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.

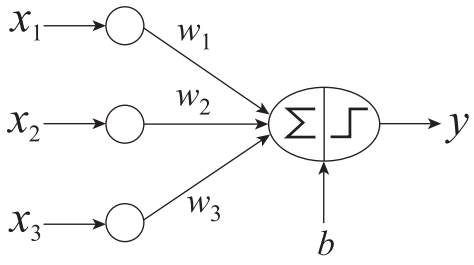


Figure 6.20. Basic architecture of a perceptron.

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:

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

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 γ

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

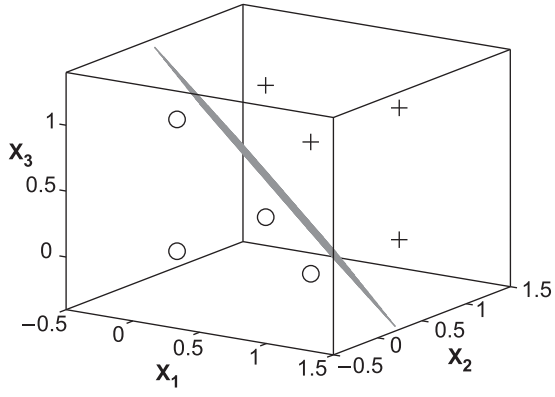


Figure 6.21. Perceptron decision boundary for the data given on the left (+ represents a positively labeled instance while o represents a negatively labeled instance.

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

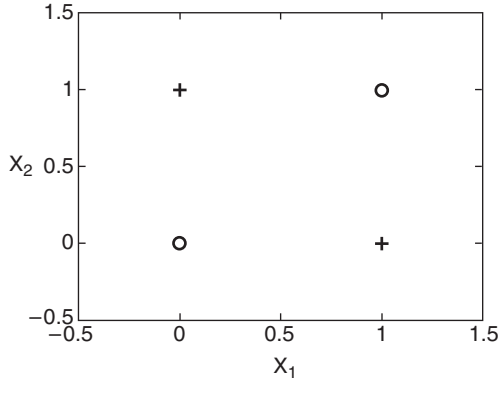


Figure 6.22. XOR classification problem. No linear hyperplane can separate the two classes.

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.

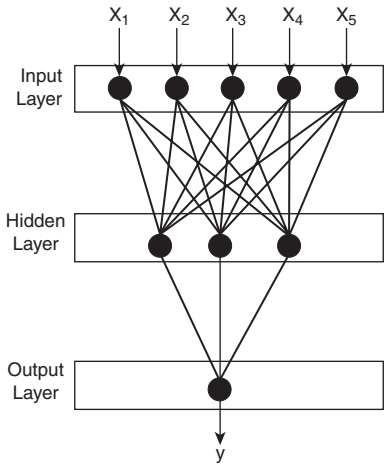
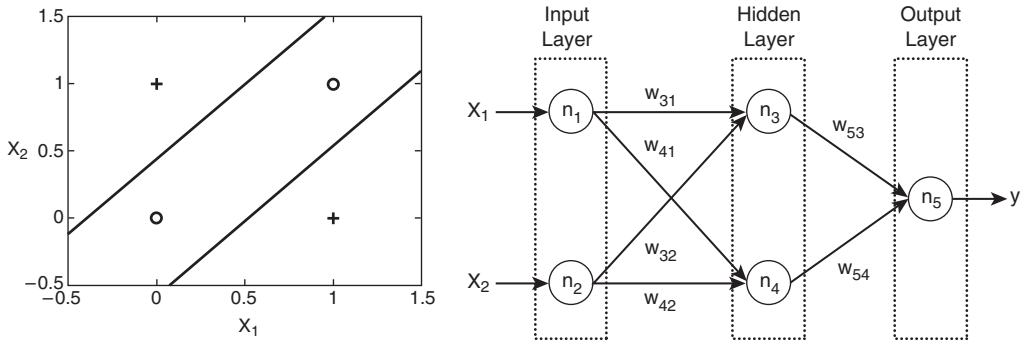


Figure 6.23. Example of a multi-layer artificial neural network (ANN).

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

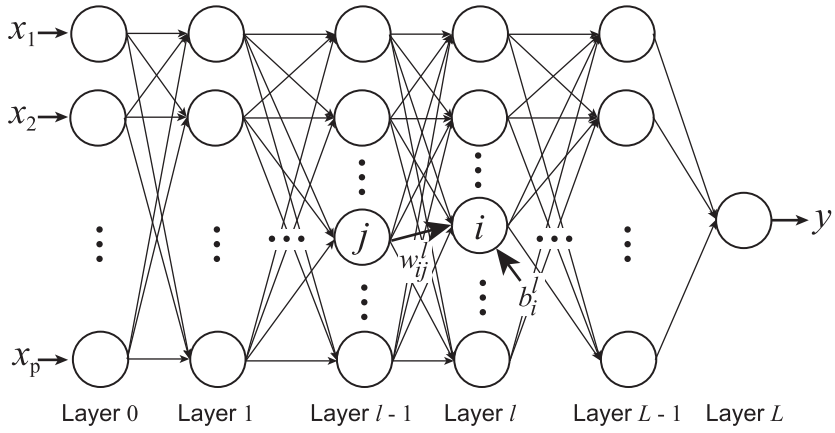


Figure 6.25. Schematic illustration of the parameters of an ANN model with $(L-1)$ hidden layers.

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

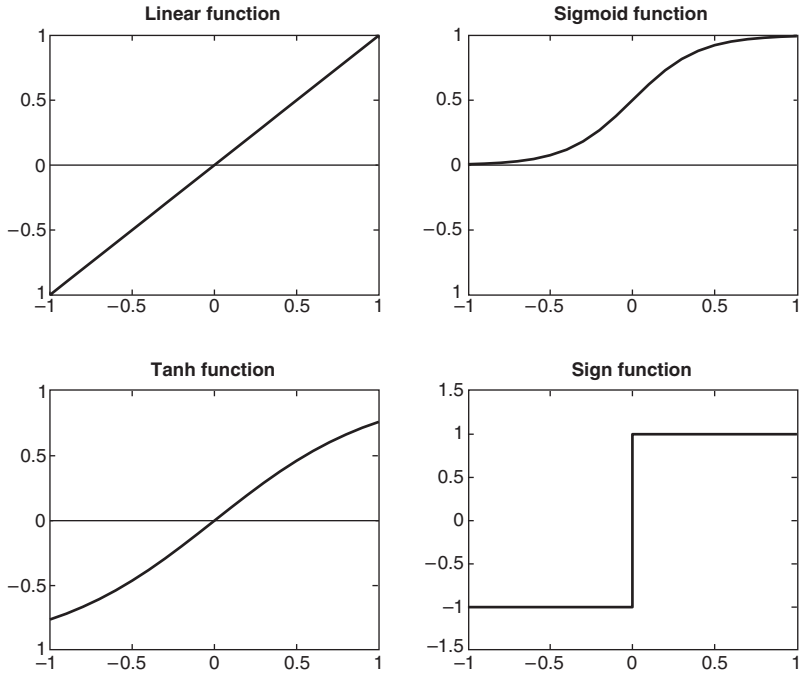


Figure 6.26. Types of activation functions used in multi-layer neural networks.

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

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

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.

6. Training an ANN is a time consuming process, especially when the number of hidden nodes is large. Nevertheless, test examples can be classified rapidly.

7. Just like logistic regression, ANN can learn in the presence of interacting variables, since the model parameters are jointly learned over all variables together. In addition, ANN cannot handle instances with missing values in the training or testing phase.

6.8 Deep Learning

As described above, the use of hidden layers in ANN is based on the general belief that complex high-level features can be constructed by combining simpler lower-level features. Typically, the greater the number of hidden layers, the deeper the hierarchy of features learned by the network. This motivates the learning of ANN models with long chains of hidden layers, known as **deep neural networks**. In contrast to "shallow" neural networks that involve only a small number of hidden layers, deep neural networks are able to represent features at multiple levels of abstraction and often require far fewer nodes per layer to achieve generalization performance similar to shallow networks.

Despite the huge potential in learning deep neural networks, it has remained challenging to learn ANN models with a large number of hidden layers using classical approaches. Apart from reasons related to limited computational resources and hardware architectures, there have been a number of algorithmic challenges in learning deep neural networks. First, learning a deep neural network with low training error has been a daunting task because of the saturation of sigmoid activation functions, resulting in slow convergence of gradient descent. This problem becomes even more serious as we move away from the output node to the hidden layers, because of the compounded effects of saturation at multiple layers, known as the **vanishing gradient problem**. Because of this reason, classical ANN models have suffered from slow and ineffective learning, leading to poor training and test performance. Second, the learning of deep neural networks is quite sensitive to the initial values of model parameters, chiefly because of the non-convex nature of the optimization function and the slow convergence of gradient descent. Third, deep neural networks with a large number of hidden layers have high model complexity, making them susceptible to overfitting. Hence, even if a deep neural network has been trained to show low training error, it can still suffer from poor generalization performance.

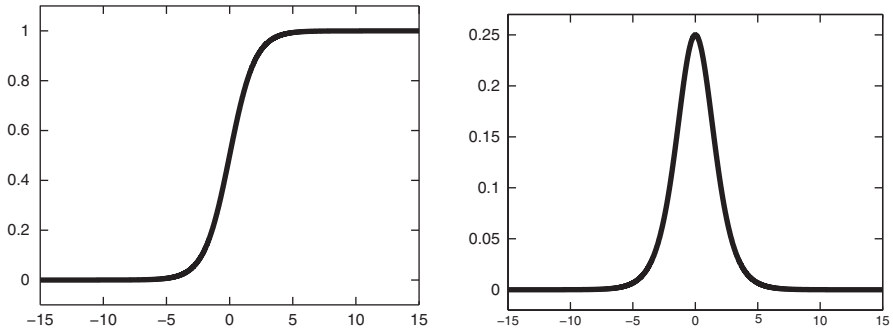
These challenges have deterred progress in building deep neural networks for several decades and it is only recently that we have started to unlock their immense potential with the help of a number of advances being made in the area of deep learning. Although some of these advances have been around for some time, they have only gained mainstream attention in the last decade, with deep neural networks continually beating records in various competitions and solving problems that were too difficult for other classification approaches.

There are two factors that have played a major role in the emergence of deep learning techniques. First, the availability of larger labeled data sets, e.g., the ImageNet data set contains more than 10 million labeled images, has made it possible to learn more complex ANN models than ever before, without falling easily into the traps of model overfitting. Second, advances in computational abilities and hardware infrastructures, such as the use of graphical processing units (GPU) for distributed computing, have greatly helped in experimenting with deep neural networks with larger architectures that would not have been feasible with traditional resources.

In addition to the previous two factors, there have been a number of algorithmic advancements to overcome the challenges faced by classical methods in learning deep neural networks. Some examples include the use of more responsive combinations of loss functions and activation functions, better initialization of model parameters, novel regularization techniques, more agile architecture designs, and better techniques for model learning and hyper-parameter selection. In the following, we describe some of the deep learning advances made to address the challenges in learning deep neural networks. Further details on recent developments in deep learning can be obtained from the Bibliographic Notes.

6.8.1 Using Synergistic Loss Functions

One of the major realizations leading to deep learning has been the importance of choosing appropriate combinations of activation and loss functions. Classical ANN models commonly made use of the sigmoid activation function at the output layer, because of its ability to produce real-valued outputs between 0 and 1, which was combined with a squared loss objective to perform gradient descent. It was soon noticed that this particular combination of activation and loss function resulted in the saturation of output activation values, which can be described as follows.



(a) $\sigma(z)$. (b) $\partial\sigma(z)/\partial z$.

Figure 6.27. Plots of sigmoid function and its derivative.

Saturation of Outputs

Although the sigmoid has been widely-used as an activation function, it easily saturates at high and low values of inputs that are far away from 0. Observe from Figure 6.27(a) that $\sigma(z)$ shows variance in its values only when z is close to 0. For this reason, $\partial\sigma(z)/\partial z$ is non-zero for only a small range of z around 0, as shown in Figure 6.27(b). Since $\partial\sigma(z)/\partial z$ is one of the components in the gradient of loss (see Equation 6.55), we get a diminishing gradient value when the activation values are far from 0.

To illustrate the effect of saturation on the learning of model parameters at the output node, consider the partial derivative of loss with respect to the weight w_j^L at the output node. Using the squared loss function, we can write this as

$$\frac{\partial \text{ Loss}}{\partial w_j^L} = 2(a^L - y) \times \sigma(z^L)(1 - \sigma(z^L)) \times a_j^{L-1}. \tag{6.60}$$

In the previous equation, notice that when z^L is highly negative, $\sigma(z^L)$ (and hence the gradient) is close to 0. On the other hand, when z^L is highly positive, $(1 - \sigma(z^L))$ becomes close to 0, nullifying the value of the gradient. Hence, irrespective of whether the prediction a^L matches the true label y or not, the gradient of the loss with respect to the weights is close to 0 whenever z^L is highly positive or negative. This causes an unnecessarily slow convergence of the model parameters of the ANN model, often resulting in poor learning.

Note that it is the combination of the squared loss function and the sigmoid activation function at the output node that together results in diminishing gradients (and thus poor learning) upon saturation of outputs. It is thus

important to choose a synergistic combination of loss function and activation function that does not suffer from the saturation of outputs.

Cross entropy loss function

The cross entropy loss function, which was described in the context of logistic regression in Section 6.6.2, can significantly avoid the problem of saturating outputs when used in combination with the sigmoid activation function. The cross entropy loss function of a real-valued prediction $\hat{y} \in (0, 1)$ on a data instance with binary label $y \in \{0, 1\}$ can be defined as

$$Loss\ (y,\ \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}), \tag{6.61}$$

where log represents the natural logarithm (to base e) and $0 \log(0) = 0$ for convenience. The cross entropy function has foundations in information theory and measures the amount of disagreement between y and $\hat{y}$. The partial derivative of this loss function with respect to $\hat{y} = a^L$ can be given as

$$\begin{aligned} \delta^L = \frac{\partial\ Loss}{\partial a^L} &= \frac{-y}{a^L} + \frac{(1-y)}{(1-a^L)}. \\ &= \frac{(a^L - y)}{a^L(1-a^L)}. \end{aligned} \tag{6.62}$$

Using this value of δ^L in Equation 6.56, we can obtain the partial derivative of the loss with respect to the weight w_j^l at the output node as

$$\begin{aligned} \frac{\partial\ \text{Loss}}{\partial w_j^L} &= \frac{(a^L - y)}{a^L(1-a^L)} \times a^L(1-a^L) \times a_j^{L-1}. \\ &= (a^L - y) \times a_j^{L-1}. \end{aligned} \tag{6.63}$$

Notice the simplicity of the previous formula using the cross entropy loss function. The partial derivatives of the loss with respect to the weights at the output node depend only on the difference between the prediction a^L and the true label y. In contrast to Equation 6.60, it does not involve terms such as $\sigma(z^L)(1 - \sigma(z^L))$ that can be impacted by saturation of z^L. Hence, the gradients are high whenever $(a^L - y)$ is large, promoting effective learning of the model parameters at the output node. This has been a major breakthrough in the learning of modern ANN models and it is now a common practice to use the cross entropy loss function with sigmoid activations at the output node.

6.8.2 Using Responsive Activation Functions

Even though the cross entropy loss function helps in overcoming the problem of saturating outputs, it still does not solve the problem of saturation at hidden layers, arising due to the use of sigmoid activation functions at hidden nodes. In fact, the effect of saturation on the learning of model parameters is even more aggravated at hidden layers, a problem known as the vanishing gradient problem. In the following, we describe the vanishing gradient problem and the use of a more responsive activation function, called the **rectified linear output unit (ReLU)**, to overcome this problem.

Vanishing Gradient Problem

The impact of saturating activation values on the learning of model parameters increases at deeper hidden layers that are farther away from the output node. Even if the activation in the output layer does not saturate, the repeated multiplications performed as we backpropagate the gradients from the output layer to the hidden layers may lead to decreasing gradients in the hidden layers. This is called the vanishing gradient problem, which has been one of the major hindrances in learning deep neural networks.

To illustrate the vanishing gradient problem, consider an ANN model that consists of a single node at every hidden layer of the network, as shown in Figure 6.28. This simplified architecture involves a single chain of hidden nodes where a single weighted link w^l connects the node at layer $l-1$ to the node at layer l. Using Equations 6.56 and 6.59, we can represent the partial derivative of the loss with respect to w^l as

$$\begin{aligned} \frac{\partial \text{ Loss}}{\partial w^l} &= \delta^l \times a^l(1-a^l) \times a^{l-1}, \\ \text{where } \delta^l &= 2(a^L - y) \times \prod_{r=l}^{L-1} (a^{r+1}(1-a^{r+1}) \times w^{r+1}). \end{aligned} \tag{6.64}$$

Notice that if any of the linear predictors z^{r+1} saturates at subsequent layers, then the term $a^{r+1}(1-a^{r+1})$ becomes close to 0, thus diminishing the overall gradient. The saturation of activations thus gets compounded and has

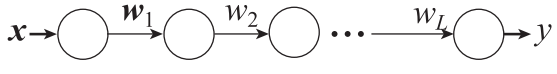


Figure 6.28. An example of an ANN model with only one node at every hidden layer.

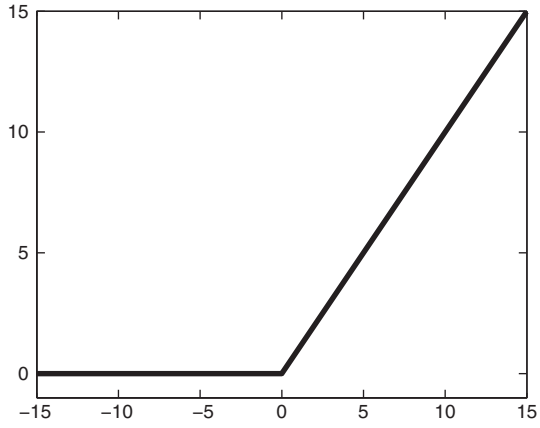


Figure 6.29. Plot of the rectified linear unit (ReLU) activation function.

multiplicative effects on the gradients at hidden layers, making them highly unstable and thus, unsuitable for use with gradient descent. Even though the previous discussion only pertains to the simplified architecture involving a single chain of hidden nodes, a similar argument can be made for any generic ANN architecture involving multiple chains of hidden nodes. Note that the vanishing gradient problem primarily arises because of the use of sigmoid activation function at hidden nodes, which is known to easily saturate especially after repeated multiplications.

Rectified Linear Units (ReLU)

To overcome the vanishing gradient problem, it is important to use an activation function $f(z)$ at the hidden nodes that provides a stable and significant value of the gradient whenever a hidden node is active, i.e., $z > 0$. This is achieved by using rectified linear units (ReLU) as activation functions at hidden nodes, which can be defined as

$$a = f(z) \quad = \quad \begin{cases} z, & \text{if } z > 0. \\ 0, & \text{otherwise.} \end{cases} \tag{6.65}$$

The idea of ReLU has been inspired from biological neurons, which are either in an inactive state ($f(z) = 0$) or show an activation value proportional to the input. Figure 6.29 shows a plot of the ReLU function. We can see that it is linear with respect to z when $z > 0$. Hence, the gradient of the activation

value with respect to z can be written as

$$\frac{\partial a}{\partial z} = \begin{cases} 1, & \text{if } z > 0. \\ 0, & \text{if } z < 0. \end{cases} \tag{6.66}$$

Although $f(z)$ is not differentiable at 0, it is common practice to use $\partial a/\partial z = 0$ when $z = 0$. Since the gradient of the ReLU activation function is equal to 1 whenever $z > 0$, it avoids the problem of saturation at hidden nodes, even after repeated multiplications. Using ReLU, the partial derivatives of the loss with respect to the weight and bias parameters can be given by

$$\frac{\partial \text{ Loss}}{\partial w_{ij}^l} = \delta_i^l \times I(z_i^l) \times a_j^{l-1}, \tag{6.67}$$

$$\frac{\partial \text{ Loss}}{\partial b_i^l} = \delta_i^l \times I(z_i^l), \tag{6.68}$$

$$\text{where } \delta_i^l = \sum_{i=1}^{n} (\delta_i^{l+1} \times I(z_i^{l+1}) \times w_{ij}^{l+1}),$$

$$\text{and } I(z) = \begin{cases} 1, & \text{if } z > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Notice that ReLU shows a linear behavior in the activation values whenever a node is active, as compared to the nonlinear properties of the sigmoid function. This linearity promotes better flows of gradients during backpropagation, and thus simplifies the learning of ANN model parameters. The ReLU is also highly responsive at large values of z away from 0, as opposed to the sigmoid activation function, making it more suitable for gradient descent. These differences give ReLU a major advantage over the sigmoid function. Indeed, ReLU is used as the preferred choice of activation function at hidden layers in most modern ANN models.

6.8.3 Regularization

A major challenge in learning deep neural networks is the high model complexity of ANN models, which grows with the addition of hidden layers in the network. This can become a serious concern, especially when the training set is small, due to the phenomena of model overfitting. To overcome this challenge, it is important to use techniques that can help in reducing

the complexity of the learned model, known as **regularization techniques**. Classical approaches for learning ANN models did not have an effective way to promote regularization of the learned model parameters. Hence, they had often been sidelined by other classification methods, such as support vector machines (SVM), which have in-built regularization mechanisms. (SVMs will be discussed in more detail in Section 6.9).

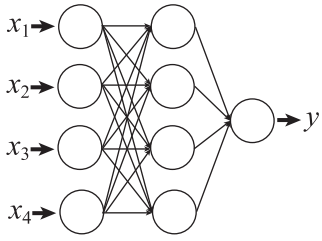
One of the major advancements in deep learning has been the development of novel regularization techniques for ANN models that are able to offer significant improvements in generalization performance. In the following, we discuss one of the regularization techniques for ANN, known as the **dropout** method, that have gained a lot of attention in several applications.

Dropout

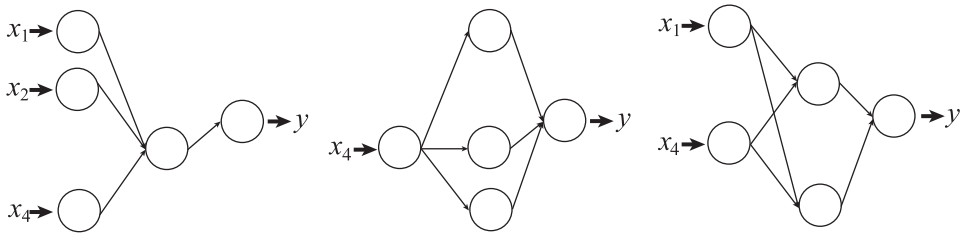
The main objective of dropout is to avoid the learning of spurious features at hidden nodes, occurring due to model overfitting. It uses the basic intuition that spurious features often "co-adapt" themselves such that they show good training performance only when used in highly selective combinations. On the other hand, relevant features can be used in a diversity of feature combinations and hence are quite resilient to the removal or modification of other features. The dropout method uses this intuition to break complex "co-adaptations" in the learned features by randomly dropping input and hidden nodes in the network during training.

Dropout belongs to a family of regularization techniques that uses the criteria of resilience to random perturbations as a measure of the robustness (and hence, simplicity) of a model. For example, one approach to regularization is to inject noise in the input attributes of the training set and learn a model with the noisy training instances. If a feature learned from the training data is indeed generalizable, it should not be affected by the addition of noise. Dropout can be viewed as a similar regularization approach that perturbs the information content of the training set not only at the level of attributes but also at multiple levels of abstractions, by dropping input and hidden nodes.

The dropout method draws inspiration from the biological process of gene swapping in sexual reproduction, where half of the genes from both parents are combined together to create the genes of the offspring. This favors the selection of parent genes that are not only useful but can also inter-mingle with diverse combinations of genes coming from the other parent. On the other hand, co-adapted genes that function only in highly selective combinations are soon eliminated in the process of evolution. This idea is used in the dropout



(a) Original network.



(b) Sub-networks.

Figure 6.30. Examples of sub-networks generated in the dropout method using $\gamma = 0.5$.

method for eliminating spurious co-adapted features. A simplified description of the dropout method is provided in the rest of this section.

Let $(\mathbf{w}^k, \mathbf{b}^k)$ represent the model parameters of the ANN model at the k^{th} iteration of the gradient descent method. At every iteration, we randomly select a fraction γ of input and hidden nodes to be dropped from the network, where $\gamma \in (0, 1)$ is a hyper-parameter that is typically chosen to be 0.5. The weighted links and bias terms involving the dropped nodes are then eliminated, resulting in a "thinned" sub-network of smaller size. The model parameters of the sub-network $(\mathbf{w}_s^k, \mathbf{b}_s^k)$ are then updated by computing activation values and performing backpropagation on this smaller sub-network. These updated values are then added back in the original network to obtain the updated model parameters, $(\mathbf{w}^{k+1}, \mathbf{b}^{k+1})$, to be used in the next iteration.

Figure 6.30 shows some examples of sub-networks that can be generated at different iterations of the dropout method, by randomly dropping input and hidden nodes. Since every sub-network has a different architecture, it is difficult to learn complex co-adaptations in the features that can result in overfitting. Instead, the features at the hidden nodes are learned to be more

agile to random modifications in the network structure, thus improving their generalization ability. The model parameters are updated using a different random sub-network at every iteration, till the gradient descent method converges.

Let $(\mathbf{w}^{k_{max}}, \mathbf{b}^{k_{max}})$ denote the model parameters at the last iteration k_{max} of the gradient descent method. These parameters are finally scaled down by a factor of $(1 - \gamma)$, to produce the weights and bias terms of the final ANN model, as follows:

$$(\mathbf{w}^*, \mathbf{b}^*) = ((1 - \gamma) \times \mathbf{w}^{k_{max}}, (1 - \gamma) \times \mathbf{b}^{k_{max}})$$

We can now use the complete neural network with model parameters $(\mathbf{w}^*, \mathbf{b}^*)$ for testing. The dropout method has been shown to provide significant improvements in the generalization performance of ANN models in a number of applications. It is computationally cheap and can be applied in combination with any of the other deep learning techniques. It also has a number of similarities with a widely-used ensemble learning method known as **bagging**, which learns multiple models using random subsets of the training set, and then uses the *average output* of all the models to make predictions. (Bagging will be presented in more detail later in Section 6.10.4). In a similar vein, it can be shown that the predictions of the final network learned using dropout approximates the average output of all possible 2^n sub-networks that can be formed using n nodes. This is one of the reasons behind the superior regularization abilities of dropout.

6.8.4 Initialization of Model Parameters

Because of the non-convex nature of the loss function used by ANN models, it is possible to get stuck in locally optimal but globally inferior solutions. Hence, the initial choice of model parameter values plays a significant role in the learning of ANN by gradient descent. The impact of poor initialization is even more aggravated when the model is complex, the network architecture is deep, or the classification task is difficult. In such cases, it is often advisable to first learn a simpler model for the problem, e.g., using a single hidden layer, and then incrementally increase the complexity of the model, e.g., by adding more hidden layers. An alternate approach is to train the model for a simpler task and then use the learned model parameters as initial parameter choices in the learning of the original task. The process of initializing ANN model parameters before the actual training process is known as **pretraining**.

Pretraining helps in initializing the model to a suitable region in the parameter space that would otherwise be inaccessible by random initialization.

Pretraining also reduces the variance in the model parameters by fixing the starting point of gradient descent, thus reducing the chances of overfitting due to multiple comparisons. The models learned by pretraining are thus more consistent and provide better generalization performance.

Supervised Pretraining

A common approach for pretraining is to incrementally train the ANN model in a layer-wise manner, by adding one hidden layer at a time. This approach, known as **supervised pretraining**, ensures that the parameters learned at every layer are obtained by solving a simpler problem, rather than learning all model parameters together. These parameter values thus provide a good choice for initializing the ANN model. The approach for supervised pretraining can be briefly described as follows.

We start the supervised pretraining process by considering a reduced ANN model with only a single hidden layer. By applying gradient descent on this simple model, we are able to learn the model parameters of the first hidden layer. At the next run, we add another hidden layer to the model and apply gradient descent to learn the parameters of the newly added hidden layer, while keeping the parameters of the first layer fixed. This procedure is recursively applied such that while learning the parameters of the l^{th} hidden layer, we consider a reduced model with only l hidden layers, whose first $(l-1)$ hidden layers are not updated on the l^{th} run but are instead fixed using pretrained values from previous runs. In this way, we are able to learn the model parameters of all $(L-1)$ hidden layers. These pretrained values are used to initialize the hidden layers of the final ANN model, which is fine-tuned by applying a final round of gradient descent over all the layers.

Unsupervised Pretraining

Supervised pretraining provides a powerful way to initialize model parameters, by gradually growing the model complexity from shallower to deeper networks. However, supervised pretraining requires a sufficient number of labeled training instances for effective initialization of the ANN model. An alternate pretraining approach is **unsupervised pretraining**, which initializes model parameters by using unlabeled instances that are often abundantly available. The basic idea of unsupervised pretraining is to initialize the ANN model in such a way that the learned features capture the latent structure in the unlabeled data.

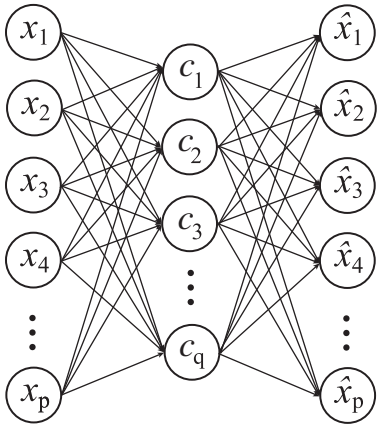


Figure 6.31. The basic architecture of a single-layer autoencoder.

Unsupervised pretraining relies on the assumption that learning the distribution of the input data can indirectly help in learning the classification model. It is most helpful when the number of labeled examples is small and the features for the supervised problem bear resemblance to the factors generating the input data. Unsupervised pretraining can be viewed as a different form of regularization, where the focus is not explicitly toward finding simpler features but instead toward finding features that can best explain the input data. Historically, unsupervised pretraining has played an important role in reviving the area of deep learning, by making it possible to train any generic deep neural network without requiring specialized architectures.

Use of Autoencoders

One simple and commonly used approach for unsupervised pretraining is to use an unsupervised ANN model known as an **autoencoder**. The basic architecture of an autoencoder is shown in Figure 6.31. An autoencoder attempts to learn a *reconstruction* of the input data by mapping the attributes $\mathbf{x}$ to latent features $\mathbf{c}$, and then re-projecting $\mathbf{c}$ back to the original attribute space to create the reconstruction $\hat{\mathbf{x}}$. The latent features are represented using a hidden layer of nodes, while the input and output layers represent the attributes and contain the same number of nodes. During training, the goal is to learn an autoencoder model that provides the lowest reconstruction error, $RE(\mathbf{x}, \hat{\mathbf{x}})$, on all input data instances. A typical choice of the reconstruction error is the squared loss function:

$$RE(\mathbf{x}, \hat{\mathbf{x}}) = ||\mathbf{x} - \hat{\mathbf{x}}||^2.$$

The model parameters of the autoencoder can be learned by using a similar gradient descent method as the one used for learning supervised ANN models for classification. The key difference is the use of the reconstruction error on all training instances as the training loss. Autoencoders that have multiple layers of hidden layers are known as **stacked autoencoders**.

Autoencoders are able to capture complex representations of the input data by the use of hidden nodes. However, if the number of hidden nodes is large, it is possible for an autoencoder to learn the identity relationship, where the input $\mathbf{x}$ is just copied and returned as the output $\hat{\mathbf{x}}$, resulting in a trivial solution. For example, if we use as many hidden nodes as the number of attributes, then it is possible for every hidden node to copy an attribute and simply pass it along to an output node, without extracting any useful information. To avoid this problem, it is common practice to keep the number of hidden nodes smaller than the number of input attributes. This forces the autoencoder to learn a compact and useful *encoding* of the input data, similar to a dimensionality reduction technique. An alternate approach is to corrupt the input instances by adding random noise, and then learn the autoencoder to reconstruct the original instance from the noisy input. This approach is known as the **denoising autoencoder**, which offers strong regularization capabilities and is often used to learn complex features even in the presence of a large number of hidden nodes.

To use an autoencoder for unsupervised pretraining, we can follow a similar layer-wise approach like supervised pretraining. In particular, to pretrain the model parameters of the l^{th} hidden layer, we can construct a reduced ANN model with only l hidden layers and an output layer containing the same number of nodes as the attributes and is used for reconstruction. The parameters of the l^{th} hidden layer of this network are then learned using a gradient descent method to minimize the reconstruction error. The use of unlabeled data can be viewed as providing *hints* to the learning of parameters at every layer that aid in generalization. The final model parameters of the ANN model are then learned by applying gradient descent over all the layers, using the initial values of parameters obtained from pretraining.

Hybrid Pretraining

Unsupervised pretraining can also be combined with supervised pretraining by using two output layers at every run of pretraining, one for reconstruction and the other for supervised classification. The parameters of the l^{th} hidden layer are then learned by jointly minimizing the losses on both output layers, usually weighted by a trade-off hyper-parameter α. Such a combined approach often

shows better generalization performance than either of the approaches, since it provides a way to balance between the competing objectives of representing the input data and improving classification performance.

6.8.5 Characteristics of Deep Learning

Apart from the basic characteristics of ANN discussed in Section 6.7.3, the use of deep learning techniques provides the following additional characteristics:

1. An ANN model trained for some task can be easily re-used for a different task that involves the same attributes, by using pretraining strategies. For example, we can use the learned parameters of the original task as initial parameter choices for the target task. In this way, ANN promotes *re-usability* of learning, which can be quite useful when the target application has a smaller number of labeled training instances.

2. Deep learning techniques for regularization, such as the dropout method, help in reducing the model complexity of ANN and thus promoting good generalization performance. The use of regularization techniques is especially useful in high-dimensional settings, where the number of training labels is small but the classification problem is inherently difficult.

3. The use of an autoencoder for pretraining can help eliminate irrelevant attributes that are not related to other attributes. Further, it can help reduce the impact of redundant attributes by representing them as copies of the same attribute.

4. Although the learning of an ANN model can succumb to finding inferior and locally optimal solutions, there are a number of deep learning techniques that have been proposed to ensure adequate learning of an ANN. Apart from the methods discussed in this section, some other techniques involve novel architecture designs such as skip connections between the output layer and lower layers, which aids the easy flow of gradients during backpropagation.

5. A number of specialized ANN architectures have been designed to handle a variety of input data sets. Some examples include **convolutional neural networks** (CNN) for two-dimensional gridded objects such as images, and **recurrent neural network** (RNN) for sequences. While CNNs have been extensively used in the area of computer vision, RNNs have found applications in processing speech and language.