65 FEEDFORWARD NEURAL NETWORKS

We illustrated in Example 63.2 one limitation of linear separation surfaces by considering the XOR mapping (63.11). The example showed that certain feature spaces are not linearly separable and cannot be resolved by the Perceptron algorithm. The result in the example was used to motivate one powerful approach to nonlinear separation surfaces by means of kernel methods. In this chapter we describe a second powerful and popular method, based on training *feedforward neural networks*. These are also called multilayer Perceptrons and even *deep networks*, depending on the size and number of their layers. We revisit the XOR mapping in Prob. 65.1 and show how a simple network of this type can separate the features.

Feedforward neural networks are layered structures of interconnected units called *neurons*, each of which will be a modified version of the Perceptron. A neural network will consist of:

(a) one input layer, where the feature vector, $h \in \mathbb{R}^M$, is applied to;

(b) one output layer, where the predicted label, now represented by a vector $\widehat{\gamma} \in \mathbb{R}^Q$, is read from, and

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

The net effect is a nonlinear mapping from the input space, h, to the output space $\widehat{\gamma}$:

$$h \in \mathbb{R}^M \longrightarrow \widehat{\gamma} \in \mathbb{R}^Q \tag{65.1}$$

We will describe several iterative procedures for learning the internal parameters of this mapping from training data $\{\gamma_n, h_n\}$. In future chapters, we will describe alternative neural network architectures and their respective training algorithms, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and long short-term memory (LSTMs) networks, where some parameters are shared across nodes and/or layers.

Besides the ability of feedforward neural networks to model nonlinear mappings from h to $\widehat{\gamma}$, one other notable difference in relation to the learning algorithms considered so far in the text is that neural networks can deal with classification problems where the *scalar* binary class variable $\gamma \in \{+1, -1\}$ is replaced by a *vector* class variable $\gamma \in \{+1, -1\}^Q$ whose entries are again binary. This level of generality allows us to solve *multiclass* and *multilabel* classifica-

tion problems directly without the need to resort to one-versus-all (OvA) or one-versus-one (OvO) constructions.

For instance, in multiclass problems, the feature vector h can belong to one of a collection of Q classes (such as deciding whether h represents cats, dogs, or elephants), and the label vector γ will contain $+1$ in the location corresponding to the correct class and -1 in the remaining entries. In multilabel classification problems, on the other hand, the feature vector h can reveal several properties simultaneously (such as representing a male individual with high-school education who likes mystery movies). In this case, the entries of γ corresponding to these properties will be $+1$ while the remaining entries will be -1, for example,

$$\gamma = \begin{bmatrix} -1 \\ -1 \\ -1 \\ \boxed{+1} \\ -1 \end{bmatrix} \ (\textbf{multiclass problem}), \quad \gamma = \begin{bmatrix} \boxed{+1} \\ -1 \\ -1 \\ \boxed{+1} \\ \boxed{+1} \end{bmatrix} \ (\textbf{multilabel problem}) \tag{65.2}$$

In one of the most common implementations of neural networks, information flows forward from the input layer into the 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* or recursive structures, where signals from later layers feed back into neurons in earlier layers. We will encounter examples of these in the form of recurrent neural networks (RNNs) and long short-term memory (LSTMs) networks in a future chapter. We focus here on feedforward structures, which are widely used in applications; they also exhibit a universal approximation ability as explained in the comments at the end of the chapter — see expression (65.189).

65.1 ACTIVATION FUNCTIONS

The basic unit in a neural network is the neuron shown in Fig. 65.1 (*left*). 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 $-\theta$, which represents an offset parameter. The coefficients for the remaining multipliers are denoted by $w(m)$ and their respective inputs by $h(m)$. If we collect the input and scaling coefficients into column vectors:

$$h = \mathrm{col}\Big\{h(1), h(2), \ldots, h(M)\Big\} \tag{65.3a}$$

$$w = \mathrm{col}\Big\{w(1), w(2), \ldots, w(M)\Big\} \tag{65.3b}$$

then the output of the adder is the affine relation:

$$z \ \triangleq \ h^{\mathsf{T}} w - \theta \tag{65.4}$$

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

$$y \triangleq f(z) = f(h^{\mathsf{T}}w - \theta) \tag{65.5}$$

On the right-hand side of the same figure, 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 simply operate as a pure *linear combiner*.

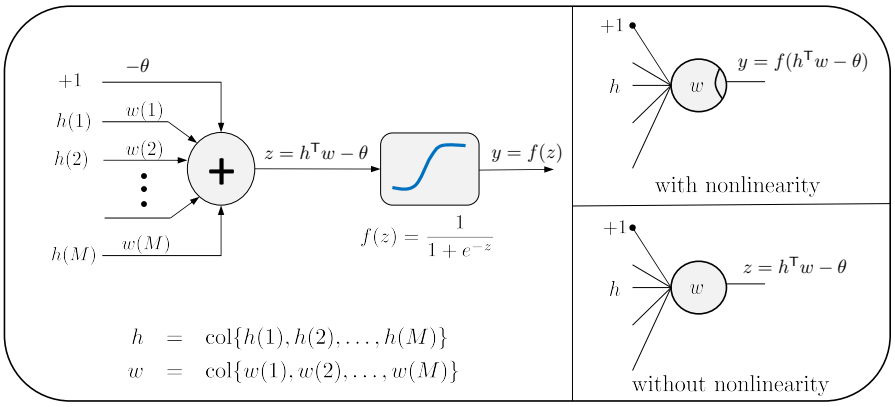

Figure 65.1 (*Left*) Structure of a neuron consisting of an offset parameter $-\theta$, and M multipliers with weights $\{w(m)\}$ and input signals $\{h(m)\}$, followed by an adder with output z and a nonlinearity $y = f(z)$. (*Right*) Compact representations for the neuron in terms of a circle with multiple input lines and one output line. Two circle representations are used to distinguish between the cases when the nonlinearity is present or not (i.e., whether $f(z) = z$ or not). When a nonlinearity is present, we will indicate its presence by an arc inside the circular representation, as shown in the top right corner.

Sigmoid and tanh functions

There are several common choices for the activation function $f(z)$, listed in Table 65.1 with some of them illustrated in Fig. 65.2. We encountered the sigmoid function earlier in (59.5a) while discussing the logistic regression problem. One useful property of the sigmoid function is that its derivative admits the representation

$$f'(z) = f(z)(1 - f(z)), \quad \textbf{(sigmoid function)} \tag{65.6}$$

We also encountered the hyperbolic tangent function earlier in (27.33) while studying the optimal mean-square-error inference problem. Its derivative is given

by any of the forms:

$$\begin{aligned} f'(z) &= 1/\cosh^2(z) \\ &= 4/(e^z + e^{-z})^2 \\ &= 1 - (\tanh(z))^2, \quad (\textbf{tanh function}) \end{aligned} \tag{65.7}$$

The sigmoid and tanh functions are related via the translation

$$\frac{1}{1+e^{-z}} = \frac{1}{2}\left(\tanh\left(z/2\right)+1\right) \iff \tanh(z/2) = 2\,\text{sigmoid}(z) - 1 \tag{65.8}$$

and satisfy

$$(\textbf{sigmoid function}): \quad \lim_{z\to+\infty} f(z) = 1, \quad \lim_{z\to-\infty} f(z) = 0 \tag{65.9a}$$

$$(\textbf{tanh function}): \quad \lim_{z\to+\infty} f(z) = 1, \quad \lim_{z\to-\infty} f(z) = -1 \tag{65.9b}$$

That is, both functions saturate for large $|z|$. This means that when $|z|$ is large, the derivatives of the sigmoid and tanh functions will assume small values close to zero. We will explain later in Sec. 65.8 that this property is problematic and is responsible for a *slowdown* in the speed of learning by neural networks. This is because small derivative values at any neuron will end up limiting the learning ability of the neurons in the preceding layers.

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. Typical choices for the parameters (a, b) are

$$b = \frac{2}{3}, \quad a = \frac{1}{\tanh(2/3)} \approx 1.7159 \tag{65.10}$$

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

ReLU and leaky-ReLU functions

In the rectifier (or hinge) function listed in Table 65.1 (also called a "rectified linear unit" or ReLU function), nonnegative values of z remain unaltered while negative values of z are set to zero. In this case, we set the derivative value at $z = 0$ to $f'(0) = 0$ by convention (we could also set it to one, if desired):

$$f(z) = \text{ReLU}(z) \implies f'(z) = \begin{cases} 0, & z < 0 \\ 0, & z = 0 \\ 1, & z > 0 \end{cases} \tag{65.11}$$

Compared with the sigmoid function, we observe that the derivative of ReLU is constant and equal to one for all positive values of z; this property will help speed up the training of neural networks and is one of the main reasons why ReLU activation functions are generally preferred over sigmoid functions. ReLU

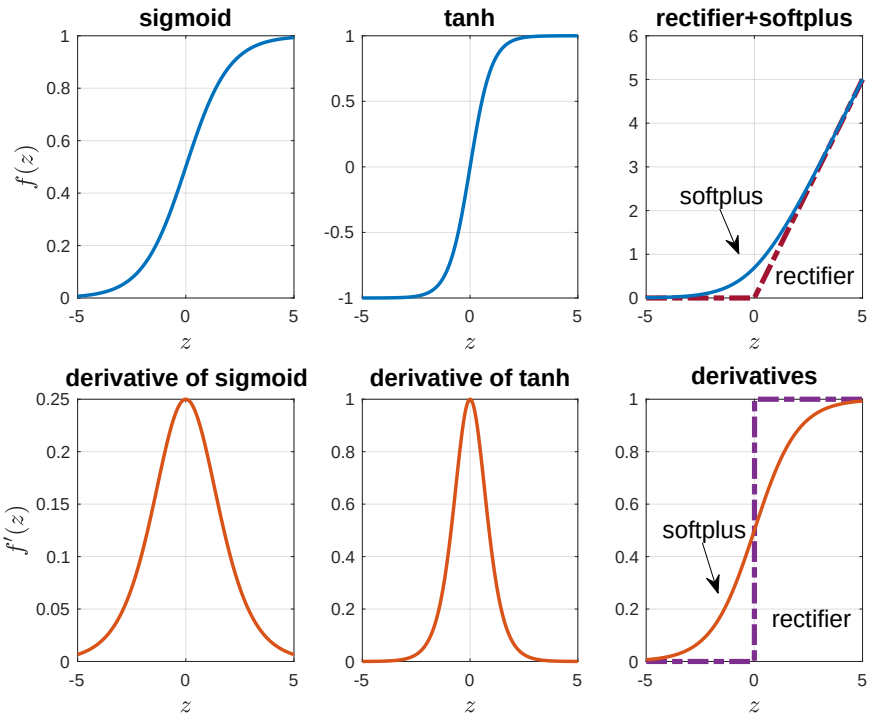

Figure 65.2 Examples of activation functions and their derivatives. (*Left*) Sigmoid function. (*Center*) Hyperbolic tangent function. (*Right*) Rectifier and softplus functions.

functions are also easy to implement and do not require the exponentiation operation that appears in the sigmoid implementation.

Unfortunately, the derivative of the ReLU function is zero for negative values of z, which will affect training when internal values in the network drop below zero. These nodes will not be able to continue learning and recover from their state of negative $z-$values. This challenge is referred to as the "dying ReLU" problem. The softplus function provides a smooth approximation for the rectifier function, tending to zero gracefully as $z \to -\infty$. The leaky-ReLU version, on the other hand, incorporates a small positive gradient for $z < 0$:

$$f(z) = \text{leaky ReLU(z)} \implies f'(z) = \begin{cases} 0.01, & z < 0 \\ 0, & z = 0 \\ 1, & z > 0 \end{cases} \tag{65.12}$$

The exponential linear unit (ELU) also addresses the problem over negative z by incorporating an exponential decay term such that as $z \to -\infty$, the function ELU(z) will tend to $-\alpha$ where $\alpha > 0$. The value of α can be selected through a (cross-validation) training process by simulating the performance of the neural network with different choices for α. The rectifier functions are widely used within

Table 65.1 Typical choices for the activation function $f(z)$ used in (65.5).

activation function	$f(z)$
sigmoid or logistic	$f(z) = \dfrac{1}{1+e^{-z}}$
softplus	$f(z) = \ln(1+e^{z})$
hyperbolic tangent (tanh)	$f(z) = \tanh(z) \triangleq \dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$
scaled tanh	$f(z) = a\tanh(bz), a, b > 0$
ReLU (hinge)	$f(z) = \max\{0, z\}$
leaky ReLU	$f(z) = \begin{cases} z, & z \geq 0 \\ 0.01z, & z < 0 \end{cases}$
ELU	$f(z) = \begin{cases} z, & z \geq 0 \\ \alpha(e^{z}-1), & z < 0 \end{cases}$
no activation	$f(z) = z$

hidden layers in the training of deep and convolutional neural networks. ELU activation functions have been observed to lead to neural networks with higher classification performance than ReLUs.

Softmax activation

The activation functions shown in Table 65.1 act on scalar arguments z; these arguments are the internal signals within the various neurons. We will encounter another popular activation function known as *softmax activation*, and written compactly as $y = \text{softmax}(z)$. This activation will be used at the output layer of the neural network, which has Q neurons, and it will operate simultaneously on all $z-$values from these neurons denoted by $\{z(q)\}$, for $q = 1, 2, \ldots, Q$. Softmax transforms these values into another set of Q values constructed as:

$$y(q) \triangleq e^{z(q)} \left(\sum_{q'=1}^{Q} e^{z(q')} \right)^{-1}, \quad q = 1, 2, \ldots, Q \tag{65.13}$$

Observe that each $y(q)$ is influenced by all $\{z(q)\}$. The exponentiation and normalization in the denominator ensure that the output variables $\{y(q)\}$ are all nonnegative and add up to one. In this way, the softmax transformation generates a Gibbs probability distribution — recall (3.168).

We explained earlier in Remark 36.2 that some care is needed in the numerical implementation of the softmax procedure due to the exponentiation in (65.13) and the possibility of overflow or underflow. For example, if the number in the exponent has a large value, then $y(q)$ can saturate in finite-precision implementations. One way to avoid this difficulty is to subtract from all the $\{z(q)\}$ their largest value and introduce the centered variables:

$$s(q) \triangleq z(q) - \left(\max_{1 \leq q' \leq Q} z(q') \right), \quad q = 1, 2, \ldots, Q \tag{65.14}$$

By doing so, the largest value of the $\{s(q)\}$ will be zero. It is easy to see that using the $\{s(q)\}$ instead of the $\{z(q)\}$ does not change the values of the $\{y(q)\}$:

$$y(q) = e^{s(q)} \left(\sum_{q'=1}^{Q} e^{s(q')} \right)^{-1} \tag{65.15}$$

Comparing with Perceptron

In Fig. 65.3 we compare the structure of the sigmoidal neuron with Perceptron, which uses the sign function for activation with its sharp discontinuous transition. Recall from (60.25) that Perceptron predicts the label for a feature vector h by using

$$\widehat{\gamma} = h^{\mathsf{T}} w - \theta \tag{65.16}$$

which agrees with the expression for z in the figure. Subsequently, the class for h is decided based on the sign of $\widehat{\gamma}$. In other words, the Perceptron unit operates on z by means of an implicit sign function. One of the key advantages of using continuous (smooth) activation functions in constructing neural networks, such as the sigmoid or tanh functions, over the discontinuous sign function, is that the resulting networks will respond more gracefully to small changes in their internal signals. For example, networks consisting solely of interconnected Perceptron neurons can find their output signals change dramatically in response to small internal signal variations; this is because the outputs of the sign functions can change suddenly from -1 to $+1$ for slight changes at their inputs. Smooth activation functions limit this sensitivity.

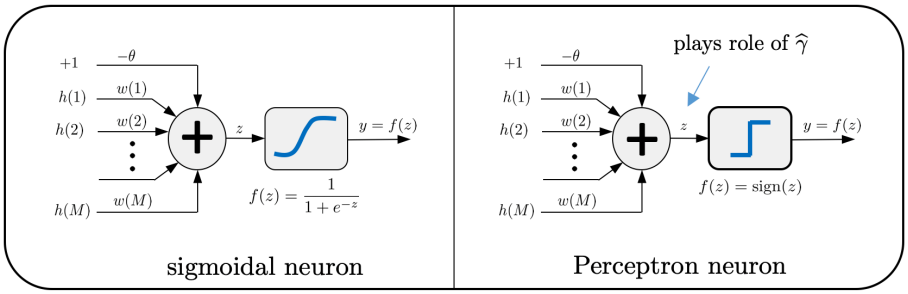

Figure 65.3 (*Left*) Neuron where the output of the linear combiner is smoothed through a sigmoid activation function. (*Right*) Perceptron neuron where the output of the linear combiner is applied to a hard-thresholding sign function.

65.2 FEEDFORWARD NETWORKS

We explain next how to combine several neurons to form a feedforward multi-layer neural network, which we will subsequently train to solve classification

and regression problems. In the feedforward implementation, information flows forward in the network and there is no loop to feed signals from future layers back to earlier layers.

Figure 65.4 illustrates this structure for a network consisting of an input layer, three hidden layers, and an output layer. In this example, there are two output nodes in the output layer, denoted by $\widehat{\gamma}(1)$ and $\widehat{\gamma}(2)$, 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 neurons in the hidden layers, again excluding the bias sources. The neurons in each layer are numbered with the numbers placed inside the symbol for the neuron. We will be using the terminology "node" to refer to any arbitrary element in the network, whether it is a neuron or an input node. In this example, the nodes in the output layer employ activation functions. There are situations where output nodes may be simple combiners without activation functions (for example, when the network is applied to the solution of regression problems).

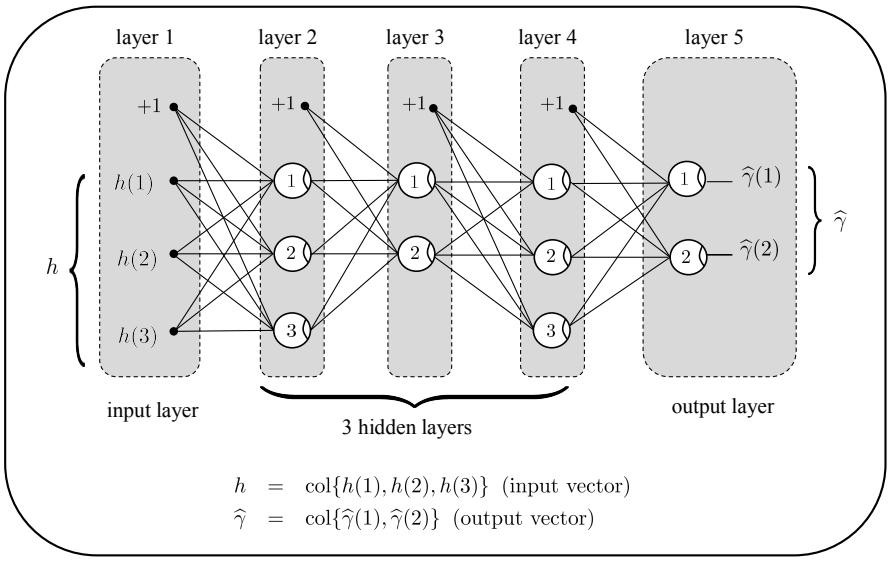

Figure 65.4 A feedforward neural network consisting of an input layer, three hidden layers, and an output layer. There are three input nodes in the input layer (excluding the bias source denoted by $+1$) and two output nodes in the output layer denoted by $\widehat{\gamma}(1)$ and $\widehat{\gamma}(2)$.

For convenience, we will 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. 65.4 we have $L = 5$ layers, three of which are hidden. Usually, large networks with many hidden layers are referred to as *deep* networks. For 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

$$n_1 = 3, \quad n_2 = 3, \quad n_3 = 2, \quad n_4 = 3, \quad n_5 = 2 \tag{65.17}$$

Now, between any two layers, there will be 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, as shown in Fig. 65.5, these coefficients can be collected into a matrix W_2 of size $n_2 \times n_3$ with individual entries:

$$W_2 \triangleq \begin{bmatrix} w_{11}^{(2)} & w_{12}^{(2)} \\ w_{21}^{(2)} & w_{22}^{(2)} \\ w_{31}^{(2)} & w_{32}^{(2)} \end{bmatrix}, \quad (n_2 \times n_3) \tag{65.18}$$

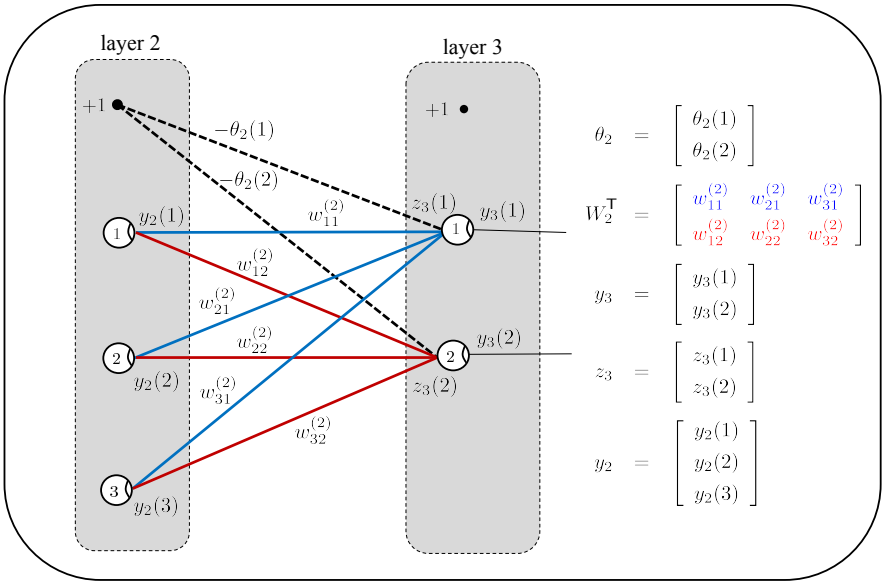

Figure 65.5 Combination and bias weights between layers 2 and 3 for the network shown in Fig. 65.4. The combination weights between nodes are collected into a matrix W_2 of size $n_2 \times n_3$, while the bias weights are collected into a vector θ_2 of size $n_3 \times 1$.

In the notation (65.18), the scalar $w_{ij}^{(\ell)}$ has the following interpretation:

$$w_{ij}^{(\ell)} = \text{weight from node } i \text{ in layer } \ell \text{ to node } j \text{ in layer } \ell + 1 \tag{65.19}$$

In the expressions used to describe the operation of a neural network, and its training algorithms, we will also be dealing with the *transpose* of the weight

matrix, which has size $n_3 \times n_2$ and is given by

$$W_2^{\mathsf{T}} \triangleq \begin{bmatrix} w_{11}^{(2)} & w_{21}^{(2)} & w_{31}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} & w_{32}^{(2)} \end{bmatrix}, \quad (n_3 \times n_2) \tag{65.20}$$

We also associate with layers 2 and 3 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

$$\theta_2 \triangleq \begin{bmatrix} \theta_2(1) \\ \theta_2(2) \end{bmatrix}, \quad (n_3 \times 1) \tag{65.21}$$

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

$$-\ \theta_\ell(j) = \text{weight from } +1 \text{ bias source in layer } \ell \text{ to node } j \text{ in layer } \ell+1 \tag{65.22}$$

Figure 65.5 illustrates these definitions for the parameters linking layers 2 and 3. In the figure we are illustrating the common situation when every node from a preceding layer feeds into every node in the succeeding layer, thus leading to a *fully-connected* interface between the layers. One can consider situations where only a subset of these connections are active (selected either at random or according to some policy); we will focus for now on the fully-connected case and discuss later the dropout strategy where some of the links will be deactivated. Later in Chapter 67, when we discuss convolutional neural networks, we will encounter other strategies for connecting nodes between successive layers.

Figure 65.6 illustrates the four weight matrices, $\{W_1^{\mathsf{T}}, W_2^{\mathsf{T}}, W_3^{\mathsf{T}}, W_4^{\mathsf{T}}\}$, and four bias vectors, $\{\theta_1, \theta_2, \theta_3, \theta_4\}$, that are associated with the network of Fig. 65.4. More generally, the combination weights between two layers ℓ and $\ell+1$ will be collected into a matrix W_ℓ^{T} of size $n_{\ell+1} \times n_\ell$ and the bias weights into layer $\ell+1$ will be collected in a column vector θ_ℓ of size $n_{\ell+1} \times 1$. Continuing with Fig. 65.5, we denote the outputs of the nodes in layer 3 by $y_3(1)$ and $y_3(2)$ and collect them into the vector:

$$y_3 \triangleq \begin{bmatrix} y_3(1) \\ y_3(2) \end{bmatrix}, \quad (n_3 \times 1) \tag{65.23}$$

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

$$y_\ell(j) = \text{output of node } j \text{ in layer } \ell \tag{65.24}$$

Likewise, the output vector for layer 2 is

$$y_2 \triangleq \begin{bmatrix} y_2(1) \\ y_2(2) \\ y_2(3) \end{bmatrix}, \quad (n_2 \times 1) \tag{65.25}$$

Using the vector and matrix quantities so defined, we can now examine the flow

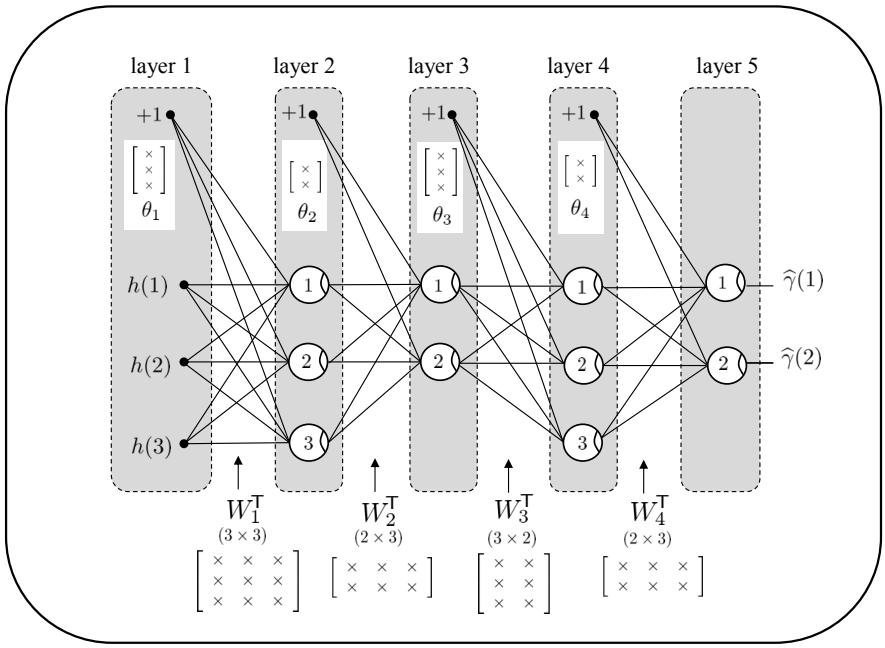

Figure 65.6 The combination weights between successive layers are collected into matrices, W_ℓ^{T}, for $\ell = 1, 2, \ldots, L-1$. Likewise, the bias coefficients for each hidden layer are collected into vectors, θ_ℓ, for $\ell = 1, 2, \ldots, L-1$.

of signals through the network. For instance, it is clear that the output vector for layer $\ell = 3$ is given by

$$y_3 = f\left(W_2^{\mathsf{T}} y_2 - \theta_2\right) \tag{65.26}$$

in terms of the output vector for layer 2 and where the notation $f(z)$ 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:

$$z_3 \triangleq \begin{bmatrix} z_3(1) \\ z_3(2) \end{bmatrix}, \quad (n_3 \times 1) \tag{65.27}$$

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

$$z_\ell(j) = \text{signal at node } j \text{ of layer } \ell \text{ prior to activation function} \tag{65.28}$$

That is,

$$y_3(1) = f(z_3(1)), \quad y_3(2) = f(z_3(2)) \tag{65.29}$$

If we now let $\widehat{\gamma} = \text{col}\{\widehat{\gamma}(1), \widehat{\gamma}(2)\}$ 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 — this flow is depicted schematically in Fig. 65.7. In this description, the vectors $\{z_\ell, y_\ell\}$ denote the pre- and post-activation signals for the internal layer of index ℓ. For the output layer, we will interchangeably use either the notation $\{z_L, y_L\}$ or the notation $\{z, \widehat{\gamma}\}$ for the pre- and post-activation signals. We will also employ the following compact notation to refer to the forward recursions (65.31), which feed a feature vector h into a network with parameters $\{W_\ell, \theta_\ell\}$ and generates the output signals $(z, \widehat{\gamma})$ along with the intermediate signals $\{z_\ell, y_\ell\}$ at the internal layers:

$$\Big(\widehat{\gamma}, z, \{y_\ell, z_\ell\}\Big) \;=\; \textbf{forward}\Big(h, \{W_\ell, \theta_\ell\}\Big) \tag{65.30}$$

Feedforward propagation through a neural network with L layers

given feedforward network with L layers (input+output+hidden);
start with $y_1 = h$;
repeat for $\ell = 1, \ldots, L-1$:

$$\begin{aligned} z_{\ell+1} &= W_\ell^{\mathsf{T}} y_\ell - \theta_\ell \\ y_{\ell+1} &= f\,(z_{\ell+1}) \end{aligned} \tag{65.31}$$

end
$z = z_L$
$\widehat{\gamma} = y_L$

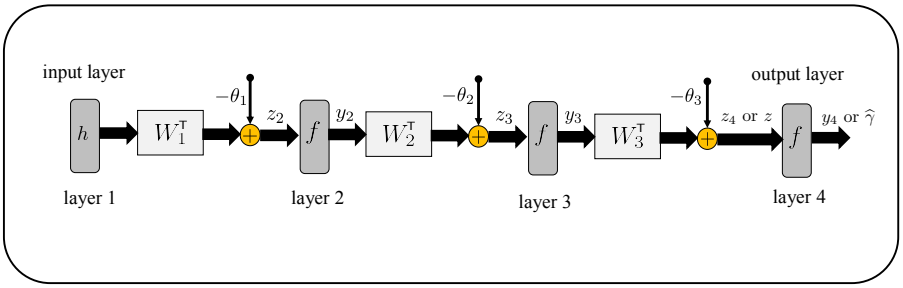

Figure 65.7 A block diagram representation of the feedforward recursions (65.31) through a succession of four layers, where the notation f denotes the activation function.

Column and row partitioning

Continuing with layers 2 and 3 from Fig. 65.5, 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 as follows:

$$W_2^{\mathsf{T}} = \left[\begin{array}{ccc} w_{11}^{(2)} & w_{21}^{(2)} & w_{31}^{(2)} \\ \hline w_{12}^{(2)} & w_{22}^{(2)} & w_{32}^{(2)} \end{array}\right] = \left[\begin{array}{c|c|c} w_{11}^{(2)} & w_{21}^{(2)} & w_{31}^{(2)} \\ w_{12}^{(2)} & w_{22}^{(2)} & w_{32}^{(2)} \end{array}\right] \triangleq \left[\begin{array}{c|c|c} w_1^{(2)} & w_2^{(2)} & w_3^{(2)} \end{array}\right] \tag{65.32}$$

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

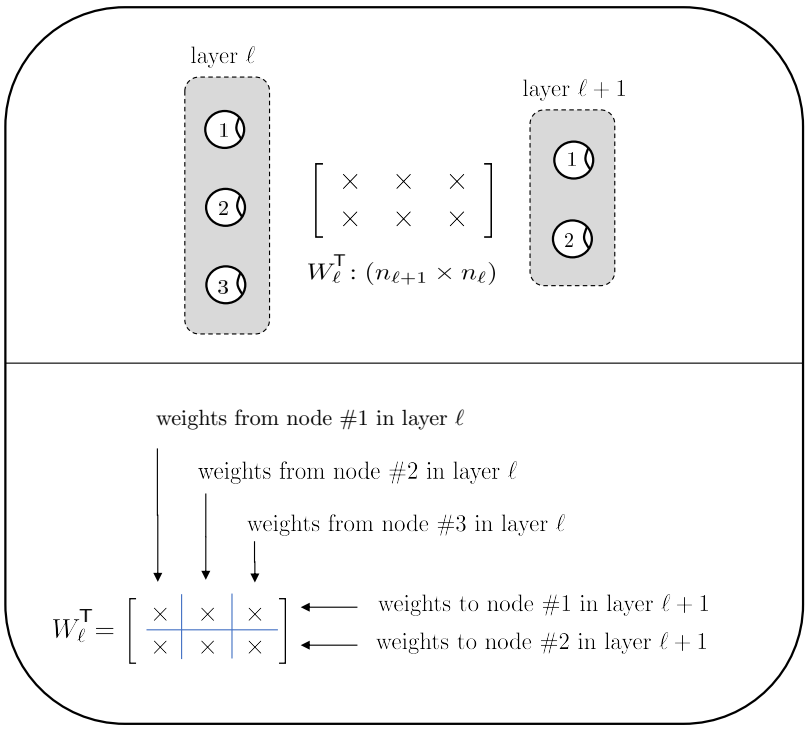

Figure 65.8 The weight matrix W_ℓ^{T} can be partitioned either in column or row form. The columns of W_ℓ^{T} represent the weights emanating from the nodes in layer ℓ. The rows of W_ℓ^{T} represent the weights arriving at the nodes in layer $\ell+1$.

In the second case, when the partitioning is column-wise, we observe that W_2^{T} consists of three columns; one column for each of the nodes in the *originating* layer 2. The entries of the first column 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 column are the weighting coefficients on the edges emanating from the second node in layer 2 to the nodes in layer 3, and likewise for the third column of W_2^{T}. In other words, each column of W_2^{T} consists of the weighting coefficients on the edges *emanating* from the nodes in layer 2. Figure 65.8 illustrates this partitioning for a generic combination matrix between two arbitrary layers ℓ and $\ell+1$. For later use, we find it convenient to denote the *columns* of W_ℓ^{T} by the notation — as depicted in the second line in (65.32):

$$w_i^{(\ell)} \;=\; \text{weight vector emanating from node } i \text{ in layer } \ell \tag{65.33}$$

For example, from (65.32) we have

$$w_1^{(2)} = \operatorname{col}\left\{w_{11}^{(2)}, w_{12}^{(2)}\right\} \;=\; \text{weights from node 1 in layer 2} \tag{65.34a}$$

$$w_2^{(2)} = \operatorname{col}\left\{w_{21}^{(2)}, w_{22}^{(2)}\right\} \;=\; \text{weights from node 2 in layer 2} \tag{65.34b}$$

$$w_3^{(2)} = \operatorname{col}\left\{w_{31}^{(2)}, w_{32}^{(2)}\right\} \;=\; \text{weights from node 3 in layer 2} \tag{65.34c}$$

65.3 REGRESSION AND CLASSIFICATION

Feedforward neural networks can be viewed as systems that map multi-dimensional input vectors $\{h \in \mathbb{R}^M\}$ into multi-dimensional output vectors $\{\widehat{\gamma} \in \mathbb{R}^Q\}$, so that the network is effectively a multi-input multi-output system. We can exploit this level of generality to solve regression problems as well as *multiclass* and *multilabel* classification problems.

Regression

In regression problems, the nodes at the output layer of the network do not employ activation functions. They will consist solely of linear combiners so that $\widehat{\gamma} = z$, where $\widehat{\gamma}$ denotes the output vector of the network and z denotes the pre-activation signal at the output layer. The individual entries of $\widehat{\gamma}$ will assume real values so that $\widehat{\gamma} \in \mathbb{R}^Q$.

Classification

In classification problems, the output layer of the network will employ activation functions; it is also often modified by replacing the activation functions by the softmax layer described earlier. Regardless of how the output vector $\widehat{\gamma}$ is generated, there will generally be an additional step that transforms it into a second vector $\gamma^\star$ with Q *discrete* entries, just like we used $\operatorname{sign}(\widehat{\gamma})$ in previous chapters to transform a scalar real-valued $\widehat{\gamma}$ into $+1$ or -1 by examining its sign. In the neural network context, we will perform similar operations, described below, to generate the discrete vector $\gamma^\star$ whose entries will belong, out of convenience, either to the choices $\{+1, -1\}$ or $\{1, 0\}$, e.g.,

$$\underset{\text{(real output)}}{\widehat{\gamma}(h) \in \mathbb{R}^Q} \quad \Longrightarrow\!\!\!\!\Longrightarrow \quad \underset{\text{(discrete output)}}{\gamma^\star(h) \in \{+1, -1\}^Q} \tag{65.35}$$

The choice of which discrete values to use, $\{+1, -1\}$ or $\{1, 0\}$, is usually dictated by the type of the activation function used. For instance, note that the sigmoid function in Table 65.1 generates values in the range $(0, 1)$, while the hyperbolic tangent function generates values in the range $(-1, 1)$. Therefore, for classification problems, we would:

(i) Employ the discrete values $\{1, 0\}$ when the sigmoid function is used. We can, for instance, obtain these discrete values by performing the following transformation on the entries of the output vector $\widehat{\gamma} \in \mathbb{R}^Q$:

$$\begin{cases} \text{if } \widehat{\gamma}(q) \geq 1/2, \text{ set } \gamma^\star(q) = 1 \\ \text{if } \widehat{\gamma}(q) < 1/2, \text{ set } \gamma^\star(q) = 0 \end{cases} \tag{65.36}$$

for $q = 1, 2, \ldots, Q$.

(ii) Employ the discrete values $\{+1, -1\}$ when the hyperbolic tangent function is used. We can obtain these discrete values by performing the following transformation on the entries of the output vector $\widehat{\gamma} \in \mathbb{R}^Q$:

$$\gamma^\star(q) = \text{sign}\Big(\widehat{\gamma}(q)\Big), \quad q = 1, 2, \ldots, Q \tag{65.37}$$

We can exploit the vector nature of $\{\widehat{\gamma}, \gamma^\star\}$ to solve two types of classification problems:

(a) (**Multilabel classification**) In some applications, one is interested in determining whether a feature vector, h, implies the presence of certain conditions $\mathbb{A}$ or $\mathbb{B}$ or more, for example, such as checking whether an individual with feature h is overweight or not (condition $\mathbb{A}$) and smokes or not (condition $\mathbb{B}$). In cases like these, we would train a neural network to generate two labels, i.e., a two-dimensional vector $\gamma^\star$ with individual discrete entries denoted by $\gamma^\star(\mathbb{A})$ and $\gamma^\star(\mathbb{B})$:

$$\widehat{\gamma}(h) \in \mathbb{R}^2 \implies \gamma^\star(h) = \begin{bmatrix} \gamma^\star(\mathbb{A}) \\ \gamma^\star(\mathbb{B}) \end{bmatrix} \tag{65.38}$$

One of the labels relates to condition $\mathbb{A}$ and would indicate whether the condition is present or not by assuming binary state values such as $\{+1, -1\}$ or $\{1, 0\}$. Similarly, for $\gamma^\star(\mathbb{B})$. This example corresponds to a *multilabel* classification problem. Such problems arise, for example, when training a neural network to solve a multitask problem such as deciding whether an image contains instances of streets, cars, and traffic signs.

(b) (**Multiclass classification**) In other applications, one is interested in classifying a feature vector, h, into only one of a collection of classes. For example, given a feature vector extracted from the image of a handwritten digit, one would want to identify the digit (i.e., classify h into one of ten classes: $0, 1, 2, \ldots, 9$). We encountered multiclass classification problems of

this type earlier in Examples 59.3.1 and 59.3.2 while discussing the one-versus-all (OvA) and one-versus-one (OvO) strategies. These solution methods focused on reducing the *multiclass* classification problem into a collection of binary classifiers. In comparison, in the neural network approach, the multiclass classification problem will be solved directly by generating a *vector*-valued class variable, $\gamma^\star \in \mathbb{R}^Q$ — see future Example 65.9. Each entry of this vector will correspond to one class. In particular, when h belongs to some class r, the $r-$th entry of $\gamma^\star$ will be activated at $+1$, while all other entries will be -1 (or zero, depending on which convention is used, $\{+1, -1\}$ or $\{1, 0\}$).

Softmax formulation

In most multiclass classification problems, the output of the feedforward network is a softmax layer where the entries $\widehat{\gamma}(q)$ are generated by employing the softmax function described before:

$$\widehat{\gamma}(q) \triangleq e^{z(q)} \left(\sum_{q'=1}^{Q} e^{z(q')} \right)^{-1} \tag{65.39}$$

Here, the variable $z(q)$ denotes the $q-$th entry of the output vector z prior to activation. Observe that the computation of $\widehat{\gamma}(q)$ is now influenced by *other* signals $\{z(q')\}$ from the other output neurons, and is not solely dependent on $z(q)$. For convenience, we will express the transformation (65.39) more succinctly by writing

$$\widehat{\gamma} \triangleq \text{softmax}(z) \tag{65.40}$$

The exponentiation and normalization in (65.39) ensure that the output variables $\{\widehat{\gamma}(q)\}$ are all nonnegative and add up to one. As a result, in multiclass classification problems, each $\widehat{\gamma}(q)$ can be interpreted as corresponding to the likelihood that the feature vector h belongs to class q. The label for vector h is then selected as the class $r^\star$ corresponding to the highest likelihood value

$$r^\star = \underset{1 \le q \le Q}{\text{argmax}} \; \widehat{\gamma}(q) \tag{65.41}$$

In other words, the predicted label $\gamma^\star$ can be set to the $r^\star-$basis vector in $\mathbb{R}^Q$:

$$\gamma^\star = e_{r^\star} \triangleq \text{col}\Big\{0, \ldots, 0, 1, 0, \ldots, 0\Big\}, \qquad (\textbf{basis vector}) \tag{65.42}$$

where the notation e_r refers to the basis vector with the value one at location r and zeros elsewhere. When the output of the network structure includes the softmax transformation, the last line of the feedforward propagation recursion (65.31) is modified as indicated below with $\widehat{\gamma} = y_L$ replaced by (65.40); the vector y_L does not need to be generated anymore. We continue to use the compact description (65.30) to refer to this implementation.

Feedforward propagation through L layers with softmax output

given feedforward network with L layers (input+output+hidden);
output layer is softmax;
start with $y_1 = h$;
repeat for $\ell = 1, \ldots, L-1$: (65.43)
  $z_{\ell+1} = W_\ell^{\mathsf{T}} y_\ell - \theta_\ell$
  $y_{\ell+1} = f\,(z_{\ell+1})$
end
$z = z_L$
$\widehat{\gamma} = \text{softmax}(z)$

65.4 CALCULATION OF GRADIENT VECTORS

Now that we have described the structure of feedforward neural networks, we can proceed to explain how to train them, i.e., how to determine their weight matrices $\{W_\ell\}$ and bias vectors $\{\theta_\ell\}$ to solve classification or regression problems. We will do so in two steps. First, we will derive the famed *backpropagation algorithm*, which is a popular procedure for evaluating gradient vectors. Subsequently, we will combine this procedure with *stochastic gradient approximation* to arrive at an effective training method. 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 necessary to pursue a systematic presentation to facilitate the derivation. We focus initially on the case in which all layers, including the output layer, employ activation functions; later, we explain the adjustments that are needed when the output layer relies on a softmax construction.

To begin with, since we will now be dealing with feature vectors $h \in \mathbb{R}^M$ that are indexed by a subscript n, say, $h_n \in \mathbb{R}^M$ (e.g., h_n can refer to vectors selected from a training set or to vectors streaming in over time), we will similarly denote the output vector $\widehat{\gamma} \in \mathbb{R}^Q$ corresponding to h_n by $\widehat{\gamma}_n$, with the same subscript n. We will also let $z_n \in \mathbb{R}^Q$ denote the output vector prior to activation so that

$$\widehat{\gamma}_n = f(z_n) \tag{65.44}$$

Thus, consider a collection of N data pairs $\{\gamma_n, h_n\}$ for $n = 0, 1, \ldots, N-1$, where γ_n denotes the actual discrete label vector corresponding to the $n-$th feature vector h_n. The objective is to train the neural network to result in an input-output mapping $h \to \widehat{\gamma}$ that matches reasonably well the mapping $h \to \gamma$ that is reflected by the training data.

65.4.1 Regularized Least-Squares Risk

We formulate initially a regularized empirical risk optimization problem of the following form:

$$\{W_\ell^\star, \theta_\ell^\star\} \triangleq \underset{\{W_\ell, \theta_\ell\}}{\operatorname{argmin}} \left\{ \sum_{\ell=1}^{L-1} \rho \|W_\ell\|_{\mathrm{F}}^2 + \frac{1}{N} \sum_{n=0}^{N-1} \|\gamma_n - \widehat{\gamma}_n\|^2 \right\} \tag{65.45}$$

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):

$$\|W_\ell\|_{\mathrm{F}}^2 = \sum_{i=1}^{n_\ell} \sum_{j=1}^{n_\ell+1} \left| w_{ij}^{(\ell)} \right|^2 \tag{65.46}$$

Therefore, the regularization term is in effect adding the squares of all combination weights in the network. As already explained in Sec. 51.2, regularization helps avoid overfitting and improves the generalization ability of the network. Other forms of regularization are possible, including ℓ_1-regularization (see Prob. 65.15), as well as other risk functions (see Probs. 65.15–65.18). We will motivate some of these alternative costs later in Sec. 65.8; their main purpose is to avoid the slowdown in learning that arises from using (65.45).

Continuing with (65.45), we denote the regularized empirical risk by

$$\mathcal{P}(W, \theta) \triangleq \sum_{\ell=1}^{L-1} \rho \|W_\ell\|_{\mathrm{F}}^2 + \frac{1}{N} \sum_{n=0}^{N-1} \|\gamma_n - \widehat{\gamma}_n\|^2 \tag{65.47}$$

where we are denoting the arguments of $\mathcal{P}(\cdot)$ generically by $\{W, \theta\}$; these refer to the collection of all parameters $\{W_\ell, \theta_\ell\}$ across all layers. The loss function that is associated with each data point in (65.47) is seen to be (we used the letter Q to refer to loss functions in earlier chapters; we will use the calligraphic letter $\mathcal{Q}$ here to avoid confusion with the number of classes Q at the output of the network):

$$\mathcal{Q}(W, \theta; \gamma, h) \triangleq \|\gamma - \widehat{\gamma}\|^2 \tag{65.48}$$

This loss value depends on all weight and bias parameters of the network. For simplicity, we will often drop the parameters (W, θ) and write only $\mathcal{Q}(\gamma, h)$.

It is useful to note from (65.47) that regularization is not applied to the bias vectors $\{\theta_\ell\}$; these entries are embedded into the output signals $\{\widehat{\gamma}_n\}$, as is evident from (65.31). Observe further that the risk (65.47) is *not* quadratic in the unknown variables $\{W_\ell, \theta_\ell\}$. Actually, the dependency of $\widehat{\gamma}_n$ on the variables $\{W_\ell, \theta_\ell\}$ is highly nonlinear due to the activation functions at the successive nodes. As a result, the risk function $\mathcal{P}(W, \theta)$ is non-convex over its parameters and will generally have multiple local minima. We will still apply stochastic-gradient constructions to seek a local minimizer, especially since it has been observed in

practice, through extensive experimentation, that the training algorithm works well despite the nonlinear and non-convex nature of the risk function.

In order to implement iterative procedures for "minimizing" $\mathcal{P}(W,\theta)$, we need to evaluate the gradients of $\mathcal{P}(W,\theta)$ relative to the individual entries of $\{W_\ell, \theta_\ell\}$, i.e., we need to evaluate quantities of the form:

$$\frac{\partial \mathcal{P}(W,\theta)}{\partial w_{ij}^{(\ell)}} \quad \text{and} \quad \frac{\partial \mathcal{P}(W,\theta)}{\partial \theta_\ell(j)} \tag{65.49}$$

for each layer ℓ and entries $w_{ij}^{(\ell)}$ and $\theta_\ell(j)$. The backpropagation algorithm is the procedure that enables us to compute these gradients in an effective manner, as we explain in the remainder of this section.

65.4.2 Sensitivity Factors

To simplify the notation in this section, we drop the subscript n and reinstate it later when it is time to list the final algorithm. The subscript is not necessary for the gradient calculations.

We thus consider a generic feedforward neural network consisting of L layers, including the input and output layers. We denote the vector signals at the output layer by $\{z, \widehat{\gamma}\}$, with the letter z representing the signal prior to the activation function, i.e.,

$$\widehat{\gamma} = f(z) \tag{65.50}$$

We denote the pre- and post-activation signals at the $\ell-$th hidden layer by $\{z_\ell, y_\ell\}$, which satisfy

$$y_\ell = f\left(z_\ell\right) \tag{65.51}$$

with individual entries indexed by $\{z_\ell(j), y_\ell(j)\}$ for $j = 1, 2, \ldots, n_\ell$. The number of nodes in the layer is n_ℓ (which excludes the bias source). We associate with each layer ℓ a *sensitivity* vector of size n_ℓ denoted by $\delta_\ell \in \mathbb{R}^{n_\ell}$ and whose individual entries are defined as follows:

$$\delta_\ell(j) \triangleq \frac{\partial \|\gamma - \widehat{\gamma}\|^2}{\partial z_\ell(j)} = \frac{\partial \mathcal{Q}(\gamma, h)}{\partial z_\ell(j)}, \quad j = 1, 2, \ldots, n_\ell \tag{65.52}$$

These factors measure how the unregularized term $\|\gamma - \widehat{\gamma}\|^2$ (i.e., the loss value $\mathcal{Q}(\gamma, h)$) varies in response to changes in the pre-activation signals, $z_\ell(j)$; we show later in (65.79) that this same quantity also measures the sensitivity to changes in the bias coefficients. It turns out that knowledge of the sensitivity variables facilitates evaluation of the partial derivatives (65.49).

We therefore examine how to compute the $\{\delta_\ell\}$ for all layers. We will show that these variables satisfy a backward recursive relation that tells us how to construct δ_ℓ from knowledge of $\delta_{\ell+1}$. We start with the output layer for which $\ell = L$. We denote its individual entries by $\{\widehat{\gamma}(1), \ldots, \widehat{\gamma}(Q)\}$. Likewise, we denote

the pre-activation entries by $\{z(1),\ldots,z(Q)\}$. In this way, the chain rule for differentiation gives

$$\begin{aligned}\delta_L(j) &\triangleq \frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial z(j)}\\ &\overset{(a)}{=} \sum_{k=1}^{Q}\frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial\widehat{\gamma}(k)}\,\frac{\partial\widehat{\gamma}(k)}{\partial z(j)}\\ &= \sum_{k=1}^{Q}2(\widehat{\gamma}(k)-\gamma(k))\,\frac{\partial\widehat{\gamma}(k)}{\partial z(j)}\\ &\overset{(b)}{=} 2\Big(\widehat{\gamma}(j)-\gamma(j)\Big)f'(z(j))\end{aligned} \tag{65.53}$$

where step (a) applies the following general chain-rule property. Assume y is a function of several variables, $\{x_1,x_2,\ldots,x_Q\}$, which in turn are themselves functions of some variable z, i.e.,

$$y=f(x_1,x_2,\ldots,x_Q),\quad x_q=g_q(z) \tag{65.54}$$

Then, it holds that

$$\frac{\partial y}{\partial z}=\sum_{q=1}^{Q}\frac{\partial y}{\partial x_q}\frac{\partial x_q}{\partial z},\qquad (\textbf{chain rule}) \tag{65.55}$$

Step (b) in (65.53) is because only $\widehat{\gamma}(j)$ depends on $z(j)$ through the relation $\widehat{\gamma}(j)=f(z(j))$. Consequently, using the Hadamard product notation we can write

$$\boxed{\delta_L \;=\; 2(\widehat{\gamma}-\gamma)\odot f'(z)} \qquad (\textbf{terminal sensitivity value}) \tag{65.56}$$

where $a\odot b$ denotes elementwise multiplication for two vectors a and b. It is important to note that the activation function whose derivative appears in (65.56) is the one associated with the *output layer* of the network. If desired, we can express the above relation in matrix-vector form by writing

$$\delta_L \;=\; 2J(\widehat{\gamma}-\gamma) \tag{65.57a}$$

where J is the diagonal matrix

$$J \;\triangleq\; \mathrm{diag}\Big\{f'(z(1)),\,f'(z(2)),\,\ldots,\,f'(z(Q))\Big\} \tag{65.57b}$$

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

$$\begin{aligned}\delta_\ell(j) &\triangleq \frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial z_\ell(j)}\\ &= \sum_{k=1}^{n_{\ell+1}} \frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial z_{\ell+1}(k)}\,\frac{\partial z_{\ell+1}(k)}{\partial z_\ell(j)}\\ &= \sum_{k=1}^{n_{\ell+1}} \delta_{\ell+1}(k)\frac{\partial z_{\ell+1}(k)}{\partial z_\ell(j)}\end{aligned} \tag{65.58}$$

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

$$z_{\ell+1}(k) \;=\; f\left(z_\ell(j)\right) w_{jk}^{(\ell)} \;+\; \text{terms independent of } z_\ell(j) \tag{65.59}$$

It follows that

$$\begin{aligned}\delta_\ell(j) &= \left(\sum_{k=1}^{n_{\ell+1}} \delta_{\ell+1}(k) w_{jk}^{(\ell)}\right) f'\left(z_\ell(j)\right)\\ &= f'\left(z_\ell(j)\right)\left(w_j^{(\ell)}\right)^{\mathsf{T}}\delta_{\ell+1}\end{aligned} \tag{65.60}$$

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 (65.33). In vector form, we arrive at the following recursion for the sensitivity vector δ_ℓ, which runs backward from $\ell = L-1$ down to $\ell = 2$ with the boundary condition δ_L given by (65.56):

$$\boxed{\delta_\ell = f'\left(z_\ell\right) \odot \left(W_\ell \delta_{\ell+1}\right)} \tag{65.61}$$

It is again useful to note that the activation function whose derivative appears in (65.61) is the one associated with the $\ell-$th layer of the network. If desired, we can also express this relation in matrix-vector form by writing

$$\delta_\ell \;=\; J_\ell W_\ell \delta_{\ell+1} \tag{65.62a}$$

where J_ℓ is now the diagonal matrix

$$J_\ell \;\triangleq\; \text{diag}\Big\{f'(z_\ell(1)),\, f'(z_\ell(2)),\, \ldots,\, f'(z_\ell(Q))\Big\} \tag{65.62b}$$

We therefore arrive at the following description for the flow of sensitivity signals backward through the network — this flow is depicted schematically in Fig. 65.9.

Backward propagation through a network with L layers

given a feedforward network with L layers (input+output+hidden);
pre- and post-activation signals at output layer are $(z, \widehat{\gamma})$;
internal pre-activation signals are $\{z_\ell\}$;
given feature vector $h \in \mathbb{R}^M$ with label vector $\gamma \in \mathbb{R}^Q$;
start from $\delta_L = 2(\widehat{\gamma} - \gamma) \odot f'(z)$
repeat for $\ell = L-1, , \ldots, 3, 2$:
$\quad \delta_\ell = f'(z_\ell) \odot (W_\ell \delta_{\ell+1})$
end (65.63)

For compactness of notation, we will sometimes express the backward recursion (65.63), which starts from a terminal condition δ_L and feeds it through a network with L layers, activation functions $f(\cdot)$, and combination matrices $\{W_\ell\}$, to generate the sensitivity factors $\{\delta_\ell,\ \ell = 2, \ldots, L-1\}$ by writing

$$\{\delta_\ell\} \ = \ \textbf{backward}\Big(\delta_L, f, \{z_\ell, W_\ell\}\Big) \tag{65.64}$$

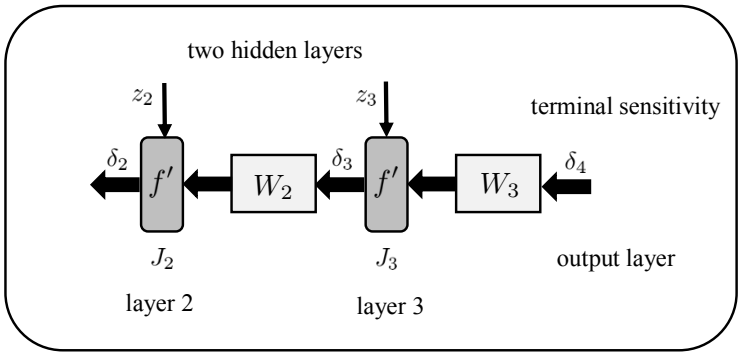

Figure 65.9 A block diagram representation of the backward recursion (65.63) for the same scenario shown earlier in Fig. 65.7, which involves only two hidden layers.

Example 65.1 (**Terminal sensitivity factor for softmax layer**) The backward recursion (65.61) continues to be valid when the output layer of the neural network is modified to be the softmax construction (65.39), in which case $\widehat{\gamma} = \text{softmax}(z)$. The only change will be in the terminal or boundary value, δ_L, as we now explain.

If we refer to expressions (65.52) and (65.53), where the subscript n was dropped for convenience, we have

$$\delta_L(j) = \sum_{k=1}^{Q} 2\Big(\widehat{\gamma}(k) - \gamma(k)\Big)\,\frac{\partial \widehat{\gamma}(k)}{\partial z(j)} \tag{65.65}$$

where now, in view of the normalization (65.39):

$$\frac{\partial\widehat{\gamma}(k)}{\partial z(j)} = \begin{cases} -\widehat{\gamma}(j)\widehat{\gamma}(k), & k \neq j \\ (1-\widehat{\gamma}(k))\widehat{\gamma}(k), & k = j \end{cases} \tag{65.66}$$

Substituting into (65.65) gives

$$\delta_L(j) = 2\Big(\widehat{\gamma}(j) - \gamma(j)\Big)\widehat{\gamma}(j) - \left(\sum_{k=1}^{Q} 2(\widehat{\gamma}(k) - \gamma(k))\widehat{\gamma}(k)\right)\widehat{\gamma}(j) \tag{65.67}$$

We can express this relation in a more compact form by collecting the partial derivatives (65.66) into a $Q \times Q$ symmetric matrix:

$$[J]_{jk} \triangleq \frac{\partial\widehat{\gamma}(k)}{\partial z(j)}, \quad j,k = 1,2,\ldots,Q \tag{65.68}$$

That is, for $Q = 3$:

$$J = \begin{bmatrix} (1-\widehat{\gamma}(1))\widehat{\gamma}(1) & -\widehat{\gamma}(1)\widehat{\gamma}(2) & -\widehat{\gamma}(1)\widehat{\gamma}(3) \\ -\widehat{\gamma}(2)\widehat{\gamma}(1) & (1-\widehat{\gamma}(2))\widehat{\gamma}(2) & -\widehat{\gamma}(2)\widehat{\gamma}(3) \\ -\widehat{\gamma}(3)\widehat{\gamma}(1) & -\widehat{\gamma}(3)\widehat{\gamma}(2) & (1-\widehat{\gamma}(3))\widehat{\gamma}(3) \end{bmatrix} = \mathrm{diag}(\widehat{\gamma}) - \widehat{\gamma}\widehat{\gamma}^{\mathsf{T}} \tag{65.69}$$

Then, we can rewrite (65.67) in the following matrix-vector product form:

$$\boxed{\delta_L = 2J(\widehat{\gamma} - \gamma)} \tag{65.70}$$

Comparing this expression with (65.56), we see that (65.70) *does not* depend on the terminal derivative term $f'(z)$.

65.4.3 Expressions for the Gradients

We are ready to evaluate the partial derivatives in (65.49). Following arguments similar to the above, we note that

$$\begin{aligned} \frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial w_{ij}^{(\ell)}} &= \sum_{k=1}^{n_{\ell+1}} \frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial z_{\ell+1}(k)} \frac{\partial z_{\ell+1}(k)}{\partial w_{ij}^{(\ell)}} \\ &= \frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial z_{\ell+1}(j)} \frac{\partial z_{\ell+1}(j)}{\partial w_{ij}^{(\ell)}} \\ &= \delta_{\ell+1}(j) y_\ell(i) \end{aligned} \tag{65.71}$$

where the second equality is because only $z_{\ell+1}(j)$ depends on $w_{ij}^{(\ell)}$. If we apply result (65.71) to the combination matrix W_2 defined earlier in (65.18), we find that these gradient calculations lead to (where we are writing $\partial\|\gamma-y\|^2/\partial W_2$ to refer to the resulting matrix):

$$\frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial W_2} = \begin{bmatrix} \delta_3(1)y_2(1) & \delta_3(2)y_2(1) \\ \delta_3(1)y_2(2) & \delta_3(2)y_2(2) \\ \delta_3(1)y_2(3) & \delta_3(2)y_2(3) \end{bmatrix} = y_2\delta_3^{\mathsf{T}} \tag{65.72}$$

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

$$\frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial W_\ell} = y_\ell\delta_{\ell+1}^{\mathsf{T}} \tag{65.73}$$

so that from (65.47), and after restoring the subscript n,

$$\frac{\partial\mathcal{P}(W,\theta)}{\partial W_\ell} = 2\rho W_\ell \;+\; \frac{1}{N}\sum_{n=0}^{N-1} y_{\ell,n}\delta_{\ell+1,n}^{\mathsf{T}}, \quad (\textbf{a matrix}) \tag{65.74}$$

In this notation, $y_{\ell,n}$ is the output vector for layer ℓ at time n.

A similar argument can be employed to compute the gradients of $\|\gamma-\widehat{\gamma}\|^2$ relative to the bias weights, $\theta_\ell(i)$, across the layers. Thus, note that

$$\frac{\partial\mathcal{P}(W,\theta)}{\partial\theta_\ell(i)} = \frac{1}{N}\sum_{n=0}^{N-1}\frac{\partial\|\gamma_n-\widehat{\gamma}_n\|^2}{\partial\theta_\ell(i)} \tag{65.75}$$

so that, in a manner similar to the calculation (65.71),

$$\begin{aligned}\frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial\theta_\ell(i)} &= \sum_{k=1}^{n_{\ell+1}}\frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial z_{\ell+1}(k)}\,\frac{\partial z_{\ell+1}(k)}{\partial\theta_\ell(i)}\\ &= \frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial z_{\ell+1}(i)}\,\frac{\partial z_{\ell+1}(i)}{\partial\theta_\ell(i)}\\ &= -\delta_{\ell+1}(i)\end{aligned} \tag{65.76}$$

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

$$z_{\ell+1}(i) \;=\; -\theta_\ell(i) \;+\; \text{terms independent of } \theta_\ell(i) \tag{65.77}$$

If we apply result (65.76) to the bias vector θ_2 defined earlier in (65.21), we find that these gradient calculations lead to (where we are writing $\partial\|\gamma-\widehat{\gamma}\|^2/\partial\theta_2$ to refer to the resulting gradient vector):

$$\frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial\theta_2} = \begin{bmatrix} -\delta_3(1) \\ -\delta_3(2) \end{bmatrix} \tag{65.78}$$

More generally, we have

$$\frac{\partial\|\gamma-\widehat{\gamma}\|^2}{\partial\theta_\ell} = -\delta_{\ell+1} \tag{65.79}$$

so that from (65.47), and after restoring the subscript n,

$$\frac{\partial\mathcal{P}(W,\theta)}{\partial\theta_\ell} = -\frac{1}{N}\sum_{n=0}^{N-1}\delta_{\ell+1,n}, \quad (\textbf{a column vector}) \tag{65.80}$$

In summary, we arrive at the following listing for the main steps involved in computing the partial derivatives in (65.49) relative to all combination matrices and bias vectors in a feedforward network consisting of L layers. In the description below, we reinstate the subscript n to refer to the sample index. Moreover, the

quantities $\{y_{\ell,n}, \delta_{\ell,n}, z_{\ell,n}\}$ are all vectors associated with layer ℓ, while $\{z_n, \widehat{\gamma}_n\}$ are the pre- and post-activation vectors at the output layer of the network. When the softmax construction (65.39) is employed at the output layer, we simply replace the boundary condition for $\delta_{L,n}$ by (65.70).

Computation of partial derivatives for empirical risk (65.47)

given a feedforward network with L layers (input+output+hidden);
pre- and post-activation signals at output layer are $(z_n, \widehat{\gamma}_n)$;
internal pre- and post-activation signals are $\{z_{\ell,n}, y_{\ell,n}\}$;
given N training data samples $\{\gamma_n, h_n\}$;
repeat for $n = 0, 1, \ldots, N-1$**:**

$$\begin{aligned}
&\Big(\widehat{\gamma}_n, z_n, \{y_{\ell,n}, z_{\ell,n}\}\Big) = \text{forward}\Big(h_n, \{W_\ell, \theta_\ell\}\Big) \\
&\delta_{L,n} = 2(\widehat{\gamma}_n - \gamma_n) \odot f'(z_n) \\
&\{\delta_{\ell,n}\} = \text{backward}\Big(\delta_{L,n}, f, \{z_{\ell,n}, W_\ell\}\Big)
\end{aligned} \tag{65.81}$$

end

compute for $\ell = 1, 2, \ldots, L-1$:

$$\frac{\partial \mathcal{P}(W,\theta)}{\partial W_\ell} = 2\rho W_\ell \; + \; \frac{1}{N}\sum_{n=0}^{N-1} y_{\ell,n}\delta_{\ell+1,n}^{\mathsf{T}}, \qquad (n_\ell \times n_{\ell+1})$$

$$\frac{\partial \mathcal{P}(W,\theta)}{\partial \theta_\ell} = -\frac{1}{N}\sum_{n=0}^{N-1} \delta_{\ell+1,n}, \qquad (n_{\ell+1} \times 1)$$

65.5 BACKPROPAGATION ALGORITHM

We can now use the forward and backward recursions from the previous section to train the neural network by writing down a stochastic-gradient implementation with step-size $\mu > 0$. In this implementation, either one randomly-selected data point (γ_n, h_n) may be used per iteration or a mini-batch block of size B. We describe the implementation in the mini-batch mode. By setting $B = 1$, we recover a stochastic-gradient version with one data point per iteration. Moving forward, we will need to introduce an iteration index, m, and attach it to the combination matrices and bias vectors since they will now be adjusted from one iteration to the other. We will therefore write $W_{\ell,m}$ and $\theta_{\ell,m}$ for the parameter values a iteration m.

Mini-batch backpropagation algorithm for solving (65.45)

given a feedforward network with L layers (input+output+hidden);
pre- and post-activation signals at output layer are $(z_n, \widehat{\gamma}_n)$;
internal pre- and post-activation signals are $\{z_{\ell,n}, y_{\ell,n}\}$;
given N training data samples $\{\gamma_n, h_n\},\ n = 0, 1, \ldots, N-1$;
given small step-size $\mu > 0$ and regularization parameter $\rho \geq 0$;
start from random initial parameters $\{\boldsymbol{W}_{\ell,-1}, \boldsymbol{\theta}_{\ell,-1}\}$.

repeat until convergence over $m = 0, 1, 2, \ldots$**:**

select B random data pairs $\{\boldsymbol{\gamma}_b, \boldsymbol{h}_b\}$

(*forward processing*)

repeat for $b = 0, 1, \ldots, B-1$**:**

$$
\begin{aligned}
&\boldsymbol{y}_{1,b} = \boldsymbol{h}_b \\
&\textbf{repeat for } \ell = 1, 2, \ldots, L-1: \\
&\quad \boldsymbol{z}_{\ell+1,b} = \boldsymbol{W}_{\ell,m-1}^{\mathsf{T}} \boldsymbol{y}_{\ell,b} - \boldsymbol{\theta}_{\ell,m-1} \\
&\quad \boldsymbol{y}_{\ell+1,b} = f(\boldsymbol{z}_{\ell+1,b}) \\
&\textbf{end} \\
&\widehat{\boldsymbol{\gamma}}_b = \boldsymbol{y}_{L,b} \\
&\boldsymbol{z}_b = \boldsymbol{z}_{L,b} \\
&\boldsymbol{\delta}_{L,b} = 2(\widehat{\boldsymbol{\gamma}}_b - \boldsymbol{\gamma}_b) \odot f'(\boldsymbol{z}_b)
\end{aligned}
\tag{65.82}
$$

end

(*backward processing*)

repeat for $\ell = L-1, \ldots, 2, 1$:

$$
\begin{aligned}
&\boldsymbol{W}_{\ell,m} = (1 - 2\mu\rho)\boldsymbol{W}_{\ell,m-1} - \frac{\mu}{B} \sum_{b=0}^{B-1} \boldsymbol{y}_{\ell,b} \boldsymbol{\delta}_{\ell+1,b}^{\mathsf{T}} \\
&\boldsymbol{\theta}_{\ell,m} = \boldsymbol{\theta}_{\ell,m-1} + \frac{\mu}{B} \sum_{b=0}^{B-1} \boldsymbol{\delta}_{\ell+1,b} \\
&\boldsymbol{\delta}_{\ell,b} = f'(\boldsymbol{z}_{\ell,b}) \odot \Big(\boldsymbol{W}_{\ell,m-1} \boldsymbol{\delta}_{\ell+1,b} \Big),\ \ell \geq 2, b = 0, 1, \ldots, B-1
\end{aligned}
$$

end

end

$\{W_\ell^\star, \theta_\ell^\star\} \leftarrow \{\boldsymbol{W}_{\ell,m}, \boldsymbol{\theta}_{\ell,m}\}$

In the listing (65.82), we are denoting the training data and the network parameters in boldface to highlight their random nature. In contrast to (65.81), we are also blending the backward update for the sensitivity factors into the same loop for updating the parameters of the network for improved computational efficiency; this is because the sensitivity factors and the network parameters are updated one layer at a time. Clearly, if desired, we can perform multiple passes over the data and repeat (65.82) for several epochs. The training of the algorithm is performed until sufficient convergence is attained, which can be met by either

training until a certain maximum number of iterations has been reached (i.e., for $m \leq M_{\text{iter}}$), or until the improvement in the risk function $\mathcal{P}(W,\theta)$ is negligible, say,

$$\Big|\mathcal{P}(\{W_{\ell,m},\theta_{\ell,m}\}) \;-\; \mathcal{P}(\{W_{\ell,m-1},\theta_{\ell,m-1}\})\Big| \leq \epsilon \tag{65.83}$$

over two successive iterations and for some small enough $\epsilon > 0$. When the softmax construction (65.39) is employed at the output layer, the only adjustment that is needed to the algorithm is to replace the boundary condition for $\boldsymbol{\delta}_{L,b}$ by (65.70), namely,

$$\boldsymbol{\delta}_{L,b} = 2\boldsymbol{J}(\widehat{\boldsymbol{\gamma}}_b - \boldsymbol{\gamma}_b), \quad \boldsymbol{J} = \text{diag}(\widehat{\boldsymbol{\gamma}}_b) - \widehat{\boldsymbol{\gamma}}_b\widehat{\boldsymbol{\gamma}}_b^{\mathsf{T}} \tag{65.84}$$

Stochastic gradient implementation

When the batch size is $B = 1$, the above mini-batch recursions simplify to the listing shown in (65.87). Again, when the softmax construction is employed in the last layer, the expression for the boundary condition $\boldsymbol{\delta}_{L,m}$ in (65.82) would be replaced by:

$$\boldsymbol{\delta}_{L,m} = 2\boldsymbol{J}(\widehat{\boldsymbol{\gamma}}_m - \boldsymbol{\gamma}_m), \quad \boldsymbol{J} = \text{diag}(\widehat{\boldsymbol{\gamma}}_m) - \widehat{\boldsymbol{\gamma}}_m\widehat{\boldsymbol{\gamma}}_m^{\mathsf{T}} \tag{65.85}$$

In both implementations, the parameter values $\{W_\ell^\star, \theta_\ell^\star\}$ at the end of the training phase are used for testing purposes. For example, assuming a softmax output layer, the predicted class $r^\star$ corresponding to a feature vector h would be the index of the highest value within $\widehat{\gamma}$ (whose entries have the interpretation of a probability measure):

$$\widehat{\gamma} = \text{softmax}(z) \tag{65.86a}$$

$$\widehat{\gamma}(q) \approx \mathbb{P}(\boldsymbol{h} \in \text{class } q) \tag{65.86b}$$

$$r^\star = \underset{1\leq q\leq Q}{\text{argmax}} \;\widehat{\gamma}(q) \tag{65.86c}$$

where z is the vector prior to the activation at the last layer of the neural network, and which results from feeding the test feature h through the feedforward layers after training.

Stochastic-gradient backpropagation for solving (65.45)

given a feedforward network with L layers (input+output+hidden);
pre- and post-activation signals at output layer are $(z_n, \widehat{\gamma}_n)$;
internal pre- and post-activation signals are $\{z_{\ell,n}, y_{\ell,n}\}$;
given N training data samples $\{\gamma_n, h_n\},\ n = 0, 1, \ldots, N-1$;
given small step-size $\mu > 0$ and regularization parameter $\rho \geq 0$;
start from random initial parameters $\{\boldsymbol{W}_{\ell,-1}, \boldsymbol{\theta}_{\ell,-1}\}$.
repeat until convergence over $m = 0, 1, 2, \ldots$**:**

$$
\begin{aligned}
&\text{select one random data pair } (\boldsymbol{h}_m, \boldsymbol{\gamma}_m)\\
&\boldsymbol{y}_{1,m} = \boldsymbol{h}_m\\
&(\textit{forward processing})\\
&\textbf{repeat for } \ell = 1, 2, \ldots, L-1\textbf{:}\\
&\quad \boldsymbol{z}_{\ell+1,m} = \boldsymbol{W}_{\ell,m-1}^{\mathsf{T}} \boldsymbol{y}_{\ell,m} - \boldsymbol{\theta}_{\ell,m-1}\\
&\quad \boldsymbol{y}_{\ell+1,m} = f(\boldsymbol{z}_{\ell+1,m})\\
&\textbf{end}\\
&\widehat{\boldsymbol{\gamma}}_m = \boldsymbol{y}_{L,m}\\
&\boldsymbol{z}_m = \boldsymbol{z}_{L,m}\\
&\boldsymbol{\delta}_{L,m} = 2(\widehat{\boldsymbol{\gamma}}_m - \boldsymbol{\gamma}_m) \odot f'(\boldsymbol{z}_m)\\
&(\textit{backward processing})\\
&\textbf{repeat for } \ell = L-1, \ldots, 2, 1\textbf{:}\\
&\quad \boldsymbol{W}_{\ell,m} = (1 - 2\mu\rho)\boldsymbol{W}_{\ell,m-1} \ - \ \mu \boldsymbol{y}_{\ell,m} \boldsymbol{\delta}_{\ell+1,m}^{\mathsf{T}}\\
&\quad \boldsymbol{\theta}_{\ell,m} = \boldsymbol{\theta}_{\ell,m-1} + \mu \boldsymbol{\delta}_{\ell+1,m}\\
&\quad \boldsymbol{\delta}_{\ell,m} = f'(\boldsymbol{z}_{\ell,m}) \odot \Big(\boldsymbol{W}_{\ell,m-1} \boldsymbol{\delta}_{\ell+1,m}\Big), \ \ \ell \geq 2\\
&\textbf{end}\\
&\textbf{end}\\
&\{W_\ell^\star, \theta_\ell^\star\} \leftarrow \{\boldsymbol{W}_{\ell,m}, \boldsymbol{\theta}_{\ell,m}\}
\end{aligned}
\tag{65.87}
$$

Initialization

At step $n = -1$, it is customary to select the entries of the initial bias vector $\{\theta_{\ell,-1}\}$ randomly by following a Gaussian distribution with zero mean and unit variance:

$$
\boldsymbol{\theta}_{\ell,-1} \ \sim \mathcal{N}_{\boldsymbol{\theta}_\ell}(0, I_{n_{\ell+1}}) \tag{65.88}
$$

The combination weights $\{w_{ij,-1}^{(\ell)}\}$ across the network 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. It

is customary to select the variance of the Gaussian distribution as

$$\boldsymbol{w}_{ij,-1}^{(\ell)} \sim \mathcal{N}_{\boldsymbol{w}_{ij}^{(\ell)}}\Big(0, 1/n_\ell\Big) \tag{65.89}$$

The reason for this normalization is to limit the variation of the signals in the subsequent layer $\ell+1$. Note, in particular, that for an arbitrary node j in layer $\ell+1$, its initial pre-activation signal would be given by

$$\boldsymbol{z}_{\ell+1,-1}(j) = \sum_{i=1}^{n_\ell} \boldsymbol{w}_{ij,-1}^{(\ell)} \boldsymbol{y}_{\ell,-1}(i) - \boldsymbol{\theta}_{\ell,-1}(j) \tag{65.90}$$

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 follows that the variance of $\boldsymbol{z}_{\ell+1,-1}(j)$, denoted by σ_z^2, will be given by

$$\sigma_z^2 = 1 + \sum_{i=1}^{n_\ell} \frac{1}{n_\ell}\sigma_y^2 = 1 + \sigma_y^2, \quad \textbf{(with normalization)} \tag{65.91}$$

Without the variance scaling by $1/n_\ell$ in (65.89), the above variance would instead be given by

$$\sigma_z^2 = 1 + n_\ell \sigma_y^2, \quad \textbf{(without normalization)} \tag{65.92}$$

which grows linearly with n_ℓ. In this case, the pre-activation signals in layer $\ell+1$ will be more likely to assume larger (negative or positive) values when n_ℓ is large, which in turn means that the activation functions will saturate. This fact slows down the learning process in the network because small changes in internal signals (or weights) will have little effect on the output of a saturated node and on subsequent layers.

Other forms of initialization include selecting the weight variables by *uniformly* sampling within the following ranges, where the second and third choices are recommended when tanh and sigmoidal activation functions are used:

$$\boldsymbol{w}_{ij,-1}^{(\ell)} \in \mathcal{U}\left[-\frac{1}{\sqrt{n_\ell}}, \frac{1}{\sqrt{n_\ell}}\right] \tag{65.93a}$$

$$\boldsymbol{w}_{ij,-1}^{(\ell)} \in \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{n_{\ell+1}+n_\ell}}, \frac{\sqrt{6}}{\sqrt{n_{\ell+1}+n_\ell}}\right], \quad \text{(tanh activation)} \tag{65.93b}$$

$$\boldsymbol{w}_{ij,-1}^{(\ell)} \in \mathcal{U}\left[-\frac{4\sqrt{6}}{\sqrt{n_{\ell+1}+n_\ell}}, \frac{4\sqrt{6}}{\sqrt{n_{\ell+1}+n_\ell}}\right], \quad \text{(sigmoid activation)} \tag{65.93c}$$

These choices are meant to facilitate the reliable flow of information in the forward and backward directions in the network away from saturation to avoid the difficulties caused by the *vanishing gradient problem* (discussed later in Sec. 65.8).

Example 65.2 (**Early stopping procedure**) Regularization reduces overfitting and improves generalization. In the least-squares risk formulation described so far, and in the cross-entropy formulation described further ahead, regularization is attained by adding

a penalty term to the risk function. There are other more implicit ways by which regularization (i.e., improvement of the generalization ability) can be attained in order to reduce the gap between the training error and the test error. *Early stopping* is one such procedure; it relies on the use of *pocket variables* and a *validation test set*. During training, we split the original training data of size N into two groups: one larger group of size N_T used exclusively for the standard training procedure, say, by means of the stochastic gradient backpropagation algorithm, and a smaller disjoint collection of data points of size N_V used for validation purposes (with $N = N_T + N_V$). Early stopping operates as follows.

Assume we continually train the parameters $\{W_\ell, \theta_\ell\}$ of a neural network and test the learned parameters against the validation data after each iteration. It has been observed in practice that while the training error improves with training and continues to decrease, the same is not true for the validation error. The latter may improve initially but will start deteriorating after some time. This suggests that rather than train the network continually, we should stop and freeze the network parameters at those values where the validation error was the smallest. These parameter values are likely to lead to better generalization. We use pocket variables to keep track of these specific parameters. The details of the implementation are shown in (65.94).

Early-stopping procedure for training neural networks (65.94)

split the N training samples into two disjoint sets of size (N_T, N_V);
denote pocket variables by $\{W_{\ell,p}, \theta_{\ell,p}\}$;
denote initial conditions by $\{W_{\ell,-1}, \theta_{\ell,-1}\}$;
set pocket variables to the initial conditions;
set initial error on validation set to a very large value, denoted by R_p.
repeat for sufficient time:
  run N_T iterations of the stochastic-gradient algorithm to update the network parameters to $\{W_{\ell,N_T}, \theta_{\ell,N_T}\}$.
  evaluate the network error, R_v, on the validation set.
  if $R_v < R_p$
    update $R_p \leftarrow R_v$
    set the pocket variables to $\{W_{\ell,p}, \theta_{\ell,p}\} \leftarrow \{W_{\ell,N_T}, \theta_{\ell,N_T}\}$
  end
end
return $\{W_{\ell,p}, \theta_{\ell,p}\}$

Example 65.3 (**Training using ADAM**) Algorithm (65.87) relies on the use of stochastic-gradient updates for $\{\boldsymbol{W}_{\ell,m}, \boldsymbol{\theta}_{\ell,m}\}$. There are of course other methods to update the network parameters, including the use of *adaptive* gradients. Here, we describe how the updates for $\{\boldsymbol{W}_{\ell,m}, \boldsymbol{\theta}_{\ell,m}\}$ should be modified if we were to employ instead the ADAM recursions from Sec. 17.4.

Note first that the gradient matrix for the update of $\boldsymbol{W}_{\ell,m}$ at iteration m is given by

$$\boldsymbol{G}_{\ell,m} \triangleq 2\rho \boldsymbol{W}_{\ell,m-1} + \boldsymbol{y}_{\ell,m}\boldsymbol{\delta}_{\ell+1,m}^{\mathsf{T}} \tag{65.95}$$

while the gradient vector for updating $\boldsymbol{\theta}_{\ell,m}$ is given by $-\boldsymbol{\delta}_{\ell+1,m}$. Now, let $\bar{\boldsymbol{W}}_{\ell,m}$ and $\bar{\boldsymbol{\theta}}_{\ell,m}$ denote smoothed quantities that are updated as follows starting from zero initial conditions at $m = -1$:

$$\bar{\boldsymbol{W}}_{\ell,m} = \beta_{w,1}\bar{\boldsymbol{W}}_{\ell,m-1} + (1-\beta_{w,1})\boldsymbol{G}_{\ell,m} \tag{65.96a}$$

$$\bar{\boldsymbol{\theta}}_{\ell,m} = \beta_{\theta,1}\bar{\boldsymbol{\theta}}_{\ell,m-1} - (1-\beta_{\theta,1})\boldsymbol{\delta}_{\ell+1,m} \tag{65.96b}$$

A typical value for the forgetting factors $\beta_{w,1}, \beta_{\theta,1} \in (0,1)$ is 0.9. Let further $\boldsymbol{S}_{\ell,m}$ and $\boldsymbol{s}_{\ell,m}$ denote variance quantities associated with $\{\boldsymbol{W}_\ell, \boldsymbol{\theta}_\ell\}$ and updated as follows starting again from zero initial conditions for $m = -1$:

$$\boldsymbol{S}_{\ell,m} = \beta_{w,2}\boldsymbol{S}_{\ell,m-1} + (1-\beta_{w,2})\Big(\boldsymbol{G}_{\ell,m} \odot \boldsymbol{G}_{\ell,m}\Big) \tag{65.97a}$$

$$\boldsymbol{s}_{\ell,m} = \beta_{\theta,2}\boldsymbol{s}_{\ell,m-1} + (1-\beta_{\theta,2})\Big(\boldsymbol{\delta}_{\ell+1,m} \odot \boldsymbol{\delta}_{\ell+1,m}\Big) \tag{65.97b}$$

where $\odot$ denotes the Hadamard (elementwise) product of two matrices or vectors. A typical value for the forgetting factors $\beta_{w,2}, \beta_{\theta,2} \in (0,1)$ is 0.999.

Let the notation $A^{\odot\frac{1}{2}}$ refer to the elementwise computation of the square-roots of the entries of a matrix or vector argument. The ADAM updates for $\{\boldsymbol{W}_{\ell,m}, \boldsymbol{\theta}_{\ell,m}\}$ then take the form:

$$\boldsymbol{W}_{\ell,m} = \boldsymbol{W}_{\ell,m-1} - \mu \times \frac{\sqrt{1-\beta_{w,2}^{m+1}}}{1-\beta_{w,1}^{m+1}} \times \left\{ \bar{\boldsymbol{W}}_{\ell,m} \oslash \left(\epsilon \mathbb{1}_{n_\ell} \mathbb{1}_{n_{\ell+1}}^{\mathsf{T}} + \boldsymbol{S}_{\ell,m}^{\odot\frac{1}{2}} \right) \right\} \tag{65.98a}$$

$$\boldsymbol{\theta}_{\ell,m} = \boldsymbol{\theta}_{\ell,m-1} - \mu \times \frac{\sqrt{1-\beta_{\theta,2}^{m+1}}}{1-\beta_{\theta,1}^{m+1}} \times \left\{ \bar{\boldsymbol{\theta}}_{\ell,m} \oslash \left(\epsilon \mathbb{1}_{n_{\ell+1}} + \boldsymbol{s}_{\ell,m}^{\odot\frac{1}{2}} \right) \right\} \tag{65.98b}$$

where ϵ is a small positive number to avoid division by zero, and where we are using the symbol $\oslash$ to refer to elementwise division.

Example 65.4 (**Use of neural networks as autoencoders**) We can use three-layer neural networks to act as "autoencoders." The network consists of an input layer, one hidden layer, and an output layer, and its objective would be to map the input space back onto itself. This is achieved by constructing a feedforward network with the same number of output nodes as input nodes, and by setting $\gamma = h$:

$$Q = M, \quad \gamma = h \tag{65.99}$$

If the individual entries of h happen to lie within the interval $(0,1)$, then the output layer will include sigmoidal nonlinearities so that the output signals will lie within the same interval. On the other hand, if the entries of h lie within the interval $(-1,1)$, then the neurons in the output layer will include hyperbolic-tangent nonlinearities. If, however, the individual entries of h are arbitrary real numbers, then the neurons in the output layer should not include nonlinearities. For illustration purposes, we assume the entries of h lie within $(0,1)$ and consider an autoencoder structure of the form shown in Fig. 65.10 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 (65.87) or in the mini-batch form (65.82), to a collection of N feature vectors h_n using $\gamma_n = h_n$, we end up with an *unsupervised* learning procedure that trains the network to learn how to map h_n onto itself (i.e., it learns how to recreate the input data). This situation is illustrated in Fig. 65.10, which shows an autoencoder fed by h. The hidden layer in this example has three nodes. If we denote their outputs by

$$\{y_2(1),\, y_2(2),\, y_2(3)\} \tag{65.100}$$

then the network will be mapping these three signals through the output layer back into a good approximation for the five entries of the input data, h. The step of generating the outputs in the hidden layer amounts to a form of *data compression* since it generates three hidden signals that are sufficient to reproduce the five input signals.

Recall that we are denoting 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 $W_1 \in$

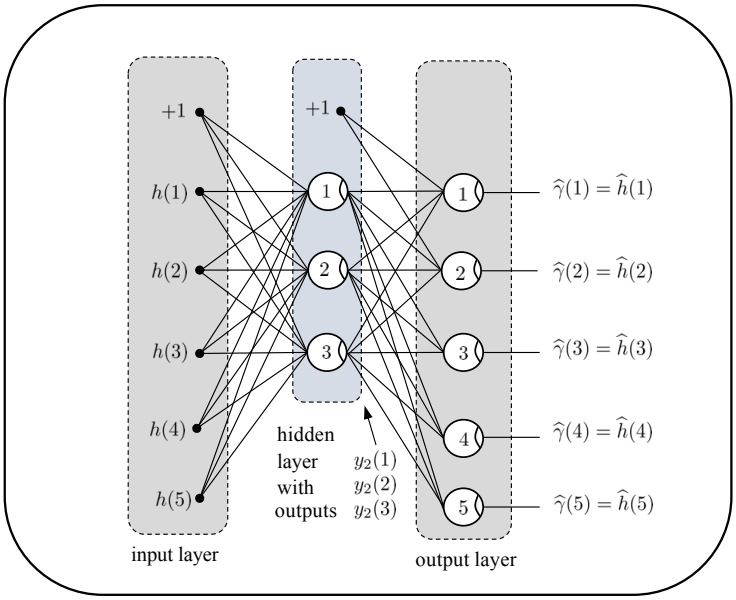

Figure 65.10 An autoencoder with a single hidden layer. The network is trained to map the feature vectors back to themselves using, for example, either the stochastic-gradient algorithm (65.87) or the mini-batch implementation (65.82) .

$\mathbb{R}^{M\times n_2}$, 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 of the form:

$$y_2 \;=\; f(W_1^{\mathsf{T}}h-\theta_1), \quad (\textbf{encoding}) \tag{65.101}$$

where $f(\cdot)$ denotes the activation function. Moreover, since the autoencoder is trained to recreate h, then $\widehat{\gamma}=\widehat{h}$ and we find that the representation y_2 is mapped back to the original feature vector by using

$$\widehat{h} \;=\; f(W_2^{\mathsf{T}}y_2-\theta_2), \quad (\textbf{decoding}) \tag{65.102}$$

in terms of the bias vector, θ_2, for the output layer and the weight matrix $W_2\in\mathbb{R}^{n_2\times M}$ between the hidden layer and the output layer. We refer to these two transformations as "encoding" and "decoding" steps, respectively. In some implementations, the weight matrices of the encoder and decoder sections are tied together by setting $W_2=W_1^{\mathsf{T}}$ — see Prob. 65.16.

Autoencoders provide a useful structure to encode data, i.e., to learn a compressed representation for the data and to perform dimensionality reduction. The reduced representation will generally extract 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 a feature extraction/detection module. Compact representations of this type are useful in reducing the possibility of overfitting in learning algorithms. This is because the reduced features can be used to drive learning algorithms 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., for which $n_2\geq M$. Obviously, this

case will not lead to dimensionality reduction. However, it has been observed in practice that such "overcomplete" representations are useful in that the features y_2 that they produce can lead to reduced classification errors. 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 often used, known as a "*denoising*" autoencoder. Here, a fraction of the input entries in h are randomly set to zero (possibly as many as 50% of them). The perturbed input vector, denoted by h', is then applied to the input of the autoencoder while the original vector h continues to be used as the reference signal, γ. 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 learning the correlations that may exist among the entries of the input vector in order to be able to "recover" the missing data.

One can further consider autoencoder implementations that involve multiple hidden layers, thus leading to *deep autoencoder* architectures. Obviously, the training of these layered structures becomes more challenging due to the vanishing gradient problem that we will be discussing in a future section.

Example 65.5 (**Linear autoencoders and PCA**) Consider a collection of N feature vectors $\{h_n \in \mathbb{R}^M\}$ and assume they have already been preprocessed according to procedure (57.6), i.e., each vector is centered around the ensemble mean and each entry is scaled by the ensemble standard deviation. We continue to denote the preprocessed features by $\{h_n\}$. We introduce the sample covariance matrix

$$\widehat{R} = \frac{1}{N-1}\sum_{n=0}^{N-1} h_n h_n^{\mathsf{T}} \tag{65.103}$$

and define its eigendecomposition $\widehat{R} = U\Lambda U^{\mathsf{T}}$, where U is $M \times M$ orthogonal and Λ is diagonal with nonnegative ordered entries

$$\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_M \geq 0 \tag{65.104}$$

Assume we wish to reduce the dimension of the feature data to $M' < M$. We retain the leading $M' \times M'$ block of Λ and the leading M' columns of U denoted by U_1 (which is $M \times M'$). We showed in listing (57.34) and Fig. 57.2 that the PCA implementation admits an encoder-decoder structure where the transformations from the original feature space h to the reduced space h' and back to the original space $\widehat{h}$ are given by

$$h'_n = U_1^{\mathsf{T}} h_n, \quad \widehat{h}_n = U_1 h'_n \tag{65.105}$$

If we make the identification $y_2 \leftarrow h'_n$, $W_1 = U_1$, and $W_2 = U_1^{\mathsf{T}}$, we conclude that PCA is a special case of the structure shown in Fig. 65.10 with the nonlinearities removed and with the weight matrices tied together (since we now have $W_2 = W_1^{\mathsf{T}}$). It should be noted that referring to PCA as a "linear" autoencoder, which is common in the literature, is technically inaccurate because the weight matrix U_1 depends on the feature data in a rather nonlinear fashion. The "linear" qualification is meant to refer to the fact that there are no nonlinearities when PCA is represented according to Fig. 65.10.

Interestingly, it turns out that the modeling capability of the PCA construction is comparable to autoencoders that employ nonlinearities. To see this, we collect all feature vectors into the $N \times M$ matrix (where $N \geq M$):

$$H = \begin{bmatrix} h_0 & h_1 & \ldots & h_{N-1} \end{bmatrix}^{\mathsf{T}}, \quad (N \times M) \tag{65.106}$$

and formulate the problem of seeking an approximation for H of rank $M' < M$ that is

optimal in the following sense:

$$\widehat{H} \triangleq \underset{X}{\text{argmin}} \, \|H - X\|_{\text{F}}^2, \;\; \text{subject to } \text{rank}(X) = M' \tag{65.107}$$

Then, we know from (57.49) that the solution is constructed as follows. We introduce the singular value decomposition:

$$H = V \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} U^{\mathsf{T}} \tag{65.108}$$

where V is $N \times N$ orthogonal, U is $M \times M$ orthogonal, and Σ is $M \times M$. We let U_1 denote the leading $M \times M'$ submatrix of U. Then, it holds that

$$\widehat{H} = HU_1U_1^{\mathsf{T}} \tag{65.109}$$

In other words, the optimal approximation for H is given by the PCA construction. It involves applying U_1 followed by U_1^{T} to the input matrix H, as already described by (65.105).

Example 65.6 (**Graph neural network**) A standing assumption in our treatment of feedforward neural networks will be that all nodes from one layer are connected to all nodes in the subsequent layer. There are variations, however, in the form of *graph neural networks* where sparse connections are allowed. We describe one of them here to illustrate the main idea. Consider for instance the relations:

$$z_{\ell+1} = W_\ell^{\mathsf{T}} y_\ell - \theta_\ell, \qquad y_{\ell+1} = f(z_{\ell+1}) \tag{65.110a}$$

which map the output vector y_ℓ for the $\ell-$th layer to the output vector $y_{\ell+1}$ for the subsequent layer. If we use j to index the individual entries, then we can rewrite the above relations more explicitly as

$$z_{\ell+1}(j) = \sum_{i=1}^{n_\ell} w_{ij}^{(\ell)} y_\ell(i) - \theta_\ell(j), \quad y_{\ell+1}(j) = f(z_{\ell+1}(j)) \tag{65.111a}$$

in terms of the weights $\{w_{ij}^{(\ell)}\}$ that appear on the $j-$row of W_ℓ^{T}. It is seen that all $n_\ell-$outputs $\{y_\ell(i)\}$ from layer ℓ contribute to the formation of each $y_{\ell+1}(j)$. We may consider instead a sparse structure where only a subset of the nodes from the previous layer contribute to $y_{\ell+1}(j)$. For example, let $\mathcal{N}_j^{(\ell+1)}$ denote the subset of nodes from layer ℓ that contribute to $y_{\ell+1}(j)$. Then, we can replace the above expressions by writing

$$z_{\ell+1}(j) = \sum_{i \in \mathcal{N}_j^{(\ell+1)}} w_{ij}^{(\ell)} y_\ell(i) - \theta_\ell(j) \tag{65.112a}$$

$$y_{\ell+1}(j) = f(z_{\ell+1}(j)) \tag{65.112b}$$

In this way, procedure (65.31) for propagating signals forward through the network will need to be adjusted to (65.114).

For the backward recursions, we start from relation (65.61), which shows how to update the sensitivity factors for two consecutive layers:

$$\delta_\ell = f'(z_\ell) \odot (W_\ell \delta_{\ell+1}) \tag{65.113}$$

Feedforward propagation: graph neural network with L layers

given a feedforward network with L layers (input+output+hidden);
given neighborhoods $\{\mathcal{N}_j^{(\ell+1)}\}$ for every node j in every layer $\ell > 1$;
start with $y_1 = h$;
repeat for $\ell = 1, \ldots, L-1$:
  for $j = 1, 2, \ldots, n_{\ell+1}$:

$$\begin{aligned} z_{\ell+1}(j) &= \sum_{i \in \mathcal{N}_j^{(\ell+1)}} w_{ij}^{(\ell)} y_\ell(i) - \theta_\ell(j) \\ y_{\ell+1}(j) &= f(z_{\ell+1}(j)) \end{aligned} \tag{65.114}$$

  end
end
$z = z_L$
$\widehat{\gamma} = y_L$

If we again use j to index individual entries we have

$$\delta_\ell(j) = f'\left(z_\ell(j)\right) \odot \left(\sum_{i=1}^{n_{\ell+1}} w_{ji}^{(\ell)} \delta_{\ell+1}(i) \right) \tag{65.115}$$

in terms of the weights $\{w_{ji}^{(\ell)}\}$ that appear on the $j-$th row of W_ℓ. It is seen once more that all entries of $\delta_{\ell+1}$ contribute to the formation of $\delta_\ell(j)$. Motivated by the derivation (65.58)–(65.60), we consider a similar sparse construction and use instead

$$\delta_\ell(j) = f'\left(z_\ell(j)\right) \odot \left(\sum_{i=1}^{n_{\ell+1}} \mathbb{I}\Big[j \in \mathcal{N}_i^{(\ell+1)}\Big] w_{ji}^{(\ell)} \delta_{\ell+1}(i) \right) \tag{65.116}$$

In this way, the backward recursions appearing in procedure (65.87) to propagate signals back through the network will need to be adjusted to (65.119). If the neighborhoods happen to satisfy the symmetry condition:

$$j \in \mathcal{N}_{i+1}^{(\ell+1)} \iff i \in \mathcal{N}_{j+1}^{(\ell+1)} \tag{65.117}$$

then we can simplify (65.116) to

$$\delta_\ell(j) = f'\left(z_\ell(j)\right) \odot \left(\sum_{i \in \mathcal{N}_{j+1}^{(\ell+1)}} w_{ji}^{(\ell)} \delta_{\ell+1}(i) \right) \tag{65.118}$$

Other variations are possible.

Backward propagation: graph neural network with L layers

for every pair (γ, h), feed h forward through network using (65.114);
determine $\{z_\ell, y_\ell\}$ across layers and output $\widehat{\gamma}$;
$\delta_L = 2(\widehat{\gamma} - \gamma) \odot f'(z)$.
repeat for $\ell = L-1, \ldots, 2, 1$**:**
  $\theta_\ell \leftarrow \theta_\ell + \mu\delta_{\ell+1}$
  for $j = 1, 2, \ldots, n_\ell$**:**

$$\delta_\ell(j) \leftarrow f'(z_\ell(j)) \odot \left(\sum_{i=1}^{n_{\ell+1}} \mathbb{I}\Big[j \in \mathcal{N}_i^{(\ell+1)}\Big] w_{ji}^{(\ell)} \delta_{\ell+1}(i) \right), \quad \ell \geq 2 \tag{65.119}$$

  end
  for $j = 1, 2, \ldots, n_{\ell+1}$**:**
    for $i \in \mathcal{N}_j^{(\ell+1)}$**:**
      $w_{ij}^{(\ell)} \leftarrow (1 - 2\mu\rho) w_{ij}^{(\ell)} \; - \; \mu y_\ell(i) \delta_{\ell+1}(j)$
    end
  end
end

65.6 DROPOUT STRATEGY

We described in Sec. 62.1 the bagging technique for enhancing the performance of classifiers and combating overfitting. The same technique can in principle be applied to neural networks. As explained in that section, bagging is based on the idea of training multiple networks, by sampling from the *same* dataset, and on combining the classification decisions, e.g., by taking a majority vote. In order to ensure variability across the networks, the training data for each network is obtained by sampling from the original dataset *with* replacement. While the bagging procedure is justified for simple network architectures, it can nevertheless become expensive for networks with many hidden layers that require training a large number of combination weights and offset coefficients. One useful alternative to reduce overfitting and provide a form of regularization is to employ the *dropout* method.

We refer to the mini-batch implementation (65.82), which trains a network with L layers. Each iteration m involves feeding a batch of B randomly-selected data points $\{\gamma_b, h_b\}$ through the network and adjusting its parameters from $\{W_{\ell,m-1}, \theta_{\ell,m-1}\}$ to $\{W_{\ell,m}, \theta_{\ell,m}\}$. Dropout is based on the idea that during each iteration m, only a random fraction of the connections in the network are retained and their parameters adjusted. The thinning operation is achieved by switching a good portion of the nodes into *sleeping mode* where nodes are *turned off* with probability p (usually, p is close to $1/2$ for internal nodes and close to 0.1 for the input nodes). When a node is turned off, it does not feed any signals into subsequent layers and its incoming and outgoing combination weights and

bias coefficient are frozen and not adjusted during the iteration. The other active nodes in the network participate in the training and operate as if the sleeping nodes do not exist. The ultimate effect is that the size of the original network is scaled down by p (e.g., halved when $p = 1/2$). Only the combination weights and bias coefficients of the active nodes are adjusted by the backpropagation algorithm (65.82). This situation is illustrated in Fig 65.11.

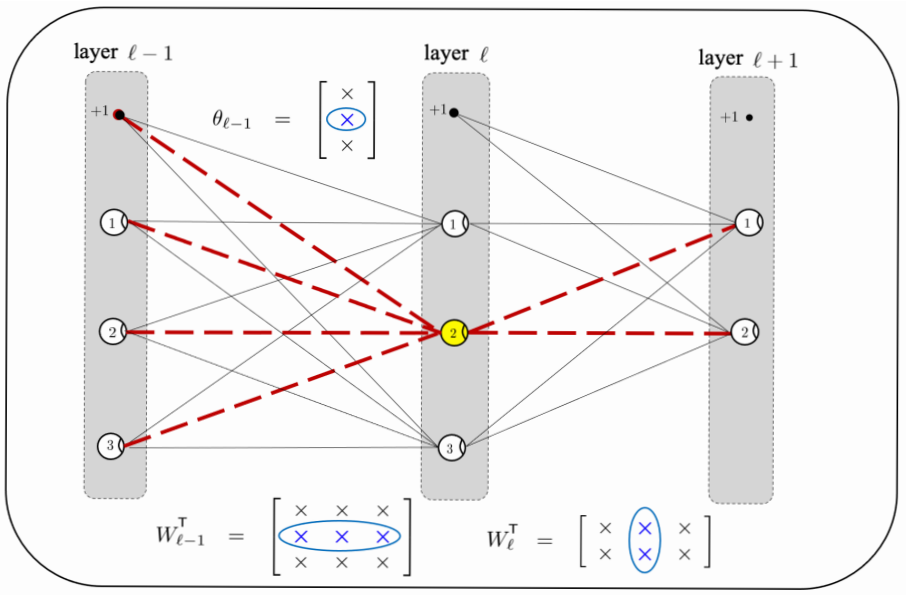

Figure 65.11 A succession of three layers where node 2 in layer ℓ is turned off at random. The dashed lines arriving at this node from the preceding layer $\ell - 1$, and also leaving from it towards layer $\ell + 1$, represent the weighting and bias coefficients that will be frozen and not adjusted by the training algorithm. These coefficients correspond to one column in W_{ℓ}^{T}, one row in $W_{\ell-1}^{\mathsf{T}}$, and one entry in $\theta_{\ell-1}$.

Forward propagation

The training of the network proceeds as follows. During each iteration m, we associate a Bernoulli variable with each node in a generic layer ℓ: it is equal to zero with probability p_ℓ and one with probability $1 - p_\ell$. The variable will be zero when the node is turned off. We collect the Bernoulli variables for layer ℓ into a vector $a_\ell \in \{0, 1\}^{n_\ell}$ and denote it by writing

$$a_\ell \stackrel{\Delta}{=} \text{Bernoulli}(p_\ell, n_\ell) \tag{65.120}$$

This notation means that a_ℓ is a vector of dimension $n_\ell \times 1$ and its entries are Bernoulli variables, each with success probability $1-p_\ell$. Then, during the forward propagation of signals through the feedforward network by means of algorithm

(65.31), the expression for $z_{\ell+1}$ will be modified to:

$$z_{\ell+1} = W_\ell^{\mathsf{T}} (y_\ell \odot a_\ell) - \theta_\ell \tag{65.121}$$

where the Hadamard product is used to annihilate the output signals from sleeping nodes. During normal operation of the forward step, the quantities $\{y_\ell, z_{\ell+1}\}$ in this expression will be computed for every index b within a mini-batch, while the parameters $\{W_\ell, \theta_\ell, a_\ell\}$ will be the ones available at the start of iteration m, so that we should write more explicitly:

$$z_{\ell+1,b} = W_{\ell,m-1}^{\mathsf{T}} (y_{\ell,b} \odot a_{\ell,m}) - \theta_{\ell,m-1}, \quad b = 0, 1, \ldots, B-1 \tag{65.122}$$

The Bernoulli vectors $\{a_{\ell,m}\}$ only vary with m and, therefore, remain fixed for all samples within the $B-$size mini-batch. In other words, the thinned network structure remains invariant during the processing of each mini-batch of samples.

Backward propagation

During the backward step of the training algorithm, at iteration m, only the combination weights and bias coefficients of active nodes are updated while the combination weights and bias coefficients of sleeping nodes remain intact. For example, in the context of Fig. 65.11, the second column of $W_{\ell,m-1}^{\mathsf{T}}$, the second row of $W_{\ell-1,m-1}^{\mathsf{T}}$, and the second entry of $\theta_{\ell-1,m-1}$ are not updated; they remain frozen at their existing values within the new $W_{\ell,m}$, $W_{\ell-1,m}$, and $\theta_{\ell-1,m}$. At the next iteration, $m+1$, involving a new mini-batch of samples, the process is repeated. A new collection of Bernoulli vectors $\{a_{\ell,m+1}\}$ is generated resulting in a new thinned network. The batch of B samples is fed forward through this network and its thinned parameters adjusted to $\{W_{\ell,m+1}, \theta_{\ell,m+1}\}$, and so on.

If we repeat the derivation of the backpropagation algorithm under the dropout condition, we arrive at listing (65.124). Again, if the softmax construction (65.39) is employed at the output layer, then the expression for the boundary condition $\boldsymbol{\delta}_{L,b}$ would be replaced by

$$\boldsymbol{\delta}_{L,b} = 2\boldsymbol{J}(\widehat{\boldsymbol{\gamma}}_b - \boldsymbol{\gamma}_b), \quad \boldsymbol{J} = \mathrm{diag}(\widehat{\boldsymbol{\gamma}}_b) - \widehat{\boldsymbol{\gamma}}_b \widehat{\boldsymbol{\gamma}}_b^{\mathsf{T}} \tag{65.123}$$

Observe that the recursions in the forward and backward passes are similar to the implementation without dropout, with the main difference being the incorporation of the Bernoulli vector $\boldsymbol{a}_{\ell,m}$ in three locations: during the generation of $\boldsymbol{z}_{\ell+1,b}$ in the forward pass, and during the generation of $\boldsymbol{W}_{\ell,m}$ and $\boldsymbol{\delta}_{\ell,b}$ in the backward pass. The net effect of the dropout step is the following:

(a) During the backward pass, at the first iteration corresponding to $\ell = L-1$, the columns of $\boldsymbol{W}_{L-1,m}^{\mathsf{T}}$ corresponding to sleeping nodes in $\boldsymbol{a}_{L-1,m}$ (i.e., to its zero entries) would not be updated and stay at the values in $\boldsymbol{W}_{L-1,m-1}^{\mathsf{T}}$.

(b) For the subsequent stages $\ell = L-2, L-3, \ldots, 1$, the columns of $\boldsymbol{W}_{\ell,m}^{\mathsf{T}}$ corresponding to sleeping nodes in $\boldsymbol{a}_{\ell,m}$ are not updated and stay at the values in $\boldsymbol{W}_{\ell,m-1}^{\mathsf{T}}$. Likewise, the rows of $\boldsymbol{W}_{\ell,m}^{\mathsf{T}}$ corresponding to sleeping nodes in $\boldsymbol{a}_{\ell+1,m}$ are not updated and stay at the values in $\boldsymbol{W}_{\ell,m-1}^{\mathsf{T}}$. In the

same token, all entries in $\boldsymbol{\theta}_{\ell,m}$ corresponding to the sleeping nodes in $\boldsymbol{a}_{\ell+1,m}$ are not updated and stay at the values in $\boldsymbol{\theta}_{\ell,m-1}$.

Mini-batch backpropagation for solving (65.45) with dropout

given a feedforward network with L layers (input+output+hidden);
pre- and post-activation signals at output layer are $(z_n, \widehat{\gamma}_n)$;
internal pre- and post-activation signals are $\{z_{\ell,n}, y_{\ell,n}\}$;
given N training data samples $\{\gamma_n, h_n\}$, $n = 0, 1, \ldots, N-1$;
given Bernoulli probabilities $\{p_\ell\}, \ell = 1, 2, \ldots, L-1$;
given small step-size $\mu > 0$ and regularization parameter $\rho \geq 0$;
start from random initial parameters $\{\boldsymbol{W}_{\ell,-1}, \boldsymbol{\theta}_{\ell,-1}\}$.
repeat until convergence over $m = 0, 1, 2, \ldots$:
  select B random data pairs $\{\boldsymbol{\gamma}_b, \boldsymbol{h}_b\}$
  generate $\boldsymbol{a}_{\ell,m} = \text{Bernoulli}(p_\ell, n_\ell)$, $\ell = 1, 2, \ldots, L-1$

  (*forward processing*)
  repeat for $b = 0, 1, \ldots, B-1$**:**
    $\boldsymbol{y}_{1,b} = \boldsymbol{h}_b$
    repeat for $\ell = 1, 2, \ldots, L-1$
      $\boldsymbol{z}_{\ell+1,b} = \boldsymbol{W}_{\ell,m-1}^{\mathsf{T}}(\boldsymbol{y}_{\ell,b} \odot \boldsymbol{a}_{\ell,m}) - \boldsymbol{\theta}_{\ell,m-1}$
      $\boldsymbol{y}_{\ell+1,b} = f(\boldsymbol{z}_{\ell+1,b})$
    end
    $\widehat{\boldsymbol{\gamma}}_b = \boldsymbol{y}_{L,b}$
    $\boldsymbol{z}_b = \boldsymbol{z}_{L,b}$
    $\boldsymbol{\delta}_{L,b} = 2(\widehat{\boldsymbol{\gamma}}_b - \boldsymbol{\gamma}_b) \odot f'(\boldsymbol{z}_b)$
  end

  (*backward processing*)
  repeat for $\ell = L-1, \ldots, 2, 1$**:**

$$\boldsymbol{W}_{\ell,m} = (1 - 2\mu\rho)\boldsymbol{W}_{\ell,m-1} \;-\; \frac{\mu}{B}\sum_{b=0}^{B-1}(\boldsymbol{y}_{\ell,b} \odot \boldsymbol{a}_{\ell,m})\boldsymbol{\delta}_{\ell+1,b}^{\mathsf{T}}$$

$$\boldsymbol{\theta}_{\ell,m} = \boldsymbol{\theta}_{\ell,m-1} + \frac{\mu}{B}\sum_{b=0}^{B-1}\boldsymbol{\delta}_{\ell+1,b}$$

$$\boldsymbol{\delta}_{\ell,b} = \left\{f'(\boldsymbol{z}_{\ell,b}) \odot \left(\boldsymbol{W}_{\ell,m-1}\boldsymbol{\delta}_{\ell+1,b}\right)\right\} \odot \boldsymbol{a}_{\ell,m},\ \ell \geq 2,\ \forall b$$

  end
end

$$\{W_\ell^\star, \theta_\ell^\star\} \leftarrow \left\{(1-p_\ell)\,\boldsymbol{W}_{\ell,m},\ (1-p_\ell)\,\boldsymbol{\theta}_{\ell,m}\right\}$$

(65.124)

The convergence time of the dropout implementation is expected to be worse

than an implementation without dropout. Once training is completed, all nodes in the network are activated again for testing purposes without any dropout. As indicated in the last line of the algorithm, the combination weights and bias coefficients for the network will be set to scaled versions of the parameter values obtained at the end of training. If a particular node was dropped with probability p during training, then its outgoing combination weights will need to be scaled by $1-p$ (e.g., they should be scaled by $2/3$ if $p = 1/3$) in order to account for the fact that these weights were determined with only a fraction of the nodes active. By doing so, we are in effect averaging the performance of a collection of randomly thinned networks, in a manner that mimics the bagging technique, except that the training was performed on successive networks with a reduced number of parameters. As a result, the possibility of overfitting is reduced since each iteration involves training a sparse version of the network.

65.7 REGULARIZED CROSS-ENTROPY RISK

The derivations in the earlier sections illustrated the training of feedforward neural networks by minimizing the regularized least-squares empirical risk (65.47). Other risk functions are of course possible. In this section we examine one popular scheme that is suitable for *multiclass* classification problems where the label vector γ is assumed to be *one-hot encoded*. That is, the entries of γ assume binary values in $\{0,1\}$ with only one entry equal to one while all other entries are zero:

$$\gamma \in \{0,1\}^Q \tag{65.125}$$

In this setting, there are Q classes and a feature vector $h \in \mathbb{R}^M$ can belong to one of the classes. The corresponding label $\gamma \in \mathbb{R}^Q$ will have the form of a basis vector and the location of its unit entry will identify the class of h.

The output layer of the neural network will now be a softmax layer where the entries of the output vector $\widehat{\gamma}$ are computed as follows:

$$\widehat{\gamma}(q) \triangleq e^{z(q)} \left(\sum_{q'=1}^{Q} e^{z(q')} \right)^{-1} \tag{65.126}$$

where the symbol z refers to the pre-activation signal at the last layer. The parameters $\{W_\ell, \theta_\ell\}$ of the network will be determined by minimizing the following regularized cross-entropy empirical risk:

$$\{W_\ell^\star, \theta_\ell^\star\} \triangleq \underset{\{W_\ell, \theta_\ell\}}{\operatorname{argmin}} \left\{ \mathcal{P}(W,\theta) \triangleq \sum_{\ell=1}^{L-1} \rho \|W_\ell\|_{\mathrm{F}}^2 - \frac{1}{N} \sum_{n=0}^{N-1} \sum_{q=1}^{Q} \gamma_n(q) \ln\left(\widehat{\gamma}_n(q)\right) \right\} \tag{65.127}$$

where γ_n is the label vector associated with the $n-$th feature vector h_n and $\widehat{\gamma}_n$ is

the corresponding output vector. The loss function that is associated with each data point in (65.127) is seen to be:

$$\mathcal{Q}(W,\theta;\gamma,h) \triangleq -\sum_{q=1}^{Q}\gamma(q)\ln(\widehat{\gamma}(q)) \tag{65.128}$$

This loss value depends on all weight and bias parameters. For simplicity, we will drop the parameters and write $\mathcal{Q}(\gamma,h)$.

Example 65.7 (**Binary labels**) Consider the special case in which $Q=2$ so that the network has only two outputs, denoted by

$$\widehat{\gamma} = \begin{bmatrix} \widehat{\gamma}(1) \\ \widehat{\gamma}(2) \end{bmatrix} \tag{65.129}$$

These outputs add up to one in view of the softmax calculation (65.126). The feature vectors belong to one of two classes so that

$$\gamma = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{or} \quad \gamma = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{65.130}$$

In this case, expression (65.127) for the empirical risk simplifies to

$$\mathcal{P}(W,\theta) \triangleq \sum_{\ell=1}^{L-1}\rho\|W_\ell\|_{\mathrm{F}}^2 - \frac{1}{N}\sum_{n=0}^{N-1}\Big(\gamma_n(1)\ln(\widehat{\gamma}_n(1)) + (1-\gamma_n(1))\ln(1-\widehat{\gamma}_n(1))\Big) \tag{65.131}$$

where, for each n, only one of the terms inside the rightmost summation is nonzero since either $\gamma_n(1)=0$ or $\gamma_n(1)=1$.

65.7.1 Motivation for Cross-Entropy

We already know from the result of the earlier Example 31.5 that cross-entropy minimization is related to maximum-likelihood inference and to the minimization of the KL divergence between a true distribution and an empirical approximation for it. We provide further motivation for this observation here and explain how it applies to the choice of risk function in (65.127).

Consider two probability mass functions, denoted generically by $p_{\boldsymbol{x}}(x)$ and $s_{\boldsymbol{x}}(x)$, for a random variable $\boldsymbol{x}$. Their cross-entropy is denoted by $H(p,s)$ and defined as the value:

$$\begin{aligned} H(p,s) &\triangleq -\mathbb{E}_p \log_2 s_{\boldsymbol{x}}(x) \\ &= -\sum_x p_{\boldsymbol{x}}(x)\log_2 s_{\boldsymbol{x}}(x) \\ &= -\sum_x \mathbb{P}_p(\boldsymbol{x}=x)\log_2 \mathbb{P}_s(\boldsymbol{x}=x) \end{aligned} \tag{65.132}$$

where the sum is over the discrete realizations for $\boldsymbol{x}$, and the notation $\mathbb{P}_p(\boldsymbol{x}=x)$ refers to the probability that event $\boldsymbol{x}=x$ occurs under the discrete distribution

$p_{\boldsymbol{x}}(x)$. Similarly, for $\mathbb{P}_s(\boldsymbol{x}=x)$ under distribution $s_{\boldsymbol{x}}(x)$. 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,

$$H(p,s) \;=\; H(p) + D_{\mathrm{KL}}(p,s) \tag{65.133}$$

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

$$H(p) \;\triangleq\; -\sum_{x} \mathbb{P}_p(\boldsymbol{x}=x)\log_2 \mathbb{P}_p(\boldsymbol{x}=x) \tag{65.134}$$

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

To illustrate the relevance of this conclusion in the context of classification problems, let us consider a multiclass classification scenario with Q classes where the label vectors $\gamma \in \mathbb{R}^Q$ are one-hot encoded. For example, if a feature vector h belongs to some class r, then the $r-$th entry of its label vector γ will be equal to one while all other entries will be zero. Let $\widehat{\gamma}$ denote the output vector that is generated by the neural network for this feature vector. Recall that the output layer is based on a softmax calculation and, hence, we can interpret $\widehat{\gamma}$ as defining a probability distribution over the class variable, denoted by

$$\mathbb{P}_s(\boldsymbol{r}=q) = \widehat{\gamma}(q), \quad q=1,2,\ldots,Q \tag{65.135}$$

Likewise, if γ is the true label vector, we can use its entries to define a second probability distribution on the same class variable, denoted by

$$\mathbb{P}_p(\boldsymbol{r}=q) = \gamma(q), \quad q=1,2,\ldots,Q \tag{65.136}$$

Since only one entry of the vector γ is equal to one, this second distribution will be zero everywhere except at location $q=r$. Ideally, we would like the distribution that results from $\widehat{\gamma}$ to match the distribution from γ. The cross-entropy between the two distributions is given by

$$H(p,s) = -\sum_{q=1}^{Q} \gamma(q)\log_2 \widehat{\gamma}(q) \tag{65.137}$$

This result is the reason for the form of the rightmost term in (65.127), where the outer sum is over all training samples.

Example 65.8 (**Log loss function**) The rightmost term in (65.127) is often referred to as the *log-loss function*. We denote it by

$$\mathrm{LogLoss} \;\triangleq\; -\frac{1}{N}\sum_{n=0}^{N-1}\sum_{q=1}^{Q} \gamma_n(q)\ln\left(\widehat{\gamma}_n(q)\right) \tag{65.138}$$

which in view of the above discussion has the following interpretation in terms of the class variables:

$$\mathrm{LogLoss} \;\triangleq\; -\frac{1}{N}\sum_{n=0}^{N-1}\sum_{q=1}^{Q} \mathbb{I}[r(h_n)=q]\,\mathbb{P}(r^{\star}(h_n)=q) \tag{65.139}$$

Here, the indicator term $\mathbb{I}[r(h_n) = q]$ is equal to one or zero depending on whether the true label for feature h_n is q, while the term $\mathbb{P}(r^\star(h_n) = q)$ represents the probability that the classifier will assign label q to h_n. The notation $r(h_n)$ and $r^\star(h_n)$ refers to the true and assigned labels for feature h_n, respectively; they both assume integer values in the set $\{1, 2, \ldots, Q\}$.

65.7.2 Sensitivity Factors

We return to problem (65.127) where the objective is to minimize the risk function over the parameters $\{W_\ell, \theta_\ell\}$. The derivation that follows is meant to show that the same backpropagation algorithm from before will continue to apply, where the only adjustment that is needed is in the value of the boundary sensitivity vector.

We drop the subscript n for convenience of the derivation; we restore it in the listing of the algorithm. As before, we associate with each layer ℓ a *sensitivity* vector of size n_ℓ denoted by $\delta_\ell \in \mathbb{R}^{n_\ell}$ and with entries $\{\delta_\ell(j)\}$ defined by

$$\delta_\ell(j) \triangleq \frac{\partial \mathcal{Q}(\gamma, h)}{\partial z_\ell(j)}, \quad j = 1, 2, \ldots, n_\ell \tag{65.140}$$

in terms of the partial derivative of the loss function (65.128). We can derive a recursive update for the vector δ_ℓ. We consider first the output layer for which $\ell = L$. The chain rule for differentiation gives

$$\delta_L(j) \triangleq \frac{\partial \mathcal{Q}(\gamma, h)}{\partial z(j)} = -\frac{\partial}{\partial z(j)} \left(\sum_{q=1}^{Q} \gamma(q) \ln(\widehat{\gamma}(q)) \right) \tag{65.141}$$

Using the fact that

$$\frac{\partial \ln(\widehat{\gamma}(q))}{\partial z(j)} = \begin{cases} 1 - \widehat{\gamma}(j), & q = j \\ -\widehat{\gamma}(j), & q \neq j \end{cases} \tag{65.142}$$

we find that

$$\delta_L(j) = \widehat{\gamma}(j) \underbrace{\Big(\sum_{q=1}^{Q} \gamma(q) \Big)}_{=1} - \gamma(j) = \widehat{\gamma}(j) - \gamma(j) \tag{65.143}$$

In other words, the boundary sensitivity vector is now given by

$$\boxed{\delta_L = \widehat{\gamma} - \gamma} \tag{65.144}$$

Next, repeating the same arguments after (65.57b), we can similarly derive a backward recursion for updating the sensitivity vectors as

$$\boxed{\delta_\ell = f'(z_\ell) \odot (W_\ell \delta_{\ell+1})} \tag{65.145}$$

along with the same gradient expressions:

$$\frac{\partial \mathcal{P}(W,\theta)}{\partial W_\ell} = 2\rho W_\ell \;+\; \frac{1}{N}\sum_{n=0}^{N-1} y_{\ell,n}\delta_{\ell+1,n}^{\mathsf{T}} \tag{65.146a}$$

$$\frac{\partial \mathcal{P}(W,\theta)}{\partial \theta_\ell} = -\frac{1}{N}\sum_{n=0}^{N-1} \delta_{\ell+1,n} \tag{65.146b}$$

In summary, the same backpropagation algorithm (65.82), and its dropout version (65.124), continue to hold with the main difference being that the boundary condition for $\boldsymbol{\delta}_{L,b}$ should be replaced by

$$\boldsymbol{\delta}_{L,b} = \widehat{\boldsymbol{\gamma}}_b - \boldsymbol{\gamma}_b \tag{65.147}$$

Recall that this conclusion assumes a neural network with softmax construction at the output layer, and one-hot encoded label vectors of the form $\gamma \in \{0,1\}^Q$.

When the batch size is $B = 1$, the mini-batch recursions (65.149) simplify to the listing shown in (65.150). In both implementations, the parameters $\{W_\ell^\star, \theta_\ell^\star\}$ at the end of the training phase are used for testing purposes. For example, assume a new feature vector h is fed into the network and let $\widehat{\gamma}$ denote the corresponding output vector. We declare the class of h to be the index that corresponds to the largest value within $\widehat{\gamma}$:

$$r^\star = \underset{1\le q\le Q}{\text{argmax}}\;\; \widehat{\gamma}(q) \tag{65.148}$$

For ease of reference, we collect in Table 65.2 the boundary values for the sensitivity factor under different scenarios.

Table 65.2 List of terminal sensitivity factors for different risk functions and/or network structure.

risk function	output layer	boundary sensitivity factor, $\boldsymbol{\delta}_{L,m}$
least-squares (65.47)	activation, $f(\cdot)$	$2(\widehat{\gamma}_m - \gamma_m) \odot f'(z_m)$
least-squares (65.47)	softmax (65.39)	$2J(\widehat{\gamma}_m - \gamma_m)$, $J = \text{diag}(\widehat{\gamma}_m) - \widehat{\gamma}_m(\widehat{\gamma}_m)^{\mathsf{T}}$
cross-entropy (65.127)	softmax (65.39)	$\widehat{\gamma}_m - \gamma_m$

Mini-batch backpropagation algorithm for solving (65.127)

given a feedforward network with L layers (input+output+hidden);
pre- and post-activation signals at output layer are $(z_n, \widehat{\gamma}_n)$;
internal pre- and post-activation signals are $\{z_{\ell,n}, y_{\ell,n}\}$;
given N training data samples $\{\gamma_n, h_n\},\ n = 0, 1, \ldots, N-1$;
given small step-size $\mu > 0$ and regularization parameter $\rho \geq 0$;
start from random initial parameters $\{\boldsymbol{W}_{\ell,-1}, \boldsymbol{\theta}_{\ell,-1}\}$.
repeat until convergence over $m = 0, 1, 2, \ldots$**:**
  select B random data pairs $\{\boldsymbol{\gamma}_b, \boldsymbol{h}_b\}$

  (*forward processing*)
  repeat for $b = 0, 1, \ldots, B-1$**:**
    $\boldsymbol{y}_{1,b} = \boldsymbol{h}_b$
    repeat for $\ell = 1, 2, \ldots, L-1$ **:**
      $\boldsymbol{z}_{\ell+1,b} = \boldsymbol{W}_{\ell,m-1}^{\mathsf{T}} \boldsymbol{y}_{\ell,b} - \boldsymbol{\theta}_{\ell,m-1}$
      $\boldsymbol{y}_{\ell+1,b} = f(\boldsymbol{z}_{\ell+1,b})$
    end
    $\boldsymbol{z}_b = \boldsymbol{z}_{L,b}$
    $\widehat{\boldsymbol{\gamma}}_b = \mathrm{softmax}(\boldsymbol{z}_b)$
    $\boldsymbol{\delta}_{L,b} = \widehat{\boldsymbol{\gamma}}_b - \boldsymbol{\gamma}_b$
  end

  (*backward processing*)
  repeat for $\ell = L-1, \ldots, 2, 1$ **:**

$$\boldsymbol{W}_{\ell,m} = (1 - 2\mu\rho)\boldsymbol{W}_{\ell,m-1} \ - \ \frac{\mu}{B}\sum_{b=0}^{B-1} \boldsymbol{y}_{\ell,b}\boldsymbol{\delta}_{\ell+1,b}^{\mathsf{T}}$$

$$\boldsymbol{\theta}_{\ell,m} = \boldsymbol{\theta}_{\ell,m-1} + \frac{\mu}{B}\sum_{b=0}^{B-1} \boldsymbol{\delta}_{\ell+1,b}$$

$$\boldsymbol{\delta}_{\ell,b} = f'(\boldsymbol{z}_{\ell,b}) \odot \Big(\boldsymbol{W}_{\ell,m-1}\boldsymbol{\delta}_{\ell+1,b}\Big),\ \ell \geq 2, b = 0, 1, \ldots, B-1$$

  end
end
$\{W_\ell^\star, \theta_\ell^\star\} \leftarrow \{\boldsymbol{W}_{\ell,m}, \boldsymbol{\theta}_{\ell,m}\}$

(65.149)

Stochastic-gradient backpropagation for solving (65.127)

given a feedforward network with L layers (input+output+hidden);
pre- and post-activation signals at output layer are $(z_n, \widehat{\gamma}_n)$;
internal pre- and post-activation signals are $\{z_{\ell,n}, y_{\ell,n}\}$;
given N training data samples $\{\gamma_n, h_n\}$, $n = 0, 1, \ldots, N-1$;
given small step-size $\mu > 0$ and regularization parameter $\rho \geq 0$;
start from random initial parameters $\{\boldsymbol{W}_{\ell,-1}, \boldsymbol{\theta}_{\ell,-1}\}$.
repeat until convergence over $m = 0, 1, 2, \ldots$**:**

select one random data pair $(\boldsymbol{h}_m, \boldsymbol{\gamma}_m)$

$$\boldsymbol{y}_{1,m} = \boldsymbol{h}_m$$

(*forward processing*)

repeat for $\ell = 1, 2, \ldots, L-1$**:**

$$\begin{aligned} \boldsymbol{z}_{\ell+1,m} &= \boldsymbol{W}_{\ell,m-1}^{\mathsf{T}} \boldsymbol{y}_{\ell,m} - \boldsymbol{\theta}_{\ell,m-1} \\ \boldsymbol{y}_{\ell+1,m} &= f(\boldsymbol{z}_{\ell+1,m}) \end{aligned}$$

end

$$\begin{aligned} \boldsymbol{z}_m &= \boldsymbol{z}_{L,m} \\ \widehat{\boldsymbol{\gamma}}_m &= \mathrm{softmax}(\boldsymbol{z}_m) \\ \boldsymbol{\delta}_{L,m} &= \widehat{\boldsymbol{\gamma}}_m - \boldsymbol{\gamma}_m \end{aligned}$$

(*backward processing*)

repeat for $\ell = L-1, \ldots, 2, 1$**:**

$$\begin{aligned} \boldsymbol{W}_{\ell,m} &= (1 - 2\mu\rho)\boldsymbol{W}_{\ell,m-1} \;-\; \mu \boldsymbol{y}_{\ell,m} \boldsymbol{\delta}_{\ell+1,m}^{\mathsf{T}} \\ \boldsymbol{\theta}_{\ell,m} &= \boldsymbol{\theta}_{\ell,m-1} + \mu \boldsymbol{\delta}_{\ell+1,m} \\ \boldsymbol{\delta}_{\ell,m} &= f'(\boldsymbol{z}_{\ell,m}) \odot \Big(\boldsymbol{W}_{\ell,m-1} \boldsymbol{\delta}_{\ell+1,m} \Big), \quad \ell \geq 2 \end{aligned}$$

end

end

(65.150)

Example 65.9 (**Classification of handwritten digits**) We illustrate the operation of a neural network by applying it to the problem of identifying handwritten digits using the same MNIST dataset considered earlier in Example 52.3. The dataset consists of 60,000 labeled training samples and 10,000 labeled testing samples. Each entry in the dataset is a 28×28 grayscale image, which we transform into an $M = 784-$long feature vector, h_n. Each pixel in the image and, therefore, each entry in h_n, assumes integer values in the range $[0, 255]$. Every feature vector (or image) is assigned an integer label in the range 0-9 depending on which digit the image corresponds to. The earlier Fig. 52.6, which is repeated here, shows randomly selected images from the training dataset.

We pre-process the images (or the corresponding feature vectors $\{h_n\}$) by scaling their entries by 255 (so that they assume values in the range $[0, 1]$). We subsequently com-

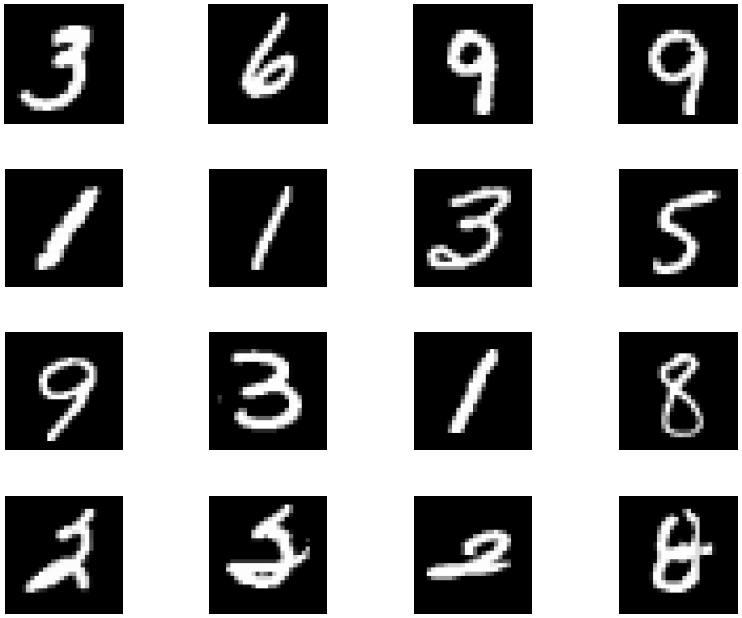

Figure 65.12 Randomly selected images from the MNIST dataset for handwritten digits. Each image is 28×28 grayscale with pixels assuming integer values in the range $[0, 255]$.

pute the mean feature vectors for both the training and test sets. We center the scaled feature vectors around these means in both sets. The earlier Fig. 52.7 showed randomly selected images for the digits $\{0, 1\}$ before and after processing.

We construct a neural network with a total of 4 layers: one input layer, one output layer, and two hidden layers. The size of the input layer is $n_1 = 784$ (which agrees with the size of the feature vectors), while the size of the output layer is $n_4 = 10$ (which agrees with the number of classes). The size of the hidden layers is set to $n_2 = n_3 = 512$ neurons. We employ a softmax layer at the output and train the network using a regularized cross-entropy criterion with parameters

$$\mu = 0.001, \quad \rho = 0.0001 \tag{65.151}$$

We run $P = 200$ passes of the stochastic-gradient algorithm (65.150) over the training data, with the data being randomly reshuffled at the start of each pass. At the end of the training phase, we evaluate the empirical error rate over the 10,000 test samples, as well as over the 60,000 training samples. We simulate three different scenarios where we vary the nonlinearity at the output of the internal neurons: sigmoid, rectifier, and tanh. We also simulate a dropout implementation with $p_1 = 0.1$ for the input layer and $p_2 = p_3 = 0.5$ for the two hidden layers; in this last simulation, we use the sigmoid activation function for the internal nodes and perform the same number of 200 passes over the data. The results are summarized in Table 65.3. The performance under dropout can be improved by using a larger number of passes due to the slower convergence in this case.

Example 65.10 (**Classification of tiny color images**) We again illustrate the operation of neural networks by applying them to the problem of classifying color images into

Table 65.3 The table lists the empirical error rates over both the test and training samples from the MNIST dataset for three types of internal nonlinearities: sigmoid, rectifier, and tanh. The last row in the table corresponds to a dropout implementation using 200 passes over the data, the sigmoid activation function, and putting 50% of the neurons in the hidden layers to sleep at each iteration.

nonlinearity	empirical test error (%)	number of test errors	empirical training error (%)	number of training errors
sigmoid	2.18%	218	1.02%	613
tanh	1.84%	184	0.00167%	1
rectifier	1.82%	182	0.00167%	1
dropout	6.22%	622	6.25%	3752

one of ten classes using the CIFAR-10 dataset. This dataset consists of color images that can belong to one of 10 classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Figure 65.13 shows random selections of images from the dataset. The images in the dataset have low resolution and that is why they appear blurred.

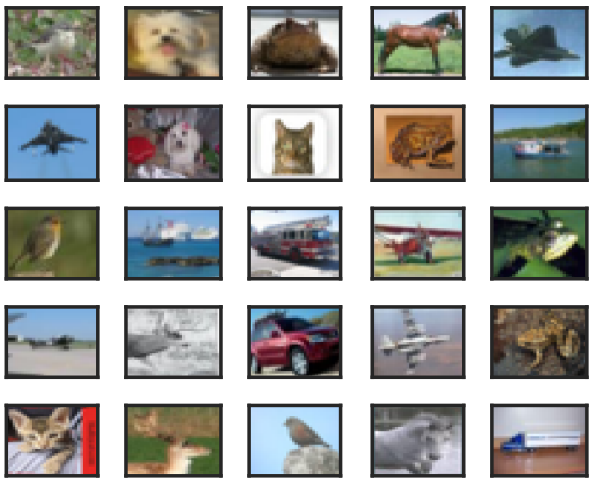

Figure 65.13 Randomly selected color images from the CIFAR-10 dataset. Each image has 3 channels (red, green, blue) of size 32×32 each. The pixels in each channel assume integer values in the range $[0, 255]$. The CIFAR-10 dataset is found at http://www.cs.toronto.edu/~kriz/cifar.html.

There are 6,000 images per class for a total of 60,000 images in the dataset. There are 50,000 training images and 10,000 test images. There are 1,000 random images from each class in the test collection of 10,000 images. The training images are divided into 5 batches of 10,000 images each. Each training batch may contain more images from one class or another.

Each image has size 32×32 in the red, green, and color channels, which we transform into an $M = 32 \times 32 \times 3 = 3072-$long feature vector, h_n. Each pixel in the image assumes integer values in the range $[0, 255]$. Each feature vector (or image) is assigned

an integer class label in the range 0-9. We pre-process the images (or the corresponding feature vectors $\{h_n\}$) by scaling their entries by 255 (so that they assume values in the range $[0, 1]$). We subsequently compute the mean feature vectors for the training and test sets and center the scaled feature vectors in both sets around these means.

We construct a neural network with a total of 4 layers: one input layer, one output layer, and two hidden layers. The size of the input layer is $n_1 = 3072$ (which agrees with the size of the feature vectors), while the size of the output layer is $n_4 = 10$ (which agrees with the number of classes). The size of the hidden layers is set to $n_2 = n_3 = 2048$ neurons. We employ a softmax layer at the output of the network, and rectifier units at the internal neurons. We train the network using a regularized cross-entropy criterion with parameters

$$\mu = 0.001, \quad \rho = 0.0001 \tag{65.152}$$

We run a stochastic-gradient version of the backpropagation algorithm (65.82) with mini-batches of size equal to one sample, and adjusted to the cross-entropy scenario where the boundary condition $\boldsymbol{\delta}_{L,b}$ is replaced by

$$\boldsymbol{\delta}_{L,b} = \widehat{\boldsymbol{\gamma}}_b - \boldsymbol{\gamma}_b \tag{65.153}$$

We run $P = 200$ passes of the stochastic-gradient algorithm (65.150) over the training data, with the data being randomly reshuffled at the start of each pass. At the end of the training phase, we evaluate the empirical error rate over the 10,000 test samples and also over the 50,000 training samples. We also simulate a dropout implementation with $p_1 = 0.1$ for the input layer and $p_2 = p_3 = 0.5$ for the two hidden layers using now $P = 300$ passes over the data. The results are summarized in Table 65.4. It is seen from the results on the test data that this is a more challenging classification problem.

Table 65.4 The table lists the empirical error rates over 10,000 test samples and 50,000 training samples from the CIFAR-10 dataset with and without dropout.

setting	empirical test error (%)	number of test errors	empirical training error (%)	number of training errors
w/ dropout	42.28%	4228	0.02%	10
with dropout	42.92%	4292	5.62%	2810

Example 65.11 (**Transfer learning**) Assume a neural network has been trained to perform a certain task $\mathbb{A}$ such as detecting images of cars ("car" versus "no car"). This assumes that a large amount of training data is available so that the network can be trained well (say, by minimizing a regularized cross-entropy empirical risk) to perform its intended task with minimal classification error. Now assume we wish to train a second neural network to perform another task $\mathbb{B}$, which also involves classifying images, say, detecting whether an image is showing a bird or not. This objective is different from detecting the presence of cars. If we happen to have a sufficient amount of training data under task $\mathbb{B}$, then we could similarly train this second network to perform its task well. However, it may be the case that while we had a large amount of data to train network $\mathbb{A}$, we may only have a limited amount of data to train network $\mathbb{B}$. Transfer learning provides one useful method to transfer the knowledge acquired from training the first network and apply it to assist in training the second network. The approach exploits the fact that the input data to both tasks, $\mathbb{A}$ and $\mathbb{B}$ (i.e, to both neural networks), are of similar type: they are images of the same size. There are of course other methods to transfer learning/knowledge from one situation to another, and we will describe one such method later in Chapter 72 when we study meta learning. Here, we continue with transfer learning.

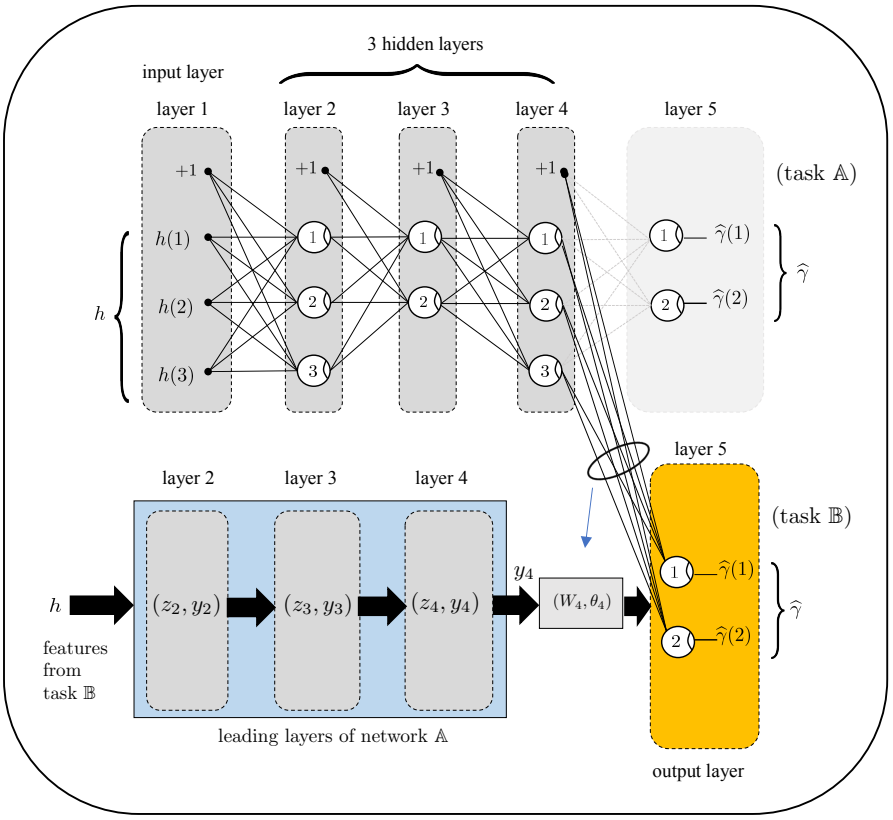

Figure 65.14 The top part shows the neural network for solving task $\mathbb{A}$; it consists of 3 hidden layers. The last layer is replaced, with new parameters (W_4, θ_4), as shown in the lower part of the figure. These parameters are trained using the data $\{\gamma(n), y_{4,n}\}$ for task $\mathbb{B}$.

The main idea is to replace the last output layer of network $\mathbb{A}$ by a new weight matrix and a new bias vector, and to retrain only these last-layer parameters, while keeping the weights and biases from all prior layers fixed at the values obtained during the training of network $\mathbb{A}$. The main reason why this approach works reasonably well is because the earlier layers from network $\mathbb{A}$ have been well trained to identify many low-level features (such as edges, texture) that continue to be useful for task $\mathbb{B}$. It is generally the last layer that is responsible for performing the final step of prediction or classification in a network. We can therefore limit our training for network $\mathbb{B}$ by re-training the weight matrix and bias vector for this last layer using the data $\{\gamma(n), y_{4,n}\}$ from network $\mathbb{B}$. Some variations are possible:

(a) We can use the training data available for task $\mathbb{B}$ to train the last layer only, while keeping the weights and bias vectors of all prior layers fixed at the values obtained from training network $\mathbb{A}$. This is the approach described above.

(b) Once the training from step (a) is concluded, we can consider fine-tuning all weight and bias vector parameters across all layers by using the training data from task $\mathbb{B}$. That is, we can retrain all parameters starting from their current values as initial conditions.

(c) Under step (a), we can consider replacing the last layer of network $\mathbb{A}$ by two or

more layers and retrain these using the data from task $\mathbb{B}$.

The situation under option (a) is illustrated in Fig. 65.14 for a network $\mathbb{A}$ with three hidden layers shown in the top part of the figure. The new weight matrix and bias vector of the last layer are denoted by (W_4, θ_4) and shown in the lower part of the figure. The hidden layers of network $\mathbb{B}$ continue to be the same as those trained under network $\mathbb{A}$. Fixing the parameters of these earlier layers, we can feed the training data $\{\gamma(n), h_n\}$ under task $\mathbb{B}$ and generate realizations $\{y_{4,n}\}$ for the vector y_4 shown in network $\mathbb{B}$. We are then faced with the problem of training a single-layer neural network: its training data are $\{\gamma(n), y_{4,n}\}$ and its output are $\{\widehat{\gamma}(1), \widehat{\gamma}(2)\}$ in the lower part of the figure. The parameters to be trained are (W_4, θ_4).

Example 65.12 (**Multitask learning**) The objective in multitask learning is to design a *single* neural network to identify the presence or absence of several labels at once, as is the case with multilabel classification. For example, the purpose may be to examine an image and to indicate which of the following objects appear in the image: a car, a street, a traffic signal, a pedestrian, or a bicycle. The network should be able to detect the presence of several of these objects simultaneously, such as indicating that the image shows a pedestrian, a stop sign, and a bicycle. In principle, if we have sufficient amount of training data for each situation, then we could design 5 separate neural networks: one for detecting cars in images, a second one for detecting streets, a third one for detecting traffic signals, a fourth one for detecting pedestrians, and a fifth one for detecting bicycles. Once trained, these networks would operate separately.

Multitask learning provides an alternative approach to the problem; it relies on training a *single* network to detect the presence of any of the objects of interest *simultaneously*. This is possible when the multiple tasks benefit from some shared low-level features (such as edges or texture), and the amount of training data available for each task is more or less uniform.

Assume there are T separate tasks. Motivated by expression (65.131), one way to design a multitask neural network is to minimize an empirical risk function of the form:

$$\mathcal{P}(W,\theta) \triangleq \sum_{\ell=1}^{L-1} \rho \|W_\ell\|_{\mathrm{F}}^2 - \frac{1}{N}\sum_{n=0}^{N-1}\sum_{t=1}^{T}\Big(\gamma_{n,t}(1)\ln(\widehat{\gamma}_{n,t}(1)) + (1-\gamma_{n,t}(1))\ln(1-\widehat{\gamma}_{n,t}(1))\Big) \tag{65.154}$$

where we associate a pair of outputs $\{\widehat{\gamma}_{n,t}(1), \widehat{\gamma}_{n,t}(2)\}$ with each task t; these outputs continue to add up to one because they are defined by applying the softmax construction to their respective pre-activation signals, denoted by $\{z_{n,t}(1), z_{n,t}(2)\}$:

$$\widehat{\gamma}_{n,t}(q) = \frac{e^{z_{n,t}(q)}}{e^{z_{n,t}(1)} + e^{z_{n,t}(2)}}, \quad q = 1, 2 \tag{65.155}$$

Thus, the value of $\widehat{\gamma}_{n,t}(1)$ represents the likelihood that the $n-$th feature vector contains the attribute that is present under task t. If desired, it is sufficient to have a single output $\widehat{\gamma}_{n,t}(1)$ associated with each task t. We continue with $2-$dimensional label vectors to remain consistent with the assumed one-hot encoding formulation from the earlier sections.

Figure 65.15 shows a network structure for a two-task learning problem ($T = 2$); we are dropping the iteration index n from the variables in the figure. The network consists of three hidden layers and one output layer. The top part shows the network with output vectors $\widehat{\gamma}_1 \in \mathbb{R}^2$ for task $t = 1$ and $\widehat{\gamma}_2 \in \mathbb{R}^2$ for task $t = 2$ (i.e., the subscripts here refer to the task number). The lower part of the figure shows in greater detail the

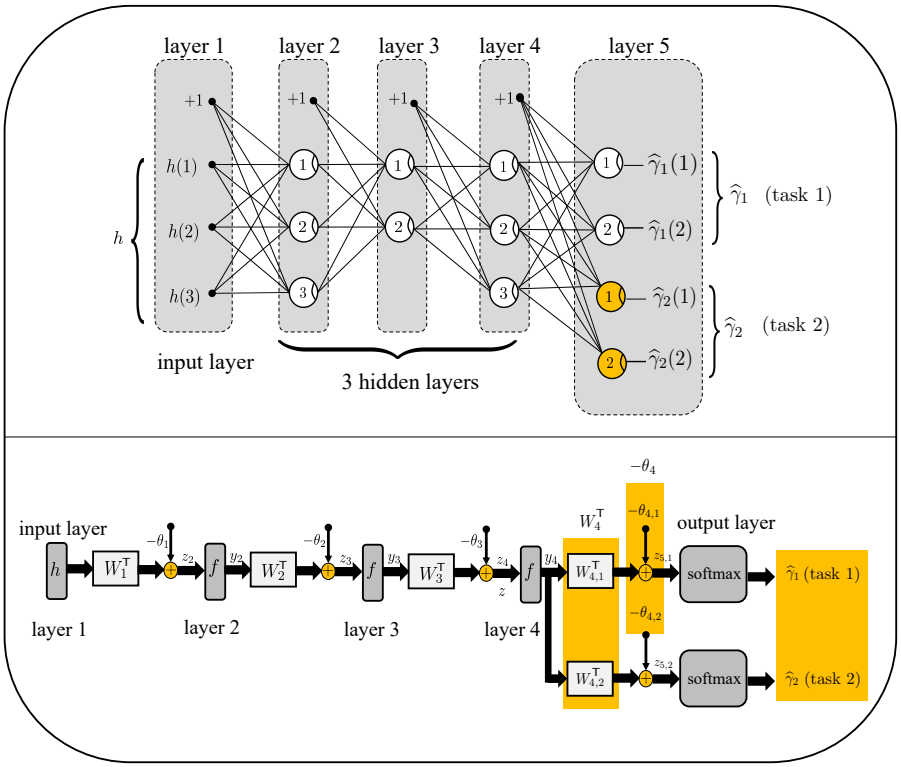

Figure 65.15 The top part shows a neural network for solving a two-task problem with outputs $\widehat{\gamma}_1$ for task $t = 1$ and $\widehat{\gamma}_2$ for task $t = 2$. The lower part of the figure shows the forward processing of signals through the layers. In the last layer, and for added clarity, we are splitting the weight matrix W_4 and the bias vector θ_4 into two components to highlight the parts that relate to the output vectors for the separate tasks.

forward processing of signals through the layers. In the last layer, we are splitting for illustration purposes the weight matrix W_4 and the bias vector θ_4 into two components to highlight the parts that relate to the output vectors for the separate tasks:

$$\underbrace{\begin{bmatrix} z_{5,1} \\ \hline z_{5,2} \end{bmatrix}}_{z_5} = \underbrace{\begin{bmatrix} W_{4,1}^{\mathsf{T}} \\ \hline W_{4,2}^{\mathsf{T}} \end{bmatrix}}_{=W_4^{\mathsf{T}}} y_4 - \underbrace{\begin{bmatrix} \theta_{4,1} \\ \hline \theta_{4,2} \end{bmatrix}}_{\theta_4} \tag{65.156a}$$

$$\widehat{\gamma}_1 = \mathrm{softmax}(z_{5,1}), \quad \text{for task } t = 1 \tag{65.156b}$$

$$\widehat{\gamma}_2 = \mathrm{softmax}(z_{5,2}), \quad \text{for task } t = 2 \tag{65.156c}$$

The same algorithms (65.149) or (65.150) will continue to be valid for minimizing (65.155) with the main difference being that the boundary sensitivity factor (65.144)

should be replaced by — see Prob. 65.21:

$$\delta_L = \mathrm{col}\Big\{\widehat{\gamma}_t - \gamma_t\Big\}_{t=1}^{T} = \begin{bmatrix} \widehat{\gamma}_1 - \gamma_1 \\ \widehat{\gamma}_2 - \gamma_2 \\ \vdots \\ \widehat{\gamma}_T - \gamma_T \end{bmatrix} \tag{65.157}$$

where $\widehat{\gamma}_t$ denotes the output vector for task t and γ_t the corresponding one-hot encoded label; each of the vectors γ_t and $\widehat{\gamma}_t$ has dimensions 2×1 and their entries add up to one:

$$\widehat{\gamma}_t \triangleq \begin{bmatrix} \widehat{\gamma}_t(1) \\ \widehat{\gamma}_t(2) \end{bmatrix}, \quad \gamma_t = \begin{bmatrix} \gamma_t(1) \\ \gamma_t(2) \end{bmatrix} \tag{65.158}$$

with $\widehat{\gamma}_t(1) + \widehat{\gamma}_t(2) = 1$ and $\gamma_t(q) \in \{0, 1\}$ for $q = 1, 2$. Moreover, in a manner similar to (65.140), the sensitivity factors for this problem are now defined by

$$\delta_\ell(j) \triangleq -\frac{\partial}{\partial z_\ell(j)} \left\{ \sum_{t=1}^{T} \Big(\gamma_t(1) \ln(\widehat{\gamma}_t(1)) + (1 - \gamma_t(1)) \ln(1 - \widehat{\gamma}_t(1)) \Big) \right\} \tag{65.159}$$

with the subscript n dropped for convenience of notation.

Consider an example with two tasks, $T = 2$, where the network is tasked with detecting whether images have instances of cats (task 1) and dogs (task 2) in them. For example, a training sample h with an aggregate label vector of the form

$$\gamma \triangleq \begin{bmatrix} \gamma_1 \\ \hline \gamma_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ \hline 1 \\ 0 \end{bmatrix} \tag{65.160}$$

corresponds to an image that has both a cat and a dog in it, while

$$\gamma \triangleq \begin{bmatrix} \gamma_1 \\ \hline \gamma_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ \hline 1 \\ 0 \end{bmatrix} \tag{65.161}$$

corresponds to an image with only a dog in it. It may happen that not all training samples are completely labeled. For example, some images may have been labeled in relation to the presence of dogs in them, but without paying attention to whether cats are present. For instance, one image in the training set may have an aggregate label vector of the form

$$\gamma \triangleq \begin{bmatrix} \gamma_1 \\ \hline \gamma_2 \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ \hline 1 \\ 0 \end{bmatrix} \tag{65.162}$$

where the labeling under task 1 is missing and represented by the question marks. Multitask learning allows us to perform training even when some labeling information is missing. To do so, we modify the empirical risk function and redefine it as follows:

$$\mathcal{P}(W, \theta) \triangleq \sum_{\ell=1}^{L-1} \rho \|W_\ell\|_{\mathrm{F}}^2 - \tag{65.163}$$

$$\frac{1}{N} \sum_{n=0}^{N-1} \sum_{t=1}^{T} \mathbb{I}[\gamma_{n,t} \in \mathrm{valid}] \Big(\gamma_{n,t}(1) \ln(\widehat{\gamma}_{n,t}(1)) + (1 - \gamma_{n,t}(1)) \ln(1 - \widehat{\gamma}_{n,t}(1)) \Big)$$

where we added an indicator function that assumes the value one when the label vector

for the $t-$th task has valid entries in $\{0,1\}$. In this way, in the sum over tasks, only samples that have valid labels are taken into account. The ultimate effect on the training algorithm is that the boundary sensitivity factor (65.157) would be replaced by

$$\delta_L = \mathrm{col}\Big\{\mathbb{I}[\gamma_t \in \text{valid}]\Big(\widehat{\gamma}_t - \gamma_t\Big)\Big\}_{t=1}^{T} \tag{65.164}$$

65.8 SLOWDOWN IN LEARNING

There is one important impairment that arises in the training of feedforward networks with a large number of hidden layers (such as deep networks; these may include dozens or hundreds of layers and millions of parameters). The impairment is already evident from examining the backward recursion (65.63) for updating the sensitivity vectors, $\delta_{\ell,n}$. The recursion shows that the flow of information back to the earlier layers is hindered by saturation effects that cause gradient values, which depend on $\delta_{\ell,n}$ through (65.74) and (65.80), to become small or negligible.

To see this effect more clearly, consider an arbitrary combination weight, $w_{ij}^{(\ell)}$, in some internal layer of index ℓ. It is clear from recursion (65.87) for $\delta_{\ell,m}$ that, starting from the terminal vector $\delta_{L,m}$ and propagating it backwards, the entries in $\delta_{\ell,m}$ will involve a product of derivatives of the activation function, $f'(\cdot)$, at successive output signals, namely,

$$\text{entries of } \delta_{\ell,m} \;\propto\; f'\left(z_{\ell,m}\right) f'\left(z_{\ell+1,m}\right) \ldots f'\left(z_{L,m}\right) \tag{65.165}$$

For the sigmoid and hyperbolic tangent functions listed in Table 65.1 it holds that

$$\begin{cases} f'(x) = f(x)(1-f(x)), & 0 \leq f'(x) \leq 1/4, \quad (\textbf{sigmoid}) \\ f'(x) = 1/\cosh(x), & 0 \leq f'(x) \leq 1, \quad (\textbf{tanh}) \end{cases} \tag{65.166}$$

such that products of a large collection of derivative values for these functions can result in a small number, especially when some nodes are close to saturation levels where the corresponding derivative values $f'(\cdot)$ will be practically zero. In general, saturation is more likely to occur at the output nodes resulting in a small $f'(z_{L,m})$. This problem, known as the *vanishing gradient problem*, is more pronounced for the sigmoid activation function since its derivative function is bounded by the smaller value of $1/4$ rather than one. When the entries of $\delta_{\ell,m}$ become small, the updates for $W_{\ell,m}$ and $\theta_{\ell,m}$ in algorithm (65.87) end up evolving slowly. Observe, in particular, that this effect is magnified for the earlier layers of the network due to the backward nature of the recursion for $\delta_{\ell,m}$: as we move further back into the earlier layers, more terms are included in the product (65.165) and it is more likely that it will assume small values. This means that earlier layers in the network will learn at a relatively slower rate compared to the subsequent layers.

These observations help explain why other choices for the activation function are considered, such as the rectifier function from Table 65.1, although this function still suffers from the problem of turning off learning for negative values of its argument, z. Note further that when the output layer of the network is modified to include the softmax construction (65.39), then the boundary sensitivity factor, $\delta_{L,m}$, given by expression (65.70) does not depend on the terminal derivative term $f'(z_{L,m})$. This fact helps ameliorate the vanishing gradient problem but does not resolve it completely because the sensitivity factors for the earlier (hidden) layers will continue to depend on the derivatives $f'(z_{\ell,m})$.

Autoencoders and RBMs

There are several mechanisms by which the difficulties arising from the gradient vanishing problem can be ameliorated. Some of the techniques are more successful than others. In the next chapter, we will describe two mechanisms that have been used in earlier implementations of neural networks: one is based on cascading layers of autoencoders, while the other is based on cascading layers of restricted Boltzmann machines (RBMs). This latter method was originally devised for training deep belief networks and was subsequently observed to also provide good initialization for feedforward networks. In the context of the gradient vanishing problem, the purpose of these methods is to generate convenient *initial conditions* for the network parameters $\{W_\ell, \theta_\ell\}$ from which training of the network by backpropagation can subsequently be launched. These two methods (autoencoders and RBMs), however, are not as effective or popular today, as the more powerful approach of batch normalization described in the next section. Nevertheless, we will still discuss autoencoders and RBMs because these techniques have independent value of their own. In particular, deep belief networks serve as *generative network models*.

65.9 BATCH NORMALIZATION

The batch normalization method is effective in speeding up the convergence of neural networks and ameliorating the degrading effect of the vanishing gradient problem. We motivate the method by referring to the *mini-batch* stochastic gradient implementation (65.82) of the backpropagation algorithm. During any iteration m, the signal that feeds forward into layer $\ell+1$ is denoted by $\boldsymbol{y}_{\ell,b} \in \mathbb{R}^{n_\ell}$ with b representing the index within the mini-batch of size B. This signal generates the output signal, $\boldsymbol{y}_{\ell+1,b}$, for layer $\ell+1$ via the calculations:

$$\boldsymbol{z}_{\ell+1,b} = \boldsymbol{W}_{\ell,m-1}^{\mathsf{T}} \boldsymbol{y}_{\ell,b} - \boldsymbol{\theta}_{\ell,m-1} \tag{65.167a}$$

$$\boldsymbol{y}_{\ell+1,b} = f(\boldsymbol{z}_{\ell+1,b}) \tag{65.167b}$$

in terms of the parameters $\{\boldsymbol{W}_{\ell,m-1}, \boldsymbol{\theta}_{\ell,m-1}\}$ that scale the signals feeding into the nonlinearities in layer $\ell+1$. These parameters are updated during the back-

ward processing step and therefore evolve with the iteration index m. Recall that $\boldsymbol{z}_{\ell+1,b}$ and $\boldsymbol{y}_{\ell+1,b}$ are vectors of size $n_{\ell+1}$ each. Likewise, $\boldsymbol{W}_{\ell,m-1}$ has dimensions $n_\ell \times n_{\ell+1}$ while $\boldsymbol{\theta}_{\ell,m-1}$ has dimension $n_{\ell+1}$.

Now note that the statistical distribution of the signals $\{\boldsymbol{y}_{\ell+1,b}, \boldsymbol{z}_{\ell+1,b}\}$, including the range of values for their individual entries, change continually during training as we iterate over m. This change is due to two effects: (a) the change in the parameters $\{\boldsymbol{W}_{\ell,m-1}, \boldsymbol{\theta}_{\ell,m-1}\}$, which scale the signals feeding directly into layer $\ell+1$, and (b) the change in the parameters $\{\boldsymbol{W}_{\ell',m-1}, \boldsymbol{\theta}_{\ell',m-1}\}$ for the earlier layers as well. This constant change in the statistical distribution of the signals in the internal layers in response to changes in the values of the parameters helps compound the saturation effect in the presence of nonlinearities due to changes that occur in the dynamic range of the signals. The same change in the statistical distribution of the data in the internal layers interferes with the task of locating the minimizing (or optimal) parameters that the network is seeking, which further contributes to slowdown in convergence.

65.9.1 Centering and Normalization

Batch normalization is based on the idea of normalizing the signals feeding into each layer so that their range of values would fall within controlled bounds and away from saturation effects. Doing so would enable the use of saturating nonlinearities without serious concerns. As we explain in the sequel, the normalization is achieved by scaling the input signals at each layer by certain amounts, the values of which will also be learned (i.e., adjusted) by the training algorithm. Since the values of the scaling constants will be learned by relying on batches of data, the resulting procedure is referred to as *batch normalization*. It has been observed through experimental validation that batch normalization generally eliminates the need for dropout and speeds up the training of neural networks.

More specifically, batch normalization normalizes the individual entries of the vectors $\{\boldsymbol{z}_{\ell+1,b}\}$ by centering them around "zero-mean" and by scaling them to "unit-variance." By performing these transformations at every layer, we end up ensuring that the inputs to all layers will have a uniform "white" distribution. One way to achieve these transformations is to employ a construction similar to the procedure described earlier in Sec. 57.1 to normalize the feature vectors feeding into PCA. Thus, consider a mini-batch of size B involving the signals $\{\boldsymbol{z}_{\ell+1,b},\ b = 0, 1, \ldots, B-1\}$ feeding into the nonlinearities in layer $\ell+1$. We denote the individual entries of each $\boldsymbol{z}_{\ell+1,b}$ by

$$\boldsymbol{z}_{\ell+1,b} = \mathrm{col}\Big\{\boldsymbol{z}_{\ell+1,b}(1),\, \boldsymbol{z}_{\ell+1,b}(2),\, \ldots,\, \boldsymbol{z}_{\ell+1,b}(n_{\ell+1})\Big\} \tag{65.168}$$

with the index running from 1 to $n_{\ell+1}$. Then, each of these entries will first be transformed as follows:

$$\boldsymbol{z}'_{\ell+1,b}(k) \;\leftarrow\; \frac{1}{\sqrt{\sigma^2_{\ell+1}(k)+\epsilon}}\Big(\boldsymbol{z}_{\ell+1,b}(k) - \bar{z}_{\ell+1}(k)\Big) \tag{65.169}$$

where ϵ is a small positive number to avoid division by zero, and where the mean and variance parameters $\{\bar{z}_{\ell+1}(k), \sigma^2_{\ell+1}(k)\}$ are estimated from the realizations within the batch as:

$$\bar{z}_{\ell+1}(k) = \frac{1}{B}\sum_{b=0}^{B-1} z_{\ell+1,b}(k), \quad k = 1, 2, \ldots, n_\ell + 1 \tag{65.170a}$$

$$\sigma^2_{\ell+1}(k) = \frac{1}{B-1}\sum_{b=0}^{B-1}\Big(z_{\ell+1,b}(k) - \bar{z}_{\ell+1}(k)\Big)^2 \tag{65.170b}$$

This construction is illustrated inside the first dotted box in Fig. 65.16 for two successive layers ℓ and $\ell+1$ with $n_\ell = 3$ and $n_{\ell+1} = 2$ internal nodes, respectively. Note in the figure that we have one pair of parameters $\{\bar{z}_{\ell+1}(k), \sigma^2_{\ell+1}(k)\}$ for each entry of $\boldsymbol{z}_{\ell+1,b}$; moreover, this pair is the same for all vectors $\{\boldsymbol{z}_{\ell+1,b}\}$ within the same mini-batch of data of size B, i.e., for $b = 0, 1, \ldots, B-1$.

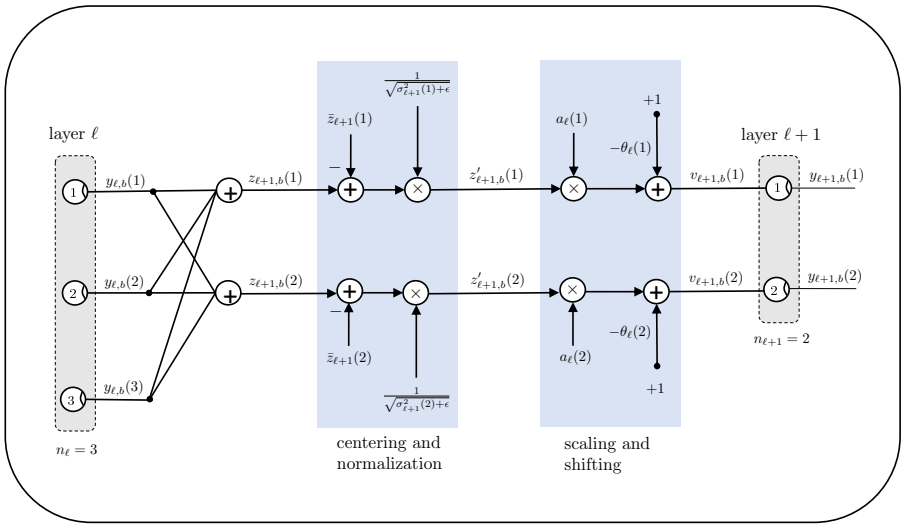

Figure 65.16 Illustration of the centering, normalization, scaling, and shifting operations that are applied to the entries of the internal signals $\boldsymbol{z}_{\ell+1,b}$.

In this way, we end up transforming the vector $\boldsymbol{z}_{\ell+1,b} = \text{col}\{\boldsymbol{z}_{\ell+1,b}(k)\}$ at the input of the nonlinearities for layer $\ell+1$ into the vector $\boldsymbol{z}'_{\ell+1,b} = \text{col}\{\boldsymbol{z}'_{\ell+1,b}(k)\}$. We can represent this transformation in vector form. To do so, we collect the sample means $\{\bar{z}_{\ell+1}(k)\}$ into the vector:

$$\bar{z}_{\ell+1} \triangleq \text{col}\Big\{\bar{z}_{\ell+1}(1),\, \bar{z}_{\ell+1}(2),\, \ldots,\, \bar{z}_{\ell+1}(n_{\ell+1})\Big\} \tag{65.171}$$

which can be calculated directly from the batch vectors $\{z_{\ell+1,b}\}$:

$$\boxed{\bar{z}_{\ell+1} \;=\; \frac{1}{B}\sum_{b=0}^{B-1} z_{\ell+1,b}} \tag{65.172}$$

We also collect the sample variances into the *diagonal* matrix

$$S_{\ell+1}^2 \;\triangleq\; \epsilon\, I_{n_{\ell+1}} + \mathrm{diag}\Big\{\sigma_{\ell+1}^2(1),\,\ldots,\,\sigma_{\ell+1}^2(n_{\ell+1})\Big\} \tag{65.173}$$

Then, it holds that

$$\boxed{\boldsymbol{z}'_{\ell+1,b} \;=\; S_{\ell+1}^{-1}\Big(\boldsymbol{z}_{\ell+1,b} - \bar{z}_{\ell+1}\Big)} \tag{65.174}$$

This normalization is applied to each internal layer, as well as to the input layer (i.e., to the feature vectors $\{h_n\}$).

Observe that, during the training procedure, the values of the sample mean and variance parameters $\{\bar{z}_{\ell+1}(k), \sigma_{\ell+1}^2(k)\}$ vary from one batch of data to another. For this reason, we will smooth them out during the entire training session over the successive mini-batches of data by means of a first-order running filter and generate the smoothed values (with a subscript s):

$$\bar{z}_{\ell+1,s}(k) = \lambda\,\bar{z}_{\ell+1,s}(k) + (1-\lambda)\,\bar{z}_{\ell+1}(k) \tag{65.175a}$$

$$\sigma_{\ell+1,s}^2(k) = \lambda\,\sigma_{\ell+1,s}^2(k) + (1-\lambda)\,\sigma_{\ell+1}^2(k) \tag{65.175b}$$

Here, the scalar λ is a smoothing parameter assuming values close to one, say, $\lambda = 0.95$. These smoothed values will be used *during testing*. That is, following training, the parameters of the neural network will be fixed at their learned values, whereas the mean and variance parameters for the successive layers will be set to the smoothed values $\{\bar{z}_{\ell+1,s}(k), \sigma_{\ell+1,s}^2(k)\}$. We can rewrite the above recursions in vector and matrix forms as well, in terms of smoothed quantities $\bar{\boldsymbol{z}}_{\ell+1,s}$ and $S_{\ell+1,s}$:

$$\bar{z}_{\ell+1,s} = \lambda\,\bar{z}_{\ell+1,s} + (1-\lambda)\,\bar{z}_{\ell+1} \tag{65.176a}$$

$$S_{\ell+1,s}^2 = \lambda\,S_{\ell+1,s}^2 + (1-\lambda)\,S_{\ell+1}^2 \tag{65.176b}$$

65.9.2 Scaling and Shifting

The normalization (65.174) is insufficient because it increases the likelihood that the transformed entries of $\boldsymbol{z}'_{\ell+1,b}$ will assume values within the linear regime of the nonlinearities, thus reducing the modeling power of the neural network. To overcome this possibility, we apply a second linear transformation to the normalized values by scaling and shifting them as follows:

$$\boldsymbol{z}''_{\ell+1,b}(k) \;\leftarrow\; a_\ell(k)\boldsymbol{z}'_{\ell+1,b}(k) - \theta_\ell(k) \tag{65.177}$$

for some scalars $\{a_\ell(k), \theta_\ell(k)\}$ to be learned by the training algorithm (along with the weights $\{W_\ell\}$). The ultimate effect is to transform $\boldsymbol{z}'_{\ell+1,b} = \text{col}\{\boldsymbol{z}'_{\ell+1,b}(k)\}$ into a new vector $\boldsymbol{z}''_{\ell+1,b} = \text{col}\{\boldsymbol{z}''_{\ell+1,b}(k)\}$. Note again that we have a pair of parameters $\{a_\ell(k), \theta_\ell(k)\}$ for each entry of $\boldsymbol{z}'_{\ell+1,b}$, which means that we have $n_{\ell+1}$ such pairs of parameters for $\boldsymbol{z}'_{\ell+1,b}$. The same parameters will be used for all entries within the mini-batch of size B (and will only be updated from one mini-batch to another). Note further that if we set

$$a_\ell(k) \leftarrow \Big(\sigma^2_{\ell+1}(k) + \epsilon\Big)^{1/2}, \quad \theta_\ell(k) \leftarrow -\bar{z}_{\ell+1}(k) \tag{65.178}$$

then the above transformation leads to $\boldsymbol{z}''_{\ell+1,b}(k) = \boldsymbol{z}_{\ell+1,b}(k)$. In other words, these particular choices for $\{a_\ell(k), \theta_\ell(k)\}$ lead us back to the original setting with the input vector $\boldsymbol{z}_{\ell+1,b}$ without any modification to the internal signals in the network. Continuing, we collect the scaling factors $\{a_\ell(k), \theta_\ell(k)\}$ into the matrix and vector quantities:

$$a_\ell = \text{col}\Big\{a(1), a(2), \ldots, a(n_{\ell+1})\Big\}, \quad (n_{\ell+1} \times 1) \tag{65.179}$$

$$\theta_\ell = \text{col}\Big\{\theta(1), \theta(2), \ldots, \theta(n_{\ell+1})\Big\}, \quad (n_{\ell+1} \times 1) \tag{65.180}$$

$$A_\ell = \text{diag}\{a_\ell\}, \quad (n_{\ell+1} \times n_{\ell+1} \text{ diagonal matrix}) \tag{65.181}$$

We are reusing the symbol θ_ℓ here because there will be no need anymore in the batch normalization implementation for the separate bias correction term $-\boldsymbol{\theta}_{\ell,m-1}$ appearing in (65.167a). This is because the correction by the $\theta_\ell(k)$ in (65.177) will achieve the same effect. Using the notation (A_ℓ, θ_ℓ), the transformation from $\boldsymbol{z}'_{\ell+1,b}$ to $\boldsymbol{z}''_{\ell+1,b}$ can be written as

$$\boldsymbol{z}''_{\ell+1,b} \;=\; A_\ell\, \boldsymbol{z}'_{\ell+1,b} - \theta_\ell \tag{65.182}$$

For simplicity of notation, we drop the double prime superscript and replace the symbol $\boldsymbol{z}''_{\ell+1,b}$ by $\boldsymbol{v}_{\ell+1,b}$ so that, in terms of the original internal signal,

$$\boxed{\boldsymbol{v}_{\ell+1,b} = A_\ell\, S^{-1}_{\ell+1}\Big(\boldsymbol{z}_{\ell+1,b} \;-\; \bar{z}_{\ell+1}\Big) \;-\; \theta_\ell} \tag{65.183}$$

We therefore started with an input vector $\boldsymbol{z}_{\ell+1,b}$ and transformed it into the vector $\boldsymbol{v}_{\ell+1,b}$ as illustrated schematically by the diagram in Fig. 65.17.

The entries of $\boldsymbol{v}_{\ell+1,b}$ are fed into the subsequent nonlinearities. We express the batch normalization (BN) transformation (65.183) more compactly by writing

$$\boldsymbol{v}_{\ell+1,b} \;=\; \text{BN}\Big(\boldsymbol{z}_{\ell+1,b};\, a_\ell,\, \theta_\ell,\, S_{\ell+1},\, \bar{z}_{\ell+1}\Big) \tag{65.184}$$

We still need to explain how the parameters $\{a_\ell, \theta_\ell\}$ are determined; it turns out that the training algorithm will need to learn the values of $\{W_\ell, \theta_\ell, a_\ell\}$ as opposed to just $\{W_\ell, \theta_\ell\}$. In summary, under batch normalization, relations (65.167a)–

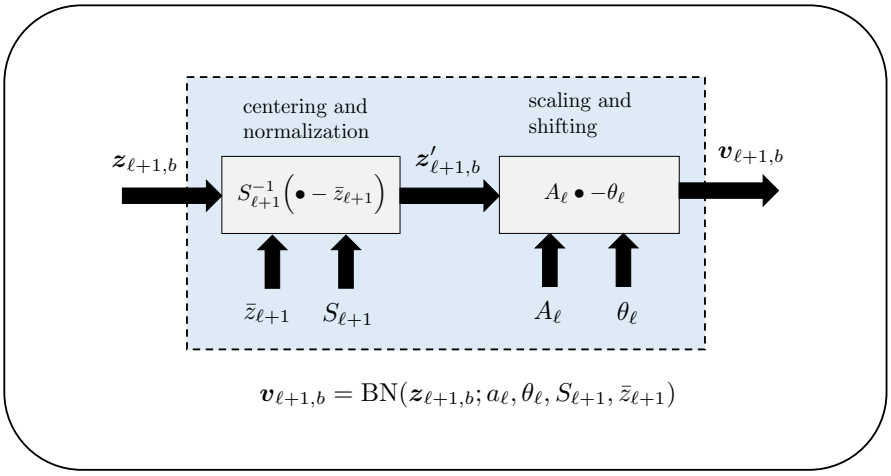

Figure 65.17 Schematic representation of the transformation from $\boldsymbol{z}_{\ell+1,b}$ to $\boldsymbol{v}_{\ell+1,b}$. The first block on the left performs centering and normalization, while the second block performs scaling and shifting.

(65.167b) are replaced by

$$\left\{\begin{array}{l} \boldsymbol{z}_{\ell+1,b} = \boldsymbol{W}_{\ell,m-1}^{\mathsf{T}}\,\boldsymbol{y}_{\ell,b} \\ \text{consider the mini-batch } \mathcal{B} = \{\boldsymbol{z}_{\ell+1,b}\}_{b=0}^{B-1} \text{ of size } B. \\ \text{estimate the sample mean } \bar{z}_{\ell+1} \text{ from } \mathcal{B} \text{ using (65.172)} \\ \text{estimate the sample variance } S_{\ell+1} \text{ from } \mathcal{B} \text{ using (65.173)} \\ \boldsymbol{v}_{\ell+1,b} = \mathrm{BN}\Big(\boldsymbol{z}_{\ell+1,b};\, \boldsymbol{a}_{\ell,m-1},\, \boldsymbol{\theta}_{\ell,m-1},\, S_{\ell+1},\, \bar{z}_{\ell+1}\Big) \\ \boldsymbol{y}_{\ell+1,b} = f(\boldsymbol{v}_{\ell+1,b}) \end{array}\right. \tag{65.185}$$

where $\{\boldsymbol{W}_{\ell,m-1},\, \boldsymbol{\theta}_{\ell,m-1},\, \boldsymbol{a}_{\ell,m-1}\}$ denote the values of the parameters $\{W_\ell, \theta_\ell, a_\ell\}$ at iteration $m-1$. Although the batch normalization step $\mathrm{BN}(\cdot)$ depends on the parameters $(S_{\ell+1}, \bar{z}_{\ell+1})$, these values do not need to be learned recursively by the training algorithm because they are computed as sample averages over the successive mini-batches of data. Observe from (65.185) that each layer $\ell+1$ now receives $\boldsymbol{v}_{\ell+1,b}$ as input rather than $\boldsymbol{z}_{\ell+1,b}$.

65.9.3 Training Algorithm

We derive in Appendix 65.A the algorithm for learning the parameters $\{W_\ell, \theta_\ell, a_\ell\}$ by showing how to adjust the earlier backpropagation recursions. The result is listing (65.187) for minimizing the regularized least-squares risk:

$$\{W_\ell^\star, \theta_\ell^\star, a_\ell^\star\} \;\triangleq\; \underset{\{W_\ell,\theta_\ell,a_\ell\}}{\mathrm{argmin}} \left\{ \mathcal{P}(W,\theta,a) \;\triangleq\; \sum_{\ell=1}^{L-1} \rho \|W_\ell\|_{\mathrm{F}}^2 \;+\; \frac{1}{N}\sum_{n=0}^{N-1} \|\gamma_n - \widehat{\gamma}_n\|^2 \right\} \tag{65.186}$$

Backpropagation with batch normalization for solving (65.186).

given a feedforward network with L layers (input+output+hidden);
given N data samples $\{\gamma_n, h_n\}$ preprocessed by (57.5b);
given small step-size $\mu > 0$ and regularization parameter $\rho \geq 0$;
start from random initial parameters $\{\boldsymbol{W}_{\ell,-1}, \boldsymbol{\theta}_{\ell,-1}\}$;
set initial parameters $\{\boldsymbol{a}_{\ell,-1}\}$ to zero;
set initial smoothed values $\{\bar{z}_{\ell+1,s} = 0, S^2_{\ell+1,s} = \epsilon I_{n_{\ell+1}}\}$;
set $\lambda = 0.95$, $\epsilon = 10^{-6}$ (or similar values).
repeat until convergence over $m = 0, 1, 2, \ldots$:

select B random data pairs $\{\boldsymbol{h}_b, \boldsymbol{\gamma}_b\}$ and set $\boldsymbol{y}_{1,b} = \boldsymbol{h}_b$.
repeat for $\ell = 1, 2, \ldots, L-1$: (*forward processing*)

$$\boldsymbol{z}_{\ell+1,b} = \boldsymbol{W}^{\mathsf{T}}_{\ell,m-1}\boldsymbol{y}_{\ell,b},\ b = 0, 1, \ldots, B-1$$
$$\bar{z}_{\ell+1} = \frac{1}{B}\sum_{b=0}^{B-1}\boldsymbol{z}_{\ell+1,b}$$
$$\sigma^2_{\ell+1}(k) = \frac{1}{B-1}\sum_{b=0}^{B-1}\Big(\boldsymbol{z}_{\ell+1,b}(k) - \bar{z}_{\ell+1}(k)\Big)^2,\ \ k = 1, \ldots, n_{\ell+1}$$
$$S^2_{\ell+1} = \epsilon I_{n_{\ell+1}} + \mathrm{diag}\left\{\sigma^2_{\ell+1}(1), \ldots, \sigma^2_{\ell+1}(n_\ell+1)\right\}$$
$$\bar{z}_{\ell+1,s} = \lambda\bar{z}_{\ell+1,s} + (1-\lambda)\bar{z}_{\ell+1}$$
$$S^2_{\ell+1,s} = \lambda S^2_{\ell+1,s} + (1-\lambda)S^2_{\ell+1}$$
$$\boldsymbol{z}'_{\ell+1,b} = S^{-1}_{\ell+1}\Big(\boldsymbol{z}_{\ell+1,b} - \bar{z}_{\ell+1}\Big)$$
$$\boldsymbol{v}_{\ell+1,b} = \mathrm{diag}\{\boldsymbol{a}_{\ell,m-1}\}\,\boldsymbol{z}'_{\ell+1,b} - \boldsymbol{\theta}_{\ell,m-1}$$
$$\boldsymbol{y}_{\ell+1,b} = f(\boldsymbol{v}_{\ell+1,b})$$

end

set $\widehat{\boldsymbol{\gamma}}_b = \boldsymbol{y}_{L,b},\ \ \boldsymbol{v}_b = \boldsymbol{v}_{L,b},\ \ \boldsymbol{\delta}_{L,b} = 2(\widehat{\boldsymbol{\gamma}}_b - \boldsymbol{\gamma}_b)\odot f'(\boldsymbol{v}_b)$

repeat for $\ell = L-1, \ldots, 2, 1$: (*backward processing*)

$$\boldsymbol{W}_{\ell,m-1} = [\boldsymbol{w}^{(\ell)}_{ij,m-1}],\ i = 1, \ldots, n_\ell,\ j = 1, \ldots, n_{\ell+1}$$
compute $\boldsymbol{c}^{(\ell+1)}_{b,b'}(j)$ from (65.221) for $j = 1, \ldots, n_{\ell+1},\ \forall\, b, b'$
$$\boldsymbol{D}_{\ell,b,m-1} = \mathrm{diag}\left\{\boldsymbol{a}_{\ell,m-1}(j)\boldsymbol{c}^{(\ell+1)}_{b,b}(j)\right\},\ \ (n_{\ell+1}\times n_{\ell+1})$$
$$\boldsymbol{x}^{(\ell)}_{ij,m-1} \ \triangleq\ (1-2\mu\rho)\boldsymbol{w}^{(\ell)}_{ij,m-1}$$
$$\boldsymbol{w}^{(\ell)}_{ij,m} = \boldsymbol{x}^{(\ell)}_{ij,m-1} - \frac{\mu}{B}\sum_{b=0}^{B-1}\boldsymbol{\delta}_{\ell+1,b}(j)\boldsymbol{a}_{\ell,m-1}(j)\left(\sum_{b'=0}^{B-1}\boldsymbol{c}^{(\ell+1)}_{b,b'}(j)\boldsymbol{y}_{\ell,b'}(i)\right)$$
$$\boldsymbol{\theta}_{\ell,m} = \boldsymbol{\theta}_{\ell,m-1} + \frac{\mu}{B}\sum_{b=0}^{B-1}\boldsymbol{\delta}_{\ell+1,b}$$
$$\boldsymbol{a}_{\ell,m} = \boldsymbol{a}_{\ell,m-1} - \frac{\mu}{B}\,\mathrm{diag}\left\{\sum_{b=0}^{B-1}\boldsymbol{\delta}_{\ell+1,b}\left(\boldsymbol{z}'_{\ell+1,b}\right)^{\mathsf{T}}\right\}$$
$$\boldsymbol{\delta}_{\ell,b} = f'(\boldsymbol{v}_{\ell,b})\odot\Big(\boldsymbol{W}_{\ell,m-1}\boldsymbol{D}_{\ell,b,m-1}\boldsymbol{\delta}_{\ell+1,b}\Big),\ \ell \geq 2, \forall b$$

end

end

$\{W^\star_\ell, \theta^\star_\ell, a^\star_\ell, \bar{z}^\star_{\ell+1}, S^\star_{\ell+1}\} \leftarrow \{\boldsymbol{W}_{\ell,m}, \boldsymbol{\theta}_{\ell,m}, \boldsymbol{a}_{\ell,m}, \bar{z}_{\ell+1,s}, S_{\ell+1,s}\}$

(65.187)

The description shows how to adjust the earlier mini-batch listing (65.187), which did not include batch normalization. The listing here assumes that the feature vectors $\{\boldsymbol{h}_n\}$ entering the algorithm have already been centered and normalized in the same manner used earlier for PCA in (57.5b); this step is similar to (65.174) with minimal differences in notation.

We can extend the same batch normalization analysis to the cross-entropy formulation (65.127)–(65.126) and minimize its empirical risk over the augmented parameters $\{W_\ell, \theta_\ell, a_\ell\}$. The same algorithm will continue to hold with the only modification being the boundary condition for the sensitivity factor. Specifically, expression (65.226) would be replaced by

$$\delta_{L,b} \;=\; \widehat{\gamma}_b - \gamma_b, \quad \text{(for cross-entropy risk)} \tag{65.188}$$

Once the neural network is trained, we freeze the coefficients $\{W_\ell, \theta_\ell, a_\ell\}$, and replace the parameters $\{\bar{z}_{\ell+1}(k), \sigma^2_{\ell+1}(k)\}$ by the smoothed versions denoted by $\{\bar{z}_{\ell+1,s}(k), \sigma^2_{\ell+1,s}(k)\}$ and computed via (65.175a)–(65.175b).

65.10 COMMENTARIES AND DISCUSSION

Perceptron and neural networks. We explained in the concluding remarks of Chapter 60 that the Perceptron rule was introduced and implemented into a hardware unit in 1957 by the American psychologist **Frank Rosenblatt (1928–1971)**. Rosenblatt (1957,1958,1962) showed that his Perceptron rule converges to a separating hyperplane if one exists and promoted its potential. Soon thereafter, Minsky and Papert (1969) published a textbook in which they highlighted the modeling limitations of Perceptrons such as their inability to emulate certain logical functions including the XOR function. However, it was realized soon thereafter that *networks* of Perceptron units are able to implement any logical function of binary inputs. This is because the single Perceptron neuron can implement the NAND gate defined in Table 65.5, and NAND gates are known to be universal building blocks for logical functions — see Prob. 60.3. An overview of the history of Perceptron and early developments of the theory of learning and neural networks can be found in the texts by Rosenblatt (1962), Nilsson (1965), and Duda and Hart (1973), as well as in more recent texts by Siu, Roychowdhury, and Kailath (1995), Haykin (1999, 2009), Duda, Hart, and Stork (2000), Theodoridis and Koutroumbas (2008), and Theodoridis (2015), and in the overview article by Widrow and Lehr (1990).

Table 65.5 Input-output mapping of the NAND logical operation.

input a	input b	NAND
0	0	1
0	1	1
1	0	1
1	1	0

Universal approximation theorem. The criticism by Minsky and Papert (1969) motivated a flurry of work on multi-layer neural networks, leading to a powerful *universal*

approximation theorem. Consider a feedforward neural network architecture consisting of a single hidden layer containing a finite number of neurons. Each neuron has a non-constant nonlinearity that is bounded, continuous, and monotonically increasing (such as the sigmoidal function). The output layer of the network consists of a single linear neuron without a nonlinearity. In one of its versions, the universal approximation theorem asserts that such networks have the ability to approximate uniformly any continuous function $g(h) : \mathbb{R}^M \rightarrow [0\,1]$ over any compact subset $\mathcal{S} \subset \mathbb{R}^M$. Specifically, if we denote by $\widehat{g}(h)$ the mapping from the input feature space to the output of the neural network, then a network exists such that

$$|g(h) - \widehat{g}(h)| \; < \; \epsilon, \quad \text{for all } h \in \mathcal{S} \text{ and any small } \epsilon > 0 \tag{65.189}$$

where the neural network mapping has the form — see Fig. 65.18

$$\widehat{g}(h) \; = \; \sum_{j=1}^{K} \alpha(j)\, f\Big(h^{\mathsf{T}} w_j - \theta(j)\Big) \tag{65.190}$$

In this notation, K is the number of neurons in the hidden layer (which we denoted by n_2 in the body of the chapter). We are also using $\alpha(j)$ to designate the weight that scales the signal generated by the $j-$th neuron in the hidden layer and feeding into the output node; these scalars correspond to the notation $w_{j1}^{(2)}$ used in the body of the chapter. We are further using w_j to refer to the weight vector that aggregates the factors that scale the feature entries into the $j-$th neuron in the hidden layer; the entries of w_j correspond to the $j-$th row in the weight matrix W_1^{T} in the notation used in the body of the chapter. Moreover, the offset parameter $\theta(j)$ corresponds to the scalars $\theta_1(j)$ used to denote the offset feeding from the first layer into the $j-$th neuron in the second layer.

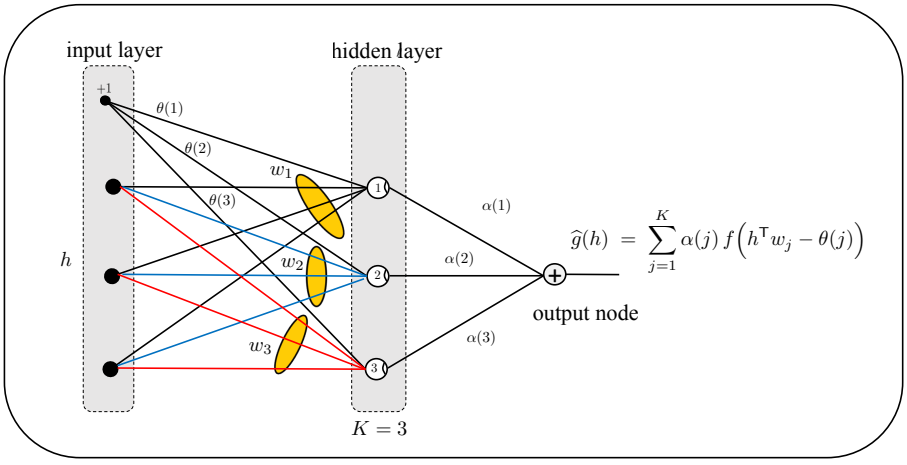

Figure 65.18 Graphical representation of the approximation model (65.190) with $K = 3$ neurons in the hidden layer for illustration purposes. The output node is a simple linear combiner without an activation function.

The universal approximation result was established by Hecht-Nielsen (1989), Hornik, Stinchcombe, and White (1989), Stinhcombe and White (1989), Cybenko (1989), and Funahashi (1989) for sigmoidal activation functions and, more broadly, by Hornik, Stinchcombe and White (1989) and Hornik (1991) for bounded, continuous, and monotonically increasing activation functions. In particular, one of the results from Hornik

(1991) affirms that multilayer feedforward networks can approximate any continuous function on compact subsets of $\mathbb{R}^M$ to arbitrary precision "provided a sufficient number of hidden units are used and the activation functions are continuous, bounded, and non-constant." The arguments in these articles are quite technical for the untrained reader relying on important results from functional analysis such as the *Hahn-Banach theorem* (which allows extending a linear functional defined on a subspace to the entire vector space) and the *Riesz Representation theorem* (which relates linear functionals on a space to measure theory). The work by Leshno *et al.* (1993) provides a generalization that showed, as stated in the abstract of the paper, that any "standard multilayer feedforward network with a locally bounded piecewise continuous activation function can approximate any, continuous function to any degree of accuracy if, and only if, the network's activation function is not a polynomial." This conclusion therefore applies to the important class of ReLU activation functions.

Training of neural networks. The universal approximation theorem implies that the class of feedforward networks is rich enough to model almost any arbitrary function even with a single hidden layer. This is a remarkable conclusion and it has motivated expansive research on neural networks. When trained with the powerful backpropagation algorithm, developed by Werbos (1974,1988,1990,1994) and rediscovered by Rumelhart, Hinton, and Williams (1985,1986), these networks end up providing a rich class of learning machines with powerful variations in the form of deep learning architectures (discussed in the next chapter), autoencoder architectures, and convolutional architectures (also discussed in a future chapter). We introduced autoencoders in Example 65.4. These structures will be used in the design of deep architectures in the next chapter. Some of the early works on the concept of autoencoders are those by Boulard and Kamp (1988), Hinton and Zemel (1994), and Schwenk and Milgram (1995). The first work by Boulard and Kamp (1988) comments on the connection of PCA to autoencoders, as illustrated in Example 65.5. The work by Bengio *et al.* (2006) showed how overcomplete autoencoders can lead to lower classification errors, while the work by Vincent *et al.* (2008) introduced denoising autoencoders by masking a random subset of the input entries for enhanced representation capabilities.

Studies on the use of different activation functions in training neural networks appear in Jarrett *et al.* (2009), Glorot, Bordes, and Bengio (2011), and Goodfellow, Bengio, and Courville (2016). It is indicated in the last reference that the hyperbolic tangent activation function performs better than the sigmoidal function for feedforward and convolutional neural networks, and that the rectifier function yields similar performance to the hyperbolic tangent. Discussions on the initialization of neural networks appear in Duda, Hart, and Stork (2000) and Glorot and Bengio (2010). One of the first studies to explain the origin of the slowdown in learning (also known as the vanishing gradient problem discussed in Sec. 65.8) is the work by Hochreiter (1991); see also Hochreiter *et al.* (2001). An interpretation for the backpropagation procedure as the solution to a min-max optimization problem is given by Hassibi, Sayed, and Kailath (1994a,b).

Some of the earliest discussions on the use of the cross-entropy measure in the training of neural networks appears in Hinton (1987), Solla, Levin, and Fleisher (1988), Bourlard and Wellekens (1989), Zhou and Austin (1989), Bridle (1990a,b), Bourlard and Morgan (1993), Ney (1995), and Bishop (1995). Several subsequent works considered other entropy-based measures such as Linsker (1998), Xu and Principe (1999), Principe, Xu, and Fisher (2000), Erdogmus and Principe (2002). Useful references on the cross-entropy design criterion are Rubinstein and Kroese (2004) and De Boer *et al.* (2005). The use of the cross-entropy risk function is nowadays the preferred design methodology, motivated in large part by its close statistical connection to the maximum-likelihood and KL divergence formalisms as explained earlier in Example 31.5. The performance of the networks has greatly improved with the use of such cross-entropy losses, as well as with the employment of rectified linear units (ReLU) as shown in the works by Jarrett *et al.* (2009) and Glorot, Bordes, and Bengio (2011).

For further reading on the history and impact of these developments, readers may

consult the overviews by Bengio (2009), Bengio, Courville, and Vincent (2013), Schmidhuber (2015), and LeCun, Bengio, and Hinton (2015), and also the pioneering works by Rumelhart, Hinton, and Williams (1986) on backpropagation, LeCun *et al.* (1998) on convolutional (deep) networks, Hinton and Salakhutdinov (2006) on autoencoders, Hinton, Osindero, and Teh (2006) on deep belief networks, and Hinton *et al.* (2012b), Krizhevsky, Sutskever, and Hinton (2012), and Srivastava *et al.* (2014) on the dropout strategy.

Graph neural networks. The feedforward neural network structures discussed in the body of the chapter assume full connectivity between nodes in adjacent layers. Sparse connections can also be considered where *neighborhoods* are defined for every node. For example, we can associate with every generic node j in layer $\ell+1$, a subset of the nodes from layer ℓ and denote this set by $\mathcal{N}_j^{(\ell+1)}$. Only neighbors i within this neighborhood will contribute to the formation of the signals $\{z_{\ell+1}(j), y_{\ell+1}(j)\}$ in the subsequent layer. We illustrated one such construction in Example 65.6 and showed how the forward and backward passes are adjusted. There are other variations along these lines; in particular, we will discuss in Chapter 67 the related class of convolutional neural networks, which are particularly useful to exploit graph-structured data. Some of the earliest references on the topic of graph neural networks are the works by Scarselli *et al.* (2004) and Gori, Monfardini and Scarselli (2005). For further discussions, readers may refer to Scarselli *et al.* (2008), Bui, Ravi, and Ramavajjala (2018), Zhou *et al.* (2019), Ward *et al.* (2020), Zhou, Zheng, and Huang (2020), Wu *et al.* (2020), and Liu and Zhou (2020).

Initialization and batch normalization. Useful references on the initialization of the weights of neural networks are the works by Glorot and Bengio (2010), Bengio (2012), and LeCun *et al.* (2012). The first two references motivate the choices (65.93b)–(65.93c) in lieu of the popular heuristics (65.93a). The original reference on the batch normalization procedure described in Sec. 65.9 is Ioffe and Szegedy (2015). The article motivates the scaling and normalization steps and comments on the implementation of the algorithm, although actual derivations for the training recursions are missing. It is still not well understood why batch normalization is so effective at improving network performance, e.g., see the discussion in Bjorck *et al.* (2018), Santurkar *et al.* (2018), and Kohler *et al.* (2018).

Multitask and transfer learning. We encountered instances of multitask and transfer learning in Examples 65.11 and 65.12. The main idea of multitask learning is to exploit the presence of some low-level common features among different tasks to enable a single trained classifier to address multiple tasks. In a similar vein, under transfer learning, knowledge acquired by a classifier trained on a particular task can be "transfered" to enable the same classifier to perform well on another "related" task for which there is only a limited amount of training data. In Chapter 72 we will consider similar concepts under the broader framework of *meta learning*. Some of the earlier references on multitask and transfer learning are the works by Suddarth and Kergosien (1990), Dietterich, Hild, and Bakiri (1990), Suddarth and Holden (1991), and Caruana (1993,1997). Applications to image processing and language models appear, for example, in Deng *et al.* (2009), Donahue *et al.* (2014), Yosinki *et al.* (2014), Dai and Le (2015), Russakovsky *et al.* (2015), and Radford *et al.* (2018).

Gaussian process modeling. We indicated earlier that networks with a single hidden layer satisfy a useful universal approximation property. Following Neal (1995,1996), we can further argue that, when the number of hidden units is large, the output of the network actually tends to a Gaussian process (recall the definition from Sec. 4.5). Moreover, as is shown by Williams (1996), the covariance matrix of this process can be specified in terms of the moments of the parameters of the network. This conclusion is useful for analyzing the behavior and modeling capabilities of networks. More specifically, consider the neural network shown in Fig. 65.19, and which consists of one

hidden layer and one output node. The hidden layer is assumed to have K units (the figure shows $K = 3$ units for illustration purposes only). The output node does not involve a nonlinearity and only combines the outputs of the hidden layer. We denote the weight vector for the output layer by w. It follows that

$$y_2 = f(W_1^{\mathsf{T}} h - \theta_1) \tag{65.191a}$$

$$y_3 = w^{\mathsf{T}} y_2 - \theta_2 \tag{65.191b}$$

where $f(\cdot)$ is the activation function. In this example, we select $f(z)$ as the error function that is associated with the standard Gaussian distribution, namely,

$$\text{erf}(z) \triangleq \frac{2}{\sqrt{\pi}} \int_0^z e^{-x^2/2} dx \tag{65.192}$$

This function behaves like $\tanh(z)$: its value is zero at $z = 0$ and it tends to ± 1 as $z \to \pm\infty$. We will view the output of the network as the response to the input feature vector, h. To make this interpretation explicit, we write $g(h)$ instead of y_3, where the function $g(\cdot)$ represents the input-output mapping of the network. It is given by

$$g(h) = w^{\mathsf{T}} \text{erf}(W_1^{\mathsf{T}} h - \theta_1) - \theta_2 \tag{65.193}$$

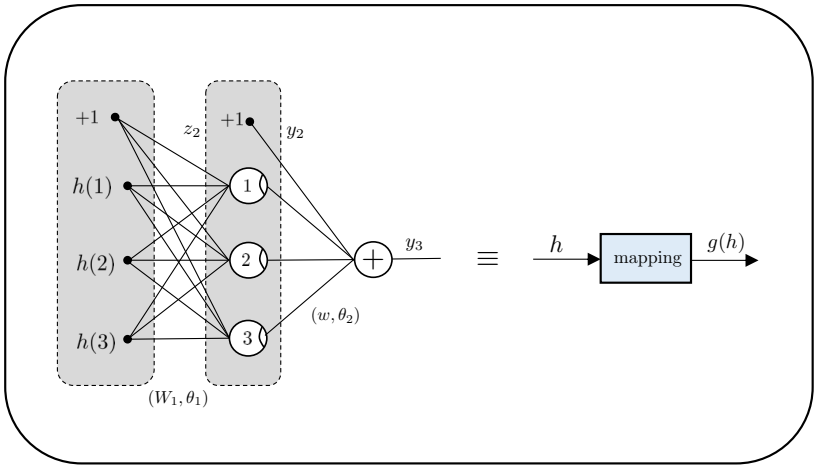

Figure 65.19 A network with a single hidden layer and one output node. It can be represented as inducing a mapping from the input h to the output denoted by $g(h)$.

We model $(\boldsymbol{w}, \boldsymbol{\theta}_2)$ as independent random variables with zero mean and second-order moments given by $\mathbb{E}\,\boldsymbol{\theta}_2 = \sigma_\theta^2$ and $\mathbb{E}\,\boldsymbol{w}\boldsymbol{w}^{\mathsf{T}} = (\sigma_w^2/K) I_K$, where the latter moment is scaled by K. We also model all entries of $\boldsymbol{W}_1$ and $\boldsymbol{\theta}_1$ to be independent and identically distributed (iid) with bounded variances. All random variables are independent of each other. It follows from the discussion on the central limit theorem in (4.158) and (4.159) that $\boldsymbol{g}(h)$ approaches a Gaussian distribution as $K \to \infty$. We can evaluate the first and second-order moments of this Gaussian distribution as follows. For any two feature vectors (h, h'), we have

$$\mathbb{E}\,\boldsymbol{g}(h) = 0 \tag{65.194a}$$

$$\mathbb{E}\,\boldsymbol{g}(h)\boldsymbol{g}(h') = \sigma_\theta^2 + \frac{\sigma_w^2}{K}\,\mathbb{E}\left\{ \Big(\text{erf}(\boldsymbol{W}_1^{\mathsf{T}} h - \boldsymbol{\theta}_1)\Big)^{\mathsf{T}} \text{erf}(\boldsymbol{W}_1^{\mathsf{T}} h' - \boldsymbol{\theta}_1) \right\} \tag{65.194b}$$

If we let $\boldsymbol{r}^{\mathsf{T}}$ denote any generic row in $\boldsymbol{W}_1^{\mathsf{T}}$ and $\boldsymbol{\alpha}$ the corresponding entry in $\boldsymbol{\theta}_1$, then we can write by virtue of the iid assumption:

$$\begin{aligned}\mathbb{E}\,\boldsymbol{g}(h)\boldsymbol{g}(h') &= \sigma_\theta^2 + \sigma_w^2\,\mathbb{E}\,\Big\{\mathrm{erf}(\boldsymbol{r}^{\mathsf{T}}h - \boldsymbol{\alpha})\mathrm{erf}(\boldsymbol{r}^{\mathsf{T}}h' - \boldsymbol{\alpha})\Big\}\\ &= \sigma_\theta^2 + \sigma_w^2\,\mathbb{E}\,\Big\{\mathrm{erf}((\boldsymbol{r}^e)^{\mathsf{T}}h^e)\mathrm{erf}((\boldsymbol{r}^e)^{\mathsf{T}}h^{e'})\Big\}\end{aligned} \tag{65.195}$$

in terms of the extended vectors

$$\boldsymbol{r}^e = \begin{bmatrix} -\boldsymbol{\alpha} \\ \boldsymbol{r} \end{bmatrix}, \quad h^e = \begin{bmatrix} 1 \\ h \end{bmatrix} \tag{65.196}$$

We assume that $\boldsymbol{r}^e$ is Gaussian distributed with covariance matrix Σ. We can then appeal to the result of Prob. 4.11 to conclude that

$$\mathbb{E}\;\boldsymbol{g}(h)\boldsymbol{g}(h') = \sigma_\theta^2 + \frac{2\sigma_w^2}{\pi}\arcsin\left\{\frac{2(h^e)^{\mathsf{T}}\Sigma h^{e'}}{\sqrt{(1+2(h^e)^{\mathsf{T}}\Sigma h^e)(1+2(h_{e'})^{\mathsf{T}}\Sigma h^{e'})}}\right\} \tag{65.197}$$

We therefore conclude that, for a sufficient large number of hidden units, the input-output mapping induced by the neural network of Fig. 65.19 generates a Gaussian process $\boldsymbol{g}(h)$, with covariance matrix specified by (65.197) — see Prob. 65.25.

CIFAR dataset. Figure 65.13 illustrates images from the CIFAR-10 dataset. It consists of color images that can belong to one of 10 classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The dataset is described in Krizhevsky (2009) and can be downloaded from `http://www.cs.toronto.edu/~kriz/cifar.html`.

PROBLEMS

65.1 Refer to the earlier Example 63.2 dealing with the XOR mapping and how linear classifiers are unable to discriminate among the four possible feature vectors $h = \mathrm{col}\{a, b\}$ where $a = \pm 1$ and $b = \pm 1$. In this problem, we verify that a three-layer feedforward network is able to separate the four feature locations and assign them to the correct labels. We denote the labels by $\gamma = -1$ for $h = \{+1, +1\}$ and $h = \{-1, -1\}$ and $\gamma = +1$ for $h = \{+1, -1\}$ and $h = \{-1, +1\}$. The network is shown in Fig. 65.20: it consists of an input layer with two input nodes, a hidden layer with two nodes and an output layer with a single node. The activation functions in the hidden units are ReLU, whereas the output node is linear. The network parameters are set to:

$$W_1^{\mathsf{T}} = -\frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \theta_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad w = \begin{bmatrix} 2 \\ -4 \end{bmatrix}, \quad \theta = 1$$

Verify that this structure is capable of implementing the XOR mapping.

65.2 Refer to the sigmoidal and tanh functions defined in Table 65.1. Verify that $f(z) = 0.5(1 + \tanh(z/2))$.

65.3 Consider a feedforward neural network with tanh activation functions. Change the signs of all weights for two successive hidden layers and their respective offset parameters. Does the output of the network change? What do you conclude?

65.4 One generalization of the ReLU is $f(z) = \max\{0, z\} + \alpha\,\min\{0, z\}$, for some nonzero scalar α. What does $f(z)$ simplify to when $\alpha = -1$? *Remark.* See Jarrett *et al.* (2009) for an application in the context of object recognition in images.

65.5 How would the listing of the stochastic-gradient backpropagation algorithm (65.87) change if the nodes in the output layer of the network do not include activation functions and are simply linear combiners?

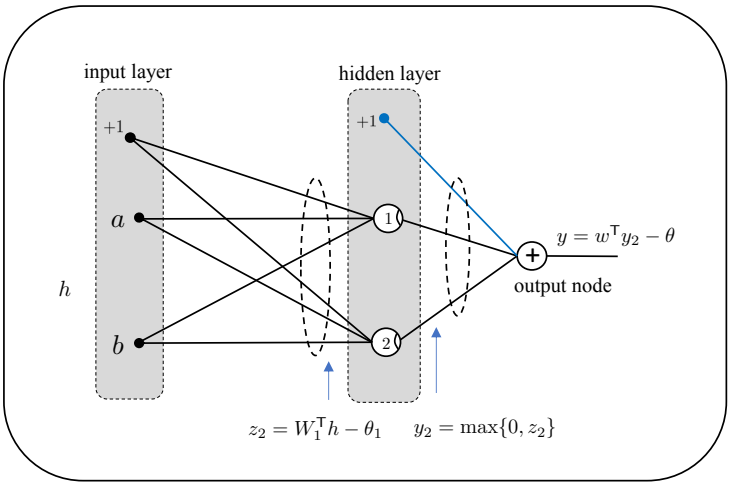

Figure 65.20 A three-layer feedforward neural network for implementing the XOR mapping.

65.6 Assume each node in a feedforward neural network employs an individual activation function, denoted by $f_{\ell,i}(x)$ for node i in layer ℓ. How would the listing of the stochastic-gradient backpropagation algorithm (65.87) change in this case?

65.7 Assume all nodes in a feedforward neural network employ the rectifier function listed in Table 65.1. How would the listing of the stochastic-gradient backpropagation algorithm (65.87) change in this case?

65.8 Assume all nodes in a feedforward neural network are simply linear combiners without activation functions. How would the listing of the stochastic-gradient backpropagation algorithm (65.87) change in this case?

65.9 Consider a feedforward neural network classifier with L layers, including the input and output layers. The network receives input vectors $h \in \mathbb{R}^M$ and generates output vectors $\widehat{\gamma} \in \mathbb{R}^Q$ with entries $\{\widehat{\gamma}(q)\}$. The output layer employs a softmax mapping and the network is trained by minimizing a regularized cross-entropy empirical risk of the same form studied in the body of the chapter. Select any output entry of index q and define the sensitivity vector λ_1^q with entries $\lambda_1^q(j) = \partial\widehat{\gamma}(q)/\partial h(j)$. Define also similar sensitivity vectors λ_ℓ^q for the internal layers with entries $\lambda_\ell^q(j) = \partial\widehat{\gamma}(q)/\partial z_\ell(j)$.

(a) Determine the boundary condition λ_L^q.

(b) Determine a backward recursion for evaluating λ_1^q.

(c) How does the recursion simplify when ReLu activation functions are employed in the input and hidden layers?

65.10 Consider the same setting as Prob. 65.9. Introduce the $Q \times M$ terminal sensitivity Jacobian matrix Λ_L with entries $[\Lambda_L]_{ij} = \partial\widehat{\gamma}(i)/\partial h(j)$. Introduce similar Jacobian matrices Λ_ℓ for the internal layers with entries $[\Lambda_\ell]_{ij} = \partial y_\ell(i)/\partial h(j)$. Verify that $\Lambda_1 = I_M$ and derive a forward recursion to update these matrices.

65.11 Consider a feedforward neural network classifier with L layers, including the input and output layers. The network receives input vectors $h \in \mathbb{R}^M$ and generates output vectors $\widehat{\gamma} \in \mathbb{R}^Q$ with entries $\{\widehat{\gamma}(q)\}$. The output layer employs a softmax mapping and the network is trained by minimizing a regularized empirical risk of the following generic form

$$\mathcal{P}(W,\theta) = \sum_{\ell=1}^{L-1} \rho \|W_\ell\|_{\mathrm{F}}^2 \;+\; \frac{1}{N}\sum_{n=0}^{N-1} \mathcal{Q}(W,\theta;\gamma(n),h_n)$$

where the notation $\mathcal{Q}(\cdot)$ denotes some generic loss function, which we also write more compactly as $\mathcal{Q}(\gamma, h)$ by dropping the (W, θ) arguments. Define the sensitivity vector δ_ℓ with entries $\delta_\ell(j) = \partial\mathcal{Q}(\gamma, h)/\partial z_\ell(j)$. Derive a a backward recursion for updating the sensitivity vectors.

65.12 Refer to expression (65.71) for the gradient of $Q(\gamma, h) = \|\gamma - \widehat{\gamma}\|^2$ relative to $w_{ij}^{(\ell)}$. Show that the second-order derivative is given by

$$\frac{\partial^2 \mathcal{Q}(\gamma, h)}{\partial (w_{ij}^{(\ell)})^2} \;=\; y_\ell^2(i)\, f''(z_{\ell+1}(j))\, (w_j^{\ell+1})^{\mathsf{T}} \delta_{\ell+2}$$

where $f''(\cdot)$ denotes the second-order derivative of the activation function. Verify that the same expression holds for cross-entropy risk minimization where $\mathcal{Q}(\gamma, h)$ is instead given by (65.128). Verify further that for both cases, we can aggregate the second-order gradients into the following rank-one matrix product representation:

$$\frac{\partial^2}{\partial W_\ell^2}\, \mathcal{Q}(\gamma, h) \;=\; (y_\ell \odot y_\ell) \Big\{ f''(z_{\ell+1}) \odot (W_{\ell+1}\delta_{\ell+2}) \Big\}^{\mathsf{T}}$$

65.13 Refer to expression (65.76) for the gradient of $\mathcal{Q}(\gamma, h) = \|\gamma - \widehat{\gamma}\|^2$ relative to $\theta_\ell(i)$. Show that the second-order derivative is given by

$$\frac{\partial^2 \mathcal{Q}(\gamma, h)}{\partial (\theta_\ell(i))^2} \;=\; f''(z_{\ell+1}(j))\, (w_i^{\ell+1})^{\mathsf{T}} \delta_{\ell+2}$$

where $f''(\cdot)$ denotes the second-order derivative of the activation function. Verify that the same expression holds for cross-entropy risk minimization where $\mathcal{Q}(\gamma, h)$ is given by (65.128). Verify further that for both cases, we can aggregate the second-order gradients into the following vector representation:

$$\frac{\partial^2}{\partial \theta_\ell^2}\, \mathcal{Q}(\gamma, h) \;=\; f''(z_{\ell+1}) \odot (W_{\ell+1}\delta_{\ell+2})$$

65.14 Refer to listing (65.81) which provides expressions for the gradients of the empirical risk relative to the weight matrices and bias vectors for the regularized empirical risk (65.47). Use the results of Probs. 65.12 and 65.13 to derive the following expressions for the second-order derivatives of the empirical risk relative to the same weight matrices and bias vectors:

$$\frac{\partial^2 \mathcal{P}(W, \theta)}{\partial W_\ell^2} = 2\rho \mathbb{1}_{n_\ell} \mathbb{1}_{n_{\ell+1}}^{\mathsf{T}} \;+\; \frac{1}{N} \sum_{n=0}^{N-1} (y_{\ell,n} \odot y_{\ell,n}) \Big\{ f''(z_{\ell+1,n}) \odot (W_{\ell+1}\delta_{\ell+2,n}) \Big\}^{\mathsf{T}}$$

$$\frac{\partial^2 \mathcal{P}(W, \theta)}{\partial \theta_\ell^2} = \frac{1}{N} \sum_{n=0}^{N-1} f''(z_{\ell+1,n}) \odot (W_{\ell+1}\delta_{\ell+2,n})$$

Verify that the same expressions hold for the cross-entropy empirical risk (65.127). In this problem, the notation $\partial^2\alpha/\partial W^2$ is a matrix having the same size as W; its individual entries consist of the second-order partial derivatives of the scalar-valued function α relative the individual entries of W:

$$\Big[\partial^2\alpha/\partial W^2\Big]_{ij} \;=\; \partial^2\alpha/\partial w_{ij}^2$$

65.15 Refer to the listing of the stochastic-gradient backpropagation algorithm (65.87). Assume we replace the original regularized empirical risk (65.47) by the ℓ_1-regularized form:

$$\mathcal{P}(W, \theta) \;\triangleq\; \sum_{\ell=1}^{L-1} \alpha \|\mathrm{vec}(W_\ell)\|_1 \;+\; \frac{1}{N} \sum_{n=0}^{N-1} \|\gamma_n - \widehat{\gamma}_n\|^2$$

where $\alpha > 0$ and the notation $\mathrm{vec}(A)$ refers to replacing a matrix A by a vector constructed by stacking its columns on top of each other.

(a) Argue that the expression for the gradient of the above empirical risk relative to W_ℓ in (65.81) now becomes

$$\frac{\partial \mathcal{P}(W,\theta)}{\partial W_\ell} = \alpha\,\mathrm{sign}(W_\ell) \;+\; \frac{1}{N}\sum_{n=0}^{N-1} y_{\ell,n}\delta_{\ell+1,n}^{\mathsf{T}} \quad (n_\ell \times n_{\ell+1})$$

where the sign function is applied to the individual entries of W_ℓ.

(b) Conclude that the only change in the listing of the backpropagation algorithm (65.87) is the update for $\boldsymbol{W}_{\ell,m}$, which now becomes

$$\boldsymbol{W}_{\ell,m} = \boldsymbol{W}_{\ell,m-1} \;-\; \mu\alpha\,\mathrm{sign}(\boldsymbol{W}_{\ell,m-1}) - \mu \boldsymbol{y}_{\ell,m}\boldsymbol{\delta}_{\ell+1,m}^{\mathsf{T}}$$

65.16 Refer to the autoencoder structure shown in Fig. 65.10 and impose the constraint $W_2 = W_1^{\mathsf{T}}$. Use a regularized cross entropy criterion and derive a stochastic gradient algorithm for training the autoencoder. Use sigmoidal units and assume the entries of the feature vectors lie within the interval $(0, 1)$.

65.17 Refer to the listing of the stochastic-gradient backpropagation algorithm (65.87), which was derived for the ℓ_2-regularized empirical risk (65.47). It is known that quadratic costs are sensitive to outliers. Consider instead the following risk function:

$$\mathcal{P}(W,\theta) \;\triangleq\; \sum_{\ell=1}^{L-1}\rho\|W_\ell\|_{\mathrm{F}}^2 \;+\; \frac{1}{N}\sum_{n=0}^{N-1} H_\Delta(\gamma_n - \widehat{\gamma}_n)$$

in terms of the Huber loss, which was defined earlier in (11.86), namely,

$$H_\Delta(x) = \begin{cases} \frac{1}{2\Delta}\|x\|^2, & \|x\| \le \Delta \\ \|x\| - \frac{\Delta}{2}, & \|x\| > \Delta \end{cases}$$

for some scalar parameter $\Delta > 0$. The Huber loss is linear in $\|x\|$ over the range $\|x\| > \Delta$ and, therefore, it penalizes less drastically large values for $\|x\|$ in comparison with the quadratic loss, $\|x\|^2$. Repeat the derivation that led to the backpropagation algorithm and determine the necessary adjustments.

65.18 Consider a feedforward neural network with a single output node; its output signals are denoted by $\{z(n), \widehat{\gamma}(n)\}$. Set all activation functions to the hyperbolic tangent function from Table 65.1. Replace the regularized empirical risk (65.47) by the following logistic risk:

$$\mathcal{P}(W,\theta) \;\triangleq\; \sum_{\ell=1}^{L-1}\rho\|W_\ell\|_{\mathrm{F}}^2 \;+\; \frac{1}{N}\sum_{n=0}^{N-1} \ln\left(1 + e^{-\gamma(n)\widehat{\gamma}(n)}\right)$$

where $\gamma(n) = \pm 1$. Repeat the derivation that led to the backpropagation algorithm (65.81) and determine the necessary adjustments.

65.19 Let $\mathcal{P}(W,\theta,a)$ denote a differentiable empirical risk function that is dependent on three sets of parameters $\{W_\ell, \theta_\ell, a_\ell\}$ similar to the least-squares risk (65.186) introduced during our treatment of the batch normalization procedure in Sec. 65.9. Using

the notation of that section, establish the validity of the following expressions:

$$\begin{aligned}
\frac{\partial \mathcal{P}}{\partial z'_{\ell+1,b}(k)} &= a_\ell(k)\,\frac{\partial \mathcal{P}}{\partial z''_{\ell+1,b}(k)} \\
\frac{\partial \mathcal{P}}{\partial \sigma^2_{\ell+1}(k)} &= -\frac{1}{2}\,\frac{1}{(\sigma^2_{\ell+1}(k)+\epsilon)^{3/2}}\sum_{b=0}^{B-1}\frac{\partial \mathcal{P}}{\partial z'_{\ell+1,b}(k)}\Big(z_{\ell+1,b}(k)-\bar{z}_{\ell+1}(k)\Big) \\
\frac{\partial \mathcal{P}}{\partial \bar{z}_{\ell+1}(k)} &= -\frac{1}{(\sigma^2_{\ell+1}(k)+\epsilon)^{1/2}}\sum_{b=0}^{B-1}\frac{\partial \mathcal{P}}{\partial z'_{\ell+1,b}(k)} \;- \\
&\qquad \frac{2}{B}\,\frac{\partial \mathcal{P}}{\partial \sigma^2_{\ell+1}(k)}\sum_{b=0}^{B-1}(z_{\ell+1,b}(k)-\bar{z}_{\ell+1}(k)) \\
\frac{\partial \mathcal{P}}{\partial z_{\ell+1,b}(k)} &= \frac{1}{(\sigma^2_{\ell+1}(k)+\epsilon)^{1/2}}\,\frac{\partial \mathcal{P}}{\partial z'_{\ell+1,b}(k)} + \\
&\qquad \frac{2}{B}\Big(z_{\ell+1,b}(k)-\bar{z}_{\ell+1}(k)\Big)\frac{\partial \mathcal{P}}{\partial \sigma^2_{\ell+1}(k)} + \frac{1}{B}\,\frac{\partial \mathcal{P}}{\partial \bar{z}_{\ell+1}(k)} \\
\frac{\partial \mathcal{P}}{\partial a_\ell(k)} &= \sum_{b=0}^{B-1}\frac{\partial \mathcal{P}}{\partial z''_{\ell+1,b}(k)}\,z'_{\ell+1,b}(k) \\
\frac{\partial \mathcal{P}}{\partial \theta_\ell(k)} &= -\sum_{b=0}^{B-1}\frac{\partial \mathcal{P}}{\partial z''_{\ell+1,b}(k)}
\end{aligned}$$

65.20 Refer to expression (65.212b) for $\sigma^2_{\ell+1}(j)$ and note that $\bar{z}_\ell(j)$ also depends on $z_{\ell+1,b}(j)$. Verify that, for any batch index b:

$$\frac{\partial \sigma^2_{\ell+1}(j)}{\partial z_{\ell+1,b}(j)} = \frac{2}{B}\Big(z_{\ell+1,b}(j)-\bar{z}_{\ell+1}(j)\Big)$$

65.21 Establish the validity of expressions (65.157) and (65.164) for the boundary sensitivity factors under multitask learning.

65.22 Consider two identical feedforward neural networks. The input to one network consists of feature vectors $\{h_n^{(1)}\} \in \mathbb{R}^M$, while the input to the second network consists of feature vectors $\{h_n^{(2)}\} \in \mathbb{R}^M$. For example, the inputs to each of the networks could correspond to images of signatures written by individuals on a device. The $Q-$dimensional outputs of the networks are similarly denoted by $\{\widehat{\gamma}_n^{(1)}\}$ and $\{\widehat{\gamma}_n^{(2)}\}$. They are assumed to be generated by sigmoidal activation functions in the last layer. The cosine of the angle (denoted by $\angle$) between the output vectors is used as a measure of similarity between them:

$$\widehat{\gamma}(n) = \cos\angle(\widehat{\gamma}_n^{(1)}, \widehat{\gamma}_n^{(2)}) \;=\; \frac{(\widehat{\gamma}_n^{(1)})^{\mathsf{T}}\widehat{\gamma}_n^{(2)}}{\|\widehat{\gamma}_n^{(1)}\|\,\|\widehat{\gamma}_n^{(2)}\|}$$

If the angle is small then the cosine value is close to 1, and if the angle is large then the cosine value is close to -1. In this way, class $+1$ would correspond to a situation in which the two signatures are more or less matching while the class -1 would correspond to a situation in which one of the signatures is a forgery. We refer to the output vectors $\{\widehat{\gamma}_n^{(1)}, \widehat{\gamma}_n^{(2)}\}$ as *embeddings* for the original input images. We impose the condition that the weight matrices and bias vectors of both networks should be the same. The resulting architecture is related to the "Siamese" network studied later in Sec. 72.2. Develop a backpropagation algorithm to train the parameters of the network, say, by minimizing

an empirical risk of the form

$$\min_{W,\theta}\left\{\mathcal{P}(W,\theta)\;\triangleq\;\sum_{\ell=0}^{L-1}\rho\|W_\ell\|_{\mathrm{F}}^2\;+\;\frac{1}{N}\sum_{n=0}^{N-1}(\gamma(n)-\widehat{\gamma}(n))^2\right\}$$

where $\gamma(n)\in\{+1,-1\}$ are the true labels and $\widehat{\gamma}(n)$ is the predicted label (i.e., the cosine output of the network). *Remark.* For more motivation, see the work by Bromley *et al.* (1994) where Siamese networks are introduced and applied to the verification of signatures written on pen-input tablets.

65.23 Consider the same setting of the Siamese network described in Prob. 65.22. We incorporate two modifications. First, we compute the absolute element-wise difference between the entries of the embedding vectors $\{\widehat{\gamma}_n^{(1)},\,\widehat{\gamma}_n^{(2)}\}$ and determine the vector r_n with entries

$$r_n\;\triangleq\;\mathrm{col}\Big\{\,\big|\widehat{\gamma}_n^{(1)}(q)-\widehat{\gamma}_n^{(2)}(q)\big|\,\Big\}$$

The vector $r_n\in\mathbb{R}^Q$ is then fed into a neural layer with a single output node employing the sigmoidal activation function and generating the scalar output $\widehat{\gamma}(n)$:

$$\widehat{\gamma}(n)=\mathrm{sigmoid}(w_r^{\mathsf{T}}r_n-\theta_r),\quad w_r\in\mathbb{R}^Q,\quad\theta_r\in\mathbb{R}$$

where (w_r,θ_r) are the weight and bias parameters for the output layer. The value of $\widehat{\gamma}$ can be interpreted as a probability measure indicating which class is more likely (matching signatures or forgery). Note that, for all practical purposes, the output layer operates as an affine classifier, with values of $w_r^{\mathsf{T}}r_n-\theta_r$ larger than $1/2$ corresponding to one class and smaller values corresponding to another class. We continue to impose the condition that the weight matrices and bias vectors of both networks should be the same. Develop a backpropagation algorithm to train the parameters of the Siamese network by minimizing now a cross-entropy empirical risk of the form

$$\min_{W,\theta,w_r,\theta_r}\left\{\rho\|w_r\|_2^2+\sum_{\ell=0}^{L-1}\rho\|W_\ell\|_{\mathrm{F}}^2-\frac{1}{N}\sum_{n=0}^{N-1}\Big\{\gamma(n)\ln(\widehat{\gamma}(n))+(1-\gamma(n))\ln(1-\widehat{\gamma}(n))\Big\}\right\}$$

where $\gamma(n)\in\{1,0\}$ are the true labels with $\gamma(n)=1$ corresponding to matching signatures and $\gamma(n)=0$ corresponding to forgery. *Remark.* For more motivation, see the work by Koch *el al.* (2015), which applies this structure to one-shot image recognition problems.

65.24 Consider the same setting of the Siamese network described in Prob. 65.23 with the following modification. We concatenate the outputs of the two Siamese branches into

$$d_n\;\triangleq\;\mathrm{col}\Big\{\widehat{\gamma}_n^{(1)},\;\widehat{\gamma}_n^{(2)}\Big\}\in\mathbb{R}^Q$$

and feed d_n into another feedforward network with L' layers and a scalar softmax output node denoted by $\widehat{\gamma}(n)$. This node is referred to as the "*relation score*" between the two inputs $\{h^{(1)},h^{(2)}\}$. We denote the parameters of the third network by $\{W_\ell',\theta_\ell'\}$. We continue to impose the condition that the weight matrices and bias vectors of the first two Siamese branches are the same. Develop a backpropagation algorithm to train the parameters of this new architecture by minimizing:

$$\min_{W,\theta,W',\theta'}\;\left\{\sum_{\ell=0}^{L-1}\rho\|W_\ell\|_{\mathrm{F}}^2+\sum_{\ell=0}^{L'-1}\rho\|W_\ell'\|_{\mathrm{F}}^2\;+\;\frac{1}{N}\sum_{n=0}^{N-1}(\gamma(n)-\widehat{\gamma}(n))^2\right\}$$

where $\gamma(n)\in\{1,0\}$ are the true labels with $\gamma(n)=1$ corresponding to matching signatures and $\gamma(n)=0$ corresponding to forgery. *Remark.* For more motivation, see the work by Sung *et al.* (2018) on relation networks in meta-learning.

65.25 Refer to expression (65.197) for the variance of the Gaussian process at the output of the neural network. Assume Σ is diagonal with $\Sigma = \mathrm{diag}\{\sigma_\alpha^2,\ \sigma_a^2 I_K\}$ where the variance of $\boldsymbol{\alpha}$ is different from the variance of the entries of $\boldsymbol{r}$. Let λ denote the angle between vectors h and h'. Argue that when the squared norms $\|h\|^2$ and $\|h'\|^2$ are much larger than $(1+2\sigma_\alpha^2)/2\sigma_a^2$, it holds that

$$\mathbb{E}\ \boldsymbol{g}(h)\boldsymbol{g}(h') \approx \sigma_\theta^2 + \sigma_w^2(1 - 2\lambda/\pi)$$

65.A DERIVATION OF BATCH NORMALIZATION ALGORITHM

We derive in this appendix algorithm (65.187) for learning the parameters of a batch-normalized neural network. We illustrate the learning procedure by reconsidering the regularized least-squares risk (65.45):

$$\{W_\ell^\star, \theta_\ell^\star, a_\ell^\star\} \ \triangleq\ \underset{\{W_\ell,\theta_\ell,a_\ell\}}{\mathrm{argmin}} \left\{ \mathcal{P}(W,\theta,a) \ \triangleq\ \sum_{\ell=1}^{L-1} \rho\|W_\ell\|_{\mathrm{F}}^2 \ +\ \frac{1}{N}\sum_{n=0}^{N-1} \|\gamma_n - \widehat{\gamma}_n\|^2 \right\} \tag{65.198}$$

where the arguments of $\mathcal{P}(\cdot)$ are augmented to include the vectors $\{a_\ell\}$. Here, the notation $\{W,\theta,a\}$ is referring to the collection of all parameters $\{W_\ell,\theta_\ell,a_\ell\}$ from across all layers. In order to implement iterative procedures for minimizing $\mathcal{P}(W,\theta,a)$, we need to know how to evaluate (or approximate) the gradients of $\mathcal{P}(W,\theta,a)$ relative to the individual entries of $\{W_\ell,\theta_\ell,a_\ell\}$, namely, the quantities

$$\frac{\partial\mathcal{P}(W,\theta,a)}{\partial w_{ij}^{(\ell)}}, \quad \frac{\partial\mathcal{P}(W,\theta,a)}{\partial\theta_\ell(i)}, \quad \frac{\partial\mathcal{P}(W,\theta,a)}{\partial a_\ell(i)} \tag{65.199}$$

for each layer ℓ and entries $\{w_{ij}^{(\ell)}, \theta_\ell(i), a_\ell(i)\}$; the notation $\{\theta_\ell(i), a_\ell(i)\}$ refers to the $i-$th entries in the vectors θ_ℓ and a_ℓ defined earlier in (65.179)–(65.180). To compute the gradients in (65.199), we repeat the arguments from Sec. 65.4.1.

Expressions for the gradients

Since, under batch normalization, it is the signal vector denoted generically by v that is fed into the nonlinear activation functions (rather than the original vector z), we replace the earlier definition (65.52) for the sensitivity factor by

$$\delta_\ell(j) \ \triangleq\ \frac{\partial\|\gamma - \widehat{\gamma}\|^2}{\partial v_\ell(j)} \tag{65.200}$$

where the differentiation is now relative to the $j-$th entry of the vector v_ℓ feeding into layer ℓ; the output of this layer is given by

$$y_\ell = f(v_\ell) \tag{65.201}$$

The above two expressions are written for a generic internal vector v_ℓ, regardless of its batch index (i.e., we are simply writing v_ℓ instead of $v_{\ell,b}$). However, we will soon restore the index within the batch for clarity and completeness.

We are ready to evaluate the partial derivatives in (65.199). To begin with, note that

$$\frac{\partial\mathcal{P}(W,\theta,a)}{\partial\theta_\ell(i)} = \frac{1}{N}\sum_{n=0}^{N-1} \frac{\partial\|\gamma_n - \widehat{\gamma}_n\|^2}{\partial\theta_\ell(i)} \tag{65.202}$$

Using a mini-batch of size B to approximate the gradient we replace (65.202) by

$$\begin{aligned}\frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial\theta_\ell(i)} &= \frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial v_{\ell+1,b}(i)}\,\frac{\partial v_{\ell+1,b}(i)}{\partial\theta_\ell(i)}\\ &\overset{(65.183)}{=} -\frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial v_{\ell+1,b}(i)}\\ &\overset{(65.200)}{=} -\frac{1}{B}\sum_{b=0}^{B-1}\delta_{\ell+1,b}(i)\end{aligned}\tag{65.203}$$

where the notation $\delta_{\ell+1,b}(i)$ denotes the sensitivity factor relative to the $b-$th signal $v_{\ell+1,b}(i)$ in the batch. It follows that, in terms of the gradients relative to the vectors θ_ℓ and not only relative to their individual entries, we have

$$\boxed{\frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial\theta_\ell} = -\frac{1}{B}\sum_{b=0}^{B-1}\delta_{\ell+1,b}}\tag{65.204}$$

where $\delta_{\ell+1,b}$ is the sensitivity vector that collects the factors $\{\delta_{\ell+1,b}(i)\}$.

Next, we consider partial derivative relative to the individual entries of the parameter vector a_ℓ. Thus, note that

$$\frac{\partial\mathcal{P}(W,\theta,a)}{\partial a_\ell(i)} = \frac{1}{N}\sum_{n=0}^{N-1}\frac{\partial\|\gamma_n-\widehat{\gamma}_n\|^2}{\partial a_\ell(i)}\tag{65.205}$$

Using a mini-batch of size B to approximate the gradient we have

$$\begin{aligned}\frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial a_\ell(i)} &= \frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial v_{\ell+1,b}(i)}\,\frac{\partial v_{\ell+1,b}(i)}{\partial a_\ell(i)}\\ &\overset{(65.182)}{=} \frac{1}{B}\sum_{b=0}^{B-1}\delta_{\ell+1,b}(i)\,z'_{\ell+1,b}(i)\end{aligned}\tag{65.206}$$

so that, in terms of the gradient relative to the vector a_ℓ and not only relative to their individual entries, we can write

$$\boxed{\frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial a_\ell} = \frac{1}{B}\,\mathrm{diag}\left\{\sum_{b=0}^{B-1}\delta_{\ell+1,b}\left(z'_{\ell+1,b}\right)^{\mathsf{T}}\right\}}\tag{65.207}$$

where the diag$\{\cdot\}$ operation applied to a matrix argument returns a column vector with the diagonal entries of the matrix. Moreover,

$$z'_{\ell+1,b} = S_{\ell+1}^{-1}\left(z_{\ell+1,b}-\bar{z}_{\ell+1}\right)\tag{65.208}$$

We now compute the partial derivatives of the risk function relative to the individual entries of the weight matrices. Thus, note that for the regularized least-squares risk:

$$\frac{\partial\mathcal{P}(W,\theta,a)}{\partial w_{ij}^{(\ell)}} = 2\rho w_{ij}^{(\ell)} + \frac{1}{N}\sum_{n=0}^{N-1}\frac{\partial\|\gamma_n-\widehat{\gamma}_n\|^2}{\partial w_{ij}^{(\ell)}}\tag{65.209}$$

Using a mini-batch of size B to approximate the rightmost term we have

$$\begin{aligned}&\frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial w_{ij}^{(\ell)}}\\&=\frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial v_{\ell+1,b}(j)}\,\frac{\partial v_{\ell+1,b}(j)}{\partial z'_{\ell+1,b}(j)}\left(\sum_{p=0}^{B-1}\frac{\partial z'_{\ell+1,b}(j)}{\partial z_{\ell+1,p}(j)}\,\frac{\partial z_{\ell+1,p}(j)}{\partial w_{ij}^{(\ell)}}\right)\end{aligned}\tag{65.210}$$

where the rightmost sum over the batch index $p=0,1,\ldots,B-1$ appears in view of the chain rule of differentiation and the fact that the variable $z'_{\ell+1,b}(j)$ depends on all vectors $z_{\ell+1,p}$ within the mini-batch of size B (this dependence is through the mean and variance variables $\bar{z}_{\ell+1}$ and $S_{\ell+1}$ — see second equation below). To evaluate the partial derivatives in (65.210) we first recall the relations:

$$v_{\ell+1,b}(j) = a_\ell(j)\,z'_{\ell+1,b}(j)-\theta_\ell(j)\tag{65.211a}$$

$$z'_{\ell+1,b}(j) = \frac{1}{(\sigma^2_{\ell+1}(j)+\epsilon)^{1/2}}\Big(z_{\ell+1,b}(j)-\bar{z}_{\ell+1}(j)\Big)\tag{65.211b}$$

$$z_{\ell+1,p}(j) = f(v_{\ell,p}(i))\,w_{ij}^{(\ell)} \;+\; \text{terms independent of } w_{ij}^{(\ell)}\tag{65.211c}$$

$$y_{\ell,p}(i) \triangleq f(v_{\ell,p}(i))\tag{65.211d}$$

as well as the definitions

$$\bar{z}_{\ell+1}(j)=\frac{1}{B}\sum_{q=0}^{B-1}z_{\ell+1,q}(j)\tag{65.212a}$$

$$\sigma^2_{\ell+1}(j)=\frac{1}{B}\sum_{q=0}^{B-1}\Big(z_{\ell+1,q}(j)-\bar{z}_{\ell+1}(j)\Big)^2\tag{65.212b}$$

We conclude from the expression for $\sigma^2_{\ell+1}(j)$ that, for any batch index $p=0,1,\ldots,B-1$ — see Prob. 65.20:

$$\begin{aligned}\frac{\partial(\sigma^2_{\ell+1}(j)+\epsilon)^{-1/2}}{\partial z_{\ell+1,p}(j)}&=-\frac{1}{2}\,\frac{1}{(\sigma^2_{\ell+1}(j)+\epsilon)^{3/2}}\,\frac{\partial(\sigma^2_{\ell+1}(j)+\epsilon)}{\partial z_{\ell+1,p}(j)}\\&=-\frac{1}{2}\,\frac{1}{(\sigma^2_{\ell+1}(j)+\epsilon)^{3/2}}\,\frac{\partial\sigma^2_{\ell+1}(j)}{\partial z_{\ell+1,p}(j)}\\&=-\frac{1}{B}\,\frac{1}{(\sigma^2_{\ell+1}(j)+\epsilon)^{3/2}}\,\Big(z_{\ell+1,p}(j)-\bar{z}_{\ell+1}(j)\Big)\end{aligned}\tag{65.213}$$

Returning to the partial derivatives in (65.210) we deduce from (65.200), (65.211a), and (65.211c) that

$$\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial v_{\ell+1,b}(j)}=\delta_{\ell+1,b}(j)\tag{65.214}$$

$$\frac{\partial v_{\ell+1,b}(j)}{\partial z'_{\ell+1,b}(j)}=a_\ell(j)\tag{65.215}$$

$$\frac{\partial z_{\ell+1,b'}(j)}{\partial w_{ij}^{(\ell)}}=y_{\ell,b'}(i)\tag{65.216}$$

while (65.211b) gives

$$\frac{\partial z'_{\ell+1,b}(j)}{\partial z_{\ell+1,p}(j)} = (\sigma^2_{\ell+1}(j)+\epsilon)^{-1/2}\,\frac{\partial\Big(z_{\ell+1,b}(j)-\bar{z}_{\ell+1}(j)\Big)}{\partial z_{\ell+1,p}(j)} \;+\; \Big(z_{\ell+1,b}(j)-\bar{z}_{\ell+1}(j)\Big)\,\frac{\partial(\sigma^2_{\ell+1}(j)+\epsilon)^{-1/2}}{\partial z_{\ell+1,p}(j)} \tag{65.217}$$

Note from the definition of the mean value $\bar{z}_{\ell+1}(j)$ that

$$\frac{\partial\Big(z_{\ell+1,b}(j)-\bar{z}_{\ell+1}(j)\Big)}{\partial z_{\ell+1,p}(j)} \;=\; \begin{cases} -1/B, & \text{when } p\neq b \\ (1-1/B), & \text{when } p=b \end{cases} \tag{65.218}$$

Combining with (65.213) we get

$$\frac{\partial z'_{\ell+1,b}(j)}{\partial z_{\ell+1,p}(j)} \;\triangleq\; c^{(\ell+1)}_{b,p}(j) \tag{65.219}$$

where the scalar $c^{(\ell+1)}_{b,p}(j)$, whose value depends on the layer index, is computed as follows. Let

$$\xi_\ell(j) \;\triangleq\; \frac{1}{(\sigma^2_{\ell+1}(j)+\epsilon)^{1/2}} \tag{65.220}$$

$$c^{(\ell+1)}_{b,p}(j) \;\triangleq\; \begin{cases} \Big(1-\dfrac{1}{B}\Big)\xi_\ell(j)\left(1-\dfrac{1}{B-1}\,\xi^2_\ell(j)\,\Big(z_{\ell+1,b}(j)-\bar{z}_{\ell+1}(j)\Big)^2\right) \\ \text{(when } p=b\text{)} \\ -\dfrac{1}{B}\xi_\ell(j)\left(1+\xi^2_\ell(j)\,\Big(z_{\ell+1,b}(j)-\bar{z}_{\ell+1}(j)\Big)\Big(z_{\ell+1,p}(j)-\bar{z}_{\ell+1}(j)\Big)\right) \\ \text{(when } p\neq b\text{)} \end{cases} \tag{65.221}$$

Substituting into (65.209)–(65.210) we find that a batch approximation for the gradient of the risk function relative to the individual entries $w^{(\ell)}_{ij}$ of the weight matrices can be computed as follows:

$$\begin{aligned} &2\rho w^{(\ell)}_{ij} + \frac{1}{B}\sum_{b=0}^{B-1}\frac{\partial\|\gamma_b-\widehat{\gamma}_b\|^2}{\partial w^{(\ell)}_{ij}} \\ &= 2\rho w^{(\ell)}_{ij} + \frac{1}{B}\sum_{b=0}^{B-1}\delta_{\ell+1,b}(j)a_\ell(j)\left(\sum_{p=0}^{B-1}c^{(\ell+1)}_{b,p}(j)y_{\ell,p}(i)\right) \end{aligned} \tag{65.222}$$

Sensitivity factors

It remains to show how to propagate the sensitivity vectors $\delta_{\ell,b}$ defined by (65.200). We start with the output layer for which $\ell=L$. We denote the vector signals at the output layer in the $b-$th sample of the batch by $\{v_{L,b},\widehat{\gamma}_b\}$, using the subscript b, with the letter v representing the signal prior to the activation function, i.e.,

$$\widehat{\gamma}_b = f(v_{L,b}) \tag{65.223}$$

We denote the individual entries at the output layer by $\{\widehat{\gamma}_b(1),\ldots,\widehat{\gamma}_b(Q)\}$. Likewise, we denote the pre-activation entries by $\{v_{L,b}(1),\ldots,v_{L,b}(Q)\}$. We also denote the pre- and post-activation signals at a generic $\ell-$th hidden layer by $\{v_{\ell,b},y_{\ell,b}\}$ with

$$y_{\ell,b} = f\,(v_{\ell,b}) \tag{65.224}$$

with individual entries indexed by $\{v_{\ell,b}(i), y_{\ell,b}(i)\}$. The number of nodes within hidden layer ℓ is denoted by n_ℓ.

In this way, the chain rule for differentiation gives

$$\begin{aligned}
\delta_{L,b}(j) &\triangleq \frac{\partial\|\gamma_b - \widehat{\gamma}_b\|^2}{\partial v_{L,b}(j)} \\
&= \sum_{k=1}^{Q} \frac{\partial\|\gamma_b - \widehat{\gamma}_b\|^2}{\partial\widehat{\gamma}_b(k)} \frac{\partial\widehat{\gamma}_b(k)}{\partial v_{L,b}(j)} \\
&= \sum_{k=1}^{Q} 2(\widehat{\gamma}_b(k) - \gamma_b(k)) \frac{\partial\widehat{\gamma}_b(k)}{\partial v_{L,b}(j)} \\
&= 2\Big(\widehat{\gamma}_b(j) - \gamma_b(j)\Big) f'(v_{L,b}(j))
\end{aligned} \tag{65.225}$$

since only $\widehat{\gamma}_b(j)$ depends on $v_{L,b}(j)$ through the relation $\widehat{\gamma}_b(j) = f(v_{L,b}(j))$. Consequently, using the Hadamard product notation we get

$$\boxed{\delta_{L,b} = 2(\widehat{\gamma}_b - \gamma_b) \odot f'(v_{L,b})} \tag{65.226}$$

Next we evaluate $\delta_{\ell,b}$ for the earlier layers. This calculation can be carried out recursively by relating $\delta_{\ell,b}$ to $\delta_{\ell+1,b}$. Indeed, note that

$$\begin{aligned}
\delta_{\ell,b}(j) &\triangleq \frac{\partial\|\gamma_b - \widehat{\gamma}_b\|^2}{\partial v_{\ell,b}(j)} \\
&= \sum_{k=1}^{n_{\ell+1}} \frac{\partial\|\gamma_b - \widehat{\gamma}_b\|^2}{\partial v_{\ell+1,b}(k)} \frac{\partial v_{\ell+1,b}(k)}{\partial z'_{\ell+1,b}(k)} \left(\sum_{p=0}^{B-1} \frac{\partial z'_{\ell+1,b}(k)}{\partial z_{\ell+1,p}(k)} \frac{\partial z_{\ell+1,p}(k)}{\partial v_{\ell,b}(j)}\right)
\end{aligned} \tag{65.227}$$

where the signals $z_{\ell+1,p}(k)$ and $v_{\ell,b}(j)$ are related via

$$z_{\ell+1,p}(k) = f(v_{\ell,p}(j))\, w_{jk}^{(\ell)} \tag{65.228}$$

Therefore, when $p = b$, we have

$$\frac{\partial z_{\ell+1,p}(k)}{\partial v_{\ell,b}(j)} = f'(v_{\ell,b}(j))\, w_{jk}^{(\ell)}, \quad \text{when } p = b \tag{65.229}$$

Otherwise, the above partial derivative is zero when $p \neq b$. Using this result and expression (65.219) we find that the partial derivatives in (65.227) evaluate to

$$\delta_{\ell,b}(j) = \sum_{k=1}^{n_{\ell+1}} \delta_{\ell+1,b}(k) a_\ell(k)\, c_{b,b}^{(\ell+1)}(k) f'\Big(v_{\ell,b}(j)\Big) w_{jk}^{(\ell)} \tag{65.230}$$

If we introduce the diagonal scaling matrix

$$D_{\ell,b} \triangleq \operatorname{diag}\Big\{a_\ell(1)c_{b,b}^{(\ell+1)}(1), \ldots, a_\ell(n_{\ell+1})c_{b,b}^{(\ell+1)}(n_{\ell+1})\Big\} \tag{65.231}$$

then, in vector form, we arrive at the following recursion for the sensitivity vector $\delta_{\ell,b}$, which runs backward from $\ell = L-1$ down to $\ell = 2$ with the boundary condition $\delta_{L,b}$ given by (65.226):

$$\boxed{\delta_{\ell,b} = f'(v_{\ell,b}) \odot (W_\ell D_{\ell,b} \delta_{\ell+1,b})} \tag{65.232}$$

The resulting algorithm with batch normalization for minimizing the regularized least-squares risk (65.198) is listed in (65.187).

REFERENCES

Bengio, Y. (2009), "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127.

Bengio, Y. (2012), "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*, G. Montavon, G. B. Orr, K. Muller, *Eds.*, 2nd edition, pp. 437–478, Spring-Verlag. Also available online at the link arXiv:1206.5533.

Bengio, Y., A. Courville, and P. Vincent (2013), "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Analalysis and Machine Intelligence*, vol. 35, no. 8, pp. 1798–1828.

Bengio, Y., P. Lamblin, D. Popovici, and H. Larochelle (2006), "Greedy layer-wise training of deep networks," *Proc. Advances Neural Information Processing Systems* (NIPS), pp. 153–160, Vancouver, Canada.

Bishop, C. (1995), *Neural Networks for Pattern Recognition*, Clarendon Press, Oxford.

Bjorck, J., C. Gomes, B. Selman, and K. Q. Weinberger (2018), "Understanding batch normalization," *Proc. Advances in Neural Information Processing Systems* (NIPS), pp. 7694–7705, Montreal, Canada.

Boulard, H. and Y. Kamp (1988), "Auto-association by multilayer perceptrons and singular value decomposition," *Biological Cybernetics*, vol. 59, pp. 291–294.

Bourlard, H. A. and N. Morgan (1993), *Connectionist Speech Recognition: A Hybrid Approach*, Kluwer, MA.

Bourlard, H. and C. J. Wellekens (1989), "Links between Markov models and multilayer Perceptrons," *Proc. Advances Neural Information Processing Systems* (NIPS), pp. 502–507, Denver, CO.

Bridle, J. S. (1990a), "Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters," *Proc. Advances Neural Information Processing Systems* (NIPS), pp. 211–217, Denver, CO.

Bridle, J. S. (1990b), "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition," in *Neurocomputing: Algorithms, Architectures and Applications*, F. F. Soulie and J. Herault, *Eds.*, pp. 227–236, Springer–Verlag.

Bromley, J., I. Guyon, Y. LeCun, E. Sickinger, and R. Shah (1994), "Signature verification using a Siamese time delay neural network," *Proc. Advances Neural Information Processing Systems* (NIPS), pp. 737–744, Denver, CO.

Bui, T. D., S. Ravi, and V. Ramavajjala (2018), "Neural graph learning: Training neural networks using graphs," *Proc. ACM International Conference on Web Search and Data Mining*, pp. 64–71, Los Angeles, CA.

Caruana, R. (1993), "Multitask learning: A knowledge-based source of inductive bias," *Proc.International Conference on Machine Learning* (ICML), pp. 41–48, Amherst, MA.

Caruana, R. (1997), "Multitask leaning," *Machine Learning*, vol. 28, pp. 41–75.

Cybenko, G. (1989), "Approximations by superpositions of sigmoidal functions," *Math. Control, Signals, and Systems*, vol. 2, no. 4, pp. 303–314.

Dai, A. M. and Q. V. Le (2015), "Semi-supervised sequence learning," *Proc. Advances Neural Information Processing Systems* (NIPS), pp. 1–9, Montreal, Canada.

De Boer, P.-T., D. P. Kroese, S. Mannor, and R. Y. Rubinstein (2005), "A tutorial of the cross-entropy method," *Ann. Operations Res.*, vol. 134, pp. 19–67.

Deng, J., W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei (2009). "ImageNet: A large-scale hierarchical image database," *Proc. Conference on Computer Vision and Pattern Recognition* (CVPR), pp. 248–255, Miami, FL.

Dietterich, T. G., H. Hild, and G. Bakiri (1990), "A comparative study of ID3 and backpropagation for English text-to-speech mapping," *Proc. International Conference on Artificial Intelligence*, pp. 24–31, Boston, MA.

Donahue, J., Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell (2014),

"DeCAF: A deep convolutional activation feature for generic visual recognition," *Proc. International Conference on Machine Learning* (ICML), pp. pp. 647–655.

Duda, R. O. and P. E. Hart (1973), *Pattern Classification and Scene Analysis*, Wiley, NY.

Duda, R. O., P. E. Hart, and D. G. Stork (2000), *Pattern Classification*, 2nd edition, Wiley, NY.

Erdogmus D. and J. Principe (2002), "An error-entropy minimization algorithm for supervised training of nonlinear adaptive systems," *IEEE Trans. Signal Process.*, vol. 50, no. 7, pp. 1780–1786.

Funahashi, K. (1989), "On the approximate realization of continuous mappings by neural networks," *Neural Networks*, vol. 2, pp. 183–192.

Glorot, X. and Y. Bengio (2010), "Understanding the difficulty of training deep feedforward neural networks," *Proc. Machine Learning Research*, vol. 9, pp. 249–256.

Glorot, X., A. Bordes, and Y. Bengio (2011), "Deep sparse rectifier neural networks," *J. Mach. Learn. Res.*, vol. 15, pp. 315–323.

Goodfellow, I., Y. Bengio, and A. Courville (2016), *Deep Learning*, MIT Press, Cambridge, MA.

Gori, M., G. Monfardini and F. Scarselli (2005), "A new model for learning in graph domains," *Proc. IEEE International Joint Conference on Neural Networks*, pp. 729–734, Montreal, Canada.

Hassibi, B., A. H. Sayed and T. Kailath (1994a), "$\mathcal{H}^\infty-$optimality criteria for LMS and backpropagation," *Proc. Advances Neural Information Processing Systems* (NIPS), vol. 6, pp. 351–358, Denver, CO.

Hassibi, B., A. H. Sayed and T. Kailath (1994b), "LMS and backpropagation are minimax filters," in *Theoretical Advances in Neural Computation and Learning,* V. Roychowdhury, K. Siu and A. Orlitsky, *Eds.*, pp. 425–447, Kluwer Academic Publishers, Norwell, MA.

Haykin, S. (1999), *Neural Networks: A Comprehensive Foundation*, Prentice Hall, NY.

Haykin, S. (2009), *Neural Networks and Learning Machines*, 3rd edition, Pearson Press.

Hecht-Nielsen, R. (1989), "Theory of the back-propagation neural network," *Proc. Intern. Joint Conf. Neural Networks*, pp.593–606, New York, NY.

Hinton, G. (1987), "Connectionist learning procedures," Technical Report CMU-CS-87-115, Carnegie Mellon University, Dept. Computer Science. Also published in *Artificial Intelligence*, vol. 40, pp. 185–234, 1989.

Hinton, G., S. Osindero, and Y.-W. Teh (2006), "A fast learning algorithm for deep belief nets," *Neural Computation*, vol. 18, no. 7, pp. 1527–1554.

Hinton, G. and R. Salakhutdinov (2006), "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, pp. 504–507.

Hinton, G., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2012b), "Improving neural networks by preventing co-adaptation of feature detectors," *available online at arXiv:1207.0580.*

Hinton, G. and R. S. Zemel (1994), "Autoencoders, minimum description length, and Helmholtz free energy," *Proc. Advances Neural Information Processing* (NIPS), pp. 3–10, Denver, CO.

Hochreiter, S. (1991), *Untersuchungen zu Dynamischen Neuronalen Netzen,* Diploma thesis, Institut f. Informatik, Technische Univ. Munich.

Hochreiter, S., Y. Bengio, P. Frasconi, and J. Schmidhuber (2001), "Gradient flow in recurrent nets: The difficulty of learning long-term dependencies," in *A Field Guide to Dynamical Recurrent Neural Networks*, S. C. Kremer and J. F. Kolen, *Eds.*, IEEE Press.

Hornik, K. (1991), "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257.

Hornik, K., M. Stinchcombe, H. White (1989), "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366.

Ioffe, S. and C. Szegedy (2015), "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *Proc. International Conference on Ma-*

chine Learning (ICML), vol. 37, pp. 448–456, Lille, France. Also available online at arXiv:1502.03167v3.

Jarrett, K., K. Kavukcuoglu, M. Ranzato and Y. LeCun (2009), "What is the best multi-stage architecture for object recognition," *Proc. IEEE Intern. Conf. Computer Vision*, pp. 2146–2153, Kyoto, Japan.

Koch, G., R. Zemel, and R. Salakhutdinov (2015), "Siamese neural networks for one-shot image recognition," *Proc. International Conference on Machine Learning* (ICML), pp. 1–8, Lille, France.

Kohler, J., H. Daneshmand, A. Lucchi, M. Zhou, K. Neymeyr, and T. Hofmann (2018), "Exponential convergence rates for batch normalization: The power of length-direction decoupling in non-convex optimization," *available online at arXiv:1805.10694*.

Krizhevsky, A. (2009), *Learning Multiple Layers of Features from Tiny Images*, MS dissertation, Computer Science Department, University of Toronto, Canada.

Krizhevsky, A., I. Sutskever, and G. Hinton (2012), "ImageNet classification with deep convolutional neural networks," *Proc. Advances Neural Information Processing Systems* (NIPS), vol. 25, pp. 1097–1105, Lake Tahoe, NV.

LeCun, Y., Y. Bengio, and G. Hinton (2015), "Deep learning," *Nature*, vol. 521, pp. 436–444.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998), "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324.

LeCun, Y., L. Bottou, G. B. Orr, and K. Muller (2012), "Efficient backprop," in *Neural Networks, Tricks of the Trade*, G. Montavon, G. B. Orr, and K. Muller, *Eds.*, 2nd edition, pp. 9–48, Spring-Verlag.

Leshno, M., V. Y. Lin, A. Pinkus, and S. Schocken (1993), "Multilayer feedforward networks with a non-polynomial activation function can approximate any function," *Neural Networks*, vol. 6, pp. 861–867.

Linsker, R. (1998), "Self-organization in a perceptual network," *IEEE Computer*, vol. 21.pp. 105–117.

Liu, Z. and J. Zhou (2020), *Introduction to Graph Neural Networks*, Morgan & Claypool.

Minsky, M. and S. Papert (1969), *Perceptrons*, MIT Press, Cambridge, MA. Expanded edition published in 1987.

Neal, R. (1995), *Bayesian Learning for Neural Networks*, PhD dissertation, Dept. of Computer Science, University of Toronto, Canada.

Neal, R. (1996), *Bayesian Learning for Neural Networks*, Springer, NY.

Ney, H. (1995), "On the probabilistic interpretation of neural network classifiers and discriminative training criteria," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 17, no. 2, pp. 107–119.

Nilsson, N. (1965), *Learning Machines*, McGraw-Hill, NY.

Principe, J. C., D. Xu, and J. Fisher (2000), "Information theoretic learning," in *Unsupervised Adaptive Filtering: Blind Source Separation*, S. Haykin, *Ed.*, pp. 265–319, Wiley, NY.

Radford, A. K. Narasimhan, T. Salimans, and I. Sutskever (2018), "Improving language understanding by generative pre-training," *available at* https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf

Rosenblatt, F. (1957), *The Perceptron: A Perceiving and Recognizing Automaton*, Technical Report 85-460-1, Project PARA, Cornell Aeronautical Lab.

Rosenblatt, F.(1958), "The Perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408.

Rosenblatt, F. (1962), *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Spartan Press, Washington, DC.

Rubinstein, R. Y. and D. P. Kroese (2004), *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*, Springer-Verlag, NY.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1985), "Learning internal representations by error propagation," in J. L. McClelland and D. E. Rumelhart, *Eds.*,

Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol. 1, pp. 318–362, MIT Press.

Rumelhart, D. E., G. Hinton, and R. J. Williams (1986), "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536.

Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei (2015), "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211-252, available online at arXiv:1409.0575.

Santurkar, S., D. Tsipras, A. Ilyas, and A. Madry (2018), "How does batch normalization help optimization?," *available online at arXiv:1805.11604*.

Scarselli, F., M. Gori, A. Tsoi, M. Hagenbuchner, and G. Monfardini (2008), "The graph neural network model," *IEEE Trans. Neural Networks*, vol. 20, no. 1, pp. 61–80.

Scarselli, F., A. C. Tsoi, M. Gori, and M. Hagenbuchner (2004), " Graphical-based learning environments for pattern recognition," *Proc. Joint IAPR International Workshops on SPR and SSPR*, pp. 42–56, Lisbon, Portugal.

Schmidhuber, J. (2015), "Deep learning in neural networks: An Overview," *Neural Networks*, vol. 61, pp. 85–117.

Schwenk, H. and M. Milgram (1995), "Transformation invariant autoassociation with application to handwritten character recognition," *Proc. Advances Neural Information Processing* (NIPS), pp. 991–998, Denver, CO.

Siu, K.-Y., V. P. Roychowdhury, and T. Kailath (1995), *Discrete Neural Computation: A Theoretical Foundation*, Prentice Hall, NJ.

Solla, S. A., E. Levin, and M. Fleisher (1988), "Accelerated learning in layered neural networks," *Complex Systems*, vol. 2, pp. 625–640.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014), "Dropout: A simple way to prevent neural networks from overfitting," *J. Machine Learning Research*, vol. 15, pp. 1929–1958.

Stinhcombe, M. and H. White (1989), "Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions," *Proc. International Joint Conference on Neural Networks*, pp.613–617, Washington, DC.

Suddarth, S. C. and A. D. C. Holden (1991), "Symbolic-neural systems and the use of hints for developing complex systems," *Intern. J. Man-Machine Studies*, vol. 35, no. 3, pp. 291–311.

Suddarth, S. C. and Y. L. Kergosien (1990), "Rule-injection hints as a means of improving network performance and learning time," *Proc. EURASIP Workshop on Neural Networks*, pp. 120–129, Sesimbra, Portugal.

Sung, F., Y. Yang, L. Zhang, T. Xiang, P. H. S. Torr, and T. M. Hospedales (2018), "Learning to compare: Relation network for few-shot learning," *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition* (CVPR), pp. 1199–1208, Salt Lake City, UT.

Theodoridis, S. (2015), *Machine Learning: A Bayesian and Optimization Perspective*, Academic Press.

Theodoridis, S. and K. Koutroumbas (2008), *Pattern Recognition*, 4th edition, Academic Press.

Vincent, P., H. Larochelle, Y. Bengio, and P. A. Manzagol (2008), "Extracting and composing robust features with denoising autoencoders," *Proc. International Conference on Machine Learning* (ICML), pp. 1096–1103, Helsinki, Finland.

Ward, I. R., J. Joyner, C. Lickfold, S. Rowe, Y. Guo, and M. Bennamoun (2020), "A practical guide to graph neural networks: How do graph neural networks work, and where can they be applied?" *available online at https://arxiv.org/pdf/2010.05234.pdf.*

Werbos, P. J. (1974), *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Ph.D. dissertation, Harvard University.

Werbos, P. J. (1988), "Generalization of backpropagation with application to a recurrent gas market model," *Neural Networks*, vol. 1, no. 4, pp. 339–356.

Werbos, P. J. (1990), "Backpropagation through time: What it does and how to do it," *Proc. IEEE*, vol. 78, no. 10, pp. 1550–1560.

Werbos, P. J. (1994), *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*, Wiley, NY.

Widrow, B. and M. A. Lehr (1990),"30 years of adaptive neural networks: Perceptron, Madaline, and backpropagation," *Proc. IEEE*, vol. 78, no. 9, pp. 1415–1442.

Williams, C. K. I. (1996), "Computing with infinite networks," in *Proc. Advances in Neural Information Processing* (NIPS), pp. 295–301, Denver, CO.

Wu, Z., S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu (2020), "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Networks and Learning Systems*, pp. 1–21.

Xu, D. and J. Principe (1999), "Training MLPs layer-by-layer with the information potential," *Proc. IEEE ICASSP*, pp. 1045–1048, Phoenix, AZ.

Yosinski, J., J. Clune, Y. Bengio, and H. Lipson (2014), "How transferable are features in deep neural networks?" *Proc. Advances in Neural Information Processing Systems* (NIPS), pp. 1–9, Montreal, Canada.

Zhou, P. and J. Austin (1989), "Learning criteria for training neural network classifiers," *Neural Computing and Applications*, vol. 7, no. 4, pp. 334–342, 1989.

Zhou, J., G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun (2019), "Graph neural networks: A review of methods and applications," *available online at https://arxiv.org/abs/1812.08434*.

Zhou, Y., H. Zheng, and X. Huang (2020), "Graph neural networks: Taxonomy, advances and trends," *available online at https://arxiv.org/abs/2012.08752.*