sum of all the attributes. If there are missing values in a training instance, it can be discarded from the training set. However, if there are missing values in a test instance, then logistic regression would fail to predict its class label.

6.7 Artificial Neural Network (ANN)

Artificial neural networks (ANN) are powerful classification models that are able to learn highly complex and nonlinear decision boundaries purely from the data. They have gained widespread acceptance in several applications such as vision, speech, and language processing, where they have been repeatedly shown to outperform other classification models (and in some cases even human performance). Historically, the study of artificial neural networks was inspired by attempts to emulate biological neural systems. The human brain consists primarily of nerve cells called **neurons**, linked together with other neurons via strands of fiber called **axons**. Whenever a neuron is stimulated (e.g., in response to a stimuli), it transmits nerve activations via axons to other neurons. The receptor neurons collect these nerve activations using structures called **dendrites**, which are extensions from the cell body of the neuron. The strength of the contact point between a dendrite and an axon, known as a **synapse**, determines the connectivity between neurons. Neuroscientists have discovered that the human brain learns by changing the strength of the synaptic connection between neurons upon repeated stimulation by the same impulse.

The human brain consists of approximately 100 billion neurons that are inter-connected in complex ways, making it possible for us to learn new tasks and perform regular activities. Note that a single neuron only performs a simple modular function, which is to respond to the nerve activations coming from sender neurons connected at its dendrite, and transmit its activation to receptor neurons via axons. However, it is the composition of these simple functions that together is able to express complex functions. This idea is at the basis of constructing artificial neural networks.

Analogous to the structure of a human brain, an artificial neural network is composed of a number of processing units, called nodes, that are connected with each other via directed links. The nodes correspond to neurons that perform the basic units of computation, while the directed links correspond to connections between neurons, consisting of axons and dendrites. Further, the weight of a directed link between two neurons represents the strength of the synaptic connection between neurons. As in biological neural systems, the primary objective of ANN is to adapt the weights of the links until they fit the input-output relationships of the underlying data.

The basic motivation behind using an ANN model is to extract useful features from the original attributes that are most relevant for classification. Traditionally, feature extraction has been achieved by using dimensionality reduction techniques such as PCA (introduced in Chapter 2), which show limited success in extracting nonlinear features, or by using hand-crafted features provided by domain experts. By using a complex combination of inter-connected nodes, ANN models are able to extract much richer sets of features, resulting in good classification performance. Moreover, ANN models provide a natural way of representing features at multiple levels of abstraction, where complex features are seen as compositions of simpler features. In many classification problems, modeling such a hierarchy of features turns out to be very useful. For example, in order to detect a human face in an image, we can first identify low-level features such as sharp edges with different gradients and orientations. These features can then be combined to identify facial parts such as eyes, nose, ears, and lips. Finally, an appropriate arrangement of facial parts can be used to correctly identify a human face. ANN models provide a powerful architecture to represent a hierarchical abstraction of features, from lower levels of abstraction (e.g., edges) to higher levels (e.g., facial parts).

Artificial neural networks have had a long history of developments spanning over five decades of research. Although classical models of ANN suffered from several challenges that hindered progress for a long time, they have re-emerged with widespread popularity because of a number of recent developments in the last decade, collectively known as **deep learning**. In this section, we examine classical approaches for learning ANN models, starting from the simplest model called **perceptrons** to more complex architectures called **multi-layer neural networks**. In the next section, we discuss some of the recent advancements in the area of ANN that have made it possible to effectively learn modern ANN models with deep architectures.

6.7.1 Perceptron

A perceptron is a basic type of ANN model that involves two types of nodes: input nodes, which are used to represent the input attributes, and an output node, which is used to represent the model output. Figure 6.20 illustrates the basic architecture of a perceptron that takes three input attributes, x_1, x_2, and x_3, and produces a binary output y. The input node corresponding to an attribute x_i is connected via a weighted link w_i to the output node. The weighted link is used to emulate the strength of a synaptic connection between neurons.

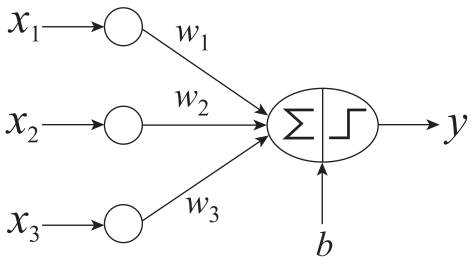


Figure 6.20. Basic architecture of a perceptron.

The output node is a mathematical device that computes a weighted sum of its inputs, adds a bias factor b to the sum, and then examines the sign of the result to produce the output $\hat{y}$ as follows:

$$\hat{3}y = \begin{cases} 1, & \text{if } \mathbf{w}^T\mathbf{x} + b > 0. \\ -1, & \text{otherwise.} \end{cases} \tag{6.48}$$

To simplify notations, $\mathbf{w}$ and b can be concatenated to form $\tilde{\mathbf{w}} = (\mathbf{w}^T\ b)^T$, while $\mathbf{x}$ can be appended with 1 at the end to form $\tilde{\mathbf{x}} = (\mathbf{x}^T\ 1)^T$. The output of the perceptron $\hat{y}$ can then be written:

$$\hat{y} = sign(\tilde{\mathbf{w}}^T\tilde{\mathbf{x}}),$$

where the sign function acts as an **activation function** by providing an output value of +1 if the argument is positive and −1 if its argument is negative.

Learning the Perceptron

Given a training set, we are interested in learning parameters $\tilde{\mathbf{w}}$ such that $\hat{y}$ closely resembles the true y of training instances. This is achieved by using the perceptron learning algorithm given in Algorithm 6.3. The key computation for this algorithm is the iterative weight update formula given in Step 8 of the algorithm:

$$w_j^{(k+1)} = w_j^{(k)} + \lambda\big(y_i - \hat{y}_i^{(k)}\big)x_{ij}, \tag{6.49}$$

where $w^{(k)}$ is the weight parameter associated with the i^{th} input link after the k^{th} iteration, λ is a parameter known as the **learning rate**, and x_{ij} is the value of the j^{th} attribute of the training example $\mathbf{x}_i$. The justification for Equation 6.49 is rather intuitive. Note that $(y_i - \hat{y}_i)$ captures the discrepancy between y_i and $\hat{y}_i$, such that its value is 0 only when the true label and the predicted output match. Assume x_{ij} is positive. If $\hat{y} = 0$ and $y = 1$, then w_j is increased at the next iteration so that $\tilde{\mathbf{w}}^T\mathbf{x}_i$ can become positive. On the other hand, if $\hat{y} = 1$ and $y = 0$, then w_j is decreased so that $\tilde{\mathbf{w}}^T\mathbf{x}_i$ can become negative. Hence, the weights are modified at every iteration to reduce the discrepancies between $\hat{y}$ and y across all training instances. The learning rate λ, a parameter whose value is between 0 and 1, can be used to control the amount of adjustments made in each iteration. The algorithm halts when the average number of discrepancies are smaller than a threshold γ.

Algorithm 6.3 Perceptron learning algorithm.

```
1: Let $D.train = \{(\tilde{\mathbf{x}}_i, y_i) \mid i = 1, 2, \ldots, n\}$ be the set of training instances.
2: Set $k \leftarrow 0$.
3: Initialize the weight vector $\tilde{\mathbf{w}}^{(0)}$ with random values.
4: repeat
5:   for each training instance $(\tilde{\mathbf{x}}_i, y_i) \in D.train$ do
6:     Compute the predicted output $\hat{y}_i^{(k)}$ using $\tilde{\mathbf{w}}^{(k)}$.
7:     for each weight component $w_j$ do
8:       Update the weight, $w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij}$.
9:     end for
10:    Update $k \leftarrow k + 1$.
11:  end for
12: until $\sum_{i=1}^{n} |y_i - \hat{y_i}^{(k)}|/n$ is less than a threshold $\gamma$
```

The perceptron is a simple classification model that is designed to learn linear decision boundaries in the attribute space. Figure 6.21 shows the decision boundary obtained by applying the perceptron learning algorithm to the data set provided on the left of the figure. However, note that there can be multiple decision boundaries that can separate the two classes, and the perceptron arbitrarily learns one of these boundaries depending on the random initial values of parameters. (The selection of the optimal decision boundary is a problem that will be revisited in the context of support vector machines in Section 6.9.) Further, the perceptron learning algorithm is only guaranteed to converge when the classes are linearly separable. However, if the classes are not linearly separable, the algorithm fails to converge. Figure 6.22 shows

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

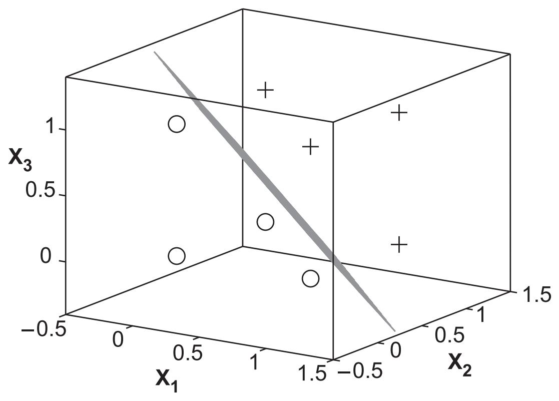


Figure 6.21. Perceptron decision boundary for the data given on the left (+ represents a positively labeled instance while o represents a negatively labeled instance.

an example of a nonlinearly separable data given by the XOR function. The perceptron cannot find the right solution for this data because there is no linear decision boundary that can perfectly separate the training instances. Thus, the stopping condition at line 12 of Algorithm 6.3 would never be met and hence, the perceptron learning algorithm would fail to converge. This is a major limitation of perceptrons since real-world classification problems often involve nonlinearly separable classes.

X_1	X_2	y
0	0	−1
1	0	1
0	1	1
1	1	−1

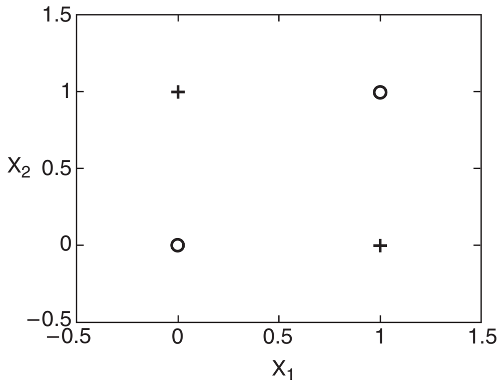


Figure 6.22. XOR classification problem. No linear hyperplane can separate the two classes.

6.7.2 Multi-layer Neural Network

A multi-layer neural network generalizes the basic concept of a perceptron to more complex architectures of nodes that are capable of learning nonlinear decision boundaries. A generic architecture of a multi-layer neural network is shown in Figure 6.23 where the nodes are arranged in groups called layers. These layers are commonly organized in the form of a chain such that every layer operates on the outputs of its preceding layer. In this way, the layers represent different levels of *abstraction* that are applied on the input features in a sequential manner. The composition of these abstractions generates the final output at the last layer, which is used for making predictions. In the following, we briefly describe the three types of layers used in multi-layer neural networks.

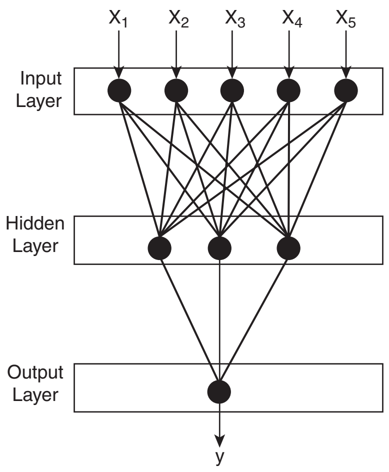
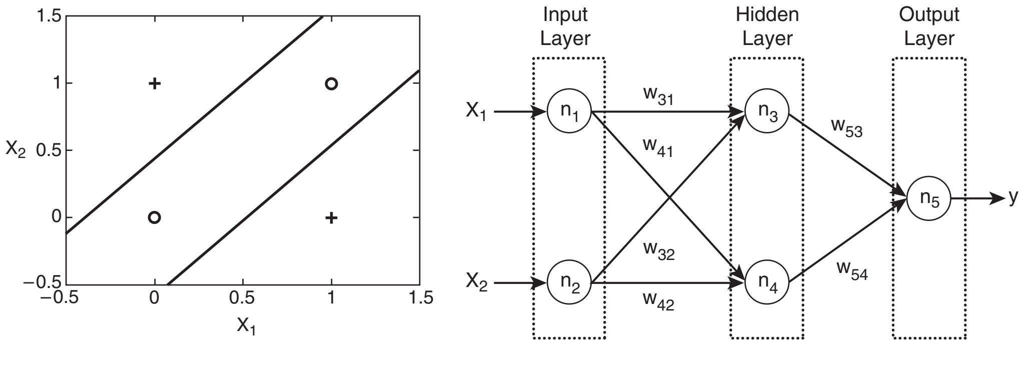


Figure 6.23. Example of a multi-layer artificial neural network (ANN).

The first layer of the network, called the **input layer**, is used for representing inputs from attributes. Every numerical or binary attribute is typically represented using a single node on this layer, while a categorical attribute is either represented using a different node for each categorical value, or by encoding the k-ary attribute using $\lceil \log_2 k \rceil$ input nodes. These inputs are fed into intermediary layers known as **hidden layers**, which are made up of processing units known as hidden nodes. Every hidden node operates on signals received from the input nodes or hidden nodes at the preceding layer, and produces an activation value that is transmitted to the next layer. The



(a) Decision boundary. (b) Neural network topology.

Figure 6.24. A two-layer neural network for the XOR problem.

final layer is called the **output layer** and processes the activation values from its preceding layer to produce predictions of output variables. For binary classification, the output layer contains a single node representing the binary class label. In this architecture, since the signals are propagated only in the forward direction from the input layer to the output layer, they are also called **feedforward neural networks**.

A major difference between multi-layer neural networks and perceptrons is the inclusion of hidden layers, which dramatically improves their ability to represent arbitrarily complex decision boundaries. For example, consider the XOR problem described in the previous section. The instances can be classified using two hyperplanes that partition the input space into their respective classes, as shown in Figure 6.24(a). Because a perceptron can create only one hyperplane, it cannot find the optimal solution. However, this problem can be addressed by using a hidden layer consisting of two nodes, as shown in Figure 6.24(b). Intuitively, we can think of each hidden node as a perceptron that tries to construct one of the two hyperplanes, while the output node simply combines the results of the perceptrons to yield the decision boundary shown in Figure 6.24(a).

The hidden nodes can be viewed as learning latent representations or *features* that are useful for distinguishing between the classes. While the first hidden layer directly operates on the input attributes and thus captures simpler features, the subsequent hidden layers are able to combine them and construct more complex features. From this perspective, multi-layer neural networks learn a hierarchy of features at different levels of abstraction that

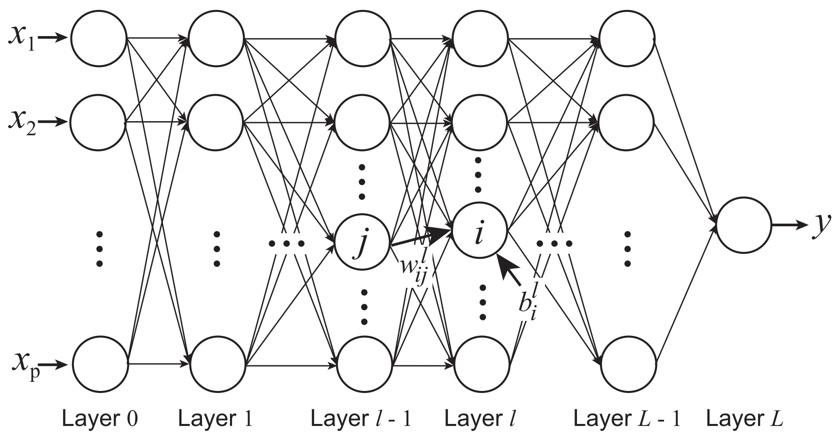


Figure 6.25. Schematic illustration of the parameters of an ANN model with $(L-1)$ hidden layers.

are finally combined at the output nodes to make predictions. Further, there are combinatorially many ways we can combine the features learned at the hidden layers of ANN, making them highly expressive. This property chiefly distinguishes ANN from other classification models such as decision trees, which can learn partitions in the attribute space but are unable to combine them in exponential ways.

To understand the nature of computations happening at the hidden and output nodes of ANN, consider the i^{th} node at the l^{th} layer of the network $(l > 0)$, where the layers are numbered from 0 (input layer) to L (output layer), as shown in Figure 6.25. The activation value generated at this node, a_i^l, can be represented as a function of the inputs received from nodes at the preceding layer. Let w_{ij}^l represent the weight of the connection from the j^{th} node at layer $(l-1)$ to the i^{th} node at layer l. Similarly, let us denote the bias term at this node as b_i^l. The activation value a_i^l can then be expressed as

$$a_i^l = f(z_i^l) = f\Big(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\Big),$$

where z is called the *linear predictor* and $f(.)$ is the activation function that converts z to a. Further, note that, by definition, $a_j^0 = x_j$ at the input layer and $a^L = \hat{y}$ at the output node.

There are a number of alternate activation functions apart from the sign function that can be used in multi-layer neural networks. Some examples include linear, sigmoid (logistic), and hyperbolic tangent functions, as shown in Figure 6.26. These functions are able to produce real-valued and nonlinear

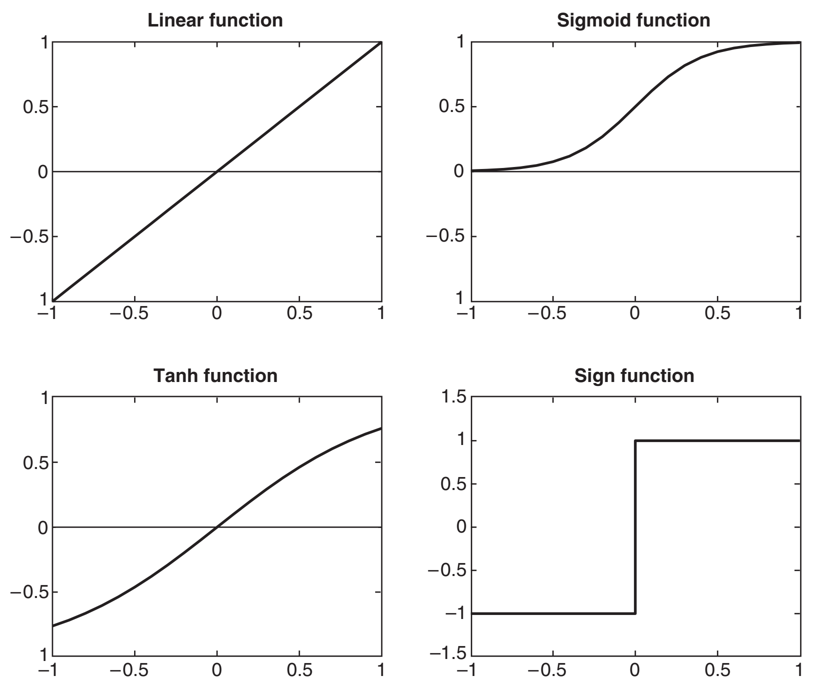


Figure 6.26. Types of activation functions used in multi-layer neural networks.

activation values. Among these activation functions, the sigmoid $\sigma(.)$ has been widely used in many ANN models, although the use of other types of activation functions in the context of deep learning will be discussed in Section 6.8. We can thus represent a_i^l as

$$a_i^l = \sigma(z_i^l) = \frac{1}{1+e^{-z_i^l}}. \tag{6.50}$$

Learning Model Parameters

The weights and bias terms $(\mathbf{w}, \mathbf{b})$ of the ANN model are learned during training so that the predictions on training instances match the true labels. This is achieved by using a loss function

$$E(\mathbf{w}, \mathbf{b}) = \sum_{k=1}^{n} \text{Loss } (y_k,\ \hat{y}_k) \tag{6.51}$$

where y_k is the true label of the k^{th} training instance and $\hat{y}_k$ is equal to a^L, produced by using $\mathbf{x}_k$. A typical choice of the loss function is the **squared loss function**:.

$$\text{Loss}\ (y_k,\ \hat{y}_k) = (y_k - \hat{y}_k)^2. \tag{6.52}$$

Note that $E(\mathbf{w}, \mathbf{b})$ is a function of the model parameters $(\mathbf{w}, \mathbf{b})$ because the output activation value a^L depends on the weights and bias terms. We are interested in choosing $(\mathbf{w}, \mathbf{b})$ that minimizes the training loss $E(\mathbf{w}, \mathbf{b})$. Unfortunately, because of the use of hidden nodes with nonlinear activation functions, $E(\mathbf{w}, \mathbf{b})$ is not a convex function of $\mathbf{w}$ and $\mathbf{b}$, which means that $E(\mathbf{w}, \mathbf{b})$ can have local minima that are not globally optimal. However, we can still apply standard optimization techniques such as the **gradient descent method** to arrive at a locally optimal solution. In particular, the weight parameter w_{ij}^l and the bias term b_i^l can be iteratively updated using the following equations:

$$w_{ij}^l \longleftarrow w_{ij}^l - \lambda \frac{\partial E}{\partial w_{ij}^l}, \tag{6.53}$$

$$b_i^l \longleftarrow b_i^l - \lambda \frac{\partial E}{\partial b_i^l}, \tag{6.54}$$

where λ is a hyper-parameter known as the learning rate. The intuition behind this equation is to move the weights in a direction that reduces the training loss. If we arrive at a minima using this procedure, the gradient of the training loss will be close to 0, eliminating the second term and resulting in the convergence of weights. The weights are commonly initialized with values drawn randomly from a Gaussian or a uniform distribution.

A necessary tool for updating weights in Equation 6.53 is to compute the partial derivative of E with respect to w_{ij}^l. This computation is non-trivial especially at hidden layers ($l < L$), since w_{ij}^l does not directly affect $\hat{y} = aL$ (and hence the training loss), but has complex chains of influences via activation values at subsequent layers. To address this problem, a technique known as **backpropagation** was developed, which propagates the derivatives backward from the output layer to the hidden layers. This technique can be described as follows.

Recall that the training loss E is simply the sum of individual losses at training instances. Hence the partial derivative of E can be decomposed as a sum of partial derivatives of individual losses.

$$\frac{\partial E}{\partial w_j^l} = \sum_{k=1}^{n} \frac{\partial\ \text{Loss}\ (y_k,\ \hat{y}_k)}{\partial w_j^l}.$$

To simplify discussions, we will consider only the derivatives of the loss at the k^{th} training instance, which will be generically represented as $\text{Loss}(y, a^L)$. By using the chain rule of differentiation, we can represent the partial derivatives of the loss with respect to w_{ij}^l as

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \frac{\partial \text{ Loss}}{\partial a_i^l} \times \frac{\partial a_i^l}{\partial z_i^l} \times \frac{\partial z_i^l}{\partial w_{ij}^l}. \tag{6.55}$$

The last term of the previous equation can be written as

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = \frac{\partial \left(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\right)}{\partial w_{ij}^l} = a_j^{l-1}.$$

Also, if we use the sigmoid activation function, then

$$\frac{\partial a_i^l}{\partial z_i^l} = \frac{\partial \ \sigma(z_i^l)}{\partial z_i^l} = a_i^l(1 - a_i^l).$$

Equation 6.55 can thus be simplified as

$$\begin{aligned} \frac{\partial \text{ Loss}}{\partial w_{ij}^l} &= \delta_i^l \times a_i^l(1 - a_i^l) \times a_j^{l-1}, \qquad (6.56) \\ \text{where } \delta_i^l &= \frac{\partial \text{ Loss}}{\partial a_i^l}. \end{aligned}$$

A similar formula for the partial derivatives with respect to the bias terms b_i^l is given by

$$\frac{\partial \text{ Loss}}{\partial b_i^l} = \delta_i^l \times a_i^l(1 - a_i^l). \tag{6.57}$$

Hence, to compute the partial derivatives, we only need to determine δ_i^l. Using a squared loss function, we can easily write δ^L at the output node as

$$\delta^L = \frac{\partial \text{ Loss}}{\partial a^L} = \frac{\partial \ (y - a^L)^2}{\partial a^L} = 2(a^L - y). \tag{6.58}$$

However, the approach for computing δ_j^l at hidden nodes $(l < L)$ is more involved. Notice that a_j^l affects the activation values a_i^{l+1} of all nodes at the next layer, which in turn influences the loss. Hence, again using the chain rule of differentiation, δ_j^l can be represented as

$$\begin{aligned} \delta_j^l &= \frac{\partial \text{ Loss}}{\partial a_j^l} = \sum_i \left(\frac{\partial \text{ Loss}}{\partial a_i^{l+1}} \times \frac{\partial a_i^{l+1}}{\partial a_j^l} \right). \\ &= \sum_i \left(\frac{\partial \text{ Loss}}{\partial a_i^{l+1}} \times \frac{\partial a_i^{l+1}}{\partial z_i^{l+1}} \times \frac{\partial z_i^{l+1}}{\partial a_j^l} \right). \\ &= \sum_i (\delta_i^{l+1} \times a_i^{l+1}(1 - a_i^{l+1}) \times w_{ij}^{l+1}). \end{aligned} \tag{6.59}$$

The previous equation provides a concise representation of the δ_j^l values at layer l in terms of the δ_i^{l+1} values computed at layer $l+1$. Hence, proceeding backward from the output layer L to the hidden layers, we can recursively apply Equation 6.59 to compute δ_i^l at every hidden node. δ_i^l can then be used in Equations 6.56 and 6.57 to compute the partial derivatives of the loss with respect to w_{ij}^l and b_i^l, respectively. Algorithm 6.4 summarizes the complete approach for learning the model parameters of ANN using backpropagation and gradient descent method.

Algorithm 6.4 Learning ANN using backpropagation and gradient descent.

```
1:  Let $D.train = \{(\mathbf{x}_k, y_k) \mid k = 1, 2, \ldots, n\}$ be the set of training instances.
2:  Set counter $c \leftarrow 0$.
3:  Initialize the weight and bias terms $(\mathbf{w}^{(0)}, \mathbf{b}^{(0)})$ with random values.
4:  **repeat**
5:      **for** each training instance $(\mathbf{x}_k, y_k) \in D.train$ **do**
6:          Compute the set of activations $(a_i^l)_k$ by making a forward pass using $\mathbf{x}_k$.
7:          Compute the set $(\delta_i^l)_k$ by backpropagation using Equations 6.58 and 6.59.
8:          Compute $(\partial \text{ Loss}/\partial w_{ij}^l,\ \partial \text{ Loss}/\partial b_i^l)_k$ using Equations 6.56 and 6.57.
9:      **end for**
10:     Compute $\partial E/\partial w_{ij}^l \longleftarrow \sum_{k=1}^n (\partial \text{ Loss}/\partial w_{ij}^l)_k$.
11:     Compute $\partial E/\partial b_i^l \longleftarrow \sum_{k=1}^n (\partial \text{ Loss}/\partial b_i^l)_k$.
12:     Update $(\mathbf{w}^{(c+1)}, \mathbf{b}^{(c+1)})$ by gradient descent using Equations 6.53 and 6.54.
13:     Update $c \leftarrow c + 1$.
14: **until** $(\mathbf{w}^{(c+1)}, \mathbf{b}^{(c+1)})$ and $(\mathbf{w}^{(c)}, \mathbf{b}^{(c)})$ converge to the same value
```

6.7.3 Characteristics of ANN

1. Multi-layer neural networks with at least one hidden layer are **universal approximators**; i.e., they can be used to approximate any target function. They are thus highly expressive and can be used to learn complex decision boundaries in diverse applications. ANN can also be used for multiclass classification and regression problems, by appropriately modifying the output layer. However, the high model complexity of classical ANN models makes it susceptible to overfitting, which can be overcome to some extent by using deep learning techniques discussed in Section 6.8.3.

2. ANN provides a natural way to represent a hierarchy of features at multiple levels of abstraction. The outputs at the final hidden layer of the ANN model thus represent features at the highest level of abstraction that are most useful for classification. These features can also be used as inputs in other supervised classification models, e.g., by replacing the output node of the ANN by any generic classifier.

3. ANN represents complex high-level features as compositions of simpler lower-level features that are easier to learn. This provides ANN the ability to gradually increase the complexity of representations, by adding more hidden layers to the architecture. Further, since simpler features can be combined in combinatorial ways, the number of complex features learned by ANN is much larger than traditional classification models. This is one of the main reasons behind the high expressive power of deep neural networks.

4. ANN can easily handle irrelevant attributes, by using zero weights for attributes that do not help in improving the training loss. Also, redundant attributes receive similar weights and do not degrade the quality of the classifier. However, if the number of irrelevant or redundant attributes is large, the learning of the ANN model may suffer from overfitting, leading to poor generalization performance.

5. Since the learning of ANN model involves minimizing a non-convex function, the solutions obtained by gradient descent are not guaranteed to be globally optimal. For this reason, ANN has a tendency to get stuck in local minima, a challenge that can be addressed by using deep learning techniques discussed in Section 6.8.4.