Hinton, G. E. (1987) Learning translation invariant recognition in a massively parallel network. In Goos, G. and Hartmanis, J., editors, *PARLE: Parallel Architectures and Languages Europe*, pages 1–13, Lecture Notes in Computer Science, Springer-Verlag, Berlin.

LEARNING TRANSLATION INVARIANT RECOGNITION IN A MASSIVELY PARALLEL NETWORK

Geoffrey E. Hinton
Computer Science Department
Carnegie-Mellon University
Pittsburgh PA 15213
U.S.A.

Abstract

One major goal of research on massively parallel networks of neuron-like processing elements is to discover efficient methods for recognizing patterns. Another goal is to discover general learning procedures that allow networks to construct the internal representations that are required for complex tasks. This paper describes a recently developed procedure that can learn to perform a recognition task. The network is trained on examples in which the input vector represents an instance of a pattern in a particular position and the required output vector represents its name. After prolonged training, the network develops canonical internal representations of the patterns and it uses these canonical representations to identify familiar patterns in novel positions.

1 Introduction

Most current models of human cognitive processes are based on logic. They assume that the formal manipulation of symbols is the essence of intelligence and they model cognitive processes by exploiting the sequential symbol processing abilities of conventional, serial computers (Newell, 1980). This approach has been very successful for modeling people's behavior when they are solving symbolic problems or playing intellectual games, and it has also had some success for expert systems which do not require much commonsense knowledge and do not involve complex interactions with the physical world. It has been much less successful for tasks like vision or commonsense reasoning that require rapid processing of large amounts of data or large amounts of stored knowledge.

An alternative approach that is much more compatible with fine-grained parallel computation ~~is~~ based on neural nets. It assumes that human abilities like perceptual interpretation, content-addressable memory, and commonsense reasoning are best understood by considering how computation might be organized in systems like the brain which consist of massive numbers of richly-interconnected but rather slow processing elements. Representations and search techniques which are efficient on serial machines are not necessarily suitable for massively parallel networks, particularly if the hardware is inhomogeneous and unreliable. The neural net approach has tended to emphasize learning from examples rather than programming. So far, it has been much less successful than the logic-based approach, partly because the ideas about representations and the procedures for learning them have been inadequate, and partly because it is very inefficient to simulate massively parallel networks with conventional computers.

The recent technological advances in VLSI and computer aided design mean that it is now much easier to build massively parallel machines and this has led to a new wave of interest in neural net models (Hinton and Anderson, 1981; Feldman and Ballard, 1982; Rumelhart, McClelland et. al., 1986). One very ambitious goal is to produce a general-purpose special-purpose chip (Parker, 1985). After learning, the chip would be special-purpose because the interactions between the processing elements would be specific to a particular task, with all the space and time efficiency which that implies. But before learning the chip would be general-purpose: The very same chip could learn any one of a large number of different tasks by being shown examples of input vectors and required output vectors from the relevant domain. We are still a long way from achieving this goal because the existing learning procedures are too slow, although one general-purpose special-purpose chip based on the Boltzmann machine learning procedure (Ackley, Hinton, and Sejnowski, 1985) has already been laid out (Alspector and Allen, 1987).

This paper describes a recent and powerful "connectionist" learning procedure called back-propagation (Rumelhart, Hinton, and Williams, 1986a, 1986b) and shows that it can overcome a major limitation of an earlier generation of learning procedures such as perceptrons (Rosenblatt, 1962) which were incapable of learning to recognize shapes that had been translated.

2 The network

The network consists of multiple layers of simple, neuron-like processing elements called "units" that interact using weighted connections. Each unit has a "state" or "activity level" that is determined by the input received from units in the layer below. The total input, x_j, received by unit j is defined to be

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

where y_i is the state of the i'th unit (which is in a lower layer), w_{ji} is the weight on the connection from the i'th to the j'th unit and θ_j is the threshold of the j'th unit. Thresholds can be eliminated by giving every unit an extra input line whose activity level is always 1. The weight on this input is the negative of the threshold, and it can be learned in just the same way as the other weights. The lowest layer contains the input units and an external input vector is supplied to the network by clamping the states of these units. The state of any other unit in the network is a monotonic non-linear function of its total input (see figure 1).

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

All the network's long-term knowledge about the function it has learned to compute is encoded by the magnitudes of the weights on the connections. This paper does not address the issue of how to choose an appropriate architecture (i.e. the number of layers, the number of units per layer, and the connectivity between layers).

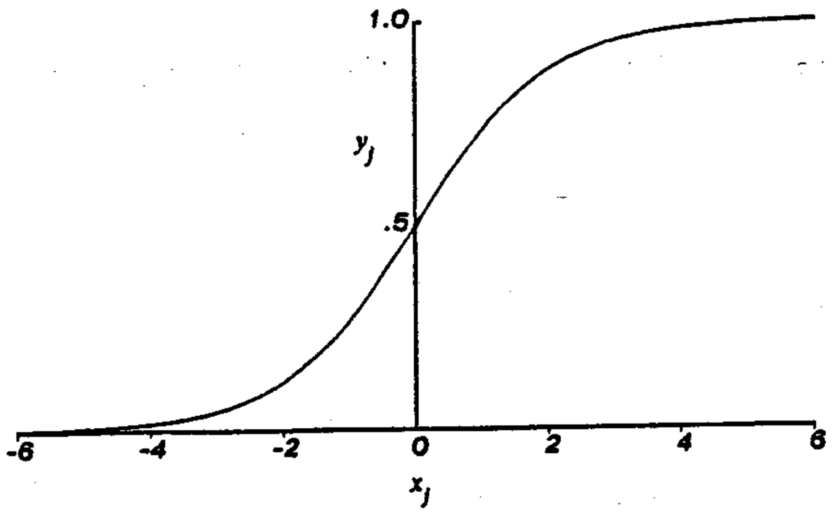

Figure 1: The non-linear transfer function defined in Eq. 2.

3 The learning procedure

Some learning procedures, like the perceptron convergence procedure (Rosenblatt, 1962), are only applicable if the actual or desired states of all the units in the network are already specified. This makes the learning task relatively easy, but it also rules out learning in networks that have intermediate layers between the input and the output. Other, more recent, learning procedures operate in networks that contain "hidden" units (Ackley, Hinton, and Sejnowski, 1985) whose desired states are not specified (either directly or indirectly) by the input or the desired output of the network. This makes learning much harder because the learning procedure must (implicitly) decide what the hidden units should represent. The learning procedure is therefore searching the space of possible representations. Since this is a very large space, it is not surprising that the learning is rather slow.

3.1 A simple LMS learning procedure

If the input units of a network are directly connected to the output units, there is a relatively simple procedure for finding the weights that give the Least Mean Square (LMS) error in the output vectors. The error, with a given set of weights, is defined as:

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

where $y_{j,c}$ is the actual state of output unit j in input-output case c, and $d_{j,c}$ is its desired state.

We can minimize the error measure given in Eq. 3 by starting with any set of weights and repeatedly changing each weight by an amount proportional to $\partial E/\partial w$.

$$\Delta w_{ji} = -\varepsilon\frac{\partial E}{\partial w_{ji}} \tag{4}$$

Provided the weight increments are sufficiently small this learning procedure is guaranteed to find the set of weights that minimizes the error. The value of $\partial E/\partial w$ is obtained by differentiating Eq. 3 and Eq. 1.

$$\frac{\partial E}{\partial w_{ji}} = \sum_c \frac{\partial E}{\partial y_j}\cdot\frac{dy_j}{dx_j}\cdot\frac{\partial x_j}{\partial w_{ji}} = \sum_c (y_j-d_j)\cdot\frac{dy_j}{dx_j}\cdot y_i \tag{5}$$

If the output units are linear, the term dy_j/dx_j is a constant. If the output units use the non-linear transfer function described in Eq. 2, dy_j/dx_j is equal to $y_j(1-y_j)$.

If we construct a multi-dimensional "weight-space" that has an axis for each weight and one extra axis (the "height") that corresponds to the error measure, we can interpret the simple LMS procedure geometrically. For each combination of weights, there is a height (the total error) and these heights form an error-surface. For networks with linear output units and no hidden units, the error surface always forms a concave-upward bowl whose horizontal cross-sections are ellipses and whose vertical cross-sections are parabolas. Since the bowl only has one minimum, gradient descent on the error-surface is guaranteed to find it. If the output units use the non-linear transfer function described in Eq. 2, the bowl is deformed. It still only has one minimum, so gradient descent still works, but the gradient tends to zero as we move far away from the minimum. So gradient descent can become very slow at points in weight-space where output or hidden units have incorrect activity levels near 1 or 0.

3.2 Back-propagation: A multilayer LMS procedure

In a multilayer network it is possible, using Eq. 5, to compute $\partial E/\partial w_{ji}$ for *all* the weights in the network provided we can compute $\partial E/\partial y_j$ for all the units that have modifiable incoming weights. In a system that has no hidden units, this is easy because the only relevant units are the output units, and for them $\partial E/\partial y_j$ is found by differentiating the error function in Eq. 3. But for hidden units, $\partial E/\partial y_j$ is harder to compute. The central idea of back-propagation is that these derivatives can be computed efficiently by starting with the output layer and working backwards through the layers. For each input-output case, c, we first use a forward pass, starting at the input units, to compute the activity levels of all the units in the network. Then we use a backward pass, starting at the output units, to compute $\partial E/\partial y_j$ for all the hidden units. For a hidden unit, j, in layer J the only way it can affect the error is via its effects on the units, k, in the next layer, K , (assuming units in one layer only send their outputs to units in the layer above). So we have

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{dy_k}{dx_k} \cdot \frac{\partial x_k}{\partial y_j} = \sum_k \frac{\partial E}{\partial y_k} \cdot \frac{dy_k}{dx_k} \cdot w_{kj} \tag{6}$$

where the index c has been suppressed for clarity. So if $\partial E/\partial y_k$ is already known for all units in layer K, it is easy to compute the same quantity for units in layer J. The computation performed during the backward pass is very similar in form to the computation performed during the forward pass.

Back-propagation has been tried for a wide variety of tasks (Le Cun, 1985; Rumelhart, Hinton and Williams, 1986b; Sejnowski and Rosenberg, 1986; Elman and Zipser, 1987; Plaut and Hinton, 1987). It rarely gets stuck in poor local minima, even though these can exist for networks with hidden units. A much more serious problem is the speed of convergence. High-dimensional weight spaces typically contain ravines with steep sides and a shallow gradient along the ravine. Acceleration methods can be used to speed convergence in such spaces. The idea of an acceleration method is to use the gradient to change the velocity of the current point in weight space rather than using it to directly determine the change in the position of the point. The method can be implemented by using the following weight update rule

$$\Delta w_{ji}(t) = -\varepsilon \frac{\partial E}{\partial w_{ji}} + \alpha \Delta w_{ji}(t-1) \tag{7}$$

where α is a coefficient between 0 and 1 that determines the amount of damping.

Gradient descent is still very slow for large networks, even using the acceleration method. More powerful techniques that use the second derivative would converge faster but would require more elaborate computations that would be much harder to implement directly in parallel hardware. So despite its impressive performance on relatively small problems, back-propagation is inadequate, in its current form, for larger tasks because the learning time scales poorly with the size of the task and the size of the network.

A second major issue with back-propagation concerns the way it generalizes. If the network must be shown all possible input vectors before it learns a function, it is little more than a table look-up device. We would like to be able to learn a function from many less examples than the complete set of possibilities. The assumption is that most of the functions we are interested in are highly structured. They contain regularities in the relationship between the input and output vectors, and the network should notice these regularities in the training examples and apply the same regularities to predict the correct output for the test cases.

4 Recognizing familiar shapes in novel positions

The task of recognizing familiar shapes in novel positions can be used as a test of the ability of a learning procedure to discover the regularities that underlie a set of examples. We use a very simple version of this task in which there is a one-dimensional binary image that is 12 pixels long. The shapes are all 6 pixels long and their first and last pixels always have value 1, so that they are easy to locate. The middle 4 pixels of a shape can take on values of either 1 or 0, so there are 16 possible shapes. Each image only contains one shape, and the background consists entirely of zeros. To eliminate end effects, there is wrap-around so that if a shape moves off one end of the image it reappears at the other end (see figure 3). The wrap-around also ensures that the group invariance theorem of Minsky and Papert (1969) is applicable, and this theorem can be used to prove that the task cannot be performed by a network with no hidden layers.

Since each of the 16 shapes can appear in 12 positions, there are 192 possible shape instances. 160 of these are used for training the network, and 32 are used for testing it to see if it generalizes correctly. There are two test instances, selected at random, for each shape.

The network has two hidden layers as shown in figure 2. The higher hidden layer acts as a narrow bandwidth bottleneck. Information about the identity of the shape must be squeezed through this layer in order to produce the correct answer. The aim of the simulation was to show that the network would develop canonical codes for shapes in this layer, and would use these same canonical codes for shapes in novel positions. The lower hidden layer consists of units that are needed to extract the canonical representation. Each unit in this layer receives connections from 6 adjacent input units, and sends outputs to all the units in the higher hidden layer.

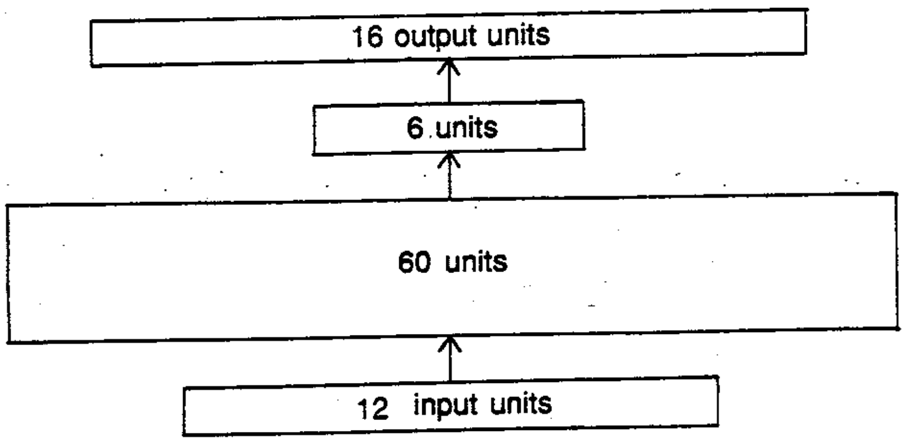

Figure 2: The architecture of the network. The activities of the input units in the bottom layer represent the image. The 60 units in the next layer up are each connected to 6 adjacent input units (with wrap-around) and to all the units in the layer above. The 6 units in the next layer up are all connected to all the output units which are in the top layer.

After extensive learning, the network performs perfectly on the training cases. To avoid requiring very large weights, the error of an output unit is defined to be zero if its activity is above 0.8 when it should be on or below 0.2 when it should be off. Otherwise the error is the square of the difference between the activity level and 0.8 or 0.2. After the network has

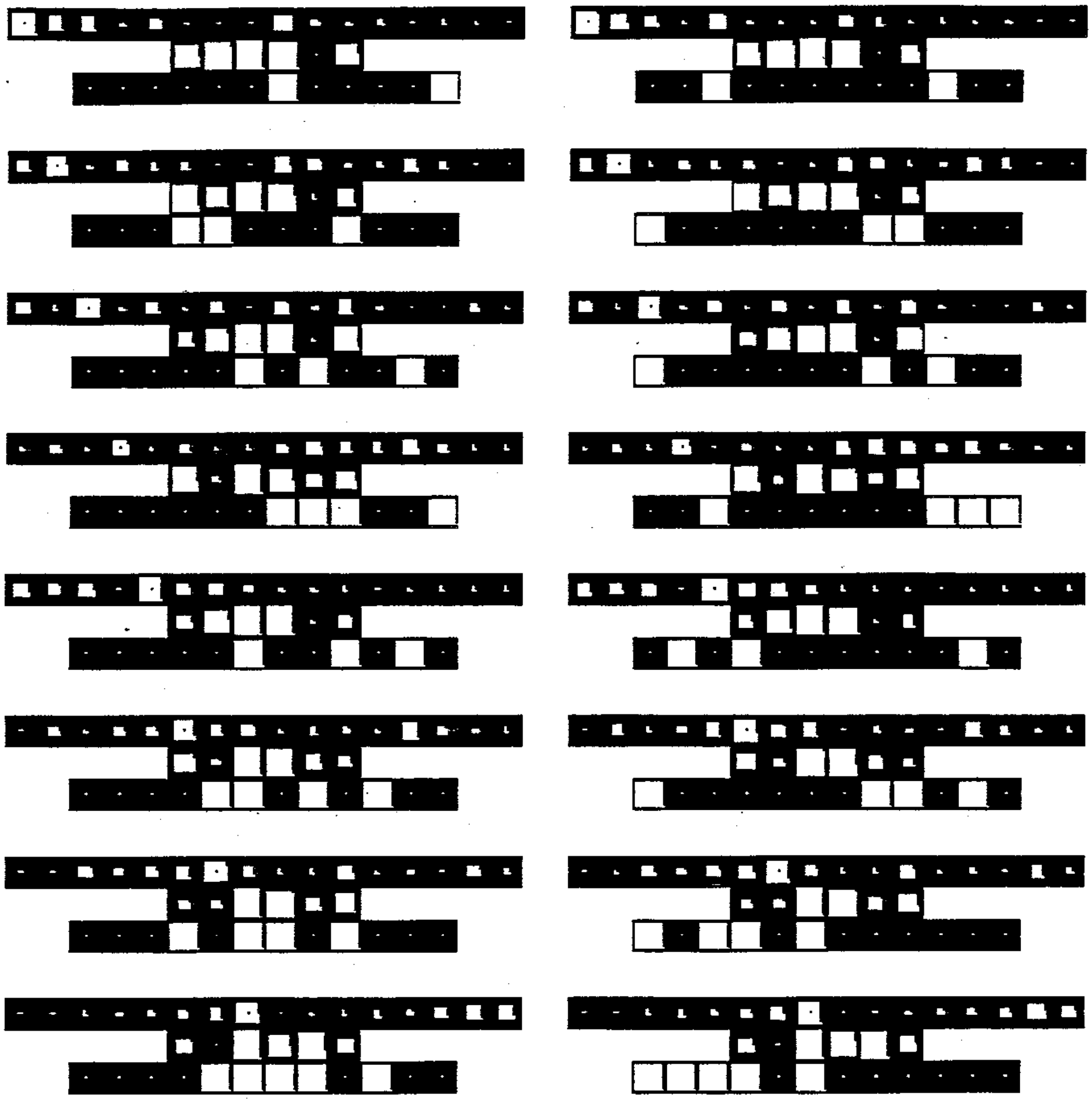

Figure 3a: The activity levels of some of the units for 16 of the 32 test cases. The bottom row of each group shows the input units (whose activity levels encode the image). The top row shows the output units and there is a small black dot on the correct answers. There are two different test cases, arranged side by side, for each shape. The middle row of each group shows the activity levels of the higher hidden layer. Notice that the two instances of each shape have very similar activity patterns in this layer. The lower hidden layer of 60 hidden units is not shown.

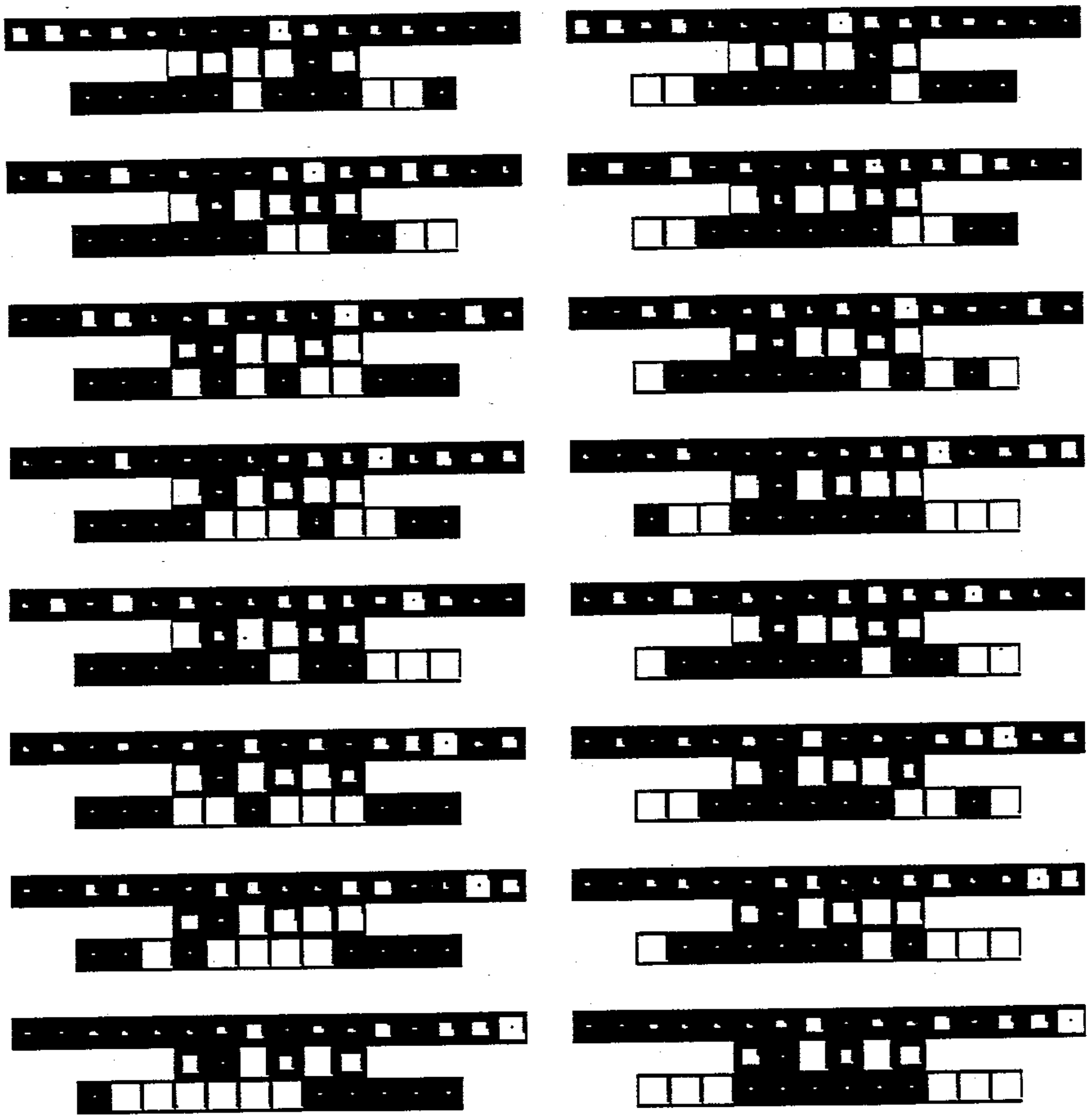

Figure 3b: The activity levels of some of the units for the remaining 16 test cases.

learned to get the training cases correct, "weight-decay" is introduced -- each time the weights are updated, their magnitude is also decremented by 0.4%. This means that only weights that are doing useful work in reducing the error can survive. It forces the network to perform the task by exploiting strong regularities that apply to many cases, rather than using the accidental structure of individual cases. The weight-decay improves the generalization performance from 17/32 correct to 30/32 correct, where a "correct" answer is one for which the correct output unit is more active than any of the others.

Figure 3 shows the activity levels of the 16 output units for each of the 32 test cases. It also shows the activity levels of the 6 units in the higher hidden layer. Notice that the network has developed a canonical "description" for each shape so that the two novel instances of each shape receive very similar descriptions even though the network has not seen either instance before. This is what allows the network to generalize correctly. When presented with a novel image, the network converts it into a canonical description. It can do this because it has seen many other shapes in the same position, and so it knows how to produce canonical descriptions of the individual pieces of the image. The use of canonical descriptions is much more powerful than simply looking for correlations between the input and output. If we consider the images alone, most of the test images have a much larger overlap with images of some other shape than they do with images of the correct shape in other positions.

4.1 The learning parameters

All the weights are updated in parallel after sweeping through the entire set of training cases. All the weights were randomly initialized to values between –0.5 and +0.5 and the network was first trained for 20 sweeps using the acceleration method (Eq. 7) with $\varepsilon=.002$ and $\alpha=.5$. Then it was trained for 8000 sweeps with $\varepsilon=.004$ and $\alpha=.95$. After this training the errors on the test cases were tiny. Weight-decay of 0.4% per update was then introduced and the network was trained for a further 15,000 sweeps with $\varepsilon=.008$ and $\alpha=.98$. The parameters used were very conservative to ensure that there were no oscillations and that the weights were not accidentally driven to large values from which they could not recover. The training time could be reduced substantially by using an adaptive scheme for dynamically controlling the parameters. It could also be reduced by using a value for ε that is inversely proportional to the number of input lines that the unit receives. Nevertheless the learning would still be very slow.

4.2 Some variations

Simulations in which the units in the lower hidden layer received inputs from the whole image did not generalize as well (20/32 correct), so the local connectivity is helpful in constraining the function that the network learns. Simulations that omitted the higher hidden layer (the narrow bandwidth channel) did not generalize nearly as well (4/32 correct).

A very different way of using back-propagation for shape recognition is described in Hinton (1987). It uses a network with recurrent connections that settles to a stable state when shown an image, and it incorporates a more elaborate mechanism for achieving invariance that includes an explicit representation of the position and orientation of the object in the image.

5 The future of back-propagation

One drawback of back-propagation is that it appears to require an external superviser to specify the desired states of the output units. It can be converted into an unsupervised procedure by using the input itself to do the supervision. Consider a multilayer "encoder" network in which the desired output vector is identical with the input vector. The network must learn to compute the identity mapping for all the input vectors in its training set. If the middle layer of the network contains fewer units than the input layer, the learning procedure must construct a compact, invertible code for each input vector. This code can then be used as the input to later stages of processing.

The use of self-supervised back-propagation to construct compact codes resembles the use of principal components analysis to perform dimensionality reduction, but it has the advantage that it allows the code to be a non-linear transform of the input vector. This form of back-propagation has been used successfully to compress images (Cottrell, personal communication, 1986) and to compress speech waves (Elman and Zipser, 1987). A variation of it has been used to extract the underlying degrees of freedom of simple shapes (Saund, 1986). It is also possible to use back-propagation to predict one part of the perceptual input from other parts. In domains with sequential structure, one portion of a sequence can be used as input and the next term in the sequence can be the desired output. This forces the network to extract features that are good predictors.

One promising technique for improving the way the learning time scales with the size of the task is to introduce modularity so that the interactions between the changes in different weights are reduced. For tasks like low-level vision, it is possible to specify in advance that each part of an image should be predictable from nearby parts. This allows a procedure like self-supervised back-propagation to be used in a local module which can learn independently and in parallel with other local modules. In addition to using this kind of innately specified modularity, it is clear that people solve complex tasks by making use of procedures and representations that they have already developed for solving simpler tasks. If back-propagation is to be effective for more realistic tasks than the toy examples that have been used so far, it probably needs to be developed in such a way that it too can exhibit this type of modularity.

6 Conclusions

There are now a number of different "connectionist" learning procedures which are capable of constructing interesting representations in the hidden units of a connectionist network. All of the existing procedures are very slow for large networks, and future progress will depend on finding faster procedures or finding ways of making the existing procedures scale better. Parallel hardware will probably be needed for this research because searching a very large space of possible representations is probably inherently expensive. If the research is successful it may lead to a new kind of chip that learns from examples rather than requiring explicit programming.

Acknowledgements

This research was supported by grant IST/8520359 from the National Science Foundation. The simulator for the learning procedure was developed by Kevin Lang and David Plaut. David Rumelhart provided helpful advice.

References

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

Alspector, J. & Allen, R. B. (1987). A neuromorphic VLSI learning system. In P. Loseleben (Ed.), *Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference.*. Cambridge, Mass.: MIT Press.

Elman, J. L. and Zipser, D. (1987). *Discovering the hidden structure of speech* (Tech. Rep.). Institute for Cognitive Science Technical Report No. 8701. University of California, San Diego.,

Feldman, J. A. & Ballard, D. H. (1982). Connectionist models and their properties. *Cognitive Science, 6*, 205-254.

Hinton, G. E., & Anderson, J. A. (1981). *Parallel models of associative memory.* Hillsdale, NJ: Erlbaum.

Hinton, G. E. (1987). Learning to recognize shapes in a parallel network. In M. Imbert (Ed.), *Proceedings of the 1986 Fyssen Conference.* Oxford: Oxford University Press.

Le Cun, Y. (1985). A learning scheme for asymmetric threshold networks. *Proceedings of Cognitiva 85.* Paris, France.

Minsky, M. & Papert, S. (1969). *Perceptrons.* Cambridge, Mass: MIT Press.

Newell, A. (1980). Physical symbol systems. *Cognitive Science, 4,* 135-183.

Parker, D. B. (April 1985). *Learning-logic* (Tech. Rep.). TR-47, Sloan School of Management, MIT, Cambridge, Mass.,

Plaut, D. C., Nowlan, S. J., & Hinton, G. E. (June 1986). *Experiments on learning by back-propagation* (Tech. Rep. CMU-CS-86-126). Pittsburgh PA 15213: Carnegie-Mellon University,

Plaut, D. C. and Hinton, G. E. (1987). Learning sets of filters using back-propagation. *Computer Speech and Language,* .

Rosenblatt, F. (1962). *Principles of neurodynamics.* New York: Spartan Books.

Rumelhart, D. E., McClelland, J. L., & the PDP research group. (1986). *Parallel distributed processing: Explorations in the microstructure of cognition. Volume I.*. Cambridge, MA: Bradford Books.

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

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, & the PDP research group (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition.* Cambridge, MA: Bradford Books.

Saund, E. (1986). Abstraction and representation of continuous variables in connectionist networks. *Proceedings of the Fifth National Conference on Artificial Intelligence.* Los Altos, California, Morgan Kauffman.

Sejnowski, T. J. & Rosenberg C. R. (1986). *NETtalk: A parallel network that learns to read aloud* Technical Report 86-01 . Department of Electrical Engineering and Computer Science, Johns Hopkins University, Baltimore, MD.,