting Strings

There are different ways to build and format strings in C#: concatenation, numeric format strings, or string interpolation. The following code listing demonstrates string concatenation.

```
string name = "Joe";
string helloViaConcatenation = "Hello, " + name + "!";
Console.WriteLine(helloViaConcatenation);
```

Code Listing 17

This prints **"Hello, Joe!"** to the console. The following example does the same thing, but uses **string.Format**.

```
string helloViaStringFormat = string.Format("Hello, {0}!", name);
Console.WriteLine(helloViaStringFormat);
```

Code Listing 18

The **string.Format** takes a format string that has numeric placeholders in curly braces. It's 0-based, so the first placeholder is **{0}**. The parameters following the string are placed into the format string in the order they appear. Since **name** is the first (and only) parameter, **string.Format** replaces **{0}** with **Joe** to create **"Hello, Joe!"** as a string. As a convenience in console applications, **WriteLine** uses the same formatting. The following code accomplishes the same task as the two lines in the previous code listing.

```
Console.WriteLine("Hello, {0}!", name);
```

Code Listing 19

Going a little further, string formatting is more powerful, allowing you to specify column lengths, alignment, and value formatting as shown in the following code.

```
string item = "bread";
decimal amount = 2.25m;
Console.WriteLine("{0,-10}{1:C}", item, amount);
```

Code Listing 20

In this example, the first placeholder consumes 10 characters in length. The default alignment is right, but the minus sign changes that to align on the left. On the second placeholder, the **C** is a currency format string.

Note: There are many string formatting options. You can visit https://msdn.microsoft.com/en-us/library/dwhawy9k(v=vs.110).aspx for standard formats, https://msdn.microsoft.com/en-us/library/0c899ak8(v=vs.110).aspx for custom formats, and https://msdn.microsoft.com/en-us/library/az4se3k1(v=vs.110).aspx for DateTime formats.

C# 6 introduced a new way to format strings, called string interpolation. It's a shorthand syntax that lets you replace numeric placeholders with expressions as follows:

```
Console.WriteLine($"{item}      {amount}");
```

Code Listing 21

The **$** prefix is required. Here, the value from the **item** variable replaces **{item}** and the value from the **amount** variable replaces **{amount}**. Similar to numeric placeholders, you can include additional formatting.

```
Console.WriteLine($"{nameof(item)}: {item,-10} {nameof(amount)}: {amount:C}");
```

Code Listing 22

The **nameof** operator prints out the name ***"item"***, demonstrating how you can use expressions in placeholders. You can also see the space and currency formatting on **item** and **amount**.

Branching Statements

You can use either an **if-else** or **switch** statement in your code for branching logic. When you only need to execute code for a true condition, use an **if** statement as in the following sample.

```
string action2 = "Sell";
```

```
if (priceGain > 2m)
{
    action2 = "Buy";
}
```

Code Listing 23

The curly braces are optional in this example because there is only one statement to execute if **`priceGain > 2m`**. However, they would be required for multiple statements. This is true for all branching and logic statements. You can also have an **`else`** case, as shown in the following listing.

```
string action3 = "Do Nothing";
if (priceGain <= 2m)
{
    action3 = "Sell";
}
else
{
    action3 = "Buy";
}
```

Code Listing 24

Whenever the Boolean condition of the **`if`** statement is false, as it is in the previous code sample where **`priceGain <= 2m`**, the **`else`** clause executes. In this case, **`action3`** becomes **`"Buy"`**. Of course, you can have multiple conditions by adding more **`else if`** clauses.

```
string action4 = null;
if (priceGain <= 2m)
{
    action4 = "Sell";
}
else if (priceGain > 2m && priceGain <= 3m)
{
    action4 = "Do Nothing";
}
else
{
    action4 = "Sell";
}
```

Code Listing 25

In the previous example, you can see a more complex Boolean expression in the **`else if`** clause. When **`priceGain`** is **`2.5`**, the value of **`action4`** becomes **`"Do Nothing"`**. The **`&&`** is a logical operator that succeeds if both the expression on the left and right are true. The logical **`||`** operator succeeds if either the expression on the left or right is true. These operators also perform short-circuit operations where the expression on the right doesn't execute if the expression on the left causes the whole expression to not be true. In the case of the **`else if`** in

Code Listing 25, if **priceGain** were **2m** or less, the **&&** operator would not evaluate the **priceGain <= 3** expression because the entire operation is already false. Once a branch of the **if** statement executes, no other branches are evaluated or executed.

Notice that I set **action4** to **null**. The **null** keyword means no value. I'll talk about **null** in the next chapter and explain where you can use it.

An **if** statement is good for either simple branching or complex conditions, such as the previous **else if** clause. However, when you have multiple cases and all expressions are constant values, such as an **int** or **string**, you might prefer a **switch** statement. The following example uses a **switch** statement to select appropriate equipment based on a weather forecast.

```
string currentWeather = "rain";
string equipment = null;
switch (currentWeather)
{
    case "sunny":
        equipment = "sunglasses";
        break;
    case "rain":
        equipment = "umbrella";
        break;
    case "cold":
    default:
        equipment = "jacket";
        break;
}
```

Code Listing 26

The **switch** statement tries to match a value, **currentWeather** in this example, with one of its **case** statements. It uses the **default** case for no match. All **case** statements must be terminated with a **break** statement. The only time fall-through is allowed is when a **case** has no body, as demonstrated with the **"cold" case** and **default**, which both set **equipment** to **"jacket"**. Since **currentWeather** is **"rain"**, **equipment** becomes **"umbrella"** and no other cases execute.

Beyond branching statements, you also need the ability to perform a set of operations multiple times, which is where C# loops come in. Before discussing loops, let's look at arrays and collections, which hold data that loops can use.

Arrays and Collections

Sometimes you need to group a number of items together in a collection to manage them in memory. For this, you can either use arrays or one of the many collection types in the .NET Framework. The following sample demonstrates how to create an array.

```
int[] oddNumbers = { 1, 3, 5 };
```

```
int firstOdd = oddNumbers[0];
int lastOdd = oddNumbers[2];
```

Code Listing 27

Here, I've declared and initialized the array with three values. Arrays and collections are 0-based, so **`firstOdd`** is **`1`** and **`lastOdd`** is **`5`**. The **`[x]`** syntax, where **`x`** is a number, is referred to as an indexer because it allows you to access the array at the location specified by the index. Here's another example that uses **`string`** instead of **`int`**.

```
string[] names = new string[3];
names[1] = "Joe";
```

Code Listing 28

In this example, I instantiated an array to hold three strings. All of the strings equal **`null`** by default. This code sets the second string to **`"Joe"`**.

In addition to arrays, you can use all types of data structures, such as **`List`**, **`Stack`**, **`Queue`**, and more, which are part of the FCL. The following example shows how to use a **`List`**. Remember to add a **`using`** clause for **`System.Collections.Generic`** to use the **`List<T>`** type.

```
List<decimal> stockPrices = new List<decimal>();
stockPrices.Add(56.23m);
stockPrices.Add(72.80m);
decimal secondStockPrice = stockPrices[1];
```

Code Listing 29

In this sample, I instantiated a new **`List`** collection. The **`<decimal>`** is a generic type indicating that this is a strongly typed list that can only hold values of type **`decimal`**; it's a **`List`** of **`decimal`**. That list has two items. Notice how I used the array-like indexer syntax to read the second item in the (0-based) **`stockPrices`** list.

Looping Statements

C# supports several loops, including **`for`**, **`foreach`**, **`while`**, and **`do`**. The code listings that follow perform similar logic.

```
double[] temperatures = { 72.3, 73.8, 75.1, 74.9 };
for (int i = 0; i < temperatures.Length; i++)
{
    Console.WriteLine(i);
}
```

Code Listing 30

The **for** loop initializes **i** to **0**, makes sure **i** is less than the number of items in the **temperature** array, executes the **Console.WriteLine**, and then increments **i**. It continues executing until the condition (**i < temperatures.Length**) is false, and then moves on to the next statement in the program.

```
foreach (int temperature in temperatures)
{
    Console.WriteLine(temperature);
}
```

Code Listing 31

The **foreach** loop used in Code Listing 31 is simpler and will execute for each value in the **temperatures** array.

Next is an example of a **while** loop.

```
int tempCount = 0;
while (tempCount < temperatures.Length)
{
    Console.WriteLine(tempCount);
    tempCount++;
}
```

Code Listing 32

The **while** loop evaluates the condition and executes if it's true. Notice that I initialized **tempCount** to **0** and increment **tempCount** inside of the loop on each iteration.

Finally, the following example shows how to write a **do-while** loop.

```
int tempCount2 = 0;
do
{
    Console.WriteLine(tempCount2++);
}
while (tempCount2 <= temperatures.Length);
```

Code Listing 33

A **do-while** loop is good for when you want to execute logic at least one time. This example increments **tempCount2** as a parameter to **Console.WriteLine**. Remember, the postfix operator changes the variable after evaluation.

Wrapping Up

Here's a calculator program that pulls together some of the concepts from this chapter, plus some extra features. You can type this into your editor and execute it for practice.

```
using System;
using System.Text;

/*
    Title: Calculator
    By: Joe Mayo
*/

class Calculator
{
    /// <summary>
    /// This is the entry point.
    /// </summary>
    static void Main()
    {
        char firstChar = 'Q';
        bool keepRunning = true;

        do
        {
            Console.WriteLine();
            Console.Write("What do you want to do? (Add, Subtract, Multiply, Divide,
Quit): ");
            string input = Console.ReadLine();
            firstChar = input[0];

            // This is used in both the if statement and the do-while loop.
            keepRunning = !(firstChar == 'q' || firstChar == 'Q');

            double firstNumber = 0;
            double secondNumber = 0;

            if (keepRunning)
            {
                Console.Write("First Number: ");
                string firstNumberInput = Console.ReadLine();
                firstNumber = double.Parse(firstNumberInput);

                Console.Write("Second Number: ");
                string secondNumberInput = Console.ReadLine();
                secondNumber = double.Parse(secondNumberInput);
            }

            double result = 0;
            switch (firstChar)
            {
                case 'a':
                case 'A':
                    result = firstNumber + secondNumber;
                    break;
```

```
                case 's':
                case 'S':
                    result = firstNumber - +secondNumber;
                    break;
                case 'm':
                case 'M':
                    result = firstNumber * secondNumber;
                    break;
                case 'd':
                case 'D':
                    result = firstNumber / secondNumber;
                    break;
                default:
                    result = -1;
                    break;
            }

            Console.WriteLine();
            Console.WriteLine("Your result is " + result);

        } while (keepRunning);
    }
}
```

Code Listing 34

The previous program demonstrates a **do-while** loop, an **if** statement, a **switch** statement, and a basic console communication with the user.

There are a couple string features here that you haven't seen yet. The first is where the program uses **Console.ReadLine** to read input text from the user for the input string. Notice the indexer syntax to read the first character from the string. You can read any character of a string this way. Also, look at the bottom of the program where it prints **"Your result is " + result**, which concatenates a string with the number. Using the **+** operator for concatenation is a simple way to build strings. Another way to build a string is with a type named **StringBuilder**, which you can use like this:

```
StringBuilder sb = new StringBuilder();
sb.Append("Your result is ");
sb.Append(result.ToString());
Console.WriteLine(sb.ToString());
```

Code Listing 35

You'll also need to add a **using System.Text;** clause to the top of the file. After you've used the concatenate operator, **+**, about four times on the same string, you might consider rewriting with a **StringBuilder** instead. The **string** type is immutable, meaning that you can't modify it. This also means that every concatenation operation causes the CLR to create a new string in-memory.

The calculator program also has multiline and single-line comments that aren't compiled, but help you document the code as you need. Here's the multi-line comment:

```
/*
    Title: Calculator
    By: Joe Mayo
*/
```

Code Listing 36

Here's the single-line comment:

```
// This is used in both the if statement and the do-while loop.
```

Code Listing 37

An extension of the single-line comment is a convention that uses three slashes and a set of XML tags, known as documentation comments.

```
/// <summary>
/// This is the entry point.
/// </summary>
```

Code Listing 38

Summary

C# has a full set of operators and types that allow you to write a wide range of expressions and statements. With branching statements and loops, you can write logic of your choosing. All of the code in this chapter has been in the **`Main`** method, but clearly that's inadequate and you'll quickly grow out of that. The next chapter explores some new C# features to help organize code with methods and properties.

Chapter 3 Methods and Properties

Previous chapters show how to write code in the **Main** method. That's the program entry point, but it's normally a lightweight method without too much code. For this chapter, you'll learn how to move your code out of the **Main** method and modularize it so you can manage the code better. You'll learn how to define methods with parameters and return values. You'll also learn about properties, which let you encapsulate object state.

Starting at Main

We'll use a simpler version of the calculator from the previous chapter to get started. This calculator only performs addition and stops running after one operation.

```
using System;

class Calculator1
{
    static void Main()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);

        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);

        double result = firstNumber + secondNumber;

        Console.WriteLine($"\n\tYour result is {result}.");

        Console.ReadKey();
    }
}
```

Code Listing 39

The part of this program that might be new is the **Console.ReadKey** statement at the end of the **Main** method. This allows users to see the result and keeps the program from ending until they press a key. The **\n** in the interpolated string is a newline and **\t** is a tab.

Modularizing with Methods

Although the previous program is small, a first glance doesn't really tell you what it does. Imagine if it was like the calculator in Chapter 2 or even longer; it would eventually become difficult to understand. When you have to work on this again later, you might need to read many

lines of code to understand it. So, it would be better to refactor this. Refactoring is the practice of changing the design of code without changing its functionality; the purpose is to improve the program. The following code sample is a first draft of refactoring this program into methods.

```
using System;

class Calculator2
{
    static void Main()
    {
        double firstNumber = GetFirstNumber();

        double secondNumber = GetSecondNumber();

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetFirstNumber()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);
        return firstNumber;
    }

    static double GetSecondNumber()
    {
        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);
        return secondNumber;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

Code Listing 40

Looking at **`Main`**, you can tell what the program does. It reads two numbers, adds the results, and then shows the results to the user. Each of those lines is a method call. The first three methods—**`GetFirstNumber`**, **`GetSecondNumber`**, and **`AddNumbers`**—return a value that is assigned to a variable. The last method, **`PrintResult`**, performs an action without returning a

result. Before moving to the next refactoring, let's walk through these methods. The following code listing shows the **GetFirstNumber** method.

```
static double GetFirstNumber()
{
    Console.Write("First Number: ");
    string firstNumberInput = Console.ReadLine();
    double firstNumber = double.Parse(firstNumberInput);
    return firstNumber;
}
```

Code Listing 41

At first glance, the signature of this method looks similar to the **Main** method. The differences are that the return type of this method is **double** and the method is named **GetFirstNumber**. All we did was write the method and the code that creates the **firstNumber**. When a method has a return type, a value of that type must be returned. **GetFirstNumber** does that with the **return** statement.

The **GetSecondNumber** method is nearly identical to **GetFirstNumber**. Let's examine **AddNumbers** next.

```
static double AddNumbers(double firstNumber, double secondNumber)
{
    return firstNumber + secondNumber;
}
```

Code Listing 42

Notice that **Main** passes the **firstNumber** and **secondNumber** variables to **AddNumbers** as arguments that the **AddNumbers** method can work with as parameters. The return type of **AddNumbers** is **double**, so the method adds and returns the result of the add operation.

Finally, we have the **PrintResult** method.

```
static void PrintResult(double result)
{
    Console.WriteLine($"\nYour result is {result}.");
}
```

Code Listing 43

The **PrintResult** method writes the results from its parameter to the console. Notice that **PrintResult** does not have a return type, as indicated by the **void** keyword.

Simplifying Code with Methods

The last section improved the program because a huge block of code was broken into more meaningful pieces. We can improve this code with some extra refactoring. In particular, the **`GetFirstNumber`** and **`GetSecondNumber`** methods are largely redundant. The following sample shows how to refactor those two methods into one and reduce the amount of code.

```
using System;

class Calculator3
{
    static void Main()
    {
        double firstNumber = GetNumber("First");
        double secondNumber = GetNumber("Second");

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);
        return number;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

Code Listing 44

This time I removed **`GetFirstNumber`** and **`GetSecondNumber`** and replaced them with **`GetNumber`**. The only real difference besides variable names is the **`whichNumber string`** parameter.

Adding Properties

The previous examples performed all of the operations inside of the same class. It was driven from the **`Main`** method and serviced through methods. What if I wanted to reuse the calculator

functions in that class and wanted the new class to hold its own values, or state? In this case, moving the calculator methods into a separate **`Calculator`** class would be useful.

The next question to ask is, "How do we get to the state of the class?" For example, if I want to read the result from the **`Calculator`** class, what is the best way to do so? One approach is to use a method named **`GetResult`** that returns the value. Another way in C# is to use a property, which you can use like a field, but works like a method. The following version of the calculator program shows how to refactor methods into a separate class and add properties.

Note: Refactoring is the practice of changing the design of code without changing its behavior with the goal of improving the code. Martin Fowler's book,* Refactoring: Improving the Design of Existing Code*, is a good reference.

```
using System;

public class Calculator4
{

    double[] numbers = new double[2];

    public double First
    {
        get
        {
            return numbers[0];
        }
    }

    public double Second
    {
        get
        {
            return numbers[1];
        }
    }

    double result;

    public double Result
    {
        get { return result; }
        set { result = value; }
    }

    public void GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);

        if (whichNumber == "First")
            numbers[0] = number;
        else
```

```
            numbers[1] = number;
    }

    public void AddNumbers()
    {
        Result = First + Second;
    }

    public void PrintResult()
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

Code Listing 45

In the previous code listing, **`First`**, **`Second`**, and **`Result`** are properties. I'll break down the syntax shortly, but first look at how these properties are used inside of the **`AddNumbers`** and **`PrintResults`** methods. **`AddNumbers`** reads the values of **`First`** and **`Second`** and adds those values together and writes to **`Result`**.

Each of these properties looks just like a field or variable; you just read from and write to them. **`PrintResult`** reads the **`Result`** property. However, looking at the definitions of the properties, you can tell right away that they aren't fields.

The **`Result`** property is a typical read and write property with **`get`** and **`set`** accessors. When you read the property, the **`get`** accessor executes. When you write to the property, the **`set`** accessor executes. Notice that there is a **`result`** field (lowercase) prior to the **`Result`** (uppercase) property. The **`get`** accessor reads the value of **`result`** and the **`set`** accessor writes to **`result`**. When using **`set`**, the **`value`** keyword represents what is being written to the property.

This pattern of reading and writing from a single backing store is so common that C# has a shortcut syntax you can use instead. The following code sample shows **`Result`** rewritten as an auto-implemented property.

```
    public double Result { get; set; }
```

Code Listing 46

The backing store in auto-implemented properties is implied and handled behind the scenes by the C# compiler. If you need to provide validation on the value being assigned or have a unique way of storing the value, you should resort to full properties where the **`get`** accessor, **`set`** accessor, or both are defined.

In fact, **`First`** and **`Second`** properties have a unique backing store, requiring a fully implemented **`get`** accessor. They read from an array position. Notice that the **`GetNumber`** method figures out which array position to put each number into.

Properties give you the ability to encapsulate the internal operations of your class so you are free to modify the implementation without breaking the interface for consumers of your class.

The following code sample demonstrates how consuming code might use this new **Calculator4** class.

```
using System;

class Program
{
    static void Main()
    {
        var calc4 = new Calculator4();

        calc4.GetNumber("First");
        calc4.GetNumber("Second");

        calc4.AddNumbers();

        PrintResult(calc4);

        Console.ReadKey();
    }

    static void PrintResult(Calculator4 calc)
    {
        Console.WriteLine($"Your result is {result}.");
    }
}
```

Code Listing 47

The **Main** method creates a new instance of **Calculator4** and calls public methods. All of the strange internals of **Calculator4** are hidden and **Main** only cares about the public interface, exposing **Calculator4** services for reuse. The **PrintResult** method reads the **Calculator4** instance **Result** property. Again, that's the benefit of methods and properties: callers can use a class without caring how that class does what it does.

Exception Handling

C# has a feature called structured exception handling that lets you work with situations where your methods aren't able to fulfill their intended purpose. The syntax for managing exception handling is the **try-catch** block. All the code to be monitored for exceptions goes in the **try** block, and the code that handles a potential exception goes in a **catch** block. The following code listing shows an example.

```
        static void HandleNullReference()
        {
            Program prog = null;

            try
            {
                Console.WriteLine(prog.ToString());
```

```
        }
        catch (NullReferenceException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
```

Code Listing 48

In C#, any time you try to use a member of a **`null`** object, you'll receive a **`NullReferenceException`**. The solution to fix the problem is to assign a value to the variable. The previous example causes a **`NullReferenceException`** to be thrown in the **`try`** block by calling **`ToString`** on the **`prog`** variable, which is **`null`**.

Since the code that threw the exception is inside the **`try`** block, the code stops execution of any of the code in the **`try`** block and starts looking for an exception handler. The **`catch`** block **`parameter`** indicates that it can catch a **`NullReferenceException`** if the code inside of the **`try`** block throws that exception type. The body of the **`catch`** block is where you perform any exception handling.

You can customize exception handling with multiple **`catch`** blocks. The following example shows code that throws an exception in the **`try`** block, which is subsequently handled by a **`catch`** block.

```
    static void HandleUncaughtException()
    {
        Program prog = null;

        try
        {
            Console.WriteLine(prog.ToString());
        }
        catch (ArgumentNullException ex)
        {
            Console.WriteLine("From ArgumentNullException: " + ex.Message);
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("From ArgumentException: " + ex.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine("From Exception: " + ex.Message);
        }
        finally
        {
            Console.WriteLine("Finally always executes.");
        }
    }
```

Code Listing 49

The method name is **HandleUncaughtException** because there isn't a specific **catch** block to handle a **NullReferenceException**; the exception will be handled by the **catch** block for the **Exception** type.

You list exceptions by their inheritance hierarchy, with top-level exceptions lower in the list of **catch** blocks. A thrown exception will move down this list of handlers, looking for a matching exception type, and only execute in the **catch** block of the first handler that matches. **ArgumentNullException** derives from **ArgumentException**, and **ArgumentException** derives from **Exception**.

If no **catch** block can handle an exception, the code goes up the stack looking for a potential **catch** block in calling code that can handle the exception type. If no code in the call stack is able to handle the exception, your program will terminate.

The **finally** block always executes if the program begins executing code in the **try** block. If an exception occurs and is not caught, the **finally** block will execute before the program looks at the calling code for a matching catch handler.

You can write a **try-finally** block (without **catch** blocks) to guarantee that certain code will execute once the **try** block begins. This is useful for opening a resource, like a file or database connection, and then guaranteeing you will be able to close that resource regardless of whether an exception occurs.

If you encounter a reason why your method can't perform its intended purpose, **throw** an exception. There are many **Exception**-derived types in the .NET FCL that you can use. The following code example pulls together a few concepts you might want to use, such as validating method input and throwing an **ArgumentNullException**.

```
public class Address
{
    public string City { get; set; }
}

internal class Company
{
    public Address Address { get; set; }
}

        // Inside of a class...
        static void ThrowException()
        {
            try
            {
                ValidateInput("something", new Company());
            }
            catch (ArgumentNullException ex) when (ex.ParamName == "inputString")
            {
                Console.WriteLine("From ArgumentNullException: " + ex.Message);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("From ArgumentException: " + ex.Message);
```

```
        }
    }

    static void ValidateInput(string inputString, Company cmp)
    {
        if (inputString == null)
            throw new ArgumentNullException(nameof(inputString));

        if (cmp?.Address?.City == null)
            throw new ArgumentNullException(nameof(cmp));
    }
```

Code Listing 50

The previous code shows an **Address** class and a **Company** class with a property of the **Address** type. The **try** block of the **ThrowException** message passes a new instance of **Company**, but doesn't instantiate **Address**, meaning that the **Company** instance's **Address** property is **null**.

Inside **ValidateInput**, the **if** statement uses the null referencing operator, **?.**, to check if any of the values between **Company**, **Company**'s **Address** property, or **Address**'s **City** property is **null**. This is a convenient way to check for **null** without a group of individual checks, producing less syntax and simpler code. If any of these values are **null**, the code throws an **ArgumentNullException**.

The argument to the **ArgumentNullException** uses the **nameof** operator, which evaluates to a **string** representation of the value passed to it; it is **"cmp"** in this case. This code isn't enclosed in a **try** block, so control returns to the code calling this method.

Back in the **ThrowException** method, the thrown exception causes the code to look for a handler suitable for this exception type. The exception type is **ArgumentNullException**, but the **catch** block for **ArgumentNullException** won't execute. That's because the **when** clause following the **ArgumentNullException catch** block parameter is checking for a **ParamName** of **"inputString"**. This **when** clause is called an exception filter. As mentioned previously, the parameter name passed to the **ArgumentNullException** during instantiation was **"cmp"**, so there isn't a match. Therefore, the code continues looking at **catch** handlers.

Since **ArgumentNullException** derives from **ArgumentException** and there is no exception filter, the **catch** handler for **ArgumentException** executes. The exception is now handled.

Tip: It's typically better to throw and handle specific exceptions, rather than their parent exceptions. This adds more fidelity and meaning to the exception and makes debugging easier.

Summary

Methods help you organize code into named functions that you can call to perform operations. Their name documents what the code does. Also, methods are useful to help avoid duplicating the same code in multiple places. Properties are used like fields and look like fields from the perspective of code using the property's class. However, properties are more sophisticated in that they have **`get`** and **`set`** accessors that let them work like methods and perform more sophisticated work, like validation or special value handling. Both methods and properties help define the interface of a class to consumers and let you encapsulate the internal operations of a class, which makes it more reusable. You can use **`try-catch`** blocks to handle exceptions and **`try-finally`** blocks to guarantee critical code executes. Use the **`throw`** statement whenever a method you're writing can't fulfill its intended purpose.

Chapter 4 Writing Object-Oriented Code

C# is an object-oriented programming (OOP) language. It supports inheritance, encapsulation, polymorphism, and abstraction. This chapter shows you how C# supports OOP.

Implementing Inheritance

In C#, inheritance defines a relationship between two classes where a derived class can reuse members of a base class. A simple way to view this is that a derived class is a more specific version of a base class. We will reuse the calculator example from previous chapters, but alter it to provide two different types of calculators: scientific and programmer. Since they're both calculators, it could be useful to create a relationship with a **`Calculator`** base class and **`ScientificCalculator`** and **`ProgrammerCalculator`** derived classes, like this:

- **`Calculator`**
  - **`ScientificCalculator`**
  - **`ProgrammerCalculator`**

In C#, you would express this relationship as follows.

```
public class Calculator { }
public class ScientificCalculator : Calculator { }
public class ProgrammerCalculator : Calculator { }
```

Code Listing 51

As you can see, we added a colon suffix (as the inheritance operator) to the derived class and specified the base class it is derived from. The following figure illustrates the inheritance relationship between these classes.

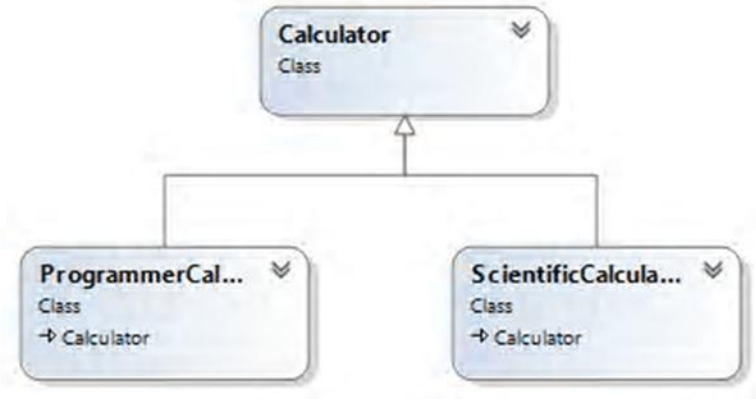


Figure 1: Calculator Inheritance Diagram

You can assume that **`Calculator`** will have all of the standard operations like addition, subtraction, and more that all calculators have. The following code listing is an expanded example that shows the base class with a common method, and the derived classes with specialized methods that only belong to those classes.

```
using System;

public class Calculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ScientificCalculator : Calculator
{
    public double Power(double num, double power)
    {
        return Math.Pow(num, power);
    }
}

public class ProgrammerCalculator : Calculator
{
    public int Or(int num1, int num2)
    {
        return num1 | num2;
    }
}
```

Code Listing 52

The methods of the derived classes in the previous example used the FCL **`Math`** class for **`Power`**, which has many more math methods you can use in your own code, and also used the built-in C# **`|`** operator for **`Or`**. The following example shows how to write code that takes advantage of inheritance with the previous classes.

```
using System;

public class Program
{
    public static void Main()
    {
        ScientificCalculator sciCalc = new ScientificCalculator();
        double powerResult = sciCalc.Power(2, 5);
        Console.WriteLine($"Scientific Calculator 2**5: {powerResult}");
        double sciSum = sciCalc.Add(3, 3);
        Console.WriteLine($"Scientific Calculator 3 + 3: {sciSum}");

        ProgrammerCalculator prgCalc = new ProgrammerCalculator();
        double orResult = prgCalc.Or(5, 10);
        Console.WriteLine($"Programmer Calculator 5 | 10: {orResult}");
```

```
        double prgSum = prgCalc.Add(3, 3);
        Console.WriteLine($"Programmer Calculator 3 + 3: {prgSum}");

        Console.ReadKey();
    }
}
```

Code Listing 53

Both the **ScientificCalculator** instance, **sciCalc**, and the **ProgrammerCalculator** instance, **prgCalc**, call the **Add** method. Further, those classes don't define their own **Add**, but they do derive from **Calculator** and therefore inherit **Calculator**'s **Add**.

The ability to inherit isn't always guaranteed; the next section explains more about when a class member is visible to other classes.

Access Modifiers and Encapsulation

In the previous example, all classes and methods had **public** modifiers, meaning that any other class or code can see and access them in code. You can leave off access modifiers and accept defaults. In that case, a class access becomes what is known as **internal**, and class members default to **private**.

Classes can only be **internal** or **public**. If they're **internal**, they can only be accessed by code inside of the assembly they are contained in.

Available access modifiers for class members include **public**, **private**, **internal**, **internal protected**, and **protected**. The **public** and **internal** modifiers have the same meaning for class members as they do for classes.

The default modifier for class members, **private**, means that code outside the class can't use that member; only other members within the same class can use it. This is useful if you want to modularize a method by breaking it into different supporting methods, but the supporting methods have no meaning outside of the class.

The **protected** modifier allows derived classes inside and outside the assembly to use the **protected** base class member. The **internal protected** modifier further restricts protected behavior only to derived classes inside the same assembly.

Most of the access modifier defaults and behaviors apply to **struct** types as well as classes, except for **protected** and **internal protected**, as I'll explain next.

Designing Types: Class vs. Struct

A **struct** is another C# type that looks similar to a **class**, but has different behavior. A **struct** can't derive from another **class** or **struct**. Since implementation inheritance doesn't apply to a **struct**, neither do **protected** and **internal protected** modifiers. A **struct** does have

interface inheritance, which I'll explain more in the Exposing Interfaces section later in this chapter.

Additionally, a **struct** copies by value, as opposed to a **class** which copies by reference. The difference is that if you pass a **struct** instance to a method, the method gets a brand new copy of the **struct** value. If you copy a **class** instance to a method, the method gets a copy of a reference to the **class** in the heap, which is an area in computer memory that the CLR uses to allocate space for reference type objects. These facts help you decide whether you should design a type as a **class** or **struct**. Imagine a type with many properties and how it would hurt performance if you had to pass it by value to a method as a **struct**; the state of that type is copied to the stack, which is memory the CLR allocates for every method call to hold items like parameters and local variables. In that case, the proper design decision might be to define the type as a **class** so that only the reference is copied.

Most of the built-in types, such as **int**, **double**, and **char**, are value types. If you have a type with those semantics—small and a single value—then designing a type as a **struct** might be a benefit. Otherwise, designing a type as a **class** is fine. Here's an example of a type that might make a good **struct**.

```
public struct Complex
{
    public Complex(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public double Real { get; set; }

    public double Imaginary { get; set; }

    public static Complex operator +(Complex complex1, Complex complex2)
    {
        Complex complexSum = new Complex();
        complexSum.Real = complex1.Real + complex2.Real;
        complexSum.Imaginary = complex1.Imaginary + complex2.Imaginary;
        return complexSum;
    }
    public static implicit operator Complex(double dbl)
    {
        Complex cmplx = new Complex();
        cmplx.Real = dbl;
        return cmplx;
    }
    // This is not a safe operation.
    public static explicit operator double(Complex cmplx)
    {
        return cmplx.Real;
    }
}
```

Code Listing 54

Complex could make a good **struct** because you might have a lot of mathematical operations, and it would be more efficient to pass a copy of the numbers on the stack rather than letting the CLR allocate memory as it does for a **class**.

Complex has a constructor, named after the class itself, with a couple parameters. This makes it easy to initialize a new instance of **Complex**.

There are a few operator overloads in **Complex**: an addition operator and two conversion operators. The addition operator lets you add two complex numbers. Where the operator identifier (**+**) precedes the parameter list, the values to be added are specified in the parameters, and the return type is part of the signature. Operators are always **static**.

The two conversion operators let you make assignments between the containing type and another type of your choice. The type assigned to is the operator identifier and the type being assigned is the parameter. The **implicit** modifier means the conversion is safe and the **explicit** modifier means the conversion has the potential to lose data or provide an invalid result. For example, assigning a **double** to an **int** would be **explicit** because of loss of precision, and the explicit conversion in the previous example causes loss of the imaginary part of the number. The following code sample demonstrates how **Complex** could be used.

```
using System;

class Program
{
    static void Main()
    {
        Complex complex1 = new Complex();
        complex1.Real = 3;
        complex1.Imaginary = 1;

        Complex complex2 = new Complex(7, 5);

        Complex complexSum = complex1 + complex2;

        Console.WriteLine(
            $"Complex sum - Real: {complexSum.Real}, " +
            $"Imaginary: {complexSum.Imaginary}");

        Complex complex3 = 9;

        double realPart = (double)complex3;

        Console.ReadKey();
    }
}
```

Code Listing 55

The **Main** method instantiates **complex1** and then populates its values. Next, **Main** instantiates **complex2** by using the **Complex** constructor, which is simpler initialization code.

You can also see how natural it is to use the addition operator, rather than an **Add** method used in previous **Calculator** demos.

Because there's an implicit conversion from **int** to **double** and **Complex** has an implicit conversion operator from **double** to **Complex**, **Main** is able to assign **9** to **complex3**. The same can't be said for assigning **complex3** to **realPart** because **Complex** to **double** is an **explicit** conversion operator in the **Complex** type. Any time you have an **explicit** conversion, you must use a cast operator, as in **(double)complex3**.

One of the items you need to watch out for when working with value types is a concept referred to as boxing and unboxing. Boxing occurs any time you assign a value type to **object**, and unboxing occurs when you assign **object** to a value type. The following code demonstrates one scenario where this could happen.

```
ArrayList intCollection = new ArrayList();
intCollection.Add(7);
int number = (int)intCollection[0];
```

Code Listing 56

An **ArrayList** is a collection class belonging to the **System.Collections** namespace. It is more powerful than an array and operates on type **object**. The **Add** method accepts an argument of type **object**. Since all types derive from **object**, an **ArrayList** is flexible enough to allow you to work with objects of any type. Boxing occurs when passing **7** to the **Add** method because **7** is an **int** (a value type) and is converted to **object**. What is really happening is that the CLR creates a boxed **int** in memory. Since the **ArrayList** holds type **object**, you also need to perform a conversion to unbox a value when reading from the **ArrayList**. The **(int)** cast operator converts from **object** (the boxed **int**) to **int** when reading the first element of **intCollection**.

The problem that boxing and unboxing cause is related to performance. In this situation, the reason you would use a collection is because you want to hold a lot of **int** values, which could be hundreds or thousands. Think about all the time spent accessing that **ArrayList** and incurring the performance penalty of boxing and unboxing on each operation.

***Note:** `ArrayList` is an old collection class that existed in C# v1.0 and is no longer used in modern development. C# v2.0 introduced generics, which use new collection classes that are strongly typed and avoid the boxing and unboxing penalties. While the `ArrayList` example is unlikely today, this scenario still highlights the performance penalty of any other situation where you might be assigning a value type to an object type.*

Another difference between **class** (reference types) and **struct** (value types) is equality evaluation. Value type equality works by comparing the corresponding members of the **struct**. Reference type equality works by verifying that references are equal. In other words, structs are equal if their values match, but classes are equal if they reference the same object in memory.

In the later section on polymorphism, you'll learn how to override the **`object.Equals`** method to give you more control over class equality.

Creating Enums

An **`enum`** is a value type that lets you create a set of strongly typed mnemonic values. They're useful when you have a finite set of values and don't want to represent those values as strings or numbers. Here's an example of an **`enum`**.

```
public enum MathOperation
{
    Add,
    Subtract,
    Multiply,
    Divide
}
```

Code Listing 57

Like a **`struct`**, an **`enum`** is a value type. You use the **`enum`** keyword as the type definition. The previous **`enum`** is named **`MathOperation`** and has four members. The following example shows how you can use this **`enum`**.

```
using System;
using static MathOperation;

class Program
{
    static void Main()
    {
        string[] possibleOperations = Enum.GetNames(typeof(MathOperation));

        Console.Write($"Please select ({string.Join(", ", possibleOperations)}): ");

        string operationString = Console.ReadLine();

        MathOperation selectedOperation;

        if (!Enum.TryParse<MathOperation>(operationString, out selectedOperation))
            selectedOperation = MathOperation.Add;

        switch (selectedOperation)
        {
            case MathOperation.Add:
                Console.WriteLine($"You selected {nameof(Add)}");
                break;
            case MathOperation.Subtract:
                Console.WriteLine($"You selected {nameof(Subtract)}");
                break;
            case MathOperation.Multiply:
                Console.WriteLine($"You selected {nameof(Multiply)}");
```

```
                break;
            case MathOperation.Divide:
                Console.WriteLine($"You selected {nameof(Divide)}");
                break;
        }

        Console.ReadKey();
    }
}
```

Code Listing 58

The FCL has an **Enum** class that lets you work with enums and the previous **Main** method shows how to use a couple of its methods. **Enum.GetNames** returns a **string** array, representing the names in the **enum**, specified with the **typeof** operator. The **string.Join** method, the expression in the interpolated string of the **Console.WriteLine**, creates a comma-separated string of these names.

The **Enum.TryParse** method in the previous example takes a string and produces an **enum** of the type specified in the type parameter, which is **MathOperation** in this case. The **out** parameter means that **TryParse** will return the parsed value in the **selectedOperation** variable. This is practical because the return type of the **TryParse** is **bool**, allowing you to know whether the input string, **operationString**, is valid.

The **selectedOperation** variable is of type **MathOperation**. The default syntax for enums is to prefix them with the enum type name, as in **MathOperation.Add**. However, you can also add a **using static** clause to the top of the file, allowing you to only specify the member name, as the previous example shows in the **switch** statement. A **switch** statement can operate on numbers, strings, or enums.

Enabling Polymorphism

Polymorphism lets derived classes specialize a base class implementation. The mechanism to allow polymorphism is to decorate a base class method with the **virtual** modifier and decorate the derived class method with the **override** modifier. If you were designing the **Calculator** class, you could allow derived classes to implement their own improved or specialized versions of the **Add** method, as shown in the following sample.

```
using System;

public class Calculator
{
    public virtual double Add(double num1, double num2)
    {
        Console.WriteLine("Calculator Add called.");
        return num1 + num2;
    }
}
```

```
public class ProgrammerCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ProgrammerCalculator Add called.");
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ScientificCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ScientificCalculator Add called.");
        return base.Add(num1, num2);
    }
}
```

Code Listing 59

Polymorphism is opt-in for C#. Notice that the **`Add`** method in the base class **`Calculator`** has a **`virtual`** modifier. Polymorphism doesn't occur unless a base class method has the **`virtual`** modifier. Also, notice that the derived classes **`ScientificCalculator`** and **`ProgrammerCalculator`** have **`override`** modifiers. Again, these methods won't be called polymorphically unless they have the **`override`** modifier. Additionally, a method with the **`override`** modifier is also **`virtual`** for any of its derived classes.

With polymorphism, the overridden method in derived classes executes at runtime. If you wanted to call the base class implementation of that method, call the base class method with the **`base`** keyword. **`ScientificCalculator`** calls **`base.Add(num1, num2)`** to call the **`Add`** method in **`Calculator`**. Here's an example of how this works.

```
using System;

public class Program
{
    public static void Main()
    {
        Calculator sciCalc = new ScientificCalculator();
        double sciCalcResult = sciCalc.Add(2, 5);
        Console.WriteLine($"Scientific Calculator 2 + 5: {sciCalcResult}");

        Calculator prgCalc = new ProgrammerCalculator();
        double prgCalcResult = prgCalc.Add(5, 10);
        Console.WriteLine($"Programmer Calculator 5 + 10: {prgCalcResult}");
```

```
        Console.ReadKey();
    }
}
```

Code Listing 60

The output for this program would be:

```
ScientificCalculator Add called.

Calculator Add called.

Scientific Calculator 2 + 5: 7

ProgrammerCalculator Add called.

Programmer Calculator 5 + 10: 15
```

Main assigns instances of **ScientificCalculator** and **ProgrammerCalculator** to variables of type **Calculator**. As you saw in the previous listing, **ScientificCalculator** and **ProgrammerCalculator** are derived types and **Calculator** is their base type. The derived instances are the runtime type—the actual type when the program is running—and the base class is the compile-time type. The runtime-type overrides execute at runtime.

Looking at the definition of **Add** in **ScientificCalculator**, **Calculator**, and **Main**, and looking at the output, you can trace the polymorphic behavior of this program. **Main** calls **Add** on the **ScientificCalculator** instance. **ScientificCalculator.Add** executes because it overrides the **virtual Calculator.Add** method. After writing the first line of output, **ScientificCalculator.Add** calls the **Calculator.Add** method with the **base** keyword. **Calculator.Add** prints the second line to the output, performs the addition calculation, and returns the sum. **ScientificCalculator.Add** returns the return value from **Calculator.Add**. **Main** assigns the return value from **ScientificCalculator.Add** to the **sciCalc** variable and prints the results into the third line of the output. Tracing the call to **ProgrammerCalculator.Add** is similar, except that there is no call to the **Calculator.Add** in the base class.

Another example of when you would want to use polymorphism is in defining reference type equality. By default, reference types are only equal if their references are the same. The following example shows how to control reference type equality.

```
public class Customer
{
    int id;
    string name;

    public Customer(int id, string name)
    {
        this.id = id;
        this.name = name;
```

```
    }

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;

        if (obj.GetType() != typeof(Customer))
            return false;

        Customer cust = obj as Customer;

        return id == cust.id;
    }

    public static bool operator ==(Customer cust1, Customer cust2)
    {
        return cust1.Equals(cust2);
    }

    public static bool operator !=(Customer cust1, Customer cust2)
    {
        return !cust1.Equals(cust2);
    }

    public override int GetHashCode()
    {
        return id;
    }

    public override string ToString()
    {
        return $"{{ id: {id}, name: {name} }}";
    }
}
```

Code Listing 61

Since all classes implicitly derive from **`object`**, they can **`override`** object **`virtual`** methods **`Equals`**, **`GetHashCode`**, and **`ToString`**. **`Customer`** overrides **`Equals`**. When you override **`Equals`**, check for **`null`** and type equality before working with the objects to prevent callers from accidentally comparing **`null`** or incompatible types. **`Customer`** instances are equal if they have the same **`id`**.

`Customer` has a constructor that initializes the state of the class. The **`this`** operator lets you access members of the containing instance and helps avoid ambiguity.

When implementing custom equality, you should also overload the equals and not equals and override the **`GetHashCode`** method. The default implementation of **`GetHashCode`** is a system-defined object **`id`**, so you could override this to achieve a better distribution of values in a hash.

Tip: You can escape { and } characters that you don't want to evaluate as expressions by doubling them as {{ and }} respectively in string interpolation.

The following is an example of how to check equality of **`Customer`** instances.

```
using System;

class Program
{
    static void Main()
    {
        Customer cust1 = new Customer(1, "May");
        Customer cust2 = new Customer(2, "Joe");

        Console.WriteLine($"cust1 == cust2: {cust1 == cust2}");

        Customer cust3 = new Customer(1, "May");

        Console.WriteLine($"\ncust1 == cust3: {cust1 == cust3}");
        Console.WriteLine($"cust1.Equals(cust3): {cust1.Equals(cust3)}");
        Console.WriteLine($"object.ReferenceEquals(cust1, cust3):
{object.ReferenceEquals(cust1, cust3)}");

        Console.WriteLine($"\ncust1: {cust1}");
        Console.WriteLine($"cust2: {cust2}");
        Console.WriteLine($"cust3: {cust3}");

        Console.ReadKey();
    }
}
```

Code Listing 62

When using the **`==`** operator, the code calls the operator overload and **`Equals`** calls the equals method as expected. **`ReferenceEquals`** is an **`object`** method that is useful because it allows reference equality checking in case the type defined a custom **`Equals`** override.

If **`Customer`** had not overridden **`ToString`**, the last three **`Console.WriteLine`** statements in the previous code listing would have printed the type name, which is the default behavior of **`ToString`**.

Writing Abstract Classes

In previous examples, you could create an instance of **`Calculator`**. However, it may or may not make sense to instantiate a base class. A base class may serve only as a reusable type for common functionality of similar derived classes and to enable polymorphism, yet not have substance enough to be used on its own. In that case, you can modify the class definition as **`abstract`**, as shown in the following sample.

```
public abstract class Calculator
{
    // ...
}
```

Code Listing 63

In an **abstract** class, you can have **virtual** or non-virtual members. Additionally, you can have **abstract** methods. An **abstract** method doesn't have an implementation. Derived classes should specify the implementation and you don't want a default implementation in the base class that might not make sense. The purpose of an **abstract** method is to specify an interface that derived classes must implement. In the case of **Calculator**, you could define an **abstract Add** method as shown in the following code example.

```
public abstract class Calculator
{
    public abstract double Add(double num1, double num2);
}
```

Code Listing 64

The **Add** method has an **abstract** modifier. This method is implicitly virtual, but can't be called by a derived class because it doesn't have an implementation. The semicolon is required to terminate the **abstract** method signature. When an **abstract** class has **abstract** methods, all derived classes must **override** the **abstract** method. The **Main** method in the previous section still runs if you change the definition of the non-abstract **Calculator** class to the previous **abstract Calculator**.

Abstract classes are nice for situations where you want to have some default behavior, specify what the public interface of a class is, and support polymorphism. However, there are some limitations in that a C# class can have only one base class. Additionally, a struct can't inherit another class or struct, so they don't help if you need to write code that allows you to replace any number of value types with a base class implementation. There is an alternative, which I'll discuss next.

Exposing Interfaces

If you only wanted a base class that specified an interface for a common set of operations, you could create an abstract class with only abstract methods. This ensures that all derived classes have those abstract methods. However, there's a better alternative, named after what it does: an **interface**.

The benefit of the **interface** type is that both **class** and **struct** types can inherit multiple interfaces. You can also implement polymorphism with interfaces. They don't have any implementation and you must write the implementation in your derived class. The following code listing shows the **Calculator** class rewritten as an interface.

```
public interface ICalculator
{
    double Add(double num1, double num2);
}
```

Code Listing 65

Instead of **class** or **struct**, you'll use the **interface** type. A common convention for interface identifiers is the **I** prefix, as in **ICalculator**. Interface methods are implicitly public and virtual, so you don't need access, abstract, or virtual modifiers. Like **abstract** methods, **interface** methods have a signature, but don't have an implementation. Developers provide that implementation in their classes that derive from interfaces. The following code sample is a revision of the previous classes to implement the **ICalculator** interface.

```
public class ScientificCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ProgrammerCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

Code Listing 66

Deriving from an interface uses the same syntax as deriving from a class in that you add a colon and interface name after the class name. Unlike virtual methods, you don't use an **override** modifier on methods.

A derived class implementation must be **public**. This make sense because an interface defines a contract that any derived class will have members defined in the interface. That means any time you use a class through its interface, you know that it will have the members defined by an interface. The following code example is a modification of the **Main** method that uses the **ICalculator** interface.

```
using System;
```

```
public class Program
{
    public static void Main()
    {
        ICalculator sciCalc = new ScientificCalculator();
        double sciCalcResult = sciCalc.Add(2, 5);
        Console.WriteLine($"Scientific Calculator 2 + 5: {sciCalcResult}");

        ICalculator prgCalc = new ProgrammerCalculator();
        double prgCalcResult = prgCalc.Add(5, 10);
        Console.WriteLine($"Programmer Calculator 5 + 10: {prgCalcResult}");

        Console.ReadKey();
    }
}
```

Code Listing 67

The only syntax difference between this example and the one previous to that is the compile-time type of **`sciCalc`** and **`prgCalc`** is **`ICalculator`**. Because each variable is an **`ICalculator`**, you can be guaranteed that the runtime type implements the members of that interface.

Interfaces can also inherit other interfaces. In that case, derived classes must implement all members of each interface in the inheritance chain. Also, a **`class`** or **`struct`** can implement multiple interfaces, which is demonstrated in the following sample.

```
public interface ICalculator { }
public interface IMath { }

public class ScientificCalculator : ICalculator, IMath
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

Code Listing 68

After the first interface, additional interfaces appear in a comma-separated list. A class or struct must implement the methods of all interfaces it is derived from.

Object Lifetime

The lifetime of a value type (**`struct`** or **`enum`**) depends on where it's allocated. Parameter and variable value type instances reside on the stack and exist for as long as they are in scope. Reference type instances (**`class`**) begin life when their constructors execute. The CLR allocates their space on the managed heap, and they exist until the CLR garbage collector (GC) cleans them up.

You can use constructors to initialize a class. While doing so, you can also affect initialization of static state, base types, and other constructor overloads. The following demo shows several features of class initialization.

```
using System;

public class Calculator
{
    static double pi = Math.PI;
    double startAngle = 0;

    public DateTime Created { get; } = DateTime.Now;

    static Calculator()
    {
        Console.WriteLine("static Calculator()");
    }

    public Calculator()
    {
        Console.WriteLine("public Calculator()");
    }

    public Calculator(int val)
    {
        Console.WriteLine("public Calculator(int)");
    }
}
```

Code Listing 69

Calculator has a **static** constructor and two instance constructor overloads. A **static** constructor executes one time for the lifetime of the object and before the first constructor executes. The following sample is a derived class with similar members.

```
using System;

public class ScientificCalculator : Calculator
{
    static double pi = Math.PI;
    double startAngle = 0;

    static ScientificCalculator()
    {
        Console.WriteLine("static ScientificCalculator()");
    }

    public ScientificCalculator() : this(0)
    {
        Console.WriteLine("public ScientificCalculator()");
    }

    public ScientificCalculator(int val)
    {
```

```
        Console.WriteLine("public ScientificCalculator(int)");
    }

    public ScientificCalculator(int val, string word) : base(val)
    {
        Console.WriteLine("public ScientificCalculator(int, string)");
    }

    public double EndAngle { get; set; }
}
```

Code Listing 70

ScientificCalculator derives from **Calculator** and has similar constructors, except for the **this** and **base** operators. Using the **this** operator calls the constructor overload with the matching parameters. Since **0** is an **int**, the default (no parameter) constructor calls **ScientificCalculator(int val)** first. The **base** operator calls the matching constructor in the base class, so calling **base(0)** calls **Calculator(int val)** first. The following code listing is a program that instantiates these classes.

```
using System;

class Program
{
    static void Main()
    {
        var calc1 = new ScientificCalculator();

        var calc2 = new ScientificCalculator(0, "x")
        {
            EndAngle = 360
        };

        Console.ReadKey();
    }
}
```

Code Listing 71

And here is the program's output:

```
static ScientificCalculator()

static Calculator()

public Calculator()

public ScientificCalculator(int)

public ScientificCalculator()

public Calculator(int)
```

public ScientificCalculator(int, string)

Viewing the output, you can see what executes first. Here are the rules that govern the instantiation of these classes:

- Static constructors execute before instance constructors.
- Static constructors execute one time for the life of the program.
- Base class constructors execute before derived class constructors.
- The **this** operator causes an overloaded constructor that matches the **this** parameter list to execute first.
- The base class default constructor executes, unless the derived class uses base to explicitly select a different base class constructor overload.
- This is not shown in the output of the previous example, but static fields initialize before the static constructor and instance fields initialize before instance constructors.
- Auto-implemented property initializers, such as **Created**, initialize at the same time as fields.
- Properties in object initialization syntax execute last as object initialization syntax is equivalent to populating the property through the instance variable after instantiation.

Note: In Visual Studio, you can set break points in the code and use the Immediate Window to inspect field values. You can experiment with different object initialization scenarios to get a feel for the initialization sequence.

Of all these lifecycle events, garbage collection (GC) is the least predictable. The CLR optimizes resources and runs GC when it needs to. This means that the lifetime of a reference type object is non-deterministic. There's a rich body of theoretical discussion around the how and why of GC, but I'll restrict that debate to the practical consideration of resource management. This includes closing files, database connections, operating system handles, and more.

To release resources, there's a pattern commonly referred to as the Dispose Pattern. It relies on the **IDisposable** interface, flags that manage the disposal state of the object, and a destructor. The following code has constructor and **Dispose** method comments that imply a scenario where the class could be logging operations during its lifetime, and the log should be opened during instantiation and closed when the object is no longer needed.

```
using System;

public class Calculator : IDisposable
{
    static Calculator()
    {
        // Initialize log file stream.
    }

    #region IDisposable Support
    private bool disposedValue = false; // To detect redundant calls.

    protected virtual void Dispose(bool disposing)
    {
        if (!disposedValue)
```

```
        {
            if (disposing)
            {
                // TODO: dispose managed state (managed objects).
                // Close log file stream.
            }

            // TODO: free unmanaged resources (unmanaged objects) and override a
finalizer below.
            // TODO: set large fields to null.

            disposedValue = true;
        }
    }

    // TODO: override a finalizer only if Dispose(bool disposing) above has code to
free unmanaged resources.
    // ~Calculator() {
    //   // Do not change this code. Put cleanup code in Dispose(bool disposing) above.
    //   Dispose(false);
    // }

    // This code added to correctly implement the disposable pattern.
    public void Dispose()
    {
        // Do not change this code. Put cleanup code in Dispose(bool disposing) above.
        Dispose(true);
        // TODO: uncomment the following line if the finalizer is overridden above.
        // GC.SuppressFinalize(this);
    }
    #endregion
}
```

Code Listing 72

The code between **#region** and **#endregion** is automatically generated by VS. To generate this code, select **IDisposable** in the editor and the Quick Action icon (a light bulb) will appear. Open the Quick Action menu and select **Implement interface with Dispose pattern**. The **#region** and **#endregion** let VS fold the code so you won't have to see it in the editor.

The **Calculator** class implements the **IDisposable** interface, which is only the **Dispose** method. The constructor initializes a resource you want to open, like a file handle or database, and the GC calls the destructor, **~Calculator()**, if it's uncommented. The **Dispose()** method calls **Dispose(bool)** with a **true** argument and **~Calculator()** calls **Dispose(bool)** with a **false** argument. This lets **Dispose(bool)** know whether it should clean up managed resources that belong to the CLR or unmanaged resources that belong to the operating system. The flag **disposedValue** helps to prevent the object from being disposed more than one time.

The following sample shows how calling code can use this class, disposing it when it is no longer needed.

```
        ScientificCalculator calc3 = null;
```

```
try
{
    calc3 = new ScientificCalculator();
    // Do stuff.
}
finally
{
    if (calc3 != null)
        calc3.Dispose();
}
```

Code Listing 73

This shows the reason **try-finally** exists, to guarantee that resources can be closed or disposed. Because the **finally** block is guaranteed to execute after code in the **try** block starts, **calc3** can be safely disposed. While that works, it's more verbose than necessary. The following listing simplifies the code.

```
using (var calc4 = new ScientificCalculator())
{
    // Do stuff.
}
```

Code Listing 74

The **using** statement accepts parameters with any type that implements **IDisposable**. It takes care of calling **Dispose()** after the block completes execution. Behind the scenes, the logic is similar to the previous **try-finally** block.

Summary

C# supports object-oriented programming. For inheritance, you have single inheritance for classes, multiple inheritance for interfaces, and structs that can only inherit interfaces. Use abstract classes for classes that shouldn't stand alone, but provide interface and structure to derived classes. Use interfaces when you don't have implementation, need value type (struct) polymorphism, or need to implement multiple interfaces. I also discussed structs and how they are ideal for situations where copy by value leads to performance gains and value type semantics make sense. Unlike interfaces that you need to be public, use encapsulation to hide the internal workings that you don't want other developers to use in their code. Polymorphism is a powerful concept that allows you to write a single algorithm that is coded the same for every instance, yet allows each instance to vary with an implementation specific to the runtime type of the instance. Pay attention to the sequence of object instantiation to ensure your types initialize correctly. If you need to dispose a type, make that type implement **IDisposable** with the Dispose Pattern. You can use a **using** statement to simplify the instantiation and safe cleanup of the type.

Chapter 5 Handling Delegates, Events, and Lambdas

Much of the user interface work you'll do is event based, and C# supports this through type members called events. For events to work, you need some infrastructure to specify methods that can be called, which surfaces through delegates and lambdas. This chapter will explain how delegates, events, and lambdas work in C#.

Referencing Methods with Delegates

Delegates have a few capabilities in C#: referencing methods, dispatching multiple methods, asynchronous execution, and event typing. This can be confusing because many other language features serve only a single purpose. Differentiating and comparing all of these capabilities of C# delegates adds complexity that you might not be familiar with. This discussion is going to cut the feature list somewhat to hopefully illuminate delegates and make them less complex as you move forward with practical implementation. In particular, I'll focus on delegates as method references and event types.

Note: I'll avoid deep discussion of delegate multi-cast and asynchronous execution because they're rarely used and largely replaced by other language features. For example, events support multi-cast dispatch and C# 5.0 introduces a capability referred to as async.

Let's first examine the role of a delegate as a reference to a method. To do this, the delegate specifies the signature of a method that it can reference, like this:

```
public delegate double Add(double num1, double num2);
```

Code Listing 75

You might notice that a delegate looks like an abstract method, except it has the **`delegate`** type definition keyword. A **`delegate`** definition is a reference type, just like a class, struct, or interface. The previous **`delegate`** definition is for a delegate type named **`Add`** that takes two **`double`** arguments and returns a result of type **`double`**. Just like other types, delegate accessibility can only be **`public`** or **`internal`** and is **`internal`** by default.

There are esoteric uses of delegates that I won't get into, but I do want to focus on the most practical and common way to use delegates: as event types.

Firing Events

Events are type members that allow a type to notify other types of things that have happened. A very common example is a user interface with a button. You'll want to write code that does something when a user clicks that button. Here are the pieces you need to make that happen:

1. A **`Button`** class, which is typically supplied by the UI technology you're using.
2. An event member of the **`Button`** class, named **`Clicked`**.
3. The UI technology takes care of knowing when that **`Clicked`** event should fire. I'll use a **`SimulateClick`** method in an upcoming code listing.
4. The event has a delegate type. Only methods that conform to the signature of that delegate type can assign to this delegate.
5. Your code defines a method to be called when that event fires.
6. The method you write must have a signature that matches the delegate type of the event. If the method signature doesn't match the event delegate type signature, the compiler won't let you assign that method to the delegate.

As you can see, delegates have a lot of moving parts. In particular, pay attention to #6. Delegates prevent you, or anyone, or anything from assigning an arbitrary method to an event. Here's an example that defines a delegate and a class with an event of that delegate type.

```
using System;

public class ClickEventArgs : EventArgs
{
    public string Name { get; set; }
}

public delegate void ClickHandler(object sender, ClickEventArgs e);

public class CalculatorButton
{
    public event ClickHandler Clicked;
}
```

Code Listing 76

An event can be a member of a class, struct, or interface. If it is an interface member, it means that classes or structs that implement that interface must also have the event in their definitions. An event has the **`event`** modifier and adheres to the same accessibility rules as other type members like methods and properties.

If a delegate serves the purpose you need, you can use it. In fact, the FCL includes many reusable types, including reusable delegate types that you can use without needing to create your own. There's even a .NET type named **`EventHandler`** that nearly matches the signature of **`ClickHandler`**, where the **`sender`** is typically the source of the event, and **`EventArgs`** is a base class you can derive from to create your own custom type for sharing information with methods and event calls when fired. The previous code, with **`ClickEventArgs`**, derives from **`EventArgs`**, a type that comes with the .NET Framework. The following example simulates an event to demonstrate how to write a method that handles that event in code.

```
using System;

public class CalculatorButton
{
    public event ClickHandler Clicked;
    public void SimulateClick()
    {
        if (Clicked != null)
        {
            ClickEventArgs args = new ClickEventArgs();
            args.Name = "Add";

            Clicked(this, args);
        }
    }
}

public class Program
{
    public static void Main()
    {
        Program prg = new Program();
        CalculatorButton calcBtn = new CalculatorButton();

        calcBtn.Clicked += new ClickHandler(prg.CalculatorBtnClicked);
        calcBtn.Clicked += prg.CalculatorBtnClicked;

        calcBtn.SimulateClick();

        Console.ReadKey();
    }

    public void CalculatorBtnClicked(object sender, ClickEventArgs e)
    {
        Console.WriteLine(
            $"Caller is a CalculatorButton: {sender is CalculatorButton} and is named
{e.Name}");
    }
}
```

Code Listing 77

Again, this example has a lot of moving parts, but they follow the list at the start of this section about defining and using events. First, notice that **`CalculateButton`** has a new method, **`SimulateClick`**. Since we simplified the code by avoiding the UI, we have to fake a user clicking a button. That said, **`SimulateClick`** demonstrates the proper way to fire an event in your own code. Before firing an event, make sure that a user has assigned methods to the event by checking for **`null`**. Whenever no methods are assigned, the event will be **`null`**. **`SimulateClick`** sets up the **`ClickEventArgs`** parameter. In this case, it's only a **`Name`** property, but you would provide any relevant information available for the **`EventArgs`** type you were using and what information a method that received this event might need. Next, fire the event by calling it like a method. This causes the event to call each method assigned to it, one by one, in the order they were assigned. The first parameter is the **`this`** keyword, which indicates that the

instance of the containing type, **CalculatorButton**, gets passed to the methods. The second is the **ClickEventArgs** variable that was previously instantiated and had its **Name** property set.

The **Main** method shows how to assign methods to events. Notice the **+=** operator is used twice to assign two methods to the **CalculateButton** instance, **calcBtn**, and **Clicked** event. The **CalculatorBtnClicked** method is an instance method, so the **prg** instance provides access to that method during assignment.

The first assignment creates a new instance of the **ClickHandler** delegate. Delegates are types and you can instantiate them. You instantiate delegates with a method, which becomes the method the delegate refers to. Remember how I explained that delegates are references to methods? In this case, the new instance of the **ClickHandler** delegate refers to the **CalculatorBtnClicked** method. The second assignment shows a newer and simpler syntax for accomplishing the same task as the first; it just uses the method name. This is called delegate inference and means that since the method assigned to the event has the same signature of the event's delegate, the C# compiler will take care of instantiating the delegate with that method behind the scenes for you.

Finally, calling **SimulateClick** on the **CalculatorButton** instance, **calcBtn**, fires the event as explained previously. Regardless of whether the program assigns the same method to the event twice, firing the event causes all assigned delegates to fire, which calls the methods assigned to each delegate to execute. Therefore, the **CalculatorBtnClicked** method will execute two times.

You might wonder why I had to define **SimulateClick** inside of **CalculatorButton** instead of just firing the event from **Main**. The reason is because external code can't fire an event. An event can only be fired from inside of its containing type.

Instead of assigning named methods, you can assign code blocks directly to events.

Working with Lambdas

A lambda is a nameless method. Sometimes you have a block of code that serves one specific purpose and you don't need to define it as a method. Methods are nice because they let you modularize your code and have something to refer to with delegates and call from multiple places. But a lot of times you just need to run some code for a specific operation. Lambdas are quick and simple ways to assign and execute a block of code.

Note: A lambda is also a very sophisticated language feature that lets you translate between parse trees and code. This is a core feature of Language Integrated Query (LINQ), which I'll discuss more in Chapter 7. For daily practical use, working with lambdas as parse trees is rare.

Just like methods, lambdas can have parameters, a body, and can return values. The following code listing is an example of a lambda.

```
using System;

public class Program
{
    public static void Main()
    {
        Action hello = () => Console.WriteLine("Hello!");
        hello();

        Console.ReadKey();
    }
}
```

Code Listing 78

Action is a reusable delegate in the .NET Framework, and **hello** is variable of the **Action** delegate type. The lambda starts with an empty parameter list, meaning no parameters. The **=>** operator separates the parameter list and body and is referred to as either “such that” or “goes to”. Next, you see the body, which is a single statement. Since hello is a delegate, you can call it just like a method and it will execute the lambda, which prints “Hello!” to the console.

With a single statement, you don’t need to use curly braces on the body, but you do with multiple statements, as in the following example.

```
using System;

public class Program
{
    public static void Main()
    {
        Predicate<string> validator =
            word =>
            {
                int count = word.Length;
                return count > 3;
            };
        ValidateInput(validator);
        ValidateInput(word =>
        {
            int count = word.Length;
            return count > 3;
        });

        Console.ReadKey();
    }




    public static void ValidateInput(Predicate<string> validator)
    {
        string input = "Hello";
```

```
        bool isValid = validator(input);
        Console.WriteLine($"Is Valid: {isValid}");
    }
}
```

Code Listing 79

The previous example assigns a lambda to the .NET Framework's **`Predicate`** delegate, which is designed to return a **`bool`**. The lambda has a single parameter, **`word`**, of type **`string`**. Since the lamba has more than one statement, it requires curly braces. The example shows how to pass the lambda both as a variable and as an entire lambda.

`Predicate` is a generic delegate. The type parameter is set to **`<string>`**, meaning that the lambda parameter is type **`string`**. You'll learn more about generics in Chapter 6.

The **`ValidateInput`** method passes a **`string`** to **`validator`** and assigns the results to the **`isValid`** variable. It's just like a method call, except there isn't a method, just code; it is quick to write and limited in scope.

Another way to use lambdas is with events. The following example shows a different way to write an event handler method for the **`Clicked`** event in the previous **`CalculatorButton`** example.

```
using System;

public class Program
{
    public static void Main()
    {
        CalculatorButton calcBtn = new CalculatorButton();

        calcBtn.Clicked += (object sender, ClickEventArgs e) =>
        {
            Console.WriteLine(
                $"Caller is a CalculatorButton: {sender is CalculatorButton} and is
named {e.Name}");

            Console.WriteLine(message);
        };

        calcBtn.SimulateClick();

        Console.ReadKey();
    }
}
```

Code Listing 80

The first thing you might notice in this example is that the delegate assignment to the **`Clicked`** event is now a lambda. If you have two or more parameters, they must be enclosed in parentheses as a comma-separated list. If the body of the lambda includes two or more lines,

they must be terminated with semicolons and enclosed in braces. Notice that the signature of the lambda matches the **`ClickEventHandler`** defined previously.

More FCL Delegate Types

In addition to the **`Action`** and **`Predicate<T>`** delegates you've seen in previous examples, the FCL has a set of delegates named **`Func<T>`** that you can reuse at will. Here's an example that rewrites the previous example, using **`Func<T, TResult>`** instead of **`Predicate<T>`**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

class Program
{
    public static void Main()
    {
        Func<string, bool> validator =
            word =>
            {
                int count = word.Length;
                return count > 3;
            };
        ValidateInput(validator);
        ValidateInput(word =>
        {
            int count = word.Length;
            return count > 3;
        });

        Console.ReadKey();
    }

    public static void ValidateInput(Func<string, bool> validator)
    {
        string input = "Hello";
        bool isValid = validator(input);
        Console.WriteLine($"Is Valid: {isValid}");
    }
}
```

Code Listing 81

This is nearly identical to the previous program, except it uses **`Func<string, bool>`** instead of **`Predicate<string>`**. As mentioned previously, both **`Func<T, TResult>`** and **`Predicate<T>`** are generic delegates. The type specifications in angle brackets are plug-ins for types applied to parameters and return types. The following listing shows the **`Predicate<T>`** delegate as defined in the FCL.

```
public delegate bool Predicate<T>(T obj);
```

Code Listing 82

It refers to a method that returns a **bool**, but accepts a parameter of type **T**. So, **Predicate<string>** means that the parameter to the method referred to is a **string**. Similarly, here's the FCL definition of **Func<T, TResult>**.

```
public delegate TResult Func<T, TResult>(T arg);
```

Code Listing 83

This shows that **Func<T, TResult>** accepts a parameter of type **T** and returns a value of type **TResult**. In Code Listing 81, **Func<string, bool>** refers to a method with a parameter of type **string** that returns type **bool**.

For convenience, the FCL offers 18 overloads of the **Func** delegate, allowing between 0 and 16 input parameters and 1 return parameter type. This covers many scenarios, and you can reuse the provided delegates from the FCL to go a long way. You can create your own delegates only when you need to.

Expression-Bodied Members

While not necessarily lambdas, expression-bodied members give you some shorthand syntax for properties and methods. The following listing provides an example.

```
using System;

class Program
{
    public static string Today => DateTime.Now.ToShortDateString();

    public static void Log(string message) => Console.WriteLine(message);

    public static void Main()
    {
        Log($"{Today} is a good day.");

        Console.ReadKey();
    }
}
```

Code Listing 84

The **Program** class defines the **Today** property and **Log** method as expression-bodied members. **Main** shows how these members are used like normal methods and properties.

Summary

You learned about delegates, events, and lambdas. A delegate is a reference to a method. You can pass a delegate around in code or assign it to an event. The method a delegate refers to must have the same signature of the delegate. An event is a type member, defined with a delegate type. You can assign as many delegates to an event as you need and each one executes when the event fires. You can only fire an event from within the type the event is defined in. Instead of methods, you can use lambdas when you don't need to define a separate method. You can refer to a lambda with a delegate, pass a lambda as a parameter, or assign a lambda to an event. Expression-bodied members let you write properties and methods with a shorthand syntax.

Chapter 6 Working with Collections and Generics

You've seen arrays in previous chapters and they can be useful in scenarios where you need a fixed size, strongly typed list of objects. However, there are many times when you need to organize objects into different types of data structures like lists, queues, stacks, and dictionaries. These capabilities are available to C# developers via collection classes in the .NET Framework.

A core part of working with collections is the use of generics, which allow you to use parameterized code. This allows you to strongly type your collections. You can even write your own code that uses generics, allowing you to create strongly typed reusable libraries.

Note: The first version of .NET offered a collection library based on `Object` which isn't strongly typed. Since all .NET types are assignable to `Object`, this worked. However, you had to write a lot of code that used cast operators to convert from `Object` back to the type you added to the collection. Generics solves this problem, and using generic collections is standard practice in .NET today.

Using Collections

.NET collection classes let you work with data in many different ways. Instead of an array, you can use a **`List`**. If you need a first-in first-out set of items, you can use a **`Queue`**. If you need to work with items that have unique IDs, you can use a **`Dictionary`**. With generics, you can build your own collection to manage data any way that you need.

Tip: Check out the System.Collections.Generic namespace before writing your own collection; you might find that what you need is already written.

A common collection is a **`List`**, which is a nice replacement for an array. The following listing provides an example.

```
using System;
using System.Collections.Generic;

public class Company
{
    public string Name { get; set; }
}

public class Program
```

```
{
    public static void Main()
    {
        List<string> names = new List<string>();
        names.Add("Joe");
        names.Insert(0, "Car");
        names.Add("Jill");
        names[0] = "Building";
        names.RemoveAt(0);
        Console.WriteLine($"First name: {names[0]}");

        IList<Company> companies = new List<Company>
        {
            new Company { Name = "Syncfusion" },
            new Company { Name = "Microsoft" },
            new Company { Name = "Acme" }
        };

        foreach (Company cmp in companies)
            Console.WriteLine(cmp.Name);
        Console.ReadKey();
    }
}
```

Code Listing 85

The previous program demonstrates the versatility of collections. You have a **`List`** of **`string`**, which only holds objects of type **`string`**. This is a generic collection, meaning that its type parameters, inside **`<`** and **`>`**, specify the type of object the collection works with. Each item added is appended to the list and the list grows dynamically. The **`Insert`** operation adds a new **`string`** at the first position of the list and pushes down the first, **`"Joe"`**, into the second position at index 1. The second **`Add`** puts **`"Jill"`** at index 2. Notice how you can use indexer (array-like) syntax to access elements of the list. The **`RemoveAt`** deleted the string at the first index of the collection, moving **`"Joe"`** to 0 and **`"Jill"`** to 1.

The second **`List`** in **`Main`** shows how to use custom types. Since **`List`** derives from **`IList`**, you can assign the instance to that interface. This is convenient because it means you can create code that operates on an **`IList`**; whether the caller passes in a **`List<T>`** or any other collection type that derives from **`IList`**, your code will still work.

The example also uses collection initialization syntax, where you can instantiate a comma-separated list of the collection type that populates the **`List`**. The **`foreach`** statement iterates through the collection, printing each item.

The previous example uses a **`foreach`** loop, but you could also use the **`ForEach`** method of **`List`**, as shown in the following example.

```
        List<Company> companyList = companies as List<Company>;
        companyList.ForEach(cmp => Console.WriteLine(cmp.Name));
```

Code Listing 86

The first line uses the **as** operator to convert the **IList<Company>** to **List<Company>**. With an instance of **List<T>**, you can call the **ForEach** method, which takes a lambda parameter. This lambda executes for each of the items in the **List<T>** and the lambda parameter, **cmp**, contains the current item.

This should give you an idea of how **List** works. There are more methods available that you can learn about by reading the documentation for the **List** class.

Another useful collection is a **Dictionary**. It works like a hash table where you store and retrieve objects by index as shown in the following sample.

```
using System;
using System.Collections.Generic;

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}

public class Program
{
    public static void Main()
    {
        Dictionary<int, Customer> customers = new Dictionary<int, Customer>();
        Customer jane = new Customer { ID = 0, Name = "Jane" };
        Customer joe = new Customer { ID = 1, Name = "Joe" };
        customers.Add(jane.ID, jane);
        customers[joe.ID] = joe;

        foreach (int key in customers.Keys)
            Console.WriteLine(customers[key].Name);

        Dictionary<int, Customer> customers2 = new Dictionary<int, Customer>
        {
            [0] = new Customer { ID = 0, Name = "Chris" },
            [1] = new Customer { ID = 1, Name = "Alex" }
        };

        Console.ReadKey();
    }
}
```

Code Listing 87

A **Dictionary** in the previous example takes two type parameters for the key and value, respectively. The first example instantiates the dictionary to take an **int** key and **Customer** value. The **Customer** class has two properties, where the **ID** will be used as a key for the dictionary. Notice the two different ways you can add values to a dictionary, via the **Add** method or indexer assignment. The first parameter to **Add** is the index and the second is the value. When using the indexer, put the index in brackets and assign the value. Just as in other collections, there are many methods available and you should review the documentation of that collection.

The **foreach** loop shows how to iterate through **Dictionary** items. A **Dictionary** has a **Keys** property, which is a collection of keys, and a **Values** property, which is a collection of values (the **Customer** instances in the previous example). Notice how the loop uses the indexer **customers[key]** to access the value associated with each key.

The second **Dictionary** in the example shows how to use the dictionary initializer syntax. Just assign the value to the index that matches the key for that value.

Writing Generic Code

One of the primary applications of generics is to support collections. In the previous section, you saw how to use collections. You could also write your own collection class. If you wrote a generic linked list, you would need a **Node** class to hold an object and reference the next in the list, and a **LinkedList** collection class that performed list operations. The **Node** class in the following listing contains an object instance.

```
class Node<T>
{
    public T Item { get; set; }
    public Node<T> Next;

    public Node(T item)
    {
        Item = item;
    }
}
```

Code Listing 88

The **<T>** syntax makes the **Node<T>** class generic. Whenever code instantiates a **Node**, it specifies a type that replaces **T**. Anywhere you're using an object of that type, specify **T**. **Node<T>** doesn't have an access modifier because it's only used with the code inside this assembly and the default internal accessibility is appropriate. The following sample shows how to instantiate a **Node<T>**.

```
Node<string> name = new Node<string>("May");
```

Code Listing 89

Here, you see **Node<string>** as the type, meaning that all of the places you see **T** inside of the **Node** class are now **string**. You're protected from passing an **int**, **decimal**, or any other type to the constructor of this class because it will only hold a **string**. It is strongly typed.

Next, you need a collection that holds **Node<T>** instances as a linked list, as shown in the following listing.

```
Using System;
using System.Collections;
using System.Collections.Generic;

public class LinkedList<T> : IList<T>
{
    Node<T> head;
    Node<T> tail;

    public void Add(T item)
    {
        var node = new Node<T>(item);

        if (head == null)
            head = node;
        else
            tail.Next = node;

        tail = node;
    }

    // Other IList members…
}
```

Code Listing 90

The **LinkedList** class is generic and holds items of the type it's instantiated as. The **IList<T>** interface belongs to the FCL and facilitates creating collections. As you would expect with interfaces, developers who write code to the **IList** interface can use this collection too. The **LinkedList** class implements all the members of the **IList<T>** interface, as it must.

Add is a minimal implementation, but illustrates some concepts of working with generics. Even though the code instantiates a new **Node<T>**, the actual type will be the same as the type that **LinkedList** is defined as. The same concept applies to the interface where **IList<T>** becomes the same type as **LinkedList**. The following example instantiates a **LinkedList<T>**.

```
public class Program
{
    public static void Main()
    {
        var llist = new LinkedList<string>();
        llist.Add("Jamie");
        llist.Add("Ron");
        //...

        Node<string> name = new Node<string>("May");
    }
}
```

Code Listing 91

This shows that you instantiate and use your generic collection like any other collection. Just supply the type during instantiation and the collection will work with objects of that type.

Any place you see the **`object`** type being used is a potential candidate for creating a generic type. All types inherit the **`object`** type, which is why you'll see types in the FCL and elsewhere work with object type values.

You can also create generic methods. The following example shows a couple factory methods where one is type **`object`** and the other is generic.

```
using System;

public class CustomerReport
{
    public DateTime Date { get; set; }
}

public class OrdersReport
{
    public DateTime Date { get; set; }
}

public class ReportFactory
{
    public static object Create(Type reportType)
    {
        switch (reportType.ToString())
        {
            case "CustomerReport":
                var custRpt = new CustomerReport();
                custRpt.Date = DateTime.Now;
                return custRpt;
            default:
            case "OrdersReport":
                var ordsRpt = new OrdersReport();
                ordsRpt.Date = DateTime.Now;
                return ordsRpt;
        }
    }
}

public class Program
{
    public static void Main()
    {
        var rpt = (CustomerReport)ReportFactory.Create(typeof(CustomerReport));
        Console.ReadKey();
    }
}
```

Code Listing 92

What you should get out of the previous **`ReportFactory`** implementation is that there's a lot of duplication in the code and the use of cast and **`typeof`** operators in the **`Main`** method includes

more syntax than necessary. You can probably see where this code might become less maintainable with more complexity. The following example shows how to refactor the **`Create`** method into a generic method.

```
using System;

public abstract class Report { }

public class CustomerReport : Report
{
    public DateTime Date { get; set; }
}

public class OrdersReport : Report
{
    public DateTime Date { get; set; }
}



public class ReportFactory
{
    public static TReport Create<TReport>()
        where TReport : Report
    {
        switch (typeof(TReport).Name)
        {
            case "CustomerReport":
                var custRpt = new CustomerReport();
                custRpt.Date = DateTime.Now;
                return (TReport)(Report)custRpt;
            default:
            case "OrdersReport":
                var ordsRpt = new OrdersReport();
                ordsRpt.Date = DateTime.Now;
                return (TReport)(Report)ordsRpt;
        }
    }
}

public class Program
{
    public static void Main()
    {
        var rpt2 = ReportFactory.Create<CustomerReport>();

        Console.ReadKey();
    }
}
```

Code Listing 93

The **`Create`** method has a new type parameter, **`TReport`**. You've seen the use of just **`T`** in previous examples, but sometimes—as in **`Dictionary<TKey`** and **`TValue>`**—you have to

differentiate between multiple type parameters or make the code more self-documenting. The return type is now strongly typed too. The code is able to cast from the derived type to **`Report`**, and then to **`TReport`** to return the proper type. This is allowed because of the generic constraint, where **`TReport : Report`** says that **`TReport`** must derive from **`Report`**. The calling code is much simpler.

The **`Create<TReport>`** method is still longer than it has to be and contains too much duplication. We can solve that problem with generic constraints. A constraint does what the name implies: it limits how generic a type can be. You saw the base class constraint on **`Report`** in the previous code. The following table describes all available constraints.

Table 3: Generic Type Constraints

Constraint	Description
interface	Type must implement specified interfaces.
base class	Type must derive from specified base class.
class	Type must be a reference type.
struct	Type must be a value type.
new	Type must have a default (no parameter) constructor.

We need two constraints to simplify our code: **`interface`** and **`new`**. The following example shows how they can be used.

```
using System;

public interface IReport
{
    DateTime Date { get; set; }
}

public class CustomerReport : IReport
{
    public DateTime Date { get; set; }
}

public class OrdersReport : Report, IReport
{
    public DateTime Date { get; set; }
}

public class ReportFactory
{
    public static TReport Create<TReport>()
        where TReport : IReport, new()
    {
        return new TReport() { Date = DateTime.Now };
    }
```

```
}

public class Program
{
    public static void Main()
    {
        var rpt2 = ReportFactory.Create<CustomerReport>();

        Console.ReadKey();
    }
}
```

Code Listing 94

In this demo, there's a new interface, **`IReport`**, which **`CustomerReport`** and **`OrdersReport`** derive from. Since we know the classes we expect are **`IReport`**, we can make assumptions about the type and write code that operates on any **`IReport`**.

The **`Create<TReport>`** method has additional syntax following the method signature. To specify a constraint, follow the **`where`** keyword with the type being constrained, append a semicolon, and then add a comma-separated list of constraints from the previous table. This example uses an interface and **`new()`** constraint. The **`new()`** constraint means we can create a new instance of a type, **`new TReport()`**. Further, since the type is an **`IReport`**, we know it has a **`Date`** property and can populate its **`Date`** property. Gone are the duplication and excessive code, simplified by generic code in both implementation and use.

Tip: You can also create generic delegates. As usual, you should seek to reuse typ already present in the FCL. A popular reusable delegate in the .NET Framework is EventHandler<TEventArgs>. In fact, you can replace all the references to ClickHandler in Chapter 5 with EventHandler<ClickEventArgs> and your code will still work.

Summary

You've seen how to use generics and that they let you write reusable code. The .NET collection classes are more versatile than arrays and allow you to manage your objects in ways that better help the design of your application.

Chapter 7 Querying Objects with LINQ

Language-Integrated Query (LINQ) allows you to query data with a SQL-like syntax. LINQ can be used with many different types of data and both Microsoft and third parties have built LINQ providers to access a wide range of data sources. This chapter narrows that list by showing you how to use LINQ to Objects. Once you know LINQ to Objects, understanding other LINQ providers is easy because of similar syntax.

Getting Started

Before you write any LINQ code, remember to add a **`using`** declaration to the **`System.Linq`** namespace at the top of your file. Each example in this chapter will use the following class, containing collections to work with:

```
using System.Collections.Generic;

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}
public class Order
{
    public int CustomerID { get; set; }
    public string Description { get; set; }
}

public static class Company
{
    static Company()
    {
        Customers = new List<Customer>
        {
            new Customer { ID = 0, Name = "May" },
            new Customer { ID = 1, Name = "Gary" },
            new Customer { ID = 2, Name = "Jennifer" }
        };
        Orders = new List<Order>
        {
            new Order { CustomerID = 0, Description = "Shoes" },
            new Order { CustomerID = 0, Description = "Purse" },
            new Order { CustomerID = 2, Description = "Headphones" }
        };
    }
```

```
    public static List<Customer> Customers { get; set; }
    public static List<Order> Orders { get; set; }
}
```

Code Listing 95

These are collections of objects in memory. To make the collection easier to query, **`Company`** is a **`static`** class, with a **`static`** constructor that initializes **`static`** properties. If you abstract this concept, that data could have been read from a file, database, or REST service. Regardless of the data source or the LINQ provider, the basic LINQ syntax remains the same.

Querying Collections

To query data, you only need the **`from`** and **`select`** keywords. Remember to add a **`using`** clause for the **`System.Linq`** namespace. The syntax looks like SQL, as you can see in the following example.

```
using System;
using System.Linq;
using System.Collections.Generic;

public class Program
{
    public void Main()
    {
        IEnumerable<Customer> customers =
            from cust in Company.Customers
            select cust;

        foreach (Customer cust in customers)
            Console.WriteLine(cust.Name);
    }
}
```

Code Listing 96

LINQ to Objects queries result in a collection of type **`IEnumerable<T>`**. In this case, it's a collection of **`Customer`** objects. The **`from`** keyword specifies a range variable, **`cust`**, which holds each object from the collection. You specify the collection after the **`in`** keyword.

The **`select`** defines what to query. In this example, you're just returning the whole object. In fact, the collection you get is identical to what is in **`Company.Customers`**. This isn't particularly useful in LINQ to Objects, but is very useful if the data was read from an external data source, like a database where you just wanted to get a collection of objects into memory for further manipulation. The **`select`** allows you to reshape the data you get back into various projections. The following is a query that gets the customer name.

```
        IEnumerable<string> customers2 =
            from cust2 in Company.Customers
            select cust2.Name;
```

Code Listing 97

The **select** uses the **cust2** variable to access the **Name**, resulting in a collection of **string** (the **Name** property's type). Sometimes you need a whole different object, where that object might be defined as:

```
public class CustomerViewModel
{
    public string Name { get; set; }
}
```

Code Listing 98

And a new projection could be written as:

```
        IEnumerable<CustomerViewModel> customerVMs =
            from custVM in Company.Customers
            select new CustomerViewModel
            {
                Name = custVM.Name
            };
```

Code Listing 99

Here, **select** instantiates a new **CustomerViewModel**. Then it populates values, using object initialization syntax, to assign the **custVM.Name** to the new object's **Name** property. This results in a collection of type **CustomerViewModel**.

The previous example assumed you needed to work with a specifically typed collection. However, what if you don't care what type the collection is and what if you didn't want to create a new class just to do manipulation in a single algorithm? In that case, you could use an anonymous type, as shown in the following listing.

```
        var customers3 =
            from cust3 in Company.Customers
            select new
            {
                Name = cust3.Name
            };
        foreach (var cust3 in customers3)
            Console.WriteLine(cust3.Name);
```

Code Listing 100

Anonymous types don't have names you can use, even though C# might create an internal name for its own use. To work around this problem, use the **var** keyword as the type. Notice how the projection uses **new** without a type name: an anonymous type. You can define whatever properties you want for an anonymous type; just write them in. Notice also that you can use **var** in the **foreach** loop.

If you need to return a collection from a method, create a new (named) type and project into that. Anonymous types are designed for situations limited to the scope in which they are used. You'll see the **var** keyword used elsewhere in code, but the reason it was added to the language was to support this scenario. The following listing shows a common way to use **var**, other than the previous scenario.

```
var customer = new Customer();
```

Code Listing 101

The previous statement is shorter than specifying the object type of the variable, which is redundant in this case and is obviously **Customer**. However, the following example is less obvious.

```
var response = DoSomethingAndReturnResults();
```

Code Listing 102

The problem in the previous statement is that just reading the code doesn't tell you what type **var** is. You don't know whether it's a single object or a collection. In this case, the code might be more maintainable by specifying the type.

Note: A common misconception is that `var` is dangerous because it behaves like `object`, allowing you to set the variable to any type. This is not true. When you use `var`, the code is still strongly typed. Once you assign a value to a variable of type `var`, you can't assign any other type to that variable. In the previous examples, `customers` is an instance of type `Customer`. You can't write code later to assign an object of another instance type—for example, an `Order` type—to that variable.

Filtering Data

You can filter a collection with the **where** clause, as shown in the following example.

```
        var customers4 =
            from cust4 in Company.Customers
            where cust4.Name.Length > 3 && !cust4.Name.StartsWith("G")
```

```
            select cust4;

        foreach (var cust4 in customers4)
            Console.WriteLine(cust4.Name);
```

Code Listing 103

In the previous listing, a customer's name must be longer than **3**, which filters the list down to **Gary** and **Jennifer**. The clause to the right of the **&&** operator filters that list even further to the name whose first character is not **"G"**.

In LINQ to Objects, you can create complex conditions in the **where** clause using logical operators, parentheses for grouping, and any other logic to filter results. You can even call another method that will evaluate the current object being evaluated. The result of the **where** clause must evaluate to a **bool**. Other LINQ providers might restrict the type of expressions in a **where** clause, so you'll have to review documentation for that particular provider to learn more.

Ordering Collections

In LINQ, the **orderby** clause lets you sort collection results. The following listing demonstrates this.

```
        var customers5 =
            from cust5 in Company.Customers
            orderby cust5.Name descending
            select cust5;

        foreach (var cust5 in customers5)
            Console.WriteLine(cust5.Name);
```

Code Listing 104

In this example, the **orderby** clause sorts the list by the customer name in **descending** order. The default order is ascending, which you'll get by either omitting **descending** or specifying **ascending** instead. The output is:

```
May
Jennifer
Gary
```

Joining Objects

Sometimes you'll have two different collections of objects or related tables in a database and you need to join them together. To do this, use the **join** clause.

```
var customerOrders =
    from cust in Company.Customers
    join ord in Company.Orders
        on cust.ID equals ord.CustomerID
    select new
    {
        ID = cust.ID,
        Customer = cust.Name,
        Item = ord.Description
    };

foreach (var custOrd in customerOrders)
    Console.WriteLine(
        $"Customer: {custOrd.Customer}, Item: {custOrd.Item}");
```

Code Listing 105

After the **from** clause, you can use one or more **join** clauses to access the types you need. The **on** keyword lets you specify the keys to match between tables. This example creates a projection on an anonymous type to create a report based on the joined information. This was a normal join, which omits any **Customers** where there isn't a matching **Order**. The following example lets you do the equivalent of a left join.

```
var customerOrders2 =
    from cust in Company.Customers
    join ord in Company.Orders.DefaultIfEmpty()
        on cust.ID equals ord.CustomerID
    select new
    {

        ID = cust.ID,
        Customer = cust.Name,
        Item = ord.Description
    };

foreach (var custOrd2 in customerOrders)
    Console.WriteLine(
        $"Customer: {custOrd2.Customer}, Item: {custOrd2.Item}");
```

Code Listing 106

The difference here is the call to **DefaultIfEmpty**, which includes the **Customer** with the **Name Gary**, even though there aren't any orders in the join that match his **ID**.

Using Standard Operators

You've seen basic LINQ syntax, but there's much more available in the form of standard query operators. There are literally dozens of standard query operators, and you can view all of them on MSDN at https://msdn.microsoft.com/en-us/library/vstudio/bb397896(v=vs.120).aspx.

The following code listings are a grab bag of examples, demonstrating how to use standard query operators that you might find useful.

So far, you've been working with **IEnumerable<T>**, where **T** is the projected type of the query. There are a set of standard query operators that will return different collection types, including **ToList**, **ToArray**, **ToDictionary**, and more. Here's an example that turns the results into a **List**.

```
var custList =
    (from cust in Company.Customers
     select cust)
    .ToList();
custList.ForEach(cust => Console.WriteLine(cust.Name));
```

Code Listing 107

The previous code enclosed the query in parentheses and then called the **ToList** operator. The **ForEach** method on **List<T>** lets you pass a lambda.

LINQ queries use deferred execution. This means that the query doesn't execute until you execute a **foreach** loop or call one of the standard query operators, like **ToList**, that requests the data.

You've seen how the C# **select**, **where**, **orderby**, and **join** keywords help build queries. Each of these queries have a standard query operator equivalent. These standard query operators use a fluent syntax and give you a different way to perform the same query as their matching language syntax. Some people prefer the fluent style and others prefer the language syntax, but the method you choose is really a personal preference. The following is an example of the **Where** and **Select** operators, which mirror the **where** and **select** language syntax clauses.

```
var customers6 =
    Company.Customers
           .Where(cust => cust.Name.StartsWith("J"));
foreach (var cust6 in customers6)
    Console.WriteLine(cust6.Name);

var customers7 =
    Company.Customers.Select(cust => cust.Name);
foreach (var cust7 in customers7)
    Console.WriteLine(cust7);
```

Code Listing 108

The **Where** lambda must evaluate to a **bool** and the **Select** lambda lets you specify the projection.

You can perform set operations like **Union**, **Except**, and **Intersect**. The following listing is an example of **Union**.

```
var additionalCustomers =
    new List<Customer>
    {
        new Customer { ID = 1, Name = "Gary" }
    };
var customerUnion =
    Company.Customers
           .Union(additionalCustomers)
           .ToArray();
foreach (var cust in customerUnion)
    Console.WriteLine(cust.Name);
```

Code Listing 109

Just pass a compatible collection and **`Union`** will produce a combined collection of all objects. I used the **`ToArray`** operator in this example too, which results in an array of the collection type, **`Customer`**.

There is a useful set of operators for selecting **`First`**, **`FirstOrDefault`**, **`Single`**, **`SingleOrDefault`**, **`Last`**, and **`LastOrDefault`**. The following example demonstrates **`First`**.

```
Console.WriteLine(Company.Customers.First().Name);
```

Code Listing 110

The only thing about using **`First`** this way is the possibility of an **`InvalidOperationException`** with the message "Sequence contains no elements." This sequence contains elements, but this isn't guaranteed. You would be safer using the operator with the **`OrDefault`** suffix, as in the following listing.

```
var empty =
    Company.Customers
           .Where(cust => cust.ID == 999)
           .SingleOrDefault();

if (empty == null)
    Console.WriteLine("No values returned.");
```

Code Listing 111

The previous example writes **`"No values returned."`** Because there isn't a customer with **`ID == 999`**, the **`SingleOrDefault`** returns **`null`**, which is the default value of a reference type object.

These were only a handful of available operators, but hopefully you have a sense for the wealth of support in language syntax as well as the standard query operators that comprise LINQ.

Summary

LINQ allows you to use SQL-like syntax to query data. The LINQ provider used in this chapter is LINQ to Objects, which lets you query objects in memory, but there are many other LINQ providers for other data sources. Use a **`from`** to specify the collection being queried and a **`select`** to shape the results. The **`where`** clause lets you filter results and takes a **`bool`** expression to evaluate if a given object should be included. The **`orderby`** clause lets you sort results. The **`join`** clause lets you combine two collections. Standard query operators extend LINQ and make it even more powerful than the language keywords.

Chapter 8 Making Your Code Asynchronous

In version 5, C# introduced the capability to write and call code asynchronously, commonly referred to as async. To understand async, it's useful to consider the normal behavior of code, which is synchronous. In synchronous code you call a method, wait for it to complete, and move on to the rest of the code. The primary point of this behavior is that the thread calling the synchronous method is also executing the synchronous code in that method. If that synchronous method runs for a long time, your UI will become unresponsive and your users might not know whether the program crashed or if they should just wait.

Asynchronous code improves this situation by allowing the long-running operation to continue on a separate thread, and free your calling thread to resume its responsibilities. When the calling thread is the UI thread, the application becomes responsive again and you can show a status or busy indicators, or let the user operate another part of the program while the asynchronous process runs. When the asynchronous process returns, you can interact with users in some way, if that makes sense for your application. In the past, writing this asynchronous code has been a challenge. Though the task of writing asynchronous code has improved with new patterns and libraries, the C# async features makes asynchronous programming much easier.

There are a couple different perspectives of async that determine how you code: library consumer or library creator. From a consumer perspective, you make assumptions about async code based on an implied contract. However, from the library creator perspective, you have additional responsibilities to ensure your code provides the async contract users expect.

Consuming Async Code

C# has two keywords that support async: **`async`** and **`await`**. Decorating a method with the **`async`** modifier says that the method can contain async code. You use the **`await`** keyword on a **`Task`** to start an async operation.

```
using System.IO;
using System.Threading.Tasks;

public class Program
{
    public static void Main()
    {
        Program.CreateFileAsync("test.txt").Wait();
    }




    public static async Task CreateFileAsync(string filename)
    {
```

```
        using (StreamWriter writer = File.CreateText(filename))
            await writer.WriteAsync("This is a test.");
    }
}
```

Code Listing 112

In the previous program, the **CreateFileAsync** method is asynchronous. You can tell by the **async** modifier on the method. You need to add **using** clauses for the **System.IO** and **System.Threading.Tasks** namespaces for writing to a file and async **Task** support, respectively. The **File** class is part of the FCL and its **CreateText** method returns a **StreamWriter** that you use to write to the file.

Note: Appending a method name with `Async` is not required, but it is a common convention.

The proper way to call an async method is to **await** its **Task** or **Task<T>**. The **WriteAsync** method returns **Task**, which means you can **await** it.

The **using** statement closes the file when its enclosing block completes. In this case, the block is only a single line, so no curly braces are required.

Part of the async contract is an expectation that some code in the library you're using will run the operation on another thread, releasing your thread for other operations; that's what **WriteAsync** does too. So, the thread returns to the code calling this async method. But the caller in this program is the **Main** method, which is calling **Wait()** on the **Task** returned from **CreateFileAsync**. This keeps the program from ending before the thread that's running the async operation completes.

Warning: The previous example is a console application, which doesn't have the underlying infrastructure (referred to as a synchronization context) to manage proper thread handling. Therefore, it was necessary to `Wait()` on the task returned from `CreateFileAsync`. In a normal UI application, you will have a synchronization context, meaning you won't have to worry about the program ending, and won't need to call `Wait()` on an async method. The preferred method of waiting on an async method is via `async` and `await` as shown in the `CreateFileAsync` method. In fact, you should never call `Wait` on an async method. That's because when the second thread returns from doing work on the `async` call, it will attempt to marshal the call back onto the calling thread. If that calling thread is in a synchronous `Wait()`, the thread will be blocked, preventing the second thread from performing that marshaling operation. Then you'll have a deadlock. To prevent deadlock, never call `Wait()`, use `async` and `await` instead.

The **async** modifier is required on the method if you use **await**. If a method has the **async** modifier, but no **await**s, C# will give you a compiler warning and let you know that the method will run synchronously.

Async Return Types

With **async**, you can **await** any awaitable type. The FCL has **Task** and **Task<T>**, which are awaitable and are what you should use in most situations. Returning **Task** means that the method does not return a value, which is what you saw with the previous **CreateFileAsync** method.

Tip: Stephen Toub's blog post "await anything;" explains how to create a custom awaitable type and is a good reference if you see it as a way to improve your code. You can read it at http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx.

Use **Task<T>** when your method returns a value. The following listing shows an example.

```
public async Task<string> ReturnGreeting()
{
    await Task.Delay(1000);
    return "Hello";
}
```

Code Listing 113

Task.Delay is a way to sleep the thread by a number of milliseconds, but I'll be using it in more examples to simplify code and as a placeholder for where you would normally add async code.

The previous example shows a return type of **Task<string>**. The method only returns the string **"Hello"** instead of an instance of **Task<string>** because the C# compiler takes care of that for you.

An async method can return **void** rather than an awaitable type too. This is done in the following listing.

```
public async void SayGreeting()
{
    await Task.Delay(1000);
    Console.WriteLine("Hello");
}
```

Code Listing 114

This method executes asynchronously, but async **void** methods have important caveats you must be aware of: they aren't awaitable, and they don't protect against exceptions, but they are necessary for scenarios like event handling where the method must be **void**.

Since you can only await awaitable types like **Task** and **Task<T>**, there's no way to await an async **void** method. The implication of this is that when a library's code starts another thread, it

allows the calling thread to return. Calling an async **void** method means you can't wait until that method completes and you won't ever know when or if the method completes. As with anything, there are no absolutes and one could argue that it would be possible to write some cross-thread communication mechanism, but I'm referring to the general out of the box behavior, which will lead to some important implications. Because of this behavior, there are pros and cons on when you should use an async **void** method.

The largest problem with async **void** methods is that you can't throw an exception back to the calling code. With **Task** and **Task<T>** returning methods, you can **await** and wrap the async method call in a **try-catch**, but you can't do that with async **void** methods. If an async **void** method throws an unhandled exception, the application will crash.

With such problems, it would be easy to assume that async **void** should not be used at all. However, the C# language designers added async **void** for one specific reason: to support event handling. Event handlers in the .NET Framework follow a pattern where their delegates return **void**. Therefore, you can't use an awaitable type, like **Task** or **Task<T>**, and must assign async **void** methods as event handlers.

In UI applications, a UI control might fire an event, async **void** methods assigned to the event execute, the async code starts a new thread and releases the UI thread, and the UI thread returns and processes messages to keep the UI responsive. So, using async **void** as event handlers is appropriate.

Developing Async Libraries

Writing an async library is mostly normal coding, but the key thing to keep in mind is what is happening to the thread. First, all code executes by default on the calling thread. Second, you need to marshal execution onto a new thread and release the calling thread to the caller.

Understanding What Thread the Code is Running On

The following code doesn't necessarily make any logical sense, but represents the potential structure of some library code that you might write. In particular, it demonstrates what happens with threads before and after the first **await** in your **async** method. In the following code, **UserInfo** is just a type to hold and return data. **UserService** and **AddressService** have async methods that the **GetUserInfoAsync** method calls.

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

public class UserInfo
{
    public string Info { get; set; }
    public string Address { get; set; }
}
```

```
class UserService
{
    internal static async Task<string> GetUserAsync(string user)
    {
        // Do some long running synchronous processing.
        return await Task.FromResult(user);
    }
}

class AddressService
{
    internal static async Task<string> GetAddressAsync(string user)
    {
        return await Task.FromResult(user);
    }
}

public class UserSearch
{
    public async Task<UserInfo> GetUserInfoAsync(string term, List<string> names)
    {
        var userName =
            (from name in names
             where name.StartsWith(term)
             select name)
            .FirstOrDefault();

        var user = new UserInfo();
        user.Info = await UserService.GetUserAsync(userName);
        user.Address = await AddressService.GetAddressAsync(userName);

        return user;
    }
}
```

Code Listing 115

Remember, you're writing reusable library code, so it could be called from many different technologies, such as WPF, Windows Store apps, Windows Phone, and more. What's common about these type of applications is that a user interacts with the UI controls and those UI controls fire events. This means an async **`void`** event handler awaits your **`GetUserInfoAsync`** method.

When the event handler code calls your code, it's running on the UI thread. Your code will continue running on the UI thread until some other code explicitly marshals the call to another thread and releases the UI thread.

Note: More accurately, the thread calling your code might not necessarily be the UI thread if there was another async method that called your code and already released the UI thread. However, defensive coding is a safe approach because

there exists the possibility that your code will be called on the UI thread by some developer in the future.

Notice the LINQ query in **GetUserInfoAsync** before reaching the first **await**. That is synchronous code that runs on the calling thread, which could also be the UI thread. The issue here is that the UI thread is tied up doing work in your async method, rather than returning to the UI. Imagine a UI with a progress indicator that locks up because your async method is holding onto the UI thread and doing a lot of processing before the first async call.

The code is still on the UI thread when it calls **UserService.GetUserAsync**. I added a comment to **GetUserAsync** to represent more long-running synchronous processing that is also running on the UI thread. Finally, awaiting **Task.FromResult** releases the UI thread and the rest of the code runs asynchronously. That's because **Task.FromResult** implements the async contract properly. Before showing you how to fix this problem, let's look at the rest of the code so you can understand how it runs.

When the code returns from **Task.FromResult**, the UI thread has been released and the code is running on the new async thread. When returning from **GetUserAsync** to its caller, **GetUserInfoAsync**, the call automatically marshals back to the calling thread, which could be the UI thread. Again, this program eats up CPU cycles on the UI thread, making the application less responsive. Fortunately, there's a way to fix this problem.

Fulfilling the Async Contract

The previous section explained how the code runs on the calling thread by default, which could be the UI thread. Whenever you call an async method in the FCL, that code will release the calling thread and continue on a new thread, which is proper behavior of the async contract that developers expect. You should do the same in your code.

To do this, use the **Task.ConfigureAwait** method, passing **false** as the parameter. The following is an example that fixes the problem in **GetUserInfoAsync**.

```
public async Task<UserInfo> GetUserInfoAsync(string term, List<string> names)
{
    var userName =
        (from name in names
         where name.StartsWith(term)
         select name)
        .FirstOrDefault();

    var user = new UserInfo();
    user.Info = await UserService.GetUserAsync(userName).ConfigureAwait(false);
    user.Address = await AddressService.GetAddressAsync(userName);

    return user;
}
```

Code Listing 116

The **GetUserInfoAsync** method appends **ConfigureAwait(false)** to the call to **GetUserAsync**. **GetUserAsync** returns a **Task<string>** and **ConfigureAwait(false)** operates on that return value, releasing the calling thread and running the rest of the method on a new async thread. That's it; that's all you have to do.

You still have the issue of synchronous processing before the first call to **ConfigureAwait**. Sometimes, you can't do anything about it because it's necessary to execute that code before the first **await**. However, if it's possible to rearrange the code to do any processing after the first **await**, you should do so.

A Few More Notes on Async

I made this point previously, but I feel it bears repeating. Especially for library code, you should prefer async methods that return **Task** or **Task<T>**. In the UI you don't have a choice if you're writing an event handler. If you're using a Model View ViewModel (MVVM) architecture, you'll also need **void Command** handlers. You shouldn't have these issues in reusable library code and in this scenario, async **void** methods are dangerous.

Async void methods that throw exceptions will crash your application.

Much of the discussion in this chapter is around how async improves the user experience by releasing the UI thread. In addition to that, async also improves application performance by not blocking threads. These scenarios usually involve some type of out-of-process operation such as network communication, file I/O, or REST service calls. These operations can use Windows operating system services such as I/O completion ports to free threads while the long-running out-of-process operation executes; they can then reallocate those threads when the operation completes and needs to return to your code. In addition to performance increases through efficient thread management, you can also improve the scalability of a server application by using async to avoid blocking threads more than you have to.

For all the seeming complexity that this chapter introduces, attempting to perform many of the operations associated with managing threads for application responsiveness, performance, and scalability is made much easier through the use of async.

Summary

Async is a useful capability that allows your application to be responsive and perform well. The user experience of async is a method with an **async** modifier and the ability to await an async method's **Task**. In addition to the user experience, there are additional considerations for writing async libraries. You should be aware of the threading behavior and how an **async** method runs on the caller's thread by default. Remember that you should minimize synchronous code before the first **await** and that you should call **ConfigureAwait(false)** at the earliest opportunity, releasing the UI thread and running the remaining algorithm on the new async thread.

Chapter 9 Moving Forward and More Things to Know

To keep subject matter succinct, I've passed up features that could evolve into deeper discussions. This chapter is about some of those features, if only to highlight that they are part of the C# language and that you are likely to encounter them regularly.

Decorating Code with Attributes

An attribute is a feature of C# that lets you decorate code with meta-information for various tools. I use the term "tool" loosely, but it could be the C# compiler, a testing framework, or a UI technology. Essentially, these tools read the attributes to make some decision on how to work with your code. I'll show you a few examples so you can be familiar with attribute syntax when you encounter it in code.

The **Obsolete** attribute lets you indicate that some code has been deprecated. It's a C# compiler attribute and the compiler will emit a message regarding its use. The following code shows an example.

```
using System;

public class ShoppingCart
{
    [Obsolete("Method planned for deprecation on date - use … instead.")]
    public void Add(string item) { }

    [Obsolete("Method is obsolete and can no longer be used", error: true)]
    public decimal CalculateTax(decimal[] prices) { return 0; }
}
```

Code Listing 117

When the C# compiler sees the **Obsolete** attribute decorating **Add**, it will show a warning with the message argument matching the parameter to the **Obsolete** attribute. In the second example, the compiler shows the message as an error and you won't be able to compile because the second parameter, **true**, indicates that the compiler should treat usage of that method as an error.

The next example uses attributes for a unit test with MSTest, Microsoft's unit testing software.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

```
namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Code Listing 118

As with most unit testing frameworks, MSTest has a test runner that loads the unit test code, looks for classes decorated with the **`TestClass`** attribute, and executes methods with the **`TestMethod`** attribute. It does this with another capability of C# called reflection.

Using Reflection

Reflection gives you the ability to examine compiled .NET code. With reflection, you can build useful tools like MSTest, dynamically instantiate types and execute their code, and more. The following code uses reflection to dynamically instantiate a type and execute one of its methods.

```
using System.Linq;
using System;
using System.Reflection;

public class FinancialCalculator
{
    public decimal Sum(decimal[] numbers)
    {
        return numbers.Sum();
    }
}

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        Type calcType = typeof(FinancialCalculator);
        MethodInfo sumMethod = calcType.GetMethod("Sum");
        FinancialCalculator calc =
            (FinancialCalculator)Activator.CreateInstance(calcType);
        decimal sum = (decimal)sumMethod.Invoke(calc, new object[] { prices });

        Console.WriteLine($"Sum: {sum}\nPress any key to continue.");
```

```
        Console.ReadKey();
    }
}
```

Code Listing 119

FinancialCalculator.Sum uses the LINQ **Sum** method, so add a **using** clause for **System.Linq**. Add a **using** declaration for **System.Reflection** to support reflection too.

With reflection, a **Type** instance gives you access to all of the information about a type. The **Main** method calls **GetMethod** to obtain a **MethodInfo** reference to the **Add** method, but there are many more methods that let you look at various parts of a type. As an example of a subset of capabilities available, you can call **GetMethods**, **GetProperties**, or **GetFields** to get an array of **MethodInfo**, **PropertyInfo**, or **FieldInfo** respectively. There are many more methods in the **Type** class you can use, and it's a fun exercise to write code to practice with this.

Activator.CreateInstance creates a new instance of the **Type** it's passed. Calling **Invoke** lets you run a method and get the results. Much of the previous reflection code is hard-coded for simplicity, but it's very useful for when you need to write code that examines the capabilities of another piece of code and optionally work with the member of a type.

Working with Code Dynamically

C# also has a type called dynamic. Its purpose is to allow you to interoperate with dynamic languages, like IronPython and IronRuby, and makes reflection easier. Microsoft also has a technology called Silverlight. Dynamic could make working with the HTML DOM easier, but Silverlight has been largely replaced by HTML 5 as a dynamic web application technology.

The **dynamic** type lets you assign any value to a **dynamic** variable and use any typed members on that variable. Rather than the C# compiler emitting errors, any errors are handled by the CLR at runtime. You might see where this has a lot of power through coding flexibility, yet a drawback of dynamic typing is that it offers no indication of type-related errors until runtime. Using the **FinancialCalculator** class from the previous example, the following is an example of some dynamic code.

```
using System;
using System.Reflection;

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        Type calcType = typeof(FinancialCalculator);
        MethodInfo sumMethod = calcType.GetMethod("Sum");
        dynamic calc = Activator.CreateInstance(calcType);
```

```
        dynamic sum = calc.Sum(prices);

        Console.WriteLine($"Sum: {sum}\nPress any key to continue.");
        Console.ReadKey();
    }
}
```

Code Listing 120

You might notice that this code is simpler than the full reflection implementation. You don't have an interface and have no guarantee that the **`calc`** instance has a member named **`Sum`**, but since the alternative is to use reflection, you're still in the same situation of runtime evaluation. Therefore, this might be a reasonable approach for this particular scenario.

Pulling together what you learned about generics, it might be useful to improve the algorithm even further, as shown in the following listing.

```
using System;

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        decimal sum = GetSum<FinancialCalculator, decimal>(prices);

        Console.WriteLine("Sum: {0}\nPress any key to continue.", sum);
        Console.ReadKey();
    }

    public static TValue GetSum<TCalc, TValue>(TValue[] prices)
        where TCalc : new()
    {
        dynamic calc = new TCalc();
        TValue sum = calc.Sum(prices);
        return sum;
    }
}
```

Code Listing 121

The previous example totally eliminates the need for reflection, reduces code, makes the algorithm strongly typed where it needed to be, and makes it dynamic where it helps. It would have been possible to use an interface constraint to make **`GetSum`** more strongly typed, but I used this as an exercise to help you think about where dynamic might be useful.

Summary

Attributes are C# features that tell a tool something about your code. Reflection helps you write meta-code that can evaluate and execute other code. There is a dynamic type that lets you make assumptions about the code you're writing, interfacing with dynamic languages, and making it easy to perform reflection.

This completes *C# Succinctly*. I hope it has been useful for you. I wish you the best in your further studies.

Sunday, March 06, 2016
12:09 AM

Artificial Neural Networks: A Tutorial

Anil K. Jain
Michigan State University

Jianchang Mao
K.M. Mohiuddin
IBM Almaden Research Center

These massively parallel systems with large numbers of interconnected simple processors may solve a variety of challenging computational problems. This tutorial provides the background and the basics.

Numerous advances have been made in developing intelligent systems, some inspired by biological neural networks. Researchers from many scientific disciplines are designing artificial neural networks (ANNs) to solve a variety of problems in pattern recognition, prediction, optimization, associative memory, and control (see the "Challenging problems" sidebar).

Conventional approaches have been proposed for solving these problems. Although successful applications can be found in certain well-constrained environments, none is flexible enough to perform well outside its domain. ANNs provide exciting alternatives, and many applications could benefit from using them.[1-3]

This article is for those readers with little or no knowledge of ANNs to help them understand the other articles in this issue of *Computer*. We discuss the motivations behind the development of ANNs, describe the basic biological neuron and the artificial computational model, outline network architectures and learning processes, and present some of the most commonly used ANN models. We conclude with character recognition, a successful ANN application.

WHY ARTIFICIAL NEURAL NETWORKS?

The long course of evolution has given the human brain many desirable characteristics not present in von Neumann or modern parallel computers. These include

- massive parallelism,
- distributed representation and computation,
- learning ability,
- generalization ability,
- adaptivity,
- inherent contextual information processing,
- fault tolerance, and
- low energy consumption.

It is hoped that devices based on biological neural networks will possess some of these desirable characteristics.

Modern digital computers outperform humans in the domain of numeric computation and related symbol manipulation. However, humans can effortlessly solve complex perceptual problems (like recognizing a man in a crowd from a mere glimpse of his face) at such a high speed and extent as to dwarf the world's fastest computer. Why is there such a remarkable difference in their performance? The biological neural system architecture is completely different from the von Neumann architecture (see Table 1). This difference significantly affects the type of functions each computational model can best perform.

Numerous efforts to develop "intelligent" programs based on von Neumann's centralized architecture have not resulted in general-purpose intelligent programs. Inspired by biological neural networks, ANNs are massively parallel computing systems consisting of an exremely large number of simple processors with many interconnections. ANN models attempt to use some "organizational" principles believed to be used in the human

0018-9162/96/$5.00 © 1996 IEEE

Challenging problems

Let us consider the following problems of interest to computer scientists and engineers.

Pattern classification

The task of pattern classification is to assign an input pattern (like a speech waveform or handwritten symbol) represented by a feature vector to one of many prespecified classes (see Figure A1). Well-known applications include character recognition, speech recognition, EEG waveform classification, blood cell classification, and printed circuit board inspection.

Clustering/categorization

In clustering, also known as unsupervised pattern classification, there are no training data with known class labels. A clustering algorithm explores the similarity between the patterns and places similar patterns in a cluster (see Figure A2). Well-known clustering applications include data mining, data compression, and exploratory data analysis.

Function approximation

Suppose a set of n labeled training patterns (input-output pairs), $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, have been generated from an unknown function $\mu(\mathbf{x})$ (subject to noise). The task of function approximation is to find an estimate, say $\hat{\mu}$, of the unknown function μ (Figure A3). Various engineering and scientific modeling problems require function approximation.

Prediction/forecasting

Given a set of n samples $\{y(t_1), y(t_2), \ldots, y(t_n)\}$ in a time sequence, $t_1, t_2, \ldots, t_n$, the task is to predict the sample $y(t_{n+1})$ at some future time t_{n+1}. Prediction/forecasting has a significant impact on decision-making in business, science, and engineering. Stock market prediction and weather forecasting are typical applications of prediction/forecasting techniques (see Figure A4).

Optimization

A wide variety of problems in mathematics, statistics, engineering, science, medicine, and economics can be posed as optimization problems. The goal of an optimization algorithm is to find a solution satisfying a set of constraints such that an objective function is maximized or minimized. The Traveling Salesman Problem (TSP), an *NP-complete* problem, is a classic example (see Figure A5).

Content-addressable memory

In the von Neumann model of computation, an entry in memory is accessed only through its address, which is independent of the content in the memory. Moreover, if a small error is made in calculating the address, a completely different item can be retrieved. Associative memory or content-addressable memory, as the name implies, can be accessed by their content. The content in the memory can be recalled even by a partial input or distorted content (see Figure A6). Associative memory is extremely desirable in building multimedia information databases.

Control

Consider a dynamic system defined by a tuple $\{u(t), y(t)\}$, where $u(t)$ is the control input and $y(t)$ is the resulting output of the system at time t. In model-reference adaptive control, the goal is to generate a control input $u(t)$ such that the system follows a desired trajectory determined by the reference model. An example is engine idle-speed control (Figure A7).

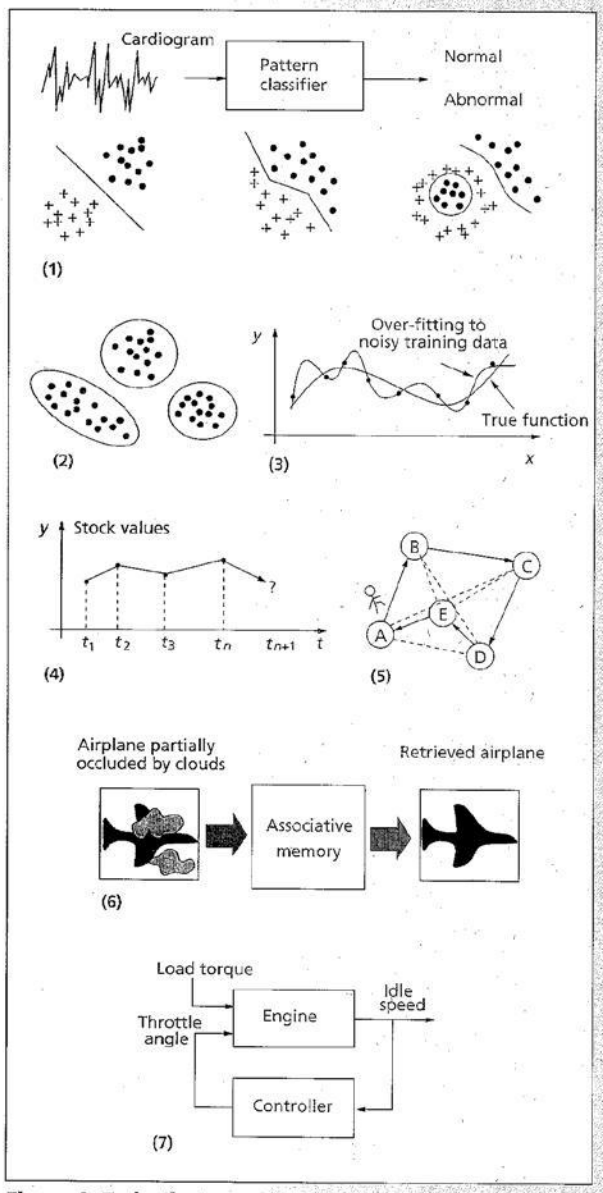


Figure A. Tasks that neural networks can perform: (1) pattern classification; (2) clustering/categorization; (3) function approximation; (4) prediction/forecasting; (5) optimization (a TSP problem example); (6) retrieval by content; and (7) control (engine idle speed). (Adapted from *DARPA Neural Network Study*[1])

brain. Modeling a biological nervous system using ANNs can also increase our understanding of biological functions. State-of-the-art computer hardware technology (such as *VLSI* and optical) has made this modeling feasible.

A thorough study of ANNs requires knowledge of neurophysiology, cognitive science/psychology, physics (statistical mechanics), control theory, computer science, artificial intelligence, statistics/mathematics, pattern recognition, computer vision, parallel processing, and hardware (digital/analog/VLSI/optical). New developments in these disciplines continuously nourish the field. On the other hand, ANNs also provide an impetus to these disciplines in the form of new tools and representations. This symbiosis is necessary for the vitality of neural network research. Communications among these disciplines ought to be encouraged.

Brief historical review

ANN research has experienced three periods of extensive activity. The first peak in the 1940s was due to McCulloch and Pitts' pioneering work.[4] The second occurred in the 1960s with Rosenblatt's perceptron convergence theorem[5] and Minsky and Papert's work showing the limitations of a simple perceptron.[6] Minsky and Papert's results dampened the enthusiasm of most researchers, especially those in the computer science community. The resulting lull in neural network research lasted almost 20 years. Since the early 1980s, ANNs have received considerable renewed interest. The major developments behind this resurgence include Hopfield's energy approach[7] in 1982 and the back-propagation learning algorithm for multilayer perceptrons (multilayer feedforward networks) first proposed by Werbos,[8] reinvented several times, and then popularized by Rumelhart et al.[9] in 1986. Anderson and Rosenfeld[10] provide a detailed historical account of ANN developments.

Biological neural networks

A *neuron* (or nerve cell) is a special biological cell that processes information (see Figure 1). It is composed of a cell body, or *soma*, and two types of out-reaching tree-like branches: the *axon* and the *dendrites*. The cell body has a nucleus that contains information about hereditary traits and a plasma that holds the molecular equipment for producing material needed by the neuron. A neuron receives signals (impulses) from other neurons through its dendrites (receivers) and transmits signals generated by its cell body along the axon (transmitter), which eventually branches into strands and substrands. At the terminals of these strands are the *synapses*. A synapse is an elementary structure and functional unit between two neurons (an axon strand of one neuron and a dendrite of another). When the impulse reaches the synapse's terminal, certain chemicals called neurotransmitters are released. The neurotransmitters diffuse across the synaptic gap, to enhance or inhibit, depending on the type of the synapse, the receptor neuron's own tendency to emit electrical impulses. The synapse's effectiveness can be adjusted by the signals passing through it so that the synapses can *learn* from the activities in which they participate. This dependence on history acts as a memory, which is possibly responsible for human memory.

The cerebral cortex in humans is a large flat sheet of neurons about 2 to 3 millimeters thick with a surface area of about 2,200 cm^2, about twice the area of a standard computer keyboard. The cerebral cortex contains about 10^{11} neurons, which is approximately the number of stars in the Milky Way.[11] Neurons are massively connected, much more complex and dense than telephone networks. Each neuron is connected to 10^3 to 10^4 other neurons. In total, the human brain contains approximately 10^{14} to 10^{15} interconnections.

Neurons communicate through a very short train of pulses, typically milliseconds in duration. The *message* is modulated on the pulse-transmission frequency. This frequency can vary from a few to several hundred hertz, which is a million times slower than the fastest switching speed in electronic circuits. However, complex perceptual decisions such as face recognition are typically made by humans within a few hundred milliseconds. These decisions are made by a network of neurons whose operational speed is only a few milliseconds. This implies that the computations cannot take more than about 100 serial stages. In other

Table 1. Von Neumann computer versus biological neural system.

	Von Neumann computer	Biological neural system
Processor	Complex High speed One or a few	Simple Low speed A large number
Memory	Separate from a processor Localized Noncontent addressable	Integrated into processor Distributed Content addressable
Computing	Centralized Sequential Stored programs	Distributed Parallel Self-learning
Reliability	Very vulnerable	Robust
Expertise	Numerical and symbolic manipulations	Perceptual problems
Operating environment	Well-defined, well-constrained	Poorly defined, unconstrained

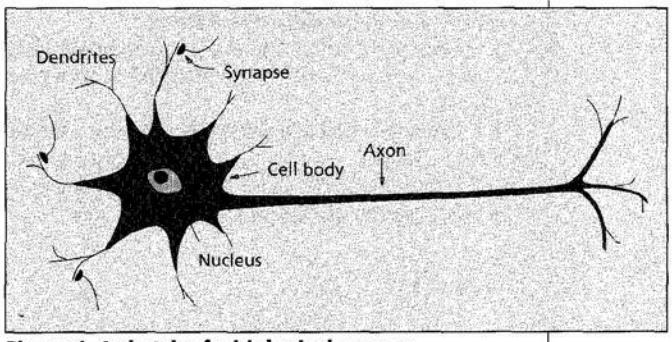


Figure 1. A sketch of a biological neuron.

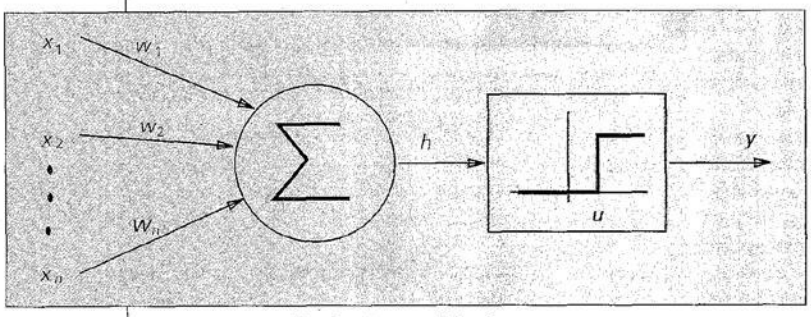


Figure 2. McCulloch-Pitts model of a neuron.

words, the brain runs parallel programs that are about 100 steps long for such perceptual tasks. This is known as the *hundred step rule.*[12] The same timing considerations show that the amount of information sent from one neuron to another must be very small (a few bits). This implies that critical information is not transmitted directly, but captured and distributed in the interconnections—hence the name, *connectionist* model, used to describe ANNs.

Interested readers can find more introductory and easily comprehensible material on biological neurons and neural networks in Brunak and Lautrup.[11]

ANN OVERVIEW

Computational models of neurons

McCulloch and Pitts[4] proposed a binary threshold unit as a computational model for an artificial neuron (see Figure 2).

This mathematical neuron computes a weighted sum of its n input signals, $x_j, j = 1, 2, \ldots, n$, and generates an output of 1 if this sum is above a certain threshold u. Otherwise, an output of 0 results. Mathematically,

$$y = \theta\left(\sum_{j=1}^{n} w_j x_j - u\right),$$

where $\theta(\cdot)$ is a unit step function at 0, and w_j is the synapse weight associated with the jth input. For simplicity of notation, we often consider the threshold u as another weight $w_0 = -u$ attached to the neuron with a constant input $x_0 = 1$. Positive weights correspond to *excitatory* synapses, while negative weights model *inhibitory* ones. McCulloch and Pitts proved that, in principle, suitably chosen weights let a synchronous arrangement of such neurons perform universal computations. There is a crude analogy here to a biological neuron: wires and interconnections model axons and dendrites, connection weights represent synapses, and the threshold function approximates the activity in a soma. The McCulloch and Pitts model, however, contains a number of simplifying assumptions that do not reflect the true behavior of biological neurons.

The McCulloch-Pitts neuron has been generalized in many ways. An obvious one is to use activation functions other than the threshold function, such as piecewise linear, sigmoid, or Gaussian, as shown in Figure 3. The sigmoid function is by far the most frequently used in ANNs. It is a strictly increasing function that exhibits smoothness and has the desired asymptotic properties. The standard sigmoid function is the *logistic* function, defined by

$$g(x) = 1/(1 + \exp\{-\beta x\}),$$

where β is the slope parameter.

Network architectures

ANNs can be viewed as weighted directed graphs in which artificial neurons are nodes and directed edges (with weights) are connections between neuron outputs and neuron inputs.

Based on the connection pattern (architecture), ANNs can be grouped into two categories (see Figure 4):

- *feed-forward* networks, in which graphs have no loops, and
- *recurrent* (or *feedback*) networks, in which loops occur because of feedback connections.

In the most common family of feed-forward networks, called multilayer perceptron, neurons are organized into layers that have unidirectional connections between them. Figure 4 also shows typical networks for each category.

Different connectivities yield different network behaviors. Generally speaking, feed-forward networks are *static*, that is, they produce only one set of output values rather than a sequence of values from a given input. Feed-forward networks are memory-less in the sense that their response to an input is independent of the previous network state. Recurrent, or feedback, networks, on the other hand, are dynamic systems. When a new input pattern is presented, the neuron outputs are computed. Because of the feedback paths, the inputs to each neuron are then modified, which leads the network to enter a new state.

Different network architectures require appropriate learning algorithms. The next section provides an overview of learning processes.

Learning

The ability to learn is a fundamental trait of intelligence. Although a precise definition of learning is difficult to formulate, a learning process in the ANN context can be viewed as the problem of updating network architecture and connection weights so that a network can efficiently perform a specific task. The network usually must learn the connection weights from available training patterns. Performance is improved over time by iteratively updating the weights in the network. ANNs' ability to automatically *learn from examples* makes them attractive and exciting. Instead of following a set of *rules* specified by human experts, ANNs appear to learn underlying rules (like input-output relationships) from the given collection of representative examples. This is one of the major advantages of neural networks over traditional expert systems.

To understand or design a learning process, you must first have a model of the environment in which a neural network operates, that is, you must know what information is available to the network. We refer to this model as

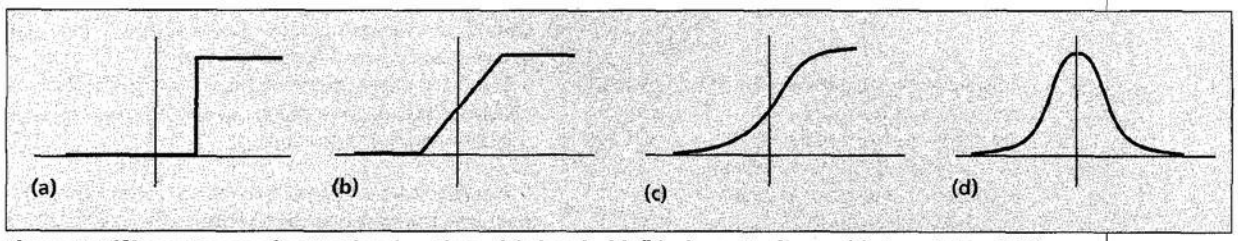


Figure 3. Different types of activation functions: (a) threshold, (b) piecewise linear, (c) sigmoid, and (d) Gaussian.

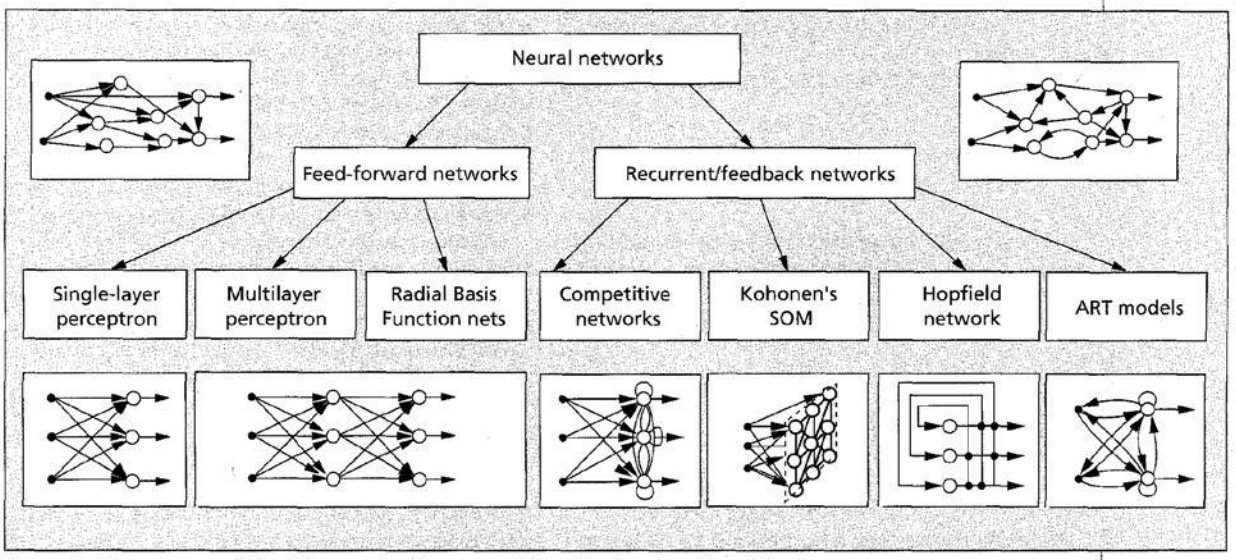


Figure 4. A taxonomy of feed-forward and recurrent/feedback network architectures.

a learning paradigm.[3] Second, you must understand how network weights are updated, that is, which *learning rules* govern the updating process. A *learning algorithm* refers to a procedure in which learning rules are used for adjusting the weights.

There are three main learning paradigms: supervised, unsupervised, and hybrid. In supervised learning, or learning with a "teacher," the network is provided with a correct answer (output) for every input pattern. Weights are determined to allow the network to produce answers as close as possible to the known correct answers. Reinforcement learning is a variant of supervised learning in which the network is provided with only a critique on the correctness of network outputs, not the correct answers themselves. In contrast, unsupervised learning, or learning without a teacher, does not require a correct answer associated with each input pattern in the training data set. It explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories from these correlations. Hybrid learning combines supervised and unsupervised learning. Part of the weights are usually determined through supervised learning, while the others are obtained through unsupervised learning.

Learning theory must address three fundamental and practical issues associated with learning from samples: capacity, sample complexity, and computational complexity. Capacity concerns how many patterns can be stored, and what functions and decision boundaries a network can form.

Sample complexity determines the number of training patterns needed to train the network to guarantee a valid generalization. Too few patterns may cause "over-fitting" (wherein the network performs well on the training data set, but poorly on independent test patterns drawn from the same distribution as the training patterns, as in Figure A3).

Computational complexity refers to the time required for a learning algorithm to estimate a solution from training patterns. Many existing learning algorithms have high computational complexity. Designing efficient algorithms for neural network learning is a very active research topic.

There are four basic types of learning rules: error-correction, Boltzmann, Hebbian, and competitive learning.

ERROR-CORRECTION RULES. In the supervised learning paradigm, the network is given a desired output for each input pattern. During the learning process, the actual output y generated by the network may not equal the desired output d. The basic principle of error-correction learning rules is to use the error signal $(d - y)$ to modify the connection weights to gradually reduce this error.

The perceptron learning rule is based on this error-correction principle. A perceptron consists of a single neuron with adjustable weights, $w_j, j = 1, 2, \ldots, n$, and threshold u, as shown in Figure 2. Given an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^t$, the net input to the neuron is

Perceptron learning algorithm

1. Initialize the weights and threshold to small random numbers.
2. Present a pattern vector $(x_1, x_2, \ldots, x_n)^t$ and evaluate the output of the neuron.
3. Update the weights according to

   $$w_j(t+1) = w_j(t) + \eta\,(d - y)\,x_j,$$

   where d is the desired output, t is the iteration number, and η $(0.0 < \eta < 1.0)$ is the gain (step size).

$$v = \sum_{j=1}^{n} w_j x_j - u$$

The output y of the perceptron is $+1$ if $v > 0$, and 0 otherwise. In a two-class classification problem, the perceptron assigns an input pattern to one class if $y = 1$, and to the other class if $y = 0$. The linear equation

$$\sum_{j=1}^{n} w_j x_j - u = 0$$

defines the decision boundary (a hyperplane in the n-dimensional input space) that halves the space.

Rosenblatt[5] developed a learning procedure to determine the weights and threshold in a perceptron, given a set of training patterns (see the "Perceptron learning algorithm" sidebar).

Note that learning occurs only when the perceptron makes an error. Rosenblatt proved that when training patterns are drawn from two linearly separable classes, the perceptron learning procedure converges after a finite number of iterations. This is the *perceptron convergence theorem*. In practice, you do not know whether the patterns are linearly separable. Many variations of this learning algorithm have been proposed in the literature.[2] Other activation functions that lead to different learning characteristics can also be used. However, *a single-layer perceptron can only separate linearly separable patterns as long as a monotonic activation function is used.*

The back-propagation learning algorithm (see the "Back-propagation algorithm sidebar") is also based on the error-correction principle.

BOLTZMANN LEARNING. Boltzmann machines are symmetric recurrent networks consisting of binary units (+1 for "on" and –1 for "off"). By symmetric, we mean that the weight on the connection from unit i to unit j is equal to the weight on the connection from unit j to unit i ($w_{ij} = w_{ji}$). A subset of the neurons, called *visible*, interact with the environment; the rest, called *hidden*, do not. Each neuron is a stochastic unit that generates an output (or state) according to the Boltzmann distribution of statistical mechanics. Boltzmann machines operate in two modes: *clamped*, in which visible neurons are clamped onto specific states determined by the environment; and *free-running*, in which both visible and hidden neurons are allowed to operate freely.

Boltzmann learning is a stochastic learning rule derived from information-theoretic and thermodynamic principles.[10] The objective of Boltzmann learning is to adjust the connection weights so that the states of visible units satisfy a particular desired probability distribution. According to the Boltzmann learning rule, the change in the connection weight w_{ij} is given by

$$\Delta w_{ij} = \eta(\overline{\rho}_{ij} - \rho_{ij}),$$

where η is the learning rate, and $\overline{\rho}_{ij}$ and ρ_{ij} are the correlations between the states of units i and j when the network operates in the clamped mode and free-running mode, respectively. The values of $\overline{\rho}_{ij}$ and ρ_{ij} are usually estimated from Monte Carlo experiments, which are extremely slow.

Boltzmann learning can be viewed as a special case of error-correction learning in which error is measured not as the direct difference between desired and actual outputs, but as the difference between the correlations among the outputs of two neurons under clamped and free-running operating conditions.

HEBBIAN RULE. The oldest learning rule is *Hebb's postulate of learning*.[13] Hebb based it on the following observation from neurobiological experiments: If neurons on both sides of a synapse are activated synchronously and repeatedly, the synapse's strength is selectively increased.

Mathematically, the Hebbian rule can be described as

$$w_{ij}(t+1) = w_{ij}(t) + \eta\, y_j(t)\, x_i(t),$$

where x_i and y_j are the output values of neurons i and j, respectively, which are connected by the synapse w_{ij}, and η is the learning rate. Note that x_i is the input to the synapse.

An important property of this rule is that learning is done locally, that is, the change in synapse weight depends only on the activities of the two neurons connected by it. This significantly simplifies the complexity of the learning circuit in a VLSI implementation.

A single neuron trained using the Hebbian rule exhibits an orientation selectivity. Figure 5 demonstrates this property. The points depicted are drawn from a two-dimen-

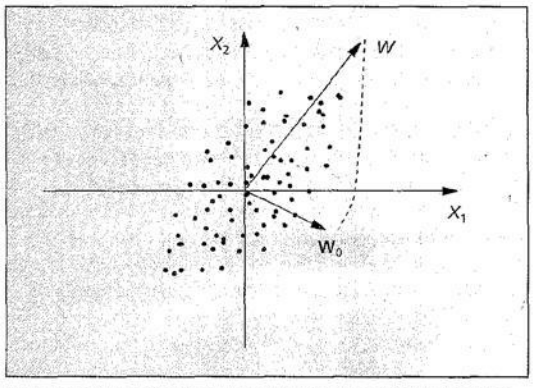


Figure 5. Orientation selectivity of a single neuron trained using the Hebbian rule.

sional Gaussian distribution and used for training a neuron. The weight vector of the neuron is initialized to $\mathbf{w}_0$ as shown in the figure. As the learning proceeds, the weight vector moves progressively closer to the direction $\mathbf{w}$ of maximal variance in the data. In fact, $\mathbf{w}$ is the eigenvector of the covariance matrix of the data corresponding to the largest eigenvalue.

COMPETITIVE LEARNING RULES. Unlike Hebbian learning (in which multiple output units can be fired simultaneously), competitive-learning output units compete among themselves for activation. As a result, only one output unit is active at any given time. This phenomenon is known as *winner-take-all*. Competitive learning has been found to exist in biological neural networks.[3]

Competitive learning often clusters or categorizes the input data. Similar patterns are grouped by the network and represented by a single unit. This grouping is done automatically based on data correlations.

The simplest competitive learning network consists of a single layer of output units as shown in Figure 4. Each output unit i in the network connects to all the input units (x_j's) via weights, $w_{ij}, j=1, 2, \ldots, n$. Each output unit also connects to all other output units via inhibitory weights but has a self-feedback with an excitatory weight. As a result of competition, only the unit i^* with the largest (or the smallest) net input becomes the winner, that is, $\mathbf{w}_{i^*} \cdot \mathbf{x} \geq \mathbf{w}_i \cdot \mathbf{x}$, $\forall i$, or $\| \mathbf{w}_{i^*} - \mathbf{x} \| \leq \| \mathbf{w}_i - \mathbf{x} \|$, $\forall i$. When all the weight vectors are normalized, these two inequalities are equivalent.

A simple competitive learning rule can be stated as

$$\Delta w_{ij} = \begin{cases} \eta(x_j^u - w_{i^*j}), & i = i^*, \\ 0, & i \neq i^*. \end{cases} \quad (1)$$

Note that only the weights of the winner unit get updated. The effect of this learning rule is to move the stored pattern in the winner unit (weights) a little bit closer to the input pattern. Figure 6 demonstrates a geometric interpretation of competitive learning. In this example, we assume that all input vectors have been normalized to have unit length. They are depicted as black dots in Figure 6. The weight vectors of the three units are randomly initialized. Their initial and final positions on the sphere after competitive learning are marked as Xs in Figures 6a and 6b, respectively. In Figure 6, each of the three natural groups (clusters) of patterns has been discovered by an output unit whose weight vector points to the center of gravity of the discovered group.

You can see from the competitive learning rule that the network will not stop learning (updating weights) unless the learning rate η is 0. A particular input pattern can fire different output units at different iterations during learning. This brings up the stability issue of a learning system. The system is said to be *stable* if no pattern in the training data changes its category after a finite number of learning iterations. One way to achieve stability is to force the learning rate to decrease gradually as the learning process proceeds towards 0. However, this artificial freezing of learning causes another problem termed *plasticity*, which is the ability to adapt to new data. This is known as Grossberg's *stability-plasticity* dilemma in competitive learning.

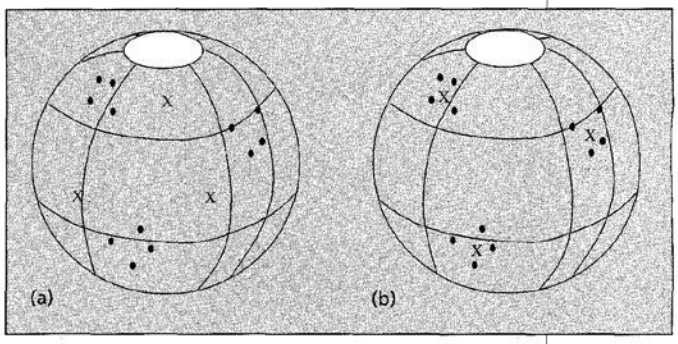


Figure 6. An example of competitive learning: (a) before learning; (b) after learning.

The most well-known example of competitive learning is *vector quantization* for data compression. It has been widely used in speech and image processing for efficient storage, transmission, and modeling. Its goal is to represent a set or distribution of input vectors with a relatively small number of prototype vectors (weight vectors), or a codebook. Once a codebook has been constructed and agreed upon by both the transmitter and the receiver, you need only transmit or store the index of the corresponding prototype to the input vector. Given an input vector, its corresponding prototype can be found by searching for the nearest prototype in the codebook.

SUMMARY. Table 2 summaries various learning algorithms and their associated network architectures (this is not an exhaustive list). Both supervised and unsupervised learning paradigms employ learning rules based

Back-propagation algorithm

1. Initialize the weights to small random values.
2. Randomly choose an input pattern $\mathbf{x}^{(u)}$.
3. Propagate the signal forward through the network.
4. Compute δ_i^L in the output layer $(o_i = y_i^L)$

   $$\delta_i^L = g'(h_i^L)\left[d_i^u - y_i^L\right],$$

   where h_i^l represents the net input to the ith unit in the lth layer, and g' is the derivative of the activation function g.
5. Compute the deltas for the preceding layers by propagating the errors backwards;

   $$\delta_i^l = g'(h_i^l)\sum_j w_{ij}^{l+1}\delta_j^{l+1},$$

   for $l = (L-1), \ldots, 1$.
6. Update weights using

   $$\Delta w_{ji}^l = \eta \delta_i^l y_j^{l-1}$$

7. Go to step 2 and repeat for the next pattern until the error in the output layer is below a prespecified threshold or a maximum number of iterations is reached.

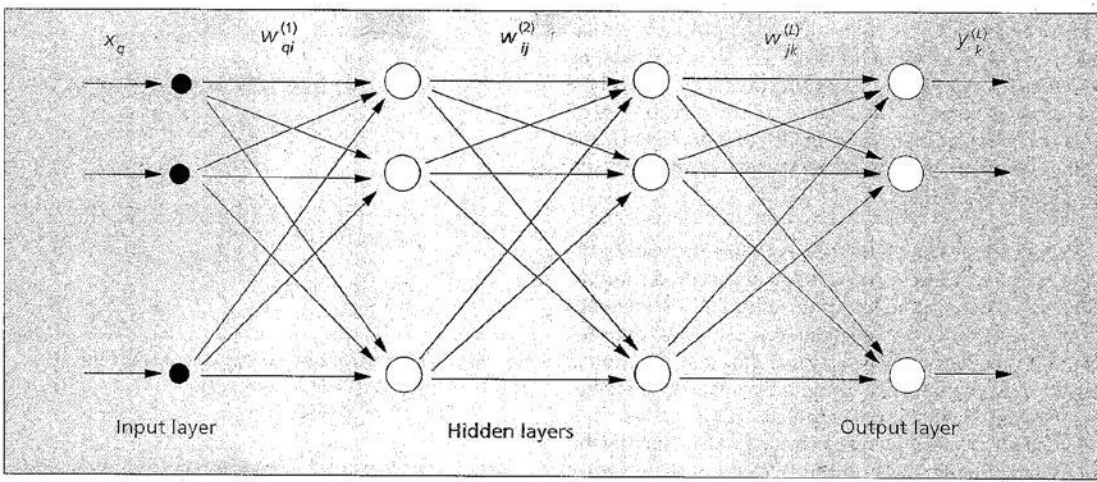


Figure 7. A typical three-layer feed-forward network architecture.

on error-correction, Hebbian, and competitive learning. Learning rules based on error-correction can be used for training feed-forward networks, while Hebbian learning rules have been used for all types of network architectures. However, each learning algorithm is designed for training a specific architecture. Therefore, when we discuss a learning algorithm, a particular network architecture association is implied. Each algorithm can

Table 2. Well-known learning algorithms.

Paradigm	Learning rule	Architecture	Learning algorithm	Task
Supervised	Error-correction	Single- or multilayer perceptron	Perceptron learning algorithms Back-propagation Adaline and Madaline	Pattern classification Function approximation Prediction, control
	Boltzmann	Recurrent	Boltzmann learning algorithm	Pattern classification
	Hebbian	Multilayer feed-forward	Linear discriminant analysis	Data analysis Pattern classification
	Competitive	Competitive	Learning vector quantization	Within-class categorization Data compression
		ART network	ARTMap	Pattern classification Within-class categorization
Unsupervised	Error-correction	Multilayer feed-forward	Sammon's projection	Data analysis
	Hebbian	Feed-forward or competitive	Principal component analysis	Data analysis Data compression
		Hopfield Network	Associative memory learning	Associative memory
	Competitive	Competitive	Vector quantization	Categorization Data compression
		Kohonen's SOM	Kohonen's SOM	Categorization Data analysis
		ART networks	ART1, ART2	Categorization
Hybrid	Error-correction and competitive	RBF network	RBF learning algorithm	Pattern classification Function approximation Prediction, control

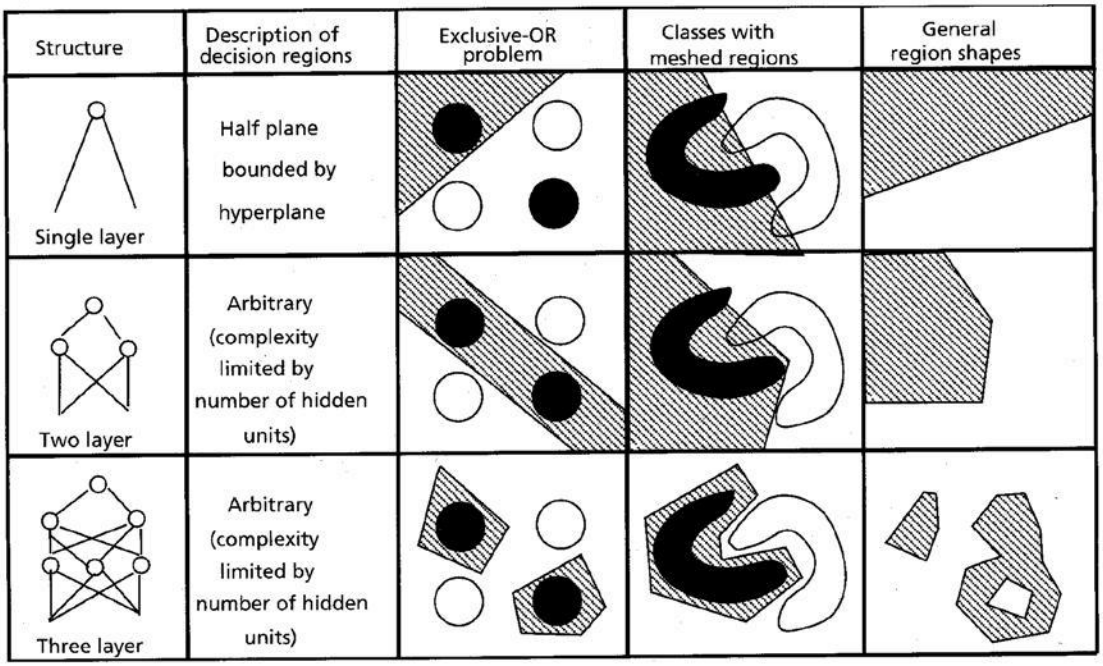


Figure 8. A geometric interpretation of the role of hidden unit in a two-dimensional input space.

perform only a few tasks well. The last column of Table 2 lists the tasks that each algorithm can perform. Due to space limitations, we do not discuss some other algorithms, including Adaline, Madaline,[14] linear discriminant analysis,[15] Sammon's projection,[15] and principal component analysis.[2] Interested readers can consult the corresponding references (this article does not always cite the first paper proposing the particular algorithms).

MULTILAYER FEED-FORWARD NETWORKS

Figure 7 shows a typical three-layer perceptron. In general, a standard L-layer feed-forward network (we adopt the convention that the input nodes are not counted as a layer) consists of an input stage, $(L-1)$ hidden layers, and an output layer of units successively connected (fully or locally) in a feed-forward fashion with no connections between units in the same layer and no feedback connections between layers.

Multilayer perceptron

The most popular class of multilayer feed-forward networks is *multilayer perceptrons* in which each computational unit employs either the thresholding function or the sigmoid function. Multilayer perceptrons can form arbitrarily complex decision boundaries and represent any Boolean function.[6] The development of the *back-propagation* learning algorithm for determining weights in a multilayer perceptron has made these networks the most popular among researchers and users of neural networks.

We denote $w_{ij}^{(l)}$ as the weight on the connection between the ith unit in layer $(l-1)$ to jth unit in layer l.

Let $\{(\mathbf{x}^{(1)}, \mathbf{d}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{d}^{(2)}), \ldots, (\mathbf{x}^{(p)}, \mathbf{d}^{(p)})\}$ be a set of p training patterns (input-output pairs), where $\mathbf{x}^{(i)} \in R^n$ is the input vector in the n-dimensional pattern space, and $\mathbf{d}^{(i)} \in [0, 1]^m$, an m-dimensional hypercube. For classification purposes, m is the number of classes. The squared-error cost function most frequently used in the ANN literature is defined as

$$E = \frac{1}{2}\sum_{i=1}^{p} \left\| \mathbf{y}^{(i)} - \mathbf{d}^{(i)} \right\|^2 \qquad (2)$$

The back-propagation algorithm[9] is a gradient-descent method to minimize the squared-error cost function in Equation 2 (see "Back-propagation algorithm" sidebar).

A geometric interpretation (adopted and modified from Lippmann[16]) shown in Figure 8 can help explicate the role of hidden units (with the threshold activation function).

Each unit in the first hidden layer forms a hyperplane in the pattern space; boundaries between pattern classes can be approximated by hyperplanes. A unit in the second hidden layer forms a hyperregion from the outputs of the first-layer units; a decision region is obtained by performing an AND operation on the hyperplanes. The output-layer units combine the decision regions made by the units in the second hidden layer by performing logical OR operations. Remember that this scenario is depicted only to explain the role of hidden units. Their actual behavior, after the network is trained, could differ.

A two-layer network can form more complex decision boundaries than those shown in Figure 8. Moreover, multilayer perceptrons with sigmoid activation functions can form smooth decision boundaries rather than piecewise linear boundaries.

Radial Basis Function network

The Radial Basis Function (RBF) network,[3] which has two layers, is a special class of multilayer feed-forward net-

works. Each unit in the hidden layer employs a radial basis function, such as a Gaussian kernel, as the activation function. The radial basis function (or kernel function) is centered at the point specified by the weight vector associated with the unit. Both the positions and the widths of these kernels must be learned from training patterns. There are usually many fewer kernels in the RBF network than there are training patterns. Each output unit implements a linear combination of these radial basis functions. From the point of view of function approximation, the hidden units provide a set of functions that constitute a basis set for representing input patterns in the space spanned by the hidden units.

There are a variety of learning algorithms for the RBF network.[3] The basic one employs a two-step learning strategy, or hybrid learning. It estimates kernel positions and kernel widths using an unsupervised clustering algorithm, followed by a supervised least mean square (LMS) algorithm to determine the connection weights between the hidden layer and the output layer. Because the output units are linear, a noniterative algorithm can be used. After this initial solution is obtained, a supervised gradient-based algorithm can be used to refine the network parameters.

This hybrid learning algorithm for training the RBF network converges much faster than the back-propagation algorithm for training multilayer perceptrons. However, for many problems, the RBF network often involves a larger number of hidden units. This implies that the runtime (after training) speed of the RBF network is often slower than the runtime speed of a multilayer perceptron. The efficiencies (error versus network size) of the RBF network and the multilayer perceptron are, however, problem-dependent. It has been shown that the RBF network has the same asymptotic approximation power as a multilayer perceptron.

SOM learning algorithm

1. Initialize weights to small random numbers; set initial learning rate and neighborhood.
2. Present a pattern $\mathbf{x}$, and evaluate the network outputs.
3. Select the unit (c_i, c_j) with the minimum output:

$$\|\mathbf{x} - \mathbf{w}_{c_i c_j}\| = \min_{ij} \|\mathbf{x} - \mathbf{w}_{ij}\|$$

4. Update all weights according to the following learning rule:

$$\mathbf{w}_{ij}(t+1) = \begin{cases} \mathbf{w}_{ij}(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{w}_{ij}(t)], & \text{if } (i,j) \in N_{c_i c_j}(t), \\ \mathbf{w}_{ij}(t), & \text{otherwise,} \end{cases}$$

where $N_{c_i c_j}(t)$ is the neighborhood of the unit (c_i, c_j) at time t, and $\alpha(t)$ is the learning rate.

5. Decrease the value of $\alpha(t)$ and shrink the neighborhood $N_{c_i c_j}(t)$.
6. Repeat steps 2 through 5 until the change in weight values is less than a prespecified threshold or a maximum number of iterations is reached.

Issues

There are many issues in designing feed-forward networks, including

- how many layers are needed for a given task,
- how many units are needed per layer,
- how will the network perform on data not included in the training set (generalization ability), and
- how large the training set should be for "good" generalization.

Although multilayer feed-forward networks using back propagation have been widely employed for classification and function approximation,[2] many design parameters still must be determined by trial and error. Existing theoretical results provide only very loose guidelines for selecting these parameters in practice.

KOHONEN'S SELF-ORGANIZING MAPS

The self-organizing map (SOM)[16] has the desirable property of topology preservation, which captures an important aspect of the feature maps in the cortex of highly developed animal brains. In a topology-preserving mapping, nearby input patterns should activate nearby output units on the map. Figure 4 shows the basic network architecture of Kohonen's SOM. It basically consists of a two-dimensional array of units, each connected to all n input nodes. Let $\mathbf{w}_{ij}$ denote the n-dimensional vector associated with the unit at location (i, j) of the 2D array. Each neuron computes the Euclidean distance between the input vector $\mathbf{x}$ and the stored weight vector $\mathbf{w}_{ij}$.

This SOM is a special type of competitive learning network that defines a spatial neighborhood for each output unit. The shape of the local neighborhood can be square, rectangular, or circular. Initial neighborhood size is often set to one half to two thirds of the network size and shrinks over time according to a schedule (for example, an exponentially decreasing function). During competitive learning, all the weight vectors associated with the winner and its neighboring units are updated (see the "SOM learning algorithm" sidebar).

Kohonen's SOM can be used for projection of multivariate data, density approximation, and clustering. It has been successfully applied in the areas of speech recognition, image processing, robotics, and process control.[2] The design parameters include the dimensionality of the neuron array, the number of neurons in each dimension, the shape of the neighborhood, the shrinking schedule of the neighborhood, and the learning rate.

ADAPTIVE RESONANCE THEORY MODELS

Recall that the *stability-plasticity* dilemma is an important issue in competitive learning. How do we learn new things (plasticity) and yet retain the stability to ensure that existing knowledge is not erased or corrupted? Carpenter and Grossberg's Adaptive Resonance Theory models (ART1, ART2, and ARTMap) were developed in an attempt to overcome this dilemma.[17] The network has a sufficient supply of output units, but they are not used until deemed necessary. A unit is said to be *committed* (*uncommitted*) if it is (is not) being used. The learning algorithm updates

the stored prototypes of a category only if the input vector is sufficiently similar to them. An input vector and a stored prototype are said to resonate when they are sufficiently similar. The extent of similarity is controlled by a *vigilance parameter*, ρ, with $0 < \rho < 1$, which also determines the number of categories. When the input vector is not sufficiently similar to any existing prototype in the network, a new category is created, and an uncommitted unit is assigned to it with the input vector as the initial prototype. If no such uncommitted unit exists, a novel input generates no response.

We present only ART1, which takes binary (0/1) input to illustrate the model. Figure 9 shows a simplified diagram of the ART1 architecture.[2] It consists of two layers of fully connected units. A top-down weight vector $\mathbf{w}_j$ is associated with unit j in the input layer, and a bottom-up weight vector $\overline{\mathbf{w}}_i$ is associated with output unit i; $\overline{\mathbf{w}}_i$ is the normalized version of $\mathbf{w}_i$.

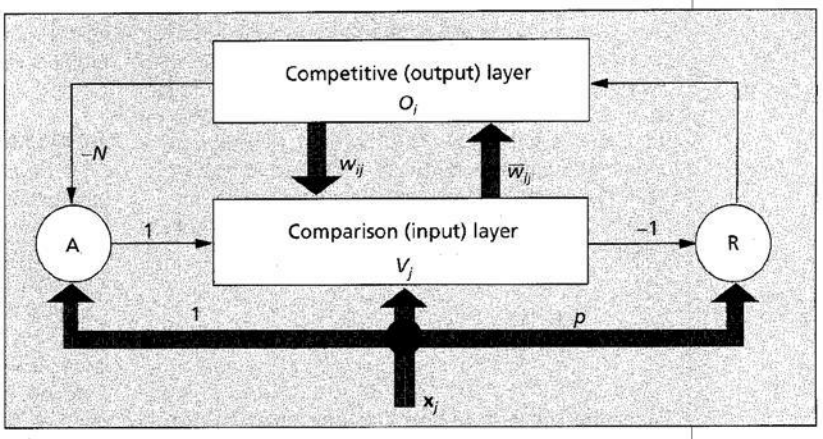


Figure 9. ART1 network.

$$\overline{\mathbf{w}}_i = \frac{\mathbf{w}_i}{\varepsilon + \sum_j w_{ji}}, \tag{3}$$

where ε is a small number used to break the ties in selecting the winner. The top-down weight vectors $\mathbf{w}_j$'s store cluster prototypes. The role of normalization is to prevent prototypes with a long vector length from dominating prototypes with a short one. Given an n-bit input vector $\mathbf{x}$, the output of the auxiliary unit A is given by

$$A = Sgn_{0/1}\left(\sum_j x_j - n\sum_i O_i - 0.5\right),$$

where $Sgn_{0/1}(x)$ is the *signum* function that produces +1 if $x \geq 0$ and 0 otherwise, and the output of an input unit is given by

$$V_j = Sgn_{0/1}\left(x_j + \sum_i w_{ji} O_i + A - 1.5\right)$$
$$= \begin{cases} x_j, & \text{if no output } O_j \text{ is "on",} \\ x_j \wedge \Sigma_i w_{ji} O_i, & \text{otherwise.} \end{cases}$$

A reset signal R is generated only when the similarity is less than the vigilance level. (See the "ART1 learning algorithm" sidebar.)

The ART1 model can create new categories and reject an input pattern when the network reaches its capacity. However, the number of categories discovered in the input data by ART1 is sensitive to the vigilance parameter.

HOPFIELD NETWORK

Hopfield used a network *energy* function as a tool for designing recurrent networks and for understanding their dynamic behavior.[7] Hopfield's formulation made explicit the principle of storing information as dynamically stable attractors and popularized the use of recurrent networks for associative memory and for solving combinatorial optimization problems.

A Hopfield network with n units has two versions: binary and continuously valued. Let v_i be the state or output of the ith unit. For binary networks, v_i is either +1 or −1, but for continuous networks, v_i can be any value between 0 and 1. Let w_{ij} be the synapse weight on the connection from units i to j. In Hopfield networks, $w_{ij} = w_{ji}$, $\forall i, j$ (symmetric networks), and $w_{ii} = 0$, $\forall i$ (no self-feedback connections). The network dynamics for the binary Hopfield network are

$$v_i = Sgn\left(\sum_j w_{ij} v_j - \theta_i\right) \tag{4}$$

ART1 learning algorithm

1. Initialize $w_{ij} = 1$, for all i, j. Enable all the output units.
2. Present a new pattern $\mathbf{x}$.
3. Find the winner unit i^* among the enabled output units

   $\overline{\mathbf{w}}_{i^*} \cdot \mathbf{x} \geq \overline{\mathbf{w}}_i \cdot \mathbf{x}, \forall i$

4. Perform vigilance test

   $$r = \frac{\mathbf{w}_{i^*} \cdot \mathbf{x}}{\Sigma_j x_j}$$

   If $r \geq \rho$ (resonance), go to step 5. Otherwise, disable unit i^* and go to step 3 (until all the output units are disabled).
5. Update the winning weight vector $\mathbf{w}_{i^*}$, enable all the output units, and go to step 2

   $\Delta w_{ji^*} = \eta (V_j - w_{ji^*})$

6. If all output units are disabled, select one of the uncommitted output units and set its weight vector to x. If there is no uncommitted output unit (capacity is reached), the network rejects the input pattern.

The dynamic update of network states in Equation 4 can be carried out in at least two ways: *synchronously* and *asynchronously*. In a synchronous updating scheme, all units are updated simultaneously at each time step. A central clock must synchronize the process. An asynchronous updating scheme selects one unit at a time and updates its state. The unit for updating can be randomly chosen.

The energy function of the binary Hopfield network in a state $\mathbf{v} = (v_1, v_2, \ldots, v_n)^T$ is given by

$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} v_i v_j \quad (5)$$

The central property of the energy function is that as network state evolves according to the network dynamics (Equation 4), the network energy always decreases and eventually reaches a local minimum point (attractor) where the network stays with a constant energy.

Associative memory

When a set of patterns is stored in these network attractors, it can be used as an *associative memory*. Any pattern present in the basin of attraction of a stored pattern can be used as an index to retrieve it.

An associative memory usually operates in two phases: storage and retrieval. In the storage phase, the weights in the network are determined so that the attractors of the network memorize a set of p n-dimensional patterns $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^p\}$ to be stored. A generalization of the Hebbian learning rule can be used for setting connection weights w_{ij}. In the retrieval phase, the input pattern is used as the initial state of the network, and the network evolves according to its dynamics. A pattern is produced (or retrieved) when the network reaches equilibrium.

How many patterns can be stored in a network with n binary units? In other words, what is the memory capacity of a network? It is finite because a network with n binary units has a maximum of 2^n distinct states, and not all of them are attractors. Moreover, not all attractors (stable states) can store useful patterns. Spurious attractors can also store patterns different from those in the training set.[2]

It has been shown that the maximum number of random patterns that a Hopfield network can store is $P_{max} \approx 0.15n$. When the number of stored patterns $p < 0.15n$, a nearly perfect recall can be achieved. When memory patterns are orthogonal vectors instead of random patterns, more patterns can be stored. But the number of spurious attractors increases as p reaches capacity. Several learning rules have been proposed for increasing the memory capacity of Hopfield networks.[2] Note that we require n^2 connections in the network to store p n-bit patterns.

Energy minimization

Hopfield networks always evolve in the direction that leads to lower network energy. This implies that if a combinatorial optimization problem can be formulated as minimizing this energy, the Hopfield network can be used to find the optimal (or suboptimal) solution by letting the network evolve freely. In fact, any quadratic objective function can be rewritten in the form of Hopfield network energy. For example, the classic Traveling Salesman Problem can be formulated as such a problem.

APPLICATIONS

We have discussed a number of important ANN models and learning algorithms proposed in the literature. They have been widely used for solving the seven classes of problems described in the beginning of this article. Table 2 showed typical suitable tasks for ANN models and learning algorithms. Remember that to successfully work with real-world problems, you must deal with numerous design issues, including network model, network size, activation function, learning parameters, and number of training samples. We next discuss an optical character recognition (OCR) application to illustrate how multilayer feed-forward networks are successfully used in practice.

OCR deals with the problem of processing a scanned image of text and transcribing it into machine-readable form. We outline the basic components of OCR and explain how ANNs are used for character classification.

An OCR system

An OCR system usually consists of modules for preprocessing, segmentation, feature extraction, classification, and contextual processing. A paper document is scanned to produce a gray-level or binary (black-and-white) image. In the preprocessing stage, filtering is applied to remove noise, and text areas are located and converted to a binary image using a global or local adaptive thresholding method. In the segmentation step, the text image is separated into individual characters. This is a particularly difficult task with handwritten text, which contains a proliferation of touching characters. One effective technique is to break the composite pattern into smaller patterns (over-segmentation) and find the correct character segmentation points using the output of a pattern classifier.

Because of various degrees of slant, skew, and noise level, and various writing styles, recognizing segmented characters is not easy. This is evident from Figure 10, which shows the size-normalized character bitmaps of a sample set from the NIST (National Institute of Standards and Technology) hand-print character database.[16]

Schemes

Figure 11 shows the two main schemes for using ANNs in an OCR system. The first one employs an explicit feature extractor (not necessarily a neural network). For instance, contour direction features are used in Figure 11. The extracted features are passed to the input stage of a multilayer feed-forward network.[19] This scheme is very flexible in incorporating a large variety of features. The other scheme does not explicitly extract features from the raw data. The feature extraction implicitly takes place within the intermediate stages (hidden layers) of the ANN. A nice property of this scheme is that feature extraction and classification are integrated and trained simultaneously to produce optimal classification results. It is not clear whether the types of features that can be extracted by this integrated architecture are the most effective for character recognition. Moreover, this scheme requires a much larger network than the first one.

A typical example of this integrated feature extraction-classification scheme is the network developed by Le Cun et al.[20] for zip code recognition. A 16 × 16 normalized gray-level image is presented to a feed-forward network with three hidden layers. The units in the first layer are locally connected to the units in the input layer, forming a set of local feature maps. The second hidden layer is constructed in a similar way. Each unit in the second layer also combines local information coming from feature maps in the first layer.

The activation level of an output unit can be interpreted as an approximation of the a posteriori probability of the input pattern's belonging to a particular class. The output categories are ordered according to activation levels and passed to the post-processing stage. In this stage, contextual information is exploited to update the classifier's output. This could, for example, involve looking up a dictionary of admissible words, or utilizing syntactic constraints present, for example, in phone or social security numbers.

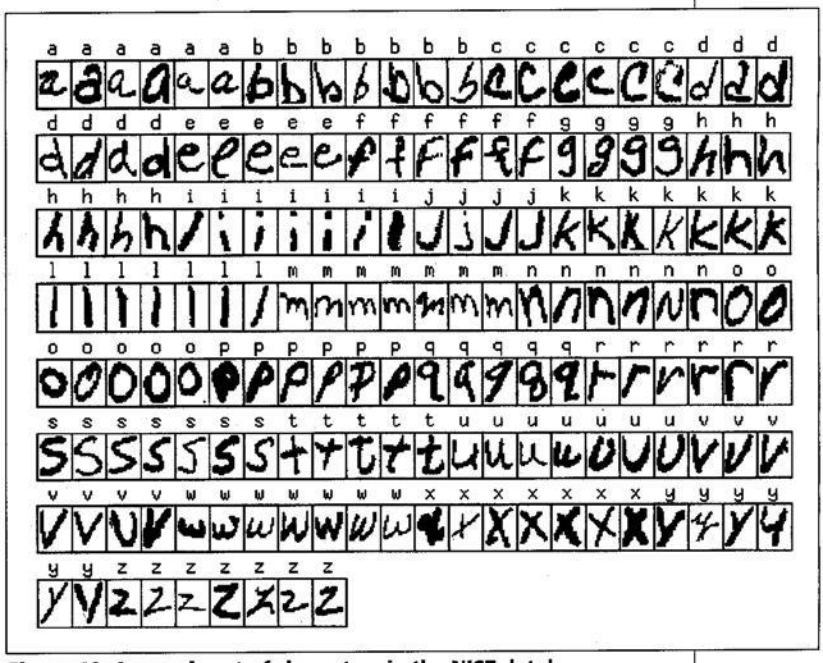


Figure 10. A sample set of characters in the NIST database.

Results

ANNs work very well in the OCR application. However, there is no conclusive evidence about their superiority over conventional statistical pattern classifiers. At the First Census Optical Character Recognition System Conference held in 1992,[18] more than 40 different handwritten character recognition systems were evaluated based on their performance on a common database. The top 10 performers used either some type of multilayer feed-forward network or a nearest neighbor-based classifier. ANNs tend to be superior in terms of speed and memory requirements compared to nearest neighbor methods. Unlike the nearest neighbor methods, classification speed using ANNs is independent of the size of the training set. The recognition accuracies of the top OCR systems on the NIST isolated (presegmented) character data were above 98 percent for digits, 96 percent for uppercase characters, and 87 percent for lowercase characters. (Low recognition accuracy for lowercase characters was largely due to the fact that the test data differed significantly from the training data, as well as being due to "ground-truth" errors.) One conclusion drawn from the test is that OCR system performance on isolated characters compares well with human performance. However, humans still outperform OCR systems on unconstrained and cursive handwritten documents.

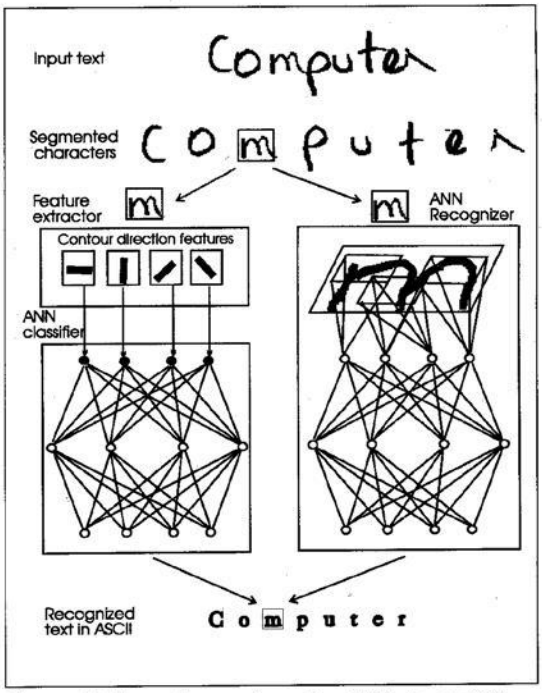


Figure 11. Two schemes for using ANNs in an OCR system.

DEVELOPMENTS IN ANNS HAVE STIMULATED a lot of enthusiasm and criticism. Some comparative studies are optimistic, some offer pessimism. For many tasks, such as pattern recognition, no one approach dominates the others. The choice of the best technique should be driven by the given application's nature. We should try to understand the capacities, assumptions, and applicability of various approaches and maximally exploit their complementary advantages to develop better intelligent systems. Such an effort may lead to a synergistic approach that combines the strengths of ANNs with other technologies to achieve significantly better performance for challenging problems. As Minsky[21] recently observed, the time has come to build systems out of diverse components. Individual modules are important, but we also need a good methodology for integration. It is clear that communication and cooperative work between

researchers working in ANNs and other disciplines will not only avoid repetitious work but (and more important) will stimulate and benefit individual disciplines. ■

Acknowledgments

We thank Richard Casey (IBM Almaden); Pat Flynn (Washington State University); William Punch, Chitra Dorai, and Kalle Karu (Michigan State University); Ali Khotanzad (Southern Methodist University); and Ishwar Sethi (Wayne State University) for their many useful suggestions.

References

1. *DARPA Neural Network Study*, AFCEA Int'l Press, Fairfax, Va., 1988.
2. J. Hertz, A. Krogh, and R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Reading, Mass., 1991.
3. S. Haykin, *Neural Networks: A Comprehensive Foundation*, MacMillan College Publishing Co., New York, 1994.
4. W.S. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bull. Mathematical Biophysics*, Vol. 5, 1943, pp. 115-133.
5. R. Rosenblatt, *Principles of Neurodynamics*, Spartan Books, New York, 1962.
6. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, Mass., 1969.
7. J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," in *Proc. Nat'l Academy of Sciences*, USA 79, 1982, pp. 2,554-2,558.
8. P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," PhD thesis, Dept. of Applied Mathematics, Harvard University, Cambridge, Mass., 1974.
9. D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, MIT Press, Cambridge, Mass., 1986.
10. J.A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, Mass., 1988.
11. S. Brunak and B. Lautrup, *Neural Networks, Computers with Intuition*, World Scientific, Singapore, 1990.
12. J. Feldman, M.A. Fanty, and N.H. Goddard, "Computing with Structured Neural Networks," *Computer*, Vol. 21, No. 3, Mar. 1988, pp. 91-103.
13. D.O. Hebb, *The Organization of Behavior*, John Wiley & Sons, New York, 1949.
14. R.P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, Vol. 4, No. 2, Apr. 1987, pp. 4-22.
15. A.K. Jain and J. Mao, "Neural Networks and Pattern Recognition," in *Computational Intelligence: Imitating Life*, J.M. Zurada, R. J. Marks II, and C.J. Robinson, eds., IEEE Press, Piscataway, N.J., 1994, pp. 194-212.
16. T. Kohonen, *Self Organization and Associative Memory*, Third Edition, Springer-Verlag, New York, 1989.
17. G.A. Carpenter and S. Grossberg, *Pattern Recognition by Self-Organizing Neural Networks*, MIT Press, Cambridge, Mass., 1991.
18. "The First Census Optical Character Recognition System Conference," R.A. Wilkinson et al., eds., . Tech. Report, NISTIR 4912, US Dept. Commerce, NIST, Gaithersburg, Md., 1992.
19. K. Mohiuddin and J. Mao, "A Comparative Study of Different Classifiers for Handprinted Character Recognition," in *Pattern Recognition in Practice IV*, E.S. Gelsema and L.N. Kanal, eds., Elsevier Science, The Netherlands, 1994, pp. 437-448.
20. Y. Le Cun et al., "Back-Propagation Applied to Handwritten Zip Code Recognition, *Neural Computation*, Vol. 1, 1989, pp. 541–551.
21. M. Minsky, "Logical Versus Analogical or Symbolic Versus Connectionist or Neat Versus Scruffy," *AI Magazine*, Vol. 65, No. 2, 1991, pp. 34-51.

Anil K. Jain *is a University Distinguished Professor and the chair of the Department of Computer Science at Michigan State University. His interests include statistical pattern recognition, exploratory pattern analysis, neural networks, Markov random fields, texture analysis, remote sensing, interpretation of range images, and 3D object recognition.*

Jain served as editor-in-chief of IEEE Transactions on Pattern Analysis and Machine Intelligence *from 1991 to 1994, and currently serves on the editorial boards of* Pattern Recognition, Pattern Recognition Letters, Journal of Mathematical Imaging, Journal of Applied Intelligence, *and* IEEE Transactions on Neural Networks. *He has coauthored, edited, and coedited numerous books in the field. Jain is a fellow of the IEEE and a speaker in the IEEE Computer Society's Distinguished Visitors Program for the Asia-Pacific region. He is a member of the IEEE Computer Society.*

Jianchang Mao *is a research staff member at the IBM Almaden Research Center. His interests include pattern recognition, neural networks, document image analysis, image processing, computer vision, and parallel computing.*

Mao received the BS degree in physics in 1983 and the MS degree in electrical engineering in 1986 from East China Normal University in Shanghai. He received the PhD in computer science from Michigan State University in 1994. Mao is the abstracts editor of IEEE Transactions on Neural Networks. *He is a member of the IEEE and the IEEE Computer Society.*

K.M. Mohiuddin *is the manager of the Document Image Analysis and Recognition project in the Computer Science Department at the IBM Almaden Research Center. He has led IBM projects on high-speed reconfigurable machines for industrial machine vision, parallel processing for scientific computing, and document imaging systems. His interests include document image analysis, handwriting recognition/OCR, data compression, and computer architecture.*

Mohiuddin received the MS and PhD degrees in electrical engineering from Stanford University in 1977 and 1982, respectively. He is an associate editor of IEEE Transactions on Pattern Analysis and Machine Intelligence. *He served on Computer's editorial board from 1984 to 1989, and is a senior member of the IEEE and a member of the IEEE Computer Society.*

Readers can contact Anil Jain at the Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, MI 48824; jain@cps.msu.edu.

Sunday, March 06, 2016
12:18 AM

NEURAL NETWORKS

by Christos Stergiou and Dimitrios Siganos

Abstract

This report is an introduction to Artificial Neural Networks. The various types of neural networks are explained and demonstrated, applications of neural networks like ANNs in medicine are described, and a detailed historical background is provided. The connection between the artificial and the real thing is also investigated and explained. Finally, the mathematical models involved are presented and demonstrated.

Contents:

1. Introduction to neural networks

1.1 What is a Neural Network?

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a

specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well.

1.2 Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras.

Many important advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few reserchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

For a more detailed description of the history click here

The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits. But the technology available at that time did not allow them to do too much.

1.3 Why use neural networks?

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyse. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

1.4 Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements(neurones) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmic approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

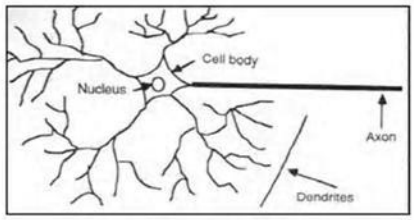
Neural networks do not perform miracles. But if used sensibly they can produce some amazing results.

Back to Contents

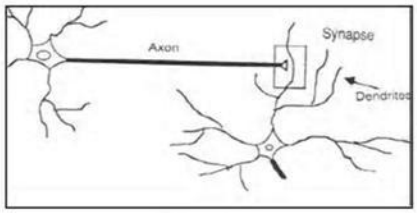
2. Human and Artificial Neurones - investigating the similarities

2.1 How the Human Brain Learns?

Much is still unknown about how the brain trains itself to process information, so theories abound. In the human brain, a typical neuron collects signals from others through a host of fine structures called *dendrites*. The neuron sends out spikes of electrical activity through a long, thin stand known as an *axon*, which splits into thousands of branches. At the end of each branch, a structure called a *synapse* converts the activity from the axon into electrical effects that inhibit or excite activity from the axon into electrical effects that inhibit or excite activity in the connected neurones. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.



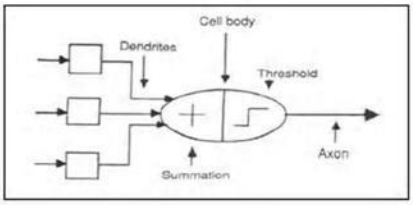
Components of a neuron



The synapse

2.2 From Human Neurones to Artificial Neurones

We conduct these neural networks by first trying to deduce the essential features of neurones and their interconnections. We then typically program a computer to simulate these features. However because our knowledge of neurones is incomplete and our computing power is limited, our models are necessarily gross idealisations of real networks of neurones.



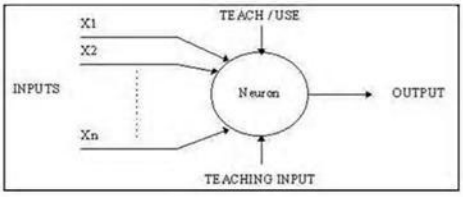
The neuron model

Back to Contents

3. An engineering approach

3.1 A simple neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.



A simple neuron

3.2 Firing rules

The firing rule is an important concept in neural networks and accounts for their high flexibility. A firing rule determines how one calculates whether a neuron should fire for any input pattern. It relates to all the input patterns, not only the ones on which the node was trained.

A simple firing rule can be implemented by using Hamming distance technique. The rule goes as follows:

> Take a collection of training patterns for a node, some of which cause it to fire (the 1-taught set of patterns) and others which prevent it from doing so (the 0-taught set). Then the patterns not in the collection cause the node to fire if, on comparison , they have more input elements in common with the 'nearest' pattern in the 1-taught set than with the 'nearest' pattern in the 0-taught set. If there is a tie, then the pattern remains in the undefined state.

For example, a 3-input neuron is taught to output 1 when the input (X1,X2 and X3) is 111 or 101 and to output 0 when the input is 000 or 001. Then, before applying the firing rule, the truth table is;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0/1	0/1	0/1	1	0/1	1

As an example of the way the firing rule is applied, take the pattern 010. It differs from 000 in 1 element, from 001 in 2 elements, from 101 in 3 elements and from 111 in 2 elements. Therefore, the 'nearest' pattern is 000 which belongs in the 0-taught set. Thus the firing rule requires that the neuron should not fire when the input is 001. On the other hand, 011 is equally distant from two taught patterns that have different outputs and thus the output stays undefined (0/1).

By applying the firing in every column the following truth table is obtained;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0	0/1	0/1	1	1	1

The difference between the two truth tables is called the *generalisation of the neuron*. Therefore the firing rule gives the neuron a sense of similarity and enables it to respond 'sensibly' to patterns not seen during training.

3.3 Pattern Recognition - an example

An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward (figure 1) neural network that has been trained accordingly. During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the input pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is given as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

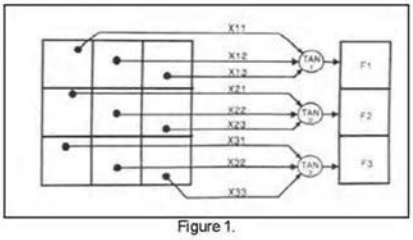


Figure 1.

For example:
The network of figure 1 is trained to recognise the patterns T and H. The associated patterns are all black and all white respectively as shown below.

If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurones after generalisation are;

X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Top neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

Middle neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1

OUT:	1	0	1	1	0	0	1	0

Bottom neuron

From the tables it can be seen the following associasions can be extracted:

In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern.

Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.

Here, the top row is 2 errors away from the a T and 3 from an H. So the top output is black. The middle row is 1 error away from both T and H so the output is random. The bottom row is 1 error away from T and 2 away from H. Therefore the output is black. The total output of the network is still in favour of the T shape.

3.4 A more complicated neuron

The previous neuron doesn't do anything that conventional conventional computers don't do already. A more sophisticated neuron (figure 2) is the McCulloch and Pitts model (MCP). The difference from the previous model is that the inputs are 'weighted', the effect that each input has at decision making is dependent on the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs are then added together and if they exceed a pre-set threshold value, the neuron fires. In any other case the neuron does not fire.

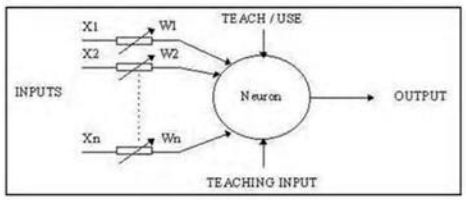


Figure 2. An MCP neuron

In mathematical terms, the neuron fires if and only if;

$$X_1W_1 + X_2W_2 + X_3W_3 + \ldots > T$$

The addition of input weights and of the threshold makes this neuron a very flexible and powerful one. The MCP neuron has the ability to adapt to a particular situation by changing its weights and/or threshold. Various algorithms exist that cause the neuron to 'adapt'; the most used ones are the Delta rule and the back error propagation. The former is used in feed-forward networks and the latter in feedback networks.

Back to Contents

4 Architecture of neural networks

4.1 Feed-forward networks

Feed-forward ANNs (figure 1) allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organisation is also referred to as bottom-up or top-down.

4.2 Feedback networks

Feedback networks (figure 1) can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations.

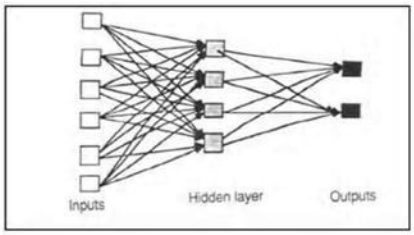


Figure 4.1 An example of a simple feedforward network

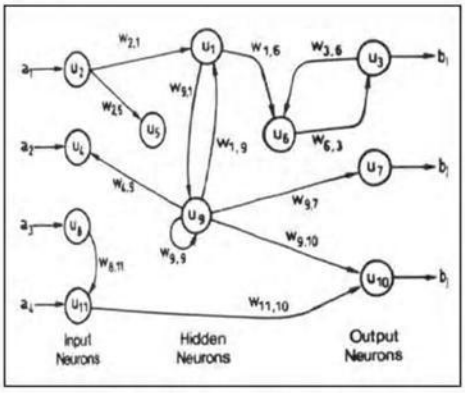


Figure 4.2 An example of a complicated network

4.3 Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units: a layer of "**input**" units is connected to a layer of "**hidden**" units, which is connected to a layer of "**output**" units. (see Figure 4.1)

- The activity of the input units represents the raw information that is fed into the network.
- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish single-layer and multi-layer architectures. The single-layer organisation, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organisations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

4.4 Perceptrons

The most influential work on neural nets in the 60's went under the heading of 'perceptrons' a term coined by Frank Rosenblatt. The perceptron (figure 4.4) turns out to be an MCP model (neuron with weighted inputs) with some additional, fixed, pre--processing. Units labelled A1, A2, Aj , Ap are called association units and their task is to extract specific, localised featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They were mainly used in pattern recognition even though their capabilities extended a lot more.

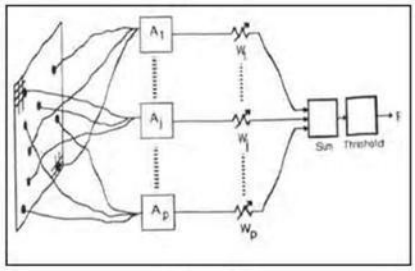


Figure 4.4

In 1969 Minsky and Papert wrote a book in which they described the limitations of single layer Perceptrons. The impact that the book had was tremendous and caused a lot of neural network researchers to loose their interest. The book was very well written and showed mathematically that *single layer* perceptrons could not do some basic pattern recognition operations like determining the parity of a shape or determining whether a shape is connected or not. What they did not realised, until the 80's, is that given the appropriate training, multilevel perceptrons can do these operations.

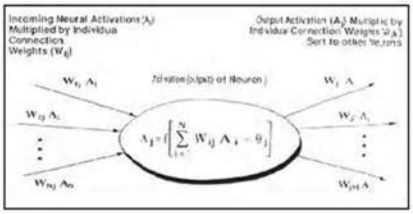
Back to Contents

5. The Learning Process

The memorisation of patterns and the subsequent response of the network can be categorised into two general paradigms:

- **associative mapping** in which the network learns to produce a particular pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associtive mapping can generally be broken down into two mechanisms:
  - *auto-association*: an input pattern is associated with itself and the states of input and output units coincide. This is used to provide pattern completition, ie to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network actually stores pairs of patterns building an association between two sets of patterns.
  - *hetero-association*: is related to two recall mechanisms:
    - *nearest-neighbour* recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and
    - *interpolative* recall, where the output pattern is a similarity dependent interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, ie when there is a fixed set of categories into which the input patterns are to be classified.
- **regularity detection** in which units learn to respond to particular properties of the input patterns. Whereas in asssociative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network posseses knowledge which is contained in the values of the connections weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights.



Information is stored in the weight matrix W of a neural network. Learning is the determination of the weights. Following the way learning is performed, we can distinguish two major categories of neural networks:

- **fixed networks** in which the weights cannot be changed, ie dW/dt=0. In such networks, the weights are fixed a priori according to the problem to solve.
- **adaptive networks** which are able to change their weights, ie dW/dt not= 0.

All learning methods used for adaptive neural networks can be classified into two major categories:

- **Supervised learning** which incorporates an external teacher, so that each output unit is told what its desired response to input signals ought to be. During the learning process global information may be required. Paradigms of supervised learning include error-correction learning, reinforcement learning and stochastic learning.
  An important issue conserning supervised learning is the problem of error convergence, ie the minimisation of error between the desired and computed unit values. The aim is to determine a set of weights which minimises the error. One well-known method, which is common to many learning paradigms is the least mean square (LMS) convergence.
- **Unsupervised learning** uses no external teacher and is based upon only local information. It is also referred to as self-organisation, in the sense that it self-organises data presented to the network and detects their emergent collective properties. Paradigms of unsupervised learning are Hebbian lerning and competitive learning.
  Ano2.2 From Human Neurones to Artificial Neuronesther aspect of learning concerns the distinction or not of a seperate phase, during which the network is trained, and a subsequent operation phase. We say that a neural network learns off-line if the learning phase and the operation phase are distinct. A neural network learns on-line if it learns and operates at the same time. Usually, supervised learning is performed off-line, whereas usupervised learning is performed on-line.

5.1 Transfer Function

The behaviour of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- linear (or ramp)
- threshold
- sigmoid

For **linear units**, the output activity is proportional to the total weighted output.

For **threshold units**, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.

For **sigmoid units**, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurones than do linear or threshold units, but all three must be considered rough approximations.

To make a neural network that performs some specific task, we must choose how the units are connected to one another (see figure 4.1), and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

We can teach a three-layer network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

5.2 An Example to illustrate the above teaching procedure:

Assume that we want a network to recognise hand-written digits. We might use an array of, say, 256 sensors, each recording the presence or absence of ink in a small area of a single digit. The network would therefore need 256 input units (one for each sensor), 10 output units (one for each kind of digit) and a number of hidden units.

For each kind of digit recorded by the sensors, the network should produce high activity in the appropriate output unit and low activity in the other output units.

To train the network, we present an image of a digit and compare the actual activity of the 10 output units with the desired activity. We then calculate the error, which is defined as the square of the difference between the actual and the desired activities. Next we change the weight of each connection so as to reduce the error.We repeat this training process for many different images of each different images of each kind of digit until the network classifies every image correctly.

To implement this procedure we need to calculate the error derivative for the weight (EW) in order to change the weight by an amount that is proportional to the rate at which the error changes as the weight is changed. One way to calculate the EW is to perturb a weight slightly and observe how the error changes. But that method is inefficient because it requires a separate perturbation for each of the many weights.

Another way to calculate the EW is to use the Back-propagation algorithm which is described below, and has become nowadays one of the most important tools for training neural networks. It was developed independently by two teams, one (Fogelman-Soulie, Gallinari and Le Cun) in France, the other (Rumelhart, Hinton and Williams) in U.S.

5.3 The Back-Propagation Algorithm

In order to train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and the actual output is reduced. This process requires that the neural network compute the error derivative of the weights (**EW**). In other words, it must calculate how the error changes as each weight is increased or decreased slightly. The back propagation algorithm is the most widely used method for determining the **EW**.

The back-propagation algorithm is easiest to understand if all the units in the network are linear. The algorithm computes each **EW** by first computing the **EA**, the rate at which the error changes as the activity level of a unit is changed. For output units, the **EA** is simply the difference between the actual and the desired output. To compute the **EA** for a hidden unit in the layer just before the output layer, we first identify all the weights between that hidden unit and the output units to which it is connected. We then multiply those weights by the **EA**s of those output units and add the products. This sum equals the **EA** for the chosen hidden unit. After calculating all the **EA**s in the hidden layer just before the output layer, we can compute in like fashion the **EA**s for other layers, moving from layer to layer in a direction opposite to the way activities propagate through the network. This is what gives back propagation its name. Once the **EA** has been computed for a unit, it is straight forward to compute the **EW** for each incoming connection of the unit. The **EW** is the product of the EA and the activity through the incoming connection.

Note that for non-linear units, (see Appendix C) the back-propagation algorithm includes an extra step. Before back-propagating, the **EA** must be converted into the **EI**, the rate at which the error changes as the total input received by a unit is changed.

Back to Contents

6. Applications of neural networks

6.1 Neural Networks in Practice

Given this description of neural networks and how they work, what real world applications are they suited for? Neural networks have broad applicability to real world business problems. In fact, they have already been successfully applied in many industries.

Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs including:

- sales forecasting
- industrial process control
- customer research
- data validation
- risk management
- target marketing

But to give you some more specific examples; ANN are also used in the following specific paradigms: recognition of speakers in communications; diagnosis of hepatitis; recovery of telecommunications from faulty software; interpretation of multimeaning Chinese words; undersea mine detection; texture analysis; three-dimensional object recognition; hand-written word recognition; and facial recognition.

6.2 Neural networks in medicine

Artificial Neural Networks (ANN) are currently a 'hot' research area in medicine and it is believed that they will receive extensive application to biomedical systems in the next few years. At the moment, the research is mostly on modelling parts of the human body and recognising diseases from various scans (e.g. cardiograms, CAT scans, ultrasonic scans, etc.).

Neural networks are ideal in recognising diseases using scans since there is no need to provide a specific algorithm on how to identify the disease. Neural networks learn by example so the details of how to recognise the disease are not needed. What is needed is a set of examples that are representative of all the variations of the disease. The quantity of examples is not as important as the 'quantity'. The examples need to be selected very carefully if the system is to perform reliably and efficiently.

6.2.1 Modelling and Diagnosing the Cardiovascular System

Neural Networks are used experimentally to model the human cardiovascular system. Diagnosis can be achieved by building a model of the cardiovascular system of an individual and comparing it with the real time physiological measurements taken from the patient. If this routine is carried out regularly, potential harmful medical conditions can be detected at an early stage and thus make the process of combating the disease much easier.

A model of an individual's cardiovascular system must mimic the relationship among physiological variables (i.e., heart rate, systolic and diastolic blood pressures, and breathing rate) at different physical activity levels. If a model is adapted to an individual, then it becomes a model of the physical condition of that individual. The simulator will have to be able to adapt to the features of any individual without the supervision of an expert. This calls for a neural network.

Another reason that justifies the use of ANN technology, is the ability of ANNs to provide sensor fusion which is the combining of values from several different sensors. Sensor fusion enables the ANNs to learn complex relationships among the individual sensor values, which would otherwise be lost if the values were individually analysed. In medical modelling and diagnosis, this implies that even though each sensor in a set may be sensitive only to a specific physiological variable, ANNs are capable of detecting complex medical conditions by fusing the data from the individual biomedical sensors.

6.2.2 Electronic noses

ANNs are used experimentally to implement electronic noses. Electronic noses have several potential applications in telemedicine. Telemedicine is the practice of medicine over long distances via a communication link. The electronic nose would identify odours in the remote surgical environment. These identified odours would then be electronically transmitted to another site where an door generation system would recreate them. Because the sense of smell can be an important sense to the surgeon, telesmell would enhance telepresent surgery.
For more information on telemedicine and telepresent surgery click here.

6.2.3 Instant Physician

An application developed in the mid-1980s called the "instant physician" trained an autoassociative memory neural network to store a large number of medical records, each of which includes information on symptoms, diagnosis, and treatment for a particular case. After training, the net can be presented with input consisting of a set of symptoms; it will then find the full stored pattern that represents the "best" diagnosis and treatment.

6.3 Neural Networks in business

Business is a diverted field with several general areas of specialisation such as accounting or financial analysis. Almost any neural network application would fit into one business area or financial analysis.
There is some potential for using neural networks for business purposes, including resource allocation and scheduling. There is also a strong potential for using neural networks for database mining, that is, searching for patterns implicit within the explicitly stored information in databases. Most of the funded work in this area is classified as proprietary. Thus, it is not possible to report on the full extent of the work going on. Most work is applying neural networks, such as the Hopfield-Tank network for optimization and scheduling.

6.3.1 Marketing

There is a marketing application which has been integrated with a neural network system. The Airline Marketing Tactician (a trademark abbreviated as AMT) is a computer system made of various intelligent technologies including expert systems. A feedforward neural network is integrated with the AMT and was trained using back-propagation to assist the marketing control of airline seat allocations. The adaptive neural approach was amenable to rule expression. Additionaly, the application's environment changed rapidly and constantly, which required a continuously adaptive solution. The system is used to monitor and recommend booking advice for each departure. Such information has a direct impact on the profitability of an airline and can provide a technological advantage for users of the system. [Hutchison & Stephens, 1987]

While it is significant that neural networks have been applied to this problem, it is also important to see that this intelligent technology can be integrated with expert systems and other approaches to make a functional system. Neural networks were used to discover the influence of undefined interactions by the various variables. While these interactions were not defined, they were used by the neural system to develop useful conclusions. It is also noteworthy to see that neural networks can influence the bottom line.

6.3.2 Credit Evaluation

The HNC company, founded by Robert Hecht-Nielsen, has developed several neural network applications. One of them is the Credit Scoring system which increase the profitability of the existing model up to 27%. The HNC neural systems were also applied to mortgage screening. A neural network automated mortgage insurance underwritting system was developed by the Nestor Company. This system was trained with 5048 applications of which 2597 were certified. The data related to property and borrower qualifications. In a conservative mode the system agreed on the underwritters on 97% of the cases. In the liberal model the system agreed 84% of the cases. This is system run on an Apollo DN3000 and used 250K memory while processing a case file in approximately 1 sec.

Back to Contents

7. Conclusion

The computing world has a lot to gain fron neural networks. Their ability to learn by example makes them very flexible and powerful. Furthermore there is no need to devise an algorithm in order to perform a specific task; i.e. there is no need to understand the internal mechanisms of that task. They are also very well suited for real time systems because of their fast responseand computational times which are due to their parallel architecture.

Neural networks also contribute to other areas of research such as neurology and psychology. They are regularly used to model parts of living organisms and to investigate the internal mechanisms of the brain.

Perhaps the most exciting aspect of neural networks is the possibility that some day 'consious' networks might be produced. There is a number of scientists arguing that conciousness is a 'mechanical' property and that 'consious' neural networks are a realistic possibility.

Finally, I would like to state that even though neural networks have a huge potential we will only get the best of them when they are intergrated with computing, AI, fuzzy logic and related subjects.

Back to Contents

References:

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html
3. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
4. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html
5. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht
6. Neural Networks by Eric Davalo and Patrick Naim
7. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
8. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
9. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab. Neural Networks, Eric Davalo and Patrick Naim
10. Assimov, I (1984, 1950), Robot, Ballatine, New York.
11. Electronic Noses for Telemedicine
    http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html
12. Pattern Recognition of Pathology Images
    http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html

Back to Contents

Appendix A - Historical background in detail

The history of neural networks that was described above can be divided into several periods:

1. **First Attempts**: There were some initial simulations using formal logic. McCulloch and Pitts (1943) developed models of neural networks based on their understanding of neurology. These models made several assumptions about how neurons worked. Their networks were based on simple neurons which were considered to be binary devices with fixed thresholds. The results of their model were simple logic functions such as "a or b" and "a and b". Another attempt was by using computer simulations. Two groups (Farley and Clark, 1954; Rochester, Holland, Haibit and Duda, 1956). The first group (IBM reserchers) maintained closed contact with neuroscientists at McGill University. So whenever their models did not work, they consulted the neuroscientists. This interaction established a multidiscilinary trend which continues to the present day.
2. **Promising & Emerging Technology:** Not only was neroscience influential in the development of neural networks, but psychologists and engineers also contributed to the progress of neural network simulations. Rosenblatt (1958) stirred considerable interest and activity in the field when he designed and developed the Perceptron. The Perceptron had three layers with the middle layer known as the association layer.This system could learn to connect or associate a given input to a random output unit.

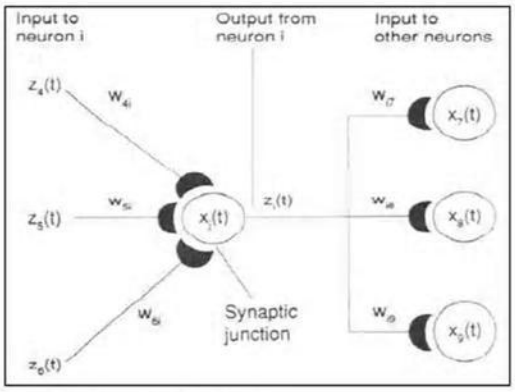
Another system was the ADALINE (ADAptive LInear Element) which was developed in 1960 by Widrow and Hoff (of Stanford University). The ADALINE was an analogue electronic device made from simple components. The method used for learning was different to that of the Perceptron, it employed the Least-Mean-Squares (LMS) learning rule.

3. **Period of Frustration & Disrepute:** In 1969 Minsky and Papert wrote a book in which they generalised the limitations of single layer Perceptrons to multilayered systems. In the book they said: "...our intuitive judgment that the extension (to multilayer systems) is sterile". The significant result of their book was to eliminate funding for research with neural network simulations. The conclusions supported the disenhantment of reserchers in the field. As a result, considerable prejudice against this field was activated.
4. **Innovation:** Although public interest and available funding were minimal, several researchers continued working to develop neuromorphically based computaional methods for problems such as pattern recognition.
   During this period several paradigms were generated which modern work continues to enhance.Grossberg's (Steve Grossberg and Gail Carpenter in 1988) influence founded a school of thought which explores resonating algorithms. They developed the ART (Adaptive Resonance Theory) networks based on biologically plausible models. Anderson and Kohonen developed associative techniques independent of each other. Klopf (A. Henry Klopf) in 1972, developed a basis for learning in artificial neurons based on a biological principle for neuronal learning called heterostasis.
   Werbos (Paul Werbos 1974) developed and used the back-propagation learning method, however several years passed before this approach was popularized. Back-propagation nets are probably the most well known and widely applied of the neural networks today. In essence, the back-propagation net. is a Perceptron with multiple layers, a different thershold function in the artificial neuron, and a more robust and capable learning rule.
   Amari (A. Shun-Ichi 1967) was involved with theoretical developments: he published a paper which established a mathematical theory for a learning basis (error-correction method) dealing with adaptive patern classification. While Fukushima (F. Kunihiko) developed a step wise trained multilayered neural network for interpretation of handwritten characters. The original network was published in 1975 and was called the Cognitron.
5. **Re-Emergence:** Progress during the late 1970s and early 1980s was important to the re-emergence on interest in the neural network field. Several factors influenced this movement. For example, comprehensive books and conferences provided a forum for people in diverse fields with specialized technical languages, and the response to conferences and publications was quite positive. The news media picked up on the increased activity and tutorials helped disseminate the technology. Academic programs appeared and courses were inroduced at most major Universities (in US and Europe). Attention is now focused on funding levels throughout Europe, Japan and the US and as this funding becomes available, several new commercial with applications in industry and finacial institutions are emerging.
6. **Today:** Significant progress has been made in the field of neural networks-enough to attract a great deal of attention and fund further research. Advancement beyond current commercial applications appears to be possible, and research is advancing the field on many fronts. Neurally based chips are emerging and applications to complex problems developing. Clearly, today is a period of transition for neural network technology.

Back to Contents

Appendix B - The back-propagation Algorithm - a mathematical approach

Units are connected to one another. Connections correspond to the edges of the underlying directed graph. There is a real number associated with each connection, which is called the weight of the connection. We denote by W_{ij} the weight of the connection from unit u_i to unit u_j. It is then convenient to represent the pattern of connectivity in the network by a weight matrix W whose elements are the weights W_{ij}. Two types of connection are usually distinguished: excitatory and inhibitory. A positive weight represents an excitatory connection whereas a negative weight represents an inhibitory connection. The pattern of connectivity characterises the architecture of the network.



A unit in the output layer determines its activity by following a two step procedure.

- First, it computes the total weighted input x_j, using the formula:

$$x_j = \sum_i y_i w_{ij}$$

  where y_i is the activity level of the jth unit in the previous layer and W_{ij} is the weight of the connection between the ith and the jth unit.

- Next, the unit calculates the activity y_j using some function of the total weighted input. Typically we use the sigmoid function:

$$y_j = \frac{1}{1+e^{-x_j}}$$

Once the activities of all output units have been determined, the network computes the error E, which is defined by the expression:

$$E = \frac{1}{2}\sum_i (y_i - d_i)^2$$

where y_i is the activity level of the jth unit in the top layer and d_i is the desired output of the jth unit.

The back-propagation algorithm consists of four steps:

1. Compute how fast the error changes as the activity of an output unit is changed. This error derivative (EA) is the difference between the actual and the desired activity.

$$EA_j = \frac{\partial E}{\partial y_j} = y_j - d_j$$

2. Compute how fast the error changes as the total input received by an output unit is changed. This quantity (EI) is the answer from step 1 multiplied by the rate at which the output of a unit changes as its total input is changed.

$$EI_j = \frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \times \frac{dy_j}{dx_j} = EA_j y_j (1 - y_j)$$

3. Compute how fast the error changes as a weight on the connection into an output unit is changed. This quantity (EW) is the answer from step 2 multiplied by the activity level of the unit from which the connection emanates.

$$EW_{ij} = \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial W_{ij}} = EI_j y_i$$

4. Compute how fast the error changes as the activity of a unit in the previous layer is changed. This crucial step allows back propagation to be applied to multilayer networks. When the activity of a unit in the previous layer changes, it affects the activites of all the output units to which it is connected. So to compute the overall effect on the error, we add together all these seperate effects on output units. But each effect is simple to calculate. It is the answer in step 2 multiplied by the weight on the connection to that output unit.

$$EA_i = \frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial y_i} = \sum_j EI_j W_{ij}$$

By using steps 2 and 4, we can convert the EAs of one layer of units into EAs for the previous layer. This procedure can be repeated to get the EAs for as many previous layers as desired. Once we know the EA of a unit, we can use steps 2 and 3 to compute the EWs on its incoming connections.

Back to Contents

Appendix C - References used throughout the review

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html
3. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht
4. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
5. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html
6. Electronic Noses for Telemedicine
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html
7. An Introduction to Computing with Neural Nets (Richard P. Lipmann, IEEE ASSP Magazine, April 1987)
8. Pattern Recognition of Pathology Images
   http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html
9. Developments in autonomous vehicle navigation. Stefan Neuber, Jos Nijhuis, Lambert Spaanenburg. Institut fur Mikroelektronik Stuttgart, Allmandring 30A, 7000 Stuttgart-80
10. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
11. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab.
12. Neural Networks, Eric Davalo and Patrick Naim.
13. Assimov, I (1984, 1950), Robot, Ballatine, New York.
14. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
15. Alkon, D.L 1989, Memory Storage and Neural Systems, Scientific American, July, 42-50
16. Minsky and Papert (1969) Perceptrons, An introduction to computational geometry, MIT press, expanded edition.
17. Neural computers, NATO ASI series, Editors: Rolf Eckmiller Christoph v. d. Malsburg

Back to Contents

Theme Feature

Artificial Neural Networks: A Tutorial

Anil K. Jain
Michigan State University

Jianchang Mao
K.M. Mohiuddin
IBM Almaden Research Center

These massively parallel systems with large numbers of interconnected simple processors may solve a variety of challenging computational problems. This tutorial provides the background and the basics.

Numerous advances have been made in developing intelligent systems, some inspired by biological neural networks. Researchers from many scientific disciplines are designing artificial neural networks (ANNs) to solve a variety of problems in pattern recognition, prediction, optimization, associative memory, and control (see the "Challenging problems" sidebar).

Conventional approaches have been proposed for solving these problems. Although successful applications can be found in certain well-constrained environments, none is flexible enough to perform well outside its domain. ANNs provide exciting alternatives, and many applications could benefit from using them.[1-3]

This article is for those readers with little or no knowledge of ANNs to help them understand the other articles in this issue of *Computer*. We discuss the motivations behind the development of ANNs, describe the basic biological neuron and the artificial computational model, outline network architectures and learning processes, and present some of the most commonly used ANN models. We conclude with character recognition, a successful ANN application.

WHY ARTIFICIAL NEURAL NETWORKS?

The long course of evolution has given the human brain many desirable characteristics not present in von Neumann or modern parallel computers. These include

- massive parallelism,
- distributed representation and computation,
- learning ability,
- generalization ability,
- adaptivity,
- inherent contextual information processing,
- fault tolerance, and
- low energy consumption.

It is hoped that devices based on biological neural networks will possess some of these desirable characteristics.

Modern digital computers outperform humans in the domain of numeric computation and related symbol manipulation. However, humans can effortlessly solve complex perceptual problems (like recognizing a man in a crowd from a mere glimpse of his face) at such a high speed and extent as to dwarf the world's fastest computer. Why is there such a remarkable difference in their performance? The biological neural system architecture is completely different from the von Neumann architecture (see Table 1). This difference significantly affects the type of functions each computational model can best perform.

Numerous efforts to develop "intelligent" programs based on von Neumann's centralized architecture have not resulted in general-purpose intelligent programs. Inspired by biological neural networks, ANNs are massively parallel computing systems consisting of an exremely large number of simple processors with many interconnections. ANN models attempt to use some "organizational" principles believed to be used in the human

0018-9162/96/$5.00 © 1996 IEEE

Challenging problems

Let us consider the following problems of interest to computer scientists and engineers.

Pattern classification

The task of pattern classification is to assign an input pattern (like a speech waveform or handwritten symbol) represented by a feature vector to one of many prespecified classes (see Figure A1). Well-known applications include character recognition, speech recognition, EEG waveform classification, blood cell classification, and printed circuit board inspection.

Clustering/categorization

In clustering, also known as unsupervised pattern classification, there are no training data with known class labels. A clustering algorithm explores the similarity between the patterns and places similar patterns in a cluster (see Figure A2). Well-known clustering applications include data mining, data compression, and exploratory data analysis.

Function approximation

Suppose a set of n labeled training patterns (input-output pairs), $\{(\mathbf{x}_1, y_1),(\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, have been generated from an unknown function $\mu(\mathbf{x})$ (subject to noise). The task of function approximation is to find an estimate, say $\hat{\mu}$, of the unknown function μ (Figure A3). Various engineering and scientific modeling problems require function approximation.

Prediction/forecasting

Given a set of n samples $\{y(t_1), y(t_2), \ldots, y(t_n)\}$ in a time sequence, $t_1, t_2, \ldots, t_n$, the task is to predict the sample $y(t_{n+1})$ at some future time t_{n+1}. Prediction/forecasting has a significant impact on decision-making in business, science, and engineering. Stock market prediction and weather forecasting are typical applications of prediction/forecasting techniques (see Figure A4).

Optimization

A wide variety of problems in mathematics, statistics, engineering, science, medicine, and economics can be posed as optimization problems. The goal of an optimization algorithm is to find a solution satisfying a set of constraints such that an objective function is maximized or minimized. The Traveling Salesman Problem (TSP), an *NP-complete* problem, is a classic example (see Figure A5).

Content-addressable memory

In the von Neumann model of computation, an entry in memory is accessed only through its address, which is independent of the content in the memory. Moreover, if a small error is made in calculating the address, a completely different item can be retrieved. Associative memory or content-addressable memory, as the name implies, can be accessed by their content. The content in the memory can be recalled even by a partial input or distorted content (see Figure A6). Associative memory is extremely desirable in building multimedia information databases.

Control

Consider a dynamic system defined by a tuple $\{u(t), y(t)\}$, where $u(t)$ is the control input and $y(t)$ is the resulting output of the system at time t. In model-reference adaptive control, the goal is to generate a control input $u(t)$ such that the system follows a desired trajectory determined by the reference model. An example is engine idle-speed control (Figure A7).

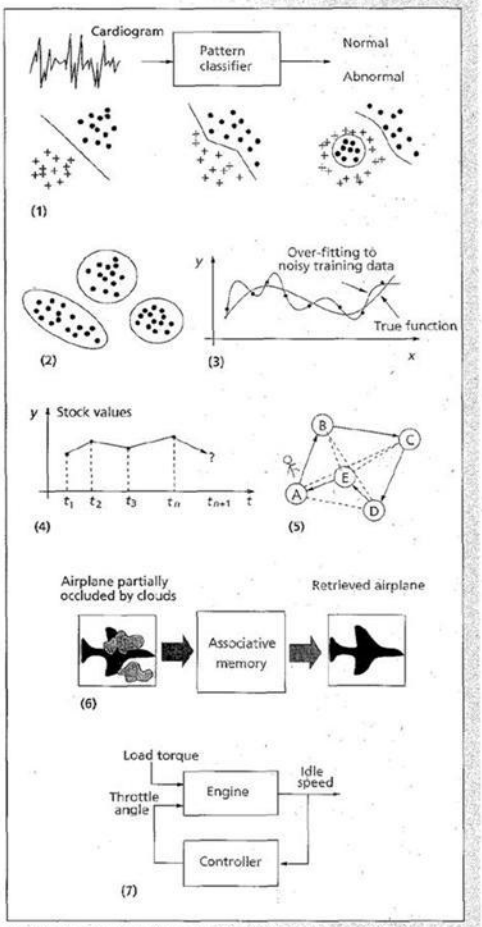


Figure A. Tasks that neural networks can perform: (1) pattern classification; (2) clustering/categorization; (3) function approximation; (4) prediction/forecasting; (5) optimization (a TSP problem example); (6) retrieval by content; and (7) control (engine idle speed). (Adapted from *DARPA Neural Network Study*[1])

brain. Modeling a biological nervous system using ANNs can also increase our understanding of biological functions. State-of-the-art computer hardware technology (such as *VLSI* and optical) has made this modeling feasible.

A thorough study of ANNs requires knowledge of neurophysiology, cognitive science/psychology, physics (statistical mechanics), control theory, computer science, artificial intelligence, statistics/mathematics, pattern recognition, computer vision, parallel processing, and hardware (digital/analog/VLSI/optical). New developments in these disciplines continuously nourish the field. On the other hand, ANNs also provide an impetus to these disciplines in the form of new tools and representations. This symbiosis is necessary for the vitality of neural network research. Communications among these disciplines ought to be encouraged.

Brief historical review

ANN research has experienced three periods of extensive activity. The first peak in the 1940s was due to McCulloch and Pitts' pioneering work.[4] The second occurred in the 1960s with Rosenblatt's perceptron convergence theorem[5] and Minsky and Papert's work showing the limitations of a simple perceptron.[6] Minsky and Papert's results dampened the enthusiasm of most researchers, especially those in the computer science community. The resulting lull in neural network research lasted almost 20 years. Since the early 1980s, ANNs have received considerable renewed interest. The major developments behind this resurgence include Hopfield's energy approach[7] in 1982 and the back-propagation learning algorithm for multilayer perceptrons (multilayer feedforward networks) first proposed by Werbos,[8] reinvented several times, and then popularized by Rumelhart et al.[9] in 1986. Anderson and Rosenfeld[10] provide a detailed historical account of ANN developments.

Biological neural networks

A *neuron* (or nerve cell) is a special biological cell that processes information (see Figure 1). It is composed of a cell body, or *soma*, and two types of out-reaching tree-like branches: the *axon* and the *dendrites*. The cell body has a nucleus that contains information about hereditary traits and a plasma that holds the molecular equipment for producing material needed by the neuron. A neuron receives signals (impulses) from other neurons through its dendrites (receivers) and transmits signals generated by its cell body along the axon (transmitter), which eventually branches into strands and substrands. At the terminals of these strands are the *synapses*. A synapse is an elementary structure and functional unit between two neurons (an axon strand of one neuron and a dendrite of another). When the impulse reaches the synapse's terminal, certain chemicals called neurotransmitters are released. The neurotransmitters diffuse across the synaptic gap, to enhance or inhibit, depending on the type of the synapse, the receptor neuron's own tendency to emit electrical impulses. The synapse's effectiveness can be adjusted by the signals passing through it so that the synapses can *learn* from the activities in which they participate. This dependence on history acts as a memory, which is possibly responsible for human memory.

The cerebral cortex in humans is a large flat sheet of neurons about 2 to 3 millimeters thick with a surface area of about 2,200 cm^2, about twice the area of a standard computer keyboard. The cerebral cortex contains about 10^{11} neurons, which is approximately the number of stars in the Milky Way.[11] Neurons are massively connected, much more complex and dense than telephone networks. Each neuron is connected to 10^3 to 10^4 other neurons. In total, the human brain contains approximately 10^{14} to 10^{15} interconnections.

Neurons communicate through a very short train of pulses, typically milliseconds in duration. The *message* is modulated on the pulse-transmission frequency. This frequency can vary from a few to several hundred hertz, which is a million times slower than the fastest switching speed in electronic circuits. However, complex perceptual decisions such as face recognition are typically made by humans within a few hundred milliseconds. These decisions are made by a network of neurons whose operational speed is only a few milliseconds. This implies that the computations cannot take more than about 100 serial stages. In other

Table 1. Von Neumann computer versus biological neural system.

	Von Neumann computer	Biological neural system
Processor	Complex High speed One or a few	Simple Low speed A large number
Memory	Separate from a processor Localized Noncontent addressable	Integrated into processor Distributed Content addressable
Computing	Centralized Sequential Stored programs	Distributed Parallel Self-learning
Reliability	Very vulnerable	Robust
Expertise	Numerical and symbolic manipulations	Perceptual problems
Operating environment	Well-defined, well-constrained	Poorly defined, unconstrained

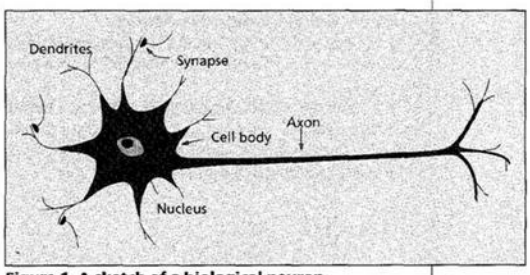


Figure 1. A sketch of a biological neuron.

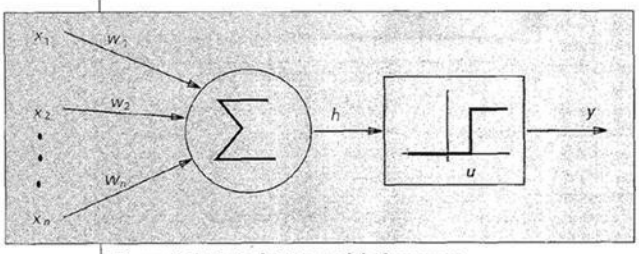


Figure 2. McCulloch-Pitts model of a neuron.

words, the brain runs parallel programs that are about 100 steps long for such perceptual tasks. This is known as the *hundred step rule.*[12] The same timing considerations show that the amount of information sent from one neuron to another must be very small (a few bits). This implies that critical information is not transmitted directly, but captured and distributed in the interconnections—hence the name, *connectionist* model, used to describe ANNs.

Interested readers can find more introductory and easily comprehensible material on biological neurons and neural networks in Brunak and Lautrup.[11]

ANN OVERVIEW

Computational models of neurons

McCulloch and Pitts[4] proposed a binary threshold unit as a computational model for an artificial neuron (see Figure 2).

This mathematical neuron computes a weighted sum of its n input signals, $x_j, j = 1, 2, \ldots, n$, and generates an output of 1 if this sum is above a certain threshold u. Otherwise, an output of 0 results. Mathematically,

$$y = \theta\left(\sum_{j=1}^{n} w_j x_j - u\right),$$

where $\theta(\cdot)$ is a unit step function at 0, and w_j is the synapse weight associated with the jth input. For simplicity of notation, we often consider the threshold u as another weight $w_0 = -u$ attached to the neuron with a constant input $x_0 = 1$. Positive weights correspond to *excitatory* synapses, while negative weights model *inhibitory* ones. McCulloch and Pitts proved that, in principle, suitably chosen weights let a synchronous arrangement of such neurons perform universal computations. There is a crude analogy here to a biological neuron: wires and interconnections model axons and dendrites, connection weights represent synapses, and the threshold function approximates the activity in a soma. The McCulloch and Pitts model, however, contains a number of simplifying assumptions that do not reflect the true behavior of biological neurons.

The McCulloch-Pitts neuron has been generalized in many ways. An obvious one is to use activation functions other than the threshold function, such as piecewise linear, sigmoid, or Gaussian, as shown in Figure 3. The sigmoid function is by far the most frequently used in ANNs. It is a strictly increasing function that exhibits smoothness and has the desired asymptotic properties. The standard sigmoid function is the *logistic* function, defined by

$$g(x) = 1/(1 + \exp\{-\beta x\}),$$

where β is the slope parameter.

Network architectures

ANNs can be viewed as weighted directed graphs in which artificial neurons are nodes and directed edges (with weights) are connections between neuron outputs and neuron inputs.

Based on the connection pattern (architecture), ANNs can be grouped into two categories (see Figure 4):

- *feed-forward* networks, in which graphs have no loops, and
- *recurrent* (or *feedback*) networks, in which loops occur because of feedback connections.

In the most common family of feed-forward networks, called multilayer perceptron, neurons are organized into layers that have unidirectional connections between them. Figure 4 also shows typical networks for each category.

Different connectivities yield different network behaviors. Generally speaking, feed-forward networks are *static*, that is, they produce only one set of output values rather than a sequence of values from a given input. Feed-forward networks are memory-less in the sense that their response to an input is independent of the previous network state. Recurrent, or feedback, networks, on the other hand, are dynamic systems. When a new input pattern is presented, the neuron outputs are computed. Because of the feedback paths, the inputs to each neuron are then modified, which leads the network to enter a new state.

Different network architectures require appropriate learning algorithms. The next section provides an overview of learning processes.

Learning

The ability to learn is a fundamental trait of intelligence. Although a precise definition of learning is difficult to formulate, a learning process in the ANN context can be viewed as the problem of updating network architecture and connection weights so that a network can efficiently perform a specific task. The network usually must learn the connection weights from available training patterns. Performance is improved over time by iteratively updating the weights in the network. ANNs' ability to automatically *learn from examples* makes them attractive and exciting. Instead of following a set of *rules* specified by human experts, ANNs appear to learn underlying rules (like input-output relationships) from the given collection of representative examples. This is one of the major advantages of neural networks over traditional expert systems.

To understand or design a learning process, you must first have a model of the environment in which a neural network operates, that is, you must know what information is available to the network. We refer to this model as

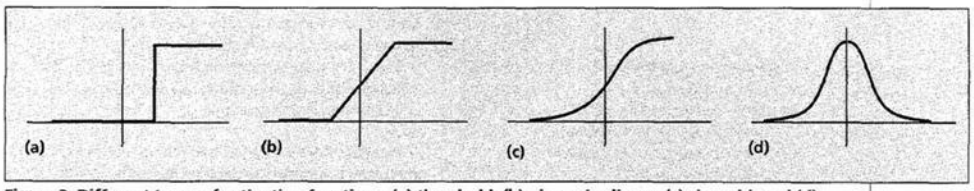


Figure 3. Different types of activation functions: (a) threshold, (b) piecewise linear, (c) sigmoid, and (d) Gaussian.

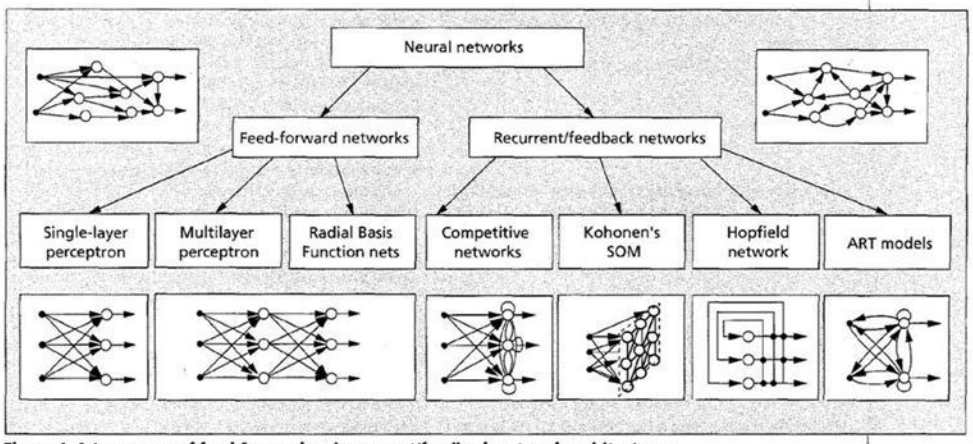


Figure 4. A taxonomy of feed-forward and recurrent/feedback network architectures.

a learning paradigm.[3] Second, you must understand how network weights are updated, that is, which *learning rules* govern the updating process. A *learning algorithm* refers to a procedure in which learning rules are used for adjusting the weights.

There are three main learning paradigms: supervised, unsupervised, and hybrid. In supervised learning, or learning with a "teacher," the network is provided with a correct answer (output) for every input pattern. Weights are determined to allow the network to produce answers as close as possible to the known correct answers. Reinforcement learning is a variant of supervised learning in which the network is provided with only a critique on the correctness of network outputs, not the correct answers themselves. In contrast, unsupervised learning, or learning without a teacher, does not require a correct answer associated with each input pattern in the training data set. It explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories from these correlations. Hybrid learning combines supervised and unsupervised learning. Part of the weights are usually determined through supervised learning, while the others are obtained through unsupervised learning.

Learning theory must address three fundamental and practical issues associated with learning from samples: capacity, sample complexity, and computational complexity. Capacity concerns how many patterns can be stored, and what functions and decision boundaries a network can form.

Sample complexity determines the number of training patterns needed to train the network to guarantee a valid generalization. Too few patterns may cause "over-fitting" (wherein the network performs well on the training data set, but poorly on independent test patterns drawn from the same distribution as the training patterns, as in Figure A3).

Computational complexity refers to the time required for a learning algorithm to estimate a solution from training patterns. Many existing learning algorithms have high computational complexity. Designing efficient algorithms for neural network learning is a very active research topic.

There are four basic types of learning rules: error-correction, Boltzmann, Hebbian, and competitive learning.

ERROR-CORRECTION RULES. In the supervised learning paradigm, the network is given a desired output for each input pattern. During the learning process, the actual output y generated by the network may not equal the desired output d. The basic principle of error-correction learning rules is to use the error signal $(d - y)$ to modify the connection weights to gradually reduce this error.

The perceptron learning rule is based on this error-correction principle. A perceptron consists of a single neuron with adjustable weights, $w_j, j = 1, 2, \ldots, n$, and threshold u, as shown in Figure 2. Given an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^t$, the net input to the neuron is

Perceptron learning algorithm

1. Initialize the weights and threshold to small random numbers.
2. Present a pattern vector $(x_1, x_2, \ldots, x_n)^t$ and evaluate the output of the neuron.
3. Update the weights according to

$$w_j(t+1) = w_j(t) + \eta\,(d - y)\,x_j,$$

where d is the desired output, t is the iteration number, and η $(0.0 < \eta < 1.0)$ is the gain (step size).

$$v = \sum_{j=1}^{n} w_j x_j - u$$

The output y of the perceptron is $+1$ if $v > 0$, and 0 otherwise. In a two-class classification problem, the perceptron assigns an input pattern to one class if $y = 1$, and to the other class if $y = 0$. The linear equation

$$\sum_{j=1}^{n} w_j x_j - u = 0$$

defines the decision boundary (a hyperplane in the n-dimensional input space) that halves the space.

Rosenblatt[5] developed a learning procedure to determine the weights and threshold in a perceptron, given a set of training patterns (see the "Perceptron learning algorithm" sidebar).

Note that learning occurs only when the perceptron makes an error. Rosenblatt proved that when training patterns are drawn from two linearly separable classes, the perceptron learning procedure converges after a finite number of iterations. This is the *perceptron convergence theorem*. In practice, you do not know whether the patterns are linearly separable. Many variations of this learning algorithm have been proposed in the literature.[2] Other activation functions that lead to different learning characteristics can also be used. However, *a single-layer perceptron can only separate linearly separable patterns as long as a monotonic activation function is used.*

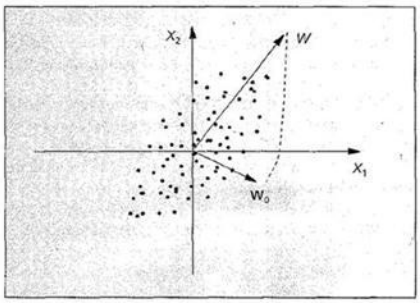


Figure 5. Orientation selectivity of a single neuron trained using the Hebbian rule.

The back-propagation learning algorithm (see the "Back-propagation algorithm sidebar") is also based on the error-correction principle.

BOLTZMANN LEARNING. Boltzmann machines are symmetric recurrent networks consisting of binary units (+1 for "on" and −1 for "off"). By symmetric, we mean that the weight on the connection from unit i to unit j is equal to the weight on the connection from unit j to unit i ($w_{ij} = w_{ji}$). A subset of the neurons, called *visible*, interact with the environment; the rest, called *hidden*, do not. Each neuron is a stochastic unit that generates an output (or state) according to the Boltzmann distribution of statistical mechanics. Boltzmann machines operate in two modes: *clamped*, in which visible neurons are clamped onto specific states determined by the environment; and *free-running*, in which both visible and hidden neurons are allowed to operate freely.

Boltzmann learning is a stochastic learning rule derived from information-theoretic and thermodynamic principles.[10] The objective of Boltzmann learning is to adjust the connection weights so that the states of visible units satisfy a particular desired probability distribution. According to the Boltzmann learning rule, the change in the connection weight w_{ij} is given by

$$\Delta w_{ij} = \eta(\bar{\rho}_{ij} - \rho_{ij}),$$

where η is the learning rate, and $\bar{\rho}_{ij}$ and ρ_{ij} are the correlations between the states of units i and j when the network operates in the clamped mode and free-running mode, respectively. The values of $\bar{\rho}_{ij}$ and ρ_{ij} are usually estimated from Monte Carlo experiments, which are extremely slow.

Boltzmann learning can be viewed as a special case of error-correction learning in which error is measured not as the direct difference between desired and actual outputs, but as the difference between the correlations among the outputs of two neurons under clamped and free-running operating conditions.

HEBBIAN RULE. The oldest learning rule is *Hebb's postulate of learning*.[13] Hebb based it on the following observation from neurobiological experiments: If neurons on both sides of a synapse are activated synchronously and repeatedly, the synapse's strength is selectively increased.

Mathematically, the Hebbian rule can be described as

$$w_{ij}(t+1) = w_{ij}(t) + \eta\, y_j(t)\, x_i(t),$$

where x_i and y_j are the output values of neurons i and j, respectively, which are connected by the synapse w_{ij}, and η is the learning rate. Note that x_i is the input to the synapse.

An important property of this rule is that learning is done locally, that is, the change in synapse weight depends only on the activities of the two neurons connected by it. This significantly simplifies the complexity of the learning circuit in a VLSI implementation.

A single neuron trained using the Hebbian rule exhibits an orientation selectivity. Figure 5 demonstrates this property. The points depicted are drawn from a two-dimen-

sional Gaussian distribution and used for training a neuron. The weight vector of the neuron is initialized to $\mathbf{w}_0$ as shown in the figure. As the learning proceeds, the weight vector moves progressively closer to the direction $\mathbf{w}$ of maximal variance in the data. In fact, $\mathbf{w}$ is the eigenvector of the covariance matrix of the data corresponding to the largest eigenvalue.

COMPETITIVE LEARNING RULES. Unlike Hebbian learning (in which multiple output units can be fired simultaneously), competitive-learning output units compete among themselves for activation. As a result, only one output unit is active at any given time. This phenomenon is known as *winner-take-all*. Competitive learning has been found to exist in biological neural networks.[3]

Competitive learning often clusters or categorizes the input data. Similar patterns are grouped by the network and represented by a single unit. This grouping is done automatically based on data correlations.

The simplest competitive learning network consists of a single layer of output units as shown in Figure 4. Each output unit i in the network connects to all the input units (x_j's) via weights, $w_{ij}, j=1, 2, \ldots, n$. Each output unit also connects to all other output units via inhibitory weights but has a self-feedback with an excitatory weight. As a result of competition, only the unit i^* with the largest (or the smallest) net input becomes the winner, that is, $\mathbf{w}_{i^*} \cdot \mathbf{x} \geq \mathbf{w}_i \cdot \mathbf{x}, \forall i$, or $\| \mathbf{w}_{i^*} - \mathbf{x} \| \leq \| \mathbf{w}_i - \mathbf{x} \|, \forall i$. When all the weight vectors are normalized, these two inequalities are equivalent.

A simple competitive learning rule can be stated as

$$\Delta w_{ij} = \begin{cases} \eta(x_j^u - w_{i^*j}), & i = i^*, \\ 0, & i \neq i^*. \end{cases} \quad (1)$$

Note that only the weights of the winner unit get updated. The effect of this learning rule is to move the stored pattern in the winner unit (weights) a little bit closer to the input pattern. Figure 6 demonstrates a geometric interpretation of competitive learning. In this example, we assume that all input vectors have been normalized to have unit length. They are depicted as black dots in Figure 6. The weight vectors of the three units are randomly initialized. Their initial and final positions on the sphere after competitive learning are marked as Xs in Figures 6a and 6b, respectively. In Figure 6, each of the three natural groups (clusters) of patterns has been discovered by an output unit whose weight vector points to the center of gravity of the discovered group.

You can see from the competitive learning rule that the network will not stop learning (updating weights) unless the learning rate η is 0. A particular input pattern can fire different output units at different iterations during learning. This brings up the stability issue of a learning system. The system is said to be *stable* if no pattern in the training data changes its category after a finite number of learning iterations. One way to achieve stability is to force the learning rate to decrease gradually as the learning process proceeds towards 0. However, this artificial freezing of learning causes another problem termed *plasticity*, which is the ability to adapt to new data. This is known as Grossberg's *stability-plasticity* dilemma in competitive learning.

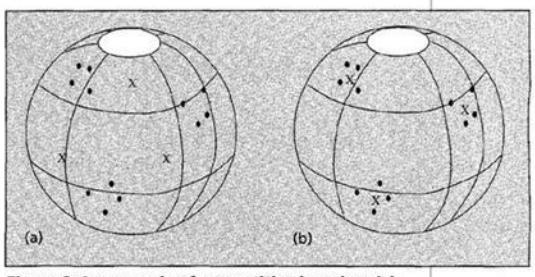


Figure 6. An example of competitive learning: (a) before learning; (b) after learning.

The most well-known example of competitive learning is *vector quantization* for data compression. It has been widely used in speech and image processing for efficient storage, transmission, and modeling. Its goal is to represent a set or distribution of input vectors with a relatively small number of prototype vectors (weight vectors), or a codebook. Once a codebook has been constructed and agreed upon by both the transmitter and the receiver, you need only transmit or store the index of the corresponding prototype to the input vector. Given an input vector, its corresponding prototype can be found by searching for the nearest prototype in the codebook.

SUMMARY. Table 2 summaries various learning algorithms and their associated network architectures (this is not an exhaustive list). Both supervised and unsupervised learning paradigms employ learning rules based

Back-propagation algorithm

1. Initialize the weights to small random values.
2. Randomly choose an input pattern $\mathbf{x}^{(u)}$.
3. Propagate the signal forward through the network.
4. Compute δ_i^L in the output layer ($o_i = y_i^L$)

   $$\delta_i^L = g'(h_i^L)\left[d_i^u - y_i^L\right],$$

   where h_i^l represents the net input to the ith unit in the lth layer, and g' is the derivative of the activation function g.
5. Compute the deltas for the preceding layers by propagating the errors backwards;

   $$\delta_i^l = g'(h_i^l)\sum_j w_{ij}^{l+1}\,\delta_j^{l+1},$$

   for $l = (L-1), \ldots, 1$.
6. Update weights using

   $$\Delta w_{ij}^l = \eta \delta_i^l y_j^{l-1}$$

7. Go to step 2 and repeat for the next pattern until the error in the output layer is below a prespecified threshold or a maximum number of iterations is reached.

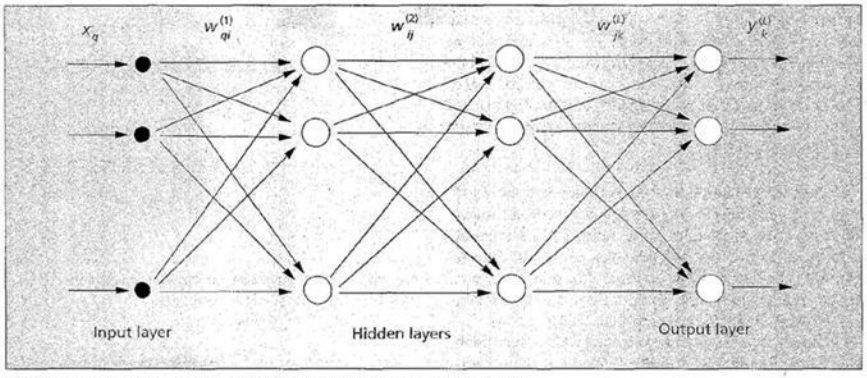


Figure 7. A typical three-layer feed-forward network architecture.

on error-correction, Hebbian, and competitive learning. Learning rules based on error-correction can be used for training feed-forward networks, while Hebbian learning rules have been used for all types of network architectures. However, each learning algorithm is designed for training a specific architecture. Therefore, when we discuss a learning algorithm, a particular network architecture association is implied. Each algorithm can

Table 2. Well-known learning algorithms.

Paradigm	Learning rule	Architecture	Learning algorithm	Task
Supervised	Error-correction	Single- or multilayer perceptron	Perceptron learning algorithms Back-propagation Adaline and Madaline	Pattern classification Function approximation Prediction, control
	Boltzmann	Recurrent	Boltzmann learning algorithm	Pattern classification
	Hebbian	Multilayer feed-forward	Linear discriminant analysis	Data analysis Pattern classification
	Competitive	Competitive	Learning vector quantization	Within-class categorization Data compression
		ART network	ARTMap	Pattern classification Within-class categorization
Unsupervised	Error-correction	Multilayer feed-forward	Sammon's projection	Data analysis
	Hebbian	Feed-forward or competitive	Principal component analysis	Data analysis Data compression
		Hopfield Network	Associative memory learning	Associative memory
	Competitive	Competitive	Vector quantization	Categorization Data compression
		Kohonen's SOM	Kohonen's SOM	Categorization Data analysis
		ART networks	ART1, ART2	Categorization
Hybrid	Error-correction and competitive	RBF network	RBF learning algorithm	Pattern classification Function approximation Prediction, control

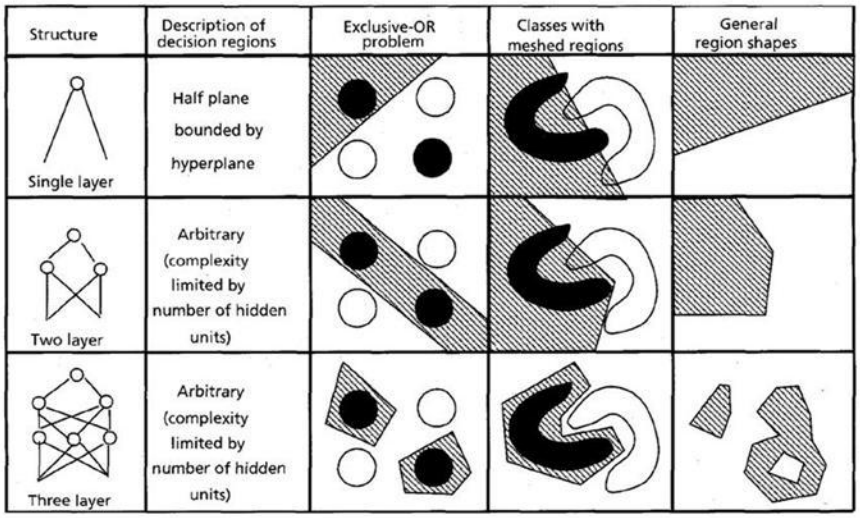


Figure 8. A geometric interpretation of the role of hidden unit in a two-dimensional input space.

perform only a few tasks well. The last column of Table 2 lists the tasks that each algorithm can perform. Due to space limitations, we do not discuss some other algorithms, including Adaline, Madaline,[14] linear discriminant analysis,[15] Sammon's projection,[15] and principal component analysis.[2] Interested readers can consult the corresponding references (this article does not always cite the first paper proposing the particular algorithms).

MULTILAYER FEED-FORWARD NETWORKS

Figure 7 shows a typical three-layer perceptron. In general, a standard L-layer feed-forward network (we adopt the convention that the input nodes are not counted as a layer) consists of an input stage, $(L-1)$ hidden layers, and an output layer of units successively connected (fully or locally) in a feed-forward fashion with no connections between units in the same layer and no feedback connections between layers.

Multilayer perceptron

The most popular class of multilayer feed-forward networks is *multilayer perceptrons* in which each computational unit employs either the thresholding function or the sigmoid function. Multilayer perceptrons can form arbitrarily complex decision boundaries and represent any Boolean function.[5] The development of the *back-propagation* learning algorithm for determining weights in a multilayer perceptron has made these networks the most popular among researchers and users of neural networks.

We denote $w_{ij}^{(l)}$ as the weight on the connection between the ith unit in layer $(l-1)$ to jth unit in layer l.

Let $\{(\mathbf{x}^{(1)}, \mathbf{d}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{d}^{(2)}), \ldots, (\mathbf{x}^{(p)}, \mathbf{d}^{(p)})\}$ be a set of p training patterns (input-output pairs), where $\mathbf{x}^{(i)} \in R^n$ is the input vector in the n-dimensional pattern space, and $\mathbf{d}^{(i)} \in [0, 1]^m$, an m-dimensional hypercube. For classification purposes, m is the number of classes. The squared-error cost function most frequently used in the ANN literature is defined as

$$E = \frac{1}{2}\sum_{i=1}^{p} \left\| \mathbf{y}^{(i)} - \mathbf{d}^{(i)} \right\|^2 \qquad (2)$$

The back-propagation algorithm[9] is a gradient-descent method to minimize the squared-error cost function in Equation 2 (see "Back-propagation algorithm" sidebar).

A geometric interpretation (adopted and modified from Lippmann[16]) shown in Figure 8 can help explicate the role of hidden units (with the threshold activation function).

Each unit in the first hidden layer forms a hyperplane in the pattern space; boundaries between pattern classes can be approximated by hyperplanes. A unit in the second hidden layer forms a hyperregion from the outputs of the first-layer units; a decision region is obtained by performing an AND operation on the hyperplanes. The output-layer units combine the decision regions made by the units in the second hidden layer by performing logical OR operations. Remember that this scenario is depicted only to explain the role of hidden units. Their actual behavior, after the network is trained, could differ.

A two-layer network can form more complex decision boundaries than those shown in Figure 8. Moreover, multilayer perceptrons with sigmoid activation functions can form smooth decision boundaries rather than piecewise linear boundaries.

Radial Basis Function network

The Radial Basis Function (RBF) network,[3] which has two layers, is a special class of multilayer feed-forward net-

works. Each unit in the hidden layer employs a radial basis function, such as a Gaussian kernel, as the activation function. The radial basis function (or kernel function) is centered at the point specified by the weight vector associated with the unit. Both the positions and the widths of these kernels must be learned from training patterns. There are usually many fewer kernels in the RBF network than there are training patterns. Each output unit implements a linear combination of these radial basis functions. From the point of view of function approximation, the hidden units provide a set of functions that constitute a basis set for representing input patterns in the space spanned by the hidden units.

There are a variety of learning algorithms for the RBF network.[3] The basic one employs a two-step learning strategy, or hybrid learning. It estimates kernel positions and kernel widths using an unsupervised clustering algorithm, followed by a supervised least mean square (LMS) algorithm to determine the connection weights between the hidden layer and the output layer. Because the output units are linear, a noniterative algorithm can be used. After this initial solution is obtained, a supervised gradient-based algorithm can be used to refine the network parameters.

This hybrid learning algorithm for training the RBF network converges much faster than the back-propagation algorithm for training multilayer perceptrons. However, for many problems, the RBF network often involves a larger number of hidden units. This implies that the runtime (after training) speed of the RBF network is often slower than the runtime speed of a multilayer perceptron. The efficiencies (error versus network size) of the RBF network and the multilayer perceptron are, however, problem-dependent. It has been shown that the RBF network has the same asymptotic approximation power as a multilayer perceptron.

SOM learning algorithm

1. Initialize weights to small random numbers; set initial learning rate and neighborhood.
2. Present a pattern $\mathbf{x}$, and evaluate the network outputs.
3. Select the unit (c_i, c_j) with the minimum output:

$$\|\mathbf{x} - \mathbf{w}_{c_i c_j}\| = \min_{ij} \|\mathbf{x} - \mathbf{w}_{ij}\|$$

4. Update all weights according to the following learning rule:

$$\mathbf{w}_{ij}(t+1) = \begin{cases} \mathbf{w}_{ij}(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{w}_{ij}(t)], & \text{if } (i,j) \in N_{c_i c_j}(t), \\ \mathbf{w}_{ij}(t), & \text{otherwise,} \end{cases}$$

where $N_{c_i c_j}(t)$ is the neighborhood of the unit (c_i, c_j) at time t, and $\alpha(t)$ is the learning rate.

5. Decrease the value of $\alpha(t)$ and shrink the neighborhood $N_{c_i c_j}(t)$.
6. Repeat steps 2 through 5 until the change in weight values is less than a prespecified threshold or a maximum number of iterations is reached.

Issues

There are many issues in designing feed-forward networks, including

- how many layers are needed for a given task,
- how many units are needed per layer,
- how will the network perform on data not included in the training set (generalization ability), and
- how large the training set should be for "good" generalization.

Although multilayer feed-forward networks using back-propagation have been widely employed for classification and function approximation,[2] many design parameters still must be determined by trial and error. Existing theoretical results provide only very loose guidelines for selecting these parameters in practice.

KOHONEN'S SELF-ORGANIZING MAPS

The self-organizing map (SOM)[16] has the desirable property of topology preservation, which captures an important aspect of the feature maps in the cortex of highly developed animal brains. In a topology-preserving mapping, nearby input patterns should activate nearby output units on the map. Figure 4 shows the basic network architecture of Kohonen's SOM. It basically consists of a two-dimensional array of units, each connected to all n input nodes. Let $\mathbf{w}_{ij}$ denote the n-dimensional vector associated with the unit at location (i, j) of the 2D array. Each neuron computes the Euclidean distance between the input vector $\mathbf{x}$ and the stored weight vector $\mathbf{w}_{ij}$.

This SOM is a special type of competitive learning network that defines a spatial neighborhood for each output unit. The shape of the local neighborhood can be square, rectangular, or circular. Initial neighborhood size is often set to one half to two thirds of the network size and shrinks over time according to a schedule (for example, an exponentially decreasing function). During competitive learning, all the weight vectors associated with the winner and its neighboring units are updated (see the "SOM learning algorithm" sidebar).

Kohonen's SOM can be used for projection of multivariate data, density approximation, and clustering. It has been successfully applied in the areas of speech recognition, image processing, robotics, and process control.[2] The design parameters include the dimensionality of the neuron array, the number of neurons in each dimension, the shape of the neighborhood, the shrinking schedule of the neighborhood, and the learning rate.

ADAPTIVE RESONANCE THEORY MODELS

Recall that the *stability-plasticity* dilemma is an important issue in competitive learning. How do we learn new things (plasticity) and yet retain the stability to ensure that existing knowledge is not erased or corrupted? Carpenter and Grossberg's Adaptive Resonance Theory models (ART1, ART2, and ARTMap) were developed in an attempt to overcome this dilemma.[17] The network has a sufficient supply of output units, but they are not used until deemed necessary. A unit is said to be *committed* (*uncommitted*) if it is (is not) being used. The learning algorithm updates

the stored prototypes of a category only if the input vector is sufficiently similar to them. An input vector and a stored prototype are said to resonate when they are sufficiently similar. The extent of similarity is controlled by a *vigilance parameter*, ρ, with $0 < \rho < 1$, which also determines the number of categories. When the input vector is not sufficiently similar to any existing prototype in the network, a new category is created, and an uncommitted unit is assigned to it with the input vector as the initial prototype. If no such uncommitted unit exists, a novel input generates no response.

We present only ART1, which takes binary (0/1) input to illustrate the model. Figure 9 shows a simplified diagram of the ART1 architecture.[2] It consists of two layers of fully connected units. A top-down weight vector $\mathbf{w}_j$ is associated with unit j in the input layer, and a bottom-up weight vector $\overline{\mathbf{w}}_i$ is associated with output unit i; $\overline{\mathbf{w}}_i$ is the normalized version of $\mathbf{w}_i$.

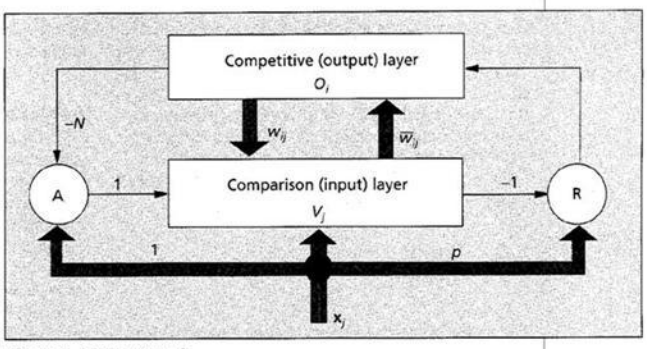


Figure 9. ART1 network.

$$\overline{\mathbf{w}}_i = \frac{\mathbf{w}_i}{\varepsilon + \sum_j w_{ji}}, \tag{3}$$

where ε is a small number used to break the ties in selecting the winner. The top-down weight vectors $\mathbf{w}_j$'s store cluster prototypes. The role of normalization is to prevent prototypes with a long vector length from dominating prototypes with a short one. Given an n-bit input vector $\mathbf{x}$, the output of the auxiliary unit A is given by

$$A = Sgn_{0/1}\left(\sum_j x_j - n\sum_i O_i - 0.5\right),$$

where $Sgn_{0/1}(x)$ is the *signum* function that produces +1 if $x \geq 0$ and 0 otherwise, and the output of an input unit is given by

$$V_j = Sgn_{0/1}\left(x_j + \sum_i w_{ji}O_i + A - 1.5\right) = \begin{cases} x_j, & \text{if no output } O_j \text{ is "on",} \\ x_j \wedge \Sigma_i w_{ji}O_i, & \text{otherwise.} \end{cases}$$

A reset signal R is generated only when the similarity is less than the vigilance level. (See the "ART1 learning algorithm" sidebar.)

The ART1 model can create new categories and reject an input pattern when the network reaches its capacity. However, the number of categories discovered in the input data by ART1 is sensitive to the vigilance parameter.

HOPFIELD NETWORK

Hopfield used a network *energy* function as a tool for designing recurrent networks and for understanding their dynamic behavior.[7] Hopfield's formulation made explicit the principle of storing information as dynamically stable attractors and popularized the use of recurrent networks for associative memory and for solving combinatorial optimization problems.

A Hopfield network with n units has two versions: binary and continuously valued. Let v_i be the state or output of the ith unit. For binary networks, v_i is either +1 or −1, but for continuous networks, v_i can be any value between 0 and 1. Let w_{ij} be the synapse weight on the connection from units i to j. In Hopfield networks, $w_{ij} = w_{ji}$, $\forall i, j$ (symmetric networks), and $w_{ii} = 0$, $\forall i$ (no self-feedback connections). The network dynamics for the binary Hopfield network are

$$v_i = Sgn\left(\sum_j w_{ij}v_j - \theta_i\right) \tag{4}$$

ART1 learning algorithm

1. Initialize $w_{ij} = 1$, for all i, j. Enable all the output units.
2. Present a new pattern $\mathbf{x}$.
3. Find the winner unit i^* among the enabled output units

   $\overline{\mathbf{w}}_{i^*} \cdot \mathbf{x} \geq \overline{\mathbf{w}}_i \cdot \mathbf{x}, \forall i$

4. Perform vigilance test

   $r = \dfrac{\mathbf{w}_{i^*} \cdot \mathbf{x}}{\Sigma_j x_j}$

   If $r \geq \rho$ (resonance), go to step 5. Otherwise, disable unit i^* and go to step 3 (until all the output units are disabled).
5. Update the winning weight vector $\mathbf{w}_{i^*}$, enable all the output units, and go to step 2

   $\Delta w_{ji^*} = \eta\,(V_j - w_{ji^*})$

6. If all output units are disabled, select one of the uncommitted output units and set its weight vector to x. If there is no uncommitted output unit (capacity is reached), the network rejects the input pattern.

The dynamic update of network states in Equation 4 can be carried out in at least two ways: *synchronously* and *asynchronously*. In a synchronous updating scheme, all units are updated simultaneously at each time step. A central clock must synchronize the process. An asynchronous updating scheme selects one unit at a time and updates its state. The unit for updating can be randomly chosen.

The energy function of the binary Hopfield network in a state $\mathbf{v} = (v_1, v_2, \ldots, v_n)^T$ is given by

$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} v_i v_j \qquad (5)$$

The central property of the energy function is that as network state evolves according to the network dynamics (Equation 4), the network energy always decreases and eventually reaches a local minimum point (attractor) where the network stays with a constant energy.

Associative memory

When a set of patterns is stored in these network attractors, it can be used as an *associative memory*. Any pattern present in the basin of attraction of a stored pattern can be used as an index to retrieve it.

An associative memory usually operates in two phases: storage and retrieval. In the storage phase, the weights in the network are determined so that the attractors of the network memorize a set of p n-dimensional patterns $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^p\}$ to be stored. A generalization of the Hebbian learning rule can be used for setting connection weights w_{ij}. In the retrieval phase, the input pattern is used as the initial state of the network, and the network evolves according to its dynamics. A pattern is produced (or retrieved) when the network reaches equilibrium.

How many patterns can be stored in a network with n binary units? In other words, what is the memory capacity of a network? It is finite because a network with n binary units has a maximum of 2^n distinct states, and not all of them are attractors. Moreover, not all attractors (stable states) can store useful patterns. Spurious attractors can also store patterns different from those in the training set.[2]

It has been shown that the maximum number of random patterns that a Hopfield network can store is $P_{max} \approx 0.15n$. When the number of stored patterns $p < 0.15n$, a nearly perfect recall can be achieved. When memory patterns are orthogonal vectors instead of random patterns, more patterns can be stored. But the number of spurious attractors increases as p reaches capacity. Several learning rules have been proposed for increasing the memory capacity of Hopfield networks.[2] Note that we require n^2 connections in the network to store p n-bit patterns.

Energy minimization

Hopfield networks always evolve in the direction that leads to lower network energy. This implies that if a combinatorial optimization problem can be formulated as minimizing this energy, the Hopfield network can be used to find the optimal (or suboptimal) solution by letting the network evolve freely. In fact, any quadratic objective function can be rewritten in the form of Hopfield network energy. For example, the classic Traveling Salesman Problem can be formulated as such a problem.

APPLICATIONS

We have discussed a number of important ANN models and learning algorithms proposed in the literature. They have been widely used for solving the seven classes of problems described in the beginning of this article. Table 2 showed typical suitable tasks for ANN models and learning algorithms. Remember that to successfully work with real-world problems, you must deal with numerous design issues, including network model, network size, activation function, learning parameters, and number of training samples. We next discuss an optical character recognition (OCR) application to illustrate how multilayer feed-forward networks are successfully used in practice.

OCR deals with the problem of processing a scanned image of text and transcribing it into machine-readable form. We outline the basic components of OCR and explain how ANNs are used for character classification.

An OCR system

An OCR system usually consists of modules for preprocessing, segmentation, feature extraction, classification, and contextual processing. A paper document is scanned to produce a gray-level or binary (black-and-white) image. In the preprocessing stage, filtering is applied to remove noise, and text areas are located and converted to a binary image using a global or local adaptive thresholding method. In the segmentation step, the text image is separated into individual characters. This is a particularly difficult task with handwritten text, which contains a proliferation of touching characters. One effective technique is to break the composite pattern into smaller patterns (over-segmentation) and find the correct character segmentation points using the output of a pattern classifier.

Because of various degrees of slant, skew, and noise level, and various writing styles, recognizing segmented characters is not easy. This is evident from Figure 10, which shows the size-normalized character bitmaps of a sample set from the NIST (National Institute of Standards and Technology) hand-print character database.[16]

Schemes

Figure 11 shows the two main schemes for using ANNs in an OCR system. The first one employs an explicit feature extractor (not necessarily a neural network). For instance, contour direction features are used in Figure 11. The extracted features are passed to the input stage of a multilayer feed-forward network.[39] This scheme is very flexible in incorporating a large variety of features. The other scheme does not explicitly extract features from the raw data. The feature extraction implicitly takes place within the intermediate stages (hidden layers) of the ANN. A nice property of this scheme is that feature extraction and classification are integrated and trained simultaneously to produce optimal classification results. It is not clear whether the types of features that can be extracted by this integrated architecture are the most effective for character recognition. Moreover, this scheme requires a much larger network than the first one.

A typical example of this integrated feature extraction-classification scheme is the network developed by Le Cun et al.[20] for zip code recognition. A 16 × 16 normalized gray-level image is presented to a feed-forward network with three hidden layers. The units in the first layer are locally connected to the units in the input layer, forming a set of local feature maps. The second hidden layer is constructed in a similar way. Each unit in the second layer also combines local information coming from feature maps in the first layer.

The activation level of an output unit can be interpreted as an approximation of the a posteriori probability of the input pattern's belonging to a particular class. The output categories are ordered according to activation levels and passed to the post-processing stage. In this stage, contextual information is exploited to update the classifier's output. This could, for example, involve looking up a dictionary of admissible words, or utilizing syntactic constraints present, for example, in phone or social security numbers.

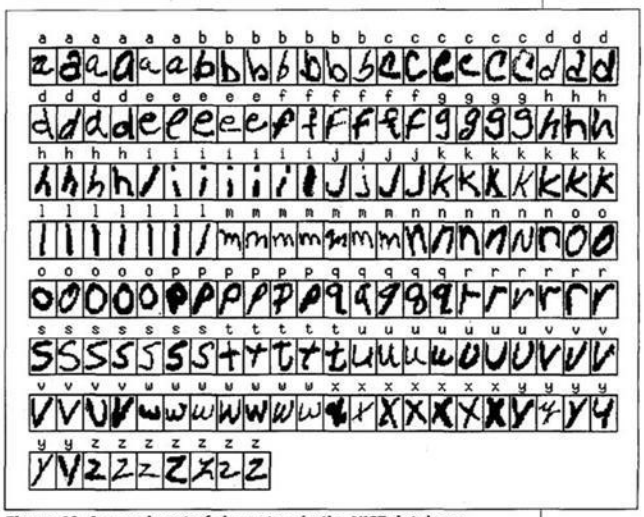


Figure 10. A sample set of characters in the NIST database.

Results

ANNs work very well in the OCR application. However, there is no conclusive evidence about their superiority over conventional statistical pattern classifiers. At the First Census Optical Character Recognition System Conference held in 1992,[18] more than 40 different handwritten character recognition systems were evaluated based on their performance on a common database. The top 10 performers used either some type of multilayer feed-forward network or a nearest neighbor-based classifier. ANNs tend to be superior in terms of speed and memory requirements compared to nearest neighbor methods. Unlike the nearest neighbor methods, classification speed using ANNs is independent of the size of the training set. The recognition accuracies of the top OCR systems on the NIST isolated (presegmented) character data were above 98 percent for digits, 96 percent for uppercase characters, and 87 percent for lowercase characters. (Low recognition accuracy for lowercase characters was largely due to the fact that the test data differed significantly from the training data, as well as being due to "ground-truth" errors.) One conclusion drawn from the test is that OCR system performance on isolated characters compares well with human performance. However, humans still outperform OCR systems on unconstrained and cursive handwritten documents.

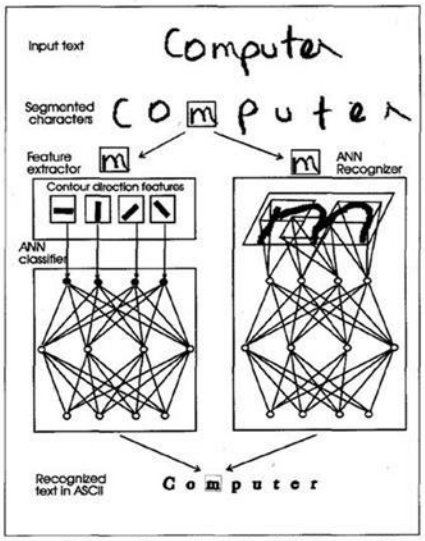


Figure 11. Two schemes for using ANNs in an OCR system.

DEVELOPMENTS IN ANNS HAVE STIMULATED a lot of enthusiasm and criticism. Some comparative studies are optimistic, some offer pessimism. For many tasks, such as pattern recognition, no one approach dominates the others. The choice of the best technique should be driven by the given application's nature. We should try to understand the capacities, assumptions, and applicability of various approaches and maximally exploit their complementary advantages to develop better intelligent systems. Such an effort may lead to a synergistic approach that combines the strengths of ANNs with other technologies to achieve significantly better performance for challenging problems. As Minsky[21] recently observed, the time has come to build systems out of diverse components. Individual modules are important, but we also need a good methodology for integration. It is clear that communication and cooperative work between

researchers working in ANNs and other disciplines will not only avoid repetitious work but (and more important) will stimulate and benefit individual disciplines. ■

Acknowledgments

We thank Richard Casey (IBM Almaden); Pat Flynn (Washington State University); William Punch, Chitra Dorai, and Kalle Karu (Michigan State University); Ali Khotanzad (Southern Methodist University); and Ishwar Sethi (Wayne State University) for their many useful suggestions.

References

1. *DARPA Neural Network Study*, AFCEA Int'l Press, Fairfax, Va., 1988.
2. J. Hertz, A. Krogh, and R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Reading, Mass., 1991.
3. S. Haykin, *Neural Networks: A Comprehensive Foundation*, MacMillan College Publishing Co., New York, 1994.
4. W.S. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bull. Mathematical Biophysics*, Vol. 5, 1943, pp. 115-133.
5. R. Rosenblatt, *Principles of Neurodynamics*, Spartan Books, New York, 1962.
6. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, Mass., 1969.
7. J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," in *Proc. Nat'l Academy of Sciences*, USA 79, 1982, pp. 2,554-2,558.
8. P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," PhD thesis, Dept. of Applied Mathematics, Harvard University, Cambridge, Mass., 1974.
9. D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, MIT Press, Cambridge, Mass., 1986.
10. J.A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, Mass., 1988.
11. S. Brunak and B. Lautrup, *Neural Networks, Computers with Intuition*, World Scientific, Singapore, 1990.
12. J. Feldman, M.A. Fanty, and N.H. Goddard, "Computing with Structured Neural Networks," *Computer*, Vol. 21, No. 3, Mar. 1988, pp. 91-103.
13. D.O. Hebb, *The Organization of Behavior*, John Wiley & Sons, New York, 1949.
14. R.P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, Vol. 4, No. 2, Apr. 1987, pp. 4-22.
15. A.K. Jain and J. Mao, "Neural Networks and Pattern Recognition," in *Computational Intelligence: Imitating Life*, J.M. Zurada, R. J. Marks II, and C.J. Robinson, eds., IEEE Press, Piscataway, N.J., 1994, pp. 194-212.
16. T. Kohonen, *Self Organization and Associative Memory*, Third Edition, Springer-Verlag, New York, 1989.
17. G.A. Carpenter and S. Grossberg, *Pattern Recognition by Self-Organizing Neural Networks*, MIT Press, Cambridge, Mass., 1991.
18. "The First Census Optical Character Recognition System Conference," R.A. Wilkinson et al., eds., . Tech. Report, NISTIR 4912, US Dept. Commerce, NIST, Gaithersburg, Md., 1992.
19. K. Mohiuddin and J. Mao, "A Comparative Study of Different Classifiers for Handprinted Character Recognition," in *Pattern Recognition in Practice IV*, E.S. Gelsema and L.N. Kanal, eds., Elsevier Science, The Netherlands, 1994, pp. 437-448.
20. Y. Le Cun et al., "Back-Propagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, Vol. 1, 1989, pp. 541–551.
21. M. Minsky, "Logical Versus Analogical or Symbolic Versus Connectionist or Neat Versus Scruffy," *AI Magazine*, Vol. 65, No. 2, 1991, pp. 34-51.

Anil K. Jain *is a University Distinguished Professor and the chair of the Department of Computer Science at Michigan State University. His interests include statistical pattern recognition, exploratory pattern analysis, neural networks, Markov random fields, texture analysis, remote sensing, interpretation of range images, and 3D object recognition.*

Jain served as editor-in-chief of IEEE Transactions on Pattern Analysis and Machine Intelligence *from 1991 to 1994, and currently serves on the editorial boards of* Pattern Recognition, Pattern Recognition Letters, Journal of Mathematical Imaging, Journal of Applied Intelligence, *and* IEEE Transactions on Neural Networks. *He has coauthored, edited, and coedited numerous books in the field. Jain is a fellow of the IEEE and a speaker in the IEEE Computer Society's Distinguished Visitors Program for the Asia-Pacific region. He is a member of the IEEE Computer Society.*

Jianchang Mao *is a research staff member at the IBM Almaden Research Center. His interests include pattern recognition, neural networks, document image analysis, image processing, computer vision, and parallel computing.*

Mao received the BS degree in physics in 1983 and the MS degree in electrical engineering in 1986 from East China Normal University in Shanghai. He received the PhD in computer science from Michigan State University in 1994. Mao is the abstracts editor of IEEE Transactions on Neural Networks. *He is a member of the IEEE and the IEEE Computer Society.*

K.M. Mohiuddin *is the manager of the Document Image Analysis and Recognition project in the Computer Science Department at the IBM Almaden Research Center. He has led IBM projects on high-speed reconfigurable machines for industrial machine vision, parallel processing for scientific computing, and document imaging systems. His interests include document image analysis, handwriting recognition/OCR, data compression, and computer architecture.*

Mohiuddin received the MS and PhD degrees in electrical engineering from Stanford University in 1977 and 1982, respectively. He is an associate editor of IEEE Transactions on Pattern Analysis and Machine Intelligence. *He served on Computer's editorial board from 1984 to 1989, and is a senior member of the IEEE and a member of the IEEE Computer Society.*

Readers can contact Anil Jain at the Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, MI 48824; jain@cps.msu.edu.

AI : Neural Network for beginners (Part 2 of 3)

Sacha Barber, 29 Jan 2007

AI : An Introduction into Neural Networks (Multi-layer networks / Back Propagation)

Download demo project (includes source code) - 812 Kb

Introduction

This article is part 2 of a series of 3 articles that I am going to post. The proposed article content will be as follows:

1. Part 1 : Is an introduction into Perceptron networks (single layer neural networks).
2. Part 2 : This one, is about multi layer neural networks, and the back propagation training method to solve a non linear classification problem such as the logic of an XOR logic gate. This is something that a Perceptron can't do. This is explained further within this article.
3. Part 3 : Will be about how to use a genetic algorithm (GA) to train a multi layer neural network to solve some logic problem.

Summary

This article will show how to use a multi-layer neural network to solve the XOR logic problem.

A Brief Recap (From part 1 of 3)

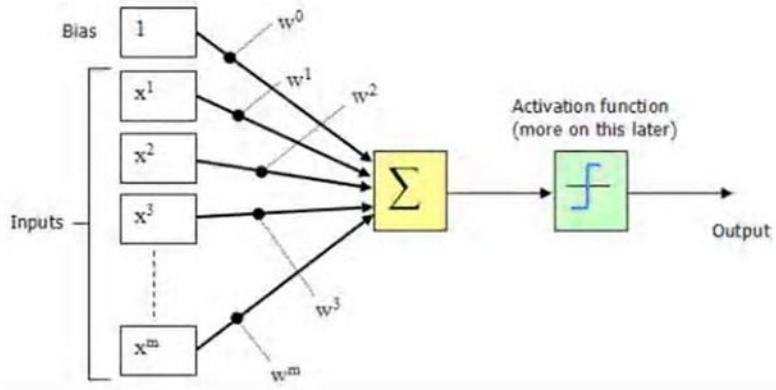
Before we commence with the nitty gritty of this new article which deals with multi layer Neural Networks, let just revisit a few key concepts. If you haven't read Part 1, perhaps you should start there.

Perceptron Configuration (Single layer network)

The inputs and connection weights shown below are typically real values, both positive (+) and negative (-).

The perceptron itself, consists of weights, the summation processor, an activation function, and an adjustable threshold processor (called bias here after).

For convenience, the normal practice is to treat the bias as just another input. The following diagram illustrates the revised configuration.



The bias can be thought of as the propensity (a tendency towards a particular way of behaving) of the perceptron to fire irrespective of it's inputs. The perceptron configuration network shown above fires if the weighted sum > 0, or if you have into maths type explanations

$$\sum_{i=1}^{m} bias + (w^i x^i)$$

So that's the basic operation of a perceptron. But we now want to build more layers of these, so let's carry on to the new stuff.

So Now The New Stuff (More layers)

From this point on, anything that is being discussed relates directly to this article's code.

In the summary at the top, the problem we are trying to solve was how to use a multi-layer neural network to solve the XOR logic problem. So how is this done. Well it's really an incremental build on what Part 1 already discussed. So let's march on.

What does the XOR logic problem look like? Well, it looks like the following truth table:

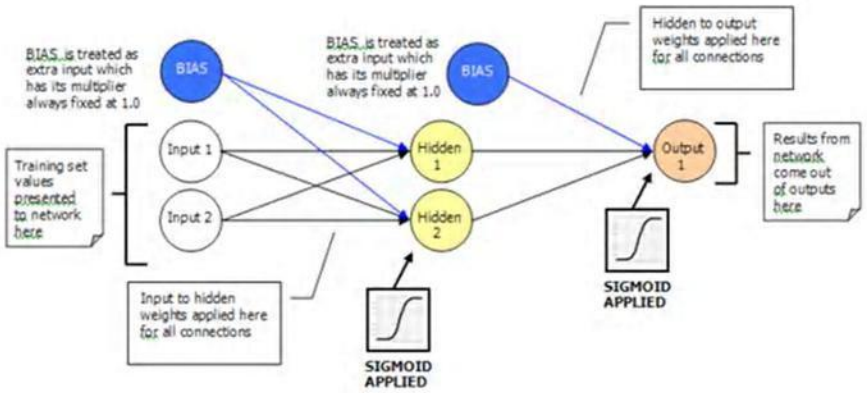
I1	I2	Output
0	0	0
0	1	1
1	0	1
1	1	0

XOR Logic Gate

Remember with a single layer (perceptron) we can't actually achieve the XOR functionality, as it is not linearly separable. But with a multi-layer network, this is achievable.

What Does The New Network Look Like

The new network that will solve the XOR problem will look similar to a single layer network. We are still dealing with inputs / weights / outputs. What is new is the addition of the hidden layer.



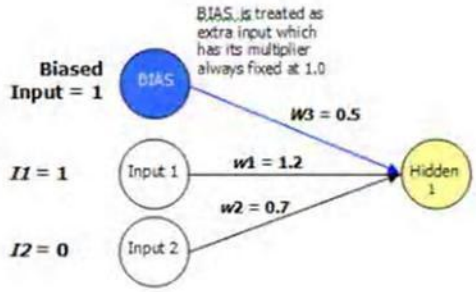
As already explained above, there is one input layer, one hidden layer and one output layer.

It is by using the inputs and weights that we are able to work out the activation for a given node. This is easily achieved for the hidden layer as it has direct links to the actual input layer.

The output layer, however, knows nothing about the input layer as it is not directly connected to it. So to work out the activation for an output node we need to make use of the output from the hidden layer nodes, which are used as inputs to the output layer nodes.

This entire process described above can be thought of as a pass forward from one layer to the next.

This still works like it did with a single layer network; the activation for any given node is still worked out as follows:



$$A = \sum_{n=1}^{N+1} wi * Ii$$

NOTE : N + 1 is bias and the value of the input for the bias is 1.

Where (wi is the weight(i), and Ii is the input(i) value)

You see it the same old stuff, no demons, smoke or magic here. It's stuff we've already covered.

So that's how the network looks/works. So now I guess you want to know how to go about training it.

Types Of Learning

There are essentially 2 types of learning that may be applied, to a Neural Network, which is "Reinforcement" and "Supervised"

Reinforcement

In Reinforcement learning, during training, a set of inputs is presented to the Neural Network, the Output is 0.75, when the target was expecting 1.0.

The error (1.0 - 0.75) is used for training ('wrong by 0.25').

What if there are 2 outputs, then the total error is summed to give a single number (typically sum of squared errors). Eg "your total error on all outputs is 1.76"

Note that this just tells you how wrong you were, not in which direction you were wrong.

Using this method we may never get a result, or it could be a case of 'Hunt the needle'.

NOTE : Part 3 of this series will be using a GA to train a Neural Network, which is Reinforcement learning. The GA simply does what a GA does, and all the normal GA phases to select weights for the Neural Network. There is no back propagation of values. The Neural Network is just good or just bad. As one can imagine, this process takes a lot more steps to get to the same result.

Supervised

In Supervised Learning the Neural Network is given more information.
Not just 'how wrong' it was, but 'in what direction it was wrong' like 'Hunt the needle' but where you are told 'North a bit', 'West a bit'.

So you get, and use, far more information in Supervised Learning, and this is the normal form of Neural Network learning algorithm. Back Propagation (what this article uses, is Supervised Learning)

Learning Algorithm

In brief, to train a multi-layer Neural Network, the following steps are carried out:

- Start off with random weights (and biases) in the Neural Network
- Try one or more members of the training set, see how badly the output(s) are compared to what they should be (compared to the target output(s))
- Jiggle weights a bit, aimed at getting improvement on outputs
- Now try with a new lot of the training set, or repeat again, jiggling weights each time
- Keep repeating until you get quite accurate outputs

This is what this article submission uses to solve the XOR problem. This is also called "Back Propagation" (normally called BP or BackProp)

Backprop allows you to use this error at output, to adjust the weights arriving at the output layer, but then also allows you to calculate the effective error 1 layer back, and use this to adjust the weights arriving there, and so on, back-propagating errors through any number of layers.

The trick is the use of a sigmoid as the non-linear transfer function (which was covered in Part 1. The sigmoid is used as it offers the ability to apply differentiation techniques.

$$y = g(x) = \frac{1}{1 + e^{-x}}$$

Because this is nicely differentiable —it so happens that

$$\frac{dg}{dx} = g'(x) = g(x)(1 - g(x))$$

Which in context of the article can be written as

```
delta_outputs[i] = outputs[i] * (1.0 - outputs[i]) * (targets[i] - outputs[i])
```

It is by using this calculation that the weight changes can be applied back through the network.

Things To Watch Out For

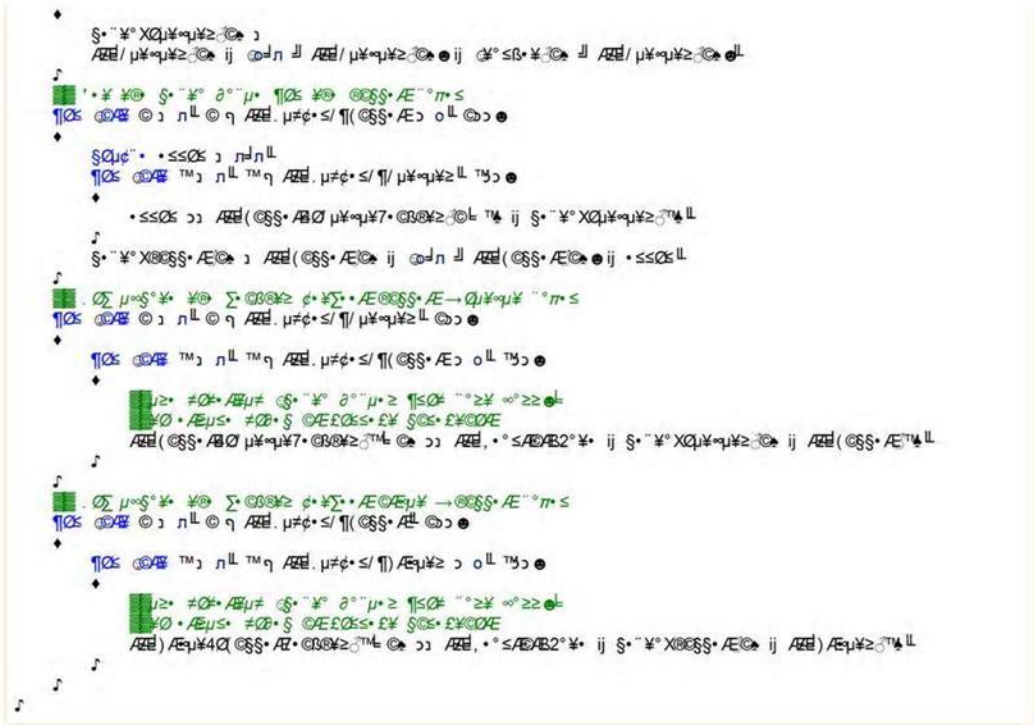
Valleys: Using the rolled ball metaphor, there may well be valleys like this, with steep sides and a gently sloping floor. Gradient descent tends to waste time swooshing up and down each side of the valley (think ball!)

So what can we do about this. Well we add a momentum term, that tends to cancel out the back and forth movements and emphasizes any consistent direction, then this will go down such valleys with gentle bottom-slopes much more successfully (faster)

Starting The Training

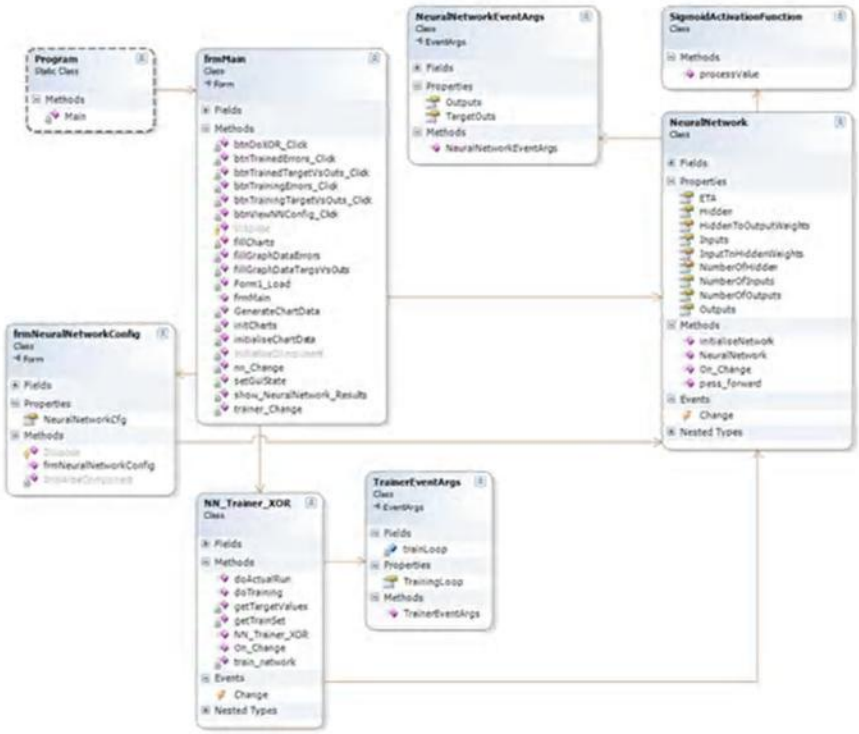
This is probably best demonstrated with a code snippet from the article's actual code:

Hide ¡ Shrink ¡ Copy Code



So Finally The Code

Well, the code for this article looks like the following class diagram (It's Visual Studio 2005 C#, .NET v2.0)



The main classes that people should take the time to look at would be :

- . . X4≤° ©Æ≤X8/ 2 : Trains a Neural Network to solve the XOR problem
- 4≤° ©Æ≤%∂• Æ! ≤ß≥ : Training event args, for use with a GUI
- . • µ≤° ¨ . • ¥∑Ø≤´ : A configurable Neural Network
- . • µ≤° ¨ . • ¥∑Ø≤´ %∂• Æ! ≤ß≥ : Training event args, for use with a GUI
- 3©ß≠Ø©§! £¥©∂ ° ¥©Ø∆Æ¥©Ø∆E A static method to provide the sigmoid activation function

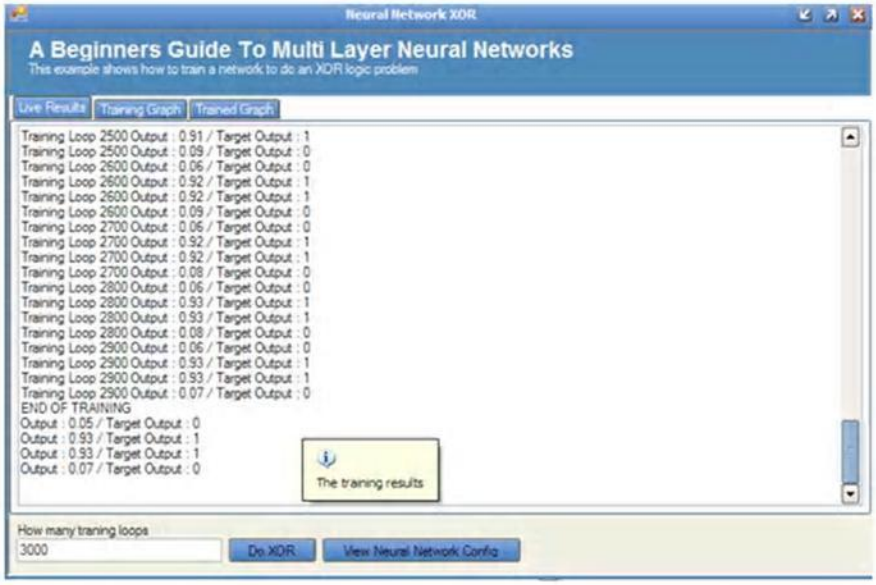
The rest are a GUI I constructed simply to show how it all fits together.

NOTE : the demo project contains all code, so I won't list it here.

Code Demos

The DEMO application attached has 3 main areas which are described below:

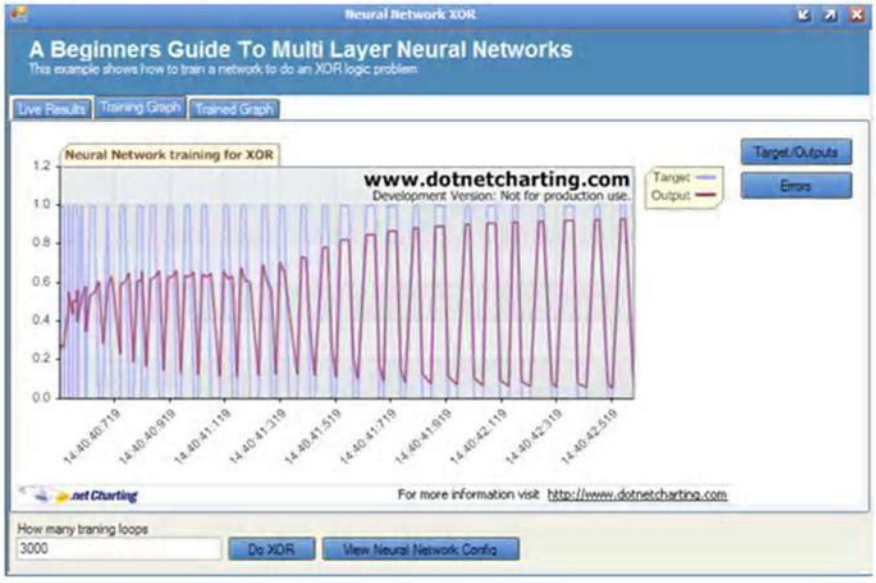
LIVE RESULTS Tab



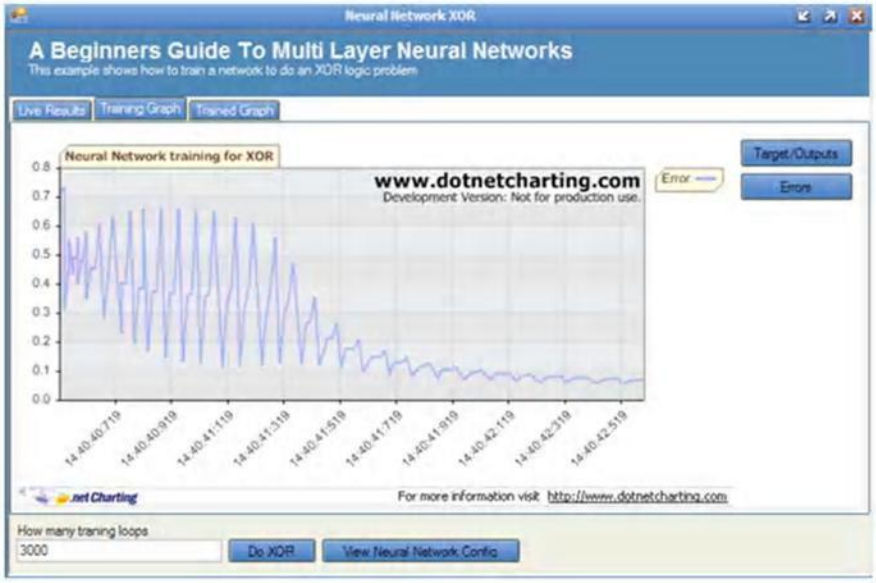
It can be seen that this has very nearly solved the XOR problem (You will probably never get it 100% accurate)

TRAINING RESULTS Tab

Viewing the training phase target/outputs together

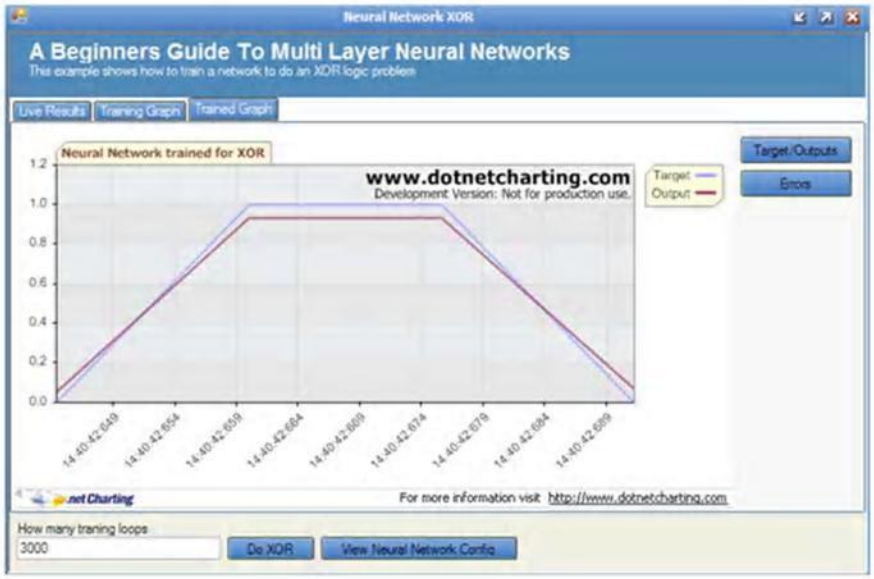


Viewing the training phase errors

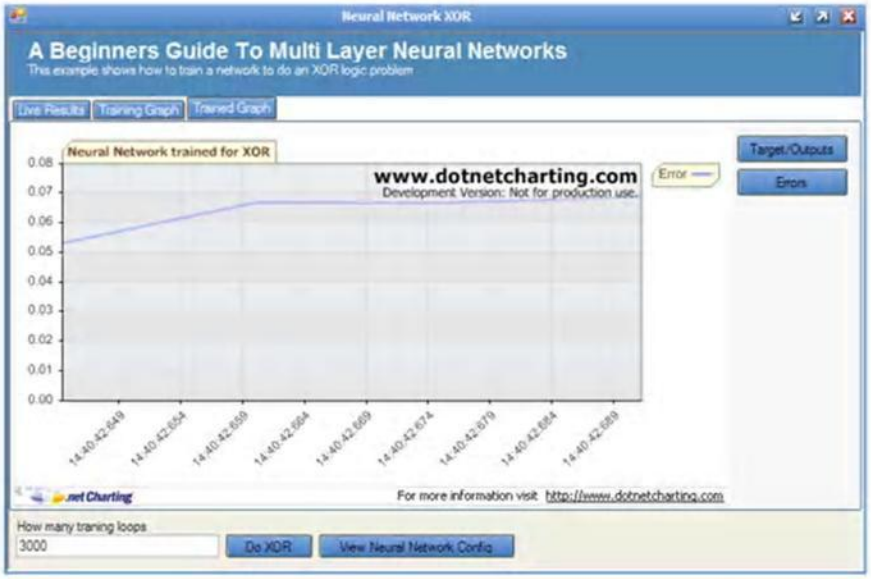


TRAINED RESULTS Tab

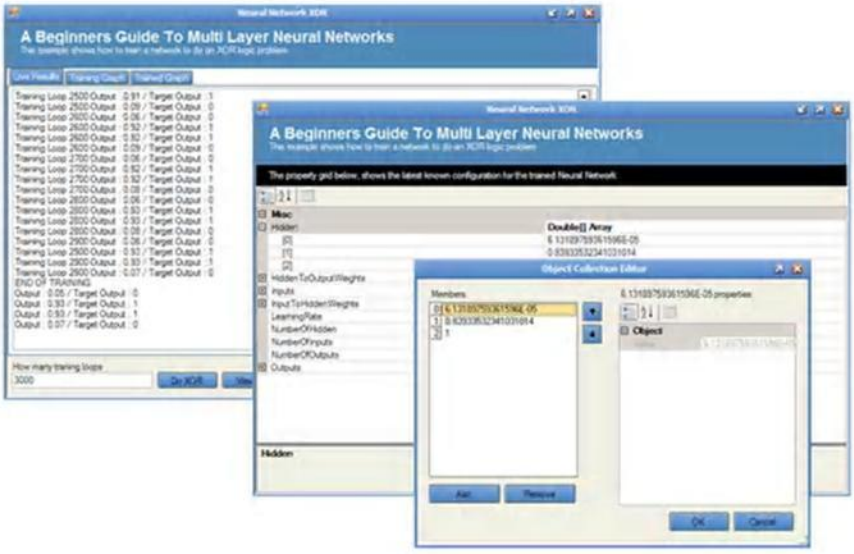
Viewing the trained target/outputs together



Viewing the trained errors



It is also possible to view the Neural Networks final configuration using the "View Neural Network Config" button. If people are interested in what weights the Neural Network ended up with, this is the place to look.



What Do You Think ?

That's it. I would just like to ask, if you liked the article, please vote for it.

Points of Interest

I think AI is fairly interesting, that's why I am taking the time to publish these articles. So I hope someone else finds it interesting, and that it might help further someone's knowledge, as it has my own.

Anyone that wants to look further into AI type stuff, that finds the content of this article a bit basic should check out Andrew Krillovs articles, at Andrew Krillov CP articles as his are more advanced, and very good. In fact anything Andrew seems to do, is very good.

History

- v1.0 24/11/06

Bibliography

- Artificial Intelligence 2nd edition, Elaine Rich / Kevin Knight. McGraw Hill Inc.
- Artificial Intelligence, A Modern Approach, Stuart Russell / Peter Norvig. Prentice Hall.

License

Share

About the Author

Sacha Barber

Software Developer (Senior)
United Kingdom

I currently hold the following qualifications (amongst others, I also studied Music Technology and Electronics, for my sins)

- MSc (Passed with distinctions), in Information Technology for E-Commerce
- BSc Hons (1st class) in Computer Science & Artificial Intelligence

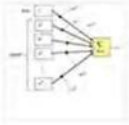
Both of these at Sussex University UK.

Award(s)

I am lucky enough to have won a few awards for Zany Crazy code articles over the years

- Microsoft C# MVP 2015
- Codeproject MVP 2015
- Microsoft C# MVP 2014
- Codeproject MVP 2014
- Microsoft C# MVP 2013
- Codeproject MVP 2013
- Microsoft C# MVP 2012
- Codeproject MVP 2012
- Microsoft C# MVP 2011
- Codeproject MVP 2011
- Microsoft C# MVP 2010
- Codeproject MVP 2010
- Microsoft C# MVP 2009
- Codeproject MVP 2009
- Microsoft C# MVP 2008
- Codeproject MVP 2008
- And numerous codeproject awards which you can see over at my blog

You may also be interested in...



AI: Neural Network for Beginners (Part 3 of 3)

Speed Up Your Git Repository: Introducing Git-Over-FASP

AI : Neural Network for beginners (Part 1 of 3)

Taking COBOL mobile

AI Life

COBOL programmers: Skill up and save timeÂ

Comments and Discussions

You must Sign In to use this message board.

Search Comments

Profile popups Spacing Relaxed Layout Normal Per page 25 Update

First Prev Next

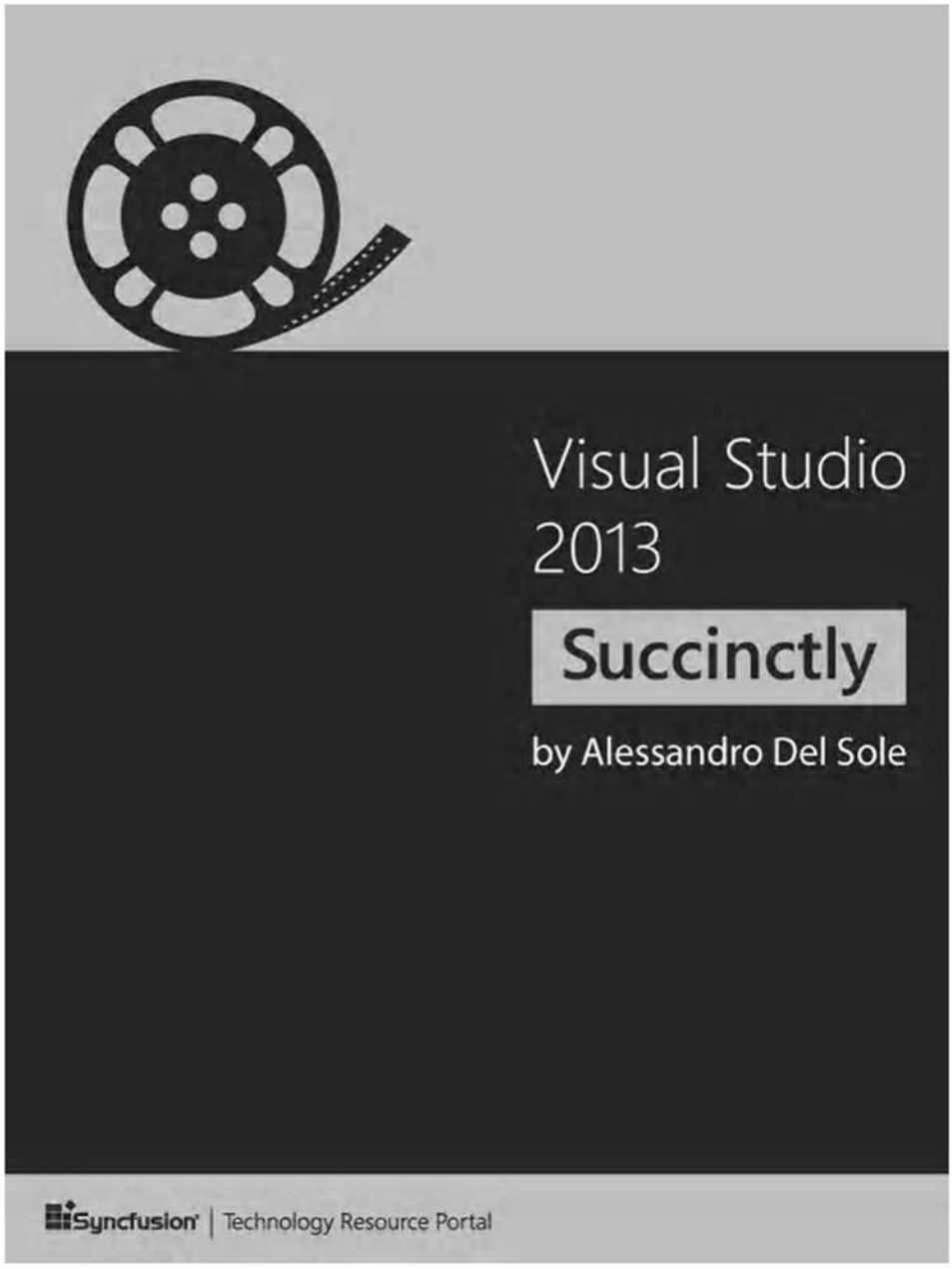
help	Member 11265993	26-Nov-14 22:42
Sigmoid function	Member 10973839	26-Jul-14 23:17
Sigmoid function	Member 10973839	26-Jul-14 23:16
Takes a long time to converge	Member 9401821	2-Mar-14 0:13
Re: Takes a long time to converge	LudemeGames	2-Mar-14 4:38
I have a question,thank you for telling me .	fengyelan	16-Apr-13 22:31
My vote of 5	Nickydo	10-Sep-12 2:30
Part 3?	Mauro Leggieri	5-Apr-09 7:41
Re: Part 3?	Sacha Barber	5-Apr-09 9:55
Re: Part 3?	Mauro Leggieri	6-Apr-09 3:17
About parameterizing the 'momentum' factor	mahabir	23-Sep-08 20:48
Re: About parameterizing the 'momentum' factor	Sacha Barber	23-Sep-08 22:57
Re: About parameterizing the 'momentum' factor	Sacha Barber	23-Sep-08 22:59
Re: About parameterizing the 'momentum' factor	ramesh0285	26-Nov-12 18:14
part 1	gholamabbas Sayyad	18-Sep-08 21:21
Re: part 1	Sacha Barber	18-Sep-08 22:49
Solution for getTrainSet(int idx)	DKHVC	16-Apr-08 21:09
[Message Deleted]	Danny Rodriguez	27-Jan-08 10:05
Hello	MohamadJaber	11-Dec-07 0:33
Erratic Bahaviour?	rampantandroid	15-Oct-07 18:17
Re: Erratic Bahaviour?	rampantandroid	15-Oct-07 19:56
Small Suggestion	dfhgesart	28-Jul-07 16:26
Re: Small Suggestion	Sacha Barber	29-Jul-07 0:35
Excellent!	merlin981	17-May-07 5:31
license?	famousj.dejazzd.com	17-Jan-07 9:32

Last Visit: 31-Dec-99 19:00 ¡ ¡ Last Update: 29-Dec-15 6:41 Refresh 1 2 Next ...

General ¡¡ News ¡¡ Suggestion ¡¡ Question ¡¡ Bug ¡¡ Answer ¡¡ Joke ¡¡ Praise ¡¡ Rant ¡¡ Admin ¡¡

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Wednesday, December 30, 2015
2:46 AM



Visual Studio 2013 Succinctly

By
Alessandro Del Sole

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: Jeff Boenig
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Alessandro Del Sole has been a Microsoft Most Valuable Professional (MVP) since 2008. Awarded MVP of The Year in 2009, 2010, 2011, and 2012, he is internationally considered a Visual Studio expert. Alessandro has authored six printed books and three e-books on programming with Visual Studio, including *Visual Basic 2012 Unleashed*, *Visual Studio LightSwitch Unleashed*, *Hidden Visual Studio LightSwitch*, *Hidden WPF*, and *Visual Basic 2010 Unleashed*. He has written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and in English for many developer portals, including the Visual Basic Developer Center from Microsoft. He is a frequent speaker at Italian conferences, and he has released a number of apps for Windows Phone and Windows 8. He has also produced a number of instructional videos both in English and Italian. You can follow him on Twitter at @progalex.

Introduction

Microsoft Visual Studio 2013 is the new version of the popular integrated development environment for building modern, high-quality applications for a number of platforms such as Windows, the web, Microsoft cloud, tablets running Windows 8, and Windows Phone devices.

The key word in Visual Studio 2013 is "productivity." Microsoft well knows that developers spend most of their time writing code, so the new version adds many tools to increase productivity and to help developers be faster and more efficient. The .NET Framework 4.5.1 does not introduce any new features to managed languages such as Visual Basic and Visual C#; on the other side, lots of enhancements have been made to the integrated development environment.

In this book you will learn what's new in Visual Studio 2013 for the code editor, for the debugger, for Windows 8.1, for the web and the cloud (including the new integrated support for Windows Azure subscriptions), and much more. There are so many improvements to support new and updated technologies that you will easily understand why a new release of Visual Studio was important after only one year.

Visual Studio 2013 ships with the following editions: Ultimate, Premium, Professional, and Test Professional, plus the free Express editions. Most features described in this book require Visual Studio 2013 Professional, but some of them require Visual Studio 2013 Ultimate, which is the most complete edition available. I will specify where the Ultimate edition is required. For a full comparison, you can look at this page in the Visual Studio portal from Microsoft. You can also download a fully-functional, 90-day trial of Visual Studio 2013 Ultimate (and other editions) from the Visual Studio Downloads page. The Express Editions are lightweight, free of charge editions of specific development environments for non-professional developers, hobbyists, and students that can be used even for commercial purposes.

Available products are Visual Studio 2013 Express for Windows Desktop (which you use to build WPF, Windows Forms, and Console apps), Visual Studio 2013 Express for Windows (which you use to build Windows Store apps for Windows 8.x), and Visual Studio 2013 Express for Web (which you use to build apps and sites for the web and the cloud). You can download the Express Editions from the same download page as above.

It is worth mentioning that Visual Studio 2013 allows building apps for Windows Phone 8, but not for Windows Phone 7.x. If you still need to build apps for Windows Phone 7.x, you will need to use Visual Studio 2012 and the Windows Phone 7.1 SDK. Visual Studio 2013 can be safely installed side-by-side with Visual Studio 2012. Also, Visual Studio 2013 allows opening most Visual Studio 2012 projects without modifying files for a perfect backward compatibility. A full list of conversion scenarios is provided in the MSDN documentation.

Chapter 1 Synchronized Settings and Notifications

Most developers work on different computers, such as desktop workstations, laptops, and servers. In most situations, developers install Visual Studio onto each computer they work with. As you know, the IDE (integrated development environment) is customizable and allows adjusting a number of settings, such as adding buttons to toolbars, changing colors, using different fonts, and so on. Before Visual Studio 2013, you had to adjust settings manually on every installation of Visual Studio, which requires more time and the risk of forgetting to change some settings. Visual Studio 2013 introduces *synchronized settings*, so that every time you make customizations in the environment, these will be automatically applied to the other installations of Visual Studio on different computers. This chapter explains how this new feature works and how you can customize your work environment just once.

Sign in to Visual Studio

The first time you start Visual Studio 2013, you will be asked to specify a default profile, such as web programming, Visual Basic programming, general development, and other profiles from previous versions of the IDE. This is a step you've already taken many times, so I will not spend much time here. After selecting the profile, Visual Studio 2013 will ask you to sign in with a Microsoft Account (formerly known as Windows Live ID). A Microsoft Account is an email address based on one of the Microsoft providers such as Hotmail, Live, or Outlook. Figure 1 demonstrates this.

Figure 1: Visual Studio 2013 asks you to sign in with a Microsoft Account.

Signing in with a Microsoft Account is not mandatory; you will certainly be able to use Visual Studio without an email address. However, signing in is advantageous for the following reasons:

- You can take advantage of Synchronized Settings, as described later in this chapter.
- Signing in with a Microsoft Account permanently unlocks any Visual Studio Express you have installed.
- You will be automatically logged in to the Team Foundation Service account associated with your email address if you subscribed to this service.
- You can use a trial version of Visual Studio for 90 days instead of 30.
- Signing in will unlock Visual Studio if your Microsoft Account is associated with an MSDN subscription.

Assuming you already have a Microsoft Account, click **Sign in**. At this point you will be asked to enter your email address and password, as represented in Figure 2.

Figure 2: Enter your credentials to get started.

Click **Sign in**. At this point Visual Studio will recognize your profile and will show some information while preparing the environment for the first use (see Figure 3).

Tip: If you are installing Visual Studio 2013 for the first time on a computer but you already installed it onto a different machine, this is also the moment in which settings are synchronized. Information on how settings are synchronized is coming shortly.

Figure 3: Enter your credentials to get started.

Visual Studio will also ask you to select one of the available development settings. If you have a previous version installed, such as Visual Studio 2012, the new IDE provides an option to import customizations from the previous version. You can choose from among General, JavaScript, SQL Server, Visual Basic, Visual C#, Visual C++, Visual F#, Web Development, and Web Development (Code Only). Choose the one that is closest to your interest. If you do not know what the best choice for you is, simply choose the General settings. Also, you will be able to select one of the available graphic themes (Light, Dark, Blue). Once signed in, Visual Studio shows your profile name and picture at the upper right corner of the IDE, including shortcuts to access your profiled detailed information and to connect to Team Foundation Service.

Note: Team Foundation Service is a cloud-based version of Team Foundation Server, the popular Microsoft platform for team collaboration. With Team Foundation Service you can host team projects and take advantage of source control and other team development tools wherever you are. Another important reason for signing into Visual Studio with a Microsoft Account is that the IDE automatically connects your account to the associated Team Foundation Service account. To create your Team Foundation Service account, visit http://tfs.visualstudio.com.

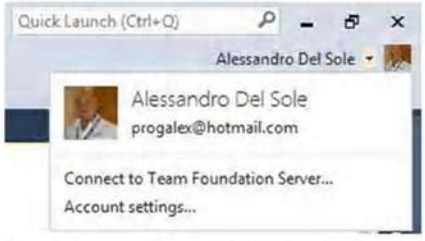


Figure 4: The IDE recognizes the user and provides useful shortcuts.

Once you have signed in, your settings are ready to be synchronized and shared across multiple computers.

Synchronized settings

By default, Visual Studio 2013 can synchronize the following settings:

- Development settings. These are related to the development profile selected the first time you ran Visual Studio and can be changed by selecting Tools, Import and Export Settings, Reset All Settings.
- Theme settings, available in Options, Environment page, General tab
- Startup settings available in Options, StartUp
- All settings for the text editor available in Options, Text Editor
- All settings for fonts and colors available in Options, Environment, Fonts and Colors
- All default and custom keyboard shortcuts, available in Options, Environment, Keyboard
- Customized command aliases

Tip: Command aliases are a way to enter commands inside the Command window and allow opening dialogs or launch other tasks in Visual Studio, instead of using menus and menu items. The full list of built-in aliases is available at: http://msdn.microsoft.com/en-us/library/c3a0kd3x.aspx

When you make changes or customizations, Visual Studio stores the aforementioned settings on the cloud, that is, on Microsoft servers, and associates those settings to the Microsoft Account you used to sign in. When you sign into Visual Studio with the same account on a different computer, the IDE downloads settings associated to your account and applies them to the active environment. Developers have been requesting this feature for a long time and finally Visual Studio 2013 solves the problem. This is just another example of how the cloud can make your life easier as a developer. Remember that synchronization works even if you have different editions of Visual Studio 2013, such as Ultimate, Premium, and Professional. Synchronization also applies to Express Editions, but it does not work if you have Express and non-Express editions on the same machine.

Selective synchronization

You can disable settings synchronization or choose what you want to synchronize among the settings listed in the previous paragraph. To accomplish this, go to the **Tools** menu, then select the **Options** submenu, then the **Synchronized Settings** command. Figure 5 shows how the Options dialog appears at this point.

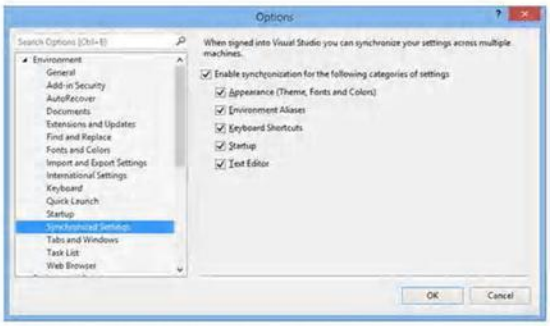


Figure 5: Enabling Synchronization and Settings Selection

The **Enable synchronization for the following categories of settings** check box is selected by default. If you unselect it, synchronization will stop until you explicitly re-enable it. You can also select one or more specific settings you want to synchronize, excluding settings you do not use. Click **OK** to apply your changes.

Synchronization conflicts

For various reasons, synchronization across multiple machines can occasionally fail. If this happens, Visual Studio shows a message in the Notification Hub (see the next topic of this chapter). The MSDN documentation describes three possible solutions with some manual work. As a personal suggestion, turn synchronization off on the computer where it was not successful, and then sign out. Next, sign in again and turn synchronization on again.

Notifications Hub

Visual Studio 2013 introduces a new concept of notifications. The goal is keeping the developer informed about product updates, extension updates, documentation updates, license issues, problems with the Microsoft Account, unresolved conflicts, and other errors. The IDE presents notifications to you via the Notifications Hub. The Notification Hub consists of a new tool window called Notifications and of a small flag icon (the Notifications button) placed near the Quick Launch bar, indicating the number of available notifications. To open the Notifications window:

1. Click the **View** menu.
2. Select the **Notifications** entry.

You can also click the **Notifications** button on the Quick Launch bar for faster opening. Figure 6 shows the Notification Hub.

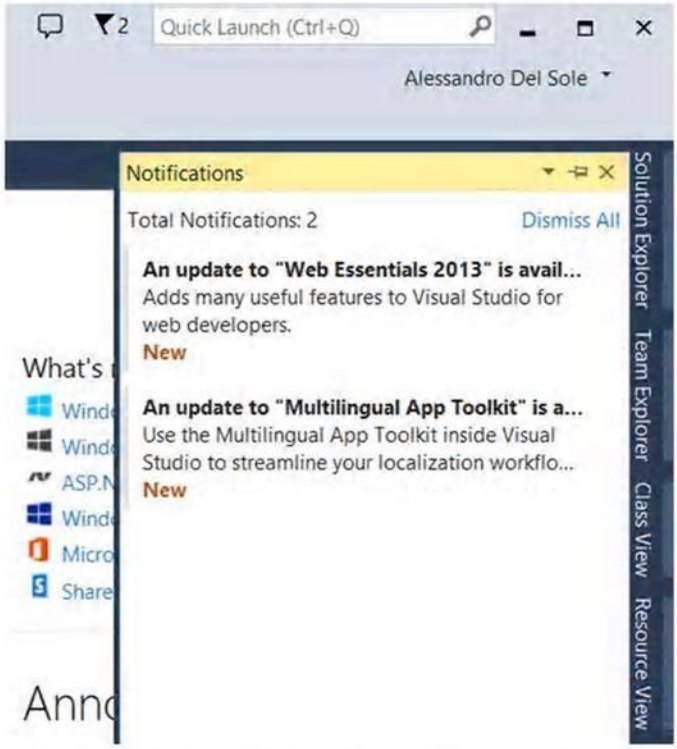


Figure 6: Enabling Synchronization and Settings Selection

If you click the button, the Notifications tool window opens and shows a full list of notifications. You can expand the notification description and, in case the notification is about an update, you will be able to click a hyperlink that will redirect you to the download page. You can also ignore all notifications by clicking **Dismiss All**.

Chapter summary

Among the new features in the IDE, Visual Studio 2013 makes it easy to share settings across multiple computers with Synchronized Settings; with this feature, most settings are saved to the cloud and applied to all of your other installations of Visual Studio. The Notifications Hub provides an easy way to download updates and to present information about other issues related to Visual Studio. Both features require you to sign into Visual Studio with your Microsoft Account, which also allows connecting to other Microsoft services without additional effort.

Chapter 2 The Start Page Revisited

The Start Page has been an important place in Visual Studio since the early days. In the first versions of Visual Studio .NET, it was a static page containing shortcuts for creating new or opening existing projects, and a place to get the latest announcements from Microsoft. In Visual Studio 2010, the Start Page was completely redesigned; it was built upon Windows Presentation Foundation (WPF), providing not only a better integration with the IDE but also offering an opportunity to build completely customized entry points. In Visual Studio 2013 the Start Page has evolved even more, becoming the place where you start your work as well as learn about new and updated technologies.

A new Start experience

An important concept behind the development experience in Visual Studio 2013 is that programmers should have everything they need inside the active page. The Start Page in Visual Studio 2013 has been reorganized based on this concept and includes not only shortcuts for working with projects, but also updated links to learning resources and announcements, all in one place. The Start Page has a dynamic layout, meaning that items inside the page are automatically rearranged when you resize the Visual Studio's window. Figure 7 shows how the Start Page appears when you run Visual Studio 2013.

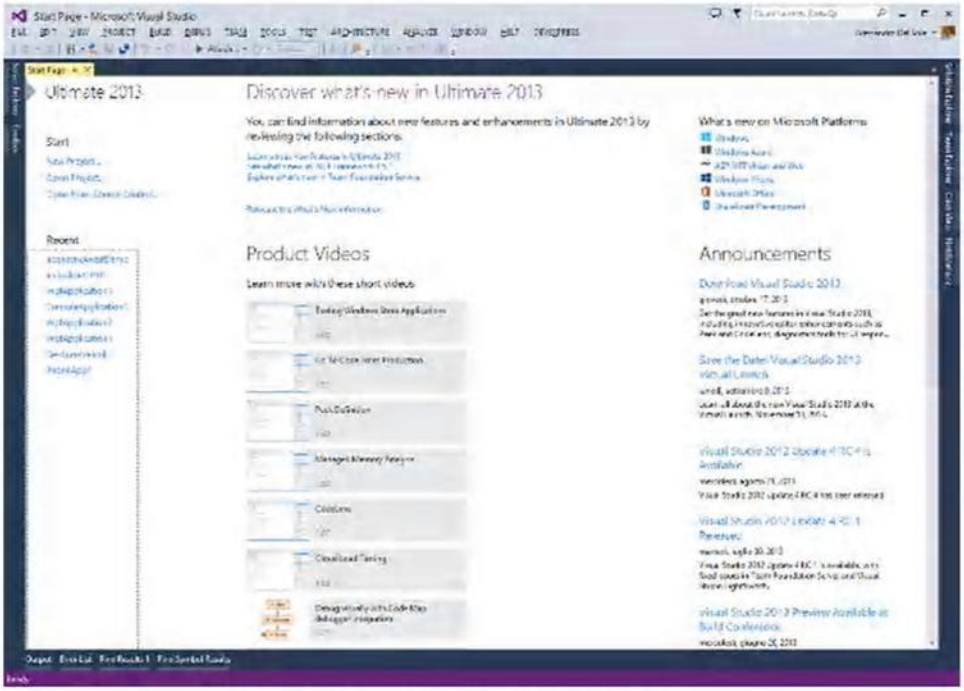


Figure 7: The Start Page in Visual Studio 2013

The Start Page is made of several areas, each described in the next sections of this chapter. Of course, you can still create and use custom start pages based on WPF (as you could do in Visual Studio 2010 and 2012) or you can disable the Start Page and choose a different entry point by using Tools, Options, Startup.

Note: This chapter does not cover how to build custom start pages. If you wish to create your own start page, read Start Pages in the MSDN documentation.

What you find here is a description of what the Start Page contains and how it can make your life easier.

Work with projects

Tip: In this and the next subsections, use Figure 7 as a reference to locate items in the Start Page.

The first thing you probably do when you launch Visual Studio is open a project. On the left side of the Start Page you will find two areas related to working with projects, Start and Recent. Start contains shortcuts for creating new projects or opening existing ones, including from source controls platforms such as Team Foundation Server, Team Foundation Services, and GIT. Recent shows a list of recent projects; if you right-click the name of a project in the recent list, you will be able to open the project, open the containing folder, or remove the project from the list.

Staying up to date: Announcements

The Announcements area shows news about product updates, new releases, events/conferences, and technical content from the various teams in Redmond working on Visual Studio. This is not new in the Start Page, but the behavior is different. First, you can no longer customize the source of the announcements; in the previous versions of Visual Studio you could specify a different RSS feed to show contents, but now the news channel cannot be changed. However, the news channel is now filtered with information that you actually need to stay up to date with new releases and with events focused on Visual Studio 2013.

Learning

The Start Page now has more content for getting started with Microsoft technologies and with specific product features, as described in this section.

What's new on Microsoft platforms

The **What's new on Microsoft platforms** area has shortcuts that make it easier to access the MSDN documentation for each of the most recent development platforms, operating systems, and collaboration platforms, such as Windows 8, Windows Azure, the web and ASP.NET, Windows Phone, Office, and SharePoint.

Product Videos

The **Product Videos** area allows watching short instructional videos about specific features in the Visual Studio IDE. This is very useful for a better understanding of most of the new features, because the videos show them in action with practical examples. You might see the following text:

```
We have a lot of great content to show you, but we need your permission to get it and
keep it updated.
```

If you see this message, you need to click **Tools**, then click **Options**, and select **Startup** under the **Environment** node in the **Options** dialog; finally, check the **Download content every** check box. The default time interval is 60 minutes but you can increase or decrease the value. The reason for this is that Visual Studio uses your Internet connection to retrieve the list of available contents, so it needs your permission first.

Discover what's new

At the top of the Start Page you can find an area that offers shortcuts to learn what new features are available in Visual Studio 2013, the .NET Framework 4.5.1, and Team Foundation Services. Such shortcuts will direct you to the appropriate page of the MSDN documentation.

Chapter summary

With its revisited and dynamic layout, the Start Page in Visual Studio 2013 is more than a simple place where you create new projects or pick up existing ones; the Start Page is now the place where you can easily find all the learning resources and product releases you need to start building applications for the most recent Microsoft platforms.

Chapter 3 Code Editor Improvements

The code editor in Visual Studio 2013 is one of the areas of the IDE where Microsoft made many investments. The goal is to make developers stay focused on the code they are writing, helping them perform common tasks more quickly and save time. This chapter describes new features in the code editor that will help you be more productive when writing code.

Peek Definition

Peek Definition is a new feature that you can use to see and edit the definition of a class or class member inside a popup shown within the active code editor window. This helps you avoid the need to leave the active window in order to open the code file that contains the code block you need to edit. To understand how it works, create a new Console application then add a **Person** class like the following.

Visual C#

```
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime DateOfBirth { get; set; }

        public override string ToString()
        {
            return string.Format("{0} {1}, born {2}",
                this.FirstName, this.LastName,
                this.DateOfBirth.ToShortDateString());
        }
    }
```

Visual Basic

```
Public Class Person
    Public Property FirstName As String
    Public Property LastName As String
    Public Property DateOfBirth As Date

    Public Overrides Function ToString() As String
        Return String.Format("{0} {1}, born {2}",
            Me.FirstName, Me.LastName,
            Me.DateOfBirth.ToShortDateString())
    End Function
```

```
End Class
```

The **Main** method of the sample application simply creates a new instance of the **Person** class and assigns some values as in the following code.

Visual C#

```
    class Program
    {
        static void Main(string[] args)
        {
            Person person = new Person();

            person.FirstName = "Alessandro";
            person.LastName = "Del Sole";
            person.DateOfBirth = new DateTime(1977, 5, 10);

            Console.WriteLine(person.ToString());
            Console.ReadLine();
        }
    }
```

Visual Basic

```
Module Module1
    Sub Main()
        Dim person As New Person

        person.FirstName = "Alessandro"
        person.LastName = "Del Sole"
        person.DateOfBirth = New DateTime(1977, 5, 10)

        Console.WriteLine(person.ToString())
        Console.ReadLine()
    End Sub
End Module
```

Now suppose you want to make some edits to the **Person** class, such as renaming members or adding new ones. Right-click the type name (**Person** in our example) and select **Peek Definition**. As you can see from Figure 8, a pop-up appears showing the code of the **Person** class and the name of the code file where it is defined.

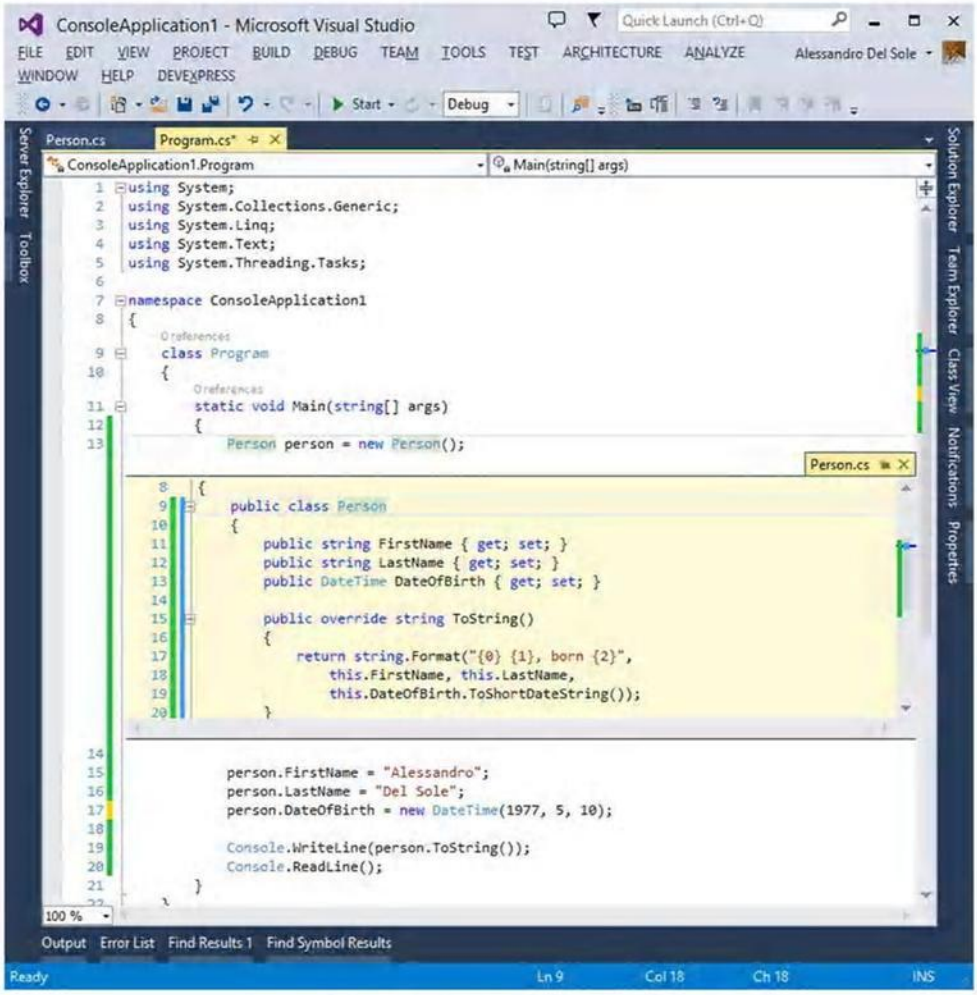


Figure 8: Editing Code with Peek Definition

Peek Definition offers a fully functional editor, so you can change your class (or member) definition according to your needs without leaving the active window. If you make any changes, these are immediately visible in the code that uses the class. When done, you can simply click the usual Close button to hide the Peek Definition content. This tool is a very useful addition, because it not only allows you to stay in the active editor window while making changes to a type, but it also makes it easier to find type definitions among millions of lines of code and thousands of code files.

CodeLens

Note: This feature is available only in the Ultimate edition.

In many situations, you might need to know how many times an object has been used in your code and where. The previous (and current) versions of Visual Studio provide a tool called **Find All References**, which shows a list of references to an object inside a tool window called Find Symbols Results that you can invoke by right-clicking an object's name in the code editor and then selecting **Find All References**. Visual Studio 2013 makes a step forward, offering an additional integrated view of code references called **CodeLens**. Take a look at Figure 9, which shows the `Person` class definition inside the code editor.

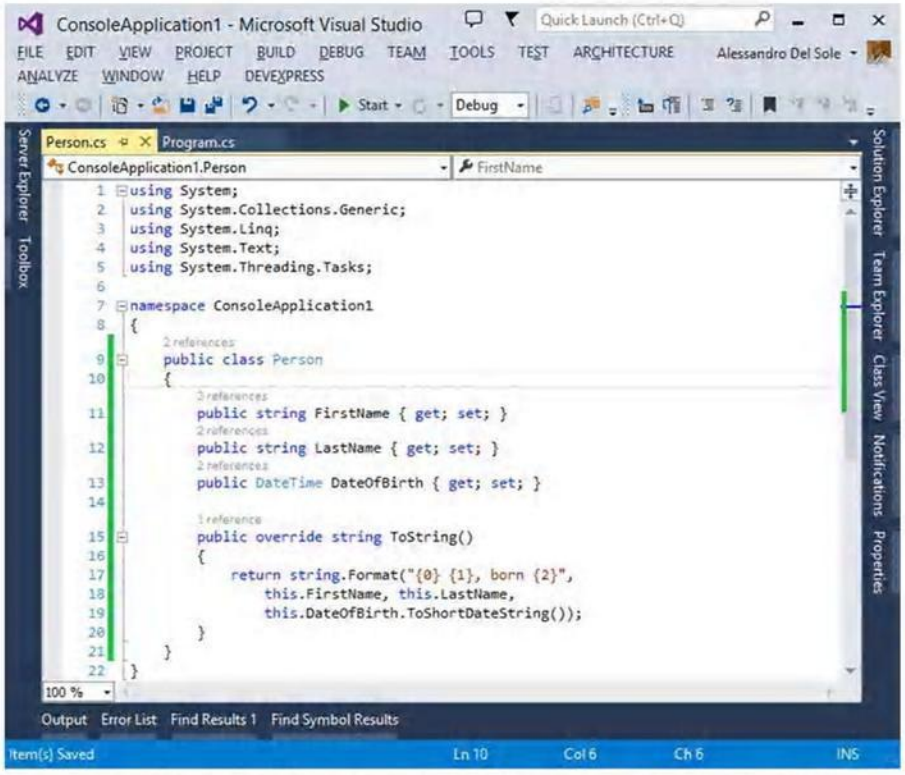


Figure 9: Visual Studio 2013 shows the number of references for each type and member.

As you can see, above each type and member name Visual Studio shows the number of references. If you click that number, a tooltip will show where the object is used; if you pass the mouse pointer over the line of code that contains the reference, another tooltip will show the full code block containing the reference (see Figure 10).

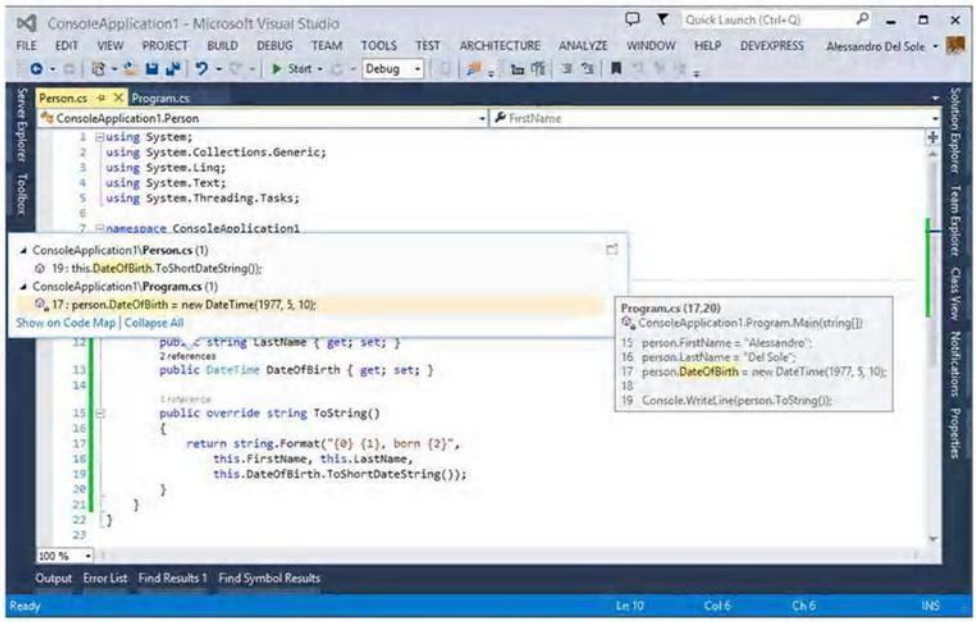


Figure 10: Finding Object References from within the Code Editor

CodeLens also shows the containing code file for each reference and the line number where the object is used, and allows fast navigation to the reference by double-clicking the line of code in the tooltip. Actually CodeLens does also an amazing job when your solution is under the source control of Team Foundation Server; in fact, it can show information about unit tests, passed tests, failed tests, and edits made by other team members.

Enhanced Scroll Bar

In most of real world projects, code files are made of hundreds of lines of code, so finding specific code blocks inside a file can become difficult. In order to make it easier to browse very long code files, Visual Studio 2013 provides an improved scroll bar in the code editor window, known as enhanced scroll bar. Basically the scroll bar can show a map of the code (map mode) so that when you move the mouse pointer up and down, a magnifier shows a preview of the code block. This is very useful with long code files, if you want to see some code definition without jumping from one position to another in the code file.

To enable the map mode, right-click the scroll bar, then select **Scroll Bar Options**. In the **Options** dialog, locate the **Behavior** group and then select the **Use map mode for vertical scroll bar** option, as shown in Figure 11.

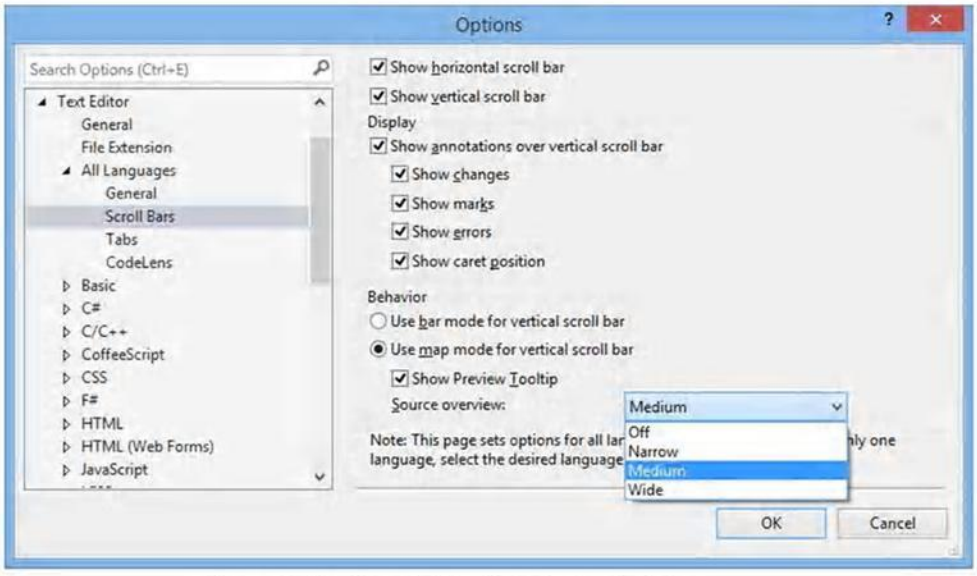


Figure 11: Enabling the Map Mode for the Scroll Bar

Note: Enabling the code preview is optional, but I encourage you to leave it selected. After all, it's the real benefit of this tool.

You can also choose the size of the map by changing the value of the **Source overview** box. The default value is Medium, which is a good choice for most situations. Click **OK** to enable the map mode. When you go back to the code editor, you can see the scroll bar's new look. Figure 12 shows an example.

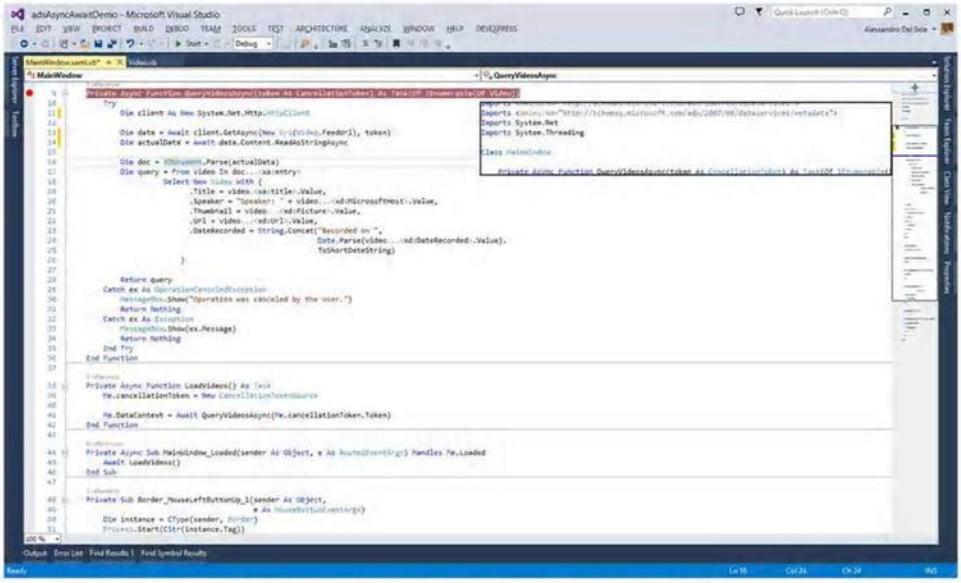


Figure 12: Browsing Code with the Scroll Bar in Map Mode

When you move the mouse pointer over the scroll bar, a tooltip shows a preview of the code for the current position on the map. A blue line indicates the cursor position, yellow dots indicate edited lines of code, and red dots represent breakpoints. You can simply revert to the classic scroll bar by going back to the scroll bar options and selecting the **Use bar mode for vertical scroll bar** option.

Tip: The enhanced scroll bar works with all languages supported by Visual Studio 2013. This means that when you enable the map mode, the scroll bar will show the map for every code file in any language until you disable it again.

Navigate To

Another key feature of Visual Studio 2013 in the code editor is called **Navigate To**. With this feature, you can easily find the definition of a type or member by placing the cursor on the type or member and then pressing **CTRL + ,** . Figure 13 demonstrates how Visual Studio 2013 shows types and members that contain the name selected in the code editor.

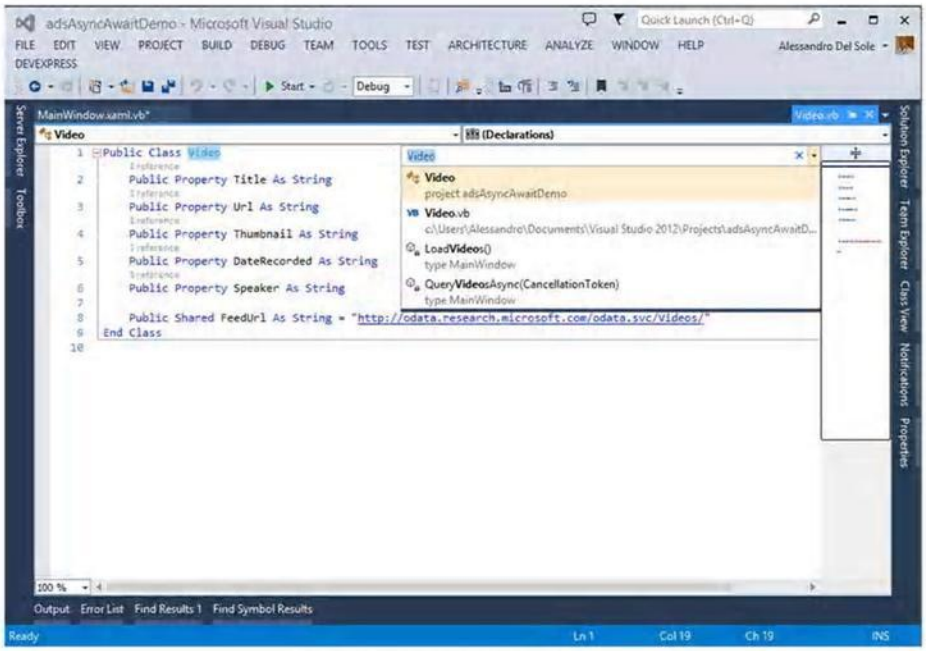


Figure 13: Using Navigate To

As you can see from Figure 13, a list of objects matching the type or member name is shown. You can press the up and down arrows on your keyboard to see every definition in the code editor without actually activating a window. As you can easily understand, Navigate To becomes particularly useful if you have multiple definitions of the same type in your solution and you want to go to its definition quickly.

Chapter summary

The code editor in Visual Studio 2013 has been dramatically enhanced with new features that increase the developer's productivity by making it easier to complete common tasks. Peek Definition, integrated references, the enhanced scroll bar, and Navigate To give their contribution to make the new IDE a great place for writing code.

Chapter 4 XAML IntelliSense Improvements

XAML (*eXtensible Application Markup Language*) is a markup language used to design the user interface in technologies such as Windows Presentation Foundation (WPF), Silverlight, Windows Phone, and Windows Store Apps for Windows 8. If you have experience with at least one of these technologies, which I assume you have, you know that Visual Studio, especially in the latest versions, offers a pretty good designer. In addition, Microsoft offers another tool called Expression Blend, which is dedicated to professional designers and allows working on the user interface with professional tools without interfering with the managed code behind. As a developer, most of the time you will work with Visual Studio. Although the designer has reached a very good level of productivity, a lot of times you will need to write XAML code manually; this is a very common practice, for example when you need to resize elements in the UI with exact proportions or when you need to assign styles or set data-bindings to controls. When you edit the XAML manually, you use the XAML code editor, which implements the IntelliSense technology, allowing you to write code faster. However, IntelliSense for XAML has always lacked some important points, such as recognizing available data sources and resources when using data-binding or assigning styles. Finally, Visual Studio 2013 addresses this issue and introduces a lot of new goodies into the XAML code editor.All of these new features are available to all technologies based on XAML.

Note: In the first preview of Visual Studio 2013, the new features in the XAML code editor were only available to Windows Store Apps. This limitation has been removed in the Visual Studio 2013 Release Candidate.

XAML IntelliSense for data-binding and resources

In Visual Studio 2013 you can now take advantage of IntelliSense when assigning a data source to a binding expression or when you assign a resource such as a style.

Note: This feature only works with data sources and resources that you declare in XAML. If you create an instance of a collection in managed code (at runtime), this cannot be recognized by the IntelliSense. It also works with design-time information that you declare through the d: XML namespace.

Let's use an example to see how this feature works.

The goal of the example is declaring a collection of objects and binding the collection to the user interface using the new IntelliSense features. Data will be shown inside a **`ListBox`** control. Create a new WPF Application project called *Chapter 4*, for the sake of consistency. Add a new folder to the project and call it Model. Add a new class to the folder, called **`Person`**. The code for the new class looks like the following.

Visual C#

```
namespace Chapter4.Model
{
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
    }
}
```

Visual Basic

```
Namespace Model
    Public Class Person
        Public Property FirstName As String
        Public Property LastName As String
        Public Property Age As Integer
    End Class
End Namespace
```

Note: While Visual C# automatically adds a namespace definition for each subfolder you create in the project, Visual Basic doesn't; it just recognizes objects under the root namespace. For the sake of consistency in both languages, we are adding a namespace declaration in Visual Basic so that we can use the same XAML with both.

This is a very simple class with only three properties, but we are focusing on the new tools now rather than on writing complex code. The next step is adding to the Model folder a new collection of **`Person`** objects, called **`People`**. The code is the following.

Visual C#

```
using System.Collections.Generic;
using System.Collections.ObjectModel;

namespace Chapter4.Model
{
    //add a using System.Collections.ObjectModel; directive
    public class People: ObservableCollection<Person>
    {
        public People()
        {
            Person one = new Person {LastName="Del Sole",
                                     FirstName="Alessandro", Age=36};
            Person two = new Person { LastName = "White",
                                      FirstName = "Robert", Age = 39};
```

```
            Person three = new Person { LastName = "Red",
                                        FirstName = "Stephen", Age = 42 };

            this.Add(one);
            this.Add(two);
            this.Add(three);
        }
    }
}
```

Visual Basic

```
Imports System.Collections.ObjectModel
Namespace Model
    Public Class People

        Inherits ObservableCollection(Of Person)
        Public Sub New()
            Dim one As New Person() With {.LastName = "Del Sole",
                                          .FirstName = "Alessandro",
                                          .Age = 36}
            Dim two As New Person() With {.LastName = "White",
                                          .FirstName = "Robert", .Age = 39}
            Dim three As New Person() With {.LastName = "Red",
                                            .FirstName = "Stephen",
                                            .Age = 42}

            Me.Add(one)
            Me.Add(two)
            Me.Add(three)
        End Sub
    End Class
End Namespace
```

The **People** class inherits from **ObservableCollection<Person>**. The constructor of the **People** collection creates three instances of the **Person** class, and populates them with sample data. The reason why we are creating a collection this way is that IntelliSense for XAML does not support collections created at runtime. Instead, with this approach we can declare the collection in the application's resources; every time a class is declared in the XAML resources, the constructor of the class is invoked, so in our case an instance of the collection is automatically created and populated when added to the XAML resources. Such an instance can then be data-bound to controls in the user interface. To do so, double-click the **MainWindow.xaml** file in Solution Explorer. When the designer and the XAML editor appear, first add the following namespace declaration within the **Window** tag.

```
<Window x:Class="Chapter4.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Chapter4.Model"
        xmlns:controls="clr-namespace:Chapter4"
        Title="MainWindow" Height="350" Width="525">
```

This is necessary in order to reference the **People** and **Person** classes. The next step is declaring the data source as a resource, as shown in the following code.

```
    <Window.Resources>
        <local:People x:Key="PeopleData"/>
    </Window.Resources>
```

This is the point in which an instance of the **People** collection is declared, so we are ready to bind data to a **ListBox** control. As you know, in order to present information coming from a collection, the so-called item controls (like the **ListBox**) need to implement a **DataTemplate**. Let's add a **ListBox** and its **DataTemplate** without pointing to any data source, by writing the following code within the **Window** tag.

```
    <Grid>
        <ListBox Name="PeopleBox" >
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="Black"
                            BorderThickness="2">
                        <StackPanel Orientation="Vertical">
                            <TextBlock />
                            <TextBlock />
                            <TextBlock />
                        </StackPanel>
                    </Border>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
```

The data template simply presents the value of each property of the **Person** class with a **TextBlock** control, arranged inside a **StackPanel** container. The **Border** adorner is used for a better highlighting inside the designer, but it is optional. For a full demonstration of the new IntelliSense features, we can also add a new style for **TextBlock** controls. In Solution Explorer, double-click the **App.xaml** file. Within **the Application.Resources** tag, add the following style, which allows presenting text in red and with different size and weight for the current font.

```
<Application.Resources>
    <Style x:Key="MyTextBlockStyle" TargetType="TextBlock">
        <Setter Property="Foreground" Value="Red"/>
        <Setter Property="FontSize" Value="16"/>
        <Setter Property="FontWeight" Value="SemiBold"/>
    </Style>
</Application.Resources>
```

Now you are ready to test the new amazing IntelliSense for XAML.

IntelliSense for data-binding

Switch back to the MainWindow.xaml file locate the **`ListBox`** control. As you know, item controls are populated by assigning their **`ItemsSource`** property with an instance of a collection, either at design-time (with XAML code) or at runtime (in managed code). We previously declared a data source as a resource, so a Source binding expression is needed to assign it as the **`ItemsSource`** property for the **`ListBox`**. To understand the benefit of XAML IntelliSense at this point, type the following code (not just copy/paste).

```
<ListBox Name="PeopleBox"
         ItemsSource="{Binding Source={StaticResource PeopleData}}">
```

After you type **`StaticResource`**, you will see how the IntelliSense will show a list of available objects that can be used for data-binding, as demonstrated in Figure 14.

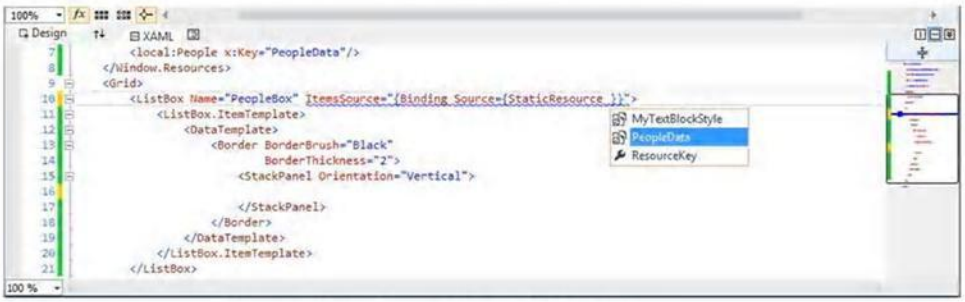


Figure 14: IntelliSense for Data-Binding in Action

Select the **`PeopleData`** object to finalize data-binding. As in every other scenario where you write code, IntelliSense will help you complete the expression while you type. You can also select the data source with the arrows on your keyboard and then press **Tab**.

Tip: If you have multiple data-bound controls in the Window, you might want to bind the parent container's DataContext property instead (in this case the Grid) and then assign the ItemsSource property with the {Binding} expression.

Similarly, you can bind **TextBlock** controls to properties of the collection with the help of IntelliSense, as demonstrated in Figure 15.

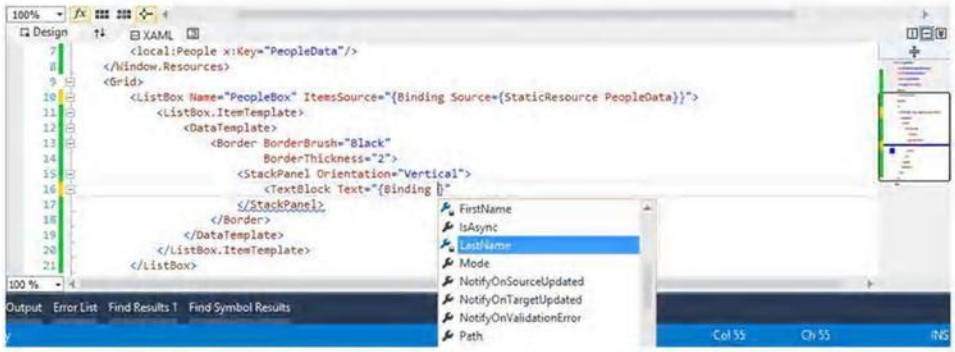


Figure 15: IntelliSense shows properties that can be data-bound.

This is a tremendous benefit for several reasons. First, you can write code faster. Secondly, you do not need to remember the name or the casing of properties, thus minimizing the risk of typos. The complete code of the **ListBox**'s data template is the following.

```
        <DataTemplate>
           <Border BorderBrush="Black"
                   BorderThickness="2">
                <StackPanel Orientation="Vertical">
                    <TextBlock Text="{Binding LastName}"/>
                    <TextBlock Text="{Binding FirstName}"/>
                    <TextBlock Text="{Binding Age}"/>
                </StackPanel>
            </Border>
        </DataTemplate>
```

IntelliSense for resources

The next step is using IntelliSense to assign resources. We previously defined a style that must be now assigned to each **TextBlock** control in the Window. The code for the first **TextBlock** looks like the following. You might want to type the style assignment manually in order to see the XAML IntelliSense feature in action.

```
<TextBlock Text="{Binding LastName}"
           Style="{StaticResource MyTextBlockStyle}"/>
```

As it happened for data-binding, when you are assigning the **StaticResource** expression the IntelliSense will show available resources, as shown in Figure 16.

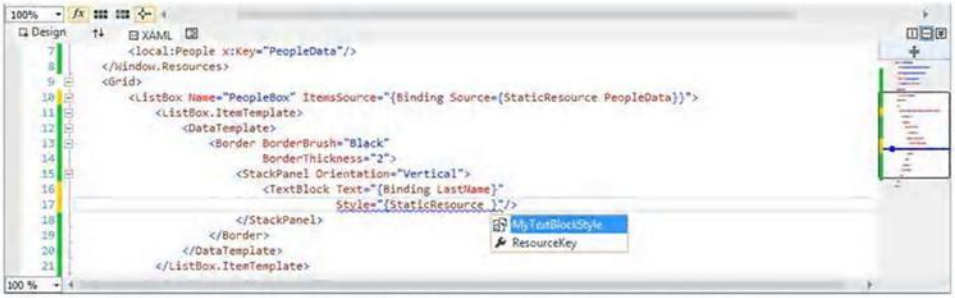


Figure 16: IntelliSense shows available resources for the specified control.

It is worth mentioning that, in the case of styles, IntelliSense will only show styles valid for the control that you are working on, either defined in the application or defined in the .NET Framework or SDK extensions. This is an additional benefit, since you not only will avoid errors and will write code faster, but you will be also be picking up only resources that are specific for the selected element of the user interface. For the sake of completeness, Figure 17 shows how the designer looks at this point.

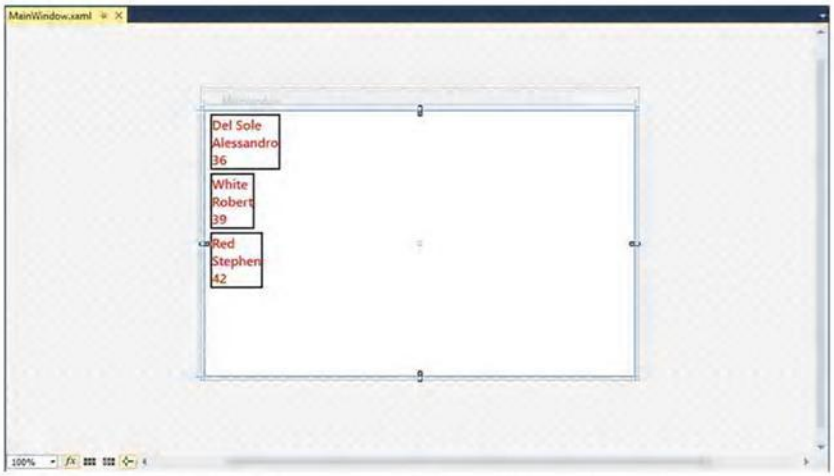


Figure 17: The Designer after the Data and Style Assignments

You can run the application by pressing **F5** to see how it displays data.

Go To Definition

Go To Definition is a feature that you already know from the managed code editor. With this feature, you can right-click an object's name, select **Go To Definition** from the context menu, and see how the object is defined in the Object Browser window, if it is a built-in object from the .NET Framework, or in the appropriate code file if it is an object you wrote. This feature is now available to the XAML editor too, as demonstrated in Figure 18.

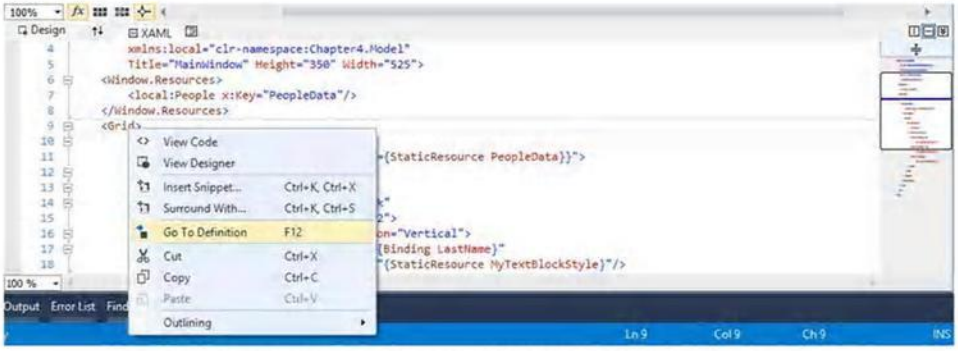


Figure 18: Go To Definition is now available in the XAML code editor.

Tip: The keyboard shortcut for Go To Definition is also F12 in the XAML code editor.

Go To Definition is available for the following objects:

- Resources
- System types
- Local types (custom controls)
- Binding expressions

Let's walk through every object to see the different behavior of Go To Definition.

Resources

In an XAML-based application, resources can be of two types: resources defined in an assembly (from .NET, from the SDK, or from a 3rd party library) and resources defined in the current application. In the first case, Go To Definition will open the Object Browser window pointing to the specified resource definition. For example, in the sample application created previously, place the cursor on the **`MyTextBlockStyle`** assignment in any of the **`TextBlock`** controls, then right-click and select **Go To Definition** (see Figure 19).

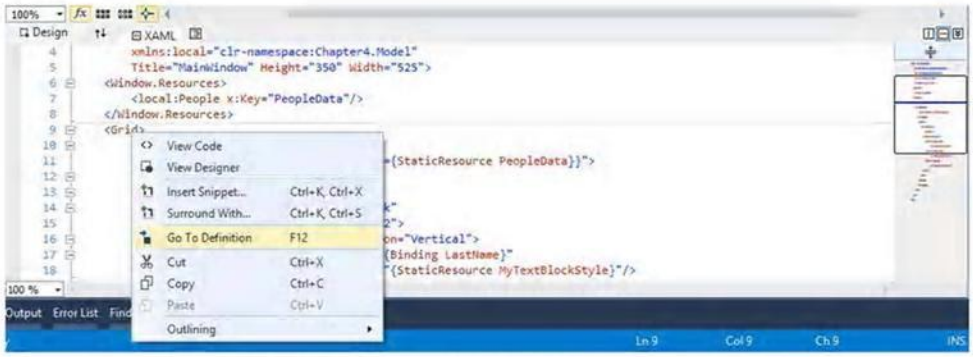


Figure 19: Go To Definition Over the Style Defined in the Sample Application

At this point, the code editor will open the definition of the resource at the exact position in the appropriate code file; in our case, the style definition inside the App.xaml file. As you can see (Figure 20), the cursor is placed at the beginning of the definition and the **`Style`** tag is selected.

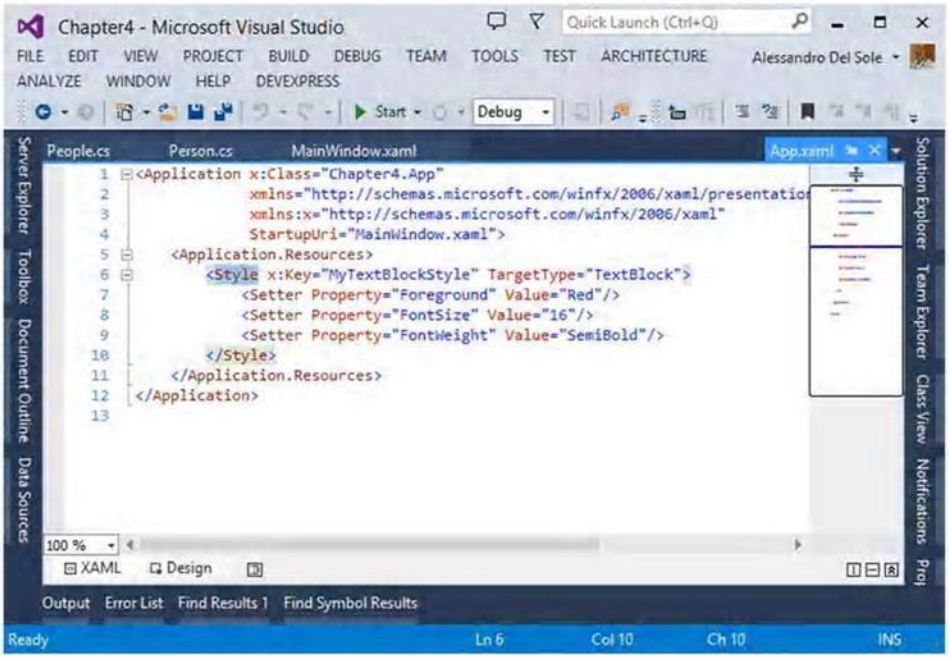


Figure 20: Go To Definition opens the resource definition at the exact position.

System Types

Go To Definition works with types defined in the .NET Framework and SDK extensions. Since the source code of these types is not available, Visual Studio shows the definition inside the Object Browser window. For example, if you select Go To Definition on the **`Grid`** control in the MainWindow.xaml file of the sample application, the Object Browser will be opened, showing the control's definition (see Figure 21).

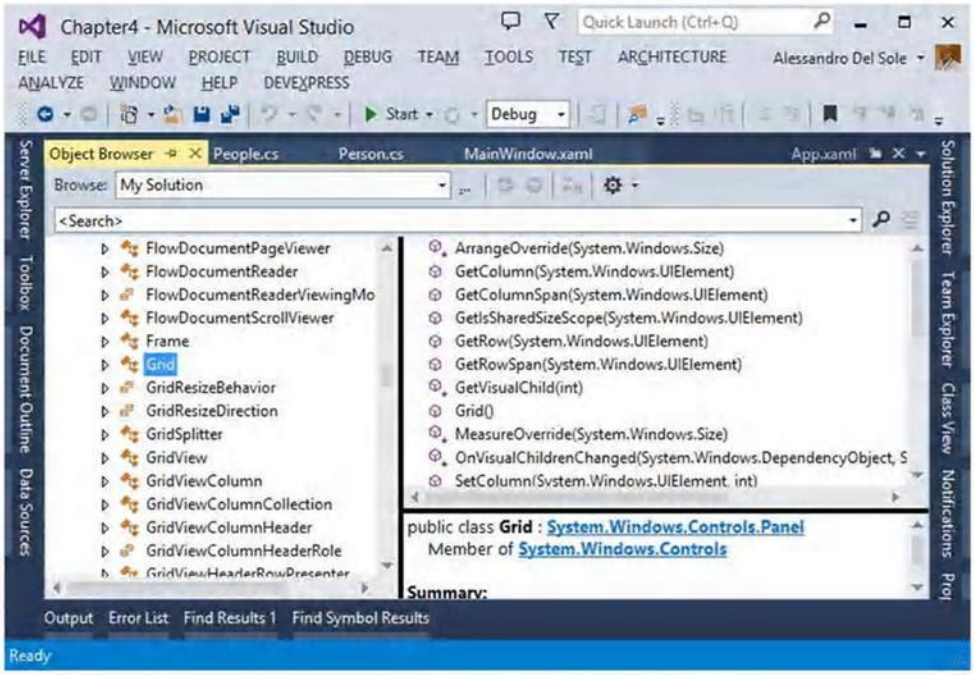


Figure 21: Seeing a Control's Definition with the Object Browser

Local Types

Go To Definition has a particular behavior with local types; these are user controls and custom controls created by developers.

Note: This is not a book about WPF and other XAML-based technologies, so I will not cover the difference between user controls and custom controls in detail. As a hint, user controls are the result of the composition of existing controls; custom controls extend existing built-in controls with additional functionalities in code, and provide templating, theming, and styling entry points. For further information, read the Control Authoring Overview in the MSDN Library.

To understand how it works, let's make some edits to the sample application. In **Solution Explorer**, right-click the project name, select **Add New Item**. Then, in the **Add New Item** dialog, select the **User Control (WPF)** template and name the new control **CustomBoxControl.xaml** (see Figure 22).

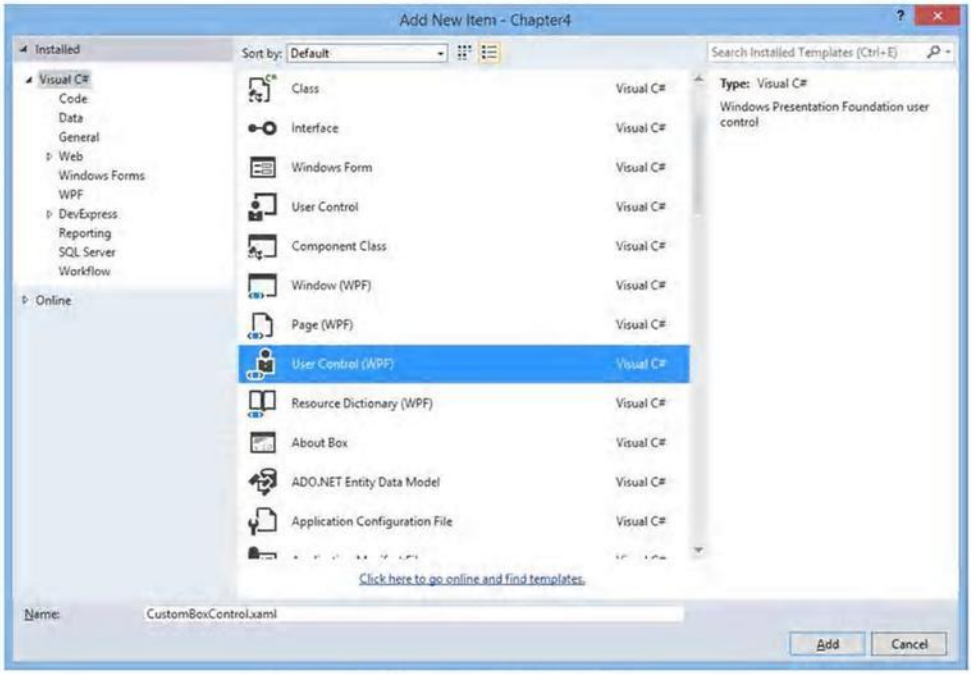


Figure 22: Adding a New User Control

Cut and paste the **`ListBox`** definition from MainWindow.xaml to the new control, and repeat this step for the **`local`** XML namespace declaration. Finally, add a local resource that points to the **`People`** collection as you did in MainWindow.xaml. The full code of the user control looks like the following.

```
<UserControl x:Class="Chapter4.CustomBoxControl"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d" xmlns:local="clr-namespace:Chapter4.Model"
             d:DesignHeight="300" d:DesignWidth="300">
    <UserControl.Resources>
        <local:People x:Key="PeopleData"/>
    </UserControl.Resources>
```

```
    <Grid>
        <ListBox Name="PeopleBox"
                 ItemsSource="{Binding
                 Source={StaticResource PeopleData}}">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Border BorderBrush="Black"
                            BorderThickness="2">
                        <StackPanel Orientation="Vertical">
                            <TextBlock Text="{Binding LastName}"
                                       Style="{StaticResource
                                       MyTextBlockStyle}"/>
                            <TextBlock Text="{Binding FirstName}"
                                       Style="{StaticResource
                                       MyTextBlockStyle}"/>
                            <TextBlock Text="{Binding Age}"
                                       Style="{StaticResource
                                       MyTextBlockStyle}"/>
                        </StackPanel>
                    </Border>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</UserControl>
```

In MainWindow.xaml, add the following XML namespace to include the user control.

```
xmlns:controls="clr-namespace:Chapter4"
```

Then, add the user control as follows.

```
    <Grid>
        <controls:CustomBoxControl/>
    </Grid>
```

If you did everything correctly, the designer now should still look like in Figure 17. Now, right-click **`CustomBoxControl`** inside the **`Grid`** and select **Go To Definition**. As you can see (Figure 23), Visual Studio 2013 opens the Find Symbol Results window and shows two results, one for XAML and one for managed code.

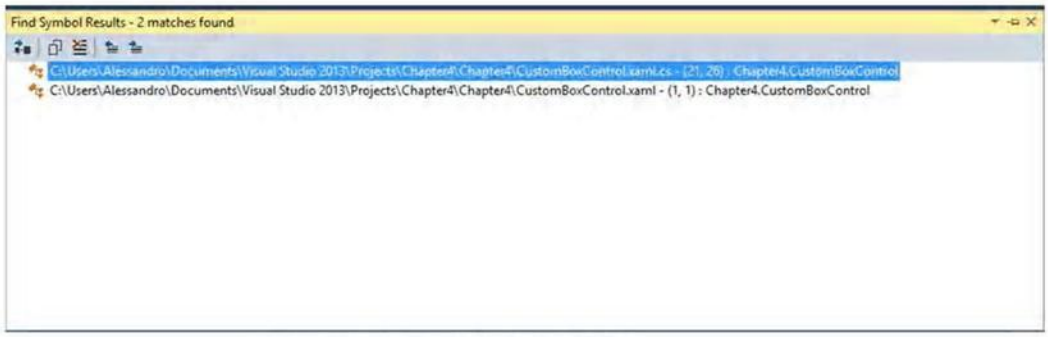


Figure 23: Adding a New User Control

The reason is that user controls (and custom controls as well) are made of two components, the XAML file defining the user interface and a code-behind file. You can then double-click the file you need and see the appropriate definition. The Find Symbol Result is a good choice, because you can also see the line number where the definition begins.

Binding expressions

Go To Definition also works with binding expressions. For instance, you can right-click the name of the data source or of the bound property inside a **`Binding`** expression and select Go To Definition. With data sources (collections) defined as static resources, Go To Definition moves you to the definition of the resource; with data sources defined in managed code, Go To Definition attempts to make a full symbol search in the code, showing search results in the Find Symbol Results window. In the case of data-bound properties, Go To Definition moves you to the code of the class that exposes such a property.

Automatic closing tag

When you add an item in XAML, the code editor automatically adds the closing tag. For instance, when you add a **`<Button>`** tag, Visual Studio adds the matching **`</Button>`** tag. This is not new, since it is the normal behavior in the earlier versions. The new feature is that if you add the slash before the **>** symbol in the first tag, the closing tag is automatically removed. In other words, in this code:

```
<Button Width="100" Height="50" Click="Button_Click" Name="Button1" Content="Click
me!"></Button>
```

If you type the slash before the **>** symbol, it automatically turns into the following code.

```
<Button Width="100" Height="50" Click="Button_Click" Name="Button1" Content="Click
me!"/>
```

This is another way the editor can help you write code faster.

IntelliSense matching

Continuing its purpose to make your coding experience better, Visual Studio 2013 adds another feature to the XAML code editor, known as IntelliSense matching. Basically, when you start typing the name of a control or resource, the IntelliSense will help you find the appropriate control as you type, even if you enter the wrong characters. For example, Figure 24 shows how IntelliSense understands you need a **`StackPanel`** even if you are typing it incorrectly.

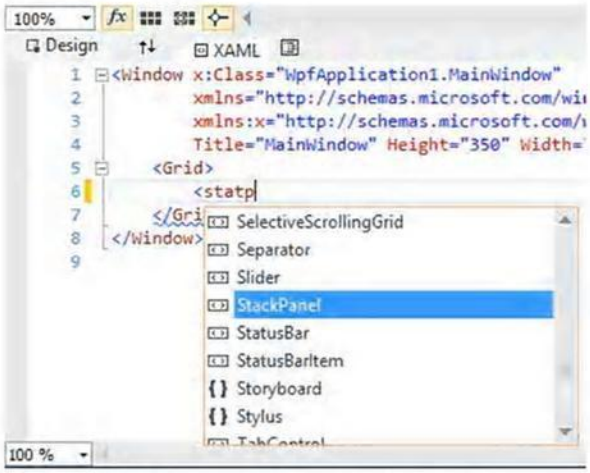


Figure 24: IntelliSense Matching makes it easy to select controls as you type.

IntelliSense does an excellent job, depending on how many identifiers can match what you are typing. For instance, in a Windows Store App, if you type **`Abbb`** it will suggest the **`AppBar`** control, which is probably your choice.

Better support for comments

A common issue in previous versions of Visual Studio is that when you add a comment to a code block containing another comment, it causes the code editor to show an error message. Figure 25 shows how Visual Studio 2012 handles this kind of situation.

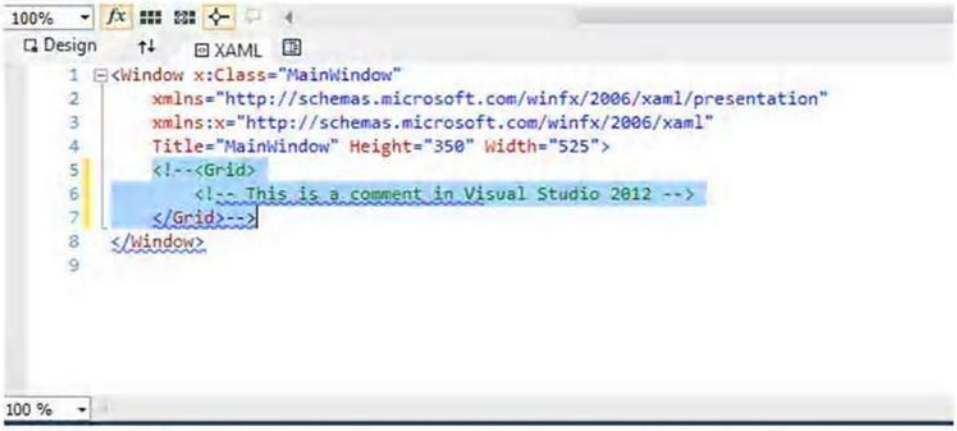


Figure 25: Visual Studio 2012 does not recognize nested comments correctly.

The problem was that the code editor did not recognize comment closing tags correctly. Visual Studio 2013 addresses this issue, so when you add a comment to a code block containing another comment, the entire code block gets commented, as demonstrated in Figure 26.

Figure 26: Visual Studio 2013 correctly recognizes nested comments.

Reusable XAML code snippets

Reusable code snippets for IntelliSense have been a very popular feature since Visual Studio 2005, but have always been limited to managed languages, XML, and HTML/JavaScript in version 2012. With code snippets you can take advantage of a huge code library offered by Visual Studio or create your own code snippets, so that it is easier to reuse your code with the support of IntelliSense. The need of code snippets for XAML has always been very strong, so many developers used different techniques to store their reusable code. I wrote myself an extension for Visual Studio 2010 to support XAML code snippets. Visual Studio 2013 makes another step forward, introducing built-in support for code snippets in the XAML code editor.

To use code snippets, simply right-click in the code editor and select **Insert Snippet** or **Surround With**, as shown in Figure 27.

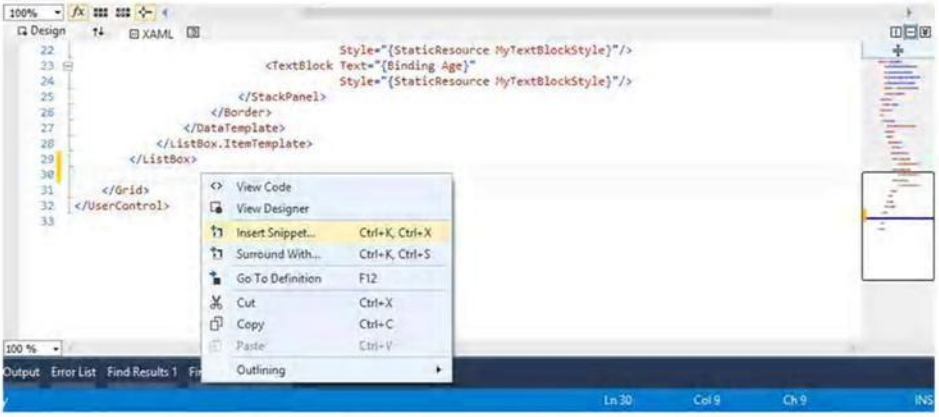


Figure 27: Visual Studio 2013 provides support for XAML code snippets.

At this point, a list of available code snippets appears. You can select the code snippet you need from the list and the related code will be placed there (see Figure 28).

Figure 28: Selecting a Code Snippet from the List

Note: This chapter is based on Visual Studio 2013 RTM released to the MSDN subscribers in October, 2013. In this release only one XAML snippet is supplied. At the time of this writing, we cannot predict if additional snippets will be provided, or if you will only be able to import your own, custom code snippets. This chapter does not explain how to build custom code snippet files, which is out of its scope. However, you can read this interesting article from Tim Heuer, which explains XAML code snippets from creation to deployment.

You can manage code snippet files with the Code Snippets Manager tool (available from the Tools menu), as you already did for other languages. This is the place where you can import, remove, and view detailed information about code snippet files. By adding XAML code snippets, Visual Studio 2013 bridges the gap with code editors for other languages.

Chapter summary

Without a doubt, adjusting the user interface and supplying data-bindings manually in XAML code is a very common task in any XAML-based technology, including WPF, Silverlight, Windows Phone, and Windows Store Apps, despite the existence of specialized tools for designing. Visual Studio 2013 finally provides important improvements to the XAML code editor, such as IntelliSense for recognizing data sources and styles; Go To Definition for system types, local types, custom, and user controls; commenting nested code blocks; better handling of closing tags, and IntelliSense matching to help you select controls quickly; and reusable code snippets, finally added to XAML completing the availability of this tool to all of the supported languages.

Chapter 5 Visual Studio 2013 for the web and Windows Azure

Programming for the web is the core business for many companies and developers. Creating websites means making an application available to potential customers worldwide through the Internet or providing internal portals or departmental applications through a local Intranet. Because of the importance of the web in a programmer's life, cloud computing platforms are becoming more and more important every day. With Windows Azure, Microsoft has released one of the most powerful and complete cloud infrastructures ever. With a platform like Windows Azure you no longer need an in-house data-center, removing the need of purchasing physical servers and paying for their maintenance; with Windows Azure, you only pay for services you actually use. This chapter does not explain what ASP.NET and Windows Azure are, nor does it explain how to create applications for both platforms; that's the goal of other resources. Instead, here we focus on how the new tooling available in Visual Studio 2013 makes programming for the web and the cloud an even more amazing experience with a deeper integration with the IDE.

Note: The .NET Framework 4.5.1 introduces some new features to ASP.NET. If you want to learn about what's new, you can visit the appropriate page in the MSDN Library. Here you learn about new features in the IDE for ASP.NET, not about the runtime.

What's new in the IDE for ASP.NET

Visual Studio 2013 introduces new tools and updates the existing environment for web development with ASP.NET. This chapter focuses on the most important features that you must know, as they will change the way you create web applications.

One ASP.NET: A new, unified experience

In the past, Microsoft released several technologies for creating web applications, like Web Forms, ASP.NET Dynamic Data, and MVC. Similarly, a number of frameworks and libraries were released, such as jQuery, jQuery Mobile, Web API, and Windows Identity Foundation. You could choose among several kinds of project templates in order to build web applications and add references to your desired frameworks later. Visual Studio 2013 dramatically simplifies this process by introducing the so-called **One ASP.NET**, which provides a unified development experience and makes it easy to use any of the available platforms, as well as making libraries interchangeable. But what does One ASP.NET mean in practice? To understand what it is, open Visual Studio 2013 and select **File**, **New Project**. Select the **Web** templates folder. As you can see from Figure 29, now there is only one project template.

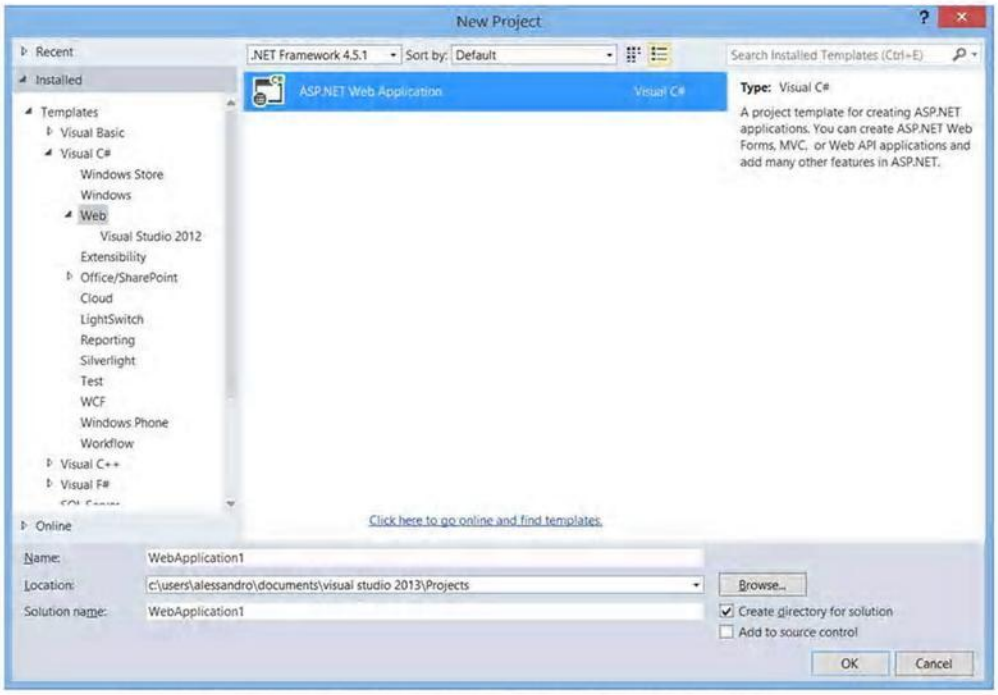


Figure 29: One ASP.NET also means simplifying a project's creation.

Unlike in the past, where you had to choose among a number of several project templates, now you have only one template. Don't be scared of this; in the next steps you will learn the reasons behind this feature and how to take advantage of it. For backward compatibility, you can still create web applications using templates inherited from Visual Studio 2012. You can simply expand the **Web** template folder and select the **Visual Studio 2012** subfolder (which is visible in Figure 29). You will see the classic list of available project templates based on Web Forms, MVC, and Ajax. Let's focus on One ASP.NET and double-click the single project template. At this point Visual Studio 2013 will show a new dialog, represented in Figure 30.

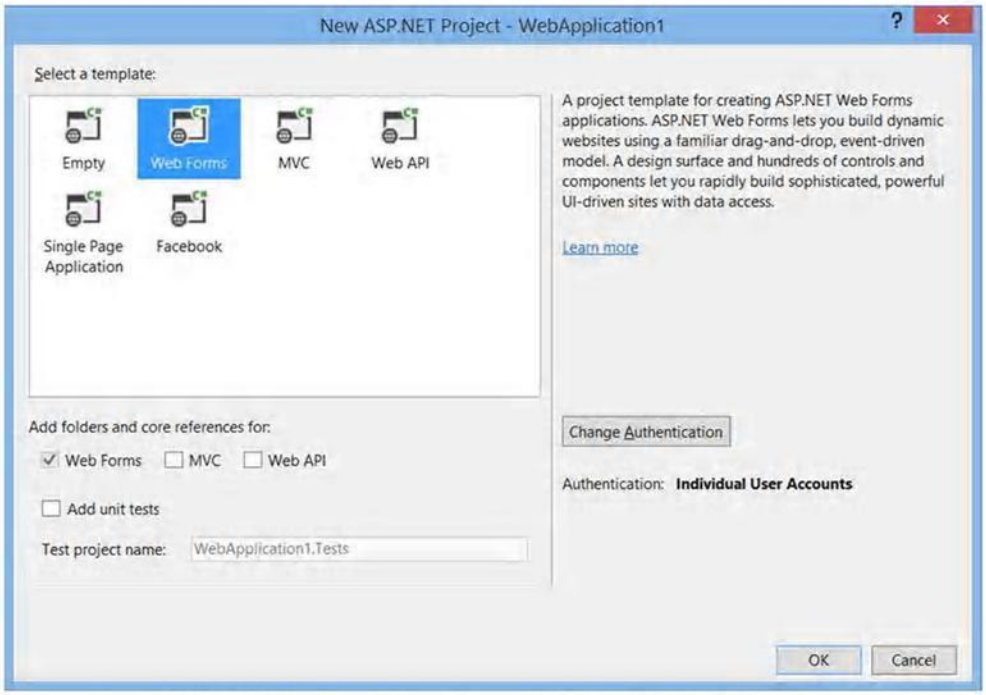


Figure 30: Selecting the Presentation Framework, Authentication, References

It's becoming clearer why the new approach is called One ASP.NET. In one place, you can:

- Select the presentation framework, such as Web Forms or MVC.
- Select a ready-to-use application stub, such as the Single Page Application or the Facebook application templates.
- Choose the authentication model: anonymous, Individual (ASP.NET), Windows (for Windows domains), Organizational (based on Active Directory, Office 365, and Windows Azure Active Directory).
- Add unit tests.
- Add folders and references to libraries that are specific to other frameworks; for instance, in a Web Forms application you can use MVC libraries and vice versa.

This new way of creating web applications gives you an opportunity to work with multiple kinds of libraries, taking full advantage of all the libraries from ASP.NET 4.5.1. Also, One ASP.NET simplifies the process by adding references for you. Experimenting with Web Forms, MVC, and other project templates is left to you as an exercise. Let's now take a closer look at new features from the IDE.

Scaffolding for Web Forms

Scaffolding is all about data. Basically with scaffolding the IDE can generate view models and views based on a modeled data source (such as the ADO.NET Entity Framework); with scaffolding, Visual Studio generates for you pages that can read, insert, delete, or update data without you writing a single line of code. Technically speaking, Visual Studio generates a **controller**, which is a class containing the necessary code to perform C.R.U.D. (Create, Read, Update, Delete) operations against data, and pages to work with data (**Views**), one for each of the C.R.U.D. operations. Scaffolding is not a new concept in the ASP.NET development; it was first introduced with ASP.NET MVC. The good news is that Visual Studio 2013 brings scaffolding to Web Forms as well. This is possible because of the One ASP.NET experience; in fact, Visual Studio injects the appropriate MVC dependencies into a Web Form project and then does most of the work for you. If you already used this technique in MVC projects, you will be familiar with most of the concepts.

Note: With Visual Studio 2008 and .NET Framework 3.5 Service Pack 1, Microsoft introduced ASP.NET Dynamic Data, an innovative way of creating modern, data-centric applications for the web. Dynamic Data is still available in later versions. The concept of scaffolding was the base of ASP.NET Dynamic Data, but what we mean today by scaffolding is pretty different, and is based on new frameworks, libraries, and implements different code-behind. For this reason, be sure you have clear that in this chapter we refer to scaffolding by indicating the MVC (and now Web Forms) implementations.

To understand how scaffolding works, we will now create a sample ASP.NET application based on Web Forms, then we will add a reference to a database. Then we will use the new tooling in Visual Studio 2013 to generate data-bound pages without writing a single line of code. Before you continue, ensure you have downloaded and installed the following prerequisites:

- Microsoft SQL Server 2012 Express Edition, as the database engine required for data access. I recommend you download either the "With Tools" or the "With Advanced Services" editions that will also install SQL Server Management Studio Express for database management.
- The Adventure Works sample database from Microsoft. It can be downloaded for free from this page of the CodePlex website.

We assume that you already know how to install a database like Adventure Works to SQL Server, so let's go ahead.

Create a sample project

First you will create a sample project. This is also the first time for you to see the One ASP.NET tooling in action. To create the project, follow these steps:

1. Select **File**, **New Project.**
2. Select the **Web** templates folder (see Figure 1).
3. Select the one **ASP.NET Web Application** and call the new project **ScaffoldingDemo.**
4. Click **OK.**
5. In the **New ASP.NET Project** dialog (take Figure 30 as a reference), select **Web Forms** as the presentation framework, then select the **MVC** checkbox.

6. For the sake of simplicity, change the authentication mode to anonymous. This is just for testing purposes; in a real world scenario you must select the appropriate authentication type according to your needs. To change the authentication, click **Change Authentication**, then in the **Change Authentication** dialog, select **No Authentication** (see Figure 31).
7. Click **OK** to create the project.

Figure 31: Changing the Authentication Type

Tip: For your curiosity or for real needs, while you are in the Change Authentication dialog, try to click on each option to see how you can implement different authentication types and how each type satisfies specific platform requirements.

Adding Data Connection and Entity Data Model

When the project is ready, in **Solution Explorer** right-click the project name, select **Add New Item**, and select the **Data** template folder. This is the point in which a data connection will be added, as demonstrated in Figure 32. Select the **ADO.NET Entity Data Model** item template; call the new model **AdventureWorks.edmx** and click **Add**.

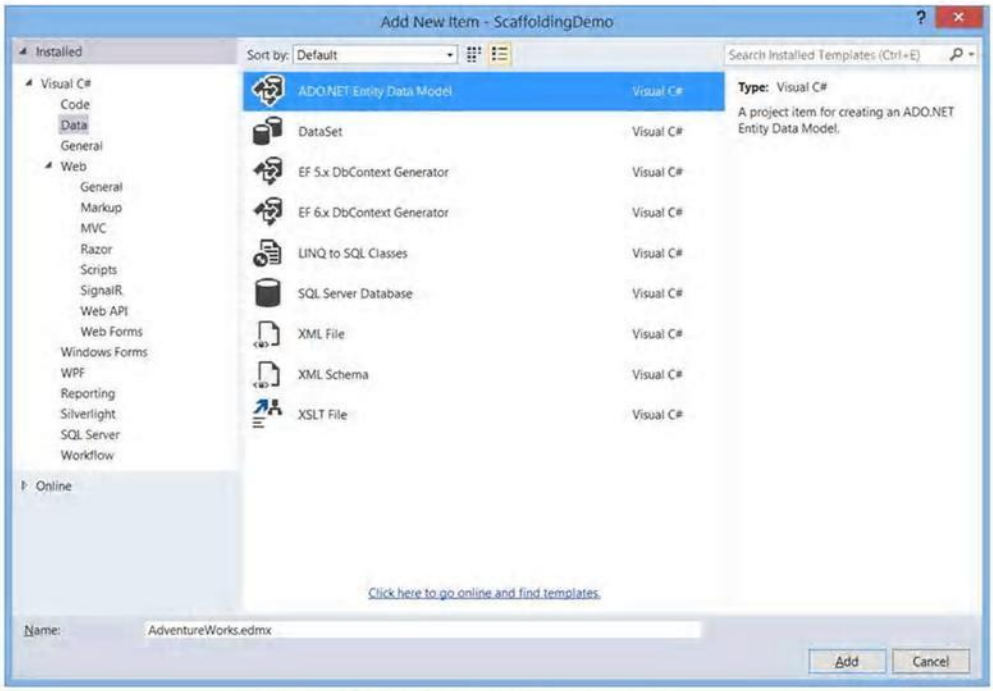


Figure 32: Adding a New Entity Data Model

As you know, the Entity Data Model is based on the ADO.NET Entity Framework. In the **Entity Data Model** wizard, select the **Generate From Database** option first, then click **Next**. Click the **New Connection** button, then add a new connection that points to the **AdventureWorks** database. Figure 33 shows how the Connection Properties window will look like at this point; of course, you can have a different server name on your machine.

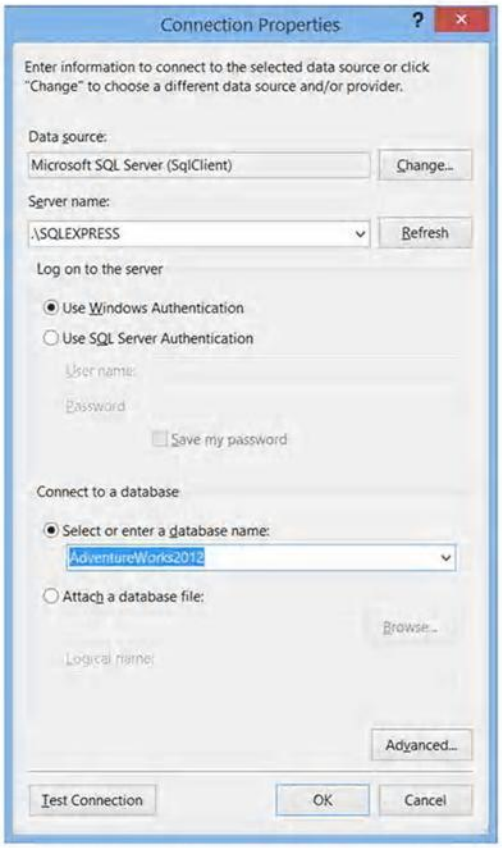


Figure 33: Adding a New Entity Data Model

At this point the Entity Data Model Wizard will show summary information for the newly created connection (see Figure 6). You can definitely leave unchanged the identifier for the connection settings or provide a different one (at the bottom of Figure 34).

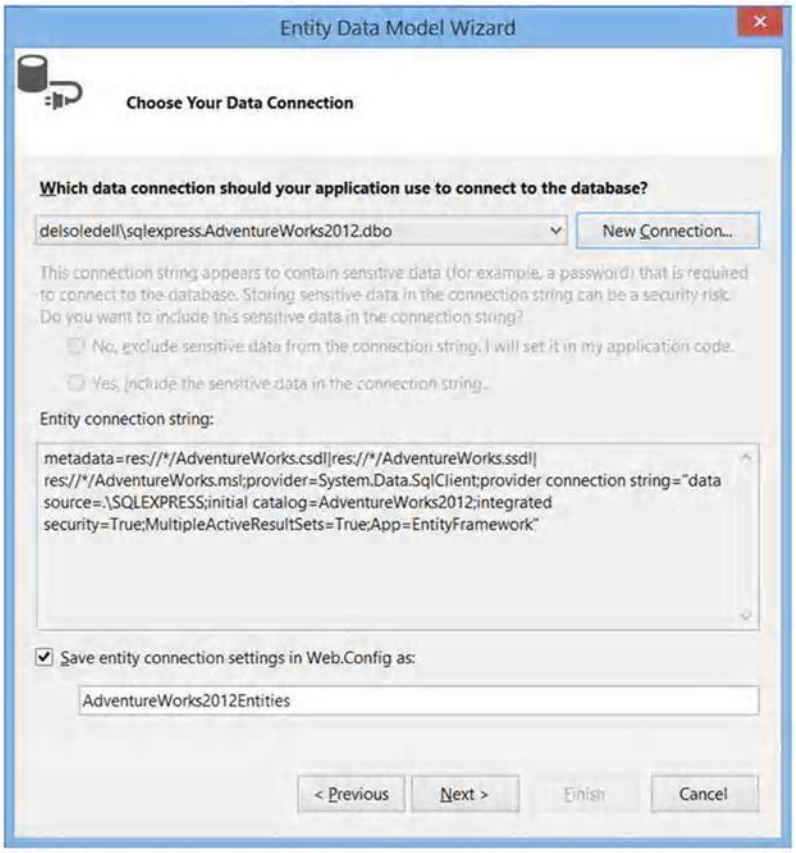


Figure 34: Summary Information for the New Entity Data Model

When you click **Next**, Visual Studio will ask you to specify the version of the Entity Framework you want to use, between 5.0 and 6.0 (default option). Leave unchanged the selection on version 6.0 and click **Next**. At this point you will need to specify the tables or views you want to add to the model. Just select the **Person** table, as we want to demonstrate concepts easily without having a complex data structure. Figure 35 shows how to make such a selection.

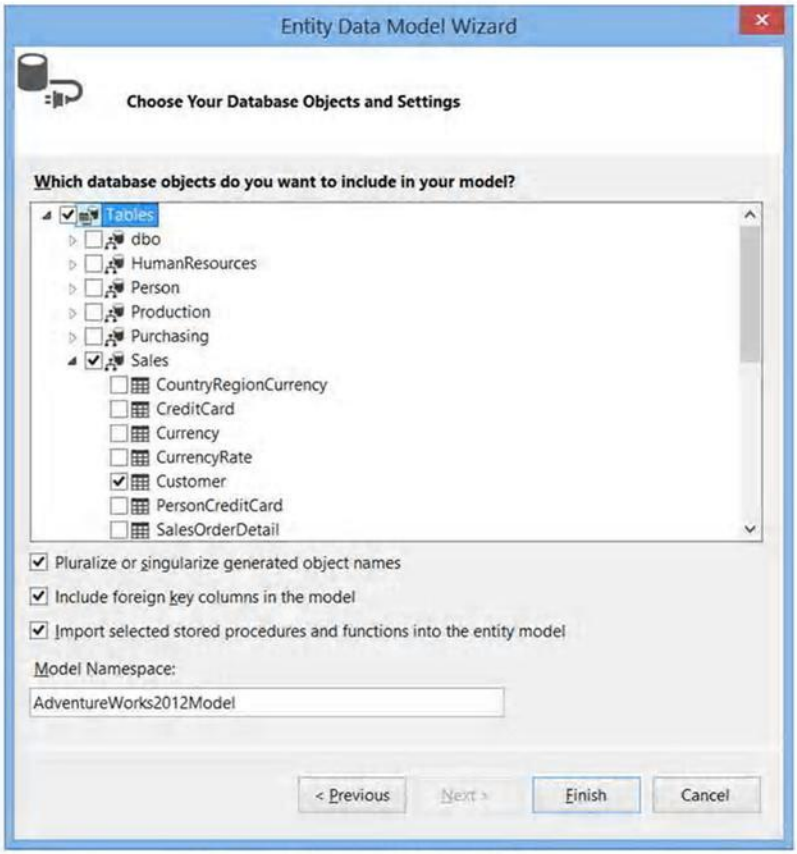


Figure 35: Selecting Database Objects

Click **Finish**. After a few seconds the Entity Data Model designer shows the .NET representation of the selected table (see Figure 36).

Tip: If you see a message that says "Running this template can potentially harm your computer," ignore it. Visual Studio always analyzes code snippets that execute actions against local resources, such as a database, including auto-generated snippets, but of course executing such actions is safe at this point.

Before doing anything else, rebuild the project (**CTRL** + **Shift** + **B**) so that all references are updated.

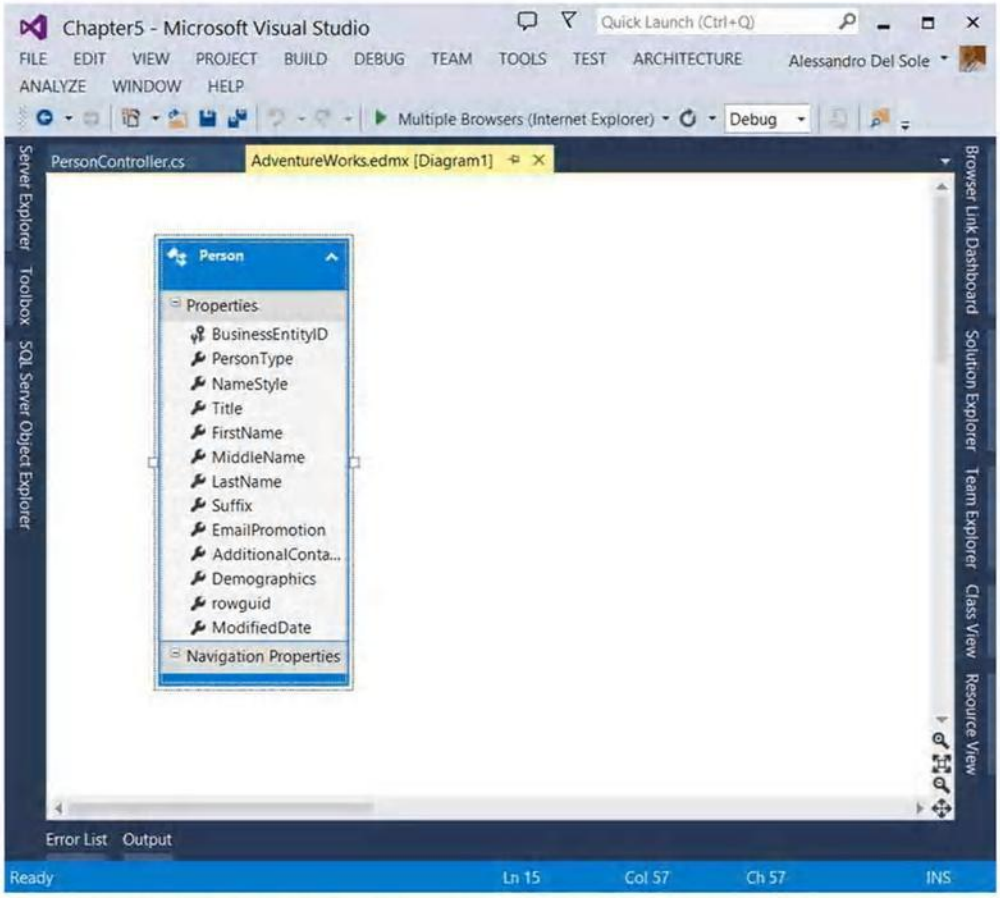


Figure 36: The Entity Data Model designer shows the table representation.

Depending on the database version you have installed, you might see additional entities in your designer. In fact, there are some differences between versions 2008 and 2012. You can ignore those additional entities and focus on entities you see in Figure 36. In the designer, right-click the **`Demographics`** property and select **Delete from Model**. The reason for this is that the **`Demographics`** property contains long XML markup that would make it difficult to provide readable figures for this e-book. Now that you have a connection to your data source, you need to present data and give users an opportunity of editing data. So, let's dive into scaffolding.

Generate data-bound pages with scaffolding

The benefit of scaffolding is the ability to generate data-bound views without writing a single line of code. Visual Studio 2013 generates a controller for each entity set and one page per action, which means one page for listing a collection of items, one for adding an item, one for editing, and one for deleting. Since scaffolding generates views, in **Solution Explorer** right-click the **Views** folder, then select **Add**, **New Scaffolded Item**. The **Add Scaffold** dialog appears and allows specifying the item you need, as you can see in Figure 37.

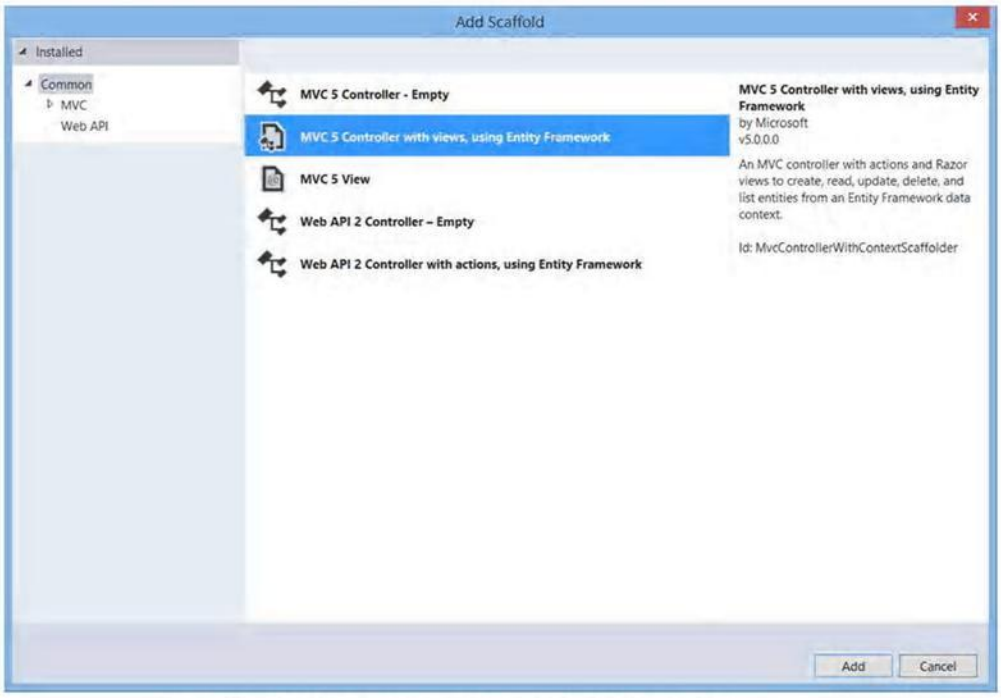


Figure 37: Selecting an Appropriate Controller for Scaffolding

As you can see, you can choose among different kinds of controllers. In this case you are working with the Entity Framework, so the appropriate controller is MVC 5 Controller with views, using Entity Framework. You can also choose an empty controller or a view. Also notice how you can take advantage of the Web API framework to create controllers that can be exposed to other consumers. When you click Add, Visual Studio shows the Add Controller dialog (see Figure 38).

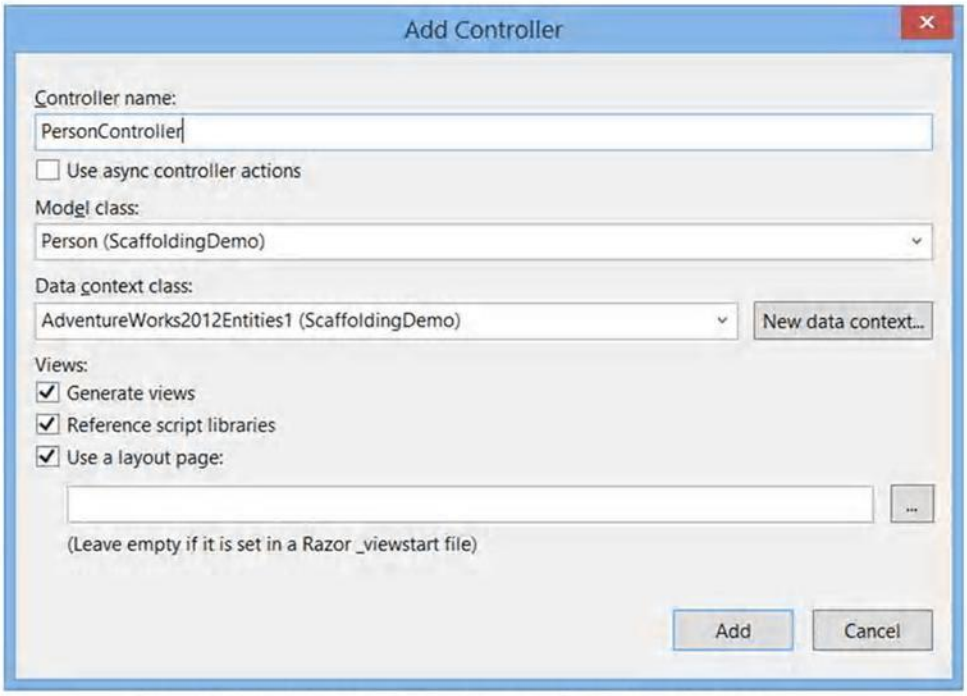


Figure 38: Specifying Properties and Settings for the New Controller

Here you can specify a number of settings for the new controller. The following table describes these settings and indicates how to rename items.

Table 1: Members and settings for the Add Controller dialog

Item name	Description	Value
Controller name	The name of the controller class that will be generated to interact with data	PersonController
Model class	The entity class that will be managed by the controller	Person

Item name	Description	Value
Data Context class	The context class generated by the Entity Framework to represent the database in a modeled way	AdventureWorks2012Entities
Generate views	Select this checkbox to make Visual Studio generate views (pages) for you	True
Reference script libraries	Select this checkbox to import scripting libraries	True
Use a layout page	Select this checkbox if you want to use a custom page for the layout; no need for this example	False

Click **Add**. In a few seconds, Visual Studio 2013 generates the **`PersonController`** class and a number of .cshtml files (under the Views\Person subfolder); each file is a page whose name is self-explanatory, such as Create.cshtml or Delete.cshtml. If you double-click the **`PersonController`** class in Solution Explorer, you will see the code that interacts with the data model. Figure 39 shows the result of scaffolding that is the controller and generated files in Solution Explorer.

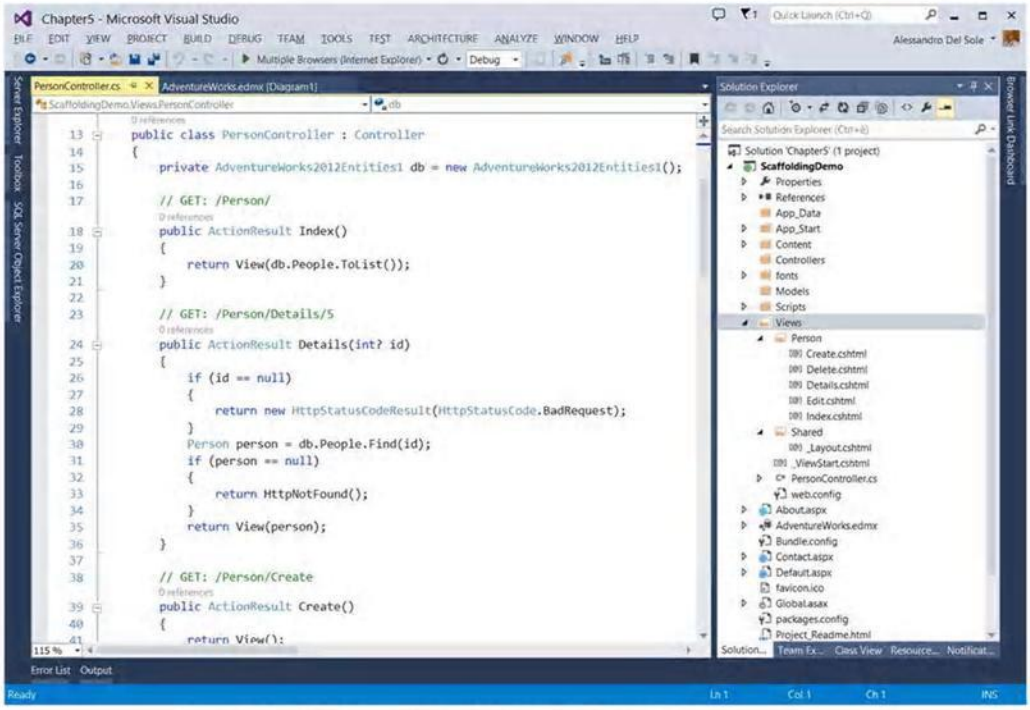


Figure 39: The Result of Scaffolding in Solution Explorer

The full code for the **PersonController** class follows.

Visual C#

```
    public class PersonController : Controller
    {
        private AdventureWorks2012Entities1 db = new
AdventureWorks2012Entities1();

        // GET: /Person/
        public ActionResult Index()
        {
            return View(db.People.ToList());
        }

        // GET: /Person/Details/5
        public ActionResult Details(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
```

```
            }
            Person person = db.People.Find(id);
            if (person == null)
            {
                return HttpNotFound();
            }
            return View(person);
        }

        // GET: /Person/Create
        public ActionResult Create()
        {
            return View();
        }

        // POST: /Person/Create
        // To protect from overposting attacks, please enable the specific
properties you want to bind to, for
        // more details see http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult
Create([Bind(Include="BusinessEntityID,PersonType,NameStyle,Title,FirstName
,MiddleName,LastName,Suffix,EmailPromotion,AdditionalContactInfo,rowguid,Mo
difiedDate")] Person person)
        {
            if (ModelState.IsValid)
            {
                db.People.Add(person);
                db.SaveChanges();
                return RedirectToAction("Index");
            }

            return View(person);
        }

        // GET: /Person/Edit/5
        public ActionResult Edit(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Person person = db.People.Find(id);
            if (person == null)
            {
                return HttpNotFound();
            }
            return View(person);
        }
```

```
        // POST: /Person/Edit/5
        // To protect from overposting attacks, please enable the specific
properties you want to bind to, for
        // more details see http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult
Edit([Bind(Include="BusinessEntityID,PersonType,NameStyle,Title,FirstName,M
iddleName,LastName,Suffix,EmailPromotion,AdditionalContactInfo,rowguid,Modi
fiedDate")] Person person)
        {
            if (ModelState.IsValid)
            {
                db.Entry(person).State = EntityState.Modified;
                db.SaveChanges();
                return RedirectToAction("Index");
            }
            return View(person);
        }

        // GET: /Person/Delete/5
        public ActionResult Delete(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Person person = db.People.Find(id);
            if (person == null)
            {
                return HttpNotFound();
            }
            return View(person);
        }

        // POST: /Person/Delete/5
        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public ActionResult DeleteConfirmed(int id)
        {
            Person person = db.People.Find(id);
            db.People.Remove(person);
            db.SaveChanges();
            return RedirectToAction("Index");
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
```

```
            {
                db.Dispose();
            }
            base.Dispose(disposing);
        }
    }
```

Visual Basic

```
Imports System
Imports System.Collections.Generic
Imports System.Data
Imports System.Data.Entity
Imports System.Linq
Imports System.Net
Imports System.Web
Imports System.Web.Mvc

Namespace ScaffoldingDemo
    Public Class PersonController
        Inherits System.Web.Mvc.Controller

        Private db As New AdventureWorks2012Entities

        ' GET: /Person/
        Function Index() As ActionResult
            Return View(db.People.ToList())
        End Function

        ' GET: /Person/Details/5
        Function Details(ByVal id As Integer?) As ActionResult
            If IsNothing(id) Then
                Return New HttpStatusCodeResult(HttpStatusCode.BadRequest)
            End If
            Dim person As Person = db.People.Find(id)
            If IsNothing(person) Then
                Return HttpNotFound()
            End If
            Return View(person)
        End Function

        ' GET: /Person/Create
        Function Create() As ActionResult
            Return View()
        End Function

        ' POST: /Person/Create
```

```
        'To protect from overposting attacks, please enable the specific
properties you want to bind to, for
        'more details see http://go.microsoft.com/fwlink/?LinkId=317598.
        <HttpPost()>
        <ValidateAntiForgeryToken()>
        Function Create(<Bind(Include :=
"BusinessEntityID,PersonType,NameStyle,Title,FirstName,MiddleName,LastName,
Suffix,EmailPromotion,AdditionalContactInfo,rowguid,ModifiedDate")> ByVal
person As Person) As ActionResult
            If ModelState.IsValid Then
                db.People.Add(person)
                db.SaveChanges()
                Return RedirectToAction("Index")
            End If
            Return View(person)
        End Function

        ' GET: /Person/Edit/5
        Function Edit(ByVal id As Integer?) As ActionResult
            If IsNothing(id) Then
                Return New HttpStatusCodeResult(HttpStatusCode.BadRequest)
            End If
            Dim person As Person = db.People.Find(id)
            If IsNothing(person) Then
                Return HttpNotFound()
            End If
            Return View(person)
        End Function

        ' POST: /Person/Edit/5
        'To protect from overposting attacks, please enable the specific
properties you want to bind to, for
        'more details see http://go.microsoft.com/fwlink/?LinkId=317598.
        <HttpPost()>
        <ValidateAntiForgeryToken()>
        Function Edit(<Bind(Include :=
"BusinessEntityID,PersonType,NameStyle,Title,FirstName,MiddleName,LastName,
Suffix,EmailPromotion,AdditionalContactInfo,rowguid,ModifiedDate")> ByVal
person As Person) As ActionResult
            If ModelState.IsValid Then
                db.Entry(person).State = EntityState.Modified
                db.SaveChanges()
                Return RedirectToAction("Index")
            End If
            Return View(person)
        End Function

        ' GET: /Person/Delete/5
        Function Delete(ByVal id As Integer?) As ActionResult
            If IsNothing(id) Then
```

```
                Return New HttpStatusCodeResult(HttpStatusCode.BadRequest)
            End If
            Dim person As Person = db.People.Find(id)
            If IsNothing(person) Then
                Return HttpNotFound()
            End If
            Return View(person)
        End Function

        ' POST: /Person/Delete/5
        <HttpPost()>
        <ActionName("Delete")>
        <ValidateAntiForgeryToken()>
        Function DeleteConfirmed(ByVal id As Integer) As ActionResult
            Dim person As Person = db.People.Find(id)
            db.People.Remove(person)
            db.SaveChanges()
            Return RedirectToAction("Index")
        End Function

        Protected Overrides Sub Dispose(ByVal disposing As Boolean)
            If (disposing) Then
                db.Dispose()
            End If
            MyBase.Dispose(disposing)
        End Sub
    End Class
End Namespace
```

Although lengthy, the code is not difficult; you have a number of methods responsible for C.R.U.D. operations, such as **`Create`**, **`Edit`**, **`Details`**, **`Delete`**, and **`Index`**. The latter returns the full list of records in the table mapped by the model class. Every method returns an object of type **`ActionResult`**, which is exposed by the MVC framework (remember you are using Web Forms here). It encapsulates the result of the aforementioned methods and is used to perform framework-level operations on behalf of the method. Of course, you can make additional edits to the controller class if you wish. In this particular example, it is a good idea to restrict the number of people returned by the **`Index`** method, in order to speed up the process. Imagine you want to retrieve only the first ten people in the table. You can edit the **`Index`** method as follows:

Visual C#

```
        // GET: /Person/
        public ActionResult Index()
        {
            return View(db.People.Take(10).ToList());
        }
```

Visual Basic

```
        ' GET: /Person/
        Function Index() As ActionResult
            Return View(db.People.Take(10).ToList())
        End Function
```

You can use any LINQ operator to edit the query in a way that best fits your needs. In this case, the code uses **Take** to retrieve the first 10 records. It is worth mentioning that for each method in the code you will find some comments that explain how to invoke the related page in the browser. For instance, if you want to view the list of records in the table, you will use the /Person relative URL (see the code in the previous listing). You will get a demonstration shortly.

Test the application

Press **F5** to start debugging the application. Your default web browser will open and show the default page of the application. In the address bar, type the /Person relative URL at the end (see the highlight in Figure 40). The application will show the first ten people in the table at this point, as you can see in Figure 40.

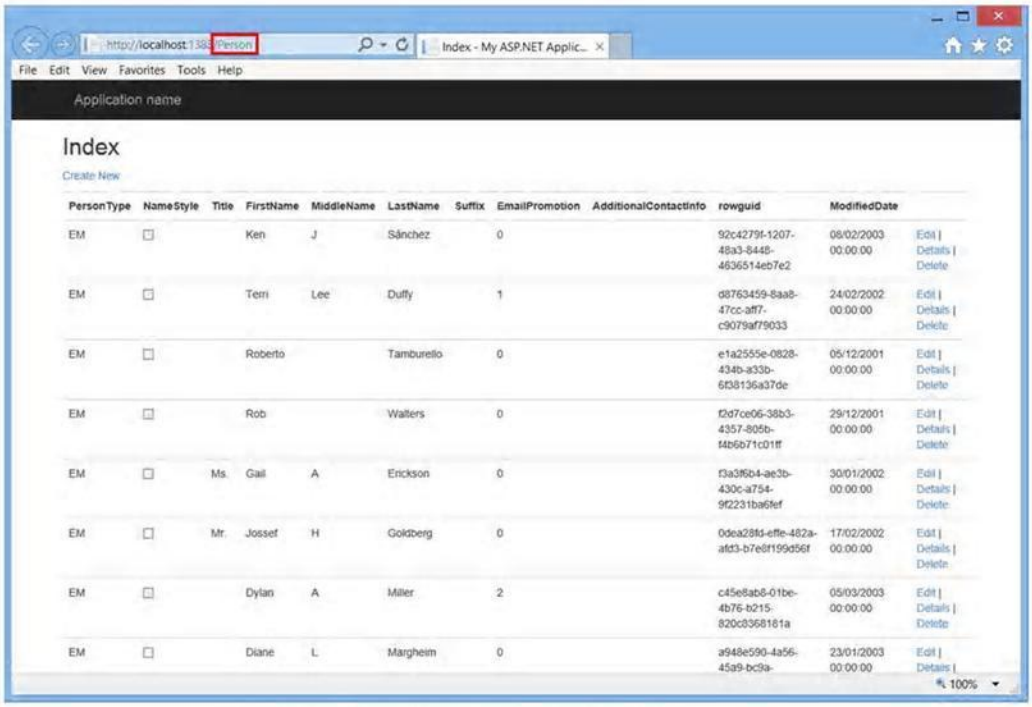


Figure 40: The application shows a list of items.

Notice the shortcuts to pages for data operations. For example, you can click **Edit** to see and change details of an item. This is also useful to understand how the application invokes the appropriate methods in the controller. Take a look at Figure 41, which shows how to edit an existing item.

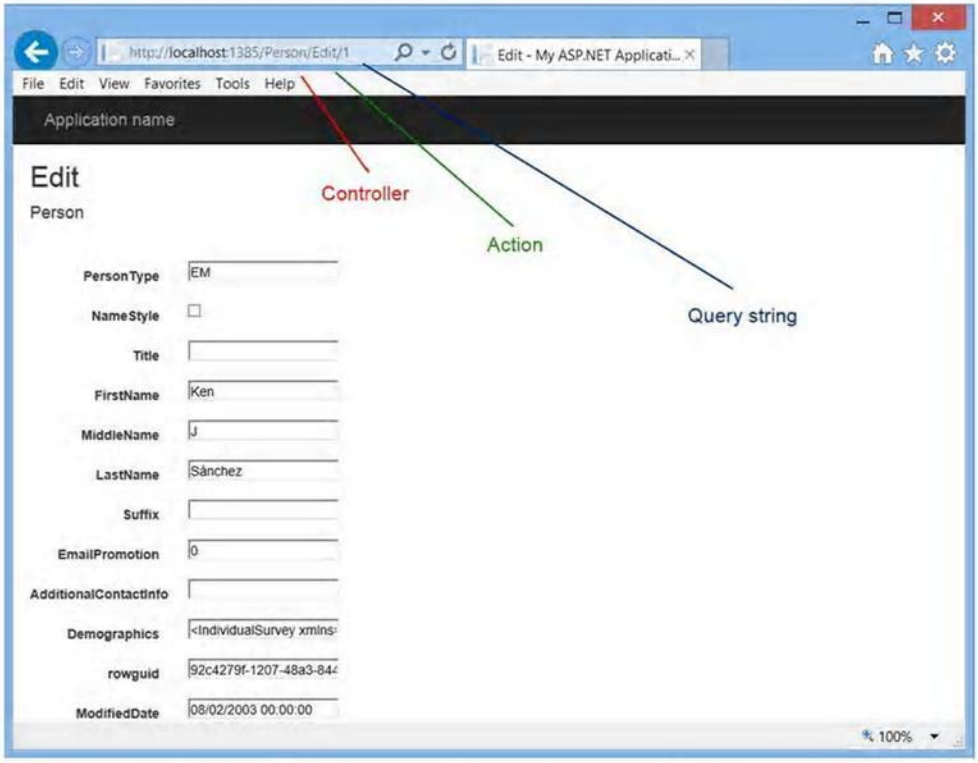


Figure 41: Editing an Item and the Anatomy of the Address

Apart from seeing how easy it is to edit an item without writing any code, notice look at the address bar. The address is made of the following elements:

- The web address for the application
- The name of the controller (in this case Person)
- The name of the method in the controller (in this case Edit)
- The value for the query string that will be used by the invoked method to retrieve a specific item

You can go back to the previous page and select **Create New** to see how easy it is to add a new person to the database (see Figure 42).

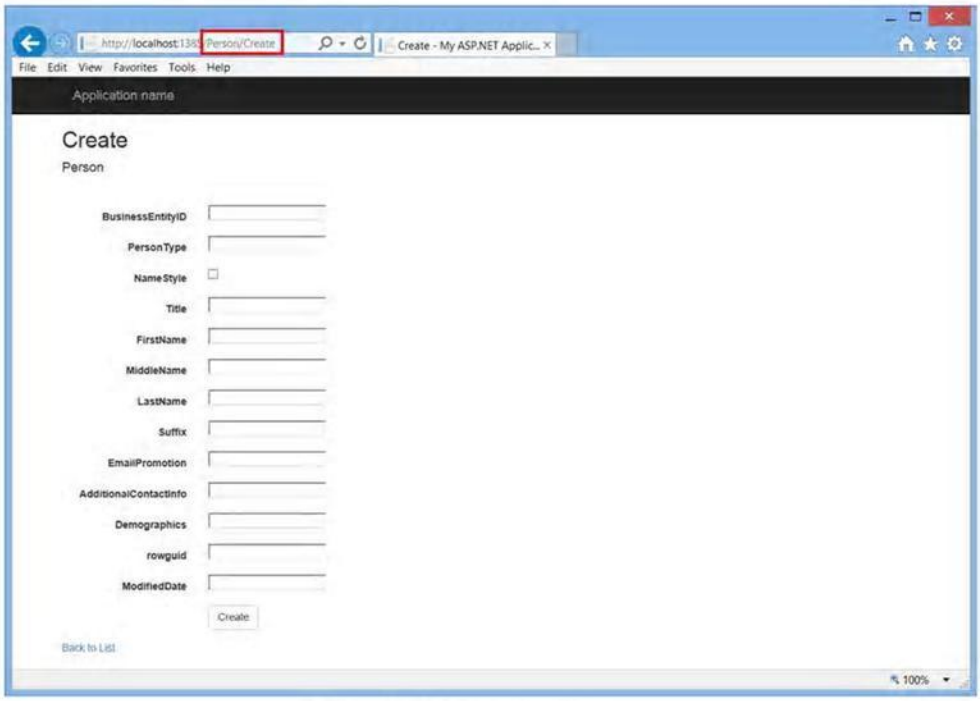


Figure 42: Adding a New Item

Figure 42 also highlights how both the controller name and the method name are used in the address. So you have seen how to take advantage of scaffolding in a Web Forms application due to the new One ASP.NET experience. Pages generated by Visual Studio can be edited to provide a different layout, but it is evident how with a minimal effort you can create powerful data-centric applications.

Browsers Link Dashboard

You already know that with Visual Studio you can test your web applications with different browsers. In Visual Studio 2013, if you have your application running in different browsers and you make changes in the IDE, such changes can be refreshed to every browser with a single click. This is possible because Visual Studio 2013 and the .NET Framework 4.5.1 use SignalR 2.0, the popular library that allows sending real-time notifications.

To understand how it works, first create a new ASP.NET project called BrowserLinkDemo. By following the lesson learned in the previous section, select the **Web Forms** template and the anonymous authentication. Of course, this feature works not only with Web Forms, but also with ASP.NET MVC.

Note: Do not delete the new project until you complete this chapter. It will be used again later when discussing the new tools for Windows Azure.

The next step is telling Visual Studio to use multiple browsers to run the application, so if you did not do this before, select the arrow near the Start button on the toolbar (see Figure 43), then **Browse With**.

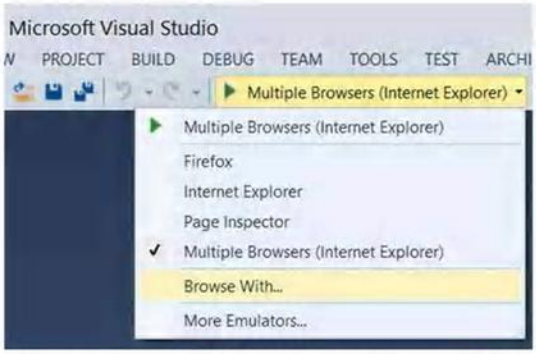


Figure 43: Accessing the Command to Change the Default Browser(s) for Testing

In the Browse With dialog, select two or more web browsers. On my machine, I have Internet Explorer and FireFox installed, so my selection includes both (see Figure 44).

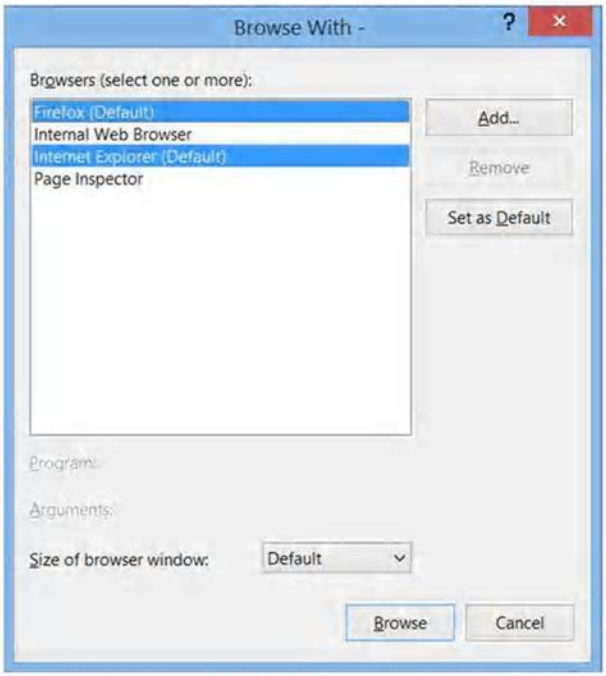


Figure 44: Selecting Two or More Web Browsers for Testing

Press **CTRL** and click on each browser you want to use, then select **Set as Default** and close the dialog. Now, instead of debugging the usual way with F5, press **CTRL + F5** to start without debugging. This way, the application will be launched in all of the browsers you selected. Figure 45 shows the application running inside Internet Explorer.

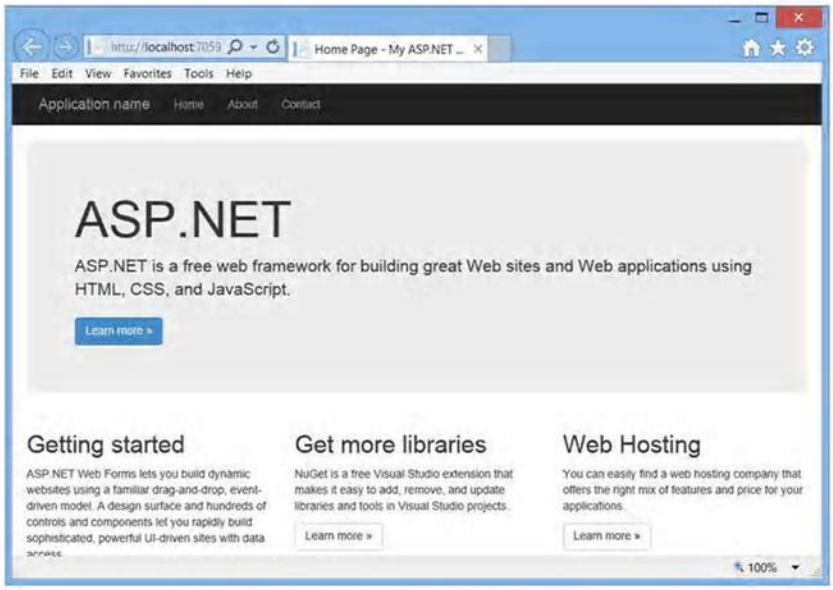


Figure 45: The Sample Application Running

Now let's make a very simple edit in Visual Studio. Open the Default.aspx page in the designer and replace the ASP.NET string with **Visual Studio 2013 Succinctly**. On the **Standard** toolbar, click the **Refresh** button, which you can see highlighted in Figure 46, or press **CTRL+ALT+Enter**.

Figure 46: The Refresh Button for Linked Browsers

Now switch back to both browsers; you will see how they show the updated string. With this technique, you do not need to stop the application, make your edits, and restart debugging. By clicking the arrow near the Refresh button, you can access additional shortcuts, including the one to enable the Browser Link Dashboard tool window. With the Browser Link Dashboard you can see connections for each application in the solution and you can take specific action for each linked browser, such as refreshing only one instead. Figure 47 shows what the Browser Link Dashboard looks like against a solution containing both sample projects described in this chapter.

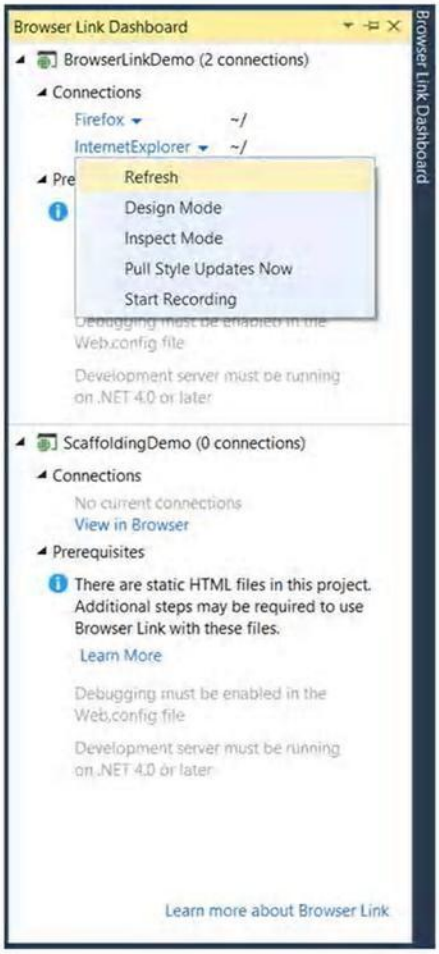


Figure 47: The Browser Link Dashboard allows managing actions for single connections.

With Scaffolding and Browsers Link, you have seen two relevant features in Visual Studio 2013. But this new release has a great focus on the cloud. This is discussed in the next section.

What's new in Windows Azure

Windows Azure is the popular cloud computing platform from Microsoft, first unveiled at the Professional Developer Conference (PDC) in 2008. Over the years, Windows Azure has dramatically evolved by introducing tons of services, and the cloud has become an important part of our daily lives. Many websites, mobile devices, and applications use Windows Azure's services. If you've ever developed applications for Windows Azure before, you know that you had to do most things outside of Visual Studio, using the Windows Azure Developer Portal on the web or special client applications; in fact, Visual Studio lacked a good integration with the platform. As for other platforms, Visual Studio 2013 solves the problem and provides deep integration with Windows Azure making it easy to manage many services from within the IDE. This chapter provides guidance on how Visual Studio 2013 integrates with Azure and on how you can leverage this integration to build applications faster.

What you need before reading this section

Because we focus on the new tooling in Visual Studio 2013 for Windows Azure, we assume you already have at least a basic knowledge of the platform, including information about paid services and pricing. In fact, this chapter can neither summarize all services offered by Windows Azure nor it can explain programming for Azure, since this would require an entire book. If you need an overview of programming for Windows Azure before reading this chapter, refer to the official documentation available at http://www.windowsazure.com/en-us/documentation/ . In order to complete the steps described in this chapter, you must install the Windows Azure SDK 2.2 or later.

Note: This chapter describes Windows Azure from the developer's perspective and will describe services that you can access only if you have a current subscription. But Windows Azure is a paid service. Fortunately, you can enable a 30-day trial at http://www.windowsazure.com/en-us/pricing/free-trial/. If you want to fully understand the rest of this chapter, you are encouraged to subscribe the trial.

Server Explorer window

You've probably already used the Server Explorer tool window in Visual Studio many times, for different purposes such as managing connections to databases, or web servers. In Visual Studio 2013, Server Explorer offers a new node called Windows Azure, which allows connecting to your subscription and managing services from within the IDE. Figure 48 shows how the Windows Azure node looks.

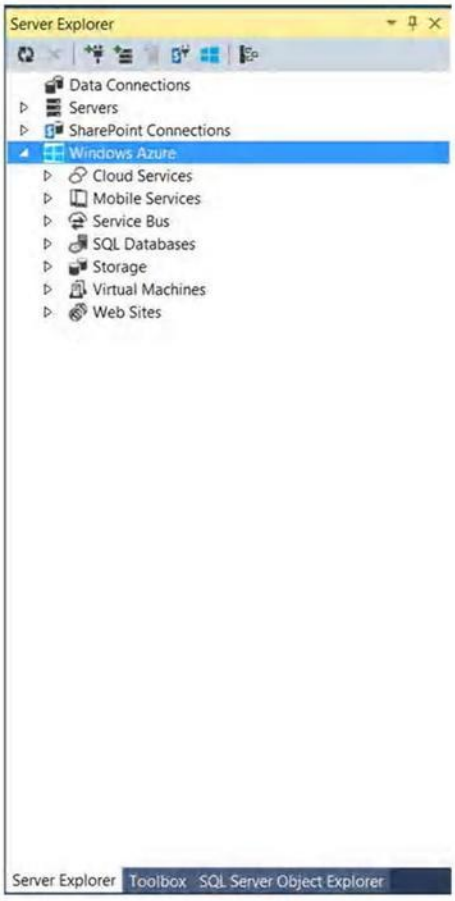


Figure 48: The New Windows Azure Tooling in the Server Explorer Window

As you can see, you can manage most services without leaving Visual Studio. The first thing you need to do is associate a valid Windows Azure subscription to Visual Studio 2013. To do so, right-click **Windows Azure** and select **Connect to Windows Azure**. You will be asked to enter the Microsoft Account of your current subscription. If you have multiple subscriptions associated with your Microsoft Account, you will be able to manage subscriptions. Simply right-click again **Windows Azure** and select **Manage Subscriptions**. At this point, the Manage Windows Azure Subscriptions appears, as shown in Figure 49.

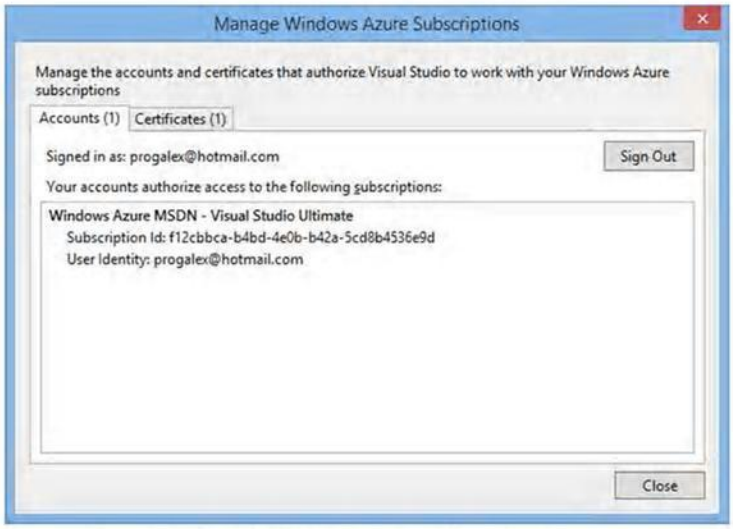


Figure 49: Managing Windows Azure Subscriptions

You will need to import your subscription settings in Visual Studio 2013 at this point. Click **Certificates**, then **Import**. In the appearing Import Windows Azure Subscriptions dialog (see Figure 50), click the **Download subscription file** hyperlink.

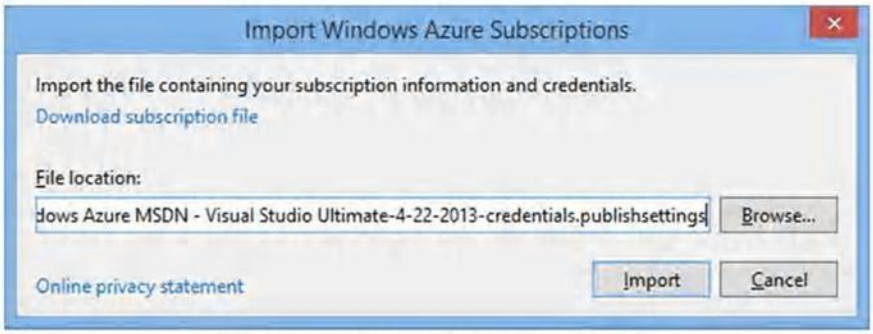


Figure 50: Managing Windows Azure Subscriptions

At this point, a page in the Windows Azure's website will be opened and a file with your subscription's settings will be automatically available for downloading. Once you've downloaded the settings file, click **Browse** and in the dialog search for the downloaded settings file, select it, and finally click **Import**. By following these steps, your subscription will be added to the list of subscriptions in Visual Studio. If you have added multiple subscriptions, in the Manage Windows Azure Subscriptions dialog you will be able to select the subscription you want to work with. Click **Close** when you have made your selection. Let's now see in more detail what you can do with Server Explorer.

Integration with mobile services

Mobile Services can be used as a backend for mobile applications. A very common use of Mobile Services is storing data inside tables. In Visual Studio 2013 you can now create a mobile service directly, add tables, and see logs for the service. To create a new service, right-click **Mobile Services**, and then select **Create Service**. In the Create Mobile Service dialog you will have to specify the new service's details, as shown in Figure 51.

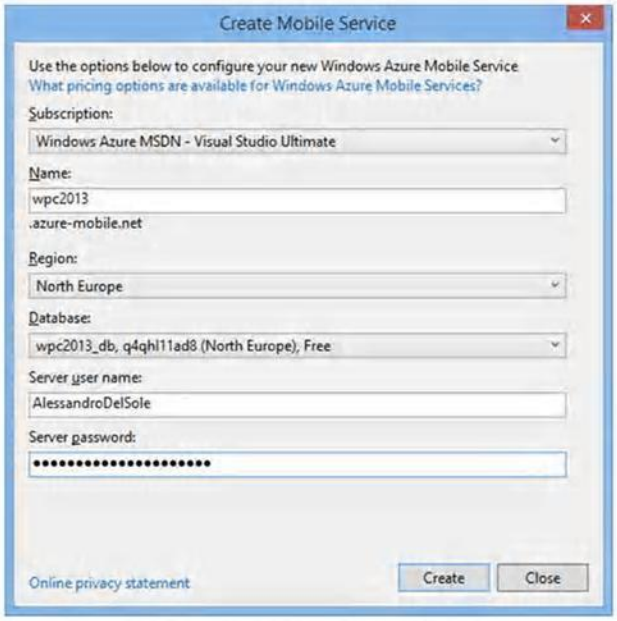


Figure 51: Creating a New Mobile Service

You will need to specify the following information:

- The subscription where the service will be created. You can leave the selection unchanged if you want the service to be created in the current subscription.
- The service name. Visual Studio 2013 will check the availability of the service name as you type.
- The Region. Remember to select the region that is nearest to your location.
- A new or existing SQL Azure database to be associated with the new service (optional). If you did not create a SQL Azure database in the Management Portal before, you can use the **Create a free SQL database** option or the **Create a billed SQL database**. Of course, I strongly recommend to create a free SQL Azure database (up to 20 MB) rather than a billed one.
- User name and password for the SQL Azure server in order to access the database.

Note: I'm assuming you have already configured a SQL Azure server, since I'm talking about server user name and password required to access a database on the cloud. If you

did not configure your SQL Azure server, you can get started by reading the official documentation.

When you're finished, click **Create**. The new service will appear under the Mobile Services node of Server Explorer. You can also add tables to the service directly. Right-click the service's name and select **Create Table**. In the Create Table dialog (see Figure 52) you can specify the table name and permission for each of the C.R.U.D. operations.

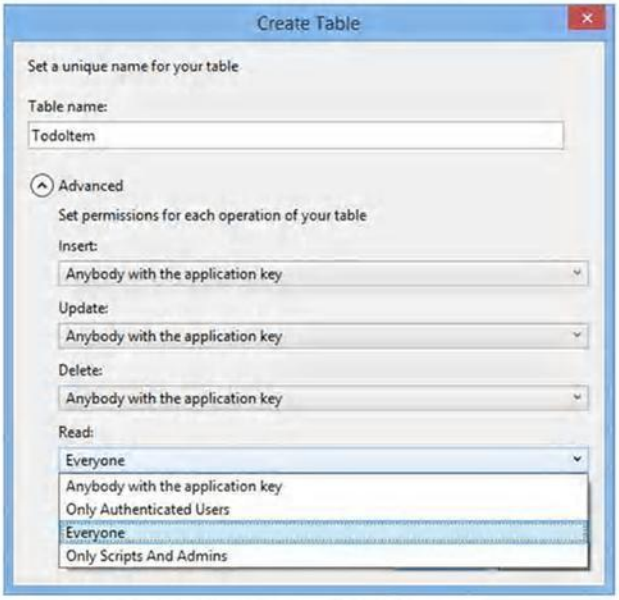


Figure 52: Creating a New Mobile Service

Now create a new table called **TodoItem**. We will use this table in the next chapter when demonstrating Windows 8 support for mobile services. Notice how you have deep control over permissions; you can allow everyone, authenticated users, administrators, or users having the application key (managing the application key is only available in the management portal). Click **Create** when you're ready. You will be able to see the new table as a node under the mobile service's name. Also, if you expand the table name, you will see a number of JavaScript code files, which are responsible of performing operations against data, such as read, insert, update, and delete. Figure 53 shows how the Server Explorer window appears at this point.

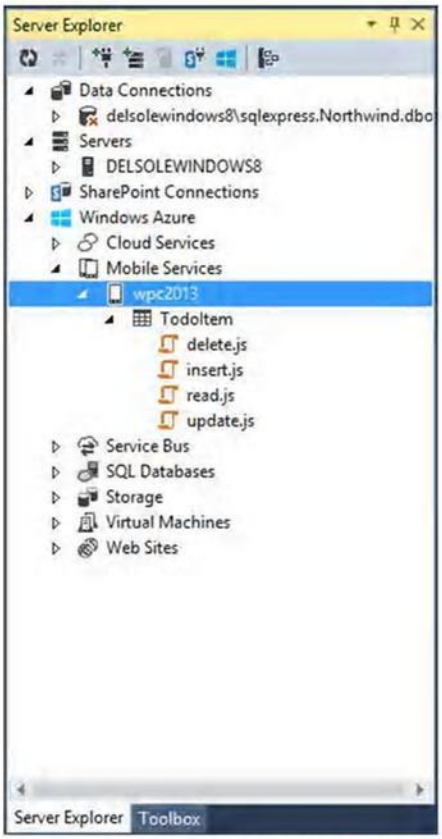


Figure 53: Mobile service, table, and JavaScript files as they appear in Server Explorer.

With only a few steps, you have created a mobile service that can be used as a backend in your mobile applications. You can use mobile services in the following applications:

- Windows Store apps for Windows 8.x
- Windows Phone apps
- ASP.NET web applications

You will need to right-click the project name in Solution Explorer and select **Add Connected Service**. This is discussed further in Chapter 7.

Integration with Azure websites and cloud services

In Windows Azure, you publish your web applications through websites or cloud services. A website is a simplified, pre-configured environment for easy deployment. A cloud service, instead, is a highly customizable environment that requires you to make some configurations before deployment, and where you can take a lot of actions over the system. You can find the full description of both environments in this page of the Windows Azure documentation (I recommend that you read it). Whatever your choice is, Visual Studio 2013 supports publishing applications to both web sites and cloud services.

Note: You can connect to Virtual Machines from Server Explorer, but you can only publish to websites and cloud services directly.

For instance, imagine you want to deploy the sample application created earlier in this chapter to demonstrate the Browser Link feature. In Server Explorer, right-click the **Web Sites** node and then select **Add New Site**. A new dialog called Create site on Windows Azure appears. Here you will have to specify details for the new website, as shown in Figure 54.

Figure 54: Creating a New Website

The following information is required:

- The site name, an identifier that will be used to construct the website's URL. Visual Studio will check for the availability of the name as you type.
- The location: Microsoft has several data centers across the world, so ensure you select the location nearest to you.
- Database server, database username, and database password. These are optional fields; you only need to supply them if your application will use an SQL Azure database.

Tip: The sample application does not use any database. For the sake of completeness, Figure 54 shows how to fill database-related fields.

Once you have entered all the required information, click **Create**. Once the website has been created, double-click its name in Server Explorer so that Visual Studio shows a configuration window (see Figure 55).

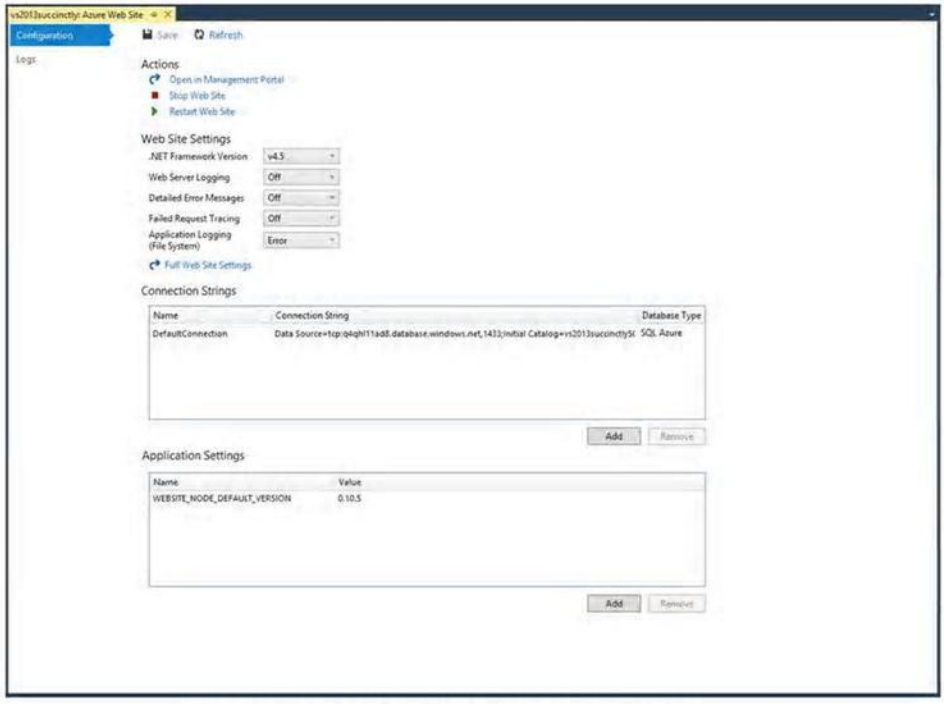


Figure 55: Configuration Window for the New Website

The window is divided into four main areas:

- Actions: here you find shortcuts to common tasks such as opening the site in the Azure's management portal, stopping or restarting the site.
- Web Site Settings: here you can fine-tune the configuration for your website, including error management and tracing.
- Connection Strings: here you can manage connection strings that your application uses to connect to data sources.
- Application Settings: this allows listing and adding environment variables for your websites. You should never change or remove settings added by Visual Studio.

Now you want to publish the BrowserLinkDemo sample application to the newly created website. First, right-click the project name in Solution Explorer, then select Convert, Convert to

Windows Azure Cloud Service Project. This action will add a new project with a Windows Azure role to the solution. By default, the new project's name has the same name of the original project plus the .Azure suffix, in our example it is BrowserLinkDemo.Azure. Now, in Solution Explorer right-click the original project (BrowserLinkDemo) and then click Publish.

Tip: Use the Publish command if you want to deploy your application to a website. Because this is our current case, we are using this option. If you want to deploy to a Cloud Service instead, right-click the project name and select Publish to Windows Azure. A specific dialog will allow creating a new Cloud Service and easily deploy the application.

The Publish Web dialog appears and has no preconfigured settings, so you need to click the **Import** button to specify the target website. Visual Studio shows the Import Publish Settings dialog at this point, which is visible in Figure 55.

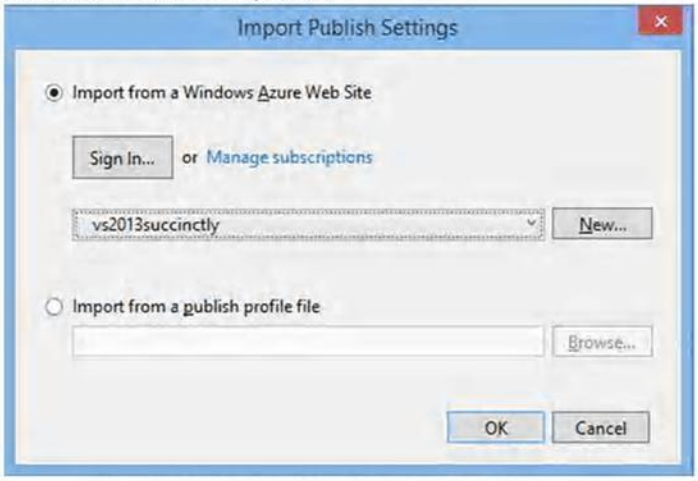


Figure 56: Selecting the Website for Publishing

Since the IDE is still connected to your Azure subscription, you can simply select the target website from the first combo box. You can also select a different subscription by clicking **Sign In** or by importing an existing publish profile file.

Tip: Visual Studio 2013 makes it easy to download and publish profiles from your Windows Azure subscription. Just right-click a website in Server Explorer and then click Download Publish Profile.

When you're ready, click **OK**. The Profile form in the Publish Web dialog will be filled with information coming from the publish profile you just selected. Go ahead to the **Connection** form (see Figure 57).

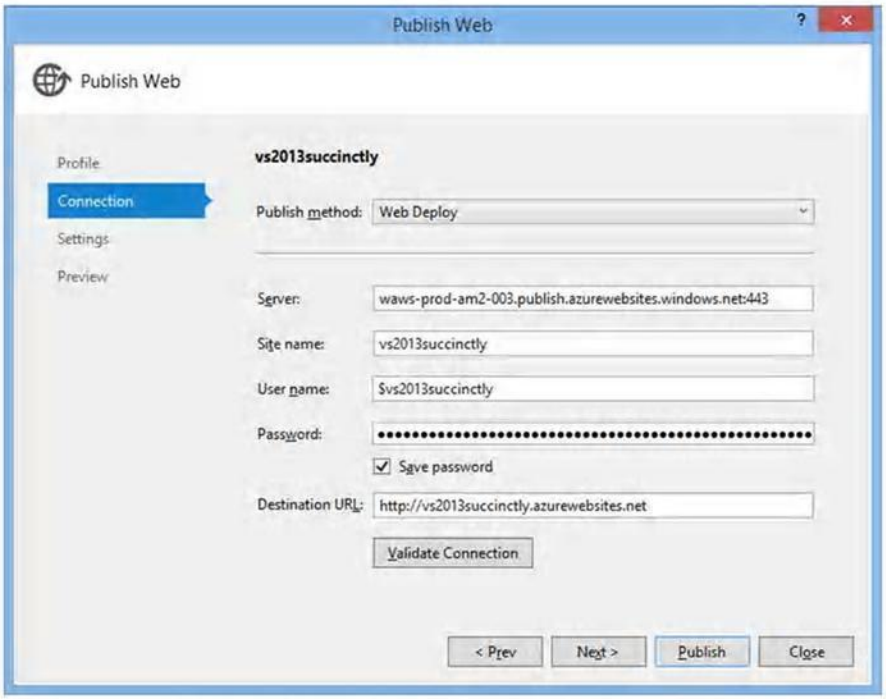


Figure 57: Connection Settings for the Application

Visual Studio 2013 automatically provides information for this form so you do not need to change any field. Notice the content of the Destination URL field, which contains the web address for your application once published. Click **Next** to access the Settings form (see Figure 58).

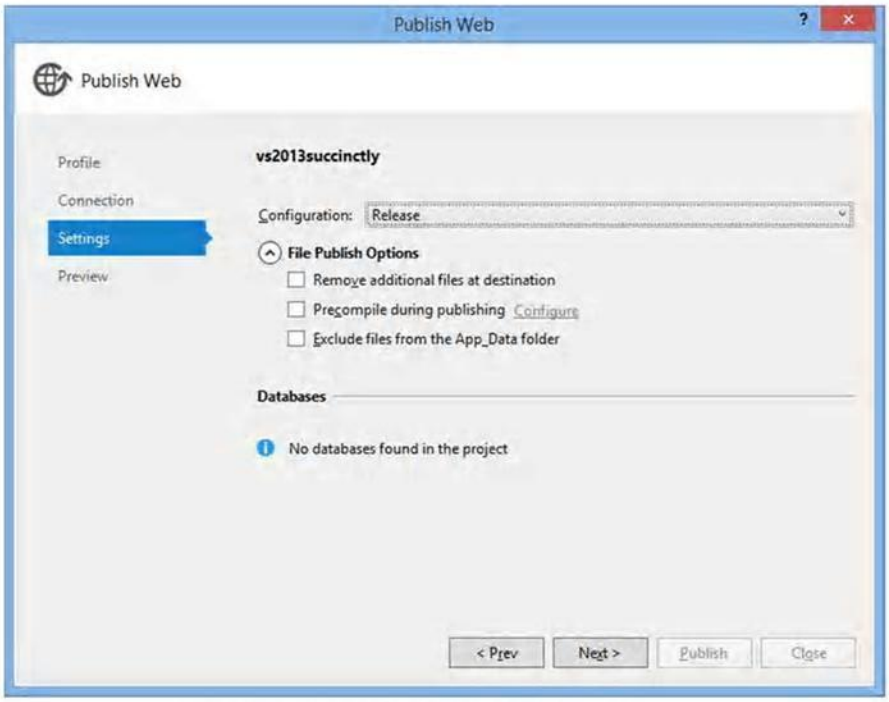


Figure 58: Publish Settings for the Application

In this form of the Publish Web dialog you can change the configuration between **Release** (default) and **Debug**, or decide if you want to remove additional files at destination, precompile managed code during publishing, or exclude files from the App_Data folder. If you are also using a database, here you will be able to set a default connection string. When you click **Next**, you access the Preview form where you can optionally view the full list of files that will be published to the websites, by clicking **Start Preview**. The result of pressing this button is shown in Figure 59.

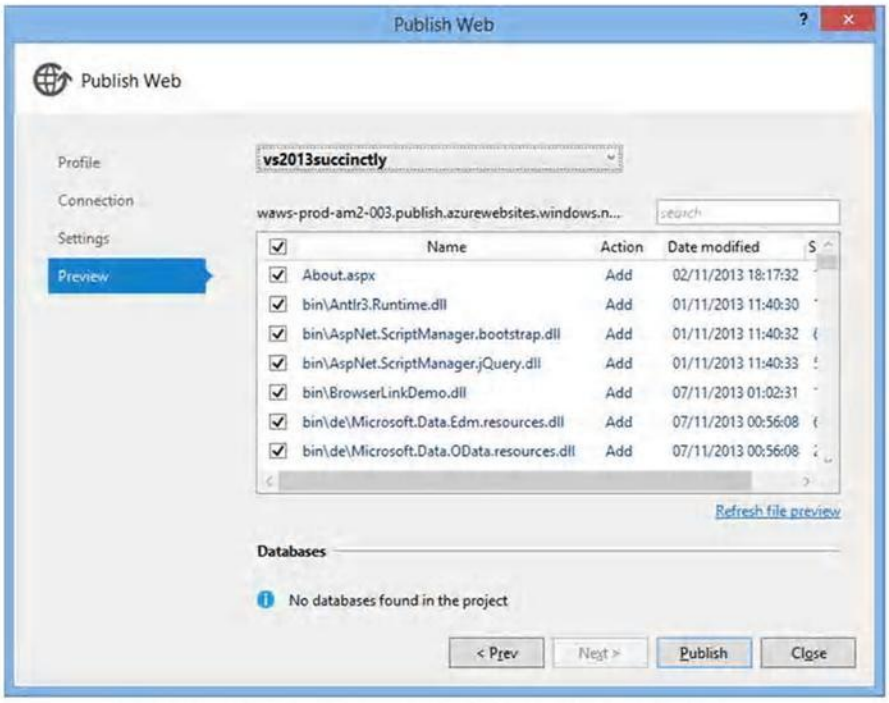


Figure 59: The List of Files that Will Be Published to the Website

You can finally click **Publish**. The progress of publishing will be shown in the Output window and the speed may vary depending on your Internet connection. When the application has been published and is online, you will see a message in the Output window and you will be able to start the application by using the appropriate web address. Figure 60 shows the sample application running on the website created previously.

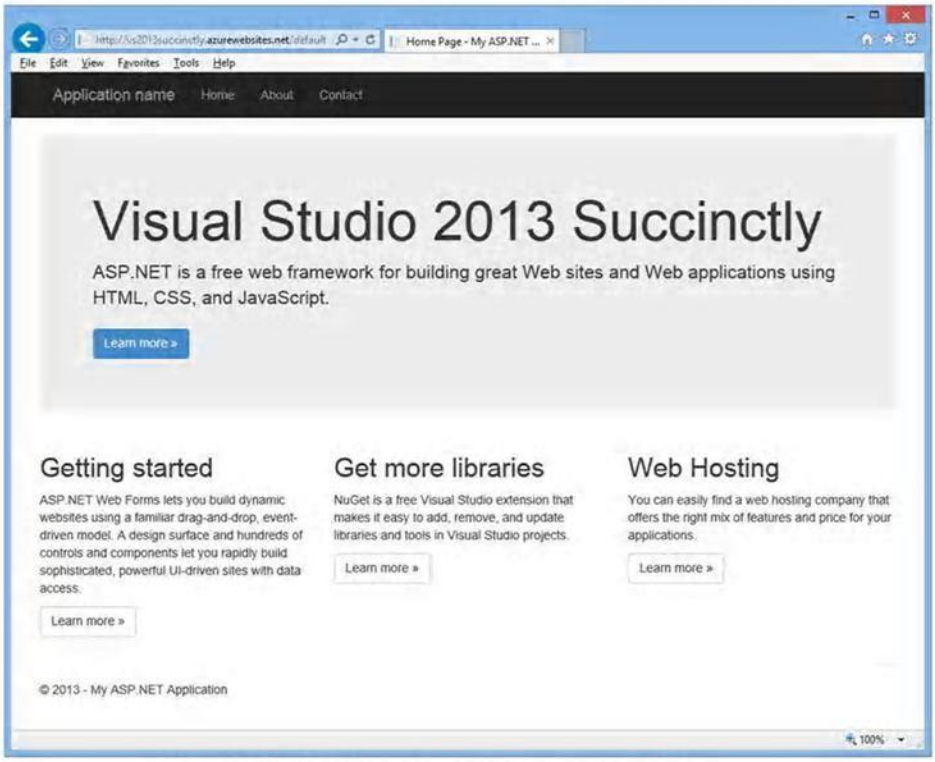


Figure 60: The Sample Application Running on the Website

As you can see, with a very few steps you have been able to publish an ASP.NET application to Windows Azure, without the need of having an in-house server and hosting infrastructure.

Note: If you create websites for testing purposes like in the current scenario, remember to delete the site when finished, even if you have the spending limit enabled. Websites consume resources even if they are offline, so the only way to ensure they do not consume any unneeded resources to delete them.

Integration with Azure Storage

Note: For the rest of Azure services described in this chapter, we focus on the new tools in the IDE. If you need guidance on how to access resources in the Azure Storage from your applications, make sure you read the appropriate documentation.

The Windows Azure Storage is the place where you can store data and files for your cloud-based applications. For instance, if your application needs to show some pictures or allows downloading files, these will be first saved onto the Storage and then accessed via their web address (HTTP). The Windows Azure Storage provides the following services:

- Blob Storage: here you can upload unstructured binary data, such as files.

- Queue Storage: this is used to save and retrieve messages for workflows and communications.
- Table Storage: here you can store non-relational and unschematized data with support for queries.

The Blob Storage can also recognize VHD files and allows mounting these files as virtual hard disks on the cloud. Visual Studio 2013 finally provides integration with the Windows Azure Storage from within the IDE. This is a tremendous benefit and a significant step forward, because before Visual Studio 2013, the only way to upload to or manage information on the Storage was by using 3rd party client applications. Now you can definitely use Visual Studio to perform operations such as uploading files to the Blob Storage. Any Windows Azure subscription supports multiple storages. For each storage, you need to create a storage account. Since any transaction (download, upload, login) has a cost, Microsoft offers the Development Account, which consists of the Windows Azure Emulator and other components that allow simulating a production environment on your development machine. This way, you can freely test transactions locally, and move to the cloud only when you are ready to go to production. Remember to check the MSDN documentation for the storage programmability and for understanding how to access information on the storage from your applications.

Creating a storage account

In Server Explorer, expand the **Storage** node under Windows Azure. The local development storage will be shown by default. If it's not already running, Visual Studio will start the Windows Azure Storage Emulator. The development storage works exactly like an online storage; however, it is a good idea to see how to work with an online account. Unfortunately, you cannot create new storage accounts from Visual Studio; instead, you can connect to existing accounts. So, open the Windows Azure Management Portal in your favorite web browser. Once logged in, click **Storage** in the dashboard on the left. Figure 61 shows how the management portal appears at this point, with no accounts available yet.

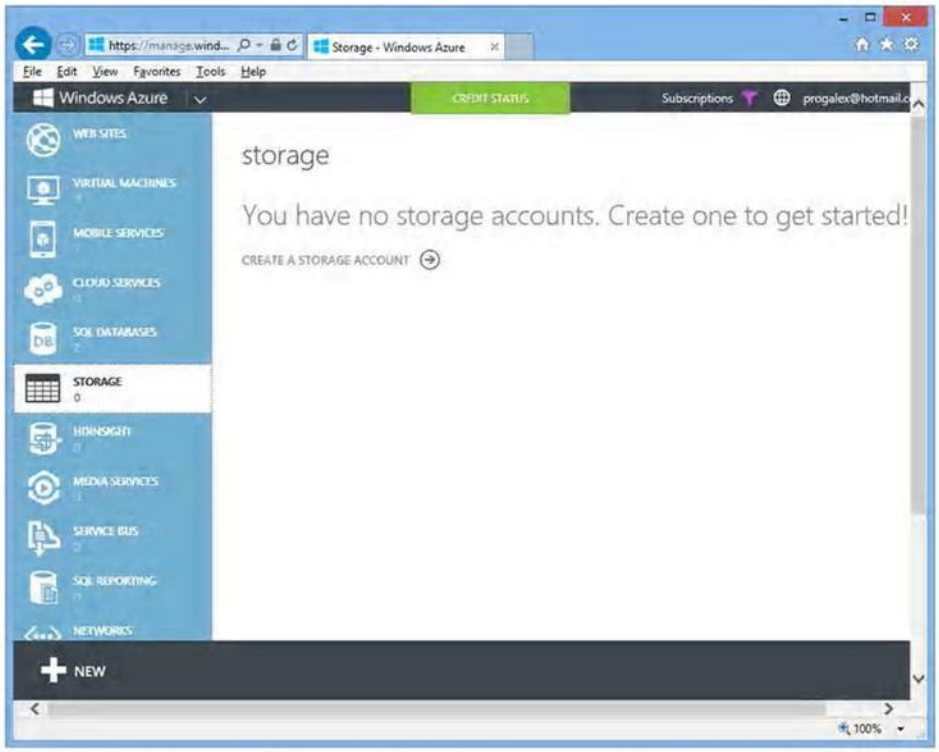


Figure 61: The Windows Azure subscription has no storages at the moment.

Click **CREATE A STORAGE ACCOUNT**. The Management Portal will open a new page where you can easily create a new storage account by supplying the account name and the location. Figure 62 shows how the management portal appears at this point.

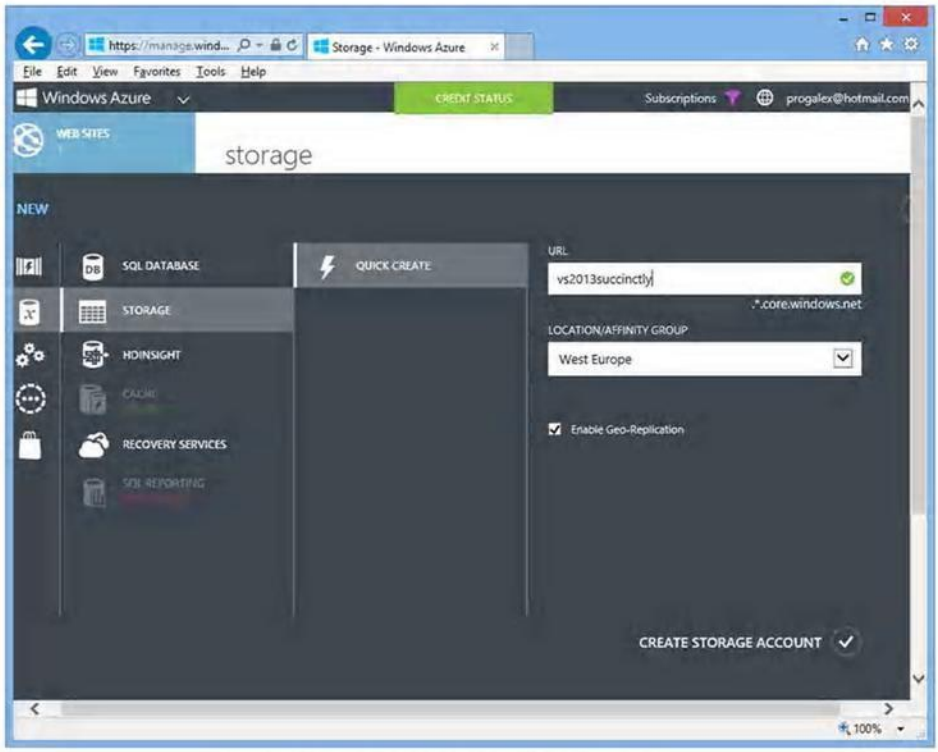


Figure 62: Creating a New Storage Account

Always remember to select the nearest location to where you live. As you might know, the account name becomes the prefix of the URL for services exposed by the storage. Table 2 shows URLs for each service (where storagename is the name you entered when creating the storage account).

Table 2: Members and settings for the Add Controller dialog

Storage	URL
Blob Storage	http://storagename.blob.core.windows.net
Table Storage	http://storagename.table.core.windows.net
Queue Storage	http://storagename.queue.core.windows.net

Addresses shown in Table 2 are very important, since they are the way you access each storage. Click **CREATE STORAGE ACCOUNT**. After about one minute, you will be able to see the storage account online in the Management Portal. Now close the management portal and go back to Visual Studio 2013. In Server Explorer, right-click the **Storage** node and then click **Refresh**. You will now see the newly created storage account (see Figure 63).

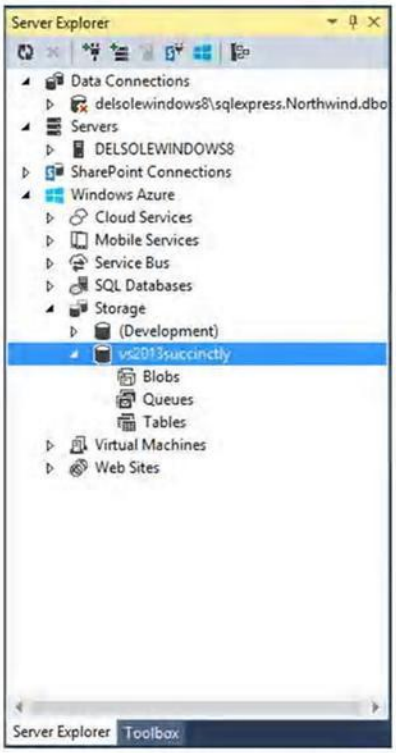


Figure 63: The new Storage Account is visible inside Server Explorer.

Now you will see how to interact with the storage.

Managing the Blob Storage

Say you want to upload an image file to the Blob Storage. Blobs are organized within folders called containers, so you first need to create a container. Follow these steps:

1. Click the Storage Account of your interest (in this case, vs2013succinctly) and expand it in order to make the Blobs node visible.
2. Select **Create Blob Container**.
3. In the Create Blob Container dialog (see Figure 64), specify the name of your new container all lower case. For example, enter **pictures**.
4. The new container will be visible in Server Explorer. Right-click it and select **View Blob Container**. You will see a new window called Container whose purpose is showing and

filtering the list of blobs in a container. It also allows uploading and/or deleting blobs (see Figure 65).

5. Click the **Upload Blob** button (the one with the black arrow). In the file selection dialog, select an image file (possibly of type Jpeg, in order to save space and time due to the lower size of this format). Visual Studio will start uploading your file to the storage, showing the progress of the operation. As you can see from Figure 66, the window shows metadata information for the file and, most importantly, the URL to access it.

Figure 64: Creating a New Blob Container

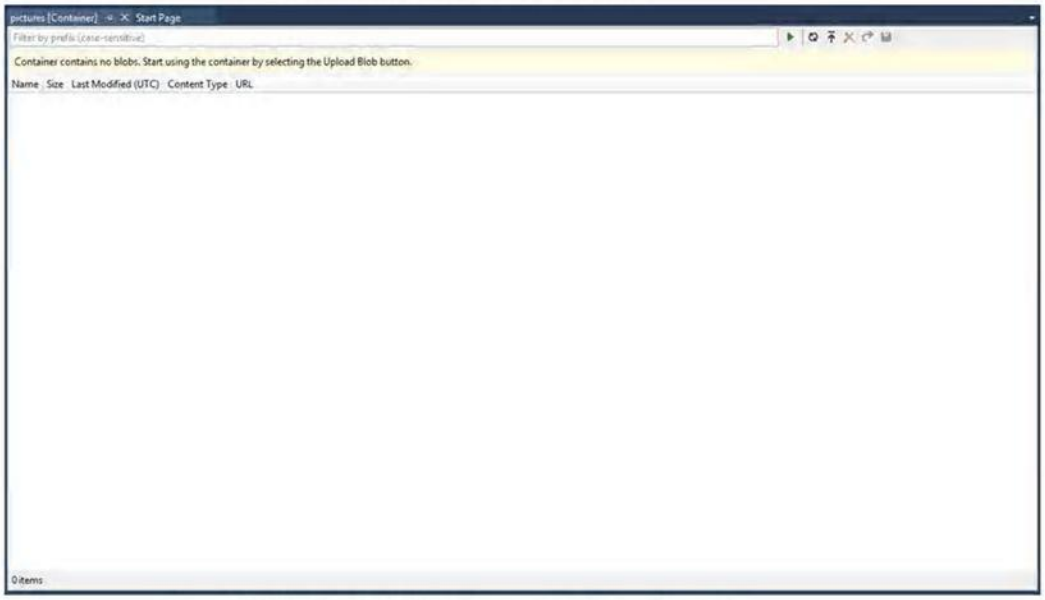


Figure 65: The list of blobs for the current container is empty.

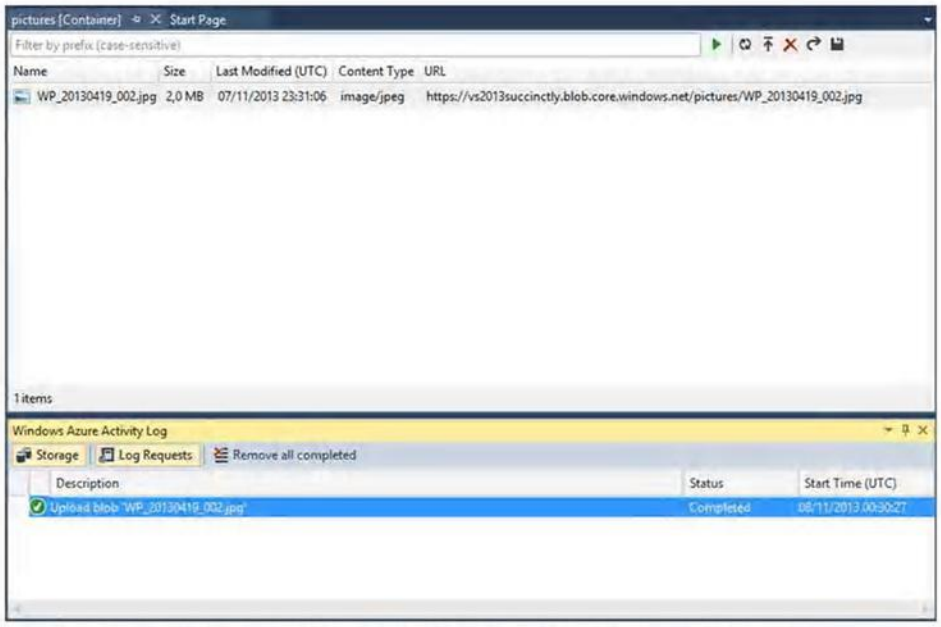


Figure 66: The blob has been uploaded and you can see its metadata and URL.

If you right-click the blob, you will get a popup menu showing a list of interesting commands. You can download the blob to disk (Save As), open the blob (Open), delete the blob (Delete), view more detailed properties in the Properties window (Properties), and copy the blob's URL to the clipboard (Copy URL). In order to access the blob in code, you will need to supply a connection string and credentials for your Windows Azure subscription. Writing code to accomplish this is beyond the scope of this chapter, so read this page in the Windows Azure documentation for this purpose. What you really need to understand is how you can upload the blob (and how you can manage blobs and containers) with the new tooling in Visual Studio 2013, without the need of 3rd party client applications.

Create and query tables

The Windows Azure storage supports creating tables to store nonrelational data. Right-click **Tables** in your storage account and select **Create Table**. In the **Create Table** dialog (see Figure 67), enter the name of your table, for example **mydata**.

Figure 67: Creating a New Table

When the new table is created, double-click it: you will see a window similar to the one you saw for containers. The window's toolbar (see Figure 68) has a button called Add Entity, represented by three drawers and a green addition symbol.

Figure 68: The Table's Designer Toolbar

Click **Add Entity** to enter new data. Figure 69 shows how you can enter data for the new entity.

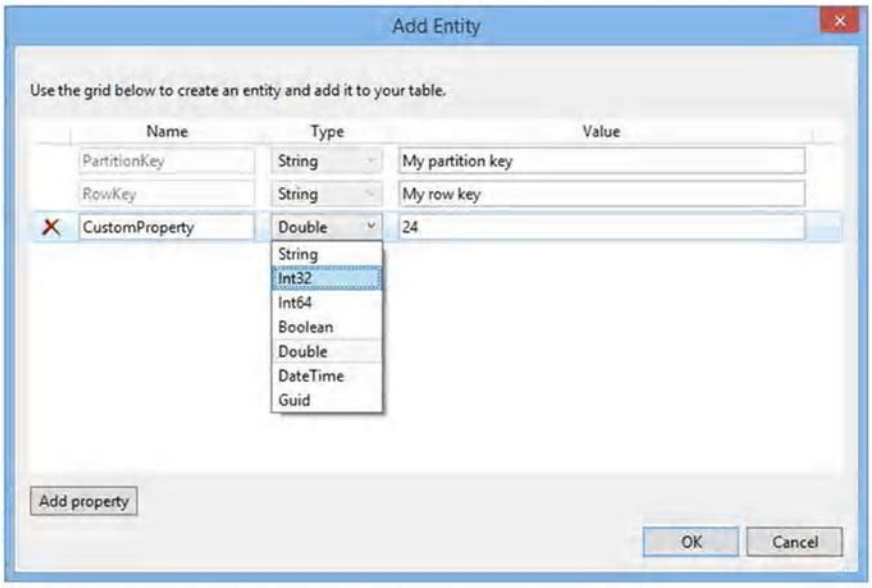


Figure 69: Entering Data for the New Entity and Property Customization

Every entity has some predefined properties, such as PartitionKey and RowKey, both of type String, that can be assigned with your values. Also, you can add custom properties of different types (see Figure 69). Click **OK** when ready. The new entity will be now visible inside the window, as shown in Figure 70.

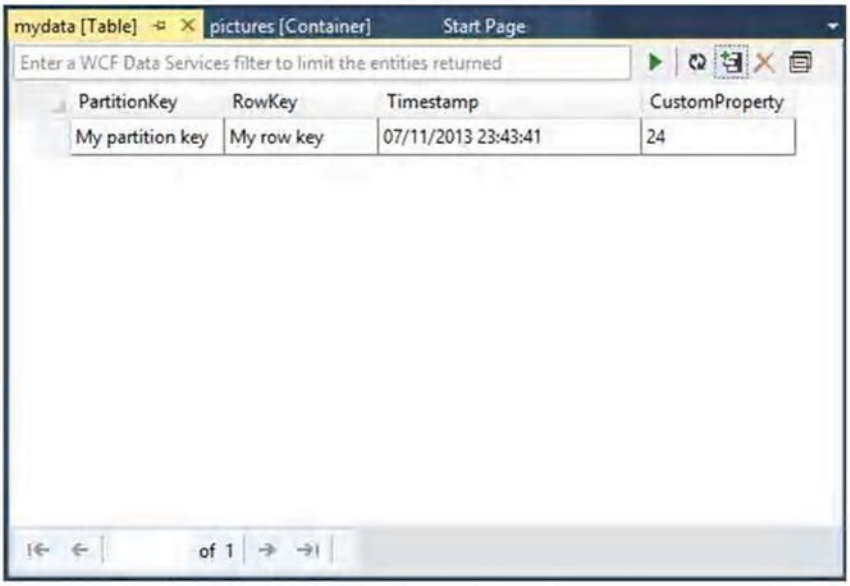


Figure 70: Listing Tables in your Storage Account

The toolbar has another button called Query Builder (the one on the right represented by two overlaid squares as shown in Figure 68), which allows executing queries against tables. You can specify one or more filters against different properties, as shown in Figure 71.

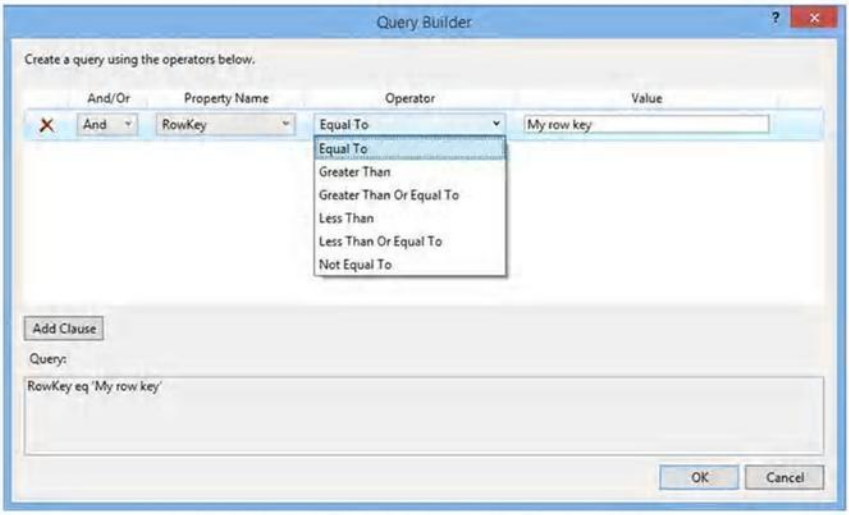


Figure 71: Defining Queries with the Query Builder

Notice that custom properties are not available in the Property Name dropdown. However, you can type queries and add filters by custom properties in the textbox inside the designer's toolbar (see Figure 68). When you have defined all the required filters, click **OK**. You will see the query syntax in the text box; if you click **Execute**, the query will be executed and only entities matching the specified criteria will be shown.

Create message queues

Visual Studio 2013 supports creating queues in the storage. With queues, cloud-based applications can easily share messages. To create a queue, in Server Explorer expand **Storage**, then expand the storage account of your choice (you can use **Development**), then right-click **Queues**. In the **Create Queue** dialog, enter the name for the new queue, again lower case (see Figure 72).

Figure 72: Creating a New Queue

When the new queue is visible in Server Explorer, double-click it to open the Queue window. Here is the place where you add and manage messages. To add one, click **Add Message** on the toolbar (the button with the icon of a letter and the green + symbol). In the appearing **Add Message** dialog, enter a text message and define when the message will expire (see Figure 73), and then click **OK**.

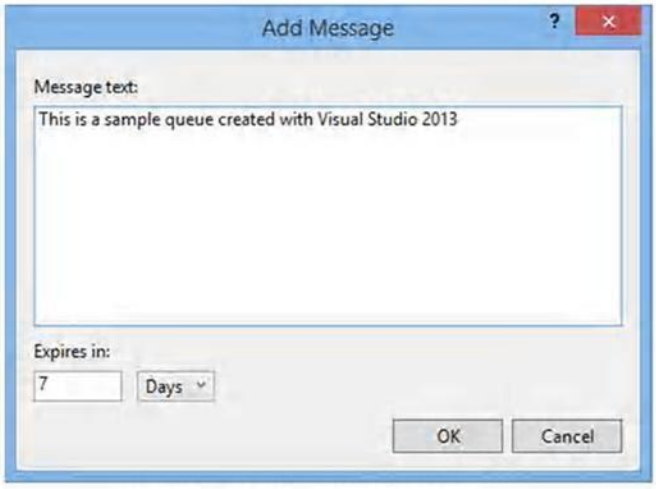


Figure 73: Creating a New Message

The new message will be added to the queue and will be visible in the Queue window, as you can see in Figure 74.

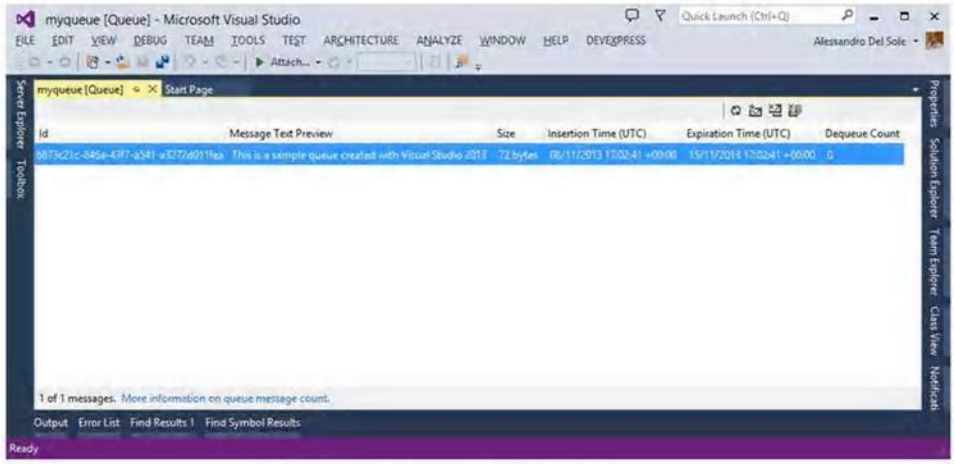


Figure 74: Creating a New Queue

Interesting information is shown, such as the insertion time, expiration time, and ID. By using the appropriate buttons on the toolbar, you can also de-queue or remove messages. If you are not familiar with using queues in your applications, check out the Windows Azure documentation about programming with queues. As for other storage types, Visual Studio 2013 makes it very easy to create and manage queues avoiding the need of using 3rd party applications.

Chapter summary

With the idea of offering the most productive environment ever, Visual Studio 2013 introduces many new tools for web development, including building applications for Windows Azure. On the ASP.NET side, Visual Studio 2013 introduces the One ASP.NET experience, which provides a unified approach to web development making it easy to use libraries from different frameworks into one application. This includes the use of scaffolding (formerly available only for MVC applications) in Web Forms applications. With scaffolding you can easily build data-centric applications and take advantage of auto-generated code for data access and data-bound, ready-to-use pages. Also, Visual Studio 2013 makes it easier to test applications in different web browsers with the new Browser Link feature, which allows refreshing all browsers with one click. For Windows Azure, Visual Studio 2013 enhances the Server Explorer tool window, which now offers all you need to work with most services exposed by the platform from within the IDE.

Chapter 6 New and Enhanced Tools for Debugging

As a developer, you probably spend a lot of time testing and debugging your code. Visual Studio 2013 introduces new debugging tools and updates some existing ones, continuing in its purpose of offering the most productive environment ever.

64-bit Edit and Continue

Visual Studio 2013 finally introduces Edit and Continue for 64-bit applications. As you know, with Edit and Continue, you can break the application's execution, edit your code, and then restart. So far, this has been available only for 32-bit applications. It is very easy to demonstrate how this feature works. Consider a very simple Console application, whose goal is retrieving the list of running processes and displaying the name of the first process in the list; the code is the following.

Visual C#

```
    class Program
    {
        static void Main(string[] args)
        {
            var runningProcesses = System.Diagnostics.
                                   Process.GetProcesses();
            Console.WriteLine(runningProcesses.First().ProcessName);
            Console.ReadLine();
        }
    }
```

Visual Basic

```
Module Module1

    Sub Main()
        'Add a breakpoint here and make your edits at 64-bits!
        Dim runningProcesses = System.Diagnostics.Process.GetProcesses()
        Console.WriteLine(runningProcesses.First().ProcessName)
        Console.ReadLine()
    End Sub

End Module
```

Before running the application, open the project's properties, select the **Build** tab, and change the platform target to **x64**, as shown in Figure 75.

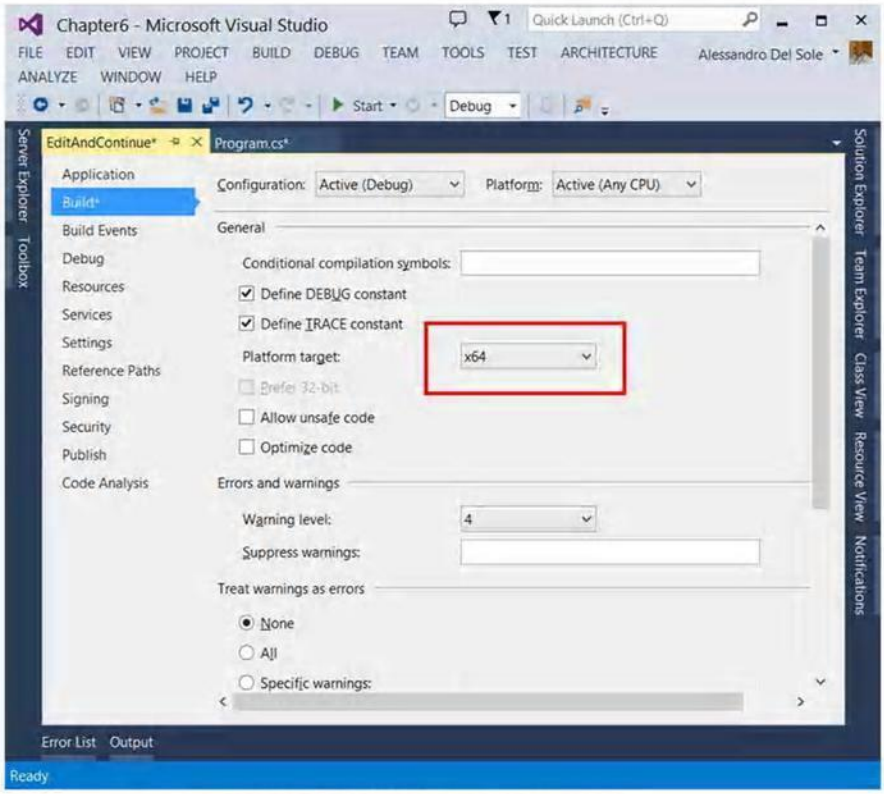


Figure 75: Selecting 64-bit Target Architectures

Now go back to the code, and place a breakpoint on the line containing the declaration of the **`runningProcesses`** variable by pressing **F9**. Finally, press **F5** to run the application. When the debugger encounters the breakpoint, the code editor is shown. You can simply rename the **`runningProcesses`** variable into **`currentProcesses`** (see Figure 76); this is enough to demonstrate how Edit and Continue is now working. Before Visual Studio 2013, if you tried to edit your code, at this point you would receive an error saying that Edit and Continue is only supported in 32-bit applications.

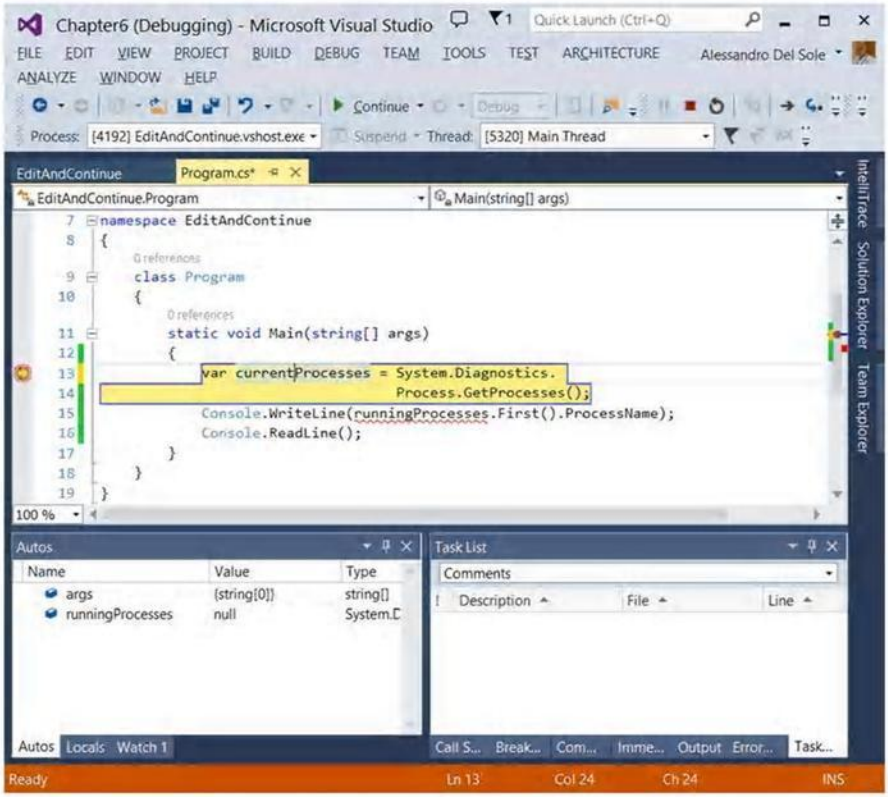


Figure 76: You can edit your code before resuming the execution.

Asynchronous debugging

Visual Studio 2012 and the .NET Framework 4.5 introduced a new pattern for coding asynchronous operations, known as the Async/Await pattern based on the new **async** and **await** keywords in the managed languages. The goal of this pattern is making the UI thread always responsive; the compiler can generate appropriate instances of the `Task` class and execute an operation asynchronously even in the same thread. You will see shortly a code example that will make your understanding easier, however there is very much more to say about Async/Await, so you are strongly encouraged to read the MSDN documentation if you've never used it. I

f you are already familiar with this pattern, you know that it is pretty difficult to get information about the progress and the state of an asynchronous operation at debugging time. For this reason, Visual Studio 2013 introduces a new tool window called Tasks. The purpose of this new tool window is to show the list of running tasks and provide information on active and pending tasks, time of execution, and executing code. The Tasks window has been very much publicized as a new addition to Windows Store apps development, but it is actually available to a number of other technologies, such as WPF. This is the reason why this feature is discussed in this chapter rather than in the next one about Windows 8.1.

Create a sample project

To understand how this feature works, let's create a new WPF Application project called *AsyncDebugging*. This application will create a new text file when the user clicks a button. The XAML code for the user interface is very simple, as represented in the following listing.

```
<Window x:Class="AsyncDebugging.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Width="100" Height="30" Name="FileButton" Content="Create
file" Click="FileButton_Click"/>
    </Grid>
</Window>
```

The code-behind file for the main window will contain the following code (see comments inside).

Visual C#

```
using System.IO;

        //Asynchronous method that passes some variables to
        //the other async method that will write the file
        //You wait for the async operation to be completed by using
        //the await operator. This method cannot be awaited itself
        //because it returns void.
        private async void WriteFile()
        {
            string filePath = @"C:\temp\testFile.txt";
            string text = "Visual Studio 2013 Succinctly\r\n";

            await WriteTextAsync(filePath, text);
        }


        //Asynchronous method that writes some text into a file
        //Marked with "async"
```

```
        private async Task WriteTextAsync(string filePath, string text)
        {
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            using (FileStream sourceStream = new FileStream(filePath,
                FileMode.Append, FileAccess.Write, FileShare.None,
                bufferSize: 4096, useAsync: true))
            {
                //new APIs since .NET 4.5 offer async methods to read
                //and write files
                //you use "await" to wait for the async operation to be
                //completed and to get the result
                await sourceStream.WriteAsync(encodedText, 0,
                      encodedText.Length);
            };
        }

        private void FileButton_Click(object sender, RoutedEventArgs e)
        {
            //Place a breakpoint here...
            WriteFile();
        }
```

Visual Basic

```
    Imports System.IO

    'Asynchronous method that passes some variables to
    'the other async method that will write the file
    'You wait for the async operation to be completed by using
    'the await operator. This method cannot be awaited itself
    'because it returns void.
    Private Async Sub WriteFile()
        Dim filePath As String = "C:\temp\testFile.txt"
        Dim text As String = "Visual Studio 2013 Succinctly"

        Await WriteTextAsync(filePath, text)
    End Sub

    'Requires Imports System.IO
    'Asynchronous method that writes some text into a file
    'Marked with "async"
    Private Async Function WriteTextAsync(filePath As String,
            text As String) As Task
        Dim encodedText As Byte() = Encoding.Unicode.GetBytes(text)

        Using sourceStream As New FileStream(filePath, FileMode.Append,
                                             FileAccess.Write,
```

```
                                        FileShare.None,
                                        bufferSize:=4096,
                                        useAsync:=True)
            'new APIs since .NET 4.5: async methods to read and write files
            'you use "await" to wait for the async operation
            'to be completed and to get the result
            Await sourceStream.WriteAsync(encodedText, 0,
                  encodedText.Length)
        End Using
    End Function

    Private Sub FileButton_Click(sender As Object, e As RoutedEventArgs)
        WriteFile()
    End Sub
```

In order to run the code without any errors, ensure you have a C:\Temp folder; if not, create one or edit the code to point to a different folder. If you start the application normally, after a few seconds you will see that the text file has been created correctly into the C:\Temp folder. If you already have used the Async/Await pattern in the past, you know that the debugging tools available until Visual Studio 2012 could not show the lifecycle of tasks; you could not know what task was active and which one was waiting. Let's see how Visual Studio 2013 changes things at this point.

Understanding the Tasks lifecycle with the Tasks window

Place a breakpoint on the **`WriteFile`** method invocation inside the button's click event handler (see the comment in the previous listing). Start the application and, when ready, click the button. When Visual Studio breaks the execution on the breakpoint, go to **Debug**, **Windows**, and select **Tasks**. The Tasks tool window will be opened and docked inside the IDE. Start debugging with Step Into by pressing **F11**. While asynchronous methods are invoked, the Tasks window shows their status, as demonstrated in Figure 77.

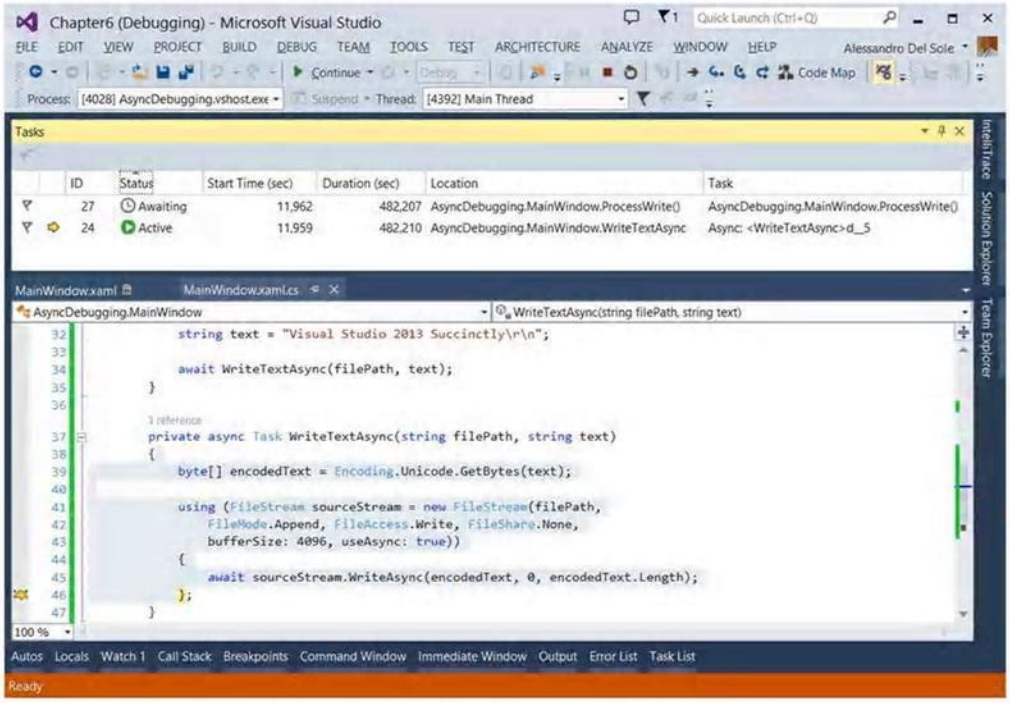


Figure 77: The Tasks window shows the status of asynchronous tasks.

By default, the Tasks window shows the following columns and related information:

- ID, which represents the task identifier.
- Status, which indicates whether the task is active or awaiting.
- Start Time (sec), which indicates the start time in seconds for the tasks.
- Location, which shows the name of the method where the task has been invoked.
- Task, which summarizes the operation in progress.

You can customize the Task window by adding or removing columns. If you right-click any column and then select **Columns**, a popup menu will show the full list of available columns; for instance, you might be interested in the Thread Assignment column to see what thread contains the selected task. The Tasks window is definitely useful when you need a better understanding of asynchronous operations' lifecycle, including when you need to analyze a task's performance. If the Tasks window does not display information as you step through lines of code using F11, and you are working with a desktop application, restart debugging and retry. This is a known issue. If you are working with a Windows Store app instead, you will not encounter this problem.

Performance and Diagnostics Hub

Analyzing performance and the behavior of an application is crucial. If your application is fast, fluid, and does not consume a lot of system resources (including battery for mobile apps), users will love it. Visual Studio has been offering analysis tools for many years, focusing on different areas such as memory usage, CPU usage, unit tests, and code analysis. With the big growth of mobile apps, Visual Studio has also been offering analysis tools specific to mobile platforms. In Visual Studio 2013, Microsoft has made another step forward, introducing a new unique place where you find such analysis tools. This place is called Performance and Diagnostics Hub. You can reach it by selecting **Debug**, **Performance** and **Diagnostics** or by pressing **ALT+F2**. Figure 78 shows how the Performance and Diagnostics Hub appears with a Windows Store app project.

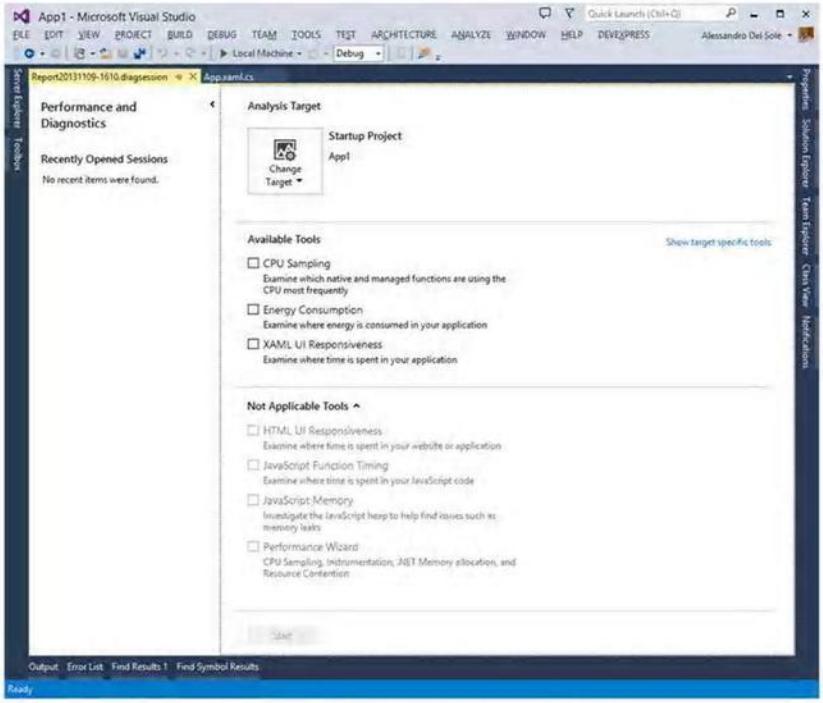


Figure 78: The Performance and Diagnostics Hub

Visual Studio 2013 will enable only target-specific tools. Figure 78 refers to a XAML Windows Store app, so all the other tools that target HTML Windows Store apps are disabled. For ASP.NET and desktop applications, only the CPU Sampling is available. Table 3 shows the list of available analysis tools per project type.

Table 3: Analysis Tools per Project Type

Analysis Tool	Purpose	Project Type(s)
Performance Wizard (includes CPU Sampling)	Analyze CPU usage, managed memory allocation, runtime diagnostics of the application state	All project types
Energy Consumption	Analyze potential battery usage through the Windows simulator	Windows Store apps
XAML UI Responsiveness	Analyze how time is spent in rendering layout	XAML Windows Store apps
HTML UI Responsiveness	Analyze how time is spent in rendering layout	HTML Windows Store apps
JavaScript Memory	Analyze the JavaScript heap to help find issues such as memory leaks	HTML Windows Store apps
JavaScript Function Timing	Analyze how time is spent in executing JavaScript code	HTML Windows Store apps

The CPU Sampling analysis tool invokes the Profiler that ships with Visual Studio, which you already know from previous versions. To start a diagnostics session you just select the tool you need and then click **Start** at the bottom of the page. When you close the application or break the diagnostic session manually, Visual Studio will generate a report based on the analysis type you selected. In the next chapter, when we discuss new features for Windows 8.1, you will get a more detailed demonstration of this tool. Remember that you can still access analysis tools via the Analyze menu as you did with previous versions of the IDE.

Code Map debugging

Note: Code Map is available only in Visual Studio 2013 Ultimate.

Another interesting addition to Visual Studio 2013 is Code Map. Actually, Code Map is available in Visual Studio 2012 with Update 1, but now the tool is integrated in the IDE. With Code Map, you can get an incremental visualization of your application and dependencies. In simpler words, you can get a visual representation of method calls, references, and fields while debugging, inside an interactive window where you can also add comments, flag an item for follow up, and export graphics to an image file.

To understand how Code Map works, let's consider the WPF sample application we created to demonstrate asynchronous debugging earlier in this chapter. Ensure a breakpoint is still inside the button's click event handler, then start the application with **F5**. Click the button in the application, then when the debugger encounters the breakpoint and breaks, click the **Code Map** on the toolbar (see Figure 79).

Figure 79: The Code Map Button

Visual Studio will start generating a map at this point. After a few seconds, you will see the method call in the Code Map, as represented in Figure 80.

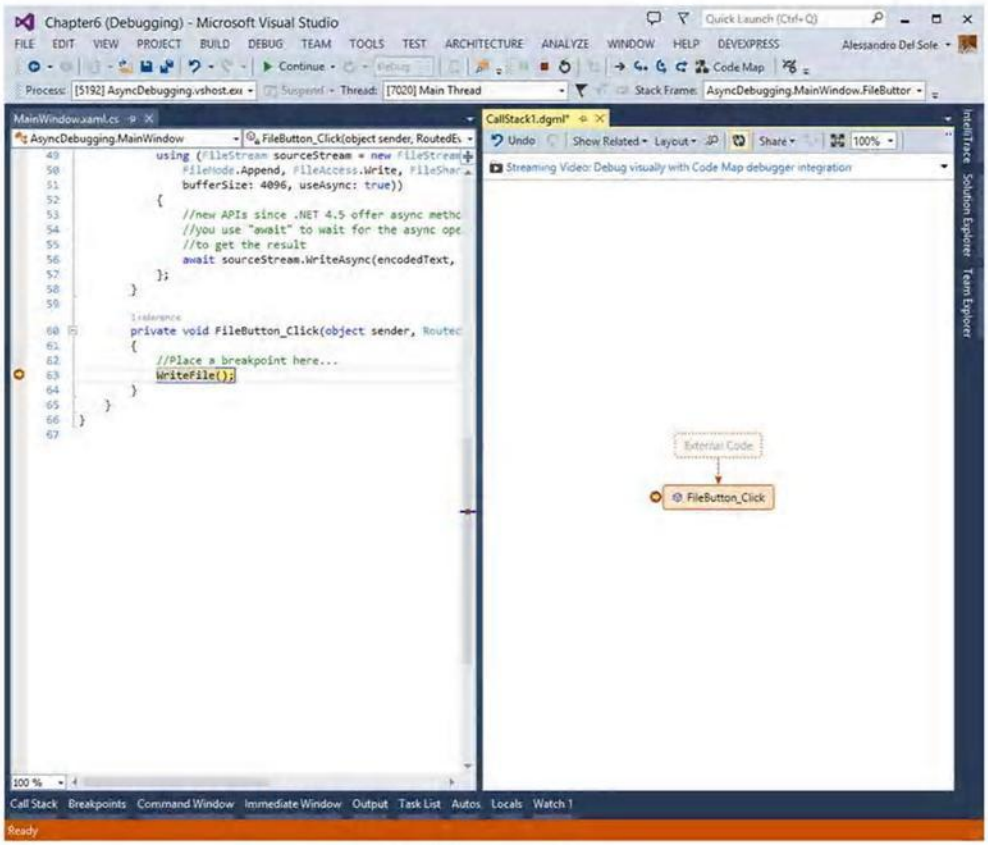


Figure 80: A New Code Map

Before continuing, you can play with the various buttons on the window's toolbar. For instance, if you check the Share button, you will see how you can easily export or email the diagram as an image file or as a portable XPS file. The Layout button offers an option to show the code map in different ways, whereas Show Related allows finding references to methods and types for the selected item in the map. Now press **F11** to execute the next line of code. The Code Map is immediately updated with the call to the **`WriteFile`** method, as shown in Figure 81.

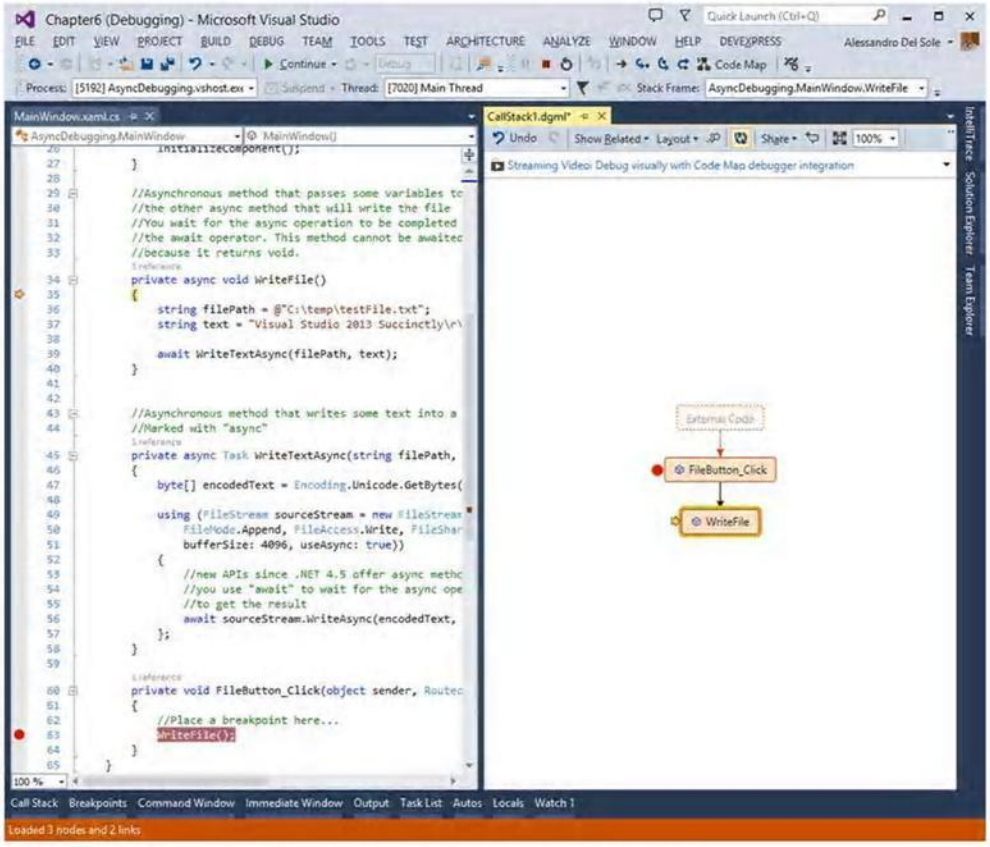


Figure 81: The code map is updated while debugging.

Objects are also represented on the Code Map. For example, while you are debugging the **`WriteFile`** method, right-click the **`text`** variable and then click **Show On Code Map**. The map will be updated (see Figure 82) with the referenced variable, shown inside its containing object.

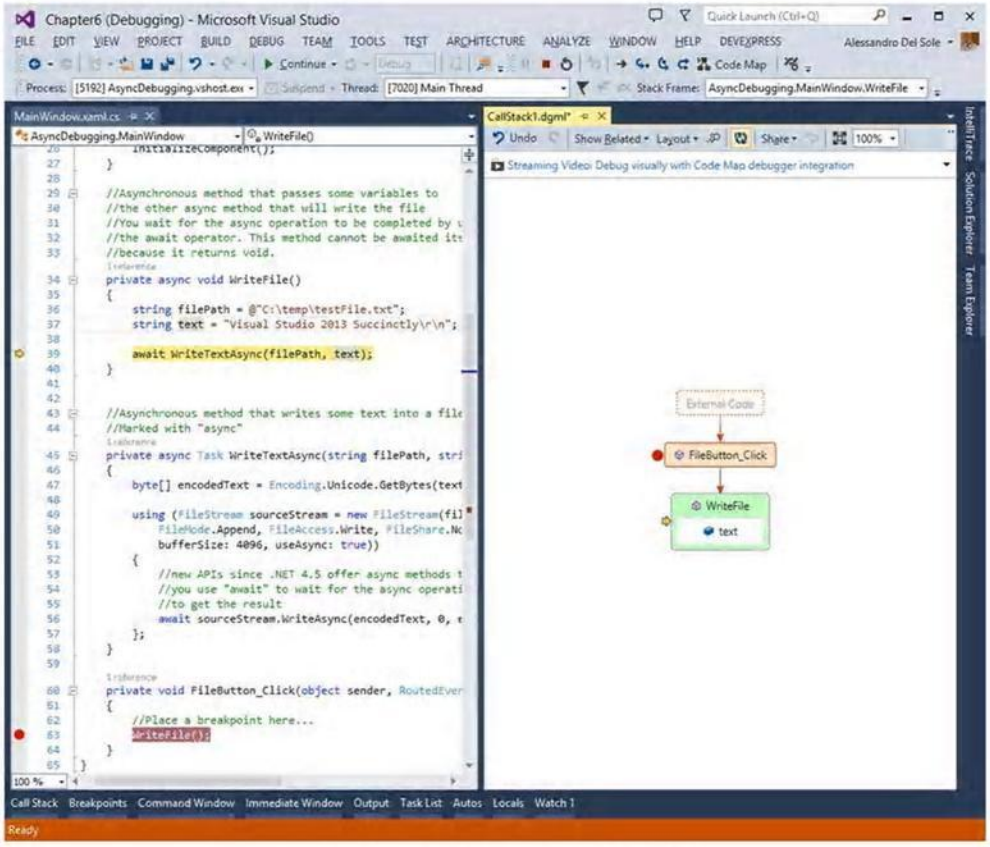


Figure 82: The code map is updated while debugging.

If you right-click a method in the map, you will be able to display a number of data points such as calls to other methods, fields the method references, and the containing type. For example, right-click the **`WriteFile`** method and then select **Show Methods This Call**. Visual Studio will show calls to other methods made by **`WriteFile`**, as shown in Figure 83.

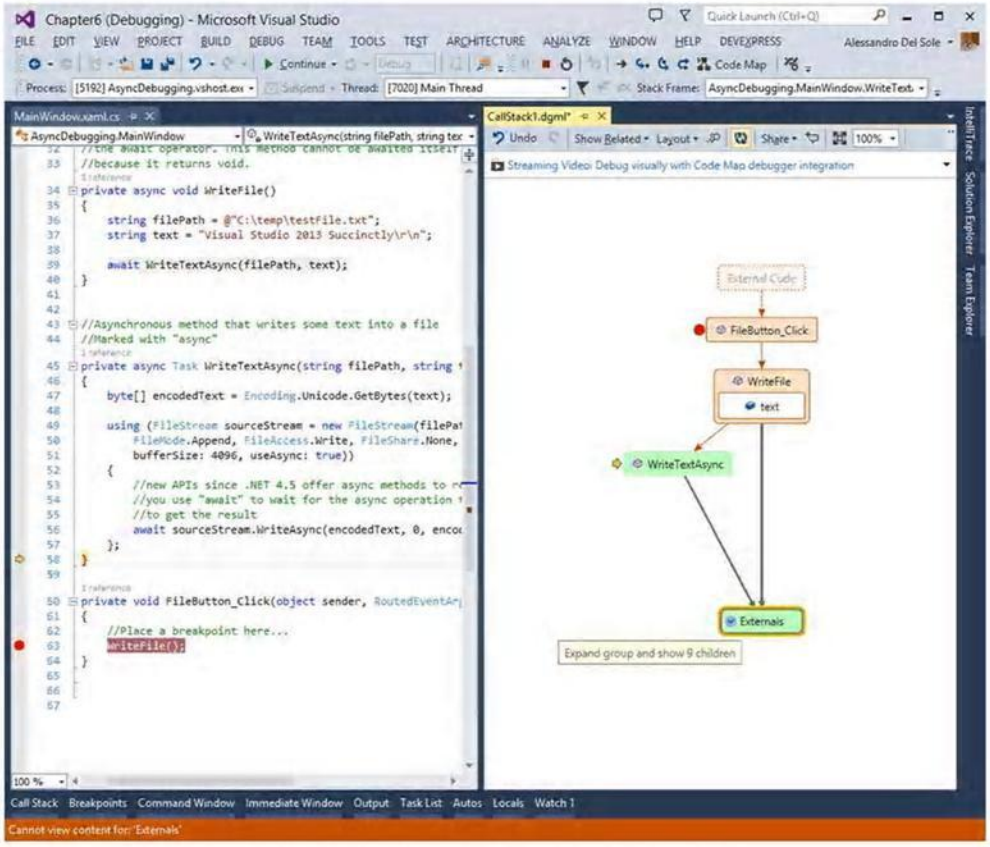


Figure 83: Showing method calls from the selected method.

The method calls **`WriteTextAsync`**, which invokes external code. Such an external code is how the runtime translates the Async/Await pattern into the backing .NET methods. This can be easily demonstrated by expanding the **Externals** node by clicking the expansion button inside. As a tooltip suggests, if you expand the Externals node you will be able to see nine children objects, as represented in Figure 84.

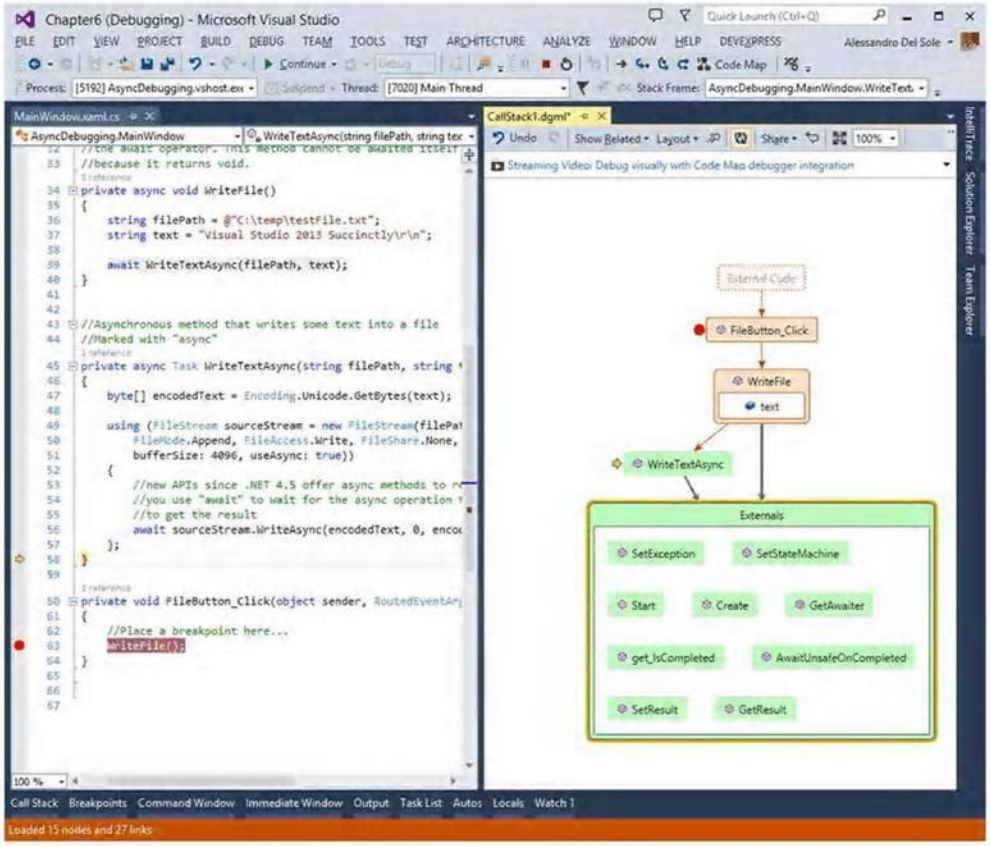


Figure 84: Investigating External Calls

All the method calls you see in the map are handled by the runtime to manage asynchronous operations on your behalf. It is worth mentioning that every time you pass the mouse pointer over a method, a tooltip shows the method definition in code. You can also add comments and flag items for follow up. To add a comment, right-click an item and then select **New Comment**. You will be able to enter your comment inside a text box. To flag an item for follow up, right-click it and then select **Flag for Follow Up**. In Figure 85, you can see a comment and the `WriteTextAsync` method flagged for follow up.

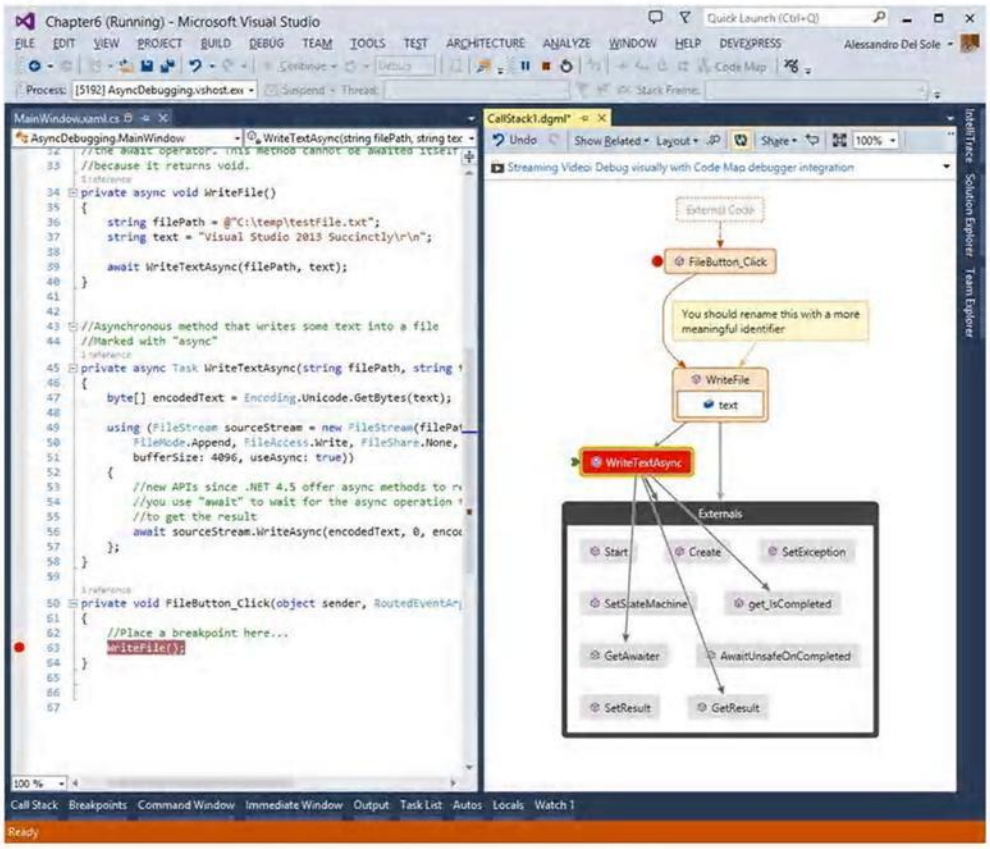


Figure 85: Adding Comments and Flags

You can finally right-click an item and see advanced properties by selecting the **Advanced** group in the context menu. Figure 86 shows the result of the command **Show Containing Type**, **Namespace**, and **Assembly**.

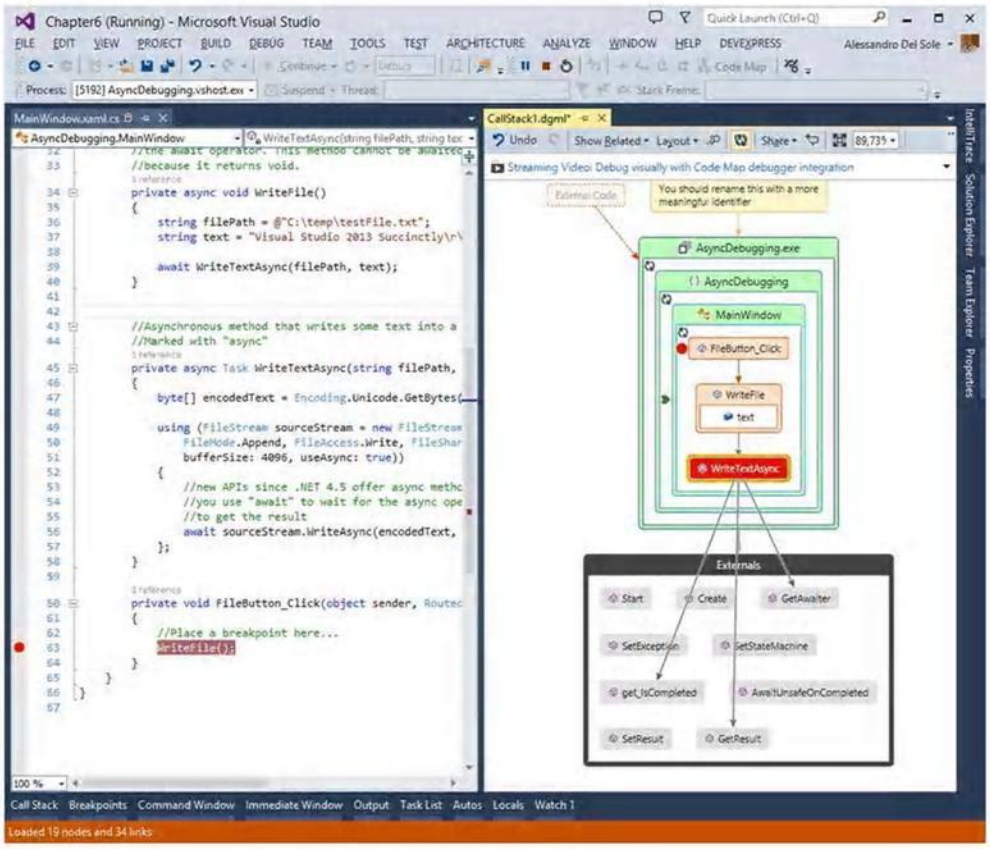


Figure 86: Visualizing Advanced Properties

It is worth mentioning that the context menu you see when you right-click any items will show the Go To Definition command, which will redirect you to the object definition in either the code editor or the Object Browser window. As you can easily understand, Code Map provides a great benefit because it allows debugging while literally seeing what is happening; this makes it easier to discover the most subtle bugs.

Method Return Value

Visual Studio 2013 brings to Visual C# and Visual Basic a feature that was already available to C++, which is the ability to view a method's return value inside the Autos window without the need to step into the code. To understand how this feature works, create a new Console application. Now consider the following code.

Visual C#

```
    class Program
    {
        static void Main(string[] args)
        {
            //Step Over (F10)
            int result = Multiply(Five(), Six());
        }

        private static int Multiply(int num1, int num2)
        {
            return (num1 * num2);
        }

        private static int Five()
        {
            return (5);
        }

        private static int Six()
        {
            return (6);
        }
    }
```

Visual Basic

```
Module Module1

    Sub Main()
        'Step over (F10)
        Dim result As Integer = Multiply(Five(), Six())
    End Sub

    Private Function Multiply(num1 As Integer, num2 As Integer) As Integer
        Return (num1 * num2)
    End Function

    Private Function Five() As Integer
        Return (5)
    End Function

    Private Function Six() As Integer
        Return (6)
    End Function
End Module
```

As you can see, this simplified code returns the result of a multiplication by invoking two methods, each returning an integer value. As suggested in the code, place a breakpoint on the only line of code in the **`Main`** method and start the application by pressing **F5**. You can step over (**F10**) to execute the method without executing the other methods line by line. At this point, you will be able to see the value returned by every intermediate method call in the Autos window, as shown in Figure 87.

Tip: If the Autos window is not displayed automatically, go to Debug, then select Windows, then Autos.

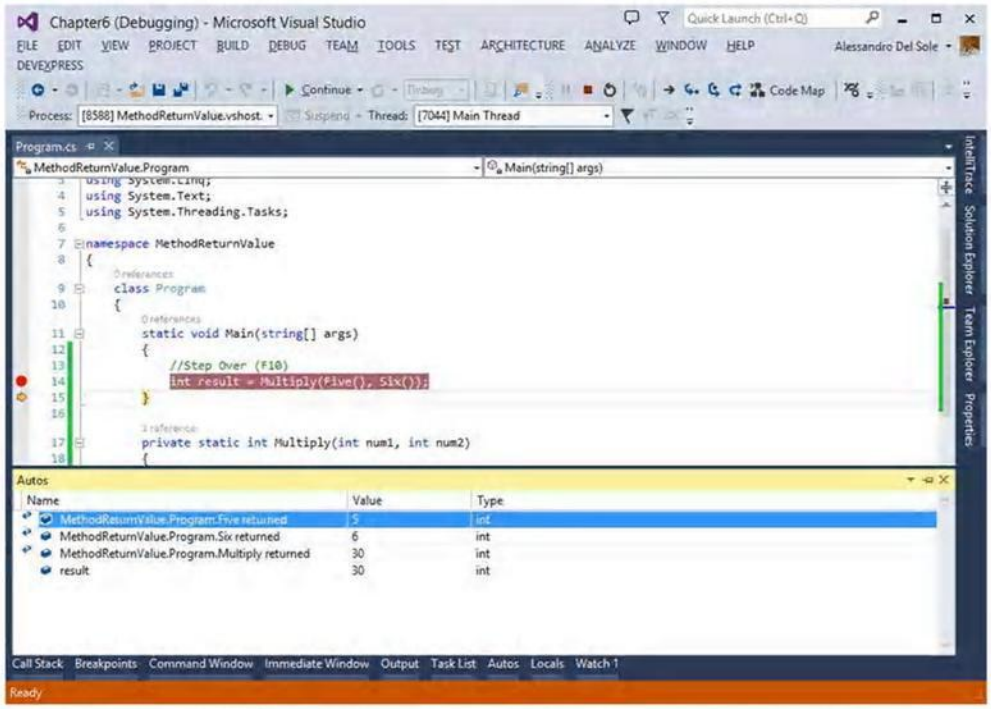


Figure 87: Method Return Values Shown in the Autos Window Without Executing Line by Line

This feature is useful when you need to focus on one piece of code and you do not want to step into every single line, but you still want to see the result of every method call.

Chapter summary

Because debugging is one of the most important activities in application development, Microsoft has made a significant investment to make the debugging experience in Visual Studio 2013 even more productive. Now you can finally use the popular Edit and Continue feature against 64-bit applications. You can take advantage of asynchronous debugging to understand the lifecycle of asynchronous operations based on the Async/Await pattern. You now have a unified place to analyze your applications' performances and behavior with the new Performance and Diagnostics Hub. You can get a graphical representation of your code execution while debugging with Code Map. Finally, you can now get method return values without stepping into every single line of code, just by stepping over the caller method. All these new features will save you time and help you write high-quality code.

Chapter 7 Visual Studio 2013 for Windows 8.1

One reason for the release of a new version of Visual Studio after only one year is that several technologies other have been updated. Probably the most important update has been Windows 8.1., which introduces many new APIs and changes in the existing infrastructure. Because of this, developers need an updated version of the .NET Framework (the 4.5.1) to support Windows 8.1 and a new version of Visual Studio based on .NET 4.5.1. This chapter covers new features in the IDE related to Windows Store app development. If you instead wish to learn about the new APIs in Windows 8.1, you can refer to the MSDN documentation.

Tip: XAML IntelliSense improvements discussed in Chapter 4 are certainly valid for Windows 8.1 app development. Since I already talked about such improvements in detail before, I will not repeat them here.

New project templates

Windows 8.1 introduces a new control called **Hub**, which provides the ability to create a central hub in your Windows Store apps. Basically the concept of a hub is providing users with a landing page that gives an overview of different parts of the app in one place. Before Windows 8.1, developers had to do a bit of work to manually create a hub. To highlight the importance of this control, Visual Studio 2013 introduces the **Hub** control and a specific project template called Hub App, which is available for both XAML and HTML modes, and only if you run Visual Studio 2013 on Windows 8.1. Figure 88 shows the New Project dialog with the new template selected.

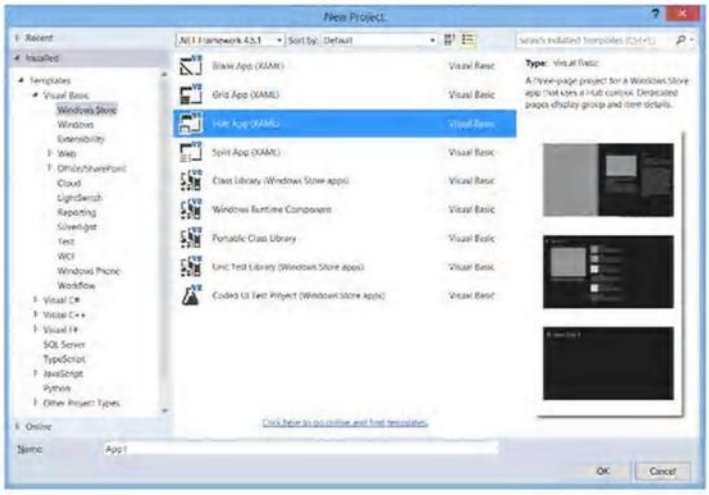


Figure 88: The New Hub App Project Template

Understanding how the **Hub** control works is very easy. You can just create a new project based on the Hub App template. When the project is ready, start the application before looking at the code. By either using the mouse or your finger, you can scroll the main page horizontally to see how the Hub allows creating sections of contents or shortcuts to additional pages. Figure 89 shows the sample app running.

Figure 89: The Hub Control allows organizing content and shortcuts.

Now look at the XAML code. The built-in project template provides a very rich and powerful example, but what you need to know at the higher-level is represented in the following code.

```
<Hub SectionHeaderClick="Hub_SectionHeaderClick">
    <Hub.Header>
        <Grid>
        <!-- Controls here... -->
        </Grid>
    </Hub.Header>
    <HubSection Width="780" Margin="0,0,80,0">
        <HubSection.Background>
        <!-- Your brush here... -->
        </HubSection.Background>
        <Grid>
        <!-- Controls here... -->
        </Grid>
    </HubSection>
    <HubSection Width="500" Header="Section 1">
        <DataTemplate>
        <!-- Your data-bound items here... -->
        </DataTemplate>
    </HubSection>
```

```
        </Hub>
```

Among the others, the **Hub** control exposes the **Header** property that shows content summarizing the topic of the section. Because of the XAML hierarchical nature, **Header** can be not only text, but also a set of nested controls. The **Hub** control contains **HubSection** controls for as many topics as you need to summarize. The **HubSection** control is very versatile, since it can store any kind of content. As you can see from the code snippets in the previous listing, you can put text or panels, set the background, and even place data-bound controls via **DataTemplate** elements. The MSDN Code Gallery contains a very good example of the **Hub** control for both XAML and HTML that you can download for additional testing. Of course, the MSDN documentation provides everything you need to know for building apps with the **Hub** control. Since explaining how to program this control in detail is beyond the scope of this book, you can check out the related page on MSDN.

Improved Device tool window

When you work with the designer on a Windows Store app, you can take advantage of a useful tool window called Device (also known as Device panel). It allows changing some properties of your application so that you can get a preview of your edits at design time, avoiding the need to launch the application every time. The Device panel is not new in Visual Studio 2013; it was already available in Visual Studio 2012, but has now been reorganized and updated with new features. Figure 90 shows the Device panel.

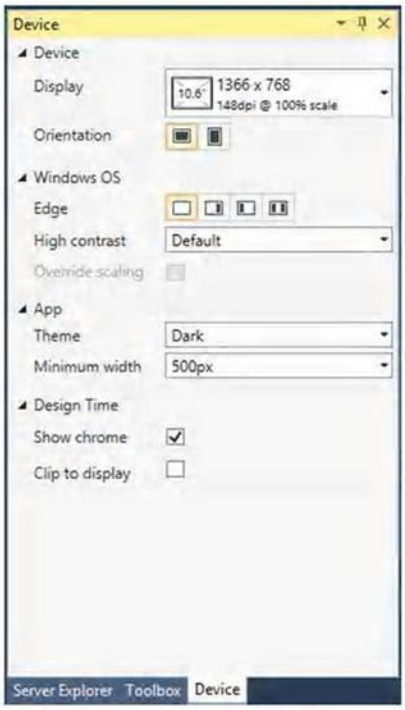


Figure 90: The Device Tool Window

Properties on the window are self-explanatory, and it's easy to see the result on the designer when you change them. In summary you can:

- Change the app resolution in the designer with the Display property.
- Switch between horizontal and vertical orientation with the Orientation property.
- Test how the app will appear on screen with the Edge property.
- Select the screen contrast with the High contrast property.
- Test the app on a different scaling with the Override scaling property. Scaling is increased by 40%.
- Change the theme to see how the app responds to system settings with the Theme property.
- Establish a minimum app width with the Minimum width property
- Show or remove the device chrome in the designer with the Chrome property.
- Clip the entire document or show the document display with the Clip to display property.

The Device panel is a good companion to get a preview of the behavior of the app directly in the designer, so that you can change the app layout and appearance and see if the result you get is what you (and your users) expect.

Connect to Windows Azure mobile services

In Chapter 5, you discovered new features in the Server Explorer window to provide integration with Windows Azure services from within the IDE. You learned what a mobile service is and how to create one from Server Explorer. Continuing the integration with the cloud platform, Visual Studio 2013 allows connecting easily to a mobile service in Windows 8.1 applications.

In a Store app, first save the project—otherwise the tooling will not work without giving you any warning. Then right-click the project name in Solution Explorer, then select **Add**, **Connected Service**. At this point the Services Manager dialog appears. Select the **Windows Azure** node to see a list of available services, as shown in Figure 91.

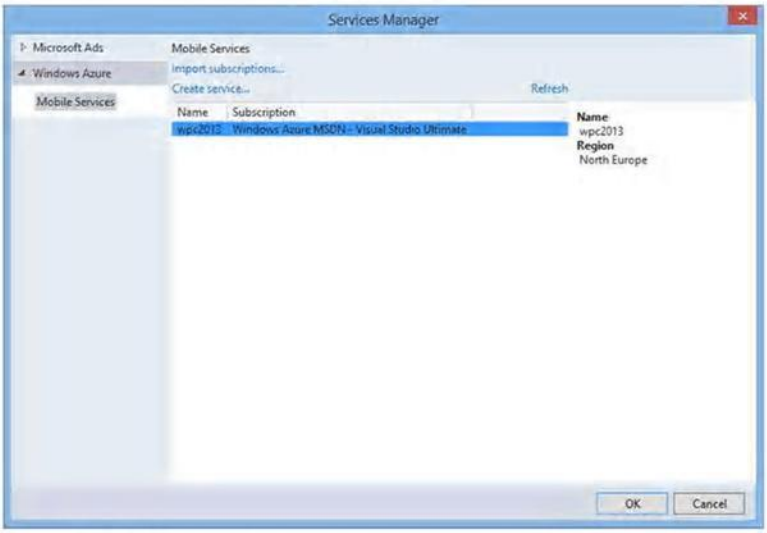


Figure 91: The Services Manager dialog shows available Mobile Services.

Once you click **OK**, a service reference is added. Visual Studio 2013 will automatically add references to assemblies that are required in order to connect to the service in code and to manage data stored inside the service. You can expand the **References** node in Solution Explorer to see what libraries have been referenced; Figure 92 demonstrates this.

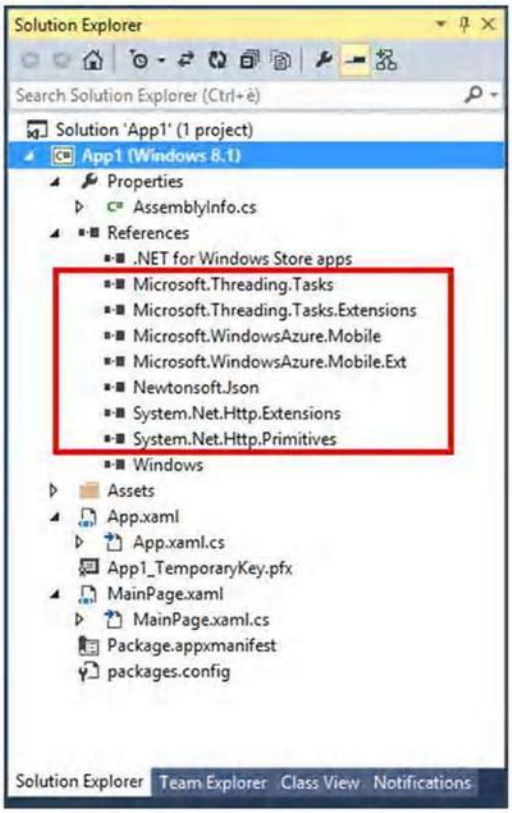


Figure 92: References Added to Support Coding against Mobile Services

The new assemblies required to interact with a Mobile Service are Microsoft.WindowsAzure.Mobile.dll and Microsoft.WindowsAzure.Mobile.Ext.dll. Other assemblies are part of the .NET runtime and are required for serializing and deserializing data through the JSON format over the network. Not limited to this, Visual Studio 2013 will also add to the **App** class code the following line (**yourmobileservice** stands for the name of your service and **YOURSECRETKEY** stands for the client secret key, both added appropriately):

Visual C#

```
public static Microsoft.WindowsAzure.MobileServices.MobileServiceClient
yourmobileserviceClient = new
Microsoft.WindowsAzure.MobileServices.MobileServiceClient(
        "https://yourmobileservice.azure-mobile.net/",
        "YOURSECRETKEY");
```

Visual Basic

```
Public Shared yourmobileserviceClient As New
Microsoft.WindowsAzure.MobileServices.MobileServiceClient(
        "https://yourmobileservice.azure-mobile.net/",
        "YOURSECRETKEY");
```

By creating an instance of the **Microsoft.WindowsAzure.MobileServices.MobileServiceClient** class, your app will be able to connect to the specified mobile service. For a deeper understanding of the code you need to write to manage data and C.R.U.D. operations from your app, you can follow the example shown in the Getting started with Mobile Services page in the Windows Azure documentation, which also provides guidance to use these services in other platforms.

Asynchronous debugging

Visual Studio 2013 introduces a new tool window called Tasks, which helps developers debug asynchronous operations written according to the Async/Await pattern introduced with the previous version. This feature was discussed in detail in the previous chapter, so you should be able to use it successfully against Windows Store apps.

Analyze performance with the XAML UI Responsiveness Tool

Visual Studio 2013 brings to XAML Store apps the UI Responsiveness Tool that was already available for HTML/JavaScript Store applications. As the name implies, this tool analyzes the user interface performance and generates a detailed report about the app behavior. In order to use this tool, you need to open the Performance and Diagnostics Hub that you discovered in the previous chapter. When visible, you have to select the **XAML UI Responsiveness** check box, as shown in Figure 93.

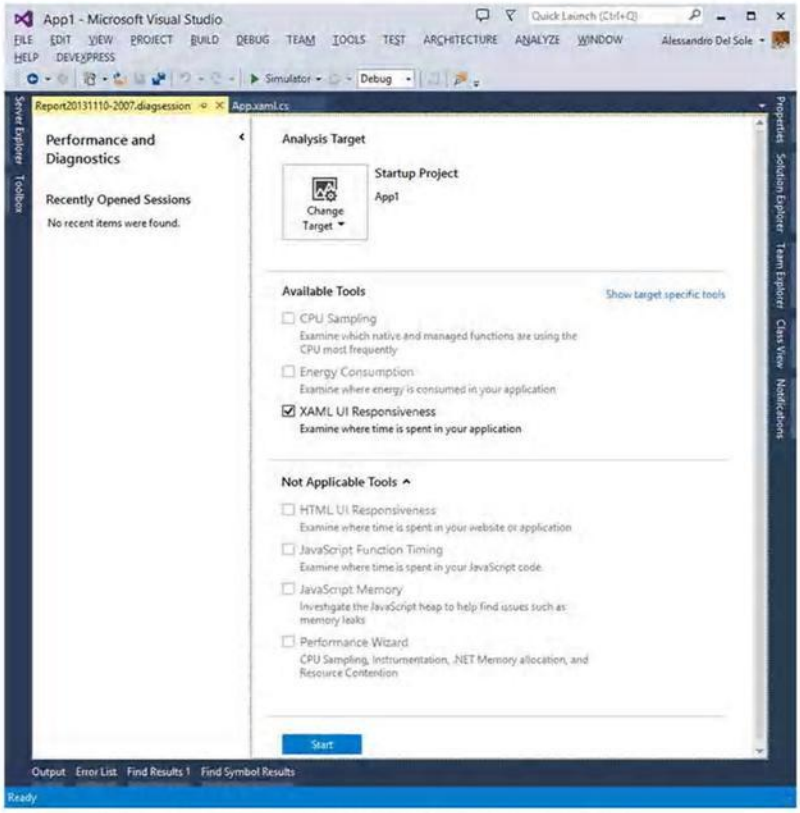


Figure 93: Selecting the XAML UI Responsiveness Tool

When ready, click the **Start** button. Use your app for a while, then close it or go back to Visual Studio and click the **Stop Collection** hyperlink. After a few seconds, Visual Studio shows a detailed report about the collected information about performances; Figure 94 shows an example.

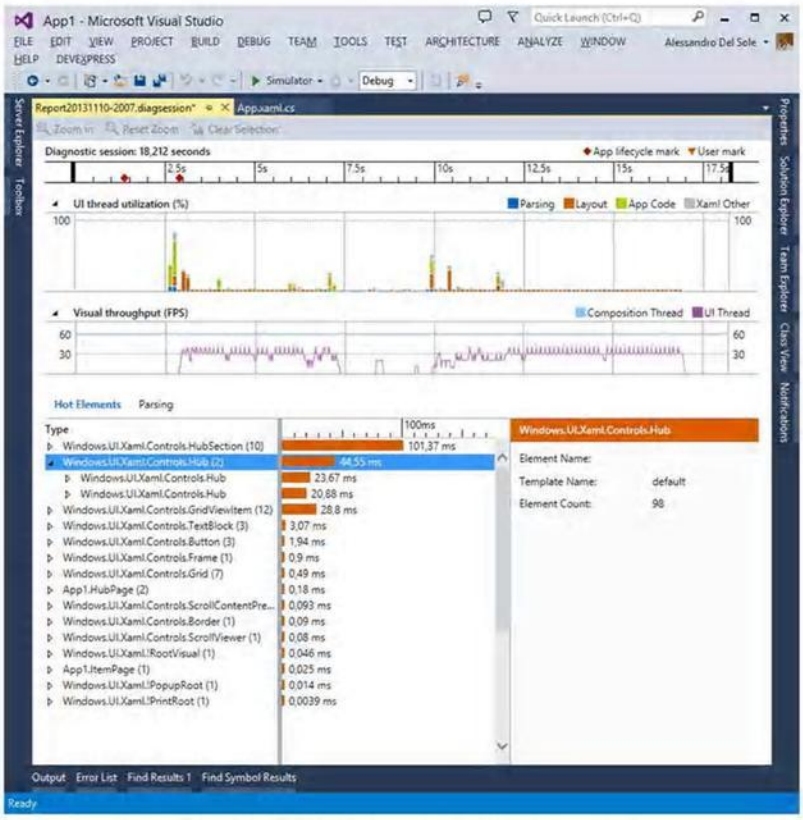


Figure 94: Selecting the XAML UI Responsiveness Tool

The report can be divided into four main parts. Let's discuss them in more detail.

Diagnostic Session

The Diagnostic Session report shows information about the application lifecycle and user interaction. It displays the test duration and it shows how much time the app required for activation. This section can be useful to discover important performance problems that you must avoid for a successful app submission.

UI Thread Utilization

The UI Thread Utilization section shows how the UI thread has been exploited in percentage by different tasks managed by the runtime. You can understand how many resources have been consumed by the XAML parser (blue), how many in rendering the user interface (dark orange), in executing the app code (light green), and in other tasks related to XAML (not parsing). This can be very useful to understand what areas of your code have the most negative impact on the overall performance.

Visual Throughput (FPS)

This section shows how many frames per second (FPS) have been rendered during the application lifecycle; for timing, you can take the Diagnostic Session as a reference. This tool is very straightforward, and can show frames in both the UI thread and the composition thread. If you pass the mouse pointer over the graphic, you will see a tooltip showing frames per second for both threads at the given time.

Tip: If you are new to Windows Store application development, you might not be familiar with the concept of composition thread. The composition thread was introduced with Windows Phone and is a companion thread for the UI thread, in that it does some work that the UI thread would normally do. The composition thread is normally responsible for combining graphics texture and for sending them to the GPU for rendering. This is all managed by the runtime; by invoking the composition thread, the runtime can make an app stay much more responsive and you, as the developer, will not need to do any additional work.

Hot Elements and Parsing

At the bottom of the report, you will find two tabs, **Hot Elements** and **Parsing**. Hot Elements shows the list of UI elements (.NET objects with their fully qualified name) and the time in milliseconds they have been busy with the application execution. Objects in the list can be expanded to show nested controls and types. When you click an object, you will also be able to see additional information such as nested elements count, XAML code file (if not a system object), and the control template, on the right side of the window. The Parsing tab shows the list of XAML files in the application package and how much time in milliseconds has been required for parsing. This tool is not limited to XAML files you see in Solution Explorer, but also works against built-in XAML files prepackaged in the application.

Chapter summary

Windows 8.1 is the new major upgrade for the Windows 8 operating system, which introduces tons of new APIs and features that developers can leverage to build amazing apps. Visual Studio 2013 is the environment you need to build apps for Windows 8.1. This new version introduces a new **Hub** control, which Visual Studio 2013 supports with the Hub App project template. At design time, you can get previews of your app's look and feel via the Device panel. You can now easily connect to Windows Azure Mobile Services and Visual Studio will do most of the work for you. Great apps are fluid and fast apps, so with Visual Studio 2013 you have a new analysis tool called XAML UI Responsiveness, which helps you understand how and where your app spends more time and consumes more resources.

AI : Neural Network for beginners (Part 2 of 3)

Sacha Barber, 29 Jan 2007

AI : An Introduction into Neural Networks (Multi-layer networks / Back Propagation)

Download demo project (includes source code) - 812 Kb

Introduction

This article is part 2 of a series of 3 articles that I am going to post. The proposed article content will be as follows:

1. Part 1 : Is an introduction into Perceptron networks (single layer neural networks).
2. Part 2 : This one, is about multi layer neural networks, and the back propagation training method to solve a non linear classification problem such as the logic of an XOR logic gate. This is something that a Perceptron can't do. This is explained further within this article.
3. Part 3 : Will be about how to use a genetic algorithm (GA) to train a multi layer neural network to solve some logic problem.

Summary

This article will show how to use a multi-layer neural network to solve the XOR logic problem.

A Brief Recap (From part 1 of 3)

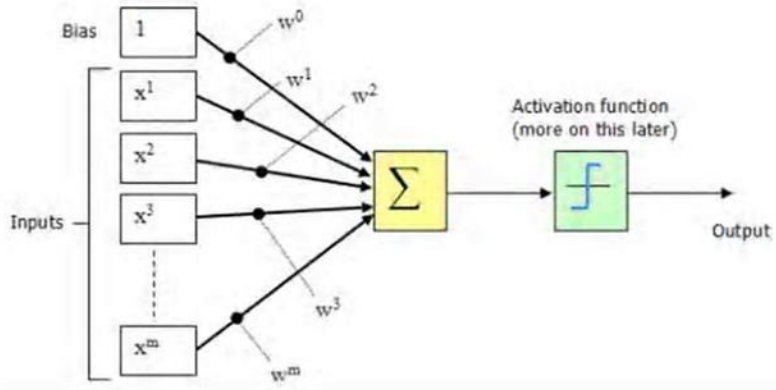
Before we commence with the nitty gritty of this new article which deals with multi layer Neural Networks, let just revisit a few key concepts. If you haven't read Part 1, perhaps you should start there.

Perceptron Configuration (Single layer network)

The inputs and connection weights shown below are typically real values, both positive (+) and negative (-).

The perceptron itself, consists of weights, the summation processor, an activation function, and an adjustable threshold processor (called bias here after).

For convenience, the normal practice is to treat the bias as just another input. The following diagram illustrates the revised configuration.



The bias can be thought of as the propensity (a tendency towards a particular way of behaving) of the perceptron to fire irrespective of it's inputs. The perceptron configuration network shown above fires if the weighted sum > 0, or if you have into maths type explanations

$$\sum_{i=1}^{m} bias + (w^i x^i)$$

So that's the basic operation of a perceptron. But we now want to build more layers of these, so let's carry on to the new stuff.

So Now The New Stuff (More layers)

From this point on, anything that is being discussed relates directly to this article's code.

In the summary at the top, the problem we are trying to solve was how to use a multi-layer neural network to solve the XOR logic problem. So how is this done. Well it's really an incremental build on what Part 1 already discussed. So let's march on.

What does the XOR logic problem look like? Well, it looks like the following truth table:

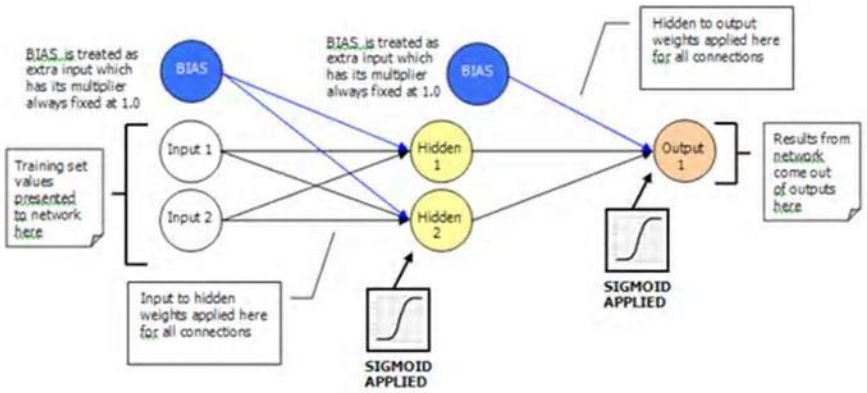
I1	I2	Output
0	0	0
0	1	1
1	0	1
1	1	0

XOR Logic Gate

Remember with a single layer (perceptron) we can't actually achieve the XOR functionality, as it is not linearly separable. But with a multi-layer network, this is achievable.

What Does The New Network Look Like

The new network that will solve the XOR problem will look similar to a single layer network. We are still dealing with inputs / weights / outputs. What is new is the addition of the hidden layer.



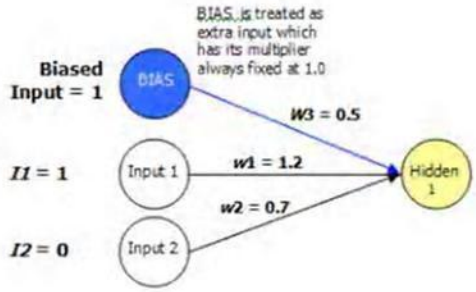
As already explained above, there is one input layer, one hidden layer and one output layer.

It is by using the inputs and weights that we are able to work out the activation for a given node. This is easily achieved for the hidden layer as it has direct links to the actual input layer.

The output layer, however, knows nothing about the input layer as it is not directly connected to it. So to work out the activation for an output node we need to make use of the output from the hidden layer nodes, which are used as inputs to the output layer nodes.

This entire process described above can be thought of as a pass forward from one layer to the next.

This still works like it did with a single layer network; the activation for any given node is still worked out as follows:



$$A = \sum_{n=1}^{N+1} wi * Ii$$

NOTE : N + 1 is bias and the value of the input for the bias is 1.

Where (wi is the weight(i), and Ii is the input(i) value)

You see it the same old stuff, no demons, smoke or magic here. It's stuff we've already covered.

So that's how the network looks/works. So now I guess you want to know how to go about training it.

Types Of Learning

There are essentially 2 types of learning that may be applied, to a Neural Network, which is "Reinforcement" and "Supervised"

Reinforcement

In Reinforcement learning, during training, a set of inputs is presented to the Neural Network, the Output is 0.75, when the target was expecting 1.0.

The error (1.0 - 0.75) is used for training ('wrong by 0.25').

What if there are 2 outputs, then the total error is summed to give a single number (typically sum of squared errors). Eg "your total error on all outputs is 1.76"

Note that this just tells you how wrong you were, not in which direction you were wrong.

Using this method we may never get a result, or it could be a case of 'Hunt the needle'.

NOTE : Part 3 of this series will be using a GA to train a Neural Network, which is Reinforcement learning. The GA simply does what a GA does, and all the normal GA phases to select weights for the Neural Network. There is no back propagation of values. The Neural Network is just good or just bad. As one can imagine, this process takes a lot more steps to get to the same result.

Supervised

In Supervised Learning the Neural Network is given more information.
Not just 'how wrong' it was, but 'in what direction it was wrong' like 'Hunt the needle' but where you are told 'North a bit', 'West a bit'.

So you get, and use, far more information in Supervised Learning, and this is the normal form of Neural Network learning algorithm. Back Propagation (what this article uses, is Supervised Learning)

Learning Algorithm

In brief, to train a multi-layer Neural Network, the following steps are carried out:

- Start off with random weights (and biases) in the Neural Network
- Try one or more members of the training set, see how badly the output(s) are compared to what they should be (compared to the target output(s))
- Jiggle weights a bit, aimed at getting improvement on outputs
- Now try with a new lot of the training set, or repeat again, jiggling weights each time
- Keep repeating until you get quite accurate outputs

This is what this article submission uses to solve the XOR problem. This is also called "Back Propagation" (normally called BP or BackProp)

Backprop allows you to use this error at output, to adjust the weights arriving at the output layer, but then also allows you to calculate the effective error 1 layer back, and use this to adjust the weights arriving there, and so on, back-propagating errors through any number of layers.

The trick is the use of a sigmoid as the non-linear transfer function (which was covered in Part 1. The sigmoid is used as it offers the ability to apply differentiation techniques.

$$y = g(x) = \frac{1}{1 + e^{-x}}$$

Because this is nicely differentiable —it so happens that

$$\frac{dg}{dx} = g'(x) = g(x)(1 - g(x))$$

Which in context of the article can be written as

```
delta_outputs[i] = outputs[i] * (1.0 - outputs[i]) * (targets[i] - outputs[i])
```

It is by using this calculation that the weight changes can be applied back through the network.

Things To Watch Out For

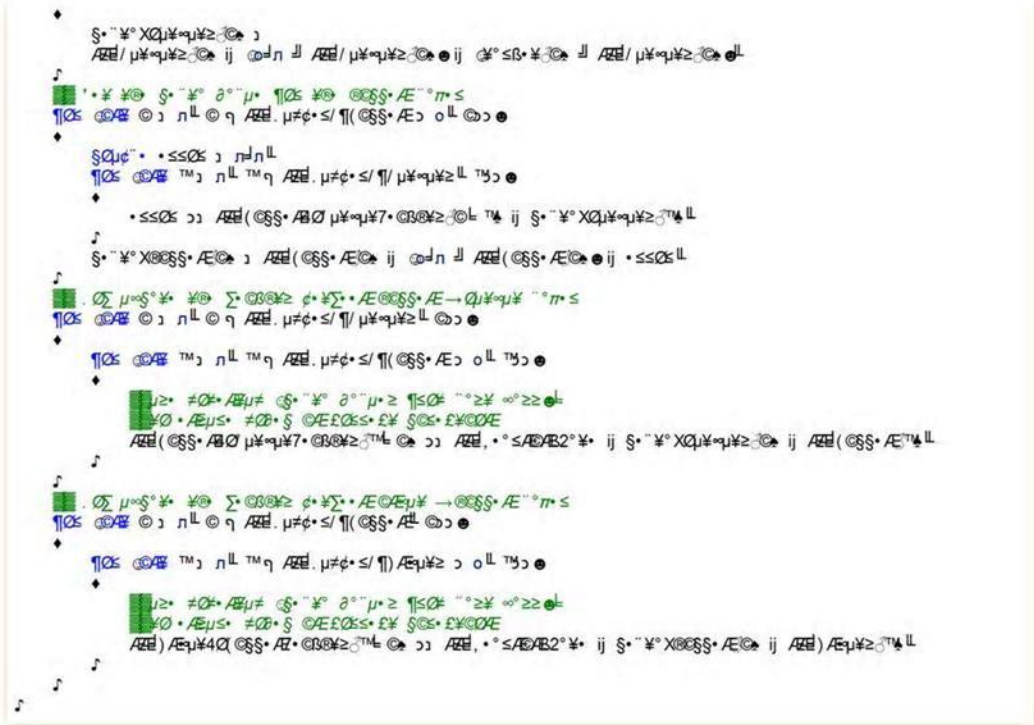
Valleys: Using the rolled ball metaphor, there may well be valleys like this, with steep sides and a gently sloping floor. Gradient descent tends to waste time swooshing up and down each side of the valley (think ball!)

So what can we do about this. Well we add a momentum term, that tends to cancel out the back and forth movements and emphasizes any consistent direction, then this will go down such valleys with gentle bottom-slopes much more successfully (faster)

Starting The Training

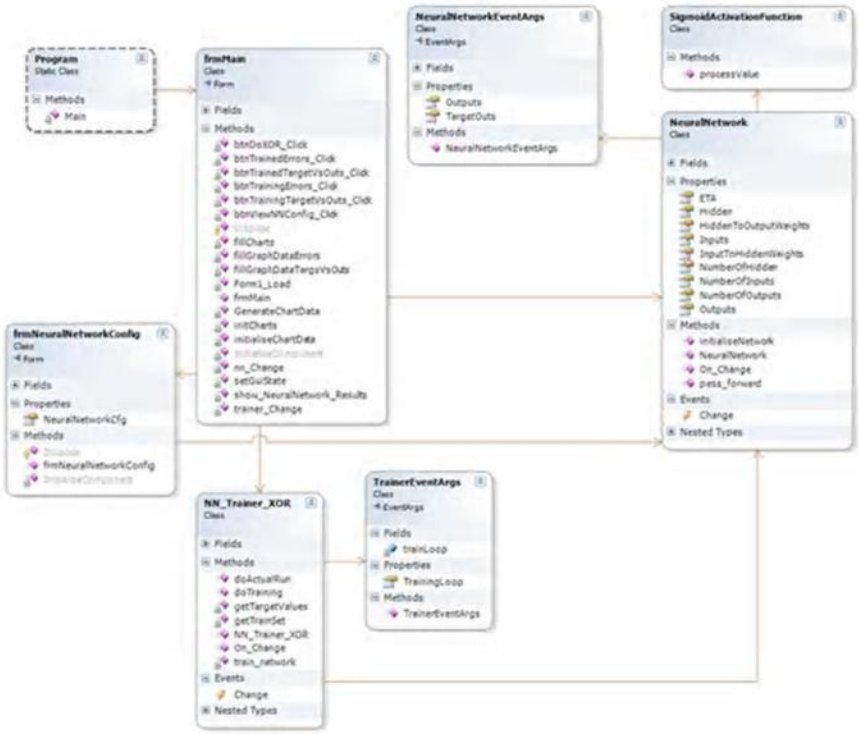
This is probably best demonstrated with a code snippet from the article's actual code:

Hide ¡ Shrink ¡ Copy Code



So Finally The Code

Well, the code for this article looks like the following class diagram (It's Visual Studio 2005 C#, .NET v2.0)



The main classes that people should take the time to look at would be :

- . . X4≤° ©Æ≤X8/ 2 : Trains a Neural Network to solve the XOR problem
- 4≤° ©Æ≤%∂• Æ! ≤ß≥ : Training event args, for use with a GUI
- . • µ≤° ¨ . • ¥∑Ø≤´ : A configurable Neural Network
- . • µ≤° ¨ . • ¥∑Ø≤´ %∂• Æ! ≤ß≥ : Training event args, for use with a GUI
- 3©ß≠Ø©§! £¥©∂ ° ¥©Ø∆Æ¥©Ø∆E A static method to provide the sigmoid activation function

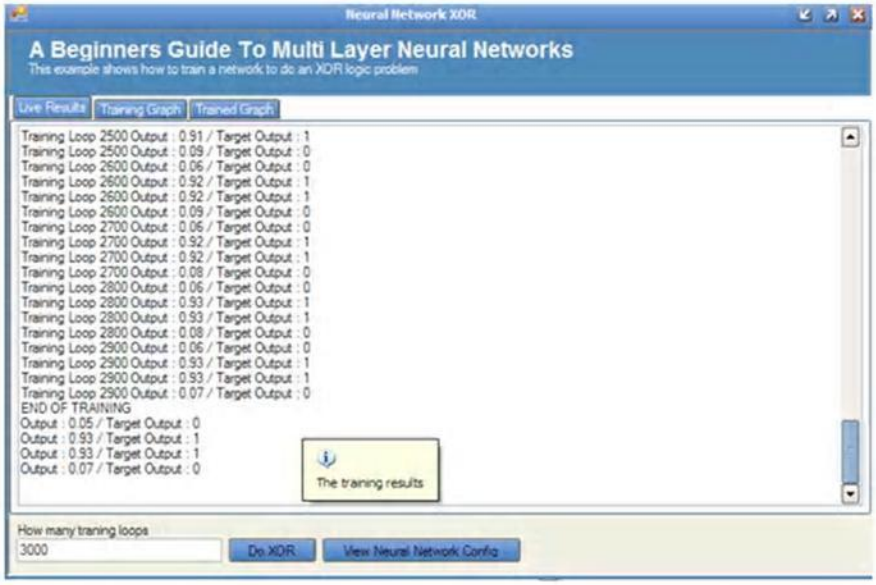
The rest are a GUI I constructed simply to show how it all fits together.

NOTE : the demo project contains all code, so I won't list it here.

Code Demos

The DEMO application attached has 3 main areas which are described below:

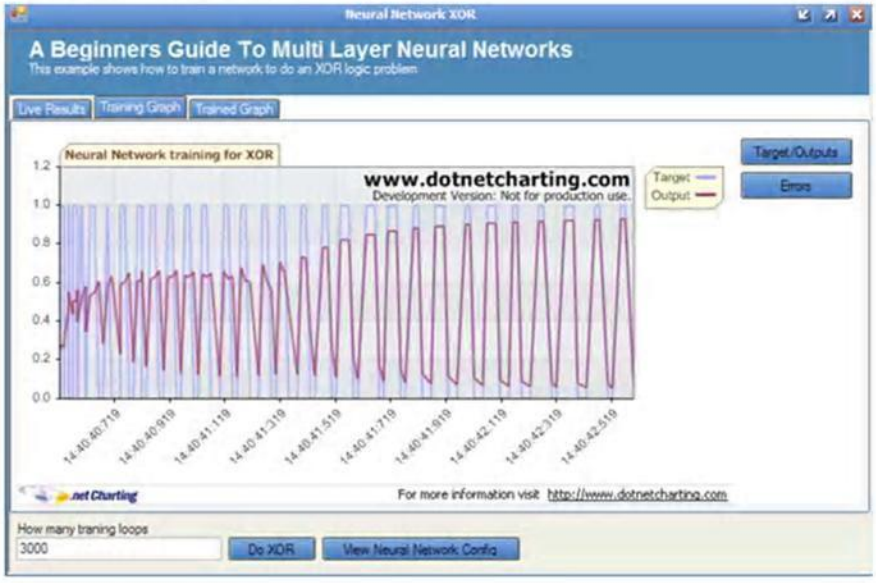
LIVE RESULTS Tab



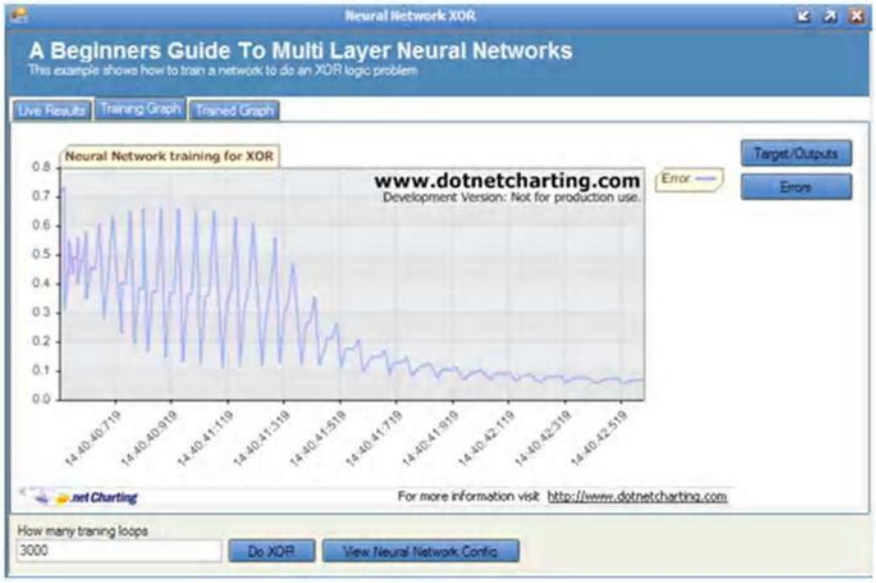
It can be seen that this has very nearly solved the XOR problem (You will probably never get it 100% accurate)

TRAINING RESULTS Tab

Viewing the training phase target/outputs together

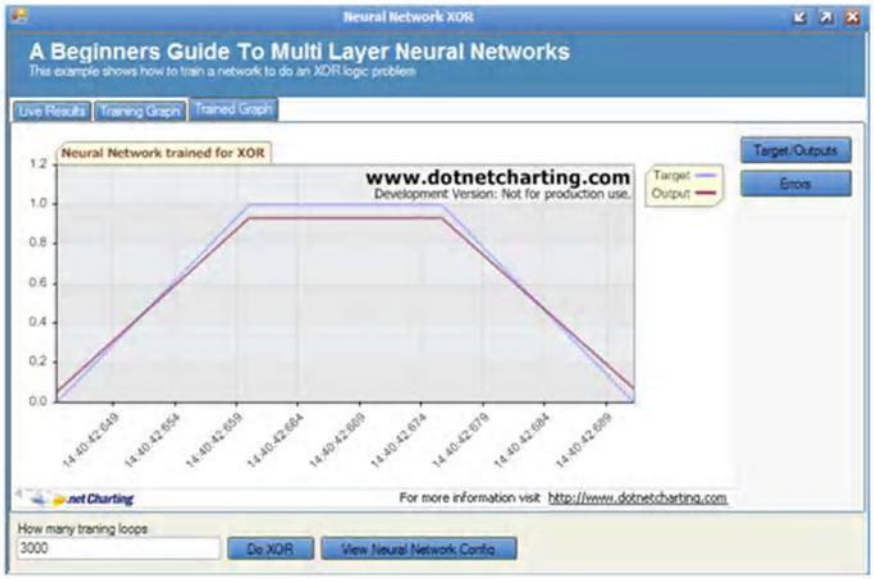


Viewing the training phase errors

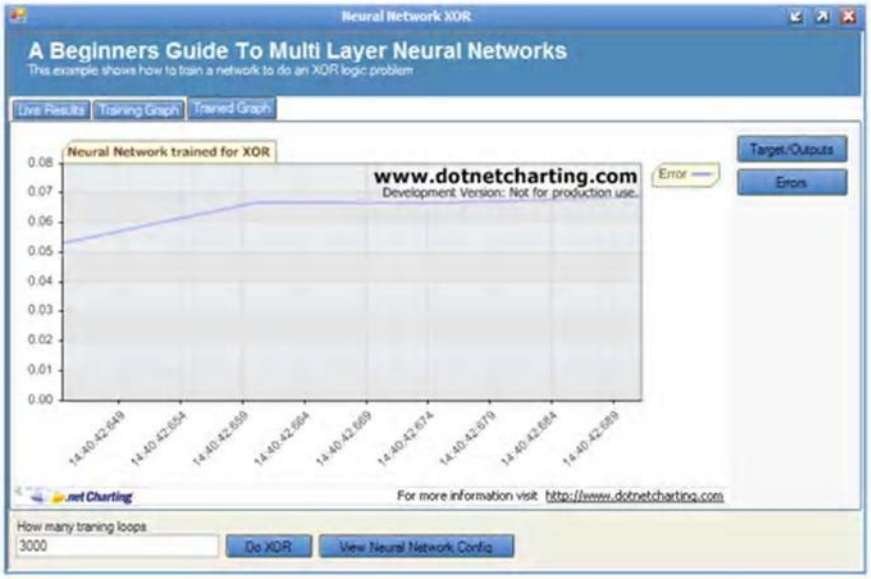


TRAINED RESULTS Tab

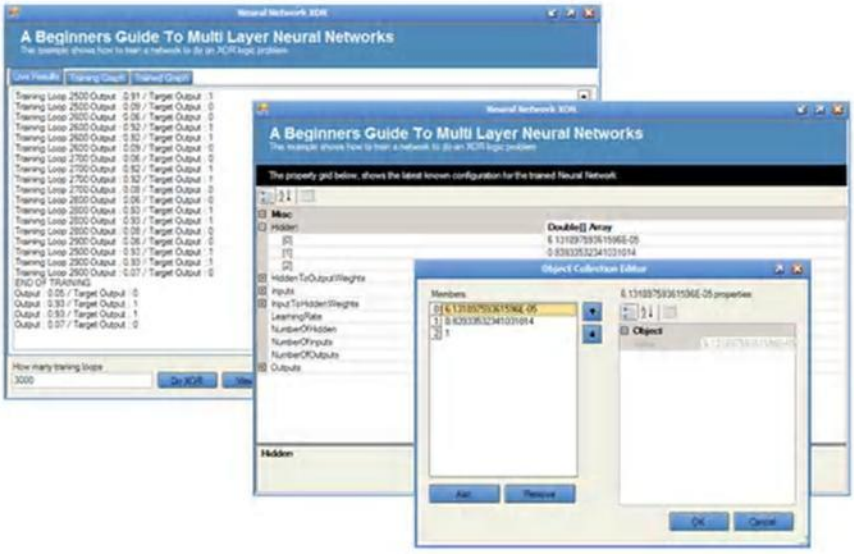
Viewing the trained target/outputs together



Viewing the trained errors



It is also possible to view the Neural Networks final configuration using the "View Neural Network Config" button. If people are interested in what weights the Neural Network ended up with, this is the place to look.



What Do You Think ?

That's it. I would just like to ask, if you liked the article, please vote for it.

Points of Interest

I think AI is fairly interesting, that's why I am taking the time to publish these articles. So I hope someone else finds it interesting, and that it might help further someone's knowledge, as it has my own.

Anyone that wants to look further into AI type stuff, that finds the content of this article a bit basic should check out Andrew Krillovs articles, at Andrew Krillov CP articles as his are more advanced, and very good. In fact anything Andrew seems to do, is very good.

History

- v1.0 24/11/06

Bibliography

- Artificial Intelligence 2nd edition, Elaine Rich / Kevin Knight. McGraw Hill Inc.
- Artificial Intelligence, A Modern Approach, Stuart Russell / Peter Norvig. Prentice Hall.

License

Share

About the Author

Sacha Barber

Software Developer (Senior)
United Kingdom

I currently hold the following qualifications (amongst others, I also studied Music Technology and Electronics, for my sins)

- MSc (Passed with distinctions), in Information Technology for E-Commerce
- BSc Hons (1st class) in Computer Science & Artificial Intelligence

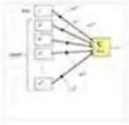
Both of these at Sussex University UK.

Award(s)

I am lucky enough to have won a few awards for Zany Crazy code articles over the years

- Microsoft C# MVP 2015
- Codeproject MVP 2015
- Microsoft C# MVP 2014
- Codeproject MVP 2014
- Microsoft C# MVP 2013
- Codeproject MVP 2013
- Microsoft C# MVP 2012
- Codeproject MVP 2012
- Microsoft C# MVP 2011
- Codeproject MVP 2011
- Microsoft C# MVP 2010
- Codeproject MVP 2010
- Microsoft C# MVP 2009
- Codeproject MVP 2009
- Microsoft C# MVP 2008
- Codeproject MVP 2008
- And numerous codeproject awards which you can see over at my blog

You may also be interested in...



AI: Neural Network for Beginners (Part 3 of 3)

Speed Up Your Git Repository: Introducing Git-Over-FASP

AI : Neural Network for beginners (Part 1 of 3)

Taking COBOL mobile

AI Life

COBOL programmers: Skill up and save timeÂ

Comments and Discussions

You must Sign In to use this message board.

Search Comments

Profile popups Spacing Relaxed Layout Normal Per page 25 Update

First Prev Next

help	Member 11265993	26-Nov-14 22:42
Sigmoid function	Member 10973839	26-Jul-14 23:17
Sigmoid function	Member 10973839	26-Jul-14 23:16
Takes a long time to converge	Member 9401821	2-Mar-14 0:13
Re: Takes a long time to converge	LudemeGames	2-Mar-14 4:38
I have a question,thank you for telling me .	fengyelan	16-Apr-13 22:31
My vote of 5	Nickydo	10-Sep-12 2:30
Part 3?	Mauro Leggieri	5-Apr-09 7:41
Re: Part 3?	Sacha Barber	5-Apr-09 9:55
Re: Part 3?	Mauro Leggieri	6-Apr-09 3:17
About parameterizing the 'momentum' factor	mahabir	23-Sep-08 20:48
Re: About parameterizing the 'momentum' factor	Sacha Barber	23-Sep-08 22:57
Re: About parameterizing the 'momentum' factor	Sacha Barber	23-Sep-08 22:59
Re: About parameterizing the 'momentum' factor	ramesh0285	26-Nov-12 18:14
part 1	gholamabbas Sayyad	18-Sep-08 21:21
Re: part 1	Sacha Barber	18-Sep-08 22:49
Solution for getTrainSet(int idx)	DKHVC	16-Apr-08 21:09
[Message Deleted]	Danny Rodriguez	27-Jan-08 10:05
Hello	MohamadJaber	11-Dec-07 0:33
Erratic Bahaviour?	rampantandroid	15-Oct-07 18:17
Re: Erratic Bahaviour?	rampantandroid	15-Oct-07 19:56
Small Suggestion	dfhgesart	28-Jul-07 16:26
Re: Small Suggestion	Sacha Barber	29-Jul-07 0:35
Excellent!	merlin981	17-May-07 5:31
license?	famousj.dejazzd.com	17-Jan-07 9:32

Last Visit: 31-Dec-99 19:00 ¡ ¡ Last Update: 29-Dec-15 6:41 Refresh 1 2 Next ...

General ¡¡ News ¡¡ Suggestion ¡¡ Question ¡¡ Bug ¡¡ Answer ¡¡ Joke ¡¡ Praise ¡¡ Rant ¡¡ Admin ¡¡

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

NEURAL NETWORKS

by Christos Stergiou and Dimitrios Siganos

Abstract

This report is an introduction to Artificial Neural Networks. The various types of neural networks are explained and demonstrated, applications of neural networks like ANNs in medicine are described, and a detailed historical background is provided. The connection between the artificial and the real thing is also investigated and explained. Finally, the mathematical models involved are presented and demonstrated.

Contents:

1. Introduction to neural networks

1.1 What is a Neural Network?

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a

specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well.

1.2 Historical background

Neural network simulations appear to be a recent development. However, this field was established before the advent of computers, and has survived at least one major setback and several eras.

Many important advances have been boosted by the use of inexpensive computer emulations. Following an initial period of enthusiasm, the field survived a period of frustration and disrepute. During this period when funding and professional support was minimal, important advances were made by relatively few reserchers. These pioneers were able to develop convincing technology which surpassed the limitations identified by Minsky and Papert. Minsky and Papert, published a book (in 1969) in which they summed up a general feeling of frustration (against neural networks) among researchers, and was thus accepted by most without further analysis. Currently, the neural network field enjoys a resurgence of interest and a corresponding increase in funding.

For a more detailed description of the history click here

The first artificial neuron was produced in 1943 by the neurophysiologist Warren McCulloch and the logician Walter Pits. But the technology available at that time did not allow them to do too much.

1.3 Why use neural networks?

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyse. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

1.4 Neural networks versus conventional computers

Neural networks take a different approach to problem solving than that of conventional computers. Conventional computers use an algorithmic approach i.e. the computer follows a set of instructions in order to solve a problem. Unless the specific steps that the computer needs to follow are known the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. But computers would be so much more useful if they could do things that we don't exactly know how to do.

Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected processing elements(neurones) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform a specific task. The examples must be selected carefully otherwise useful time is wasted or even worse the network might be functioning incorrectly. The disadvantage is that because the network finds out how to solve the problem by itself, its operation can be unpredictable.

On the other hand, conventional computers use a cognitive approach to problem solving; the way the problem is to solved must be known and stated in small unambiguous instructions. These instructions are then converted to a high level language program and then into machine code that the computer can understand. These machines are totally predictable; if anything goes wrong is due to a software or hardware fault.

Neural networks and conventional algorithmic computers are not in competition but complement each other. There are tasks are more suited to an algorithmic approach like arithmetic operations and tasks that are more suited to neural networks. Even more, a large number of tasks, require systems that use a combination of the two approaches (normally a conventional computer is used to supervise the neural network) in order to perform at maximum efficiency.

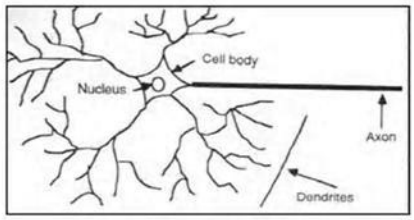
Neural networks do not perform miracles. But if used sensibly they can produce some amazing results.

Back to Contents

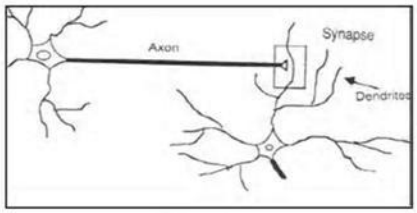
2. Human and Artificial Neurones - investigating the similarities

2.1 How the Human Brain Learns?

Much is still unknown about how the brain trains itself to process information, so theories abound. In the human brain, a typical neuron collects signals from others through a host of fine structures called *dendrites*. The neuron sends out spikes of electrical activity through a long, thin stand known as an *axon*, which splits into thousands of branches. At the end of each branch, a structure called a *synapse* converts the activity from the axon into electrical effects that inhibit or excite activity from the axon into electrical effects that inhibit or excite activity in the connected neurones. When a neuron receives excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.



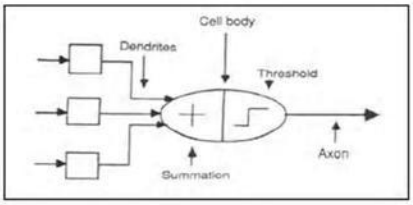
Components of a neuron



The synapse

2.2 From Human Neurones to Artificial Neurones

We conduct these neural networks by first trying to deduce the essential features of neurones and their interconnections. We then typically program a computer to simulate these features. However because our knowledge of neurones is incomplete and our computing power is limited, our models are necessarily gross idealisations of real networks of neurones.



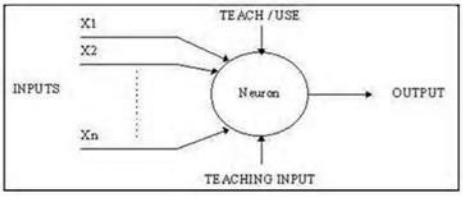
The neuron model

Back to Contents

3. An engineering approach

3.1 A simple neuron

An artificial neuron is a device with many inputs and one output. The neuron has two modes of operation; the training mode and the using mode. In the training mode, the neuron can be trained to fire (or not), for particular input patterns. In the using mode, when a taught input pattern is detected at the input, its associated output becomes the current output. If the input pattern does not belong in the taught list of input patterns, the firing rule is used to determine whether to fire or not.



A simple neuron

3.2 Firing rules

The firing rule is an important concept in neural networks and accounts for their high flexibility. A firing rule determines how one calculates whether a neuron should fire for any input pattern. It relates to all the input patterns, not only the ones on which the node was trained.

A simple firing rule can be implemented by using Hamming distance technique. The rule goes as follows:

> Take a collection of training patterns for a node, some of which cause it to fire (the 1-taught set of patterns) and others which prevent it from doing so (the 0-taught set). Then the patterns not in the collection cause the node to fire if, on comparison , they have more input elements in common with the 'nearest' pattern in the 1-taught set than with the 'nearest' pattern in the 0-taught set. If there is a tie, then the pattern remains in the undefined state.

For example, a 3-input neuron is taught to output 1 when the input (X1,X2 and X3) is 111 or 101 and to output 0 when the input is 000 or 001. Then, before applying the firing rule, the truth table is;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0/1	0/1	0/1	1	0/1	1

As an example of the way the firing rule is applied, take the pattern 010. It differs from 000 in 1 element, from 001 in 2 elements, from 101 in 3 elements and from 111 in 2 elements. Therefore, the 'nearest' pattern is 000 which belongs in the 0-taught set. Thus the firing rule requires that the neuron should not fire when the input is 001. On the other hand, 011 is equally distant from two taught patterns that have different outputs and thus the output stays undefined (0/1).

By applying the firing in every column the following truth table is obtained;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0	0/1	0/1	1	1	1

The difference between the two truth tables is called the *generalisation of the neuron*. Therefore the firing rule gives the neuron a sense of similarity and enables it to respond 'sensibly' to patterns not seen during training.

3.3 Pattern Recognition - an example

An important application of neural networks is pattern recognition. Pattern recognition can be implemented by using a feed-forward (figure 1) neural network that has been trained accordingly. During training, the network is trained to associate outputs with input patterns. When the network is used, it identifies the input pattern and tries to output the associated output pattern. The power of neural networks comes to life when a pattern that has no output associated with it, is given as an input. In this case, the network gives the output that corresponds to a taught input pattern that is least different from the given pattern.

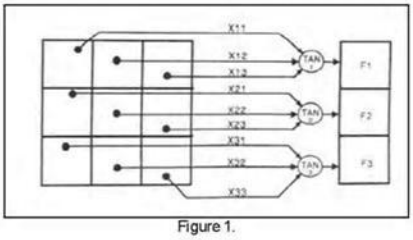


Figure 1.

For example:
The network of figure 1 is trained to recognise the patterns T and H. The associated patterns are all black and all white respectively as shown below.

If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurones after generalisation are;

X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Top neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

Middle neuron

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1

OUT:	1	0	1	1	0	0	1	0

Bottom neuron

From the tables it can be seen the following associasions can be extracted:

In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern.

Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.

Here, the top row is 2 errors away from the a T and 3 from an H. So the top output is black. The middle row is 1 error away from both T and H so the output is random. The bottom row is 1 error away from T and 2 away from H. Therefore the output is black. The total output of the network is still in favour of the T shape.

3.4 A more complicated neuron

The previous neuron doesn't do anything that conventional conventional computers don't do already. A more sophisticated neuron (figure 2) is the McCulloch and Pitts model (MCP). The difference from the previous model is that the inputs are 'weighted', the effect that each input has at decision making is dependent on the weight of the particular input. The weight of an input is a number which when multiplied with the input gives the weighted input. These weighted inputs are then added together and if they exceed a pre-set threshold value, the neuron fires. In any other case the neuron does not fire.

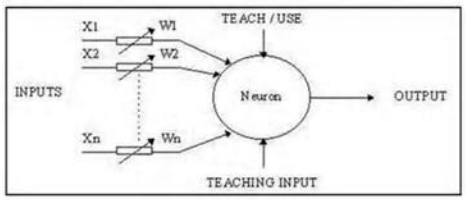


Figure 2. An MCP neuron

In mathematical terms, the neuron fires if and only if;

$$X_1W_1 + X_2W_2 + X_3W_3 + \ldots > T$$

The addition of input weights and of the threshold makes this neuron a very flexible and powerful one. The MCP neuron has the ability to adapt to a particular situation by changing its weights and/or threshold. Various algorithms exist that cause the neuron to 'adapt'; the most used ones are the Delta rule and the back error propagation. The former is used in feed-forward networks and the latter in feedback networks.

Back to Contents

4 Architecture of neural networks

4.1 Feed-forward networks

Feed-forward ANNs (figure 1) allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organisation is also referred to as bottom-up or top-down.

4.2 Feedback networks

Feedback networks (figure 1) can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organisations.

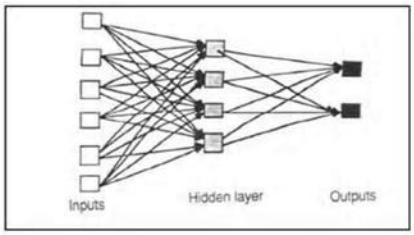


Figure 4.1 An example of a simple feedforward network

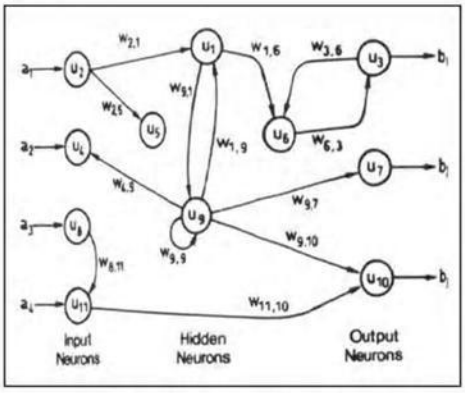


Figure 4.2 An example of a complicated network

4.3 Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units: a layer of "**input**" units is connected to a layer of "**hidden**" units, which is connected to a layer of "**output**" units. (see Figure 4.1)

- The activity of the input units represents the raw information that is fed into the network.
- The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
- The behaviour of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This simple type of network is interesting because the hidden units are free to construct their own representations of the input. The weights between the input and hidden units determine when each hidden unit is active, and so by modifying these weights, a hidden unit can choose what it represents.

We also distinguish single-layer and multi-layer architectures. The single-layer organisation, in which all units are connected to one another, constitutes the most general case and is of more potential computational power than hierarchically structured multi-layer organisations. In multi-layer networks, units are often numbered by layer, instead of following a global numbering.

4.4 Perceptrons

The most influential work on neural nets in the 60's went under the heading of 'perceptrons' a term coined by Frank Rosenblatt. The perceptron (figure 4.4) turns out to be an MCP model (neuron with weighted inputs) with some additional, fixed, pre--processing. Units labelled A1, A2, Aj , Ap are called association units and their task is to extract specific, localised featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They were mainly used in pattern recognition even though their capabilities extended a lot more.

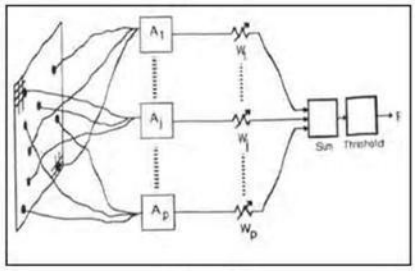


Figure 4.4

In 1969 Minsky and Papert wrote a book in which they described the limitations of single layer Perceptrons. The impact that the book had was tremendous and caused a lot of neural network researchers to loose their interest. The book was very well written and showed mathematically that *single layer* perceptrons could not do some basic pattern recognition operations like determining the parity of a shape or determining whether a shape is connected or not. What they did not realised, until the 80's, is that given the appropriate training, multilevel perceptrons can do these operations.

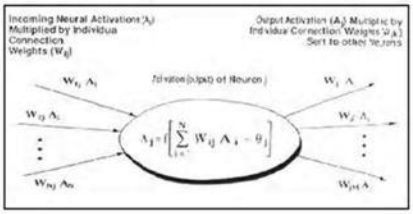
Back to Contents

5. The Learning Process

The memorisation of patterns and the subsequent response of the network can be categorised into two general paradigms:

- **associative mapping** in which the network learns to produce a particular pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associtive mapping can generally be broken down into two mechanisms:
  - *auto-association*: an input pattern is associated with itself and the states of input and output units coincide. This is used to provide pattern completition, ie to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network actually stores pairs of patterns building an association between two sets of patterns.
  - *hetero-association*: is related to two recall mechanisms:
    - *nearest-neighbour* recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and
    - *interpolative* recall, where the output pattern is a similarity dependent interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, ie when there is a fixed set of categories into which the input patterns are to be classified.
- **regularity detection** in which units learn to respond to particular properties of the input patterns. Whereas in asssociative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network posseses knowledge which is contained in the values of the connections weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights.



Information is stored in the weight matrix W of a neural network. Learning is the determination of the weights. Following the way learning is performed, we can distinguish two major categories of neural networks:

- **fixed networks** in which the weights cannot be changed, ie dW/dt=0. In such networks, the weights are fixed a priori according to the problem to solve.
- **adaptive networks** which are able to change their weights, ie dW/dt not= 0.

All learning methods used for adaptive neural networks can be classified into two major categories:

- **Supervised learning** which incorporates an external teacher, so that each output unit is told what its desired response to input signals ought to be. During the learning process global information may be required. Paradigms of supervised learning include error-correction learning, reinforcement learning and stochastic learning.
  An important issue conserning supervised learning is the problem of error convergence, ie the minimisation of error between the desired and computed unit values. The aim is to determine a set of weights which minimises the error. One well-known method, which is common to many learning paradigms is the least mean square (LMS) convergence.
- **Unsupervised learning** uses no external teacher and is based upon only local information. It is also referred to as self-organisation, in the sense that it self-organises data presented to the network and detects their emergent collective properties. Paradigms of unsupervised learning are Hebbian lerning and competitive learning.
  Ano2.2 From Human Neurones to Artificial Neuronesther aspect of learning concerns the distinction or not of a seperate phase, during which the network is trained, and a subsequent operation phase. We say that a neural network learns off-line if the learning phase and the operation phase are distinct. A neural network learns on-line if it learns and operates at the same time. Usually, supervised learning is performed off-line, whereas usupervised learning is performed on-line.

5.1 Transfer Function

The behaviour of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

- linear (or ramp)
- threshold
- sigmoid

For **linear units**, the output activity is proportional to the total weighted output.

For **threshold units**, the output is set at one of two levels, depending on whether the total input is greater than or less than some threshold value.

For **sigmoid units**, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurones than do linear or threshold units, but all three must be considered rough approximations.

To make a neural network that performs some specific task, we must choose how the units are connected to one another (see figure 4.1), and we must set the weights on the connections appropriately. The connections determine whether it is possible for one unit to influence another. The weights specify the strength of the influence.

We can teach a three-layer network to perform a particular task by using the following procedure:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

5.2 An Example to illustrate the above teaching procedure:

Assume that we want a network to recognise hand-written digits. We might use an array of, say, 256 sensors, each recording the presence or absence of ink in a small area of a single digit. The network would therefore need 256 input units (one for each sensor), 10 output units (one for each kind of digit) and a number of hidden units.

For each kind of digit recorded by the sensors, the network should produce high activity in the appropriate output unit and low activity in the other output units.

To train the network, we present an image of a digit and compare the actual activity of the 10 output units with the desired activity. We then calculate the error, which is defined as the square of the difference between the actual and the desired activities. Next we change the weight of each connection so as to reduce the error.We repeat this training process for many different images of each different images of each kind of digit until the network classifies every image correctly.

To implement this procedure we need to calculate the error derivative for the weight (EW) in order to change the weight by an amount that is proportional to the rate at which the error changes as the weight is changed. One way to calculate the EW is to perturb a weight slightly and observe how the error changes. But that method is inefficient because it requires a separate perturbation for each of the many weights.

Another way to calculate the EW is to use the Back-propagation algorithm which is described below, and has become nowadays one of the most important tools for training neural networks. It was developed independently by two teams, one (Fogelman-Soulie, Gallinari and Le Cun) in France, the other (Rumelhart, Hinton and Williams) in U.S.

5.3 The Back-Propagation Algorithm

In order to train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and the actual output is reduced. This process requires that the neural network compute the error derivative of the weights (**EW**). In other words, it must calculate how the error changes as each weight is increased or decreased slightly. The back propagation algorithm is the most widely used method for determining the **EW**.

The back-propagation algorithm is easiest to understand if all the units in the network are linear. The algorithm computes each **EW** by first computing the **EA**, the rate at which the error changes as the activity level of a unit is changed. For output units, the **EA** is simply the difference between the actual and the desired output. To compute the **EA** for a hidden unit in the layer just before the output layer, we first identify all the weights between that hidden unit and the output units to which it is connected. We then multiply those weights by the **EA**s of those output units and add the products. This sum equals the **EA** for the chosen hidden unit. After calculating all the **EA**s in the hidden layer just before the output layer, we can compute in like fashion the **EA**s for other layers, moving from layer to layer in a direction opposite to the way activities propagate through the network. This is what gives back propagation its name. Once the **EA** has been computed for a unit, it is straight forward to compute the **EW** for each incoming connection of the unit. The **EW** is the product of the EA and the activity through the incoming connection.

Note that for non-linear units, (see Appendix C) the back-propagation algorithm includes an extra step. Before back-propagating, the **EA** must be converted into the **EI**, the rate at which the error changes as the total input received by a unit is changed.

Back to Contents

6. Applications of neural networks

6.1 Neural Networks in Practice

Given this description of neural networks and how they work, what real world applications are they suited for? Neural networks have broad applicability to real world business problems. In fact, they have already been successfully applied in many industries.

Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs including:

- sales forecasting
- industrial process control
- customer research
- data validation
- risk management
- target marketing

But to give you some more specific examples; ANN are also used in the following specific paradigms: recognition of speakers in communications; diagnosis of hepatitis; recovery of telecommunications from faulty software; interpretation of multimeaning Chinese words; undersea mine detection; texture analysis; three-dimensional object recognition; hand-written word recognition; and facial recognition.

6.2 Neural networks in medicine

Artificial Neural Networks (ANN) are currently a 'hot' research area in medicine and it is believed that they will receive extensive application to biomedical systems in the next few years. At the moment, the research is mostly on modelling parts of the human body and recognising diseases from various scans (e.g. cardiograms, CAT scans, ultrasonic scans, etc.).

Neural networks are ideal in recognising diseases using scans since there is no need to provide a specific algorithm on how to identify the disease. Neural networks learn by example so the details of how to recognise the disease are not needed. What is needed is a set of examples that are representative of all the variations of the disease. The quantity of examples is not as important as the 'quantity'. The examples need to be selected very carefully if the system is to perform reliably and efficiently.

6.2.1 Modelling and Diagnosing the Cardiovascular System

Neural Networks are used experimentally to model the human cardiovascular system. Diagnosis can be achieved by building a model of the cardiovascular system of an individual and comparing it with the real time physiological measurements taken from the patient. If this routine is carried out regularly, potential harmful medical conditions can be detected at an early stage and thus make the process of combating the disease much easier.

A model of an individual's cardiovascular system must mimic the relationship among physiological variables (i.e., heart rate, systolic and diastolic blood pressures, and breathing rate) at different physical activity levels. If a model is adapted to an individual, then it becomes a model of the physical condition of that individual. The simulator will have to be able to adapt to the features of any individual without the supervision of an expert. This calls for a neural network.

Another reason that justifies the use of ANN technology, is the ability of ANNs to provide sensor fusion which is the combining of values from several different sensors. Sensor fusion enables the ANNs to learn complex relationships among the individual sensor values, which would otherwise be lost if the values were individually analysed. In medical modelling and diagnosis, this implies that even though each sensor in a set may be sensitive only to a specific physiological variable, ANNs are capable of detecting complex medical conditions by fusing the data from the individual biomedical sensors.

6.2.2 Electronic noses

ANNs are used experimentally to implement electronic noses. Electronic noses have several potential applications in telemedicine. Telemedicine is the practice of medicine over long distances via a communication link. The electronic nose would identify odours in the remote surgical environment. These identified odours would then be electronically transmitted to another site where an door generation system would recreate them. Because the sense of smell can be an important sense to the surgeon, telesmell would enhance telepresent surgery.
For more information on telemedicine and telepresent surgery click here.

6.2.3 Instant Physician

An application developed in the mid-1980s called the "instant physician" trained an autoassociative memory neural network to store a large number of medical records, each of which includes information on symptoms, diagnosis, and treatment for a particular case. After training, the net can be presented with input consisting of a set of symptoms; it will then find the full stored pattern that represents the "best" diagnosis and treatment.

6.3 Neural Networks in business

Business is a diverted field with several general areas of specialisation such as accounting or financial analysis. Almost any neural network application would fit into one business area or financial analysis.
There is some potential for using neural networks for business purposes, including resource allocation and scheduling. There is also a strong potential for using neural networks for database mining, that is, searching for patterns implicit within the explicitly stored information in databases. Most of the funded work in this area is classified as proprietary. Thus, it is not possible to report on the full extent of the work going on. Most work is applying neural networks, such as the Hopfield-Tank network for optimization and scheduling.

6.3.1 Marketing

There is a marketing application which has been integrated with a neural network system. The Airline Marketing Tactician (a trademark abbreviated as AMT) is a computer system made of various intelligent technologies including expert systems. A feedforward neural network is integrated with the AMT and was trained using back-propagation to assist the marketing control of airline seat allocations. The adaptive neural approach was amenable to rule expression. Additionaly, the application's environment changed rapidly and constantly, which required a continuously adaptive solution. The system is used to monitor and recommend booking advice for each departure. Such information has a direct impact on the profitability of an airline and can provide a technological advantage for users of the system. [Hutchison & Stephens, 1987]

While it is significant that neural networks have been applied to this problem, it is also important to see that this intelligent technology can be integrated with expert systems and other approaches to make a functional system. Neural networks were used to discover the influence of undefined interactions by the various variables. While these interactions were not defined, they were used by the neural system to develop useful conclusions. It is also noteworthy to see that neural networks can influence the bottom line.

6.3.2 Credit Evaluation

The HNC company, founded by Robert Hecht-Nielsen, has developed several neural network applications. One of them is the Credit Scoring system which increase the profitability of the existing model up to 27%. The HNC neural systems were also applied to mortgage screening. A neural network automated mortgage insurance underwritting system was developed by the Nestor Company. This system was trained with 5048 applications of which 2597 were certified. The data related to property and borrower qualifications. In a conservative mode the system agreed on the underwritters on 97% of the cases. In the liberal model the system agreed 84% of the cases. This is system run on an Apollo DN3000 and used 250K memory while processing a case file in approximately 1 sec.

Back to Contents

7. Conclusion

The computing world has a lot to gain fron neural networks. Their ability to learn by example makes them very flexible and powerful. Furthermore there is no need to devise an algorithm in order to perform a specific task; i.e. there is no need to understand the internal mechanisms of that task. They are also very well suited for real time systems because of their fast responseand computational times which are due to their parallel architecture.

Neural networks also contribute to other areas of research such as neurology and psychology. They are regularly used to model parts of living organisms and to investigate the internal mechanisms of the brain.

Perhaps the most exciting aspect of neural networks is the possibility that some day 'consious' networks might be produced. There is a number of scientists arguing that conciousness is a 'mechanical' property and that 'consious' neural networks are a realistic possibility.

Finally, I would like to state that even though neural networks have a huge potential we will only get the best of them when they are intergrated with computing, AI, fuzzy logic and related subjects.

Back to Contents

References:

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html
3. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
4. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html
5. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht
6. Neural Networks by Eric Davalo and Patrick Naim
7. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
8. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
9. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab. Neural Networks, Eric Davalo and Patrick Naim
10. Assimov, I (1984, 1950), Robot, Ballatine, New York.
11. Electronic Noses for Telemedicine
    http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html
12. Pattern Recognition of Pathology Images
    http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html

Back to Contents

Appendix A - Historical background in detail

The history of neural networks that was described above can be divided into several periods:

1. **First Attempts**: There were some initial simulations using formal logic. McCulloch and Pitts (1943) developed models of neural networks based on their understanding of neurology. These models made several assumptions about how neurons worked. Their networks were based on simple neurons which were considered to be binary devices with fixed thresholds. The results of their model were simple logic functions such as "a or b" and "a and b". Another attempt was by using computer simulations. Two groups (Farley and Clark, 1954; Rochester, Holland, Haibit and Duda, 1956). The first group (IBM reserchers) maintained closed contact with neuroscientists at McGill University. So whenever their models did not work, they consulted the neuroscientists. This interaction established a multidiscilinary trend which continues to the present day.
2. **Promising & Emerging Technology:** Not only was neroscience influential in the development of neural networks, but psychologists and engineers also contributed to the progress of neural network simulations. Rosenblatt (1958) stirred considerable interest and activity in the field when he designed and developed the Perceptron. The Perceptron had three layers with the middle layer known as the association layer.This system could learn to connect or associate a given input to a random output unit.

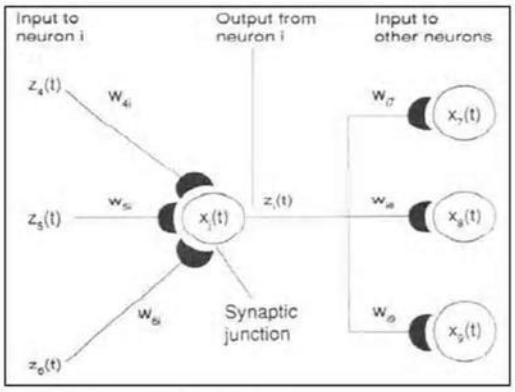
Another system was the ADALINE (ADAptive LInear Element) which was developed in 1960 by Widrow and Hoff (of Stanford University). The ADALINE was an analogue electronic device made from simple components. The method used for learning was different to that of the Perceptron, it employed the Least-Mean-Squares (LMS) learning rule.

3. **Period of Frustration & Disrepute:** In 1969 Minsky and Papert wrote a book in which they generalised the limitations of single layer Perceptrons to multilayered systems. In the book they said: "...our intuitive judgment that the extension (to multilayer systems) is sterile". The significant result of their book was to eliminate funding for research with neural network simulations. The conclusions supported the disenhantment of reserchers in the field. As a result, considerable prejudice against this field was activated.
4. **Innovation:** Although public interest and available funding were minimal, several researchers continued working to develop neuromorphically based computaional methods for problems such as pattern recognition.
   During this period several paradigms were generated which modern work continues to enhance.Grossberg's (Steve Grossberg and Gail Carpenter in 1988) influence founded a school of thought which explores resonating algorithms. They developed the ART (Adaptive Resonance Theory) networks based on biologically plausible models. Anderson and Kohonen developed associative techniques independent of each other. Klopf (A. Henry Klopf) in 1972, developed a basis for learning in artificial neurons based on a biological principle for neuronal learning called heterostasis.
   Werbos (Paul Werbos 1974) developed and used the back-propagation learning method, however several years passed before this approach was popularized. Back-propagation nets are probably the most well known and widely applied of the neural networks today. In essence, the back-propagation net. is a Perceptron with multiple layers, a different thershold function in the artificial neuron, and a more robust and capable learning rule.
   Amari (A. Shun-Ichi 1967) was involved with theoretical developments: he published a paper which established a mathematical theory for a learning basis (error-correction method) dealing with adaptive patern classification. While Fukushima (F. Kunihiko) developed a step wise trained multilayered neural network for interpretation of handwritten characters. The original network was published in 1975 and was called the Cognitron.
5. **Re-Emergence:** Progress during the late 1970s and early 1980s was important to the re-emergence on interest in the neural network field. Several factors influenced this movement. For example, comprehensive books and conferences provided a forum for people in diverse fields with specialized technical languages, and the response to conferences and publications was quite positive. The news media picked up on the increased activity and tutorials helped disseminate the technology. Academic programs appeared and courses were inroduced at most major Universities (in US and Europe). Attention is now focused on funding levels throughout Europe, Japan and the US and as this funding becomes available, several new commercial with applications in industry and finacial institutions are emerging.
6. **Today:** Significant progress has been made in the field of neural networks-enough to attract a great deal of attention and fund further research. Advancement beyond current commercial applications appears to be possible, and research is advancing the field on many fronts. Neurally based chips are emerging and applications to complex problems developing. Clearly, today is a period of transition for neural network technology.

Back to Contents

Appendix B - The back-propagation Algorithm - a mathematical approach

Units are connected to one another. Connections correspond to the edges of the underlying directed graph. There is a real number associated with each connection, which is called the weight of the connection. We denote by W_{ij} the weight of the connection from unit u_i to unit u_j. It is then convenient to represent the pattern of connectivity in the network by a weight matrix W whose elements are the weights W_{ij}. Two types of connection are usually distinguished: excitatory and inhibitory. A positive weight represents an excitatory connection whereas a negative weight represents an inhibitory connection. The pattern of connectivity characterises the architecture of the network.



A unit in the output layer determines its activity by following a two step procedure.

- First, it computes the total weighted input x_j, using the formula:

$$x_j = \sum_i y_i w_{ij}$$

  where y_i is the activity level of the jth unit in the previous layer and W_{ij} is the weight of the connection between the ith and the jth unit.

- Next, the unit calculates the activity y_j using some function of the total weighted input. Typically we use the sigmoid function:

$$y_j = \frac{1}{1+e^{-x_j}}$$

Once the activities of all output units have been determined, the network computes the error E, which is defined by the expression:

$$E = \frac{1}{2}\sum_i (y_i - d_i)^2$$

where y_i is the activity level of the jth unit in the top layer and d_i is the desired output of the jth unit.

The back-propagation algorithm consists of four steps:

1. Compute how fast the error changes as the activity of an output unit is changed. This error derivative (EA) is the difference between the actual and the desired activity.

$$EA_j = \frac{\partial E}{\partial y_j} = y_j - d_j$$

2. Compute how fast the error changes as the total input received by an output unit is changed. This quantity (EI) is the answer from step 1 multiplied by the rate at which the output of a unit changes as its total input is changed.

$$EI_j = \frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \times \frac{dy_j}{dx_j} = EA_j y_j (1 - y_j)$$

3. Compute how fast the error changes as a weight on the connection into an output unit is changed. This quantity (EW) is the answer from step 2 multiplied by the activity level of the unit from which the connection emanates.

$$EW_{ij} = \frac{\partial E}{\partial W_{ij}} = \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial W_{ij}} = EI_j y_i$$

4. Compute how fast the error changes as the activity of a unit in the previous layer is changed. This crucial step allows back propagation to be applied to multilayer networks. When the activity of a unit in the previous layer changes, it affects the activites of all the output units to which it is connected. So to compute the overall effect on the error, we add together all these seperate effects on output units. But each effect is simple to calculate. It is the answer in step 2 multiplied by the weight on the connection to that output unit.

$$EA_i = \frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial x_j} \times \frac{\partial x_j}{\partial y_i} = \sum_j EI_j W_{ij}$$

By using steps 2 and 4, we can convert the EAs of one layer of units into EAs for the previous layer. This procedure can be repeated to get the EAs for as many previous layers as desired. Once we know the EA of a unit, we can use steps 2 and 3 to compute the EWs on its incoming connections.

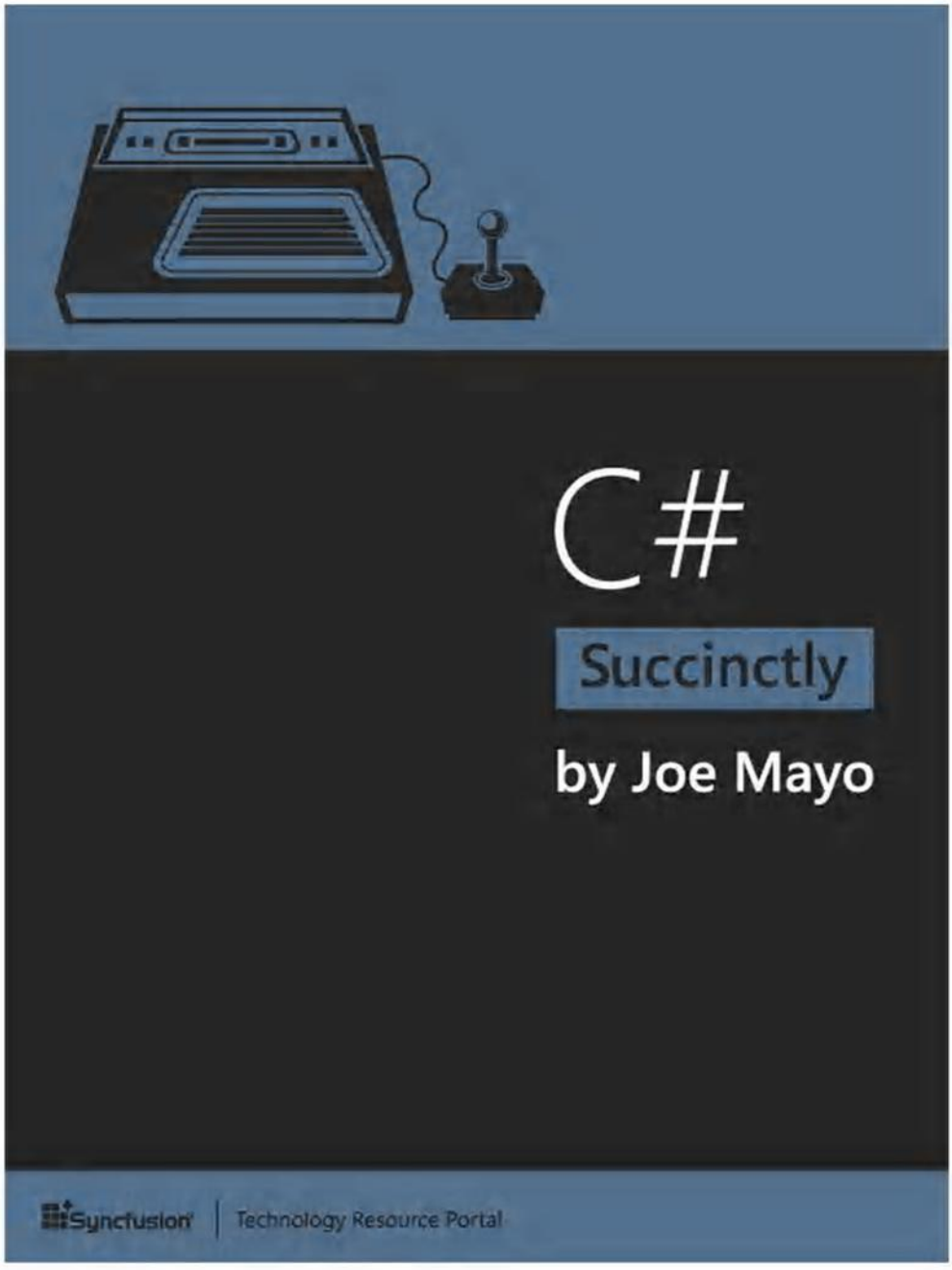
Back to Contents

Appendix C - References used throughout the review

1. An introduction to neural computing. Aleksander, I. and Morton, H. 2nd edition
2. Neural Networks at Pacific Northwest National Laboratory
   http://www.emsl.pnl.gov:2080/docs/cie/neural/neural.homepage.html
3. Artificial Neural Networks in Medicine
   http://www.emsl.pnl.gov:2080/docs/cie/techbrief/NN.techbrief.ht
4. Industrial Applications of Neural Networks (research reports Esprit, I.F.Croall, J.P.Mason)
5. A Novel Approach to Modelling and Diagnosing the Cardiovascular System
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.wcnn95.abs.html
6. Electronic Noses for Telemedicine
   http://www.emsl.pnl.gov:2080/docs/cie/neural/papers2/keller.ccc95.abs.html
7. An Introduction to Computing with Neural Nets (Richard P. Lipmann, IEEE ASSP Magazine, April 1987)
8. Pattern Recognition of Pathology Images
   http://kopernik-eth.npac.syr.edu:1200/Task4/pattern.html
9. Developments in autonomous vehicle navigation. Stefan Neuber, Jos Nijhuis, Lambert Spaanenburg. Institut fur Mikroelektronik Stuttgart, Allmandring 30A, 7000 Stuttgart-80
10. Klimasauskas, CC. (1989). The 1989 Neuro Computing Bibliography. Hammerstrom, D. (1986). A Connectionist/Neural Network Bibliography.
11. DARPA Neural Network Study (October, 1987-February, 1989). MIT Lincoln Lab.
12. Neural Networks, Eric Davalo and Patrick Naim.
13. Assimov, I (1984, 1950), Robot, Ballatine, New York.
14. Learning internal representations by error propagation by Rumelhart, Hinton and Williams (1986).
15. Alkon, D.L 1989, Memory Storage and Neural Systems, Scientific American, July, 42-50
16. Minsky and Papert (1969) Perceptrons, An introduction to computational geometry, MIT press, expanded edition.
17. Neural computers, NATO ASI series, Editors: Rolf Eckmiller Christoph v. d. Malsburg

Back to Contents

Friday, January 01, 2016
1:28 AM



C# Succinctly

By
Joe Mayo

Foreword by Daniel Jebaraj

Technical Reviewer: Stephen Haunts
Copy Editor: Ben Ball
Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.
Proofreader: Graham High, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Joe Mayo is an author, a consultant at Mayo Software, LLC, and an instructor who specializes in Microsoft .NET technology. Joe has written several books, including *C# Unleashed* (Sams) and *LINQ Programming* (McGraw-Hill), and coauthored *ASP.NET 2.0 MVP Hacks and Tips* (Wrox). His articles have been published in CODE Magazine and the online publications Inform IT and C# Station.

Joe is a regular presenter on .NET topics and has received multiple Microsoft Visual C# MVP awards. His open source project, LINQ to Twitter, is hosted on GitHub, and you can read his blog at Geeks with Blogs. You can find Joe on Twitter as @JoeMayo.

Chapter 1 Introducing C# and .NET

Welcome to *C# Succinctly*. True to the *Succinctly* series concept, this book is very focused on a single topic: the C# programming language. I might briefly mention some technologies that you can write with C# or explain how a feature fits into those technologies, but the whole of this book is about helping you become familiar with C# syntax.

In this chapter, I'll start with some introductory information and then jump straight into a simple C# program.

What can I do with C#?

C# is a general purpose, object-oriented, component-based programming language. As a general purpose language, you have a number of ways to apply C# to accomplish many different tasks. You can build web applications with ASP.NET, desktop applications with Windows Presentation Foundation (WPF), or build mobile applications for Windows Phone. Other applications include code that runs in the cloud via Windows Azure, and iOS, Android, and Windows Phone support with the Xamarin platform. There might be times when you need a different language, like C or C++, to communicate with hardware or real-time systems. However, from a general programming perspective, you can do a lot with C#.

What is .NET?

.NET is a platform that includes languages, a runtime, and framework libraries, allowing developers to create many types of applications. C# is one of the .NET languages, which also includes Visual Basic, F#, C++, and more.

The runtime is more formally named the Common Language Runtime (CLR). Programming languages that target the CLR compile to an Intermediate Language (IL). The CLR itself is a virtual machine that runs IL and provides many services such as memory management, garbage collection, exception management, security, and more.

The Framework Class Library (FCL) is a set of reusable code that provides both general services and technology-specific platforms. The general services include essential types such as collections, cryptography, networking, and more. In addition to general classes, the FCL includes technology-specific platforms like ASP.NET, WPF, web services, and more. The value the FCL offers is to have common components available for reuse, saving time and money without needing to write that code yourself.

There's a huge ecosystem of open-source and commercial software that relies on and supports .NET. If you visit CodePlex, GitHub, or any other open-source code repository site, you'll see a multitude of projects written in C#. Commercial offerings include tools and services that help you build code, manage systems, and offer applications. Syncfusion is part of this ecosystem, offering reusable components for many of the .NET technologies I have mentioned.

Writing, Running, and Deploying a C# Program

The previous section described plenty of great things you can do with C#, but most of them are so detailed that they require their own book. To stay focused on the C# programming language, the code in this book will be for the console application. A console application runs on the command line, which you'll learn about in this section. You can write your code with any editor, but this book uses Visual Studio.

Note: The code samples in this book can be downloaded at https://bitbucket.org/syncfusiontech/c-succinctly.

Starting a New Program

You'll need an editor or Integrated Development Environment (IDE) to write code. Microsoft offers Visual Studio (VS), which is available via Community Edition as a free download for training and individual purposes (https://www.visualstudio.com/en-us/products/vs-2015-product-editions.aspx). There are other development tools, but you can also use any editor, including Notepad. Notepad++ is another editor that does syntax highlighting, but there are many more available. Essentially, you just need the ability to type a text document. Pick your editor or IDE of choice and it will work for all programs in this book.

Note: You need to use Visual Studio 2015 to compile the samples in this book.

To get started, we need a program to run. In VS, select **File** > **New** > **Project**, then select **Installed** > **Templates** > **Visual C#** in the tree on the left, and finally select the **Console Application** project type. Name the solution **Chapter01**, name the project **Greetings**, set the location to your preference, and click **OK**. This will create a new solution for you. Delete the **Program.cs** file and add a **Greetings.cs** file. In any text editor, just create a file named **Greetings.cs**. The following is a C# program that prints a greeting to the command line.

```
using System;

class Greetings
{
    static void Main()
    {
        Console.WriteLine("Greetings!");
    }
}
```

Code Listing 1

The **class** is a container for code, defining a type, named **Greetings**. A **class** has members and this example shows a method member named **Main**. A method is similar to functions and

procedures in other programming languages. For desktop application types, like console or WPF, naming a method **Main** tells the C# compiler where the program begins executing. Both the **Greetings** class and **Main** method have curly braces, referred to as a block, indicating beginning and ending scope.

The **void** keyword isn't a type; it indicates that a method does not return a value. For **Main**, you can replace **void** with **int**, meaning that the program has a return code. This number can be used by command-line shell tools to evaluate the conditions under which the program ended. It is unique to each program and specified by you. Later, you'll learn more about methods and return values.

The **static** modifier indicates that there is only ever one instance of a **Greetings** class that has that **Main** method—it is the **static** instance. **Main** must be **static**, but other methods can omit **static**, which makes them instance members. This means that you can have many copies of a class or instance with their own method.

Since a program only needs a single **Main** method, **static** makes sense. A program that manages customers might have a **Customer** class and you would need multiple instances to represent each **Customer**. You'll see examples of instantiating classes in later chapters of this book.

Inside the **Main** method is a statement that prints words to the command line. The words, enclosed in double quotes, are a string. That string is passed to the **WriteLine** method, which writes that string to the command line and causes it to move to the next line. **WriteLine** is a method that belongs to a class named **Console**. You see in this example, just like the **Greetings** class, the **Console** is a class too. This **Console** class belongs to the **System** namespace, which is why the **using** clause appears at the top of the file, allowing us to use that **Console** class.

The code begins with a **using** clause for the **System** namespace. The FCL is grouped into namespaces to keep code organized and avoid clashes between identically named types. This **using** clause allows us to use the code in the **System** namespace, which we're doing with the **Console** class. Without that, the compiler doesn't know what **Console** means or how to find it, but now C# knows that we're using the **System.Console** class.

Namespaces and Code Organization

There are various ways to organize code and the choice should be based on the standards of your team and the nature of the project you're building. One of the common ways to organize code is with the C# namespace feature. Here's a hierarchical description of where namespaces fit into the overall structure of a program:

Namespace

  Type

    Type Members

Out of this hierarchy, the namespace is optional, as demonstrated in the previous program where the **Greetings** class was not contained in a namespace. This means **Greetings** is a

member of the **global** namespace. You should avoid this practice as it opens the possibility for other developers working with your code to write their own **Greetings** class in the same namespace, which will cause errors because the C# compiler can't figure out which **Greetings** class to use. While **Greetings** might seem unique and unlikely, think of common names, such as **File**, **Math**, or **Window**, that would cause problems. The following program uses namespaces appropriately.

```
using static System.Math;

namespace Syncfusion
{
    public class Calc
    {
        public static double Pythagorean(double a, double b)
        {
            double cSquared = Pow(a, 2) + Pow(b, 2);
            return Sqrt(cSquared);
        }
    }
}
```

Code Listing 2

The **Calc** class is a member of the **Syncfusion** namespace. The **Pythagorean** method is a member of the **Calc** class. A method is a block of code with a name, parameters, and return value that you can call from other code. This follows the namespace, class, member organization.

System is a namespace in the FCL and **Math** is a class in the **System** namespace. The **using static** clause allows the code to use static members of the **Math** class without full qualification. Instead of writing **Math.Pow(a, 2)**, which squares the value of **a**, you can use the shorthand syntax in the **Pythagorean** method. The **Pythagorean** method uses **Math.Sqrt**, which provides square root, similarly. The following sample shows how you can use this code.

```
using Syncfusion;
using System;

using Crypto = System.Security.Cryptography;

namespace NamespaceDemo
{
    class Program
    {
        static void Main()
        {
            double hypotenuse = Calc.Pythagorean(2, 3);
            Console.WriteLine("Hypotenuse: " + hypotenuse);

            Crypto.AesManaged aes = new Crypto.AesManaged();

            Console.ReadKey();
```

```
        }
    }
}
```

Code Listing 3

The **Main** method calls the **Pythagorean** method of the **Calc** class, passing arguments **2** and **3** and receiving a result in **hypotenuse**. Since **Calc** is in the **Syncfusion** namespace, the code adds a **using** clause for **Syncfusion** to the top of the file. Had the code not included that **using** clause, **Main** would have been required to use the fully qualified name, **Syncfusion.Calc.Pythagorean**.

Another feature of the previous program is the namespace alias, **Crypto**. This syntax allows you to use a shorthand syntax when you need to fully qualify a namespace, but want to reduce syntax in your code. If there had been another **Cryptography** namespace used in the same code, though not in this listing, full qualification would have been necessary. **Crypto** is the alias for **System.Security.Cryptography** and **Main** uses that alias in **Crypto.AesManaged** to make the code more readable.

Running the Program

The rest of this chapter returns to the previous Greetings program in this chapter.

Now the program is written and you want to continue by compiling the program and running it. You'll want to save this file with the name **Greetings.cs**. The name isn't necessarily important, but by convention should be meaningful and is often the same name as a class it contains. You're allowed to put multiple classes in the same file, but it's easier to find a class later if it is alone in its own file of the same name. C# files have a .cs extension.

In VS, click the green **Start** arrow on the toolbar and it will build and run the program. The program runs and stops so quickly that you won't see the command-line output, so you can press **Ctrl + F5** to make the command line stay open. This book uses Visual Studio 2015, but Syncfusion has published *Visual Studio 2013 Succinctly*, which explains many features that are still valid in Visual Studio 2015. In the meantime, I'm going to show you how to use the C# compiler directly—the benefit being that you see what the IDE is doing for you.

Tip: Adding `Console.ReadKey();` as the last line in `Main` makes the command line stop and wait for a key press.

Minimally, you need the .NET Framework installed on your machine, which is free for commercial as well as non-commercial use. If you installed VS, you already have the .NET Framework. Otherwise, download it from http://www.microsoft.com/en-us/download/details.aspx?id=30653 and install it. This link is for .NET Framework 4.5, but any future version should work fine.

Once .NET is installed, open Windows Explorer and do a search for the C# compiler, **csc.exe**. Since I'm using Visual Studio 2015 for the examples in this book, the C# 6 compiler on my machine is located at **C:\Program Files (x86)\MSBuild\14.0\Bin**, but yours may differ.

Next, make sure the C# compiler is in your path. Open your **System Properties** window. As of this writing, I'm on Windows 8.1 and found it via selecting **Control Panel** > **System and Security** > **System**, and then clicking **Advanced System Settings**. Select the **Advanced** tab and click the **Environment Variables** button. In the **System variables** list, select **Path**, and click **Edit**. You should see several paths separated by semicolons. At the end of that path, add your C# compiler's path that you found with the Windows Explorer search and make sure it's separated from the previous paths with a semicolon. Close out of all these windows when you're done setting the path.

Now that you have the .NET Framework installed and have the path to the C# compiler set, you can build the program that you typed in the previous example. First, open a command prompt window. On my system, I can do this by pressing the **Windows logo key + R**, typing **cmd.exe** in the **Run** dialog, and then clicking **OK**. If you've never used a command line, it's a good idea to open your favorite search engine and look for a tutorial. Alternatively, it might be good to learn PowerShell; Syncfusion has a book on it titled *PowerShell Succinctly*. In the meantime, navigate to the directory where you saved **Greetings.cs**. You can type **cd\your\path\there** and press **Enter** to get there. You can verify you're in the right location by typing **dir** to see what files reside in the current directory.

To compile the program, type **csc Greetings.cs**. If you see compiler errors, go back to Code Listing 1 and make sure you've typed the code exactly as it is there and then recompile.

Tip: Use a space separated list to compile multiple files; e.g., *csc.exe file1.cs file2.cs.* ***For C# compiler help, type*** *csc.exe /help.*

Now type **dir** and you'll see a new file named **Greetings.exe**. This is an executable assembly. In .NET, an assembly is a unit of identity, execution, and deployment, which is why it's not just called a file. For the purposes of this book, you won't be involved with all the nuances of assemblies, but it's an encompassing term that includes both executable (.exe) and library (.dll) files.

Now type **Greetings.exe** and press **Enter**. The program will print `Greetings!` on the command line. Then you'll see a new command-line prompt, meaning that the program ended. This came from the `Console.WriteLine` statement in the `Main` method. When the `Main` method finishes executing, the program finishes too.

Deploying the Program

.NET uses XCopy deployment, which means that you only need to copy the assembly to anywhere you want it to go. The one caveat is that whatever machine you run the program on must also have the .NET CLR installed. Installing VS or the .NET Framework automatically installs the CLR. Also, you can only install the .NET Framework Runtime, which doesn't include development tools, to a machine where you only want to run a C# program but not perform any

development tasks. In practical terms, most Windows systems already have .NET installed from the original installation and it is kept up-to-date via Windows Update.

Whenever you run the program, Windows looks at the executable, determines that it's a .NET assembly, loads the CLR, and then gives that assembly to the CLR to run. From the users' perspective, the CLR behavior is behind the scenes; the program appears like any other program when they run the executable.

Summary

This chapter included a couple broader takeaways regarding how C# fits into the .NET Framework ecosystem and how to create a C# program. Remember that C# is a programming language, but it builds programs that use the FCL to run applications managed by the CLR. What this gives you is the ability to compile programs into assemblies that can be deployed and run on any machine that supports the CLR. The program entry point is the **`Main`** method. You can use any editor or an IDE like Visual Studio to write your code. To run a program, press **F5** in VS or compile with **`csc.exe`** on the command line. To deploy, copy the program to a machine with the CLR installed. In the next chapter, you'll learn more about how to code logic in C# using expressions and statements.

Chapter 2 Coding Expressions and Statements

In Chapter 1, you saw how to write, compile, and execute a C# program. The example program had a single statement in the **Main** method. In this chapter, you'll learn how to write more statements and add logic to your program. For efficiency, many of the examples in the rest of the book are snippets, but you can still add these statements inside of a **Main** method to compile and get a better feel for C# syntax. There will be plenty of complete programs too.

Writing Simple Statements

By combining language operators and syntax, you can build expressions and statements in C#. Here are a few examples of simple C# statements.

```
int count = 7;
char keyPressed = 'Q';
string title = "Weekly Report";
```

Code Listing 4

Each of the examples in the previous code listing have common syntactical elements: type, variable identifier, assignment operator, value, and statement completion. The types are **int**, **char**, and **string**, which represent a number, a character, and a sequence of characters respectively. These are a few of the several built-in types that C# offers. Variables are a name that can be used in later code. The = operator assigns the right-hand side of the expression to the left-hand side. Each statement ends with a semicolon.

The previous example showed how to declare a variable and perform assignment at the same time, but that isn't necessarily required. As long as you declare a variable before trying to use it, you'll be okay. Here's a separate declaration.

```
string title;
```

Code Listing 5

And the variable's later assignment.

```
title = "Weekly Report";
```

Code Listing 6

Note: C# is case sensitive, so "title" and "Title" are two separate variable names.

Overview of C# Types and Operators

C# is a strongly typed language, meaning that the compiler won't implicitly convert between incompatible types. For example, you can't assign a **`string`** to an **`int`** or an **`int`** to a **`string`**—at least, not implicitly. The following code will not compile.

```
int total = "359";
string message = 7;
```

Code Listing 7

The "**`359`**" with double quotes is a string, and the **`7`** without quotes is an **`int`**. While you can't perform conversions implicitly, there are ways to do this explicitly. For example, you'll often receive text input from a user that should be an **`int`** or another type. The following code listing shows a couple examples of how to perform such tasks explicitly.

```
int total = int.Parse("359");
string message = 7.ToString();
```

Code Listing 8

In the previous listing, **`Parse`** will convert the string to an **`int`** if the string represents a valid **`int`**. Calling **`ToString`** on any value will always produce a **`string`** that will compile.

In addition to the previous conversion examples, C# has a cast operator that lets you convert between types that allow explicit conversions. Let's say you have a **`double`**, which is a 64-bit floating point type, and want to assign that to an **`int`**, which is a 32-bit whole number. You could cast it like this:

```
double preciseLength = 5.61;
int roundedLength = (int)preciseLength;
```

Code Listing 9

Without the cast operator, you would receive a compiler error because a **`double`** is not an **`int`**. Essentially, the C# compiler is protecting you from shooting yourself in the foot because assigning a **`double`** to an **`int`** means that you lose precision. In the previous example, **`roundedLength`** becomes **`5`**. Using the cast operator allows you to tell the C# compiler that you know this operation could be dangerous in the wrong circumstances, but makes sense for your particular situation.

The following table lists the built-in types so you can see what is available:

Table 1: Built-In Types

Type (Literal Suffix)	Description	Values/Range
byte	8-bit unsigned integer	0 to 255
sbyte	8-bit signed integer	-128 to 127
short	16-bit signed integer	-32,768 to 32,767
ushort	16-bit unsigned integer	0 to 65,535
int	32-bit signed integer	-2,147,483,648 to 2,147,483,647
uint	32-bit unsigned integer	0 to 4,294,967,295
long (l)	64-bit signed integer	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong (ul)	64-bit unsigned integer	0 to 18,446,744,073,709,551,615
float (f)	32-bit floating point	-3.4×10^{38} to $+3.4 \times 10^{38}$
double (d)	64-bit floating point	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
decimal (m)	128-bit, 28 or 29 digits of precision (ideal for financial)	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / (10^{0 \text{ to } 28})$
bool	Boolean	true or false
char	16-bit Unicode character (use single quotes)	U+0000 to U+FFFF
string	Sequence of Unicode characters (use double quotes)	E.g., “abc”

You should add a suffix to a number when the meaning would be ambiguous. In the following example, the **`m`** suffix ensures the **`9.95`** literal is treated as a **`decimal`** number:

```
decimal price = 9.95m;
```

Code Listing 10

You can assign Unicode values directly to a char. The following example shows how to assign a carriage return.

```
char cr = '\u0013';
```

Code Listing 11

You can also obtain the Unicode value of a character with a cast operator as shown here.

```
int crUnicode = (int)cr;
```

Code Listing 12

So far, you've only seen statements with the assignment operator, but C# has many other operators that allow you to perform all of the logical operations you would expect of any general purpose programming language. The following table lists some of the available operators.

Table 2: C# Operators

Category	Description
Primary	`x.y x?.y f(x) a[x] x++ x-- new typeof default checked unchecked nameof`
Unary	`+ - ! ~ ++x --x (T)x await x`
Multiplicative	`* / %`
Additive	`+ -`
Shift	`<< >>`
Relational and Type Testing	`< > <= >= is as`
Equality	`== !=`
Logical AND	`&`
Logical XOR	`^`
Logical OR	`|`
Conditional AND	`&&`
Conditional OR	`||`
Null Coalescing	`??`
Conditional	`?:`
Assignment	`= *= /= %= += -= <<= >>= &= ^= |= =>`

Prefix operators change the value of the variable before assignment, and postfix operators change a variable after assignment, as demonstrated in the following sample.

```
int val1 = 5;
int val2 = ++val1;
int val3 = 2;
int val4 = val3--;
```

Code Listing 13

In the previous code listing, both **val1** and **val2** are **6**. The **val3** variable is **1**, but **val4** is **2** because the postfix operator evaluates after assignment.

The ternary operator offers simple syntax for if-then-else logic. Here's an example:

```
decimal priceGain = 2.5m;
string action = priceGain > 2m ? "Buy" : "Sell";
```

Code Listing 14

On the left side of **?** is a Boolean expression, **priceGain > 2m**. If that is true, which it is in this example, the ternary operator returns the first value between **?** and **:**, which is **"Buy"**. Otherwise, the ternary operator would return the value after the **:**, which is **"Sell"**. This statement assigns the result of the ternary operator, **"Buy"**, to the string variable, **action**.

In addition to the built-in types, the FCL has many types you will use on a daily basis. One of these is **DateTime**, which represents a date and time. Here's a quick demo showing a couple things you can do with a **DateTime**.

```
DateTime currentTime = DateTime.Now;
string shortDateString = currentTime.ToShortDateString();
string longDateString = currentTime.ToLongDateString();
string defaultDateString = currentTime.ToString();
DateTime tomorrow = currentTime.AddDays(1);
```

Code Listing 15

The previous code shows how to get the current **DateTime**, a short representation of a date (e.g., 12/8/2014), a long representation of the date and time (everything spelled out), the default numeric representation, and how to use **DateTime** methods for calculations.

Tip: Search the FCL before creating your own library of types. Many of the common types you use every day, like `DateTime`, will already exist.

Operator Precedence and Associativity

The C# operators listed in Table 2 outlines operators in their general order of precedence. The precedence defines which operators evaluate first. Operators of higher precedence evaluate before operators of lower precedence.

Assignment and conditional operators are right-associative and all other operators are left-associative. You can change the normal order of operations by using parentheses as shown in the following code listing.

```
int result1 = 2 + 3 * 5;
int result2 = (2 + 3) * 5;
```

Code Listing 16

In the previous code, **result1** is **17**, but **result2** is **25**.

Formatting Strings

There are different ways to build and format strings in C#: concatenation, numeric format strings, or string interpolation. The following code listing demonstrates string concatenation.

```
string name = "Joe";
string helloViaConcatenation = "Hello, " + name + "!";
Console.WriteLine(helloViaConcatenation);
```

Code Listing 17

This prints **"Hello, Joe!"** to the console. The following example does the same thing, but uses **string.Format**.

```
string helloViaStringFormat = string.Format("Hello, {0}!", name);
Console.WriteLine(helloViaStringFormat);
```

Code Listing 18

The **string.Format** takes a format string that has numeric placeholders in curly braces. It's 0-based, so the first placeholder is **{0}**. The parameters following the string are placed into the format string in the order they appear. Since **name** is the first (and only) parameter, **string.Format** replaces **{0}** with **Joe** to create **"Hello, Joe!"** as a string. As a convenience in console applications, **WriteLine** uses the same formatting. The following code accomplishes the same task as the two lines in the previous code listing.

```
Console.WriteLine("Hello, {0}!", name);
```

Code Listing 19

Going a little further, string formatting is more powerful, allowing you to specify column lengths, alignment, and value formatting as shown in the following code.

```
string item = "bread";
decimal amount = 2.25m;
Console.WriteLine("{0,-10}{1:C}", item, amount);
```

Code Listing 20

In this example, the first placeholder consumes 10 characters in length. The default alignment is right, but the minus sign changes that to align on the left. On the second placeholder, the **C** is a currency format string.

Note: There are many string formatting options. You can visit https://msdn.microsoft.com/en-us/library/dwhawy9k(v=vs.110).aspx for standard formats, https://msdn.microsoft.com/en-us/library/0c899ak8(v=vs.110).aspx for custom formats, and https://msdn.microsoft.com/en-us/library/az4se3k1(v=vs.110).aspx for DateTime formats.

C# 6 introduced a new way to format strings, called string interpolation. It's a shorthand syntax that lets you replace numeric placeholders with expressions as follows:

```
Console.WriteLine($"{item}      {amount}");
```

Code Listing 21

The **$** prefix is required. Here, the value from the **item** variable replaces **{item}** and the value from the **amount** variable replaces **{amount}**. Similar to numeric placeholders, you can include additional formatting.

```
Console.WriteLine($"{nameof(item)}: {item,-10} {nameof(amount)}: {amount:C}");
```

Code Listing 22

The **nameof** operator prints out the name ***"item"***, demonstrating how you can use expressions in placeholders. You can also see the space and currency formatting on **item** and **amount**.

Branching Statements

You can use either an **if-else** or **switch** statement in your code for branching logic. When you only need to execute code for a true condition, use an **if** statement as in the following sample.

```
string action2 = "Sell";
```

```
if (priceGain > 2m)
{
    action2 = "Buy";
}
```

Code Listing 23

The curly braces are optional in this example because there is only one statement to execute if **`priceGain > 2m`**. However, they would be required for multiple statements. This is true for all branching and logic statements. You can also have an **`else`** case, as shown in the following listing.

```
string action3 = "Do Nothing";
if (priceGain <= 2m)
{
    action3 = "Sell";
}
else
{
    action3 = "Buy";
}
```

Code Listing 24

Whenever the Boolean condition of the **`if`** statement is false, as it is in the previous code sample where **`priceGain <= 2m`**, the **`else`** clause executes. In this case, **`action3`** becomes **`"Buy"`**. Of course, you can have multiple conditions by adding more **`else if`** clauses.

```
string action4 = null;
if (priceGain <= 2m)
{
    action4 = "Sell";
}
else if (priceGain > 2m && priceGain <= 3m)
{
    action4 = "Do Nothing";
}
else
{
    action4 = "Sell";
}
```

Code Listing 25

In the previous example, you can see a more complex Boolean expression in the **`else if`** clause. When **`priceGain`** is **`2.5`**, the value of **`action4`** becomes **`"Do Nothing"`**. The **`&&`** is a logical operator that succeeds if both the expression on the left and right are true. The logical **`||`** operator succeeds if either the expression on the left or right is true. These operators also perform short-circuit operations where the expression on the right doesn't execute if the expression on the left causes the whole expression to not be true. In the case of the **`else if`** in

Code Listing 25, if **priceGain** were **2m** or less, the **&&** operator would not evaluate the **priceGain <= 3** expression because the entire operation is already false. Once a branch of the **if** statement executes, no other branches are evaluated or executed.

Notice that I set **action4** to **null**. The **null** keyword means no value. I'll talk about **null** in the next chapter and explain where you can use it.

An **if** statement is good for either simple branching or complex conditions, such as the previous **else if** clause. However, when you have multiple cases and all expressions are constant values, such as an **int** or **string**, you might prefer a **switch** statement. The following example uses a **switch** statement to select appropriate equipment based on a weather forecast.

```
string currentWeather = "rain";
string equipment = null;
switch (currentWeather)
{
    case "sunny":
        equipment = "sunglasses";
        break;
    case "rain":
        equipment = "umbrella";
        break;
    case "cold":
    default:
        equipment = "jacket";
        break;
}
```

Code Listing 26

The **switch** statement tries to match a value, **currentWeather** in this example, with one of its **case** statements. It uses the **default** case for no match. All **case** statements must be terminated with a **break** statement. The only time fall-through is allowed is when a **case** has no body, as demonstrated with the **"cold" case** and **default**, which both set **equipment** to **"jacket"**. Since **currentWeather** is **"rain"**, **equipment** becomes **"umbrella"** and no other cases execute.

Beyond branching statements, you also need the ability to perform a set of operations multiple times, which is where C# loops come in. Before discussing loops, let's look at arrays and collections, which hold data that loops can use.

Arrays and Collections

Sometimes you need to group a number of items together in a collection to manage them in memory. For this, you can either use arrays or one of the many collection types in the .NET Framework. The following sample demonstrates how to create an array.

```
int[] oddNumbers = { 1, 3, 5 };
```

```
int firstOdd = oddNumbers[0];
int lastOdd = oddNumbers[2];
```

Code Listing 27

Here, I've declared and initialized the array with three values. Arrays and collections are 0-based, so **`firstOdd`** is **`1`** and **`lastOdd`** is **`5`**. The **`[x]`** syntax, where **`x`** is a number, is referred to as an indexer because it allows you to access the array at the location specified by the index. Here's another example that uses **`string`** instead of **`int`**.

```
string[] names = new string[3];
names[1] = "Joe";
```

Code Listing 28

In this example, I instantiated an array to hold three strings. All of the strings equal **`null`** by default. This code sets the second string to **`"Joe"`**.

In addition to arrays, you can use all types of data structures, such as **`List`**, **`Stack`**, **`Queue`**, and more, which are part of the FCL. The following example shows how to use a **`List`**. Remember to add a **`using`** clause for **`System.Collections.Generic`** to use the **`List<T>`** type.

```
List<decimal> stockPrices = new List<decimal>();
stockPrices.Add(56.23m);
stockPrices.Add(72.80m);
decimal secondStockPrice = stockPrices[1];
```

Code Listing 29

In this sample, I instantiated a new **`List`** collection. The **`<decimal>`** is a generic type indicating that this is a strongly typed list that can only hold values of type **`decimal`**; it's a **`List`** of **`decimal`**. That list has two items. Notice how I used the array-like indexer syntax to read the second item in the (0-based) **`stockPrices`** list.

Looping Statements

C# supports several loops, including **`for`**, **`foreach`**, **`while`**, and **`do`**. The code listings that follow perform similar logic.

```
double[] temperatures = { 72.3, 73.8, 75.1, 74.9 };
for (int i = 0; i < temperatures.Length; i++)
{
    Console.WriteLine(i);
}
```

Code Listing 30

The **for** loop initializes **i** to **0**, makes sure **i** is less than the number of items in the **temperature** array, executes the **Console.WriteLine**, and then increments **i**. It continues executing until the condition (**i < temperatures.Length**) is false, and then moves on to the next statement in the program.

```
foreach (int temperature in temperatures)
{
    Console.WriteLine(temperature);
}
```

Code Listing 31

The **foreach** loop used in Code Listing 31 is simpler and will execute for each value in the **temperatures** array.

Next is an example of a **while** loop.

```
int tempCount = 0;
while (tempCount < temperatures.Length)
{
    Console.WriteLine(tempCount);
    tempCount++;
}
```

Code Listing 32

The **while** loop evaluates the condition and executes if it's true. Notice that I initialized **tempCount** to **0** and increment **tempCount** inside of the loop on each iteration.

Finally, the following example shows how to write a **do-while** loop.

```
int tempCount2 = 0;
do
{
    Console.WriteLine(tempCount2++);
}
while (tempCount2 <= temperatures.Length);
```

Code Listing 33

A **do-while** loop is good for when you want to execute logic at least one time. This example increments **tempCount2** as a parameter to **Console.WriteLine**. Remember, the postfix operator changes the variable after evaluation.

Wrapping Up

Here's a calculator program that pulls together some of the concepts from this chapter, plus some extra features. You can type this into your editor and execute it for practice.

```
using System;
using System.Text;

/*
    Title: Calculator
    By: Joe Mayo
*/

class Calculator
{
    /// <summary>
    /// This is the entry point.
    /// </summary>
    static void Main()
    {
        char firstChar = 'Q';
        bool keepRunning = true;

        do
        {
            Console.WriteLine();
            Console.Write("What do you want to do? (Add, Subtract, Multiply, Divide,
Quit): ");
            string input = Console.ReadLine();
            firstChar = input[0];

            // This is used in both the if statement and the do-while loop.
            keepRunning = !(firstChar == 'q' || firstChar == 'Q');

            double firstNumber = 0;
            double secondNumber = 0;

            if (keepRunning)
            {
                Console.Write("First Number: ");
                string firstNumberInput = Console.ReadLine();
                firstNumber = double.Parse(firstNumberInput);

                Console.Write("Second Number: ");
                string secondNumberInput = Console.ReadLine();
                secondNumber = double.Parse(secondNumberInput);
            }

            double result = 0;
            switch (firstChar)
            {
                case 'a':
                case 'A':
                    result = firstNumber + secondNumber;
                    break;
```

```
                case 's':
                case 'S':
                    result = firstNumber - +secondNumber;
                    break;
                case 'm':
                case 'M':
                    result = firstNumber * secondNumber;
                    break;
                case 'd':
                case 'D':
                    result = firstNumber / secondNumber;
                    break;
                default:
                    result = -1;
                    break;
            }

            Console.WriteLine();
            Console.WriteLine("Your result is " + result);

        } while (keepRunning);
    }
}
```

Code Listing 34

The previous program demonstrates a **do-while** loop, an **if** statement, a **switch** statement, and a basic console communication with the user.

There are a couple string features here that you haven't seen yet. The first is where the program uses **Console.ReadLine** to read input text from the user for the input string. Notice the indexer syntax to read the first character from the string. You can read any character of a string this way. Also, look at the bottom of the program where it prints **"Your result is " + result**, which concatenates a string with the number. Using the **+** operator for concatenation is a simple way to build strings. Another way to build a string is with a type named **StringBuilder**, which you can use like this:

```
StringBuilder sb = new StringBuilder();
sb.Append("Your result is ");
sb.Append(result.ToString());
Console.WriteLine(sb.ToString());
```

Code Listing 35

You'll also need to add a **using System.Text;** clause to the top of the file. After you've used the concatenate operator, **+**, about four times on the same string, you might consider rewriting with a **StringBuilder** instead. The **string** type is immutable, meaning that you can't modify it. This also means that every concatenation operation causes the CLR to create a new string in-memory.

The calculator program also has multiline and single-line comments that aren't compiled, but help you document the code as you need. Here's the multi-line comment:

```
/*
    Title: Calculator
    By: Joe Mayo
*/
```

Code Listing 36

Here's the single-line comment:

```
// This is used in both the if statement and the do-while loop.
```

Code Listing 37

An extension of the single-line comment is a convention that uses three slashes and a set of XML tags, known as documentation comments.

```
/// <summary>
/// This is the entry point.
/// </summary>
```

Code Listing 38

Summary

C# has a full set of operators and types that allow you to write a wide range of expressions and statements. With branching statements and loops, you can write logic of your choosing. All of the code in this chapter has been in the **`Main`** method, but clearly that's inadequate and you'll quickly grow out of that. The next chapter explores some new C# features to help organize code with methods and properties.

Chapter 3 Methods and Properties

Previous chapters show how to write code in the **Main** method. That's the program entry point, but it's normally a lightweight method without too much code. For this chapter, you'll learn how to move your code out of the **Main** method and modularize it so you can manage the code better. You'll learn how to define methods with parameters and return values. You'll also learn about properties, which let you encapsulate object state.

Starting at Main

We'll use a simpler version of the calculator from the previous chapter to get started. This calculator only performs addition and stops running after one operation.

```
using System;

class Calculator1
{
    static void Main()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);

        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);

        double result = firstNumber + secondNumber;

        Console.WriteLine($"\n\tYour result is {result}.");

        Console.ReadKey();
    }
}
```

Code Listing 39

The part of this program that might be new is the **Console.ReadKey** statement at the end of the **Main** method. This allows users to see the result and keeps the program from ending until they press a key. The **\n** in the interpolated string is a newline and **\t** is a tab.

Modularizing with Methods

Although the previous program is small, a first glance doesn't really tell you what it does. Imagine if it was like the calculator in Chapter 2 or even longer; it would eventually become difficult to understand. When you have to work on this again later, you might need to read many

lines of code to understand it. So, it would be better to refactor this. Refactoring is the practice of changing the design of code without changing its functionality; the purpose is to improve the program. The following code sample is a first draft of refactoring this program into methods.

```
using System;

class Calculator2
{
    static void Main()
    {
        double firstNumber = GetFirstNumber();

        double secondNumber = GetSecondNumber();

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetFirstNumber()
    {
        Console.Write("First Number: ");
        string firstNumberInput = Console.ReadLine();
        double firstNumber = double.Parse(firstNumberInput);
        return firstNumber;
    }

    static double GetSecondNumber()
    {
        Console.Write("Second Number: ");
        string secondNumberInput = Console.ReadLine();
        double secondNumber = double.Parse(secondNumberInput);
        return secondNumber;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

Code Listing 40

Looking at **`Main`**, you can tell what the program does. It reads two numbers, adds the results, and then shows the results to the user. Each of those lines is a method call. The first three methods—**`GetFirstNumber`**, **`GetSecondNumber`**, and **`AddNumbers`**—return a value that is assigned to a variable. The last method, **`PrintResult`**, performs an action without returning a

result. Before moving to the next refactoring, let's walk through these methods. The following code listing shows the **GetFirstNumber** method.

```
static double GetFirstNumber()
{
    Console.Write("First Number: ");
    string firstNumberInput = Console.ReadLine();
    double firstNumber = double.Parse(firstNumberInput);
    return firstNumber;
}
```

Code Listing 41

At first glance, the signature of this method looks similar to the **Main** method. The differences are that the return type of this method is **double** and the method is named **GetFirstNumber**. All we did was write the method and the code that creates the **firstNumber**. When a method has a return type, a value of that type must be returned. **GetFirstNumber** does that with the **return** statement.

The **GetSecondNumber** method is nearly identical to **GetFirstNumber**. Let's examine **AddNumbers** next.

```
static double AddNumbers(double firstNumber, double secondNumber)
{
    return firstNumber + secondNumber;
}
```

Code Listing 42

Notice that **Main** passes the **firstNumber** and **secondNumber** variables to **AddNumbers** as arguments that the **AddNumbers** method can work with as parameters. The return type of **AddNumbers** is **double**, so the method adds and returns the result of the add operation.

Finally, we have the **PrintResult** method.

```
static void PrintResult(double result)
{
    Console.WriteLine($"\nYour result is {result}.");
}
```

Code Listing 43

The **PrintResult** method writes the results from its parameter to the console. Notice that **PrintResult** does not have a return type, as indicated by the **void** keyword.

Simplifying Code with Methods

The last section improved the program because a huge block of code was broken into more meaningful pieces. We can improve this code with some extra refactoring. In particular, the **`GetFirstNumber`** and **`GetSecondNumber`** methods are largely redundant. The following sample shows how to refactor those two methods into one and reduce the amount of code.

```
using System;

class Calculator3
{
    static void Main()
    {
        double firstNumber = GetNumber("First");
        double secondNumber = GetNumber("Second");

        double result = AddNumbers(firstNumber, secondNumber);

        PrintResult(result);

        Console.ReadKey();
    }

    static double GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);
        return number;
    }

    static double AddNumbers(double firstNumber, double secondNumber)
    {
        return firstNumber + secondNumber;
    }

    static void PrintResult(double result)
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

Code Listing 44

This time I removed **`GetFirstNumber`** and **`GetSecondNumber`** and replaced them with **`GetNumber`**. The only real difference besides variable names is the **`whichNumber`** **`string`** parameter.

Adding Properties

The previous examples performed all of the operations inside of the same class. It was driven from the **`Main`** method and serviced through methods. What if I wanted to reuse the calculator

functions in that class and wanted the new class to hold its own values, or state? In this case, moving the calculator methods into a separate **`Calculator`** class would be useful.

The next question to ask is, "How do we get to the state of the class?" For example, if I want to read the result from the **`Calculator`** class, what is the best way to do so? One approach is to use a method named **`GetResult`** that returns the value. Another way in C# is to use a property, which you can use like a field, but works like a method. The following version of the calculator program shows how to refactor methods into a separate class and add properties.

Note: Refactoring is the practice of changing the design of code without changing its behavior with the goal of improving the code. Martin Fowler's book,* Refactoring: Improving the Design of Existing Code*, is a good reference.

```
using System;

public class Calculator4
{

    double[] numbers = new double[2];

    public double First
    {
        get
        {
            return numbers[0];
        }
    }

    public double Second
    {
        get
        {
            return numbers[1];
        }
    }

    double result;

    public double Result
    {
        get { return result; }
        set { result = value; }
    }

    public void GetNumber(string whichNumber)
    {
        Console.Write($"{whichNumber} Number: ");
        string numberInput = Console.ReadLine();
        double number = double.Parse(numberInput);

        if (whichNumber == "First")
            numbers[0] = number;
        else
```

```
            numbers[1] = number;
    }

    public void AddNumbers()
    {
        Result = First + Second;
    }

    public void PrintResult()
    {
        Console.WriteLine($"\nYour result is {result}.");
    }
}
```

Code Listing 45

In the previous code listing, **First**, **Second**, and **Result** are properties. I'll break down the syntax shortly, but first look at how these properties are used inside of the **AddNumbers** and **PrintResults** methods. **AddNumbers** reads the values of **First** and **Second** and adds those values together and writes to **Result**.

Each of these properties looks just like a field or variable; you just read from and write to them. **PrintResult** reads the **Result** property. However, looking at the definitions of the properties, you can tell right away that they aren't fields.

The **Result** property is a typical read and write property with **get** and **set** accessors. When you read the property, the **get** accessor executes. When you write to the property, the **set** accessor executes. Notice that there is a **result** field (lowercase) prior to the **Result** (uppercase) property. The **get** accessor reads the value of **result** and the **set** accessor writes to **result**. When using **set**, the **value** keyword represents what is being written to the property.

This pattern of reading and writing from a single backing store is so common that C# has a shortcut syntax you can use instead. The following code sample shows **Result** rewritten as an auto-implemented property.

```
    public double Result { get; set; }
```

Code Listing 46

The backing store in auto-implemented properties is implied and handled behind the scenes by the C# compiler. If you need to provide validation on the value being assigned or have a unique way of storing the value, you should resort to full properties where the **get** accessor, **set** accessor, or both are defined.

In fact, **First** and **Second** properties have a unique backing store, requiring a fully implemented **get** accessor. They read from an array position. Notice that the **GetNumber** method figures out which array position to put each number into.

Properties give you the ability to encapsulate the internal operations of your class so you are free to modify the implementation without breaking the interface for consumers of your class.

The following code sample demonstrates how consuming code might use this new **Calculator4** class.

```
using System;

class Program
{
    static void Main()
    {
        var calc4 = new Calculator4();

        calc4.GetNumber("First");
        calc4.GetNumber("Second");

        calc4.AddNumbers();

        PrintResult(calc4);

        Console.ReadKey();
    }

    static void PrintResult(Calculator4 calc)
    {
        Console.WriteLine($"Your result is {result}.");
    }
}
```

Code Listing 47

The **Main** method creates a new instance of **Calculator4** and calls public methods. All of the strange internals of **Calculator4** are hidden and **Main** only cares about the public interface, exposing **Calculator4** services for reuse. The **PrintResult** method reads the **Calculator4** instance **Result** property. Again, that's the benefit of methods and properties: callers can use a class without caring how that class does what it does.

Exception Handling

C# has a feature called structured exception handling that lets you work with situations where your methods aren't able to fulfill their intended purpose. The syntax for managing exception handling is the **try-catch** block. All the code to be monitored for exceptions goes in the **try** block, and the code that handles a potential exception goes in a **catch** block. The following code listing shows an example.

```
        static void HandleNullReference()
        {
            Program prog = null;

            try
            {
                Console.WriteLine(prog.ToString());
```

```
        }
        catch (NullReferenceException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
```

Code Listing 48

In C#, any time you try to use a member of a **null** object, you'll receive a **NullReferenceException**. The solution to fix the problem is to assign a value to the variable. The previous example causes a **NullReferenceException** to be thrown in the **try** block by calling **ToString** on the **prog** variable, which is **null**.

Since the code that threw the exception is inside the **try** block, the code stops execution of any of the code in the **try** block and starts looking for an exception handler. The **catch** block **parameter** indicates that it can catch a **NullReferenceException** if the code inside of the **try** block throws that exception type. The body of the **catch** block is where you perform any exception handling.

You can customize exception handling with multiple **catch** blocks. The following example shows code that throws an exception in the **try** block, which is subsequently handled by a **catch** block.

```
    static void HandleUncaughtException()
    {
        Program prog = null;

        try
        {
            Console.WriteLine(prog.ToString());
        }
        catch (ArgumentNullException ex)
        {
            Console.WriteLine("From ArgumentNullException: " + ex.Message);
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("From ArgumentException: " + ex.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine("From Exception: " + ex.Message);
        }
        finally
        {
            Console.WriteLine("Finally always executes.");
        }
    }
```

Code Listing 49

The method name is **HandleUncaughtException** because there isn't a specific **catch** block to handle a **NullReferenceException**; the exception will be handled by the **catch** block for the **Exception** type.

You list exceptions by their inheritance hierarchy, with top-level exceptions lower in the list of **catch** blocks. A thrown exception will move down this list of handlers, looking for a matching exception type, and only execute in the **catch** block of the first handler that matches. **ArgumentNullException** derives from **ArgumentException**, and **ArgumentException** derives from **Exception**.

If no **catch** block can handle an exception, the code goes up the stack looking for a potential **catch** block in calling code that can handle the exception type. If no code in the call stack is able to handle the exception, your program will terminate.

The **finally** block always executes if the program begins executing code in the **try** block. If an exception occurs and is not caught, the **finally** block will execute before the program looks at the calling code for a matching catch handler.

You can write a **try-finally** block (without **catch** blocks) to guarantee that certain code will execute once the **try** block begins. This is useful for opening a resource, like a file or database connection, and then guaranteeing you will be able to close that resource regardless of whether an exception occurs.

If you encounter a reason why your method can't perform its intended purpose, **throw** an exception. There are many **Exception**-derived types in the .NET FCL that you can use. The following code example pulls together a few concepts you might want to use, such as validating method input and throwing an **ArgumentNullException**.

```
public class Address
{
    public string City { get; set; }
}

internal class Company
{
    public Address Address { get; set; }
}

        // Inside of a class...
        static void ThrowException()
        {
            try
            {
                ValidateInput("something", new Company());
            }
            catch (ArgumentNullException ex) when (ex.ParamName == "inputString")
            {
                Console.WriteLine("From ArgumentNullException: " + ex.Message);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("From ArgumentException: " + ex.Message);
```

```
        }
    }

    static void ValidateInput(string inputString, Company cmp)
    {
        if (inputString == null)
            throw new ArgumentNullException(nameof(inputString));

        if (cmp?.Address?.City == null)
            throw new ArgumentNullException(nameof(cmp));
    }
```

Code Listing 50

The previous code shows an **Address** class and a **Company** class with a property of the **Address** type. The **try** block of the **ThrowException** message passes a new instance of **Company**, but doesn't instantiate **Address**, meaning that the **Company** instance's **Address** property is **null**.

Inside **ValidateInput**, the **if** statement uses the null referencing operator, **?.**, to check if any of the values between **Company**, **Company**'s **Address** property, or **Address**'s **City** property is **null**. This is a convenient way to check for **null** without a group of individual checks, producing less syntax and simpler code. If any of these values are **null**, the code throws an **ArgumentNullException**.

The argument to the **ArgumentNullException** uses the **nameof** operator, which evaluates to a **string** representation of the value passed to it; it is **"cmp"** in this case. This code isn't enclosed in a **try** block, so control returns to the code calling this method.

Back in the **ThrowException** method, the thrown exception causes the code to look for a handler suitable for this exception type. The exception type is **ArgumentNullException**, but the **catch** block for **ArgumentNullException** won't execute. That's because the **when** clause following the **ArgumentNullException** **catch** block parameter is checking for a **ParamName** of **"inputString"**. This **when** clause is called an exception filter. As mentioned previously, the parameter name passed to the **ArgumentNullException** during instantiation was **"cmp"**, so there isn't a match. Therefore, the code continues looking at **catch** handlers.

Since **ArgumentNullException** derives from **ArgumentException** and there is no exception filter, the **catch** handler for **ArgumentException** executes. The exception is now handled.

Tip: It's typically better to throw and handle specific exceptions, rather than their parent exceptions. This adds more fidelity and meaning to the exception and makes debugging easier.

Summary

Methods help you organize code into named functions that you can call to perform operations. Their name documents what the code does. Also, methods are useful to help avoid duplicating the same code in multiple places. Properties are used like fields and look like fields from the perspective of code using the property's class. However, properties are more sophisticated in that they have **`get`** and **`set`** accessors that let them work like methods and perform more sophisticated work, like validation or special value handling. Both methods and properties help define the interface of a class to consumers and let you encapsulate the internal operations of a class, which makes it more reusable. You can use **`try-catch`** blocks to handle exceptions and **`try-finally`** blocks to guarantee critical code executes. Use the **`throw`** statement whenever a method you're writing can't fulfill its intended purpose.

Chapter 4 Writing Object-Oriented Code

C# is an object-oriented programming (OOP) language. It supports inheritance, encapsulation, polymorphism, and abstraction. This chapter shows you how C# supports OOP.

Implementing Inheritance

In C#, inheritance defines a relationship between two classes where a derived class can reuse members of a base class. A simple way to view this is that a derived class is a more specific version of a base class. We will reuse the calculator example from previous chapters, but alter it to provide two different types of calculators: scientific and programmer. Since they're both calculators, it could be useful to create a relationship with a **`Calculator`** base class and **`ScientificCalculator`** and **`ProgrammerCalculator`** derived classes, like this:

- **`Calculator`**
  - **`ScientificCalculator`**
  - **`ProgrammerCalculator`**

In C#, you would express this relationship as follows.

```
public class Calculator { }
public class ScientificCalculator : Calculator { }
public class ProgrammerCalculator : Calculator { }
```

Code Listing 51

As you can see, we added a colon suffix (as the inheritance operator) to the derived class and specified the base class it is derived from. The following figure illustrates the inheritance relationship between these classes.

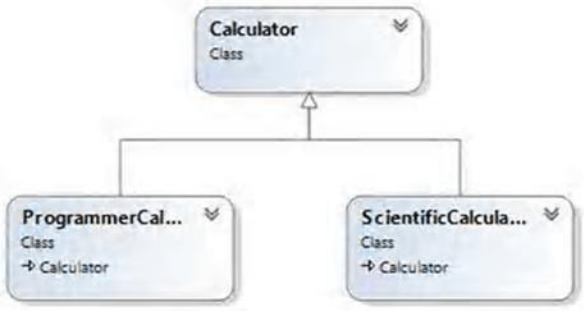


Figure 1: Calculator Inheritance Diagram

You can assume that **`Calculator`** will have all of the standard operations like addition, subtraction, and more that all calculators have. The following code listing is an expanded example that shows the base class with a common method, and the derived classes with specialized methods that only belong to those classes.

```
using System;

public class Calculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ScientificCalculator : Calculator
{
    public double Power(double num, double power)
    {
        return Math.Pow(num, power);
    }
}

public class ProgrammerCalculator : Calculator
{
    public int Or(int num1, int num2)
    {
        return num1 | num2;
    }
}
```

Code Listing 52

The methods of the derived classes in the previous example used the FCL **`Math`** class for **`Power`**, which has many more math methods you can use in your own code, and also used the built-in C# **`|`** operator for **`Or`**. The following example shows how to write code that takes advantage of inheritance with the previous classes.

```
using System;

public class Program
{
    public static void Main()
    {
        ScientificCalculator sciCalc = new ScientificCalculator();
        double powerResult = sciCalc.Power(2, 5);
        Console.WriteLine($"Scientific Calculator 2**5: {powerResult}");
        double sciSum = sciCalc.Add(3, 3);
        Console.WriteLine($"Scientific Calculator 3 + 3: {sciSum}");

        ProgrammerCalculator prgCalc = new ProgrammerCalculator();
        double orResult = prgCalc.Or(5, 10);
        Console.WriteLine($"Programmer Calculator 5 | 10: {orResult}");
```

```
        double prgSum = prgCalc.Add(3, 3);
        Console.WriteLine($"Programmer Calculator 3 + 3: {prgSum}");

        Console.ReadKey();
    }
}
```

Code Listing 53

Both the **ScientificCalculator** instance, **sciCalc**, and the **ProgrammerCalculator** instance, **prgCalc**, call the **Add** method. Further, those classes don't define their own **Add**, but they do derive from **Calculator** and therefore inherit **Calculator**'s **Add**.

The ability to inherit isn't always guaranteed; the next section explains more about when a class member is visible to other classes.

Access Modifiers and Encapsulation

In the previous example, all classes and methods had **public** modifiers, meaning that any other class or code can see and access them in code. You can leave off access modifiers and accept defaults. In that case, a class access becomes what is known as **internal**, and class members default to **private**.

Classes can only be **internal** or **public**. If they're **internal**, they can only be accessed by code inside of the assembly they are contained in.

Available access modifiers for class members include **public**, **private**, **internal**, **internal protected**, and **protected**. The **public** and **internal** modifiers have the same meaning for class members as they do for classes.

The default modifier for class members, **private**, means that code outside the class can't use that member; only other members within the same class can use it. This is useful if you want to modularize a method by breaking it into different supporting methods, but the supporting methods have no meaning outside of the class.

The **protected** modifier allows derived classes inside and outside the assembly to use the **protected** base class member. The **internal protected** modifier further restricts protected behavior only to derived classes inside the same assembly.

Most of the access modifier defaults and behaviors apply to **struct** types as well as classes, except for **protected** and **internal protected**, as I'll explain next.

Designing Types: Class vs. Struct

A **struct** is another C# type that looks similar to a **class**, but has different behavior. A **struct** can't derive from another **class** or **struct**. Since implementation inheritance doesn't apply to a **struct**, neither do **protected** and **internal protected** modifiers. A **struct** does have

interface inheritance, which I'll explain more in the Exposing Interfaces section later in this chapter.

Additionally, a **struct** copies by value, as opposed to a **class** which copies by reference. The difference is that if you pass a **struct** instance to a method, the method gets a brand new copy of the **struct** value. If you copy a **class** instance to a method, the method gets a copy of a reference to the **class** in the heap, which is an area in computer memory that the CLR uses to allocate space for reference type objects. These facts help you decide whether you should design a type as a **class** or **struct**. Imagine a type with many properties and how it would hurt performance if you had to pass it by value to a method as a **struct**; the state of that type is copied to the stack, which is memory the CLR allocates for every method call to hold items like parameters and local variables. In that case, the proper design decision might be to define the type as a **class** so that only the reference is copied.

Most of the built-in types, such as **int**, **double**, and **char**, are value types. If you have a type with those semantics—small and a single value—then designing a type as a **struct** might be a benefit. Otherwise, designing a type as a **class** is fine. Here's an example of a type that might make a good **struct**.

```
public struct Complex
{
    public Complex(double real, double imaginary)
    {
        Real = real;
        Imaginary = imaginary;
    }

    public double Real { get; set; }

    public double Imaginary { get; set; }

    public static Complex operator +(Complex complex1, Complex complex2)
    {
        Complex complexSum = new Complex();
        complexSum.Real = complex1.Real + complex2.Real;
        complexSum.Imaginary = complex1.Imaginary + complex2.Imaginary;
        return complexSum;
    }
    public static implicit operator Complex(double dbl)
    {
        Complex cmplx = new Complex();
        cmplx.Real = dbl;
        return cmplx;
    }
    // This is not a safe operation.
    public static explicit operator double(Complex cmplx)
    {
        return cmplx.Real;
    }
}
```

Code Listing 54

Complex could make a good **struct** because you might have a lot of mathematical operations, and it would be more efficient to pass a copy of the numbers on the stack rather than letting the CLR allocate memory as it does for a **class**.

Complex has a constructor, named after the class itself, with a couple parameters. This makes it easy to initialize a new instance of **Complex**.

There are a few operator overloads in **Complex**: an addition operator and two conversion operators. The addition operator lets you add two complex numbers. Where the operator identifier (**+**) precedes the parameter list, the values to be added are specified in the parameters, and the return type is part of the signature. Operators are always **static**.

The two conversion operators let you make assignments between the containing type and another type of your choice. The type assigned to is the operator identifier and the type being assigned is the parameter. The **implicit** modifier means the conversion is safe and the **explicit** modifier means the conversion has the potential to lose data or provide an invalid result. For example, assigning a **double** to an **int** would be **explicit** because of loss of precision, and the explicit conversion in the previous example causes loss of the imaginary part of the number. The following code sample demonstrates how **Complex** could be used.

```
using System;

class Program
{
    static void Main()
    {
        Complex complex1 = new Complex();
        complex1.Real = 3;
        complex1.Imaginary = 1;

        Complex complex2 = new Complex(7, 5);

        Complex complexSum = complex1 + complex2;

        Console.WriteLine(
            $"Complex sum - Real: {complexSum.Real}, " +
            $"Imaginary: {complexSum.Imaginary}");

        Complex complex3 = 9;

        double realPart = (double)complex3;

        Console.ReadKey();
    }
}
```

Code Listing 55

The **Main** method instantiates **complex1** and then populates its values. Next, **Main** instantiates **complex2** by using the **Complex** constructor, which is simpler initialization code.

You can also see how natural it is to use the addition operator, rather than an **Add** method used in previous **Calculator** demos.

Because there's an implicit conversion from **int** to **double** and **Complex** has an implicit conversion operator from **double** to **Complex**, **Main** is able to assign **9** to **complex3**. The same can't be said for assigning **complex3** to **realPart** because **Complex** to **double** is an **explicit** conversion operator in the **Complex** type. Any time you have an **explicit** conversion, you must use a cast operator, as in **(double)complex3**.

One of the items you need to watch out for when working with value types is a concept referred to as boxing and unboxing. Boxing occurs any time you assign a value type to **object**, and unboxing occurs when you assign **object** to a value type. The following code demonstrates one scenario where this could happen.

```
ArrayList intCollection = new ArrayList();
intCollection.Add(7);
int number = (int)intCollection[0];
```

Code Listing 56

An **ArrayList** is a collection class belonging to the **System.Collections** namespace. It is more powerful than an array and operates on type **object**. The **Add** method accepts an argument of type **object**. Since all types derive from **object**, an **ArrayList** is flexible enough to allow you to work with objects of any type. Boxing occurs when passing **7** to the **Add** method because **7** is an **int** (a value type) and is converted to **object**. What is really happening is that the CLR creates a boxed **int** in memory. Since the **ArrayList** holds type **object**, you also need to perform a conversion to unbox a value when reading from the **ArrayList**. The **(int)** cast operator converts from **object** (the boxed **int**) to **int** when reading the first element of **intCollection**.

The problem that boxing and unboxing cause is related to performance. In this situation, the reason you would use a collection is because you want to hold a lot of **int** values, which could be hundreds or thousands. Think about all the time spent accessing that **ArrayList** and incurring the performance penalty of boxing and unboxing on each operation.

Note:** `ArrayList` **is an old collection class that existed in C# v1.0 and is no longer used in modern development. C# v2.0 introduced generics, which use new collection classes that are strongly typed and avoid the boxing and unboxing penalties. While the** `ArrayList` **example is unlikely today, this scenario still highlights the performance penalty of any other situation where you might be assigning a value type to an object type.

Another difference between **class** (reference types) and **struct** (value types) is equality evaluation. Value type equality works by comparing the corresponding members of the **struct**. Reference type equality works by verifying that references are equal. In other words, structs are equal if their values match, but classes are equal if they reference the same object in memory.

In the later section on polymorphism, you'll learn how to override the **object.Equals** method to give you more control over class equality.

Creating Enums

An **enum** is a value type that lets you create a set of strongly typed mnemonic values. They're useful when you have a finite set of values and don't want to represent those values as strings or numbers. Here's an example of an **enum**.

```
public enum MathOperation
{
    Add,
    Subtract,
    Multiply,
    Divide
}
```

Code Listing 57

Like a **struct**, an **enum** is a value type. You use the **enum** keyword as the type definition. The previous **enum** is named **MathOperation** and has four members. The following example shows how you can use this **enum**.

```
using System;
using static MathOperation;

class Program
{
    static void Main()
    {
        string[] possibleOperations = Enum.GetNames(typeof(MathOperation));

        Console.Write($"Please select ({string.Join(", ", possibleOperations)}): ");

        string operationString = Console.ReadLine();

        MathOperation selectedOperation;

        if (!Enum.TryParse<MathOperation>(operationString, out selectedOperation))
            selectedOperation = MathOperation.Add;

        switch (selectedOperation)
        {
            case MathOperation.Add:
                Console.WriteLine($"You selected {nameof(Add)}");
                break;
            case MathOperation.Subtract:
                Console.WriteLine($"You selected {nameof(Subtract)}");
                break;
            case MathOperation.Multiply:
                Console.WriteLine($"You selected {nameof(Multiply)}");
```

```
                break;
            case MathOperation.Divide:
                Console.WriteLine($"You selected {nameof(Divide)}");
                break;
        }

        Console.ReadKey();
    }
}
```

Code Listing 58

The FCL has an **Enum** class that lets you work with enums and the previous **Main** method shows how to use a couple of its methods. **Enum.GetNames** returns a **string** array, representing the names in the **enum**, specified with the **typeof** operator. The **string.Join** method, the expression in the interpolated string of the **Console.WriteLine**, creates a comma-separated string of these names.

The **Enum.TryParse** method in the previous example takes a string and produces an **enum** of the type specified in the type parameter, which is **MathOperation** in this case. The **out** parameter means that **TryParse** will return the parsed value in the **selectedOperation** variable. This is practical because the return type of the **TryParse** is **bool**, allowing you to know whether the input string, **operationString**, is valid.

The **selectedOperation** variable is of type **MathOperation**. The default syntax for enums is to prefix them with the enum type name, as in **MathOperation.Add**. However, you can also add a **using static** clause to the top of the file, allowing you to only specify the member name, as the previous example shows in the **switch** statement. A **switch** statement can operate on numbers, strings, or enums.

Enabling Polymorphism

Polymorphism lets derived classes specialize a base class implementation. The mechanism to allow polymorphism is to decorate a base class method with the **virtual** modifier and decorate the derived class method with the **override** modifier. If you were designing the **Calculator** class, you could allow derived classes to implement their own improved or specialized versions of the **Add** method, as shown in the following sample.

```
using System;

public class Calculator
{
    public virtual double Add(double num1, double num2)
    {
        Console.WriteLine("Calculator Add called.");
        return num1 + num2;
    }
}
```

```
public class ProgrammerCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ProgrammerCalculator Add called.");
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ScientificCalculator : Calculator
{
    public override double Add(double num1, double num2)
    {
        Console.WriteLine("ScientificCalculator Add called.");
        return base.Add(num1, num2);
    }
}
```

Code Listing 59

Polymorphism is opt-in for C#. Notice that the **`Add`** method in the base class **`Calculator`** has a **`virtual`** modifier. Polymorphism doesn't occur unless a base class method has the **`virtual`** modifier. Also, notice that the derived classes **`ScientificCalculator`** and **`ProgrammerCalculator`** have **`override`** modifiers. Again, these methods won't be called polymorphically unless they have the **`override`** modifier. Additionally, a method with the **`override`** modifier is also **`virtual`** for any of its derived classes.

With polymorphism, the overridden method in derived classes executes at runtime. If you wanted to call the base class implementation of that method, call the base class method with the **`base`** keyword. **`ScientificCalculator`** calls **`base.Add(num1, num2)`** to call the **`Add`** method in **`Calculator`**. Here's an example of how this works.

```
using System;

public class Program
{
    public static void Main()
    {
        Calculator sciCalc = new ScientificCalculator();
        double sciCalcResult = sciCalc.Add(2, 5);
        Console.WriteLine($"Scientific Calculator 2 + 5: {sciCalcResult}");

        Calculator prgCalc = new ProgrammerCalculator();
        double prgCalcResult = prgCalc.Add(5, 10);
        Console.WriteLine($"Programmer Calculator 5 + 10: {prgCalcResult}");
```

```
        Console.ReadKey();
    }
}
```

Code Listing 60

The output for this program would be:

```
ScientificCalculator Add called.

Calculator Add called.

Scientific Calculator 2 + 5: 7

ProgrammerCalculator Add called.

Programmer Calculator 5 + 10: 15
```

Main assigns instances of **ScientificCalculator** and **ProgrammerCalculator** to variables of type **Calculator**. As you saw in the previous listing, **ScientificCalculator** and **ProgrammerCalculator** are derived types and **Calculator** is their base type. The derived instances are the runtime type—the actual type when the program is running—and the base class is the compile-time type. The runtime-type overrides execute at runtime.

Looking at the definition of **Add** in **ScientificCalculator**, **Calculator**, and **Main**, and looking at the output, you can trace the polymorphic behavior of this program. **Main** calls **Add** on the **ScientificCalculator** instance. **ScientificCalculator.Add** executes because it overrides the **virtual Calculator.Add** method. After writing the first line of output, **ScientificCalculator.Add** calls the **Calculator.Add** method with the **base** keyword. **Calculator.Add** prints the second line to the output, performs the addition calculation, and returns the sum. **ScientificCalculator.Add** returns the return value from **Calculator.Add**. **Main** assigns the return value from **ScientificCalculator.Add** to the **sciCalc** variable and prints the results into the third line of the output. Tracing the call to **ProgrammerCalculator.Add** is similar, except that there is no call to the **Calculator.Add** in the base class.

Another example of when you would want to use polymorphism is in defining reference type equality. By default, reference types are only equal if their references are the same. The following example shows how to control reference type equality.

```
public class Customer
{
    int id;
    string name;

    public Customer(int id, string name)
    {
        this.id = id;
        this.name = name;
```

```
    }

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;

        if (obj.GetType() != typeof(Customer))
            return false;

        Customer cust = obj as Customer;

        return id == cust.id;
    }

    public static bool operator ==(Customer cust1, Customer cust2)
    {
        return cust1.Equals(cust2);
    }

    public static bool operator !=(Customer cust1, Customer cust2)
    {
        return !cust1.Equals(cust2);
    }

    public override int GetHashCode()
    {
        return id;
    }

    public override string ToString()
    {
        return $"{{ id: {id}, name: {name} }}";
    }
}
```

Code Listing 61

Since all classes implicitly derive from **`object`**, they can **`override`** object **`virtual`** methods **`Equals`**, **`GetHashCode`**, and **`ToString`**. **`Customer`** overrides **`Equals`**. When you override **`Equals`**, check for **`null`** and type equality before working with the objects to prevent callers from accidentally comparing **`null`** or incompatible types. **`Customer`** instances are equal if they have the same **`id`**.

`Customer` has a constructor that initializes the state of the class. The **`this`** operator lets you access members of the containing instance and helps avoid ambiguity.

When implementing custom equality, you should also overload the equals and not equals and override the **`GetHashCode`** method. The default implementation of **`GetHashCode`** is a system-defined object **`id`**, so you could override this to achieve a better distribution of values in a hash.

Tip: You can escape { and } characters that you don't want to evaluate as expressions by doubling them as {{ and }} respectively in string interpolation.

The following is an example of how to check equality of **`Customer`** instances.

```
using System;

class Program
{
    static void Main()
    {
        Customer cust1 = new Customer(1, "May");
        Customer cust2 = new Customer(2, "Joe");

        Console.WriteLine($"cust1 == cust2: {cust1 == cust2}");

        Customer cust3 = new Customer(1, "May");

        Console.WriteLine($"\ncust1 == cust3: {cust1 == cust3}");
        Console.WriteLine($"cust1.Equals(cust3): {cust1.Equals(cust3)}");
        Console.WriteLine($"object.ReferenceEquals(cust1, cust3):
{object.ReferenceEquals(cust1, cust3)}");

        Console.WriteLine($"\ncust1: {cust1}");
        Console.WriteLine($"cust2: {cust2}");
        Console.WriteLine($"cust3: {cust3}");

        Console.ReadKey();
    }
}
```

Code Listing 62

When using the **`==`** operator, the code calls the operator overload and **`Equals`** calls the equals method as expected. **`ReferenceEquals`** is an **`object`** method that is useful because it allows reference equality checking in case the type defined a custom **`Equals`** override.

If **`Customer`** had not overridden **`ToString`**, the last three **`Console.WriteLine`** statements in the previous code listing would have printed the type name, which is the default behavior of **`ToString`**.

Writing Abstract Classes

In previous examples, you could create an instance of **`Calculator`**. However, it may or may not make sense to instantiate a base class. A base class may serve only as a reusable type for common functionality of similar derived classes and to enable polymorphism, yet not have substance enough to be used on its own. In that case, you can modify the class definition as **`abstract`**, as shown in the following sample.

```
public abstract class Calculator
{
    // ...
}
```

Code Listing 63

In an **abstract** class, you can have **virtual** or non-virtual members. Additionally, you can have **abstract** methods. An **abstract** method doesn't have an implementation. Derived classes should specify the implementation and you don't want a default implementation in the base class that might not make sense. The purpose of an **abstract** method is to specify an interface that derived classes must implement. In the case of **Calculator**, you could define an **abstract Add** method as shown in the following code example.

```
public abstract class Calculator
{
    public abstract double Add(double num1, double num2);
}
```

Code Listing 64

The **Add** method has an **abstract** modifier. This method is implicitly virtual, but can't be called by a derived class because it doesn't have an implementation. The semicolon is required to terminate the **abstract** method signature. When an **abstract** class has **abstract** methods, all derived classes must **override** the **abstract** method. The **Main** method in the previous section still runs if you change the definition of the non-abstract **Calculator** class to the previous **abstract Calculator**.

Abstract classes are nice for situations where you want to have some default behavior, specify what the public interface of a class is, and support polymorphism. However, there are some limitations in that a C# class can have only one base class. Additionally, a struct can't inherit another class or struct, so they don't help if you need to write code that allows you to replace any number of value types with a base class implementation. There is an alternative, which I'll discuss next.

Exposing Interfaces

If you only wanted a base class that specified an interface for a common set of operations, you could create an abstract class with only abstract methods. This ensures that all derived classes have those abstract methods. However, there's a better alternative, named after what it does: an **interface**.

The benefit of the **interface** type is that both **class** and **struct** types can inherit multiple interfaces. You can also implement polymorphism with interfaces. They don't have any implementation and you must write the implementation in your derived class. The following code listing shows the **Calculator** class rewritten as an interface.

```
public interface ICalculator
{
    double Add(double num1, double num2);
}
```

Code Listing 65

Instead of **`class`** or **`struct`**, you'll use the **`interface`** type. A common convention for interface identifiers is the **`I`** prefix, as in **`ICalculator`**. Interface methods are implicitly public and virtual, so you don't need access, abstract, or virtual modifiers. Like **`abstract`** methods, **`interface`** methods have a signature, but don't have an implementation. Developers provide that implementation in their classes that derive from interfaces. The following code sample is a revision of the previous classes to implement the **`ICalculator`** interface.

```
public class ScientificCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}

public class ProgrammerCalculator : ICalculator
{
    public double Add(double num1, double num2)
    {
        return MyMathLib.Add(num1, num2);
    }
}

public class MyMathLib
{
    public static double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

Code Listing 66

Deriving from an interface uses the same syntax as deriving from a class in that you add a colon and interface name after the class name. Unlike virtual methods, you don't use an **`override`** modifier on methods.

A derived class implementation must be **`public`**. This make sense because an interface defines a contract that any derived class will have members defined in the interface. That means any time you use a class through its interface, you know that it will have the members defined by an interface. The following code example is a modification of the **`Main`** method that uses the **`ICalculator`** interface.

```
using System;
```

```
public class Program
{
    public static void Main()
    {
        ICalculator sciCalc = new ScientificCalculator();
        double sciCalcResult = sciCalc.Add(2, 5);
        Console.WriteLine($"Scientific Calculator 2 + 5: {sciCalcResult}");

        ICalculator prgCalc = new ProgrammerCalculator();
        double prgCalcResult = prgCalc.Add(5, 10);
        Console.WriteLine($"Programmer Calculator 5 + 10: {prgCalcResult}");

        Console.ReadKey();
    }
}
```

Code Listing 67

The only syntax difference between this example and the one previous to that is the compile-time type of **sciCalc** and **prgCalc** is **ICalculator**. Because each variable is an **ICalculator**, you can be guaranteed that the runtime type implements the members of that interface.

Interfaces can also inherit other interfaces. In that case, derived classes must implement all members of each interface in the inheritance chain. Also, a **class** or **struct** can implement multiple interfaces, which is demonstrated in the following sample.

```
public interface ICalculator { }
public interface IMath { }

public class ScientificCalculator : ICalculator, IMath
{
    public double Add(double num1, double num2)
    {
        return num1 + num2;
    }
}
```

Code Listing 68

After the first interface, additional interfaces appear in a comma-separated list. A class or struct must implement the methods of all interfaces it is derived from.

Object Lifetime

The lifetime of a value type (**struct** or **enum**) depends on where it's allocated. Parameter and variable value type instances reside on the stack and exist for as long as they are in scope. Reference type instances (**class**) begin life when their constructors execute. The CLR allocates their space on the managed heap, and they exist until the CLR garbage collector (GC) cleans them up.

You can use constructors to initialize a class. While doing so, you can also affect initialization of static state, base types, and other constructor overloads. The following demo shows several features of class initialization.

```
using System;

public class Calculator
{
    static double pi = Math.PI;
    double startAngle = 0;

    public DateTime Created { get; } = DateTime.Now;

    static Calculator()
    {
        Console.WriteLine("static Calculator()");
    }

    public Calculator()
    {
        Console.WriteLine("public Calculator()");
    }

    public Calculator(int val)
    {
        Console.WriteLine("public Calculator(int)");
    }
}
```

Code Listing 69

`Calculator` has a **`static`** constructor and two instance constructor overloads. A **`static`** constructor executes one time for the lifetime of the object and before the first constructor executes. The following sample is a derived class with similar members.

```
using System;

public class ScientificCalculator : Calculator
{
    static double pi = Math.PI;
    double startAngle = 0;

    static ScientificCalculator()
    {
        Console.WriteLine("static ScientificCalculator()");
    }

    public ScientificCalculator() : this(0)
    {
        Console.WriteLine("public ScientificCalculator()");
    }

    public ScientificCalculator(int val)
    {
```

```
        Console.WriteLine("public ScientificCalculator(int)");
    }

    public ScientificCalculator(int val, string word) : base(val)
    {
        Console.WriteLine("public ScientificCalculator(int, string)");
    }

    public double EndAngle { get; set; }
}
```

Code Listing 70

ScientificCalculator derives from **Calculator** and has similar constructors, except for the **this** and **base** operators. Using the **this** operator calls the constructor overload with the matching parameters. Since **0** is an **int**, the default (no parameter) constructor calls **ScientificCalculator(int val)** first. The **base** operator calls the matching constructor in the base class, so calling **base(0)** calls **Calculator(int val)** first. The following code listing is a program that instantiates these classes.

```
using System;

class Program
{
    static void Main()
    {
        var calc1 = new ScientificCalculator();

        var calc2 = new ScientificCalculator(0, "x")
        {
            EndAngle = 360
        };

        Console.ReadKey();
    }
}
```

Code Listing 71

And here is the program's output:

```
static ScientificCalculator()

static Calculator()

public Calculator()

public ScientificCalculator(int)

public ScientificCalculator()

public Calculator(int)
```

```
public ScientificCalculator(int, string)
```

Viewing the output, you can see what executes first. Here are the rules that govern the instantiation of these classes:

- Static constructors execute before instance constructors.
- Static constructors execute one time for the life of the program.
- Base class constructors execute before derived class constructors.
- The **this** operator causes an overloaded constructor that matches the **this** parameter list to execute first.
- The base class default constructor executes, unless the derived class uses base to explicitly select a different base class constructor overload.
- This is not shown in the output of the previous example, but static fields initialize before the static constructor and instance fields initialize before instance constructors.
- Auto-implemented property initializers, such as **Created**, initialize at the same time as fields.
- Properties in object initialization syntax execute last as object initialization syntax is equivalent to populating the property through the instance variable after instantiation.

Note: In Visual Studio, you can set break points in the code and use the Immediate Window to inspect field values. You can experiment with different object initialization scenarios to get a feel for the initialization sequence.

Of all these lifecycle events, garbage collection (GC) is the least predictable. The CLR optimizes resources and runs GC when it needs to. This means that the lifetime of a reference type object is non-deterministic. There's a rich body of theoretical discussion around the how and why of GC, but I'll restrict that debate to the practical consideration of resource management. This includes closing files, database connections, operating system handles, and more.

To release resources, there's a pattern commonly referred to as the Dispose Pattern. It relies on the **IDisposable** interface, flags that manage the disposal state of the object, and a destructor. The following code has constructor and **Dispose** method comments that imply a scenario where the class could be logging operations during its lifetime, and the log should be opened during instantiation and closed when the object is no longer needed.

```
using System;

public class Calculator : IDisposable
{
    static Calculator()
    {
        // Initialize log file stream.
    }

    #region IDisposable Support
    private bool disposedValue = false; // To detect redundant calls.

    protected virtual void Dispose(bool disposing)
    {
        if (!disposedValue)
```

```
        {
            if (disposing)
            {
                // TODO: dispose managed state (managed objects).
                // Close log file stream.
            }

            // TODO: free unmanaged resources (unmanaged objects) and override a
finalizer below.
            // TODO: set large fields to null.

            disposedValue = true;
        }
    }

    // TODO: override a finalizer only if Dispose(bool disposing) above has code to
free unmanaged resources.
    // ~Calculator() {
    //   // Do not change this code. Put cleanup code in Dispose(bool disposing) above.
    //   Dispose(false);
    // }

    // This code added to correctly implement the disposable pattern.
    public void Dispose()
    {
        // Do not change this code. Put cleanup code in Dispose(bool disposing) above.
        Dispose(true);
        // TODO: uncomment the following line if the finalizer is overridden above.
        // GC.SuppressFinalize(this);
    }
    #endregion
}
```

Code Listing 72

The code between **#region** and **#endregion** is automatically generated by VS. To generate this code, select **IDisposable** in the editor and the Quick Action icon (a light bulb) will appear. Open the Quick Action menu and select **Implement interface with Dispose pattern**. The **#region** and **#endregion** let VS fold the code so you won't have to see it in the editor.

The **Calculator** class implements the **IDisposable** interface, which is only the **Dispose** method. The constructor initializes a resource you want to open, like a file handle or database, and the GC calls the destructor, **~Calculator()**, if it's uncommented. The **Dispose()** method calls **Dispose(bool)** with a **true** argument and **~Calculator()** calls **Dispose(bool)** with a **false** argument. This lets **Dispose(bool)** know whether it should clean up managed resources that belong to the CLR or unmanaged resources that belong to the operating system. The flag **disposedValue** helps to prevent the object from being disposed more than one time.

The following sample shows how calling code can use this class, disposing it when it is no longer needed.

```
        ScientificCalculator calc3 = null;
```

```
try
{
    calc3 = new ScientificCalculator();
    // Do stuff.
}
finally
{
    if (calc3 != null)
        calc3.Dispose();
}
```

Code Listing 73

This shows the reason **try-finally** exists, to guarantee that resources can be closed or disposed. Because the **finally** block is guaranteed to execute after code in the **try** block starts, **calc3** can be safely disposed. While that works, it's more verbose than necessary. The following listing simplifies the code.

```
using (var calc4 = new ScientificCalculator())
{
    // Do stuff.
}
```

Code Listing 74

The **using** statement accepts parameters with any type that implements **IDisposable**. It takes care of calling **Dispose()** after the block completes execution. Behind the scenes, the logic is similar to the previous **try-finally** block.

Summary

C# supports object-oriented programming. For inheritance, you have single inheritance for classes, multiple inheritance for interfaces, and structs that can only inherit interfaces. Use abstract classes for classes that shouldn't stand alone, but provide interface and structure to derived classes. Use interfaces when you don't have implementation, need value type (struct) polymorphism, or need to implement multiple interfaces. I also discussed structs and how they are ideal for situations where copy by value leads to performance gains and value type semantics make sense. Unlike interfaces that you need to be public, use encapsulation to hide the internal workings that you don't want other developers to use in their code. Polymorphism is a powerful concept that allows you to write a single algorithm that is coded the same for every instance, yet allows each instance to vary with an implementation specific to the runtime type of the instance. Pay attention to the sequence of object instantiation to ensure your types initialize correctly. If you need to dispose a type, make that type implement **IDisposable** with the Dispose Pattern. You can use a **using** statement to simplify the instantiation and safe cleanup of the type.

Chapter 5 Handling Delegates, Events, and Lambdas

Much of the user interface work you'll do is event based, and C# supports this through type members called events. For events to work, you need some infrastructure to specify methods that can be called, which surfaces through delegates and lambdas. This chapter will explain how delegates, events, and lambdas work in C#.

Referencing Methods with Delegates

Delegates have a few capabilities in C#: referencing methods, dispatching multiple methods, asynchronous execution, and event typing. This can be confusing because many other language features serve only a single purpose. Differentiating and comparing all of these capabilities of C# delegates adds complexity that you might not be familiar with. This discussion is going to cut the feature list somewhat to hopefully illuminate delegates and make them less complex as you move forward with practical implementation. In particular, I'll focus on delegates as method references and event types.

Note: I'll avoid deep discussion of delegate multi-cast and asynchronous execution because they're rarely used and largely replaced by other language features. For example, events support multi-cast dispatch and C# 5.0 introduces a capability referred to as async.

Let's first examine the role of a delegate as a reference to a method. To do this, the delegate specifies the signature of a method that it can reference, like this:

```
public delegate double Add(double num1, double num2);
```

Code Listing 75

You might notice that a delegate looks like an abstract method, except it has the **`delegate`** type definition keyword. A **`delegate`** definition is a reference type, just like a class, struct, or interface. The previous **`delegate`** definition is for a delegate type named **`Add`** that takes two **`double`** arguments and returns a result of type **`double`**. Just like other types, delegate accessibility can only be **`public`** or **`internal`** and is **`internal`** by default.

There are esoteric uses of delegates that I won't get into, but I do want to focus on the most practical and common way to use delegates: as event types.

Firing Events

Events are type members that allow a type to notify other types of things that have happened. A very common example is a user interface with a button. You'll want to write code that does something when a user clicks that button. Here are the pieces you need to make that happen:

1. A **`Button`** class, which is typically supplied by the UI technology you're using.
2. An event member of the **`Button`** class, named **`Clicked`**.
3. The UI technology takes care of knowing when that **`Clicked`** event should fire. I'll use a **`SimulateClick`** method in an upcoming code listing.
4. The event has a delegate type. Only methods that conform to the signature of that delegate type can assign to this delegate.
5. Your code defines a method to be called when that event fires.
6. The method you write must have a signature that matches the delegate type of the event. If the method signature doesn't match the event delegate type signature, the compiler won't let you assign that method to the delegate.

As you can see, delegates have a lot of moving parts. In particular, pay attention to #6. Delegates prevent you, or anyone, or anything from assigning an arbitrary method to an event. Here's an example that defines a delegate and a class with an event of that delegate type.

```
using System;

public class ClickEventArgs : EventArgs
{
    public string Name { get; set; }
}

public delegate void ClickHandler(object sender, ClickEventArgs e);

public class CalculatorButton
{
    public event ClickHandler Clicked;
}
```

Code Listing 76

An event can be a member of a class, struct, or interface. If it is an interface member, it means that classes or structs that implement that interface must also have the event in their definitions. An event has the **`event`** modifier and adheres to the same accessibility rules as other type members like methods and properties.

If a delegate serves the purpose you need, you can use it. In fact, the FCL includes many reusable types, including reusable delegate types that you can use without needing to create your own. There's even a .NET type named **`EventHandler`** that nearly matches the signature of **`ClickHandler`**, where the **`sender`** is typically the source of the event, and **`EventArgs`** is a base class you can derive from to create your own custom type for sharing information with methods and event calls when fired. The previous code, with **`ClickEventArgs`**, derives from **`EventArgs`**, a type that comes with the .NET Framework. The following example simulates an event to demonstrate how to write a method that handles that event in code.

```
using System;

public class CalculatorButton
{
    public event ClickHandler Clicked;
    public void SimulateClick()
    {
        if (Clicked != null)
        {
            ClickEventArgs args = new ClickEventArgs();
            args.Name = "Add";

            Clicked(this, args);
        }
    }
}

public class Program
{
    public static void Main()
    {
        Program prg = new Program();
        CalculatorButton calcBtn = new CalculatorButton();

        calcBtn.Clicked += new ClickHandler(prg.CalculatorBtnClicked);
        calcBtn.Clicked += prg.CalculatorBtnClicked;

        calcBtn.SimulateClick();

        Console.ReadKey();
    }

    public void CalculatorBtnClicked(object sender, ClickEventArgs e)
    {
        Console.WriteLine(
            $"Caller is a CalculatorButton: {sender is CalculatorButton} and is named
{e.Name}");
    }
}
```

Code Listing 77

Again, this example has a lot of moving parts, but they follow the list at the start of this section about defining and using events. First, notice that **`CalculateButton`** has a new method, **`SimulateClick`**. Since we simplified the code by avoiding the UI, we have to fake a user clicking a button. That said, **`SimulateClick`** demonstrates the proper way to fire an event in your own code. Before firing an event, make sure that a user has assigned methods to the event by checking for **`null`**. Whenever no methods are assigned, the event will be **`null`**. **`SimulateClick`** sets up the **`ClickEventArgs`** parameter. In this case, it's only a **`Name`** property, but you would provide any relevant information available for the **`EventArgs`** type you were using and what information a method that received this event might need. Next, fire the event by calling it like a method. This causes the event to call each method assigned to it, one by one, in the order they were assigned. The first parameter is the **`this`** keyword, which indicates that the

instance of the containing type, **CalculatorButton**, gets passed to the methods. The second is the **ClickEventArgs** variable that was previously instantiated and had its **Name** property set.

The **Main** method shows how to assign methods to events. Notice the **+=** operator is used twice to assign two methods to the **CalculateButton** instance, **calcBtn**, and **Clicked** event. The **CalculatorBtnClicked** method is an instance method, so the **prg** instance provides access to that method during assignment.

The first assignment creates a new instance of the **ClickHandler** delegate. Delegates are types and you can instantiate them. You instantiate delegates with a method, which becomes the method the delegate refers to. Remember how I explained that delegates are references to methods? In this case, the new instance of the **ClickHandler** delegate refers to the **CalculatorBtnClicked** method. The second assignment shows a newer and simpler syntax for accomplishing the same task as the first; it just uses the method name. This is called delegate inference and means that since the method assigned to the event has the same signature of the event's delegate, the C# compiler will take care of instantiating the delegate with that method behind the scenes for you.

Finally, calling **SimulateClick** on the **CalculatorButton** instance, **calcBtn**, fires the event as explained previously. Regardless of whether the program assigns the same method to the event twice, firing the event causes all assigned delegates to fire, which calls the methods assigned to each delegate to execute. Therefore, the **CalculatorBtnClicked** method will execute two times.

You might wonder why I had to define **SimulateClick** inside of **CalculatorButton** instead of just firing the event from **Main**. The reason is because external code can't fire an event. An event can only be fired from inside of its containing type.

Instead of assigning named methods, you can assign code blocks directly to events.

Working with Lambdas

A lambda is a nameless method. Sometimes you have a block of code that serves one specific purpose and you don't need to define it as a method. Methods are nice because they let you modularize your code and have something to refer to with delegates and call from multiple places. But a lot of times you just need to run some code for a specific operation. Lambdas are quick and simple ways to assign and execute a block of code.

Note: A lambda is also a very sophisticated language feature that lets you translate between parse trees and code. This is a core feature of Language Integrated Query (LINQ), which I'll discuss more in Chapter 7. For daily practical use, working with lambdas as parse trees is rare.

Just like methods, lambdas can have parameters, a body, and can return values. The following code listing is an example of a lambda.

```
using System;

public class Program
{
    public static void Main()
    {
        Action hello = () => Console.WriteLine("Hello!");
        hello();

        Console.ReadKey();
    }
}
```

Code Listing 78

Action is a reusable delegate in the .NET Framework, and **hello** is variable of the **Action** delegate type. The lambda starts with an empty parameter list, meaning no parameters. The **=>** operator separates the parameter list and body and is referred to as either "such that" or "goes to". Next, you see the body, which is a single statement. Since hello is a delegate, you can call it just like a method and it will execute the lambda, which prints "Hello!" to the console.

With a single statement, you don't need to use curly braces on the body, but you do with multiple statements, as in the following example.

```
using System;

public class Program
{
    public static void Main()
    {
        Predicate<string> validator =
            word =>
            {
                int count = word.Length;
                return count > 3;
            };
        ValidateInput(validator);
        ValidateInput(word =>
        {
            int count = word.Length;
            return count > 3;
        });

        Console.ReadKey();
    }



    public static void ValidateInput(Predicate<string> validator)
    {
        string input = "Hello";
```

```
        bool isValid = validator(input);
        Console.WriteLine($"Is Valid: {isValid}");
    }
}
```

Code Listing 79

The previous example assigns a lambda to the .NET Framework's **Predicate** delegate, which is designed to return a **bool**. The lambda has a single parameter, **word**, of type **string**. Since the lamba has more than one statement, it requires curly braces. The example shows how to pass the lambda both as a variable and as an entire lambda.

Predicate is a generic delegate. The type parameter is set to **<string>**, meaning that the lambda parameter is type **string**. You'll learn more about generics in Chapter 6.

The **ValidateInput** method passes a **string** to **validator** and assigns the results to the **isValid** variable. It's just like a method call, except there isn't a method, just code; it is quick to write and limited in scope.

Another way to use lambdas is with events. The following example shows a different way to write an event handler method for the **Clicked** event in the previous **CalculatorButton** example.

```
using System;

public class Program
{
    public static void Main()
    {
        CalculatorButton calcBtn = new CalculatorButton();

        calcBtn.Clicked += (object sender, ClickEventArgs e) =>
        {
            Console.WriteLine(
                $"Caller is a CalculatorButton: {sender is CalculatorButton} and is
named {e.Name}");

            Console.WriteLine(message);
        };

        calcBtn.SimulateClick();

        Console.ReadKey();
    }
}
```

Code Listing 80

The first thing you might notice in this example is that the delegate assignment to the **Clicked** event is now a lambda. If you have two or more parameters, they must be enclosed in parentheses as a comma-separated list. If the body of the lambda includes two or more lines,

they must be terminated with semicolons and enclosed in braces. Notice that the signature of the lambda matches the **`ClickEventHandler`** defined previously.

More FCL Delegate Types

In addition to the **`Action`** and **`Predicate<T>`** delegates you've seen in previous examples, the FCL has a set of delegates named **`Func<T>`** that you can reuse at will. Here's an example that rewrites the previous example, using **`Func<T, TResult>`** instead of **`Predicate<T>`**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

class Program
{
    public static void Main()
    {
        Func<string, bool> validator =
            word =>
            {
                int count = word.Length;
                return count > 3;
            };
        ValidateInput(validator);
        ValidateInput(word =>
        {
            int count = word.Length;
            return count > 3;
        });

        Console.ReadKey();
    }

    public static void ValidateInput(Func<string, bool> validator)
    {
        string input = "Hello";
        bool isValid = validator(input);
        Console.WriteLine($"Is Valid: {isValid}");
    }
}
```

Code Listing 81

This is nearly identical to the previous program, except it uses **`Func<string, bool>`** instead of **`Predicate<string>`**. As mentioned previously, both **`Func<T, TResult>`** and **`Predicate<T>`** are generic delegates. The type specifications in angle brackets are plug-ins for types applied to parameters and return types. The following listing shows the **`Predicate<T>`** delegate as defined in the FCL.

```
public delegate bool Predicate<T>(T obj);
```

Code Listing 82

It refers to a method that returns a **bool**, but accepts a parameter of type **T**. So, **Predicate<string>** means that the parameter to the method referred to is a **string**. Similarly, here's the FCL definition of **Func<T, TResult>**.

```
public delegate TResult Func<T, TResult>(T arg);
```

Code Listing 83

This shows that **Func<T, TResult>** accepts a parameter of type **T** and returns a value of type **TResult**. In Code Listing 81, **Func<string, bool>** refers to a method with a parameter of type **string** that returns type **bool**.

For convenience, the FCL offers 18 overloads of the **Func** delegate, allowing between 0 and 16 input parameters and 1 return parameter type. This covers many scenarios, and you can reuse the provided delegates from the FCL to go a long way. You can create your own delegates only when you need to.

Expression-Bodied Members

While not necessarily lambdas, expression-bodied members give you some shorthand syntax for properties and methods. The following listing provides an example.

```
using System;

class Program
{
    public static string Today => DateTime.Now.ToShortDateString();

    public static void Log(string message) => Console.WriteLine(message);

    public static void Main()
    {
        Log($"{Today} is a good day.");

        Console.ReadKey();
    }
}
```

Code Listing 84

The **Program** class defines the **Today** property and **Log** method as expression-bodied members. **Main** shows how these members are used like normal methods and properties.

Summary

You learned about delegates, events, and lambdas. A delegate is a reference to a method. You can pass a delegate around in code or assign it to an event. The method a delegate refers to must have the same signature of the delegate. An event is a type member, defined with a delegate type. You can assign as many delegates to an event as you need and each one executes when the event fires. You can only fire an event from within the type the event is defined in. Instead of methods, you can use lambdas when you don't need to define a separate method. You can refer to a lambda with a delegate, pass a lambda as a parameter, or assign a lambda to an event. Expression-bodied members let you write properties and methods with a shorthand syntax.

Chapter 6 Working with Collections and Generics

You've seen arrays in previous chapters and they can be useful in scenarios where you need a fixed size, strongly typed list of objects. However, there are many times when you need to organize objects into different types of data structures like lists, queues, stacks, and dictionaries. These capabilities are available to C# developers via collection classes in the .NET Framework.

A core part of working with collections is the use of generics, which allow you to use parameterized code. This allows you to strongly type your collections. You can even write your own code that uses generics, allowing you to create strongly typed reusable libraries.

Note: The first version of .NET offered a collection library based on `Object` which isn't strongly typed. Since all .NET types are assignable to `Object`, this worked. However, you had to write a lot of code that used cast operators to convert from `Object` back to the type you added to the collection. Generics solves this problem, and using generic collections is standard practice in .NET today.

Using Collections

.NET collection classes let you work with data in many different ways. Instead of an array, you can use a **`List`**. If you need a first-in first-out set of items, you can use a **`Queue`**. If you need to work with items that have unique IDs, you can use a **`Dictionary`**. With generics, you can build your own collection to manage data any way that you need.

Tip: Check out the System.Collections.Generic namespace before writing your own collection; you might find that what you need is already written.

A common collection is a **`List`**, which is a nice replacement for an array. The following listing provides an example.

```
using System;
using System.Collections.Generic;

public class Company
{
    public string Name { get; set; }
}

public class Program
```

```
{
    public static void Main()
    {
        List<string> names = new List<string>();
        names.Add("Joe");
        names.Insert(0, "Car");
        names.Add("Jill");
        names[0] = "Building";
        names.RemoveAt(0);
        Console.WriteLine($"First name: {names[0]}");

        IList<Company> companies = new List<Company>
        {
            new Company { Name = "Syncfusion" },
            new Company { Name = "Microsoft" },
            new Company { Name = "Acme" }
        };

        foreach (Company cmp in companies)
            Console.WriteLine(cmp.Name);
        Console.ReadKey();
    }
}
```

Code Listing 85

The previous program demonstrates the versatility of collections. You have a **List** of **string**, which only holds objects of type **string**. This is a generic collection, meaning that its type parameters, inside **<** and **>**, specify the type of object the collection works with. Each item added is appended to the list and the list grows dynamically. The **Insert** operation adds a new **string** at the first position of the list and pushes down the first, **"Joe"**, into the second position at index 1. The second **Add** puts **"Jill"** at index 2. Notice how you can use indexer (array-like) syntax to access elements of the list. The **RemoveAt** deleted the string at the first index of the collection, moving **"Joe"** to 0 and **"Jill"** to 1.

The second **List** in **Main** shows how to use custom types. Since **List** derives from **IList**, you can assign the instance to that interface. This is convenient because it means you can create code that operates on an **IList**; whether the caller passes in a **List<T>** or any other collection type that derives from **IList**, your code will still work.

The example also uses collection initialization syntax, where you can instantiate a comma-separated list of the collection type that populates the **List**. The **foreach** statement iterates through the collection, printing each item.

The previous example uses a **foreach** loop, but you could also use the **ForEach** method of **List**, as shown in the following example.

```
        List<Company> companyList = companies as List<Company>;
        companyList.ForEach(cmp => Console.WriteLine(cmp.Name));
```

Code Listing 86

The first line uses the **as** operator to convert the **IList<Company>** to **List<Company>**. With an instance of **List<T>**, you can call the **ForEach** method, which takes a lambda parameter. This lambda executes for each of the items in the **List<T>** and the lambda parameter, **cmp**, contains the current item.

This should give you an idea of how **List** works. There are more methods available that you can learn about by reading the documentation for the **List** class.

Another useful collection is a **Dictionary**. It works like a hash table where you store and retrieve objects by index as shown in the following sample.

```
using System;
using System.Collections.Generic;

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}

public class Program
{
    public static void Main()
    {
        Dictionary<int, Customer> customers = new Dictionary<int, Customer>();
        Customer jane = new Customer { ID = 0, Name = "Jane" };
        Customer joe = new Customer { ID = 1, Name = "Joe" };
        customers.Add(jane.ID, jane);
        customers[joe.ID] = joe;

        foreach (int key in customers.Keys)
            Console.WriteLine(customers[key].Name);

        Dictionary<int, Customer> customers2 = new Dictionary<int, Customer>
        {
            [0] = new Customer { ID = 0, Name = "Chris" },
            [1] = new Customer { ID = 1, Name = "Alex" }
        };

        Console.ReadKey();
    }
}
```

Code Listing 87

A **Dictionary** in the previous example takes two type parameters for the key and value, respectively. The first example instantiates the dictionary to take an **int** key and **Customer** value. The **Customer** class has two properties, where the **ID** will be used as a key for the dictionary. Notice the two different ways you can add values to a dictionary, via the **Add** method or indexer assignment. The first parameter to **Add** is the index and the second is the value. When using the indexer, put the index in brackets and assign the value. Just as in other collections, there are many methods available and you should review the documentation of that collection.

The **foreach** loop shows how to iterate through **Dictionary** items. A **Dictionary** has a **Keys** property, which is a collection of keys, and a **Values** property, which is a collection of values (the **Customer** instances in the previous example). Notice how the loop uses the indexer **customers[key]** to access the value associated with each key.

The second **Dictionary** in the example shows how to use the dictionary initializer syntax. Just assign the value to the index that matches the key for that value.

Writing Generic Code

One of the primary applications of generics is to support collections. In the previous section, you saw how to use collections. You could also write your own collection class. If you wrote a generic linked list, you would need a **Node** class to hold an object and reference the next in the list, and a **LinkedList** collection class that performed list operations. The **Node** class in the following listing contains an object instance.

```
class Node<T>
{
    public T Item { get; set; }
    public Node<T> Next;

    public Node(T item)
    {
        Item = item;
    }
}
```

Code Listing 88

The **<T>** syntax makes the **Node<T>** class generic. Whenever code instantiates a **Node**, it specifies a type that replaces **T**. Anywhere you're using an object of that type, specify **T**. **Node<T>** doesn't have an access modifier because it's only used with the code inside this assembly and the default internal accessibility is appropriate. The following sample shows how to instantiate a **Node<T>**.

```
Node<string> name = new Node<string>("May");
```

Code Listing 89

Here, you see **Node<string>** as the type, meaning that all of the places you see **T** inside of the **Node** class are now **string**. You're protected from passing an **int**, **decimal**, or any other type to the constructor of this class because it will only hold a **string**. It is strongly typed.

Next, you need a collection that holds **Node<T>** instances as a linked list, as shown in the following listing.

```
Using System;
using System.Collections;
using System.Collections.Generic;

public class LinkedList<T> : IList<T>
{
    Node<T> head;
    Node<T> tail;

    public void Add(T item)
    {
        var node = new Node<T>(item);

        if (head == null)
            head = node;
        else
            tail.Next = node;

        tail = node;
    }

    // Other IList members…
}
```

Code Listing 90

The **LinkedList** class is generic and holds items of the type it's instantiated as. The **IList<T>** interface belongs to the FCL and facilitates creating collections. As you would expect with interfaces, developers who write code to the **IList** interface can use this collection too. The **LinkedList** class implements all the members of the **IList<T>** interface, as it must.

Add is a minimal implementation, but illustrates some concepts of working with generics. Even though the code instantiates a new **Node<T>**, the actual type will be the same as the type that **LinkedList** is defined as. The same concept applies to the interface where **IList<T>** becomes the same type as **LinkedList**. The following example instantiates a **LinkedList<T>**.

```
public class Program
{
    public static void Main()
    {
        var llist = new LinkedList<string>();
        llist.Add("Jamie");
        llist.Add("Ron");
        //...

        Node<string> name = new Node<string>("May");
    }
}
```

Code Listing 91

This shows that you instantiate and use your generic collection like any other collection. Just supply the type during instantiation and the collection will work with objects of that type.

Any place you see the **object** type being used is a potential candidate for creating a generic type. All types inherit the **object** type, which is why you'll see types in the FCL and elsewhere work with object type values.

You can also create generic methods. The following example shows a couple factory methods where one is type **object** and the other is generic.

```
using System;

public class CustomerReport
{
    public DateTime Date { get; set; }
}

public class OrdersReport
{
    public DateTime Date { get; set; }
}

public class ReportFactory
{
    public static object Create(Type reportType)
    {
        switch (reportType.ToString())
        {
            case "CustomerReport":
                var custRpt = new CustomerReport();
                custRpt.Date = DateTime.Now;
                return custRpt;
            default:
            case "OrdersReport":
                var ordsRpt = new OrdersReport();
                ordsRpt.Date = DateTime.Now;
                return ordsRpt;
        }
    }
}

public class Program
{
    public static void Main()
    {
        var rpt = (CustomerReport)ReportFactory.Create(typeof(CustomerReport));
        Console.ReadKey();
    }
}
```

Code Listing 92

What you should get out of the previous **ReportFactory** implementation is that there's a lot of duplication in the code and the use of cast and **typeof** operators in the **Main** method includes

more syntax than necessary. You can probably see where this code might become less maintainable with more complexity. The following example shows how to refactor the **Create** method into a generic method.

```
using System;

public abstract class Report { }

public class CustomerReport : Report
{
    public DateTime Date { get; set; }
}

public class OrdersReport : Report
{
    public DateTime Date { get; set; }
}



public class ReportFactory
{
    public static TReport Create<TReport>()
        where TReport : Report
    {
        switch (typeof(TReport).Name)
        {
            case "CustomerReport":
                var custRpt = new CustomerReport();
                custRpt.Date = DateTime.Now;
                return (TReport)(Report)custRpt;
            default:
            case "OrdersReport":
                var ordsRpt = new OrdersReport();
                ordsRpt.Date = DateTime.Now;
                return (TReport)(Report)ordsRpt;
        }
    }
}

public class Program
{
    public static void Main()
    {
        var rpt2 = ReportFactory.Create<CustomerReport>();

        Console.ReadKey();
    }
}
```

Code Listing 93

The **Create** method has a new type parameter, **TReport**. You've seen the use of just **T** in previous examples, but sometimes—as in **Dictionary<TKey** and **TValue>**—you have to

differentiate between multiple type parameters or make the code more self-documenting. The return type is now strongly typed too. The code is able to cast from the derived type to **Report**, and then to **TReport** to return the proper type. This is allowed because of the generic constraint, where **TReport : Report** says that **TReport** must derive from **Report**. The calling code is much simpler.

The **Create<TReport>** method is still longer than it has to be and contains too much duplication. We can solve that problem with generic constraints. A constraint does what the name implies: it limits how generic a type can be. You saw the base class constraint on **Report** in the previous code. The following table describes all available constraints.

Table 3: Generic Type Constraints

Constraint	Description
interface	Type must implement specified interfaces.
base class	Type must derive from specified base class.
class	Type must be a reference type.
struct	Type must be a value type.
new	Type must have a default (no parameter) constructor.

We need two constraints to simplify our code: **interface** and **new**. The following example shows how they can be used.

```
using System;

public interface IReport
{
    DateTime Date { get; set; }
}

public class CustomerReport : IReport
{
    public DateTime Date { get; set; }
}

public class OrdersReport : Report, IReport
{
    public DateTime Date { get; set; }
}

public class ReportFactory
{
    public static TReport Create<TReport>()
        where TReport : IReport, new()
    {
        return new TReport() { Date = DateTime.Now };
    }
```

```
}

public class Program
{
    public static void Main()
    {
        var rpt2 = ReportFactory.Create<CustomerReport>();

        Console.ReadKey();
    }
}
```

Code Listing 94

In this demo, there's a new interface, **IReport**, which **CustomerReport** and **OrdersReport** derive from. Since we know the classes we expect are **IReport**, we can make assumptions about the type and write code that operates on any **IReport**.

The **Create<TReport>** method has additional syntax following the method signature. To specify a constraint, follow the **where** keyword with the type being constrained, append a semicolon, and then add a comma-separated list of constraints from the previous table. This example uses an interface and **new()** constraint. The **new()** constraint means we can create a new instance of a type, **new TReport()**. Further, since the type is an **IReport**, we know it has a **Date** property and can populate its **Date** property. Gone are the duplication and excessive code, simplified by generic code in both implementation and use.

Tip: You can also create generic delegates. As usual, you should seek to reuse types already present in the FCL. A popular reusable delegate in the .NET Framework is EventHandler<TEventArgs>. In fact, you can replace all the references to ClickHandler in Chapter 5 with EventHandler<ClickEventArgs> and your code will still work.

Summary

You've seen how to use generics and that they let you write reusable code. The .NET collection classes are more versatile than arrays and allow you to manage your objects in ways that better help the design of your application.

Chapter 7 Querying Objects with LINQ

Language-Integrated Query (LINQ) allows you to query data with a SQL-like syntax. LINQ can be used with many different types of data and both Microsoft and third parties have built LINQ providers to access a wide range of data sources. This chapter narrows that list by showing you how to use LINQ to Objects. Once you know LINQ to Objects, understanding other LINQ providers is easy because of similar syntax.

Getting Started

Before you write any LINQ code, remember to add a **`using`** declaration to the **`System.Linq`** namespace at the top of your file. Each example in this chapter will use the following class, containing collections to work with:

```
using System.Collections.Generic;

public class Customer
{
    public int ID { get; set; }
    public string Name { get; set; }
}
public class Order
{
    public int CustomerID { get; set; }
    public string Description { get; set; }
}

public static class Company
{
    static Company()
    {
        Customers = new List<Customer>
        {
            new Customer { ID = 0, Name = "May" },
            new Customer { ID = 1, Name = "Gary" },
            new Customer { ID = 2, Name = "Jennifer" }
        };
        Orders = new List<Order>
        {
            new Order { CustomerID = 0, Description = "Shoes" },
            new Order { CustomerID = 0, Description = "Purse" },
            new Order { CustomerID = 2, Description = "Headphones" }
        };
    }
```

```
    public static List<Customer> Customers { get; set; }
    public static List<Order> Orders { get; set; }
}
```

Code Listing 95

These are collections of objects in memory. To make the collection easier to query, **Company** is a **static** class, with a **static** constructor that initializes **static** properties. If you abstract this concept, that data could have been read from a file, database, or REST service. Regardless of the data source or the LINQ provider, the basic LINQ syntax remains the same.

Querying Collections

To query data, you only need the **from** and **select** keywords. Remember to add a **using** clause for the **System.Linq** namespace. The syntax looks like SQL, as you can see in the following example.

```
using System;
using System.Linq;
using System.Collections.Generic;

public class Program
{
    public void Main()
    {
        IEnumerable<Customer> customers =
            from cust in Company.Customers
            select cust;

        foreach (Customer cust in customers)
            Console.WriteLine(cust.Name);
    }
}
```

Code Listing 96

LINQ to Objects queries result in a collection of type **IEnumerable<T>**. In this case, it's a collection of **Customer** objects. The **from** keyword specifies a range variable, **cust**, which holds each object from the collection. You specify the collection after the **in** keyword.

The **select** defines what to query. In this example, you're just returning the whole object. In fact, the collection you get is identical to what is in **Company.Customers**. This isn't particularly useful in LINQ to Objects, but is very useful if the data was read from an external data source, like a database where you just wanted to get a collection of objects into memory for further manipulation. The **select** allows you to reshape the data you get back into various projections. The following is a query that gets the customer name.

```
        IEnumerable<string> customers2 =
            from cust2 in Company.Customers
            select cust2.Name;
```

Code Listing 97

The **select** uses the **cust2** variable to access the **Name**, resulting in a collection of **string** (the **Name** property's type). Sometimes you need a whole different object, where that object might be defined as:

```
public class CustomerViewModel
{
    public string Name { get; set; }
}
```

Code Listing 98

And a new projection could be written as:

```
        IEnumerable<CustomerViewModel> customerVMs =
            from custVM in Company.Customers
            select new CustomerViewModel
            {
                Name = custVM.Name
            };
```

Code Listing 99

Here, **select** instantiates a new **CustomerViewModel**. Then it populates values, using object initialization syntax, to assign the **custVM.Name** to the new object's **Name** property. This results in a collection of type **CustomerViewModel**.

The previous example assumed you needed to work with a specifically typed collection. However, what if you don't care what type the collection is and what if you didn't want to create a new class just to do manipulation in a single algorithm? In that case, you could use an anonymous type, as shown in the following listing.

```
        var customers3 =
            from cust3 in Company.Customers
            select new
            {
                Name = cust3.Name
            };
        foreach (var cust3 in customers3)
            Console.WriteLine(cust3.Name);
```

Code Listing 100

Anonymous types don't have names you can use, even though C# might create an internal name for its own use. To work around this problem, use the **var** keyword as the type. Notice how the projection uses **new** without a type name: an anonymous type. You can define whatever properties you want for an anonymous type; just write them in. Notice also that you can use **var** in the **foreach** loop.

If you need to return a collection from a method, create a new (named) type and project into that. Anonymous types are designed for situations limited to the scope in which they are used. You'll see the **var** keyword used elsewhere in code, but the reason it was added to the language was to support this scenario. The following listing shows a common way to use **var**, other than the previous scenario.

```
var customer = new Customer();
```

Code Listing 101

The previous statement is shorter than specifying the object type of the variable, which is redundant in this case and is obviously **Customer**. However, the following example is less obvious.

```
var response = DoSomethingAndReturnResults();
```

Code Listing 102

The problem in the previous statement is that just reading the code doesn't tell you what type **var** is. You don't know whether it's a single object or a collection. In this case, the code might be more maintainable by specifying the type.

Note: A common misconception is that `var` is dangerous because it behaves like `object`, allowing you to set the variable to any type. This is not true. When you use `var`, the code is still strongly typed. Once you assign a value to a variable of type `var`, you can't assign any other type to that variable. In the previous examples, `customers` is an instance of type `Customer`. You can't write code later to assign an object of another instance type—for example, an `Order` type—to that variable.

Filtering Data

You can filter a collection with the **where** clause, as shown in the following example.

```
var customers4 =
    from cust4 in Company.Customers
    where cust4.Name.Length > 3 && !cust4.Name.StartsWith("G")
```

```
            select cust4;

        foreach (var cust4 in customers4)
            Console.WriteLine(cust4.Name);
```

Code Listing 103

In the previous listing, a customer's name must be longer than **3**, which filters the list down to **Gary** and **Jennifer**. The clause to the right of the **&&** operator filters that list even further to the name whose first character is not **"G"**.

In LINQ to Objects, you can create complex conditions in the **where** clause using logical operators, parentheses for grouping, and any other logic to filter results. You can even call another method that will evaluate the current object being evaluated. The result of the **where** clause must evaluate to a **bool**. Other LINQ providers might restrict the type of expressions in a **where** clause, so you'll have to review documentation for that particular provider to learn more.

Ordering Collections

In LINQ, the **orderby** clause lets you sort collection results. The following listing demonstrates this.

```
        var customers5 =
            from cust5 in Company.Customers
            orderby cust5.Name descending
            select cust5;

        foreach (var cust5 in customers5)
            Console.WriteLine(cust5.Name);
```

Code Listing 104

In this example, the **orderby** clause sorts the list by the customer name in **descending** order. The default order is ascending, which you'll get by either omitting **descending** or specifying **ascending** instead. The output is:

```
May
Jennifer
Gary
```

Joining Objects

Sometimes you'll have two different collections of objects or related tables in a database and you need to join them together. To do this, use the **join** clause.

```
var customerOrders =
    from cust in Company.Customers
    join ord in Company.Orders
        on cust.ID equals ord.CustomerID
    select new
    {
        ID = cust.ID,
        Customer = cust.Name,
        Item = ord.Description
    };

foreach (var custOrd in customerOrders)
    Console.WriteLine(
        $"Customer: {custOrd.Customer}, Item: {custOrd.Item}");
```

Code Listing 105

After the **from** clause, you can use one or more **join** clauses to access the types you need. The **on** keyword lets you specify the keys to match between tables. This example creates a projection on an anonymous type to create a report based on the joined information. This was a normal join, which omits any **Customers** where there isn't a matching **Order**. The following example lets you do the equivalent of a left join.

```
var customerOrders2 =
    from cust in Company.Customers
    join ord in Company.Orders.DefaultIfEmpty()
        on cust.ID equals ord.CustomerID
    select new
    {

        ID = cust.ID,
        Customer = cust.Name,
        Item = ord.Description
    };

foreach (var custOrd2 in customerOrders)
    Console.WriteLine(
        $"Customer: {custOrd2.Customer}, Item: {custOrd2.Item}");
```

Code Listing 106

The difference here is the call to **DefaultIfEmpty**, which includes the **Customer** with the **Name Gary**, even though there aren't any orders in the join that match his **ID**.

Using Standard Operators

You've seen basic LINQ syntax, but there's much more available in the form of standard query operators. There are literally dozens of standard query operators, and you can view all of them on MSDN at https://msdn.microsoft.com/en-us/library/vstudio/bb397896(v=vs.120).aspx.

The following code listings are a grab bag of examples, demonstrating how to use standard query operators that you might find useful.

So far, you've been working with **IEnumerable<T>**, where **T** is the projected type of the query. There are a set of standard query operators that will return different collection types, including **ToList**, **ToArray**, **ToDictionary**, and more. Here's an example that turns the results into a **List**.

```
var custList =
    (from cust in Company.Customers
     select cust)
    .ToList();
custList.ForEach(cust => Console.WriteLine(cust.Name));
```

Code Listing 107

The previous code enclosed the query in parentheses and then called the **ToList** operator. The **ForEach** method on **List<T>** lets you pass a lambda.

LINQ queries use deferred execution. This means that the query doesn't execute until you execute a **foreach** loop or call one of the standard query operators, like **ToList**, that requests the data.

You've seen how the C# **select**, **where**, **orderby**, and **join** keywords help build queries. Each of these queries have a standard query operator equivalent. These standard query operators use a fluent syntax and give you a different way to perform the same query as their matching language syntax. Some people prefer the fluent style and others prefer the language syntax, but the method you choose is really a personal preference. The following is an example of the **Where** and **Select** operators, which mirror the **where** and **select** language syntax clauses.

```
var customers6 =
    Company.Customers
           .Where(cust => cust.Name.StartsWith("J"));
foreach (var cust6 in customers6)
    Console.WriteLine(cust6.Name);

var customers7 =
    Company.Customers.Select(cust => cust.Name);
foreach (var cust7 in customers7)
    Console.WriteLine(cust7);
```

Code Listing 108

The **Where** lambda must evaluate to a **bool** and the **Select** lambda lets you specify the projection.

You can perform set operations like **Union**, **Except**, and **Intersect**. The following listing is an example of **Union**.

```
var additionalCustomers =
    new List<Customer>
    {
        new Customer { ID = 1, Name = "Gary" }
    };
var customerUnion =
    Company.Customers
           .Union(additionalCustomers)
           .ToArray();
foreach (var cust in customerUnion)
    Console.WriteLine(cust.Name);
```

Code Listing 109

Just pass a compatible collection and **Union** will produce a combined collection of all objects. I used the **ToArray** operator in this example too, which results in an array of the collection type, **Customer**.

There is a useful set of operators for selecting **First**, **FirstOrDefault**, **Single**, **SingleOrDefault**, **Last**, and **LastOrDefault**. The following example demonstrates **First**.

```
Console.WriteLine(Company.Customers.First().Name);
```

Code Listing 110

The only thing about using **First** this way is the possibility of an **InvalidOperationException** with the message "Sequence contains no elements." This sequence contains elements, but this isn't guaranteed. You would be safer using the operator with the **OrDefault** suffix, as in the following listing.

```
var empty =
    Company.Customers
           .Where(cust => cust.ID == 999)
           .SingleOrDefault();

if (empty == null)
    Console.WriteLine("No values returned.");
```

Code Listing 111

The previous example writes **"No values returned."** Because there isn't a customer with **ID == 999**, the **SingleOrDefault** returns **null**, which is the default value of a reference type object.

These were only a handful of available operators, but hopefully you have a sense for the wealth of support in language syntax as well as the standard query operators that comprise LINQ.

Summary

LINQ allows you to use SQL-like syntax to query data. The LINQ provider used in this chapter is LINQ to Objects, which lets you query objects in memory, but there are many other LINQ providers for other data sources. Use a **`from`** to specify the collection being queried and a **`select`** to shape the results. The **`where`** clause lets you filter results and takes a **`bool`** expression to evaluate if a given object should be included. The **`orderby`** clause lets you sort results. The **`join`** clause lets you combine two collections. Standard query operators extend LINQ and make it even more powerful than the language keywords.

Chapter 8 Making Your Code Asynchronous

In version 5, C# introduced the capability to write and call code asynchronously, commonly referred to as async. To understand async, it's useful to consider the normal behavior of code, which is synchronous. In synchronous code you call a method, wait for it to complete, and move on to the rest of the code. The primary point of this behavior is that the thread calling the synchronous method is also executing the synchronous code in that method. If that synchronous method runs for a long time, your UI will become unresponsive and your users might not know whether the program crashed or if they should just wait.

Asynchronous code improves this situation by allowing the long-running operation to continue on a separate thread, and free your calling thread to resume its responsibilities. When the calling thread is the UI thread, the application becomes responsive again and you can show a status or busy indicators, or let the user operate another part of the program while the asynchronous process runs. When the asynchronous process returns, you can interact with users in some way, if that makes sense for your application. In the past, writing this asynchronous code has been a challenge. Though the task of writing asynchronous code has improved with new patterns and libraries, the C# async features makes asynchronous programming much easier.

There are a couple different perspectives of async that determine how you code: library consumer or library creator. From a consumer perspective, you make assumptions about async code based on an implied contract. However, from the library creator perspective, you have additional responsibilities to ensure your code provides the async contract users expect.

Consuming Async Code

C# has two keywords that support async: **`async`** and **`await`**. Decorating a method with the **`async`** modifier says that the method can contain async code. You use the **`await`** keyword on a **`Task`** to start an async operation.

```
using System.IO;
using System.Threading.Tasks;

public class Program
{
    public static void Main()
    {
        Program.CreateFileAsync("test.txt").Wait();
    }



    public static async Task CreateFileAsync(string filename)
    {
```

```
        using (StreamWriter writer = File.CreateText(filename))
            await writer.WriteAsync("This is a test.");
    }
}
```

Code Listing 112

In the previous program, the **CreateFileAsync** method is asynchronous. You can tell by the **async** modifier on the method. You need to add **using** clauses for the **System.IO** and **System.Threading.Tasks** namespaces for writing to a file and async **Task** support, respectively. The **File** class is part of the FCL and its **CreateText** method returns a **StreamWriter** that you use to write to the file.

Note: Appending a method name with `Async` is not required, but it is a common convention.

The proper way to call an async method is to **await** its **Task** or **Task<T>**. The **WriteAsync** method returns **Task**, which means you can **await** it.

The **using** statement closes the file when its enclosing block completes. In this case, the block is only a single line, so no curly braces are required.

Part of the async contract is an expectation that some code in the library you're using will run the operation on another thread, releasing your thread for other operations; that's what **WriteAsync** does too. So, the thread returns to the code calling this async method. But the caller in this program is the **Main** method, which is calling **Wait()** on the **Task** returned from **CreateFileAsync**. This keeps the program from ending before the thread that's running the async operation completes.

Warning: The previous example is a console application, which doesn't have the underlying infrastructure (referred to as a synchronization context) to manage proper thread handling. Therefore, it was necessary to `Wait()` on the task returned from `CreateFileAsync`. In a normal UI application, you will have a synchronization context, meaning you won't have to worry about the program ending, and won't need to call `Wait()` on an async method. The preferred method of waiting on an async method is via `async` and `await` as shown in the `CreateFileAsync` method. In fact, you should never call `Wait` on an async method. That's because when the second thread returns from doing work on the `async` call, it will attempt to marshal the call back onto the calling thread. If that calling thread is in a synchronous `Wait()`, the thread will be blocked, preventing the second thread from performing that marshaling operation. Then you'll have a deadlock. To prevent deadlock, never call `Wait()`, use `async` and `await` instead.

The **async** modifier is required on the method if you use **await**. If a method has the **async** modifier, but no **await**s, C# will give you a compiler warning and let you know that the method will run synchronously.

Async Return Types

With **async**, you can **await** any awaitable type. The FCL has **Task** and **Task<T>**, which are awaitable and are what you should use in most situations. Returning **Task** means that the method does not return a value, which is what you saw with the previous **CreateFileAsync** method.

Tip: Stephen Toub's blog post "await anything;" explains how to create a custom awaitable type and is a good reference if you see it as a way to improve your code. You can read it at http://blogs.msdn.com/b/pfxteam/archive/2011/01/13/10115642.aspx.

Use **Task<T>** when your method returns a value. The following listing shows an example.

```
public async Task<string> ReturnGreeting()
{
    await Task.Delay(1000);
    return "Hello";
}
```

Code Listing 113

Task.Delay is a way to sleep the thread by a number of milliseconds, but I'll be using it in more examples to simplify code and as a placeholder for where you would normally add async code.

The previous example shows a return type of **Task<string>**. The method only returns the string **"Hello"** instead of an instance of **Task<string>** because the C# compiler takes care of that for you.

An async method can return **void** rather than an awaitable type too. This is done in the following listing.

```
public async void SayGreeting()
{
    await Task.Delay(1000);
    Console.WriteLine("Hello");
}
```

Code Listing 114

This method executes asynchronously, but async **void** methods have important caveats you must be aware of: they aren't awaitable, and they don't protect against exceptions, but they are necessary for scenarios like event handling where the method must be **void**.

Since you can only await awaitable types like **Task** and **Task<T>**, there's no way to await an async **void** method. The implication of this is that when a library's code starts another thread, it

allows the calling thread to return. Calling an async **void** method means you can't wait until that method completes and you won't ever know when or if the method completes. As with anything, there are no absolutes and one could argue that it would be possible to write some cross-thread communication mechanism, but I'm referring to the general out of the box behavior, which will lead to some important implications. Because of this behavior, there are pros and cons on when you should use an async **void** method.

The largest problem with async **void** methods is that you can't throw an exception back to the calling code. With **Task** and **Task<T>** returning methods, you can **await** and wrap the async method call in a **try-catch**, but you can't do that with async **void** methods. If an async **void** method throws an unhandled exception, the application will crash.

With such problems, it would be easy to assume that async **void** should not be used at all. However, the C# language designers added async **void** for one specific reason: to support event handling. Event handlers in the .NET Framework follow a pattern where their delegates return **void**. Therefore, you can't use an awaitable type, like **Task** or **Task<T>**, and must assign async **void** methods as event handlers.

In UI applications, a UI control might fire an event, async **void** methods assigned to the event execute, the async code starts a new thread and releases the UI thread, and the UI thread returns and processes messages to keep the UI responsive. So, using async **void** as event handlers is appropriate.

Developing Async Libraries

Writing an async library is mostly normal coding, but the key thing to keep in mind is what is happening to the thread. First, all code executes by default on the calling thread. Second, you need to marshal execution onto a new thread and release the calling thread to the caller.

Understanding What Thread the Code is Running On

The following code doesn't necessarily make any logical sense, but represents the potential structure of some library code that you might write. In particular, it demonstrates what happens with threads before and after the first **await** in your **async** method. In the following code, **UserInfo** is just a type to hold and return data. **UserService** and **AddressService** have async methods that the **GetUserInfoAsync** method calls.

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

public class UserInfo
{
    public string Info { get; set; }
    public string Address { get; set; }
}
```

```
class UserService
{
    internal static async Task<string> GetUserAsync(string user)
    {
        // Do some long running synchronous processing.
        return await Task.FromResult(user);
    }
}

class AddressService
{
    internal static async Task<string> GetAddressAsync(string user)
    {
        return await Task.FromResult(user);
    }
}

public class UserSearch
{
    public async Task<UserInfo> GetUserInfoAsync(string term, List<string> names)
    {
        var userName =
            (from name in names
             where name.StartsWith(term)
             select name)
            .FirstOrDefault();

        var user = new UserInfo();
        user.Info = await UserService.GetUserAsync(userName);
        user.Address = await AddressService.GetAddressAsync(userName);

        return user;
    }
}
```

Code Listing 115

Remember, you're writing reusable library code, so it could be called from many different technologies, such as WPF, Windows Store apps, Windows Phone, and more. What's common about these type of applications is that a user interacts with the UI controls and those UI controls fire events. This means an async **`void`** event handler awaits your **`GetUserInfoAsync`** method.

When the event handler code calls your code, it's running on the UI thread. Your code will continue running on the UI thread until some other code explicitly marshals the call to another thread and releases the UI thread.

Note: More accurately, the thread calling your code might not necessarily be the UI thread if there was another async method that called your code and already released the UI thread. However, defensive coding is a safe approach because

there exists the possibility that your code will be called on the UI thread by some developer in the future.

Notice the LINQ query in **GetUserInfoAsync** before reaching the first **await**. That is synchronous code that runs on the calling thread, which could also be the UI thread. The issue here is that the UI thread is tied up doing work in your async method, rather than returning to the UI. Imagine a UI with a progress indicator that locks up because your async method is holding onto the UI thread and doing a lot of processing before the first async call.

The code is still on the UI thread when it calls **UserService.GetUserAsync**. I added a comment to **GetUserAsync** to represent more long-running synchronous processing that is also running on the UI thread. Finally, awaiting **Task.FromResult** releases the UI thread and the rest of the code runs asynchronously. That's because **Task.FromResult** implements the async contract properly. Before showing you how to fix this problem, let's look at the rest of the code so you can understand how it runs.

When the code returns from **Task.FromResult**, the UI thread has been released and the code is running on the new async thread. When returning from **GetUserAsync** to its caller, **GetUserInfoAsync**, the call automatically marshals back to the calling thread, which could be the UI thread. Again, this program eats up CPU cycles on the UI thread, making the application less responsive. Fortunately, there's a way to fix this problem.

Fulfilling the Async Contract

The previous section explained how the code runs on the calling thread by default, which could be the UI thread. Whenever you call an async method in the FCL, that code will release the calling thread and continue on a new thread, which is proper behavior of the async contract that developers expect. You should do the same in your code.

To do this, use the **Task.ConfigureAwait** method, passing **false** as the parameter. The following is an example that fixes the problem in **GetUserInfoAsync**.

```
public async Task<UserInfo> GetUserInfoAsync(string term, List<string> names)
{
    var userName =
        (from name in names
         where name.StartsWith(term)
         select name)
        .FirstOrDefault();

    var user = new UserInfo();
    user.Info = await UserService.GetUserAsync(userName).ConfigureAwait(false);
    user.Address = await AddressService.GetAddressAsync(userName);

    return user;
}
```

Code Listing 116

The **GetUserInfoAsync** method appends **ConfigureAwait(false)** to the call to **GetUserAsync**. **GetUserAsync** returns a **Task<string>** and **ConfigureAwait(false)** operates on that return value, releasing the calling thread and running the rest of the method on a new async thread. That's it; that's all you have to do.

You still have the issue of synchronous processing before the first call to **ConfigureAwait**. Sometimes, you can't do anything about it because it's necessary to execute that code before the first **await**. However, if it's possible to rearrange the code to do any processing after the first **await**, you should do so.

A Few More Notes on Async

I made this point previously, but I feel it bears repeating. Especially for library code, you should prefer async methods that return **Task** or **Task<T>**. In the UI you don't have a choice if you're writing an event handler. If you're using a Model View ViewModel (MVVM) architecture, you'll also need **void Command** handlers. You shouldn't have these issues in reusable library code and in this scenario, async **void** methods are dangerous.

Async void methods that throw exceptions will crash your application.

Much of the discussion in this chapter is around how async improves the user experience by releasing the UI thread. In addition to that, async also improves application performance by not blocking threads. These scenarios usually involve some type of out-of-process operation such as network communication, file I/O, or REST service calls. These operations can use Windows operating system services such as I/O completion ports to free threads while the long-running out-of-process operation executes; they can then reallocate those threads when the operation completes and needs to return to your code. In addition to performance increases through efficient thread management, you can also improve the scalability of a server application by using async to avoid blocking threads more than you have to.

For all the seeming complexity that this chapter introduces, attempting to perform many of the operations associated with managing threads for application responsiveness, performance, and scalability is made much easier through the use of async.

Summary

Async is a useful capability that allows your application to be responsive and perform well. The user experience of async is a method with an **async** modifier and the ability to await an async method's **Task**. In addition to the user experience, there are additional considerations for writing async libraries. You should be aware of the threading behavior and how an **async** method runs on the caller's thread by default. Remember that you should minimize synchronous code before the first **await** and that you should call **ConfigureAwait(false)** at the earliest opportunity, releasing the UI thread and running the remaining algorithm on the new async thread.

Chapter 9 Moving Forward and More Things to Know

To keep subject matter succinct, I've passed up features that could evolve into deeper discussions. This chapter is about some of those features, if only to highlight that they are part of the C# language and that you are likely to encounter them regularly.

Decorating Code with Attributes

An attribute is a feature of C# that lets you decorate code with meta-information for various tools. I use the term "tool" loosely, but it could be the C# compiler, a testing framework, or a UI technology. Essentially, these tools read the attributes to make some decision on how to work with your code. I'll show you a few examples so you can be familiar with attribute syntax when you encounter it in code.

The **Obsolete** attribute lets you indicate that some code has been deprecated. It's a C# compiler attribute and the compiler will emit a message regarding its use. The following code shows an example.

```
using System;

public class ShoppingCart
{
    [Obsolete("Method planned for deprecation on date - use … instead.")]
    public void Add(string item) { }

    [Obsolete("Method is obsolete and can no longer be used", error: true)]
    public decimal CalculateTax(decimal[] prices) { return 0; }
}
```

Code Listing 117

When the C# compiler sees the **Obsolete** attribute decorating **Add**, it will show a warning with the message argument matching the parameter to the **Obsolete** attribute. In the second example, the compiler shows the message as an error and you won't be able to compile because the second parameter, **true**, indicates that the compiler should treat usage of that method as an error.

The next example uses attributes for a unit test with MSTest, Microsoft's unit testing software.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

```
namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Code Listing 118

As with most unit testing frameworks, MSTest has a test runner that loads the unit test code, looks for classes decorated with the **`TestClass`** attribute, and executes methods with the **`TestMethod`** attribute. It does this with another capability of C# called reflection.

Using Reflection

Reflection gives you the ability to examine compiled .NET code. With reflection, you can build useful tools like MSTest, dynamically instantiate types and execute their code, and more. The following code uses reflection to dynamically instantiate a type and execute one of its methods.

```
using System.Linq;
using System;
using System.Reflection;

public class FinancialCalculator
{
    public decimal Sum(decimal[] numbers)
    {
        return numbers.Sum();
    }
}

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        Type calcType = typeof(FinancialCalculator);
        MethodInfo sumMethod = calcType.GetMethod("Sum");
        FinancialCalculator calc =
            (FinancialCalculator)Activator.CreateInstance(calcType);
        decimal sum = (decimal)sumMethod.Invoke(calc, new object[] { prices });

        Console.WriteLine($"Sum: {sum}\nPress any key to continue.");
```

```
        Console.ReadKey();
    }
}
```

Code Listing 119

FinancialCalculator.Sum uses the LINQ **Sum** method, so add a **using** clause for **System.Linq**. Add a **using** declaration for **System.Reflection** to support reflection too.

With reflection, a **Type** instance gives you access to all of the information about a type. The **Main** method calls **GetMethod** to obtain a **MethodInfo** reference to the **Add** method, but there are many more methods that let you look at various parts of a type. As an example of a subset of capabilities available, you can call **GetMethods**, **GetProperties**, or **GetFields** to get an array of **MethodInfo**, **PropertyInfo**, or **FieldInfo** respectively. There are many more methods in the **Type** class you can use, and it's a fun exercise to write code to practice with this.

Activator.CreateInstance creates a new instance of the **Type** it's passed. Calling **Invoke** lets you run a method and get the results. Much of the previous reflection code is hard-coded for simplicity, but it's very useful for when you need to write code that examines the capabilities of another piece of code and optionally work with the member of a type.

Working with Code Dynamically

C# also has a type called dynamic. Its purpose is to allow you to interoperate with dynamic languages, like IronPython and IronRuby, and makes reflection easier. Microsoft also has a technology called Silverlight. Dynamic could make working with the HTML DOM easier, but Silverlight has been largely replaced by HTML 5 as a dynamic web application technology.

The **dynamic** type lets you assign any value to a **dynamic** variable and use any typed members on that variable. Rather than the C# compiler emitting errors, any errors are handled by the CLR at runtime. You might see where this has a lot of power through coding flexibility, yet a drawback of dynamic typing is that it offers no indication of type-related errors until runtime. Using the **FinancialCalculator** class from the previous example, the following is an example of some dynamic code.

```
using System;
using System.Reflection;

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        Type calcType = typeof(FinancialCalculator);
        MethodInfo sumMethod = calcType.GetMethod("Sum");
        dynamic calc = Activator.CreateInstance(calcType);
```

```
        dynamic sum = calc.Sum(prices);

        Console.WriteLine($"Sum: {sum}\nPress any key to continue.");
        Console.ReadKey();
    }
}
```

Code Listing 120

You might notice that this code is simpler than the full reflection implementation. You don't have an interface and have no guarantee that the **`calc`** instance has a member named **`Sum`**, but since the alternative is to use reflection, you're still in the same situation of runtime evaluation. Therefore, this might be a reasonable approach for this particular scenario.

Pulling together what you learned about generics, it might be useful to improve the algorithm even further, as shown in the following listing.

```
using System;

public class Program
{
    public static void Main()
    {
        decimal[] prices = { 1m, 2m, 3m };

        decimal sum = GetSum<FinancialCalculator, decimal>(prices);

        Console.WriteLine("Sum: {0}\nPress any key to continue.", sum);
        Console.ReadKey();
    }

    public static TValue GetSum<TCalc, TValue>(TValue[] prices)
        where TCalc : new()
    {
        dynamic calc = new TCalc();
        TValue sum = calc.Sum(prices);
        return sum;
    }
}
```

Code Listing 121

The previous example totally eliminates the need for reflection, reduces code, makes the algorithm strongly typed where it needed to be, and makes it dynamic where it helps. It would have been possible to use an interface constraint to make **`GetSum`** more strongly typed, but I used this as an exercise to help you think about where dynamic might be useful.

Summary

Attributes are C# features that tell a tool something about your code. Reflection helps you write meta-code that can evaluate and execute other code. There is a dynamic type that lets you make assumptions about the code you're writing, interfacing with dynamic languages, and making it easy to perform reflection.

This completes *C# Succinctly*. I hope it has been useful for you. I wish you the best in your further studies.

Artificial Neural Networks: A Tutorial

Anil K. Jain
Michigan State University

Jianchang Mao
K.M. Mohiuddin
IBM Almaden Research Center

These massively parallel systems with large numbers of interconnected simple processors may solve a variety of challenging computational problems. This tutorial provides the background and the basics.

Numerous advances have been made in developing intelligent systems, some inspired by biological neural networks. Researchers from many scientific disciplines are designing artificial neural networks (ANNs) to solve a variety of problems in pattern recognition, prediction, optimization, associative memory, and control (see the "Challenging problems" sidebar).

Conventional approaches have been proposed for solving these problems. Although successful applications can be found in certain well-constrained environments, none is flexible enough to perform well outside its domain. ANNs provide exciting alternatives, and many applications could benefit from using them.[1-3]

This article is for those readers with little or no knowledge of ANNs to help them understand the other articles in this issue of *Computer*. We discuss the motivations behind the development of ANNs, describe the basic biological neuron and the artificial computational model, outline network architectures and learning processes, and present some of the most commonly used ANN models. We conclude with character recognition, a successful ANN application.

WHY ARTIFICIAL NEURAL NETWORKS?

The long course of evolution has given the human brain many desirable characteristics not present in von Neumann or modern parallel computers. These include

- massive parallelism,
- distributed representation and computation,
- learning ability,
- generalization ability,
- adaptivity,
- inherent contextual information processing,
- fault tolerance, and
- low energy consumption.

It is hoped that devices based on biological neural networks will possess some of these desirable characteristics.

Modern digital computers outperform humans in the domain of numeric computation and related symbol manipulation. However, humans can effortlessly solve complex perceptual problems (like recognizing a man in a crowd from a mere glimpse of his face) at such a high speed and extent as to dwarf the world's fastest computer. Why is there such a remarkable difference in their performance? The biological neural system architecture is completely different from the von Neumann architecture (see Table 1). This difference significantly affects the type of functions each computational model can best perform.

Numerous efforts to develop "intelligent" programs based on von Neumann's centralized architecture have not resulted in general-purpose intelligent programs. Inspired by biological neural networks, ANNs are massively parallel computing systems consisting of an exremely large number of simple processors with many interconnections. ANN models attempt to use some "organizational" principles believed to be used in the human

0018-9162/96/$5.00 © 1996 IEEE

Challenging problems

Let us consider the following problems of interest to computer scientists and engineers.

Pattern classification

The task of pattern classification is to assign an input pattern (like a speech waveform or handwritten symbol) represented by a feature vector to one of many prespecified classes (see Figure A1). Well-known applications include character recognition, speech recognition, EEG waveform classification, blood cell classification, and printed circuit board inspection.

Clustering/categorization

In clustering, also known as unsupervised pattern classification, there are no training data with known class labels. A clustering algorithm explores the similarity between the patterns and places similar patterns in a cluster (see Figure A2). Well-known clustering applications include data mining, data compression, and exploratory data analysis.

Function approximation

Suppose a set of n labeled training patterns (input-output pairs), $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, have been generated from an unknown function $\mu(\mathbf{x})$ (subject to noise). The task of function approximation is to find an estimate, say $\hat{\mu}$, of the unknown function μ (Figure A3). Various engineering and scientific modeling problems require function approximation.

Prediction/forecasting

Given a set of n samples $\{y(t_1), y(t_2), \ldots, y(t_n)\}$ in a time sequence, $t_1, t_2, \ldots, t_n$, the task is to predict the sample $y(t_{n+1})$ at some future time t_{n+1}. Prediction/forecasting has a significant impact on decision-making in business, science, and engineering. Stock market prediction and weather forecasting are typical applications of prediction/forecasting techniques (see Figure A4).

Optimization

A wide variety of problems in mathematics, statistics, engineering, science, medicine, and economics can be posed as optimization problems. The goal of an optimization algorithm is to find a solution satisfying a set of constraints such that an objective function is maximized or minimized. The Traveling Salesman Problem (TSP), an *NP-complete* problem, is a classic example (see Figure A5).

Content-addressable memory

In the von Neumann model of computation, an entry in memory is accessed only through its address, which is independent of the content in the memory. Moreover, if a small error is made in calculating the address, a completely different item can be retrieved. Associative memory or content-addressable memory, as the name implies, can be accessed by their content. The content in the memory can be recalled even by a partial input or distorted content (see Figure A6). Associative memory is extremely desirable in building multimedia information databases.

Control

Consider a dynamic system defined by a tuple $\{u(t), y(t)\}$, where $u(t)$ is the control input and $y(t)$ is the resulting output of the system at time t. In model-reference adaptive control, the goal is to generate a control input $u(t)$ such that the system follows a desired trajectory determined by the reference model. An example is engine idle-speed control (Figure A7).

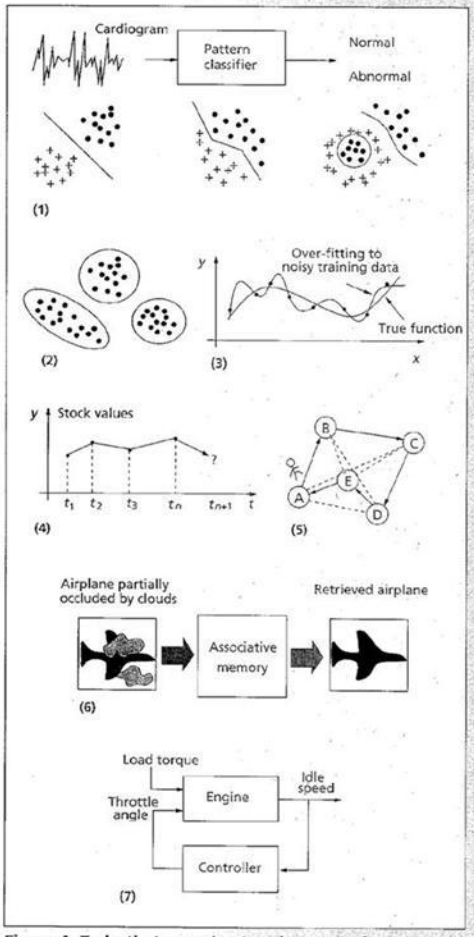


Figure A. Tasks that neural networks can perform: (1) pattern classification; (2) clustering/categorization; (3) function approximation; (4) prediction/forecasting; (5) optimization (a TSP problem example); (6) retrieval by content; and (7) control (engine idle speed). (Adapted from *DARPA Neural Network Study*[1])

brain. Modeling a biological nervous system using ANNs can also increase our understanding of biological functions. State-of-the-art computer hardware technology (such as *VLSI* and optical) has made this modeling feasible.

A thorough study of ANNs requires knowledge of neurophysiology, cognitive science/psychology, physics (statistical mechanics), control theory, computer science, artificial intelligence, statistics/mathematics, pattern recognition, computer vision, parallel processing, and hardware (digital/analog/VLSI/optical). New developments in these disciplines continuously nourish the field. On the other hand, ANNs also provide an impetus to these disciplines in the form of new tools and representations. This symbiosis is necessary for the vitality of neural network research. Communications among these disciplines ought to be encouraged.

Brief historical review

ANN research has experienced three periods of extensive activity. The first peak in the 1940s was due to McCulloch and Pitts' pioneering work.[4] The second occurred in the 1960s with Rosenblatt's perceptron convergence theorem[5] and Minsky and Papert's work showing the limitations of a simple perceptron.[6] Minsky and Papert's results dampened the enthusiasm of most researchers, especially those in the computer science community. The resulting lull in neural network research lasted almost 20 years. Since the early 1980s, ANNs have received considerable renewed interest. The major developments behind this resurgence include Hopfield's energy approach[7] in 1982 and the back-propagation learning algorithm for multilayer perceptrons (multilayer feedforward networks) first proposed by Werbos,[8] reinvented several times, and then popularized by Rumelhart et al.[9] in 1986. Anderson and Rosenfeld[10] provide a detailed historical account of ANN developments.

Biological neural networks

A *neuron* (or nerve cell) is a special biological cell that processes information (see Figure 1). It is composed of a cell body, or *soma*, and two types of out-reaching tree-like branches: the *axon* and the *dendrites*. The cell body has a nucleus that contains information about hereditary traits and a plasma that holds the molecular equipment for producing material needed by the neuron. A neuron receives signals (impulses) from other neurons through its dendrites (receivers) and transmits signals generated by its cell body along the axon (transmitter), which eventually branches into strands and substrands. At the terminals of these strands are the *synapses*. A synapse is an elementary structure and functional unit between two neurons (an axon strand of one neuron and a dendrite of another). When the impulse reaches the synapse's terminal, certain chemicals called neurotransmitters are released. The neurotransmitters diffuse across the synaptic gap, to enhance or inhibit, depending on the type of the synapse, the receptor neuron's own tendency to emit electrical impulses. The synapse's effectiveness can be adjusted by the signals passing through it so that the synapses can *learn* from the activities in which they participate. This dependence on history acts as a memory, which is possibly responsible for human memory.

The cerebral cortex in humans is a large flat sheet of neurons about 2 to 3 millimeters thick with a surface area of about 2,200 cm^2, about twice the area of a standard computer keyboard. The cerebral cortex contains about 10^{11} neurons, which is approximately the number of stars in the Milky Way.[11] Neurons are massively connected, much more complex and dense than telephone networks. Each neuron is connected to 10^2 to 10^4 other neurons. In total, the human brain contains approximately 10^{14} to 10^{15} interconnections.

Neurons communicate through a very short train of pulses, typically milliseconds in duration. The *message* is modulated on the pulse-transmission frequency. This frequency can vary from a few to several hundred hertz, which is a million times slower than the fastest switching speed in electronic circuits. However, complex perceptual decisions such as face recognition are typically made by humans within a few hundred milliseconds. These decisions are made by a network of neurons whose operational speed is only a few milliseconds. This implies that the computations cannot take more than about 100 serial stages. In other

Table 1. Von Neumann computer versus biological neural system.

	Von Neumann computer	Biological neural system
Processor	Complex High speed One or a few	Simple Low speed A large number
Memory	Separate from a processor Localized Noncontent addressable	Integrated into processor Distributed Content addressable
Computing	Centralized Sequential Stored programs	Distributed Parallel Self-learning
Reliability	Very vulnerable	Robust
Expertise	Numerical and symbolic manipulations	Perceptual problems
Operating environment	Well-defined, well-constrained	Poorly defined, unconstrained

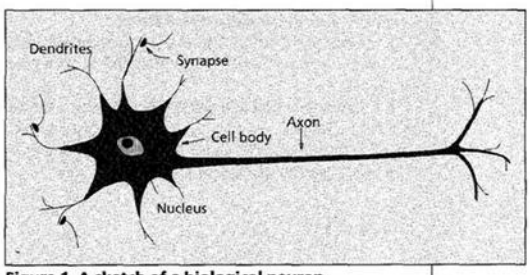


Figure 1. A sketch of a biological neuron.

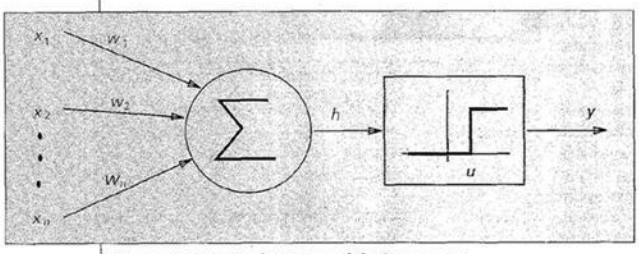


Figure 2. McCulloch-Pitts model of a neuron.

words, the brain runs parallel programs that are about 100 steps long for such perceptual tasks. This is known as the *hundred step rule.*[12] The same timing considerations show that the amount of information sent from one neuron to another must be very small (a few bits). This implies that critical information is not transmitted directly, but captured and distributed in the interconnections—hence the name, *connectionist* model, used to describe ANNs.

Interested readers can find more introductory and easily comprehensible material on biological neurons and neural networks in Brunak and Lautrup.[11]

ANN OVERVIEW

Computational models of neurons

McCulloch and Pitts[4] proposed a binary threshold unit as a computational model for an artificial neuron (see Figure 2).

This mathematical neuron computes a weighted sum of its n input signals, $x_j, j = 1, 2, \ldots, n$, and generates an output of 1 if this sum is above a certain threshold u. Otherwise, an output of 0 results. Mathematically,

$$y = \theta\left(\sum_{j=1}^{n} w_j x_j - u\right),$$

where $\theta(\cdot)$ is a unit step function at 0, and w_j is the synapse weight associated with the jth input. For simplicity of notation, we often consider the threshold u as another weight $w_0 = -u$ attached to the neuron with a constant input $x_0 = 1$. Positive weights correspond to *excitatory* synapses, while negative weights model *inhibitory* ones. McCulloch and Pitts proved that, in principle, suitably chosen weights let a synchronous arrangement of such neurons perform universal computations. There is a crude analogy here to a biological neuron: wires and interconnections model axons and dendrites, connection weights represent synapses, and the threshold function approximates the activity in a soma. The McCulloch and Pitts model, however, contains a number of simplifying assumptions that do not reflect the true behavior of biological neurons.

The McCulloch-Pitts neuron has been generalized in many ways. An obvious one is to use activation functions other than the threshold function, such as piecewise linear, sigmoid, or Gaussian, as shown in Figure 3. The sigmoid function is by far the most frequently used in ANNs. It is a strictly increasing function that exhibits smoothness and has the desired asymptotic properties. The standard sigmoid function is the *logistic* function, defined by

$$g(x) = 1/(1 + \exp\{-\beta x\}),$$

where β is the slope parameter.

Network architectures

ANNs can be viewed as weighted directed graphs in which artificial neurons are nodes and directed edges (with weights) are connections between neuron outputs and neuron inputs.

Based on the connection pattern (architecture), ANNs can be grouped into two categories (see Figure 4):

- *feed-forward* networks, in which graphs have no loops, and
- *recurrent* (or *feedback*) networks, in which loops occur because of feedback connections.

In the most common family of feed-forward networks, called multilayer perceptron, neurons are organized into layers that have unidirectional connections between them. Figure 4 also shows typical networks for each category.

Different connectivities yield different network behaviors. Generally speaking, feed-forward networks are *static*, that is, they produce only one set of output values rather than a sequence of values from a given input. Feed-forward networks are memory-less in the sense that their response to an input is independent of the previous network state. Recurrent, or feedback, networks, on the other hand, are dynamic systems. When a new input pattern is presented, the neuron outputs are computed. Because of the feedback paths, the inputs to each neuron are then modified, which leads the network to enter a new state.

Different network architectures require appropriate learning algorithms. The next section provides an overview of learning processes.

Learning

The ability to learn is a fundamental trait of intelligence. Although a precise definition of learning is difficult to formulate, a learning process in the ANN context can be viewed as the problem of updating network architecture and connection weights so that a network can efficiently perform a specific task. The network usually must learn the connection weights from available training patterns. Performance is improved over time by iteratively updating the weights in the network. ANNs' ability to automatically *learn from examples* makes them attractive and exciting. Instead of following a set of *rules* specified by human experts, ANNs appear to learn underlying rules (like input-output relationships) from the given collection of representative examples. This is one of the major advantages of neural networks over traditional expert systems.

To understand or design a learning process, you must first have a model of the environment in which a neural network operates, that is, you must know what information is available to the network. We refer to this model as

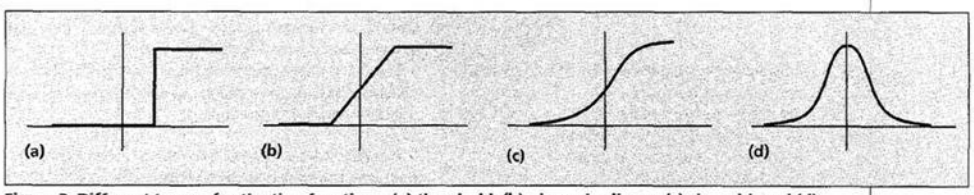


Figure 3. Different types of activation functions: (a) threshold, (b) piecewise linear, (c) sigmoid, and (d) Gaussian.

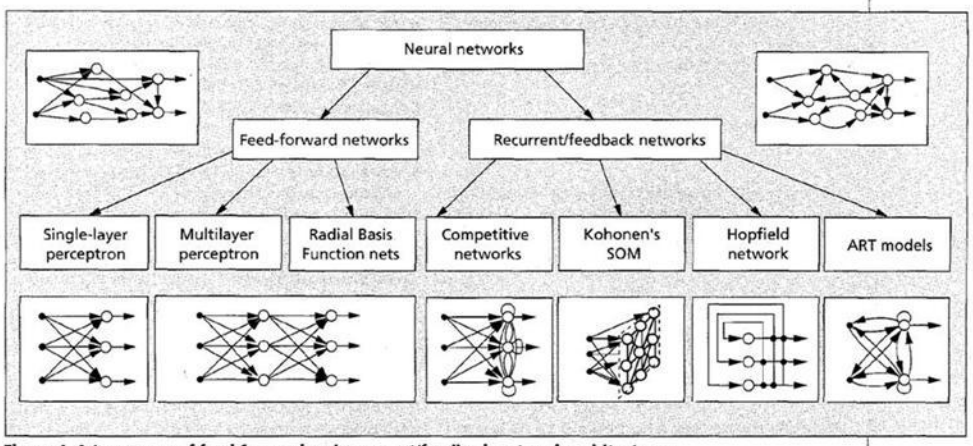


Figure 4. A taxonomy of feed-forward and recurrent/feedback network architectures.

a learning paradigm.[3] Second, you must understand how network weights are updated, that is, which *learning rules* govern the updating process. A *learning algorithm* refers to a procedure in which learning rules are used for adjusting the weights.

There are three main learning paradigms: supervised, unsupervised, and hybrid. In supervised learning, or learning with a "teacher," the network is provided with a correct answer (output) for every input pattern. Weights are determined to allow the network to produce answers as close as possible to the known correct answers. Reinforcement learning is a variant of supervised learning in which the network is provided with only a critique on the correctness of network outputs, not the correct answers themselves. In contrast, unsupervised learning, or learning without a teacher, does not require a correct answer associated with each input pattern in the training data set. It explores the underlying structure in the data, or correlations between patterns in the data, and organizes patterns into categories from these correlations. Hybrid learning combines supervised and unsupervised learning. Part of the weights are usually determined through supervised learning, while the others are obtained through unsupervised learning.

Learning theory must address three fundamental and practical issues associated with learning from samples: capacity, sample complexity, and computational complexity. Capacity concerns how many patterns can be stored, and what functions and decision boundaries a network can form.

Sample complexity determines the number of training patterns needed to train the network to guarantee a valid generalization. Too few patterns may cause "over-fitting" (wherein the network performs well on the training data set, but poorly on independent test patterns drawn from the same distribution as the training patterns, as in Figure A3).

Computational complexity refers to the time required for a learning algorithm to estimate a solution from training patterns. Many existing learning algorithms have high computational complexity. Designing efficient algorithms for neural network learning is a very active research topic.

There are four basic types of learning rules: error-correction, Boltzmann, Hebbian, and competitive learning.

ERROR-CORRECTION RULES. In the supervised learning paradigm, the network is given a desired output for each input pattern. During the learning process, the actual output y generated by the network may not equal the desired output d. The basic principle of error-correction learning rules is to use the error signal $(d - y)$ to modify the connection weights to gradually reduce this error.

The perceptron learning rule is based on this error-correction principle. A perceptron consists of a single neuron with adjustable weights, $w_j, j = 1, 2, \ldots, n$, and threshold u, as shown in Figure 2. Given an input vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^t$, the net input to the neuron is

Perceptron learning algorithm

1. Initialize the weights and threshold to small random numbers.
2. Present a pattern vector $(x_1, x_2, \ldots, x_n)^t$ and evaluate the output of the neuron.
3. Update the weights according to

$$w_j(t+1) = w_j(t) + \eta\,(d - y)\,x_j$$

where d is the desired output, t is the iteration number, and η $(0.0 < \eta < 1.0)$ is the gain (step size).

$$v = \sum_{j=1}^{n} w_j x_j - u$$

The output y of the perceptron is +1 if $v > 0$, and 0 otherwise. In a two-class classification problem, the perceptron assigns an input pattern to one class if $y = 1$, and to the other class if $y = 0$. The linear equation

$$\sum_{j=1}^{n} w_j x_j - u = 0$$

defines the decision boundary (a hyperplane in the n-dimensional input space) that halves the space.

Rosenblatt[5] developed a learning procedure to determine the weights and threshold in a perceptron, given a set of training patterns (see the "Perceptron learning algorithm" sidebar).

Note that learning occurs only when the perceptron makes an error. Rosenblatt proved that when training patterns are drawn from two linearly separable classes, the perceptron learning procedure converges after a finite number of iterations. This is the *perceptron convergence theorem*. In practice, you do not know whether the patterns are linearly separable. Many variations of this learning algorithm have been proposed in the literature.[2] Other activation functions that lead to different learning characteristics can also be used. However, *a single-layer perceptron can only separate linearly separable patterns as long as a monotonic activation function is used.*

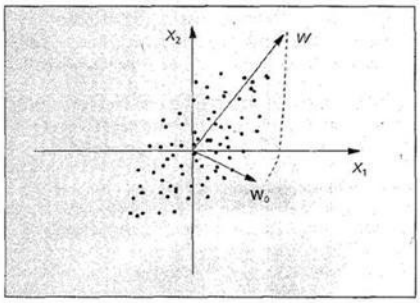


Figure 5. Orientation selectivity of a single neuron trained using the Hebbian rule.

The back-propagation learning algorithm (see the "Back-propagation algorithm sidebar") is also based on the error-correction principle.

BOLTZMANN LEARNING. Boltzmann machines are symmetric recurrent networks consisting of binary units (+1 for "on" and −1 for "off"). By symmetric, we mean that the weight on the connection from unit i to unit j is equal to the weight on the connection from unit j to unit i ($w_{ij} = w_{ji}$). A subset of the neurons, called *visible*, interact with the environment; the rest, called *hidden*, do not. Each neuron is a stochastic unit that generates an output (or state) according to the Boltzmann distribution of statistical mechanics. Boltzmann machines operate in two modes: *clamped*, in which visible neurons are clamped onto specific states determined by the environment; and *free-running*, in which both visible and hidden neurons are allowed to operate freely.

Boltzmann learning is a stochastic learning rule derived from information-theoretic and thermodynamic principles.[10] The objective of Boltzmann learning is to adjust the connection weights so that the states of visible units satisfy a particular desired probability distribution. According to the Boltzmann learning rule, the change in the connection weight w_{ij} is given by

$$\Delta w_{ij} = \eta(\bar{\rho}_{ij} - \rho_{ij}),$$

where η is the learning rate, and $\bar{\rho}_{ij}$ and ρ_{ij} are the correlations between the states of units i and j when the network operates in the clamped mode and free-running mode, respectively. The values of $\bar{\rho}_{ij}$ and ρ_{ij} are usually estimated from Monte Carlo experiments, which are extremely slow.

Boltzmann learning can be viewed as a special case of error-correction learning in which error is measured not as the direct difference between desired and actual outputs, but as the difference between the correlations among the outputs of two neurons under clamped and free-running operating conditions.

HEBBIAN RULE. The oldest learning rule is *Hebb's postulate of learning*.[13] Hebb based it on the following observation from neurobiological experiments: If neurons on both sides of a synapse are activated synchronously and repeatedly, the synapse's strength is selectively increased.

Mathematically, the Hebbian rule can be described as

$$w_{ij}(t+1) = w_{ij}(t) + \eta\, y_j(t)\, x_i(t),$$

where x_i and y_j are the output values of neurons i and j, respectively, which are connected by the synapse w_{ij}, and η is the learning rate. Note that x_i is the input to the synapse.

An important property of this rule is that learning is done locally, that is, the change in synapse weight depends only on the activities of the two neurons connected by it. This significantly simplifies the complexity of the learning circuit in a VLSI implementation.

A single neuron trained using the Hebbian rule exhibits an orientation selectivity. Figure 5 demonstrates this property. The points depicted are drawn from a two-dimen-

sional Gaussian distribution and used for training a neuron. The weight vector of the neuron is initialized to $\mathbf{w}_0$ as shown in the figure. As the learning proceeds, the weight vector moves progressively closer to the direction $\mathbf{w}$ of maximal variance in the data. In fact, $\mathbf{w}$ is the eigenvector of the covariance matrix of the data corresponding to the largest eigenvalue.

COMPETITIVE LEARNING RULES. Unlike Hebbian learning (in which multiple output units can be fired simultaneously), competitive-learning output units compete among themselves for activation. As a result, only one output unit is active at any given time. This phenomenon is known as *winner-take-all*. Competitive learning has been found to exist in biological neural networks.[3]

Competitive learning often clusters or categorizes the input data. Similar patterns are grouped by the network and represented by a single unit. This grouping is done automatically based on data correlations.

The simplest competitive learning network consists of a single layer of output units as shown in Figure 4. Each output unit i in the network connects to all the input units (x_j's) via weights, $w_{ij}, j=1, 2, \ldots, n$. Each output unit also connects to all other output units via inhibitory weights but has a self-feedback with an excitatory weight. As a result of competition, only the unit i^* with the largest (or the smallest) net input becomes the winner, that is, $\mathbf{w}_{i^*} \cdot \mathbf{x} \geq \mathbf{w}_i \cdot \mathbf{x}, \forall i$, or $\| \mathbf{w}_{i^*} - \mathbf{x} \| \leq \| \mathbf{w}_i - \mathbf{x} \|, \forall i$. When all the weight vectors are normalized, these two inequalities are equivalent.

A simple competitive learning rule can be stated as

$$\Delta w_{ij} = \begin{cases} \eta(x_j^u - w_{i^*j}), & i = i^*, \\ 0, & i \neq i^*. \end{cases} \quad (1)$$

Note that only the weights of the winner unit get updated. The effect of this learning rule is to move the stored pattern in the winner unit (weights) a little bit closer to the input pattern. Figure 6 demonstrates a geometric interpretation of competitive learning. In this example, we assume that all input vectors have been normalized to have unit length. They are depicted as black dots in Figure 6. The weight vectors of the three units are randomly initialized. Their initial and final positions on the sphere after competitive learning are marked as Xs in Figures 6a and 6b, respectively. In Figure 6, each of the three natural groups (clusters) of patterns has been discovered by an output unit whose weight vector points to the center of gravity of the discovered group.

You can see from the competitive learning rule that the network will not stop learning (updating weights) unless the learning rate η is 0. A particular input pattern can fire different output units at different iterations during learning. This brings up the stability issue of a learning system. The system is said to be *stable* if no pattern in the training data changes its category after a finite number of learning iterations. One way to achieve stability is to force the learning rate to decrease gradually as the learning process proceeds towards 0. However, this artificial freezing of learning causes another problem termed *plasticity*, which is the ability to adapt to new data. This is known as Grossberg's *stability-plasticity* dilemma in competitive learning.

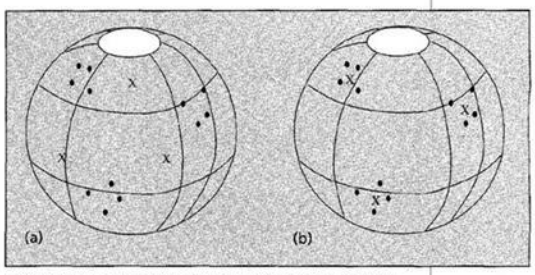


Figure 6. An example of competitive learning: (a) before learning; (b) after learning.

The most well-known example of competitive learning is *vector quantization* for data compression. It has been widely used in speech and image processing for efficient storage, transmission, and modeling. Its goal is to represent a set or distribution of input vectors with a relatively small number of prototype vectors (weight vectors), or a codebook. Once a codebook has been constructed and agreed upon by both the transmitter and the receiver, you need only transmit or store the index of the corresponding prototype to the input vector. Given an input vector, its corresponding prototype can be found by searching for the nearest prototype in the codebook.

SUMMARY. Table 2 summaries various learning algorithms and their associated network architectures (this is not an exhaustive list). Both supervised and unsupervised learning paradigms employ learning rules based

Back-propagation algorithm

1. Initialize the weights to small random values.
2. Randomly choose an input pattern $\mathbf{x}^{(u)}$.
3. Propagate the signal forward through the network.
4. Compute δ_i^L in the output layer ($o_i = y_i^L$)

   $$\delta_i^L = g'(h_i^L)\left[d_i^u - y_i^L\right],$$

   where h_i^l represents the net input to the ith unit in the lth layer, and g' is the derivative of the activation function g.
5. Compute the deltas for the preceding layers by propagating the errors backwards;

   $$\delta_i^l = g'(h_i^l)\sum_j w_{ij}^{l+1}\delta_j^{l+1},$$

   for $l = (L-1), \ldots, 1$.
6. Update weights using

   $$\Delta w_{ij}^l = \eta \delta_i^l y_j^{l-1}$$

7. Go to step 2 and repeat for the next pattern until the error in the output layer is below a prespecified threshold or a maximum number of iterations is reached.

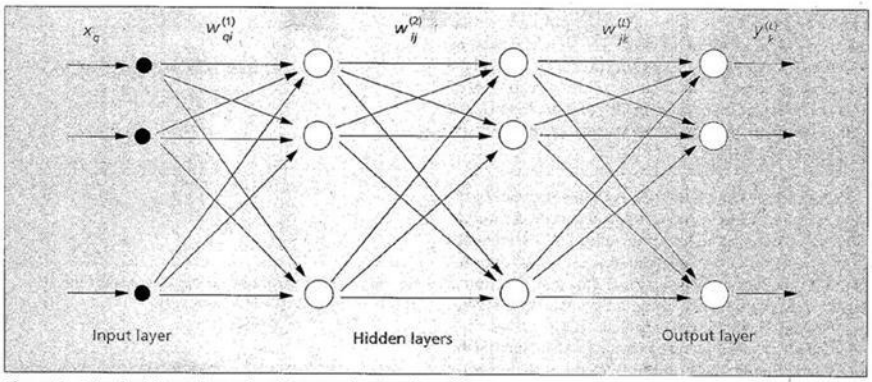


Figure 7. A typical three-layer feed-forward network architecture.

on error-correction, Hebbian, and competitive learning. Learning rules based on error-correction can be used for training feed-forward networks, while Hebbian learning rules have been used for all types of network architectures. However, each learning algorithm is designed for training a specific architecture. Therefore, when we discuss a learning algorithm, a particular network architecture association is implied. Each algorithm can

Table 2. Well-known learning algorithms.

Paradigm	Learning rule	Architecture	Learning algorithm	Task
Supervised	Error-correction	Single- or multilayer perceptron	Perceptron learning algorithms Back-propagation Adaline and Madaline	Pattern classification Function approximation Prediction, control
	Boltzmann	Recurrent	Boltzmann learning algorithm	Pattern classification
	Hebbian	Multilayer feed-forward	Linear discriminant analysis	Data analysis Pattern classification
	Competitive	Competitive	Learning vector quantization	Within-class categorization Data compression
		ART network	ARTMap	Pattern classification Within-class categorization
Unsupervised	Error-correction	Multilayer feed-forward	Sammon's projection	Data analysis
	Hebbian	Feed-forward or competitive	Principal component analysis	Data analysis Data compression
		Hopfield Network	Associative memory learning	Associative memory
	Competitive	Competitive	Vector quantization	Categorization Data compression
		Kohonen's SOM	Kohonen's SOM	Categorization Data analysis
		ART networks	ART1, ART2	Categorization
Hybrid	Error-correction and competitive	RBF network	RBF learning algorithm	Pattern classification Function approximation Prediction, control

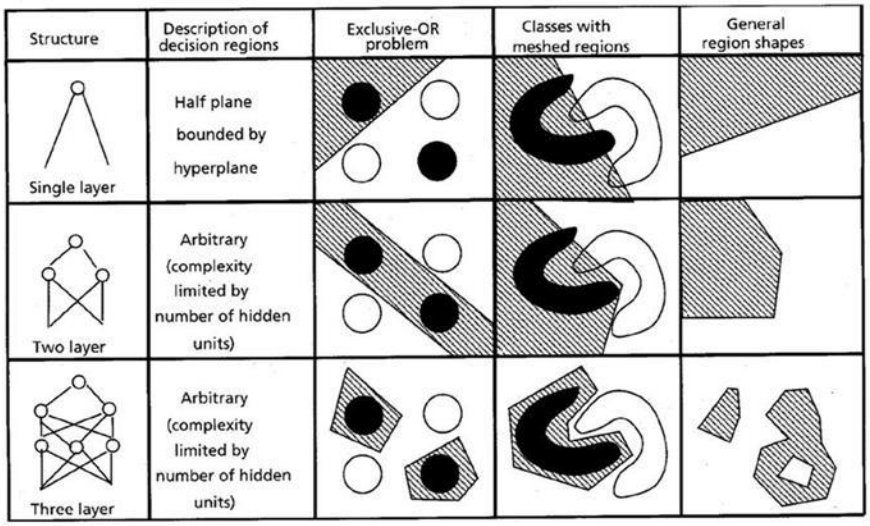


Figure 8. A geometric interpretation of the role of hidden unit in a two-dimensional input space.

perform only a few tasks well. The last column of Table 2 lists the tasks that each algorithm can perform. Due to space limitations, we do not discuss some other algorithms, including Adaline, Madaline,[14] linear discriminant analysis,[15] Sammon's projection,[15] and principal component analysis.[2] Interested readers can consult the corresponding references (this article does not always cite the first paper proposing the particular algorithms).

MULTILAYER FEED-FORWARD NETWORKS

Figure 7 shows a typical three-layer perceptron. In general, a standard L-layer feed-forward network (we adopt the convention that the input nodes are not counted as a layer) consists of an input stage, $(L-1)$ hidden layers, and an output layer of units successively connected (fully or locally) in a feed-forward fashion with no connections between units in the same layer and no feedback connections between layers.

Multilayer perceptron

The most popular class of multilayer feed-forward networks is *multilayer perceptrons* in which each computational unit employs either the thresholding function or the sigmoid function. Multilayer perceptrons can form arbitrarily complex decision boundaries and represent any Boolean function.[6] The development of the *back-propagation* learning algorithm for determining weights in a multilayer perceptron has made these networks the most popular among researchers and users of neural networks.

We denote $w_{ij}^{(l)}$ as the weight on the connection between the ith unit in layer $(l-1)$ to jth unit in layer l.

Let $\{(\mathbf{x}^{(1)}, \mathbf{d}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{d}^{(2)}), \ldots, (\mathbf{x}^{(p)}, \mathbf{d}^{(p)})\}$ be a set of p training patterns (input-output pairs), where $\mathbf{x}^{(i)} \in R^n$ is the input vector in the n-dimensional pattern space, and $\mathbf{d}^{(i)} \in [0, 1]^m$, an m-dimensional hypercube. For classification purposes, m is the number of classes. The squared-error cost function most frequently used in the ANN literature is defined as

$$E = \frac{1}{2}\sum_{i=1}^{p} \left\| \mathbf{y}^{(i)} - \mathbf{d}^{(i)} \right\|^2 \qquad (2)$$

The back-propagation algorithm[9] is a gradient-descent method to minimize the squared-error cost function in Equation 2 (see "Back-propagation algorithm" sidebar).

A geometric interpretation (adopted and modified from Lippmann[16]) shown in Figure 8 can help explicate the role of hidden units (with the threshold activation function).

Each unit in the first hidden layer forms a hyperplane in the pattern space; boundaries between pattern classes can be approximated by hyperplanes. A unit in the second hidden layer forms a hyperregion from the outputs of the first-layer units; a decision region is obtained by performing an AND operation on the hyperplanes. The output-layer units combine the decision regions made by the units in the second hidden layer by performing logical OR operations. Remember that this scenario is depicted only to explain the role of hidden units. Their actual behavior, after the network is trained, could differ.

A two-layer network can form more complex decision boundaries than those shown in Figure 8. Moreover, multilayer perceptrons with sigmoid activation functions can form smooth decision boundaries rather than piecewise linear boundaries.

Radial Basis Function network

The Radial Basis Function (RBF) network,[3] which has two layers, is a special class of multilayer feed-forward net-

works. Each unit in the hidden layer employs a radial basis function, such as a Gaussian kernel, as the activation function. The radial basis function (or kernel function) is centered at the point specified by the weight vector associated with the unit. Both the positions and the widths of these kernels must be learned from training patterns. There are usually many fewer kernels in the RBF network than there are training patterns. Each output unit implements a linear combination of these radial basis functions. From the point of view of function approximation, the hidden units provide a set of functions that constitute a basis set for representing input patterns in the space spanned by the hidden units.

There are a variety of learning algorithms for the RBF network.[3] The basic one employs a two-step learning strategy, or hybrid learning. It estimates kernel positions and kernel widths using an unsupervised clustering algorithm, followed by a supervised least mean square (LMS) algorithm to determine the connection weights between the hidden layer and the output layer. Because the output units are linear, a noniterative algorithm can be used. After this initial solution is obtained, a supervised gradient-based algorithm can be used to refine the network parameters.

This hybrid learning algorithm for training the RBF network converges much faster than the back-propagation algorithm for training multilayer perceptrons. However, for many problems, the RBF network often involves a larger number of hidden units. This implies that the runtime (after training) speed of the RBF network is often slower than the runtime speed of a multilayer perceptron. The efficiencies (error versus network size) of the RBF network and the multilayer perceptron are, however, problem-dependent. It has been shown that the RBF network has the same asymptotic approximation power as a multilayer perceptron.

SOM learning algorithm

1. Initialize weights to small random numbers; set initial learning rate and neighborhood.
2. Present a pattern $\mathbf{x}$, and evaluate the network outputs.
3. Select the unit (c_i, c_j) with the minimum output:

$$\|\mathbf{x} - \mathbf{w}_{c_i c_j}\| = \min_{ij} \|\mathbf{x} - \mathbf{w}_{ij}\|$$

4. Update all weights according to the following learning rule:

$$\mathbf{w}_{ij}(t+1) = \begin{cases} \mathbf{w}_{ij}(t) + \alpha(t)[\mathbf{x}(t) - \mathbf{w}_{ij}(t)], & \text{if } (i,j) \in N_{c_i c_j}(t), \\ \mathbf{w}_{ij}(t), & \text{otherwise,} \end{cases}$$

where $N_{c_i c_j}(t)$ is the neighborhood of the unit (c_i, c_j) at time t, and $\alpha(t)$ is the learning rate.

5. Decrease the value of $\alpha(t)$ and shrink the neighborhood $N_{c_i c_j}(t)$.
6. Repeat steps 2 through 5 until the change in weight values is less than a prespecified threshold or a maximum number of iterations is reached.

Issues

There are many issues in designing feed-forward networks, including

- how many layers are needed for a given task,
- how many units are needed per layer,
- how will the network perform on data not included in the training set (generalization ability), and
- how large the training set should be for "good" generalization.

Although multilayer feed-forward networks using back-propagation have been widely employed for classification and function approximation,[2] many design parameters still must be determined by trial and error. Existing theoretical results provide only very loose guidelines for selecting these parameters in practice.

KOHONEN'S SELF-ORGANIZING MAPS

The self-organizing map (SOM)[16] has the desirable property of topology preservation, which captures an important aspect of the feature maps in the cortex of highly developed animal brains. In a topology-preserving mapping, nearby input patterns should activate nearby output units on the map. Figure 4 shows the basic network architecture of Kohonen's SOM. It basically consists of a two-dimensional array of units, each connected to all n input nodes. Let $\mathbf{w}_{ij}$ denote the n-dimensional vector associated with the unit at location (i, j) of the 2D array. Each neuron computes the Euclidean distance between the input vector $\mathbf{x}$ and the stored weight vector $\mathbf{w}_{ij}$.

This SOM is a special type of competitive learning network that defines a spatial neighborhood for each output unit. The shape of the local neighborhood can be square, rectangular, or circular. Initial neighborhood size is often set to one half to two thirds of the network size and shrinks over time according to a schedule (for example, an exponentially decreasing function). During competitive learning, all the weight vectors associated with the winner and its neighboring units are updated (see the "SOM learning algorithm" sidebar).

Kohonen's SOM can be used for projection of multivariate data, density approximation, and clustering. It has been successfully applied in the areas of speech recognition, image processing, robotics, and process control.[2] The design parameters include the dimensionality of the neuron array, the number of neurons in each dimension, the shape of the neighborhood, the shrinking schedule of the neighborhood, and the learning rate.

ADAPTIVE RESONANCE THEORY MODELS

Recall that the *stability-plasticity* dilemma is an important issue in competitive learning. How do we learn new things (plasticity) and yet retain the stability to ensure that existing knowledge is not erased or corrupted? Carpenter and Grossberg's Adaptive Resonance Theory models (ART1, ART2, and ARTMap) were developed in an attempt to overcome this dilemma.[17] The network has a sufficient supply of output units, but they are not used until deemed necessary. A unit is said to be *committed* (*uncommitted*) if it is (is not) being used. The learning algorithm updates

the stored prototypes of a category only if the input vector is sufficiently similar to them. An input vector and a stored prototype are said to resonate when they are sufficiently similar. The extent of similarity is controlled by a *vigilance parameter*, ρ, with $0 < \rho < 1$, which also determines the number of categories. When the input vector is not sufficiently similar to any existing prototype in the network, a new category is created, and an uncommitted unit is assigned to it with the input vector as the initial prototype. If no such uncommitted unit exists, a novel input generates no response.

We present only ART1, which takes binary (0/1) input to illustrate the model. Figure 9 shows a simplified diagram of the ART1 architecture.[2] It consists of two layers of fully connected units. A top-down weight vector $\mathbf{w}_j$ is associated with unit j in the input layer, and a bottom-up weight vector $\overline{\mathbf{w}}_i$ is associated with output unit i; $\overline{\mathbf{w}}_i$ is the normalized version of $\mathbf{w}_i$.

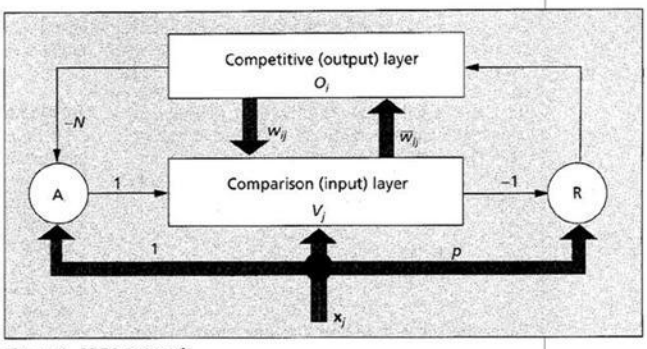


Figure 9. ART1 network.

$$\overline{\mathbf{w}}_i = \frac{\mathbf{w}_i}{\varepsilon + \sum_j w_{ji}}, \tag{3}$$

where ε is a small number used to break the ties in selecting the winner. The top-down weight vectors $\mathbf{w}_j$'s store cluster prototypes. The role of normalization is to prevent prototypes with a long vector length from dominating prototypes with a short one. Given an n-bit input vector $\mathbf{x}$, the output of the auxiliary unit A is given by

$$A = Sgn_{0/1}\left(\sum_j x_j - n\sum_i O_i - 0.5\right),$$

where $Sgn_{0/1}(x)$ is the *signum* function that produces +1 if $x \geq 0$ and 0 otherwise, and the output of an input unit is given by

$$V_j = Sgn_{0/1}\left(x_j + \sum_i w_{ji}O_i + A - 1.5\right) = \begin{cases} x_j, & \text{if no output } O_j \text{ is "on",} \\ x_j \wedge \Sigma_i w_{ji}O_i, & \text{otherwise.} \end{cases}$$

A reset signal R is generated only when the similarity is less than the vigilance level. (See the "ART1 learning algorithm" sidebar.)

The ART1 model can create new categories and reject an input pattern when the network reaches its capacity. However, the number of categories discovered in the input data by ART1 is sensitive to the vigilance parameter.

HOPFIELD NETWORK

Hopfield used a network *energy* function as a tool for designing recurrent networks and for understanding their dynamic behavior.[7] Hopfield's formulation made explicit the principle of storing information as dynamically stable attractors and popularized the use of recurrent networks for associative memory and for solving combinatorial optimization problems.

A Hopfield network with n units has two versions: binary and continuously valued. Let v_i be the state or output of the ith unit. For binary networks, v_i is either +1 or −1, but for continuous networks, v_i can be any value between 0 and 1. Let w_{ij} be the synapse weight on the connection from units i to j. In Hopfield networks, $w_{ij} = w_{ji}$, $\forall i, j$ (symmetric networks), and $w_{ii} = 0$, $\forall i$ (no self-feedback connections). The network dynamics for the binary Hopfield network are

$$v_i = Sgn\left(\sum_j w_{ij}v_j - \theta_i\right) \tag{4}$$

ART1 learning algorithm

1. Initialize $w_{ij} = 1$, for all i, j. Enable all the output units.
2. Present a new pattern $\mathbf{x}$.
3. Find the winner unit i^* among the enabled output units

   $\overline{\mathbf{w}}_{i^*} \cdot \mathbf{x} \geq \overline{\mathbf{w}}_i \cdot \mathbf{x}, \forall i$

4. Perform vigilance test

   $r = \dfrac{\mathbf{w}_{i^*} \cdot \mathbf{x}}{\Sigma_j x_j}$

   If $r \geq \rho$ (resonance), go to step 5. Otherwise, disable unit i^* and go to step 3 (until all the output units are disabled).
5. Update the winning weight vector $\mathbf{w}_{i^*}$, enable all the output units, and go to step 2

   $\Delta w_{ji^*} = \eta (V_j - w_{ji^*})$

6. If all output units are disabled, select one of the uncommitted output units and set its weight vector to x. If there is no uncommitted output unit (capacity is reached), the network rejects the input pattern.

The dynamic update of network states in Equation 4 can be carried out in at least two ways: *synchronously* and *asynchronously*. In a synchronous updating scheme, all units are updated simultaneously at each time step. A central clock must synchronize the process. An asynchronous updating scheme selects one unit at a time and updates its state. The unit for updating can be randomly chosen.

The energy function of the binary Hopfield network in a state $\mathbf{v} = (v_1, v_2, \ldots, v_n)^T$ is given by

$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} v_i v_j \qquad (5)$$

The central property of the energy function is that as network state evolves according to the network dynamics (Equation 4), the network energy always decreases and eventually reaches a local minimum point (attractor) where the network stays with a constant energy.

Associative memory

When a set of patterns is stored in these network attractors, it can be used as an *associative memory*. Any pattern present in the basin of attraction of a stored pattern can be used as an index to retrieve it.

An associative memory usually operates in two phases: storage and retrieval. In the storage phase, the weights in the network are determined so that the attractors of the network memorize a set of p n-dimensional patterns $\{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^p\}$ to be stored. A generalization of the Hebbian learning rule can be used for setting connection weights w_{ij}. In the retrieval phase, the input pattern is used as the initial state of the network, and the network evolves according to its dynamics. A pattern is produced (or retrieved) when the network reaches equilibrium.

How many patterns can be stored in a network with n binary units? In other words, what is the memory capacity of a network? It is finite because a network with n binary units has a maximum of 2^n distinct states, and not all of them are attractors. Moreover, not all attractors (stable states) can store useful patterns. Spurious attractors can also store patterns different from those in the training set.[2]

It has been shown that the maximum number of random patterns that a Hopfield network can store is $P_{max} \approx 0.15n$. When the number of stored patterns $p < 0.15n$, a nearly perfect recall can be achieved. When memory patterns are orthogonal vectors instead of random patterns, more patterns can be stored. But the number of spurious attractors increases as p reaches capacity. Several learning rules have been proposed for increasing the memory capacity of Hopfield networks.[2] Note that we require n^2 connections in the network to store p n-bit patterns.

Energy minimization

Hopfield networks always evolve in the direction that leads to lower network energy. This implies that if a combinatorial optimization problem can be formulated as minimizing this energy, the Hopfield network can be used to find the optimal (or suboptimal) solution by letting the network evolve freely. In fact, any quadratic objective function can be rewritten in the form of Hopfield network energy. For example, the classic Traveling Salesman Problem can be formulated as such a problem.

APPLICATIONS

We have discussed a number of important ANN models and learning algorithms proposed in the literature. They have been widely used for solving the seven classes of problems described in the beginning of this article. Table 2 showed typical suitable tasks for ANN models and learning algorithms. Remember that to successfully work with real-world problems, you must deal with numerous design issues, including network model, network size, activation function, learning parameters, and number of training samples. We next discuss an optical character recognition (OCR) application to illustrate how multilayer feed-forward networks are successfully used in practice.

OCR deals with the problem of processing a scanned image of text and transcribing it into machine-readable form. We outline the basic components of OCR and explain how ANNs are used for character classification.

An OCR system

An OCR system usually consists of modules for preprocessing, segmentation, feature extraction, classification, and contextual processing. A paper document is scanned to produce a gray-level or binary (black-and-white) image. In the preprocessing stage, filtering is applied to remove noise, and text areas are located and converted to a binary image using a global or local adaptive thresholding method. In the segmentation step, the text image is separated into individual characters. This is a particularly difficult task with handwritten text, which contains a proliferation of touching characters. One effective technique is to break the composite pattern into smaller patterns (over-segmentation) and find the correct character segmentation points using the output of a pattern classifier.

Because of various degrees of slant, skew, and noise level, and various writing styles, recognizing segmented characters is not easy. This is evident from Figure 10, which shows the size-normalized character bitmaps of a sample set from the NIST (National Institute of Standards and Technology) hand-print character database.[16]

Schemes

Figure 11 shows the two main schemes for using ANNs in an OCR system. The first one employs an explicit feature extractor (not necessarily a neural network). For instance, contour direction features are used in Figure 11. The extracted features are passed to the input stage of a multilayer feed-forward network.[19] This scheme is very flexible in incorporating a large variety of features. The other scheme does not explicitly extract features from the raw data. The feature extraction implicitly takes place within the intermediate stages (hidden layers) of the ANN. A nice property of this scheme is that feature extraction and classification are integrated and trained simultaneously to produce optimal classification results. It is not clear whether the types of features that can be extracted by this integrated architecture are the most effective for character recognition. Moreover, this scheme requires a much larger network than the first one.

A typical example of this integrated feature extraction-classification scheme is the network developed by Le Cun et al.[20] for zip code recognition. A 16 × 16 normalized gray-level image is presented to a feed-forward network with three hidden layers. The units in the first layer are locally connected to the units in the input layer, forming a set of local feature maps. The second hidden layer is constructed in a similar way. Each unit in the second layer also combines local information coming from feature maps in the first layer.

The activation level of an output unit can be interpreted as an approximation of the a posteriori probability of the input pattern's belonging to a particular class. The output categories are ordered according to activation levels and passed to the post-processing stage. In this stage, contextual information is exploited to update the classifier's output. This could, for example, involve looking up a dictionary of admissible words, or utilizing syntactic constraints present, for example, in phone or social security numbers.

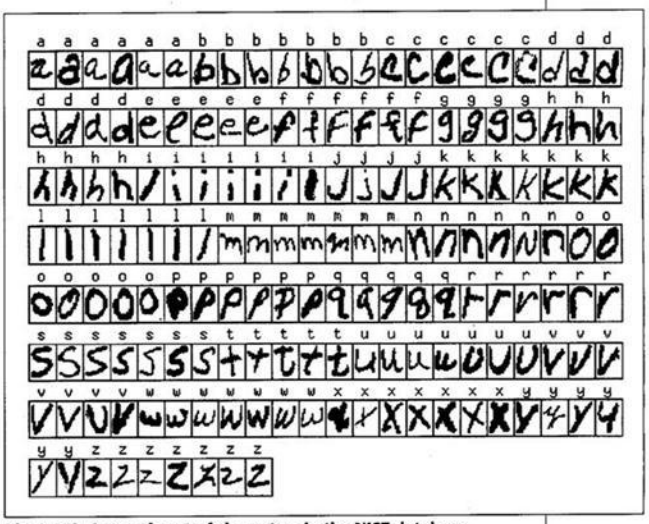


Figure 10. A sample set of characters in the NIST database.

Results

ANNs work very well in the OCR application. However, there is no conclusive evidence about their superiority over conventional statistical pattern classifiers. At the First Census Optical Character Recognition System Conference held in 1992,[18] more than 40 different handwritten character recognition systems were evaluated based on their performance on a common database. The top 10 performers used either some type of multilayer feed-forward network or a nearest neighbor-based classifier. ANNs tend to be superior in terms of speed and memory requirements compared to nearest neighbor methods. Unlike the nearest neighbor methods, classification speed using ANNs is independent of the size of the training set. The recognition accuracies of the top OCR systems on the NIST isolated (presegmented) character data were above 98 percent for digits, 96 percent for uppercase characters, and 87 percent for lowercase characters. (Low recognition accuracy for lowercase characters was largely due to the fact that the test data differed significantly from the training data, as well as being due to "ground-truth" errors.) One conclusion drawn from the test is that OCR system performance on isolated characters compares well with human performance. However, humans still outperform OCR systems on unconstrained and cursive handwritten documents.

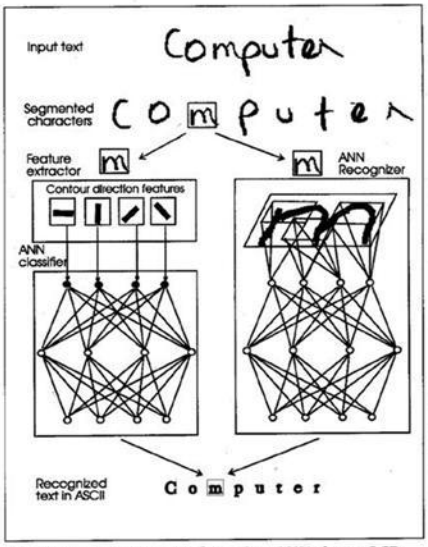


Figure 11. Two schemes for using ANNs in an OCR system.

DEVELOPMENTS IN ANNS HAVE STIMULATED a lot of enthusiasm and criticism. Some comparative studies are optimistic, some offer pessimism. For many tasks, such as pattern recognition, no one approach dominates the others. The choice of the best technique should be driven by the given application's nature. We should try to understand the capacities, assumptions, and applicability of various approaches and maximally exploit their complementary advantages to develop better intelligent systems. Such an effort may lead to a synergistic approach that combines the strengths of ANNs with other technologies to achieve significantly better performance for challenging problems. As Minsky[21] recently observed, the time has come to build systems out of diverse components. Individual modules are important, but we also need a good methodology for integration. It is clear that communication and cooperative work between

researchers working in ANNs and other disciplines will not only avoid repetitious work but (and more important) will stimulate and benefit individual disciplines. ■

Acknowledgments

We thank Richard Casey (IBM Almaden); Pat Flynn (Washington State University); William Punch, Chitra Dorai, and Kalle Karu (Michigan State University); Ali Khotanzad (Southern Methodist University); and Ishwar Sethi (Wayne State University) for their many useful suggestions.

References

1. *DARPA Neural Network Study*, AFCEA Int'l Press, Fairfax, Va., 1988.
2. J. Hertz, A. Krogh, and R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Reading, Mass., 1991.
3. S. Haykin, *Neural Networks: A Comprehensive Foundation*, MacMillan College Publishing Co., New York, 1994.
4. W.S. McCulloch and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bull. Mathematical Biophysics*, Vol. 5, 1943, pp. 115-133.
5. R. Rosenblatt, *Principles of Neurodynamics*, Spartan Books, New York, 1962.
6. M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, Mass., 1969.
7. J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," in *Proc. Nat'l Academy of Sciences*, USA 79, 1982, pp. 2,554-2,558.
8. P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," PhD thesis, Dept. of Applied Mathematics, Harvard University, Cambridge, Mass., 1974.
9. D.E. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, MIT Press, Cambridge, Mass., 1986.
10. J.A. Anderson and E. Rosenfeld, *Neurocomputing: Foundations of Research*, MIT Press, Cambridge, Mass., 1988.
11. S. Brunak and B. Lautrup, *Neural Networks, Computers with Intuition*, World Scientific, Singapore, 1990.
12. J. Feldman, M.A. Fanty, and N.H. Goddard, "Computing with Structured Neural Networks," *Computer*, Vol. 21, No. 3, Mar. 1988, pp. 91-103.
13. D.O. Hebb, *The Organization of Behavior*, John Wiley & Sons, New York, 1949.
14. R.P. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, Vol. 4, No. 2, Apr. 1987, pp. 4-22.
15. A.K. Jain and J. Mao, "Neural Networks and Pattern Recognition," in *Computational Intelligence: Imitating Life*, J.M. Zurada, R. J. Marks II, and C.J. Robinson, eds., IEEE Press, Piscataway, N.J., 1994, pp. 194-212.
16. T. Kohonen, *Self Organization and Associative Memory*, Third Edition, Springer-Verlag, New York, 1989.
17. G.A. Carpenter and S. Grossberg, *Pattern Recognition by Self-Organizing Neural Networks*, MIT Press, Cambridge, Mass., 1991.
18. "The First Census Optical Character Recognition System Conference," R.A. Wilkinson et al., eds., . Tech. Report, NISTIR 4912, US Dept. Commerce, NIST, Gaithersburg, Md., 1992.
19. K. Mohiuddin and J. Mao, "A Comparative Study of Different Classifiers for Handprinted Character Recognition," in *Pattern Recognition in Practice IV*, E.S. Gelsema and L.N. Kanal, eds., Elsevier Science, The Netherlands, 1994, pp. 437-448.
20. Y. Le Cun et al., "Back-Propagation Applied to Handwritten Zip Code Recognition, *Neural Computation*, Vol. 1, 1989, pp. 541–551.
21. M. Minsky, "Logical Versus Analogical or Symbolic Versus Connectionist or Neat Versus Scruffy," *AI Magazine*, Vol. 65, No. 2, 1991, pp. 34-51.

Anil K. Jain *is a University Distinguished Professor and the chair of the Department of Computer Science at Michigan State University. His interests include statistical pattern recognition, exploratory pattern analysis, neural networks, Markov random fields, texture analysis, remote sensing, interpretation of range images, and 3D object recognition.*

Jain served as editor-in-chief of IEEE Transactions on Pattern Analysis and Machine Intelligence *from 1991 to 1994, and currently serves on the editorial boards of* Pattern Recognition, Pattern Recognition Letters, Journal of Mathematical Imaging, Journal of Applied Intelligence, *and* IEEE Transactions on Neural Networks. *He has coauthored, edited, and coedited numerous books in the field. Jain is a fellow of the IEEE and a speaker in the IEEE Computer Society's Distinguished Visitors Program for the Asia-Pacific region. He is a member of the IEEE Computer Society.*

Jianchang Mao *is a research staff member at the IBM Almaden Research Center. His interests include pattern recognition, neural networks, document image analysis, image processing, computer vision, and parallel computing.*

Mao received the BS degree in physics in 1983 and the MS degree in electrical engineering in 1986 from East China Normal University in Shanghai. He received the PhD in computer science from Michigan State University in 1994. Mao is the abstracts editor of IEEE Transactions on Neural Networks. *He is a member of the IEEE and the IEEE Computer Society.*

K.M. Mohiuddin *is the manager of the Document Image Analysis and Recognition project in the Computer Science Department at the IBM Almaden Research Center. He has led IBM projects on high-speed reconfigurable machines for industrial machine vision, parallel processing for scientific computing, and document imaging systems. His interests include document image analysis, handwriting recognition/OCR, data compression, and computer architecture.*

Mohiuddin received the MS and PhD degrees in electrical engineering from Stanford University in 1977 and 1982, respectively. He is an associate editor of IEEE Transactions on Pattern Analysis and Machine Intelligence. *He served on Computer's editorial board from 1984 to 1989, and is a senior member of the IEEE and a member of the IEEE Computer Society.*

Readers can contact Anil Jain at the Department of Computer Science, Michigan State University, A714 Wells Hall, East Lansing, MI 48824; jain@cps.msu.edu.