Rumelhart, D. E., Durbin, R., Golden, R., and Chauvin, Y. (1995). Backpropagation: The basic theory. In Y. Chauvin and D. E. Rumelhart (eds.),Backpropagation:Theory, Architectures, and Applications. Hillsdale: NJ. Lawrence Erlbaum, 1-34. Chapter 1.

1 Backpropagation: The Basic Theory

Reprinted in:
Rumelhart, D. E., Durbin, R., Golden, R. M., Chauvin, Y. (1996). Backpropagation: The basic theory. In P. Smolensky, M. C. Mozer, and D. E. Rumelhart(eds.) Mathematical Perspectives on Neural Networks. Erlbaum: NJ, 533-566.

David E. Rumelhart
Richard Durbin
Richard Golden
Yves Chauvin
Department of Psychology, Stanford University

INTRODUCTION

Since the publication of the PDP volumes in 1986,[1] learning by backpropagation has become the most popular method of training neural networks. The reason for the popularity is the underlying simplicity and relative power of the algorithm. Its power derives from the fact that, unlike its precursors, the perceptron learning rule and the Widrow-Hoff learning rule, it can be employed for training nonlinear networks of arbitrary connectivity. Since such networks are often required for real-world applications, such a learning procedure is critical. Nearly as important as its power in explaining its popularity is its simplicity. The basic idea is old and simple; namely define an error function and use hill climbing (or gradient descent if you prefer going downhill) to find a set of weights which optimize performance on a particular task. The algorithm is so simple that it can be implemented in a few lines of code, and there have been no doubt many thousands of implementations of the algorithm by now.

The name *back propagation* actually comes from the term employed by Rosenblatt (1962) for his attempt to generalize the perceptron learning algorithm to the multilayer case. There were many attempts to generalize the perceptron learning procedure to multiple layers during the 1960s and 1970s, but none of them were especially successful. There appear to have been at least three independent inventions of the modern version of the back-propagation algorithm: Paul Werbos developed the basic idea in 1974 in a Ph.D. dissertation entitled

[1] *Parallel distributed processing: Explorations in the microstructure of cognition*. Two volumes by Rumelhart, McClelland, and the PDP Research Group.

"Beyond Regression," and David Parker and David Rumelhart apparently developed the idea at about the same time in the spring of 1982. It was, however, not until the publication of the paper by Rumelhart, Hinton, and Williams in 1986 explaining the idea and showing a number of applications that it reached the field of neural networks and connectionist artificial intelligence and was taken up by a large number of researchers.

Although the basic character of the back-propagation algorithm was laid out in the Rumelhart, Hinton, and Williams paper, we have learned a good deal more about how to use the algorithm and about its general properties. In this chapter we develop the basic theory and show how it applies in the development of new network architectures.

We will begin our analysis with the simplest cases, namely that of the feedforward network. The pattern of connectivity may be arbitrary (i.e., there need not be a notion of a layered network), but for our present analysis we will eliminate cycles. An example of such a network is illustrated in Figure 1.[2]

For simplicity, we will also begin with a consideration of a training set which consists of a set of ordered pairs $[\langle \vec{x}, \vec{d} \rangle_i]$ where we understand each pair to represent an observation in which outcome $\vec{d}$ occurred in the context of event $\vec{x}$. The goal of the network is to learn the relationship between $\vec{x}$ and $\vec{d}$. It is useful to imagine that there is some unknown function relating $\vec{x}$ to $\vec{d}$, and we are trying to find a good approximation to this function. There are, of course, many standard methods of function approximation. Perhaps the simplest is linear regression. In that case, we seek the best linear approximation to the underlying function. Since multilayer networks are typically nonlinear it is often useful to understand feedforward networks as performing a kind of *nonlinear* regression. Many of the issues that come up in ordinary linear regression also are relevant to the kind of nonlinear regression performed by our networks.

One important example comes up in the case of "overfitting." We may have too many predictor variables (or degrees of freedom) and too little training data. In this case, it is possible to do a great job of "learning" the data but a poor job of generalizing to new data. The ultimate measure of success is not how closely we approximate the training data, but how well we account for as yet unseen cases. It is possible for a sufficiently large network to merely "memorize" the training data. We say that the network has truly "learned" the function when it performs well on unseen cases. Figure 2 illustrates a typical case in which accounting exactly for noisy observed data can lead to worse performance on the new data. Combating this "overfitting" problem is a major problem for complex networks with many weights.

Given the interpretation of feedforward networks as a kind of nonlinear regression, it may be useful to ask what features the networks have which might

[2]As we indicate later, the same analysis can be applied to networks with cycles (recurrent networks), but it is easiest to understand in the simpler case.

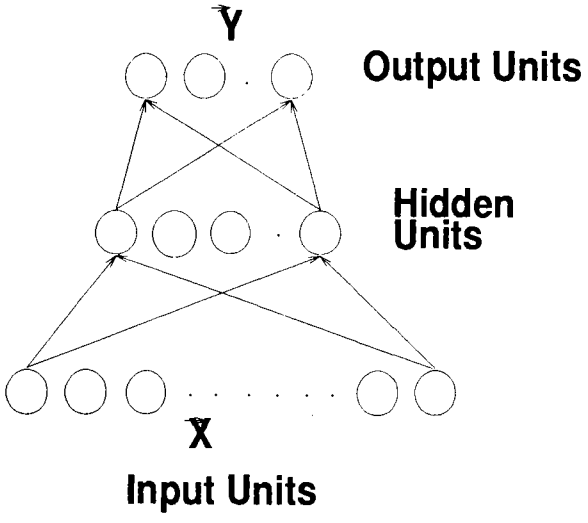

Figure 1. A simple three-layer network. The key to the effectiveness of the multilayer network is that the hidden units learn to represent the input variables in a task-dependent way.

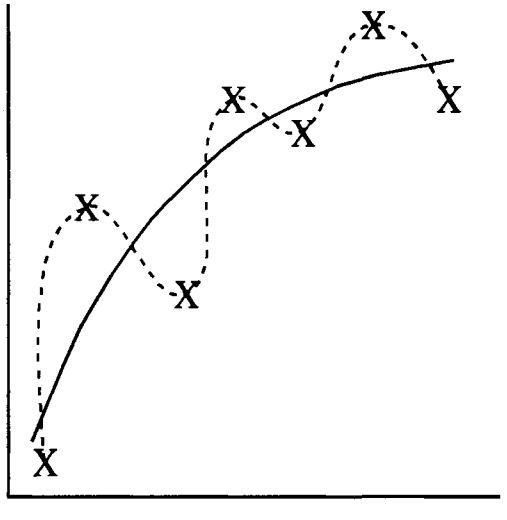

Figure 2. Even though the oscillating line passes directly through all of the data points, the smooth line would probably be the better predictor if the data were noisy.

give them an advantage over other methods. For these purposes it is useful to compare the simple feedforward network with one hidden layer to the method of polynomial regression. In the case of polynomial regression we imagine that we transform the input variables $\vec{x}$ into a large number of variables by adding a number of the cross terms $x_1x_2, x_1x_3, \ldots, x_1x_2x_3, x_1x_2x_4, \ldots$. We can also add terms with higher powers $x_1^2, x_1^3, \ldots$ as well as cross terms with higher powers. In doing this we can, of course approximate any output surface we please. Given that we can produce any output surface with a simple polynomial regression model, why should we want to use a multilayer network? The structures of these two networks are shown in Figure 3.

We might suppose that the feedforward network would have an advantage in that it might be able to represent a larger function space with fewer parameters. This does not appear to be true. Roughly, it seems to be that the "capacity" of both networks is proportional to the number of parameters in the network (cf. Cover, 1965; Mitchison & Durbin, 1989). The real difference is in the different kinds of constraints the two representations impose. Notice that for the polynomial network the number of possible terms grows rapidly with the size of the input vector. It is not, in general, possible, even to use all of the first-order cross terms since there are $n(n + 1)/2$ of them. Thus, we need to be able to select that subset of input variables that are most relevant, which often means selecting the lower-order cross terms and thereby representing only the pairwise or, perhaps, three-way interactions.

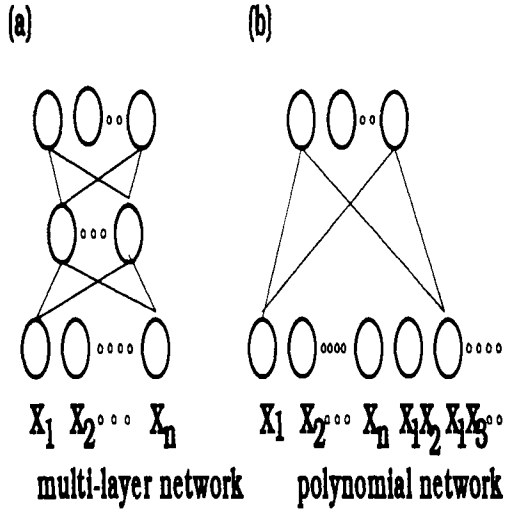

Figure 3. Two networks designed for nonlinear regression problems. The multilayer network has a set of hidden units designed to discover a "low-order" representation of the input variables. In the polynomial network the number of terms expands exponentially.

In layered networks the constraints are very different. Rather than limiting the *order of the interactions*, we limit only the *number of interactions* and let the network select the appropriate combinations of units. In many real-world situations the representation of the signal in physical terms (for example, in terms of the pixels of an image or the acoustic representation of a speech signal) may require looking at the relationships among many input variables at a time, but there may exist a description in terms of a relatively few variables if only we knew what they were. The idea is that the multilayer network is trying to find a low-order representation (a few hidden units), but that representation itself is, in general, a nonlinear function of the physical input variables which allows for the interactions of many terms.

Before we turn to the substantive issues of this chapter, it is useful to ask for what kinds of applications neural networks would be best suited. Figure 4 provides a framework for understanding these issues. The figure has two dimensions, "Theory Richness" and "Data Richness." The basic idea is that different kinds of systems are appropriate for different kinds of problems. If we have a good theory it is often possible to develop a specific "physical model" to describe the phenomena. Such a "first-principles" model is especially valuable when we

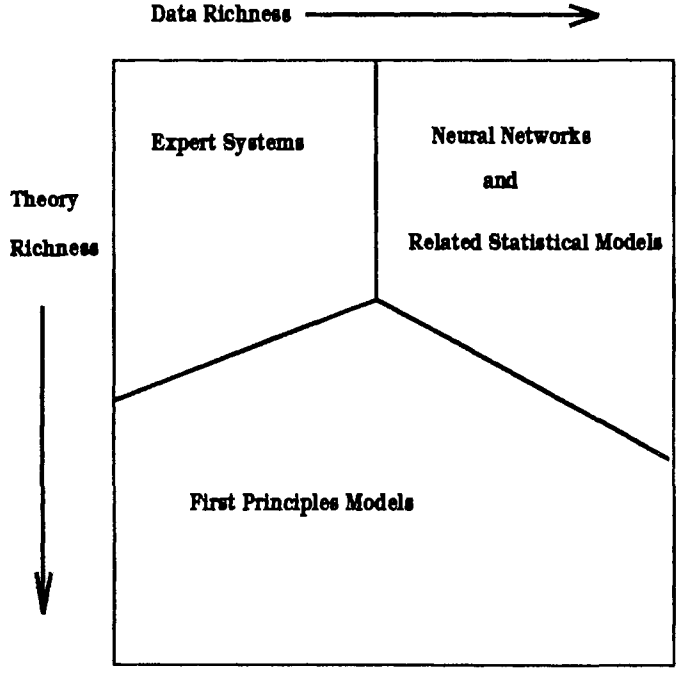

Figure 4. Neural networks and back propagation can be of the most value for a problem relatively poor in theory and relatively rich in data.

have little data. Sometimes we are "theory poor" and also "data poor." In such a case, a good model may be best determined through asking "experts" in a field and, on the basis of their understanding, devise an "expert system." The cases where networks are particularly useful are domains where we have lots of data (so we can train a complex network) but not much theory, so we cannot build a first-principles model. Note that when a situation gets sufficiently complex and we have enough data, it may be that so many approximations to the first principles models are required that in spite of a good deal of theoretical understanding better models can be constructed through learning than by application of our theoretical models.

SOME PRELIMINARY CONSIDERATIONS

There are three major issues we must address when considering networks such as these. These are:

1. *The representation problem.* What is the representational capacity of a networks of this sort? How must the size of the network grow as the complexity of the function we are attempting to approximate grows?
2. *The learning problem.* Given that a function can be approximated reasonably closely by the network, can the function be learned by the network? How does the training time scale with the size of the network and the complexity of the problem?
3. *The generalization problem.* Given a network which has learned the training set, how certain can we be of its performance on new cases? How must the size of the data set grow as the complexity of the to be approximated function grows? What strategies can be employed for improving generalization?

Representation

The original critique by Minsky and Pappert was primarily concerned with the representational capacity of the perceptron. They showed (among other things) that certain functions were simply not representable with single-layer perceptrons. It has been shown that multilayered networks do not have these limitations. In particular, we now know that with enough hidden units essentially any function can be approximated as closely as possible (cf. Hornik et al., 1989). There still is a question about the way the size of the network must scale with the complexity of the function to be approximated. There are results which indicate that smooth, continuous functions require, in general, simpler networks than functions with discontinuities.

Learning

Although there are results that indicate that the general learning problem is extremely difficult—certain representable functions may not be learnable at all—empirical results indicate that the "learning" problem is much easier than expected. Most real-world problems seem to be learnable in a reasonable time. Moreover, learning normally seems to scale linearly; that is, as the size of real problems increase, the training time seems to go up linearly (i.e., it scales with the number of patterns in the training set). Note that these results were something of a surprise. Much of the early work with the back-propagation algorithm was done with artificial problems, and there was some concern about the time that some problems, such as the parity problem, required. It now appears that these results were unduly pessimistic. It is rare that more than 100 times through the training set is required.

Generalization

Whereas the learning problem has turned out to be simpler than expected, the generalization problem has turned out to be more difficult than expected. It appears to be possible to easily build networks capable of learning fairly large data sets. Learning a data set turns out to be little guarantee of being able to generalize to new cases. Much of the most important work during recent years has been focused on the development of methods to attempt to optimize generalization rather than just the learning of the training set.

A PROBABILISTIC MODEL FOR BACK-PROPAGATION NETWORKS

The goal of the analysis which follows is to develop a theoretical framework which will allow for the development of appropriate networks for appropriate problems while optimizing generalization. The back-propagation algorithm involves specifying a cost function and then modifying the weights iteratively according to the gradient of the cost function. In this section we develop a rationale for an appropriate cost function. We propose that the goal is to find that network which is the most likely explanation of the observed data sequence. We can express this as trying to maximize the term

$$P(\mathcal{N}|\mathcal{D}) = \frac{P(\mathcal{D}|\mathcal{N})P(\mathcal{N})}{P(\mathcal{D})},$$

where $\mathcal{N}$ represents the network (with all of the weights and biases specified), $\mathcal{D}$ represents the observed data, and $P(\mathcal{D}|\mathcal{N})$ is the probability that the network $\mathcal{N}$ would have produced the observed data $\mathcal{D}$. Now since sums are easier to work

with than products, we will maximize the log of this probability. Since the log is a monotonic transformation, maximizing the log is equivalent to maximizing the probability itself. In this case we have

$$\ln P(\mathcal{N}|\mathcal{D}) = \ln P(\mathcal{D}|\mathcal{N}) + \ln P(\mathcal{N}) - \ln P(\mathcal{D}).$$

Finally, since the probability of the data is not dependent on the network, it is sufficient to maximize $\ln P(\mathcal{D}|\mathcal{N}) + \ln P(\mathcal{N})$.

Now, it is useful to understand the meaning of these two terms. The first term represents the probability of the data given the network; that is, it is a measure of how well the network accounts for the data. The second term is a representation of the probability of the network itself; that is, it is a prior probability or a prior constraint on the network. Although it is often difficult to specify the prior, doing so is an important way of inserting knowledge into the learning procedure. More will be said about this later. For the time being, however, we focus on the first term, the performance.

It is useful to begin by noticing that the data can be broken down into a set of observations, each, we will assume, chosen independently of the others. Thus, we can write the probability of the data given the network as

$$\begin{aligned}\ln P(\mathcal{D}|\mathcal{N}) &= \ln P([\langle \vec{x}, \vec{d}_i \rangle]|\mathcal{N}) \\ &= \ln \prod_i P(\langle \vec{x}, \vec{d}_i \rangle|\mathcal{N}) = \sum_i \ln P(\langle \vec{x}, \vec{d}_i \rangle|\mathcal{N}).\end{aligned}$$

Note that again this assumption allows us to express the probability of the data given the network as the sum of terms, each term representing the probability of a single observation given the network. We can take still another step. We can break the data into two parts: the outcome $\vec{d}_i$ and the observed event $\vec{x}_i$. We can write

$$\ln P(\mathcal{D}|\mathcal{N}) = \sum_i \ln P(\vec{d}_i|\vec{x}_i \wedge \mathcal{N}) + \sum_i \ln P(\vec{x}_i).$$

Now, since we suppose that the event $\vec{x}_i$ does not depend on the network, the last term of the equation will not affect the determination of the optimal network. Therefore, we need only maximize the term $\Sigma_i \ln (P(\vec{d}_i|\vec{x}_i \wedge \mathcal{N})$.

So far we have been very general; the only real assumption made is the independence of the observed data points. In order to get further, however, we need to make some specific assumptions, particularly about the relationship between the output of the network $\vec{y}_i$ and the observed outcome $\vec{d}_i$, a probabilistic assumption. First, we assume that the relationship between $\vec{x}_i$ and $\vec{d}_i$ is not deterministic, but that, for any given $\vec{x}_i$, there is a distribution of possible values of $\vec{d}_i$. The network, however, *is deterministic,* so rather than trying to predict the actual outcome we are only trying to predict the expected value of $\vec{d}_i$ given $\vec{x}_i$. Thus, the network output $\vec{y}_i$ is to be interpreted as the mean of the actual observed value. This is, of course, the standard assumption.

The Gaussian Case

To proceed further, we must specify the form of the distribution of which the network output is the mean. To decide which distribution is most appropriate, it is necessary to consider the nature of the outcomes, $\vec{d}$. In ordinary linear regression, there is an underlying assumption that the noise is normally distributed about the predicted values. In situations in which this is so, a Gaussian probability distribution is appropriate, even for nonlinear regression problems in which nonlinear networks are required. We begin our analysis in this simple case.

Under the assumption of normally distributed noise in the observations we can write

$$P(\vec{d}_i|\vec{x}_i \wedge \mathcal{N}) = K \exp\left(\sum_j \frac{(y_{ij} - d_{ij})^2}{2\sigma^2}\right),$$

where K is the normalization term for the Gaussian distribution. Now we take the log of the probability:

$$\ln P\,(\vec{d}_i|\vec{x}_i \wedge \mathcal{N}) = -\ln K - \frac{\Sigma_j (y_{ij} - d_{ij})^2}{2\sigma^2}.$$

Under the assumption that σ is fixed, we want to maximize the following term, where $\mathcal{C}$ is the function to be maximized.

$$\mathcal{C} = -\sum_i \sum_j \frac{(y_{ij} - d_{ij})^2}{2\sigma^2}.$$

Now we must consider the appropriate transfer functions for the output units. For the moment, we will consider the case of what we have termed *quasi-linear* output units in which the output is a function of the *net input* of the unit, where[3] the net input is simply a weighted sum of the inputs to the unit. That is, the net input for unit j, η_j, is given by $\eta_j = \Sigma_k w_{jk} h_k + \beta_j$. Thus, we have $y_j = \mathcal{F}(\eta_j)$.[4]

Recall that the back-propagation learning rule is determined by the derivative of the cost function with respect to the parameters of the network. In this case we can write

$$\frac{\partial \mathcal{C}}{\partial \eta_j} = \frac{(d_{ij} - y_{ij})}{\sigma^2} \frac{\partial \mathcal{F}(\eta_j)}{\partial \eta_j}.$$

This has the form of the difference between the predicted and observed values divided by the variance of the error term times the derivative of the output

[3]Note, this is not necessary. The output units could have a variety of forms, but the quasi-linear class is simple and useful.

[4]Note that η_j itself is a function of the input vector $\vec{x}_i$ and the weights and biases of the entire network.

function with respect to its net input. As we shall see, this is a very general form which occurs often.

Now what form should the output function take? It has been conventional to take it to be a sigmoidal function of its net input, but under the Gaussian assumption of error, in which the mean can, in principle, take on *any* real value, it makes more sense to let $\vec{y}$ be linear in its net input. Thus, for an assumption of Gaussian error and linear output functions we get the following very simple form of the learning rule:

$$\frac{\partial \mathcal{C}}{\partial \eta_j} \propto (d_{ij} - y_{ij}).$$

The change in η should be proportional to the difference between the observed output and its predicted value. This model is frequently appropriate for prediction problems in which the error can reasonably be normally distributed. As we shall see, classification problems in which the observations are binary are a different situation and generate a different model.

The Binomial Case

Often we use networks for classification problems—that is, for problems in which the goal is to provide a binary classification of each input vector for each of several classes. This class of problems requires a different model. In this case the outcome vectors normally consist of a sequence of 0's and 1's. The "error" cannot be normally distributed, but would be expected to be binomially distributed. In this case, we imagine that each element of $\vec{y}$ represents the *probability* that the corresponding element of the outcome vector $\vec{d}$ takes on the value 0 or 1. In this case we can write the probability of the data given the network as

$$P(\vec{d}|\vec{x} \wedge \mathcal{N}) = \prod_j y_j^{d_j}(1 - y_j)^{1-d_j}.$$

The log of probability is $\Sigma_j\, d_j \ln y_j + (1 - d_j) \ln (1 - y_j)$ and, finally,

$$\mathcal{C} = \sum_i \sum_j d_j \ln y_j + (1 - d_j) \ln(1 - y_j).$$

In the neural network world, this has been called the cross-entropy error term. As we shall see, this is just one of many such error terms. Now, the derivative of this function is

$$\frac{\partial \mathcal{C}}{\partial \eta_j} = \frac{d_j - y_j}{y_j(1 - y_j)} \frac{\partial \mathcal{F}(\eta_j)}{\partial \eta_j}.$$

Again, the derivative has the same form as before—the difference between the predicted and observed values divided by the variance (in this case the variance

of the binomial) times the derivative of the transfer function with respect to its net input.

We must now determine the meaning of the form of the output function. In this case, we want it to range between 0 and 1, so a sigmoidal function is natural. Interestingly, we see that if we choose the logistic $\mathcal{F}(\eta_j) = 1/(1 + e^{-\eta_j})$, we find an interesting result. The derivative of the logistic is $\mathcal{F}(\eta_j)(1 - \mathcal{F}(\eta_j))$ or $y_j(1 - y_j)$. It happens that this is the variance of the binomial, so it cancels the denominator in the previous equation, leaving the same simple form as we had for the Gaussian case:

$$\frac{\partial \mathcal{E}}{\partial \eta_j} \propto (d_j - y_j).$$

In the work on generalized linear models (cf. McCullagh & Nelder, 1989) such functions are called *linking functions,* and they point out that different linking functions are appropriate for different sorts of problems. It turns out to be useful to see feedforward networks as a generalization into the nonlinear realm of the work on generalized linear models. Much of the analysis given in the McCullagh and Nelder applies directly to such networks.

The Multinomial Case

In many applications we employ not multiple classification or binary classification, but "1-of-n" classification. Here we must employ still another transfer function. In this case, choose the normalized exponential output function[5]

$$\mathcal{F}_j(\vec{\eta}) = \frac{e^{\eta_j}}{\sum_k e^{\eta_k}}.$$

In this case, the $\vec{d}$ vector consists of exactly one 1 and the remaining digits are zeros. We can then interpret the output unit j, for example, as representing the probability that the input vector was a member of class j. In this case we can write the cost function as

$$\mathcal{E} = \sum_i \sum_j d_{ij} \ln \frac{e^{\eta_j}}{\sum_k e^{\eta_k}}.$$

and, again, after computing the derivative we get

$$\frac{\partial \mathcal{E}}{\partial \eta_j} \propto (d_{ij} - y_{ij}).$$

[5]This is sometimes called the "soft-max" or "Potts" unit. As we shall see, however, it is a simple generalization of the ordinary sigmoid and has a simple interpretation as representing the posterior probability of event j out of a set n of possible events.

The General Case

The fact that these cases all end up with essentially the same learning rule in spite of different models is not accidental. It requires exactly the right choice of output functions for each class of problem. It turns out that this result will occur whenever we choose a probability function from the *exponential family* of probability distributions. This family, which includes, in addition to the normal and the binomial, the gamma distribution, the exponential distribution, the Poisson distribution, the negative binomial distribution, and most other familiar probability distributions. The general form of the exponential family of probability distributions is

$$P(\vec{d}|\vec{x} \wedge \mathcal{N}) = \exp\left[\sum_i \frac{(d_i\theta - B(\theta)) + C(\vec{d}\phi)}{a(\phi)}\right],$$

where θ is the "sufficient statistic" of the distribution and is related to the mean of the distribution, ϕ is a measure of the overall variance of the distribution, and the $B(\ )$, $C(\ )$ and $a(\ )$ are different for each member of the family. It is beyond the scope of this chapter to develop the general results of this model.[6] Suffice it to say that for all members of the exponential family we get

$$\frac{\partial \mathcal{C}}{\partial \eta_j} \propto \frac{d_j - y_j}{\text{var}(y_j)}.$$

We then choose as an output function one whose derivative with respect to η is equal to the variance. For members of the exponential family of probability distributions we can always do this.

The major point of this analysis is that by using one simple cost function, a log-likelihood function, and by looking carefully at the problem at hand, we see that, unlike the original work, in which the squared error criterion was normally employed in nearly all cases, different cost functions are appropriate for different cases—prediction, cross-classification, and 1-of-*n* classification all require different forms of output units. The major advantage of this is not so much that the squared error criterion is wrong, but that by making specific probability assumptions we can get a better understanding of the *meaning* of the output units. In particular, we can interpret them as the means of underlying probability distributions. As we shall show, this understanding allows for the development of rather sophisticated architecture in a number of cases.

SOME EXAMPLES

Before discussing priors and analyzing hidden units, we sketch how to use this method of analysis to design appropriate learning rules for complex networks.

[6]See McCullagh and Nelder (1989, pp. 28–30) for a more complete description.

A Simple Clustering Network

Consider the following problem. Suppose that we wish to build a network which received a sequence of data and attempted to cluster the data points into some predefined number of clusters. The basic structure of the network, illustrated in Figure 5, consists of a set of input units, one for each element of the data vector $\vec{x}$, a set of hidden "cluster" units, and a single linear output unit. In this case, we suppose that the hidden units are "Gaussian"; that is, their output values are given by $K \exp [\Sigma_j (x_j - w_{jk})^2/2\sigma^2]$. In this case, the weights $\vec{w}_k$ can be viewed as the center of the kth cluster. The parameter σ, constant in this case, determines the spread of the cluster. We want the output unit to represent the probability of the data given the network, $P(\vec{x}_i|\mathcal{N})$.

Now if we assume that the clusters are mutually exclusive and exhaustive we can write

$$P(\vec{x}_i|\mathcal{N}) = \sum_k P(\vec{x}_i|\vec{x}_i \in c_k)P(c_k),$$

where c_k indexes the kth cluster. For simplicity we can assume that the clusters are to be equally probable, so $P(c_k) = 1/N$, where N is the number of clusters. Now, the probability of the data given the cluster is simply the output of the kth hidden unit, h_k. Therefore, the value of the output unit is $1/N \Sigma_k h_k$ and the log-likelihood of the data given the input is

$$\mathcal{C} = \ln\left(\sum_k h_k \frac{1}{N}\right).$$

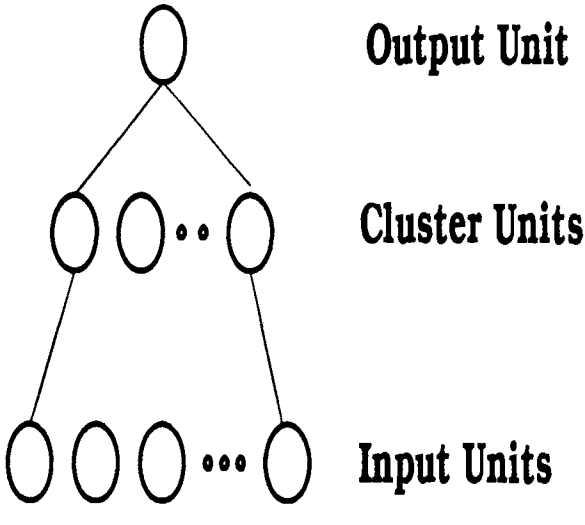

Figure 5. A simple network for clustering input vectors.

The derivative of $\mathcal{C}$ with respect to the weights is

$$\frac{\partial \mathcal{C}}{\partial w_{jk}} = \left(\frac{(K/N)\exp[\Sigma_j (x_j - w_{jk})^2/2\sigma^2]}{\Sigma_k (K/N)\exp - \Sigma_j [(x_j - w_{jk})^2/2\sigma^2]} \right) \frac{x_j - w_{jk}}{\sigma^2}.$$

The term in parentheses represents the posterior probability that the correct cluster is c_k given the input in a member of one of the clusters. We can call this posterior probability p_k. We can now see that the learning rule is again very simple:

$$\frac{\partial \mathcal{C}}{\partial w_{jk}} \propto p_k (x_j - w_{jk}).$$

This is a slight modification of the general form already discussed. It is the difference between the observed value x_j and the estimated mean value w_{jk} weighted by the probability that cluster k was the correct cluster p_k.

This simple case represents a classic mixture of Gaussian model. We assumed fixed probabilities per cluster and a fixed variance. It is not difficult to estimate the probabilities of each cluster and the variance associated with the clusters. It is also possible to add priors of various kinds. As we will explain, it is possible to order the clusters and add constraints that nearby clusters ought to have similar means. In this case, this feedforward network can be used to implement the elastic network of Durbin and Willshaw (1987) and can, for example, be used to find a solution to the traveling salesman problem.

Society of Experts

Consider the network proposed by Jacobs, Jordan, Nowlan, and Hinton (1991) and illustrated in Figure 6. The idea of this network is that instead of having a single network to solve every problem, we have a set of networks which learn to subdivide a task and thereby solve it more efficiently and elegantly. The architecture allows for all networks to look at the input units and make their best guess, but a normalized exponential "gateing" is used to weight the outputs of the individual network providing an overall best guess. The gateing network also looks at the input vector.

We must train both the gateing network and the individual "expert" networks. As before, we wish to maximize the log-likelihood of the data given the network. The final output of the network is

$$y_{ij\cdot} = \sum_k r_k y_{ijk},$$

where r_k is the probability estimated by the normalized exponential "relevance" network that subnetwork k is the correct network for the current input. At first, it may not be obvious how to train this network. Perhaps we should look at the difference between the output of the network and the observed outcome and use

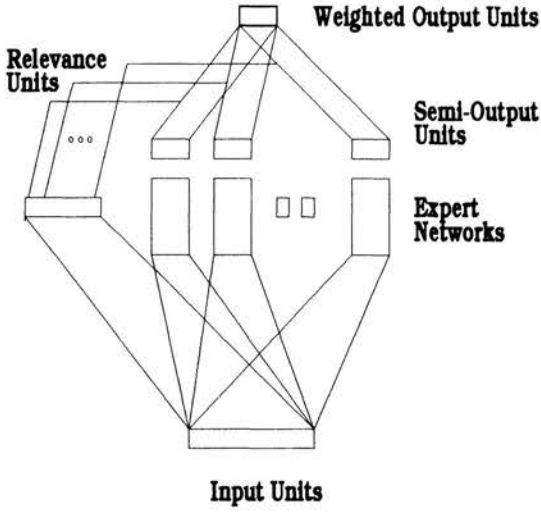

Figure 6. A simple network for clustering input vectors.

that as the error signal to be propagated back through the network. It turns out that the probabilistic analysis we have been discussing offers a different, more principled solution. We should, of course, maximize the log-likelihood—the probability of the data given the network. On the assumption that each input vector should be processed by one network and that the relevance network provides the probability that it should be network k, we can write

$$\mathcal{C} = \ln P(\vec{d}_i|\vec{x}_i \wedge \mathcal{N}) = \ln \sum_k P(\vec{d}_i|\vec{x}_i \wedge \mathcal{S}_k) r_k,$$

where $\mathcal{S}_k$ represents the kth subnet.

We must now make some specific assumptions about the form of $P(\vec{d}_i|\vec{x}_i \wedge \mathcal{S}_k)$. For concreteness, we assume a Gaussian distribution, but we could have chosen any of the other probability distributions we have discussed. In this case

$$\mathcal{C} = \ln \sum_k K r_k \exp\left[\sum_j \frac{(d_j - y_{jk})^2}{2\sigma^2}\right].$$

We now must compute the derivative of the log-likelihood function with respect to η_{jk} for each subnetwork and with respect to η_k for the relevance network. In the first case we get

$$\frac{\partial \mathcal{C}}{\partial \eta_{jk}} = \left(\frac{r_k \exp[\Sigma_j (d_j - y_{jk})^2/2\sigma^2]}{\Sigma_i r_i K \exp[\Sigma_j (d_j - y_{jk})^2/2\sigma^2]}\right) \frac{d_j - y_{jk}}{\sigma^2} = p_k \frac{d_j - y_{jk}}{\sigma^2}.$$

Note that this is precisely the same form as for the clustering network. The only real difference is that the probabilities of each class given the input were indepen-

dent of the input. In this case, the probabilities are input dependent. It is slightly more difficult to calculate, but it turns out that the derivative for the relevance units also has the simple form

$$\frac{\partial \mathcal{C}}{\partial \eta_k} = p_k - r_k,$$

the difference between the position and the prior probability that subnetwork k is the correct network.

This example, although somewhat complex, is useful for seeing how we can use our general theory to determine a learning rule in a case where it might not be immediately obvious and in which the general idea of just taking the difference between the output of the network and the target and using that as an error signal is probably the wrong thing to do. We now turn to one final example.

Integrated Segmentation and Recognition Network

A major problem with standard back-propagation algorithms is that they seem to require carefully segmented and localized input patterns for training. This is a problem for two reasons: first, it is often a labor-intensive task to provide this information and, second, the decision as to how to segment often depends on prior recognition. It is possible, however, to design a network and corresponding back-propagation learning algorithm in which we simultaneously learn to identify and segment a pattern.[7]

There are two important aspects to many pattern recognition problems which we have built directly into our network and learning algorithm. The first is that the exact location of the pattern, in space or time, is irrelevant to the classification of the pattern. It should be recognized as a member of the same class whereever or whenever it occurs. This suggests that we build translation independence directly into our network. The second aspect we wish to build into the network is that feedback about *whether* or not a pattern is present is all that should be required for training. Information about the exact location and relationship to other patterns ought not be required. The target information thus does not include information about *where* the patterns occur, but only about *whether* a pattern occurs.

We have incorporated two basic tricks into our network design to deal with these two aspects of the problem. The first is to build the assumption of translation independence into the network by using local linked receptive fields, and the

[7]The algorithm and network design presented here were first proposed by Rumelhart in a presentation entitled "Learning and generalization in multilayer networks" given at the NATO Advanced Research Workshop on Neurocomputing, Algorithms, Architecture and Applications held in Les Arcs, France, in February 1989. The algorithm can be considered a generalization and refinement of the TDNN network developed by (Waibel et al., 1989). A version of the algorithm was first published in Keeler, Rumelhart, and Loew (1991).

second is to build a fixed "forward model" (cf. Jordan & Rumelhart, 1992) which translates a location-specific recognition process into a location-independent output value and then is used to back-propagate the nonspecific error signal back through this fixed network to train the underlying location-specific network. The following sections show how these features can be realized and provide a rationale for the exact structure and assumptions of the network. The basic organization of the network is illustrated in Figure 7.

We designate the stimulus pattern by the vector $\vec{x}$. We assume that any character may occur in any position. The input features then project to a set of hidden units which are assumed to abstract hidden features from the input field. These feature abstraction units are organized into rows, one for each feature type. Each unit within a row is constrained to have the same pattern of weights as every other unit in the row. The units are thus simply translated versions of one another. This is enforced by "linking" the weights of all units in a given row, and

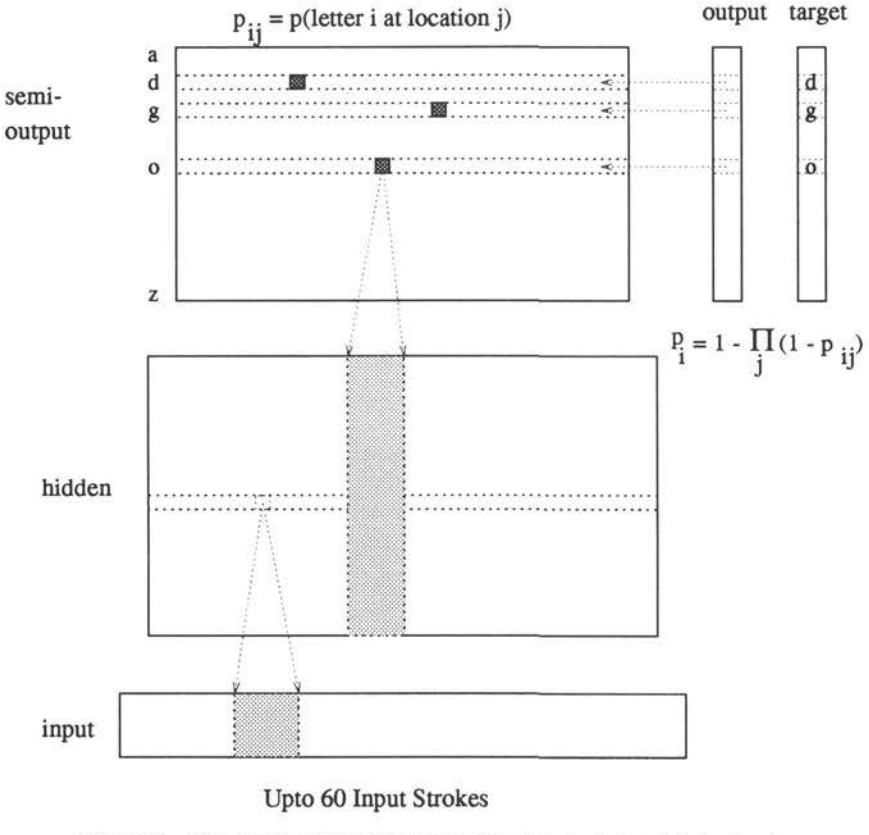

Figure 7. The basic recognition network. See text for detailed network description.

whenever one weight is changed all linked weights are changed. This is the same trick used by Rumelhart et al. (1986) to solve the so-called T/C problem and by LeCun et al. (1990) in their work on zip code recognition. We let the activation value of a hidden unit of type i at location j be a sigmoidal function of its net input and designate it h_{ij}. We interpret the activation of hidden unit h_{ij} as the probability that hidden feature f_i is present in the input at position j. The hidden units themselves have the conventional *logistic* sigmoidal transfer functions.

The hidden units then project onto a set of position-specific letter detection units. There is a row of position-specific units for each character type. Each unit in a row receives inputs from the feature units located in the immediate vicinity of the recognition unit. As with the hidden units, the units in a given row are translated versions of one another. We designate the unit for detecting character i at location j as p_{ij}. We let

$$p_{ij} = \frac{1}{1 + e^{-\eta_{ij}}},$$

where

$$\eta_{ij} = \sum_k w_{ik} h_{kj} + \beta_i$$

and w_{ik} is the weight from hidden unit h_{kj} to the detector p_{ij}. Note that since the weights from the hidden unit to the detection units are linked, this same weight will connect each feature unit in the row with a corresponding detection unit in the row above. Since we have built translational independence into the structure of the network, anything we learn about features or characters at any given location is, through the linking of weights, automatically transferred to every location.

If we were willing, or able, to carefully segment the input and tell the network exactly where each character was, we could use a standard training technique to train the network to recognize any character at any location. However, we are interested in a training algorithm in which we do not have to provide the network with specific training information. We are interested in simply telling the network which characters were present in the input, not where each character is. To implement this idea, we have built an additional network which takes the output of the p_{ij} units and computes, through a fixed output network, the probability that at least one character of a given type is present anywhere in the input field. We do this by computing the probability that at least one unit of a particular type is on. This can simply be written as

$$y_i = 1 - \prod_j (1 - p_{ij}).$$

Thus, y_i is interpreted as representing directly the probability that character i occurred at least once in the input field.

Note that *exactly the same target* would be given for the word "dog" and the word "god." Nevertheless, the network learns to properly localize the units in the p_{ij} layer. The reason is, simply, that the individual characters occur in many combinations and the only way that the network can learn to discriminate correctly is to actually detect the particular letter. The localization that occurs in the p_{ij} layer depends on each character unit seeing only a small part of the input field and on each unit of type i constrained to respond in the same way.

Important in the design of the network was an assumption as to the *meaning* of the individual units in the network. We will show why we make these interpretations and how the learning rule we derive depends on these interpretations.

To begin, we want to interpret each output unit as the probability that at least one of that character is in the input field. Assuming that the letters occurring in a given word are approximately independent of the other letters in the word, we can also assume that the probability of the target vector given the input is

$$p(\vec{d}|\vec{x}) = \prod_j y_j^{d_j}(1 - y_j)^{(1-d_j)}.$$

This is obviously an example of the binomial multiclassification model. Therefore, we get the following form of our log-likelihood function:

$$\mathcal{C} = \sum_j d_j \ln y_j + (1 - d_j)\ln(1 - y_j),$$

where d_j equals 1 if character j is presented, and zero otherwise.

On having set up our network and determined a reasonable performance criterion, we straightforwardly compute the derivative of the error function with respect to η_{ij}, the net input into the detection unit p_{ij}. We get

$$\frac{\partial \mathcal{C}}{\partial \eta_{ij}} = (d_j - y_i)\frac{p_{ij}}{y_i}.$$

This is a kind of *competitive* rule in which the learning is proportional to the relative strength of the activation of the unit at a location in the ith row to the strength of activation in the entire row. This ratio is the conditional probability that the target was at position j under the assumption that the target was, in fact, presented.This convenient interpretation is not accidental. By assigning the output units their probablistic interpretations and by selecting the appropriate, though unusual, output unit $y_i = 1 - \Pi(1 - p_{ij})$, we were able to ensure a plausible interpretation and behavior of our character detection units.

Concluding Remarks

In this section we have shown three cases in which our ability to provide useful analyses of our networks has given us important insights into network design. It is the general theory that allows us to see what assumptions to make, how to put

our networks together, and how to interpret the outputs of the networks. We have, however, attended only to one portion of the problem, namely the measure of performance. Equally important is the other term of our cost function, namely the priors over the networks. We now turn to this issue.

PRIORS

Recall that the general form of the posterior likelihood function is

$$\mathcal{C} = \sum_i \sum_j \ln P(d_{ij}|\vec{x}_i \wedge \mathcal{N}) + \ln P(\mathcal{N}).$$

In the previous section we focused on the performance term. Now we focus on the priors term. As indicated, the major point of this term is to get information and constraints into the learning procedure. The basic procedure is to modify the parameters of the network based on the derivatives of both terms of the entire cost function, not just the performance term.

Weight Decay

Perhaps the simplest case to understand is the "weight decay" term. In this case we assume that the weights are distributed normally about a zero mean. We can write this term as

$$\ln P(\mathcal{N}) = \ln \exp\left(-\frac{\Sigma_{ij} w_{ij}^2}{2\sigma^2}\right) = -\frac{1}{2\sigma^2} \sum_{ij} w_{ij}^2.$$

This amounts to a penalty for large weights. The term σ determines how important the small weight constraint is. If σ is large, the penalty term will not be very important. If it is small, than the penalty term will be heavily weighted. The derivative then is given by

$$\frac{\partial \mathcal{C}}{\partial w_{ij}} \propto -\frac{1}{\sigma^2} w_{ij}.$$

Thus, every time we see a new pattern the weights should be modified in two ways; first, they should be modified so as to reduce the overall error (as in the first time of Equation 1); then they should be moved toward zero by an amount proportional to the magnitude of the weight. The term σ determines the amount of movement that should take place.

Why should we think that the weights should be small and centered around zero? Of course, this could be a bad assumption, but it is one way of limiting the space of possible functions that the network can explore. All things being equal, the network will select a solution with small weights rather than large ones. In

linear problems this is often a useful strategy. The addition of this penalty term is known as "ridge regression," a kind of "regularization" term which limits the space of possible solutions to those with smaller weights. This is an important strategy for dealing with the overfitting problem. Weight decay was first proposed for connectionist networks by Geoffrey Hinton.

Weight Elimination

A general strategy for dealing with overfitting involves a simple application of Occam's Razor—that is, of all the networks which will fit the training data, find the simplest. The idea is to use the *prior* term to measure the complexity of the network and "prefer" simpler networks to more complex ones. But how do we measure the complexity of the network? The basic idea, due to Kolmogorov (cf. Kolmogorov, 1991), is that the complexity of a function is measured by the number of bits required to communicate the function. This is, in general, difficult to measure, but it is possible to find a set of variables which vary monotonically with the complexity of a network. For example, the more weights a network has the more complex it is—each weight has to be described. The more hidden units a network has, the greater the complexity of the network; the more bits per weight, the more complex is the network; the more symmetries there are among the weights, the simpler the network is, and so on.

Weigend et al. (1990) proposed a set of priors each of which led to a reduction in network complexity: for several priors, the weight elimination procedure has been the most useful. The idea is that the weights are not drawn from a single distribution around zero, as in weight decay, but we assume that they are drawn either from a normal distribution centered at zero or from a uniform distribution between, say, ± 20. It is possible to express this prior roughly as

$$P(\mathcal{N}) = \exp\left[- \sum_{ij} \frac{(w_{ij}/\sigma_1)^2}{1 + (w_{ij}/\sigma_2)^2} \right],$$

or, taking the log and multiplying through by the sigmas, as

$$\ln P(\mathcal{N}) = - \frac{\sigma_2^2}{\sigma_1^2} \sum_{ij} \frac{w_{ij}^2}{\sigma_2^2 + w_{ij}^2}.$$

The derivative is

$$\frac{\partial \mathcal{C}}{\partial w_{ij}} \propto - \frac{\sigma_2^2}{\sigma_1^2} \frac{w_{ij}}{(\sigma_2^2 + w_{ij}^2)^2}.$$

Note that this has the property that for small weights (weights in which w_{ij} is small relative to σ_2), the denominator is approximately constant and the change in weights is simply proportional to the numerator w_{ij}, as in weight decay. For

large weights (w_{ij} is large relative to σ_2) the change is proportional to $1/w^3$—in other words, very little change occurs. Thus, this penalty function causes small weight to move toward zero, *eliminates them* and leaves large weights alone. This has the effect of removing unneeded weights. In the nonlinear case, there is reason to believe that the weight elimination strategy is a more useful prior than weight decay since large weights are required to establish the nonlinearities. A number of successful experiments have been carried out using the strategy (cf. Weigend, Hubberman, & Rumelhart, 1990 and Weigend, Rumelhart, & Hubberman, 1991).

Although similar strategies have been suggested for eliminating unneeded hidden units and for reducing the information content of weights, these have been studied very little. Perhaps the most successful paradigm, however, is a generalization of the weight elimination paradigm to impose important weight symmetries. This work has been done by Nowlan (1991) and is described here.

Weight Symmetries

In weight decay the idea was to have a prior such that the weight distribution has a zero mean and is normally distributed. The weight elimination paradigm is more general in that it distinguishes two classes of weights, of which one is, like the weight decay case, centered on zero and normally distributed, and the other is uniformly distributed. In weight symmetries there is a small set of normally distributed weight clusters. The problem is to simultaneously estimate the mean of the priors and the weights themselves. In this case the priors are

$$P(\mathcal{N}) = \prod_i \sum_k \exp\left[-\frac{(w_i - \mu_k)^2}{2\sigma_k^2} \right] P(c_k),$$

where $P(c_k)$ is the probability of the kth weight cluster and μ_k is its center. To determine how the weights are to be changed, we must compute the derivative of the log of this probability. We get

$$\frac{\partial \mathcal{C}}{\partial w_i} \propto \sum_k \frac{\exp[(w_i - \mu_k)^2 P(c_k)/2\sigma_k^2]}{\Sigma_j \exp[(w_i - \mu_j)^2 P(c_j)/2\sigma_j^2]} \frac{(\mu_k - w_i)}{\sigma_k^2}.$$

We can similarly estimate μ_k, σ_k, and $P(c_k)$ by gradient methods as well. For example, we write the derivative of the error with respect to μ_k:

$$\frac{\partial \mathcal{C}}{\partial \mu_k} \propto \sum_j \frac{\exp[-(w_j - \mu_k)^2 P(c_k)/2\sigma_k^2]}{\Sigma_i \exp[-(w_j - \mu_i)^2 P(c_i)/2\sigma_i^2]} \frac{(w_j - \mu_k)}{\sigma_k^2}.$$

By similar methods, it is possible to estimate the other parameters of the network. Nowlan (1991) have shown that these priors go far toward solving the overfitting problem.

Elastic Network and the Traveling Salesman Problem

Earlier we showed how one could develop a clustering algorithm by using Gaussian hidden units and optimizing the log-likelihood of the data given the network. It turns out that by adding priors to the cost function we can put constraints on the clusters. Imagine that we are trying to solve the traveling salesman problem in which we are to find the shortest path through a set of cities. In this case, we represent the cities in terms of their $\langle x, y \rangle$ coordinate values so that there is a two-dimensional input vector for each city. The method is the same as that proposed by Durbin and Willshaw (1987). There is a set of clusters, each cluster located at some point in a two-dimensional space. We want to move the means of the clusters toward the cities until there is one cluster for each city and adjacent clusters are as close to one another as possible. This provides for the following cost function:

$$\begin{aligned}\mathcal{C} &= \sum_i \ln \sum_j \exp[\,[-(x_i - \mu_{x,j})^2 - y_i + \mu_{d,j})^2]/2\sigma^2] \\ &\quad + \ln \prod_j \exp[\,[-(\mu_{x,j} - \mu_{d,j+1})^2 - (\mu_{d,j} - \mu_{d,j+1})^2]/\lambda] \\ &= \sum_i \ln \sum_j \exp[\,[-(x_i - \mu_{x,j})^2 - (y_i + \mu_{d,j})^2]/2\sigma^2] \\ &\quad + \sum_j \frac{-(\mu_{x,j} - \mu_{x,j+1})^2 - (\mu_{d,j} - \mu_{d,j+1})^2}{\lambda}.\end{aligned}$$

Note that the constraint concerning the distance between successive cities is encoded in the prior term by the assumption that adjacent cluster means are near one another.

We next must compute the derivative of the likelihood function with respect to the parameters $\mu_{x,j}$ and $\mu_{d,j}$, by now this should be a familiar form. We can write

$$\begin{aligned}\frac{\partial \mathcal{C}}{\partial \mu_{x,k}} &\propto \sum_i \frac{\exp[[-(x_i - \mu_{x,k})^2 - (\mathrm{d}_i + \mu_{d,k})^2]/2\sigma^2]}{\Sigma_j \exp[[-(x_i - \mu_{x,j})^2 - (d_i + \mu_{d,j})^2/2\sigma^2]} \frac{(x_i - \mu_{x,k})}{\sigma^2} \\ &\quad + \frac{1}{\lambda}(\mu_{x,k+1} - \mu_{x,k-1}).\end{aligned}$$

In order to make this work, we must imagine that the clusters form a ring so that the cluster before the cluster -1 is the last cluster and the cluster $n + 1$ (where we have n clusters) is the same as cluster 0. Now we proceed to solve the traveling salesman problem in the following way. We start out with a rather large value of σ. The cities are presented to the network one at a time, and the weights

(i.e, the means $\mu_{x,k}$ and $\mu_{d,k}$) are adjusted until the network stabilizes. At this point it is likely that none of the cluster centers are located at any of the cities. We then decrease σ and present the cities again until it stabilizes. Then σ is decreased again. This process is repeated until there is a cluster mean located at each city. At this point we can simply follow the cluster means in order and read off the solution to the problem.

Concluding Comments

In this section we showed how knowledge and constraints can be added to the network with well-chosen priors. So far the priors have been of two types. (1) We have used priors to constrain the set of networks explored by the learning algorithm. By adding such "regularization" terms we have been able to design networks which provide much better generalization. (2) We have been able to add further constraints among the network parameter relationships. These constraints allow us to force the network to a particular set of possible solutions, such as those which minimize the tour in the traveling salesman problem.

Although not discussed here, it is possible to add knowledge to the network in another way by expressing priors about the behavior of different parts of the network. It is possible to formulate priors that, for example, constrain the output of units on successive presentations to be as similar or as dissimilar as possible to one another. The general procedure can dramatically affect the solution the network achieves.

HIDDEN UNITS

Thus far, we have focused our attention on log-likelihood cost functions, appropriate interpretation of the output units, and methods of introducing additional constraints in the network. The final section focuses on the hidden units of the network.[8] There are at least four distinct ways of viewing hidden units.

1. Sigmoidal hidden units can be viewed as approximations to linear threshold functions which divide the space into regions which can then be combined to approximate the desired function.
2. Hidden units may be viewed as a set of *basis functions,* linear combinations of which can be used to approximate the desired output function.
3. Sigmoidal hidden units can be viewed probabilistically as representing the probability that certain "hidden features" are present in the input.

[8]As an historical note, the term "hidden unit" is used to refer to those units lying between the input and output layers. The name was coined by Geoffrey Hinton, inspired by the notion of "hidden states" in hidden Markov models.

4. Layers of hidden units can be viewed as a mechanism for transforming stimuli from one representation to another from layer to layer until those stimuli which are functionally similar are near one another in hidden-unit space.

In the following sections we treat each of these conceptions.

Sigmoidal Units as Continuous Approximations to Linear Threshold Functions

Perhaps the simplest way to view sigmoidal hidden units is as continuous approximations to the linear threshold function. The best way to understand this is in terms of a two-dimensional stimulus space populated by stimuli which are labeled as members of class A or class B. Figure 8 illustrates such a space. We can imagine a single output unit which is to classify the input vectors (stimuli) as being in one or the other of these classes. A simple perceptron will be able to solve this problem if the stimuli are *linearly separable;* that is, we can draw a line which puts all of the A stimuli on one side of the line and all of the B stimuli on the other side. This is illustrated in part (a) of Figure 8. In part (b) we see that replacing the sharp line of the threshold function with a "fuzzy" line of the sigmoid causes little trouble. It tends to lead to a condition in which stimuli near the border as classified less certainly than those far from the border. This may not be a bad thing since stimuli near the border may be more ambiguous.

When the stimuli are not linearly separable (as illustrated in panel (c) of the figure), the problem is more difficult and hidden units are required. In this case, each hidden unit can be seen as putting down a dividing line segmenting the input field into regions. Ideally each region will contain stimuli of the same kind. Then the weights from the hidden-unit layer to the output units are used to combine the regions which go together to form the final classification. If the stimuli are binary, or more generally if the regions in which they lie are *convex* (as they are in panel (c)), a single layer of hidden threshold units will always be sufficient. If the space is concave, as illustrated in panel (d), then two layers of threshold units may be necessary so that the right regions can be combined. It is nevertheless possible to "approximate" the regions arbitrarily closely with a single hidden layer if enough hidden units are employed. Figure 9 shows how the problem illustrated can be solved exactly with two hidden units in the two-layer case and be approximated arbitrarily closely by many hidden units.

Hidden Units as Basis Functions for Function Approximation

It is also possible to see the hidden layers as forming a set of "basis functions" and see the output units as approximating the function through a linear combina-

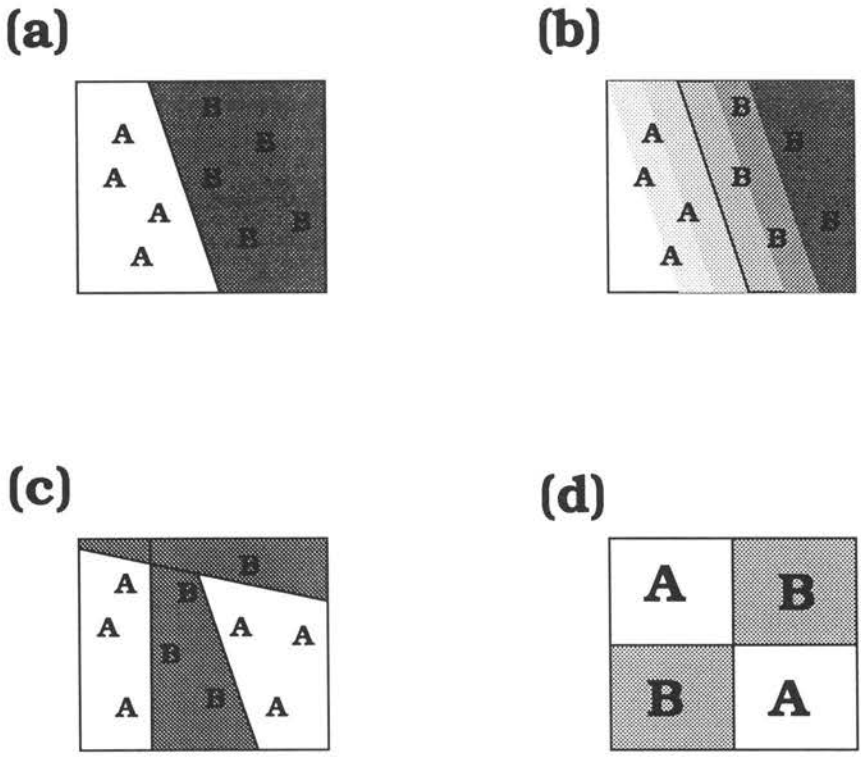

Figure 8. (a) A simple example of a linearly separable set of points. Perceptrons are capable of classifying such data sets. (b) How the same data would be classified by a sigmoid. The density of the dots indicates the magnitude of the sigmoid. If the problem is really linearly separable, the weights on the sigmoid can grow and it can act just like a perceptron. (c) A set of lines can be used to segregate a convex region. The hidden units put down a set of lines and make space that is originally not linearly separable into one that is. (d) In a concave space it might not be possible to find a set of lines which divide the two regions. In such a case two hidden layers are sometimes convenient.

tion of the hidden units. This is a view espoused by Pogio and Girosi (1989) and others employing the "radial basis function" approach. Typically, this approach involves simply substituting a Gaussian or similar radially symmetric function for the conventional sigmoidal hidden units. Of course, there is no limit to the kind of transfer function the hidden units might employ. The only real constraint (as far as back propagation is concerned) is that the functions are differentiable in their inputs and parameters. So long as this is true any hidden-unit type is possible. Certain unit types may have advantages over others, however. Among the important considerations are the problems of local minima.

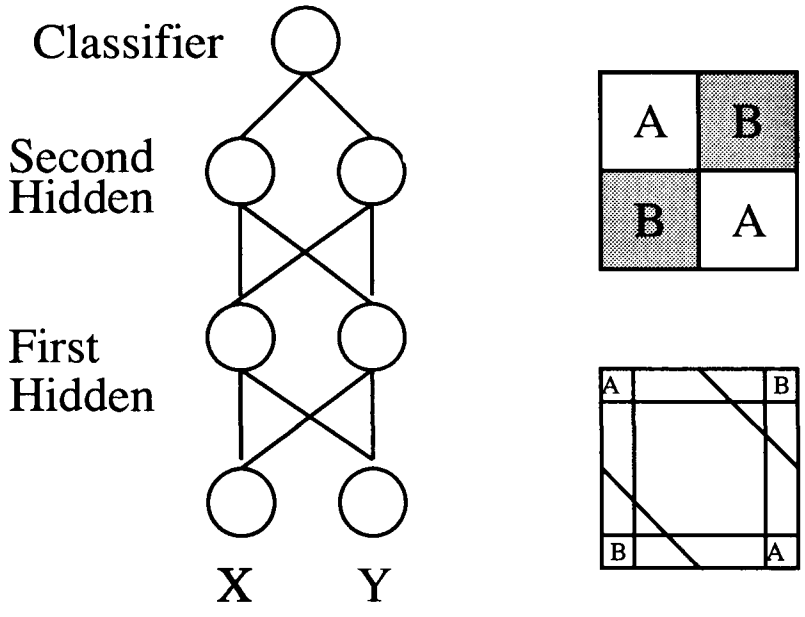

Figure 9. The first layer works by putting in the vertical and horizontal lines and moving the points to the corners of the region. This means that at the second level the problem is convex and two further hidden units divide the space and make it linearly separable.

As we will see, sigmoidal units are somewhat better behaved than many others with respect to the smoothness of the error surface. Durbin and Rumelhart (1989), for example, have found that, although "product units" $(\mathcal{F}_j(\vec{x}_i|\vec{w}_{ij}) = \Pi_i x_i^{w_{ij}})$ are much more powerful than conventional sigmoidal units (in that fewer parameters were required to represent more functions), it was a much more difficult space to search and there were more problems with local minima.

Another important consideration is the nature of the extrapolation to data points outside the local region from which the data were collected. Radial basis functions have the advantage that they go to zero as you extend beyond the region where the data were collected. Polynomial units $\mathcal{F}_j(\eta_j) = \eta_k^p$ are very ill behaved outside of the training region and for that reason are not especially good choices. Sigmoids are well behaved outside of their local region in that they saturate and are constant at 0 or 1 outside of the training region.

Sigmoidal Hidden Units as Representing Hidden Feature Probabilities

The sigmoidal hidden unit has turned out to be a serendipitous choice. It has a number of nice properties and interpretations which make it rather useful. The

first property has to do with the learning process itself. As noted from Figure 10, the sigmoidal unit is roughly linear for small weights (a net input near zero) and gets increasingly nonlinear in its response as it approaches its points of maximum curvature on either side of the midpoint. Thus, at the beginning of learning, when the weights are small, the system is mainly in its linear range and is seeking an essentially linear solution. As the weights grow, the network becomes increasing nonlinear and begins to move toward the nonlinear solution to the problem. This property of initial linearity makes the units rather robust and allows the network to reliably attain the same solution.

Sigmoidal hidden units have a useful interpretation as the posterior probability of the presence of some feature given the input. To see this, think of a sigmoidal hidden unit connected directly to the input units. Suppose that the input vectors are random variables drawn from one of two probability distributions and that the job of the hidden unit is to determine which of the two distributions is being observed. The role of the hidden unit is to give an output value equal to the probability that the input vector was drawn from distribution 1 rather than distribution 2. If drawn from distribution 1 we say that some "hidden feature" was present; otherwise we say it was absent. Denoting the hidden feature for the jth hidden unit as f_j we have

$$P(f_j = 1|\vec{x}) = \frac{P(\vec{x}|f_j = 1)P(f_j = 1)}{P(\vec{x}|f_j = 1)P(f_j = 1) + P(\vec{x}|f_j = 0)P(f_j = 0)}.$$

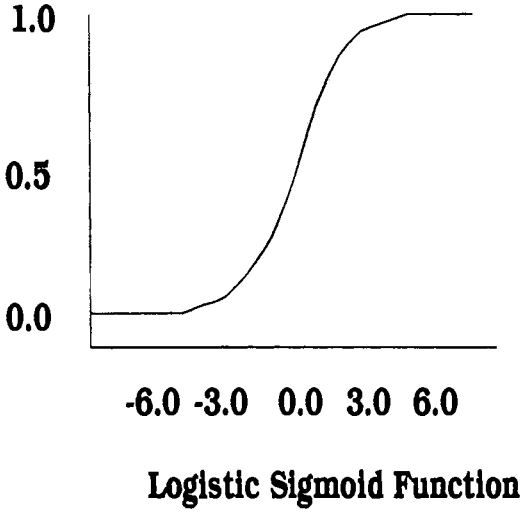

Figure 10. The logistic sigmoid is roughly linear near the middle of its range and reaches its maximum curvature.

Now, on the assumption that the x's are *conditionally independent* (i.e., if we know which distribution they were drawn from, there is a fixed probability for each input element that it will occur), we can write

$$P(\vec{x}|f_j = 1) = \Pi_i P(x_i|f_j = 1) \qquad \text{and} \qquad P(\vec{x}|f_j = 0) = \Pi_i P(x_i|f_j = 0).$$

Now, on the further assumption that the x's are binomially distributed we get

$$P(\vec{x}|f_j = 1) = \prod_i p_{ij}^{x_i}(1 - p_{ij})^{(1-x_i)},$$

$$P(\vec{x}|f_j = 0) = \prod_i q_{ij}^{x_i}(1 - q_{ij})^{(1-x_i)}.$$

So we finally have

$$P(f_j = 1|\vec{x}) = \frac{\Pi_i p_{ij}^{x_i}(1 - p_{ij})^{(1-x_i)}P(f_j = 1)}{\Pi_i p_{ij}^{x_i}(1 - p_{ij})^{(1-x_i)}P(f_j = 1) + \Pi_i q_{ij}^{x_i}(1 - q_{ij})^{(1-x_i)}P(f_j = 0}.$$

Taking logs and exponentiating give

$$P(f_j = 1|\vec{x})$$

$$= \frac{\exp\left\{\sum_i (x_i \ln p_{ij} + (1 - x_i)\ln(1 - p_{ij})) + \ln P(f_j = 1\right\}}{\exp\left\{\sum_i (x_i \ln p_{ij} + (1 - x_i)\ln(1 - p_{ij})) + \ln P(f_j = 1)\right\} + \exp\left\{\sum_i (x_i \ln q_{ij} + (1 - x_i)\ln(1 - q_{ij})) + \ln P(f_j = 0)\right\}}$$

$$= \frac{\exp\left\{\sum_i \left(x_i \ln \frac{p_{ij}}{1 - p_{ij}} + \sum_i \ln(1 - p_{ij})\right) + \ln P(f_j = 1)\right\}}{\exp\left\{\sum_i \left(x_i \ln \frac{p_{ij}}{1 - p_{ij}} + \sum_i \ln(1 - p_{ij})\right) + \ln P(f_j = 1)\right\} + \exp\left\{\sum_i \left(x_i \ln \frac{q_{ij}}{1 - q_{ij}} + \sum_i \ln(1 - q_{ij})\right) + \ln P(f_j = 0)\right\}}$$

$$= \frac{1}{1 + \exp\left\{-\sum_i \ln \frac{p_{ij}(1 - q_{ij})}{(1 - p_{ij})q_{ij}} + \sum_i \ln \frac{1 - p_{ij}}{1 - q_{ij}} + \ln \frac{P(f_j = 1)}{P(f_j = 0)}\right\}}.$$

Now, it is possible to interpret the exponent as representing η_j, the net input to the unit. If we let

$$\beta_j = \sum_i \ln \frac{1 - p_{ij}}{1 - q_{ij}} + \ln \frac{P(f_j = 1)}{P(f_j = 0)}$$

and

$$w_{ij} = \ln \frac{p_{ij}(1 - q_{ij})}{(1 - p_{ij})q_{ij}},$$

we can see the similarity. We can thus see that the sigmoid is properly understood as representing the posterior probability that some hidden feature is present given

the input. Note that, as before, the binomial assumption is not necessary. It is possible to assume that the underlying input vectors are members of any of the exponential family of probability distributions. It is easy to write the general form

$$P(f_j = 1|\vec{x}) = \frac{\exp\left\{\sum_i \frac{(x_i\theta_i - B(\theta_i)) + C(\vec{x},\phi)}{a(\phi)} + \ln P(f_j = 1)\right\}}{\exp\left\{\sum_i \frac{(x_i\theta_i - B(\theta_i)) + C(\vec{x}\phi)}{a(\phi)} + \ln P(f_j = 1)\right\} + \exp\left\{\sum_i \frac{(x_i\theta_i^* - B(\theta_i^*) + C(\vec{x},\phi))}{a(\phi)}\right\} + \ln P(f_j = 0)}.$$

By rearranging terms and dividing through by the numerator, we obtain the simple form

$$P(f_j = 1|\vec{x}) = \left(1 + \exp\left\{-\left(\sum_i x_i \frac{\theta_i - \theta_i^*}{a(\phi)} + \sum_i \frac{B(\theta_i) - B(\theta_i^*)}{a(\phi)} + \ln\frac{P(f_j = 1)}{P(f_j = 0)}\right)\right\}\right)^{-1}$$

$$= \frac{1}{1 + \exp\left[\left(\sum_i x_i w_i + \beta_i\right)\right]}$$

Thus, under the assumption that the input variables are drawn from some member of the exponential family and differ only in their means (represented by θ_i), the sigmoidal hidden unit can be interpreted as the probability that the hidden feature is present. Note the very same form is derived whether the underlying distributions are Gaussian, binomial, or any member of the exponential family. It can readily be seen that, whereas the sigmoid represents the two-alternative case, the normalized exponential clearly represents the multialternative case. Thus, we derive the normalized exponential in exactly the same way as we derive the sigmoid:

$$P(c_j = 1|\vec{x}) = \frac{\exp\{\Sigma_i[(x_{ij}\theta_i - B(\theta_{ij})) + C(\vec{x}\phi)]/a(\phi) + \ln P(c_j = 1)\}}{\Sigma_k \exp\{\Sigma_i[(x_i\theta_{ik} - B(\theta_{ik})) + C(\vec{x}\phi)]/a(\phi) + \ln P(c_k = 1)\}}$$

$$= \frac{\exp\{\Sigma_i x_{ij}\theta_i/a(\phi) - \Sigma_i B(\theta_{ij})/a(\phi) + \ln P(c_j = 1)\}}{\Sigma_k \exp\{\Sigma_i x_i \theta_{ik}/a(\phi) - B(\theta_{ik})/a(\phi) + \ln P(c_k = 1)\}}$$

$$= \frac{e^{\Sigma_i x_{ij} w_{ij} + \beta_j}}{\Sigma_k e^{\Sigma_i x_i w_{ik} + \beta_k}}.$$

Hidden-Unit Layers as Representations of the Input Stimuli

Figure 11 illustrates a very simple connectionist network consisting of two layers of units, the input units and output units, connected by a set of weights. As a result of the particular connectivity and weights of this network, each pattern of activation presented at the input units will induce another specific pattern of activation at the output units. This simple architecture is useful in various ways. If the input and output patterns all use distributed representations (i.e., can all be

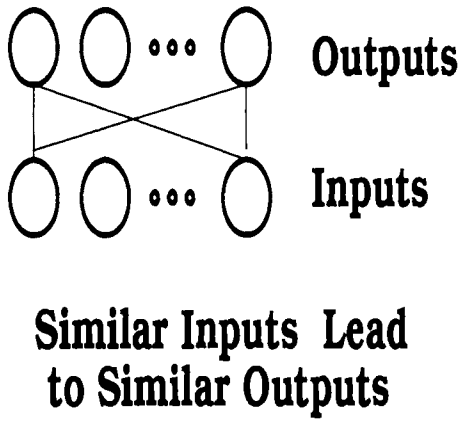

Figure 11. Similar inputs produce similar outputs.

described as sets of microfeatures), then this network will exhibit the property that "similar inputs yield similar outputs," along with the accompanying generalization and transfer of learning. Two-layer networks behave this way because the activation of an output unit is given by a relatively smooth function of the weighted sum of its inputs. Thus, a slight change in the value of an input unit will generally yield a similarly slight change in the values of the output units.

Although this similarity-based processing is mostly useful, it does not always yield the correct generalizations. In particular, in a simple two-layer network, the similarity metric employed is determined by the nature of the inputs themselves. And the "physical similarity" we are likely to have at the inputs (based on the structure of stimuli from the physical world) may not be the best measure of the "functional" or "psychological" similarity we would like to employ at the output (to group appropriate similar responses). For example, it is probably true that a lowercase *a* is physically less similar to an uppercase *A* than to a lowercase *o*, but functionally and psychologically a *a* and *A* are more similar to one another than are the two lowercase letters. Thus, physical relatedness is an inadequate similarity metric for modeling human responses to letter-shaped visual inputs. It is therefore necessary to transform these input patterns from their initial physically derived format into another representational form in which patterns requiring similar (output) responses are indeed similar to one another. This involves learning new representations.

Figure 12 illustrates a layered feedforward network in which information (activation) flows up from the input units at the bottom through successive layers of hidden units, to create the final response at the layer of output units on top. Such a network is useful for illustrating how an appropriate psychological or functional representation can be created. If we think of each input vector as a point in some multidimensional space, we can think of the similarity between

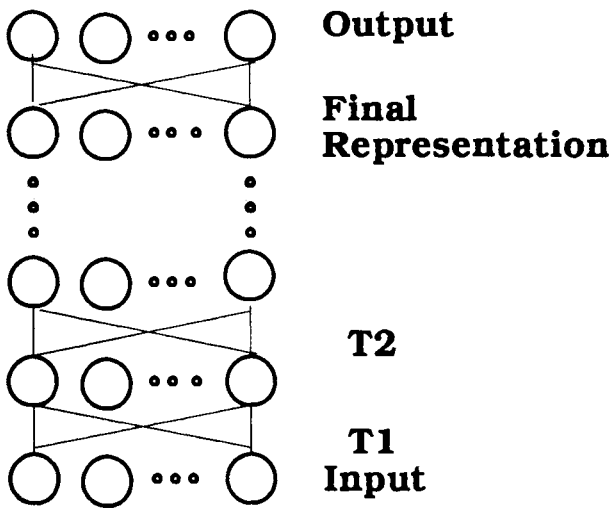

Figure 12. We can think of multilayer networks as transforming the input through a series of successive transformations so as to create a representation in which "functionally" similar stimuli are near one another when viewed as points in a multidimensional space.

two such vectors as the distance between their two corresponding points. Furthermore, we can think of the weighted connections from one layer of units to the next as implementing a transformation that maps each original input vector into some new vector. This transformation can create a new vector space in which the relative distances among the points corresponding to the input vectors are different from those in the original vector space, essentially rearranging the points. And if we use a sequence of such transformations, each involving certain nonlinearities, by "stacking" them between successive layers in the network, we can entirely rearrange the similarity relations among the original input vectors.

Thus, a layered network can be viewed simply as a mechanism for transforming the original set of input stimuli into a new similarity space with a new set of distances among the input points. For example, it is possible to move the initially distant "physical" input representations of *a* and *A* so that they are very close to one another in a transformed "psychological" output representation space, and simultaneously transform the distance between *a* and *o* output representations so that they are rather distant from one another. (Generally, we seek to attain a representation in the second-to-last layer which is sufficiently transformed that we can rely on the principle that similar patterns yield similar outputs at the final layer.) The problem is to find an appropriate sequence of transformations that accomplish the desired input-to-output change in similarity structures.

The back-propagation learning algorithm can be viewed, then, as a procedure for discovering such a sequence of transformations. In fact, we can see the role

of learning in general as a mechanism for constructing the transformations which will convert the original physically based configuration of the input vectors into an appropriate functional or psychological space, with the proper similarity relationships between concepts for making generalizations and transfer of learning occur automatically and correctly.

CONCLUSION

In this chapter we have tried to provide a kind of overview and rationale for the design and understanding of networks. Although it is possible to design and use interesting networks without any of the ideas presented here, it is, in our experience, very valuable to understand networks in terms of these probabilistic interpretations. The value is primarily in providing an understanding of the networks and their behavior so that one can craft an appropriate network for an appropriate problem. Although it has been commonplace to view networks as kinds of black boxes, this leads to inappropriate applications which may fail not because such networks cannot work but because the issues are not well understood.

REFERENCES

Cover, T. H. (1965). Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers, 14*. pp. 326–334

Durbin, R., & Rumelhart, D. E. (1989). Product units: A computationally powerful and biologically lausible extension to backpropagation networks. *Neural Computation, 1*, 133–142.

Durbin, R., & Willshaw, D. (1987). An analogue approach to the travelling salesman problem using an elastic net method. *Nature, 326*, 689–691.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer Feed-forward Networks are Universal Approximators, *Neural Networks, 2*, pp. 359–366.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., & Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural Computation, 3*(1).

Jordan, M. I., & Rumelhart, D. E. (1992). Forward models: Supervised learning with a distal teacher. *Cognitive Science, 16*, pp. 307–354.

Keeler, J. D., Rumelhart, D. E., & Loew, W. (1991). Integrated segmentation and recognition of hand-printed numerals. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky (Eds.), *Neural information processing systems* (Vol. 3). San Mateo, CA: Morgan Kaufmann.

Kolmogorov, A. N. (1991). *Selected Works of A. N. Kolmogorov,* Dordrecht; Boston; Kluwer Academic.

Le Cun, Y., Boser, Y. B., Denke, J. S., Henderson, R. D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1990). In D. S. Touretzky (Ed.), *Handwritten digit recognition with a back-propagation network* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

McCullagh, P. & Nelder, J. A. (1989). *Generalized linear models*. London: Chapman and Hall.

Mitchison, G. J., & Durbin, R. M. (1989). Bounds on the learning capacity of some multi-layer networks. *Biological Cybernetics, 60*, 345–356.

Nowlan, S. J. (1991). *Soft Competitive Adaptation: Neural Network Learning Algorithm based on*

Fitting Statistical Mixtures. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Parker, D. B. (1982). *Learning-logic* (Invention Report S81-64, File 1). Stanford, CA: Office of Technology Licensing, Stanford University.

Pogio, T., & Girosi, F. (1989). *A theory of networks for approximation and learning*. A. I. Memo No. 1140, Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

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

Rumelhart, D. E. (1990). Brain style computation: Learning and generalization. In S. F. Zornetzer, J. L. Davis, and C. Lau (Eds.), *An introduction to neural and electronic networks*. San Diego: Academic Press.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* (Vol. 1). Cambridge, MA: Bradford Books.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing. 37*, 328–338.

Weigend, A. S., Huberman, B. A., & Rumelhart, D. E. (1990). Predicting the future: A connectionist approach. *International Journal of Neural Systems, 1*, 193–209.

Weigend, A. S., Rumelhart, D. E., & Huberman, B. (1991). Generalization by weight-elimination with application to forecasting. In R. P. Lippman, J. Moody, and D. S. Touretsky (Eds.), *Advances in neural information processing* (Vol. 3, pp. 875–882). San Mateo, CA: Morgan Kaufman.

Werbos, P. (1974). *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. Unpublished dissertation, Harvard University.