The thinking behind neural networks

How we go about mimicking the brain's neurons in code

We've all got a very sophisticated processing unit – the brain – that can perform some remarkable tasks. Despite their speed and memory capacity, silicon-based computers struggle to emulate it. The branch of computer science called Artificial Intelligence tries to narrow the gap, and one of the basic tools of AI is the neural network. So let's take a look at what the neural network can do.

Over the years, Artificial Intelligence has had its ups and downs. Generally there would be a period of 'up' when, after a short run of successful papers, researchers would start making prognostications about their discipline that would grow ever more fanciful. This would naturally lead to a period of 'down' when these predictions did not come to pass.

However, just as spin-off software from the space program have made their way into retail products, spin-offs from AI are becoming part of our lives through intelligent software, even though we may not recognise it as such.

Keeping it real

One fairly recent example that comes to mind is the ability of some point-and-shoot cameras to detect when a face is in shot and hence focus on that face. The face detection software is remarkably fast and rarely wrong, so when taking portraits with these cameras it's easy to trust that the faces of the subjects will be in focus and exposed correctly.

Apple's new version of its iPhoto app goes one step further: it includes face recognition software. Import your photos into iPhoto, and it will detect faces. It's then able to recognise the same faces in different photos. Once you've 'named' the face, iPhoto will annotate the picture with the faces it recognises.

Another business-oriented application of AI algorithms is voice recognition in programs like Dragon Naturally Speaking and OSes like Windows 7. Some cars that include optional 'Technology' packages also have voice recognition for controlling the car's interior functions like the radio or the heating. (I've given up talking to my car: since I'm British but living in the States, the car's voice recognition software doesn't 'get' my voice, perhaps because it's optimised for a US accent.)

Yet another example is OCR (Optical Character Recognition). Here the state of play is quite remarkable, with the top-end packages declaring over 99 per cent accuracy for typewritten or typeset text. Even the old Palm Pilot PDAs had very constrained – yet very successful – handwriting recognition ▸

software; once you'd trained yourself to write the modified characters, the PDA recognised them as swiftly as you could write them.

Although these AI applications use many different techniques to do their magic, there is a very fundamental building block called the neural network, from which many of these techniques are but refinements.

How it works

Before we can get an appreciation of what a neural network does, we should look at the biological background from which it is derived. If you looked at a brain in a microscope you'd see that it consists of specialised cells called neurons. Neurons (Figure 1) are peculiar cells indeed. The main body of the neuron is called the soma, and it has a veritable forest of dendrites through which input signals arrive. If the number of incoming signals is sufficient, the difference in voltage potential will cause the axon hillock to fire its own signal down the axon, a comparatively long extension of the cell. The axon branches out towards the end, and at the end of each branch is a synapse that connects to a dendrite of another neuron. The signal travels through the synapse (we talk of the synapse firing) into the dendrite and this signal then participates in whether the next neuron fires or not.

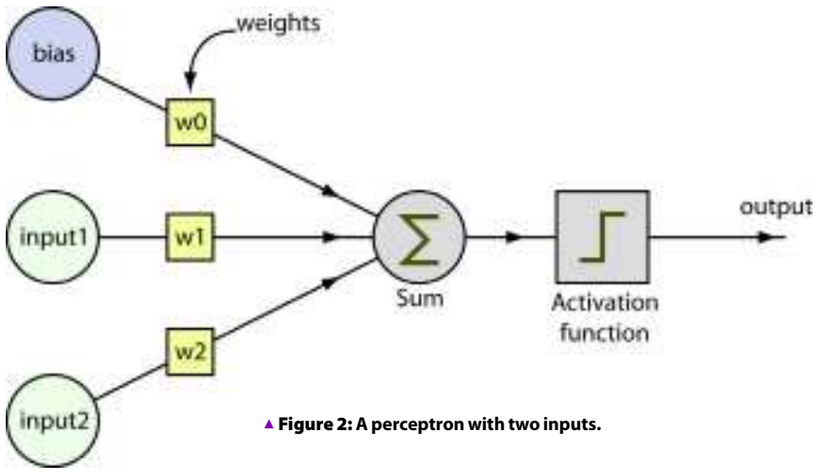
So, boiling this down to the absolute fundamentals (without worrying about the chemical processes that help the signal travel across the synaptic gap, or about the myriad other processes in the cell) we have:

- a set of input signals coming into the cell from other cells;
- if the sum of the signals reaches a threshold, the cell fires its own signal;
- the output signal from a cell will become the input signal to several other cells.

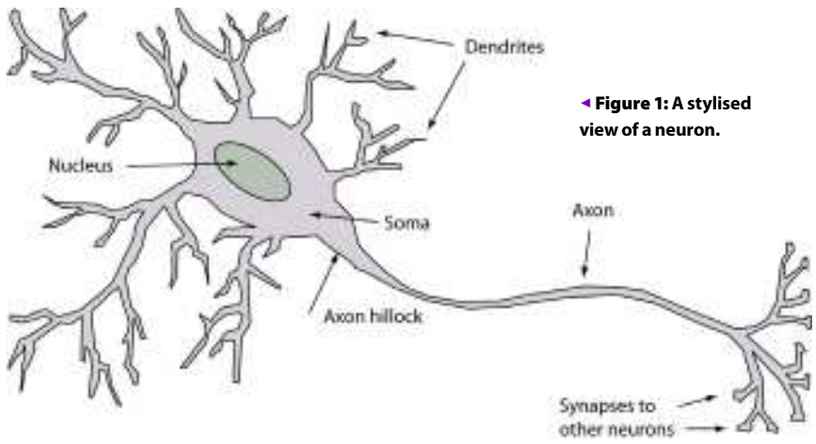
So, in short: inputs, summation and, if above threshold, output. Sounds computer-like.

In the human brain there are roughly 20 billion neurons (the number depends on various factors, including age and gender). Each neuron will be connected through synapses to roughly 10,000 other neurons. The brain is a giant, complicated network of dendritic connections. Unlike computers, it's massively parallel: computations are going on all over the brain. It boggles the mind how complex it is – indeed, how it works at all.

So let's draw back from the brink and look at how we might mimic this in computing.



▲ **Figure 2: A perceptron with two inputs.**



◀ **Figure 1: A stylised view of a neuron.**

Replicating nature

Sadly, there's no way we can mimic 20 billion neurons with 10,000 connections each, but there are several interesting things we can do with much less firepower. Way back in 1957, Frank Rosenblatt modelled a single neuron with something he called a 'perceptron', and used it to investigate pattern recognition. Unfortunately, the perceptron was unable to recognise even simple functions like XOR (proved formally by Marvin Minsky and Seymour Papert in 1969) and so it was abandoned in favour of something called multilayer feedforward networks. Nevertheless we can use many of the concepts associated with the perceptron later on.

Figure 2 shows a standard perceptron. We have a set of inputs on the left-hand side. Each input has a 'weight' associated with it. Each input signal (which is a floating-point value, positive or negative) is multiplied by its weight (another floating-point value). All of these products are summed. If the sum exceeds a threshold value (generally 0), the perceptron outputs 1 (or 'true'). If the threshold is not exceeded, the perceptron outputs 0 (or 'false'). This test is known as the activation function.

To help with the process, another fixed input is usually provided (known as the 'bias'). This models the propensity of the perceptron to fire in spite of the values of its inputs. The bias is normally 1 and will have its own weight.

All this is very well, but where do the weights come from? The inputs are obviously provided by us in some form, but who provides the weights? Look at it like this. Suppose we want a perceptron to calculate the same result as the AND operation. There will be two inputs to this perception, A and B. Each input will be constrained to two possible values, 0 and 1. If both A and B equal 1, the perceptron should output 1; otherwise it should output 0. We have to determine three weights here: the weights for A and B and the bias. Once we have these, we should be able to run the perceptron,

Spotlight on… Reinforcement learning

There are two main methods of training a network: supervised learning and reinforcement learning. Back propagation is a standard method for supervised learning: the net provides information (the error from a training set) that is then fed back into the neural network to improve the weights and thereby cause the network to learn from the training sets. This is akin to teaching a child the word 'chair' by showing them several different objects that are chairs, and several that are not. Reinforcement learning, on the other hand, stems from a different environment. The environment is dynamic, and the network must learn on the fly. The example of this is in a gaming environment, where the network is playing against other players and the whole environment is in a state of flux. Here the outputs from the network provide a reinforcement signal (good or bad). This is akin to a child learning that the stove top can be hot: every time the child touches the hot stove they get a negative reinforcement signal. They soon learn not to touch stoves. ■

The Kohonen map

Another type of neural network that works on entirely different principles is the Kohonen map. For example, instead of showing the network data chairs and tables and telling it which is which, we just show it everything and ask it to sort it out itself (unsupervised learning). The network clusters the data according to size, whether the furniture has arms or a back or not, and so on. If all goes as planned, the data will naturally cluster into 'chair-like' and 'table-like'. This is the basis of the self-organising Kohonen map. ■

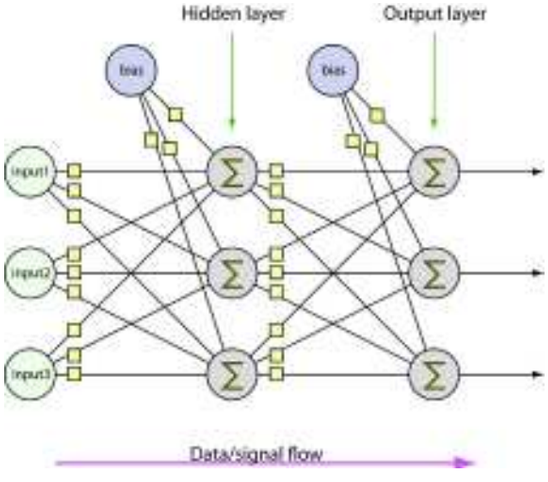
and it should produce the correct outputs for the four possible combinations of input.

The only way of doing this is to train the perceptron. First thing we need is a set of inputs and their expected outputs. For our simple example, we have four training sets: 1 and 1 gives 1, 0 and 1 gives 0, 0 and 0 gives 0, and 1 and 0 gives 0.

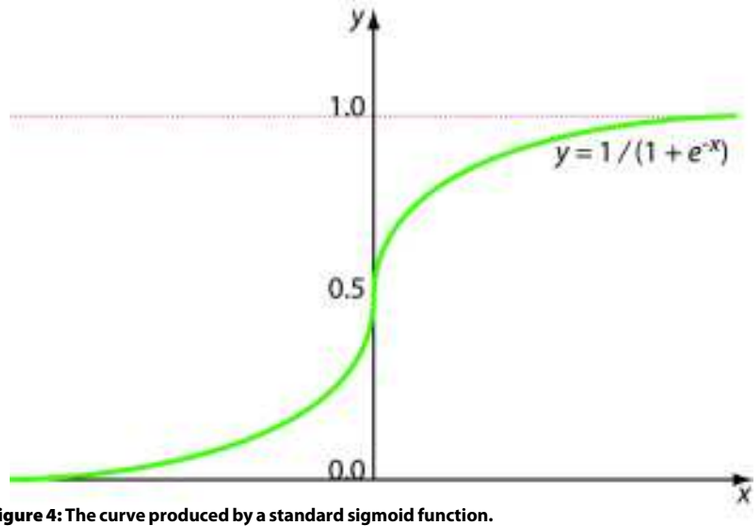
We set all weights to zero. Note that this perceptron will produce the right answer for the last three training sets automatically. The first set will produce an error (it should produce 1, but gives 0, an error of 1). What we do now is to modify the weights to take account of the error. We make use of a new constant called the learning rate (a number between 0 and 1) and modify each weight to add a term that's proportional to the error value, the learning rate and its input value. Start off with a high rate (say, 0.8).

We then let the perceptron learn using its training materials until the weights stabilise. If the weights don't converge after a few iterations, the perceptron is possibly oscillating around the solution, so it's best to reduce the learning rate. If the weights never converge, then the function being modelled by the perceptron cannot be recognised.

After Minsky and Papert showed that a single perceptron couldn't solve some simple patterns, research stagnated. Eventually, efforts shifted to studying a multilayer system instead. The first such system was called the feedforward neural network.



▲ **Figure 3: A feedforward network with three hidden perceptrons.**



▲ **Figure 4: The curve produced by a standard sigmoid function.**

In a multilayer system of perceptrons, there are at least three layers: the input layer, the hidden layer and the output layer. The latter two layers are the perceptrons. Figure 3 shows an idealised view of such a network. Notice that the data or signals travel one way, from the input layer to the output layer, hence the term feedforward. There are no cycles here.

The hidden layer is shown here to have three perceptrons, but this is by no means a fixed number. Indeed, the number of hidden perceptrons is yet another 'knob' to twiddle to tune the neural network (the weights being the only knobs so far). The number of output perceptrons is a function of the pattern you're trying to recognise, so if you were trying to perform OCR on digits, you might have 10 output perceptrons, one for each digit. Notice that all of the input signals feed into all of the perceptrons in the hidden layer, and all the outputs from the perceptrons in the hidden layer feed into the perceptrons in the output layer. If an input for a perceptron is not needed, the weight will be set to zero.

Finer tuning

The big issue with this neural network is in training it. The most successful algorithm devised is known as the back-propagation training algorithm, but it requires some changes. The first change is that the perceptron should output not just a 0 or a 1, but a floating-point value between 0 and 1. This will in turn require the perceptron to use a different activation function, one that's a curve instead of being a step function.

The functions used here are known as sigmoid functions. These are S-shaped functions with asymptotes at 0 and at 1. Figure 4 shows the standard one that's used: $f(x) = 1/(1+e^{-x})$, but others include the hyperbolic tangent (tanh) or the error function (erf). If a perceptron calculates a very large sum of its weighted inputs, it'll output a value close to 1; if the sum is very large and negative, it'll output a value close to 0; if it's close to 0, the perceptron will output a value around 0.5.

Once these changes have been made, the network is 'differentiable'; that is, it's possible to calculate gradients. The gradients we want to calculate are to help us change the weights due to a training error: a steep gradient for an input signal means a larger change in its weight, a more gradual gradient means a smaller change. The gradient also gives a direction (upwards or downwards), so we know whether to add or subtract the correction term.

And all this means that, despite the much greater complexity of a feedforward neural network, training it still only requires a few cycles. Obviously you have to pre-calculate a catalogue of training sets (so, for example, if you were creating a neural network to recognise the digits using OCR, you'd use as many different variants of the digits using all the fonts you could find), and those training sets would be fed into the network as often as needed until the weights converged.

Of course, this time around, you would have the extra knob to twiddle: the number of hidden perceptrons. Here there are no real guidelines apart from the more of them there are, the longer it will take to train the network, and you may not gain any more accuracy. Generally, though, you would aim for having at least as many hidden perceptrons as you have perceptrons in the output layer. ■

Genetic algorithms

The main method for training a neural network is back-propagation. For certain networks, a different algorithm may prove more efficient: a genetic one. Here we apply the standard genetic optimisation techniques (using chromosome fitness selection and introducing changes due to crossovers and mutations) to the weights in the network. After a few generations, the weights will converge. This type of training is of greater benefit for learning environments where the network must learn on the fly. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk