Artificial neural networks or simply neural networks are inspired by biological neuronal networks. A real biological neuron, or a nerve cell, comprises dendrites, a cell body, and an axon that leads to synaptic terminals. A neuron transmits information via electrochemical signals. When there is enough concentration of ions at the dendrites of a neuron it generates an electric pulse along its axon called an action potential, which in turn activates the synaptic terminals, releasing more ions and thus causing the information to flow to dendrites of other neurons. A human brain has on the order of 100 billion neurons, with each neuron having between 1,000 to 10,000 connections to other neurons. Thus, a human brain is a neuronal network with 100 trillion to a quadrillion ($10^{15}$) interconnections! Interestingly, as far as we know, learning happens by adjusting the synaptic strengths, since synaptic signals can be either excitatory or inhibitory, making the post-synaptic neuron either more or less likely to generate an action potential, respectively.

Artificial neural networks are comprised of abstract neurons that try to mimic real neurons at a very high level. They can be described via a weighted directed graph $G = (V, E)$, with each node $v_i \in V$ representing a neuron, and each directed edge $(v_i, v_j) \in E$ representing a synaptic to dendritic connection from $v_i$ to $v_j$. The weight of the edge $wij$ denotes the synaptic strength. Neural networks are characterized by the type of activation function used to generate an output, and the architecture of the network in terms of how the nodes are interconnected. For example, whether the graph is a directed acyclic graph or has cycles, whether the graph is layered or not, and so on. It is important to note that a neural network is designed to represent and learn information by adjusting the synaptic weights.

25.1 ARTIFICIAL NEURON: ACTIVATION FUNCTIONS

An artificial neuron acts as a processing unit, that first aggregates the incoming signals via a weighted sum, and then applies some function to generate an output. For example, a binary neuron will output a 1 whenever the combined signal exceeds a threshold, or 0 otherwise.
Figure 25.1 shows the schematic of a neuron $z_k$ that has incoming edges from neurons $x_1, \ldots, x_d$. For simplicity, both the name and the (output) value of a neuron are denoted by the same symbol. Thus, $x_i$ denotes neuron $i$, and also the value of that neuron. The net input at $z_k$, denoted $net_k$, is given as the weighted sum

$$net_k = b_k + \sum_{i=1}^{d} w_{ik} \cdot x_i = b_k + w^T x$$

(25.1)

where $w_k = (w_{1k}, w_{2k}, \ldots, w_{dk})^T \in \mathbb{R}^d$ and $x = (x_1, x_2, \ldots, x_d)^T \in \mathbb{R}^d$ is an input point. Notice that $x_0$ is a special bias neuron whose value is always fixed at 1, and the weight from $x_0$ to $z_k$ is $b_k$, which specifies the bias term for the neuron. Finally, the output value of $z_k$ is given as some activation function, $f(\cdot)$, applied to the net input at $z_k$

$$z_k = f(net_k)$$

The value $z_k$ is then passed along the outgoing edges from $z_k$ to other neurons.

Neurons differ based on the type of activation function used. Some commonly used activation functions, illustrated in Figure 25.2, are:

**Linear/Identity Function:** Identity is the simplest activation function; it simply returns its argument, and is also called a linear activation function:

$$f(net_k) = net_k$$

(25.2)

Figure 25.2(a) plots the identity function. To examine the role of the bias term, note that $net_k > 0$ is equivalent to $w^T x > -b_k$. That is, the output transitions from negative to positive when the weighted sum of the inputs exceeds $-b_k$, as shown in Figure 25.2(b).

**Step Function:** This is a binary activation function, where the neuron outputs a 0 if the net value is negative (or zero), and 1 if the net value is positive (see Figure 25.2(c)).

$$f(net_k) = \begin{cases} 0 & \text{if } net_k \leq 0 \\ 1 & \text{if } net_k > 0 \end{cases}$$

(25.3)
25.1 Artificial Neuron: Activation Functions

Figure 25.2. Neuron activation functions; also illustrating the effect of bias.
It is interesting to note that the transition from 0 to 1 happens when the weighted sum of the inputs exceeds \(-b_k\), as shown in Figure 25.2(d).

**Rectified Linear Unit (ReLU):** Here, the neuron remains inactive if the net input is less than or equal to zero, and then increases linearly with \(net_k\), as shown in Figure 25.2(c).

\[
f(\text{net}_k) = \begin{cases} 
0 & \text{if } \text{net}_k \leq 0 \\
\text{net}_k & \text{if } \text{net}_k > 0 
\end{cases}
\]  
(25.4)

An alternative expression for the ReLU activation is given as \(f(\text{net}_k) = \max(0, \text{net}_k)\). The transition from zero to linear output happens when the weighted sum of the inputs exceeds \(-b_k\) (see Figure 25.2(f)).

**Sigmoid:** The sigmoid function, illustrated in Figure 25.2(g) squashes its input so that the output ranges between 0 and 1

\[
f(\text{net}_k) = \frac{1}{1 + \exp(-\text{net}_k)}
\]  
(25.5)

When \(\text{net}_k = 0\), we have \(f(\text{net}_k) = 0.5\), which implies that the transition point where the output crosses 0.5 happens when the weighted sum of the inputs exceeds \(-b_k\) (see Figure 25.2(h)).

**Hyperbolic Tangent (tanh):** The hyperbolic tangent or tanh function is similar to the sigmoid, but its output ranges between \(-1\) and \(+1\) (see Figure 25.2(i)).

\[
f(\text{net}_k) = \frac{\exp(\text{net}_k) - \exp(-\text{net}_k)}{\exp(\text{net}_k) + \exp(-\text{net}_k)} = \frac{\exp(2 \cdot \text{net}_k) - 1}{\exp(2 \cdot \text{net}_k) + 1}
\]  
(25.6)

When \(\text{net}_k = 0\), we have \(f(\text{net}_k) = 0\), which implies that the output transitions from negative to positive when the weighted sum of the inputs exceeds \(-b_k\), as shown in Figure 25.2(j).

**Softmax:** Softmax is a generalization of the sigmoid or logistic activation function. Softmax is mainly used at the output layer in a neural network, and unlike the other functions it depends not only on the net input at neuron \(k\), but it depends on the net signal at all other neurons in the output layer. Thus, given the net input vector,
The output of the softmax function for the \( k \)th neuron is given as

\[
f(\text{net}_k | \text{net}) = \frac{\exp(\text{net}_k)}{\sum_{j=1}^{p} \exp(\text{net}_j)}
\]  

(25.7)

Figure 25.3 plots the softmax activation for \( \text{net}_k \) versus the net signal \( \text{net}_j \), with all other net values fixed at zero. The output behaves similar to a sigmoid curve for any given value of \( \text{net}_j \).

### 25.1.1 Derivatives for Activation Functions

For learning using a neural network, we need to consider the derivative of an activation function with respect to its argument. The derivatives for the activation functions are given as follows:

**Identity/Linear:** The identity (or linear) activation function has a derivative of 1 with respect to its argument, giving us:

\[
\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = 1
\]  

(25.8)

**Step:** The step function has a derivative of 0 everywhere except for the discontinuity at 0, where the derivative is \( \infty \).

**ReLU:** The ReLU function [Eq. (25.4)] is non-differentiable at 0, nevertheless for other values its derivative is 0 if \( \text{net}_j < 0 \) and 1 if \( \text{net}_j > 0 \). At 0, we can set the derivative to be any value in the range \([0, 1]\), a simple choice being 0. Putting it all together, we have

\[
\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = \begin{cases} 
0 & \text{if } \text{net}_j \leq 0 \\
1 & \text{if } \text{net}_j > 0 
\end{cases}
\]  

(25.9)

Even though ReLU has a discontinuity at 0 it is a popular choice for training deep neural networks.

**Sigmoid:** The derivative of the sigmoid function [Eq. (25.5)] is given as

\[
\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = f(\text{net}_j) \cdot (1 - f(\text{net}_j))
\]  

(25.10)

**Hyperbolic Tangent:** The derivative of the tanh function [Eq. (25.6)] is given as

\[
\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = 1 - f(\text{net}_j)^2
\]  

(25.11)

**Softmax:** The softmax activation function [Eq. (25.7)] is a vector valued function, which maps a vector input \( \text{net} = (\text{net}_1, \text{net}_2, \cdots, \text{net}_p)^T \) to a vector of probability
values. Softmax is typically used only for the output layer. The partial derivative of $f(n_{jt})$ with respect to $n_{jt}$ is given as

$$\frac{\partial f(n_{jt})}{\partial n_{jt}} = f(n_{jt}) \cdot (1 - f(n_{jt}))$$

whereas the partial derivative of $f(n_{jt})$ with respect to $n_{kt}$, with $k \neq j$ is given as

$$\frac{\partial f(n_{jt})}{\partial n_{kt}} = -f(n_{kt}) \cdot f(n_{jt})$$

Since softmax is used at the output layer, if we denote the $i$th output neuron as $o_i$, then $f(n_{it}) = o_i$, and we can write the derivative as:

$$\frac{\partial f(n_{jt})}{\partial n_{kt}} = \frac{\partial o_j}{\partial n_{kt}} = \begin{cases} o_j \cdot (1 - o_j) & \text{if } k = j \\ -o_k \cdot o_j & \text{if } k \neq j \end{cases}$$

(25.12)

25.2 NEURAL NETWORKS: REGRESSION AND CLASSIFICATION

Networks of (artificial) neurons are capable of representing and learning arbitrarily complex functions for both regression and classification tasks.

25.2.1 Regression

Consider the multiple (linear) regression problem, where given an input $x_i \in \mathbb{R}^d$, the goal is to predict the response as follows

$$\hat{y}_i = b + w_1 x_{i1} + w_2 x_{i2} + \cdots + w_d x_{id}$$

Here, $b$ is the bias term, and $w_j$ is the regression coefficient or weight for attribute $X_j$. Given a training data $D$ comprising $n$ points $x_i$ in a $d$-dimensional space, along with their corresponding true response value $y_i$, the bias and weights for linear regression are chosen so as to minimize the sum of squared errors between the true and predicted response over all data points

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

As shown in Figure 25.4(a), a neural network with $d + 1$ input neurons $x_0, x_1, \cdots, x_d$, including the bias neuron $x_0 = 1$, and a single output neuron $o$, all with identity activation functions and with $\hat{y} = o$, represents the exact same model as multiple linear regression. Whereas the multiple regression problem has a closed form solution, neural networks learn the bias and weights via a gradient descent approach that minimizes the squared error.

Neural networks can just as easily model the multivariate (linear) regression task, where we have a $p$-dimensional response vector $y_i \in \mathbb{R}^p$ instead of a single value $y_i$. That is, the training data $D$ comprises $n$ points $x_i \in \mathbb{R}^d$ and their true response vectors $y_i \in \mathbb{R}^p$. As shown in Figure 25.4(b), multivariate regression can be modeled by a neural
network with \( d + 1 \) input neurons, and \( p \) output neurons \( o_1, o_2, \ldots, o_p \), with all input and output neurons using the identity activation function. A neural network learns the weights by comparing its predicted output \( \hat{y} = o = (o_1, o_2, \ldots, o_p)^T \) with the true response vector \( y = (y_1, y_2, \ldots, y_p)^T \). That is, training happens by first computing the error function or loss function between \( o \) and \( y \). Recall that a loss function assigns a score or penalty for predicting the output to be \( o \) when the desired output is \( y \). When the prediction matches the true output the loss should be zero. The most common loss function for regression is the squared error function

\[
E_x = \frac{1}{2} \| y - o \|^2 = \frac{1}{2} \sum_{j=1}^{p} (y_j - o_j)^2
\]

where \( E_x \) denotes the error on input \( x \). Across all the points in a dataset, the total sum of squared errors is

\[
E = \sum_{i=1}^{n} E_{x_i} = \frac{1}{2} \sum_{i=1}^{n} \| y_i - o_i \|^2
\]

**Example 25.1 (Neural Networks for Multiple and Multivariate Regression).**

Consider the multiple regression of sepal length and petal length on the dependent attribute petal width for the Iris dataset with \( n = 150 \) points. From Example 23.3 we find that the solution is given as

\[
\hat{y} = -0.014 - 0.082 \cdot x_1 + 0.45 \cdot x_2
\]

The squared error for this optimal solution is 6.179 on the training data.

Using the neural network in Figure 25.4(a), with linear activation for the output and minimizing the squared error via gradient descent, results in the following learned parameters, \( b = 0.0096 \), \( w_1 = -0.087 \) and \( w_2 = 0.452 \), yielding the regression model

\[
o = 0.0096 - 0.087 \cdot x_1 + 0.452 \cdot x_2
\]

with a squared error of 6.18, which is very close to the optimal solution.
**Multivariate Linear Regression** For multivariate regression, we use the neural network architecture in Figure 25.4(b) to learn the weights and bias for the Iris dataset, where we use sepal length and sepal width as the independent attributes, and petal length and petal width as the response or dependent attributes. Therefore, each input point \( \mathbf{x} \) is 2-dimensional, and the true response vector \( \mathbf{y} \) is also 2-dimensional. That is, \( d = 2 \) and \( p = 2 \) specify the size of the input and output layers. Minimizing the squared error via gradient descent, yields the following parameters:

\[
\begin{pmatrix}
    b_1 & b_2 \\
    w_{11} & w_{12} \\
    w_{21} & w_{22}
\end{pmatrix} = \begin{pmatrix}
    -1.83 & -1.47 \\
    1.72 & 0.72 \\
    -1.46 & -0.50
\end{pmatrix}, \quad
\begin{pmatrix}
    o_1 \\
    o_2
\end{pmatrix} = \begin{pmatrix}
    -1.83 + 1.72 \cdot x_1 - 1.46 \cdot x_2 \\
    -1.47 + 0.72 \cdot x_1 - 0.50 \cdot x_2
\end{pmatrix}
\]

The squared error on the training set is 84.9. Optimal least squared multivariate regression yields a squared error of 84.16 with the following parameters:

\[
\begin{pmatrix}
    \hat{y}_1 \\
    \hat{y}_2
\end{pmatrix} = \begin{pmatrix}
    -2.56 + 1.78 \cdot x_1 - 1.34 \cdot x_2 \\
    -1.59 + 0.73 \cdot x_1 - 0.48 \cdot x_2
\end{pmatrix}
\]

### 25.2.2 Classification

Networks of artificial neurons can also learn to classify the inputs. Consider the binary classification problem, where \( y = 1 \) denotes that the point belongs to the positive class, and \( y = 0 \) means that it belongs to the negative class. Recall that in logistic regression, we model the probability of the positive class via the logistic (or sigmoid) function:

\[
\pi(\mathbf{x}) = P(y = 1|\mathbf{x}) = \frac{1}{1 + \exp\{-(b + \mathbf{w}^T\mathbf{x})\}}
\]

where \( b \) is the bias term and \( \mathbf{w} = (w_1, w_2, \cdots, w_d)^T \) is the vector of estimated weights or regression coefficients. On the other hand

\[
P(y = 0|\mathbf{x}) = 1 - P(y = 1|\mathbf{x}) = 1 - \pi(\mathbf{x})
\]

A simple change to the neural network shown in Figure 25.4(a) allows it to solve the logistic regression problem. All we have to do is use a sigmoid activation function at the output neuron \( o \), and use the cross-entropy error instead of squared error. Given input \( \mathbf{x} \), true response \( y \), and predicted response \( o \), recall that the cross-entropy error (see Eq. (24.8)) is defined as

\[
\mathcal{E}_x = -(y \cdot \ln(o) + (1 - y) \cdot \ln(1 - o))
\]

Thus, with sigmoid activation, the output of the neural network in Figure 25.4(a) is given as

\[
o = f(\text{net}_o) = \text{sigmoid}(b + \mathbf{w}^T\mathbf{x}) = \frac{1}{1 + \exp\{-(b + \mathbf{w}^T\mathbf{x})\}} = \pi(\mathbf{x})
\]

which is the same as the logistic regression model.
**Multiclass Logistic Regression** In a similar manner, the multiple output neural network architecture shown in Figure 25.4(b) can be used for multiclass or nominal logistic regression. For the general classification problem with $K$ classes $(c_1, c_2, \ldots, c_K)$, the true response $y$ is encoded as a one-hot vector. Thus, class $c_1$ is encoded as $e_1 = (1, 0, \ldots, 0)^T$, class $c_2$ is encoded as $e_2 = (0, 1, \ldots, 0)^T$, and so on, with $e_i \in \{0, 1\}^K$ for $i = 1, 2, \ldots, K$. Thus, we encode $y$ as a multivariate vector $y \in \{e_1, e_2, \ldots, e_K\}$. Recall that in multiclass logistic regression (see Section 24.2) the task is to estimate the per class bias $b_i$ and weight vector $w_i \in \mathbb{R}^d$, with the last class $c_K$ used as the base class with fixed bias $b_K = 0$ and fixed weight vector $w_K = (0, 0, \ldots, 0)^T \in \mathbb{R}^d$. The probability vector across all $K$ classes is modeled via the softmax function (see Eq. (24.17)):

$$
\pi_i(x) = \frac{\exp(b_i + w_i^T x)}{\sum_{j=1}^{K} \exp(b_j + w_j^T x)}, \quad \text{for all } i = 1, 2, \ldots, K
$$

Therefore, the neural network shown in Figure 25.4(b) (with $p = K$) can solve the multiclass logistic regression task, provided we use a softmax activation at the outputs, and use the $K$-way cross-entropy error (see Eq. (24.18)), defined as

$$
E_x = -(y_1 \cdot \ln(o_1) + \cdots + y_K \cdot \ln(o_K))
$$

where $x$ is an input vector, $o = (o_1, o_2, \ldots, o_K)^T$ is the predicted response vector, and $y = (y_1, y_2, \ldots, y_K)^T$ is the true response vector. Note that only one element of $y$ is 1, and the rest are 0, due to the one-hot encoding.

With softmax activation, the output of the neural network in Figure 25.4(b) (with $p = K$) is given as

$$
o_i = P(y = e_i|x) = f(n_{net_i}) = \frac{\exp(net_i)}{\sum_{j=1}^{p} \exp(net_j)} = \pi_i(x)
$$

which matches the multiclass logistic regression task. The only restriction we have to impose on the neural network is that the weights on edges into the last output neuron should be zero to model the base class weights $w_K$. However, in practice, we can relax this restriction, and just learn a regular weight vector for class $c_K$.

**Example 25.2 (Logistic Regression: Binary and Multiclass).** We applied the neural network in Figure 25.4(a), with logistic activation at the output neuron and cross-entropy error function, on the Iris principal components dataset. The output is a binary response indicating Iris-virginica ($Y = 1$) or one of the other Iris types ($Y = 0$). As expected, the neural network learns an identical set of weights and bias as shown for the logistic regression model in Example 24.2, namely:

$$
o = -6.79 - 5.07 \cdot x_1 - 3.29 \cdot x_2
$$

Next, we applied the neural network in Figure 25.4(b), using a softmax activation and cross-entropy error function, to the Iris principal components data with three classes: Iris-setosa ($Y = 1$), Iris-versicolor ($Y = 2$) and Iris-virginica ($Y = 3$). Thus, we need $K = 3$ output neurons, $o_1$, $o_2$, and $o_3$. Further, to obtain the same model as in the multiclass logistic regression from Example 24.3,
we fix the incoming weights and bias for output neuron $o_3$ to be zero. The model is given as

$$o_1 = -3.49 + 3.61 \cdot x_1 + 2.65 \cdot x_2$$
$$o_2 = -6.95 - 5.18 \cdot x_1 - 3.40 \cdot x_2$$
$$o_3 = 0 + 0 \cdot x_1 + 0 \cdot x_2$$

which is essentially the same as in Example 24.3.

If we do not constrain the weights and bias for $o_3$ we obtain the following model:

$$o_1 = -0.89 + 4.54 \cdot x_1 + 1.96 \cdot x_2$$
$$o_2 = -3.38 - 5.11 \cdot x_1 - 2.88 \cdot x_2$$
$$o_3 = 4.24 + 0.52 \cdot x_1 + 0.92 \cdot x_2$$

The classification decision surface for each class is illustrated in Figure 25.5. The points in class $c_1$ are shown as squares, $c_2$ as circles, and $c_3$ as triangles. This figure should be contrasted with the decision boundaries shown for multiclass logistic regression in Figure 24.3, which has the weights and bias set to 0 for the base class $c_3$.

### 25.2.3 Error Functions

Typically, for a regression task, we use squared error as the loss function, whereas for classification, we use the cross-entropy loss function. Furthermore, when learning from neural networks, we will require the partial derivatives of the error function with respect to the output neurons. Thus, the commonly used error functions and their derivatives are listed below:
Squared Error: Given an input vector $\mathbf{x} \in \mathbb{R}^d$, the squared error loss function measures the squared deviation between the predicted output vector $\mathbf{o} \in \mathbb{R}^p$ and the true response $\mathbf{y} \in \mathbb{R}^p$, defined as follows:

$$
\mathcal{E}_x = \frac{1}{2} \| \mathbf{y} - \mathbf{o} \|^2 = \frac{1}{2} \sum_{j=1}^{p} (y_j - o_j)^2
$$

(25.13)

where $\mathcal{E}_x$ denotes the error on input $\mathbf{x}$.

The partial derivative of the squared error function with respect to a particular output neuron $o_j$ is

$$
\frac{\partial \mathcal{E}_x}{\partial o_j} = \frac{1}{2} \cdot 2 \cdot (y_j - o_j) \cdot -1 = o_j - y_j
$$

(25.14)

Across all the output neurons, we can write this as

$$
\frac{\partial \mathcal{E}_x}{\partial \mathbf{o}} = \mathbf{o} - \mathbf{y}
$$

(25.15)

Cross-Entropy Error: For classification tasks, with $K$ classes $\{c_1, c_2, \ldots, c_K\}$, we usually set the number of output neurons $p = K$, with one output neuron per class. Furthermore, each of the classes is coded as a one-hot vector, with class $c_i$ encoded as the $i$th standard basis vector $\mathbf{e}_i = (e_{i1}, e_{i2}, \ldots, e_{iK})^T \in \{0, 1\}^K$, with $e_{ii} = 1$ and $e_{ij} = 0$ for all $j \neq i$. Thus, given input $\mathbf{x} \in \mathbb{R}^d$, with the true response $\mathbf{y} = (y_1, y_2, \ldots, y_K)^T$, where $\mathbf{y} \in \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_K\}$, the cross-entropy loss is defined as

$$
\mathcal{E}_x = - \sum_{i=1}^{K} y_i \cdot \ln(o_i) = - \left( y_1 \cdot \ln(o_1) + \cdots + y_K \cdot \ln(o_K) \right)
$$

(25.16)

Note that only one element of $\mathbf{y}$ is 1 and the rest are 0 due to the one-hot encoding. That is, if $\mathbf{y} = \mathbf{e}_i$, then only $y_i = 1$, and the other elements $y_j = 0$ for $j \neq i$.

The partial derivative of the cross-entropy error function with respect to a particular output neuron $o_j$ is

$$
\frac{\partial \mathcal{E}_x}{\partial o_j} = - \frac{y_j}{o_j}
$$

(25.17)

The vector of partial derivatives of the error function with respect to the output neurons is therefore given as

$$
\frac{\partial \mathcal{E}_x}{\partial \mathbf{o}} = \left( \frac{\partial \mathcal{E}_x}{\partial o_1}, \frac{\partial \mathcal{E}_x}{\partial o_2}, \ldots, \frac{\partial \mathcal{E}_x}{\partial o_K} \right)^T = \left( - \frac{y_1}{o_1}, - \frac{y_2}{o_2}, \ldots, - \frac{y_K}{o_K} \right)^T
$$

(25.18)

Binary Cross-Entropy Error: For classification tasks with binary classes, it is typical to encode the positive class as 1 and the negative class as 0, as opposed to using a one-hot encoding as in the general $K$-class case. Given an input $\mathbf{x} \in \mathbb{R}^d$, with
true response \( y \in \{0, 1\} \), there is only one output neuron \( o \). Therefore, the binary cross-entropy error is defined as

\[
E_o = - (y \cdot \ln(o) + (1 - y) \cdot \ln(1 - o))
\]  
(25.19)

Here \( y \) is either 1 or 0. The partial derivative of the binary cross-entropy error function with respect to the output neuron \( o \) is

\[
\frac{\partial E_o}{\partial o} = \frac{\partial}{\partial o} \left\{ -y \cdot \ln(o) - (1 - y) \cdot \ln(1 - o) \right\} \\
= - \left( \frac{y}{o} + \frac{1 - y}{1 - o} \cdot -1 \right) = \frac{-y \cdot (1 - o) + (1 - y) \cdot o}{o \cdot (1 - o)} \\
= \frac{o - y}{o \cdot (1 - o)}
\]  
(25.20)

25.3 MULTILAYER PERCEPTRON: ONE HIDDEN LAYER

A multilayer perceptron (MLP) is a neural network that has distinct layers of neurons. The inputs to the neural network comprise the input layer, and the final outputs from the MLP comprise the output layer. Any intermediate layer is called a hidden layer, and an MLP can have one or many hidden layers. Networks with many hidden layers are called deep neural networks. An MLP is also a feed-forward network. That is, information flows in the forward direction, and from a layer only to the subsequent layer. Thus, information flows from the input to the first hidden layer, from the first to the second hidden layer, and so on, until it reaches the output layer from the last hidden layer. Typically, MLPs are fully connected between layers. That is, each neuron in the input layer is connected to all the neurons in the first hidden layer, and each neuron in the first hidden layer is connected to all neurons in the second hidden layer, and so on, and finally, each neuron in the last hidden layer is connected to all neurons in the output layer.

For ease of explanation, in this section, we will consider an MLP with only one hidden layer, and we will later generalize the discussion to deep MLPs. For example, Figure 25.6 shows an MLP with one hidden layer. The input layer has \( d \) neurons, \( x_1, x_2, \ldots, x_d \), and an additional neuron \( x_0 \) that specifies the biases for the hidden layer. The hidden layer has \( m \) neurons, \( z_1, z_2, \ldots, z_m \), and an additional neuron \( z_0 \) that specifies the biases for the output neurons. Finally, the output layer has \( p \) neurons, \( o_1, o_2, \ldots, o_p \). The bias neurons have no incoming edges, since their value is always fixed at 1. Thus, in total there are \( d \times m + m \times p \) weight parameters \((w_{ij})\) and a further \( m + p \) bias parameters \((b_i)\) that need to be learned by the neural network. These parameters also correspond to the total number of edges in the MLP.

25.3.1 Feed-forward Phase

Let \( \mathbf{D} \) denote the training dataset, comprising \( n \) input points \( \mathbf{x}_i \in \mathbb{R}^d \) and corresponding true response vectors \( \mathbf{y}_i \in \mathbb{R}^p \). For each pair \((\mathbf{x}, \mathbf{y})\) in the data, in the feed-forward phase,
the point $\mathbf{x} = (x_1, x_2, \cdots, x_d)^T \in \mathbb{R}^d$ is supplied as an input to the MLP. The input neurons do not use any activation function, and simply pass along the supplied input values as their output value. This is equivalent to saying that the net at input neuron $k$ is $net_k = x_k$, and the activation function is the identity function $f(net_k) = net_k$, so that the output value of neuron $k$ is simply $x_k$.

Given the input neuron values, we compute the output value for each hidden neuron $z_k$ as follows:

$$z_k = f(net_k) = f\left(b_k + \sum_{i=1}^{d} w_{ik} \cdot x_i\right)$$

where $f$ is some activation function, and $w_{ik}$ denotes the weight between input neuron $x_i$ and hidden neuron $z_k$. Next, given the hidden neuron values, we compute the value for each output neuron $o_j$ as follows:

$$o_j = f(net_j) = f\left(b_j + \sum_{i=1}^{m} w_{ij} \cdot z_i\right)$$

where $w_{ij}$ denotes the weight between hidden neuron $z_i$ and output neuron $o_j$.

We can write the feed-forward phase as a series of matrix-vector operations. For this we define the $d \times m$ matrix $\mathbf{W}_h$ comprising the weights between input and hidden layer neurons, and vector $\mathbf{b}_h \in \mathbb{R}^m$ comprising the bias terms for hidden layer neurons,
given as
\[
W_h = \begin{pmatrix}
w_{11} & w_{12} & \cdots & w_{1m} \\
w_{21} & w_{22} & \cdots & w_{2m} \\
\vdots & \vdots & \ddots & \vdots \\
w_{d1} & w_{d2} & \cdots & w_{dm}
\end{pmatrix}, \quad b_h = \begin{pmatrix}
b_1 \\
b_2 \\
\vdots \\
b_m
\end{pmatrix}
\]  
(25.21)

where \(w_{ij}\) denotes the weight on the edge between input neuron \(x_i\) and hidden neuron \(z_j\), and \(b_i\) denotes the bias weight from \(x_0\) to \(z_i\). The net input and the output for all the hidden layer neurons can be computed via a matrix-vector multiplication operation, as follows:
\[
\text{net}_h = b_h + W_h^T x
\]  
(25.22)
\[
z = f(\text{net}_h) = f\left(b_h + W_h^T x\right)
\]  
(25.23)

Here, \(\text{net}_h = (\text{net}_1, \cdots, \text{net}_m)^T\) denotes the net input at each hidden neuron (excluding the bias neuron \(z_0\) whose value is always fixed at \(z_0 = 1\)), and \(z = (z_1, z_2, \cdots, z_m)^T\) denotes the vector of hidden neuron values. The activation function \(f(\cdot)\) applies to, or distributes over, each element of \(\text{net}_h\), i.e., \(f(\text{net}_h) = (f(\text{net}_1), \cdots, f(\text{net}_m))^T \in \mathbb{R}^m\). Typically, all neurons in a given layer use the same activation function, but they can also be different if desired.

Likewise, let \(W_o \in \mathbb{R}^{m \times p}\) denote the weight matrix between the hidden and output layers, and let \(b_o \in \mathbb{R}^p\) be the bias vector for output neurons, given as
\[
W_o = \begin{pmatrix}
w_{11} & w_{12} & \cdots & w_{1p} \\
w_{21} & w_{22} & \cdots & w_{2p} \\
\vdots & \vdots & \ddots & \vdots \\
w_{m1} & w_{m2} & \cdots & w_{mp}
\end{pmatrix}, \quad b_o = \begin{pmatrix}
b_1 \\
b_2 \\
\vdots \\
b_p
\end{pmatrix}
\]  
(25.24)

where \(w_{ij}\) denotes the weight on the edge between hidden neuron \(z_i\) and output neuron \(o_j\), and \(b_i\) the bias weight between \(z_0\) and output neuron \(o_i\). The output vector can then be computed as follows:
\[
\text{net}_o = b_o + W_o^T z
\]  
(25.25)
\[
o = f(\text{net}_o) = f\left(b_o + W_o^T z\right)
\]  
(25.26)

To summarize, for a given input \(x \in \mathbb{D}\) with desired response \(y\), an MLP computes the output vector via the feed-forward process, as follows:
\[
o = f\left(b_o + W_o^T z\right) = f\left(b_o + W_o^T \cdot f\left(b_h + W_h^T x\right)\right)
\]  
(25.27)

where \(o = (o_1, o_2, \cdots, o_p)^T\) is the vector of predicted outputs from the single hidden layer MLP.

### 25.3.2 Backpropagation Phase

Backpropagation is the algorithm used to learn the weights between successive layers in an MLP. The name comes from the manner in which the error gradient is propagated.
backwards from the output to input layers via the hidden layers. For simplicity of exposition, we will consider backpropagation for an MLP with a single hidden layer with \( m \) neurons, with squared error function, and with sigmoid activations for all neurons. We will later generalize to multiple hidden layers, and other error and activation functions.

Let \( \mathbf{D} \) denote the training dataset, comprising \( n \) input points \( \mathbf{x}_i = (x_{i1}, x_{i2}, \cdots, x_{id})^T \in \mathbb{R}^d \) and corresponding true response vectors \( \mathbf{y}_i \in \mathbb{R}^p \). Let \( \mathbf{W}_h \in \mathbb{R}^{d \times m} \) denote the weight matrix between the input and hidden layer, and \( \mathbf{b}_h \in \mathbb{R}^m \) the vector of bias terms for the hidden neurons from Eq. (25.21). Likewise, let \( \mathbf{W}_o \in \mathbb{R}^{m \times p} \) denote the weight matrix between the hidden and output layer, and \( \mathbf{b}_o \in \mathbb{R}^p \) the bias vector for output neurons from Eq. (25.24).

For a given input pair \( (\mathbf{x}, \mathbf{y}) \) in the training data, the MLP first computes the output vector \( \mathbf{o} \) via the feed-forward step in Eq. (25.27). Next, it computes the error in the predicted output \( \text{vis-a-vis} \) the true response \( \mathbf{y} \) using the squared error function

\[
\mathcal{E}_n = \frac{1}{2} \| \mathbf{y} - \mathbf{o} \|^2 = \frac{1}{2} \sum_{j=1}^{p} (y_j - o_j)^2
\]

(25.28)

The basic idea behind backpropagation is to examine the extent to which an output neuron, say \( o_j \), deviates from the corresponding target response \( y_j \), and to modify the weights \( w_{ij} \) between each hidden neuron \( z_i \) and \( o_j \) as some function of the error – large error should cause a correspondingly large change in the weight, and small error should result in smaller changes. Likewise, the weights between all input and hidden neurons should also be updated as some function of the error at the output, as well as changes already computed for the weights between the hidden and output layers. That is, the error propagates backwards.

The weight update is done via a gradient descent approach to minimize the error. Let \( \nabla w_{ij} \) be the gradient of the error function with respect to \( w_{ij} \), or simply the weight gradient at \( w_{ij} \). Given the previous weight estimate \( w_{ij} \), a new weight is computed by taking a small step \( \eta \) in a direction that is opposite to the weight gradient at \( w_{ij} \)

\[
w_{ij} = w_{ij} - \eta \cdot \nabla w_{ij}
\]

(25.29)

In a similar manner, the bias term \( b_j \) is also updated via gradient descent

\[
b_j = b_j - \eta \cdot \nabla b_j
\]

(25.30)

where \( \nabla b_j \) is the gradient of the error function with respect to \( b_j \), which we call the bias gradient at \( b_j \).

**Updating Parameters Between Hidden and Output Layer**

Consider the weight \( w_{ij} \) between hidden neuron \( z_i \) and output neuron \( o_j \), and the bias term \( b_j \) between \( z_0 \) and \( o_j \). Using the chain rule of differentiation, we compute the weight gradient at \( w_{ij} \) and bias gradient at \( b_j \), as follows:

\[
\nabla w_{ij} = \frac{\partial \mathcal{E}_n}{\partial w_{ij}} = \frac{\partial \mathcal{E}_n}{\partial \text{net}_i} \cdot \frac{\partial \text{net}_i}{\partial w_{ij}} = \delta_i \cdot z_i
\]

\[
\nabla b_j = \frac{\partial \mathcal{E}_n}{\partial b_j} = \frac{\partial \mathcal{E}_n}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial b_j} = \delta_j
\]

(25.31)
where we use the symbol \( \delta_j \) to denote the partial derivative of the error with respect to net signal at \( o_j \), which we also call the net gradient at \( o_j \)

\[
\delta_j = \frac{\partial E_x}{\partial \text{net}_j}
\]

(25.32)

Furthermore, the partial derivative of net \( j \) with respect to \( w_{ij} \) and \( b_j \) is given as

\[
\frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left( b_j + \sum_{k=1}^{m} w_{kj} \cdot z_k \right) = z_i \\
\frac{\partial \text{net}_j}{\partial b_j} = \frac{\partial}{\partial b_j} \left( b_j + \sum_{k=1}^{m} w_{kj} \cdot z_k \right) = 1
\]

where we used the fact that \( b_j \) and all \( w_{kj} \) for \( k \neq i \) are constants with respect to \( w_{ij} \).

Next, we need to compute \( \delta_j \), the net gradient at \( o_j \). This can also be computed via the chain rule

\[
\delta_j = \frac{\partial E_x}{\partial \text{net}_j} = \frac{\partial E_x}{\partial f(\text{net}_j)} \cdot \frac{\partial f(\text{net}_j)}{\partial \text{net}_j}
\]

(25.33)

Note that \( f(\text{net}_j) = o_j \). Thus, \( \delta_j \) is composed of two terms, namely the partial derivative of the error term with respect to the output or activation function applied to the net signal, and the derivative of the activation function with respect to its argument. Using the squared error function and from Eq. (25.14), for the former, we have

\[
\frac{\partial E_x}{\partial f(\text{net}_j)} = \frac{\partial E_x}{\partial o_j} = \frac{\partial}{\partial o_j} \left( \frac{1}{2} \sum_{k=1}^{p} (y_k - o_k)^2 \right) = (o_j - y_j)
\]

where we used the observation that all \( o_k \) for \( k \neq j \) are constants with respect to \( o_j \).

Since we assume a sigmoid activation function, for the latter, we have via Eq. (25.10)

\[
\frac{\partial f(\text{net}_j)}{\partial \text{net}_j} = o_j \cdot (1 - o_j)
\]

Putting it all together, we get

\[
\delta_j = (o_j - y_j) \cdot o_j \cdot (1 - o_j)
\]

Let \( \delta_o = (\delta_1, \delta_2, \ldots, \delta_p)^T \) denote the vector of net gradients at each output neuron, which we call the net gradient vector for the output layer. We can write \( \delta_o \) as

\[
\delta_o = o \odot (1 - o) \odot (o - y)
\]

(25.34)

where \( \odot \) denotes the element-wise product (also called the Hadamard product) between the vectors, and where \( o = (o_1, o_2, \ldots, o_p)^T \) is the predicted output vector, \( y = (y_1, y_2, \ldots, y_p)^T \) is the (true) response vector, and \( 1 = (1, \ldots, 1)^T \in \mathbb{R}^p \) is the \( p \)-dimensional vector of all ones.

Let \( z = (z_1, z_2, \ldots, z_m)^T \) denote the vector comprising the values of all hidden layer neurons (after applying the activation function). Based on Eq. (25.31), we can compute the gradients \( \nabla_{w_{ij}} \) for all hidden to output neuron connections via the outer product of
\[ \nabla_{w_o} = \left( \begin{array}{ccc} \nabla_{w_{11}} & \nabla_{w_{12}} & \cdots & \nabla_{w_{1p}} \\ \nabla_{w_{21}} & \nabla_{w_{22}} & \cdots & \nabla_{w_{2p}} \\ \vdots & \vdots & \ddots & \vdots \\ \nabla_{w_{m1}} & \nabla_{w_{m2}} & \cdots & \nabla_{w_{mp}} \end{array} \right) = \mathbf{z} \cdot \delta_o^T \] 

(25.35)

where \( \nabla_{w_o} \in \mathbb{R}^{n \times p} \) is the matrix of weight gradients. The vector of bias gradients is given as:

\[ \nabla_{b_o} = (\nabla_{b_1}, \nabla_{b_2}, \cdots, \nabla_{b_p})^T = \delta_o \] 

(25.36)

where \( \nabla_{b_o} \in \mathbb{R}^p \).

Once the gradients have been computed, we can update all the weights and biases as follows

\[ \mathbf{W}_o = \mathbf{W}_o - \eta \cdot \nabla_{w_o} \]

\[ \mathbf{b}_o = \mathbf{b}_o - \eta \cdot \nabla_{b_o} \] 

(25.37)

where \( \eta \) is the step size (also called the learning rate) for gradient descent.

**Updation Parameters Between Input and Hidden Layer**

Consider the weight \( w_{ij} \) between input neuron \( x_i \) and hidden neuron \( z_j \), and the bias term between \( x_0 \) and \( z_j \). The weight gradient at \( w_{ij} \) and bias gradient at \( b_j \) is computed similarly to Eq. (25.31)

\[ \nabla_{w_{ij}} = \frac{\partial E_x}{\partial w_{ij}} = \frac{\partial E_x}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w_{ij}} = \delta_j \cdot x_i \]

\[ \nabla_{b_j} = \frac{\partial E_x}{\partial b_j} = \frac{\partial E_x}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial b_j} = \delta_j \] 

(25.38)

which follows from

\[ \frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left\{ b_j + \sum_{k=1}^{m} w_{kj} \cdot x_k \right\} = x_i \]

\[ \frac{\partial \text{net}_j}{\partial b_j} = \frac{\partial}{\partial b_j} \left\{ b_j + \sum_{k=1}^{m} w_{kj} \cdot x_k \right\} = 1 \]

To compute the net gradient \( \delta_j \) at the hidden neuron \( z_j \) we have to consider the error gradients that flow back from all the output neurons to \( z_j \). Applying the chain rule, we get:

\[ \delta_j = \frac{\partial E_x}{\partial \text{net}_j} = \sum_{k=1}^{p} \frac{\partial E_x}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial z_j} = \frac{\partial \text{net}_j}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial z_j} = \frac{\partial E_x}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial z_j} \]

\[ = z_j \cdot (1 - z_j) \cdot \sum_{k=1}^{p} \delta_k \cdot w_{jk} \]

where \( \frac{\partial E_x}{\partial \text{net}_j} = z_j \cdot (1 - z_j) \), since we assume a sigmoid activation function for the hidden neurons. The chain rule leads to a natural interpretation for backpropagation, namely,
to find the net gradient at $z_j$ we have to consider the net gradients at each of the output neurons $\delta_k$ but weighted by the strength of the connection $w_{jk}$ between $z_j$ and $o_k$, as illustrated in Figure 25.7. That is, we compute the weighted sum of gradients $\sum_{k=1}^{p} \delta_k \cdot w_{jk}$, which is used to compute $\delta_j$, the net gradient at hidden neuron $z_j$.

Let $\delta_o = (\delta_1, \delta_2, \ldots, \delta_p)^T$ denote the vector of net gradients at the output neurons, and $\delta_h = (\delta_1, \delta_2, \ldots, \delta_m)^T$ the net gradients at the hidden layer neurons. We can write $\delta_h$ compactly as

$$\delta_h = z \odot (1 - z) \odot (W_o \cdot \delta_o)$$  \hspace{1cm} (25.39)

where $\odot$ is the element-wise product, $1 = (1, 1, \cdots, 1) \in \mathbb{R}^m$ is the vector of all ones and $z = (z_1, z_2, \cdots, z_m)^T$ is the vector of hidden layer outputs. Furthermore, $W_o \cdot \delta_o \in \mathbb{R}^m$ is the vector of weighted gradients at each hidden neuron, since

$$W_o \cdot \delta_o = \left( \sum_{k=1}^{p} \delta_k \cdot w_{1k}, \sum_{k=1}^{p} \delta_k \cdot w_{2k}, \cdots, \sum_{k=1}^{p} \delta_k \cdot w_{mk} \right)^T$$

Let $x = (x_1, x_2, \cdots, x_d)^T$ denote the input vector, then based on Eq. (25.38) we can compute the gradients $\nabla_{w_{ij}}$ for all input to hidden layer connections via the outer product:

$$\nabla_{w_{ij}} = \left( \begin{array}{cccc} \nabla_{w_{11}} & \cdots & \nabla_{w_{1m}} \\ \nabla_{w_{21}} & \cdots & \nabla_{w_{2m}} \\ \vdots & \cdots & \vdots \\ \nabla_{w_{df}} & \cdots & \nabla_{w_{dm}} \end{array} \right) = x \cdot \delta_h^T$$  \hspace{1cm} (25.40)

where $\nabla_{w_{ij}} \in \mathbb{R}^{d \times m}$ is the matrix of weight gradients. The vector of bias gradients is given as:

$$\nabla_{b_h} = (\nabla_{b_1}, \nabla_{b_2}, \cdots, \nabla_{b_m})^T = \delta_h$$  \hspace{1cm} (25.41)

where $\nabla_{b_h} \in \mathbb{R}^m$. 
Algorithm 25.1: MLP Training: Stochastic Gradient Descent

```plaintext
MLP-TRAINING (D, m, η, maxiter):
   // Initialize bias vectors
1. bh ← random m-dimensional vector with small values
2. by ← random p-dimensional vector with small values
   // Initialize weight matrices
3. Wh ← random d × m matrix with small values
4. Wo ← random m × p matrix with small values
5. t ← 0 // iteration counter
6. repeat
7.   foreach (xi, yi) ∈ D in random order do
8.      // Feed-forward phase
9.      zi ← f(bh + WhTxi)
10.     oj ← f(by + WoTzi)
      // Backpropagation phase: net gradients
11.     δo ← oj ⊙ (1 − oj) ⊙ (oj − yi)
12.     δh ← zi ⊙ (1 − zi) ⊙ (wo · δo)
      // Gradient descent for bias vectors
13.     vbh ← δo; bh ← bh − η · vbh
14.     vb ← δh; by ← by − η · vb
      // Gradient descent for weight matrices
15.     vw ← zj · δjT; wo ← wo − η · vw
16.     vh ← xi · δjT; wh ← wh − η · vh
   until t ≥ maxiter
```

Once the gradients have been computed, we can update all the weights and biases as follows

\[
\begin{align*}
Wh &= Wh - \eta \cdot vw

by &= by - \eta \cdot vb
\end{align*}
\]

(25.42)

where \(\eta\) is the step size (or learning rate).

25.3.3 MLP Training

Algorithm 25.1 shows the pseudo-code for learning the weights considering all of the input points in \(D\) via stochastic gradient descent. The code is shown for an MLP with a single hidden layer, using a squared error function, and sigmoid activations for all hidden and output neurons. The approach is called stochastic gradient descent since we compute the weight and bias gradients after observing each training point (in random order).

The MLP algorithm takes as input the dataset \(D\) (with points \(x_i\) and desired responses \(y_i\) for \(i = 1, 2, \ldots, n\)), the number of hidden layer neurons \(m\), the learning rate
\( \eta \), and an integer threshold maxiter that specifies the maximum number of iterations. The size of the input \((d)\) and output \((p)\) layers is determined automatically from \(D\). The MLP first initializes the \(d \times m\) input to hidden layer weight matrix \(W_h\), and the \(m \times p\) hidden to output layer matrix \(W_o\) to small values, for example, uniformly random in the range \([-0.01, 0.01]\). It is important to note that weights should not be set to 0, otherwise, all hidden neurons will be identical in their values, and so will be the output neurons.

The MLP training takes multiple iterations over the input points. For each input \(x_i\), the MLP computes the output vector \(o_i\) via the feed-forward step. In the backpropagation phase, we compute the error gradient vector \(\delta_h\) with respect to the net at output neurons, followed by \(\delta_o\) for hidden neurons. In the stochastic gradient descent step, we compute the error gradients with respect to the weights and biases, which are used to update the weight matrices and bias vectors. Thus, for every input vector \(x_i\), all the weights and biases are updated based on the error incurred between the predicted output \(o_i\) and the true response \(y_i\). After each input has been processed, that completes one iteration of training, called an epoch. Training stops when the maximum number of iterations, maxiter, has been reached. On the other hand, during testing, for any input \(x\), we apply the feed-forward steps and print the predicted output \(o\).

In terms of computational complexity, each iteration of the MLP training algorithm takes \(O(dm + mp)\) time for the feed-forward phase, \(p + mp + m = O(mp)\) time for the backpropagation of error gradients, and \(O(dm + mp)\) time for updating the weight matrices and bias vectors. Thus, the total training time per iteration is \(O(dm + mp)\).

**Example 25.3 (MLP with one hidden layer).** We now illustrate an MLP with a hidden layer using a non-linear activation function to learn the sine curve. Figure 25.8(a) shows the training data (the gray points on the curve), which comprises \(n = 25\) points \(x_i\) sampled randomly in the range \([-10, 10]\), with \(y_i = \sin(x_i)\). The testing data comprises 1000 points sampled uniformly from the same range. The figure also shows the desired output curve (thin line). We used an MLP with one input neuron \((d = 1)\), ten hidden neurons \((m = 10)\) and one output neuron \((p = 1)\). The hidden neurons use tanh activations, whereas the output unit uses an identity activation. The step size is \(\eta = 0.005\).

The input to hidden weight matrix \(W_h \in \mathbb{R}^{1 \times 10}\) and the corresponding bias vector \(b_h \in \mathbb{R}^{10 \times 1}\) are given as:

\[
W_h = (-0.68, 0.77, -0.42, -0.72, -0.93, -0.42, -0.66, -0.70, -0.62, -0.50)
\]

\[
b_h = (-4.36, 2.43, -0.52, 2.35 - 1.64, 3.98, 0.31, 4.45, 1.03, -4.77)^T
\]

The hidden to output weight matrix \(W_o \in \mathbb{R}^{10 \times 1}\) and the bias term \(b_o \in \mathbb{R}\) are given as:

\[
W_o = (-1.82, -1.69, -0.82, 1.37, 0.14, 2.37, -1.64, -1.92, 0.78, 2.17)^T
\]

\[
b_o = -0.16
\]

Figure 25.8(a) shows the output of the MLP on the test set, after the first iteration of training \((t = 1)\). We can see that initially the predicted response deviates
Figure 25.8. MLP for sine curve: 10 hidden neurons with hyperbolic tangent activation functions. The gray dots represent the training data. The bold line is the predicted response, whereas the thin line is the true response. (a)–(f): Predictions after different number of iterations. (g): Testing outside the training range. Good fit within the training range $[-10, 10]$ shown in the box.
significantly from the true sine response. Figure 25.8(a)–(f) show the output from the MLP after different number of training iterations. By \( t = 15000 \) iterations the output on the test set comes close to the sine curve, but it takes another 15000 iterations to get a closer fit. The final SSE is 1.45 over the 1000 test points.

We can observe that, even with a very small training data of 25 points sampled randomly from the sine curve, the MLP is able to learn the desired function. However, it is also important to recognize that the MLP model has not really learned the sine function; rather, it has learned to approximate it only in the specified range \([-10, 10]\). We can see in Figure 25.8(g) that when we try to predict values outside this range, the MLP does not yield a good fit.

**Example 25.4 (MLP for handwritten digit classification).** In this example, we apply an MLP with one hidden layer for the task of predicting the correct label for a hand-written digit from the MNIST database, which contains 60,000 training images that span the 10 digit labels, from 0 to 9. Each (grayscale) image is a 28 \times 28 matrix of pixels, with values between 0 and 255. Each pixel is converted to a value in the interval [0, 1] by dividing by 255. Figure 25.9 shows an example of each digit from the MNIST dataset.

Since images are 2-dimensional matrices, we first flatten them into a vector \( \mathbf{x} \in \mathbb{R}^{784} \) with dimensionality \( d = 28 \times 28 = 784 \). This is done by simply concatenating all of the rows of the images to obtain one long vector. Next, since the output labels are categorical values that denote the digits from 0 to 9, we need to convert them into binary (numerical) vectors, using one-hot encoding. Thus, the label 0 is encoded as \( \mathbf{e}_1 = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)^T \in \mathbb{R}^{10} \), the label 1 as \( \mathbf{e}_2 = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0)^T \in \mathbb{R}^{10} \), and so on, and finally the label 9 is encoded as \( \mathbf{e}_{10} = (0, 0, 0, 0, 0, 0, 0, 0, 1, 0)^T \in \mathbb{R}^{10} \). That is, each input image vector \( \mathbf{x} \) has a corresponding target response vector \( \mathbf{y} \in \{\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{10}\} \). Thus, the input layer for the MLP has \( d = 784 \) neurons, and the output layer has \( p = 10 \) neurons.

For the hidden layer, we consider several MLP models, each with a different number of hidden neurons \( m \). We try \( m = 0, 7, 49, 98, 196, 392 \), to study the effect of increasing the number of hidden neurons, from small to large. For the hidden layer, we use ReLU activation function, and for the output layer, we use softmax activation, since the target response vector has only one neuron with value 1, with the rest being 0. Note that \( m = 0 \) means that there is no hidden layer – the input layer is directly connected to the output layer, which is equivalent to a multiclass logistic regression model. We train each MLP for \( t = 15 \) epochs, using step size \( \eta = 0.25 \).

During training, we plot the number of misclassified images after each epoch, on the separate MNIST test set comprising 10,000 images. Figure 25.10 shows the number of errors from each of the models (with a different number of hidden neurons \( m \)), after each epoch. The final test error at the end of training is given as

<table>
<thead>
<tr>
<th>( m )</th>
<th>0</th>
<th>7</th>
<th>10</th>
<th>49</th>
<th>98</th>
<th>196</th>
<th>392</th>
</tr>
</thead>
<tbody>
<tr>
<td>errors</td>
<td>1677</td>
<td>901</td>
<td>792</td>
<td>546</td>
<td>495</td>
<td>470</td>
<td>454</td>
</tr>
</tbody>
</table>
We can observe that adding a hidden layer significantly improves the prediction accuracy. Using even a small number of hidden neurons helps, compared to the logistic regression model ($m = 0$). For example, using $m = 7$ results in 901 errors (or error rate 9.01%) compared to using $m = 0$, which results in 1677 errors (or error rate 16.77%). On the other hand, as we increase the number of hidden neurons, the error rate decreases, though with diminishing returns. Using $m = 196$, the error rate is 4.70%, but even after doubling the number of hidden neurons ($m = 392$), the error rate goes down to only 4.54%. Further increasing $m$ does not reduce the error rate.
25.4 DEEP MULTILAYER PERCEPTRONS

We now generalize the feed-forward and backpropagation steps for many hidden layers, as well as arbitrary error and neuron activation functions.

Consider an MLP with \( h \) hidden layers as shown in Figure 25.11. We assume that the input to the MLP comprises \( n \) points \( \mathbf{x}_i \in \mathbb{R}^d \) with the corresponding true response vector \( \mathbf{y}_i \in \mathbb{R}^p \). We denote the input neurons as layer \( l = 0 \), the first hidden layer as \( l = 1 \), the last hidden layer as \( l = h \), and finally the output layer as layer \( l = h + 1 \). We use \( n_l \) to denote the number of neurons in layer \( l \). Since the input points are \( d \)-dimensional, this implies \( n_0 = d \), and since the true response vector is \( p \)-dimensional, we have \( n_{h+1} = p \). The hidden layers have \( n_1 \) neurons for the first hidden layer, \( n_2 \) for the second layer, and \( n_h \) for the last hidden layer. The vector of neuron values for layer \( l \) (for \( l = 0, \ldots, h+1 \)) is denoted as

\[
\mathbf{z}^l = \left( z^l_1, \ldots, z^l_{n_l} \right)^T
\]

Each layer except the output layer has one extra bias neuron, which is the neuron at index 0. Thus, the bias neuron for layer \( l \) is denoted \( z^l_0 \) and its value is fixed at \( z^l_0 = 1 \).

(a) detail view

(b) layer view

Figure 25.11. Deep multilayer perceptron, with \( h \) hidden layers.
Figure 25.11(a) displays a detailed view of an MLP with \( h \) hidden layers, showing the individual neurons in each layer including the bias neuron. Note that the vector of input neuron values is also written as

\[
x = (x_1, x_2, \cdots, x_d)^T = (z_1^0, z_2^0, \cdots, z_d^0)^T = z^0
\]

and the vector of output neuron values is also denoted as

\[
o = (o_1, o_2, \cdots, o_p)^T = (z_1^{h+1}, z_2^{h+1}, \cdots, z_p^{h+1})^T = z^{h+1}
\]

The weight matrix between layer \( l \) and layer \( l+1 \) neurons is denoted \( W_l \in \mathbb{R}^{n_l \times n_{l+1}} \), and the vector of bias terms from the bias neuron \( z_0^l \) to neurons in layer \( l+1 \) is denoted \( b_l \in \mathbb{R}^{n_{l+1}} \), for \( l = 0, 1, \cdots, h \). Thus, \( W_0 \in \mathbb{R}^{d \times n_1} \) is the weight matrix between the input and first hidden layer, \( W_1 \in \mathbb{R}^{n_1 \times n_2} \) is the weight matrix between the first and second hidden layer, and so on; finally, \( W_h \in \mathbb{R}^{n_h \times p} \) is the weight matrix between the last hidden layer and the output layer. For the bias vectors, \( b_0 \in \mathbb{R}^{n_1} \) specifies the biases for neurons in the first hidden layer, \( b_1 \in \mathbb{R}^{n_2} \) the biases for neurons in the second hidden layer, and so on. Thus, \( b_h \in \mathbb{R}^{p} \) specifies the biases for the output neurons. Figure 25.11(b) shows a layer view for an MLP with \( h \) hidden layers. This is a more compact representation that clearly specifies the architecture or topology of the MLP. Each layer \( l \) is represented as a rectangular node, and is marked with the vector of neuron values, \( x^l \). The bias neurons are not shown, but are assumed to be present in each layer except the output. An edge between layer \( l \) and \( l+1 \) is labeled with the weight matrix \( W_l \) and bias vector \( b_l \) that specify the parameters between those layers.

For training deep MLPs, we will refer to several partial derivative vectors, described next. Define \( \delta_i^l \) as the net gradient, i.e., the partial derivative of the error function with respect to the net value at \( z_i^l \)

\[
\delta_i^l = \frac{\partial E_x}{\partial net_i^l}
\]  

(25.43)

and let \( \delta^l \) denote the net gradient vector at layer \( l \), for \( l = 1, 2, \cdots, h+1 \)

\[
\delta^l = (\delta_1^l, \cdots, \delta_{n_l}^l)^T
\]  

(25.44)

Let \( f^l \) denote the activation function for layer \( l \), for \( l = 0, 1, \cdots, h+1 \), and further let \( \partial f^l \) denote the vector of the derivatives of the activation function with respect to \( net_i^l \) for all neurons \( z_i^l \) in layer \( l \):

\[
\partial f^l = \left( \frac{\partial f^l(net_1)}{\partial net_1}, \cdots, \frac{\partial f^l(net_{n_l})}{\partial net_{n_l}} \right)^T
\]  

(25.45)

Finally, let \( \partial E_x \) denote the vector of partial derivatives of the error function with respect to the values \( o_l \) for all output neurons:

\[
\partial E_x = \left( \frac{\partial E_x}{\partial o_1}, \frac{\partial E_x}{\partial o_2}, \cdots, \frac{\partial E_x}{\partial o_p} \right)^T
\]  

(25.46)
25.4.1 Feed-forward Phase

Typically in a deep MLP, the same activation function \( f^i \) is used for all neurons in a given layer \( l \). The input layer always uses the identity activation, so \( f^0 \) is the identity function. Also, all bias neurons also use the identity function with a fixed value of 1. The output layer typically uses sigmoid or softmax activations for classification tasks, or identity activations for regression tasks. The hidden layers typically use sigmoid, tanh, or ReLU activations. In our discussion, we assume a fixed activation function \( f^i \) for all neurons in a given layer. However, it is easy to generalize to a neuron specific activation function \( f^i_j \) for neuron \( z^i_j \) in layer \( l \).

For a given input pair \((x, y) \in \mathbf{D}\), the deep MLP computes the output vector via the feed-forward process:

\[
\mathbf{o} = f^{h+1}(\mathbf{b}_h + \mathbf{W}_h^T \cdot \mathbf{z}^h)
\]

\[
= f^{h+1}(\mathbf{b}_h + \mathbf{W}_h^T \cdot f^h(\mathbf{b}_{h-1} + \mathbf{W}_{h-1}^T \cdot \mathbf{z}^{h-1}))
\]

\[
= \vdots
\]

\[
= f^{h+1}(\mathbf{b}_h + \mathbf{W}_h^T \cdot f^h(\mathbf{b}_{h-1} + \mathbf{W}_{h-1}^T \cdot f^{h-1}(\cdots f^2(\mathbf{b}_1 + \mathbf{W}_1^T \cdot f^1(\mathbf{b}_0 + \mathbf{W}_0^T \cdot \mathbf{x})))))
\]

Note that each \( f^i \) distributes over its argument. That is,

\[
f^i(\mathbf{b}_{l-1} + \mathbf{W}_{l-1}^T \cdot \mathbf{x}) = \left( f^i(\text{net}_1), f^i(\text{net}_2), \ldots, f^i(\text{net}_n) \right)^T
\]

25.4.2 Backpropagation Phase

Consider the weight update between a given layer and another, including between the input and hidden layer, or between two hidden layers, or between the last hidden layer and the output layer. Let \( z^l_j \) be a neuron in layer \( l \), and \( z^{l+1}_j \) a neuron in the next layer \( l+1 \). Let \( w^l_{ij} \) be the weight between \( z^l_i \) and \( z^{l+1}_j \), and let \( b^l_j \) denote the bias term between \( z^l_0 \) and \( z^{l+1}_j \). The weight and bias are updated using the gradient descent approach

\[
w^l_{ij} = w^l_{ij} - \eta \cdot \nabla_w w^l_{ij} \quad \quad \quad b^l_j = b^l_j - \eta \cdot \nabla_b b^l_j
\]

where \( \nabla_w w^l_{ij} \) is the weight gradient and \( \nabla_b b^l_j \) is the bias gradient, i.e., the partial derivative of the error function with respect to the weight and bias, respectively:

\[
\nabla w^l_{ij} = \frac{\partial E_x}{\partial w^l_{ij}} \quad \quad \quad \nabla b^l_j = \frac{\partial E_x}{\partial b^l_j}
\]

As noted earlier in Eq. (25.31), we can use the chain rule to write the weight and bias gradient, as follows

\[
\nabla w^l_{ij} = \frac{\partial E_x}{\partial w^l_{ij}} = \frac{\partial E_x}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial w^l_{ij}} = \delta^{l+1}_j \cdot z^l_i = \delta^l \cdot z^{l+1}_i
\]

\[
\nabla b^l_j = \frac{\partial E_x}{\partial b^l_j} = \frac{\partial E_x}{\partial \text{net}_j} \cdot \frac{\partial \text{net}_j}{\partial b^l_j} = \delta^{l+1}_j
\]
where $\delta^{l+1}_j$ is the net gradient [Eq. (25.43)], i.e., the partial derivative of the error function with respect to the net value at $z^{l+1}_j$, and we have

$$\frac{\partial \text{net}_j}{\partial w^{l}_{ij}} = \frac{\partial}{\partial w^{l}_{ij}} \left( b^l_j + \sum_{k=0}^{n_l} w^{l}_{kj} \cdot z^l_k \right) = z^l_i$$

$$\frac{\partial \text{net}_j}{\partial b^l_j} = \frac{\partial}{\partial b^l_j} \left( b^l_j + \sum_{k=0}^{n_l} w^{l}_{kj} \cdot z^l_k \right) = 1$$

Given the vector of neuron values at layer $l$, namely $\mathbf{z}^l = (z^l_1, \ldots, z^l_{n_l})^T$, we can compute the entire weight gradient matrix via an outer product operation

$$\nabla W_l = \mathbf{z}^l \cdot (\delta^{l+1})^T$$

(25.47)

and the bias gradient vector as:

$$\nabla b_l = \delta^{l+1}$$

(25.48)

with $l = 0, 1, \ldots, h$. Here $\delta^{l+1}$ is the net gradient vector at layer $l + 1$ [Eq. (25.44)].

This also allows us to update all the weights and biases as follows

$$W_l = W_l - \eta \cdot \nabla W_l$$

$$b_l = b_l - \eta \cdot \nabla b_l$$

(25.49)

where $\eta$ is the step size. However, we observe that to compute the weight and bias gradients for layer $l$ we need to compute the net gradients $\delta^{l+1}$ at layer $l + 1$.

### 25.4.3 Net Gradients at Output Layer

Let us consider how to compute the net gradients at the output layer $h + 1$. If all of the output neurons are independent (for example, when using linear or sigmoid activations), the net gradient is obtained by differentiating the error function with respect to the net signal at the output neurons. That is,

$$\delta^{h+1}_j = \frac{\partial \mathcal{E}_x}{\partial \text{net}_j} = \frac{\partial \mathcal{E}_x}{\partial f^{h+1}(\text{net}_j)} \cdot \frac{\partial f^{h+1}(\text{net}_j)}{\partial \text{net}_j} = \frac{\partial \mathcal{E}_x}{\partial o_j} \cdot \frac{\partial f^{h+1}(\text{net}_j)}{\partial \text{net}_j}$$

Thus, the gradient depends on two terms, the partial derivative of the error function with respect to the output neuron value, and the derivative of the activation function with respect to its argument. The net gradient vector across all output neurons is given as

$$\delta^{h+1} = \partial f^{h+1} \odot \frac{\partial \mathcal{E}_x}{\partial x}$$

(25.50)

where $\odot$ is the element-wise or Hadamard product, $\partial f^{h+1}$ is the vector of derivatives of the activation function with respect to its argument [Eq. (25.45)] at the output layer $l = h + 1$, and $\partial \mathcal{E}_x$ is the vector of error derivatives with respect to the output neuron values [Eq. (25.46)].
On the other hand, if the output neurons are not independent (for example, when using a softmax activation), then we have to modify the computation of the net gradient at each output neuron as follows:

$$
\delta_j^{h+1} = \frac{\partial E_x}{\partial net_j} = \sum_{i=1}^{p} \frac{\partial E_x}{\partial f^{h+1}(net_i)} \frac{\partial f^{h+1}(net_i)}{\partial net_j}
$$

Across all output neurons, we can write this compactly as follows:

$$
\delta^{h+1} = \partial F^{h+1} \cdot \partial E_x
$$

(25.51)

where $\partial F^{h+1}$ is the matrix of derivatives of $o_i = f^{h+1}(net_i)$ with respect to $net_j$ for all $i, j = 1, 2, \cdots, p$, given as

$$
\partial F^{h+1} = \begin{pmatrix}
\frac{\partial o_1}{\partial net_1} & \frac{\partial o_1}{\partial net_2} & \cdots & \frac{\partial o_1}{\partial net_p} \\
\frac{\partial o_2}{\partial net_1} & \frac{\partial o_2}{\partial net_2} & \cdots & \frac{\partial o_2}{\partial net_p} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial o_p}{\partial net_1} & \frac{\partial o_p}{\partial net_2} & \cdots & \frac{\partial o_p}{\partial net_p}
\end{pmatrix}
$$

Typically, for regression tasks, we use the squared error function with linear activation function at the output neurons, whereas for logistic regression and classification, we use the cross-entropy error function with a sigmoid activation for binary classes, and softmax activation for multiclass problems. For these common cases, the net gradient vector at the output layer is given as follows:

**Squared Error:** From Eq. (25.15), the error gradient is given as

$$
\partial E_x = \frac{\partial E_x}{\partial \o} = \o - \y
$$

The net gradient at the output layer is given as

$$
\delta^{h+1} = \partial f^{h+1} \odot \partial E_x
$$

where $\partial f^{h+1}$ depends on the activation function at the output. Typically, for regression tasks, we use a linear activation at the output neurons. In that case, we have $\partial f^{h+1} = 1$ (see Eq. (25.8)).

**Cross-Entropy Error (binary output, sigmoid activation):** Consider the binary case first, with a single output neuron $o$ with sigmoid activation. Recall that the binary cross-entropy error [Eq. (25.19)] is given as

$$
E_x = -(y \cdot \ln(o) + (1 - y) \cdot \ln(1 - o))
$$

From Eq. (25.20) we have

$$
\partial E_x = \frac{\partial E_x}{\partial o} = \frac{o - y}{o \cdot (1 - o)}
$$
Further, for sigmoid activation, we have

\[ \delta^{k+1} = \frac{\partial f(\text{net}_o)}{\partial \text{net}_o} = o \cdot (1 - o) \]

Therefore, the net gradient at the output neuron is

\[ \delta^{k+1} \cdot \nabla \mathcal{E}_x = \frac{o - y}{o \cdot (1 - o)} \cdot o \cdot (1 - o) = o - y \]

**Cross-Entropy Error** *(K outputs, softmax activation):* Recall that the cross-entropy error function [Eq. (25.16)] is given as

\[ \mathcal{E}_x = - \sum_{i=1}^{K} y_i \ln(o_i) = - \left( y_1 \ln(o_1) + \ldots + y_K \ln(o_K) \right) \]

Using Eq. (25.17), the vector of error derivatives with respect to the output neurons is given as

\[ \nabla \mathcal{E}_x = \left( \frac{\partial \mathcal{E}_x}{\partial o_1}, \frac{\partial \mathcal{E}_x}{\partial o_2}, \ldots, \frac{\partial \mathcal{E}_x}{\partial o_K} \right)^T = \left( \frac{y_1}{o_1}, \frac{y_2}{o_2}, \ldots, \frac{y_K}{o_K} \right)^T \]

where \( p = K \) is the number of output neurons.

Cross-entropy error is typically used with the softmax activation so that we get a (normalized) probability value for each class. That is,

\[ o_j = \text{softmax}(\text{net}_j) = \frac{\exp(\text{net}_j)}{\sum_{i=1}^{K} \exp(\text{net}_i)} \]

so that the output neuron values sum to one, \( \sum_{j=1}^{K} o_j = 1 \). Since an output neuron depends on all other output neurons, we need to compute the matrix of derivatives of each output with respect to each of the net signals at the output neurons [see Equations (25.12) and (25.18)]:

\[ \nabla \mathcal{F}^{k+1} = \begin{pmatrix} \frac{\partial o_1}{\partial \text{net}_1} & \frac{\partial o_1}{\partial \text{net}_2} & \ldots & \frac{\partial o_1}{\partial \text{net}_K} \\ \frac{\partial o_2}{\partial \text{net}_1} & \frac{\partial o_2}{\partial \text{net}_2} & \ldots & \frac{\partial o_2}{\partial \text{net}_K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial o_K}{\partial \text{net}_1} & \frac{\partial o_K}{\partial \text{net}_2} & \ldots & \frac{\partial o_K}{\partial \text{net}_K} \end{pmatrix} = \begin{pmatrix} o_1 \cdot (1 - o_1) & -o_1 \cdot o_2 & \ldots & -o_1 \cdot o_K \\ -o_2 \cdot o_1 & o_2 \cdot (1 - o_2) & \ldots & -o_2 \cdot o_K \\ \vdots & \vdots & \ddots & \vdots \\ -o_K \cdot o_1 & -o_K \cdot o_2 & \ldots & o_K \cdot (1 - o_K) \end{pmatrix} \]
Therefore, the net gradient vector at the output layer is
\[ \delta^{h+1} = \partial F^{h+1} \cdot \partial E_x \]
\[ = \begin{pmatrix}
  o_1 \cdot (1 - o_1) & -o_1 \cdot o_2 & \cdots & -o_1 \cdot o_K \\
  -o_1 \cdot o_2 & o_2 \cdot (1 - o_2) & \cdots & -o_2 \cdot o_K \\
  \vdots & \vdots & \ddots & \vdots \\
  -o_1 \cdot o_K & -o_2 \cdot o_K & \cdots & o_K \cdot (1 - o_K)
\end{pmatrix} \cdot \begin{pmatrix}
  -y_1 \\
  -y_2 \\
  \vdots \\
  -y_K
\end{pmatrix} \]
\[ = \begin{pmatrix}
  -y_1 + y_1 \cdot o_1 + \sum_{j \neq 1}^K y_j \cdot o_1 \\
  -y_2 + y_2 \cdot o_2 + \sum_{j \neq 2}^K y_j \cdot o_2 \\
  \vdots \\
  -y_K + y_K \cdot o_K + \sum_{j \neq K}^K y_j \cdot o_K
\end{pmatrix} \]
\[ = \begin{pmatrix}
  -y_1 + o_1 \cdot \sum_{i=1}^K y_i \\
  -y_2 + o_2 \cdot \sum_{i=1}^K y_i \\
  \vdots \\
  -y_K + o_K \cdot \sum_{i=1}^K y_i
\end{pmatrix}
\]
\[ = \begin{pmatrix}
  -y_1 + o_1 \\
  -y_2 + o_2 \\
  \vdots \\
  -y_K + o_K
\end{pmatrix}, \text{since } \sum_{i=1}^K y_i = 1
\]
\[ = 0 - y \]

25.4.4 Net Gradients at Hidden Layers

Let us assume that we have already computed the net gradients at layer \( l + 1 \), namely \( \delta^{l+1} \). Since neuron \( z^l_j \) in layer \( l \) is connected to all of the neurons in layer \( l + 1 \) (except for the bias neuron \( z_0^{l+1} \)), to compute the net gradient at \( z^l_j \), we have to account for the error from each neuron in layer \( l + 1 \), as follows:

\[ \delta^l_j = \frac{\partial E_x}{\partial \text{net}_j} = \sum_{k=1}^{n_{l+1}} \frac{\partial E_x}{\partial \text{net}_k} \cdot \frac{\partial \text{net}_k}{\partial \text{net}_j} \cdot \frac{\partial f^l(\text{net}_j)}{\partial \text{net}_j} \]
\[ = \frac{\partial f^l(\text{net}_j)}{\partial \text{net}_j} \cdot \sum_{k=1}^{n_{l+1}} \delta^{l+1}_k \cdot w^l_{jk} \]

So the net gradient at \( z^l_j \) in layer \( l \) depends on the derivative of the activation function with respect to its \( \text{net}_j \), and the weighted sum of the net gradients from all the neurons \( z^{l+1}_k \) at the next layer \( l + 1 \).

We can compute the net gradients for all the neurons in level \( l \) in one step, as follows:

\[ \delta^l = \partial f^l \odot (W^l \cdot \delta^{l+1}) \]

where \( \odot \) is the element-wise product, and \( \partial f^l \) is the vector of derivatives of the activation function with respect to its argument [Eq. (25.45)] at layer \( l \). For the commonly used activation functions at the hidden layer, using the derivatives from
Section 25.1.1, we have
\[
\partial f^i = \begin{cases} 
1 & \text{for linear} \\
\nu'(1 - \nu') & \text{for sigmoid} \\
(1 - \nu' \odot \nu') & \text{for tanh}
\end{cases}
\]

For ReLU, we have to apply Eq. (25.9) to each neuron. Note that softmax is generally not used for hidden layers.

The net gradients are computed recursively, starting from the output layer \( h + 1 \), then hidden layer \( h \), and so on, until we finally compute the net gradients at the first hidden layer \( l = 1 \). That is,
\[
\begin{align*}
\delta^h &= \partial f^h \odot (W_h \cdot \delta^{h+1}) \\
\delta^{h-1} &= \partial f^{h-1} \odot (W_{h-1} \cdot \delta^h) = \partial f^{h-1} \odot \left( W_{h-1} \cdot \left( \partial f^h \odot (W_h \cdot \delta^{h+1}) \right) \right) \\
& \vdots \\
\delta^1 &= \partial f^1 \odot \left( W_1 \cdot \left( \partial f^2 \odot \left( W_2 \cdots \left( \partial f^h \odot (W_h \cdot \delta^{h+1}) \right) \right) \right) \right) 
\end{align*}
\]

25.4.5 Training Deep MLPs

Algorithm 25.2 shows the pseudo-code for learning the weights and biases for a deep MLP. The inputs comprise the dataset \( \mathbf{D} \), the number of hidden layers \( h \), the step size or learning rate for gradient descent \( \eta \), an integer threshold \( \text{maxiter} \) denoting the number of iterations for training, parameters \( n_1, n_2, \ldots, n_h \) that denote the number of neurons (excluding bias, which will be added automatically) for each of the hidden layers \( l = 1, 2, \ldots, h \), and the type of activation functions \( f^1, f^2, \ldots, f^{h+1} \) for each of the layers (other than the input layer that uses identity activations). The size of the input (\( d \)) and output (\( p \)) layers is determined directly from \( \mathbf{D} \).

The MLP first initializes the \((n_i \times n_{i+1})\) weight matrices \( W_l \) between layers \( l \) and \( l + 1 \) with small values chosen uniformly at random, e.g., in the range \([-0.01, 0.01]\). The MLP considers each input pair \((x_i, y_i) \in \mathbf{D}\), and computes the predicted response \( \hat{y}_i \) via the feed-forward process. The backpropagation phase begins by computing the error between \( \hat{y}_i \) and true response \( y_i \), and computing the net gradient vector \( \delta^{h+1} \) at the output layer. These net gradients are backpropagated from layer \( h + 1 \) to layer \( h \), from \( h \) to \( h - 1 \), and so on until we obtain the net gradients at the first hidden layer \( l = 1 \). These net gradients are used to compute the weight gradient matrix \( \nabla W \) at layer \( l \), which can in turn be used to update the weight matrix \( W_l \). Likewise, the net gradients specify the bias gradient vector \( \nabla b_l \) at layer \( l \), which is used to update \( b_l \). After each point has been used to update the weights, that completes one iteration or epoch of training. The training stops when \text{maxiter} epochs have been reached. On the other hand, during testing, for any input \( x \), we apply the feed-forward steps and print the predicted output \( \hat{y} \).

It is important to note that Algorithm 25.2 follows a stochastic gradient descent approach, since the points are considered in random order, and the weight and bias gradients are computed after observing each training point. In practice, it is common
Algorithm 25.2: Deep MLP Training: Stochastic Gradient Descent

\textbf{DEEP-MLP-TRAINING ($D, h, \eta, \text{maxiter}, n_1, \ldots, n_h, f^1, f^2, \ldots, f^{h+1}$):}

1. $n_0 \leftarrow d$ // input layer size
2. $n_{h+1} \leftarrow p$ // output layer size
   // Initialize weight matrices and bias vectors
3. for $l = 0, 1, 2, \ldots, h$ do
4.   $b_l \leftarrow$ random $n_{l+1}$ vector with small values
5.   $W_l \leftarrow$ random $n_l \times n_{l+1}$ matrix with small values
6. $t \leftarrow 0$ // iteration counter
7. repeat
8.   foreach $(x_i, y_i) \in D$ in random order do
9.     // Feed-Forward Phase
10.    $z^0 \leftarrow x_i$
11.   for $l = 0, 1, 2, \ldots, h$ do
12.     $z^{l+1} \leftarrow \Phi^{l+1}(b_l + W^T_l \cdot z^l)$
13.     $o_l \leftarrow z^{h+1}$
14.   // Backpropagation Phase
15.   if independent outputs then
16.     $\delta^{h+1} \leftarrow \partial f^{h+1} \odot \partial \mathcal{E}_x$ // net gradients at output
17.   else
18.     $\delta^{h+1} \leftarrow \partial f^{h+1} \cdot \partial \mathcal{E}_x$ // net gradients at output
19.   for $l = h, h-1, \ldots, 1$ do
20.     $\delta^l \leftarrow \partial f^l \odot (W_l \cdot \delta^{l+1})$ // net gradients at layer $l$
21.   // Gradient Descent Step
22.   for $l = 0, 1, \ldots, h$ do
23.     $\nabla W_l \leftarrow z^l \cdot (\delta^{l+1})^T$ // weight gradient matrix at layer $l$
24.     $\nabla b_l \leftarrow \delta^{l+1}$ // bias gradient vector at layer $l$
25.   for $l = 0, 1, \ldots, h$ do
26.     $W_l \leftarrow W_l - \eta \cdot \nabla W_l$ // update $W_l$
27.     $b_l \leftarrow b_l - \eta \cdot \nabla b_l$ // update $b_l$
28.     $t \leftarrow t + 1$
29. until $t \geq \text{maxiter}$

to update the gradients by considering a fixed sized subset of the training points called a \textit{minibatch} instead of using single points. That is, the training data is divided into minibatches using an additional parameter called \textit{batch size}, and a gradient descent step is performed after computing the bias and weight gradient from each minibatch. This helps better estimate the gradients, and also allows vectorized matrix operations over the minibatch of points, which can lead to faster convergence and substantial speedups in the learning.
One caveat while training very deep MLPs is the problem of vanishing and exploding gradients. In the *vanishing gradient* problem, the norm of the net gradient can decay exponentially with the distance from the output layer, that is, as we backpropagate the gradients from the output layer to the input layer. In this case the network will learn extremely slowly, if at all, since the gradient descent method will make minuscule changes to the weights and biases. On the other hand, in the *exploding gradient* problem, the norm of the net gradient can grow exponentially with the distance from the output layer. In this case, the weights and biases will become exponentially large, resulting in a failure to learn. The gradient explosion problem can be mitigated to some extent by *gradient thresholding*, that is, by resetting the value if it exceeds an upper bound. The vanishing gradients problem is more difficult to address. Typically sigmoid activations are more susceptible to this problem, and one solution is to use alternative activation functions such as ReLU. In general, recurrent neural networks, which are deep neural networks with *feedback* connections, are more prone to vanishing and exploding gradients; we will revisit these issues in Section 26.2.

**Example 25.5 (Deep MLP).** We now examine deep MLPs for predicting the labels for the MNIST handwritten images dataset that we considered in Example 25.4. Recall that this dataset has \( n = 60000 \) grayscale images of size \( 28 \times 28 \) that we treat as \( d = 784 \) dimensional vectors. The pixel values between 0 and 255 are converted to the range 0 and 1 by dividing each value by 255. The target response vector is a one-hot encoded vector for class labels \( \{0, 1, \ldots, 9\} \). Thus, the input to the MLP \( \mathbf{x}_i \) has dimensionality \( d = 784 \), and the output layer has dimensionality \( p = 10 \). We use softmax activation for the output layer. We use ReLU activation for the hidden layers, and consider several deep models with different number and sizes of the hidden layers. We use step size \( \eta = 0.3 \) and train for \( t = 15 \) epochs. Training was done using minibatches, using batch size of 1000.

During the training of each of the deep MLPs, we evaluate its performance on the separate MNIST test dataset that contains 10,000 images. Figure 25.12 plots the

![Figure 25.12. MNIST: Deep MLPs; prediction error as a function of epochs.](image-url)
number of errors after each epoch for the different deep MLP models. The final test error at the end of training is given as

<table>
<thead>
<tr>
<th>hidden layers</th>
<th>errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_1 = 392 )</td>
<td>396</td>
</tr>
<tr>
<td>( n_1 = 196, n_2 = 49 )</td>
<td>303</td>
</tr>
<tr>
<td>( n_1 = 392, n_2 = 196, n_3 = 49 )</td>
<td>290</td>
</tr>
<tr>
<td>( n_1 = 392, n_2 = 196, n_3 = 98, n_4 = 49 )</td>
<td>278</td>
</tr>
</tbody>
</table>

We can observe that as we increase the number of layers, we do get performance improvements. The deep MLP with four hidden layers of sizes \( n_1 = 392, n_2 = 196, n_3 = 98, n_4 = 49 \) results in an error rate of 2.78\% on the training set, whereas the MLP with a single hidden layer of size \( n_1 = 392 \) has an error rate of 3.96\%. Thus, the deeper MLP significantly improves the prediction accuracy. However, adding more layers does not reduce the error rate, and can also lead to performance degradation.

25.5 FURTHER READING

Artificial neural networks have their origin in the work of McCulloch and Pitts (1943). The first application of a single neuron, called a perceptron, to supervised learning was by Rosenblatt (1958). Minsky and Papert (1969) pointed out limitations of perceptrons, which were not overcome until the development of the backpropagation algorithm, which was introduced by Rumelhart, Hinton, and Williams (1986) to train general multilayer perceptrons.


25.6 EXERCISES

**Q1.** Consider the neural network in Figure 25.13. Let bias values be fixed at 0, and let the weight matrices between the input and hidden, and hidden and output layers, respectively, be:

\[ \mathbf{W} = (w_1, w_2, w_3) = (1, 1, -1) \quad \mathbf{W'} = (w'_1, w'_2, w'_3)^T = (0.5, 1, 2)^T \]

Assume that the hidden layer uses ReLU, whereas the output layer uses sigmoid activation. Assume SSE error. Answer the following questions, when the input is \( x = 4 \) and the true response is \( y = 0 \):