What are artificial neural networks?

Anders Krogh

Artificial neural networks have been applied to problems ranging from speech recognition to prediction of protein secondary structure, classification of cancers and gene prediction. How do they work and what might they be good for?

When it comes to tasks other than number crunching, the human brain possesses many advantages over a digital computer. We can quickly recognize a face, even when seen from the side in bad lighting in a room full of other objects. We can easily understand speech, even that of an unknown person in a noisy room. Despite years of focused research, computers are far from performing at this level. The brain is also remarkably robust; it does not stop working just because a few cells die. Compare this to a computer, which will not normally survive any degradation of the CPU (central processing unit). But perhaps the most fascinating aspect of the brain is that it can learn. No programming is necessary; we do not need a software upgrade, just because we want to learn to ride a bicycle.

The computations of the brain are done by a highly interconnected network of neurons, which communicate by sending electric pulses through the neural wiring consisting of axons, synapses and dendrites. In 1943, McCulloch and Pitts modeled a neuron as a switch that receives input from other neurons and, depending on the total weighted input, is either activated or remains inactive. The weight, by which an input from another cell is multiplied, corresponds to the strength of a synapse—the neural contacts between nerve cells. These weights can be both positive (excitatory) and negative (inhibitory). In the 1960s, it was shown that networks of such model neurons have properties similar to the brain: they can perform sophisticated pattern recognition, and they can function even if some of the neurons are destroyed. The demonstration, in particular by Rosenblatt, that simple networks of such model neurons called 'perceptrons' could learn from examples stimulated interest in the field, but after Minsky and Papert[1] showed that simple perceptrons could solve only the very limited class of linearly separable problems (see below), activity in the field diminished. Nonetheless, the error back-propagation method[2], which can make fairly complex networks of simple neurons learn from examples, showed that these networks could solve problems that were not linearly separable. NETtalk, an application of an artificial neural network for machine reading of text, was one of the first widely known applications[3], and many followed soon after. In the field of biology, the exact same type of network as in NETtalk was also applied to prediction of protein secondary structure[4]; in fact, some of the best predictors still use essentially the same method. Another big wave of interest in artificial neural networks started, and led to a fair deal of hype about magical learning and thinking machines. Some of the important early works are gathered in ref. 5.

Artificial neural networks are inspired by the early models of sensory processing by the brain. An artificial neural network can be created by simulating a network of model neurons in a computer. By applying algorithms that mimic the processes of real neurons, we can make the network 'learn' to solve many types of problems. A model neuron is referred to as a threshold unit and its function is illustrated in **Figure 1a**. It receives input from a number of other units or external sources, weighs each input and adds them up. If the total input is above a threshold, the output of the unit is one; otherwise it is zero. Therefore, the output changes from 0 to 1 when the total weighted sum of inputs is equal to the threshold. The points in input space satisfying this condition define a so-called hyperplane. In two dimensions, a hyperplane is a line, whereas in three dimensions, it is a normal plane. Points on one side of the hyperplane are classified as 0 and those on the other side as 1. It means that a classification problem can be solved by a threshold unit if the two classes can be separated by a hyperplane. Such problems, as illustrated in three dimensions in **Figure 1b**, are said to be linearly separable.

Learning

If the classification problem is separable, we still need a way to set the weights and the threshold, such that the threshold unit correctly solves the classification problem. This can be done in an iterative manner by presenting examples with known classifications, one after another. This process is called learning or training, because it resembles the process we go through when learning something. Simulation of learning by a computer involves making small changes in the weights and the threshold each time a new example is presented in such a way that the classification is improved. The training can be implemented by various different algorithms, one of which will be outlined below.

During training, the hyperplane moves around until it finds its correct position in space, after which it will not change so much. This is nicely illustrated in two dimensions by the program Neural Java (http://lcn.epfl.ch/tutorial/english/index.html); follow the link "Adaline, Perceptron and Backpropagation," use red and blue dots to represent two classes and select "play."

Let us consider an example problem, for which an artificial neural network is readily applicable. Of two classes of cancer, only one

Anders Krogh is at the Bioinformatics Centre, Department of Biology and Biotech Research and Innovation Centre, University of Copenhagen, Ole Maaloes Vej 5, DK-2200 Copenhagen, Denmark.
e-mail: krogh@binf.ku.dk

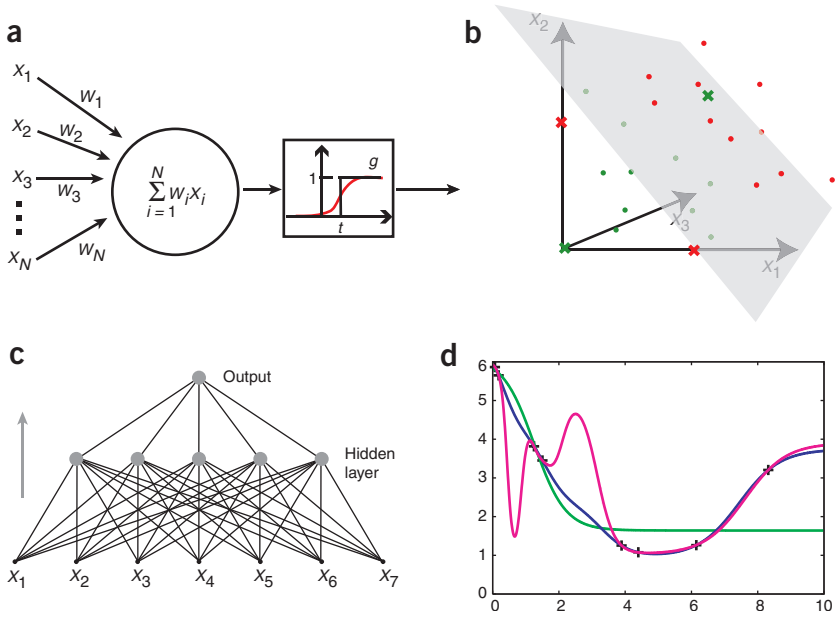

Figure 1 Artificial neural networks. (**a**) Graphical representation of the McCulloch-Pitts model neuron or threshold unit. The threshold unit receives input from *N* other units or external sources, numbered from 1 to *N*. Input *i* is called x_i and the associated weight is called w_i. The total input to a unit is the weighted sum over all inputs, $\sum_{i=1}^{N} w_i x_i = w_1 x_1 + w_2 x_2 + \ldots + w_N x_N$. If this were below a threshold *t*, the output of the unit would be 1 and 0 otherwise. Thus, the output can be expressed as $g\left(\sum_{i=1}^{N} w_i x_i - t\right)$, where *g* is the step function, which is 0 when the argument is negative and 1 when the argument is nonnegative (the actual value at zero is unimportant; here, we chose 1). The so-called transfer function, *g*, can also be a continuous sigmoid as illustrated by the red curve. (**b**) Linear separability. In three dimensions, a threshold unit can classify points that can be separated by a plane. Each dot represents input values x_1, x_2 and x_3 to a threshold unit. Green dots correspond to data points of class 0 and red dots to class 1. The green and red crosses illustrate the 'exclusive or' function—it is not possible to find a plane (or a line in the x_1, x_2 plane) that separates the green dots from the red dots. (**c**) Feed-forward network. The network shown takes seven inputs, has five units in the hidden layer and one output. It is said to be a two-layer network because the input layer does not perform any computations and is not counted. (**d**) Over-fitting. The eight points shown by plusses lie on a parabola (apart from a bit of 'experimental' noise). They were used to train three different neural networks. The networks all take an *x* value as input (one input) and are trained with a *y* value as desired output. As expected, a network with just one hidden unit (green) does not do a very good job. A network with 10 hidden units (blue) approximates the underlying function remarkably well. The last network with 20 hidden units (purple) over-fit the data; the training points are learned perfectly, but for some of the intermediate regions the network is overly creative.

responds to a certain treatment. As there is no simple biomarker to discriminate the two, you decide to try to use gene expression measurements of tumor samples to classify them. Assume you measure gene expression values for 20 different genes in 50 tumors of class 0 (nonresponsive) and 50 of class 1 (responsive). On the basis of these data, you train a threshold unit that takes an array of 20 gene expression values as input and gives 0 or 1 as output for the two classes, respectively. If the data are linearly separable, the threshold unit will classify the training data correctly.

But many classification problems are not linearly separable. We can separate the classes in such nonlinear problems by introducing more hyperplanes; that is, by introducing more than one threshold unit. This is usually done by adding an extra (hidden) layer of threshold units each of which does a partial classification of the input and sends its output to a final layer, which assembles the partial classifications to the final classification (**Fig. 1c**). Such a network is called a multi-layer perceptron or a feed-forward network. Feed-forward neural networks can also be used for regression problems, which require continuous outputs, as opposed to binary outputs (0 and 1). By replacing the step function with a continuous function, the neural network outputs a real number. Often a 'sigmoid' function—a soft version of the threshold function—is used (**Fig. 1a**). The sigmoid function can also be used for classification problems by interpreting an output below 0.5 as class 0 and an output above 0.5 as class 1; often it also makes sense to interpret the output as the probability of class 1.

In the above example, one could, for instance, have a situation where class 1 is characterized by either a highly expressed gene 1 and a silent gene 2 or a silent gene 1 and a highly expressed gene 2; if neither or both of the genes are expressed, it is a class 0 tumor. This corresponds to the 'exclusive or' function from logic, and it is the canonical example of a nonlinearly separable function (**Fig. 1b**). In this case, it would be necessary to use a multi-layer network to classify the tumors.

Back-propagation

The previously mentioned back-propagation learning algorithm works for feed-forward networks with continuous output. Training starts by setting all the weights in the network to small random numbers. Now, for each input example the network gives an output, which starts randomly. We measure the squared difference between this output and the desired output—the correct class or value. The sum of all these numbers over all training examples is called the total error of the network. If this number was zero, the network would be perfect, and the smaller the error, the better the network.

By choosing the weights that minimize the total error, one can obtain the neural network that best solves the problem at hand. This is the same as linear regression, where the two parameters characterizing the line are chosen such that the sum of squared differences between the line and the data points is minimal. This can be done analytically in linear regression, but there is no analytical solution in a feed-forward neural network with hidden units. In back-propagation, the weights and thresholds are changed each time an example is presented, such that the error gradually becomes smaller. This is repeated, often hundreds of times, until the error no longer changes. An illustration can be found at the Neural Java site above by following the link "Multi-layer Perceptron (with neuron outputs in {0;1})."

In back-propagation, a numerical optimization technique called gradient descent makes the math particularly simple; the form of the equations gave rise to the name of this method. There are some learning parameters (called learning rate and momentum) that need tuning when using back-propagation, and there are other problems to consider. For instance, gradient descent is not guaranteed to find the global minimum of the error, so the result of the training depends on the initial values of the weights. However, one problem overshadows the others: that of over-fitting.

Over-fitting occurs when the network has too many parameters to be learned from

the number of examples available, that is, when a few points are fitted with a function with too many free parameters (**Fig. 1d**). Although this is true for any method for classification or regression, neural networks seem especially prone to overparameterization. For instance, a network with 10 hidden units for solving our example problem would have 221 parameters: 20 weights and a threshold for the 10 hidden units and 10 weights and a threshold for the output unit. This is too many parameters to be learned from 100 examples. A network that overfits the training data is unlikely to generalize well to inputs that are not in the training data. There are many ways to limit over-fitting (apart from simply making small networks), but the most common include averaging over several networks, regularization and using methods from Bayesian statistics[6].

To estimate the generalization performance of the neural network, one needs to test it on independent data, which have not been used to train the network. This is usually done by cross-validation, where the data set is split into, for example, ten sets of equal size. The network is then trained on nine sets and tested on the tenth, and this is repeated ten times, so all the sets are used for testing. This gives an estimate of the generalization ability of the network; that is, its ability to classify inputs that it was not trained on. To get an unbiased estimate, it is very important that the individual sets do not contain examples that are very similar.

Extensions and applications

Both the simple perceptron with a single unit and the multi-layer network with multiple units can easily be generalized to prediction of more than two classes by just adding more output units. Any classification problem can be coded into a set of binary outputs. In the above example, we could, for instance, imagine that there are three different treatments, and for a given tumor we may want to know which of the treatments it responds to. This could be solved using three output units—one for each treatment—which are connected to the same hidden units.

Neural networks have been applied to many interesting problems in different areas of science, medicine and engineering and in some cases, they provide state-of-the-art solutions. Neural networks have sometimes been used haphazardly for problems where simpler methods would probably have given better results, giving them a somewhat poor reputation among some researchers.

There are other types of neural networks than the ones described here, such as Boltzman machines, unsupervised networks and Kohonen nets. Support vector machines[7] are closely related to neural networks. To read more, I suggest the books by Chris Bishop[6,8], a rather old book I coauthored[9] or the book by Duda *et al.*[10]. There are numerous programs to use for making artificial neural networks trained with your own data. These include extensions or plug-ins to Excel, Matlab and R (http://www.r-project.org/) as well as code libraries and large commercial packages. The FANN library (http://leenissen.dk/fann/) is recommended for serious applications. It is open source and written in the C programming language, but can be called from, for example, Perl and Python programs.

1. Minsky, M.L. & Papert, S.A. *Perceptrons* (MIT Press, Cambridge, 1969).
2. Rumelhart, D.E., Hinton, G.E. & Williams, R.J. *Nature* **323**, 533–536 (1986).
3. Sejnowski, T.J. & Rosenberg, C.R. *Complex Systems* **1**, 145–168 (1987).
4. Qian, N. & Sejnowski, T.J. *J. Mol. Biol.* **202**, 865–884 (1988).
5. Anderson, J.A. & Rosenfeld, E. (eds). *Neurocomputing: Foundations of Research* (MIT Press, Cambridge, 1988).
6. Bishop, C.M. *Neural Networks for Pattern Recognition* (Oxford University Press, Oxford, 1995).
7. Noble, W.S. *Nat. Biotechnol.* **24**, 1565–1567 (2006).
8. Bishop, C.M. *Pattern Recognition and Machine Learning* (Springer, New York, 2006).
9. Hertz, J.A., Krogh, A. & Palmer, R. *Introduction to the Theory of Neural Computation* (Addison-Wesley, Redwood City, 1991).
10. Duda, R.O., Hart, P.E. & Stork, D.G. *Pattern Classification* (Wiley Interscience, New York, 2000).