Artificial Neural Networks

Martin Anthony

Abstract

'Artificial neural networks' are machines (or models of computation) based loosely on the ways in which the brain is believed to work. In this chapter, we discuss some links between graph theory and artificial neural networks. We describe how some combinatorial optimisation tasks may be approached by using a type of artificial neural network known as a Boltzmann machine. We then focus on 'learning' in feedforward artificial neural networks, explaining how the graph structure of a network and the hardness of graph-colouring quantify the complexity of learning.

1. Introduction

There has recently been intense, and fast-growing, interest in 'artificial neural networks'. These are machines (or models of computation) based loosely on the ways in which the brain is believed to work. Neurobiologists are interested in using these machines as a means of modelling biological brains, but much of the impetus comes from their applications. For example, engineers wish to create machines that can perform 'cognitive' tasks, such as speech recognition, and economists are interested in financial time series prediction using such machines. Inevitably, there is a certain amount of hype associated with the subject, particularly in relation to neurobiological modelling.

Here, we take what may be called an 'engineering approach' to artificial neural networks. In such an approach, one is not concerned with the issue of whether artificial neural networks are plausible models of real neural networks, but rather, we start from the fact that they exist and are being used extensively. The questions that mathematicians might then ask include:

- what can artificial neural networks do for us?

- what can mathematics say about artificial neural networks that might be of interest or benefit to practitioners?

In this chapter, we discuss some connections between the theory of artificial neural networks and graphs. In one answer to the first of the above questions, we explain how certain types of artificial neural network may be useful for solving combinatorial optimization problems in graph theory. We then turn our attention to a theory of 'learning' in artificial neural networks, where the graph structure of the network and the hardness of graph-colouring tell us something about the complexity of learning.

2. Artificial neural networks

It appears that one reason why the human brain is so powerful is the sheer complexity of connections between neurons. In computer science parlance, the brain exhibits huge parallelism, with each neuron connected to many other neurons. This has been reflected in the design of artificial neural networks. One advantage of such parallelism is that the resulting network is *robust*: in a serial computer, a single fault can make computation impossible, whereas in a system with a high degree of parallelism and many computation paths, a small number of faults may be tolerated with little or no upset to the computation.

Two types of neural network are discussed here: Boltzmann machines and feedforward networks. Loosely speaking, we can say that an artificial neural network consists of a directed graph with *computation units*) situated at the vertices and weights on the arcs (directed edges). Some of the computation nodes may be distinguished as *input nodes*, which receive signals from the outside world, and some as *output nodes*. The nodes have *activations* and their activation influences those of their neighbours, either stochastically as in the Boltzmann machine, or deterministically, as in feedforward networks. The degree to which the activation of one node influences those of its neighbours is determined by the weights on the arcs. The process of 'learning' is the adjustment of the weights.

3. Boltzmann machines

In a Boltzmann machine, a type of *stochastic* artificial neural network, the underlying digraph has symmetric connections: if $(i, j) \in A$, then $(j, i) \in A$. Furthermore, the weights are always constrained so that $w_{(i,j)} = w_{(j,i)} = w_{ij}$. A Boltzmann machine M may therefore be described as an undirected graph (which may have loops) with weighted edges and stochastic computation units at the vertices. More precisely, M is a pair (G, Ω), where $G = (V, E)$ is a graph with, say, n vertices and m edges, and $\Omega \subseteq \mathbf{R}^m \times \{0, 1\}^n$ is a set of allowable *states*. For $\omega = (w_{e_1}, w_{e_2}, \ldots w_{e_m}, S_1, \ldots, S_n) \in \Omega$, the vector $w = (w_{e_1}, w_{e_2}, \ldots w_{e_m})$ describes the weights assigned to the edges $e_1, e_2, \ldots, e_m$ of the graph, and the tail of ω, $(S_1, S_2, \ldots, S_n)$, describes which of the vertices (units) $1, 2, \ldots, n$ is 'on': vertex i is 'on' in state ω if and only if $S_i = 1$. The *consensus function* of the Boltzmann machine M is the function $C : \Omega \to \mathbf{R}$ given by

$$C(\omega) = \sum_{ij \in E} w_{ij} S_i S_j.$$

Computation proceeds in the machine in a stochastic manner in such a way as to increase consensus. Thus, if w_{ij} is positive, there is a tendency for units i and j to be either both on or both off, while if the weight is negative, there is a tendency for them to have different activations. When a weight is positive, we refer to it as *excitatory* and, when negative, *inhibitory*.

We now describe how the state of the network evolves when the weights are fixed. This is, of course, a very restricted analysis since, in general, the weights may also change through 'learning'; see [1,14]. However, for the applications we have in mind, the weights will be determined explicitly by an instance of a combinatorial optimization problem.

If i is a vertex and ω a state, $\omega[i \to \bar{i}]$ is the state obtained from ω by changing S_i to $1 - S_i$; that is, by 'flipping' the activation of i. If such a flip is made, then the resulting change in the consensus function is

$$\Delta C_\omega(i) = C(\omega[i \to \bar{i}]) - C(\omega).$$

If this is fairly large, it is advantageous to flip the activation of i, and if it is negative, with large absolute value it is disadvantageous to do so. The decision on whether to flip the activation of unit i is made stochastically, based on $\Delta C_\omega(i)$. The simplest model of computation in a Boltzmann machine is the sequential model. Here, the machine has an internal clock, and on the tth tick of the clock, a vertex i_t is chosen, uniformly at random. The activation of i_t is then flipped, and the state changed to $\omega[i_t \to \bar{i}_t]$, with probability

$$\text{Prob}\,(\omega \to \omega[i_t \to \bar{i}_t]) = \frac{1}{1 + \exp(-\beta \Delta C_\omega(i_t))},$$

for some constant $\beta > 0$. A more complicated procedure is that in which the machine's computations are in parallel. Then, at time t, a subset S^t of the vertices is chosen according to some probability distribution on the subsets, and for each $i \in S^t$, a decision is made, as described above, whether to flip the activation of that unit. It should be noted that although this may involve making a large number of such decisions simultaneously, the computations involved in this procedure are local, since $\Delta C_\omega(i)$ depends only on the activations of the units j adjacent to i and on the weights w_{ij}. The result is some state ω' which is obtained from ω by flipping some of the S_i, for $i \in S^t$. It can be proved (see Aarts and Korst [1]) that the sequential mode of computation results in a stationary distribution on the set of all states, in which the probability that the state of the machine is ω is proportional to $\exp(\beta C(\omega))$. The same conclusion holds if the computations are *synchronous, with limited parallelism.* This is the special case of parallel computation, in which only independent sets of vertices are generated, and each independent set is generated with equal probability. In both cases, in the stationary distribution, states of high consensus are more likely.

The mysterious parameter β has an interesting interpretation. Its reciprocal $T = 1/\beta$ is often called *temperature*. There are indications [1] that the process of *simulated annealing*, whereby the temperature is slowly decreased as computation proceeds (so that β is no longer constant, but increases with time), may be helpful. Aarts and Korst [1] show that as $T \to 0$ and as time tends to infinity, the limiting stationary distribution of states is uniform over all states that maximize the consensus function.

4. Optimization with Boltzmann machines

A number of combinatorial optimization problems can be realized as the problem of maximizing consensus in Boltzmann machines. The book by Aarts and Korst [1] contains a full discussion of this approach; here, we shall describe the general approach and give two examples.

General approach

Suppose we have a combinatorial optimization problem. It is then often possible to construct a Boltzmann machine with fixed weights determined by the instance of the problem so that maximizing consensus in the machine is equivalent to solving the optimization problem. A general approach, as described in [1], is as follows. First, phrase the optimization problem as a $\{0,1\}$-valued linear programming problem. Secondly, construct a Boltzmann machine with vertices (nodes) corresponding to the variables in the linear programming problem. Thirdly, define the edges and weights of the machine in such a way that the following *correspondence conditions* hold:

• local maxima of the resulting consensus function correspond to feasible solutions of the optimization problem;

• If $\mathcal{S}_1, \mathcal{S}_2$ are two feasible solutions of the optimization problem, with $\mathcal{S}_1$ better than $\mathcal{S}_2$, then the consensus of the state corresponding to $\mathcal{S}_1$ is higher than that corresponding to $\mathcal{S}_2$.

Note that we may always take the underlying graph of the Boltzmann machine to be complete, since setting a weight to zero is equivalent to omitting an edge. If the procedure just described can be carried out, then, in theory, one way to attempt to solve the optimization problem is to construct the Boltzmann machine with these properties and let it evolve, perhaps decreasing the temperature parameter β^{-1} as time progresses. We have already mentioned that such an approach may be promising, since the machine has certain convergence properties. We should remark, however, that even when we know from the theory that the state of the machine will converge to one maximizing consensus, it may not do so feasibly fast.

Maximum cuts

To illustrate the Boltzmann machine approach, we start with the problem of finding a maximum cut in a weighted graph. Given a graph $G = (V, E)$ of order n having weighted edges, the problem is to determine a partition of the vertex-set into subsets V_1 and V_2 in such a way that the total sum of the weights of the edges that join vertices in the two sets is maximum.

This may easily be phrased as a $\{0,1\}$ linear programming problem: introduce a variable x_i for each vertex $i \in V$, and take $x_i = 1$ to mean that $i \in V_2$ and $x_i = 0$ to mean that

$i \in V_1$. Then the problem is to maximize the objective function

$$F = \sum_{i=1}^{n} \sum_{j=i+1}^{n} W_{ij} \left((1 - x_i)x_j + (1 - x_j)x_i \right),$$

where the W_{ij} are the weights on the edges of the graph. Our Boltzmann machine will have as its underlying graph the instance graph together with loops on all the vertices. We assign weights as follows: for $ij \in E$, set the weight w_{ij} on this edge in the Boltzmann machine to be $-2W_{ij}$, where W_{ij} is the corresponding weight in the original graph. For $i \in V$, set the weight w_{ii} on the loop, to be $\sum_{j=1}^{n} W_{ij}$. The resulting Boltzmann machine then satisfies the correspondence conditions.

The travelling salesman problem

An instance of the travelling salesman problem is a weighted graph $G = (V, E)$, where the vertices reprsent cities and the weight d_{ij} on edge ij is the distance from i to j. The aim is to find a a Hamiltonian cycle of minimum total length. A number of neural approaches to this problem have been made. We describe here a fairly simple Boltzmann machine implementation due to Garfinkel [11]. Following the general approach outlined above, we first describe the problem as a linear programming problem. Suppose that the vertex-set of the graph is $V = \{1, 2, \ldots, n\}$ and, for i and a between 1 and n, introduce a variable x_{ia}, which, for a particular tour, equals 1 if and only if vertex i is the ath vertex in the tour. For i, j, a, b between 1 and n, let

$$u_{ijab} = \begin{cases} d_{ij} & \text{if } a \equiv b + 1 \pmod{n}; \\ 0 & \text{otherwise.} \end{cases}$$

The travelling salesman problem is then equivalent to minimizing

$$F = \sum_{i,j,a,b} u_{ijab} x_{ia} x_{jb}$$

subject to the constraints

$$\sum_{i=1}^{n} x_{ia} = 1 \;\; (1 \le a \le n), \quad \text{and} \quad \sum_{a=1}^{n} x_{ia} = 1 \;\; (1 \le i \le n).$$

We now construct a Boltzmann machine whose consensus function satisfies the properties described at the beginning of this section. It has n^2 vertices, one for each pair (i, a), corresponding to x_{ia}. Define the weights as follows:

$$\begin{aligned}
w_{(i,a)(i,a)} &= 1 + \max\left\{ d_{ik} + d_{il} : l \neq k \right\}, \\
w_{(i,a)(j,b)} &= -d_{ij}, \;\text{ for } i \neq j, \text{ and } a \equiv b + 1 \pmod{n}, \\
w_{(i,a)(i,b)} &= -1 - \min(w_{(i,a)(i,a)}, w_{(i,b)(i,b)}), \;\text{ for } a \neq b, \\
w_{(i,a)(j,a)} &= -1 - \min(w_{(i,a)(i,a)}, w_{(j,a)(j,a)}), \;\text{ for } i \neq j.
\end{aligned}$$

Then the correspondence conditions are satisfied, and the Boltzmann machine will 'solve' the travelling salesman problem, although not generally in polynomial time.

5. Feedforward networks

In a feedforward network, the underlying directed graph $G = (V, A)$ is *acyclic*; that is, there are no directed cycles. For the networks considered here, there is a specified subset of *input nodes* $\mathcal{I}$ and a single output node $z \notin \mathcal{I}$. The underlying idea is that all nodes receive and transmit signals; the input nodes receive their signals from the outside world and the output node transmits a signal to the outside world, while all other nodes receive and transmit along the relevant arcs of the directed graph. Each arc ij has a weight w_{ij} that represents the strength of the connection between the nodes r and s. A positive weight corresponds to an 'excitatory' connection, a negative one to an 'inhibitory' connection. All nodes except the input nodes are 'active' in that they transmit a signal that is a predetermined function of the signals they receive. There is an *activation function* f_r for each non-input computation node r and the activity of such a node is specified in two stages. First, the signals arriving at r are aggregated by taking their weighted sum according to the connection strengths on the arcs with terminal node r, and then the function f_r of this value is computed. Thus, the action of the entire network may be described in terms of two functions $p: V \to \mathbf{R}$ and $q: V \to \mathbf{R}$, that represent the received and transmitted signals respectively. It is convenient to assume that one of the input nodes is a *bias node*, for which the applied signal is always 1. Suppose the input nodes are labelled $j_0, j_1, j_2, \ldots, j_n$, where j_0 is the bias node. We assume that a vector of real-valued signals $y = (1, y_1, y_2, \ldots, y_n)$ is applied externally to the input nodes, and $p(j_k) = q(j_k) = y_k$ for $k = 1, 2, \ldots, n$. (Note that the input vector has 1 as first entry; this is the bias signal.) For every other node l, the received and transmitted signals are defined as follows.

$$p(l) = \sum_{\{i|(i,l)\in A\}} q(i)w_{il}, \quad \text{and} \quad q(l) = f_l(p(l)).$$

The output is the value $q(z)$ transmitted by the output node z. Since the underlying digraph is acyclic, it is possible to partition the nodes into *layers* $l_1, l_2, \ldots, l_k$, in such a way that the nodes in layer l_1 are the input nodes, layer k consists solely of the output node, and all arcs go from lower to higher layers. Computation then proceeds upwards through the layers. Sometimes the activation functions f_r are chosen to be 'sigmoidal' functions; these are smooth, non-decreasing functions mapping into $[0, 1]$. A popular choice is the *standard sigmoid* function,

$$\sigma(x) = \frac{1}{(1 + \exp(-x))}.$$

Often, however, the *linear threshold* function is taken to be the activation function of each non-input node. This is the function $\mathcal{H} : \mathbf{R} \to \{0, 1\}$ such that $\mathcal{H}(x) = 1$ if $x \geq 0$ and

$\mathcal{H}(x) = 0$ if $x < 0$. A feedforward network of this type is known as a *linear threshold network*. A network which consists of a number of input nodes and just one other node, a linear threshold output node, is a special type of *perceptron*, known as the *simple perceptron* (see [20,18]). This is illustrated in Figure 1.

Figure 1 Figure 2

A feedforward linear threshold network is essentially a composition of simple perceptrons. As an example, consider the threshold network illustrated in Figure 2. It is easy to verify that if f is the function computed by this network in the state determined by the weights shown, then $f(101) = f(110) = 1$ and $f(100) = f(111) = 0$. Thus, if we restrict attention to $\{0, 1\}$-valued inputs, this network computes the exclusive-or function of its two non-bias inputs.

6. Supervised learning in feedforward networks

PAC learning

One of the main reasons why neural networks have proven so attractive is that they are, in a sense, capable of 'learning'. The use of such anthropomorphic language might be considered controversial, but in a mathematical or engineering approach to neural networks, 'learning' simply means changing the weights of the network in response to some input data. When 'successful' or 'convergent' learning is possible, there is no need to program the network explicitly to perform a particular task; in other words, we need not know in advance how to set the weights. The neural network adjusts its weights according to a *learning algorithm*, in response to some classified training examples, with (roughly speaking) the state of the network converging to the 'correct' one. In this sense, the neural network 'learns from experience'.

In any model of supervised learning, it is assumed that there is some 'target function' t, which is the function to be learned. The target function is to be thought of as the 'correct' function we should like the network to compute. It simplifies matters greatly if we assume that there *is* a correct function t, and that this function can be computed by the network with some set of weight assignments. However, the basic PAC model has been extended to deal with situations in which neither of these assumptions can be made (see [2], for example). A *labelled example* for the target function t is a pair $(x, t(x))$, where x is an input pattern to the network; for instance, if the neural network has n input nodes accepting real inputs, then $x \in \mathbf{R}^n$. The network is given a *training sample*, a sequence of such labelled examples: this constitutes its 'experience'. In response to this, its weights are altered by applying a learning algorithm. More formally, suppose that the set of all possible examples is $X = \mathbf{R}^n$ or $X = \{0,1\}^n$, where n is the number of inputs to the network, and that the target function t can be computed by the neural network in some state. A *training sample for t* of length m is an element $\mathbf{s}$ of $(X \times \{0,1\})^m$, of the form

$$\mathbf{s} = ((x_1, t(x_1)), (x_2, t(x_2)), \ldots, (x_m, t(x_m)))\,.$$

We denote the set of all training samples of length m for t by $S(m, t)$. A *learning algorithm* accepts as input the training sample $\mathbf{s}$ and alters the state of the network in some way in response to the information provided by the sample. We should like the function $L(\mathbf{s})$ that is computed by the network after 'learning' to be an approximation to the target function, or 'closer' to the target function than the function computed before learning.

A large number of learning algorithms are currently in use. One of the most popular is the *back-propagation* algorithm for feedforward networks with sigmoidal activation functions (see, for example, [14]). We describe here the very simple perceptron learning algorithm, devised by Rosenblatt [20].

For any *learning constant* $\nu > 0$, the *perceptron learning algorithm* L_ν acts on the training sample 'on-line' in the following manner. The algorithm L_ν maintains at each stage a *current state* of the network, $\mathbf{w} = (w_0, w_1, \ldots, w_n)$. Here, w_0 is the weight on the arc from the bias input node to the output node, and $w_1, w_2, \ldots, w_n$ are the weights on the arcs from the other input nodes to the output node. (See Figure 1.) This current state is updated on the basis of a labelled example $(x, t(x))$. (The initial state is, for example, that in which all weights are 0.) Suppose the current state is $\mathbf{w}$ and that a labelled example $(x, t(x))$ is presented. Denote by $h_\mathbf{w}$ the function computed by the network in its current state. Then the algorithm forms the new state $\mathbf{w}'$, where

$$\mathbf{w}' = \mathbf{w} + \nu\,(t(x) - h_\mathbf{w}(x))\,x.$$

This learning algorithm, which makes an incremental adjustment of the weights in the event of misclassification, is an example of a 'Hebbian' learning algorithm [13]. The *perceptron convergence theorem* [20,18] asserts that the perceptron learning algorithm aids convergence toward the target function: no matter how many examples are presented, the algorithm makes only a finite number of changes, or updates (provided that ν, which can be a function of n, is small enough).

In order to analyze learning in feedforward networks, we first need a mathematical framework in which to define the goal of a successful learning algorithm. We briefly describe the basic 'probably approximately correct' (PAC) model of learning introduced by Valiant [21], as it applies to feedforward networks in which there is a single output node, giving as output either 0 or 1. (A more detailed treatment may be found in [3] or [2].) A fundamental assumption of the PAC model is that the network receives training samples

$$\mathbf{s} = ((x_1, t(x_1)), (x_2, t(x_2)), \ldots, (x_m, t(x_m)))$$

in which $x_1, x_2, \ldots, x_m$ are chosen independently and at random according to some fixed (but unknown) probability distribution on the set of all examples.

In order to assess how effective a learning algorithm is, we need some measure of how close $L(\mathbf{s})$ is to t. Since there is assumed to be some probability distribution P on the set of all examples, we may define the *error*, $\mathrm{er}_P(h,t)$, of a function h (with respect to t) to be the P-probability that a randomly chosen example is classified incorrectly by h; that is,

$$\mathrm{er}_P(h,t) = P(\{x \in X : h(x) \neq t(x)\}).$$

The aim is to ensure that the error of $L(\mathbf{s})$ is 'usually small'. Since each of the m examples in the training sample is drawn randomly and independently according to P, the sample vector $\mathbf{x}$ is drawn randomly from X^m according to the product probability distribution P^m. Thus, we want it to be true that with high P^m-probability the sample $\mathbf{s}$ arising from $\mathbf{x}$ is such that the function $L(\mathbf{s})$ computed after training has small error with respect to t. This is a reasonable goal: some samples will be 'unrepresentative', but such samples will have low probability of being presented. This leads us to the following formal definition of PAC learning.

Definition *The learning algorithm L is a* PAC learning algorithm *for the network if for any given $\delta, \epsilon > 0$ there is a sample length $m_L(\delta, \epsilon)$ such that for all target functions t computable by the network and for all probability distributions P on the set of examples,*

$$P^m\left(\{\mathbf{s} \in S(m,t) : \mathrm{er}_P(L(\mathbf{s}),t) > \epsilon\}\right) < \delta.$$

whenever $m \geq m_L(\delta, \epsilon)$.

In other words, provided the sample has length at least $m_L(\delta, \epsilon)$ then it is 'probably' the case that after training on that sample, the function computed by the network is 'approximately' correct. (We should note that the product probability distribution P^m is really defined not on subsets of $S(m,t)$ but on sets of vectors in X^m. However, this abuse of notation is convenient: for a fixed t, there is a clear one-to-one correspondence between vectors $\mathbf{x} \in X^m$ and training samples $\mathbf{s} \in S(m,t)$.)

Note that the probability distribution P occurs twice in the definition: both in the requirement that the P^m-probability of a sample be small and also through the fact that the error of $L(\mathbf{s})$ is measured with reference to P. The crucial feature of the definition is that we require that the sample length $m_L(\delta, \epsilon)$ be independent of P and of t. It is not immediately clear that this is possible, but the following informal arguments explain why it may be. If a particular example has not been seen in a large sample $\mathbf{s}$, the chances are that this example has low probability, and therefore misclassification of that example contributes little to the error of the function $L(\mathbf{s})$. In other words, the penalty paid for misclassification of a particular example is its probability, and, very loosely speaking, the two occurrences of the probability distribution in the definition can therefore 'balance' each other.

An important property that a learning algorithm might have is *consistency.* We say that the learning algorithm L is *consistent* if, given any training sample

$$\mathbf{s} = ((x_1, t(x_1)), \ldots, (x_m, t(x_m))),$$

the functions $L(\mathbf{s})$ and t agree on x_i, for each $i = 1, 2, \ldots, m$. The perceptron algorithm described above is not generally a consistent algorithm. However, it is easy to construct a consistent learning algorithm L from L_ν: given a sample $\mathbf{s} = ((x_1, t(x_1), \ldots, (x_m, t(x_m))$ of examples, L acts on $\mathbf{s}$ by applying L_ν repeatedly, cycling through x_1 to x_m in turn, until no updates are made in a complete cycle.

The VC-dimension and the underlying graph

The problem of PAC learning can be addressed by means of a combinatorial parameter known as the Vapnik-Chervonenkis dimension (abbreviated to VC-dimension). Suppose that $\mathcal{N}$ is a feedforward neural network that outputs 0 or 1. We say that a set T of examples is *shattered* by $\mathcal{N}$ if for each of the $2^{|T|}$ possible ways of dividing T into two disjoint sets T_1 and T_0, there is *some* function f computable by $\mathcal{N}$ such that $f(x) = 1$ if $x \in T_1$ and $f(x) = 0$ if $x \in T_0$. The VC-*dimension* of $\mathcal{N}$, denoted $\dim_{\mathrm{VC}}(\mathcal{N})$, is defined to be the largest size of a set of examples shattered by $\mathcal{N}$. The VC-dimension may be thought of as a measure of the 'expressive power' of the network. Vapnik and Chervonenkis [22] defined this parameter (in a more general context) in studying the uniform convergence of relative frequencies to probabilities.

The VC-dimension characterizes fairly precisely the size of training sample which should be used for effective PAC-learning. The following result is due to Blumer *et al.* [8] and Ehrenfeucht *et al.* [10].

Theorem 6.1 *If a feedforward network $\mathcal{N}$ has finite* VC-*dimension $d \geq 1$, then any consistent learning algorithm L for $\mathcal{N}$ is a* PAC *learning algorithm. Moreover, there is a constant*

c_1 such that

$$\frac{c_1}{\epsilon}\left(d\ln\left(\frac{1}{\epsilon}\right)+\ln\left(\frac{1}{\delta}\right)\right)$$

is a sufficient sample length $m_L(\delta,\epsilon)$ for any such algorithm. On the other hand, there is another constant c_2 such that for any PAC *learning algorithm L for $\mathcal{N}$, the sufficient sample length $m_L(\delta,\epsilon)$ must be at least*

$$\frac{c_2}{\epsilon}\left(d+\ln\left(\frac{1}{\delta}\right)\right),$$

for all $\epsilon \le 1/8$ and $\delta \le 1/100$.

We now present a result on the VC-dimensions of feedforward linear threshold networks. The first part is due to Baum and Haussler [6] and the second part to Maass [16,17].

Theorem 6.2 *There is a constant $c_1 > 0$ such that, if $\mathcal{N}$ is any feedforward linear threshold network with one output node and whose underlying digraph has n vertices and m arcs, then $\dim_{\mathrm{VC}}(\mathcal{N}) \le c_1 m\log n$. Furthermore, there is another constant $c_2 > 0$ such that some feedforward linear threshold networks satisfy $\dim_{\mathrm{VC}}(\mathcal{N}) \ge c_2 m\log n$, where the underlying digraph has n vertices and m arcs.*

The above result relates the VC-dimension of a neural network to its underlying graph, when the activation functions are all linear threshold functions. It provides an upper bound which is tight to within a constant. It is rather disappointing that this relationship involves only the 'size' of the graph rather than its structure. In particular, this general result does not involve the number of layers in the network. Tighter bounds have been obtained for networks with very few layers; (see [2,5], for instance). Results have also been obtained for feedforward networks with sigmoidal activation functions; for results on these and other types of network, see [2,12].

Learning can be as hard as graph-colouring

If the process of PAC learning by an algorithm L is to be of practical value, it should be possible to implement the algorithm 'quickly'. We wish to quantify the behaviour of a learning algorithm for a particular neural network architecture with respect to the size of the network. In particular, we wish to consider how the running time of the algorithm varies with the number n of inputs to the network: for a learning algorithm to be efficient, this running time should increase polynomially with n.

However, there is another important consideration in any discussion of efficiency. Clearly, decreasing ϵ makes the learning task more difficult, and therefore the time taken to produce a probably approximately correct output should be constrained in some appropriate way as ϵ decreases; the appropriate condition is that the running time must be polynomial in $1/\epsilon$.

Formally, we say that a learning algorithm L is *efficient with respect to accuracy ϵ, example size n and sample length m* if its running time is polynomial in the length m of the training sample and if there is a value of $m_L(\delta, \epsilon)$ sufficient for PAC learning that is polynomial in n and $1/\epsilon$.

Judd [15] was the first to show that learning in neural networks can be hard, in the formal complexity-theoretic sense. We now describe a simple hardness result from [3,4] along the lines of one due to Blum and Rivest [7]. Before doing so, we recall that, in complexity theory, two important classes of problems, RP and NP, are defined. The class RP is the class of all problems that can be solved by 'randomized' algorithms in polynomial time, while NP is the class of problems that can be solved by non-deterministic Turing machines in polynomial time (ee, for example, [9]). It is conjectured, and widely believed, that RP is a strict subset of NP; this is known as the 'RP$\neq$NP' conjecture.

The network $\mathcal{N}_n^k$ that we consider has n inputs and $k+1$ linear threshold nodes ($k \geq 1$); see Figure 3. The first k linear threshold nodes are 'in parallel' and each of these is connected to all of the inputs. The last threshold node is the output unit; it is connected by arcs with *fixed* weight 1 to the other linear threshold nodes and to the bias node with weight $-k$. The effect of this arrangement is that the output unit acts as a multiple AND gate for the outputs of the other threshold nodes.

Figure 3

The *consistency problem* for $\mathcal{N}_n^k$, which we refer to as $\mathcal{N}^k$-CONSISTENCY, is defined as follows:

$\mathcal{N}^k-$CONSISTENCY
Instance A sequence $\mathbf{s} = ((x_1, b_1), (x_2, b_2), \ldots, (x_m, b_m))$ of labelled examples, where $x_i \in \mathbf{R}^n$ and $b_i \in \{0, 1\}$, for $1 \leq i \leq m$.
Question Is there a state of the network $\mathcal{N}_n^k$ such that the function t then computed by the network satisfies $t(x_i) = b_i$ for $1 \leq i \leq m$? In other words, is $\mathbf{s}$ a training sample for some function computed by the network?

The following result (a special case of one that appears in [19]) relates the consistency problem to the problem of efficient PAC learning.

Theorem 6.3 *If there is a* PAC *learning algorithm for* $\mathcal{N}_n^k$ *that is efficient with respect to accuracy, example size and sample length, then there is a randomized polynomial time algorithm that solves the problem* $\mathcal{N}^k$-CONSISTENCY.

One can prove that $\mathcal{N}^k$-CONSISTENCY is NP-hard for $k \geq 3$, by showing that it is as difficult as graph colouring. Here, we sketch the reduction; full details can be found in [3]. Let G be a graph with vertex-set $V = \{1, 2, \ldots, n\}$ and edge-set E. We construct a sequence $\mathbf{s}(G)$ of labelled examples as follows. For each vertex $i \in V$ we take as a labelled example $(v_i, 0)$, where v_i is the vector which has 1 in the ith coordinate position, and 0's elsewhere. For each edge $ij \in E$ we take as a labelled example $(v_i + v_j, 1)$, and we also take as a labelled example $(o, 1)$, where o is the zero vector $o = 00\ldots0$. It can be shown (see [3]) that $\mathbf{s}(G)$ is a training sample for some function in $\mathcal{N}_n^k$ if and only if G is k-colourable. It follows that if there is a polynomial time algorithm for $\mathcal{N}^k-$CONSISTENCY, then there is one for GRAPH k-COLOURING. But GRAPH k-COLOURING is NP-complete for $k \geq 3$, and it follows that the $\mathcal{N}^k-$CONSISTENCY problem is NP-hard if $k \geq 3$. (In fact, the same is true if $k = 2$: this follows from work of Blum and Rivest [7].) Theorem 6.3 enables us to move from this hardness result for the consistency problem to a hardness result for PAC learning. The theorem tells us that, unless RP=NP, there can be no computationally efficient PAC learning algorithm for this family of neural networks.

References

1. E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, Wiley-Interscience Series in Discrete Mathematics and Optimization, Wiley, 1989.

2. M. Anthony, Probabilistic analysis of learning in artificial neural networks: the PAC model and its variants, in *The Computational and Learning Complexity of Neural Networks* (ed. I. Parberry), to appear.

3. M. Anthony and N. Biggs, *Computational Learning Theory: An Introduction*, Cambridge Tracts in Theoretical Computer Science **30**, Cambridge University Press, 1992.

4. M. Anthony and N. Biggs, Computational learning theory for artificial neural networks. in *Mathematical Approaches to Neural Networks* (ed. J.G.Taylor), North Holland Mathematical Library **51**, Elsevier Science Publishers B. V., Amsterdam, 1993, pp. 25–62.

5. P.L. Bartlett, Vapnik-Chervonenkis bounds for 2-layer and 3-layer networks, *Neural Computation*, **5**(3), (1993), 371–373.

6. E. Baum and D. Haussler, What size net gives valid generalization?, *Neural Computation*, **1**(1), (1989), 151–160.

7. A. Blum and R. L. Rivest, Training a 3-node neural net is NP-Complete, in *Advances in Neural Information Processing Systems I*, (ed. D. S. Touretzky), Morgan Kaufmann, 1989, pp. 494–501.

8. A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, Learnability and the Vapnik-Chervonenkis dimension. *J. ACM*, **36**(4) (1989), 329–365.

9. T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

10. A. Ehrenfeucht, D. Haussler, M. Kearns, and L. Valiant, A general lower bound on the number of examples needed for learning, *Information and Computation*, **82** (1989), 247–261.

11. R.S. Garfinkel, Motivation and modeling, in *The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization.* (eds. E.H. Lawler, J.K. Lenstra, A.H.G. Rinooy Kan and D.B. Shmoys). Wiley, 1985, pp. 17–36.

12. D. Haussler, Decision theoretic generalizations of the PAC model for neural net and other learning applications, *Information and Computation*, **100**(1) (1992), 78–150.

13. D.O. Hebb, *The Organization of Behaviour*, Wiley, New York, 1949.

14. J. Hertz, A. Krogh, and R. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, Redwood City, 1991.

15. S. Judd, Learning in neural networks, in *Proc. 1st Annual Workshop on Computational Learning Theory*, Morgan Kaufmann, San Mateo, CA, 1988, pp. 2–8.

16. W. Maass, Bounds on the computational power and learning complexity of analog neural nets (extended abstract), in *Proc. 25th Annual ACM Symposium on the Theory of Computing*, ACM Press, 1993, pp. 335-344.

17. W. Maass, Neural nets with superlinear VC-dimension, *Neural Computation*, **6**(5),

(1994), 877–884.

18. M. Minsky and S. Papert, *Perceptrons*, MIT Press, Cambridge, MA, 1969; expanded edition, 1988.

19. L. Pitt and L. Valiant, Computational limitations on learning from examples, *J. ACM*, **35** (1988), 965–984.

20. F. Rosenblatt, Two theorems of statistical separability in the perceptron, in *Mechanisation of Thought Processes: Proceedings of a Symposium Held at the National Physical Laboratory, November 1958. Vol. 1*, H.M. Stationery Office, London, 1959.

21. L. G. Valiant. A theory of the learnable. *Comm. ACM*, **27**(11) (1984), 1134–1142.

22. V. N. Vapnik and A. Y. Chervonenkis, On the uniform convergence of relative frequencies of events to their probabilities, *Theory of Probability and its Applications*, **16**(2) (1971), 264–280.