Computer Speech and Language (1987) **2**, 35–61

Learning sets of filters using back-propagation

David C. Plaut and Geoffrey E. Hinton

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

Abstract

A learning procedure, called back-propagation, for layered networks of deterministic, neuron-like units has been described previously. The ability of the procedure automatically to discover useful internal representations makes it a powerful tool for attacking difficult problems like speech recognition. This paper describes further research on the learning procedure and presents an example in which a network learns a set of filters that enable it to discriminate formant-like patterns in the presence of noise. The generality of the learning procedure is illustrated by a second example in which a similar network learns an edge detection task. The speed of learning is strongly dependent on the shape of the surface formed by the error measure in "weight space". Examples are given of the error surface for a simple task and an acceleration method that speeds up descent in weight space is illustrated. The main drawback of the learning procedure is the way it scales as the size of the task and the network increases. Some preliminary results on scaling are reported and it is shown how the magnitude of the optimal weight changes depends on the fan-in of the units. Additional results show how the amount of interaction between the weights affects the learning speed. The paper is concluded with a discussion of the difficulties that are likely to be encounted in applying back-propagation to more realistic problems in speech recognition, and some promising approaches to overcoming these difficulties.

1. Introduction

A major difficulty in designing systems for hard perceptual tasks like speech recognition is in developing the appropriate sequence of representations, or filters, for converting the information in the input into a form that is more useful for higher-level processes. Rumelhart, Hinton & Williams (1986*a*,*b*) described a learning procedure, called *back-propagation*, that discovers useful representations in layered networks of deterministic, neuron-like units. The procedure repeatedly adjusts the weights on connections in the network to minimize a measure of the difference between the actual output vector of the network and the desired output vector given the current input vector. The power of this procedure for learning representations (Hinton, 1986) makes it plausible to train

0085–2308/87/010035 + 27 $03.00/0

networks to perform difficult tasks that have, untill recently, been too ambitious for connectionist learning procedures. The research described in this paper addresses some of the issues that must be resolved in order for back-propagation to be a viable design tool for speech recognition.

We begin in Section 2 by describing the units, the way they are connected, and the details of the learning procedure. We then give an example in which a network learns a set of filters that enable it to discriminate formant-like patterns in the presence of noise. The example shows how the learning procedure discovers weights that turn units in intermediate layers into an "ecology" of useful feature detectors, each of which complements the other detectors. The generality of the learning procedure is illustrated in Section 4 by a second example (taken from vision) in which a network, similar in structure to the one used in the first example, learns to map edge-like patterns of varying contrast into a representation of the orientation and position of the edge in ρ–θ space.

A major issue in the use of back-propagation as a design tool is how long it takes a network to learn a task. The speed of learning is strongly dependent on the shape of the surface formed by the error measure in "weight space". This space has one dimension for each weight in the network and one additional dimension (height) that represents the overall error in the network's performance for any given set of weights. For many tasks, the error surface contains ravines that cause problems for simple gradient descent procedures. Section 5 contains examples of the shape of the error surface for a simple task and illustrates the advantages of using an acceleration method to speed up progress down a ravine without causing divergent "sloshing" across the ravine.

The main drawback of the learning procedure is the way learning time scales as the size of the task and the network increases. In Section 6 we give some preliminary results on scaling and show how the magnitude of the optimal weight changes depends on the fan-in of the units. Additional results in Section 7 illustrate how the amount of interaction between the weights affects the learning speed.

In the final section, we discuss the difficulties that are likely to be ecountered when attempting to extend this approach to real speech recognition, and suggest a number of promising approaches for overcoming these difficulties.

2. Back-propagation

2.1. The units

The total input, x_j, to a unit j is a linear function of the outputs of the units, i, that are connected to j and of the weights, w_{ji}, on these connections.

$$x_j = \sum_i y_i w_{ji} \tag{1}$$

A unit has a real-valued output, y_j, (also called its *state*), that is a non-linear function of its total input.

$$y_j = \frac{1}{1+e^{-x_j}} \tag{2}$$

It is not necessary to use the exact functions given by equations (1) and (2). Any input output function that has a bounded derivative will suffice. However, the use of a linear

function for combining the inputs to a unit before applying the non-linearity greatly simplifies the learning procedure.

2.2. Layered feed-forward networks

The simplest form of the learning procedure is for networks that are organized into sequential layers of units, with a layer of input units at the bottom, any number of intermediate layers, and a layer of output units at the top. Connections are not allowed within a layer, or from higher to lower layers. The only connections allowed are ones from lower layers to higher layers, but the layers need not be adjacent; connections can skip layers.

The network is run in two stages: a forward pass in which the state of each unit in the network is set, and a backward pass in which the learning procedure operates. During the forward pass, an input vector is presented to the network by setting the states of the input units. Layers are then processed sequentially, starting at the bottom and working upwards. The states of units in each successive layer are determined in parallel by applying equations (1) and (2) to the connections coming from units in lower layers. The forward pass is complete once the states of the output units have been determined.

2.3. The learning procedure

The aim of the learning procedure is to find a set of weights such that, when the network is presented with each input vector, the output vector produced by the network is the same as (or sufficiently close to) the desired ouput vector. Given a fixed, finite set of input–output cases, the total error in the performance of the network with a particular set of weights can be computed by comparing the actual and desired output vectors for every case. The error, E, is defined by

$$E = \tfrac{1}{2}\sum_{c}\sum_{j}(y_{j,c} - d_{j,c})^2 \tag{3}$$

where c is an index over cases (input–output pairs), j is an index over output units, y is the actual state of an output unit, and d is its desired state.

The learning procedure minimizes E by performing gradient descent in weight space. This requires computing the partial derivative of E with respect to each weight in the network. This derivative is simply the sum of the partial derivatives for each of the input–output cases. For a given case, the partial derivatives of the error with respect to each weight are computed during a backward pass that follows the forward pass described above.

The backward pass starts with the output units at the top of the network and successively works down through the layers, "back-propagating" error derivatives to each weight in the network. To compute these derivatives it is necessary first to compute the error derivatives with respect to the outputs and inputs of the units.

First, the change in error is computed with respect to the output y_j of each output unit j. Differentiating equation (3) for a particular case, c, and suppressing the index c gives

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

The chain rule can then be applied to compute the change in error with respect to the input x_j of each output unit, .

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \frac{\mathrm{d}y_j}{\mathrm{d}x_j}.$$

Substituting the value of $\mathrm{d}y_j/\mathrm{d}x_j$ (from differentiation of equation 2) gives

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} y_j(1 - y_j). \tag{4}$$

It is now known how a change in the total input to each output unit will affect the error. As this total input is simply a linear function of the states of the units in the next lower layer and the weights on the connections from those units, it is easy to compute how changing these states and weights will affect the error. The change in error with respect to a weight, w_{ji}, from unit i to output unit j, is

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial x_j} \frac{\partial x_j}{\partial w_{ji}}$$

$$= \frac{\partial E}{\partial x_j} y_i. \tag{5}$$

The effect of changing the output of unit i on the error caused by output unit j is simply

$$\frac{\partial E}{\partial x_j} \frac{\partial x_j}{\partial y_i} = \frac{\partial E}{\partial x_j} w_{ji}.$$

As unit i may be connected to a number of output units, the total change in error resulting from changing its output is just the sum of the changes in the error produced by each of those output units:

$$\frac{\partial E}{\partial y_i} = \sum \frac{\partial E}{\partial x_j} w_{ji}. \tag{6}$$

Figure 1 shows these steps of the backward pass laid out graphically. It has been shown how $\partial E/\partial y$ can be computed for any unit in the penultimate layer when given $\partial E/\partial y$ for all units in the last layer. Therefore this procedure can be repeated to compute $\partial E/\partial y$ for successively earlier layers, computing $\partial E/\partial w$ for the weights in the process. The amount of computation required for the backward pass is of the same order as the forward pass (it is linear in the number of connections) and the form of the computation is also similar. In both cases, the units compute a sum by multiplying each incoming quantity by the weight on the connection (see equations 1 and 6). In the backward pass, all the connections are used backwards, and $\partial E/\partial y$ plays the role that y plays in the forward pass. The main difference is that in the forward pass the sum is put through a non-linear function, whereas in the backward pass it is simply multiplied by $y_j(1 - y_j)$.

One way of using $\partial E/\partial w$ is to change the weights after every input–output case. This has the advantage that no separate memory is required for the derivatives. An alternative scheme, which was used in the research reported here, is to accumulate $\partial E/\partial w$

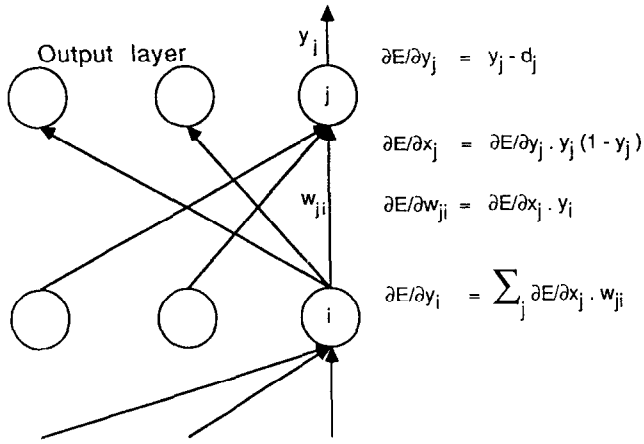

Figure 1. The steps involved in computing $\partial E/\partial y$ for the intermediate layers of a multilayer network. The backward pass starts at the top of the Figure and works downwards. Not all the connections are shown.

over all the input–output cases (or over a large number of them if it is not a finite set) before changing the weights. The advantage of this second method is that it allows the value of E in equation (3) to be monitored for a given set of weights.

The simplest version of gradient descent is to change each weight by a proportion, ε, of the accumulated $\partial E/\partial w$,

$$\Delta w = -\varepsilon \frac{\partial E}{\partial w}$$

The speed of convergence of this method can be improved by making use of the second derivatives, but the algorithm is much more complex and not as easily implemented by local computations in parallel hardware. The above method can be improved significantly without sacrificing the simplicity and locality by using an *acceleration* method in which the current gradient is used to modify the velocity of the point in weight space instead of its position.

$$\Delta w(t) = -\varepsilon \frac{\partial E}{\partial w(t)} + \alpha \Delta w(t-1)$$

where t is incremented by 1 for each sweep through the whole set of input–output cases (called an *epoch*), and α is an exponential decay factor between 0 and 1 (called *momentum*) that determines the relative contribution of the current and past gradients to the weight change. We call α the momentum because that has appropriate physical connotations, even though it is not a precise analogy. The correct analogy is to viscosity. Equation (7) can be viewed as describing the behaviour of a ball-bearing rolling down the error surface when the whole system is immersed in a liquid with viscosity determined by α. A similar acceleration method was used for estimating weights by Amari (1967). Its effectiveness is discussed in Section 5.

The learning procedure is entirely deterministic, so if two units within a layer start off with the same connectivity and weights, there is nothing to make them ever differ from each other. This symmetry is broken by starting with small random weights.

3. Learning to discriminate noisy signals

Rumelhart *et al.* (1986*a*) illustrated the performance of the learning procedure on many different, simple tasks. A further example is given here which demonstrates that the procedure can construct sets of filters that are good at discriminating between rather similar signals in the presence of a lot of noise. An artificial task (suggested by Alex Waibel) was used which was intended to resemble a task that arises in speech recognition. The authors are currently working on extending this approach to real speech data.

The input is a synthetic spectrogram that represents the energy in six different frequency bands at nine different times. Figure 2 shows examples of spectrograms with no random variation in the level of the signal or the background, as well as the same spectrograms with added noise. The problem is to decide whether the signal is simply a horizontal track or whether it rises at the beginning. There is variation in both the frequency and onset time of the signal.

It is relatively easy to decide on the frequency of the horizontal part of the track, but it is much harder to distinguish the "risers" from the "non-risers" because the noise in the signal and background obscures the rise. To make the distinction accurately, the network needs to develop a set of filters that are carefully tuned to the critical differences. The filters must cover the range of possible frequencies and onset times, and when several different filters fit quite well, their outputs must be correctly weighted to give the right answer. It should be noted, however, that due to effects such as time warping, the corresponding problem in real speech data is considerably more difficult. Possible ways of extending this approach to deal with more realistic data are described in Section 8.

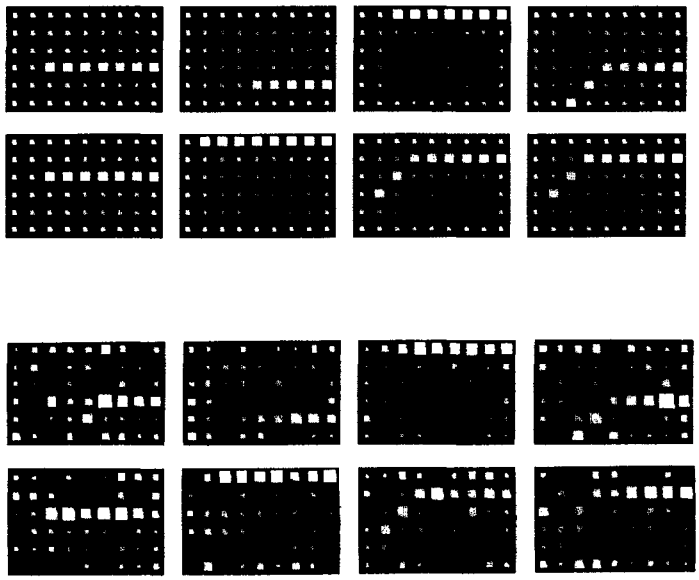

Figure 2. Synthetic spectrograms (a) without noise, and (b) with added noise. Each horizontal track represents energy in a particular frequency over time.

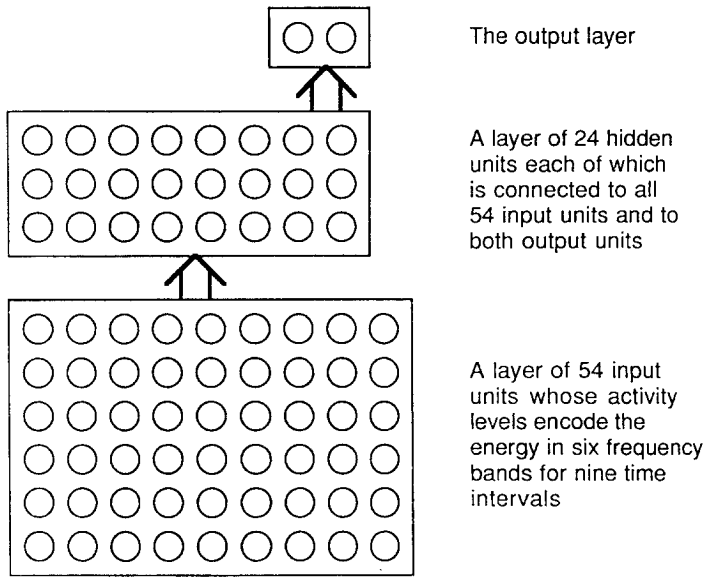

Figure 3. The network used for disciminating patterns like those in Figure 2(b).

The network used comprised three layers, as shown in Fig. 3. Initially the network was trained by repeatedly sweeping through a fixed set of 1000 examples, but it learned to use the structure of the noise to help it discriminate the difficult cases, and so it did not generalize well when tested on new examples in which the noise was different. It was decided, therefore, to generate a new example every time so that, in the long run, there were no spurious correlations between the noise and the signal. Because the network lacks a strong *a priori* model of the nature of the task, it has no way of telling the difference between a spurious correlation caused by using too small a sample and a systematic correlation that reflects the structure of the task.

Examples were generated by the following procedures.

(1) Decide to generate a riser or a non-riser with equal probability.
(2) If it is a non-riser pick one of the six frequencies at random. If it is a riser pick one of the four highest frequencies at random (the final frequency of a riser must be one of these four because it must rise through two frequency bands at the beginning).
(3) Pick one of five possible onset times at random.
(4) Give each of the input units a value of 0·4 if it is part of the signal and a value of 0·1 if it is part of the background. We now have a noise-free spectrogram of the kind shown in Figure 2(a).
(5) Add independent Gaussian noise with a mean of 0 and standard deviation of 0·15 to each unit that is part of the signal. Add independent Gaussian noise with a mean of 0 and standard deviation of 0·1 to the background. If any unit now has a negative activity level, set its level to 0.

The weights were modified after each block of 25 examples. For each weight, the values of $\partial E/\partial w$ were summed for all 25 cases and the weight increment after block t was given by equation (7). For the first 25 blocks we used $\varepsilon = 0{\cdot}005$ and $\alpha = 0{\cdot}5$. After this the weights changed rather slowly and the values were raised to $\varepsilon = 0{\cdot}07$ and $\alpha = 0{\cdot}99$. It was found that it is generally helpful to use more conservative values at the beginning because the gradients are initially very steep and the weights tend to overshoot. Once the weights have settled down, they are near the bottom of a ravine in weight space, and high values of α are required to speed progress along the ravine and to damp out oscillations across the ravine. The validity of interpreting characteristics of weight space in terms of structures such as ravines is discussed in Section 5.

In addition to the weight changes defined by equation (7) each weight was also decremented by hw each time it was changed, where h is a coefficient that was set at 0·001% for this simulation. This gives the weights a very slight tendency to decay towards zero, eliminating weights that are not doing any useful work. The hw term ensures that weights for which $\partial E/\partial w$ is near zero will keep shrinking in magnitude. Indeed, at equilibrium the magnitude of a weight will be proportional to $\partial E/\partial w$ and so it will indicate how important the weight is for performing the task correctly. This makes it much easier to understand the feature detectors produced by the learning. One way to view the term hw is as the derivative of $\frac{1}{2}hw^2$, so the learning procedure can be viewed as a compromise between minimizing E and minimizing the sum of the squares of the weights.

Figure 4 shows the activity levels of the units in all three layers for a number of examples chosen at random after the network has learned. Notice that the network is normally confident about whether the example is a riser or a non-riser, but that in difficult cases it tends to "hedge its bets". This would provide more useful information to a higher level process than a simple forced choice. Notice also that for each example, most of the units in the middle layer are firmly off.

The network was trained for 10 000 blocks of 25 examples each. After this amount of experience the weights are very stable and the performance of the network has ceased to improve. If the network is forced to make a discrete decision by interpreting the more active of the two output units as its response, it gives the "correct" response 97·8% of the time. This is better than a person can do using elaborate reasoning, and it is probably very close to the optimal possible performance. No system could be 100% correct because the very same data can be generated by adding noise to two different underlying signals, and hence it is not possible to recover the underlying signal from the data with certainty. The best that can be done is to decide which category of signal is most likely to have produced the data and this will sometimes not be the category from which the data was actually derived. For example, with the signal and noise levels used in this example, there is a probability of about 1·2% that the two crucial input units that form the rising part of a riser will have a smaller combined activity level than the two units that would form part of a non-riser with the same onset time and same final frequency. This is only one of several possible errors.

Figure 5 shows the filters that were learned in the middle layer. The ones that have positive weights to the "riser" output unit have been arranged at the top of the Figure. Their weights are mainly concentrated on the part of the input that contains the critical information, and between them they cover all the possible frequencies and onset times.

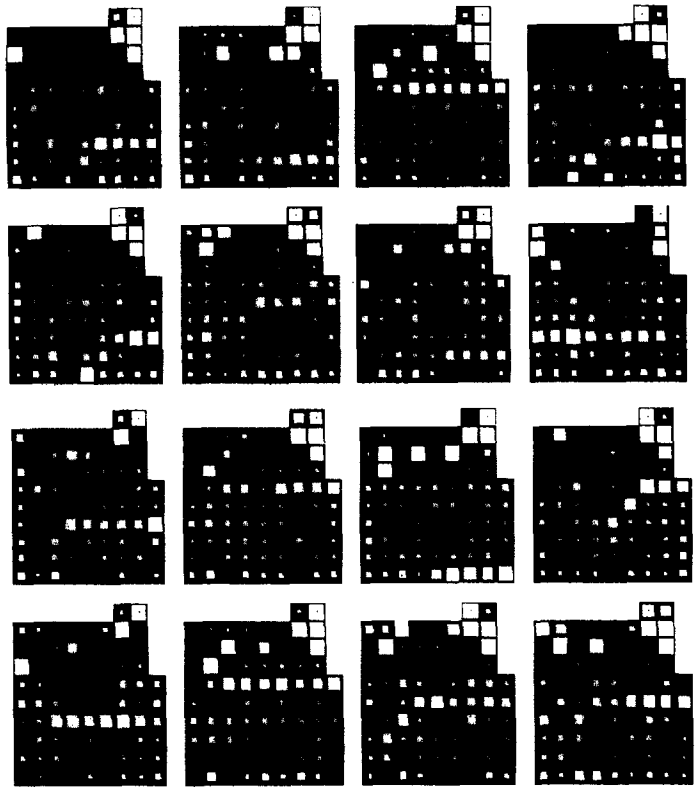

Figure 4. Activity levels of units in all three layers for a number of cases.

Notice that each filter covers several different cases and that each case is covered by several different filters. The set of filters form an "ecology" in which each one fills a niche that is left by the others. Using analytical methods it would be very hard to design a set of filters with this property, even if the precise characteristics of the process that generated the signals were explicitly given. The difficulty arises because the definition of a good set of filters is one for which there exists a set of output weights that allows the correct decision to be made as often as possible. The input weights of the filters canot be designed without considering the output weights, and an individual filter cannot be designed without considering all the other filters. This means that the optimal value of each weight depends on the value of every other weight. The learning procedure can be viewed as a numerical method of solving this analytically intractable design problem. Current analytical investigations of optimal filters (Torre & Poggio, 1986) are very helpful in providing understanding of why some filters are the way they are, but we can probably improve our understanding of the variety and robustness of possible filters by observing numerical solutions, particularly when a number of different filters must work together for making difficult discriminations. Therefore, even if back-propagation is implausible as a biological learning mechanism, it may still be useful for exploring the space of possible filter designs.

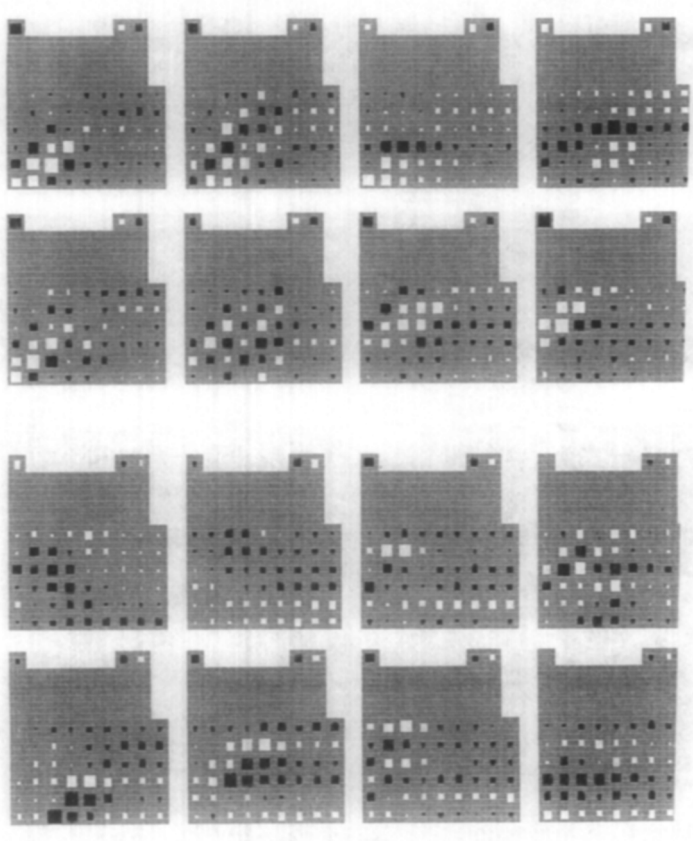

Figure 5. Some of the filters learned by the middle layer. Each weight is represented by a square whose size is proportional to the magnitude of the weight and whose color represents the sign of the weight (white for positive, black for negative).

4. Encoding edges in an image

To illustrate further the ability of the back-propagation procedure to discover efficient filters, we chose a task in which edges must be extracted from an intensity array. The 225 input units implicitly encode an edge by explicitly encoding the intensities in a 15×15 image (see Fig. 6). The output units explicitly encode the edge: an active output unit represents an edge with a particular orientation θ and a particular perpendicular distance ρ from the center of the image. Between the input and output units is a layer of 18 hidden units which act as a narrow bandwidth channel (like the optic nerve). The hidden units must learn to transmit information about the current intensity array in a form that makes it easy for the network to produce the correct output encoding. We were interested in what coding the learning procedure would discover.

All the possible infinite straight edges that could appear in the image can be represented as points in the two-dimensional space formed by the parameters ρ and θ shown in Fig. 7. To cover this space of possible edges with only 36 output units we used a standard sampling method. The output units correspond to a 6×6 grid of points in the

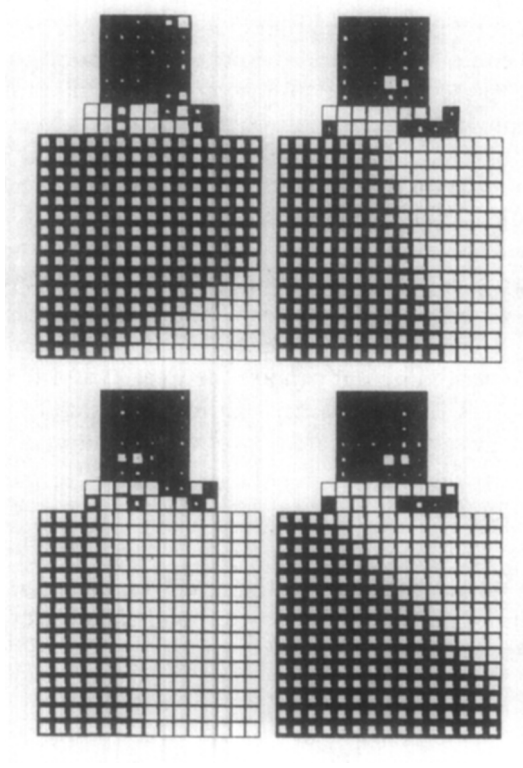

Figure 6. The activity levels of all units for a number of different inputs. The structure of the network is similar to that used to recognize synthetic spectrograms.

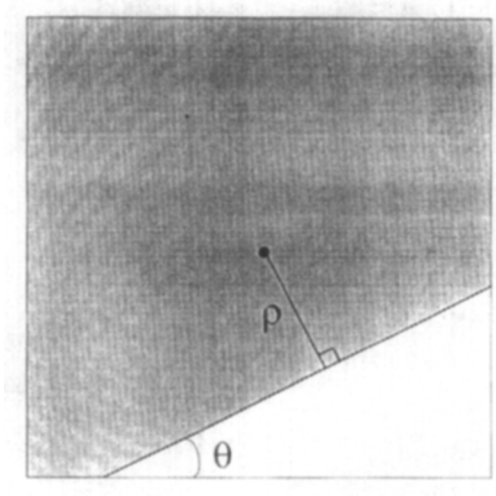

Figure 7. Representation of an edge in terms of its perpendicular distance ρ from the origin and its angle θ with the horizontal.

space. A given edge corresponds to a point that typically fails to fall exactly on one of the 36 grid points. We could have chosen the nearest grid point to represent the edge, but this is an unstable representation because a very small change in the edge can lead to a big change in its representation. A more robust method is to convolve the point with a Gaussian blurring function and then to sample this blur at the grid points. The sample values are the representation of the edge. This method has the additional advantage that it represents the orientation and position of an edge to a greater accuracy than the separation of the grid points.

To train the network, a freshly generated edge was used on every trial. To generate an edge, we first chose an orientation between 0° and 360° and a distance from the center of the image between -6 and 6. We then chose the intensities d and l of the dark and light sides of the edge: d was chosen at random between 0 and 0·6 and then l was chosen between $d+0{\cdot}4$ and 1·0. This guaranteed an intensity step of at least 0·4. Finally, the edge was smoothed by setting the intensity of each input unit to be:

$$d+\frac{l-d}{1+e^{-x/0{\cdot}25}}$$

where x is the perpendicular distance of the pixel from the edge (x is positive if the pixel lies on the light side of the edge). This corresponds, almost exactly, to convolving the image with a Gaussian blurring function.

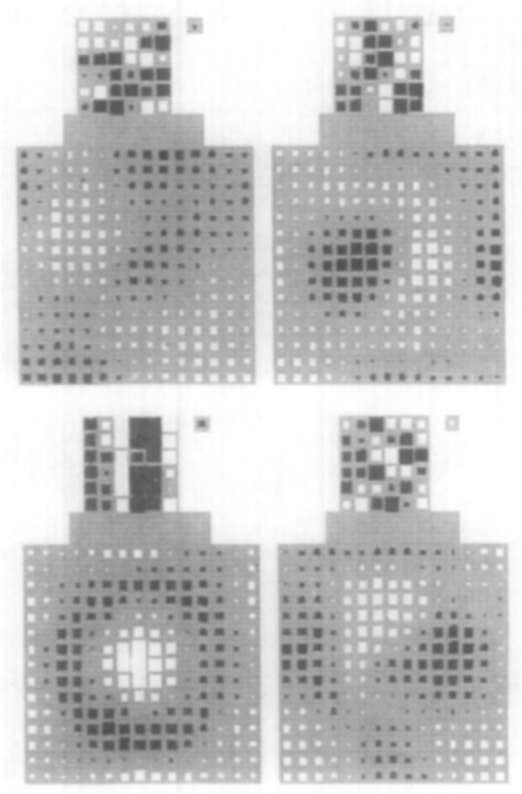

Figure 8. Some of the filters learned by the middle layer.

After prolonged learning, the network performed the task almost perfectly. The weights to and from some of the hidden units are shown in Fig. 8. Notice that the hidden units do not look like edge detectors. Each edge is detected by activity in many hidden units and each hidden unit contributes to the detection of many edges. Many of the receptive fields resemble the "on-center-off-surround" or "off-center-on-surround" fields that are found for retinal ganglion cell. The receptive fields are not localized to small parts of the image because only one edge is present at a time, so there is no reason not to combine information from all over the image.

This section and the previous one present two fairly small examples of how the learning procedure is able to construct efficient sets of filters. Determining the relevance of these examples to problems of more realistic size requires a better understanding of the properties of the learning procedure that affect how the learning time scales as the size of the network and the task increase. The following three sections are devoted to analyzing these properties.

5. Characteristics of weight space

As mentioned in Section 1, a useful way to interpret the operation of the learning procedure is in terms of movement down an error surface in a multi-dimensional *weight space*. For a network with only two connections, the characteristics of the error surface for a particular task are relatively easy to imagine by analogy with actual surfaces which curve through three-dimensional physical space. The error suface can be described as being composed of hills, valleys, ravines, ridges, plateaus, saddle points, etc. In the learning procedure, the effects of the weight-change step (ε) and momentum (α) parameters have natural interpretations in terms of physical movement among such formations. Unfortunately, for more useful networks with hundreds or thousands of connections it is not clear that these simple intuitions about the characteristics of weight space are valid guides to determining the parameters of the learning procedure.

One way to depict some of the structure of a high-dimensional weight space is to plot the error curves (i.e. cross-sections of the error surface) along significant directions in weight space and compare them to error curves along random directions. The collection of curves represents the error surface "collapsed" onto two dimensions. While such a graph gives a far from complete picture of weight space, it may give us a more direct way to test the effects of different learning parameters as well as clarify our interpretation of movement in weight space in terms of simple three-dimensional constructs.

As an example, we present a few collapsed error surface graphs of a simple learning problem at various points in the search for a good set of weights. The problem we will consider is learning the association of 20 pairs of random binary vectors of length 10. The procedure will operate on a three-layered network, with 10 input units, 10 hidden units, and 10 output units. Each input unit is connected to each hidden unit, and each hidden unit is connected to each output unit. Taking into account the connections from a permanently active unit to the hidden and output units (used to encode thresholds), the network has a total of 220 connections. This relatively simple task was chosen because it can be learned quickly enough to allow extensive exploration of the error surface. Error surfaces of more complex tasks are qualitatively similar.

Each curve in a graph is generated by: (1) choosing a direction in weight space; (2) changing the weights in the network by some factor times the unit vector representing that direction; and (3) plotting the error produced by the network with the

modified weight values. In addition to a number of random directions (dotted curves), two significant directions are shown (solid curves): the direction of maximum gradient and the direction of the next weight step (integrated gradient). Each curve is labeled on the right with its angle (in degrees) from the direction of maximum gradient. An asterisk marks the current position in weight space, and a vertical bar marks the next position.

Figures 9–12 show collapsed error surface graphs for the problem above at points throughout the operation of the learning procedure. Graphs are presented for the first 10 epochs, as well as for epochs 25, 50, 75 and 107 (when a solution is reached). For the example, $\varepsilon = 0{\cdot}1$ and initially $\alpha = 0{\cdot}5$.

During the first few epochs, the procedure repeatedly reaches a minimum along the direction of weight change and must use the influence of the calculated gradient information to change directions. As momentum contributes to maintaining movement along a particular direction, it is important in these early stages that momentum be low, so as not to dominate the new gradient information. The effect of having momentum too high at the start of learning will be illustrated in later graphs. It is not until epoch 9 or 10 (Fig. 11) that continued movement along the last weight-change direction would be beneficial.

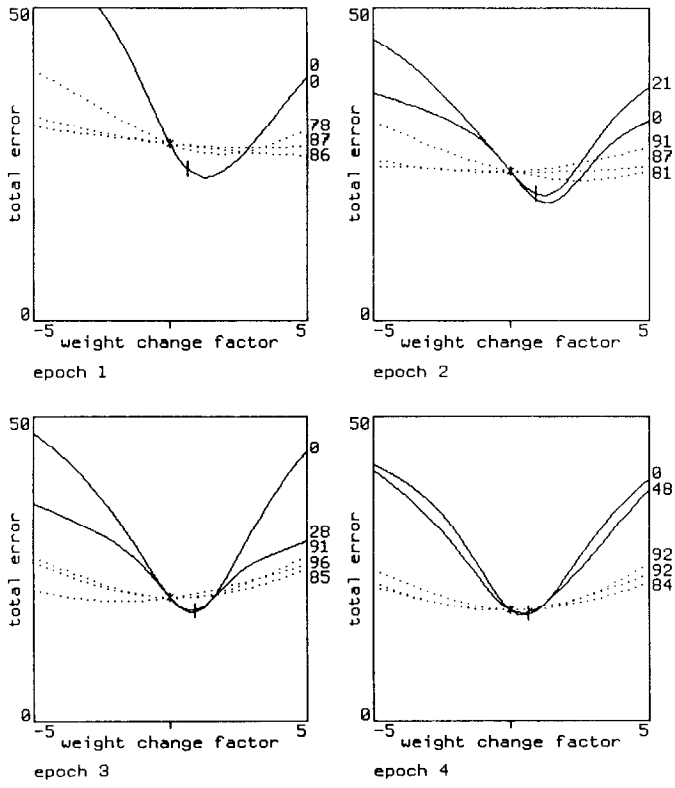

Figure 9. Collapsed error surfaces for epochs 1–4.

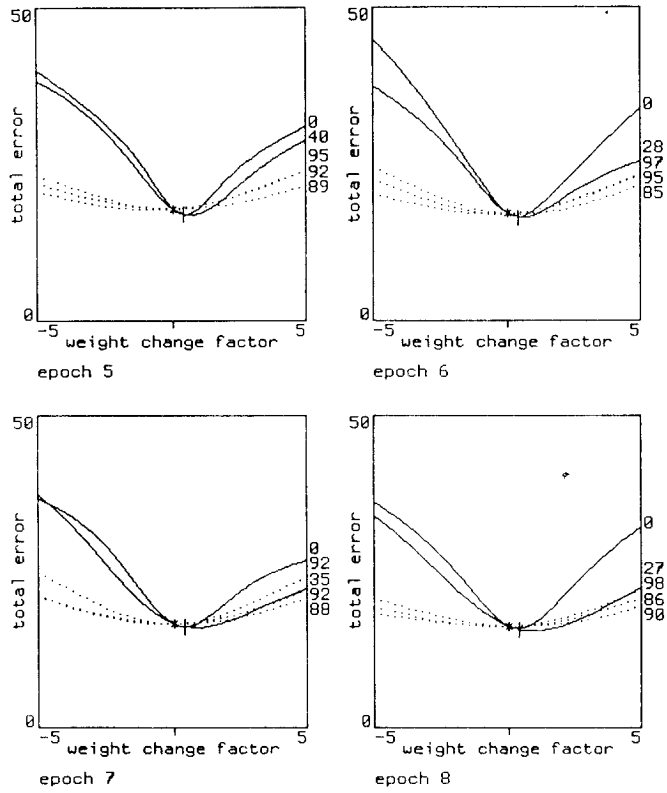

Figure 10. Collapsed error surfaces for epochs 5–8.

By epoch 25, the directions of maximum gradient and integrated gradient are practically identical and monotonically decreasing over a relatively long distance in weight space. In contrast, the error curve in each of the random directions slopes upwards almost immediately as we move away from the current point. The intuitive interpretation is that the learning procedure is moving slowly along the botom of a *ravine* in weight space. Because the correspondence of the directions of maximum gradient and integrated gradient, increasing momentum would speed up movement through the ravine without causing divergent oscillations onto the walls of the ravine. Accordingly, momentum (α) was increased to 0·95 at this point.

While the integrated gradient (and hence the direction of weight change) is still pointed along the bottom of the ravine at epoch 50, the direction of maximum gradient now points somewhat across the ravine. Without momentum, the learning procedure would "slosh" from side to side along the walls of the ravine. The high momentum both dampens this oscillatory contribution and maintains movement along the most effective direction. This effect of momentum becomes increasingly important during the later stages of learning, as is evident at epoch 75 (Fig. 12), and finally at epoch 107, when a solution is reached.

These graphs suggest that momentum should be set initially rather low, and only raised when the learning procedure has settled on a stable direction of movement. In

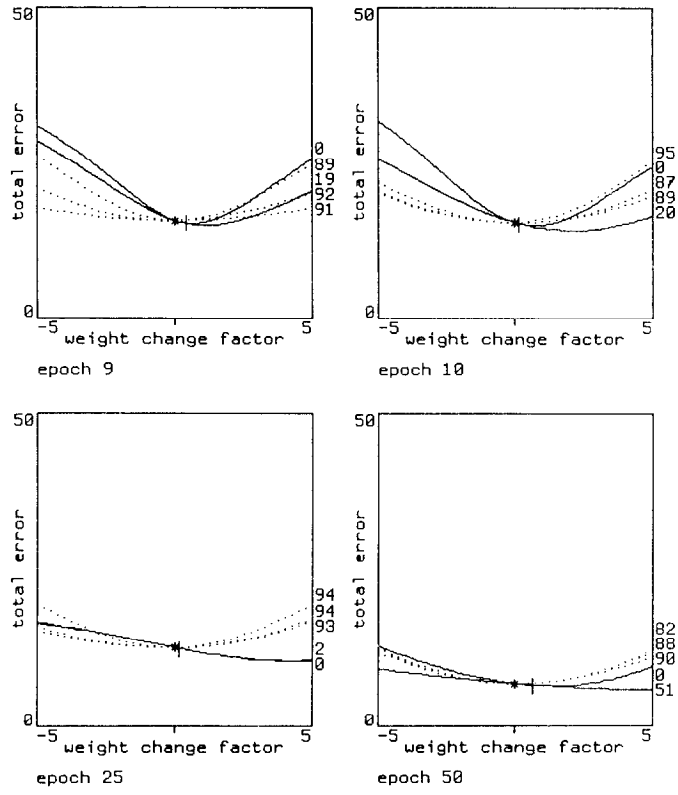

Figure 11. Collapsed error surfaces for epochs 9–50.

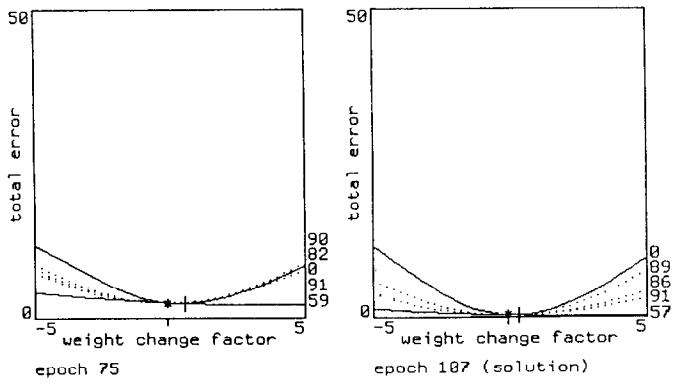

Figure 12. Collapsed error surfaces for epochs 75 and 107 (solution).

order to illustrate the behavior of the procedure when this rule is violated, Fig. 13 presents the collapsed error surface graphs of the first four epochs of a run with momentum set initially to 0·9 (instead of 0·5). The first epoch is fine, since there is no integrated gradient to affect the weight change. However, by epoch 3 the overly high momentum has caused the procedure to overshoot the minimum of the original weight-change direction and *increase* the total error over the last position in weight space. This usually means that hidden units have activities very near zero or one for all input vectors, so they have very small error derivatives and recover their sensitivity very slowly.

In the first example run, almost 50 epochs were required to reduce the total error from just over 5·0 to the solution criterion (near 0·0), even with very high momentum (0·95). This suggests the possibility of increasing the size of each weight step to speed up the later stages of learning when high momentum has essentially fixed the direction of weight change. In fact, increasing ε does significantly reduce the number of epochs to solution, as long as the weight step is not so large that the procedure drastically changes direction. However, because a number of changes of direction are required in the early stages of learning, the weight step must not be too large initially. Figure 14 illustrates the divergent behavior that results at the beginning of a run with ε set to 0·5 (instead of 0·1). The first step drastically overshoots the minimum along the direction of maximum gradient. Successive steps, though smaller, are still too large to produce coherent movement.

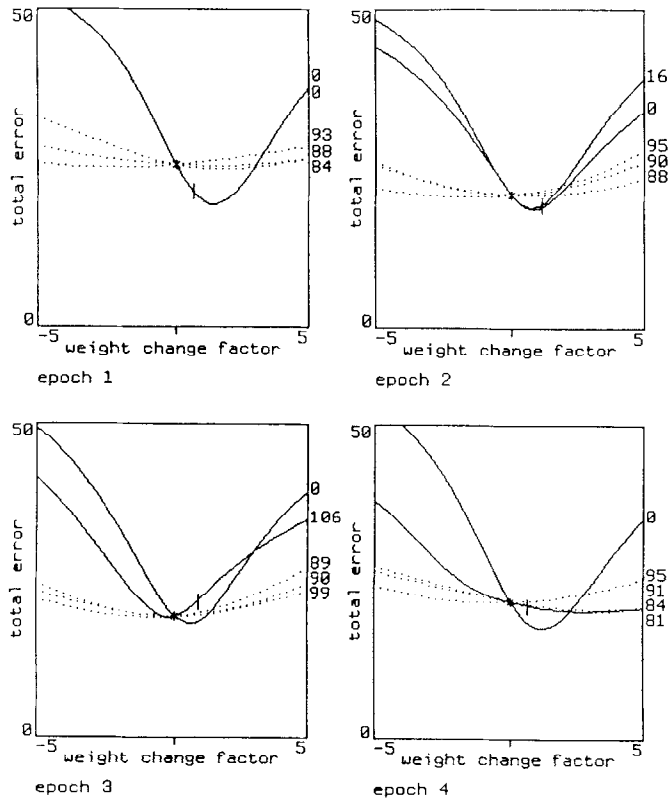

Figure 13. Collapsed error surfaces for the first four epochs of a run beginning with high momentum ($\alpha = 0{\cdot}9$).

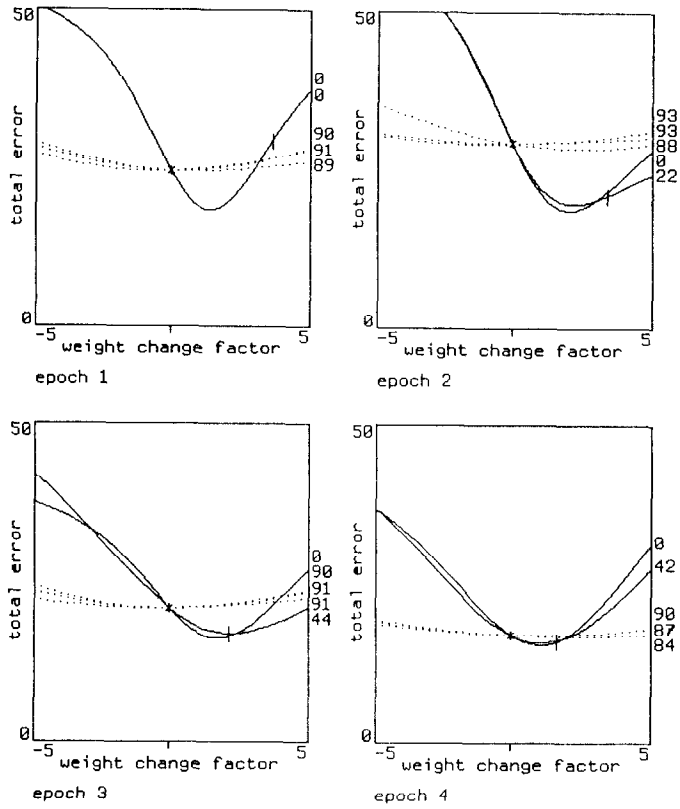

Figure 14. Collapsed error surfaces for the first four epochs of a run beginning with a large weight step ($\varepsilon = 0{\cdot}5$).

6. How the learning time scales

Small-scale simulations can only provide insight into the behavior of the learning procedure in larger networks if there is information about how the learning time scales. Procedures that are very fast for small examples but scale exponentially are of little interest if the goal is to understand learning in networks with thousands or millions of units. There are many different variables that can be scaled:

(1) the number of units used for the input and output vectors and the fraction of them that are active in any one case;
(2) the number of hidden layers;
(3) the number of units in each hidden layer;
(4) the fan-in and fan-out of the hidden units;
(5) the number of different input–output pairs that must be learned, or the complexity of the mapping from input to output.

Much research remains to be done on the effects of most of these variables. This section only addresses the question of what happens to the learning time when the number of hidden units or layers is increased but the task and the input–output encoding remain constant. If there is a fixed number of layers, we would like the learning to go faster if the network has more hidden units per layer.

6.1. Experiments

Unfortunately, two initial experiments showed that increasing the number of hidden units or hidden layers slowed down the learning. (Learning time is measured by the number of sweeps through the set of cases that are required to reach criterion. The extra time required to simulate a larger network on a serial machine is not counted.) In the first experiment, two networks were compared on the identical task: learning the associations of 20 pairs of random binary vectors of length 10. Each network consisted of three layers, with 10 input units and 10 output units. The first (called a 10–10–10 network) had 10 hidden units receiving input from all 10 input units and projecting to all 10 output units; the second (called a 10–100–10 network) had 100 hidden units fully interconnected to both input and output units. Twenty runs of each network on the task were carried out, with $\varepsilon = 0{\cdot}1$ and $\alpha = 0{\cdot}8$.

The results of this first experiment made it clear that the learning procedure in its current form does not scale well with the addition of hidden units: the 10–10–10 network took an average of 212 epochs to reach solution, while the 10–100–10 network took an average of 531 epochs.

The second experiment involved adding additional layers of hidden units to a network and seeing how the different networks compared on the same task. The task was similar to the one above, but only 10 pairs of vectors were used. Each network has 10 input units fully interconnected to units in the first hidden layer. Each hidden layer had 10 units and was fully interconnected to the following one, with the last connected to the 10 output units. Networks with one, two and four layers of hidden units were used. Twenty runs of each network were carried out, with $\varepsilon = 0{\cdot}1$ and $\alpha = 0{\cdot}8$.

The results of the second experiment were consistent with those of the first: the network with a single hidden layer solved the task in an average of 100 epochs; with two hidden layers it took 160 epochs on average, and with four hidden layers it took an average of 373 epochs to solve the task.

6.2. Unit splitting

There is one method of introducing more hidden units which has no effect on the performance of the network. Each hidden unit in the old network is replaced by n identical hidden units in the new network. The input weights of the new units are exactly the same as for the old unit, so the activity level of each new unit is exactly the same as for the old one in all circumstances. The output weights of the new units are each $1/n$ of the output weights of the old unit, and so their combined effect on any other unit is exactly the same as the effect of the single old unit. Figure 15 illustrates this invariant unit-splitting operation. To ensure that the old and new networks remain equivalent even after learning, it is necessary for the outgoing weights of the new units to change by $1/n$ times as much as the outgoing weights of the old unit. Therefore, a different value of

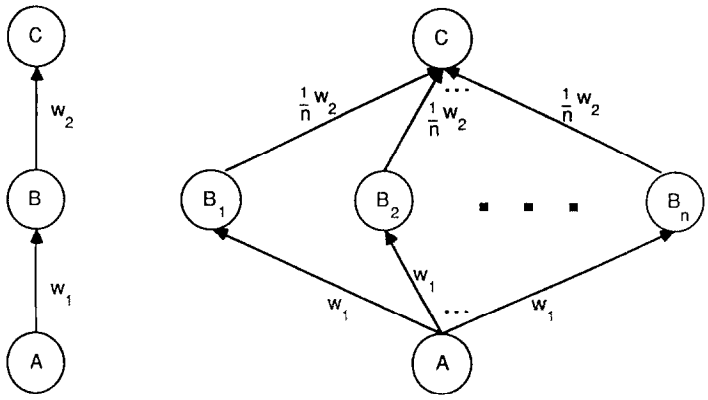

Figure 15. These two networks have identical input–output functions. The input–output behavior is invariant under the operation of splitting intermediate nodes, provided the outging weights are also decreased by the same factor.

ε must be used for the incoming and outgoing weights, and the ε for a connection emanating from a hidden unit must be inversely proportional to the fan-in of the unit receiving the connection.

6.3. Varying ε with fan-in

The fact that it is possible to increase the number of hidden units and connections in a network by a factor of n without affecting the performance of the learning procedure suggests a way to improve how well it scales. Critical to the success of the unit-splitting process is dividing the weight change step (ε) by n for weights on replicated connections. This ensures that the weight changes on incoming connections to a unit will cause the same change in total input for a given amount of error produced by the unit, even though n times as many connections are contributing to the input change. The equivalent procedure in a normal network would be to set the effective weight step for a connection, ε_{ji}, to be inversely proportional to the fan-in of the unit receiving input via that connection. Presumably such a modification would also improve the scaling of the learning procedure for networks with non-uniform fan-in.

Empirical observations of the operation of the procedure on different sized nets make it clear that larger networks (with higher fan-ins) require a much smaller value of ε for optimal learning than do smaller networks. If the change in input to a unit is too large, due to an overly ambitious value of ε, the output of the unit may overshoot its optimal value, requiring an input change in the opposite direction during the next epoch. Thus, given the fan-in of units in a network, setting ε too high results in oscillatory behavior and poor learning performance. However, if the effective ε is reduced for connections leading into units with many inputs but not reduced for other conections, this oscillatory behavior can be avoided without slowing down the learning of weights on connections providing input to units with lower fan-in.

A close look at the details of the backward pass of the learning procedure makes it clear why such a modification would be beneficial. Each connection weight w_{ji} is changed

in proportion to the error attributed to the output of unit j, independent of other inputs unit j may receive.

$$\Delta w_{ji} = \varepsilon - \frac{\partial E}{\partial y_j} y_j(1 - y_j) y_i.$$

Hence, the resulting change in total input to unit j,

$$\Delta x_j = \sum_{i=1}^{n} \Delta(w_{ji} y_i)$$

is proportional to n, the fan-in of unit j.

In order to determine if varying ε with fan-in would improve the scaling performance of the learning procedure, the scaling experiment involving the addition of hidden units to a single hidden layer was repeated using values of ε_{ji} inversely proportional to the fan-in of unit j. The constant of proportionality was set at 1·0 so that the 10–100–10 network had an effective ε on the input connections to the output units of 0·01, while the effective ε on the input connections to the hidden units remained at 0·1. It was expected that these more conservative weight change steps would prevent any oscillatory behavior and improve the learning performance.

The results bore out these expectations. The average number of epochs to solution for the 10–100–10 network was reduced from 531 to 121. By varying ε with fan-in, the addition of hidden units *speeded up* the learning by almost a factor of two, rather than slowing it down (recall from Section 6.1 that the 10–10–10 network took 212 epochs on this task). This is not a solution to the entire scaling problem, but it represents a significant improvement in the ability of the learning procedure to handle large, complex networks.

7. Reducing the interactions between the weights

The previous section demonstrated that, by varying ε inversely with fan-in, a fully interconnected network with 100 hidden units can learn a task nearly twice as fast as a similar network with only 10 hidden units. While this manipulation of ε improves the scaling performance of the learning procedure, many presentations of each environmental case are required to learn most tasks, and larger networks still generally take longer to learn than do smaller ones. The above comparison does not tell us what particular characteristics of a network most significantly influence its learning speed, because at least two important factors are confounded: (1) the number of hidden units, and (2) the fan-in of the output units.

However, the learning speed is not necessarily dependent on the *number* of units and connections in a network. This can be seen by considering a network similar to the 10–100–10 network, but in which the layers are not fully interconnected. In particular, the hidden units are partitioned into groups of 10, with each group receiving input from all input units but only projecting to a single output unit. For convenience, we will call this a 10–10of10–10 network. This structure transforms each 10 to 10 mapping into 10 *independent* 10 to 1 mappings, and so reduces the amount of *interaction* between weights on connections leading into the output layer.

7.1. Experiments

In order to investigate the relative effects on learning speed of the number of hidden units, the fan-in of the output units, and the amount of interaction between the weights, we compared the performances of the 10–10–10, 10–100–10, and 10–10of10–10 networks on the task of learning the association of twenty pairs of random binary vectors of length 10. The results of the comparison are summarized in Table I.

As the Table shows, the 10–10of10–10 network solves the task much faster than the 10–10–10 network, although both networks have uniform fan-in and the same number of connections from the hidden layer to the output layer. The 10–10of10–10 network learns more quickly because the states of units in each group of 10 hidden units are constrained only by the desired state of a single output unit, whereas the states of the 10 hidden units in the 10–10–10 network must contribute to the determination of the states of all 10 output units. The reduced constraints can be satisfied more quickly.

However, when ε is varied so that the effects of fan-in differences are eliminated, the 10–10of10–10 network learns slightly slower than the 10–100–10 network, even though both networks have the same number of hidden units and the 10–100–10 network has a much greater amount of interaction between weights. Thus a reduction in the interaction within a network does not always improve its performance. The advantage of having an additional 90 hidden units, some of which may happen to detect features that are very useful for determining the state of the output unit, seems to outweigh the difficulty caused by trying to make each of those feature detectors adapt to ten different masters. One might expect such a result for a task involving highly related environmental cases, but it is somewhat more surprising for a task involving random associations, where there is no systematic structure in the environment for the hidden units to encode. It appears that, when the magnitudes of weight changes are made sensitive to the number of sources of error by varying ε with fan-in, the learning procedure is able to take advantage of the additional flexibility afforded by an increase in the interactions between the weights.

TABLE 1. Comparison of performance of the 10–10–10, 10–100–10, and 10–10of10–10 networks on the task of learning 20 random binary associations of length 10. Data was averaged over 20 runs with $\varepsilon = 0{\cdot}1$ in the fixed ε cases, $\varepsilon_{ji} = 1{\cdot}0/\text{fan-in}_j$ in the variable ε cases, and $\alpha = 0{\cdot}8$ (varying ε has no effect on networks with uniform fan-in, and so the average number of epochs to solution for these conditions is placed in parentheses).

	Number of hidden units	Fan-in of output units	Average No. of epochs to solution	
			Fixed ε	Variable ε
10–10–10	10	10	212	(212)
10–100–10	100	100	531	121
10–10of10–10	100	10	141	(141)

7.2. Very fast learning with no generalization

Some insight into the effects of adding more hidden units can be gained by considering the extreme case in which the number of hidden units is an exponential function of the number of input units. Suppose that *binary* threshold units are used and that the biases and the weights coming from the input units are fixed in such a way that exactly one hidden unit is active for each possible input vector. Any possible mapping between input and output vectors in a single pass can now be learned. For each input vector, there is one active hidden unit, and only the signs of the weights from this hidden unit to the output units need be set. If each hidden unit is called a "memory location" and the signs of its outgoing weights are called its "contents", this is an exact model of a standard random-access memory.

This extreme case is a good illustration of the trade-off between speed of learning and generalization. It also suggests that if fast learning is required, the number of hidden units should be increased and the proportion of them that are active decreased.

8. Applications to speech recognition

It is clearly tempting to try to apply the back-propagation learning procedure to real speech recognition, because it has the ability to discover a hierarchy of non-linear feature detectors automatically. It seems more promising than earlier connectionist approaches like Boltzmann Machines (Prager, Harrison & Fallside, 1986) because it learns much faster and produces much better final performance. Several different research groups are now applying back-propagation to real speech recognition and so good data on its effectiveness will soon be available. We confine ourselves here to a few cautionary remarks about the difficulties that are likely to be encountered in this application.

One approach is to use, as input, a portion of a spectrogram that contains a known phoneme which has already been segmented out and temporally aligned (Peeling, Moore & Tomlinson, 1986). The example given earlier in this paper can be seen as an idealized version of this where the temporal alignment is somewhat variable. The network is trained on many examples of various phonemes, and then tested to see whether it can correctly identify the phonemes in new examples. The first problem with this approach is that it is bound to require an immense amount of accurately labeled training data to give good generalization. The network typically requires thousands of weights to allow it to capture the relevant structure, and so the training data must contain many thousands of bits in order to constrain these weights sufficiently to give good generalization. If the number of bits in the training data does not significantly exceed the number of degrees of freedom in the model, the network can use a form of "rote-learning" or "table look-up" to learn the training cases—it can find a setting of the weights that gives the correct output for all the training examples without capturing the underlying regularities of the task. Unfortunately, each training example only contains a few bits because, for a supervised learning procedure like back-propagation, the information in an example is the number of bits it takes to specify the correct *output*. So there is a severe practical problem in getting enough data. The results of Prager *et al.* (1986) confirm that generalization is poor when a general learning procedure is used to fit a model that has more degrees of freedom than there are bits in the training set.

One reason why this first approach requires so much data is that there is very little

prior knowledge built into the network. If the network starts with small random weights, and if each unit in the first layer of hidden units receives input from every cell in the spectrogram, the network would learn just as well if the cells were put through a fixed, random permutation before being presented as input. In other words, the network has no prior expectations that adjacent time frames or adjacent frequences are likely to be more relevant to each other than non-adjacent ones. This lack of prior expectation means that the learning is searching a huge space of possible filters most of which are almost certainly useless. Many people have suggested that the amount of training data required could be reduced by using an architecture that omits irrelevant connections, or by starting with reasonable hand-coded feature detectors instead of random weights.

A second approach, which we feel is more promising because it does not demand such accurate temporal alignment in the training data, is to use an iterative version of the back-propagation learning procedure. This version, which is explained in Appendix I, works for networks that have recurrent connections and are run for many time steps. The network receives new input at each time step, and it can either receive error-terms at each time step or receive the error after the final iteration. Using an iterative network, it is no longer necessary to turn many time frames into a single, spatially laid out input vector. Instead, the network can receive the time frames one at a time. Because the very same set of weights is applied at each time step, the network naturally captures the fact that the same kind of event may occur at different times. It does not have to learn to recognize the event separately for each possible time of occurrence as it would if the temporal input was laid out spatially. Thus, an important symmetry of the task is built into the iterative approach and does not need to be learned. Kevin Lang (pers. comm.) and Watrous & Shastri (1986) have reported some promising initial experiments using variants of this approach.

8.1. Unsupervised learning

We suspect that back-propagation may work best when it is preceded by a pre-coding stage that uses unsupervised connectionist learning procedures to reduce the bandwidth of the data or to make important features more explicit. There is a great deal of structure in speech and it is far from clear that the best way to find this structure is by attempting to predict phoneme labels or other similar categories. It is much easier to provide large training sets for an unsupervised procedure that simply looks for higher-order statistical structure in the input, because the data does not need to be labeled. Using an unsupervised procedure it may be possible to *discover* categories like phonemes (or even words) without any supervision, because these categories account for the higher-order statistical structure in the input. Pearlmutter & Hinton (1986) describe one such learning procedure. Kohonen, Makisara & Saramaki (1984) describe an interesting application of another unsupervised procedure, competitive learning, to the task of pre-coding.

It is also possible to use a form of back-propagation for unsupervised learning by structuring the task as an "encoder" problem (Ackley, Hinton & Sejnowski, 1985). If the output of a multilayer network is required to be the same as the input, the middle layers must learn to encode sufficient information about the input vector to be able to reconstruct it as the output. So the central layer must form an invertible code. By constricting this layer to a small number of units, it is possible to force the network to produce a compact invertible code that can be used as the input to later processes. The

method bears some resemblance to principle components analysis, but it works with non-linear units and so can produce much more complex encodings, especially in a network with many layers between the input and the output. Elman & Zipser (1987) have shown that this method can be used on the raw sample values of the speech wave to produce a greatly compressed encoding that nevertheless contains enough information to produce high-quality speech.

A promising variation of this approach is to use a set of adjacent time frames as input and the next time frame as the required output. The activity levels of the hidden units would then be "non-linear predictive coefficients". It remains to be seen whether these coefficients would be any more useful for recognition than LPC ones.

This research was supported by contract N00014-86-K-00167 from the Office of Naval Research and an R. K. Mellon Fellowship to David Plaut.

References

Ackley, D. H., Hinton, G. E. & Sejnowski, T. J. (1985). A learning algorithm for Boltzmann Machines. *Cognitive Science*, **9**(2), 147–169.

Amari, S. (1967). A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers*, **EC-16**(3), 299–307.

Elman, J. L. & Zipser, D. (1987). *Discovering the Hidden Structure of Speech*. Technical Report 8701, Institute for Cognitive Science, University of California, San Diego.

Hinton, G. E. (1986). Learning distributed representations of concepts. *Proceedings, 8th Annual Conference of the Cognitive Science Society*, pp. 1–12. Amherst, MA.

Kohonen, T., Makisara, K. & Saramaki, T. (1984). Phonotopic maps: insightful representation of phonological features for speech recognition. *IEEE 7th International Conference on Pattern Recognition*, pp. 182–185. Montreal, Canada.

Pearlmutter, B. A. & Hinton, G. E. (1986). G-Maximization: an unsupervised learning procedure for discovering regularities. *Proceedings, Conference on Neural Networks for Computing*, Snowbird, UT.

Peeling, S. M., More, R. K. & Tomlinson, M. J. (1986). The multi-layer perceptron as a tool for speech pattern processing research. *Proceedings, Institute of Acoustics Autumn Conference on Speech and Hearing*.

Prager, R. W., Harrison, T. D. & Fallside, F. (1986). Boltzmann machines for speech recognition. *Computer Speech and Language*, **1**, 3–27.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986*a*). Learning internal representations by error propagation. In D. E. Rumelhart & J. L. McClelland and the PDP research group, (eds). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume I: Foundations*, MIT Press, Cambridge, MA.

Rumelhart, D. E., Hinton, G. E. & Williams, R. J. (1986*b*). Learning representations by back-propagating errors. *Nature*, **323**(9), 533–536.

Torre, V. & Poggio, T. A. (1986). On edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-8**(2), 147–163.

Watrous, R. L. & Shastri L. (1986). *Learning Phonetic Features using Connectionist Networks: An Experiment in Speech Recognition*, Technical Report MS-CIS-86-78, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA.

Appendix I

Iterative networks

The description of the learning procedure in Section 2 dealt only with layered, feed-forward networks, which lack any recurrent connections. In this appendix we describe how the procedure can be extended to handle iterative networks, which have unrestricted connectivity and are run for a number of time steps when presented with input. The development follows that of Rumelhart *et al.* (1986*b*).

Figure A1 shows the equivalence between an iterative network that is run synchronously for three iterations and a layered network in which each layer after the input corresponds to one iteration of the synchronous network. Using this equivalence, it is clear that a layered network can always be constructed that will perform the same computation as an iterative network, provided the number of iterations is known in advance. Both networks have the same delay time between receiving the input and giving the output.

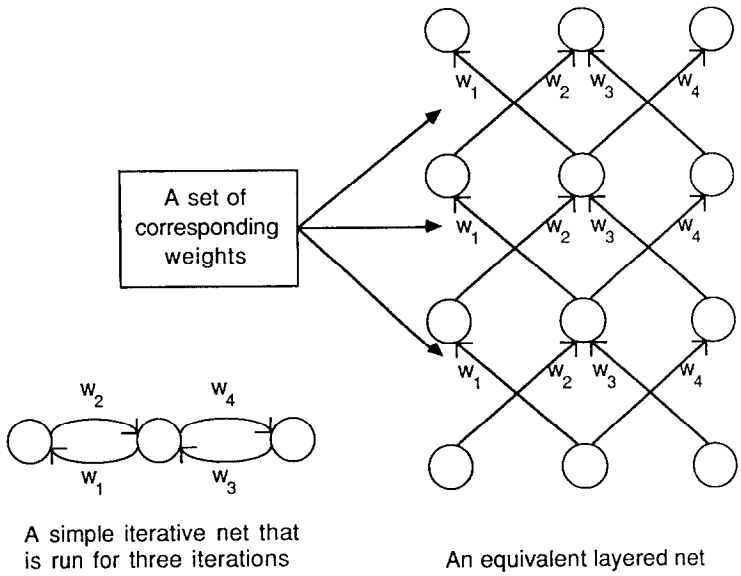

Figure A1. An iterative network and the equivalent layered network.

As we have a learning procedure for layered networks, we could learn iterative computations by first constructing the equivalent layered network, then doing the learning, then converting back to the iterative network. Or we could avoid the construction by simply mapping the learning procedure itself into the form appropriate for the iterative network. Two complications arise in performing this conversion.

(1) In a layered network, the outputs of the units in the intermediate layers during the forward pass are required for performing the backward pass (see equations 4 and 5). So in an iterative network it is necessary to store the output states of each unit that are temporally intermediate between the initial and final states.

(2) For a layered network to be equivalent to an iterative network, corresponding weights between different layers must have the same value, as in Fig. A1. There is no guarantee that the basic learning procedure for layered networks will preserve this property. However, we can easily modify it by averaging $\partial E/\partial w$ for all the weights in each set of corresponding weights, and then changing each weight by an amount proportional to this average gradient. This is equivalent to taking the weight-change vector produced by the basic learning procedure and then projecting it onto the subspace of layered networks that are equivalent to iterative ones.

With these two provisos, the learning procedure can be applied directly to iterative networks and can be used to learn sequential structures. Several examples are given in (Rumelhart *et al.*, 1986*a*).