CSCI 4588/5588: Machine Learning II

Chapter #04: Artificial Neural Networks

(Reference for Logistic Regression: Chapter 4 of [1])

(References for ANN: Chapter 11 of [1], Chapter 5 of [2], Chapter 6 of [3])

Objectives: Here we will study the fundamentals of Artificial Neural Network (ANN).

Here, we discuss the classification problem and focus on linear methods for classification. Since our predictor $G(x)$ takes values in a discrete set G, we can always divide the input space into a collection of regions labeled according to the classification. The boundaries of these regions can be rough or smooth, depending on the prediction function. For an essential class of procedures, these *decision boundaries* are linear; this is what we will mean by linear methods for classification. Next, we will particularly focus on ANN which is used for classification.

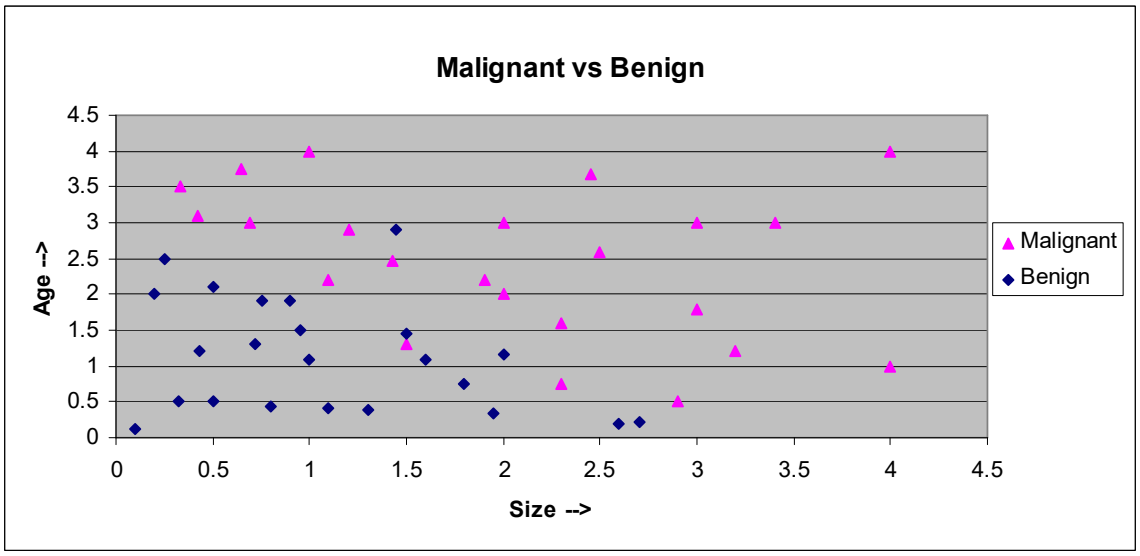


Figure 0: The (hypothetical) dataset is presenting the *Size* and *Age* of tumors, and the color label is indicating either it is 'Benign' or 'Malignant'.

Now, let us assume we are trying to model our cancer detection example (see Fig. 0, see exercise for the corresponding dataset):

$$\text{Tumor} \in \{\text{Benign}, \text{Malignant}\}$$

More precisely, we are trying to model:

$$P(G \mid X) \qquad (i)$$

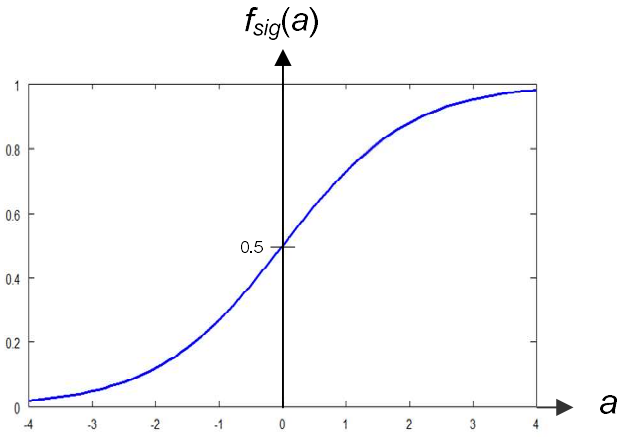
where $X = \{x_1, x_2\}$ and $\hat{G} \in \{$ Benign, Malignant$\}$, x_1= *Size of the tumor* and x_2 = *Age of the tumor*.

From our previous experience, we can express our linear equation as:

$$X_0\beta_0 + X_1\beta_1 + \cdots + X_p\beta_p = X^T\beta,$$

where we indicate the i^{th} row in **X** as: $x_i^T = (1, x_1, \cdots, x_n)$.

To convert the output of a linear equation into probability, we can plug it (i.e., $X^T\beta$) into a sigmoid function (See Figure A below for sigmoid function). The logistic function and the sigmoid functions are synonymous and hence it is called the logistic regression (i.e., logistic classification).



Sigmoid function,

$$f_{sig}(a) = \frac{1}{1+e^{-a}}$$

Octave code:

```
a=-4.0:0.1:4.0;
y=(1./(1+exp(-a)));
plot(a, y, 'LineWidth', 3);
```

Figure A: Plot of the sigmoid function.

Plugging in the sigmoid function to the Equation (*i*), we can write:

$$P(G \mid X) = f_{sig}(X^T\beta) \qquad (ii)$$

We can establish the classification rule based on Equation (ii) as:

$$\hat{G} = 1, \text{ if } [f_{sig}(X^T\beta) \geq 0.5], \text{ that is } [(X^T\beta) \geq 0],$$
$$\hat{G} = 2, \text{ if } [f_{sig}(X^T\beta) < 0.5], \text{ that is } [(X^T\beta) < 0].$$

We extend this simple idea using artificial neural network (ANN).

Artificial Neural Network **(ANN):** Here, we describe a class of learning methods that were developed separately in different fields—statistics and artificial intelligence—based on essentially identical models. The central idea is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features. The result is a powerful learning method, with widespread applications in many fields.

We have learned that the linear combination of the basis functions is effective and has useful analytic and computational properties. However, their practical applicability was limited by the curse of dimensionality. In order to apply such models to large-scale problems, it is necessary to adapt the basis functions to the data.

One way to implement such an approach is to fix the number of basis functions in advance and allow them to be adaptive. Basically, to use parametric forms of the basis functions in which the parameter values are adapted during training. The most successful model of this type in the context of pattern recognition is the *feed-forward neural network* which is comprised of multi-layers of logistic regression models.

The term 'neural network' has its origins in attempts to find mathematical representations of information processing in biological systems (McCulloch and Pitts1943; Widrow and Hoff, 1960; Rosenblatt, 1962; Rumelhart et al., 1986). Indeed, it has been used very broadly to cover a wide range of different models, many of which have been the subject of exaggerated claims regarding their biological plausibility. From the perspective of practical applications of pattern recognition, however, biological realism would impose entirely unnecessary constraints. Our focus in this chapter is therefore on neural networks as efficient models for statistical pattern recognition. In particular, we shall restrict our attention to the specific claim of neural networks that have proven to be of greatest practical value.

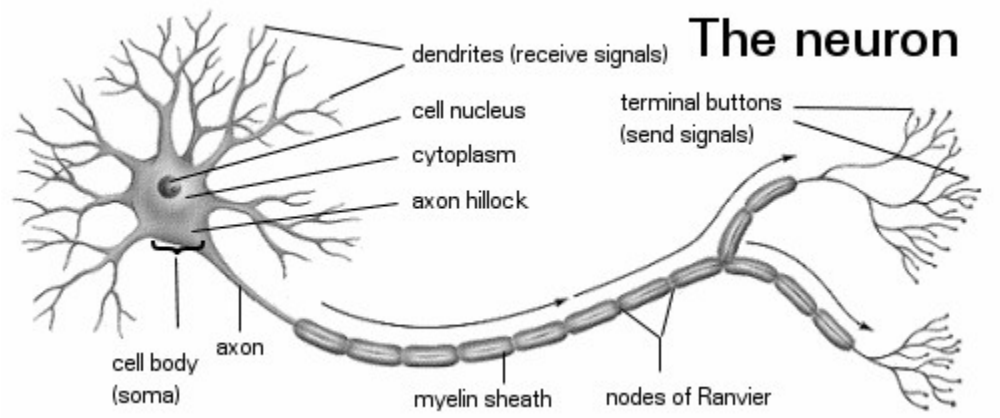


Figure A[*]: Structure of a brain cell, the neuron. The neuron has an excitable membrane that allows it to generate or propagate electrical signals, a tree of dendrites that receive signals, and an axon that transmits signals. The axon is a cable-like fiber that transmits nerve impulses from the neuron to other neurons or muscles or glands. A layer of fatty cells, the myelin sheath punctuated by the unsheathed nodes of Ranvier, insulates the axons of some neurons and speeds the impulses. Each neuron has only one axon which usually branches out extensively and passes signals to multiple target cells. Terminal buttons at the end of each axon branch connect the neuron to the receiver cells via ***synapses***. The synapse provides a functional connection between different cells. It consists of the target area, which may be a spine, a dendrite, or a cell body.

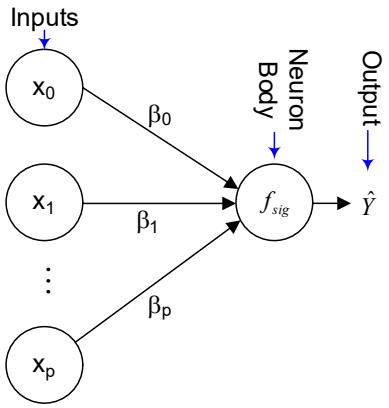
11.3 **Neural Networks**

[*] Ref: http://www.thebigview.com/mind/neuron.html

The term neural network has evolved to encompass a large class of models and learning methods. Here we describe the most widely used feed-forward neural net with hidden layer(s) and back-propagation is being applied.

The formulation of the (artificial) neural network was inspired by the idea of mimicking the brain, which is thought to be a network of billions of unit brain cells, called neuron (See Figure A*).

We can model a single neuron as a logistic unit (See Figure B).



$$\hat{Y} = f_{sig}(X_0\beta_0 + X_1\beta_1 + \cdots + X_p\beta_p) = f_{sig}(X^T\beta)$$

Figure B: Modeling a single neuron as a logistic unit.

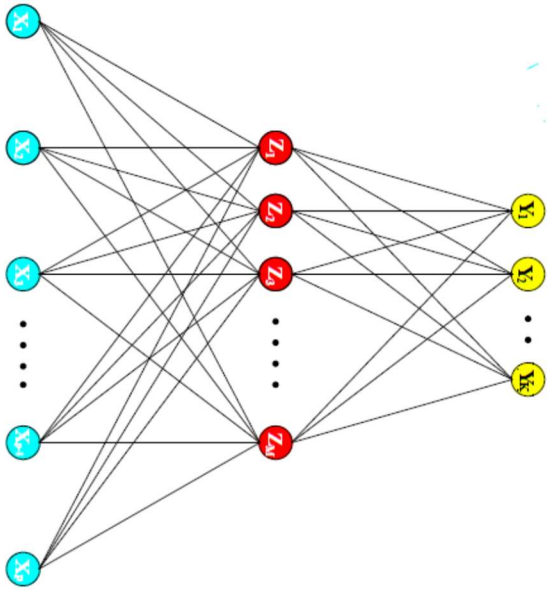
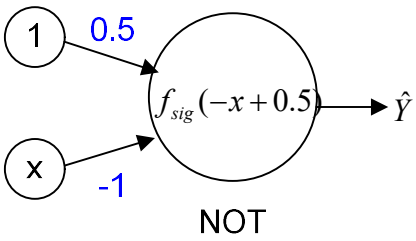


Figure 11.2. Schematic of the single hidden layer, feed-forward neural network.

A neural network, with hidden layer(s) is more than one-stage of regression or classification model, typically represented by a network diagram as in Figure 11.2.

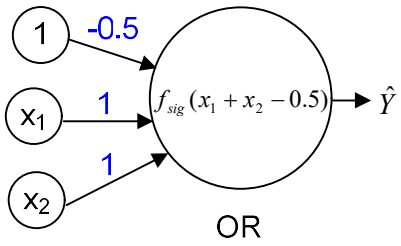
Let us first develop the intuition to see when we will need the hidden layers. We will use simple models to model popular logic functions such as AND, OR, NOT, XOR, etc and enhance our thoughts from there (See figures below).



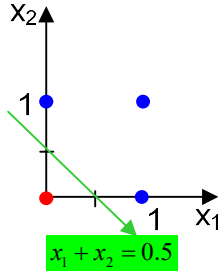
x	$f_{sig}(-x+0.5)$	$\hat{Y}$
0	f_{sig} (+0.5)	1
1	f_{sig} (-0.5)	0

(a) (b)

Figure C: Modeling (a) NOT and the corresponding (b) truth-table.

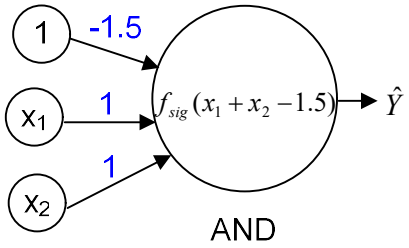


x_1	x_2	$f_{sig}(x_1+x_2-0.5)$	$\hat{Y}$
0	0	f_{sig} (-0.5)	**0**
0	1	f_{sig} (+0.5)	**1**
1	0	f_{sig} (+0.5)	**1**
1	1	f_{sig} (+1.5)	**1**

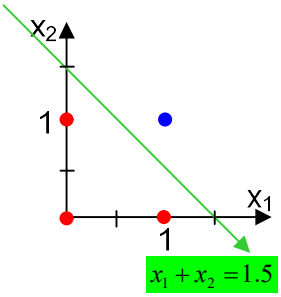


(a) (b) (c)

Figure D: Modeling (a) OR and the corresponding (b) truth-table. (c) Outputs are linearly separable.



x_1	x_2	$f_{sig}(x_1+x_2-1.5)$	$\hat{Y}$
0	0	f_{sig} (-1.5)	**0**
0	1	f_{sig} (-0.5)	**0**
1	0	f_{sig} (-0.5)	**0**
1	1	f_{sig} (+0.5)	**1**



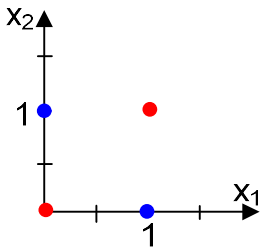
(a) (b) (c)

Figure E: Modeling (a) AND and the corresponding (b) truth-table. (c) Outputs are linearly separable.

Let us now examine the truth table and the plot of the truth-table of the XOR function (see Figure E).

x_1	x_2	$\hat{Y}$
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

(a)



(b)

x_1	x_2	$(\bar{x}_1 x_2 + x_1 \bar{x}_2)$	$\hat{Y}$
0	0	0 + 0	**0**
0	1	1 + 0	**1**
1	0	0 + 1	**1**
1	1	0 + 0	**0**

(c)

Figure F: (a) Truth-table for XOR, (b) plot of the XOR truth-table to check whether the output groups {0, 1} are linearly separable or not. (c) Formulation of XOR function using NOT, AND and OR comparable to basis expansion.

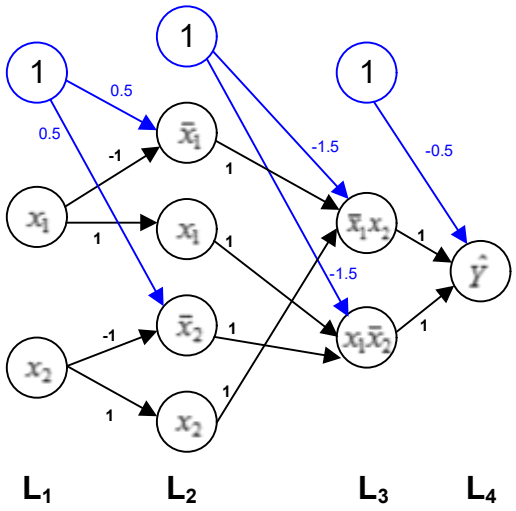


Figure G: Following the Table in Figure F(c), the implementation of the XOR function has been shown.

It is evident from Figure **G** that to solve non-linearity or, to develop XOR function, for example, we need hidden layer(s) (L_2 and L_3 in Figure **G**). Therefore, it is easy to surmise that we need hidden layer whenever the outputs are not linearly separable.

The network diagram, shown in Figure 11.2, 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 layer, with the k^{th} unit modeling the probability of class k. There are K target measurements Y_k, $k = 1, \ldots, K$, each being coded as a $0 - 1$ variable for the k^{th} class.

Let us establish the notational convention first, as demonstrated in Figure **H** to be followed in this chapter:

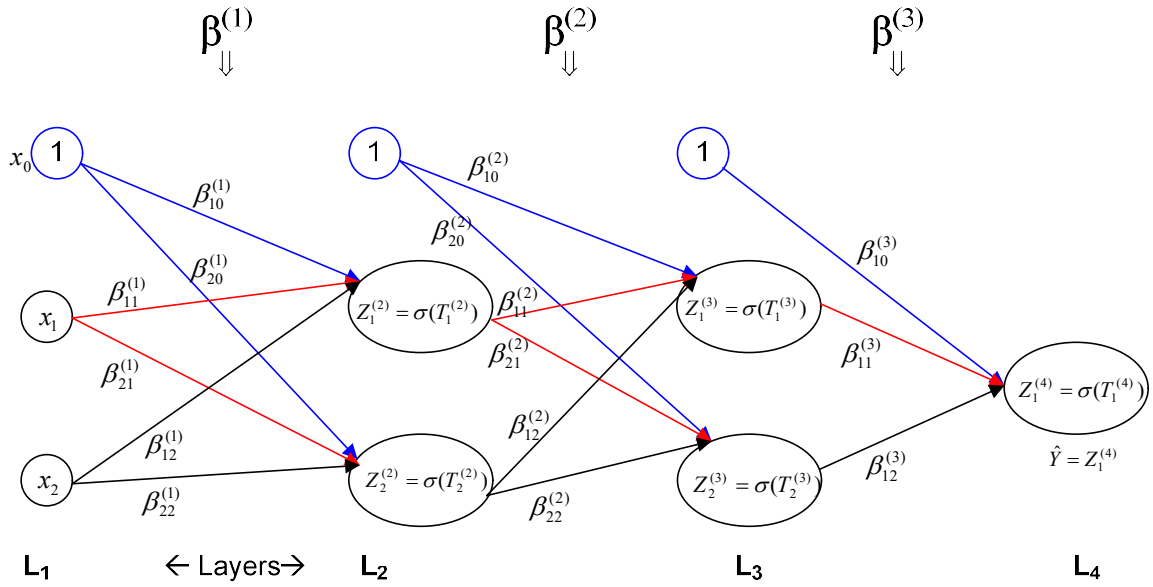


Figure H: A multilayer neural network demonstrating the notational conventions.

To indicate layers, we will use the layer number as superscript within the bracket. The *term* formed due to linear combination is indicated by T, and the result of the activation of the term T is denoted by Z. For example, at layer 2 (i.e., L_2), for the top node, we write the relationship of (T, Z) as $Z_1^{(2)} = \sigma(T_1^{(2)})$ in general. In case we use the sigmoid function, we can also write $Z_1^{(2)} = f_{sig}(T_1^{(2)})$. The linear terms that $T_1^{(2)}$ indicates with respect to Figure **H** is:

$$T_1^{(2)} = X_0\beta_{10}^{(1)} + X_1\beta_{11}^{(1)} + X_2\beta_{12}^{(1)}$$

The layer number of the corresponding β s are indicated by $\beta^{(i)}$ where the β is in between layer i and (i+1). We use, two subscripts for β as $\beta_{kl}^{(i)}$ where the β represent the weights in between k^{th} node at layer (i+1) and l^{th} node at layer i.

Basically, derived features $Z_m^{(l)}$ at layer l are created from linear combinations of the inputs, and then the target Y_k is modeled as a function of linear combinations of the $Z_m^{(l)}$.

$$\begin{aligned} &Z_m^{(l)} = \sigma(\beta_{m0}^{(l-1)} + X^T\beta_{mp}^{(l-1)}),\ m = 1,...,M, \text{ and } p = \text{number of inputs for } Z_m^{(i)\ th} \text{ node.} \\ &T_k^{(l)} = \beta_{k0}^{(l)} + Z^T\beta_{kp}^{(l)}, k = 1,...,K, \\ &f_k(X) = g_k(T),\ k = 1,...,K, \end{aligned} \qquad (11.5)$$

For example, at layer #3, the top node (i.e., #1 or, m=1) can be expressed in general as:

$$Z_1^{(3)} = \sigma(Z_0^{(2)}\beta_{10}^{(2)} + Z_1^{(2)}\beta_{11}^{(2)} + Z_2^{(2)}\beta_{12}^{(2)} + ... + Z_p^{(2)}\beta_{1p}^{(2)})$$

The activation function σ(v) is usually chosen to be the sigmoid σ(v) = 1/(1+e^{-v}), which we express as $f_{sig}(v)$; see Figure 11.3 for a plot of 1/(1+e^{-v}). Sometimes Gaussian radial basis functions (Chapter 6, Book [1]) are used for the σ(v), producing what is known as a radial basis function network.

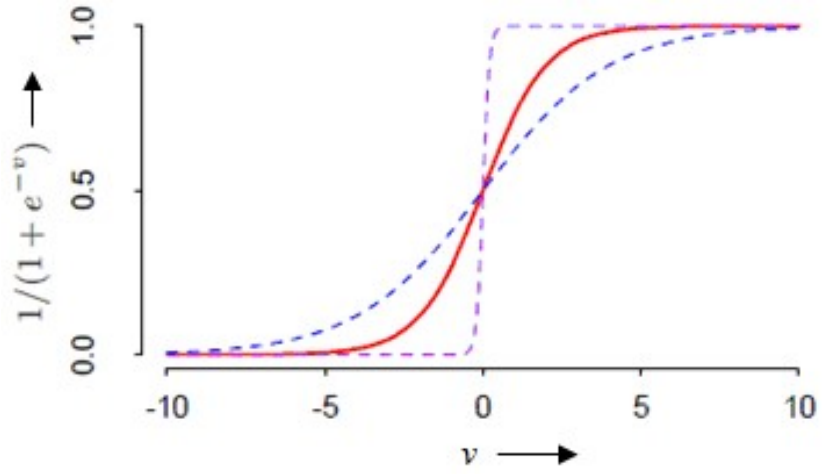


Figure 11.3: Plot of the sigmoid function $\sigma(v) = 1/(1+\exp(-v))$ (red curve), commonly used in the hidden layer of a neural network. Included are σ(sv) for $s = 1/2$ (blue curve) and $s = 10$ (purple curve). The scale parameter s controls the activation rate, and we can see that large s amounts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

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. Thinking of the constant "1" as an additional input feature, this bias unit captures the intercepts $\beta_{m0}^{(l)}$ and $\beta_{k0}^{(l)}$ 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 $g_k(T) = T_k$. For K-class classification:

$$g_k(T) = \frac{e^{T_k}}{\sum_{i=1}^{K} e^{T_i}} \tag{11.6}$$

This is of course exactly the transformation used in the *multi-logit* model (Section 4.4, Book [1]), and produces positive estimates that sum to one. In Section 4.2 (Book [1]) 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. 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 multi-logit model, using these transformations as inputs. There is, however, an important enhancement over

the basis expansion techniques discussed in Chapter 5 (Book [1]); here the parameters of the basis functions are learned from the data.

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.

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 the use of a step function for σ(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 (i.e., the β s) 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 $\forall\ \beta$, i.e,

$$\theta = \forall\ \beta \tag{11.8}$$

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

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

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

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

and the corresponding classifier is

$$G(x) = \arg\max_k f_k(x)$$

With the softmax activation function (11.6) and the cross-entropy error function (11.10), the neural network model is precisely 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 we develop the back-propagation in detail for squared error loss.

Let us define the output error,

$$E(\beta) = \frac{1}{2}\sum_{k=1}^{K}(Y_k - Z_k^{(L)})^2 \qquad [= \frac{1}{2}R(\beta)\,]$$

Here, layer l =1, 2, …,L and L = Output Layer. And, Y_k is the k^{th} target/original output and $Z_k^{(L)}$ is the k^{th} predicted/network output.

The back-propagation learning rule is based on gradient descent. This is because we want to minimize the error by changing the weights. The weights are initialized with random values, and then they are changed in a direction that will reduce the error:

$$\Delta\beta = -\alpha\frac{\partial E}{\partial \beta} \quad [\textbf{Recall}: \beta(t+1) = \beta(t) - \alpha\frac{\partial E}{\partial \beta}\text{, or,}$$

$$\Delta\beta = [\beta(t+1) - \beta(t)] = -\alpha\frac{\partial E}{\partial \beta} \qquad \textbf{(i)}$$

and α is the learning rate]

The overall idea of the error back-propagation will be to compute the discrepancy of the targeted output versus the predicted output from output node(s) first and then to propagate backward to compute the discrepancy for each of the nodes in the hidden layer except the input layer – because we do not want to change the input layer. Then we will update the weight or the parameter β for the network to minimize the error.

First, we will compute the rate of changes of error with respect to the weight connected to the output node(s):

$$\frac{\partial E}{\partial \beta^{(L-1)}} = \frac{\partial}{\partial \beta^{(L-1)}}\left(\frac{1}{2}\sum_{k=1}^{K}(Y_k - Z_k^{(L)})^2\right)$$

$$= (Y_k - Z_k^{(L)})\frac{\partial}{\partial \beta^{(L-1)}}\left(-Z_k^{(L)}\right)$$ [1. Removing Σ because all $(L\text{-}1)^{\text{th}}$ weights may not be connected to all the output nodes.
2. Expression expanded using ***chain rule***]

$$= -(Y_k - Z_k^{(L)})\frac{\partial}{\partial \beta^{(L-1)}}\left(\sigma(T_k^{(L)})\right) \qquad [\because Z_k^{(L)} = \sigma(T_k{}^{(L)})\,]$$

$= (Z_k^{(L)} - Y_k)\frac{\partial}{\partial T_k^{(L)}}\left(\sigma(T_k^{(L)})\right)\times\frac{\partial(T_k^{(L)})}{\partial\beta^{(L-1)}}$ [using chain rule]

[Note: ----------

$$Z = \sigma(T) = f_{sig}(T) = \frac{1}{1+e^{-T}}$$

$$Z' = \frac{d(\sigma(T))}{dT} = -\frac{-e^{-T}}{(1+e^{-T})^2} = \frac{e^{-T}}{(1+e^{-T})}\cdot\frac{1}{(1+e^{-T})}$$

$$Z' = (1-Z)Z$$

$$= (1-\sigma(T))(\sigma(T))$$

Therefore, $\frac{\partial}{\partial T^{(L)}}\left(\sigma(T^{(L)})\right) = Z^{(L)'} = (1-Z^{(L)})Z^{(L)}$ ***(ii)***

----------]

$= (Z_k^{(L)} - Y_k)\, Z_k^{(L)'} \times\frac{\partial(T_k^{(L)})}{\partial\beta^{(L-1)}}$

$= (Z_k^{(L)} - Y_k)\, Z_k^{(L)'} \times\frac{\partial}{\partial\beta^{(L-1)}}\left(\beta^{(L-1)^T} Z^{(L-1)}\right)$ $[\because T_k^{(L)} = (\beta^{(L-1)})^T Z^{(L-1)}]$

$= (Z_k^{(L)} - Y_k)\, Z_k^{(L)'} Z^{(L-1)}$ [Assume, error term for the output layer:

$\delta^{(L)} = (Z_k^{(L)} - Y_k)\, Z_k^{(L)'}$

Or, $\delta^{(L)} = (Z_k^{(L)} - Y_k)\, Z_k^{(L)}(1 - Z_k^{(L)})$

]

$= \delta^{(L)} Z^{(L-1)}$

Finally, for output layer we have,

$$\frac{\partial E}{\partial\beta^{(L-1)}} = \delta^{(L)} Z^{(L-1)}, \qquad \textbf{(iii)}$$

where $\delta^{(L)} = (Z_k^{(L)} - Y_k)\, Z_k^{(L)}(1 - Z_k^{(L)})$

Let us now compute the rate of changes of error with respect to the weight for the hidden layer(s) similarly:

$$\frac{\partial E}{\partial\beta^{(L-2)}} = \frac{\partial}{\partial\beta^{(L-2)}}\left(\frac{1}{2}\sum_{k=1}^{K}(Y_k - Z_k^{(L)})^2\right)$$

$$= \sum_{k=1}^{K}(Y_k - Z_k^{(L)})\times\frac{\partial}{\partial\beta^{(L-2)}}\left(-Z_k^{(L)}\right)$$

$$= \sum_{k=1}^{K}(Z_k^{(L)} - Y_k)\times\frac{\partial}{\partial\beta^{(L-2)}}\left(\sigma(T_k^{(L)})\right)$$

$$= \sum_{k=1}^{K}(Z_k^{(L)} - Y_k) \times \frac{\partial}{\partial T_k^{(L)}}\left(\sigma(T_k^{(L)})\right) \times \frac{\partial(T_k^{(L)})}{\partial \beta^{(L-2)}} \qquad \text{[using chain rule]}$$

$$= \sum_{k=1}^{K}(Z_k^{(L)} - Y_k) \times Z_k^{(L)'} \times \frac{\partial(T_k^{(L)})}{\partial \beta^{(L-2)}}$$

$$= \sum_{k=1}^{K}(Z_k^{(L)} - Y_k) \times Z_k^{(L)'} \times \frac{\partial(T_k^{(L)})}{\partial Z^{(L-1)}} \times \frac{\partial Z^{(L-1)}}{\partial \beta^{(L-2)}} \qquad \text{[using chain rule]}$$

$$= \sum_{k=1}^{K}(Z_k^{(L)} - Y_k) \times Z_k^{(L)'} \times \beta^{(L-1)} \times \frac{\partial Z^{(L-1)}}{\partial \beta^{(L-2)}} \qquad [\because T_k^{(L)} = (\beta^{(L-1)})^T Z^{(L-1)},$$

$$\therefore \frac{\partial(T_k^{(L)})}{\partial Z^{(L-1)}} = \beta^{(L-1)}]$$

$$= \frac{\partial Z^{(L-1)}}{\partial \beta^{(L-2)}} \times \sum_{k=1}^{K}(Z_k^{(L)} - Y_k) \times Z_k^{(L)'} \times \beta^{(L-1)}$$

$$= \frac{\partial Z^{(L-1)}}{\partial \beta^{(L-2)}} \times \sum_{k=1}^{K}\delta^{(L)} \times \beta^{(L-1)} \qquad [\therefore \delta^{(L)} = (Z_k^{(L)} - Y_k) \times Z_k^{(L)'}]$$

[Note:----------

$$\frac{\partial Z^{(L-1)}}{\partial \beta^{(L-2)}} = \frac{\partial}{\partial \beta^{(L-2)}}\left(\sigma(T^{(L-1)})\right) = \frac{\partial(\sigma(T^{(L-1)})}{\partial T^{(L-1)}} \times \frac{\partial T^{(L-1)}}{\partial \beta^{(L-2)}}$$

[As we have seen before in (*ii*): $Z^{(L-1)'} = \frac{\partial(\sigma(T^{(L-1)})}{\partial T^{(L-1)}} = Z^{(L-1)}(1 - Z^{(L-1)})$]

$$= Z^{(L-1)}(1 - Z^{(L-1)}) \times \frac{\partial T^{(L-1)}}{\partial \beta^{(L-2)}}$$

$$= Z^{(L-1)}(1 - Z^{(L-1)}) \times Z^{(L-2)} \qquad [\because \frac{\partial T^{(L-1)}}{\partial \beta^{(L-2)}} = \frac{\partial}{\partial \beta^{(L-2)}}\left(\beta^{(L-2)^T} Z^{(L-2)}\right) = Z^{(L-2)}$$

----------]

$$= Z^{(L-1)}(1 - Z^{(L-1)}) \times Z^{(L-2)} \times \sum_{k=1}^{K}\delta^{(L)} \times \beta^{(L-1)}$$

$$= Z^{(L-2)} \times Z^{(L-1)}(1 - Z^{(L-1)}) \times \sum_{k=1}^{K}\delta^{(L)} \times \beta^{(L-1)}$$

Finally, for the <u>hidden layer</u> we have:

$$\frac{\partial E}{\partial \beta^{(L-2)}} = Z^{(L-2)} \times Z^{(L-1)}(1 - Z^{(L-1)}) \times \sum_{k=1}^{K}\delta^{(L)} \times \beta^{(L-1)}$$

Or, we can write it as:

$$\frac{\partial E}{\partial \beta^{(L-2)}} = \delta^{(L-1)} Z^{(L-2)} \qquad \textbf{(\textit{iv})}$$

where, $\delta^{(L-1)} = Z^{(L-1)}(1 - Z^{(L-1)}) \times \sum_{k=1}^{K} \delta^{(L)} \times \beta^{(L-1)}$

Involvements of the Bias term:
If we revisit figure **H**, we see the bias terms supply +1 to the node(s) of the next layer always. If we use Equation (*iii*) for the bias term at layer (L-1), we can rewrite equation (*iii*) as:

$$\frac{\partial E}{\partial \beta_0^{(L-1)}} = \delta^{(L)} Z^{(L-1)} = \delta^{(L)} \qquad \textbf{(v)}$$

$[\because here, Z^{(L-1)} = Z_0^{(L-1)} = 1]$

Similarly for the other hidden layers following equation (*iv*) and the idea of equation (*v*), we can write:

$$\frac{\partial E}{\partial \beta_0^{(L-2)}} = \delta^{(L-1)} Z_0^{(L-2)} = \delta^{(L-1)} \qquad \textbf{(vi)}$$

Now, we have generated the required equation to describe the back-propagation algorithm. We will explain the backpropagation algorithm next, and then we will illustrate the idea in further detail:

Algorithm 1: Error Backpropagation

BEGIN

1. From a data point (x_i, y_i), apply an input vector x_i to the network and forward propagate through the network (as we computed the logistic regression) as we described in (11.5).

2. For each of the output node/unit compute the error term δ^L:

$$\delta^{(L)} = (Z_k^{(L)} - Y_k)\, Z_k^{(L)}(1 - Z_k^{(L)})$$

3. For each of the hidden layer node /unit compute the error term $\delta^{(L-1)}$:

$$\delta^{(L-1)} = Z^{(L-1)}(1 - Z^{(L-1)}) \times \sum_{k=1}^{K} \delta^{(L)} \times \beta^{(L-1)}$$

   And Compute: $\delta^{(L-2)}, \delta^{(L-3)}, ..., \delta^{(2)}$, except $\delta^{(1)}$ because it is the input layer and we do not want to change the original input data.

4. Update the weights (β) of the network [See Equations: *i*, *iii*, *iv*, *v*, *vi*]

$$\beta^{(i)}(t+1) = \beta^{(i)}(t) - \alpha\, \delta^{(i+1)}\, Z^{(i)} \qquad \textbf{(11.13)}$$

$$\beta_0^{(i)}(t+1) = \beta_0^{(i)}(t) - \alpha\, \delta^{(i+1)} \qquad \text{[For bias terms]}$$

where, i=*1*, *2*, …, (*L*-1)

END

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 the squares error function.

The advantages of back-propagation are its simple, local nature. In the backpropagation 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.

Learning can 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 as it is in (11.13).

The updates in (11.13) can also be a kind of ***batch learning***, with the parameter updates being a sum over all of the training cases, i.e.,

$$\beta^{(i)}(t+1) = \beta^{(i)}(t) - \alpha \frac{1}{N} \sum_{N} \delta^{(i+1)} Z^{(i)}$$

$$\beta_0^{(i)}(t+1) = \beta_0^{(i)}(t) - \alpha \frac{1}{N} \sum_{N} \delta^{(i+1)}$$

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. In general, online training implies that a learning step is taken place at each presentation of a randomly drawn training pattern. Online training is especially useful when the training patterns are drawn from a time-dependent environmental distribution.

The learning rate α 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 α 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 $\alpha \rightarrow 0$, $\sum_r \alpha_r = \infty$, and $\sum_r \alpha_r^2 = < \infty$ (satisfied, for example, by $\alpha = 1/r$).

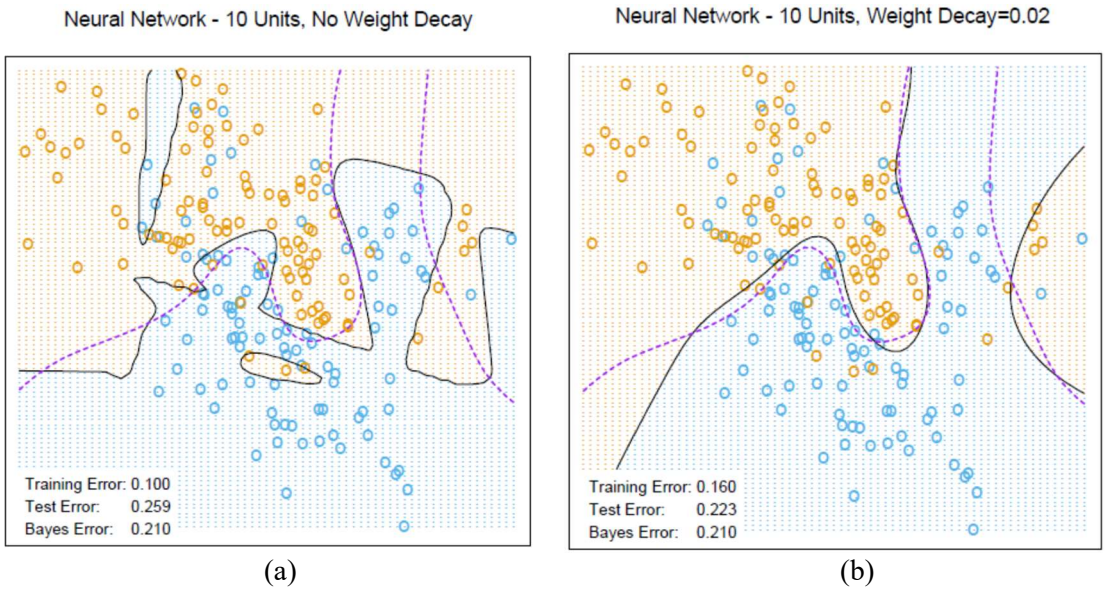
Back-propagation can be very slow, and for that reason is usually not the method of choice. Second-order techniques such as Newton's method are not attractive here, because the second derivative matrix of R (the Hessian) can be very large. Better approaches to fitting include conjugate gradients and 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 over-parametrized, 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. **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. **The 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.**



(a) (b)

Figure 11.4: A neural network on the mixture example of Chapter 2 (book [1]). Panel (a) uses no weight decay and overfits the training data. Panel (b) uses weight decay and achieves close to the Bayes error rate (broken purple boundary). Both use the softmax activation function and cross-entropy error.

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. We add a penalty to the error function:

$$E(\beta) = \frac{1}{2}\left(R(\beta) + \lambda \sum \beta^2\right) \qquad (11.16)$$

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 to the gradient expressions (11.13) (we often not penalize the bias equation, i.e., equation involving β_0). Other forms for the penalty have been proposed, for example,

$$E(\beta) = \frac{1}{2}\left(R(\beta) + \lambda \sum \frac{\beta^2}{1+\beta^2}\right) \qquad (11.17)$$

known as the ***weight elimination penalty***.

.Figure 11.4 shows the result of training a neural network with ten hidden units, without weight decay (#a) and with weight decay (#b), to the mixture example of Chapter 2 (Book [1]). Weight decay has clearly improved the prediction.

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 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. The 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. The use of multiple hidden layers allows the construction of hierarchical features at different levels of resolution. An example of the effective use of multiple layers is given in Section 11.6 (book [1]).

11.5.5 *Multiple Minima*

The error function R(θ) can be nonconvex, possessing many local minima. As a result, the final solution obtained is quite dependent on the choice of starting 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). It is preferable to average the weights since the nonlinearity of the model implies that this averaged solution could be quite poor. Another approach is via ***bagging*** (this is model averaging to avoid overfitting and reduce the variance …), which averages the predictions of network training from randomly perturbed versions of the training data. This is described in Section 8.7 (book [1]).

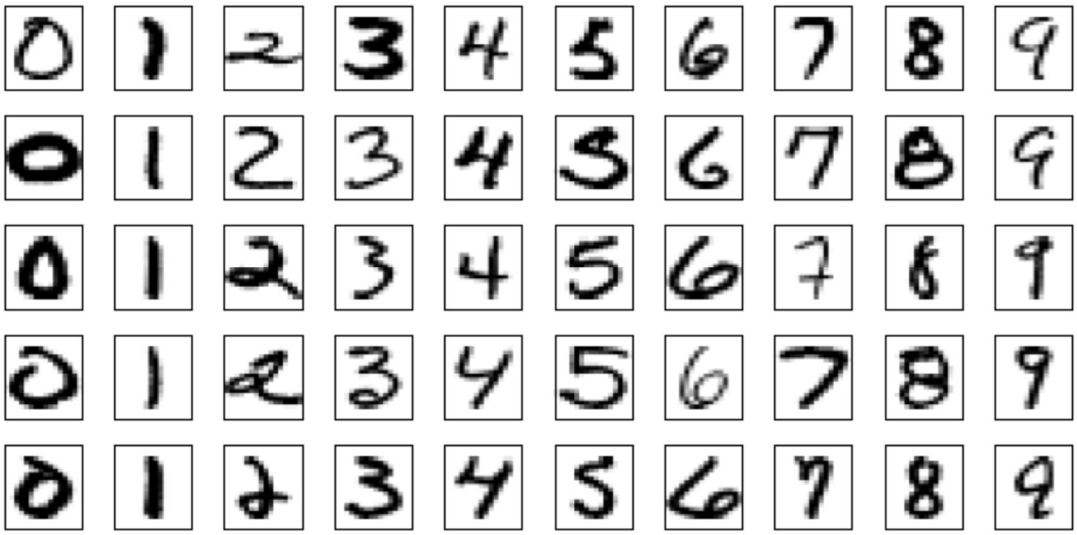


Figure 11.9. Examples of training cases from ZIP code data. Each image is a 16 × 16 8-bit grayscale representation of a handwritten digit.

11.7 **Example: ZIP Code Data**

This example is a character recognition task: classification of handwritten numerals. This problem captured the attention of the machine learning and neural network community for many years and has remained a benchmark problem in the field. Figure 11.9 shows some examples of normalized handwritten digits, automatically scanned from envelopes by the U.S. Postal Service. The original scanned digits are binary and of different sizes and orientations; the images shown here have been deslanted and size normalized, resulting in 16×16 grayscale images (Le Cun et al., 1990). These 256-pixel values are used as inputs to the neural network classifier.

A black box neural network is not ideally suited to this pattern recognition task, partly because the pixel representation of the images lack certain invariances (such as small rotations of the image). Consequently, early attempts with neural networks yielded misclassification rates around 4.5% on various examples of the problem. In this section, we show some of the pioneering efforts to handcraft the neural network to overcome some these deficiencies (Le Cun, 1989), which ultimately led to the state of the art in neural network performance (Le Cun et al., 1998).

Although current digit datasets have tens of thousands of training and test examples, the sample size here is deliberately modest in order to emphasize the effects. The examples were obtained by scanning some actual hand-drawn digits and then generating additional images by random horizontal shifts. Details may be found in Le Cun (1989). There are 320 digits in the training set, and 160 in the test set.

Five different networks were fit to the data:

Net-1: No hidden layer, equivalent to multinomial logistic regression.
Net-2: One hidden layer, 12 hidden units fully connected.
Net-3: Two hidden layers locally connected.

Net-4: Two hidden layers, locally connected with weight sharing.
Net-5: Two hidden layers, locally connected, two levels of weight sharing.

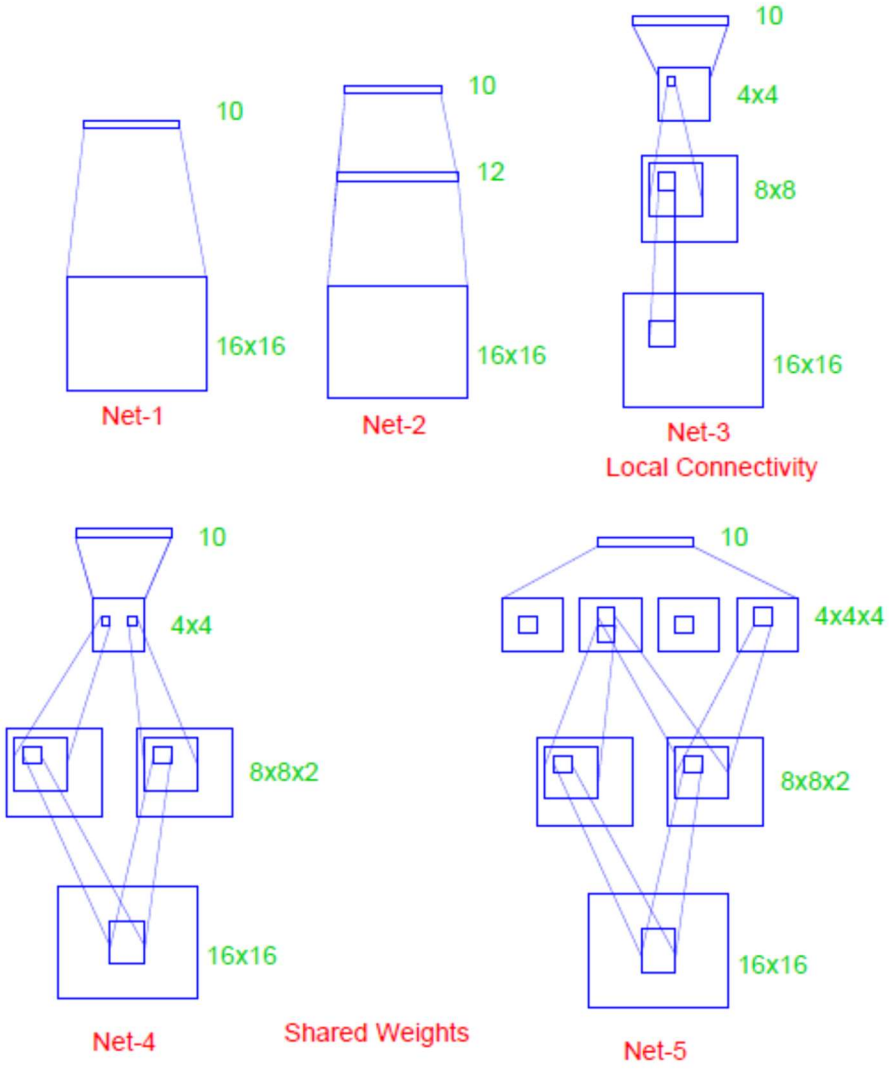


Figure 11.10. The architecture of the five networks used in the ZIP code example.

These are depicted in Figure 11.10. Net-1, for example, has 256 inputs, one each for the 16×16 input pixels, and ten output units for each of the digits 0–9. The predicted value $\hat{f}_k(x)$ represents the estimated probability that an image x has digit class k, for $k = 0, 1, 2, \ldots, 9$.

The networks all have sigmoidal output units and were all fitted 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. 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 the test performance of the other levels off at successively superior values.

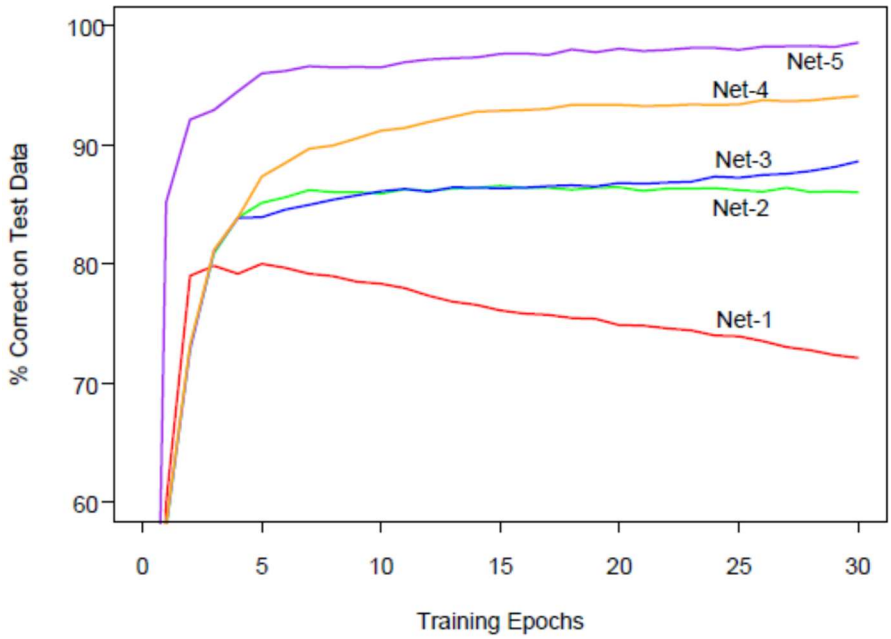


Figure 11.11. Test performance curves, as a function of the number of training epochs, for the five networks of Table 11.1 applied to the ZIP code data. (Le Cun, 1989).

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

TABLE 11.1. A 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%

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 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 shares the same set of nine weights (but have their 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 (i.e., NN with hidden layer) 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 (book [1]), which explicitly incorporates natural affine invariances. At this point, the digit recognition datasets become testbeds 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 NIST[2] 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, book [1]);

- 0.8% for a degree-9 polynomial SVM (Section 12.3, book [1]);

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

• 0.8% for LeNet-5, a more complex version of the convolutional network described here (book [1], ch-11);

• 0.7% for boosted LeNet-4. Boosting is described in Chapter 8 (book [1]). 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 **Discussions**

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 (β). Often there are solutions with β 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 the 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 (Classification And Regression Tree) and MARS (Multivariate Adaptive Regression Splines), neural networks are smooth functions of real-valued parameters. This facilitates the development of Bayesian inference for these models. The next section (see [1]) discusses a successful Bayesian implementation of neural networks.

References

1. Hastie, T., R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. 2009: Springer.
2. Bishop, C.M., *Pattern Recognition and Machine Learning*. 2009: Springer.
3. Duda, R.O., P.E. Hart, and D.G. Stork, *Pattern Classification*. 2000: Wiley.