Chapter 1

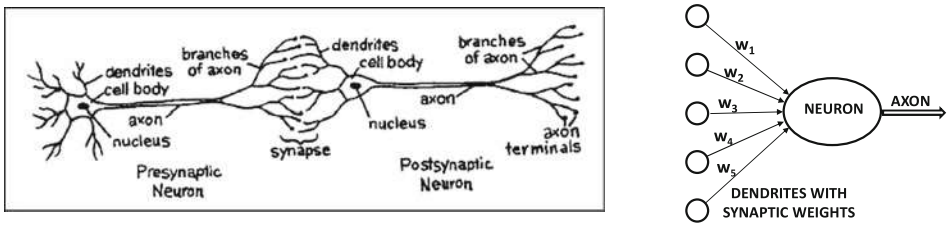
An Introduction to Neural Networks

"Thou shalt not make a machine to counterfeit a human mind."—Frank Herbert

1.1 Introduction

Artificial neural networks are popular machine learning techniques that simulate the mechanism of learning in biological organisms. The human nervous system contains cells, which are referred to as *neurons*. The neurons are connected to one another with the use of *axons* and *dendrites*, and the connecting regions between axons and dendrites are referred to as *synapses*. These connections are illustrated in Figure 1.1(a). The strengths of synaptic connections often change in response to external stimuli. This change is how learning takes place in living organisms.

This biological mechanism is simulated in *artificial* neural networks, which contain computation units that are referred to as neurons. Throughout this book, we will use the term "neural networks" to refer to artificial neural networks rather than biological ones. The computational units are connected to one another through weights, which serve the same



(a) Biological neural network (b) Artificial neural network

Figure 1.1: The synaptic connections between neurons. The image in (a) is from "*The Brain: Understanding Neurobiology Through the Study of Addiction* [598]." Copyright ©2000 by BSCS & Videodiscovery. All rights reserved. Used with permission.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_1

role as the strengths of synaptic connections in biological organisms. Each input to a neuron is scaled with a weight, which affects the function computed at that unit. This architecture is illustrated in Figure 1.1(b). An artificial neural network computes a function of the inputs by propagating the computed values from the input neurons to the output neuron(s) and using the weights as intermediate parameters. Learning occurs by changing the weights connecting the neurons. Just as external stimuli are needed for learning in biological organisms, the external stimulus in artificial neural networks is provided by the training data containing examples of input-output pairs of the function to be learned. For example, the training data might contain pixel representations of images (input) and their annotated labels (e.g., carrot, banana) as the output. These training data pairs are fed into the neural network by using the input representations to make predictions about the output labels. The training data provides feedback to the correctness of the weights in the neural network depending on how well the predicted output (e.g., probability of carrot) for a particular input matches the annotated output label in the training data. One can view the errors made by the neural network in the computation of a function as a kind of unpleasant feedback in a biological organism, leading to an adjustment in the synaptic strengths. Similarly, the weights between neurons are adjusted in a neural network in response to prediction errors. The goal of changing the weights is to modify the computed function to make the predictions more correct in future iterations. Therefore, the weights are changed carefully in a mathematically justified way so as to reduce the error in computation on that example. By successively adjusting the weights between neurons over many input-output pairs, the function computed by the neural network is refined over time so that it provides more accurate predictions. Therefore, if the neural network is trained with many different images of bananas, it will eventually be able to properly recognize a banana in an image it has not seen before. This ability to accurately compute functions of unseen inputs by training over a finite set of input-output pairs is referred to as *model generalization*. The primary usefulness of all machine learning models is gained from their ability to generalize their learning from seen training data to unseen examples.

The biological comparison is often criticized as a very poor caricature of the workings of the human brain; nevertheless, the principles of neuroscience have often been useful in designing neural network architectures. A different view is that neural networks are built as higher-level abstractions of the classical models that are commonly used in machine learning. In fact, the most basic units of computation in the neural network are inspired by traditional machine learning algorithms like *least-squares regression* and *logistic regression*. Neural networks gain their power by putting together many such basic units, and learning the weights of the different units jointly in order to minimize the prediction error. From this point of view, a neural network can be viewed as a *computational graph* of elementary units in which greater power is gained by connecting them in particular ways. When a neural network is used in its most basic form, without hooking together multiple units, the learning algorithms often reduce to classical machine learning models (see Chapter 2). The real power of a neural model over classical methods is unleashed when these elementary computational units are combined, and the weights of the elementary models are trained using their dependencies on one another. By combining multiple units, one is increasing the power of the model to learn more complicated functions of the data than are inherent in the elementary models of basic machine learning. The way in which these units are combined also plays a role in the power of the architecture, and requires some understanding and insight from the analyst. Furthermore, sufficient training data is also required in order to learn the larger number of weights in these expanded computational graphs.

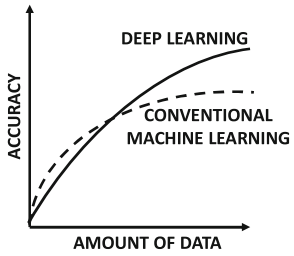


Figure 1.2: An illustrative comparison of the accuracy of a typical machine learning algorithm with that of a large neural network. Deep learners become more attractive than conventional methods primarily when sufficient data/computational power is available. Recent years have seen an increase in data availability and computational power, which has led to a "Cambrian explosion" in deep learning technology.

1.1.1 Humans Versus Computers: Stretching the Limits of Artificial Intelligence

Humans and computers are inherently suited to different types of tasks. For example, computing the cube root of a large number is very easy for a computer, but it is extremely difficult for humans. On the other hand, a task such as recognizing the objects in an image is a simple matter for a human, but has traditionally been very difficult for an automated learning algorithm. It is only in recent years that deep learning has shown an accuracy on some of these tasks that exceeds that of a human. In fact, the recent results by deep learning algorithms that surpass human performance [184] in (some narrow tasks on) image recognition would not have been considered likely by most computer vision experts as recently as 10 years ago.

Many deep learning architectures that have shown such extraordinary performance are not created by indiscriminately connecting computational units. The superior performance of *deep* neural networks mirrors the fact that biological neural networks gain much of their power from depth as well. Furthermore, biological networks are connected in ways we do not fully understand. In the few cases that the biological structure is understood at some level, significant breakthroughs have been achieved by designing artificial neural networks along those lines. A classical example of this type of architecture is the use of the *convolutional neural network* for image recognition. This architecture was inspired by Hubel and Wiesel's experiments [212] in 1959 on the organization of the neurons in the cat's visual cortex. The precursor to the convolutional neural network was the *neocognitron* [127], which was directly based on these results.

The human neuronal connection structure has evolved over millions of years to optimize survival-driven performance; survival is closely related to our ability to merge sensation and intuition in a way that is currently not possible with machines. Biological neuroscience [232] is a field that is still very much in its infancy, and only a limited amount is known about how the brain truly works. Therefore, it is fair to suggest that the biologically inspired success of convolutional neural networks might be replicated in other settings, as we learn more about how the human brain works [176]. A key advantage of neural networks over traditional machine learning is that the former provides a higher-level abstraction of expressing semantic insights about data domains by architectural design choices in the computational graph. The second advantage is that neural networks provide a simple way to adjust the

complexity of a model by adding or removing neurons from the architecture according to the availability of training data or computational power. A large part of the recent success of neural networks is explained by the fact that the increased data availability and computational power of modern computers has outgrown the limits of traditional machine learning algorithms, which fail to take full advantage of what is now possible. This situation is illustrated in Figure 1.2. The performance of traditional machine learning remains better at times for smaller data sets because of more choices, greater ease of model interpretation, and the tendency to hand-craft interpretable features that incorporate domain-specific insights. With limited data, the best of a very wide diversity of models in machine learning will usually perform better than a single class of models (like neural networks). This is one reason why the potential of neural networks was not realized in the early years.

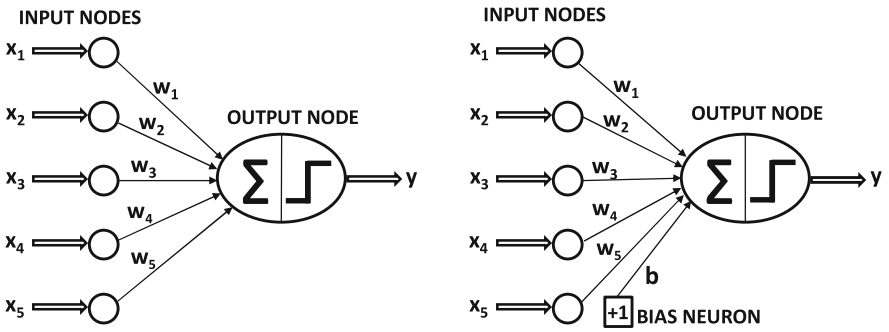
The "big data" era has been enabled by the advances in data collection technology; virtually everything we do today, including purchasing an item, using the phone, or clicking on a site, is collected and stored somewhere. Furthermore, the development of powerful Graphics Processor Units (GPUs) has enabled increasingly efficient processing on such large data sets. These advances largely explain the recent success of deep learning using algorithms that are only slightly adjusted from the versions that were available two decades back. Furthermore, these recent adjustments to the algorithms have been enabled by increased speed of computation, because reduced run-times enable efficient testing (and subsequent algorithmic adjustment). If it requires a month to test an algorithm, at most twelve variations can be tested in an year on a single hardware platform. This situation has historically constrained the intensive experimentation required for tweaking neural-network learning algorithms. The rapid advances associated with the three pillars of improved data, computation, and experimentation have resulted in an increasingly optimistic outlook about the future of deep learning. By the end of this century, it is expected that computers will have the power to train neural networks with as many neurons as the human brain. Although it is hard to predict what the true capabilities of artificial intelligence will be by then, our experience with computer vision should prepare us to expect the unexpected.

Chapter Organization

This chapter is organized as follows. The next section introduces single-layer and multi-layer networks. The different types of activation functions, output nodes, and loss functions are discussed. The backpropagation algorithm is introduced in Section 1.3. Practical issues in neural network training are discussed in Section 1.4. Some key points on how neural networks gain their power with specific choices of activation functions are discussed in Section 1.5. The common architectures used in neural network design are discussed in Section 1.6. Advanced topics in deep learning are discussed in Section 1.7. Some notable benchmarks used by the deep learning community are discussed in Section 1.8. A summary is provided in Section 1.9.

1.2 The Basic Architecture of Neural Networks

In this section, we will introduce single-layer and multi-layer neural networks. In the single-layer network, a set of inputs is directly mapped to an output by using a generalized variation of a linear function. This simple instantiation of a neural network is also referred to as the *perceptron*. In multi-layer neural networks, the neurons are arranged in layered fashion, in which the input and output layers are separated by a group of hidden layers. This layer-wise architecture of the neural network is also referred to as a *feed-forward network*. This section will discuss both single-layer and multi-layer networks.



(a) Perceptron without bias (b) Perceptron with bias

Figure 1.3: The basic architecture of the perceptron

1.2.1 Single Computational Layer: The Perceptron

The simplest neural network is referred to as the perceptron. This neural network contains a single input layer and an output node. The basic architecture of the perceptron is shown in Figure 1.3(a). Consider a situation where each training instance is of the form $(\overline{X}, y)$, where each $\overline{X} = [x_1, \ldots x_d]$ contains d feature variables, and $y \in \{-1, +1\}$ contains the *observed value* of the binary class variable. By "observed value" we refer to the fact that it is given to us as a part of the training data, and our goal is to predict the class variable for cases in which it is not observed. For example, in a credit-card fraud detection application, the features might represent various properties of a set of credit card transactions (e.g., amount and frequency of transactions), and the class variable might represent whether or not this set of transactions is fraudulent. Clearly, in this type of application, one would have historical cases in which the class variable is observed, and other (current) cases in which the class variable has not yet been observed but needs to be predicted.

The input layer contains d nodes that transmit the d features $\overline{X} = [x_1 \ldots x_d]$ with edges of weight $\overline{W} = [w_1 \ldots w_d]$ to an output node. The input layer does not perform any computation in its own right. The linear function $\overline{W} \cdot \overline{X} = \sum_{i=1}^{d} w_i x_i$ is computed at the output node. Subsequently, the sign of this real value is used in order to predict the dependent variable of $\overline{X}$. Therefore, the prediction $\hat{y}$ is computed as follows:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X}\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j\} \tag{1.1}$$

The sign function maps a real value to either $+1$ or -1, which is appropriate for binary classification. Note the circumflex on top of the variable y to indicate that it is a predicted value rather than an observed value. The error of the prediction is therefore $E(\overline{X}) = y - \hat{y}$, which is one of the values drawn from the set $\{-2, 0, +2\}$. In cases where the error value $E(\overline{X})$ is nonzero, the weights in the neural network need to be updated in the (negative) direction of the error gradient. As we will see later, this process is similar to that used in various types of linear models in machine learning. In spite of the similarity of the perceptron with respect to traditional machine learning models, its interpretation as a computational unit is very useful because it allows us to put together multiple units in order to create far more powerful models than are available in traditional machine learning.

The architecture of the perceptron is shown in Figure 1.3(a), in which a single input layer transmits the features to the output node. The edges from the input to the output contain the weights $w_1 \ldots w_d$ with which the features are multiplied and added at the output node. Subsequently, the sign function is applied in order to convert the aggregated value into a class label. The sign function serves the role of an *activation function*. Different choices of activation functions can be used to simulate different types of models used in machine learning, like *least-squares regression with numeric targets*, the *support vector machine*, or a *logistic regression classifier*. Most of the basic machine learning models can be easily represented as simple neural network architectures. It is a useful exercise to model traditional machine learning techniques as neural architectures, because it provides a clearer picture of how deep learning generalizes traditional machine learning. This point of view is explored in detail in Chapter 2. It is noteworthy that the perceptron contains two layers, although the input layer does not perform any computation and only transmits the feature values. The input layer is not included in the count of the number of layers in a neural network. Since the perceptron contains a single *computational* layer, it is considered a single-layer network.

In many settings, there is an invariant part of the prediction, which is referred to as the *bias*. For example, consider a setting in which the feature variables are mean centered, but the mean of the binary class prediction from $\{-1, +1\}$ is not 0. This will tend to occur in situations in which the binary class distribution is highly imbalanced. In such a case, the aforementioned approach is not sufficient for prediction. We need to incorporate an additional bias variable b that captures this invariant part of the prediction:

$$\hat{y} = \text{sign}\{\overline{W} \cdot \overline{X} + b\} = \text{sign}\{\sum_{j=1}^{d} w_j x_j + b\} \tag{1.2}$$

The bias can be incorporated as the weight of an edge by using a *bias neuron*. This is achieved by adding a neuron that always transmits a value of 1 to the output node. The weight of the edge connecting the bias neuron to the output node provides the bias variable. An example of a bias neuron is shown in Figure 1.3(b). Another approach that works well with single-layer architectures is to use a *feature engineering trick* in which an additional feature is created with a constant value of 1. The coefficient of this feature provides the bias, and one can then work with Equation 1.1. Throughout this book, biases will not be explicitly used (for simplicity in architectural representations) because they can be incorporated with bias neurons. The details of the training algorithms remain the same by simply treating the bias neurons like any other neuron with a fixed activation value of 1. Therefore, the following will work with the predictive assumption of Equation 1.1, which does not explicitly uses biases.

At the time that the perceptron algorithm was proposed by Rosenblatt [405], these optimizations were performed in a heuristic way with actual hardware circuits, and it was not presented in terms of a formal notion of optimization in machine learning (as is common today). However, the goal was always to minimize the error in prediction, even if a formal optimization formulation was not presented. The perceptron algorithm was, therefore, heuristically designed to minimize the number of misclassifications, and convergence proofs were available that provided correctness guarantees of the learning algorithm in simplified settings. Therefore, we can still write the (heuristically motivated) goal of the perceptron algorithm in least-squares form with respect to all training instances in a data set $\mathcal{D}$ con-

taining feature-label pairs:

$$\text{Minimize}_{\overline{W}} L = \sum_{(\overline{X},y)\in\mathcal{D}} (y - \hat{y})^2 = \sum_{(\overline{X},y)\in\mathcal{D}} \left(y - \text{sign}\{\overline{W} \cdot \overline{X}\}\right)^2$$

This type of minimization objective function is also referred to as a *loss function*. As we will see later, almost all neural network learning algorithms are formulated with the use of a loss function. As we will learn in Chapter 2, this loss function looks a lot like least-squares regression. However, the latter is defined for continuous-valued target variables, and the corresponding loss is a smooth and continuous function of the variables. On the other hand, for the least-squares form of the objective function, the sign function is non-differentiable, with step-like jumps at specific points. Furthermore, the sign function takes on constant values over large portions of the domain, and therefore the exact gradient takes on zero values at differentiable points. This results in a staircase-like loss surface, which is not suitable for gradient-descent. The perceptron algorithm (implicitly) uses a smooth approximation of the gradient of this objective function with respect to each example:

$$\nabla L_{\text{smooth}} = \sum_{(\overline{X},y)\in\mathcal{D}} (y - \hat{y})\overline{X} \tag{1.3}$$

Note that the above gradient is not a true gradient of the staircase-like surface of the (heuristic) objective function, which does not provide useful gradients. Therefore, the staircase is smoothed out into a sloping surface defined by the *perceptron criterion*. The properties of the perceptron criterion will be described in Section 1.2.1.1. It is noteworthy that concepts like the "perceptron criterion" were proposed later than the original paper by Rosenblatt [405] in order to explain the heuristic gradient-descent steps. For now, we will assume that the perceptron algorithm optimizes some unknown smooth function with the use of gradient descent.

Although the above objective function is defined over the entire training data, the training algorithm of neural networks works by feeding each input data instance $\overline{X}$ into the network one by one (or in small batches) to create the prediction $\hat{y}$. The weights are then updated, based on the error value $E(\overline{X}) = (y - \hat{y})$. Specifically, when the data point $\overline{X}$ is fed into the network, the weight vector $\overline{W}$ is updated as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha(y - \hat{y})\overline{X} \tag{1.4}$$

The parameter α regulates the learning rate of the neural network. The perceptron algorithm repeatedly cycles through all the training examples in random order and iteratively adjusts the weights until convergence is reached. A single training data point may be cycled through many times. Each such cycle is referred to as an *epoch*. One can also write the gradient-descent update in terms of the error $E(\overline{X}) = (y - \hat{y})$ as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha E(\overline{X})\overline{X} \tag{1.5}$$

The basic perceptron algorithm can be considered a *stochastic gradient-descent* method, which implicitly minimizes the squared error of prediction by performing gradient-descent updates with respect to randomly chosen training points. The assumption is that the neural network cycles through the points in random order during training and changes the weights with the goal of reducing the prediction error on that point. It is easy to see from Equation 1.5 that non-zero updates are made to the weights only when $y \neq \hat{y}$, which occurs only

when errors are made in prediction. In *mini-batch stochastic gradient descent*, the aforementioned updates of Equation 1.5 are implemented over a randomly chosen subset of training points S:

$$\overline{W} \Leftarrow \overline{W} + \alpha \sum_{\overline{X} \in S} E(\overline{X})\overline{X} \tag{1.6}$$

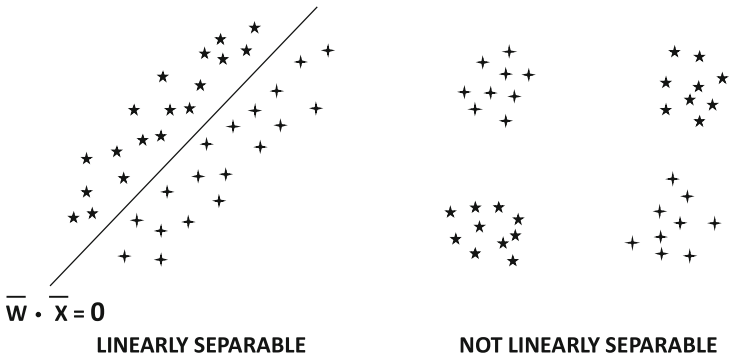


Figure 1.4: Examples of linearly separable and inseparable data in two classes

The advantages of using mini-batch stochastic gradient descent are discussed in Section 3.2.8 of Chapter 3. An interesting quirk of the perceptron is that it is possible to set the learning rate α to 1, because the learning rate only scales the weights.

The type of model proposed in the perceptron is a *linear model*, in which the equation $\overline{W} \cdot \overline{X} = 0$ defines a linear hyperplane. Here, $\overline{W} = (w_1 \ldots w_d)$ is a d-dimensional vector that is normal to the hyperplane. Furthermore, the value of $\overline{W} \cdot \overline{X}$ is positive for values of $\overline{X}$ on one side of the hyperplane, and it is negative for values of $\overline{X}$ on the other side. This type of model performs particularly well when the data is *linearly separable*. Examples of linearly separable and inseparable data are shown in Figure 1.4.

The perceptron algorithm is good at classifying data sets like the one shown on the left-hand side of Figure 1.4, when the data is linearly separable. On the other hand, it tends to perform poorly on data sets like the one shown on the right-hand side of Figure 1.4. This example shows the inherent modeling limitation of a perceptron, which necessitates the use of more complex neural architectures.

Since the original perceptron algorithm was proposed as a heuristic minimization of classification errors, it was particularly important to show that the algorithm converges to reasonable solutions in some special cases. In this context, it was shown [405] that the perceptron algorithm always converges to provide zero error on the training data when the data are linearly separable. However, the perceptron algorithm is not guaranteed to converge in instances where the data are not linearly separable. For reasons discussed in the next section, the perceptron might sometimes arrive at a very poor solution with data that are not linearly separable (in comparison with many other learning algorithms).

1.2.1.1 What Objective Function Is the Perceptron Optimizing?

As discussed earlier in this chapter, the original perceptron paper by Rosenblatt [405] did not formally propose a loss function. In those years, these implementations were achieved using actual hardware circuits. The original *Mark I perceptron* was intended to be a machine rather than an algorithm, and custom-built hardware was used to create it (cf. Figure 1.5).

The general goal was to minimize the number of classification errors with a heuristic update process (in hardware) that changed weights in the "correct" direction whenever errors were made. This heuristic update strongly resembled gradient descent but it was not derived as a gradient-descent method. Gradient descent is defined only for smooth loss functions in algorithmic settings, whereas the hardware-centric approach was designed in a more

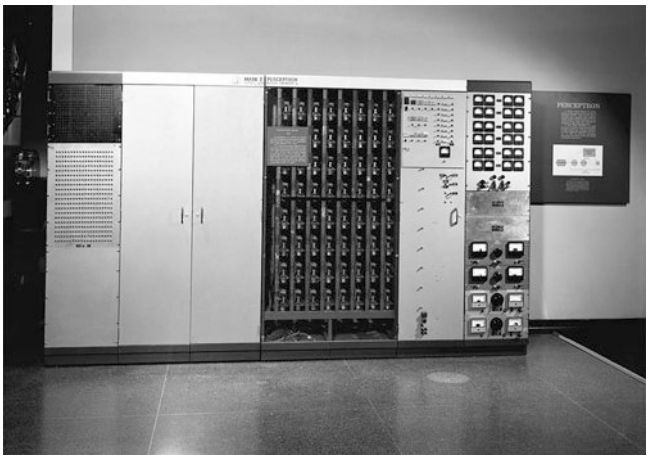


Figure 1.5: The perceptron algorithm was originally implemented using hardware circuits. The image depicts the Mark I perceptron machine built in 1958. (Courtesy: Smithsonian Institute)

heuristic way with *binary outputs*. Many of the binary and circuit-centric principles were inherited from the *McCulloch-Pitts* model [321] of the neuron. Unfortunately, binary signals are not prone to continuous optimization.

Can we find a smooth loss function, whose gradient turns out to be the perceptron update? The number of classification errors in a binary classification problem can be written in the form of a 0/1 loss function for training data point $(\overline{X_i}, y_i)$ as follows:

$$L_i^{(0/1)} = \frac{1}{2}(y_i - \text{sign}\{\overline{W} \cdot \overline{X_i}\})^2 = 1 - y_i \cdot \text{sign}\{\overline{W} \cdot \overline{X_i}\} \tag{1.7}$$

The simplification to the right-hand side of the above objective function is obtained by setting both y_i^2 and $\text{sign}\{\overline{W} \cdot \overline{X_i}\}^2$ to 1, since they are obtained by squaring a value drawn from $\{-1, +1\}$. However, this objective function is not differentiable, because it has a staircase-like shape, especially when it is added over multiple points. Note that the 0/1 loss above is dominated by the term $-y_i \text{sign}\{\overline{W} \cdot \overline{X_i}\}$, in which the sign function causes most of the problems associated with non-differentiability. Since neural networks are defined by gradient-based optimization, we need to define a smooth objective function that is responsible for the perceptron updates. It can be shown [41] that the updates of the perceptron implicitly optimize the *perceptron criterion*. This objective function is defined by dropping the sign function in the above 0/1 loss and setting negative values to 0 in order to treat all correct predictions in a uniform and lossless way:

$$L_i = \max\{-y_i(\overline{W} \cdot \overline{X_i}), 0\} \tag{1.8}$$

The reader is encouraged to use calculus to verify that the gradient of this smoothed objective function leads to the perceptron update, and the update of the perceptron is essentially

$\overline{W} \Leftarrow \overline{W} - \alpha \nabla_W L_i$. The modified loss function to enable gradient computation of a non-differentiable function is also referred to as a *smoothed surrogate loss function*. Almost all continuous optimization-based learning methods (such as neural networks) with discrete outputs (such as class labels) use some type of smoothed surrogate loss function.

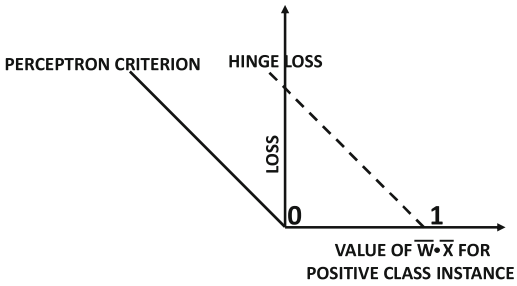


Figure 1.6: Perceptron criterion versus hinge loss

Although the aforementioned perceptron criterion was reverse engineered by working backwards from the perceptron updates, the nature of this loss function exposes some of the weaknesses of the updates in the original algorithm. An interesting observation about the perceptron criterion is that one can set $\overline{W}$ to the zero vector *irrespective of the training data set* in order to obtain the optimal loss value of 0. In spite of this fact, the perceptron updates continue to converge to a clear separator between the two classes in linearly separable cases; after all, a separator between the two classes provides a loss value of 0 as well. However, the behavior for data that are not linearly separable is rather arbitrary, and the resulting solution is sometimes not even a good approximate separator of the classes. The direct sensitivity of the loss to the *magnitude* of the weight vector can dilute the goal of class separation; it is possible for updates to worsen the number of misclassifications significantly while improving the loss. This is an example of how surrogate loss functions might sometimes not fully achieve their intended goals. Because of this fact, the approach is not stable and can yield solutions of widely varying quality.

Several variations of the learning algorithm were therefore proposed for inseparable data, and a natural approach is to always keep track of the best solution in terms of the number of misclassifications [128]. This approach of always keeping the best solution in one's "pocket" is referred to as the *pocket algorithm*. Another highly performing variant incorporates the notion of *margin* in the loss function, which creates an *identical* algorithm to the *linear support vector machine*. For this reason, the linear support vector machine is also referred to as the *perceptron of optimal stability*.

1.2.1.2 Relationship with Support Vector Machines

The perceptron criterion is a shifted version of the *hinge-loss* used in support vector machines (see Chapter 2). The hinge loss looks even more similar to the zero-one loss criterion of Equation 1.7, and is defined as follows:

$$L_i^{svm} = \max\{1 - y_i(\overline{W} \cdot \overline{X_i}), 0\} \tag{1.9}$$

Note that the perceptron does not keep the constant term of 1 on the right-hand side of Equation 1.7, whereas the hinge loss keeps this constant within the maximization function. This change does not affect the algebraic expression for the gradient, but it does change

which points are lossless and should not cause an update. The relationship between the perceptron criterion and the hinge loss is shown in Figure 1.6. This similarity becomes particularly evident when the perceptron updates of Equation 1.6 are rewritten as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha \sum_{(\overline{X},y) \in S^+} y\overline{X} \tag{1.10}$$

Here, S^+ is defined as the set of all misclassified training points $\overline{X} \in S$ that satisfy the condition $y(\overline{W} \cdot \overline{X}) < 0$. This update seems to look somewhat different from the perceptron, because the perceptron uses the error $E(\overline{X})$ for the update, which is replaced with y in the update above. A key point is that the (integer) error value $E(\overline{X}) = (y - \text{sign}\{\overline{W} \cdot \overline{X}\}) \in \{-2, +2\}$ can never be 0 for misclassified points in S^+. Therefore, we have $E(\overline{X}) = 2y$ *for misclassified points*, and $E(\overline{X})$ can be replaced with y in the updates after absorbing the factor of 2 within the learning rate. This update is identical to that used by the primal support vector machine (SVM) algorithm [448], except that the updates are performed only for the misclassified points in the perceptron, whereas the SVM also uses the marginally correct points near the decision boundary for updates. Note that the SVM uses the condition $y(\overline{W} \cdot \overline{X}) < 1$ [instead of using the condition $y(\overline{W} \cdot \overline{X}) < 0$] to define S^+, which is one of the key differences between the two algorithms. This point shows that the perceptron is fundamentally not very different from well-known machine learning algorithms like the support vector machine in spite of its different origins. Freund and Schapire provide a beautiful exposition of the role of margin in improving stability of the perceptron and also its relationship with the support vector machine [123]. It turns out that many traditional machine learning models can be viewed as minor variations of shallow neural architectures like the perceptron. The relationships between classical machine learning models and shallow neural networks are described in detail in Chapter 2.

1.2.1.3 Choice of Activation and Loss Functions

The choice of activation function is a critical part of neural network design. In the case of the perceptron, the choice of the sign activation function is motivated by the fact that a binary class label needs to be predicted. However, it is possible to have other types of situations where different target variables may be predicted. For example, if the target variable to be predicted is real, then it makes sense to use the identity activation function, and the resulting algorithm is the same as least-squares regression. If it is desirable to predict a probability of a binary class, it makes sense to use a *sigmoid* function for activating the output node, so that the prediction $\hat{y}$ indicates the probability that the observed value, y, of the dependent variable is 1. The negative logarithm of $|y/2 - 0.5 + \hat{y}|$ is used as the loss, assuming that y is coded from $\{-1, 1\}$. If $\hat{y}$ is the probability that y is 1, then $|y/2 - 0.5 + \hat{y}|$ is the probability that the correct value is predicted. This assertion is easy to verify by examining the two cases where y is 0 or 1. This loss function can be shown to be representative of the negative log-likelihood of the training data (see Section 2.2.3 of Chapter 2).

The importance of nonlinear activation functions becomes significant when one moves from the single-layered perceptron to the multi-layered architectures discussed later in this chapter. Different types of nonlinear functions such as the *sign*, *sigmoid*, or *hyperbolic tangents* may be used in various layers. We use the notation Φ to denote the activation function:

$$\hat{y} = \Phi(\overline{W} \cdot \overline{X}) \tag{1.11}$$

Therefore, a neuron really computes two functions within the node, which is why we have incorporated the summation symbol Σ as well as the activation symbol Φ within a neuron. The break-up of the neuron computations into two separate values is shown in Figure 1.7.

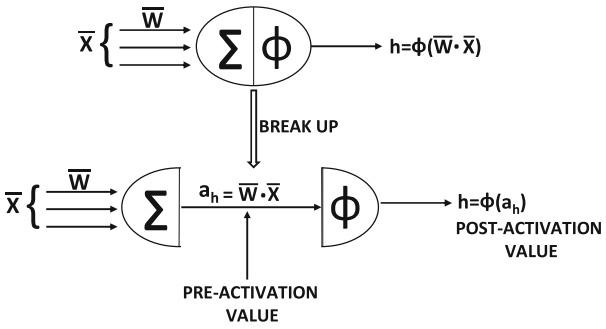


Figure 1.7: Pre-activation and post-activation values within a neuron

The value computed before applying the activation function $\Phi(\cdot)$ will be referred to as the *pre-activation value*, whereas the value computed after applying the activation function is referred to as the *post-activation value*. The output of a neuron is always the post-activation value, although the pre-activation variables are often used in different types of analyses, such as the computations of the *backpropagation algorithm* discussed later in this chapter. The pre-activation and post-activation values of a neuron are shown in Figure 1.7.

The most basic activation function $\Phi(\cdot)$ is the identity or linear activation, which provides no nonlinearity:

$$\Phi(v) = v$$

The linear activation function is often used in the output node, when the target is a real value. It is even used for discrete outputs when a smoothed surrogate loss function needs to be set up.

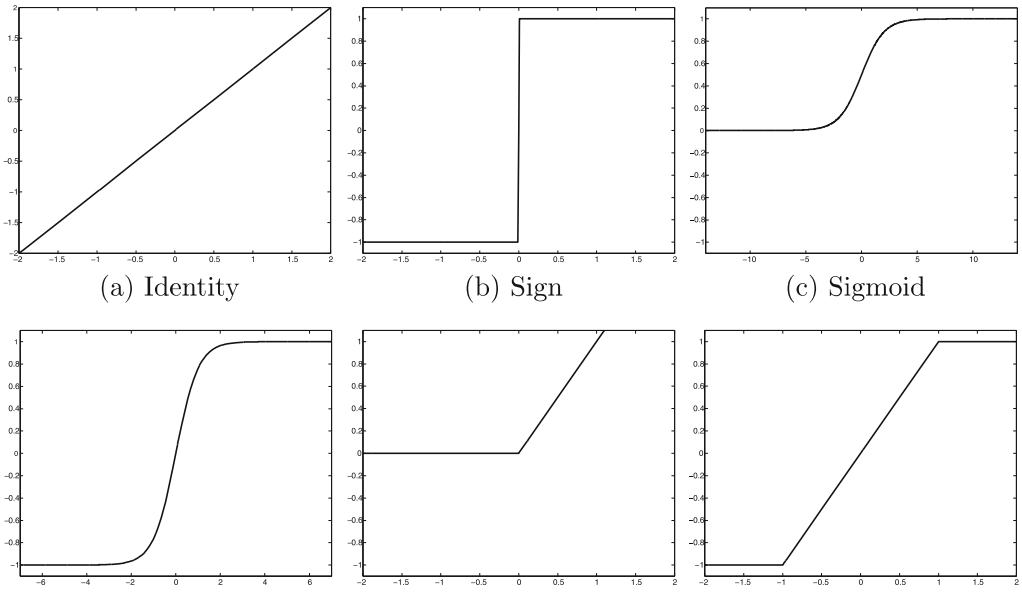
The classical activation functions that were used early in the development of neural networks were the sign, sigmoid, and the hyperbolic tangent functions:

$$\Phi(v) = \text{sign}(v) \text{ (sign function)}$$
$$\Phi(v) = \frac{1}{1 + e^{-v}} \text{ (sigmoid function)}$$
$$\Phi(v) = \frac{e^{2v} - 1}{e^{2v} + 1} \text{ (tanh function)}$$

While the sign activation can be used to map to binary outputs at prediction time, its non-differentiability prevents its use for creating the loss function at training time. For example, while the perceptron uses the sign function for prediction, the perceptron criterion in training only requires linear activation. The sigmoid activation outputs a value in $(0, 1)$, which is helpful in performing computations that should be interpreted as probabilities. Furthermore, it is also helpful in creating probabilistic outputs and constructing loss functions derived from maximum-likelihood models. The tanh function has a shape similar to that of the sigmoid function, except that it is horizontally re-scaled and vertically translated/re-scaled to $[-1, 1]$. The tanh and sigmoid functions are related as follows (see Exercise 3):

$$\tanh(v) = 2 \cdot \text{sigmoid}(2v) - 1$$

The tanh function is preferable to the sigmoid when the outputs of the computations are desired to be both positive and negative. Furthermore, its mean-centering and larger gradient



(a) Identity (b) Sign (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

Figure 1.8: Various activation functions

(because of stretching) with respect to sigmoid makes it easier to train. The sigmoid and the tanh functions have been the historical tools of choice for incorporating nonlinearity in the neural network. In recent years, however, a number of piecewise linear activation functions have become more popular:

$$\Phi(v) = \max\{v, 0\} \text{ (Rectified Linear Unit [ReLU])}$$
$$\Phi(v) = \max\left\{\min\left[v, 1\right], -1\right\} \text{ (hard tanh)}$$

The ReLU and hard tanh activation functions have largely replaced the sigmoid and soft tanh activation functions in modern neural networks because of the ease in training multi-layered neural networks with these activation functions.

Pictorial representations of all the aforementioned activation functions are illustrated in Figure 1.8. It is noteworthy that all activation functions shown here are monotonic. Furthermore, other than the identity activation function, most[1] of the other activation functions *saturate* at large absolute values of the argument at which increasing further does not change the activation much.

As we will see later, such nonlinear activation functions are also very useful in multilayer networks, because they help in creating more powerful compositions of different types of functions. Many of these functions are referred to as *squashing* functions, as they map the outputs from an arbitrary range to bounded outputs. The use of a nonlinear activation plays a fundamental role in increasing the modeling power of a network. If a network used only linear activations, it would not provide better modeling power than a single-layer linear network. This issue is discussed in Section 1.5.

[1]The ReLU shows asymmetric saturation.

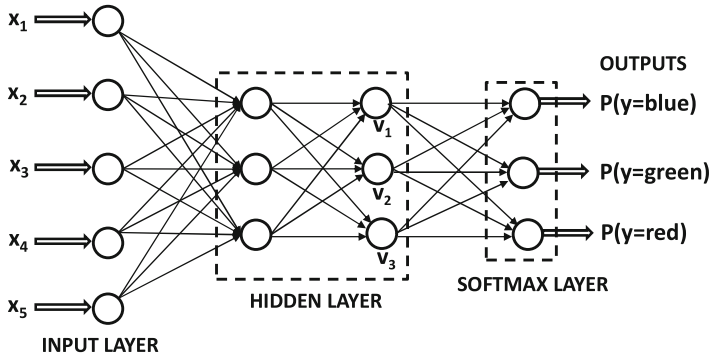


Figure 1.9: An example of multiple outputs for categorical classification with the use of a softmax layer

1.2.1.4 Choice and Number of Output Nodes

The choice and number of output nodes is also tied to the activation function, which in turn depends on the application at hand. For example, if k-way classification is intended, k output values can be used, with a softmax activation function with respect to outputs $\overline{v} = [v_1, \ldots, v_k]$ at the nodes in a given layer. Specifically, the activation function for the ith output is defined as follows:

$$\Phi(\overline{v})_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\} \tag{1.12}$$

It is helpful to think of these k values as the values output by k nodes, in which the inputs are $v_1 \ldots v_k$. An example of the softmax function with three outputs is illustrated in Figure 1.9, and the values v_1, v_2, and v_3 are also shown in the same figure. Note that the three outputs correspond to the probabilities of the three classes, and they convert the three outputs of the final hidden layer into probabilities with the softmax function. The final hidden layer often uses linear (identity) activations, when it is input into the softmax layer. Furthermore, there are no weights associated with the softmax layer, since it is only converting real-valued outputs into probabilities. The use of softmax with a single hidden layer of linear activations exactly implements a model, which is referred to as *multinomial logistic regression* [6]. Similarly, many variations like multi-class SVMs can be easily implemented with neural networks. Another example of a case in which multiple output nodes are used is the *autoencoder*, in which each input data point is fully reconstructed by the output layer. The autoencoder can be used to implement matrix factorization methods like *singular value decomposition*. This architecture will be discussed in detail in Chapter 2. The simplest neural networks that simulate basic machine learning algorithms are instructive because they lie on the continuum between traditional machine learning and deep networks. By exploring these architectures, one gets a better idea of the relationship between traditional machine learning and neural networks, and also the advantages provided by the latter.

1.2.1.5 Choice of Loss Function

The choice of the loss function is critical in defining the outputs in a way that is sensitive to the application at hand. For example, least-squares regression with numeric outputs

requires a simple squared loss of the form $(y - \hat{y})^2$ for a single training instance with target y and prediction $\hat{y}$. One can also use other types of loss like *hinge loss* for $y \in \{-1, +1\}$ and real-valued prediction $\hat{y}$ (with identity activation):

$$L = \max\{0, 1 - y \cdot \hat{y}\} \tag{1.13}$$

The hinge loss can be used to implement a learning method, which is referred to as a *support vector machine*.

For multiway predictions (like predicting word identifiers or one of multiple classes), the softmax output is particularly useful. However, a softmax output is probabilistic, and therefore it requires a different type of loss function. In fact, for probabilistic predictions, two different types of loss functions are used, depending on whether the prediction is binary or whether it is multiway:

1. **Binary targets (logistic regression):** In this case, it is assumed that the observed value y is drawn from $\{-1, +1\}$, and the prediction $\hat{y}$ is a an arbitrary numerical value on using the identity activation function. In such a case, the loss function for a single instance with observed value y and real-valued prediction $\hat{y}$ (with identity activation) is defined as follows:

$$L = \log(1 + \exp(-y \cdot \hat{y})) \tag{1.14}$$

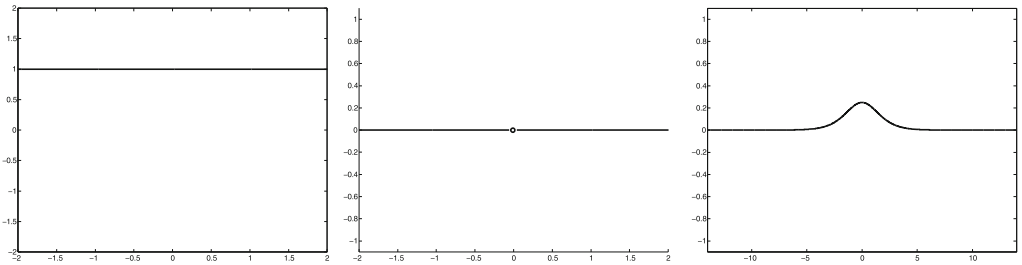
   This type of loss function implements a fundamental machine learning method, referred to as *logistic regression*. Alternatively, one can use a sigmoid activation function to output $\hat{y} \in (0, 1)$, which indicates the probability that the observed value y is 1. Then, the negative logarithm of $|y/2 - 0.5 + \hat{y}|$ provides the loss, assuming that y is coded from $\{-1, 1\}$. This is because $|y/2 - 0.5 + \hat{y}|$ indicates the probability that the prediction is correct. This observation illustrates that one can use various combinations of activation and loss functions to achieve the same result.

2. **Categorical targets:** In this case, if $\hat{y}_1 \ldots \hat{y}_k$ are the probabilities of the k classes (using the softmax activation of Equation 1.9), and the rth class is the ground-truth class, then the loss function for a single instance is defined as follows:

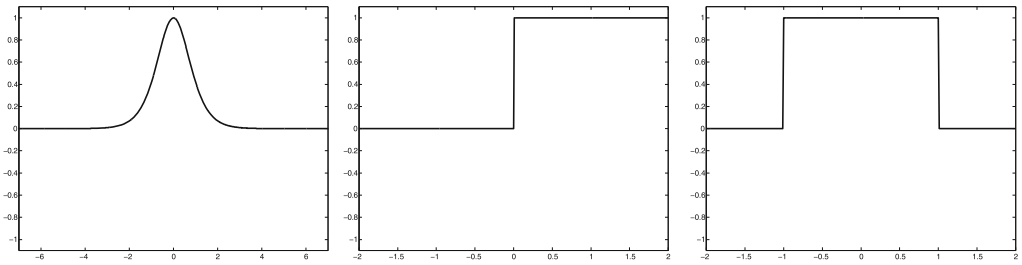
$$L = -\log(\hat{y}_r) \tag{1.15}$$

   This type of loss function implements multinomial logistic regression, and it is referred to as the *cross-entropy loss*. Note that binary logistic regression is identical to multinomial logistic regression, when the value of k is set to 2 in the latter.

The key point to remember is that the nature of the output nodes, the activation function, and the loss function depend on the application at hand. Furthermore, these choices also depend on one another. Even though the perceptron is often presented as the quintessential representative of single-layer networks, it is only a single representative out of a very large universe of possibilities. In practice, one rarely uses the perceptron criterion as the loss function. For discrete-valued outputs, it is common to use softmax activation with cross-entropy loss. For real-valued outputs, it is common to use linear activation with squared loss. Generally, cross-entropy loss is easier to optimize than squared loss.



(a) Identity (b) Sign (c) Sigmoid



(d) Tanh (e) ReLU (f) Hard Tanh

Figure 1.10: The derivatives of various activation functions

1.2.1.6 Some Useful Derivatives of Activation Functions

Most neural network learning is primarily related to gradient-descent with activation functions. For this reason, the derivatives of these activation functions are used repeatedly in this book, and gathering them in a single place for future reference is useful. This section provides details on the derivatives of these loss functions. Later chapters will extensively refer to these results.

1. *Linear and sign activations:* The derivative of the linear activation function is 1 at all places. The derivative of $\text{sign}(v)$ is 0 at all values of v other than at $v = 0$, where it is discontinuous and non-differentiable. Because of the zero gradient and non-differentiability of this activation function, it is rarely used in the loss function even when it is used for prediction at testing time. The derivatives of the linear and sign activations are illustrated in Figure 1.10(a) and (b), respectively.

2. *Sigmoid activation:* The derivative of sigmoid activation is particularly simple, when it is expressed in terms of the *output* of the sigmoid, rather than the input. Let o be the output of the sigmoid function with argument v:

$$o = \frac{1}{1 + \exp(-v)} \tag{1.16}$$

Then, one can write the derivative of the activation as follows:

$$\frac{\partial o}{\partial v} = \frac{\exp(-v)}{(1 + \exp(-v))^2} \tag{1.17}$$

The key point is that this sigmoid can be written more conveniently in terms of the outputs:

$$\frac{\partial o}{\partial v} = o(1 - o) \tag{1.18}$$

The derivative of the sigmoid is often used as a function of the output rather than the input. The derivative of the sigmoid activation function is illustrated in Figure 1.10(c).

3. *Tanh activation:* As in the case of the sigmoid activation, the tanh activation is often used as a function of the output o rather than the input v:

$$o = \frac{\exp(2v) - 1}{\exp(2v) + 1} \tag{1.19}$$

One can then compute the gradient as follows:

$$\frac{\partial o}{\partial v} = \frac{4 \cdot \exp(2v)}{(\exp(2v) + 1)^2} \tag{1.20}$$

One can also write this derivative in terms of the output o:

$$\frac{\partial o}{\partial v} = 1 - o^2 \tag{1.21}$$

The derivative of the tanh activation is illustrated in Figure 1.10(d).

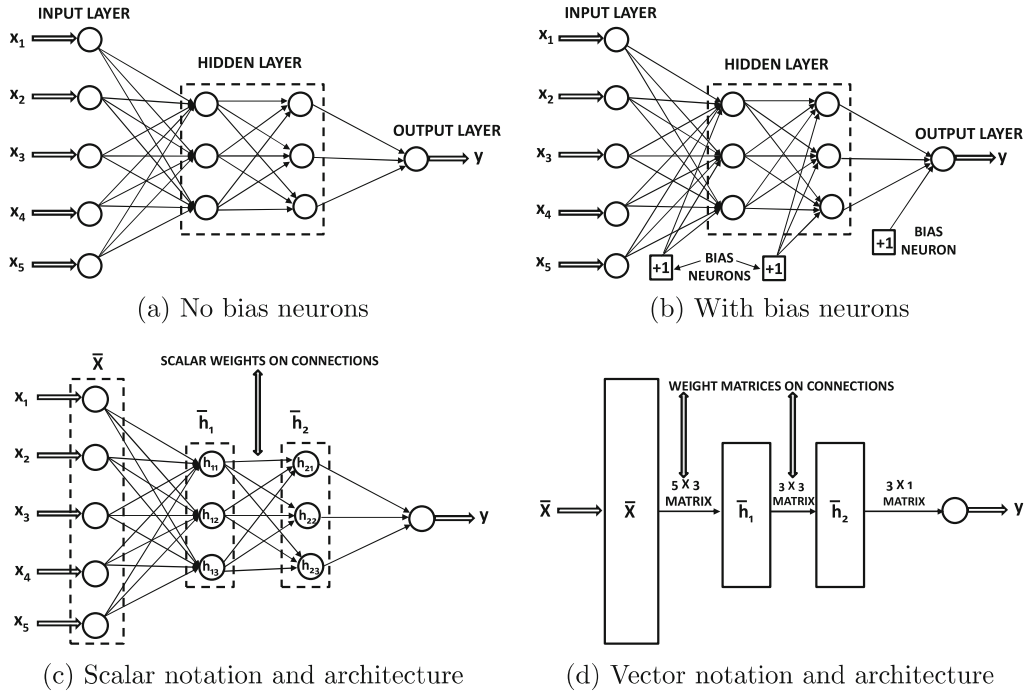
4. *ReLU and hard tanh activations:* The ReLU takes on a partial derivative value of 1 for non-negative values of its argument, and 0, otherwise. The hard tanh function takes on a partial derivative value of 1 for values of the argument in $[-1, +1]$ and 0, otherwise. The derivatives of the ReLU and hard tanh activations are illustrated in Figure 1.10(e) and (f), respectively.

1.2.2 Multilayer Neural Networks

Multilayer neural networks contain more than one computational layer. The perceptron contains an input and output layer, of which the output layer is the only computation-performing layer. The input layer transmits the data to the output layer, and all computations are completely visible to the user. Multilayer neural networks contain multiple computational layers; the additional intermediate layers (between input and output) are referred to as *hidden layers* because the computations performed are not visible to the user. The specific architecture of multilayer neural networks is referred to as *feed-forward* networks, because successive layers feed into one another in the forward direction from input to output. The default architecture of feed-forward networks assumes that all nodes in one layer are connected to those of the next layer. Therefore, the architecture of the neural network is almost fully defined, once the number of layers and the number/type of nodes in each layer have been defined. The only remaining detail is the loss function that is optimized in the output layer. Although the perceptron algorithm uses the perceptron criterion, this is not the only choice. It is extremely common to use softmax outputs with cross-entropy loss for discrete prediction and linear outputs with squared loss for real-valued prediction.

As in the case of single-layer networks, bias neurons can be used both in the hidden layers and in the output layers. Examples of multilayer networks with or without the bias neurons are shown in Figure 1.11(a) and (b), respectively. In each case, the neural network

contains three layers. Note that the input layer is often not counted, because it simply transmits the data and no computation is performed in that layer. If a neural network contains $p_1 \ldots p_k$ units in each of its k layers, then the (column) vector representations of these outputs, denoted by $\overline{h}_1 \ldots \overline{h}_k$ have dimensionalities $p_1 \ldots p_k$. Therefore, the number of units in each layer is referred to as the *dimensionality* of that layer.



(a) No bias neurons

(b) With bias neurons

(c) Scalar notation and architecture

(d) Vector notation and architecture

Figure 1.11: The basic architecture of a feed-forward network with two hidden layers and a single output layer. Even though each unit contains a single scalar variable, one often represents all units within a single layer as a single vector unit. Vector units are often represented as rectangles and have connection *matrices* between them.

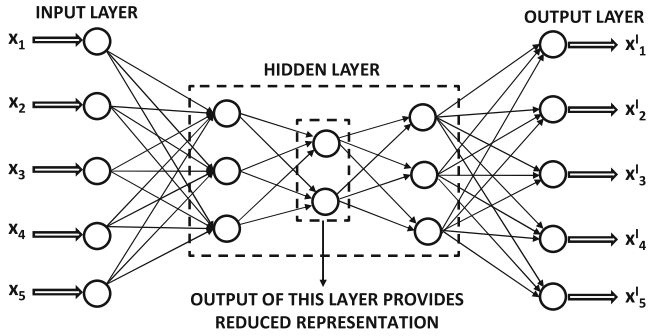


Figure 1.12: An example of an autoencoder with multiple outputs

The weights of the connections between the input layer and the first hidden layer are contained in a *matrix* W_1 with size $d \times p_1$, whereas the weights between the rth hidden layer and the $(r+1)$th hidden layer are denoted by the $p_r \times p_{r+1}$ matrix denoted by W_r. If the output layer contains o nodes, then the final matrix W_{k+1} is of size $p_k \times o$. The d-dimensional input vector $\overline{x}$ is transformed into the outputs using the following recursive equations:

$$\overline{h}_1 = \Phi(W_1^T \overline{x}) \qquad \text{[Input to Hidden Layer]}$$
$$\overline{h}_{p+1} = \Phi(W_{p+1}^T \overline{h}_p) \quad \forall p \in \{1 \ldots k-1\} \qquad \text{[Hidden to Hidden Layer]}$$
$$\overline{o} = \Phi(W_{k+1}^T \overline{h}_k) \qquad \text{[Hidden to Output Layer]}$$

Here, the activation functions like the sigmoid function are applied in *element-wise* fashion to their vector arguments. However, some activation functions such as the softmax (which are typically used in the output layers) naturally have vector arguments. Even though each unit of a neural network contains a single variable, many architectural diagrams combine the units in a single layer to create a single vector unit, which is represented as a *rectangle* rather than a *circle*. For example, the architectural diagram in Figure 1.11(c) (with scalar units) has been transformed to a vector-based neural architecture in Figure 1.11(d). Note that the connections between the vector units are now matrices. Furthermore, an implicit assumption in the vector-based neural architecture is that all units in a layer use the same activation function, which is applied in element-wise fashion to that layer. This constraint is usually not a problem, because most neural architectures use the same activation function throughout the computational pipeline, with the only deviation caused by the nature of the output layer. Throughout this book, neural architectures in which units contain vector variables will be depicted with rectangular units, whereas scalar variables will correspond to circular units.

Note that the aforementioned recurrence equations and vector architectures are valid only for layer-wise feed-forward networks, and cannot always be used for unconventional architectural designs. It is possible to have all types of unconventional designs in which inputs might be incorporated in intermediate layers, or the topology might allow connections between non-consecutive layers. Furthermore, the functions computed at a node may not always be in the form of a combination of a linear function and an activation. It is possible to have all types of arbitrary computational functions at nodes.

Although a very classical type of architecture is shown in Figure 1.11, it is possible to vary on it in many ways, such as allowing multiple output nodes. These choices are often determined by the goals of the application at hand (e.g., classification or dimensionality reduction). A classical example of the dimensionality reduction setting is the autoencoder, which recreates the outputs from the inputs. Therefore, the number of outputs and inputs is equal, as shown in Figure 1.12. The constricted hidden layer in the middle outputs the reduced representation of each instance. As a result of this constriction, there is some loss in the representation, which typically corresponds to the noise in the data. The outputs of the hidden layers correspond to the reduced representation of the data. In fact, a shallow variant of this scheme can be shown to be mathematically equivalent to a well-known dimensionality reduction method known as *singular value decomposition*. As we will learn in Chapter 2, increasing the depth of the network results in inherently more powerful reductions.

Although a fully connected architecture is able to perform well in many settings, better performance is often achieved by pruning many of the connections or sharing them in an insightful way. Typically, these insights are obtained by using a domain-specific understanding of the data. A classical example of this type of weight pruning and sharing is that of

the *convolutional neural network architecture* (cf. Chapter 8), in which the architecture is carefully designed in order to conform to the typical properties of image data. Such an approach minimizes the risk of *overfitting* by incorporating domain-specific insights (or *bias*). As we will discuss later in this book (cf. Chapter 4), overfitting is a pervasive problem in neural network design, so that the network often performs very well on the training data, but it *generalizes* poorly to unseen test data. This problem occurs when the number of free parameters, (which is typically equal to the number of weight connections), is too large compared to the size of the training data. In such cases, the large number of parameters memorize the specific nuances of the training data, but fail to recognize the statistically significant patterns for classifying unseen test data. Clearly, increasing the number of nodes in the neural network tends to encourage overfitting. Much recent work has been focused both on the architecture of the neural network as well as on the computations performed within each node in order to minimize overfitting. Furthermore, the way in which the neural network is trained also has an impact on the quality of the final solution. Many clever methods, such as *pretraining* (cf. Chapter 4), have been proposed in recent years in order to improve the quality of the learned solution. This book will explore these advanced training methods in detail.

1.2.3 The Multilayer Network as a Computational Graph

It is helpful to view a neural network as a *computational graph*, which is constructed by piecing together many basic parametric models. Neural networks are fundamentally more powerful than their building blocks because the parameters of these models are learned *jointly* to create a highly optimized composition function of these models. The common use of the term "perceptron" to refer to the basic unit of a neural network is somewhat misleading, because there are many variations of this basic unit that are leveraged in different settings. In fact, it is far more common to use logistic units (with sigmoid activation) and piecewise/fully linear units as building blocks of these models.

A multilayer network evaluates compositions of functions computed at individual nodes. A path of length 2 in the neural network in which the function $f(\cdot)$ follows $g(\cdot)$ can be considered a composition function $f(g(\cdot))$. Furthermore, if $g_1(\cdot)$, $g_2(\cdot) \ldots g_k(\cdot)$ are the functions computed in layer m, and a particular layer-$(m+1)$ node computes $f(\cdot)$, then the composition function computed by the layer-$(m+1)$ node in terms of the layer-m inputs is $f(g_1(\cdot), \ldots g_k(\cdot))$. The use of nonlinear activation functions is the key to increasing the power of multiple layers. If all layers use an identity activation function, then a multilayer network can be shown to simplify to linear regression. It has been shown [208] that a network with a single hidden layer of nonlinear units (with a wide ranging choice of squashing functions like the sigmoid unit) and a single (linear) output layer can compute almost any "reasonable" function. As a result, neural networks are often referred to as *universal function approximators*, although this theoretical claim is not always easy to translate into practical usefulness. The main issue is that the number of hidden units required to do so is rather large, which increases the number of parameters to be learned. This results in practical problems in training the network with a limited amount of data. In fact, deeper networks are often preferred because they reduce the number of hidden units in each layer as well as the overall number of parameters.

The "building block" description is particularly appropriate for multilayer neural networks. Very often, off-the-shelf softwares for building neural networks[2] provide analysts

[2]Examples include Torch [572], Theano [573], and TensorFlow [574].

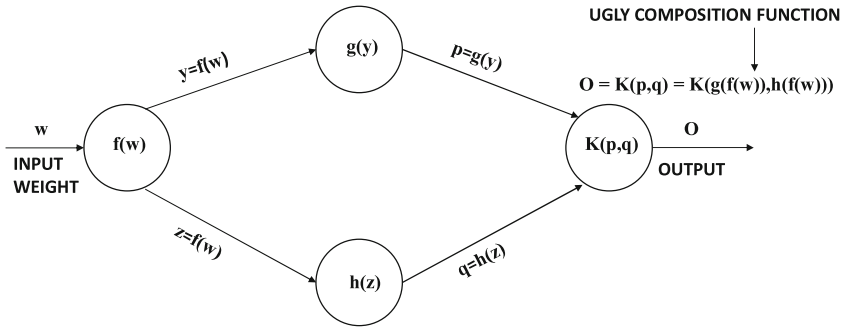
with access to these building blocks. The analyst is able to specify the number and type of units in each layer along with an off-the-shelf or customized loss function. A deep neural network containing tens of layers can often be described in a few hundred lines of code. All the learning of the weights is done automatically by the *backpropagation algorithm* that uses dynamic programming to work out the complicated parameter update steps of the underlying computational graph. The analyst does not have to spend the time and effort to explicitly work out these steps. This makes the process of trying different types of architectures relatively painless for the analyst. Building a neural network with many of the off-the-shelf softwares is often compared to a child constructing a toy from building blocks that appropriately fit with one another. Each block is like a unit (or a layer of units) with a particular type of activation. Much of this ease in training neural networks is attributable to the backpropagation algorithm, which shields the analyst from explicitly working out the parameter update steps of what is actually an extremely complicated optimization problem. Working out these steps is often the most difficult part of most machine learning algorithms, and an important contribution of the neural network paradigm is to bring modular thinking into machine learning. In other words, the modularity in neural network design translates to modularity in learning its parameters; the specific name for the latter type of modularity is "backpropagation." This makes the design of neural networks more of an (experienced) engineer's task rather than a mathematical exercise.

1.3 Training a Neural Network with Backpropagation

In the single-layer neural network, the training process is relatively straightforward because the error (or loss function) can be computed as a direct function of the weights, which allows easy gradient computation. In the case of multi-layer networks, the problem is that the loss is a complicated composition function of the weights in earlier layers. The gradient of a composition function is computed using the backpropagation algorithm. The backpropagation algorithm leverages the chain rule of differential calculus, which computes the error gradients in terms of summations of local-gradient products over the various paths from a node to the output. Although this summation has an exponential number of components (paths), one can compute it efficiently using *dynamic programming*. The backpropagation algorithm is a direct application of dynamic programming. It contains two main phases, referred to as the *forward* and *backward* phases, respectively. The forward phase is required to compute the output values and the local derivatives at various nodes, and the backward phase is required to accumulate the products of these local values over all paths from the node to the output:

1. *Forward phase:* In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the current set of weights. The final predicted output can be compared to that of the training instance and the derivative of the loss function with respect to the output is computed. The derivative of this loss now needs to be computed with respect to the weights in all layers in the backwards phase.

2. *Backward phase:* The main goal of the backward phase is to learn the gradient of the loss function with respect to the different weights by using the chain rule of differential calculus. These gradients are used to update the weights. Since these gradients are learned in the backward direction, starting from the output node, this learning process is referred to as the backward phase. Consider a sequence of hidden units



$$\begin{aligned}\frac{\partial o}{\partial w} &= \frac{\partial o}{\partial p}\cdot\frac{\partial p}{\partial w}+\frac{\partial o}{\partial q}\cdot\frac{\partial q}{\partial w} \quad \text{[Multivariable Chain Rule]}\\ &= \frac{\partial o}{\partial p}\cdot\frac{\partial p}{\partial y}\cdot\frac{\partial y}{\partial w}+\frac{\partial o}{\partial q}\cdot\frac{\partial q}{\partial z}\cdot\frac{\partial z}{\partial w} \quad \text{[Univariate Chain Rule]}\\ &= \underbrace{\frac{\partial K(p,q)}{\partial p}\cdot g'(y)\cdot f'(w)}_{\text{First path}}+\underbrace{\frac{\partial K(p,q)}{\partial q}\cdot h'(z)\cdot f'(w)}_{\text{Second path}}\end{aligned}$$

Figure 1.13: **Illustration of chain rule in computational graphs:** The products of node-specific partial derivatives along paths from weight w to output o are aggregated. The resulting value yields the derivative of output o with respect to weight w. Only two paths between input and output exist in this simplified example.

$h_1, h_2, \ldots, h_k$ followed by output o, with respect to which the loss function L is computed. Furthermore, assume that the weight of the connection from hidden unit h_r to h_{r+1} is $w_{(h_r,h_{r+1})}$. Then, in the case that a single path exists from h_1 to o, one can derive the gradient of the loss function with respect to any of these edge weights using the chain rule:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \frac{\partial L}{\partial o}\cdot\left[\frac{\partial o}{\partial h_k}\prod_{i=r}^{k-1}\frac{\partial h_{i+1}}{\partial h_i}\right]\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \quad \forall r \in 1\ldots k \tag{1.22}$$

The aforementioned expression assumes that only a *single path* from h_1 to o exists in the network, whereas an exponential number of paths might exist in reality. A generalized variant of the chain rule, referred to as the *multivariable chain rule*, computes the gradient in a computational graph, where more than one path might exist. This is achieved by adding the composition along each of the paths from h_1 to o. An example of the chain rule in a computational graph with two paths is shown in Figure 1.13. Therefore, one generalizes the above expression to the case where a set $\mathcal{P}$ of paths exist from h_r to o:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial L}{\partial o}\cdot\left[\sum_{[h_r,h_{r+1},\ldots h_k,o]\in\mathcal{P}}\frac{\partial o}{\partial h_k}\prod_{i=r}^{k-1}\frac{\partial h_{i+1}}{\partial h_i}\right]}_{\text{Backpropagation computes } \Delta(h_r,o)=\frac{\partial L}{\partial h_r}}\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \tag{1.23}$$

The computation of $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$ on the right-hand side is straightforward and will be discussed below (cf. Equation 1.27). However, the path-aggregated term above [annotated by $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$] is aggregated over an exponentially increasing number of paths (with respect to path length), which seems to be intractable at first sight. A key point is that the computational graph of a neural network does not have cycles, and it is possible to compute such an aggregation in a principled way in the backwards direction by first computing $\Delta(h_k, o)$ for nodes h_k closest to o, and then recursively computing these values for nodes in earlier layers in terms of the nodes in later layers. Furthermore, the value of $\Delta(o, o)$ for each output node is initialized as follows:

$$\Delta(o, o) = \frac{\partial L}{\partial o} \tag{1.24}$$

This type of dynamic programming technique is used frequently to efficiently compute all types of path-centric functions in directed acyclic graphs, which would otherwise require an exponential number of operations. The recursion for $\Delta(h_r, o)$ can be derived using the multivariable chain rule:

$$\Delta(h_r, o) = \frac{\partial L}{\partial h_r} = \sum_{h:h_r \Rightarrow h} \frac{\partial L}{\partial h} \frac{\partial h}{\partial h_r} = \sum_{h:h_r \Rightarrow h} \frac{\partial h}{\partial h_r} \Delta(h, o) \tag{1.25}$$

Since each h is in a later layer than h_r, $\Delta(h, o)$ has already been computed while evaluating $\Delta(h_r, o)$. However, we still need to evaluate $\frac{\partial h}{\partial h_r}$ in order to compute Equation 1.25. Consider a situation in which the edge joining h_r to h has weight $w_{(h_r,h)}$, and let a_h be the value computed in hidden unit h just *before* applying the activation function $\Phi(\cdot)$. In other words, we have $h = \Phi(a_h)$, where a_h is a linear combination of its inputs from earlier-layer units incident on h. Then, by the univariate chain rule, the following expression for $\frac{\partial h}{\partial h_r}$ can be derived:

$$\frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial a_h} \cdot \frac{\partial a_h}{\partial h_r} = \frac{\partial \Phi(a_h)}{\partial a_h} \cdot w_{(h_r,h)} = \Phi'(a_h) \cdot w_{(h_r,h)}$$

This value of $\frac{\partial h}{\partial h_r}$ is used in Equation 1.25, which is repeated recursively in the backwards direction, starting with the output node. The corresponding updates in the backwards direction are as follows:

$$\Delta(h_r, o) = \sum_{h:h_r \Rightarrow h} \Phi'(a_h) \cdot w_{(h_r,h)} \cdot \Delta(h, o) \tag{1.26}$$

Therefore, gradients are successively accumulated in the backwards direction, and each node is processed exactly once in a backwards pass. Note that the computation of Equation 1.25 (which requires proportional operations to the number of outgoing edges) needs to be repeated for each incoming edge into the node to compute the gradient with respect to all edge weights. Finally, Equation 1.23 requires the computation of $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$, which is easily computed as follows:

$$\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} = h_{r-1} \cdot \Phi'(a_{h_r}) \tag{1.27}$$

Here, the key gradient that is backpropagated is the derivative with respect to *layer activations*, and the gradient with respect to the weights is easy to compute for any incident edge on the corresponding unit.

It is noteworthy that the dynamic programming recursion of Equation 1.26 can be computed in multiple ways, depending on which variables one uses for intermediate chaining. All these recursions are equivalent in terms of the final result of backpropagation. In the following, we give an alternative version of the dynamic programming recursion, which is more commonly seen in textbooks. Note that Equation 1.23 uses the variables in the hidden layers as the "chain" variables for the dynamic programming recursion. One can also use the pre-activation values of the variables for the chain rule. The pre-activation variables in a neuron are obtained after applying the linear transform (but before applying the activation variables) as the intermediate variables. The pre-activation value of the hidden variable $h = \Phi(a_h)$ is a_h. The differences between the pre-activation and post-activation values within a neuron are shown in Figure 1.7. Therefore, instead of Equation 1.23, one can use the following chain rule:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial L}{\partial o} \cdot \Phi'(a_o) \cdot \left[\sum_{[h_r,h_{r+1},\ldots h_k,o] \in \mathcal{P}} \frac{\partial a_o}{\partial a_{h_k}} \prod_{i=r}^{k-1} \frac{\partial a_{h_{i+1}}}{\partial a_{h_i}} \right]}_{\text{Backpropagation computes } \delta(h_r,o) = \frac{\partial L}{\partial a_{h_r}}} \underbrace{\frac{\partial a_{h_r}}{\partial w_{(h_{r-1},h_r)}}}_{h_{r-1}} \tag{1.28}$$

Here, we have introduced the notation $\delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}}$ instead of $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$ for setting up the recursive equation. The value of $\delta(o, o) = \frac{\partial L}{\partial a_o}$ is initialized as follows:

$$\delta(o, o) = \frac{\partial L}{\partial a_o} = \Phi'(a_o) \cdot \frac{\partial L}{\partial o} \tag{1.29}$$

Then, one can use the multivariable chain rule to set up a similar recursion:

$$\delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}} = \sum_{h:h_r \Rightarrow h} \overbrace{\frac{\partial L}{\partial a_h}}^{\delta(h,o)} \underbrace{\frac{\partial a_h}{\partial a_{h_r}}}_{\Phi'(a_{h_r}) w_{(h_r,h)}} = \Phi'(a_{h_r}) \sum_{h:h_r \Rightarrow h} w_{(h_r,h)} \cdot \delta(h, o) \tag{1.30}$$

This recursion condition is found more commonly in textbooks discussing backpropagation. The partial derivative of the loss with respect to the weight is then computed using $\delta(h_r, o)$ as follows:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \delta(h_r, o) \cdot h_{r-1} \tag{1.31}$$

As with the single-layer network, the process of updating the nodes is repeated to convergence by repeatedly cycling through the training data in epochs. A neural network may sometimes require thousands of epochs through the training data to learn the weights at the different nodes. A detailed description of the backpropagation algorithm and associated issues is provided in Chapter 3. In this chapter, we provide a brief discussion of these issues.

1.4 Practical Issues in Neural Network Training

In spite of the formidable reputation of neural networks as universal function approximators, considerable challenges remain with respect to actually training neural networks to provide this level of performance. These challenges are primarily related to several practical problems associated with training, the most important one of which is *overfitting*.