47.23 NEURAL NETWORKS

We illustrated earlier in Example 47.55 one limitation of linear separation surfaces by considering the XOR mapping defined by (47.845). The example shows that certain feature spaces are not linearly separable. Consequently, it is not possible to learn the mapping (47.845) by means of the Perceptron algorithm, which is one of the techniques we employed to design classifiers for linearly separable data. The result in that example was used to motivate one powerful approach to nonlinear separation surfaces by means of kernel methods. In this section we describe a second powerful method, based on training neural networks, and which relies on cascading a collection of modified Perceptron units.

As we are going to see, a neural network will correspond to a layered structure of interconnected elementary elements called *neurons*; these neurons will be modified versions of the Perceptron structure we described before in Sec. 47.18. Specifically, the network will consist of:

(a) one input layer, where the feature data is applied to;

(b) one output layer, where the class labels are read from, and

(c) several hidden layers in between the input and output layers.

In one of the most common implementations for neural networks, information flows forward in the network from the input layer and into successive hidden layers until it reaches the output layer. This type of implementation is known as a *feedforward* structure. There are other implementations, known as *feedback* structures, where signals from later layers feed back into neurons in earlier layers. In this presentation we will focus on conveying the main concepts by considering the feedforward structure due to its widespread use.

47.23.1 Neurons

The basic unit in a neural network is the neuron shown in Fig. 47.68. It consists of a collection of multipliers, one adder, and a nonlinearity. The input to the first multiplier is fixed at $+1$ and its coefficient is denoted by $-\theta$. The coefficients for the remaining multipliers are denoted by $w(m)$ and their respective inputs by $h(m)$. These choices are motivated by the extended notation (47.248) introduced earlier.

If we collect the input and scaling coefficients into the column vector quantities

$$\begin{aligned} h &= \text{col}\{h(1), h(2), \ldots, h(M)\} && (47.952\text{a}) \\ w &= \text{col}\{w(1), w(2), \ldots, w(M)\} && (47.952\text{b}) \end{aligned}$$

then the output of the adder is given by the inner product

$$z = h^{\mathsf{T}} w - \theta \tag{47.953}$$

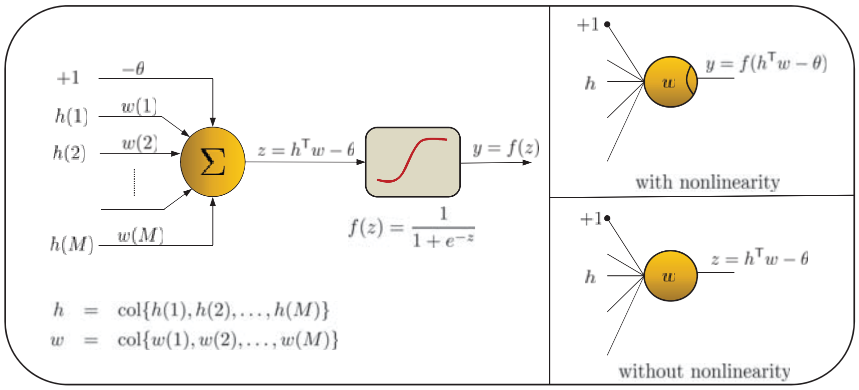
+1
$-\theta$
$h(1)$
$w(1)$
$h(2)$
$w(2)$
$h(M)$
$w(M)$
Σ
$z = h^{\mathsf{T}}w - \theta$
$y = f(z)$
$f(z) = \frac{1}{1+e^{-z}}$
$h = \text{col}\{h(1), h(2), \ldots, h(M)\}$
$w = \text{col}\{w(1), w(2), \ldots, w(M)\}$
+1
h
w
$y = f(h^{\mathsf{T}}w - \theta)$
with nonlinearity
+1
h
w
$z = h^{\mathsf{T}}w - \theta$
without nonlinearity

FIGURE 47.68 (*Left*) Structure of a neuron consisting of M multipliers with weights $\{w(m)\}$ and input signals $\{h(m)\}$, followed by an adder with output z and a nonlinearity $y = f(z)$. (*Right*) Compact representations for the neuron in terms of a circle with multiple input lines and one output line. The two representations are used to distinguish between the cases where a nontrivial nonlinearity is present or not (i.e., whether $f(z) = z$ or not). When a nonlinearity is present, we shall indicate its presence by an arc inside the circular representation, a shown in the top right corner.

where we are using the letter "z" to refer to the result of the addition. This signal is subsequently fed into a nonlinearity, called the activation function, to generate the output signal y:

$$y = f(z) \;=\; f(h^{\mathsf{T}}w - \theta) \tag{47.954}$$

There are several common choices for $f(z)$, listed in Table 47.11. We encountered the sigmoid function earlier in (47.693) while discussing the logistic regression problem. We also encountered the hyperbolic tangent function earlier in (34.43) while studying the optimal mean-square-error estimation problem. Both functions are related via the translation

$$\frac{1}{1+e^{-z}} = \frac{1}{2}\left(\tanh\left(\frac{z}{2}\right)+1\right) \tag{47.955}$$

We illustrated the behavior of both functions in Figs. 34.2 and 47.44, where it was seen that

$$\text{(sigmoid function)}: \quad \lim_{z\to+\infty} f(z) = 1, \quad \lim_{z\to-\infty} f(z) = 0 \tag{47.956a}$$

$$\text{(tanh function)}: \quad \lim_{z\to+\infty} f(z) = 1, \quad \lim_{z\to-\infty} f(z) = -1 \tag{47.956b}$$

In particular, observe from the figures that both of these choices saturate for large $|z|$. This means that when $|z|$ is large, their derivatives assume values that are close to zero. We will see later in Example 47.68 that this property is responsible for a slow down in the speed of learning by neural networks, especially for deep networks, since a small derivative value at any neuron will generally limit the learning ability of neurons in the preceding layer. In the rectifier (or hinge) function, on the other hand, nonnegative values of z remain unaltered, with a constant derivative value at one, while negative values of z are set to zero. In this case, adaptation will stop for negative z. The softplus function provides a smooth approximation for the rectified function, tending to zero gracefully as $z \to -\infty$ — see Fig. 47.69. The rectifier function is widely used in the training of deep feedforward and convolutional neural networks.

The scaled hyperbolic tangent function, $f(z) = a\tanh(bz)$, maps the real axis to the interval $[-a, a]$ and, therefore, it saturates at $\pm a$ for large values of z. A recommended choice for the parameters (a, b) is

$$b = \frac{2}{3}, \quad a = \frac{1}{\tanh(2/3)} \approx 1.7159 \tag{47.957}$$

With these values, one finds that $f(\pm 1) = \pm 1$. In other words, when the input value z approaches the typical binary values of ± 1 (which is typical in classification problems when the class variables are represented by ± 1), then the scaled hyperbolic tangent will assume the values ± 1, which are sufficiently away from its saturation levels of ± 1.7159.

TABLE 47.11 Typical choices for the activation function $f(z)$ used in (47.954).

activation function	$f(z)$
sigmoid or logistic	$f(z) = \dfrac{1}{1+e^{-z}}$
hyperbolic tangent (tanh)	$f(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$
rectifier	$f(z) = \max(0, z)$
softplus	$f(z) = \ln(1 + e^z)$
scaled tanh	$f(z) = a\tanh(bz), a, b > 0$
no activation	$f(z) = z$

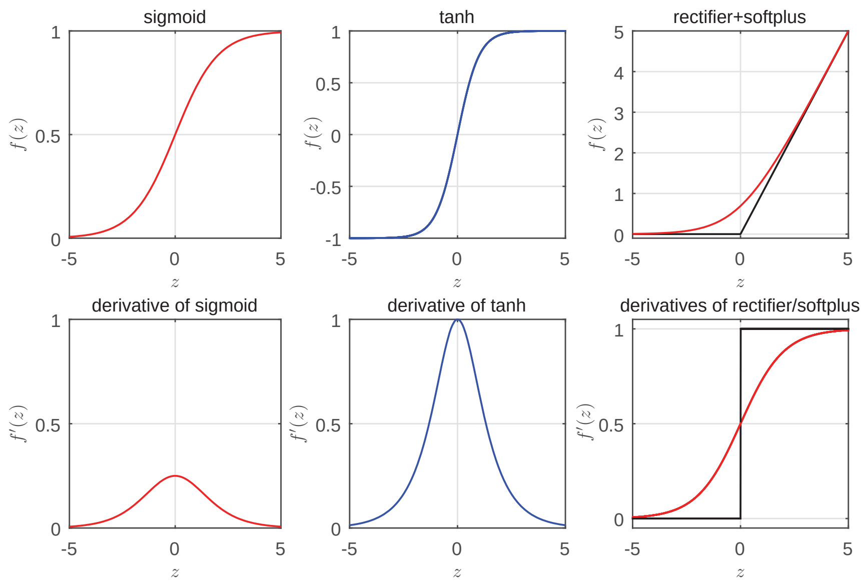

FIGURE 47.69 Examples of activation functions and their derivatives. (*Left*) Sigmoid function. (*Center*) Hyperbolic tangent function. (*Right*) Rectifier and softplus functions.

On the right-hand side of Fig. 47.68 we show two compact representations for neurons. The only difference is the additional arc that appears inside the circle in the top right corner.

This arc is used to indicate the presence of a nontrivial activation function. This is because, sometimes, the neuron may appear without the activation function (i.e., with $f(z) = z$), in which case it will be simply operating as a pure linear combiner. In Fig. 47.70 we compare the structure of the sigmoid neuron with the structure of the Perceptron neuron, which relies on the use of the sign function to drive the learning process. We encountered the Perceptron neuron earlier while studying the Perceptron algorithm — recall expressions (47.761)–(47.762). One of the key advantages in using a continuous (smooth) activation function, such as the sigmoid or tanh functions, over the discontinuous sign function is that the resulting networks of neurons will respond more gracefully to small changes in their internal signals. For example, with the output of the sign function changing drastically from 0 to 1 even for the slightest changes in its argument, networks consisting solely of Perceptron neurons can find their output signals change significantly in response to small internal signal variations. Smooth activation functions limit this sensitivity.

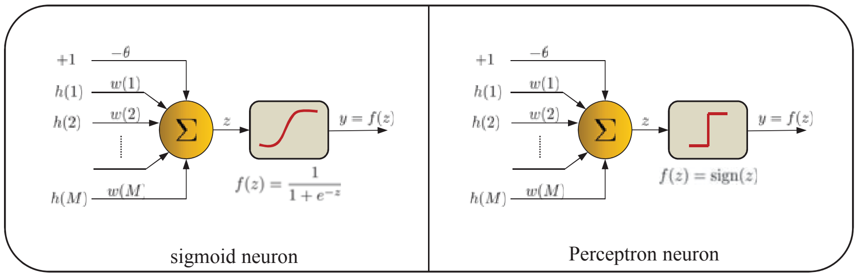

FIGURE 47.70 (*Left*) Sigmoid neuron where the output of the linear combiner is smoothed through a sigmoid activation function. (*Right*) Perceptron neuron where the output of the linear combiner is applied to a hard-thresholding sign function.

47.23.2 Feedforward Networks

We now combine several neurons in a forward structure to obtain a common and powerful implementation in the form of a feedforward multi-layer neural network. In this implementation, information flows forward in the network and there is no loop to feed signals from some future layer back to an earlier layer. Figure 47.71 illustrates this structure for a network consisting of an input layer, three hidden layers, and an output layer. There are two output nodes in the output layer, denoted by $y(1)$ and $y(2)$, and three input nodes in the input layer, denoted by $h(1), h(2)$, and $h(3)$. Note that we are excluding the bias source $+1$ from the number of input nodes. There are also successively three, two, and three nodes in the hidden layers, again excluding the bias sources. The nodes in each layer are numbered with the numbers placed inside the symbol for the neuron. Note that, contrary to all other nodes in the network, the input nodes are not actual neurons. We will be using the terminology "node" to refer to any arbitrary node in the network, whether it is a neuron or not. In this example, the nodes in the output layer employ activation functions. There are situations where these output nodes may be simple combiners without activation functions (for example, where the network is applied to the solution of regression problems).

For convenience, we shall employ the vector and matrix notation to examine how signals flow through the network. We let L denote the number of layers in the network, including the input and output layers. In the example of Fig. 47.71 we have $L = 5$ layers. Usually, networks with more than three hidden layers ($L > 5$) are referred to as *deep* networks. For

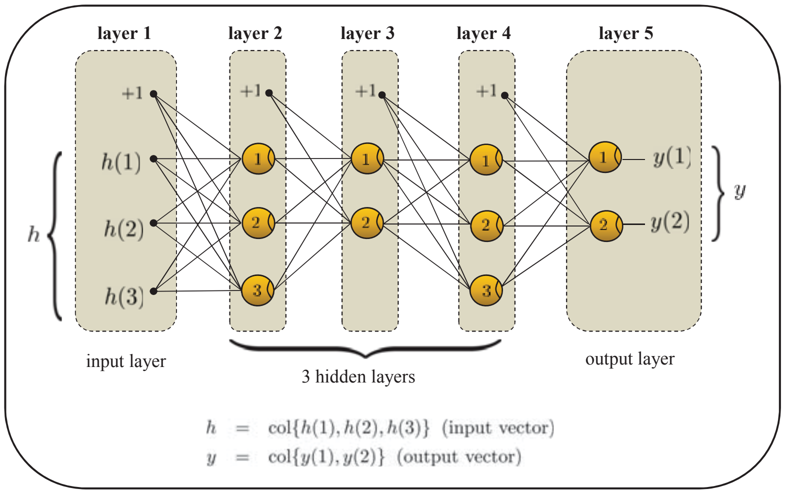

FIGURE 47.71 A feedforward neural network consisting of an input layer, three hidden layers, and an output layer. There are three input nodes in the input layer (excluding the bias source denoted by $+1$) and two output nodes in the output layer denoted by $y(1)$ and $y(2)$.

every layer $\ell = 1, 2, \ldots, L$, we let n_ℓ denote the number of nodes in that layer (again, our convention excludes the bias sources from this count). For our example, we have

$$n_1 = 3, \quad n_2 = 3, \quad n_3 = 2, \quad n_4 = 3, \quad n_5 = 2 \tag{47.958}$$

Now, between any two layers in the network, there is a collection of combination coefficients that scale the signals arriving from the nodes in the prior layer. For example, if we focus on layers 2 and 3, these coefficients can be collected into a matrix W_2^{T} of size $n_2 \times n_3$ (or W_2 of size $n_3 \times n_2$) with entries denoted individually by

$$W_2^{\mathsf{T}} \triangleq \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \\ w_{31}^{(2)} & w_{32}^{(2)} \end{bmatrix}, \quad (n_2 \times n_3) \tag{47.959}$$

In this notation, the scalar $w_{ij}^{(\ell)}$ has the following interpretation:

$$w_{ij}^{(\ell)} = \text{weight from node } i \text{ in layer } \ell \text{ to node } j \text{ in layer } \ell + 1 \tag{47.960}$$

We also associate with these layers a bias vector of size n_3, containing the coefficients that scale the bias arriving into layer 3 from layer 2, with entries denoted by

$$\theta_2 \triangleq \begin{bmatrix} \theta_2(1) \\ \theta_2(2) \end{bmatrix}, \quad (n_3 \times 1) \tag{47.961}$$

where the notation $\theta_\ell(j)$ has the following interpretation:

$$-\theta_\ell(j) = \text{weight from } +1 \text{ bias source in layer } \ell \text{ to node } j \text{ in layer } \ell + 1 \tag{47.962}$$

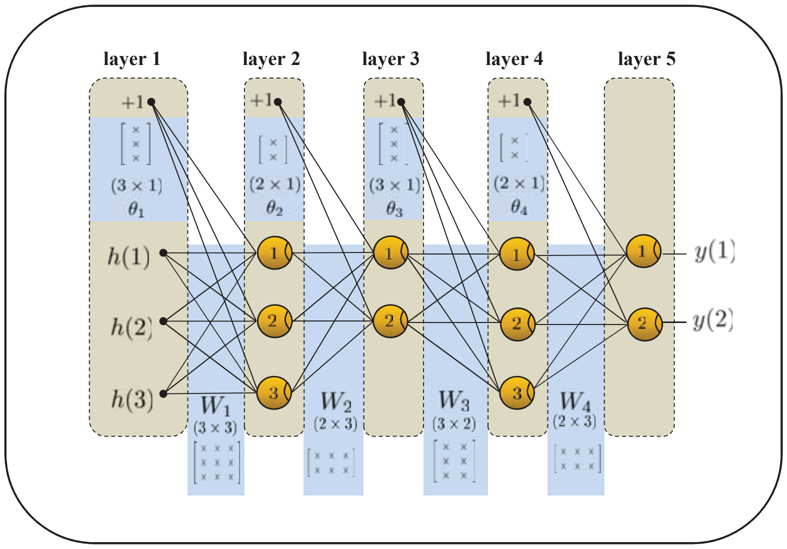

FIGURE 47.72 The combination weights between successive layers are collected into matrices, W_ℓ, for $\ell = 1, 2, \ldots, L-1$. Likewise, the bias coefficients for each hidden layer are collected into vectors, θ_ℓ, for $\ell = 1, 2, \ldots, L-1$.

Figure 47.72 illustrates the four weight matrices, $\{W_1, W_2, W_3, W_4\}$, and four bias vectors, $\{\theta_1, \theta_2, \theta_3, \theta_4\}$, that are associated with the network of Fig. 47.71. Figure 47.73 illustrates this construction more explicitly for the case of W_2. More generally, the combination weights between two layers ℓ and $\ell+1$ are collected into a matrix W_ℓ of size $n_{\ell+1} \times n_\ell$ and the bias weights of a layer $\ell+1$ are collected into a column vector θ_ℓ of size $n_{\ell+1} \times 1$.

Using the vector and matrix quantities so defined, we can now examine the flow of signals through the network. Continuing with Fig. 47.73, we collect the signals at the output of layer $\ell = 3$ into a column vector denoted by y_3 and with entries

$$y_3 = \begin{bmatrix} y_3(1) \\ y_3(2) \end{bmatrix}, \qquad (n_3 \times 1) \tag{47.963}$$

where the notation $y_\ell(j)$ has the following interpretation:

$$y_\ell(j) = \text{output of node } j \text{ at layer } \ell \tag{47.964}$$

It is clear that the output vector for layer $\ell = 3$ is given by

$$y_3 = f\left(W_2 y_2 - \theta_2\right) \tag{47.965}$$

in terms of the output vector for layer 2, and where the notation $f(z)$, when used for a *vector* argument z, means that the activation function is applied to each entry of z individually. For later use, we similarly collect the signals prior to the activation function at layer 3 into a column vector

$$z_3 = \begin{bmatrix} z_3(1) \\ z_3(2) \end{bmatrix}, \qquad (n_3 \times 1) \tag{47.966}$$

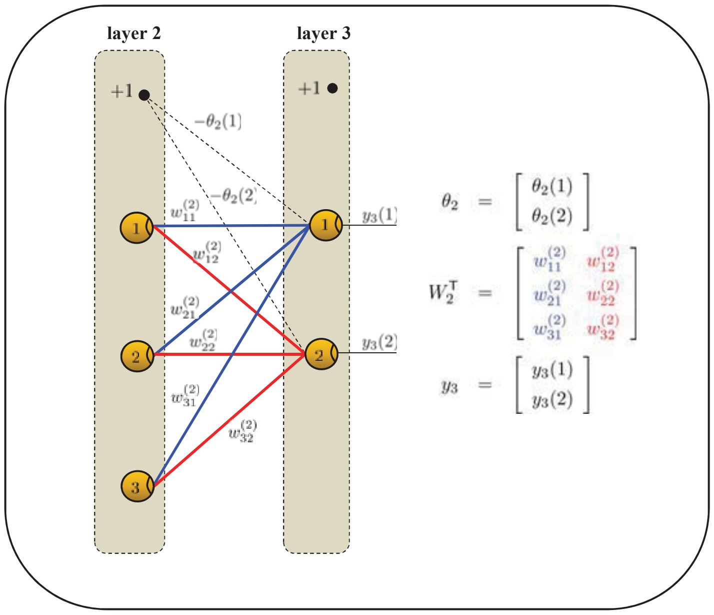
layer 2
layer 3
+1
+1
$-\theta_2(1)$
$-\theta_2(2)$
$w^{(2)}_{11}$
$w^{(2)}_{12}$
$w^{(2)}_{21}$
$w^{(2)}_{22}$
$w^{(2)}_{31}$
$w^{(2)}_{32}$
1
2
3
1
2
$y_3(1)$
$y_3(2)$
$\theta_2 = \begin{bmatrix} \theta_2(1) \\ \theta_2(2) \end{bmatrix}$
$W_2^{\mathsf{T}} = \begin{bmatrix} w^{(2)}_{11} & w^{(2)}_{12} \\ w^{(2)}_{21} & w^{(2)}_{22} \\ w^{(2)}_{31} & w^{(2)}_{32} \end{bmatrix}$
$y_3 = \begin{bmatrix} y_3(1) \\ y_3(2) \end{bmatrix}$

FIGURE 47.73 Combination and bias weights between layers 2 and 3 for the network shown in Fig. 47.71. The combination weights between nodes are collected into a matrix W_2^{T} of size $n_2 \times n_3$, while the bias weights are collected into a vector θ_2 of size $n_3 \times 1$.

where the notation $z_\ell(j)$ has the following interpretation:

$$z_\ell(j) = \text{output of node } j \text{ at layer } \ell \text{ prior to activation function} \tag{47.967}$$

If we now let $y = \text{col}\{y(1), y(2)\}$ denote the column vector that collects the outputs of the neural network, we then arrive at the following description for the flow of signals through a feedforward network. In this description, we are using the notation $z^{(\ell)}$ to denote the output vector of layer ℓ prior to the activation function. In the same token, whenever convenient, we shall also use the notation $(z^{(L)}, y^{(L)})$ to refer the pre- and post-activation vectors (z, y) for the output layer.

Propagation of signals through a feedforward neural network with L layers

```
start with $y_1 = h$.
repeat for $\ell = 1, \ldots, L-1$ :
    $z_{\ell+1} = W_\ell y_\ell - \theta_\ell$
    $y_{\ell+1} = f(z_{\ell+1})$
end
$y = y_L$
```
(47.968)

Column and Row Partitioning of Combination Matrices

Continuing with layers 2 and 3, and the corresponding combination matrix W_2^{T}, we remark for later use that we can partition W_2^{T} either in column form or in row form and write

$$W_2^{\mathsf{T}} = \left[\begin{array}{c|c} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \\ w_{31}^{(2)} & w_{32}^{(2)} \end{array}\right] = \left[\begin{array}{cc} w_{11}^{(2)} & w_{12}^{(2)} \\ \hline w_{21}^{(2)} & w_{22}^{(2)} \\ \hline w_{31}^{(2)} & w_{32}^{(2)} \end{array}\right] \tag{47.969}$$

In the first case, when the partitioning is column-wise, we observe that W_2^{T} consists of two columns; one for each of the nodes in the subsequent layer 3. The entries of the first column are the weighting coefficients on the edges arriving at the first node in layer 3. The entries of the second column are the weighting coefficients on the edges arriving at the second node in the same layer 3. In other words, each column of W_2^{T} consists of the weighting coefficients on the edges arriving at the corresponding node in the subsequent layer 3.

In the second case, when the partitioning is row-wise, we observe that W_2^{T} consists of three columns; one row for each of the nodes in the originating layer 2. The entries of the first row are the weighting coefficients on the edges emanating from the first node in layer 2 to all nodes in layer 3. The entries of the second row are the weighting coefficients on the edges emanating from second node in layer 2 to the nodes in layer 3, and likewise for the third row of W_2^{T}. In other words, each row of W_2^{T} consists of the weighting coefficients on the edges emanating from the nodes in layer 2.

Figure 47.74 illustrates this partitioning for a generic combination matrix between two arbitrary layers ℓ and $\ell+1$. For later use, we find it convenient to represent the rows of W_ℓ^{T} in column form and to introduce the column vector $w_j^{(\ell)}$ of size $(n_{\ell+1}\times 1)$ defined as follows:

$$w_i^{(\ell)} \;=\; \text{weight vector emanating from node } i \text{ in layer } \ell \tag{47.970}$$

For example, from (47.969) we have

$$w_1^{(2)} \;=\; \text{col}\left\{w_{11}^{(2)}, w_{12}^{(2)}\right\} \;=\; \text{weights from node 1 in layer 2} \tag{47.971a}$$
$$w_2^{(2)} \;=\; \text{col}\left\{w_{21}^{(2)}, w_{22}^{(2)}\right\} \;=\; \text{weights from node 2 in layer 2} \tag{47.971b}$$
$$w_3^{(2)} \;=\; \text{col}\left\{w_{31}^{(2)}, w_{32}^{(2)}\right\} \;=\; \text{weights from node 3 in layer 2} \tag{47.971c}$$

47.23.3 Regression and Classification

We next examine how a neural network can be used to solve regression or classification problems. Since we will generally be dealing with feature vectors that are indexed by the time or sample subscript n, say, $h_n \in \mathbb{R}^M$, we shall denote the corresponding output vector y at the output layer L (i.e., y_L) more compactly by $y_n \in \mathbb{R}^Q$ and remove the subscript L whenever convenient since it will be understood from the context; the more complete notation would have been to denote this vector by $y_{L,n}$ to highlight the fact that it is the output vector at time n. It is sufficient for our purposes in this section to view the network as a system that maps input vectors $\{h_n \in \mathbb{R}^M\}$ into output vectors $\{y_n \in \mathbb{R}^Q\}$. Since the network is effectively a multi-input multi-output system, we can exploit this level of generality to solve, for example, multiclass or multi-label classification problems:

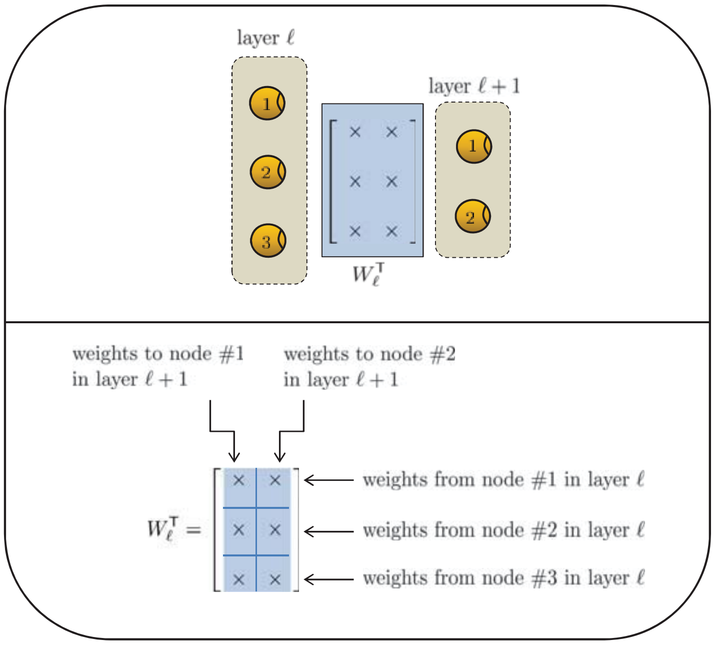
layer ℓ
layer $\ell+1$
W_ℓ^T
weights to node #1 in layer $\ell+1$
weights to node #2 in layer $\ell+1$
$W_\ell^\mathsf{T} =$
weights from node #1 in layer ℓ
weights from node #2 in layer ℓ
weights from node #3 in layer ℓ

FIGURE 47.74 The weight matrix W_ℓ^T linking layer ℓ to layer $\ell+1$ can be partitioned either in column form or in row form. The columns of W_ℓ^T represent the weights arriving at the nodes in layer $\ell+1$. The rows of W_ℓ^T represent the weights emanating from the nodes in layer ℓ.

(a) In some applications, one might be interested in determining whether the feature vector, h_n, implies that a certain condition "A" is present or not (such as checking whether h_n corresponds to a female or male individual) and whether some second condition "B" is also present or not (such as checking whether the same individual has a particular disease or not). In cases like these, we will be able to train a neural network to generate two class labels at its output, say, $\gamma_n(A)$ and $\gamma_n(B)$. One of these labels relates to condition "A" while the second label relates to condition "B". This scenario corresponds to a *multi-label* classification problem. We have two labels "A" and "B", and we would like to classify a given feature vector according to each label. We can handle such multi-label problems by replacing the usual *scalar* class variable, $\gamma(n)$, by a *vector*-valued class variable, $\gamma_n \in \mathbb{R}^Q$. Each entry of this vector variable will correspond to one label and will assume values ± 1.

(b) In some other applications, one might be interested in classifying a feature vector, h_n, into only one from among a collection of classes. For example, given a feature vector extracted from the image of a handwritten digit, one may want to identify the digit (i.e., classify the feature vector into one of ten classes, $0, 1, 2, \ldots, 9$; one class for each possible digit. We encountered such multiclass classification problems in several occasions earlier in our discussions, e.g., in Examples 47.47 and 47.51, while discussing the one-versus-all and one-versus-one strategies. These solution methods focused on reducing a *multiclass* classification problem into a collection of binary classification problems. In the neural network formulation, multiclass classification problems can be addressed directly by again replacing the *scalar* class variable, $\gamma(n)$, by a *vector*-valued class variable, $\gamma_n \in \mathbb{R}^Q$. Each entry of this vector variable will correspond to one class. When h_n belongs to some class c, the $c-$th entry of γ_n will be one, while all other entries will be zero.

(c) We observe that in example (a) on multi-label problems, the entries of γ_n assume values ± 1, while in example (b) on multiclass problems, the entries of γ_n assume values $\{1, 0\}$. The choice of the values $(\{+1, -1\}, \{1, 0\})$ is a matter of preference. For uniformity, we can redefine the labels to be the same and use either ± 1 for both types of problems or $\{1, 0\}$ for both types of problems.

We therefore consider a feedforward network with Q output nodes, leading to an output vector $y \in \mathbb{R}^Q$. The output nodes are indexed $q = 1, 2, \ldots, Q$. The input to the network is a feature vector, $h \in \mathbb{R}^M$. The mapping from h to y depends on the weight matrices $\{W_\ell\}$ and the bias vectors $\{\theta_\ell\}$ between successive layers. We assume we are given a collection of N training points $\{\gamma_n, h_n\}$, where $\gamma_n \in \mathbb{R}^Q$ and $h_n \in \mathbb{R}^M$. The following observations are now in place depending on whether the network is employed to perform regression or classification:

(a) The sigmoid function in Table 47.11 generates values in the range $(0, 1)$, while the hyperbolic tangent function generates values in the range $(-1, 1)$. Therefore, for classification problems, we can either:

(a.1) redefine the entries of γ_n to assume the values 0 or 1 when the sigmoid function is used, or

(a.2) redefine the entries of γ_n to assume the values ± 1 when the hyperbolic tangent function is used.

Under option (a.1), and during normal operation of the network following training, we can map the entries of the output vector y corresponding to a feature input h to classes $\{0, 1\}$ by performing the comparisons:

$$\begin{cases} \text{if } y(q) \geq 1/2 : \text{declare } \gamma(q) = 1 \\ \text{if } y(q) < 1/2 : \text{declare } \gamma(q) = 0 \end{cases} \tag{47.972}$$

Likewise, under option (a.2), we map any output of the network to classes ± 1 by performing the comparisons:

$$\begin{cases} \text{if } y(q) \geq 0 : \text{declare } \gamma(q) = +1 \\ \text{if } y(q) < 0 : \text{declare } \gamma(q) = -1 \end{cases} \tag{47.973}$$

(b) In regression problems, the entries of γ_n do not represent classes or labels but assume real-values. In these cases, the nodes at the output layer of the network do not employ activation functions; they consist solely of linear combiners. The output of each of these combiners provides an estimate, $\widehat{\gamma}_n(q)$, for the $q-$th entry, $\gamma_n(q)$, of γ_n.

47.23.4 Backpropagation Algorithm

The backpropagation algorithm is one of the most celebrated procedures for training neural networks. Due to the interconnected nature of the network, with signals from one layer feeding into a subsequent layer, in addition to the presence of nonlinear activation functions, it is critical to pursue a systematic and structured presentation to facilitate the derivation of the algorithm. As a byproduct, the algorithm will help provide an effective recursive construction for evaluating how the performance of the network varies in response to small changes in its internal coefficients.

Thus, let y_n denote the output vector for the neural network in response to a feature vector h_n. We also let z_n denote the output vector prior to activation so that

$$y_n = f(z_n) \tag{47.974}$$

We can view the variable y_n as a prediction for the class variable γ_n, i.e.,

$$y_n = \widehat{\gamma}_n \tag{47.975}$$

Given training data $\{\gamma_n, h_n\}$, for $n = 0, 1, \ldots, N-1$, the objective is to train the network, i.e., to select its weight matrices and bias vectors $\{W_\ell, \theta_\ell\}$, in order for the mapping from the input space, $h \in \mathbb{R}^M$, to the output space, $y \in \mathbb{R}^Q$, to approximate well the mapping from h to γ that is reflected in the training data.

Motivated by the discussion in Sec. 47.14 on regularized least-squares problems, we may consider an empirical risk optimization problem similar to (47.421), namely,

$$\{W_\ell^\circ, \theta_\ell^\circ\} \;\overset{\Delta}{=}\; \arg\min_{W_\ell, \theta_\ell} \;\frac{1}{N}\left[\sum_{\ell=1}^{L-1} \rho\|W_\ell\|_{\rm F}^2 \;+\; \sum_{n=0}^{N-1} \|\gamma_n - y_n\|^2\right] \tag{47.976}$$

where the first term applies regularization to the sum of the squared Frobenius norms of the weight matrices between successive layers. Recall that the squared Frobenius norm of a matrix is the sum of the squares of its entries (i.e., it is the squared Euclidean norm of the vectorized form of the matrix):

$$\|W_\ell\|_{\rm F}^2 \;=\; \sum_{i=1}^{n_\ell}\sum_{j=1}^{n_{\ell+1}} \left|w_{ij}^{(\ell)}\right|^2 \tag{47.977}$$

Therefore, the regularization term is in effect adding the squares of all combination weights in the network. As explained at some length earlier in Secs. 47.13 and 47.14, regularization helps avoid over-fitting and improves the generalization ability of the network. Other forms of regularization are possible, including ℓ_1-regularization (see Prob. 47.155), as well as other risk functions (see Probs. 47.159–47.162). We will motivate some of these alternative costs later in Secs. 47.23 and 47.23; their main purpose is to avoid a problem that arises from using (47.976) and which causes a slow down in the learning rate of some nodes in the network. We may also consider an alternative to (47.976) where the scaling by $1/N$ appears multiplying the second term only, namely,

$$\{W_\ell^\circ, \theta_\ell^\circ\} \;\overset{\Delta}{=}\; \arg\min_{W_\ell, \theta_\ell} \;\sum_{\ell=1}^{L-1} \rho\|W_\ell\|_{\rm F}^2 \;+\; \frac{1}{N}\sum_{n=0}^{N-1} \|\gamma_n - y_n\|^2 \tag{47.978}$$

The analysis that follows is equally applicable to this case with minimal adjustments, especially since we will be dealing with stochastic-gradient and batch algorithms. For this reason, it is sufficient to continue with (47.978) and consider the empirical risk:

$$J_{\rm emp}(W, \theta) \;\overset{\Delta}{=}\; \sum_{\ell=1}^{L-1} \rho\|W_\ell\|_{\rm F}^2 \;+\; \frac{1}{N}\sum_{n=0}^{N-1} \|\gamma_n - y_n\|^2 \tag{47.979}$$

This cost is dependent on all the unknowns $\{W_\ell, \theta_\ell\}$; we are denoting this dependency generically by using the letters W and θ as arguments for $J_{\rm emp}(\cdot, \cdot)$. In order to implement iterative procedures for minimizing $J_{\rm emp}(W, \theta)$, e.g., either in batch gradient-descent form or in stochastic gradient-descent form, we need to know how to evaluate the gradients of $J_{\text{emp}}(W,\theta)$ relative to the individual entries of $\{W_\ell,\theta_\ell\}$, namely, we need to evaluate:

$$\frac{\partial J_{\text{emp}}(W,\theta)}{\partial w_{ij}^{(\ell)}} \quad \text{and} \quad \frac{\partial J_{\text{emp}}(W,\theta)}{\partial \theta_\ell(j)} \tag{47.980}$$

The backpropagation algorithm is the procedure that enables us to compute these gradients in an effective manner, as we explain in the next section.

In the meantime, observe from (47.979) that regularization is not applied to the bias vectors $\{\theta_\ell\}$; these entries are embedded in the output signals $\{y_n\}$, as is evident from (47.968). Observe further that the cost (47.979) is not purely quadratic in the unknown variables; actually, the dependency of y_n on the variables $\{W_\ell,\theta_\ell\}$ is highly nonlinear due to the activation functions at the nodes. As a result, this risk function may have multiple local minima. We will nevertheless apply stochastic-gradient descent to seek a (local) minimizer. It has been observed in practice, through extensive experimentation, that the resulting backpropagation algorithm works well in general despite the nonlinear and also non-convex nature of the risk function.

Differentiation Relative to the Combination Weights

We first explain how to evaluate the gradients relative to the combination weights. A similar argument will apply to the bias coefficients. In the derivation that follows, we shall denote the individual entries of the output vectors $\{z_n, y_n\}$ by

$$y_n = \begin{bmatrix} y_n(1) \\ y_n(2) \\ \vdots \\ y_n(Q) \end{bmatrix}, \qquad z_n = \begin{bmatrix} z_n(1) \\ z_n(2) \\ \vdots \\ z_n(Q) \end{bmatrix} \tag{47.981}$$

We start by noting from (47.979) that

$$\begin{aligned} \frac{\partial J_{\text{emp}}(W,\theta)}{\partial w_{ij}^{(\ell)}} &= 2\rho w_{ij}^{(\ell)} + \frac{1}{N}\sum_{n=0}^{N-1} \frac{\partial \|\gamma_n - y_n\|^2}{\partial w_{ij}^{(\ell)}} \\ &= 2\left(\rho w_{ij}^{(\ell)} - \frac{1}{N}\sum_{n=0}^{N-1} (\gamma_n - y_n)^{\mathsf{T}} \frac{\partial y_n}{\partial w_{ij}^{(\ell)}}\right) \end{aligned} \tag{47.982}$$

where the rightmost gradient vector consists of the partial derivatives of the individual entries of y_n relative to the weight quantity:

$$\frac{\partial y_n}{\partial w_{ij}^{(\ell)}} = \begin{bmatrix} \partial y_n(1)/\partial w_{ij}^{(\ell)} \\ \partial y_n(2)/\partial w_{ij}^{(\ell)} \\ \vdots \\ \partial y_n(Q)/\partial w_{ij}^{(\ell)} \end{bmatrix} \tag{47.983}$$

We are therefore motivated to evaluate the individual partial derivatives that appear in this vector and which are of the form:

$$\frac{\partial y_n(q)}{\partial w_{ij}^{(\ell)}}, \quad \text{for any } q = 1,2,\ldots,Q \tag{47.984}$$

We know from (47.974) that

$$y_n(q) = f(z_n(q)) \tag{47.985}$$

Now recall the chain rule for differentiation, namely, for any two functions $f(x)$ and $g(x)$ of scalar arguments:

$$\frac{\partial f(g(x))}{\partial x} \;=\; f'(g(x))\frac{\partial g(x)}{\partial x} \tag{47.986}$$

written in terms of the derivative of $f(x)$ evaluated at $g(x)$ and where, for convenience, we are using the prime notation to refer to the derivative of $f(x)$. Applying this rule to (47.985) we get

$$\boxed{\frac{\partial y_n(q)}{\partial w_{ij}^{(\ell)}} = f'\left(z_n(q)\right)\,\frac{\partial z_n(q)}{\partial w_{ij}^{(\ell)}}} \tag{47.987}$$

Therefore, the problem of evaluating the partial derivatives (47.984) of the post-activation signals $y_n(q)$ relative to the combination weights is reduced to the problem of evaluating the partial derivatives of the pre-activation signals $z_n(q)$ relative to the same weights:

$$\frac{\partial z_n(q)}{\partial w_{ij}^{(\ell)}}, \quad \text{for any } q = 1, 2, \ldots, Q \tag{47.988}$$

Before explaining how the backpropagation algorithm carries out these calculations, we illustrate the mechanisms involved in the computation by considering an example.

Example 47.64 (Partial derivatives relative to combination weights)

The computation of the partial derivatives (47.988) is very much dependent on the *location* of the combination weight in the network. Let us consider the feedforward network shown in Fig. 47.75. The network consists of five layers with one output node (i.e., $Q = 1$). The nodes in each layer are numbered. The signals at the output node, before and after activation, are simply denoted by $\{z, y\}$; we will drop the time subscript n in this example to simplify the presentation since the notion of time is irrelevant to the calculations. We are interested in evaluating the partial derivatives

$$\frac{\partial z}{\partial w_{ij}^{(\ell)}} \tag{47.989}$$

relative to several choices for the weight $w_{ij}^{(\ell)}$.

In the figure we are highlighting three possible choices for the combination weight, denoted by

$$w_{21}^{(2)} \;=\; \text{weight from node 2 in layer 2 to node 1 in layer 3} \tag{47.990a}$$
$$w_{12}^{(3)} \;=\; \text{weight from node 1 in layer 3 to node 2 in layer 4} \tag{47.990b}$$
$$w_{21}^{(4)} \;=\; \text{weight from node 2 in layer 4 to the output node} \tag{47.990c}$$

We are also labeling the output signals for each relevant node, e.g., by using the notation $\{z_1^{(4)}, y_1^{(4)}\}$ for the output signals corresponding to node 1 in layer 4. We are interested in evaluating the three partial derivatives

$$\frac{\partial z}{\partial w_{21}^{(2)}}, \quad \frac{\partial z}{\partial w_{12}^{(3)}}, \quad \frac{\partial z}{\partial w_{21}^{(4)}} \tag{47.991}$$

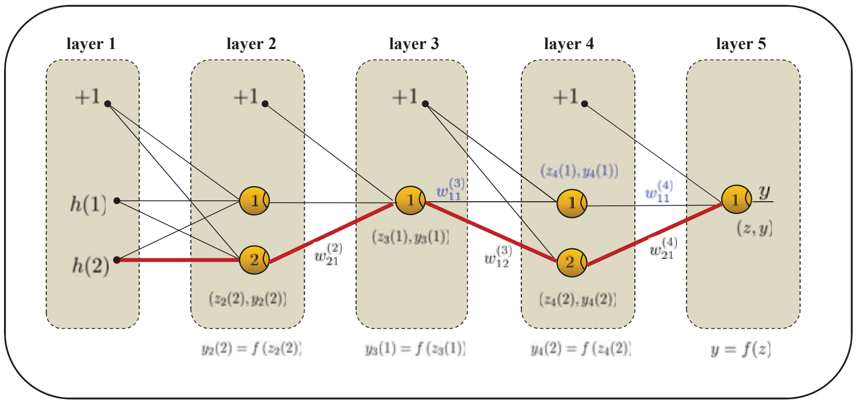

FIGURE 47.75 A feedforward network consisting of 5 layers with one output node, used to illustrate the computation of partial derivatives relative to three combination weights appearing in different layers.

The output signals of the nodes from which these weights emanate, before and after activation, satisfy

$$y_2(2) = f(z_2(2)) \tag{47.992a}$$
$$y_3(1) = f(z_3(1)) \tag{47.992b}$$
$$y_4(2) = f(z_4(2)) \tag{47.992c}$$
$$y = f(z) \tag{47.992d}$$

where $f(\cdot)$ refers to the activation function. It is clear that these output signals are related to the highlighted combination weights via the relations

$$z_3(1) = y_2(2)w_{21}^{(2)} + \text{terms independent of } w_{21}^{(2)} \tag{47.993a}$$
$$z_4(2) = y_3(1)w_{12}^{(3)} + \text{terms independent of } w_{12}^{(3)} \tag{47.993b}$$
$$z = y_4(2)w_{21}^{(4)} + \text{terms independent of } w_{21}^{(4)} \tag{47.993c}$$

For the combination weight $w_{21}^{(4)}$, which is feeding into the output layer, the derivative calculation is trivial and given by

$$\boxed{\frac{\partial z}{\partial w_{21}^{(4)}} = y_4(2)} \tag{47.994}$$

Let us consider next the combination weight $w_{12}^{(3)}$ from the preceding layer. In this case, we note that the influence of this weight on z is felt through node 2 in layer 4. Indeed, if we use generically the letter "c" to refer to terms that are independent of the variable of interest, which in this case is $w_{12}^{(3)}$, we have

$$\begin{aligned} z &= y_4(2)w_{21}^{(4)} + c \\ &= f(z_4(2))\, w_{21}^{(4)} + c \\ &= f\left(y_3(1)w_{12}^{(3)} + c\right) w_{21}^{(4)} + c \end{aligned} \tag{47.995}$$

so that, using the chain rule of differentiation, we obtain

$$\boxed{\frac{\partial z}{\partial w_{12}^{(3)}} = y_3(1)\, f'\left(z_4(2)\right)\, w_{21}^{(4)}} \tag{47.996}$$

in terms of the derivative of the activation function, denoted by means of the prime notation. Expression (47.996) reveals a useful structure. Assume, for the sake of argument, that there is no activation function in node 2 in layer 4 so that the path through the weights $\{w_{12}^{(3)}, w_{21}^{(4)}\}$ is independent of nonlinearities. In that case, we would simply have (instead of (47.995)):

$$z = y_3(1) w_{12}^{(3)} w_{21}^{(4)} \; + \; c \tag{47.997}$$

and the partial derivative of the output signal z relative to $w_{12}^{(3)}$ will be the product $y_3(1)w_{21}^{(4)}$. Comparing with (47.996) we find that the effect of the activation function is to incorporate a correction term, represented by its derivative at $z_4(2)$ (i.e., at the location of the nonlinearity). We will find that the same construction holds for derivative calculations relative to other combination weights. We carry out the next calculation explicitly, from first principles, and then comment on how the same result could have been deduced based on this intuition.

Thus, let us consider now the combination weight $w_{21}^{(2)}$ from the earliest layer. As we move backwards in the layers, the influence of an earlier combination weight on the output signal z will be felt from various sources, and not from only one source. For example, the signal with combination weight $w_{21}^{(2)}$ feeds into node 1 in layer 3, whose output in turn feeds into the two nodes in layer 4. That is why we highlighted in blue color in the figure the weights and output signals along the additional path that needs to be taken into account at this stage. Thus, observe now that

$$\begin{aligned} z &= y_4(2) w_{21}^{(4)} \; + \; y_4(1) w_{11}^{(4)} \; + \; c \\ &= f\left(z_4(2)\right) w_{21}^{(4)} \; + \; f\left(z_4(1)\right) w_{11}^{(4)} \; + \; c \\ &= f\left(y_3(1) w_{12}^{(3)} + c\right) w_{21}^{(4)} \; + \; f\left(y_3(1) w_{11}^{(3)} + c\right) w_{11}^{(4)} \; + \; c \\ &= f\left(f\left(z_3(1)\right) w_{12}^{(3)} + c\right) w_{21}^{(4)} \; + \; f\left(f\left(z_3(1)\right) w_{11}^{(3)} + c\right) w_{11}^{(4)} \; + \; c \\ &= f\left(f\left(y_2(2) w_{21}^{(2)} + c\right) w_{12}^{(3)} + c\right) w_{21}^{(4)} \; + \; f\left(f\left(y_2(2) w_{21}^{(2)} + c\right) w_{11}^{(3)} + c\right) w_{11}^{(4)} \; + \; c \end{aligned} \tag{47.998}$$

so that, using the chain rule of differentiation, we obtain

$$\boxed{\frac{\partial z}{\partial w_{21}^{(2)}} = y_2(2) w_{12}^{(3)} w_{21}^{(4)}\, f'\left(z_3(1)\right)\, f'\left(z_4(2)\right) \; + \; y_2(2) w_{11}^{(3)} w_{11}^{(4)}\, f'\left(z_3(1)\right)\, f'\left(z_4(1)\right)} \tag{47.999}$$

Here again we observe that our earlier remark on the structure of these derivative calculations continues to hold. To see this, let us ignore all the activation functions (nonlinearities) on the paths through which the signal $y_2(2)$ travels to arrive at the output node. In that case, we would replace (47.998) by

$$\begin{aligned} z &= y_4(2) w_{21}^{(4)} + y_1(4) w_{11}^{(4)} + c \\ &= z_4(2) w_{21}^{(4)} + z_1(4) w_{11}^{(4)} + c \\ &= \left(y_3(1) w_{12}^{(3)} + c\right) w_{21}^{(4)} + \left(y_3(1) w_{11}^{(3)} + c\right) w_{11}^{(4)} + c \\ &= \left(z_3(1) w_{12}^{(3)} + c\right) w_{21}^{(4)} + \left(z_3(1) w_{11}^{(3)} + c\right) w_{11}^{(4)} + c \\ &= \left(\left(y_2(2) w_{21}^{(2)} + c\right) w_{12}^{(3)} + c\right) w_{21}^{(4)} + \left(\left(y_2(2) w_{21}^{(2)} + c\right) w_{11}^{(3)} + c\right) w_{11}^{(4)} + c \end{aligned} \tag{47.1000}$$

and the derivative of the output signal z relative to $w_{21}^{(2)}$, under this contrived analysis, will be the sum of products

$$y_2(2)w_{12}^{(3)}w_{21}^{(4)} \;+\; y_2(2)w_{11}^{(3)}w_{11}^{(4)} \tag{47.1001}$$

Comparing with (47.999) we see that the actual derivative includes adjustments by the derivatives of the activation functions at the points of nonlinearities along the paths leading to the output node.

◇

Sensitivity Factors

The arguments in Example 47.64 indicate that the gradient analysis needs to be carried out in a systematic manner in order to facilitate the evaluation of the gradients. Rather than evaluate individual gradients separately, as we did in the example, we will instead rely on the convenience of the vector notation. To simplify the notation, we drop the time or iteration subscript, n, and reinstate it later when we list the final algorithm. The time subscript is not necessary for the arguments that follow.

We therefore consider a generic feedforward neural network consisting of L layers, including the input layer and the output layer. We denote the vector signals at the output layer by (z, y), with the letter z representing the signal prior to the activation function, i.e.,

$$y = f(z) \tag{47.1002}$$

We also denote the output signals at the $\ell-$th hidden layer by (z_ℓ, y_ℓ):

$$y_\ell = f\left(z_\ell\right) \tag{47.1003}$$

with individual entries indexed by $(z_\ell(i), y_\ell(i))$. The number of nodes within each hidden layer is denoted by n_ℓ (which excludes the bias source in that layer).

We further associate with each layer ℓ a sensitivity vector of size n_ℓ denoted by δ_ℓ and with entries $\delta_\ell(i)$. Each entry is defined as follows. We recall from (47.982) that the gradient calculations that we are interested in computing arise from the need to differentiate the unregularized term $\|\gamma - y\|^2$, for vectors γ and y, relative to $w_{ij}^{(\ell)}$. We define

$$\delta_\ell(j) \;\overset{\Delta}{=}\; \frac{\partial\|\gamma - y\|^2}{\partial z_\ell(j)} \tag{47.1004}$$

which measures how the (unregularized portion of the) risk varies in response to changes in the pre-activation signals, $z_\ell(j)$; we will show later in (47.1019) that this quantity also measures the sensitivity of the risk to changes in the bias coefficients. It will turn out that knowledge of the sensitivity variables will enable us to evaluate the desired partial derivatives (47.980). These quantities are related because variations in $\{w_{ij}^{(\ell)}, \theta_\ell(i)\}$ lead to variations in $z_{\ell+1}(j)$.

We can derive a recursive update for the vector δ_ℓ. Let us consider first the output layer for which $\ell = L$. We denoted these individual entries by $\{y(1), \ldots, y(Q)\}$. Likewise, we denoted the individual entries of z by $\{z_1, \ldots, z(Q)\}$. In this way, the chain rule for differentiation will imply that:

$$\begin{aligned}
\delta_L(j) &= \sum_{k=1}^{Q} \frac{\partial\|\gamma - y\|^2}{\partial y(k)}\,\frac{\partial y(k)}{\partial z(j)} \\
&= \sum_{k=1}^{Q} 2(y(k) - \gamma(k))\,\frac{\partial y(k)}{\partial z(j)} \\
&= 2(y(j) - \gamma(j))f'(z(j))
\end{aligned} \tag{47.1005}$$

Consequently, using the Hadamard product we can write

$$\boxed{\delta_L \;=\; 2(y-\gamma)\odot f'(z)} \tag{47.1006}$$

where the notation $a\odot b$ denotes elementwise multiplication for two vectors a and b. Next we evaluate δ_ℓ for the earlier layers. This calculation can be carried out recursively by relating δ_ℓ to $\delta_{\ell+1}$. Indeed, note that

$$\begin{aligned}
\delta_\ell(j) &\;\overset{\Delta}{=}\; \frac{\partial\|\gamma-y\|^2}{\partial z_\ell(j)} \\
&= \sum_{k=1}^{n_{\ell+1}} \frac{\partial\|\gamma-y\|^2}{\partial z_{\ell+1}(k)}\,\frac{\partial z_{\ell+1}(k)}{\partial z_\ell(j)} \\
&= \sum_{k=1}^{n_{\ell+1}} \delta_{\ell+1}(k)\frac{\partial z_{\ell+1}(k)}{\partial z_\ell(j)}
\end{aligned} \tag{47.1007}$$

where the right-most term involves differentiating the output of node k in layer $\ell+1$ relative to the output of node j in the previous layer, ℓ. The summation in the second equality results from the chain rule of differentiation since each entry of y is generally dependent on $z_{\ell+1}(k)$. The two signals $z_\ell(j)$ and $z_{\ell+1}(k)$ are related by the combination coefficient $w_{jk}^{(\ell)}$ since

$$z_{\ell+1}(k) \;=\; f\left(z_\ell(j)\right) w_{jk}^{(\ell)} \;+\; \text{terms independent of } w_{jk}^{(\ell)} \tag{47.1008}$$

It follows that

$$\begin{aligned}
\delta_\ell(j) &= \left(\sum_{k=1}^{n_{\ell+1}} \delta_{\ell+1}(k) w_{jk}^{(\ell)}\right) f'\left(z_\ell(j)\right) \\
&= f'\left(z_\ell(j)\right)\left(w_j^{(\ell)}\right)^{\mathsf T}\delta_{\ell+1}
\end{aligned} \tag{47.1009}$$

where we used the inner product notation in the last line by using the column vector $w_j^{(\ell)}$, which collects the combination weights emanating from node j in layer ℓ — recall the earlier definition (47.970). In vector form, we arrive at the following recursion, which runs backward from $\ell = L-1$ down to $\ell = 2$:

$$\boxed{\delta_\ell = f'\left(z_\ell\right)\odot\left(W_\ell^{\mathsf T}\delta_{\ell+1}\right)} \tag{47.1010}$$

Desired Partial Derivatives

We are now in a position to evaluate partial derivatives of the unregularized risk, $\|\gamma - y\|^2$, relative to the combination weights themselves, which are the quantities needed in

(47.982). Thus, following similar arguments to the above, we note that

$$\begin{aligned}
\frac{\partial\|\gamma-y\|^2}{\partial w_{ij}^{(\ell)}} &= \sum_{k=1}^{n_{\ell+1}} \frac{\partial\|\gamma-y\|^2}{\partial z_{\ell+1}(k)}\,\frac{\partial z_{\ell+1}(k)}{\partial w_{ij}^{(\ell)}} \\
&= \frac{\partial\|\gamma-y\|^2}{\partial z_{\ell+1}(j)}\,\frac{\partial z_{\ell+1}(j)}{\partial w_{ij}^{(\ell)}} \\
&= \delta_{\ell+1}(j)y_\ell(i) \qquad (47.1011)
\end{aligned}$$

where the second equality is because only $z_{\ell+1}(j)$ depends on $w_{ij}^{(\ell)}$. For example, if we apply this result to the combination matrix W_2^{T} defined earlier in (47.959), we find that the corresponding gradient calculations lead to (where, with some abuse of notation, we are writing $\partial\|\gamma-y\|^2/\partial W_2$ to refer to the resulting matrix):

$$\frac{\partial\|\gamma-y\|^2}{\partial W_2} = \begin{bmatrix} \delta_3(1)y_2(1) & \delta_3(1)y_2(2) & \delta_3(1)y_2(3) \\ \delta_3(2)y_2(1) & \delta_3(2)y_2(2) & \delta_3(2)y_2(3) \end{bmatrix} = \delta_3 y_2^{\mathsf{T}} \qquad (47.1012)$$

in terms of the outer product between the output vector, y_ℓ, for layer ℓ and the change vector, $\delta_{\ell+1}$, for layer $\ell+1$. Therefore, we can write

$$\boxed{\frac{\partial\|\gamma-y\|^2}{\partial W_\ell} = \delta_{\ell+1}y_\ell^{\mathsf{T}}} \qquad (47.1013)$$

so that from (47.979), and after restoring the subscript n,

$$\frac{\partial J_{\rm emp}(W,\theta)}{\partial W_\ell} = 2\rho W_\ell + \frac{1}{N}\sum_{n=0}^{N-1}\delta_{\ell+1,n}y_{n,\ell}^{\mathsf{T}} \qquad (47.1014)$$

Differentiation Relative to the Bias Weights

Similar arguments can be employed to compute the gradients of $\|\gamma-y\|^2$ relative to the bias weights, $\theta_\ell(i)$, across the layers. Thus, note that

$$\frac{\partial J_{\rm emp}(W,\theta)}{\partial \theta_\ell(i)} = \frac{1}{N}\sum_{n=0}^{N-1}\frac{\partial\|\gamma_n-y_n\|^2}{\partial\theta_\ell(i)} \qquad (47.1015)$$

so that, in a manner similar to the calculation (47.1011),

$$\begin{aligned}
\frac{\partial\|\gamma-y\|^2}{\partial \theta_\ell(i)} &= \sum_{k=1}^{n_{\ell+1}} \frac{\partial\|\gamma-y\|^2}{\partial z_{\ell+1}(k)}\,\frac{\partial z_{\ell+1}(k)}{\partial \theta_\ell(i)} \\
&= \frac{\partial\|\gamma-y\|^2}{\partial z_{\ell+1}(i)}\,\frac{\partial z_{\ell+1}(i)}{\partial \theta_\ell(i)} \\
&= -\delta_{\ell+1}(i) \qquad (47.1016)
\end{aligned}$$

where the second equality is because only $z_{\ell+1}(i)$ depends on $\theta_\ell(i)$, namely,

$$z_{\ell+1}(i) = -\theta_\ell(i) + \text{terms independent of } \theta_\ell(i) \qquad (47.1017)$$

For example, if we apply this result to the bias vector θ_2 defined earlier in (47.961), we find that the corresponding gradient calculations lead to (where, again with some abuse of notation, we are writing $\partial\|\gamma - y\|^2/\partial\theta_2$ to refer to the resulting vector):

$$\frac{\partial\|\gamma - y\|^2}{\partial\theta_2} = \begin{bmatrix} -\delta_3(1) \\ -\delta_3(2) \end{bmatrix} \tag{47.1018}$$

More generally, we have

$$\boxed{\frac{\partial\|\gamma - y\|^2}{\partial\theta_\ell} = -\delta_{\ell+1}} \tag{47.1019}$$

so that from (47.979), , and after restoring the subscript n,

$$\frac{\partial J_{\rm emp}(W,\theta)}{\partial\theta_\ell} = -\frac{1}{N}\sum_{n=0}^{N-1}\delta_{\ell+1,n} \tag{47.1020}$$

In summary, we arrive at the following listing for the main step involved in computing the desired partial derivatives of the risk function, $J_{\rm emp}(W,\theta)$, relative to all combination matrices and all bias vectors in a feedforward network consisting of L layers. In the description below, we reinstate the subscript n to refer to the time or sample index. Moreover, the quantities $\{y_{\ell,n}, \delta_{\ell,n}, z_{\ell,n}\}$ are all vectors associated with layer ℓ.

Computation of partial derivatives

```
start with training data $\{\gamma_n, h_n\}, n = 0,1,\ldots,N-1$.
repeat for $n = 0,1,\ldots,N-1$ :
  (forward pass)
    feed $h_n$ into the network and compute $\{z_{\ell,n}, y_{\ell,n}, z_n, y_n\}$ using (47.968).
    compute terminal sensitivity vector: $\delta_{L,n} = 2(y_n - \gamma_n)\odot f'(z_n)$.
  (backward pass)
    compute $\delta_{\ell,n} = f'(z_{\ell,n})\odot\left(W_\ell^{\mathsf T}\delta_{\ell+1,n}\right),\quad \ell = L-1,\ldots,3,2$        (47.1021)
end
(derivatives) compute for $\ell = 1,2,\ldots,L-1$ :
    $\frac{\partial J_{\rm emp}(W,\theta)}{\partial W_\ell} = 2\rho W_\ell + \frac{1}{N}\sum_{n=0}^{N-1}\delta_{\ell+1,n}y_{\ell,n}^{\mathsf T}$
    $\frac{\partial J_{\rm emp}(W,\theta)}{\partial\theta_\ell} = -\frac{1}{N}\sum_{n=0}^{N-1}\delta_{\ell+1,n}$
```

Observe that the sensitivity vector $\delta_{1,n}$ is not needed and is not evaluated during the backward pass. For consistency, if we were to evaluate its value, we would first note that layer $\ell = 1$ does not contain activation functions (i.e., $f(x) = x$ for this layer) and, moreover, $z_{1,n} = h_n$. Therefore, we can set $f'(z_{1,n}) = 0$ and $\delta_{1,n} = 0$. With this understanding, the evaluation of the partial derivatives of $J_{\rm emp}(W,\theta)$ relative to the individual entries of W and θ can be performed more efficiently, and recursively, by incorporating it into the backward step. Specifically, assume we introduce auxiliary variables $\{D_\ell, d_\ell\}$ whose dimensions match those of $\{W_\ell, \theta_\ell\}$ and with initial values:

$$D_\ell = 2\rho W_\ell, \quad d_\ell = 0_{n_\ell+1} \tag{47.1022}$$

Then, we can enlarge the backward step in (47.1021) to include the following calculations:

```
Enlarged backward step
for $\ell = L-1, \ldots, 3, 2, 1$
    $\delta_{\ell,n} = f'(z_{\ell,n}) \odot \left(W_\ell^{\mathsf{T}} \delta_{\ell+1,n}\right)$
    $D_\ell = D_\ell + \frac{1}{N}\delta_{\ell+1,n} y_{\ell,n}^{\mathsf{T}}$
    $d_\ell = d_\ell - \frac{1}{N}\delta_{\ell+1,n}$
end
$\partial J_{\rm emp}(W,\theta)/\partial W_\ell = D_\ell$
$\partial J_{\rm emp}(W,\theta)/\partial \theta_\ell = d_\ell$
```
(47.1023)

47.23.5 Stochastic-Gradient Training

We can now employ the backpropagation algorithm to train a feedforward neural network by employing a stochastic-gradient implementation with step-size $\mu > 0$. In this implementation, one data point (γ_n, h_n) is used per iteration. We attach a subscript n to the combination matrices and bias vectors, which will now be adjusted, and denote them by $W_{\ell,n}$ and $\theta_{\ell,n}$ at iteration n.

Initialization

At step $n = -1$, it is customary to select the bias coefficients $\{\theta_{\ell,-1}(i)\}$ randomly by following a Gaussian distribution with zero mean and variance one, i.e.,

$$\boldsymbol{\theta}_{\ell,-1}(i) \sim \mathcal{N}(0,1) \tag{47.1024}$$

The combination weights $\{w_{ij,-1}^{(\ell)}\}$ are also selected randomly according to a Gaussian distribution but one whose variance is adjusted in accordance with the number of nodes in layer ℓ, which we denoted earlier by n_ℓ. Specifically, it is customary to normalize the variance of the Gaussian distribution by $\sqrt{n_\ell}$, i.e.,

$$\boldsymbol{w}_{ij,-1}^{(\ell)} \sim \mathcal{N}\left(0, 1/\sqrt{n_\ell}\,\right) \tag{47.1025}$$

The reason for this normalization is to limit the variance of the signals in the subsequent layer $\ell+1$. Indeed, for an arbitrary node j in layer $\ell+1$, its pre-activation signal is given by

$$\boldsymbol{z}_{\ell+1,-1}(j) \;=\; \sum_{i=1}^{n_\ell} \boldsymbol{w}_{ij,-1}^{(\ell)} \boldsymbol{y}_{\ell,-1}(i) \;-\; \boldsymbol{\theta}_{\ell,-1}(j) \tag{47.1026}$$

If we assume, for illustration purposes, that the output signals, $\boldsymbol{y}_{\ell,-1}(i)$, from layer ℓ, are independent and have uniform variance, σ_y^2, it then follows that the variance of $\boldsymbol{z}_{\ell+1,-1}(j)$, denoted by σ_z^2, is given by

$$\sigma_z^2 \;=\; 1 + \sum_{i=1}^{n_\ell}\left(\frac{1}{\sqrt{n_\ell}}\right)^2 \sigma_y^2 \;=\; 1 + \sigma_y^2, \qquad \text{(with normalization)} \tag{47.1027}$$

Without normalization of the variance of the initial weights by $\sqrt{n_\ell}$, the above variance would instead be given by

$$\sigma_z^2 \;=\; 1 + n_\ell \sigma_y^2, \qquad \text{(without normalization)} \tag{47.1028}$$

which grows linearly with n_ℓ. This means that the pre-activation signal is likely to assume large (negative or positive) values when n_ℓ is large, which in turn means that their activation functions are likely to be saturated. This fact ends up slowing the learning process in the network because small changes in internal signals (or weights) have little effect on the output of saturated nodes and on subsequent layers. Other forms of initialization include selecting the weight variables by uniformly sampling within the following ranges:

$$\boldsymbol{w}_{ij,-1}^{(\ell)} \in \left[-\frac{1}{\sqrt{n_\ell}},\ \frac{1}{\sqrt{n_\ell}}\right] \tag{47.1029a}$$

$$\boldsymbol{w}_{ij,-1}^{(\ell)} \in \left[-\frac{\sqrt{6}}{\sqrt{n_{\ell+1}+n_\ell}},\ \frac{\sqrt{6}}{\sqrt{n_{\ell+1}+n_\ell}}\right], \quad \text{(for } \tanh \text{ activation)} \tag{47.1029b}$$

$$\boldsymbol{w}_{ij,-1}^{(\ell)} \in \left[-\frac{4\sqrt{6}}{\sqrt{n_{\ell+1}+n_\ell}},\ \frac{4\sqrt{6}}{\sqrt{n_{\ell+1}+n_\ell}}\right], \quad \text{(for sigmoid activation)} \tag{47.1029c}$$

where the last two intervals employ the sizes of the pre- and post-layers for the weights $\boldsymbol{w}_{ij}^{(\ell)}$. These choices are meant to ensure that during the initial stages of training, information can flow reliably forward and backward in the network away from saturation and the difficulties caused by the vanishing gradient problem (discussed later in Example 47.68).

Listing of Algorithms

Combining a stochastic-gradient step with the backpropagation algorithm (47.1021), we arrive at the following listing for a stochastic-gradient procedure for training the neural network. If desired, a different step-size value, μ_{ab}, can be used for every $(a,b)-$th entry of the weight matrices W_ℓ; likewise, for the bias coefficients.

Stochastic-gradient backpropagation algorithm

```
for each training point $(\gamma_n, h_n), n = 0, 1, \ldots, N-1$ :
  feed $h_n$ into network $\{W_{\ell,n-1}, \theta_{\ell,n-1}\}$; compute $\{z_{\ell,n}, y_{\ell,n}\}$ for all layers using (47.968).
  compute $\delta_{L,n} = 2(y_{L,n} - \gamma_n) \odot f'(z_{L,n})$.
  repeat for $\ell = L-1, \ldots, 3, 2, 1$ :
    $\delta_{\ell,n} = f'(z_{\ell,n}) \odot \left(W_{\ell,n-1}^{\mathsf{T}} \delta_{\ell+1,n}\right)$
    $W_{\ell,n} = (1 - 2\mu\rho) W_{\ell,n-1} \ - \ \mu \delta_{\ell+1,n} y_{\ell,n}^{\mathsf{T}}$
    $\theta_{\ell,n} = \theta_{\ell,n-1} + \mu \delta_{\ell+1,n}$
  end
end
```

(47.1030)

We can also train the feedforward neural networks by employing a mini-batch implementation, where a collection of B samples $\{\gamma_m, h_m\}$ are used at each iteration n to perform the gradient updates — recall (47.540). The batch samples can be chosen in various ways, e.g., as the most recent B samples in a streaming implementation or randomly from within the entire N samples. Again, if desired, a different step-size value, μ_{ab}, can be used for every $(a,b)-$th entry of the weight matrices W_ℓ; likewise, for the bias coefficients.

Mini-batch backpropagation algorithm

```
for each batch of $B$ training samples $\{\gamma_m, h_m\}$ :
    feed each $h_m$ into network $\{W_{\ell,n-1}, \theta_{\ell,n-1}\}$ and compute $\{z_{\ell,m}, y_{\ell,m}\}$ using (47.968).
    compute $\delta_{L,m} = 2(y_{L,m} - \gamma_m) \odot f'(z_{L,m})$.
    repeat for $\ell = L-1, \ldots, 3, 2, 1$ :
        $\delta_{\ell,m} = f'(z_{\ell,m}) \odot \left(W_{\ell,n-1}^{\mathsf{T}} \delta_{\ell+1,m}\right)$
        $W_{\ell,n} = (1-2\mu\rho)W_{\ell,n-1} \;-\; \mu \frac{1}{B}\sum_{m=0}^{B-1} \delta_{\ell+1,m} y_{\ell,m}^{\mathsf{T}}$
        $\theta_{\ell,n} = \theta_{\ell,n-1} + \mu \frac{1}{B}\sum_{m=0}^{B-1} \delta_{\ell+1,m}$
    end
end
```

(47.1031)

Example 47.65 (Application: Autoencoders)

The term "autoencoder" refers to a three-layer network (consisting of one input layer, one hidden layer, and one output layer) whose objective is to map the input space back into itself. This is achieved by constructing a feedforward network with the same number of output nodes as the number of input nodes and by setting γ_n equal to the feature vector, i.e.,

$$Q = M, \quad \gamma_n = h_n \tag{47.1032}$$

If the individual entries of h_n are assumed to lie within the intervals $(0,1)$ or $(-1,1)$, then the neurons in the output layer will include sigmoidal or hyperbolic-tangent nonlinearities so that their output signals will also lie within the same interval. If, on the other hand, the individual entries of h_n are arbitrary real numbers, then the neurons in the output layer will not include nonlinearities. For illustration purposes, we assume the entries of h_n lie within $(0,1)$ and consider an autoencoder structure of the form shown in Fig. 47.76 where the output neurons contain sigmoidal nonlinearities.

Now, by applying the training algorithm in any of its forms, e.g., in the stochastic-gradient form (47.1030) or in the mini-batch form (47.1031), we end up with an *unsupervised* learning procedure that trains the network to learn how to map h_n into itself (i.e., it learns how to recreate the input data). This situation is illustrated in Fig. 47.76, which shows an autoencoder trained by using h_n both as the feature vector and as the variable γ_n. The hidden layer in this example has three nodes. If we denote their outputs by

$$\{y_2(1),\, y_2(2),\, y_2(3)\} \tag{47.1033}$$

then the network will be mapping these three signals through the output layer back into good approximations for the five entries of the input data. This step, when successful, amounts to a form of data compression since it essentially shows that the three hidden signals are sufficient to reproduce the five input signals.

Recall that we denote the post-activation output vector of the hidden layer by $y_2 \in \mathbb{R}^{n_2}$, the weight matrix between the input layer and the hidden layer by $W_1 \in \mathbb{R}^{n_2 \times M}$, and the bias vector feeding into the hidden layer by θ_1. Then, using these symbols, the auto-encoder effectively determines a representation y_2 for h_n in the form:

$$y_2 \;=\; f\left(W_1 h_n - \theta_1\right), \quad \text{(encoding)} \tag{47.1034}$$

where $f(\cdot)$ denotes the activation function. Moreover, since the autoencoder is trained to recreate h_n, then $y_3 = \widehat{h}_n$ and we find that the representation y_3 is mapped back to the original feature vector

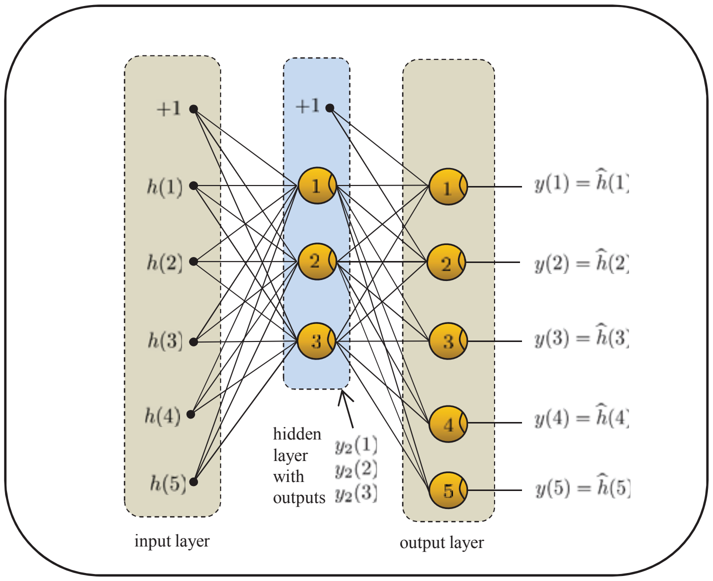

FIGURE 47.76 An autoencoder with a single hidden layer. The network is trained to map the feature vectors back to themselves using, for example, either the stochastic-gradient algorithm (47.1030) or the mini-batch implementation (47.1031) .

by using

$$\widehat{h}_n \;=\; f\left(W_2 y_2 - \theta_2\right), \quad \text{(decoding)} \tag{47.1035}$$

in terms of the bias vector, θ_2, for the output layer and the weight matrix $W_2 \in \mathbb{R}^{M\times n_2}$ between the hidden layer and the output layer. We refer to these two transformations as "encoding" and "decoding" steps. In this way, autoencoders provide a useful structure to encode the data, i.e., to learn a compressed representation for the data and to perform dimensionality reduction. The reduced representation will generally extract and reflect the most significant features present in the data. For example, if the input data happens to be a raw signal, then the autoencoder can function as an effective feature extraction/detection module. Compact representations of this type are also useful in reducing the possibility of over-fitting in learning algorithms. This is because the reduced features can be used to drive the algorithms, which would then attempt to perform classification based on less complex models.

One can also consider designing autoencoders where the number of hidden units is larger than the dimension of the input vector, i.e., $n_2 \geq M$. Obviously, this case will not lead to dimensionality reduction. However, it has been observed that such "overcomplete" representations can still be useful in that the features they produce can lead to reduced classification errors. For such situations, and in order to prevent the autoencoder from mapping the input vector back to itself at the hidden layer (i.e., in order to prevent the autoencoder from learning the identity mapping), a variation of the autoencoder structure is used; it is known as a "denoising" autoencoder. Here, a fraction of the input entries in h_n are randomly set to zero (possibly as many as 50% of them). The perturbed input vector, denoted by h_n', is applied to the input of the autoencoder while the original vector h_n continues to be used as the reference, γ_n. By doing so, the autoencoder ends up performing two tasks: encoding the input data and predicting the missing entries (i.e., countering the effect of the corruption). In order to succeed at the second task, the autoencoder ends up being endowed with a useful capability: it will need to learn the correlations among the entries of the input vector in order to be able to "recover" the missing entries.

◇

Example 47.66 (Dropout strategy)

We described in Sec. 47.21 the bagging technique for enhancing the performance of classifiers. The same technique can be applied to neural networks. As explained in that section, bagging is based on the idea of training multiple networks, using the *same* data set, and on combining their classification decisions (e.g., by taking a majority vote). In order to ensure variability across the networks, the training data for each network is generated randomly by sampling the original training data *with* replacement. The bagging procedure can be expensive for neural networks with many hidden layers or many nodes since it involves training a large number of combination weights multiple times. A powerful alternative is to employ the *dropout* method to emulate the behavior of training multiple networks albeit at a considerably lower cost. The dropout method operates as follows.

We refer to the stochastic-gradient backpropagation algorithm (47.1030) or its mini-batch form (47.1031). These algorithms train a feedforward neural network with L layers; we describe the dropout method by assuming the mini-batch implementation; the same description applies to the stochastic-gradient implementation.

At every iteration n, a new mini-batch of data $\{\gamma_m, h_m\}$ of size B is available. Before using this data to train the network, the connectivity structure of the network is modified in a random manner by switching a large portion of the nodes into a sleeping mode. Specifically, every node is turned off with probability $1/2$. When a node is turned off, its links to all other nodes are de-activated and its combination weights and bias coefficient are frozen for future use but are not used during this particular training stage. The remaining active nodes in the network operate as if these sleeping nodes do not exist. The ultimate effect is that the size of the original network is halved. The training is then performed on the active nodes by adjusting their combination weights and bias coefficients using the same recursions from the mini-batch implementation (47.1031). At the next time instant, $n + 1$, the process is repeated. Again, a collection of nodes is randomly turned off and the new mini-batch of data is used to train the remaining network.

Once training is completed, all nodes in the network are activated. However, their combination weights and bias coefficients are halved in order to account for the fact that these weights were determined with only half of the nodes in the network active. One of the main advantages for using this dropout procedure is that the performance of the resulting network can be viewed as the averaged performance across many random network realizations. By doing so, the possibility of over-fitting is reduced, especially in large network implementations.

◇

Example 47.67 (Numerical checking of gradient calculations)

It is apparent that the computation of the gradient quantities in the backpropagation implementation is less trivial than usual. It is not uncommon for software implementations of the procedure to suffer from subtle errors in the code. These errors are generally difficult to debug and can go unnoticed. One way to check the validity of the gradient calculations is by evaluating them numerically in an alternative manner and then comparing the result with the one that follows from the derived expressions. If the values are close to each other, then one gains confidence in the validity of the implementation. We illustrate this procedure for the stochastic-gradient implementation (47.1030). In this implementation, the gradient quantities that are need to perform the updates for $W_{\ell,n}$ and $\theta_{\ell,n}$ are mainly (recall (47.1013) and (47.1019)):

$$\frac{\partial\|\gamma_n - y_n\|^2}{\partial W_\ell} = \delta_{\ell+1,n} y_{\ell,n}^{\mathsf{T}} \tag{47.1036}$$

$$\frac{\partial\|\gamma_n - y_n\|^2}{\partial \theta_\ell} = -\delta_{\ell+1,n} \tag{47.1037}$$

Let us focus on the gradient calculation relative to the combination weights; the same argument applies to the bias weights. For convenience, we denote the cost that is being differentiated by $C(W,\theta) = \|\gamma_n - y_n\|^2$.

Let us pick one of the weights in W_ℓ, say, $w_{ij}^{(\ell)}$, and perturb it by $\pm\epsilon$, where ϵ is a small positive number, say, $\epsilon = 10^{-5}$ or smaller at $\epsilon = 10^{-8}$. We keep all other combination weights and bias weights fixed in the network. By perturbing $w_{ij}^{(\ell)}$, the value of y_n will be perturbed and likewise the value of the cost C. We evaluate the cost at these perturbed weights and approximate the gradient relative to $w_{ij}^{(\ell)}$ by using the finite difference expression:

$$\frac{\partial C(W,\theta)}{\partial w_{ij}^{(\ell)}} \approx \frac{1}{2\epsilon}\left[C\left(w_{ij}^{(\ell)}+\epsilon\right) \;-\; C\left(w_{ij}^{(\ell)}-\epsilon\right)\right] \tag{47.1038}$$

We subsequently compare this numerical value with the $(i,j)-$entry in the matrix $\delta_{\ell+1,n}y_{\ell,n}^{\mathsf{T}}$. If the magnitude of the difference is small, then this verification would pass the test. We repeat this same procedure for all combination weights and bias coefficients in the network.

◇

Example 47.68 (Slow down in learning)

The backward recursion for updating the sensitivity vector, $\delta_{\ell,n}$, highlights one difficulty that arises in the training of feedforward networks with a *large* number of hidden layers (i.e., deep networks, which typically have at least three hidden layers). This is because the flow of information by the backpropagation procedure to the earlier layers becomes less significant and can be hindered by saturation effects that cause gradient values to become small or negligible.

Consider an arbitrary combination weight, $w_{ij}^{(\ell)}$, in some early layer of index ℓ. It is clear from the recursion for $\delta_{\ell,n}$ that, starting from the terminal vector $\delta_{L,n}$ and propagating it backwards, the entries in $\delta_{\ell,n}$ will involve a product of derivatives of the activation function, $f'(\cdot)$, at successive output signals, namely,

$$\text{entries of } \delta_{\ell,n} \;\propto\; f'\left(z_{\ell,n}\right)\, f'\left(z_{\ell+1,n}\right)\ldots f'\left(z_{L,n}\right) \tag{47.1039}$$

For the sigmoid and hyperbolic tangent functions listed in Table 47.11 we have

$$\begin{cases} f'(x) = f(x)(1-f(x)), & 0 \le f'(x) \le 1/4 \quad \text{(sigmoid)} \\ f'(x) = 1/\cosh(x), & 0 \le f'(x) \le 1 \quad\;\;\;\, \text{(tanh)} \end{cases} \tag{47.1040}$$

It is seen that products of a large collection of derivative values for these activation functions can result in a small number, especially when the nodes in the output layer are close to saturation in which case $f'\left(z_{L,n}\right)$ will already be close to zero. This problem, known as the *vanishing gradient problem*, is more pronounced for the sigmoid activation function since its derivative function is bounded by $1/4$ rather than one. In this way, the entries of $\delta_{\ell,n}$ will become relatively small and, consequently, the updates to $W_{\ell,n-1}$ and $\theta_{\ell,n-1}$ in algorithm (47.1030) will be slow. Observe that this effect is magnified for the earlier layers in the network due to the backward nature of the recursion for $\delta_{\ell,n}$: as we move further back in the layers, more terms are included in the product (47.1039) and it will become more likely for the product to assume small values. This means that earlier layers in the network will end up learning at a relatively slower rate compared to the later layers.

For similar reasons, the learning process in the network is also slowed down when some nodes are saturated. This is because the derivatives of the corresponding activation functions will be close to zero, and the product (47.1039) will again be close to zero. As already indicated, saturation is more likely to occur at the output nodes, thus leading to small values for $f'(z_{L,n})$. These observations help explain why other choices for the activation function are considered, such as the rectifier function from Table 47.11, although this function still suffers from the problem of turning off learning for negative values of its arguments x. These observations also motivate considering adjustments to the network structure or the risk function, as we discuss next, in order to eliminate the effect of the derivatives of the activation functions at the output nodes.

◇

47.23.6 Softmax Training

One useful option to address the slowdown in learning discussed in the last example involves modifying the risk function or modifying the network structure itself. Both these modifications are done with the intent of arriving at a backward recursion for the variable $\delta_{\ell,n}$ that would be independent of the derivatives of the activation functions at the output nodes. By doing so, the resulting training algorithms will be less prune to slowdown in its learning process. We pursue the details of these modifications in Probs. 47.159, 47.160, and 47.162, where alternatives to the least-squares risk (47.979) are considered including the cross-entropy risk and the logistic risk, as well as an alternative network structure in the form of a softmax implementation. In this section and the next we explain the motivation behind the softmax and cross-entropy formulations.

In the softmax formulation, the empirical risk continues to be the least-squares risk (47.979) except that the structure of the output layer in the network is modified in one important manner. The activation functions in this layer are replaced by the following normalization step:

$$y_n(q) \;\overset{\Delta}{=}\; e^{z_n(q)}\left(\sum_{k=1}^{Q} e^{z_n(k)}\right)^{-1} \tag{47.1041}$$

Here, the variable $z_n(q)$ refers to the $q-$th entry of the output vector z_n prior to activation and $y_n(q)$ refers to the $q-$th entry at the output of the network; this latter entry is now being computed through the above softmax operation and not by feeding $z_n(q)$ through an activation function. Observe that $y_n(q)$ is now influenced by other signals $\{z_n(k)\}$ and not only by $z_n(q)$. The normalization in (47.1041) ensures that the output variables $\{y_n(q)\}$ are all nonnegative and add up to one. In this way, the softmax transformation allows us to interpret the $\{y_n(q)\}$ as probability measures (actually, as a Gibbs distribution). For example, in a multiclass classification problem, each $y_n(q)$ can now be interpreted as corresponding to the likelihood that the feature vector h_n belongs to class q. We note in passing that we also encounter the softmax operation in the context of multinomial logistic regression problems — studied in Prob. 47.134 and expression (47.1528).

In Prob. 47.160 we confirm that the listings of the stochastic-gradient and mini-batch backpropagation algorithms (47.1030) and (47.1031) remain intact except for the expression for the terminal sensitivity vector, $\delta_{L,n}$, which is replaced by

$$\begin{aligned} x &\overset{\Delta}{=} 2(y_{L,n}-\gamma_n)\odot y_{L,n} && \text{(47.1042a)}\\ \delta_{L,n} &= x \;-\; \left(\mathbb{1}^{\mathsf T}x\right) y_{L,n} && \text{(47.1042b)} \end{aligned}$$

This expression can be motivated as follows. If we refer to expressions (47.1004) and (47.1005), where the subscript n has been dropped, we have

$$\begin{aligned} \delta_L(j) &= \sum_{k=1}^{Q} \frac{\partial\|\gamma-y\|^2}{\partial y(k)}\,\frac{\partial y(k)}{\partial z(j)}\\ &= \sum_{k=1}^{Q} 2(y(k)-\gamma(k))\,\frac{\partial y(k)}{\partial z(j)} \end{aligned} \tag{47.1043}$$

where now, in view of the normalization (47.1041):

$$\frac{\partial y(k)}{\partial z(j)} = \begin{cases} -y(j)y(k), & k\neq j\\ (1-y(k))y(k), & k=j \end{cases} \tag{47.1044}$$

Substituting into (47.1043) gives

$$\begin{aligned}\delta_L(j) &= 2(y(j)-\gamma(j))y(j)(1-y(j)) - \sum_{k\neq j}^{Q} 2(y(k)-\gamma(k))y(j)y(k) \\ &= 2(y(j)-\gamma(j))y(j) - \left(\sum_{k=1}^{Q} 2(y(k)-\gamma(k))y(k)\right) y(j) \qquad (47.1045)\end{aligned}$$

which justifies expression (47.1042b). Alternatively, we can rewrite (47.1044) more compactly by using the Kronecker delta notation as

$$\frac{\partial y(k)}{\partial z(j)} = (\delta_{jk} - y(j))y(k) \qquad (47.1046)$$

where the notation δ_{jk} is used here to denote the Kronecker delta sequence, namely, $\delta_{jk} = 1$ when $j = k$ and is zero otherwise. We further collect these partial derivatives into a $Q \times Q$ symmetric matrix:

$$[J]_{jk} \overset{\Delta}{=} (\delta_{jk} - y(j))y(k) \qquad (47.1047)$$

That is, for $Q = 3$:

$$J = \begin{bmatrix} (1-y(1))y(1) & -y(1)y(2) & -y(1)y(3) \\ -y(2)y(1) & (1-y(2))y(2) & -y(2)y(3) \\ -y(3)y(1) & -y(3)y(2) & (1-y(3))y(3) \end{bmatrix} \qquad (47.1048)$$

Then, we can also express the terminal vector $\delta_{L,n}$ in the matrix-vector product (after restoring the subscript n):

$$\delta_{L,n} = 2J(y_{L,n} - \gamma_n) \qquad (47.1049)$$

Accordingly, the listing of the stochastic-gradient backpropagation algorithm will become the following (similarly for the mini-batch version). Different step-size values, μ_{ab}, can also be used for different entries of the weight matrices W_ℓ; likewise, for the bias coefficients.

Softmax stochastic-gradient backpropagation algorithm

```
**for each training point** $(\gamma_n, h_n)$ :
  feed $h_n$ into network $\{W_{\ell,n-1}, \theta_{\ell,n-1}\}$; compute $\{z_{\ell,n}, y_{\ell,n}\}$ for all layers using (47.968).
  compute $\delta_{L,n} = 2J(y_{L,n} - \gamma_n)$.
  **repeat for** $\ell = L-1, \ldots, 3, 2, 1$ :
    $\delta_{\ell,n} = f'\left(z_n^{(\ell)}\right) \odot \left(W_{\ell,n-1}^{\mathsf{T}} \delta_{\ell+1,n}\right)$
    $W_{\ell,n} = (1-2\mu\rho)W_{\ell,n-1} - \mu\delta_{\ell+1,n} y_{\ell,n}^{\mathsf{T}}$
    $\theta_{\ell,n} = \theta_{\ell,n-1} + \mu\delta_{\ell+1,n}$
  **end**
**end**
```

(47.1050)

47.23.7 Cross-Entropy Training

In the cross-entropy formulation, the structure of the neural network remains intact with *sigmoid* activation functions present at all nodes. However, the least-squares empirical risk (47.979) is replaced by

$$J_{\rm emp}(W,\theta) \;\overset{\Delta}{=}\; \sum_{\ell=1}^{L-1} \rho\|W_\ell\|_{\rm F}^2 \;-\; \sum_{n=0}^{N-1}\sum_{q=1}^{Q} \ln\left(y_n(q)^{\gamma_n(q)}(1-y_n(q))^{(1-\gamma_n(q))}\right) \tag{47.1051}$$

where $0 \le \gamma_n(q) \le 1$ and $0 < y_n(q) < 1$ denote the $q-$th entries of the vectors γ_n and y_n, and assumed to lie in the indicated ranges. For example, the condition on $y_n(q)$ is satisfied when the sigmoidal activation function of Table 47.11 is employed or when the softmax transformation (47.1041) is used. In Prob. 47.159 we examine how the backpropagation equations are modified for sigmoidal activation functions and show that the expression for $\delta_{\ell,n}$, for all layers, becomes *independent* of the derivatives of the activation function. Likewise, in Prob. 47.161 we consider an extension for multiclass classification problems where the $\gamma_n(q)$ are not limited to the range $[0,1]$ and replace the above risk function by one of the form:

$$J_{\rm emp}(W,\theta) \;\overset{\Delta}{=}\; \sum_{\ell=1}^{L-1} \rho\|W_\ell\|_{\rm F}^2 \;-\; \sum_{n=0}^{N-1}\sum_{q=1}^{Q} \gamma_n(q)\ln\left(y_n(q)\right) \tag{47.1052}$$

where the output signals are further assumed to be generated via the softmax transformation. We continue with (47.1051) without loss in generality and leave the extension to the problems.

The motivation for the risk function (47.1051) can be explained by referring to the concept of the cross-entropy between two probability distributions. Thus, consider two discrete probability density functions, denoted generically by $p_{\boldsymbol{x}}(x)$ and $q_{\boldsymbol{x}}(x)$, for a random variable $\boldsymbol{x}$. Their cross-entropy is denoted by $H(p,q)$ and is defined as

$$\begin{aligned} H(p,q) \;&\overset{\Delta}{=}\; -\sum_x p_{\boldsymbol{x}}(x)\log_2\left(q_{\boldsymbol{x}}(x)\right) \\ &=\; -\sum_x \mathbb{P}_p(\boldsymbol{x}=x)\log_2 \mathbb{P}_q(\boldsymbol{x}=x) \end{aligned} \tag{47.1053}$$

where the sum is over the discrete realizations of the random variable $\boldsymbol{x}$, and the notation $\mathbb{P}_p(\boldsymbol{x}=x)$ denotes the probability that $\boldsymbol{x}$ assumes the value x under distribution $p(x)$. Similarly, for $\mathbb{P}_q(\boldsymbol{x}=x)$. Using the earlier definition (47.700), with the natural logarithm replaced by the logarithm relative to base 2, it is straightforward to verify that the cross-entropy is, apart from an offset value, equal to the Kullback-Leibler divergence measure between the two distributions, namely,

$$H(p,q) \;=\; H(p) + D_{\rm KL}(p,q) \tag{47.1054}$$

in terms of the entropy of the distribution $p_{\boldsymbol{x}}(x)$:

$$H(p) \;=\; -\sum_x \mathbb{P}_p(\boldsymbol{x}=x)\log_2 \mathbb{P}_p(\boldsymbol{x}=x) \tag{47.1055}$$

In this way, the cross-entropy between two distributions is in effect a measure of how close these distributions are to each other.

To illustrate the relevance of this measure in the context of classification problems, let us consider a scalar class variable $\boldsymbol{\gamma} \in \{0,1\}$ (rather than $\boldsymbol{\gamma} = \pm 1$). Let $\boldsymbol{y}$ denote the output signal that is generated by the neural network, assumed to satisfy $\boldsymbol{y} \in (0,1)$; this condition is automatically satisfied under the sigmoid activation function. Then, we can associate two distributions with the random variable $\boldsymbol{\gamma}$ as follows:

$$\begin{aligned}\mathbb{P}_p(\boldsymbol{\gamma}=+1) &= \gamma, \quad \mathbb{P}_p(\boldsymbol{\gamma}=0)=1-\gamma && (47.1056a)\\ \mathbb{P}_q(\boldsymbol{\gamma}=+1) &= y, \quad \mathbb{P}_q(\boldsymbol{\gamma}=0)=1-y && (47.1056b)\end{aligned}$$

Ideally, we would like the distribution that results for $\boldsymbol{y}$ to match the distribution for $\boldsymbol{\gamma}$. The cross-entropy between these two distributions is given by

$$\begin{aligned}H(p,q) &= -\gamma\log_2(y)-(1-\gamma)\log_2(1-y)\\ &= -\frac{1}{\ln 2}\left(\ln y^{\gamma}(1-y)^{(1-\gamma)}\right) && (47.1057)\end{aligned}$$

This expression is the motivation for the rightmost term in (47.1051), where the sums are over all training data and over the entries $(\boldsymbol{\gamma},\boldsymbol{y})$ in the vector case. In Prob. 47.159, we show that the listings of the stochastic-gradient and mini-batch backpropagation algorithms (47.1030) and (47.1031) remain intact except for the expression for $\delta_{L,n}$, which is replaced by

$$\delta_{L,n} \;=\; y_{L,n}-\gamma_n \qquad (47.1058)$$

For example, the listing of the mini-batch backpropagation algorithm will become the following. Different step-size values, μ_{ab}, can also be used for different entries of the weight matrices W_ℓ; likewise, for the bias coefficients.

Cross-entropy mini-batch backpropagation algorithm

```
for each batch of $B$ training samples $\{\gamma_m, h_m\}$ :
    feed each $h_m$ into network $\{W_{\ell,n-1},\theta_{\ell,n-1}\}$ and compute $\{z_{\ell,m},y_{\ell,m}\}$ using (47.968).
    compute $\delta_{L,m}=y_{L,m}-\gamma_m$.
    repeat for $\ell=L-1,\ldots,3,2,1$ :
        $\delta_{\ell,m}=f'(z_{\ell,m})\odot\left(W^{\mathsf{T}}_{\ell,n-1}\delta_{\ell+1,m}\right)$
        $W_{\ell,n}=(1-2\mu\rho)W_{\ell,n-1} - \mu\frac{1}{B}\sum_{m=0}^{B-1}\delta_{\ell+1,m}y^{\mathsf{T}}_{\ell,m}$
        $\theta_{\ell,n}=\theta_{\ell,n-1}+\mu\frac{1}{B}\sum_{m=0}^{B-1}\delta_{\ell+1,m}$
    end
end
```

(47.1059)