

There are many other applications, such as density estimation (Friedman et al., 1984; Friedman, 1987), where the projection pursuit idea can be used. In particular, see the discussion of ICA in Section 14.7 and its relationship with exploratory projection pursuit. However the projection pursuit regression model has not been widely used in the field of statistics, perhaps because at the time of its introduction (1981), its computational demands exceeded the capabilities of most readily available computers. But it does represent an important intellectual advance, one that has blossomed in its reincarnation in the field of neural networks; the topic of the rest of this chapter.

11.3 Neural Networks

The term *neural network* has evolved to encompass a large class of models and learning methods. Here we describe the most widely used “vanilla” neural net, sometimes called the single hidden layer back-propagation network, or single layer perceptron. There has been a great deal of *hype* surrounding neural networks, making them seem magical and mysterious. As we make clear in this section, they are just nonlinear statistical models, much like the projection pursuit regression model discussed above.

A neural network is a two-stage regression or classification model, typically represented by a *network diagram* as in Figure 11.2. This network applies both to regression or classification. For regression, typically $K = 1$ and there is only one output unit Y_1 at the top. However, these networks can handle multiple quantitative responses in a seamless fashion, so we will deal with the general case.

For K -class classification, there are K units at the top, with the k th unit modeling the probability of class k . There are K target measurements Y_k , $k = 1, \dots, K$, each being coded as a 0–1 variable for the k th class.

Derived features Z_m are created from linear combinations of the inputs, and then the target Y_k is modeled as a function of linear combinations of the Z_m ,

$$\begin{aligned} Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), \quad m = 1, \dots, M, \\ T_k &= \beta_{0k} + \beta_k^T Z, \quad k = 1, \dots, K, \\ f_k(X) &= g_k(T), \quad k = 1, \dots, K, \end{aligned} \quad (11.5)$$

where $Z = (Z_1, Z_2, \dots, Z_M)$, and $T = (T_1, T_2, \dots, T_K)$.

The activation function $\sigma(v)$ is usually chosen to be the *sigmoid* $\sigma(v) = 1/(1 + e^{-v})$; see Figure 11.3 for a plot of $1/(1 + e^{-v})$. Sometimes Gaussian radial basis functions (Chapter 6) are used for the $\sigma(v)$, producing what is known as a *radial basis function network*.

Neural network diagrams like Figure 11.2 are sometimes drawn with an additional *bias* unit feeding into every unit in the hidden and output layers.

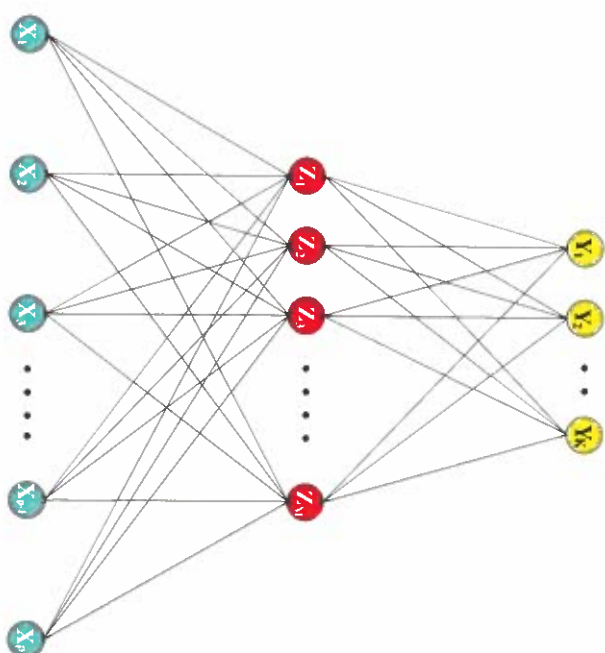


FIGURE 11.2. Schematic of a single hidden layer, feed-forward neural network.

Thinking of the constant “1” as an additional input feature, this bias unit captures the intercepts α_{0m} and β_{0k} in model (11.5).

The output function $g_k(T)$ allows a final transformation of the vector of outputs T . For regression we typically choose the identity function $g_k(T) = T_k$. Early work in K -class classification also used the identity function, but this was later abandoned in favor of the *softmax* function

$$g_k(T) = \frac{e^{T_k}}{\sum_{\ell=1}^K e^{T_\ell}}. \quad (11.6)$$

This is of course exactly the transformation used in the multilogit model (Section 4.4), and produces positive estimates that sum to one. In Section 4.2 we discuss other problems with linear activation functions, in particular potentially severe masking effects.

The units in the middle of the network, computing the derived features Z_m , are called *hidden units* because the values Z_m are not directly observed. In general there can be more than one hidden layer, as illustrated in the example at the end of this chapter. We can think of the Z_m as a basis expansion of the original inputs X ; the neural network is then a standard linear model, or linear multilogit model, using these transformations as inputs. There is, however, an important enhancement over the basis-expansion techniques discussed in Chapter 5; here the parameters of the basis functions are learned from the data.

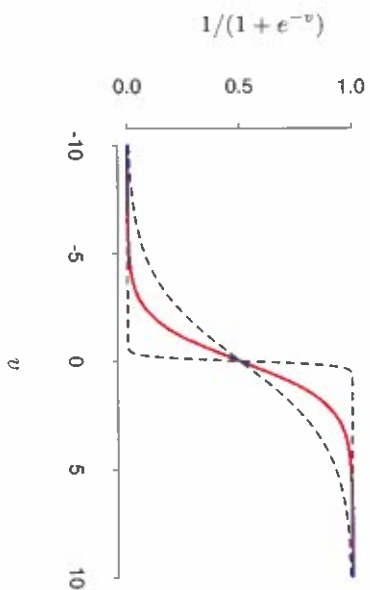


FIGURE 11.3. Plot of the sigmoid function $\sigma(v) = 1/(1 + \exp(-v))$ (red curve), commonly used in the hidden layer of a neural network. Included are $\sigma(sv)$ for $s = \frac{1}{2}$ (blue curve) and $s = 10$ (purple curve). The scale parameter s controls the activation rate, and we can see that large s amounts to a hard activation at $v = 0$. Note that $\sigma(s(v - v_0))$ shifts the activation threshold from 0 to v_0 .

Notice that if σ is the identity function, then the entire model collapses to a linear model in the inputs. Hence a neural network can be thought of as a nonlinear generalization of the linear model, both for regression and classification. By introducing the nonlinear transformation σ , it greatly enlarges the class of linear models. In Figure 11.3 we see that the rate of activation of the sigmoid depends on the norm of α_m , and if $\|\alpha_m\|$ is very small, the unit will indeed be operating in the *linear part* of its activation function.

Notice also that the neural network model with one hidden layer has exactly the same form as the projection pursuit model described above. The difference is that the PPR model uses nonparametric functions $g_m(v)$, while the neural network uses a far simpler function based on $\sigma(v)$, with three free parameters in its argument. In detail, viewing the neural network model as a PPR model, we identify

$$\begin{aligned} g_m(\omega_m^T X) &= \beta_m \sigma(\alpha_0 m + \alpha_m^T X) \\ &= \beta_m \sigma(\alpha_0 m + \|\alpha_m\|(\omega_m^T X)), \end{aligned} \quad (11.7)$$

where $\omega_m = \alpha_m / \|\alpha_m\|$ is the m th unit-vector. Since $\sigma_{\beta, \alpha_0, s}(v) = \beta \sigma(\alpha_0 + sv)$ has lower complexity than a more general nonparametric $g(v)$, it is not surprising that a neural network might use 20 or 100 such functions, while the PPR model typically uses fewer terms ($M = 5$ or 10, for example).

Finally, we note that the name “neural networks” derives from the fact that they were first developed as models for the human brain. Each unit represents a neuron, and the connections (links in Figure 11.2) represent synapses. In early models, the neurons fired when the total signal passed to that unit exceeded a certain threshold. In the model above, this corresponds

to use of a step function for $\sigma(Z)$ and $g_m(T)$. Later the neural network was recognized as a useful tool for nonlinear statistical modeling, and for this purpose the step function is not smooth enough for optimization. Hence the step function was replaced by a smoother threshold function, the sigmoid in Figure 11.3.

11.4 Fitting Neural Networks

The neural network model has unknown parameters, often called *weights*, and we seek values for them that make the model fit the training data well. We denote the complete set of weights by θ , which consists of

$$\begin{aligned} \{\alpha_{0m}, \alpha_m; m = 1, 2, \dots, M\} \quad M(p + 1) \text{ weights,} \\ \{\beta_{0k}, \beta_k; k = 1, 2, \dots, K\} \quad K(M + 1) \text{ weights.} \end{aligned} \quad (11.8)$$

For regression, we use sum-of-squared errors as our measure of fit (error function)

$$R(\theta) = \sum_{k=1}^K \sum_{i=1}^N (y_{ik} - f_k(x_i))^2. \quad (11.9)$$

For classification we use either squared error or cross-entropy (deviance):

$$R(\theta) = - \sum_{i=1}^N \sum_{k=1}^K y_{ik} \log f_k(x_i), \quad (11.10)$$

and the corresponding classifier is $G(x) = \operatorname{argmax}_k f_k(x)$. With the softmax activation function and the cross-entropy error function, the neural network model is exactly a linear logistic regression model in the hidden units, and all the parameters are estimated by maximum likelihood.

Typically we don't want the global minimizer of $R(\theta)$, as this is likely to be an overfit solution. Instead some regularization is needed: this is achieved directly through a penalty term, or indirectly by early stopping. Details are given in the next section.

The generic approach to minimizing $R(\theta)$ is by gradient descent, called *back-propagation* in this setting. Because of the compositional form of the model, the gradient can be easily derived using the chain rule for differentiation. This can be computed by a forward and backward sweep over the network, keeping track only of quantities local to each unit.

Here is back-propagation in detail for squared error loss. Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^T x_i)$, from (11.5) and let $z_i = (z_{i1}, z_{i2}, \dots, z_{Mi})$. Then we have

$$\begin{aligned} R(\theta) &= \sum_{i=1}^N R_i \\ &= \sum_{i=1}^N \sum_{k=1}^K (y_{ik} - f_k(x_i))^2, \end{aligned} \quad (11.11)$$

with derivatives

$$\begin{aligned} \frac{\partial R_i}{\partial \beta_{km}} &= -2(y_{ik} - f_k(x_i))g'_k(\beta_k^T z_i)z_{mi}, \\ \frac{\partial R_i}{\partial \alpha_{m\ell}} &= -\sum_{k=1}^K 2(y_{ik} - f_k(x_i))g'_k(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{i\ell}. \end{aligned} \quad (11.12)$$

Given these derivatives, a gradient descent update at the $(r+1)$ st iteration has the form

$$\begin{aligned} \beta_{km}^{(r+1)} &= \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^N \frac{\partial R_i}{\partial \beta_{km}}^{(r)}, \\ \alpha_{m\ell}^{(r+1)} &= \alpha_{m\ell}^{(r)} - \gamma_r \sum_{i=1}^N \frac{\partial R_i}{\partial \alpha_{m\ell}}^{(r)}, \end{aligned} \quad (11.13)$$

where γ_r is the *learning rate*, discussed below.

Now write (11.12) as

$$\begin{aligned} \frac{\partial R_i}{\partial \beta_{km}} &= \delta_{ki} z_{mi}, \\ \frac{\partial R_i}{\partial \alpha_{m\ell}} &= s_{mi} x_{i\ell}. \end{aligned} \quad (11.14)$$

The quantities δ_{ki} and s_{mi} are “errors” from the current model at the output and hidden layer units, respectively. From their definitions, these errors satisfy

$$s_{mi} = \sigma'(\alpha_m^T x_i) \sum_{k=1}^K \beta_{km} \delta_{ki}, \quad (11.15)$$

known as the *back-propagation equations*. Using this, the updates in (11.13) can be implemented with a two-pass algorithm. In the *forward pass*, the current weights are fixed and the predicted values $f_k(x_i)$ are computed from formula (11.5). In the *backward pass*, the errors δ_{ki} are computed, and then back-propagated via (11.15) to give the errors s_{mi} . Both sets of errors are then used to compute the gradients for the updates in (11.13), via (11.14).

This two-pass procedure is what is known as back-propagation. It has also been called the *delta rule* (Widrow and Hoff, 1960). The computational components for cross-entropy have the same form as those for the sum of squares error function, and are derived in Exercise 11.3.

The advantages of back-propagation are its simple, local nature. In the back propagation algorithm, each hidden unit passes and receives information only to and from units that share a connection. Hence it can be implemented efficiently on a parallel architecture computer.

The updates in (11.13) are a kind of *batch learning*, with the parameter updates being a sum over all of the training cases. Learning can also be carried out online—processing each observation one at a time, updating the gradient after each training case, and cycling through the training cases many times. In this case, the sums in equations (11.13) are replaced by a single summand. A *training epoch* refers to one sweep through the entire training set. Online training allows the network to handle very large training sets, and also to update the weights as new observations come in.

The learning rate γ_r for batch learning is usually taken to be a constant, and can also be optimized by a line search that minimizes the error function at each update. With online learning γ_r should decrease to zero as the iteration $r \rightarrow \infty$. This learning is a form of *stochastic approximation* (Robbins and Munro, 1951): results in this field ensure convergence if $\gamma_r \rightarrow 0$, $\sum_r \gamma_r = \infty$, and $\sum_r \gamma_r^2 < \infty$ (satisfied, for example, by $\gamma_r = 1/r$).

Back-propagation can be very slow, and for that reason is usually not the method of choice. Second-order techniques such as Newton’s method are not attractive here, because the second derivative matrix of R (the Hessian) can be very large. Better approaches to fitting include conjugate gradients and variable metric methods. These avoid explicit computation of the second derivative matrix while still providing faster convergence.

11.5 Some Issues in Training Neural Networks

There is quite an art in training neural networks. The model is generally overparameterized, and the optimization problem is nonconvex and unstable unless certain guidelines are followed. In this section we summarize some of the important issues.

11.5.1 Starting Values

Note that if the weights are near zero, then the operative part of the sigmoid (Figure 11.3) is roughly linear, and hence the neural network collapses into an approximately linear model (Exercise 11.2). Usually starting values for weights are chosen to be random values near zero. Hence the model starts out nearly linear, and becomes nonlinear as the weights increase. Individual

units localize to directions and introduce nonlinearities where needed. Use of exact zero weights leads to zero derivatives and perfect symmetry, and the algorithm never moves. Starting instead with large weights often leads to poor solutions.

11.5.2 Overfitting

Often neural networks have too many weights and will overfit the data at the global minimum of R . In early developments of neural networks, either by design or by accident, an early stopping rule was used to avoid overfitting. Here we train the model only for a while, and stop well before we approach the global minimum. Since the weights start at a highly regularized (linear) solution, this has the effect of shrinking the final model toward a linear model. A validation dataset is useful for determining when to stop, since we expect the validation error to start increasing.

A more explicit method for regularization is *weight decay*, which is analogous to ridge regression used for linear models (Section 3.4.1). We add a penalty to the error function $R(\theta) + \lambda J(\theta)$, where

$$J(\theta) = \sum_{k,m} \beta_{km}^2 + \sum_{m,\ell} \alpha_{m\ell}^2 \quad (11.16)$$

and $\lambda \geq 0$ is a tuning parameter. Larger values of λ will tend to shrink the weights toward zero: typically cross-validation is used to estimate λ . The effect of the penalty is to simply add terms $2\lambda\beta_{km}$ and $2\lambda\alpha_{m\ell}$ to the respective gradient expressions (11.13). Other forms for the penalty have been proposed, for example,

$$J(\theta) = \sum_{k,m} \frac{\beta_{km}^2}{1 + \beta_{km}^2} + \sum_{m,\ell} \frac{\alpha_{m\ell}^2}{1 + \alpha_{m\ell}^2}; \quad (11.17)$$

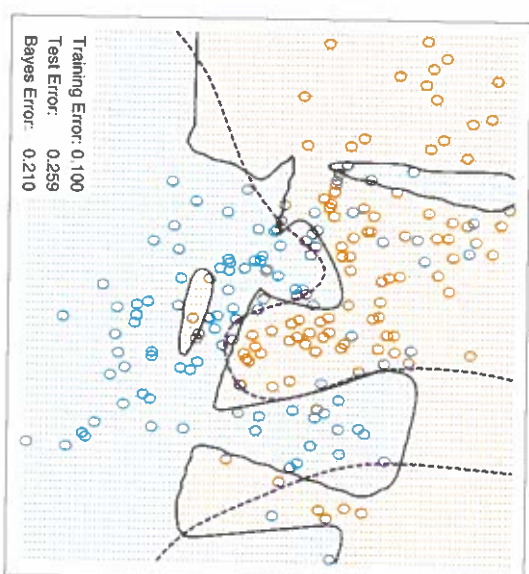
known as the *weight elimination* penalty. This has the effect of shrinking smaller weights more than (11.16) does.

Figure 11.4 shows the result of training a neural network with ten hidden units, without weight decay (upper panel) and with weight decay (lower panel), to the mixture example of Chapter 2. Weight decay has clearly improved the prediction. Figure 11.5 shows heat maps of the estimated weights from the training (grayscale versions of these are called *Hinton diagrams*.) We see that weight decay has dampened the weights in both layers: the resulting weights are spread fairly evenly over the ten hidden units.

11.5.3 Scaling of the Inputs

Since the scaling of the inputs determines the effective scaling of the weights in the bottom layer, it can have a large effect on the quality of the final

Neural Network - 10 Units, No Weight Decay



Neural Network - 10 Units, Weight Decay=0.02

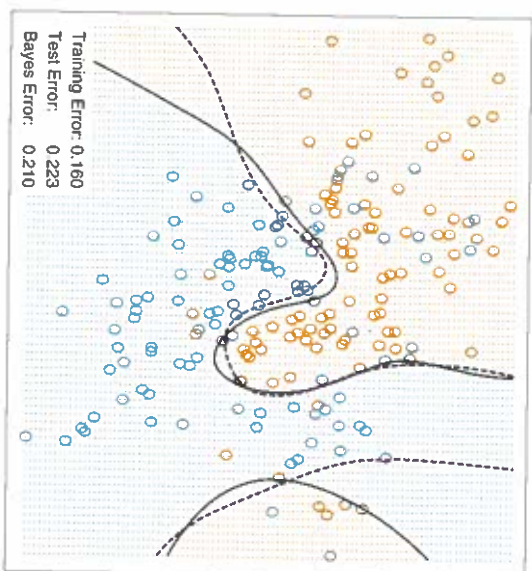


FIGURE 11.4. A neural network on the mixture example of Chapter 2. The upper panel uses no weight decay, and overfits the training data. The lower panel uses weight decay, and achieves close to the Bayes error rate (broken purple boundary). Both use the softmax activation function and cross-entropy error.

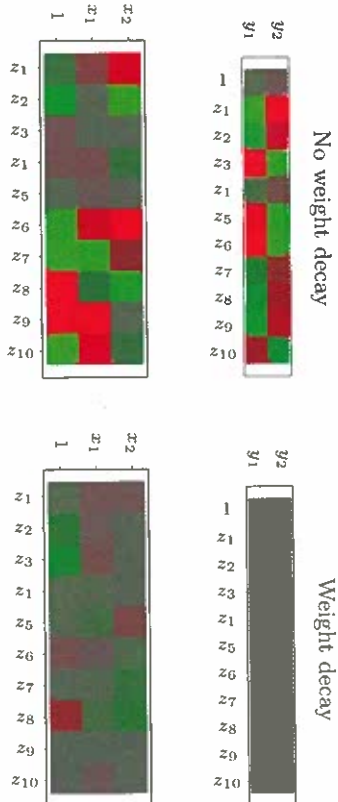


FIGURE 11.5. Heat maps of the estimated weights from the training of neural networks from Figure 11.4. The display ranges from bright green (negative) to bright red (positive).

solution. At the outset it is best to standardize all inputs to have mean zero and standard deviation one. This ensures all inputs are treated equally in the regularization process, and allows one to choose a meaningful range for the random starting weights. With standardized inputs, it is typical to take random uniform weights over the range $[-0.7, +0.7]$.

11.5.4 Number of Hidden Units and Layers

Generally speaking it is better to have too many hidden units than too few. With too few hidden units, the model might not have enough flexibility to capture the nonlinearities in the data; with too many hidden units, the extra weights can be shrunk toward zero if appropriate regularization is used. Typically the number of hidden units is somewhere in the range of 5 to 100, with the number increasing with the number of inputs and number of training cases. It is most common to put down a reasonably large number of units and train them with regularization. Some researchers use cross-validation to estimate the optimal number, but this seems unnecessary if cross-validation is used to estimate the regularization parameter. Choice of the number of hidden layers is guided by background knowledge and experimentation. Each layer extracts features of the input for regression or classification. Use of multiple hidden layers allows construction of hierarchical features at different levels of resolution. An example of the effective use of multiple layers is given in Section 11.6.

11.5.5 Multiple Minima

The error function $R(\theta)$ is nonconvex, possessing many local minima. As a result, the final solution obtained is quite dependent on the choice of start-

ing weights. One must at least try a number of random starting configurations, and choose the solution giving lowest (penalized) error. Probably a better approach is to use the average predictions over the collection of networks as the final prediction (Ripley, 1996). This is preferable to averaging the weights, since the nonlinearity of the model implies that this averaged solution could be quite poor. Another approach is via *bagging*, which averages the predictions of networks training from randomly perturbed versions of the training data. This is described in Section 8.7.

11.6 Example: Simulated Data

We generated data from two additive error models $Y = f(X) + \epsilon$:

$$\text{Sum of sigmoids: } Y = \sigma(a_1^T X) + \sigma(a_2^T X) + \epsilon_1;$$

$$\text{Radial: } Y = \prod_{m=1}^{10} \phi(X_m) + \epsilon_2.$$

Here $X^T = (X_1, X_2, \dots, X_p)$, each X_j being a standard Gaussian variate, with $p = 2$ in the first model, and $p = 10$ in the second.

For the sigmoid model, $a_1 = (3, 3)$, $a_2 = (3, -3)$; for the radial model, $\phi(t) = (1/2\pi)^{1/2} \exp(-t^2/2)$. Both ϵ_1 and ϵ_2 are Gaussian errors, with variance chosen so that the signal-to-noise ratio

$$\frac{\text{Var}(E(Y|X))}{\text{Var}(Y - E(Y|X))} = \frac{\text{Var}(f(X))}{\text{Var}(\epsilon)} \quad (11.18)$$

is 4 in both models. We took a training sample of size 100 and a test sample of size 10,000. We fit neural networks with weight decay and various numbers of hidden units, and recorded the average test error $\text{Err}_{\text{test}}(Y - f(X))^2$ for each of 10 random starting weights. Only one training set was generated, but the results are typical for an “average” training set. The test errors are shown in Figure 11.6. Note that the zero hidden unit model refers to linear least squares regression. The neural network is perfectly suited to the sum of sigmoids model, and the two-unit model does perform the best, achieving an error close to the Bayes rate. (Recall that the Bayes rate for regression with squared error is the error variance; in the figures, we report test error relative to the Bayes error). Notice, however, that with more hidden units, overfitting quickly creeps in, and with some starting weights the model does worse than the linear model (zero hidden unit) model. Even with two hidden units, two of the ten starting weight configurations produced results no better than the linear model, confirming the importance of multiple starting values.

A radial function is in a sense the most difficult for the neural net, as it is spherically symmetric and with no preferred directions. We see in the right

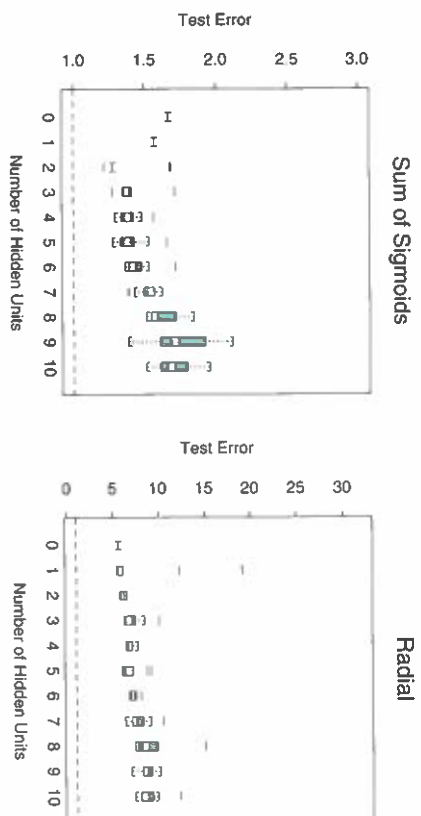


FIGURE 11.6. Boxplots of test error, for simulated data example, relative to the Bayes error (broken horizontal line). True function is a sum of two sigmoids on the left, and a radial function is on the right. The test error is displayed for 10 different starting weights, for a single hidden layer neural network with the number of units as indicated.

panel of Figure 11.6 that it does poorly in this case, with the test error staying well above the Bayes error (note the different vertical scale from the left panel). In fact, since a constant fit (such as the sample average) achieves a relative error of 5 (when the SNR is 4), we see that the neural networks perform increasingly worse than the mean.

In this example we used a fixed weight decay parameter of 0.0005, representing a mild amount of regularization. The results in the left panel of Figure 11.6 suggest that more regularization is needed with greater numbers of hidden units.

In Figure 11.7 we repeated the experiment for the sum of sigmoids model, with no weight decay in the left panel, and stronger weight decay ($\lambda = 0.1$) in the right panel. With no weight decay, overfitting becomes even more severe for larger numbers of hidden units. The weight decay value $\lambda = 0.1$ produces good results for all numbers of hidden units, and there does not appear to be overfitting as the number of units increase. Finally, Figure 11.8 shows the test error for a ten hidden unit network, varying the weight decay parameter over a wide range. The value 0.1 is approximately optimal.

In summary, there are two free parameters to select: the weight decay λ and number of hidden units M . As a learning strategy, one could fix either parameter at the value corresponding to the least constrained model, to ensure that the model is rich enough, and use cross-validation to choose the other parameter. Here the least constrained values are zero weight decay and ten hidden units. Comparing the left panel of Figure 11.7 to Figure 11.8, we see that the test error is less sensitive to the value of the weight

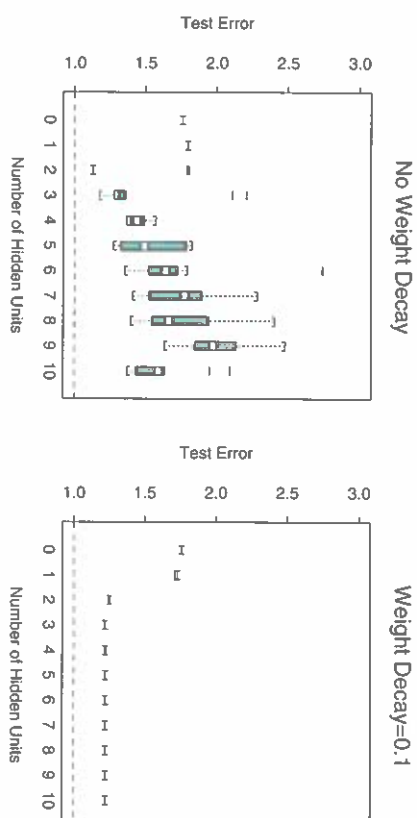


FIGURE 11.7. Boxplots of test error, for simulated data example, relative to the Bayes error. True function is a sum of two sigmoids. The test error is displayed for ten different starting weights, for a single hidden layer neural network with the number units as indicated. The two panels represent no weight decay (left) and strong weight decay $\lambda = 0.1$ (right).

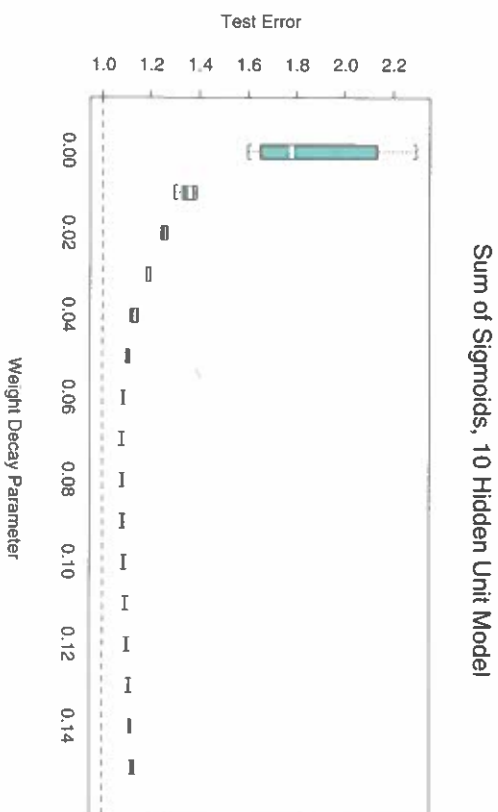


FIGURE 11.8. Boxplots of test error, for simulated data example. True function is a sum of two sigmoids. The test error is displayed for ten different starting weights, for a single hidden layer neural network with ten hidden units and weight decay parameter value as indicated.

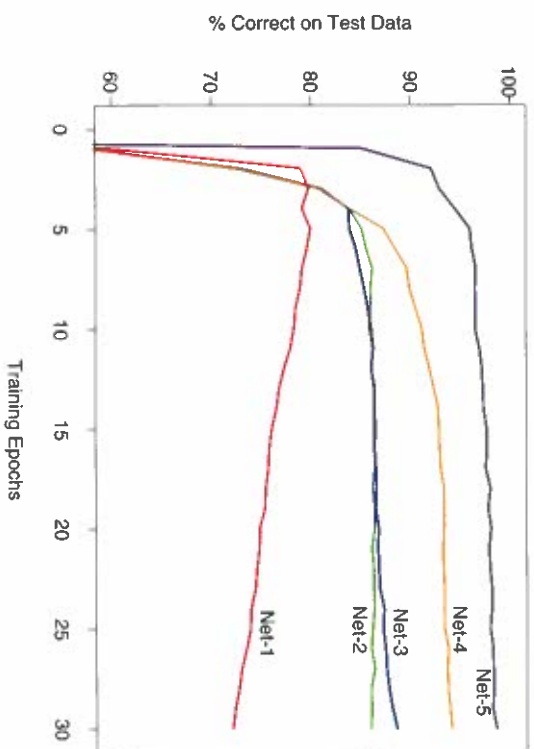


FIGURE 11.11. Test performance curves, as a function of the number of training epochs, for the five networks of Table 11.1 applied to the ZIP code data. (Le Cun, 1989)

The networks all have sigmoidal output units, and were all fit with the sum-of-squares error function. The first network has no hidden layer, and hence is nearly equivalent to a linear multinomial regression model (Exercise 11.4). Net-2 is a single hidden layer network with 12 hidden units, of the kind described above.

The training set error for all of the networks was 0%, since in all cases there are more parameters than training observations. The evolution of the test error during the training epochs is shown in Figure 11.11. The linear network (Net-1) starts to overfit fairly quickly, while test performance of the others level off at successively superior values.

The other three networks have additional features which demonstrate the power and flexibility of the neural network paradigm. They introduce constraints on the network, natural for the problem at hand, which allow for more complex connectivity but fewer parameters.

Net-3 uses local connectivity: this means that each hidden unit is connected to only a small patch of units in the layer below. In the first hidden layer (an 8×8 array), each unit takes inputs from a 3×3 patch of the input layer; for units in the first hidden layer that are one unit apart, their receptive fields overlap by one row or column, and hence are two pixels apart. In the second hidden layer, inputs are from a 5×5 patch, and again units that are one unit apart have receptive fields that are two units apart. The weights for all other connections are set to zero. Local connectivity makes each unit responsible for extracting local features from the layer below, and

TABLE 11.1. Test set performance of five different neural networks on a handwritten digit classification example (Le Cun, 1989).

Network Architecture	Links	Weights	% Correct
Net-1: Single layer network	2570	2570	80.0%
Net-2: Two layer network	3214	3214	87.0%
Net-3: Locally connected	1226	1226	88.5%
Net-4: Constrained network 1	2266	1132	94.0%
Net-5: Constrained network 2	5194	1060	98.4%

reduces considerably the total number of weights. With many more hidden units than Net-2, Net-3 has fewer links and hence weights (1226 vs. 3214), and achieves similar performance.

Net-4 and Net-5 have local connectivity with shared weights. All units in a local feature map perform the same operation on different parts of the image, achieved by sharing the same weights. The first hidden layer of Net-4 has two 8×8 arrays, and each unit takes input from a 3×3 patch just like in Net-3. However, each of the units in a single 8×8 feature map share the same set of nine weights (but have their own bias parameter). This forces the extracted features in different parts of the image to be computed by the same linear functional, and consequently these networks are sometimes known as *convolutional networks*. The second hidden layer of Net-4 has no weight sharing, and is the same as in Net-3. The gradient of the error function R with respect to a shared weight is the sum of the gradients of R with respect to each connection controlled by the weights in question.

Table 11.1 gives the number of links, the number of weights and the optimal test performance for each of the networks. We see that Net-4 has more links but fewer weights than Net-3, and superior test performance. Net-5 has four 4×4 feature maps in the second hidden layer, each unit connected to a 5×5 local patch in the layer below. Weights are shared in each of these feature maps. We see that Net-5 does the best, having errors of only 1.6%, compared to 13% for the “vanilla” network Net-2. The clever design of network Net-5, motivated by the fact that features of handwriting style should appear in more than one part of a digit, was the result of many person years of experimentation. This and similar networks gave better performance on ZIP code problems than any other learning method at that time (early 1990s). This example also shows that neural networks are not a fully automatic tool, as they are sometimes advertised. As with all statistical models, subject matter knowledge can and should be used to improve their performance.

This network was later outperformed by the tangent distance approach (Simard et al., 1993) described in Section 13.3.3, which explicitly incorporates natural affine invariances. At this point the digit recognition datasets become test beds for every new learning procedure, and researchers worked

hard to drive down the error rates. As of this writing, the best error rates on a large database (60,000 training, 10,000 test observations), derived from standard NIST² databases, were reported to be the following: (Le Cun et al., 1998):

- 1.1% for tangent distance with a 1-nearest neighbor classifier (Section 13.3.3);
- 0.8% for a degree-9 polynomial SVM (Section 12.3);
- 0.8% for *LeNet-5*, a more complex version of the convolutional network described here;
- 0.7% for boosted *LeNet-4*. Boosting is described in Chapter 8. *LeNet-4* is a predecessor of *LeNet-5*.

Le Cun et al. (1998) report a much larger table of performance results, and it is evident that many groups have been working very hard to bring these test error rates down. They report a standard error of 0.1% on the error estimates, which is based on a binomial average with $N = 10,000$ and $p \approx 0.01$. This implies that error rates within 0.1–0.2% of one another are statistically equivalent. Realistically the standard error is even higher, since the test data has been implicitly used in the tuning of the various procedures.

11.8 Discussion

Both projection pursuit regression and neural networks take nonlinear functions of linear combinations (“derived features”) of the inputs. This is a powerful and very general approach for regression and classification, and has been shown to compete well with the best learning methods on many problems.

These tools are especially effective in problems with a high signal-to-noise ratio and settings where prediction without interpretation is the goal. They are less effective for problems where the goal is to describe the physical process that generated the data and the roles of individual inputs. Each input enters into the model in many places, in a nonlinear fashion. Some authors (Hinton, 1989) plot a diagram of the estimated weights into each hidden unit, to try to understand the feature that each unit is extracting. This is limited however by the lack of identifiability of the parameter vectors α_m , $m = 1, \dots, M$. Often there are solutions with α_m spanning the same linear space as the ones found during training, giving predicted values that

are roughly the same. Some authors suggest carrying out a principal component analysis of these weights, to try to find an interpretable solution. In general, the difficulty of interpreting these models has limited their use in fields like medicine, where interpretation of the model is very important.

There has been a great deal of research on the training of neural networks. Unlike methods like CART and MARS, neural networks are smooth functions of real-valued parameters. This facilitates the development of Bayesian inference for these models. The next sections discuss a successful Bayesian implementation of neural networks.

11.9 Bayesian Neural Nets and the NIPS 2003 Challenge

A classification competition was held in 2003, in which five labeled training datasets were provided to participants. It was organized for a Neural Information Processing Systems (NIPS) workshop. Each of the data sets constituted a two-class classification problem, with different sizes and from a variety of domains (see Table 11.2). Feature measurements for a validation dataset were also available.

Participants developed and applied statistical learning procedures to make predictions on the datasets, and could submit predictions to a website on the validation set for a period of 12 weeks. With this feedback, participants were then asked to submit predictions for a separate test set and they received their results. Finally, the class labels for the validation set were released and participants had one week to train their algorithms on the combined training and validation sets, and submit their final predictions to the competition website. A total of 75 groups participated, with 20 and 16 eventually making submissions on the validation and test sets, respectively.

There was an emphasis on feature extraction in the competition. Artificial “probes” were added to the data: these are noise features with distributions resembling the real features but independent of the class labels. The percentage of probes that were added to each dataset, relative to the total set of features, is shown on Table 11.2. Thus each learning algorithm had to figure out a way of identifying the probes and downweighting or eliminating them.

A number of metrics were used to evaluate the entries, including the percentage correct on the test set, the area under the ROC curve, and a combined score that compared each pair of classifiers head-to-head. The results of the competition are very interesting and are detailed in Guyon et al. (2006). The most notable result: the entries of Neal and Zhang (2006) were the clear overall winners. In the final competition they finished first

²The National Institute of Standards and Technology maintain large databases, including handwritten character databases; <http://www.nist.gov/srd/>.