Electronic Delivery Cover Sheet

CHAPTER 5

YASER ABU-MOSTAFA
DAVID SCHWEIZER
California Institute of Technology

Neural Networks

Introduction

These sections address the popular subject of neural networks, a delicate subject because of its history and diversity. *The sections are not, by any means, a comprehensive survey of the subject.* They are intended only to be an introduction to the field. Several important areas and results are not addressed here.

There are two very popular models of neural networks: the feedback model and the feedforward model. I will start with the feedback model because historically that is what triggered the current wave of interest. If we went back twenty-five years we would find interest in the same models. That wave, however, died out, and so did at least one earlier wave. Before we get to the specifics of the feedback model, however, we need some general discussion of neural networks.

5.1 Networks and Neurons

The architecture of neural networks can be described as an undirected graph. We call the nodes neurons and the edges synapses, and we have

a neural network. What characterizes a neural architecture in general, whether it is feedback or feedforward, is that the number of neurons is huge, and each neuron does a very simple task. If we are considering neural networks as a subset of all distributed computations or parallel architectures, that is the main characterization. The number of units is very large, and the task of each unit is very simple, much simpler than the tasks performable by a microprocessor, for example. In many models the nodes perform threshold logic only. Furthermore, the number of synapses per neuron is large. Usually a neuron is connected to a large subset of all other neurons, for example, a fraction αN of all neurons, rather than $\sqrt{N}$ or $\log N$, as in other architectures such as the hypercube.

What does a neuron do? In both the feedback model and the feedforward model, the neuron performs a simple threshold function. It has N inputs, called $x_1 \ldots x_N$; and a single output, called y. For the moment, we consider all of these variables to be binary, and, for convenience, take the binary values to be -1 or $+1$ instead of 0 or 1. The neuron calculates a function from $\{-1,+1\}^N$ (i.e., N-tuples of -1's and $+1$'s) to $\{-1,+1\}$. The relation of the output to the input depends on a set of real numbers called the weights. There is one weight for each input variable, and what the neuron does is sum up the variables times the weights and subtract a threshold, t. If the result happens to exceed 0, it sets y to be $+1$; if it is less than 0, it sets y to be -1, and if it happens to hit zero, well, there are many technical variations of what to do.

Given this definition, the set of functions we can implement using a single neuron is well understood: It's the set of threshold functions, also called linearly separable functions. If we take the hypercube $\{-1,+1\}^N$, we can implement any dichotomy, that is, any hyperplane that separates the points on the hypercube.

There are other models for what the neuron can do, but this is the most popular one and the one on which both models, the feedback and the feedforward, are based, so this is the one I am going to consider.

5.2 Feedback Networks

The feedback network [1], which was introduced and reintroduced by several people, is a fully interconnected network of neurons, that is, every neuron is connected to every other neuron by a synapse. Let's call the states of the neurons u_i. Thus we have a vector of states, $(u_1 \ldots u_N)$. The rule is that the future state of neuron i is simply the value y, the output of the threshold function computed by that neuron.

For this particular model, we can express the entire network by the matrix W of weights simply because the thresholds have been chosen to be zero. That's part of the specification of the model, that all thresholds

are zero. Furthermore, $w_{ij} = w_{ji}$: The weights are symmetric. The final restriction is that it has a zero diagonal, which means that the neuron is not connected to itself.

Now, how does the network operate? We have the neurons, and we can initialize them to anything we want, so they have initial states. A neuron decides to update its state, so it looks at its surroundings, multiplies its inputs by the right weights, evaluates the sum, and calculates the sign. (Recall that the threshold is zero.) If it happens that this sign is different from the current state, it flips to the new sign. These updates are done by all the neurons asynchronously, that is, no two neurons ever try to update at the same moment.

Conceivably, neurons could continue to do updates forever. It could be the case that we have chosen the weights and the initial states poorly. It might be the case that every time a neuron updates, it changes its sign, and that that change would require some other neuron to change its state, and so on. The network would go around and around, and never settle down to an unchanging state, and so on. Such a process would be useless. However, there is a result [1] that says that a feedback network is guaranteed to go to a stable state after a finite number of updates. There is obviously an assumption of fairness, that is, that every neuron will eventually get to update as many times as it wants.

Do the updates have to be random? For the proof that the network actually gets to a stable state, they must be asynchronous, but not necessarily random. If we were to allow two neurons to evaluate and update simultaneously, the network could end up in a limit cycle. However, we can scan the neurons sequentially, and the network will still reach a stable state.

How do we show that a feedback network will always go to a stable state? We will define a scalar-valued energy function, and show that the energy of the network always decreases. Let the states be represented by a vector of N bits, denoted $\mathbf{u}$. We propose the following function, which is called E for energy:

$$E = -\frac{1}{2}\,\mathbf{u}^T\, W\, \mathbf{u}.$$

A helpful way to organize our thoughts about the energy function is to write the N^2 terms of the summation as a matrix. This matrix corresponds directly to the weight matrix, with each term being $u_i w_{ij} u_j$ instead of just w_{ij}. In order to evaluate E, we add up all the terms in this matrix and multiply that sum by $-1/2$. It is convenient to keep this matrix, illustrated in Figure 5.1, in mind as we go through the proof.

Let's see what happens when neuron i updates. When neuron i updates, the only element of $\mathbf{u}$ that can change is u_i, and thus all the terms in the matrix will remain the same, except possibly the ith row and the ith column because these are the only places u_i appears.

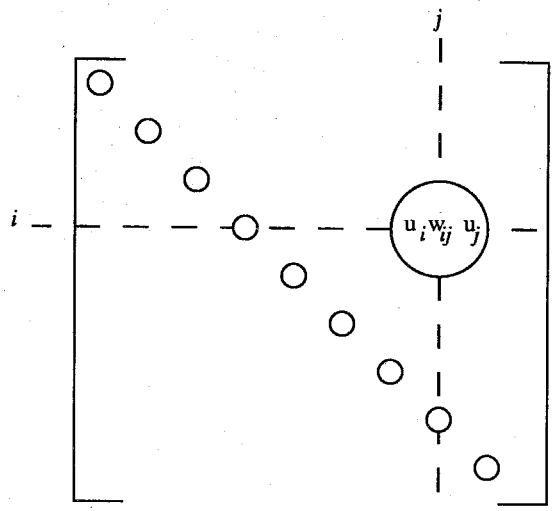

Figure 5.1 Matrix tabulation of the energy function.

Now suppose that neuron i changes state, that is, that neuron i calculates its threshold function and decides to flip the sign of u_i. We know that the new u_i is the sign of $\sum_j w_{ij} u_j$, and, since the neuron changed state, that the old u_i must have been different from that sign. Thus when we multiply the new u_i by the sum, we always get a positive number. When we multiply the old u_i by the sum we always get a negative number. If we put these facts together, and consider summing the entries in the i^{th} row (or the i^{th} column), we will find that ΔE, the change in energy, is always negative.

Why is this useful at all? This is useful because E has a lower bound, and the magnitude of ΔE for actual updates also has a lower bound. Both of these are clear because the matrix is finite, so there is only a finite number of combinations of the signed terms. Therefore, if each time the network updates, E decreases, the network is bound to reach a state where E cannot decrease further, that is, no further updates would change the state of a neuron, that is, a stable state of the entire network. This proves that with any update scheme using the rules we have, we must eventually get to a state where every neuron is happy. Every neuron looks at the states of the other neurons, multiplies them by their weights, sums the results, and finds that the sign of that sum agrees with the state it is already in.

When the feedback model is used for computation, the time evolution from the initial state to the stable state is usually what constitutes the computing. The output of the computation is deciphered from the

stable state. One designs the network to perform a certain computation by choosing the weights so that the network dynamics fulfill the specification of some algorithm. Later, we will discuss computation on feedback networks further, but first we need to investigate our control over the stable states.

5.3 Choosing the Stable States

The mere convergence to a stable state is not very useful unless we have some control over the choice of stable states. To perform a computation, we need to be able to demand that particular initial states converge to particular stable states. We need an algorithm for constructing a network with a prescribed set of stable states. If we have a network that has N neurons, the only free parameters left are the weights. We want to take a set of vectors, $\mathbf{u}^1$ through $\mathbf{u}^K$, which we want to be stable states, and produce a set of weights, W, that makes them stable. We will use an algorithm known as the sum-of-outer-products algorithm [1]. This algorithm sets

$$W = \sum_{k=1}^{K} (\mathbf{u}^k \, \mathbf{u}^{k^T} - I),$$

where I is the identity matrix. Each term of this sum is an outer product, that is, a rank 1 matrix with the entry in the i^{th} row and j^{th} column equal to $u_i u_j$. We subtract I from each term to make the diagonal be zero; this is just a technicality. This produces a weight matrix W, but we are supposed to make the $\mathbf{u}^k$'s stable, so we will argue shortly that this choice of W does indeed, with high probability, make them stable.

An interesting feature of this algorithm for the weights is that although it looks like an elaborate algebraic formula, the implementation on a neural network is very simple. In order to modify w_{ij}, we do not have to know the entire state vector, just the two bits of neurons i and j. Consider what neuron i and neuron j see when we load $\mathbf{u}^k$ as one of the states. Each of the two neurons gets a single bit of $\mathbf{u}^k$, and there is weight between them, w_{ij}. If the bits agree in sign, w_{ij} is incremented, and if they disagree in sign, it is decremented. If we do this for every weight, then when we have gone through all the $\mathbf{u}^k$'s, we have the sum of outer products. In a different scheme it could be that two bits do not suffice to calculate what w_{ij} should become, but here we just enter the first vector and adjust each w_{ij} using two bits and we get the first outer product. We input the second vector and accumulate the second outer product, and so on. This is a nice property because it is a local rule. This rule is called the Hebbian rule.

Now we are going to convince you that the $\mathbf{u}^k$'s are stable. Let us assume that the way the $\mathbf{u}^k$'s were chosen in the first place was by flipping a fair coin independently to determine each bit of each vector. This is

the same as saying that the bits of the vectors are independent *Bernoulli trials*. Assume W was constructed by the sum-of-outer-products formula. Let's take one of the vectors, say $\mathbf{u}^1$, and check that it is stable. First we multiply W by $\mathbf{u}^1$. This product gives a vector of the signals seen by each of the neurons at the beginning. The first component of the vector is the signal available to neuron 1, the second is the signal available to neuron 2, and so on, up to the signal available to neuron N. If the signal seen by a given neuron is positive, that neuron wants its state to be $+1$; if it is negative, it wants its state to be -1. If these signals happen to agree in sign with the initial state vector that we want to be stable, we are in good shape. We want to go to $\mathbf{u}^1$, and we have a vector of real numbers that happens to agree with the sign of $\mathbf{u}^1$ in every entry. We are all set because when the neurons look at this signal, they will decide to stay where they are, which means that the state is stable.

So we multiply W by $\mathbf{u}^1$. Recall that W is

$$\sum_{k=1}^{K}(\mathbf{u}^k\ \mathbf{u}^{k^T} - I).$$

The first term involves the quantity $\mathbf{u}^1\mathbf{u}^{1^T}$, which, as it is the inner product of a vector of -1's and $+1$'s with itself, is N. Thus the first term equals

$$N \times \mathbf{u}^1 - I \times \mathbf{u}^1,$$

which reduces to $(N-1) \times \mathbf{u}^1$. That's good news. We want the signals to agree in sign with $\mathbf{u}^1$, and we start with a signal that is $\mathbf{u}^1$ multiplied by a big positive number. Thus, unless the second term provides enough 'noise' to take one of the components all the way to the opposite sign, $\mathbf{u}^1$ will indeed be stable. See Figure 5.2.

When we evaluate the noise term, we find that we add up a number of independent Bernoulli trials to make up each component of the noise vector. For example, some of these Bernoulli trials come up when we compute $\mathbf{u}^1\mathbf{u}^{2^T}$. Since, by assumption, the bits of $\mathbf{u}^2$ were selected at random independently from those of $\mathbf{u}^1$, we will end up with a variety of ± 1's that are unrelated. (We call these terms noise because the bits are unrelated.) On the average, we will have as many $+1$'s as -1's, and we expect a zero sum. However, we probably won't be that lucky and have the noise cancel out exactly. There will be some variance around zero, and the variance will be controlled by how many Bernoulli trials we add. As a matter of fact, the variance will be proportional to the number of Bernoulli trials, a result that is very simple to derive.

Thus, there are many Bernoulli trials that are not really conspiring against the stability of $\mathbf{u}^1$. They are giving arbitrary values, and although they are not conspiring, if there are too many of them the variance will be

$$W\mathbf{u}^1 = (\mathbf{u}^1\mathbf{u}^{1^T} - I)\mathbf{u}^1 + \underbrace{\sum_{k=2}^{K}(\mathbf{u}^k\mathbf{u}^{k^T} - I)\,\mathbf{u}^1}$$

$$= \underbrace{(N-1)\mathbf{u}^1}_{\text{signal}} + \quad \text{noise}$$

noise = $\sum$ Bernoulli trials

variance $\propto$ Number of trials

Figure 5.2 Noise in the sum-of-outer-products.

so large that there is a possibility that one of them will swing some component all the way to the opposite sign. The number of Bernoulli trials grows with the number of 'parasitic' vectors $\mathbf{u}^2, \ldots, \mathbf{u}^K$. If this number is small, then the variance of the noise will not be sufficient to affect stability. How small should it be? That's the question of capacity [2]: how many states can be added without disrupting the behavior of the first stable state? The answer is that K can be at most the order of $\frac{N}{\log N}$ vectors [3].

5.4 Feedforward Networks

Feedforward networks are currently receiving more attention than feedback networks. The feedforward network is an architecture where the neurons are grouped in layers. There are directed connections between layers, with no loops anywhere and no connections among the neurons in the same layer. The network starts with inputs; these neurons simply hold the values of the inputs, x_i through x_N. These values are passed to the next layer through weights. The neurons in the second layer perform some computation, then pass that result to the next layer through weights, and so on. There can be many layers. The neurons in the first layer are called input units. They don't really do anything, so we could delete them and show the signals going into the second layer. The neurons in the final layer are called output units; they hold the result of the computation. The neurons in between are called hidden units because they're hidden. The relation between these networks and combination circuits is obvious. The main issue in this case is that the weights will be allowed to be real numbers.

If a feedforward network doesn't have hidden layers, there will be functions it cannot implement. The argument is actually trivial—it's the old

Perceptron [7] argument. Essentially, if the network doesn't have any hidden layers, it computes a single threshold function; if the output happens to be several bits, it computes several threshold functions. So suppose we want to implement the parity function of x_1 through x_N. Let's look at the case of two variables, x_1 and x_2, in Figure 5.3. The values $\{-1,-1\}$ and $\{+1,+1\}$ give even parity, and $\{-1,+1\}$ and $\{+1,-1\}$ give odd parity.

We would like to set the weights and the threshold. When we choose the w's, we will compute the inner product with the vector (x_1, x_2), and that will correspond to a plane. The points on one side of the plane will be mapped to $+1$, and those on the other side will be mapped to -1. We look at the hypercube, which in this case is a very simple one, and we want to be able to find a hyperplane that separates the *black* nodes, which we want to map to $+1$, from the *white* nodes, which we want to map to -1. In the case of the parity function, this is clearly impossible.

Adding a hidden layer turns out to be a very good idea. With a hidden layer, we can implement any Boolean function. The reason is very simple: We can implement any function using a canonical or-of-ands form, and we can implement those gates using threshold functions. (See Figure 5.4.) Thus, if we replace each logic gate with a neuron that computes the same function—and that is a very simple derivation—we end up with a canonical form that can implement any Boolean function of the inputs.

Although one layer suffices, sometimes it is convenient to use more than one. This is a topic in the current research, as is the number of units per layer. At this point, the rules are completely heuristic. We have intuitive arguments for what network structure to use, how many layers and how many units per layer, but no theoretical foundation whatsoever.

Now let's look more closely at the units. Again, we need a threshold

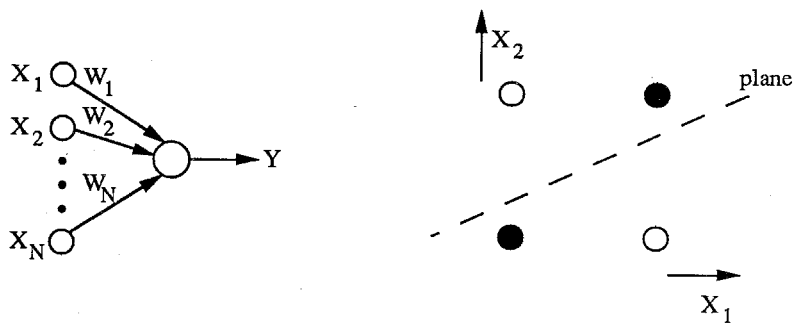

Figure 5.3 The parity is not a threshold function.

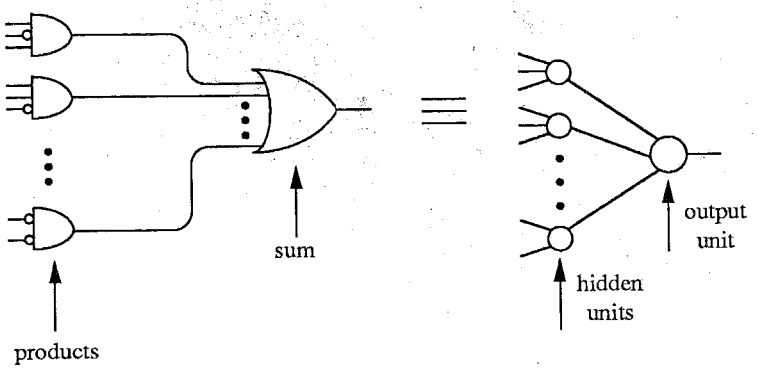

Figure 5.4 One hidden layer suffices.

function, but this time we have a feature that is sometimes very useful. In the feedback model we used a hard threshold: If the weighted sum of the inputs was positive, the output was $+1$; if negative, -1. In this model we are going to make a soft decision. If the weighted sum is very positive, the output will be very close to $+1$; if it's very negative, very close to -1, but if it is only slightly positive or slightly negative, the output will be of smaller magnitude. This nonlinear input-output relation, $y = f(\sum w_i x_i)$, has a sigmoid shape, as shown in Figure 5.5.

Only in the limit do these neurons compute Boolean functions. The final output will be passed through a hard threshold, but we would like to keep the reliability of intermediate decisions as the computation proceeds. There is no reason to make hard decisions in the intermediate stages. There is also another reason for having a soft decision, which will come from the learning rule. We will get there shortly. We also set the threshold to zero. A threshold can be added by having one of the inputs be $+1$ all the time, and using the weight on that input to give the effect of a threshold. Thus there is no loss of generality brought about by omitting the threshold.

Let us look at the learning rule. If all we were to do with multilayer networks was to take a function and implement it on a network, then we would have nothing more to offer than simple circuit design. We would even be restricting ourselves to stupid circuit elements when we might be able to build better ones. The whole idea of using multilayer networks is to be able in some sense to make the circuit build itself. We will start with just the architecture of the network: How many units and how many layers. And then we won't be given the function we want to implement, but only examples from that function. Given an input and output pair, we will use a learning rule to adjust the weights so that the circuit is more likely to give

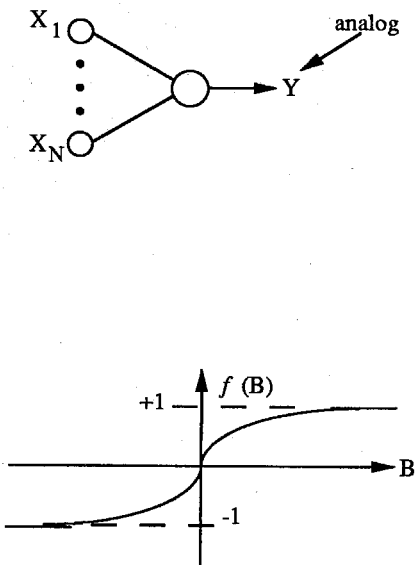

Figure 5.5 Nonlinearity of the neurons.

that output when it sees that input. We'll do that, example after example, until finally, we hope, we have a circuit that actually computes the original function.

Suppose we are training just one unit to learn a function. Assume the weights start at any random initial set of values. Given an input, $(x_1 \ldots x_N)$, suppose we want the output to be $\hat{y}$. We feed the inputs as they are to the circuit. We have no idea what the output will be, but it's a starting point. Now we get something that could, by chance, very closely agree with the desired output, or could disagree terribly. Or, as the output is an analog value, it could be in between. Now we would like to perturb the w's so that after the perturbation, if we re-enter the inputs $(x_1 \ldots x_N)$, the output will be closer to $\hat{y}$.

Unfortunately, these perturbations may disturb a value the circuit has already learned. We use the same noise argument as in the feedback model. It's a small perturbation and is optimally directed toward accommodating a particular input/output pair. If the other pairs are unrelated, we have a good chance that this behaves as noise for them. Empirically, focusing on one pair at a time and doing several runs actually works.

5.5 Back Error Propagation

Now, how do we perturb the weights? Let's assume we have an error function, not specified at the moment but well defined for any input/output pair. The error is a function of the weights $w_1, \ldots, w_N$ in the network:

$$E = E(w_1, \ldots, w_N).$$

Some sets of weights will give one value for the error function, and other sets will give smaller or larger values. We will perturb the weights in a manner that depends on the error, and would like to make ΔE, the change in the error, of large magnitude. It's a gradient, so we'll take it negative or positive. The size of the perturbations permitted is bounded by

$$\sum_i (\Delta w_i)^2 = const.$$

(This limits the distance moved in weight space.)

The problem is how to decide how much to modify each weight to maximize ΔE, subject to the constraint. We begin by assuming that we can approximate the change in the energy by:

$$\Delta E = \sum \frac{\partial E}{\partial w_i} \Delta w_i.$$

Since we want to maximize ΔE subject to a constraint, we will use the method of Lagrange multipliers. We set the term

$$\frac{\partial}{\partial \Delta w_i} \left[\sum \frac{\partial E}{\partial w_i} \Delta w_i + \lambda \left(\sum (\Delta w_i)^2 - const. \right) \right]$$

equal to 0 for all i from 1 to N, solve, and end up having to perturb each weight by an amount proportional to the rate of variation of E with respect to that weight, but in the opposite direction. That is, we get

$$\Delta w_i \propto -\frac{\partial E}{\partial w_i}.$$

This is a very satisfactory result. If the error is extremely sensitive to one of the weights, then we will perturb that weight a lot because it will affect E a lot. If it doesn't matter, then we shouldn't bother because we have a constraint on the total perturbation. We want to make our modifications where they will have the most effect. We will go for the bigger ones, and although we might think at the beginning that we would want to pick the largest one and perturb it all the way, we now know that isn't optimal. Note that this is independent of the choice of the error function. We know now

if we have any error function and would like to change the weights, with this restriction on the perturbation, such that we get closer to the correct output, we should pick the perturbation proportional to the negative of the derivative.

This gives rise to what's called the Delta rule [4]. First, we need a specific error function. A good choice is the mean square error,

$$E = \frac{1}{2}(\hat{y} - y)^2$$

where $\hat{y}$ is the target output, and y is the actual output. We could start with another function and would end up with a different rule. Mostly, however, we use this one. For convenience, we will write θ for the weighted sum of the inputs to the neuron, that is,

$$\theta = \sum_i w_i x_i.$$

We still have a sigmoid for the output; recall Figure 5.5. We already have

$$\Delta w_i = -\alpha \frac{\partial E}{\partial w_i}.$$

Now,

$$-\frac{\partial E}{\partial w_i} = -\frac{\partial E}{\partial \theta}\frac{\partial \theta}{\partial w_i}.$$

(The term $-\frac{\partial E}{\partial \theta}$ is called δ, and that's why this whole procedure is called the Delta rule.) Further,

$$-\frac{\partial E}{\partial \theta} = -\frac{\partial E}{\partial y}\frac{\partial y}{\partial \theta} = -(\hat{y} - y) f'(\theta),$$

and obviously

$$\frac{\partial \theta}{\partial w_i} = x_i.$$

All of this together yields our rule for changing the weights:

$$\Delta w_i = \alpha (\hat{y} - y) f'(\theta) x_i = \alpha \delta x_i.$$

Now let us explain this intuitively; we don't have to do the math. First, if we are way out on one of the tails of the sigmoid, we don't care what happens to w because a change in w will barely affect the output. The neuron is in saturation. So if $f'(\theta)$ is close to zero, we just forget about it. But if we are near the origin, that's very important, because perturbing w could drive the output to the positive side or the negative side. In that case,

the change is very important. The rest of the result is simply agreement in sign. If the target output minus the actual output agrees with x_i, we would like to increase w_i because that increase will push θ in the direction we already want. If they disagree, the negative sign means the weight will be decreased, and that will drive it in the other direction. Thus, at least in the first-order checks, we see that this is a plausible formula.

Now we can see another reason for maintaining analog values instead of using a hard threshold, and that is to be able to use intermediate information. When we want to adjust things slightly, we would like to have some flexibility in the perturbation. If the output is required to be $+1$ or -1, we either leave it entirely unchanged or flip it completely. There is no measure for how severe or mild the perturbation is. But in this case, we have a spectrum of values to use, which is usually useful to have around in these algorithms.

The generalization of the Delta rule for the multilayer networks is the famous Back Error Propagation Algorithm [4], and it is because of that algorithm that a lot of attention is paid to multilayer networks.

Suppose we have a fragment of a multilayer network that looks like Figure 5.6. We will again be using gradient descent. We start with the equation

$$-\frac{\partial E}{\partial w_{ij}} = -\frac{\partial E}{\partial \theta_j}\frac{\partial \theta_j}{\partial w_{ij}}.$$

Note that since E is indeed affected by a perturbation of w_{ij} (even though in a rather elaborate way), it is legitimate to define the partial derivative of E with respect to w_{ij}. Gradient descent requires that we make Δw_{ij} proportional to that derivative. Note also that the only way w_{ij} comes into the equation is by affecting the input signal to the neuron, so it is legitimate to apply the chain rule as we have.

There are now two cases: Either j is an output unit, or it is a hidden

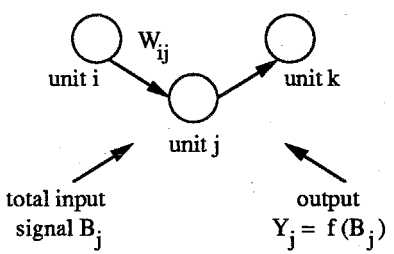

Figure 5.6 Intermediate unit.

unit. If j is an output unit, then we use the Delta rule. Once we have δ_j, we know how to compute Δw. So there is nothing new here, if j is an output unit. The situation becomes very interesting when we are dealing not with an output unit, but with a hidden unit.

In that case, the formula is much more elaborate. As before, we start with

$$\delta_j = -\frac{\partial E}{\partial \theta_j}.$$

We've added some subscripts, but the definition is exactly what it was. We can apply the chain rule and we get

$$-\frac{\partial E}{\partial \theta_j} = -\frac{\partial E}{\partial u_j}\frac{\partial u_j}{\partial \theta_j}.$$

The second term of the product is easy: it's just $f'(\theta_j)$. For the first part, we need to remember that we have more layers after this neuron. We can write

$$\frac{\partial E}{\partial u_j} = \sum_k \frac{\partial E}{\partial \theta_k}\frac{\partial \theta_k}{\partial u_j},$$

where k indexes the neurons in the next layer. The second term of the product is easy: The rate of change of the output of a neuron with respect to one of its inputs is just the weight on that input line, w_{jk}. Now comes the trick. We don't know very much about $\frac{\partial E}{\partial \theta_k}$, but it's enough to be able to write

$$\frac{\partial E}{\partial \theta_k} = -\delta_k.$$

We put it all together and get:

$$\delta_j = f'(\theta_j)\sum_k w_{jk}\delta_k.$$

We already know how to compute the errors at the output layer. Once we've done that, we can move back one layer and compute the δ's there. Notice that the summation we need to compute looks just like the one the neuron uses to compute θ. That's why the algorithm is called back error propagation. We propagate answers in one direction and errors in the other (see Figure 5.7).

So, how well does this actually perform? It performs extremely well on toy examples. It does so extremely well that it not only implements the examples it has seen, but also extrapolates correctly in many cases for examples it hasn't seen [4].

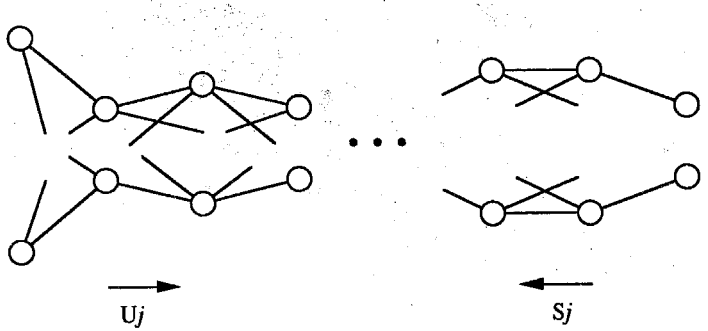

Figure 5.7 Back error propagation.

5.6 Collective Computation

The idea of collective computation [1] is to work with a very large system made up of a lot of simple components. The components may be unreliable: They could execute their instructions inexactly or even die completely. But the hope is that the overall system will reliably perform some sophisticated task despite the individual failures.

Do feedback neural networks perform that kind of computation? In some sense, yes. See Figure 5.8. During the programming phase, when we load in a state we want to be stable, the change in the weight on each

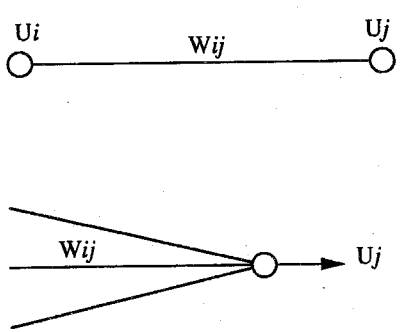

Figure 5.8 Simple rules leading to sophisticated function.

synapse depends only on the states of the neurons at each end of that synapse. This is physically implementable by burning, or not burning, a wire, a simple operation. But collectively, the neurons make a specific set of vectors stable in the network. That may not be too sophisticated, but it is definitely more sophisticated than the individual operations.

During the computation phase we would like to initialize the network to some state and have it converge to one of the stable states. Although there are no theoretical results, we find that experimental [1] networks—where the weights are not very exact, and some of the neurons fail, and some of the neurons are very lazy about updating, and so on—actually operate correctly. The overall operation is reliable, although the individual components are not.

Now let's look at two specific examples of computation on neural networks. We would like to justify the examples with an analogy. Suppose we were to go back in time 40 or 50 years and try to sell someone on the idea of computers. We would like to claim that computers can do wonderful things, but as we don't have a whole system, we can only give an example. So we build a full adder from nand gates. That's simple enough, but it's rather surprising the first time one sees it. It looks very good because the full adder performs an operation that isn't very simple, and it demonstrates the idea of computers. Unfortunately, we couldn't really say that once we have a full adder, we have all kinds of computers. The question for neural networks is similar: We have this model, which does some very specific tasks quite nicely, but can we extrapolate further?

We are not claiming that these two examples are good applications or bad applications. We just want to state what the applications are to give you some idea of the use of these models for computation. We know that the individual components are simple and inexact because we build them that way. The rest of the story comes at the system level.

5.7 Nearest Neighbor Search

Our first example is the nearest neighbor search problem. In this example, the neural network solution will be apparent to you as soon as we state the problem. We have a fixed set of M binary vectors of length N. The problem is this: Given an N-bit binary vector, find the vector in the set that is closest to it in the Hamming sense. This problem comes from the theory of error-correcting codes. The binary vectors are the legitimate code words that could be transmitted over a channel. When a message is sent over a noisy channel, what is received is only an approximate version of what was sent. Finding the code word closest in the Hamming sense to the received message is a maximum likelihood decoding.

So here we have a useful problem: optimal decoding for linear codes.

Linear codes are a subclass of all codes and have the property that the code can be specified without listing all the code words. There is a simple matrix from which all code words can be generated. The whole problem can be expressed by something of size polynomial in N. Performing a maximum likelihood decoding in a linear code is $\mathcal{NP}$-complete in N. We emphasize N because, in the case of linear codes, the list of code words $\mathbf{x}_1$ through $\mathbf{x}_M$, is already of exponential length. Even if the computation runs in time polynomial in M, that's still exponential in N. The problem may still be easy in terms of M, if M is not too large.

The feedback network solution is what one would expect. Take the code words, that is, the vectors $\mathbf{x}_1$ through $\mathbf{x}_M$, and use the sum of outer products algorithm to make them be the stable states of the network. To do maximum likelihood decoding, simply initialize the network to the received vector, let it run, and after a while it will converge to the closest code word. We have not proven to you that it will converge to the closest, but it has been demonstrated experimentally.

The problem is that the capacity of a network of N neurons is of order $N/\log N$. That is, this scheme will work properly as long as the number of code words is, at most, essentially of order $N/\log N$. Let's be generous and call it order N. Now, for a linear code, or any other reasonable code, the number of code words is exponential in N. If we were to apply the sum of outer products to exponentially many code words, we would end up with a very noisy matrix that would exhibit all kinds of irrelevant dynamics.

But the number of code words is big. What can we do? We'll just make N big. We would like the length to be the same as the number, so we will pad every code word with a bunch of extra bits, say +1's. Then apply the sum of outer products to set the weights. Each time we receive a vector, we'll initialize the network to the vector we received, padded with +1's, and let it converge. That will definitely do it, but now we have a problem: The network size is proportional to M, which is exponential in N. The exponential pops somewhere every time we try a new solution. We will argue that this is not a coincidence. *When we try to use neural networks to solve a problem that is inherently exponential, the size of the network has to be exponential.*

5.8 Traveling Salesman Problem

Now let's look at the Traveling Salesman problem [5]. When we looked at the nearest neighbor search problem, we were able to embed the problem directly into a neural network. A typical Traveling Salesman problem is stated in a way that doesn't lend itself to any direct implementation, so we have to do something to make it fit. The problem is this: Given a list of M cities (labeled 1 through M) and a matrix $[d_{xy}]$ of intercity distances, find

a minimum length tour of the cities. That is, find a permutation $x_1 \ldots x_M$ of $1 \ldots M$ that minimizes

$$\sum_{m=1}^{M} d_{x_m x_{m+1}},$$

where we take $M + 1$ to be 1. There may be more than one solution.

How can we solve this on a feedback neural network? In general, if we have a problem that doesn't fit directly into a feedback network, we use the energy as a catalyst to make it fit. The steps are the following.

First, we have to encode the problem into a feedback neural network. Feedback neural networks have weights and initial states, and problems have inputs. We have to make them correspond. Also, problems have solutions and neural networks have stable states, so we must be able to encode the solution to the problem as a stable state. We just have to choose some convention. Sometimes the choice of that convention is crucial.

In this case, we have an optimization problem, so the second step is to find a scalar function such that minimizing that function will yield a solution to the problem. Find just one scalar such that the absolute minimum of that scalar corresponds to a minimum tour.

Finally, consider the scalar function to be the energy function of a network. Since energy has a specific expression in terms of the states and weights of the network, if the scalar function happens to be in the right form, we will end up with a correspondence between the weights and thresholds of the network and the constants that arise in the problem. This is abstract. We will apply it directly to the Traveling Salesman problem and you will see the application.

There is one small change from our earlier discussion of feedback networks. In the proof that a network will converge to a stable state, we didn't have any internal thresholds—they were all set to 0. It gives us some more room to encode different scalar functions if we allow linear threshold terms as well. The new expression for the energy is

$$E = -\frac{1}{2}(\mathbf{u}^T W \mathbf{u} - 2\mathbf{u}^T \mathbf{t}),$$

where $\mathbf{t}$ is the threshold vector.The energy function is still monotonically decreasing with every update, and exactly the same analysis applies.

Now we'll look at the feedback network encoding of the Traveling Salesman problem. If there are M cities, we will build a network with M^2 neurons. For the moment, think of the neurons as being arranged in a matrix and labeled by a double index. We can relabel everything in a linear manner later.

The rows of the matrix correspond to the cities and are indexed by x, $1 \leq x \leq M$. The columns correspond to the rank, that is, the order in the tour of each city, and are indexed by i, $1 \leq i \leq M$. (See Figure 5.9.) We

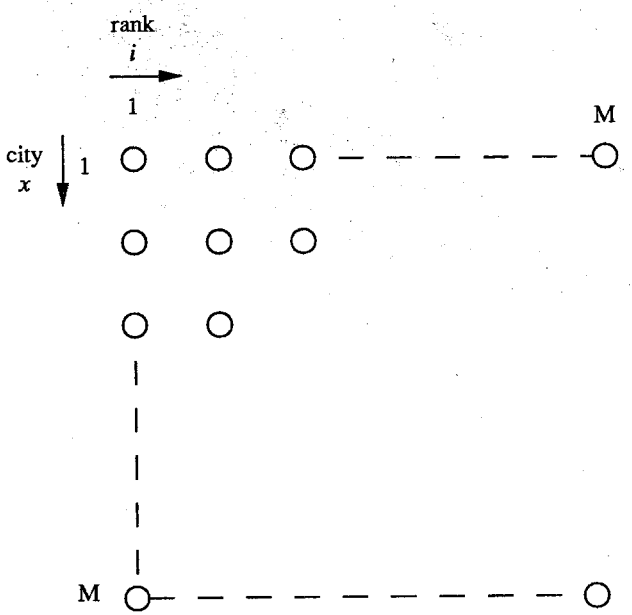

Figure 5.9 Traveling Salesman matrix.

require that the solution be a permutation matrix, that is, that there is exactly one +1 in each row and in each column. If there is a +1 in, for example, the fifth position of the first row, then city 1 is to be visited fifth on the tour. If we want to read off the tour, we find the +1 in the first column, which indicates the first city to visit, then the +1 in the second column, and so on. If the matrix is a permutation matrix, then this will be a legitimate tour, because every city will be visited once and only once.

Now we need to find a function to optimize such that optimizing that function will correspond to solving the Traveling Salesman problem, and then equate that function with an energy function. We might as well call the function an energy function right away. In the expression of the energy function, we are going to use $\{0,1\}$-valued variables, v_{xi}, instead of the $\{-1,+1\}$-valued u_{xi}. It's easier to express, and at the end we can substitute the expression

$$\frac{1}{2}(u_{xi}+1)$$

for each occurence of v_{xi}. The energy function will incorporate four conditions, each of which describes some part of the requirements of the problem.

The first condition is that a solution have at most one 1 per row. (We actually need exactly one 1 in each row, but we'll deal with that requirement in a moment.) Consider the expression

$$E_1 = \sum_x \sum_i \sum_{j \neq i} v_{xi} v_{xj}.$$

We sum over all cities over all pairs of distinct columns. If there is at most a single 1 in each row, then every term will be 0. Furthermore, since the v_{xi} are non-negative numbers, $E_1 \geq 0$. So here is a function which is greater than or equal to 0, and furthermore, is exactly 0 if and only if there is at most one 1 per row. That's a function I'd like to minimize.

The second condition is just the same—one 1 per column. We use the same idea, and get

$$E_2 = \sum_i \sum_x \sum_{y \neq x} v_{xi} v_{yi}.$$

Nothing to it.

We need exactly one 1 in each row and each column. If we have at most one 1 in each row, and at most one 1 in each column, and M 1's in the whole matrix, then we're done—we have a permutation matrix. The expression for the third condition is:

$$E_3 = \left(\sum_x \sum_i v_{xi} - M \right)^2.$$

If there are M 1's, this expression is 0. If there are more or fewer, it is positive. So now we have three conditions that we would like to minimize.

We could write an entirely different set of conditions to accomplish the same purpose, and that set might make the network perform better or worse. But the claim we will make later is that once you choose the size of the network, there is no good way of doing it. The network is too small.

The fourth condition is the real condition. The conditions we have written so far take care of only the syntax—they make sure that the solution we read off is a tour. Now we would like to minimize the length of the tour, so we look at this expression:

$$E_4 = \frac{1}{2} \sum_x \sum_{y \neq x} \sum_i d_{xy} v_{xi} \left(v_{y,i+1} + v_{y,i-1} \right),$$

where (in the subscripts) we take $M+1$ to be 1, and $1-1$ to be M. We start with the input matrix of distances d_{xy}. Whenever v_{xi} is 1, the distance will matter, because $v_{xi} = 1$ means that we pass through city x at time i. Now suppose that $v_{y,i+1} = 1$. This means that city y is visited at time $i+1$. The

tour goes from city x is to city y, and therefore the distance between x and y matters. The same reasoning applies to the preceding city; the 1/2 in front fixes counting everything twice. This is exactly the way it was done originally, and we are doing it the same way in order to follow experimental results. If we minimize this expression, we will minimize the length of the tour—if the matrix of v_{xi}'s is a permutation matrix. If the matrix has 1's all over the place, then this expression corresponds to complete nonsense; but if we have a tour, and we minimize E_4, then we have the minimum tour.

Now we take a big step. We would like to minimize E_1, E_2, E_3, and E_4. If we absolutely minimize each of them, then we will get an exact, correct solution: the minimum tour. But unfortunately we have only one energy (because we have only one network) so we have to combine the four components and hope for the best. We use a linear combination:

$$E = AE_1 + BE_2 + CE_3 + DE_4.$$

Recall that E_1, E_2, and E_3 make the solution a tour, and E_4 makes it of minimal length. Suppose we set A, B, and C to one million, and set D to 1. We are sure to get a tour. It could be disastrously long, but it will be a tour. If we set A, B, and C to 1, and D to one million, then we would get a wonderful non-tour. We can spend some time choosing the constants optimally, and all kinds of questions about stability and getting stuck in local minima arise, but at least we have an energy function.

If we look at all the expressions, we find that we can write the energy in the canonical form

$$E = -\frac{1}{2}\left[\sum_x \sum_y \sum_i \sum_j \alpha_{xyij} u_{xi} u_{yj} - \sum_x \sum_i \beta_{xi} u_{xi} + \delta\right].$$

Originally, we wanted to design a network. Since we know that the energy function corresponds exactly to the network, we can simply read off the coefficients as the weights and thresholds. Whatever is multiplied by $u_{xi}\ u_{yj}$ becomes $w_{xi,yj}$, the weight between them. Whatever is multiplied by u_{xi} is t_{xi}, the internal threshold. We can forget the constants, since if we minimize an expression, we minimize that expression plus a constant. That's the solution to the Traveling Salesman problem. We have a network, that—cross your fingers—will solve the Traveling Salesman problem.

Now for some remarks. These are not meant to be positive or negative; they are just observations.

First, the network size is quadratic in the number of cities. Next, since we are minimizing a linear combination of the conditions, we are not sure that any one of the energies is actually minimized, and therefore we are not sure that the solution is legitimate. Even if it is legitimate, we are not sure

it's optimal. We have a curious situation: Suppose we run this network and get an illegitimate solution. We visit city 5 in positions 3 and 5. How does one translate that into an optimal tour? This example actually happens some of the time.

In the description of the feedback model, we took the non-linearity of the neurons to be hard threshold. When optimizing, it is advisable to soften things because every hard decision creates stability problems. If we choose a sigmoid with a reasonable slope for the non-linearity, we will end up with a solution. It can be proved that if the slope is not too low—higher than the line $y = x$ will do—then the network will always end up in a stable state of, asymptotically, $+1$'s or -1's. We won't get the network stabilizing to a state with values like .7 and $-.8$. We will always gets $+1$'s and -1's eventually, at least asymptotically, and that helps the convergence greatly.

We need a new set of weights for every problem instance, even if the number of cities doesn't change. If someone gave us a feedback network and told us that we could use it for a computation, we would expect that the problem was embedded in the weights and that the network would take any problem instance of that size as initial values of the neurons and return the solution as the final values of the neurons. But here we have to change the weights in order to load a problem instance.

The choice of A, B, C, and D, is entirely heuristic and absolutely crucial. If we choose them wrong, we end up with a disaster. Even the simplest heuristic algorithm for solving the Traveling Salesman problem will yield a better result than the neural network method, if these constants are wrong. However, it was experimentally observed that the solutions this method gets are very good. When it was tried on $M = 10$—this example was reported—people were very enthusiastic.

But it has been observed experimentally [5], that when M is larger—and never mind 100, 15 is enough—the solution collapses in a ridiculous way. The method hardly ever gives a legitimate solution, and even when it does give a legitimate solution, it gives an inadequate tour. Something is just not working.

5.9 Limitations

We are going to ask a little bit theoretically, whether we stand a chance when we try to solve a hard problem, let's say a hard optimization problem, on a neural network. Maybe we can actually answer the question before we try several experiments. We are going to investigate the network size and the computation time as they relate to the problem complexity [6]. It may be that complex problems need bigger networks, or maybe we need to wait more, or something of this sort. As Chuck Seitz noted, theoreticians love simulations, so we have a simulation here.

First we are going to ask ourselves how quickly the neural network converges to a solution. It has been observed that it converges to a solution extremely quickly—sublinearly. But since we promised theory, we will use pessimistic estimates, which we can prove. We took the case with finite resolution for the weights, which is true for all the experiments that have been run. Nobody uses real numbers as genuine real numbers. If each weight has a finite number of levels, big as they may be, each term in the energy function assumes a finite number of values. The energy function has N^2 terms because of the double summation, and we end up with the energy function having the order of N^2.

We noted that every update decreases E. If something has, at most, order N^2 levels, and every step decreases it, then by the time we get to the minimum of it, we must have not spent more than the order of N^2 units of time. So the time for convergence is at most N^2.

We come to the real question. Is this good or bad? It looks like a good thing to have a system find the solution quickly. Except that it's what you consider to be constant, and what you consider to be variable, because you can always get the solutions quickly by getting wrong solutions. There's nothing to it. Cut the computation when you are tired, and read off the "solution," before it has converged.

Suppose we try to simulate a neural network. We want to simulate a full computation of a neural network of N neurons. We are doing it using a very simple sequential machine, so we have N^2 iterations at most to worry about. We are going to consider that we don't have a built-in threshold of N variables. We are going to consider it variable by variable and add them up, and that takes linear time.

We are going to be very pessimistic. We are going to scan the neurons one by one to look for one to update, and each time we scan them, the last one we look at is the one to be updated. Each update takes N evaluations to compute. Thus, we can simulate the entire computation of a feedback model in order N^4 cycles.

Now let's say that the network does solve a truly hard problem. Simulation will usually tell you that there is no magic, because whatever you can do, we can do. Maybe it will take us longer, but the mystique of having a new system solve a class of problems which we never solved before can always be killed by simulation arguments. The usefulness of the model will depend on whether the simulation overhead is big or small. If it's big, the model may be worthwhile, because the simulation is not that effective. If it's small, then it may not be worthwhile at all.

So suppose we have a genuinely hard problem with proven exponential time complexity. Most of the $\mathcal{NP}$-complete problems are conjectured to be in that class. So we have the time complexity $t(n)$ lower bounded by some constant times α^n. We are using big N for the number of neurons, and small n for the size of input. If we have a solution using a neural network,

then give us the neural network that you used, and we are going to simulate it. When we simulate it, it will take us $O(N^4)$ steps. However, we just observed that this problem cannot be solved in less than α^n steps, and by simulating your solution, we were able to solve it in $O(N^4)$ steps. It must be that $O(N^4)$ is at least α^n. If the time complexity of a problem is at least N^{100}, it means that no solution whatsoever will take less than N^{100}. If a solution is found on a different machine that we happen to have simulated in N^{50} steps, then the complexity of the problem could not have been N^{100} because here is a solution which took only N^{50}. This implies that N has to be exponential in n. *So if you ever succeed in solving a problem that is exponential-time using a neural network, then you must have used an exponential number of neurons.* That's why we said that converging to a point quickly was not a good idea, because if the exponential is there, you will either absorb it in the time, or absorb it in the size. In this case you absorb it in the size, which is unfortunate, because you are building this whole network to solve an instance of a problem, and we would rather wait for the solution than fill a room with the network!

Bibliography

[1] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Natl. Acad. Sci. USA*, vol. 79, pp. 2554–2558, 1982.

[2] Y. S. Abu-Mostafa and J. St. Jacques, "Information capacity of the Hopfield model," *IEEE Trans. Inform. Theory*, vol. IT-31, pp. 461–464, 1985.

[3] R. J. McEliece, E. C. Posner, E. R. Rodemich, and S. S. Venkatesh, "The capacity of the Hopfield associative memory," *IEEE Trans. Inform. Theory*, vol. 33, pp. 461–482, 1987.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing, vol. 1.* Cambridge, MA: MIT Press, 1986.

[5] J. J. Hopfield, "Collective computation, content-addressable memory, and optimization problems," in *Complexity in Information Theory,* Y. S. Abu-Mostafa (ed.), Springer-Verlag, pp. 99–114, 1988.

[6] Y. S. Abu-Mostafa, "Neural networks for computing?" in *Neural Networks for Computing.* New York: AIP Conf. Proc., vol. 151, pp. 1–6, 1986.

[7] P. Lewis and C. Coates, *Threshold Logic*, John Wiley, 1967.

[8] J. Stephen Judd, "On the complexity of loading shallow neural networks," *Journal of Complexity*, vol. 4, pp. 177–192, 1988.