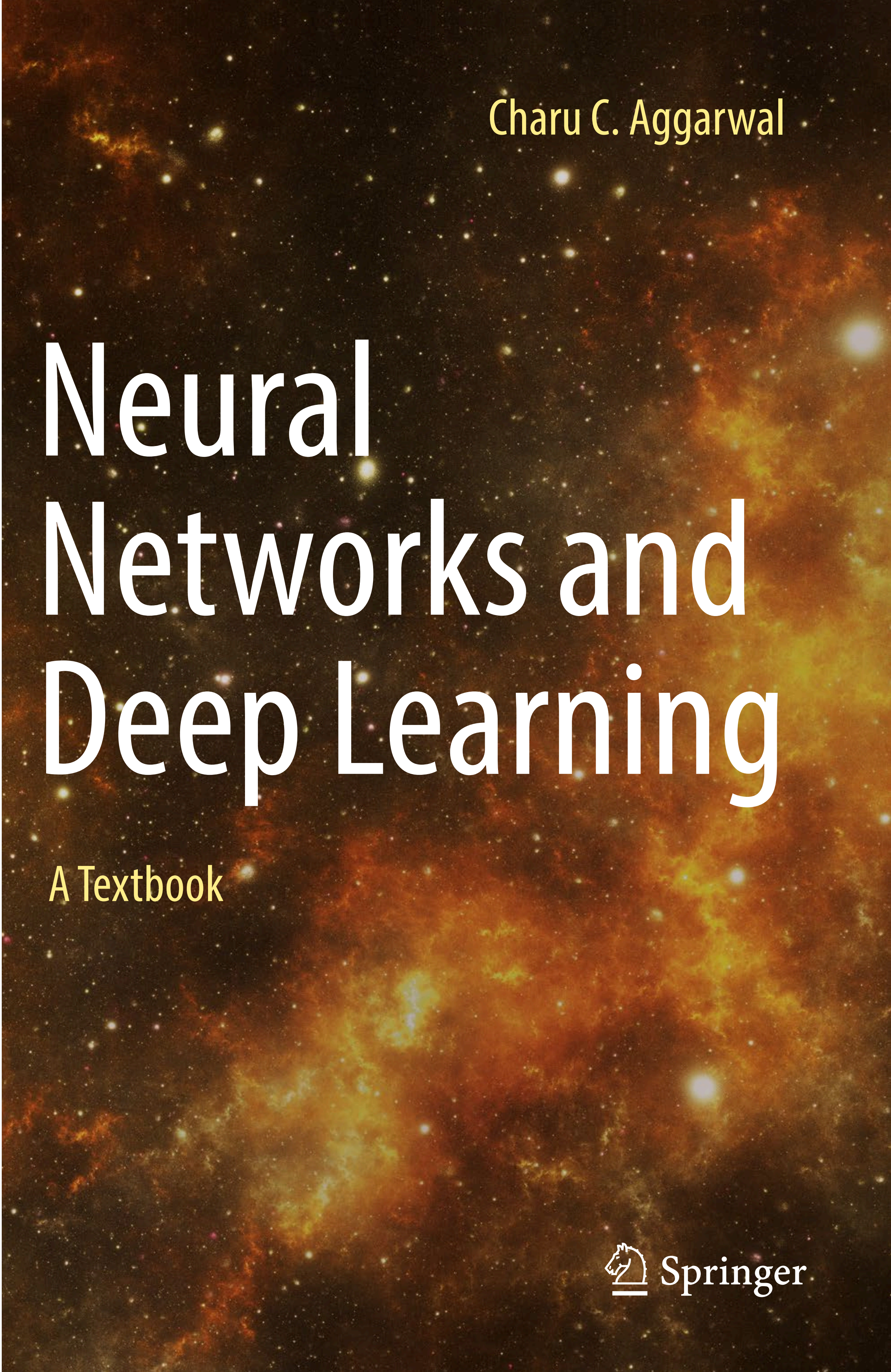


Charu C. Aggarwal

Neural Networks and Deep Learning

A Textbook

Springer

Preface

> "Any A.I. smart enough to pass a Turing test is smart enough to know to fail it."—Ian McDonald

Neural networks were developed to simulate the human nervous system for machine learning tasks by treating the computational units in a learning model in a manner similar to human neurons. The grand vision of neural networks is to create artificial intelligence by building machines whose architecture simulates the computations in the human nervous system. This is obviously not a simple task because the computational power of the fastest computer today is a minuscule fraction of the computational power of a human brain. Neural networks were developed soon after the advent of computers in the fifties and sixties. Rosenblatt's perceptron algorithm was seen as a fundamental cornerstone of neural networks, which caused an initial excitement about the prospects of artificial intelligence. However, after the initial euphoria, there was a period of disappointment in which the data hungry and computationally intensive nature of neural networks was seen as an impediment to their usability. Eventually, at the turn of the century, greater data availability and increasing computational power lead to increased successes of neural networks, and this area was reborn under the new label of "deep learning." Although we are still far from the day that artificial intelligence (AI) is close to human performance, there are specific domains like image recognition, self-driving cars, and game playing, where AI has matched or exceeded human performance. It is also hard to predict what AI might be able to do in the future. For example, few computer vision experts would have thought two decades ago that any automated system could ever perform an intuitive task like categorizing an image more accurately than a human.

Neural networks are *theoretically* capable of learning any mathematical function with sufficient training data, and some variants like recurrent neural networks are known to be *Turing complete*. Turing completeness refers to the fact that a neural network can simulate any learning algorithm, *given sufficient training data*. The sticking point is that the amount of data required to learn even simple tasks is often extraordinarily large, which causes a corresponding increase in training time (if we assume that enough training data is available in the first place). For example, the training time for image recognition, which is a simple task for a human, can be on the order of weeks even on high-performance systems. Furthermore, there are practical issues associated with the stability of neural network training, which are being resolved even today. Nevertheless, given that the speed of computers is expected to increase rapidly over time, and fundamentally more powerful paradigms like quantum computing are on the horizon, the computational issue might not eventually turn out to be quite as critical as imagined.

Although the biological analogy of neural networks is an exciting one and evokes comparisons with science fiction, the mathematical understanding of neural networks is a more mundane one. The neural network abstraction can be viewed as a modular approach of enabling learning algorithms that are based on continuous optimization on a computational graph of dependencies between the input and output. To be fair, this is not very different from traditional work in control theory; indeed, some of the methods used for optimization in control theory are strikingly similar to (and historically preceded) the most fundamental algorithms in neural networks. However, the large amounts of data available in recent years together with increased computational power have enabled experimentation with deeper architectures of these computational graphs than was previously possible. The resulting success has changed the broader perception of the potential of deep learning.

The chapters of the book are organized as follows:

1. *The basics of neural networks:* Chapter 1 discusses the basics of neural network design. Many traditional machine learning models can be understood as special cases of neural learning. Understanding the relationship between traditional machine learning and neural networks is the first step to understanding the latter. The simulation of various machine learning models with neural networks is provided in Chapter 2. This will give the analyst a feel of how neural networks push the envelope of traditional machine learning algorithms.

2. *Fundamentals of neural networks:* Although Chapters 1 and 2 provide an overview of the training methods for neural networks, a more detailed understanding of the training challenges is provided in Chapters 3 and 4. Chapters 5 and 6 present radial-basis function (RBF) networks and restricted Boltzmann machines.

3. *Advanced topics in neural networks:* A lot of the recent success of deep learning is a result of the specialized architectures for various domains, such as recurrent neural networks and convolutional neural networks. Chapters 7 and 8 discuss recurrent and convolutional neural networks. Several advanced topics like deep reinforcement learning, neural Turing mechanisms, and generative adversarial networks are discussed in Chapters 9 and 10.

We have taken care to include some of the "forgotten" architectures like RBF networks and Kohonen self-organizing maps because of their potential in many applications. The book is written for graduate students, researchers, and practitioners. Numerous exercises are available along with a solution manual to aid in classroom teaching. Where possible, an application-centric view is highlighted in order to give the reader a feel for the technology.

Throughout this book, a vector or a multidimensional data point is annotated with a bar, such as $\overline{X}$ or $\overline{y}$. A vector or multidimensional point may be denoted by either small letters or capital letters, as long as it has a bar. Vector dot products are denoted by centered dots, such as $\overline{X} \cdot \overline{Y}$. A matrix is denoted in capital letters without a bar, such as R. Throughout the book, the $n \times d$ matrix corresponding to the entire training data set is denoted by D, with n documents and d dimensions. The individual data points in D are therefore d-dimensional row vectors. On the other hand, vectors with one component for each data point are usually n-dimensional column vectors. An example is the n-dimensional column vector $\overline{y}$ of class variables of n data points. An observed value y_i is distinguished from a predicted value $\hat{y}_i$ by a circumflex at the top of the variable.

Yorktown Heights, NY, USA — Charu C. Aggarwal

Acknowledgments

I would like to thank my family for their love and support during the busy time spent in writing this book. I would also like to thank my manager Nagui Halim for his support during the writing of this book.

Several figures in this book have been provided by the courtesy of various individuals and institutions. The Smithsonian Institution made the image of the Mark I perceptron (cf. Figure 1.5) available at no cost. Saket Sathe provided the outputs in Chapter 7 for the tiny Shakespeare data set, based on code available/described in [233, 580]. Andrew Zisserman provided Figures 8.12 and 8.16 in the section on convolutional visualizations. Another visualization of the feature maps in the convolution network (cf. Figure 8.15) was provided by Matthew Zeiler. NVIDIA provided Figure 9.10 on the convolutional neural network for self-driving cars in Chapter 9, and Sergey Levine provided the image on self-learning robots (cf. Figure 9.9) in the same chapter. Alec Radford provided Figure 10.8, which appears in Chapter 10. Alex Krizhevsky provided Figure 8.9(b) containing *AlexNet*.

This book has benefitted from significant feedback and several collaborations that I have had with numerous colleagues over the years. I would like to thank Quoc Le, Saket Sathe, Karthik Subbian, Jiliang Tang, and Suhang Wang for their feedback on various portions of this book. Shuai Zheng provided feedbback on the section on regularized autoencoders in Chapter 4. I received feedback on the sections on autoencoders from Lei Cai and Hao Yuan. Feedback on the chapter on convolutional neural networks was provided by Hongyang Gao, Shuiwang Ji, and Zhengyang Wang. Shuiwang Ji, Lei Cai, Zhengyang Wang and Hao Yuan also reviewed the Chapters 3 and 7, and suggested several edits. They also suggested the ideas of using Figures 8.6 and 8.7 for elucidating the convolution/deconvolution operations.

For their collaborations, I would like to thank Tarek F. Abdelzaher, Jinghui Chen, Jing Gao, Quanquan Gu, Manish Gupta, Jiawei Han, Alexander Hinneburg, Thomas Huang, Nan Li, Huan Liu, Ruoming Jin, Daniel Keim, Arijit Khan, Latifur Khan, Mohammad M. Masud, Jian Pei, Magda Procopiuc, Guojun Qi, Chandan Reddy, Saket Sathe, Jaideep Srivastava, Karthik Subbian, Yizhou Sun, Jiliang Tang, Min-Hsuan Tsai, Haixun Wang, Jianyong Wang, Min Wang, Suhang Wang, Joel Wolf, Xifeng Yan, Mohammed Zaki, ChengXiang Zhai, and Peixiang Zhao. I would also like to thank my advisor James B. Orlin for his guidance during my early years as a researcher.

I would like to thank Lata Aggarwal for helping me with some of the figures created using PowerPoint graphics in this book. My daughter, Sayani, was helpful in incorporating special effects (e.g., image color, contrast, and blurring) in several JPEG images used at various places in this book.

Contents

Author Biography

Charu C. Aggarwal is a Distinguished Research Staff Member (DRSM) at the IBM T. J. Watson Research Center in Yorktown Heights, New York. He completed his undergraduate degree in Computer Science from the Indian Institute of Technology at Kanpur in 1993 and his Ph.D. from the Massachusetts Institute of Technology in 1996. He has worked extensively in the field of data mining. He has published more than 350 papers in refereed conferences and journals and authored over 80 patents. He is the author or editor of 18 books, including textbooks on data mining, recommender systems, and outlier analysis. Because of the commercial value of his patents, he has thrice been designated a Master Inventor at IBM. He is a recipient of an IBM Corporate Award (2003) for his work on bio-terrorist threat detection in data streams, a recipient of the IBM Outstanding Innovation Award (2008) for his scientific contributions to privacy technology, and a recipient of two IBM Outstanding Technical Achievement Awards (2009, 2015) for his work on data streams/high-dimensional data. He received the EDBT 2014 Test of Time Award for his work on condensation-based privacy-preserving data mining. He is also a recipient of the IEEE ICDM Research Contributions Award (2015), which is one of the two highest awards for influential research contributions in the field of data mining.

He has served as the general co-chair of the IEEE Big Data Conference (2014) and as the program co-chair of the ACM CIKM Conference (2015), the IEEE ICDM Conference (2015), and the ACM KDD Conference (2016). He served as an associate editor of the IEEE Transactions on Knowledge and Data Engineering from 2004 to 2008. He is an associate editor of the IEEE Transactions on Big Data, an action editor of the Data Mining and Knowledge Discovery Journal, and an associate editor of the Knowledge and Information Systems Journal. He serves as the editor-in-chief of the ACM Transactions on Knowledge Discovery from Data as well as the ACM SIGKDD Explorations. He serves on the advisory board of the Lecture Notes on Social Networks, a publication by Springer. He has served as the vice-president of the SIAM Activity Group on Data Mining and is a member of the SIAM industry committee. He is a fellow of the SIAM, ACM, and the IEEE, for “contributions to knowledge discovery and data mining algorithms.”

Chapter 1

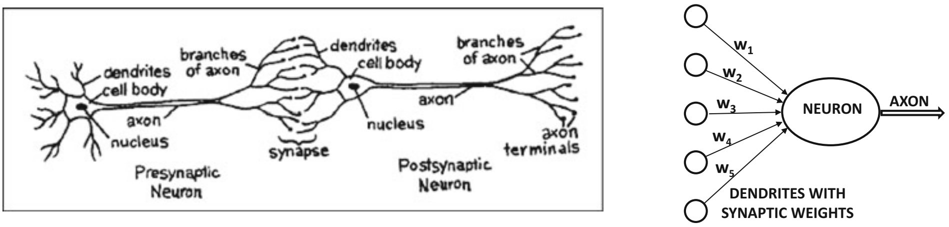
An Introduction to Neural Networks

"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.

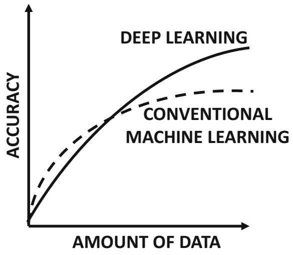


Figure 1.2: An illustrative comparison of the accuracy of a typical machine learning algorithm with that of a large neural network. Deep learners become more attractive than conventional methods primarily when sufficient data/computational power is available. Recent years have seen an increase in data 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.

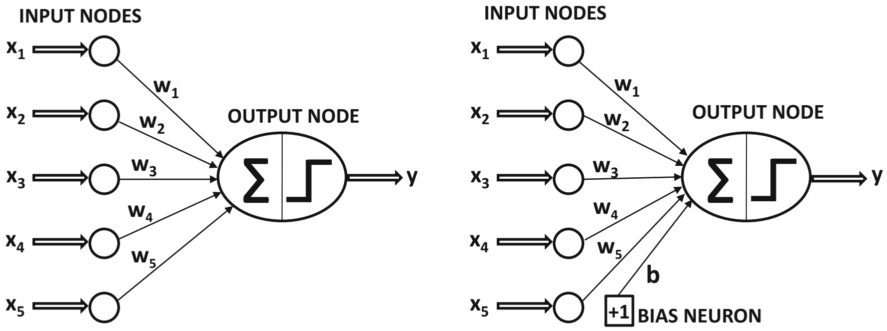
The "big data" era has been enabled by the advances in data collection technology; virtually everything we do today, including purchasing an item, using the phone, or clicking on a site, is collected and stored somewhere. Furthermore, the development of powerful Graphics Processor Units (GPUs) has enabled 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 $\mathcal{D}$ containing 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:

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

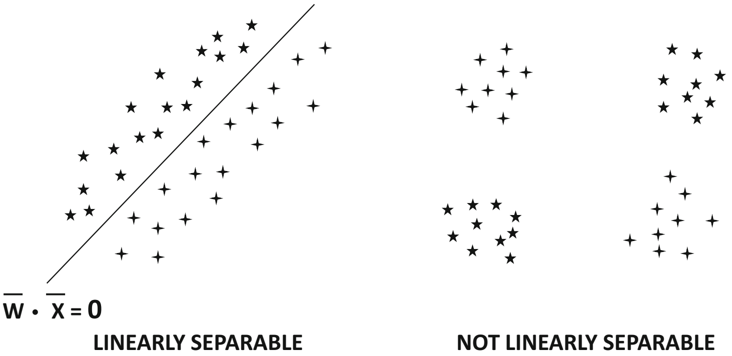


Figure 1.4: Examples of linearly separable and inseparable data in two classes

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 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.

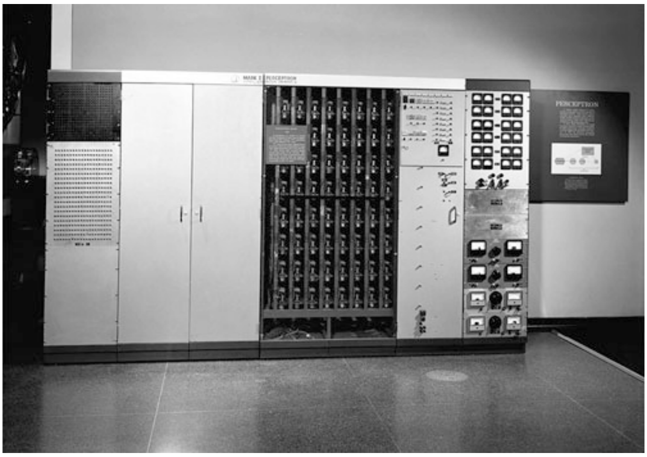


Figure 1.5: The perceptron algorithm was originally implemented using hardware circuits. The image depicts the Mark I perceptron machine built in 1958. (Courtesy: Smithsonian Institute)

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.

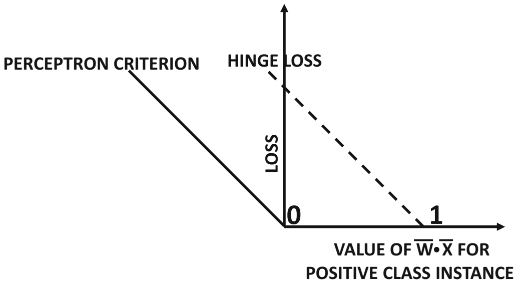


Figure 1.6: Perceptron criterion versus hinge loss

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.

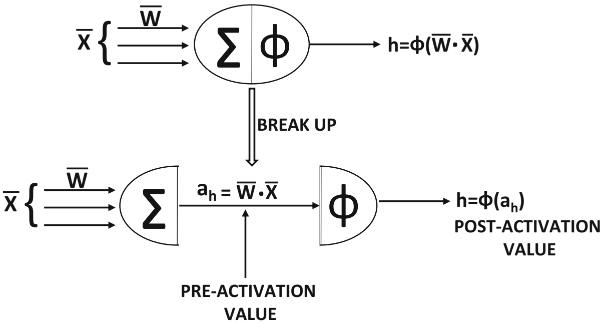


Figure 1.7: Pre-activation and post-activation values within a neuron

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.

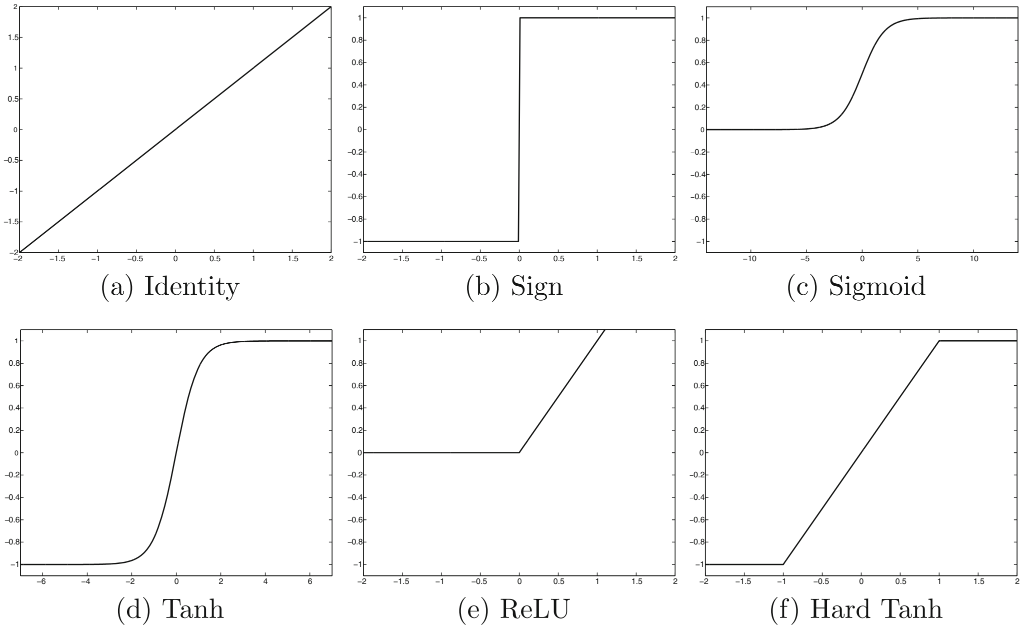
The classical activation functions that were used early in the development of neural networks were the sign, sigmoid, and the hyperbolic tangent functions:

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

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.

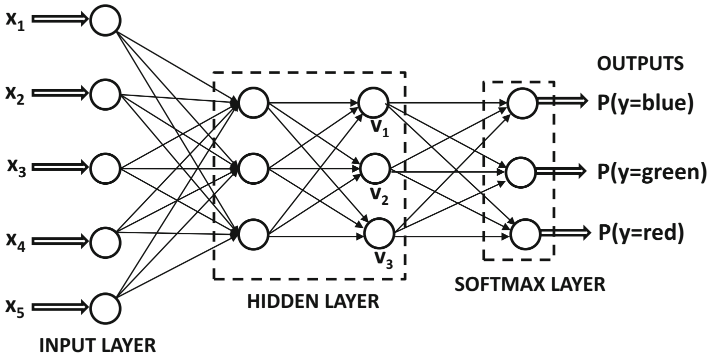


Figure 1.9: An example of multiple outputs for categorical classification with the use of a softmax layer

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:

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

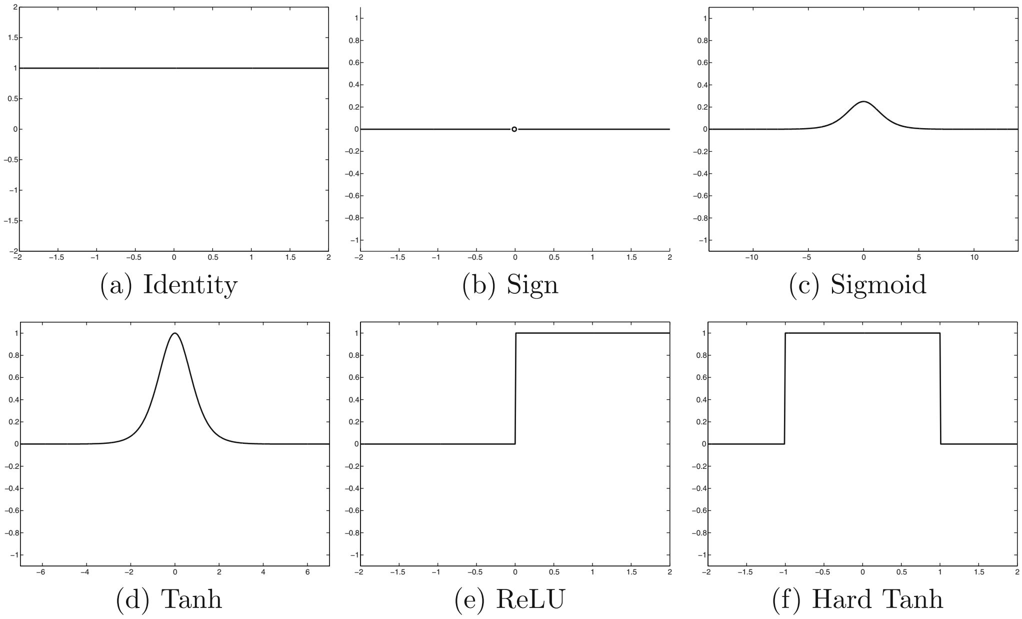
   This type of loss function implements a fundamental machine learning method, referred to as *logistic regression.* Alternatively, one can use a sigmoid activation function to output $\hat{y} \in (0, 1)$, which indicates the probability that the observed value y is 1. Then, the negative logarithm of $|y/2 - 0.5 + \hat{y}|$ 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 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:

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

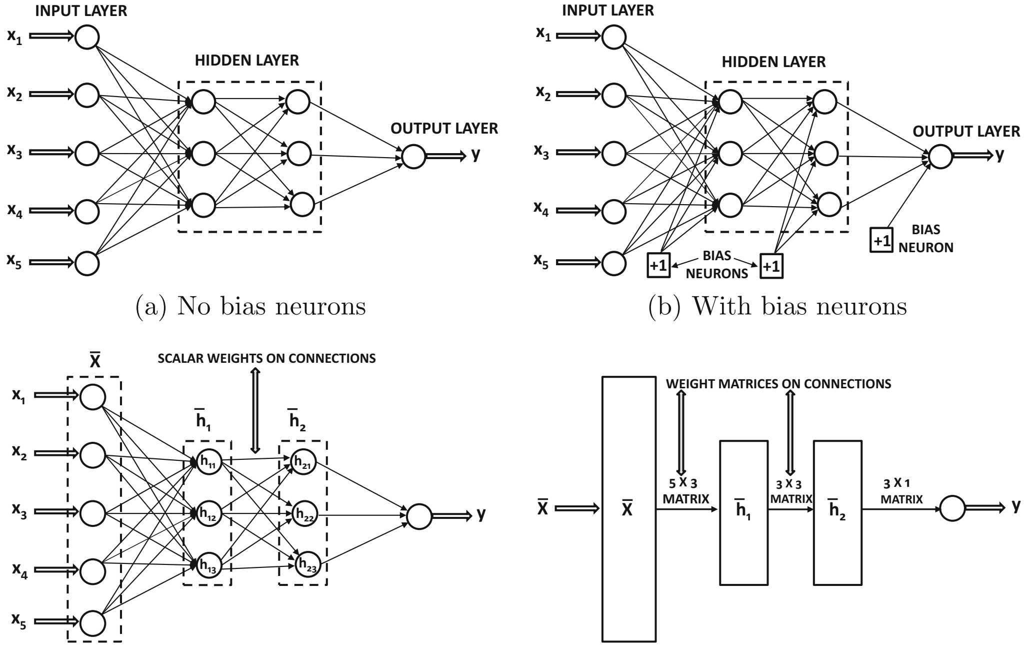
The derivative of the tanh activation is illustrated in Figure 1.10(d).

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.

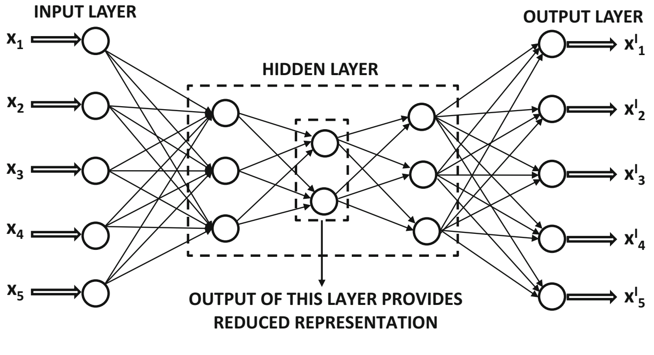


Figure 1.12: An example of an autoencoder with multiple outputs

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:

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

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].

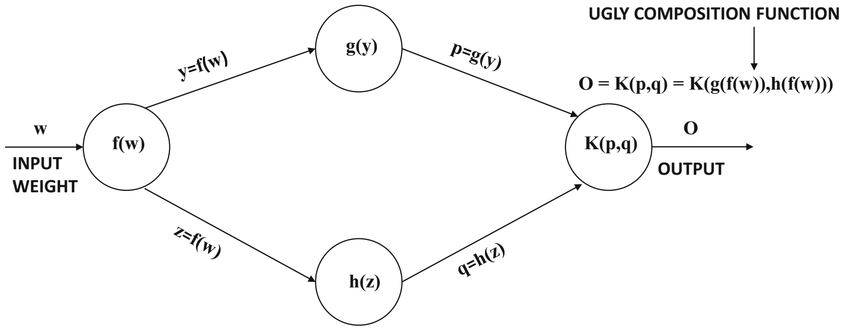
with access to these building blocks. The analyst is able to specify the number and type of units in each layer along with an off-the-shelf or customized loss function. A deep neural network containing tens of layers can often be described in a few hundred lines of code. All the learning of the weights 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:

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

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:

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

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*.

1.4.1 The Problem of Overfitting

The problem of overfitting refers to the fact that fitting a model to a particular training data set does not guarantee that it will provide good prediction performance on unseen test data, even if the model predicts the targets on the training data perfectly. In other words, there is always a gap between the training and test data performance, which is particularly large when the models are complex and the data set is small.

In order to understand this point, consider a simple single-layer neural network on a data set with five attributes, where we use the identity activation to learn a real-valued target variable. This architecture is almost identical to that of Figure 1.3, except that the identity activation function is used in order to predict a real-valued target. Therefore, the network tries to learn the following function:

$$\hat{y} = \sum_{i=1}^{5} w_i \cdot x_i \tag{1.32}$$

Consider a situation in which the observed target value is real and is always twice the value of the first attribute, whereas other attributes are completely unrelated to the target. However, we have only four training instances, which is one less than the number of features (free parameters). For example, the training instances could be as follows:

x_1	x_2	x_3	x_4	x_5	y
1	1	0	0	0	2
2	0	1	0	0	4
3	0	0	1	0	6
4	0	0	0	1	8

The correct parameter vector in this case is $\overline{W} = [2, 0, 0, 0, 0]$ based on the known relationship between the first feature and target. The training data also provides zero error with this solution, although the relationship needs to be *learned* from the given instances since it is not given to us a priori. However, the problem is that the number of training points is fewer than the number of parameters and it is possible to find an infinite number of solutions with zero error. For example, the parameter set $[0, 2, 4, 6, 8]$ also provides zero error *on the training data*. However, if we used this solution on unseen test data, it is likely to provide very poor performance because the learned parameters are spuriously inferred and are unlikely to *generalize* well to new points in which the target is twice the first attribute (and other attributes are random). This type of spurious inference is caused by the paucity of training data, where random nuances are encoded into the model. As a result, the solution does not generalize well to unseen test data. This situation is almost similar to learning by rote, which is highly predictive for training data but not predictive for unseen test data. Increasing the number of training instances improves the generalization power of the model, whereas increasing the complexity of the model reduces its generalization power. At the same time, when a lot of training data is available, an overly simple model is unlikely to capture complex relationships between the features and target. A good rule of thumb is that the total number of training data points should be at least 2 to 3 times larger than the number of parameters in the neural network, although the precise number of data instances depends on the specific model at hand. In general, models with a larger number of parameters are said to have *high capacity*, and they require a larger amount of data in order to gain generalization power to unseen test data. The notion of overfitting is often understood in the trade-off between *bias* and *variance* in machine learning. The key take-away from the notion of bias-variance trade-off is that one does not always win with more powerful (i.e., less *biased*) models when working with limited training data, because of the higher *variance* of these models. For example, if we change the training data in the table above to a different set of four points, we are likely to learn a completely different set of parameters (from the random nuances of those points). This new model is likely to yield a completely different prediction *on the same test instance* as compared to the predictions using the first training data set. This type of variation in the prediction of the same test instance using different training data sets is a manifestation of *model variance*, which also adds to the error of the model; after all, both predictions of the same test instance could not possibly be correct. More complex models have the drawback of seeing spurious patterns in random nuances, especially when the training data are insufficient. One must be careful to pick an optimum point when deciding the complexity of the model. These notions are described in detail in Chapter 4.

Neural networks have always been known to theoretically be powerful enough to approximate any function [208]. However, the lack of data availability can result in poor performance; this is one of the reasons that neural networks only recently achieved prominence. The greater availability of data has revealed the advantages of neural networks over traditional machine learning (cf. Figure 1.2). In general, neural networks require careful design to minimize the harmful effects of overfitting, even when a large amount of data is available. This section provides an overview of some of the design methods used to mitigate the impact of overfitting.

1.4.1.1 Regularization

Since a larger number of parameters causes overfitting, a natural approach is to constrain the model to use fewer non-zero parameters. In the previous example, if we constrain the vector $\overline{W}$ to have only one non-zero component out of five components, it will correctly obtain the solution $[2, 0, 0, 0, 0]$. Smaller absolute values of the parameters also tend to overfit less. Since it is hard to constrain the values of the parameters, the softer approach of adding the penalty $\lambda ||\overline{W}||^p$ to the loss function is used. The value of p is typically set to 2, which leads to *Tikhonov regularization*. In general, the squared value of each parameter (multiplied with the regularization parameter $\lambda > 0$) is added to the objective function. The practical effect of this change is that a quantity proportional to λw_i is subtracted from the update of the parameter w_i. An example of a regularized version of Equation 1.6 for mini-batch S and update step-size $\alpha > 0$ is as follows:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \sum_{\overline{X} \in S} E(\overline{X})\overline{X} \tag{1.33}$$

Here, $E[\overline{X}]$ represents the current error $(y - \hat{y})$ between observed and predicted values of training instance $\overline{X}$. One can view this type of penalization as a kind of weight decay during the updates. Regularization is particularly important when the amount of available data is limited. A neat biological interpretation of regularization is that it corresponds to gradual forgetting, as a result of which "less important" (i.e., *noisy*) patterns are removed. In general, it is often advisable to use more complex models with regularization rather than simpler models without regularization.

As a side note, the general form of Equation 1.33 is used by many regularized machine learning models like *least-squares regression* (cf. Chapter 2), where $E(\overline{X})$ is replaced by the error-function of that specific model. Interestingly, weight decay is only sparingly used in the single-layer perceptron[3] because it can sometimes cause overly rapid forgetting with a small number of recently misclassified training points dominating the weight vector; the main issue is that the perceptron criterion is already a degenerate loss function with a minimum value of 0 at $\overline{W} = 0$ (unlike its hinge-loss or least-squares cousins). This quirk is a legacy of the fact that the single-layer perceptron was originally defined in terms of biologically inspired updates rather than in terms of carefully thought-out loss functions. Convergence to an optimal solution was never guaranteed other than in linearly separable cases. For the single-layer perceptron, some other regularization techniques, which are discussed below, are more commonly used.

1.4.1.2 Neural Architecture and Parameter Sharing

The most effective way of building a neural network is by constructing the architecture of the neural network after giving some thought to the underlying data domain. For example, the successive words in a sentence are often related to one another, whereas the nearby pixels in an image are typically related. These types of insights are used to create specialized architectures for text and image data with fewer parameters. Furthermore, many of the parameters might be shared. For example, a convolutional neural network uses the same set of parameters to learn the characteristics of a local block of the image. The recent advancements in the use of neural networks like *recurrent neural networks* and *convolutional neural networks* are examples of this phenomena.

1.4.1.3 Early Stopping

Another common form of regularization is *early stopping*, in which the gradient descent is ended after only a few iterations. One way to decide the stopping point is by holding out a part of the training data, and then testing the error of the model on the held-out set. The gradient-descent approach is terminated when the error on the held-out set begins to rise. Early stopping essentially reduces the size of the parameter space to a smaller neighborhood within the initial values of the parameters. From this point of view, early stopping acts as a regularizer because it effectively restricts the parameter space.

1.4.1.4 Trading Off Breadth for Depth

As discussed earlier, a two-layer neural network can be used as a universal function approximator [208], if a large number of hidden units are used within the hidden layer. It turns out that networks with more layers (i.e., greater *depth*) tend to require far fewer units per layer because the composition functions created by successive layers make the neural network more powerful. Increased depth is a form of regularization, as the features in later layers are forced to obey a particular type of structure imposed by the earlier layers. Increased constraints reduce the capacity of the network, which is helpful when there are limitations on the amount of available data. A brief explanation of this type of behavior is given in Section 1.5. The number of units in each layer can typically be reduced to such an extent that a deep network often has far fewer parameters even when added up over the greater number of layers. This observation has led to an explosion in research on the topic of *deep learning*.

[3] Weight decay is generally used with other loss functions in single-layer models and in all multi-layer models with a large number of parameters.

Even though deep networks have fewer problems with respect to overfitting, they come with a different family of problems associated with ease of training. In particular, the loss derivatives with respect to the weights in different layers of the network tend to have vastly different magnitudes, which causes challenges in properly choosing step sizes. Different manifestations of this undesirable behavior are referred to as the *vanishing* and *exploding gradient* problems. Furthermore, deep networks often take unreasonably long to converge. These issues and design choices will be discussed later in this section and at several places throughout the book.

1.4.1.5 Ensemble Methods

A variety of ensemble methods like *bagging* are used in order to increase the generalization power of the model. These methods are applicable not just to neural networks but to any type of machine learning algorithm. However, in recent years, a number of ensemble methods that are specifically focused on neural networks have also been proposed. Two such methods include *Dropout* and *Dropconnect*. These methods can be combined with many neural network architectures to obtain an additional accuracy improvement of about 2% in many real settings. However, the precise improvement depends to the type of data and the nature of the underlying training. For example, normalizing the activations in hidden layers can reduce the effectiveness of *Dropout* methods, although one can gain from the normalization itself. Ensemble methods are discussed in Chapter 4.

1.4.2 The Vanishing and Exploding Gradient Problems

While increasing depth often reduces the number of parameters of the network, it leads to different types of practical issues. Propagating backwards using the chain rule has its drawbacks in networks with a large number of layers in terms of the stability of the updates. In particular, the updates in earlier layers can either be negligibly small (vanishing gradient) or they can be increasingly large (exploding gradient) in certain types of neural network architectures. This is primarily caused by the chain-like product computation in Equation 1.23, which can either exponentially increase or decay over the length of the path. In order to understand this point, consider a situation in which we have a multi-layer network with one neuron in each layer. Each local derivative along a path can be shown to be the product of the weight and the derivative of the activation function. The overall backpropagated derivative is the product of these values. If each such value is randomly distributed, and has an expected value less than 1, the product of these derivatives in Equation 1.23 will drop off exponentially fast with path length. If the individual values on the path have expected values greater than 1, it will typically cause the gradient to explode. Even if the local derivatives are randomly distributed with an expected value of exactly 1, the overall derivative will typically show instability depending on how the values are actually distributed. In other words, the vanishing and exploding gradient problems are rather natural to deep networks, which makes their training process unstable.

Many solutions have been proposed to address this issue. For example, a sigmoid activation often encourages the vanishing gradient problem, because its derivative is less than 0.25 at all values of its argument (see Exercise 7), and is extremely small at saturation. A ReLU activation unit is known to be less likely to create a vanishing gradient problem because its derivative is always 1 for positive values of the argument. More discussions on this issue are provided in Chapter 3. Aside from the use of the ReLU, a whole host of gradient-descent tricks are used to improve the convergence behavior of the problem. In particular, the use of *adaptive learning rates* and *conjugate gradient methods* can help in many cases. Furthermore, a recent technique called *batch normalization* is helpful in addressing some of these issues. These techniques are discussed in Chapter 3.

1.4.3 Difficulties in Convergence

Sufficiently fast convergence of the optimization process is difficult to achieve with very deep networks, as depth leads to increased resistance to the training process in terms of letting the gradients smoothly flow through the network. This problem is somewhat related to the vanishing gradient problem, but has its own unique characteristics. Therefore, some "tricks" have been proposed in the literature for these cases, including the use of *gating networks* and *residual networks* [184]. These methods are discussed in Chapters 7 and 8, respectively.

1.4.4 Local and Spurious Optima

The optimization function of a neural network is highly nonlinear, which has lots of local optima. When the parameter space is large, and there are many local optima, it makes sense to spend some effort in picking good initialization points. One such method for improving neural network initialization is referred to as *pretraining.* The basic idea is to use either supervised or unsupervised training on *shallow sub-networks* of the original network in order to create the initial weights. This type of pretraining is done in a *greedy and layerwise fashion* in which a single layer of the network is trained at one time in order to learn the initialization points of that layer. This type of approach provides initialization points that ignore drastically irrelevant parts of the parameter space to begin with. Furthermore, unsupervised pretraining often tends to avoid problems associated with overfitting. The basic idea here is that some of the minima in the loss function are spurious optima because they are exhibited only in the training data and not in the test data. Using unsupervised pretraining tends to move the initialization point closer to the basin of "good" optima in the test data. This is an issue associated with model generalization. Methods for pretraining are discussed in Section 4.7 of Chapter 4.

Interestingly, the notion of spurious optima is often viewed from the lens of model generalization in neural networks. This is a different perspective from traditional optimization. In traditional optimization, one does not focus on the differences in the loss functions of the training and test data, but on the shape of the loss function in only the training data. Surprisingly, the problem of local optima (from a traditional perspective) is a smaller issue in neural networks than one might normally expect from such a nonlinear function. Most of the time, the nonlinearity causes problems during the training process itself (e.g., failure to converge), rather than getting stuck in a local minimum.

1.4.5 Computational Challenges

A significant challenge in neural network design is the running time required to train the network. It is not uncommon to require weeks to train neural networks in the text and image domains. In recent years, advances in hardware technology such as Graphics Processor Units (GPUs) have helped to a significant extent. GPUs are specialized hardware processors that can significantly speed up the kinds of operations commonly used in neural networks. In this sense, some algorithmic frameworks like *Torch* are particularly convenient because they have GPU support tightly integrated into the platform.

Although algorithmic advancements have played a role in the recent excitement around deep learning, a lot of the gains have come from the fact that the same algorithms can do much more on modern hardware. Faster hardware also supports algorithmic development, because one needs to repeatedly test computationally intensive algorithms to understand what works and what does not. For example, a recent neural model such as the long short-term memory has changed only modestly [150] since it was first proposed in 1997 [204]. Yet, the potential of this model has been recognized only recently because of the advances in computational power of modern machines and algorithmic tweaks associated with improved experimentation.

One convenient property of the vast majority of neural network models is that most of the computational heavy lifting is *front loaded* during the training phase, and the prediction phase is often computationally efficient, because it requires a small number of operations (depending on the number of layers). This is important because the prediction phase is often far more time-critical compared to the training phase. For example, it is far more important to classify an image in real time (with a pre-built model), although the actual building of that model might have required a few weeks over millions of images. Methods have also been designed to compress trained networks in order to enable their deployment in mobile and space-constrained settings. These issues are discussed in Chapter 3.

1.5 The Secrets to the Power of Function Composition

Even though the biological metaphor sounds like an exciting way to intuitively justify the computational power of a neural network, it does not provide a complete picture of the settings in which neural networks perform well. At its most basic level, a neural network is a computational graph that performs compositions of simpler functions to provide a more complex function. Much of the power of deep learning arises from the fact that *repeated* composition of multiple nonlinear functions has significant expressive power. Even though the work in [208] shows that the single composition of a large number of squashing functions can approximate almost any function, this approach will require an extremely large number of units (i.e., parameters) of the network. This increases the capacity of the network, which causes overfitting unless the data set is extremely large. Much of the power of deep learning arises from the fact that the *repeated composition of certain types of functions increases the representation power of the network, and therefore reduces the parameter space required for learning.*

Not all base functions are equally good at achieving this goal. In fact, the nonlinear squashing functions used in neural networks are not arbitrarily chosen, but are carefully designed because of certain types of properties. For example, imagine a situation in which the identity activation function is used in each layer, so that only linear functions are computed. In such a case, the resulting neural network is no stronger than a single-layer, linear network:

Theorem 1.5.1 *A multi-layer network that uses only the identity activation function in all its layers reduces to a single-layer network performing linear regression.*

Proof: Consider a network containing k hidden layers, and therefore contains a total of $(k+1)$ computational layers (including the output layer). The corresponding $(k+1)$ weight matrices between successive layers are denoted by $W_1 \ldots W_{k+1}$. Let $\overline{x}$ be the d-dimensional column vector corresponding to the input, $\overline{h}_1 \ldots \overline{h}_k$ be the column vectors corresponding to the hidden layers, and $\overline{o}$ be the m-dimensional column vector corresponding to the output.

Then, we have the following recurrence condition for multi-layer networks:

$$\begin{aligned}
\overline{h}_1 &= \Phi(W_1^T \overline{x}) = W_1^T \overline{x} \\
\overline{h}_{p+1} &= \Phi(W_{p+1}^T \overline{h}_p) = W_{p+1}^T \overline{h}_p \quad \forall p \in \{1 \ldots k-1\} \\
\overline{o} &= \Phi(W_{k+1}^T \overline{h}_k) = W_{k+1}^T \overline{h}_k
\end{aligned}$$

In all the cases above, the activation function $\Phi(\cdot)$ has been set to the identity function. Then, by eliminating the hidden layer variables, it is easy to show the following:

$$\begin{aligned}
\overline{o} &= W_{k+1}^T W_k^T \ldots W_1^T \overline{x} \\
&= \underbrace{(W_1 W_2 \ldots W_{k+1})^T}_{W_{xo}^T} \overline{x}
\end{aligned}$$

Note that one can replace the matrix $W_1 W_2 \ldots W_{k+1}$ with the new $d \times m$ matrix W_{xo}, and learn the coefficients of W_{xo} instead of those of all the matrices W_1, $W_2 \ldots W_{k+1}$, without loss of expressivity. In other words, we have the following:

$$\overline{o} = W_{xo}^T \overline{x}$$

However, this condition is exactly identical to that of linear regression with multiple outputs [6]. In fact, it is a bad idea to learn the redundant matrices $W_1 \ldots W_{k+1}$ instead of W_{xo}, because doing so increases the number of parameters to be learned without increasing the power of the model in any way. Therefore, a multilayer neural network with identity activations does not gain over a single-layer network in terms of expressivity. ■

The aforementioned result is for the case of regression modeling with numeric target variables. A similar result holds true for binary target variables. In the special case, where all layers use identity activation and the final layer uses a single output with sign activation for prediction, the multilayer neural network reduces to the perceptron.

Lemma 1.5.1 *Consider a multilayer network in which all hidden layers use identity activation and the single output node uses the perceptron criterion as the loss function and the sign activation for prediction. This neural network reduces to the single-layer perceptron.*

The proof of this result is almost identical to that of the one discussed above. In fact, as long as the hidden layers are linear, nothing is gained using the additional layers.

This result shows that deep networks largely make sense only when the activation functions in intermediate layers are non-linear. Typically, the functions like sigmoid and tanh are *squashing* functions in which the output is bounded within an interval, and the gradients are largest near zero values. For large absolute values of their arguments, these functions are said to reach *saturation* where increasing the absolute value of the argument further does not change its value significantly. This type of function in which values do not vary significantly at large absolute values of their arguments is shared by another family of functions, referred to as *Gaussian kernels*, which are commonly used in non-parametric density estimation:

$$\Phi(v) = \exp(-v^2/2) \tag{1.34}$$

The only difference is that Gaussian kernels saturate to 0 at large values of their argument, whereas functions like sigmoid and tanh can also saturate to values of +1 and −1. It is well known in the literature on density estimation [451] that the sum of many small Gaussian kernels can be used to approximate any density function. Density functions have a special nonnegative structure in which extremes of the data distribution always saturate to zero density, and therefore the underlying kernels also show the same behavior. A similar principle holds true (more generally) for squashing functions in which the linear combination of many small activation functions can be used to approximate an arbitrary function; however, squashing functions do not saturate to zero in order to handle arbitrary behavior at extreme values. The universal approximation result of neural networks [208] posits that a linear combination of sigmoid units (and/or most other reasonable squashing functions) in a single hidden layer can be used to approximate any function well. Note that the linear combination can be performed by a single output node. Therefore, a two-layer network is sufficient as long as the number of hidden units is large enough. However, some kind of basic non-linearity in the activation function is always required in order to model the turns and twists in an arbitrary function. To understand this point, note that all 1-dimensional functions can be approximated as a sum of scaled/translated step functions and most of the activation functions discussed in this chapter (e.g., sigmoid) look awfully like step functions (see Figure 1.8). This basic idea is the essence of the universal approximation theorem of neural networks. In fact, the proof of the ability of squashing functions to approximate any function is conceptually similar to that of kernels at least at an intuitive level. However, the number of base functions required to reach a high level of approximation can be extremely large in both cases, potentially increasing the data-centric requirements to an unmanageable level. For this reason, shallow networks face the persistent problem of overfitting. The universal approximation theorem asserts the ability to well-approximate the function implicit in the training data, but makes no guarantee about whether the function can be generalized to unseen test data.

1.5.1 The Importance of Nonlinear Activation

The previous section provides a concrete proof of the fact that a neural network with only linear activations does not gain from increasing the number of layers in it. For example, consider the two-class data set illustrated in Figure 1.14, which is represented in two dimensions denoted by x_1 and x_2. There are two instances, A and B, of the class denoted by '*' with coordinates $(1, 1)$ and $(-1, 1)$, respectively. There is also a single instance B of the class denoted by '+' with coordinates $(0, 1)$, A neural network with only linear activations will never be able to classify the training data perfectly because the points are not linearly separable.

On the other hand, consider a situation in which the hidden units have ReLU activation, and they learn the two new features h_1 and h_2, which are as follows:

$$h_1 = \max\{x_1, 0\}$$
$$h_2 = \max\{-x_1, 0\}$$

Note that these goals can be achieved by using appropriate weights from the input to hidden layer, and also applying a ReLU activation unit. The latter achieves the goal of thresholding negative values to 0. We have indicated the corresponding weights in the neural network shown in Figure 1.14. We have shown a plot of the data in terms of h_1 and h_2 in the same figure. The coordinates of the three points in the 2-dimensional hidden layer are $\{(1, 0), (0, 1), (0, 0)\}$. It is immediately evident that the two classes become linearly separable in terms of the new hidden representation. In a sense, the task of the first layer was *representation learning* to enable the solution of the problem with a linear classifier. Therefore, if we add a single linear output layer to the neural network, it will be able to

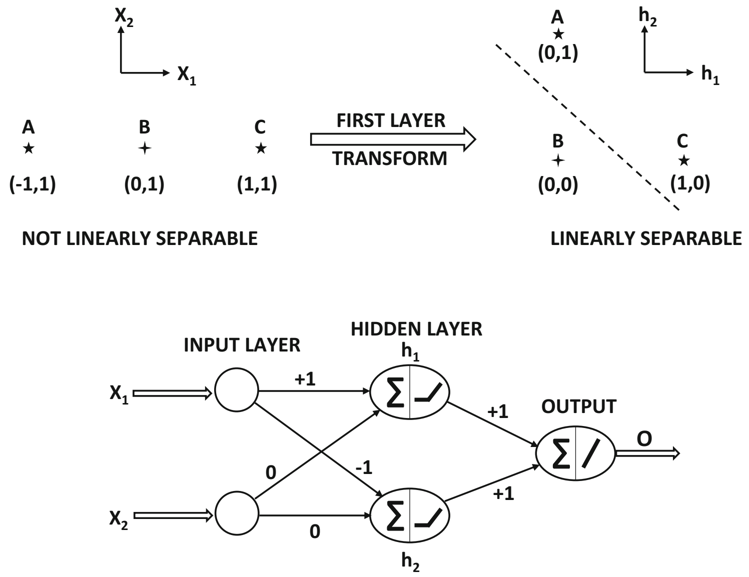


Figure 1.14: The power of nonlinear activation functions in transforming a data set to linear separability

classify these training instances perfectly. The key point is that the use of the nonlinear ReLU function is crucial in ensuring this linear separability. *Activation functions enable nonlinear mappings of the data, so that the embedded points can become linearly separable.* In fact, if both the weights from hidden to output layer are set to 1 with a linear activation function, the output O will be defined as follows:

$$O = h_1 + h_2 \tag{1.35}$$

This simple linear function separates the two classes because it always takes on the value of 1 for the two points labeled '*' and takes on 0 for the point labeled '+'. Therefore, much of the power of neural networks is hidden in the use of activation functions. The weights shown in Figure 1.14 will need to be *learned* in a data-driven manner, although

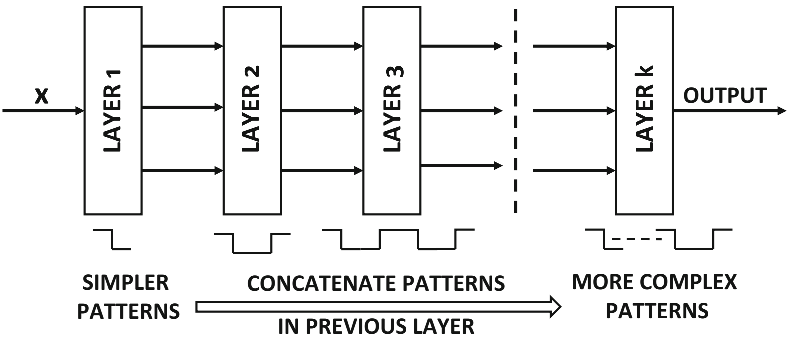


Figure 1.15: Deeper networks can learn more complex functions by composing the functions learned in earlier layers.

there are many alternative choices of weights that can make the hidden representation linearly separable. Therefore, the learned weights may be different than the ones shown in Figure 1.14 if actual training is performed. Nevertheless, in the case of the perceptron, there is no choice of weights at which one could hope to classify this training data set perfectly because the data set is not linearly separable in the original space. In other words, the activation functions enable nonlinear transformations of the data, that become increasingly powerful with multiple layers. A sequence of nonlinear activations imposes a specific type of structure on the learned model, whose power increases with the depth of the sequence (i.e., number of layers in the neural network).

Another classical example is the XOR function in which the two points $\{(0,0),(1,1)\}$ belong to one class, and the other two points $\{(1,0),(0,1)\}$ belong to the other class. It is possible to use ReLU activation to separate these two classes as well, although bias neurons will be needed in this case (see Exercise 1). The original backpropagation paper [409] discusses the XOR function, because this function was one of the motivating factors for designing multilayer networks and the ability to train them. The XOR function is considered a litmus test to determine the basic feasibility of a particular family of neural networks to properly predict nonlinearly separable classes. Although we have used the ReLU activation function above for simplicity, it is possible to use most of the other nonlinear activation functions to achieve the same goals.

1.5.2 Reducing Parameter Requirements with Depth

The basic idea of deep learning is that repeated composition of functions can often reduce the requirements on the number of base functions (computational units) by a factor that is exponentially related to the number of layers in the network. Therefore, even though the number of layers in the network increases, the number of parameters required to approximate the same function reduces drastically. This increases the generalization power of the network.

The idea behind deeper architectures is that they can better leverage *repeated regularities* in the data patterns in order to reduce the number of computational units and therefore *generalize* the learning even to areas of the data space where one does not have examples. Often these repeated regularities are learned by the neural network within the weights as the basis vectors of *hierarchical features*. Although a detailed proof [340] of this fact is beyond the scope of this book, we provide a simple example to elucidate this point. Consider a situation in which a 1-dimensional function is defined by 1024 repeated steps of the same size and height. A shallow network with one hidden layer and step activation functions would require at least 1024 units in order to model the function. However, a multilayer network would model a pattern of 1 step in the first layer, 2 steps in the next, 4 steps in the third, and 2^r steps in the rth layer. This situation is illustrated in Figure 1.15. Note that the pattern of 1 step is the simplest feature because it is repeated 1024 times, whereas a pattern of 2 steps is more complex. Therefore, the features (and the functions learned) in successive layers are hierarchically related. In this case, a total of 10 layers are required and a small number of constant nodes are required in each layer to model the joining of the two patterns from the previous layer.

Another way to understand this point is as follows. Consider a 1-dimensional function which takes one the value of 1 and -1 in alternate intervals, and this value switches 1024 times at regular intervals of the argument. The only way to simulate this function with a linear combination of step activation functions (containing only one switch in value) is to use 1024 of them (or a small constant factor of this number). However, a neural network with 10 hidden layers and only 2 units per layer has $2^{10} = 1024$ paths from the source to the output. As long as the function to be learned is regular in some way, it is often possible to learn parameters for the layers so that these 1024 paths are able to capture the complexity of 1024 different value switches in the function. The earlier layers learn more detailed patterns, whereas the later layers learn higher-level patterns. Therefore, the overall number of nodes required is an *order of magnitude less* than that required in the single-layer network. This means that the amount of data required for learning is also an order of magnitude less. The reason for this is that the multilayer network implicitly *looks for the repeated regularities and learns them* with less data, rather than trying to explicitly learn every turn and twist of the target function. When using convolutional neural networks with image data, this behavior becomes intuitively obvious in which earlier layers model simple features like lines, a middle layer might model elementary shapes, and a later layer might model a complex shape like a face. On the other hand, a single layer would have difficulty in modeling every twist and turn of a face. This provides the deeper model with better generalization power and also the ability to learn with less data.

However, increasing the depth of the network is not without its disadvantages. Deeper networks are often harder to train, and they show all types of unstable behavior such as the vanishing and exploding gradient problems. Deep networks are also notoriously unstable to parameter choice. These issues are often addressed with careful design of the functions computed within nodes, as well as the use of pretraining procedures to improve performance.

1.5.3 Unconventional Neural Architectures

The aforementioned discussion provides an overview of the most common ways in which the operations and structures of typical neural networks are constructed. However, there are many variations of this common theme. The following will discuss some of these variations.

1.5.3.1 Blurring the Distinctions Between Input, Hidden, and Output Layers

In general, there is a heavy emphasis on layer-wise feed-forward networks in the neural network domain with a sequential arrangement between input, hidden, and output layers. In other words, all input nodes feed into the first hidden layer, the hidden layers successively feed into one another, and the final hidden layer feeds into the output layer. The compu-

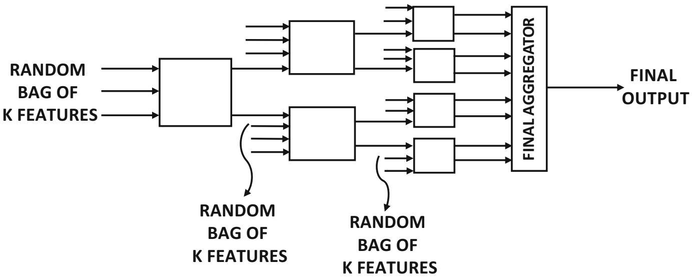


Figure 1.16: An example of an unconventional architecture in which inputs occur to layers other than the first hidden layer. As long as the neural network is acyclic (or can be transformed into an acyclic representation), the weights of the underlying computation graph can be learned using dynamic programming (backpropagation).

tational units are often defined by squashing functions applied to linear combinations of input. The hidden layer generally does not take inputs, and the loss is generally not computed over the values in the hidden layers. Because of this focus, it is easy to forget that a neural network can be defined as *any type of parameterized computation graph*, where these restrictions are not necessary for the backpropagation algorithm to work. In general, it is possible to have input and loss computation in intermediate layers, although this is less common. For example, a neural network is proposed in [515] that is inspired by the notion of a *random forest* [49], and it allows input in different layers of the network. An example of this type of network is shown in Figure 1.16. In this case, it is clear that the distinction between the input layers and the hidden layers has been blurred.

In other variations of the basic feed-forward architecture, loss functions are computed not just at the output nodes, but also at the hidden nodes. The contributions at the hidden nodes are often in the form of *penalties* that act as regularizers. For example, these types of methods are used to perform sparse feature learning by imposing penalties on the hidden nodes (cf. Chapters 2 and 4). In this case, the distinction between the hidden layers and output layers is blurred.

Another recent example of a design choice is the use of *skip connections* [184] in which the inputs from a particular layer are allowed to connect to layers beyond the immediate next layer. This type of approach leads to truly deep models. For example, a 152-layer architecture, referred to as *ResNet* [184], has reached human-level performance in the image recognition task. Although this architecture does not blur the distinction between input, hidden, and output layers, its structure differs from a traditional feed-forward network in which connections are placed only between successive layers. These networks have an *iterative view* of feature engineering [161], in which the features in later layers are iterative refinements of those in previous layers. In contrast, the traditional approach to feature engineering is *hierarchical*, in which features in later layers are increasingly abstract representations obtained from those in previous layers.

1.5.3.2 Unconventional Operations and Sum-Product Networks

Some neural networks like *long short-term memory* and convolutional neural networks define various types of multiplicative "forgetting," convolution, and pooling operations between variables that are not strictly in any of the forms discussed in this chapter. In fact, these architectures are now used so heavily in the text and image domains that they are no longer considered unusual.

Another unique type of architecture is the *sum-product network* [383]. In this case, the nodes are either summation nodes or product nodes. Summation nodes are similar to the traditional linear transformation with a set of weighted edges. However, the weights are constrained to be positive. The product nodes simply multiply its inputs without the need for weights. It is noteworthy that there are many variations in terms of how products can be computed. For example, if the inputs are two scalars, then one can simply compute their product. If the inputs are two vectors of equal length, one can compute their element-wise product. Several deep learning libraries do support these types of product operations. It is natural for the summation layers and the product layers to alternate in order to maximize expressivity.

Sum-product networks are quite expressive, and it is often possible to build deep variations with a high level of expressivity [30, 93]. A key point is that almost any mathematical function can be approximately written as a polynomial function of its inputs. Therefore, almost any function can be expressed using the sum-product architecture, although deeper architectures allow modeling with greater structure. Unlike traditional neural networks in which nonlinearity is incorporated with activation functions, the product operation is the key to nonlinearity in the sum-product network.

Training Issues

It is often helpful to be flexible in using different types of computational operations within the nodes beyond the known transformations and activation functions. Furthermore, the connections between nodes need not be structured in layer-wise fashion and nodes in the hidden layer can be included in the loss computation. As long as the underlying computational graph is acyclic, it is easy to generalize the backpropagation algorithm to any type of architecture and computational operation. After all, a dynamic programming algorithm (like backpropagation) can be used on virtually any type of directed acyclic graph in which multiple nodes can be used for initializing the dynamic programming recursion. It is important to keep in mind that architectures that are designed with a proper domain-specific understanding can often provide superior results to black-box methods that use fully connected feed-forward networks.

1.6 Common Neural Architectures

There are several types of neural architectures that are used commonly in various machine learning applications. This section will provide a brief overview of some of these architectures, which will be discussed in greater detail in later chapters.

1.6.1 Simulating Basic Machine Learning with Shallow Models

Most of the basic machine learning models like linear regression, classification, support vector machines, logistic regression, singular value decomposition, and matrix factorization can be simulated with shallow neural networks containing no more than one or two layers. It is instructive to explore these basic architectures, because it indirectly showcases the power of neural networks; most of what we know about machine learning can be simulated with relatively simple models! Furthermore, many basic neural network models like the *Widrow-Hoff learning model* are directly related to traditional machine learning models like the Fisher's discriminant, even though they were proposed independently. A noteworthy observation is that deeper architectures are often created by stacking these simpler models in a creative way. The neural architectures for basic machine learning models are discussed in Chapter 2. A number of applications to text mining, graphs, and recommender systems will also be discussed in this chapter.

1.6.2 Radial Basis Function Networks

Radial basis function (RBF) networks represent the forgotten architecture from the rich history of neural networks. They are not commonly used in the modern era, although they do have significant potential for specific types of problems. One limiting issue is that these networks are not deep, and they typically use only two layers. The first layer is constructed in an unsupervised way, whereas the second layer is trained using supervised methods. These networks are fundamentally different from feed-forward networks, and gain their power from the larger number of nodes in the unsupervised layer. The basic principles of using RBF networks are fundamentally very different from those of feed-forward networks, in the sense that the former gains its power from expanding the size of the feature space rather than depth. This approach is based on *Cover's theorem on separability of patterns* [84], which states that pattern classification problems are more likely to be linearly separable when cast into a high-dimensional space with a nonlinear transformation. The second layer of the network contains a prototype in each node and the activation is defined by the similarity of the input data to the prototype. These activations are then combined with trained weights of the next layer to create a final prediction. This approach is very similar to that of nearest-neighbor classifiers, except that the weights in the second layer provide an additional level of supervision. In other words, the approach is a *supervised* nearest-neighbor method.

Notably, support vector machines are known to be supervised variants of nearest-neighbor classifiers in which a kernel function is combined with supervised weights to weight the neighboring points in the final prediction [6]. Radial basis function networks can be used to simulate kernel methods like support vector machines. For specific types of problems like classification, one can use these architectures more effectively than an off-the-shelf kernel support vector machine. This is because these models are more general, providing more opportunities for experimentation than a kernel support vector machine. Furthermore, it is sometimes possible to gain some advantages from increased depth in the supervised layers. The full potential of radial basis function networks remains unexplored in the literature, because this architecture has largely been forgotten with the increased focus on vanilla feed-forward methods. A discussion of radial basis function networks is provided in Chapter 5.

1.6.3 Restricted Boltzmann Machines

Restricted Boltzmann machines (RBMs) use the notion of energy minimization in order to create neural network architectures for modeling data in an unsupervised way. These methods are particularly useful for creating generative models of the data, and they are closely related to probabilistic graphical models [251]. Restricted Boltzmann machines owe their origins to the use of *Hopfield networks* [207], which can be used to store memories. Stochastic variants of these networks were generalized to *Boltzmann machines*, in which hidden layers modeled generative aspects of the data.

Restricted Boltzmann machines are often used for unsupervised modeling and dimensionality reduction, although they can also be used for supervised modeling. However, since they were not naturally suited to supervised modeling, the supervised training was often preceded by an unsupervised phase. This naturally led to the discovery of the notion of pretraining, which was found to be extremely beneficial for supervised learning. RBMs were among the first models that were used for deep learning, especially in the unsupervised setting. The pretraining approach was eventually adopted by other types of models. Therefore, RBMs also have a historical significance in terms of motivating some training methodologies for deep models.

The training process of a restricted Boltzmann machine is quite different from that of a feed-forward network. In particular, these models cannot be trained using backpropagation, and they require Monte Carlo sampling in order to perform the training. The particular algorithm that is used commonly for training an RBM is the *contrastive divergence algorithm*. A discussion of restricted Boltzmann machines is provided in Chapter 6.

1.6.4 Recurrent Neural Networks

Recurrent neural networks are designed for sequential data like text sentences, time-series, and other discrete sequences like biological sequences. In these cases, the input is of the form $\overline{x}_1 \ldots \overline{x}_n$, where $\overline{x}_t$ is a d-dimensional point received at the time-stamp t. For example, the vector $\overline{x}_t$ might contain the d values at the tth tick of a multivariate time-series (with d different series). In a text-setting, the vector $\overline{x}_t$ will contain the *one-hot encoded* word at the tth time-stamp. In one-hot encoding, we have a vector of length equal to the lexicon size, and the component for the relevant word has a value of 1. All other components are 0.

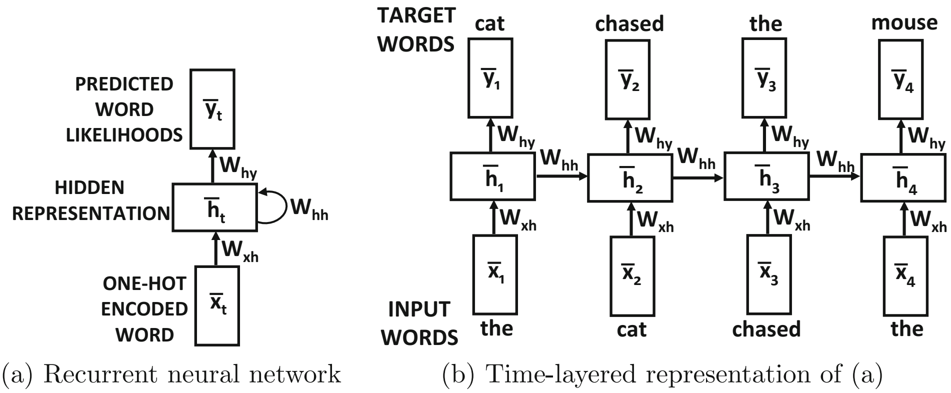
An important point about sequences is that successive words are dependent on one another. Therefore, it is helpful to receive a particular input $\overline{x}_t$ only *after* the earlier inputs have already been received and converted into a hidden state. The traditional type of feed-forward network in which all inputs feed into the first layer does not achieve this goal. Therefore, the recurrent neural network allows the input $\overline{x}_t$ to interact directly with the hidden state created from the inputs at previous time stamps. The basic architecture of the recurrent neural network is illustrated in Figure 1.17(a). The key point is that there is an input $\overline{x}_t$ at each time-stamp, and a hidden state $\overline{h}_t$ that changes at each time stamp as new data points arrive. Each time-stamp also has an output value $\overline{y}_t$. For example, in a time-series setting, the output $\overline{y}_t$ might be the forecasted prediction of $\overline{x}_{t+1}$. When used in the text-setting of predicting the next word, this approach is referred to as *language modeling*. In some applications, we do not output $\overline{y}_t$ at each time stamp, but only at the end of the sequence. For example, if one is trying the classify the sentiment of a sentence as "*positive*" or "*negative*," the output will occur only at the final time stamp.

The hidden state at time t is given by a function of the input vector at time t and the hidden vector at time $(t-1)$:

$$\overline{h}_t = f(\overline{h}_{t-1}, \overline{x}_t) \tag{1.36}$$

A separate function $\overline{y}_t = g(\overline{h}_t)$ is used to learn the output probabilities from the hidden states. Note that the functions $f(\cdot)$ and $g(\cdot)$ are the same at each time stamp. The implicit assumption is that the time-series exhibits a certain level of *stationarity*; the underlying properties do not change with time. Although this property is not exactly true in real settings, it is a good assumption to use for regularization.

A key point here is the presence of the self-loop in Figure 1.17(a), which will cause the hidden state of the neural network to change after the input of each $\overline{x}_t$. In practice, one only works with sequences of finite length, and it makes sense to unfurl the loop into a "time-layered" network that looks more like a feed-forward network. This network is shown in Figure 1.17(b). Note that in this case, we have a different node for the hidden



(a) Recurrent neural network (b) Time-layered representation of (a)

Figure 1.17: A recurrent neural network and its time-layered representation

state at each time-stamp and the self-loop has been unfurled into a feed-forward network. This representation is mathematically equivalent to Figure 1.17(a), but is much easier to comprehend because of its similarity to a traditional network. Note that unlike traditional feed-forward networks, the inputs also occur to intermediate layers in this unfurled network. The weight matrices of the connections *are shared by multiple connections* in the time-layered network to ensure that the same function is used at each time stamp. This sharing is the key to the domain-specific insights that are learned by the network. The backpropagation algorithm takes the sharing and temporal length into account when updating the weights during the learning process. This special type of backpropagation algorithm is referred to as *backpropagation through time (BPTT).* Because of the recursive nature of Equation 1.36, the recurrent network has the *ability to compute a function of variable-length inputs.* In other words, one can expand the recurrence of Equation 1.36 to define the function for $\overline{h}_t$ in terms of t inputs. For example, starting at $\overline{h}_0$, which is typically fixed to some constant vector, we have $\overline{h}_1 = f(\overline{h}_0, \overline{x}_1)$ and $\overline{h}_2 = f(f(\overline{h}_0, \overline{x}_1), \overline{x}_2)$. Note that $\overline{h}_1$ is a function of only $\overline{x}_1$, whereas $\overline{h}_2$ is a function of both $\overline{x}_1$ and $\overline{x}_2$. Since the output $\overline{y}_t$ is a function of $\overline{h}_t$, these properties are inherited by $\overline{y}_t$ as well. In general, we can write the following:

$$\overline{y}_t = F_t(\overline{x}_1, \overline{x}_2, \ldots \overline{x}_t) \tag{1.37}$$

Note that the function $F_t(\cdot)$ varies with the value of t. Such an approach is particularly useful for variable-length inputs like text sentences. More details of recurrent neural networks are provided in Chapter 7; this chapter will also discuss the applications of recurrent neural networks in various domains.

An interesting theoretical property of recurrent neural networks is that they are *Turing complete* [444]. What this means is that *given enough data and computational resources*, a recurrent neural network can simulate any algorithm. In practice, however, this theoretical property is not useful because recurrent networks have significant practical problems with generalization for long sequences. The amount of data and the size of the hidden states required for longer sequences increases in a way that is not realistic. Furthermore, there are practical issues in finding the optimum choices of parameters because of the vanishing and exploding gradient problems. As a result, specialized variants of the recurrent neural network architecture have been proposed, such as the use of long short-term memory. These advanced architectures will also be discussed in Chapter 7. Furthermore, some advanced variants of the recurrent architecture, such as neural Turing machines, have shown improvements over the recurrent neural network in some applications.

1.6.5 Convolutional Neural Networks

Convolutional neural networks are biologically inspired networks that are used in computer vision for image classification and object detection. The basic motivation for the convolutional neural network was obtained from Hubel and Wiesel's understanding [212] of the workings of the cat's visual cortex, in which specific portions of the visual field seemed to excite particular neurons. This broader principle was used to design a sparse architecture for convolutional neural networks. The first basic architecture based on this biological inspiration was the *neocognitron*, which was then generalized to the *LeNet-5* architecture [279]. In the convolutional neural network architecture, each layer of the network is 3-dimensional, which has a spatial extent and a depth corresponding to the number of features. The notion of depth of a single layer in a convolutional neural network is distinct[4] from the notion of

[4] This is an overloading of the terminology used in convolutional neural networks. The meaning of the word "depth" is inferred from the context in which it is used.

depth in terms of the number of layers. In the input layer, these features correspond to the color channels like RGB (i.e., red, green, blue), and in the hidden channels these features represent hidden feature maps that encode various types of shapes in the image. If the input is in grayscale (like *LeNet-5*), then the input layer will have a depth of 1, but later layers will still be 3-dimensional. The architecture contains two types of layers, referred to as the *convolution* and *subsampling* layers, respectively.

For the convolution layers, a *convolution operation* is defined, in which a filter is used to map the activations from one layer to the next. A convolution operation uses a 3-dimensional filter of weights with the same depth as the current layer but with a smaller spatial extent. The dot product between all the weights in the filter and any choice of spatial region (of the same size as the filter) in a layer defines the value of the hidden state in the next layer (after applying an activation function like ReLU). The operation between the filter and the spatial regions in a layer is performed at every possible position in order to define the next layer (in which the activations retain their spatial relationships from the previous layer).

The connections in a convolutional neural network are very sparse, because any activation in a particular layer is a function of only a small spatial region in the previous layer. All layers other than the final set of two of three layers maintain their spatial structure. Therefore, it is possible to spatially visualize what parts of the image affect particular portions of the activations in a layer. The features in lower-level layers capture lines or other primitive shapes, whereas the features in higher-level layers capture more complex shapes like loops (which commonly occur in many digits). Therefore, later layers can create digits by composing the shapes in these intuitive features. This is a classical example of the way in which semantic insights about specific data domains are used to design clever architectures. In addition, a subsampling layer simply averages the values in the local regions of size 2 × 2 in order to compress the spatial footprints of the layers by a factor of 2. An illustration of the architecture of *LeNet-5* is shown in Figure 1.18. In the early years, *LeNet-5* was used by several banks to recognize hand-written numbers on checks.

Convolutional neural networks have historically been the most successful of all types of neural networks. They are used widely for image recognition, object detection/localization, and even text processing. The performance of these networks has recently exceeded that of humans in the problem of image classification [184]. Convolutional neural networks provide a very good example of the fact that architectural design choices in a neural network should be performed with semantic insight about the data domain at hand. In the particular case

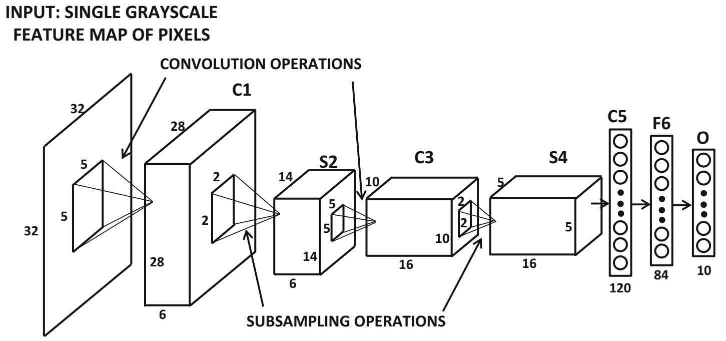


Figure 1.18: LeNet-5: One of the earliest convolutional neural networks.

of the convolutional neural network, this insight was obtained by observing the biological workings of a cat's visual cortex, and heavily using the spatial relationships among pixels. This fact also provides some evidence that a further understanding of neuroscience might also be helpful for the development of methods in artificial intelligence.

Pretrained convolutional neural networks from publicly available resources like *ImageNet* are often available for use in an off-the-shelf manner for other applications and data sets. This is achieved by using most of the pretrained weights in the convolutional network without any change except for the final classification layer. The weights of the final classification layer are learned from the data set at hand. The training of the final layer is necessary because the class labels in a particular setting may be different from those of *ImageNet*. Nevertheless, the weights in the early layers are still useful because they learn various types of shapes in the images that can be useful for virtually any type of classification application. Furthermore, the feature activations in the penultimate layer can even be used for unsupervised applications. For example, one can create a multidimensional representation of an arbitrary image data set by passing each image through the convolutional neural network and extracting the activations of the penultimate layer. Subsequently, any type of indexing can be applied to this representation for retrieving images that are similar to a specific target image. Such an approach often provides surprisingly good results in image retrieval because of the semantic nature of the features learned by the network. It is noteworthy that the use of pretrained convolutional networks is so popular that training is rarely started from scratch. Convolutional neural networks are discussed in detail in Chapter 8.

1.6.6 Hierarchical Feature Engineering and Pretrained Models

Many deeper architectures with feed-forward architectures have multiple layers in which successive transformations of the inputs from the previous layer lead to increasingly sophisticated representations of the data. The values of each hidden layer for a particular input contain a transformed representation of the input point, which becomes increasingly informative about the target value we are trying to learn, as the layer gets closer to the output node. As shown in Section 1.5.1, appropriately transformed feature representations are more amenable to simple types of predictions in the output layer. This sophistication is a result of the nonlinear activations in intermediate layers. Traditionally, the sigmoid and tanh activations were the most popular choices in the hidden layers, but the ReLU activation has become increasingly popular in recent years because of the desirable property that it is better at avoiding the vanishing and exploding gradient problems (cf. Section 3.4.2 of Chapter 3). For classification, the final layer can be viewed as a relatively simple prediction layer which contains a single linear neuron in the case of regression, and is a sigmoid/sign function in the case of binary classification. More complex outputs might require multiple nodes. One way of viewing this division of labor between the hidden layers and final prediction layer is that the early layers create a feature representation that is more amenable to the task at hand. The final layer then leverages this learned feature representation. This division of labor is shown in Figure 1.19. A key point is that the features learned in the hidden layers are often (but not always) generalizable to other data sets and problem settings in the same domain (e.g., text, images, and so on). This property can be leveraged in various ways by simply replacing the output node(s) of a pretrained network with a different application-specific output layer (e.g., linear regression layer instead of sigmoid classification layer) for the data set and problem at hand. Subsequently, only the weights of the newly replaced output layer may need to be learned for the new data set and application, whereas the weights of other layers are fixed.

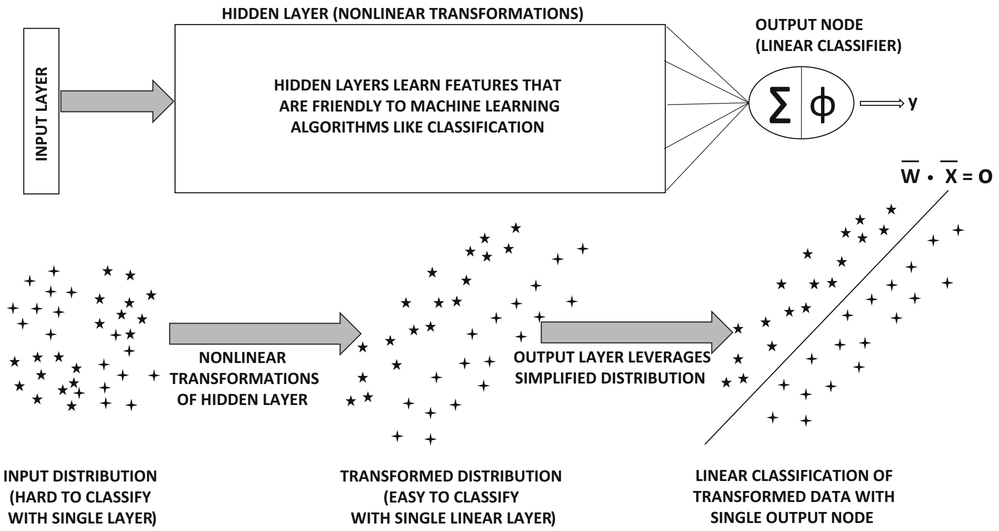

HIDDEN LAYER (NONLINEAR TRANSFORMATIONS)
INPUT LAYER
HIDDEN LAYERS LEARN FEATURES THAT ARE FRIENDLY TO MACHINE LEARNING ALGORITHMS LIKE CLASSIFICATION
OUTPUT NODE (LINEAR CLASSIFIER)
Σ
Φ
y
W · X = 0
NONLINEAR TRANSFORMATIONS OF HIDDEN LAYER
OUTPUT LAYER LEVERAGES SIMPLIFIED DISTRIBUTION
INPUT DISTRIBUTION (HARD TO CLASSIFY WITH SINGLE LAYER)
TRANSFORMED DISTRIBUTION (EASY TO CLASSIFY WITH SINGLE LINEAR LAYER)
LINEAR CLASSIFICATION OF TRANSFORMED DATA WITH SINGLE OUTPUT NODE

Figure 1.19: The feature engineering role of the hidden layers

The output of each hidden layer is a transformed feature representation of the data, in which the dimensionality of the representation is defined by the number of units in that layer. One can view this process as a kind of hierarchical feature engineering in which the features in earlier layers represent primitive characteristics of the data, whereas those in later layers represent complex characteristics with semantic significance to the class labels. Data represented in the terms of the features of later layers are often more well behaved (e.g., linearly separable) because of the semantic nature of the features learned by the transformation. This type of behavior is particularly evident in a visually interpretable way in some domains like convolutional neural networks for image data. In convolutional neural networks, the features in earlier layers capture detailed but primitive shapes like lines or edges from the data set of images. On the other hand, the features in later layers capture shapes of greater complexity like hexagons, honeycombs, and so forth, depending on the type of images provided as training data. Note that such semantically interpretable shapes often have closer correlations with class labels in the image domain. For example, almost any image will contain lines or edges, but images belonging to particular classes will be more likely to have hexagons or honeycombs. This property tends to make the representations of later layers easier to classify with simple models like linear classifiers. This process is illustrated in Figure 1.19. The features in earlier layers are used repeatedly as building blocks to create more complex features. This general principle of "putting together" simple features to create more complex features lies at the core of the successes achieved with neural networks. As it turns out, this property is also useful in leveraging pretrained models in a carefully calibrated way. The practice of using pretrained models is also referred to as *transfer learning*.

A particular type of transfer learning, which is used commonly in neural networks, is that the data and structure available in a given data set are used to learn features for that entire domain. A classical example of this setting is that of text or image data. In text data, the representations of text words are created using standardized benchmark data sets like *Wikipedia* [594] and models like *word2vec*. These can be used in almost any text application, since the nature of text data does not change very much with the application. A similar approach is often used for image data, in which the *ImageNet data set* (cf. Section 1.8.2) is used to pretrain convolutional neural networks, and provide ready-to-use features. One can download a pretrained convolutional neural network model and convert any image data set into a multidimensional representation by passing the image through the pretrained network. Furthermore, if additional application-specific data is available, one can regulate the level of transfer learning depending on the amount of available data. This is achieved by fine-tuning a subset of the layers in the pretrained neural network with this additional data. If a small amount of application-specific data is available, one can fix the weights of the early layers to their pretrained values and fine-tune only the last few layers of the neural network. The early layers often contain primitive features, which are more easily generalizable to arbitrary applications. For example, in a convolutional neural network, the early layers learn primitive features like edges, which are useful across diverse images like trucks or carrots. On the other hand, the later layers contain complex features which might depend on the image collection at hand (e.g., truck wheel versus carrot top). Fine-tuning only the weights of the later layers makes sense in such cases. If a large amount of application-specific data is available, one can fine-tune a larger number of layers. Therefore, deep networks provide significant flexibility in terms of how transfer learning is done with pretrained neural network models.

1.7 Advanced Topics

Several topics in deep learning have increasingly gained attention, and have had significant successes. Although some of these methods are limited by current computational considerations, their potential is quite significant. This section will discuss some of these topics.

1.7.1 Reinforcement Learning

In general forms of artificial intelligence, the neural network must learn to take actions in ever-changing and dynamic situations. Examples include learning robots and self-driving cars. In these cases, a critical assumption is that the learning system has no knowledge of the appropriate sequence of actions up front, and it learns through reward-based *reinforcement* as it takes various actions. These types of learning correspond to dynamic sequences of actions that are hard to model using traditional machine learning methods. The key assumption here is that these systems are too complex to explicitly model, but they are simple enough to evaluate, so that a *reward value* can be assigned for each action of the learner.

Imagine a setting in which one wishes to train a learning system to play a video game from scratch without any prior knowledge of the rules. Video games are excellent test beds for reinforcement learning methods because they are microcosms of living the "game" of life. As in real-world settings, the number of possible *states* (i.e., unique position in game) might be too large to even enumerate, and the optimal choice of move depends critically on the knowledge of what is truly important to model from a particular state. Furthermore, since one does not start with any knowledge of the rules, the learning system would need to collect the data through its actions much as a mouse explores a maze to learn its structure. Therefore, the collected data is highly biased by the user actions, which provides a particularly challenging landscape for learning. The successful training of reinforcement learning methods is a critical gateway for *self-learning systems*, which is the holy grail of artificial intelligence. Although the field of reinforcement learning was developed independently of the field of neural networks, the strong complementarity of the two fields has brought them together. Deep learning methods can be useful in learning feature representations from high-dimensional sensory inputs (e.g., the video screens of pixels in a video game or the screen of pixels in a robot's "vision"). Furthermore, reinforcement learning methods are often used to support various types of neural network algorithms like attention mechanisms. Reinforcement learning methods are discussed in Chapter 9.

1.7.2 Separating Data Storage and Computations

An important aspect of neural networks is that the data storage and computations are tightly integrated. For example, the states in a neural network can be considered a type of transient memory, which behave much like the ever-changing registers in the central processing unit of a computer. But what if we want to construct a neural network where one can control where to read data from, and where to write the data to. This goal is achieved with the notion of *attention* and *external memory*. Attention mechanisms can be used in various applications like image processing where one focuses on small parts of the image to gain successive insights. These techniques are also used for machine translation. Neural networks that can tightly control access in reading and writing to an external memory are referred to as *neural Turing machines* [158] or *memory networks* [528]. Although these methods are advanced variants of recurrent neural networks, they show significantly improved potential than their predecessors in terms of the types of problems they can handle. These methods are discussed in Chapter 10.

1.7.3 Generative Adversarial Networks

Generative adversarial networks are a model of data generation that can create a generative model of a base data set by using an adversarial game between two players. The two players correspond to a generator and a discriminator. The generator takes Gaussian noise as input and produces an output, which is a *generated* sample like the base data. The discriminator is typically a probabilistic classifier like logistic regression whose job is to distinguish real samples from the base data set and the generated sample. The generator tries to create samples that are as realistic as possible; its job is to fool the discriminator, whereas the job of the discriminator is to identify the fake samples irrespective of how well the generator tries to fool it. The problem can be understood as an adversarial game between the generator and discriminator, and the formal optimization model is a minimax learning problem. The *Nash equilibrium* of this minimax game provides the final trained model. Typically, this equilibrium point is one at which the discriminator is unable to distinguish between real and fake samples.

Such methods can create realistic fantasy samples using a base data set, and are used commonly in the image domain. For example, if the approach is trained using a data set containing images of bedrooms, it will produce realistic looking bedrooms that are not actually a part of the base data. Therefore, the approach can be used for artistic or creative endeavors. These methods can also be conditioned on specific types of context, which could be any type of object such as label, text caption, or an image with missing details. In these cases, *pairs* of related training objects are used. A typical pair could be a caption (context) and an image (base object). Similarly, one might have pairs corresponding to sketches of objects and actual photographs. Therefore, starting with a captioned image data set of various types of animals, it is possible to create a fantasy image that is not a part of the base data by using a contextual caption such as "*blue bird with sharp claws.*" Similarly, starting with an artist's sketch of a purse, the approach can create a realistic and colored image of a purse. Generative adversarial networks are discussed in Chapter 10.

1.8 Two Notable Benchmarks

The benchmarks used in the neural network literature are dominated by data from the domain of computer vision. Although traditional machine learning data sets like the UCI repository [601] can be used for testing neural networks, the general trend is towards using data sets from perceptually oriented data domains that can be visualized well. Although there are a variety of data sets drawn from the text and image domains, two of them stand out because of their ubiquity in deep learning papers. Although both are data sets drawn from computer vision, the first of them is simple enough that it can also be used for testing generic applications beyond the field of vision. In the following, we provide a brief overview of these two data sets.

1.8.1 The MNIST Database of Handwritten Digits

The MNIST database, which stands for *Modified National Institute of Standards and Technology* database, is a large database of handwritten digits [281]. As the name suggests, this data set was created by modifying an original database of handwritten digits provided by NIST. The data set contains 60,000 training images and 10,000 testing images. Each image is a scan of a handwritten digit from 0 to 9, and the differences between different images are a result of the differences in the handwriting of different individuals. These individuals were American Census Bureau employees and American high school students. The original black and white images from NIST were size normalized to fit in a 20×20 pixel box while preserving their aspect ratio and centered in a 28×28 image by computing the center of mass of the pixels. The images were translated to position this point at the center of the 28×28 field. Each of these 28×28 pixel values takes on a value from 0 to 255, depending on where it lies in the grayscale spectrum. The labels associated with the images correspond to the ten digit values. Examples of the digits in the MNIST database are illustrated in Figure 1.20. The size of the data set is rather small, and it contains only a simple object corresponding to a digit. Therefore, one might argue that the MNIST database is a toy data set. However, its small size and simplicity is also an advantage because it can be used as a laboratory for quick testing of machine learning algorithms. Furthermore, the simplification of the data set by virtue of the fact that the digits are (roughly) centered makes it easy to use it to test algorithms beyond computer vision. Computer vision algorithms require specialized assumptions such as translation invariance. The simplicity of this data set makes these assumptions unnecessary. It has been remarked by Geoff Hinton [600] that the MNIST

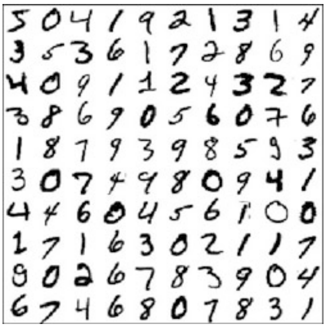


Figure 1.20: Examples of handwritten digits in the MNIST database

database is used by neural network researchers in much the same way as biologists use fruit flies for early and quick results (before serious testing on more complex organisms).

Although the matrix representation of each image is suited to a convolutional neural network, one can also convert it into a multidimensional representation of $28 \times 28 = 784$ dimensions. This conversion loses some of the spatial information in the image, but this loss is not debilitating (at least in the case of the MNIST data set) because of its relative simplicity. In fact, the use of a simple support vector machine on the 784-dimensional representation can provide an impressive error rate of about 0.56%. A straightforward 2-layer neural network on the multidimensional representation (without using the spatial structure in the image) generally does worse than the support vector machine across a broad range of parameter choices! A deep neural network without any special convolutional architecture can achieve an error rate of 0.35% [72]. Deeper neural networks and convolutional neural networks (that do use spatial structure) can reduce the error rate to as low as 0.21% by using an ensemble of five convolutional networks [402]. Therefore, even on this simple data set, one can see that the relative performance of neural networks with respect to traditional machine learning is sensitive to the specific architecture used in the former.

Finally, it should be noted that the 784-dimensional non-spatial representation of the MNIST data is used for testing all types of neural network algorithms beyond the domain of computer vision. Even though the use of the 784-dimensional (flattened) representation is not appropriate for a vision task, it is still useful for testing the general effectiveness of non-vision oriented (i.e., generic) neural network algorithms. For example, the MNIST data is frequently used to test generic autoencoders and not just convolutional ones. Even when the non-spatial representation of an image is used to reconstruct it with an autoencoder, one can still visualize the results with the original spatial positions of the reconstructed pixels to obtain a feel of what the algorithm is doing with the data. This visual exploration often gives the researcher some insights that are not available with arbitrary data sets like those obtained from the UCI Machine Learning Repository [601]. In this sense, the MNIST data set tends to have broader usability than many other types of data sets.

1.8.2 The ImageNet Database

The *ImageNet* database [581] is a huge database of over 14 million images drawn from 1000 different categories. Its class coverage is exhaustive enough that it covers most types of images that one would encounter in everyday life. This database is organized according to a *WordNet* hierarchy of nouns [329]. The *WordNet* database is a data set containing the relationships among English words using the notion of *synsets*. The *WordNet* hierarchy has been successfully used for machine learning in the natural language domain, and therefore it is natural to design an image data set around these relationships.

The *ImageNet* database is famous for the fact that an annual *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* [582] is held using this dataset. This competition has a very high profile in the vision community and receives entries from most major research groups in computer vision. The entries to this competition have resulted in many of the state-of-the-art image recognition architectures today, including the methods that have surpassed human performance on some narrow tasks like image classification [184]. Because of the wide availability of known results on these data sets, it is a popular alternative for benchmarking. We will discuss some of the state-of-the-art algorithms submitted to the *ImageNet* competition in Chapter 8 on convolutional neural networks.

Another important significance of the *ImageNet* data set is that it is large and diverse enough to be representative of the key visual concepts within the image domain. As a result, convolutional neural networks are often trained on this data set; the pretrained network can be used to extract features from an arbitrary image. This image representation is defined by the hidden activations in the penultimate layer of the neural network. Such an approach creates new multidimensional representations of image data sets that are amenable for use with traditional machine learning methods. One can view this approach as a kind of transfer learning in which the visual concepts in the *ImageNet* data set are transferred to unseen data objects for other applications.

1.9 Summary

Although a neural network can be viewed as a simulation of the learning process in living organisms, a more direct understanding of neural networks is as computational graphs. Such computational graphs perform recursive composition of simpler functions in order to learn more complex functions. Since these computational graphs are parameterized, the problem generally boils down to learning the parameters of the graph in order to optimize a loss function. The simplest types of neural networks are often basic machine learning models like least-squares regression. The real power of neural networks is unleashed by using more complex combinations of the underlying functions. The parameters of such networks are learned by using a dynamic programming method, referred to as backpropagation. There are several challenges associated with learning neural network models, such as overfitting and training instability. In recent years, numerous algorithmic advancements have reduced these problems. The design of deep learning methods in specific domains such as text and images requires carefully crafted architectures. Examples of such architectures include recurrent neural networks and convolutional neural networks. For dynamic settings in which a sequence of decisions need to be learned by a system, methods like reinforcement learning are useful.

1.10 Bibliographic Notes

A proper understanding of neural network design requires a solid understanding of machine learning algorithms, and especially the linear models based on gradient descent. The reader is recommended to refer to [2, 3, 40, 177] for basic knowledge on machine learning methods. Numerous surveys and overviews of neural networks in different contexts may be found in [27, 28, 198, 277, 345, 431]. Classical books on neural networks for pattern recognition may be found in [41, 182], whereas more recent perspectives on deep learning may be found in [147]. A recent text mining book [6] also discusses recent advances in deep learning for text analytics. An overview of the relationships between deep learning and computational neuroscience may be found in [176, 239].

The perceptron algorithm was proposed by Rosenblatt [405]. To address the issue of stability, the *pocket algorithm* [128], the *Maxover* algorithm [523], and other margin-based methods [123]. Other early algorithms of a similar nature included the Widrow-Hoff [531] and the Winnow algorithms [245]. The Winnow algorithm uses multiplicative updates instead of additive updates, and is particularly useful when many features are irrelevant. The original idea of backpropagation was based on the idea of differentiation of composition of functions as developed in control theory [54, 237]. The use of dynamic programming to perform gradient-based optimization of variables that are related via a directed acyclic graph has been a standard practice since the sixties. However, the ability to use these methods for neural network training had not yet been observed at the time. In 1969, Minsky and Papert published a book on perceptrons [330], which was largely negative about the potential of being able to properly train multilayer neural networks. The book showed that a single perceptron had limited expressiveness, and no one knew how to train multiple layers of perceptrons anyway. Minsky was an influential figure in artificial intelligence, and the negative tone of his book contributed to the first winter in the field of neural networks. The adaptation of dynamic programming methods to backpropagation in neural networks was first proposed by Paul Werbos in his PhD thesis in 1974 [524]. However, Werbos's work could not overcome the strong views against neural networks that had already become entrenched at the time. The backpropagation algorithm was proposed again by Rumelhart *et al.* in 1986 [408, 409]. Rumelhart *et al.*'s work is significant for the beauty of its presentation, and it was able to address at least some of the concerns raised earlier by Minsky and Papert. This is one of the reasons that the Rumelhart *et al.* paper is considered very influential from the perspective of backpropagation, even though it was certainly not the first to propose the method. A discussion of the history of the backpropagation algorithm may be found in the book by Paul Werbos [525].

At this point, the field of neural networks was only partially resurrected, as there were still problems with training neural networks. Nevertheless, pockets of researchers continued to work in the area, and had already set up most of the known neural architectures, such as convolution neural networks, recurrent neural networks, and LSTMs, before the year 2000. The accuracy of these methods was still quite modest because of data and computation limitations. Furthermore, backpropagation turned out to be less effective at training deeper networks because of the vanishing and exploding gradient problems. However, by this time, it was already hypothesized by several prominent researchers that existing algorithms would yield large performance improvements with increases in data, computational power, and algorithmic experimentation. The coupling of big data frameworks with GPUs turned out to be a boon for neural network research in the late 2000s. With reduced cycle times for experimentation enabled by increased computational power, tricks like pretraining started showing up in the late 2000s [198]. The publicly obvious resurrection of neural networks occurred after the year 2011 with the resounding victories [255] of neural networks in deep learning competitions for image classification. The consistent victories of deep learning algorithms in these competitions laid the foundation for the explosion in popularity we see today. Notably, the differences of these winning architectures from the ones that were developed more than two decades earlier are modest (but essential).

Paul Werbos was a pioneer of recurrent neural networks, and proposed the original version of backpropagation through time [526]. The basics of the convolutional neural network were proposed in the context of the *neocognitron* in [127]. This idea was then generalized to *LeNet-5*, which was one of the first convolutional neural networks. The ability of neural networks to perform universal function approximation is discussed in [208]. The beneficial effect of depth on reducing the number of parameters is discussed in [340].

The theoretical expressiveness of neural networks was recognized early in its development. For example, early work recognized that a neural network with a single hidden layer can be used to approximate any function [208]. A further result is that certain neural architectures like recurrent networks are Turing complete [444]. The latter means that neural networks can potentially simulate any algorithm. Of course, there are numerous practical issues associated with neural network training, as to why these exciting theoretical results do not always translate into real-world performance. The foremost problem among them is the data-hungry nature of shallow architectures, which is ameliorated with increased depth. Increased depth can be viewed as a form of regularization in which one is forcing the neural network to identify and learn repeating patterns in data points. Increased depth, however, makes the neural network harder to train from an optimization point of view. A discussion on some of these issues may be found in [41, 140, 147]. An experimental evaluation showing the advantages of deeper architectures is provided in [267].

1.10.1 Video Lectures

Deep learning has a significant number of free video lectures available on resources such as *YouTube* and *Coursera*. Two of the most authoritative resources include Geoff Hinton's course at *Coursera* [600]. *Coursera* has multiple offerings on deep learning, and offers a group of related courses in the area. During the writing of this book, an accessible course from Andrew Ng was also added to the offerings. A course on convolutional neural networks from Stanford University is freely available on *YouTube* [236]. The Stanford class by Karpathy, Johnson, and Fei-Fei [236] is on convolutional neural networks, although it does an excellent job in covering broader topics in neural networks. The initial parts of the course deal with vanilla neural networks and training methods.

Numerous topics in machine learning [89] and deep learning [90] are covered by Nando de Freitas in a lectures available on *YouTube*. Another interesting class on neural networks is available from Hugo Larochelle at the Universite de Sherbrooke [262]. A deep learning course by Ali Ghodsi at the University of Waterloo is available at [137]. Video lectures by Christopher Manning on natural language processing methods for deep learning may be found in [312]. David Silver's course on reinforcement learning is available at [619].

1.10.2 Software Resources

Deep learning is supported by numerous software frameworks like *Caffe* [571], *Torch* [572], *Theano* [573], and *TensorFlow* [574]. Extensions of *Caffe* to Python and MATLAB are available. *Caffe* was developed at the University of California at Berkeley, and it is written in C++. It provides a high-level interface in which one can specify the architecture of the network, and it enables the construction of neural networks with very little code writing and relatively simple scripting. The main drawback of *Caffe* is the relatively limited documentation available. *Theano* [35] is Python-based, and it provides high-level packages like *Keras* [575] and *Lasagne* [576] as interfaces. *Theano* is based on the notion of computational graphs, and most of the capabilities provided around it use this abstraction explicitly. *TensorFlow* [574] is also strongly oriented towards computational graphs, and is the framework proposed by Google. *Torch* [572] is written in a high-level language called *Lua*, and it is relatively friendly to use. In recent years, *Torch* has gained some ground compared to other frameworks. Support for GPUs is tightly integrated in *Torch*, which makes it relatively easy to deploy *Torch*-based applications on GPUs. Many of these frameworks contain pretrained models from computer vision and text mining, which can be used to extract features. Many off-the-shelf tools for deep learning are available from the **DeepLearning4j** repository [590]. IBM has a *PowerAI* platform that offers many machine learning and deep learning frameworks on top of IBM Power Systems [599]. Notably, as of the writing of this book, this platform also has a free edition available for certain uses.

1.11 Exercises

1. Consider the case of the XOR function in which the two points $\{(0, 0), (1, 1)\}$ belong to one class, and the other two points $\{(1, 0), (0, 1)\}$ belong to the other class. Show how you can use the ReLU activation function to separate the two classes in a manner similar to the example in Figure 1.14.

2. Show the following properties of the sigmoid and tanh activation functions (denoted by $\Phi(\cdot)$ in each case):

   (a) Sigmoid activation: $\Phi(-v) = 1 - \Phi(v)$

   (b) Tanh activation: $\Phi(-v) = -\Phi(v)$

   (c) Hard tanh activation: $\Phi(-v) = -\Phi(v)$

3. Show that the tanh function is a re-scaled sigmoid function with both horizontal and vertical stretching, as well as vertical translation:

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

4. Consider a data set in which the two points $\{(-1, -1), (1, 1)\}$ belong to one class, and the other two points $\{(1, -1), (-1, 1)\}$ belong to the other class. Start with perceptron parameter values at $(0, 0)$, and work out a few stochastic gradient-descent updates with $\alpha = 1$. While performing the stochastic gradient-descent updates, cycle through the training points in any order.

   (a) Does the algorithm converge in the sense that the change in objective function becomes extremely small over time?

   (b) Explain why the situation in (a) occurs.

5. For the data set in Exercise 4, where the two features are denoted by (x_1, x_2), define a new 1-dimensional representation z denoted by the following:

$$z = x_1 \cdot x_2$$

   Is the data set linearly separable in terms of the 1-dimensional representation corresponding to z? Explain the importance of nonlinear transformations in classification problems.

6. Implement the perceptron in a programming language of your choice.

7. Show that the derivative of the sigmoid activation function is at most 0.25, irrespective of the value of its argument. At what value of its argument does the sigmoid activation function take on its maximum value?

8. Show that the derivative of the tanh activation function is at most 1, irrespective of the value of its argument. At what value of its argument does the tanh activation take on its maximum value?

9. Consider a network with two inputs x_1 and x_2. It has two hidden layers, each of which contain two units. Assume that the weights in each layer are set so that top unit in each layer applies sigmoid activation to the sum of its inputs and the bottom unit in each layer applies tanh activation to the sum of its inputs. Finally, the single output

node applies ReLU activation to the sum of its two inputs. Write the output of this neural network *in closed form* as a function of x_1 and x_2. This exercise should give you an idea of the complexity of functions computed by neural networks.

10. Compute the partial derivative of the closed form computed in the previous exercise with respect to x_1. Is it practical to compute derivatives for gradient descent in neural networks by using closed-form expressions (as in traditional machine learning)?

11. Consider a 2-dimensional data set in which all points with $x_1 > x_2$ belong to the positive class, and all points with $x_1 \leq x_2$ belong to the negative class. Therefore, the true separator of the two classes is linear hyperplane (line) defined by $x_1 - x_2 = 0$. Now create a training data set with 20 points randomly generated inside the unit square in the positive quadrant. Label each point depending on whether or not the first coordinate x_1 is greater than its second coordinate x_2.

(a) Implement the perceptron algorithm without regularization, train it on the 20 points above, and test its accuracy on 1000 randomly generated points inside the unit square. Generate the test points using the same procedure as the training points.

(b) Change the perceptron criterion to hinge-loss in your implementation for training, and repeat the accuracy computation on the same test points above. Regularization is not used.

(c) In which case do you obtain better accuracy and why?

(d) In which case do you think that the classification of the same 1000 test instances will not change significantly by using a different set of 20 training points?

Chapter 2

Machine Learning with Shallow Neural Networks

"Simplicity is the ultimate sophistication."—Leonardo da Vinci

2.1 Introduction

Conventional machine learning often uses optimization and gradient-descent methods for learning parameterized models. Examples of such models include linear regression, support vector machines, logistic regression, dimensionality reduction, and matrix factorization. Neural networks are also parameterized models that are learned with continuous optimization methods. This chapter will show that a wide variety of optimization-centric methods in machine learning can be captured with *very simple* neural network architectures containing one or two layers. In fact, neural networks can be viewed as more powerful versions of these simple models, with this power being achieved by combining the basic models into a comprehensive neural architecture (i.e., computational graph). It is useful to show these parallels early on, as this allows the understanding of the design of a deep network as a composition of the basic units that one often uses in machine learning. Furthermore, showing this relationship provides an appreciation of the specific way in which traditional machine learning is different from neural networks, and of the cases in which one can hope to do better with neural networks. In many cases, minor variations of these simple neural network architectures (corresponding to traditional machine learning methods) provide useful variations of machine learning models that have not been studied elsewhere. In a sense, the number of ways in which one can combine the different elements of a computational graph is far greater than what is studied in traditional machine learning, even when shallow models are used.

Complex or deep neural architectures are often an overkill in instances where only a small amount of data are available. Additionally, it is easier to optimize traditional machine

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_2

learning models in data-lean settings as these models are more interpretable. On the other hand, as the amount of data increases, neural networks have an advantage because they retain the flexibility to model more complex functions with the addition of neurons to the computational graph. Figure 2.1 illustrates this point.

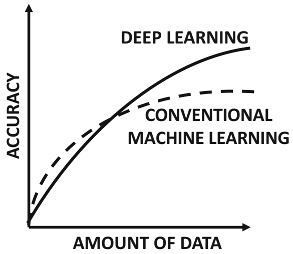


Figure 2.1: Re-visiting Figure 1.2: The effect of increased data availability on accuracy.

One way of viewing deep learning models is as a stacking of simpler models like logistic or linear regression. The coupling of a linear neuron with the sigmoid activation leads to *logistic regression*, which will be discussed in detail in this chapter. The coupling of a linear unit with sigmoid activation is also used[1] extensively for building complex neural networks. Therefore, it is natural to ask the following question [312]:

> *Is deep learning simply a stacking of simpler models like logistic or linear regression?*

Although many neural networks can be viewed in this way, this point of view does not fully capture the complexity and the style of thinking involved in deep learning models. For example, several models (such as recurrent neural networks or convolutional neural networks) perform this stacking in a particular way with a *domain-specific understanding* of the input data. Furthermore, the parameters of different units are sometimes shared in order to force the solution to obey specific types of properties. The ability to put together the basic units in a clever way is a key architectural skill required by practitioners in deep learning. Nevertheless, it is also important to learn the properties of the basic models in machine learning, since they are used repeatedly in deep learning as elementary units of computation. This chapter will, therefore, explore these basic models.

It is noteworthy that there are close relationships between some of the earliest neural networks (e.g., perceptron and Widrow-Hoff learning) and traditional machine learning models (e.g., support vector machine and Fisher discriminant). In some cases, these relationships remained unnoticed for several years, as these models were proposed independently by different communities. As a specific example, the loss function of the L_2-support vector machine was proposed by Hinton [190] in the context of a neural architecture in 1989. When used with regularization, the resulting neural network would behave identically to an L_2-support vector machine. In comparison, Cortes and Vapnik's paper on the support vector machine [82] appeared several years later with an L_1-loss function. These relationships are not surprising because the best way to define a shallow neural network is often closely related to a known machine learning algorithm. Therefore, it is important to explore these basic neural models in order to develop an integrated view of neural networks and traditional machine learning.

[1] In recent years, the sigmoid unit has fallen out of favor compared to the ReLU.

This chapter will primarily discuss two classes of models for machine learning:

1. *Supervised models:* The supervised models discussed in this chapter primarily correspond to linear models and their variants. These include methods like least-squares regression, support vector machines, and logistic regression. Multiclass variants of these models will also be studied.

2. *Unsupervised models:* The unsupervised models discussed in this chapter primarily correspond to dimensionality reduction and matrix factorization. Traditional methods like principal component analysis can also be presented as simple neural network architectures. Minor variations of these models can provide reductions of vastly different properties, which will be discussed later. The neural network framework also provides a way of understanding the relationships between widely different unsupervised methods like linear dimensionality reduction, nonlinear dimensionality reduction, and sparse feature learning, thereby providing an integrated view of traditional machine learning algorithms.

This chapter assumes that the reader has a basic familiarity with the classical machine learning models. Nevertheless, a brief overview of each model will also be provided to the uninitiated reader.

Chapter Organization

The next section will discuss some basic models for classification and regression, such as least-squares regression, binary Fisher discriminant, support vector machine, and logistic regression. The multiway variants of these models will be discussed in Section 2.3. Feature selection methods for neural networks are discussed in Section 2.4. The use of autoencoders for matrix factorization is discussed in Section 2.5. As a specific application of simple neural architectures, the *word2vec* method is discussed in Section 2.6. Simple methods for creating node embeddings in graphs are introduced in Section 2.7. A summary is given in Section 2.8.

2.2 Neural Architectures for Binary Classification Models

In this section, we will discuss some basic architectures for machine learning models such as least-squares regression and classification. As we will see, the corresponding neural architectures are minor variations of the perceptron model in machine learning. The main difference is in the choice of the activation function used in the final layer, and the loss function used on these outputs. This will be a recurring theme throughout this chapter, where we will see that small changes in neural architectures can result in distinct models from traditional machine learning. Presenting traditional machine learning models in the form of neural architectures also helps one appreciate the true closeness among various machine learning models.

Throughout this section, we will work with a single-layer network with d input nodes and a single output node. The coefficients of the connections from the d input nodes to the output node are denoted by $\overline{W} = (w_1 \ldots w_d)$. Furthermore, the bias will not be explicitly shown because it can be seamlessly modeled as the coefficient of an additional dummy input with a constant value of 1.

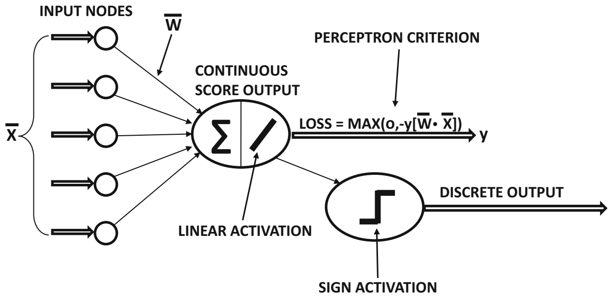


Figure 2.2: An extended architecture of the perceptron with both discrete and continuous predictions

2.2.1 Revisiting the Perceptron

Let $(\overline{X_i}, y_i)$ be a training instance, in which the observed value y_i is predicted from the feature variables $\overline{X_i}$ using the following relationship:

$$\hat{y}_i = \text{sign}(\overline{W} \cdot \overline{X_i}) \tag{2.1}$$

Here, $\overline{W}$ is the d-dimensional coefficient vector learned by the perceptron. Note the circumflex on top of $\hat{y}_i$ to indicate that it is a predicted value rather than an observed value. In general, the goal of training is to ensure that the prediction $\hat{y}_i$ is as close as possible to the observed value y_i. The gradient-descent steps of the perceptron are focused on reducing the number of misclassifications, and therefore the updates are proportional to the difference $(y_i - \hat{y}_i)$ between the observed and predicted values based on Equation 1.33 of Chapter 1:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha(y_i - \hat{y}_i)\overline{X_i} \tag{2.2}$$

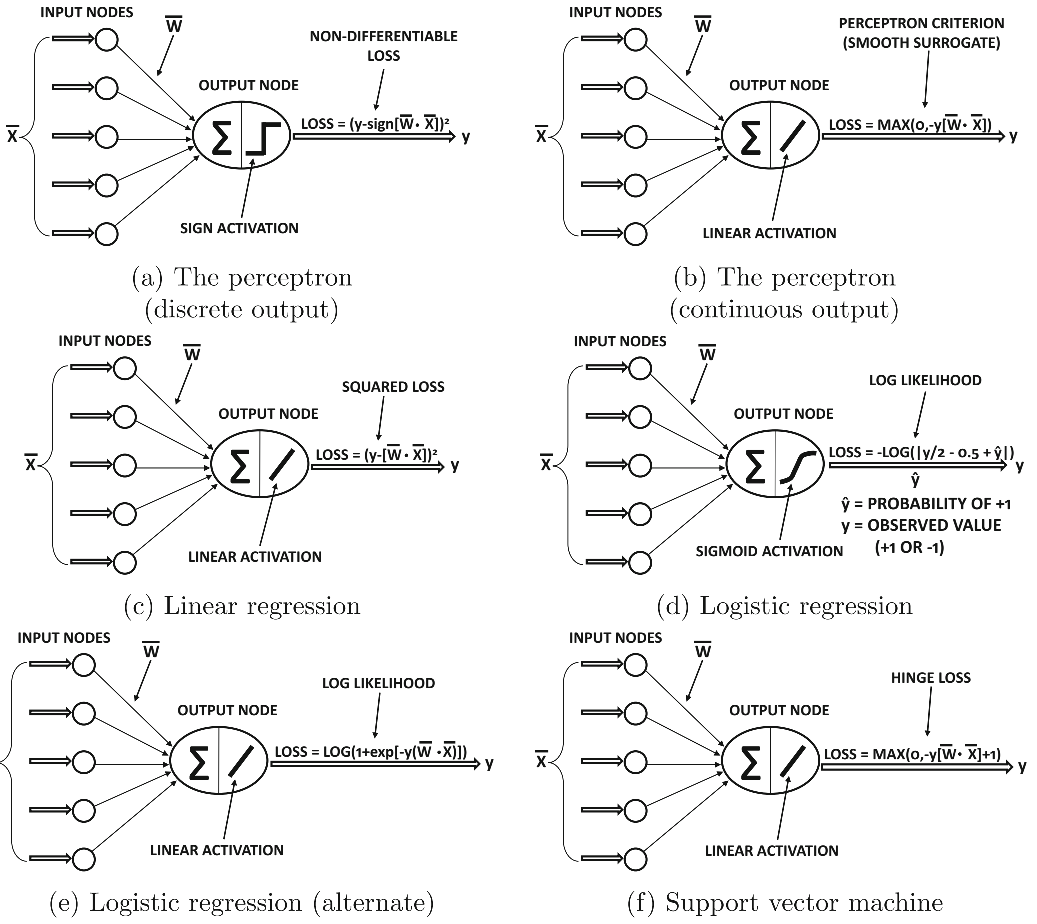
A gradient-descent update that is proportional to the difference between the observed and predicted values is naturally caused by a squared loss function such as $(y_i - \hat{y}_i)^2$. Therefore, one possibility is to consider the squared loss between the predicted and observed values as the loss function. This architecture is shown in Figure 2.3(a), and the output is a discrete value. However, the problem is that this loss function is discrete because it takes on the value of either 0 or 4. Such a loss function is not differentiable because of its staircase-like jumps.

The perceptron is one of the few learning models in which the gradient-descent updates were proposed historically before the loss function was proposed. What *differentiable* objective function does the perceptron really optimize? The answer to this question may be found in Section 1.2.1.1 of Chapter 1 by observing that the updates are performed only for misclassified training instances (i.e., $y_i\hat{y}_i < 0$), and may be written using the indicator function $I(\cdot) \in \{0, 1\}$ that takes on 1 when the condition in its argument is satisfied:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha y_i \overline{X_i} \left[I(y_i\hat{y}_i < 0) \right] \tag{2.3}$$

This rewrite from Equation 2.2 to Equation 2.3 uses the fact that $y_i = (y_i - \hat{y}_i)/2$ for misclassified points, and one can absorb a constant factor of 2 within the learning rate. This update can be shown to be consistent with the loss function L_i (specific to the ith training example) as follows:

$$L_i = \max\{0, -y_i(\overline{W} \cdot \overline{X_i})\} \tag{2.4}$$



(a) The perceptron (discrete output)

(b) The perceptron (continuous output)

(c) Linear regression

(d) Logistic regression

(e) Logistic regression (alternate)

(f) Support vector machine

Figure 2.3: Different variants of the perceptron

This loss function is referred to as the *perceptron criterion*, which is correspondingly reflected in Figure 2.3(b). Note that Figure 2.3(b) uses *linear* activations to compute the continuous loss function, although it still uses sign activations to compute the discrete predictions for a given test instance. In many discrete variable prediction settings, the output is often a predicted score (e.g., probability of class or the value of $\overline{W} \cdot \overline{X_i}$), which is then converted into a discrete prediction. Nevertheless, the final prediction need not always be converted into a discrete value, and one can simply output the relevant score for the class (which is often used for computing the loss function anyway). The sign activation is rarely used in most neural-network implementations, as most class-variable predictions of neural-network implementations are continuous scores. One can, in fact, create an extended architecture for the perceptron (cf. Figure 2.2), in which both discrete and continuous values are output. However, since the discrete part is not relevant to the loss computation and most outputs are reported as scores anyway, one rarely uses this type of extended representation. Therefore, throughout the remainder of this book, the activation in the output node is based on the score output (and how the loss function is computed), rather than on how a test instance is predicted as a discrete value.

2.2.2 Least-Squares Regression

In least-squares regression, the training data contains n different training pairs $(\overline{X_1}, y_1) \ldots (\overline{X_n}, y_n)$, where each $\overline{X_i}$ is a d-dimensional representation of the data points, and each y_i is a *real-valued* target. The fact that the target is real-valued is important, because the underlying problem is then referred to as *regression* rather than *classification*. Least-squares regression is the oldest of all learning problems, and the gradient-descent methods proposed by Tikhonov and Arsenin in the 1970s [499] are very closely related to the gradient-descent updates of Rosenblatt [405] for the perceptron algorithm. In fact, as we will see later, one can also use least-squares regression on binary targets by "pretending" that these targets are real-valued. The resulting approach is equivalent to the Widrow-Hoff learning algorithm, which is famous in the neural network literature as the second learning algorithm proposed after the perceptron.

In least-squares regression, the target variable is related to the feature variables using the following relationship:

$$\hat{y}_i = \overline{W} \cdot \overline{X_i} \tag{2.5}$$

Note the presence of the circumflex on top of $\hat{y}_i$ to indicate that it is a predicted value. The bias is missing in the relationship of Equation 2.5. Throughout this section, it will be assumed that one of the features in the training data has a constant value of 1, and the coefficient of this dummy feature is the bias. This is a standard feature engineering trick borrowed from conventional machine learning. In neural networks, the bias is often represented with the use of a bias neuron (cf. Section 1.2.1 of Chapter 1) with a constant output of 1. Although the bias neuron is almost always used in real settings, we avoid showing it explicitly throughout this book in order to maintain simplicity in presentation.

The error of the prediction, e_i, is given by $e_i = (y_i - \hat{y}_i)$. Here, $\overline{W} = (w_1 \ldots w_d)$ is a d-dimensional coefficient vector that needs to be learned so as to minimize the total squared error on the training data, which is $\sum_{i=1}^{n} e_i^2$. The portion of the loss that is specific to the ith training instance is given by the following:

$$L_i = e_i^2 = (y_i - \hat{y}_i)^2 \tag{2.6}$$

This loss can be simulated with the use of an architecture similar to the perceptron except that the squared loss is paired with the identity activation function. This architecture is shown in Figure 2.3(c), whereas the perceptron architecture is shown in Figure 2.3(a). Both the perceptron and least-squares regression have the same goal of minimizing the prediction error. However, since the loss function in classification is inherently discrete, the perceptron algorithm uses a smooth approximation of the desired goal. This results in the smoothed *perceptron criterion* shown in Figure 2.3(b). As we will see below, the gradient-descent update in least-squares regression is very similar to that in the perceptron, with the main difference being that real-valued errors are used in regression rather than discrete errors drawn from $\{-2, +2\}$.

As in the perceptron algorithm, the stochastic gradient-descent steps are determined by computing the gradient of e_i^2 with respect to $\overline{W}$, when the training pair $(\overline{X_i}, y_i)$ is presented to the neural network. This gradient can be computed as follows:

$$\frac{\partial e_i^2}{\partial \overline{W}} = -e_i \overline{X_i} \tag{2.7}$$

Therefore, the gradient-descent updates for $\overline{W}$ are computed using the above gradient and step-size α:

$$\overline{W} \Leftarrow \overline{W} + \alpha e_i \overline{X}$$

One can rewrite the above update as follows:

$$\overline{W} \Leftarrow \overline{W} + \alpha(y_i - \hat{y}_i)\overline{X} \tag{2.8}$$

It is possible to modify the gradient-descent updates of least-squares regression to incorporate forgetting factors. Adding regularization is equivalent to penalizing the loss function of least-squares classification with the additional term proportional to $\lambda \cdot ||\overline{W}||^2$, where $\lambda > 0$ is the regularization parameter. With regularization, the update can be written as follows:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha \cdot \lambda) + \alpha(y_i - \hat{y}_i)\overline{X} \tag{2.9}$$

Note that the update above looks identical to the perceptron update of Equation 2.2. The updates are, however, not exactly identical because of how the predicted value $\hat{y}_i$ is computed in the two cases. In the case of the perceptron, the sign function is applied to $\overline{W} \cdot \overline{X_i}$ in order to compute the binary value $\hat{y}_i$ and therefore the error $(y_i - \hat{y}_i)$ can only be drawn from $\{-2, +2\}$. In least-squares regression, the prediction $\hat{y}_i$ is a real value without the application of the sign function.

This observation naturally leads to the following question; what if we applied least-squares regression directly to minimize the squared distance of the *real-valued* prediction $\hat{y}_i$ from the observed binary targets $y_i \in \{-1, +1\}$? The direct application of least-squares regression to binary targets is referred to as *least-squares classification*. The gradient-descent update is the same as the one shown in Equation 2.9, which *looks* identical to that of the perceptron. However, the least-squares classification method does not yield the same result as the perceptron algorithm, because the *real-valued* training errors $(y_i - \hat{y}_i)$ in least-squares classification are computed differently from the *integer* error $(y_i - \hat{y}_i)$ in the perceptron. This direct application of least-squares regression to binary targets is referred to as *Widrow-Hoff learning*.

2.2.2.1 Widrow-Hoff Learning

Following the perceptron, the Widrow-Hoff learning rule was proposed in 1960. However, the method was not a fundamentally new one, as it is a direct application of least-squares regression to binary targets. Although the sign function is applied to the real-valued prediction *of unseen test instances* to convert them to binary predictions, the error of training instances is computed directly using real-valued predictions (unlike the perceptron). Therefore, it is also referred to as *least-squares classification* or *linear least-squares method* [6]. Remarkably, a seemingly unrelated method proposed in 1936, known as the Fisher discriminant, also reduces to Widrow-Hoff learning in the special case of binary targets.

The Fisher discriminant is formally defined as a direction $\overline{W}$ along which the ratio of inter-class variance to the intra-class variance is maximized in the projected data. By choosing a scalar b in order to define the hyperplane $\overline{W} \cdot \overline{X} = b$, it is possible to model the separation between the two classes. This hyperplane is used for classification. Although the definition of the Fisher discriminant seems quite different from least-squares regression/classification at first sight, a remarkable result is that the Fisher discriminant for binary targets is identical to the least-squares regression as applied to binary targets (i.e., least-squares classification). Both the data and the targets need to be mean-centered, which allows the bias variable b to be set to 0. Several proofs of this result are available in the literature [3, 6, 40, 41].

The neural architecture for classification with the Widrow-Hoff method is illustrated in Figure 2.3(c). The gradient-descent steps in both the perceptron and the Widrow-Hoff would be given by Equation 2.8, except for differences in how $(y_i - \hat{y}_i)$ is computed. In the case of the perceptron, this value will always be drawn from $\{-2, +2\}$. In the case of Widrow-Hoff, these errors can be arbitrary real values, since $\hat{y}_i$ is set to $\overline{W} \cdot \overline{X_i}$ without using the sign function. This difference is important because the perceptron algorithm never penalizes a positive class point for $\overline{W} \cdot \overline{X_i}$ being "too correct" (i.e., larger than 1), whereas using real-valued predictions to compute the error has the unfortunate effect of penalizing such points. The inappropriate penalization of over-performance is the Achilles heel of Widrow-Hoff learning and the Fisher discriminant [6].

It is noteworthy that least-squares regression/classification, Widrow-Hoff learning, and the Fisher discriminant were proposed independently in very different eras and by different communities of researchers. Indeed, the Fisher discriminant, which is oldest of these methods and dates back to 1936, is often viewed as a method for finding class-sensitive directions rather than as a classifier. It can, however, also be used as a classifier by using the resulting direction $\overline{W}$ to create a linear prediction. The completely different origins and seemingly different motives of all these methods make the equivalence in their solutions all the more noticeable. The Widrow-Hoff learning rule is also referred to as *Adaline*, which is short for *adaptive linear neuron*. It is also referred to as the *delta rule*. To recap, the learning rule of Equation 2.8, when applied to binary targets in $\{-1, +1\}$, can be alternatively referred to as least-squares classification, least mean-squares algorithm (LMS), Fisher[2] discriminant classifier, the Widrow-Hoff learning rule, delta rule, or Adaline. Therefore, the family of least-squares classification methods has been rediscovered several times in the literature under different names and with different motivations.

The loss function of the Widrow-Hoff method can be rewritten slightly from least-squares regression because of its binary responses:

$$\begin{aligned} L_i &= (y_i - \hat{y}_i)^2 = \underbrace{y_i^2}_{1}(y_i - \hat{y}_i)^2 \\ &= (\underbrace{y_i^2}_{1} - \hat{y}_i y_i)^2 = (1 - \hat{y}_i y_i)^2 \end{aligned}$$

This type of encoding is possible when the target variable y_i is drawn from $\{-1, +1\}$ because we can use $y_i^2 = 1$. It is helpful to convert the Widrow-Hoff objective function to this form because it can be more easily related to other objective functions like the perceptron and the support vector machine. For example, the loss function of the support vector machine is obtained by "repairing" the above loss so that over-performance is not penalized. One can repair the loss function by changing the objective function to $[\max\{(1 - \hat{y}_i y_i), 0\}]^2$, which was Hinton's L_2-loss support vector machine (SVM) [190]. Almost all the binary classification models discussed in this chapter can be shown to be closely related to the Widrow-Hoff loss function by using different ways of repairing the loss, so that over-performance is not penalized.

The gradient-descent updates (cf. Equation 2.9) of least-squares regression can be rewritten slightly for Widrow-Hoff learning because of binary response variables:

$$\begin{aligned} \overline{W} &\Leftarrow \overline{W}(1 - \alpha \cdot \lambda) + \alpha(y_i - \hat{y}_i)\overline{X} && \text{[For numeric as well as binary responses]} \\ &= \overline{W}(1 - \alpha \cdot \lambda) + \alpha y_i(1 - y_i \hat{y}_i)\overline{X} && \text{[Only for binary responses, since } y_i^2 = 1\text{]} \end{aligned}$$

[2] In order to obtain exactly the same direction as the Fisher method with Equation 2.8, it is important to mean-center both the feature variables and the binary targets. Therefore, each binary target will be one of two real values with different signs. The real values will contain the fraction of instances belonging to the other class. Alternatively, one can use a bias neuron to absorb the constant offsets.

The second form of the update is helpful in relating it to perceptron and SVM updates, in each of which $(1 - y_i \hat{y}_i)$ is replaced with an indicator variable that is a function of $y_i \hat{y}_i$. This point will be discussed in a later section.

2.2.2.2 Closed Form Solutions

The special case of least-squares regression and classification is solvable in closed form (without gradient-descent) by using the *pseudo-inverse* of the $n \times d$ training data matrix D, whose rows are $\overline{X_1} \ldots \overline{X_n}$. Let the n-dimensional column vector of dependent variables be denoted by $\overline{y} = [y_1 \ldots y_n]^T$. The pseudo-inverse of matrix D is defined as follows:

$$D^+ = (D^T D)^{-1} D^T \tag{2.10}$$

Then, the row-vector $\overline{W}$ is defined by the following relationship:

$$\overline{W}^T = D^+ \overline{y} \tag{2.11}$$

If regularization is incorporated, the coefficient vector $\overline{W}$ is given by the following:

$$\overline{W}^T = (D^T D + \lambda I)^{-1} D^T \overline{y} \tag{2.12}$$

Here, $\lambda > 0$ is the regularization parameter. However, inverting a matrix like $(D^T D + \lambda I)$ is typically done using numerical methods that require gradient descent anyway. One rarely inverts large matrices like $D^T D$. In fact, the Widrow-Hoff updates provide a very efficient way of solving the problem without using the closed-form solution.

2.2.3 Logistic Regression

Logistic regression is a probabilistic model that classifies the instances in terms of probabilities. Because the classification is probabilistic, a natural approach for optimizing the parameters is to ensure that the predicted probability of the observed class for each training instance is as large as possible. This goal is achieved by using the notion of *maximum-likelihood estimation* in order to learn the parameters of the model. The likelihood of the training data is defined as the product of the probabilities of the observed labels of each training instance. Clearly, larger values of this objective function are better. By using the negative logarithm of this value, one obtains an a loss function in minimization form. Therefore, the output node uses the negative *log-likelihood* as a loss function. This loss function replaces the squared error used in the Widrow-Hoff method. The output layer can be formulated with the sigmoid activation function, which is very common in neural network design.

Let $(\overline{X_1}, y_1), (\overline{X_2}, y_2), \ldots (\overline{X_n}, y_n)$ be a set of n training pairs in which $\overline{X_i}$ contains the d-dimensional features and $y_i \in \{-1, +1\}$ is a binary class variable. As in the case of a perceptron, a single-layer architecture with weights $\overline{W} = (w_1 \ldots w_d)$ is used. Instead of using the hard sign activation on $\overline{W} \cdot \overline{X_i}$ to predict y_i, logistic regression applies the soft sigmoid function to $\overline{W} \cdot \overline{X_i}$ in order to estimate the *probability* that y_i is 1:

$$\hat{y}_i = P(y_i = 1) = \frac{1}{1 + \exp(-\overline{W} \cdot \overline{X_i})} \tag{2.13}$$

For a test instance, it can be predicted to the class whose predicted probability is greater than 0.5. Note that $P(y_i = 1)$ is 0.5 when $\overline{W} \cdot \overline{X_i} = 0$, and $\overline{X_i}$ lies on the separating hyperplane. Moving $\overline{X_i}$ in either direction from the hyperplane results in different signs of $\overline{W} \cdot \overline{X_i}$ and corresponding movements in the probability values. Therefore, the sign of $\overline{W} \cdot \overline{X_i}$ also yields the same prediction as picking the class with probability larger than 0.5.

We will now describe how the loss function corresponding to likelihood estimation is set up. This methodology is important because it is used widely in many neural models. For positive samples in the training data, we want to maximize $P(y_i = 1)$ and for negative samples, we want to maximize $P(y_i = -1)$. For positive samples satisfying $y_i = 1$, one wants to maximize $\hat{y}_i$ and for negative samples satisfying $y_i = -1$, one wants to maximize $1 - \hat{y}_i$. One can write this casewise maximization in the form of a consolidated expression of always maximizing $|y_i/2 - 0.5 + \hat{y}_i|$. The products of these probabilities must be maximized over all training instances to maximize the likelihood $\mathcal{L}$:

$$\mathcal{L} = \prod_{i=1}^{n} |y_i/2 - 0.5 + \hat{y}_i| \tag{2.14}$$

Therefore, the loss function is set to $L_i = -\log(|y_i/2 - 0.5 + \hat{y}_i|)$ for each training instance, so that the product-wise maximization is converted to additive minimization over training instances.

$$\mathcal{LL} = -\log(\mathcal{L}) = \sum_{i=1}^{n} \underbrace{-\log(|y_i/2 - 0.5 + \hat{y}_i|)}_{L_i} \tag{2.15}$$

Additive forms of the objective function are particularly convenient for the types of stochastic gradient updates that are common in neural networks. The overall architecture and loss function is illustrated in Figure 2.3(d). For each training instance, the predicted probability $\hat{y}_i$ is computed by passing it through the neural network, and the loss is used to determine the gradient for each training instance.

Let the loss for the ith training instance be denoted by L_i, which is also annotated in Equation 2.15. Then, the gradient of L_i with respect to the weights in $\overline{W}$ can be computed as follows:

$$\begin{aligned}
\frac{\partial L_i}{\partial \overline{W}} &= -\frac{\text{sign}(y_i/2 - 0.5 + \hat{y}_i)}{|y_i/2 - 0.5 + \hat{y}_i|} \cdot \frac{\partial \hat{y}_i}{\partial \overline{W}} \\
&= -\frac{\text{sign}(y_i/2 - 0.5 + \hat{y}_i)}{|y_i/2 - 0.5 + \hat{y}_i|} \cdot \frac{\overline{X_i}}{1 + \exp(-\overline{W} \cdot \overline{X_i})} \cdot \frac{1}{1 + \exp(\overline{W} \cdot \overline{X_i})} \\
&= \begin{cases} -\frac{\overline{X_i}}{1+\exp(\overline{W} \cdot \overline{X_i})} & \text{if } y_i = 1 \\ \frac{\overline{X_i}}{1+\exp(-\overline{W} \cdot \overline{X_i})} & \text{if } y_i = -1 \end{cases}
\end{aligned}$$

Note that one can concisely write the above gradient as follows:

$$\frac{\partial L_i}{\partial \overline{W}} = -\frac{y_i \overline{X_i}}{1 + \exp(y_i \overline{W} \cdot \overline{X_i})} = -\left[\text{Probability of mistake on } (\overline{X_i}, y_i)\right] (y_i \overline{X_i}) \tag{2.16}$$

Therefore, the gradient-descent updates of logistic regression are given by the following (including regularization):

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha \frac{y_i \overline{X_i}}{1 + \exp[y_i(\overline{W} \cdot \overline{X_i})]} \tag{2.17}$$

Just as the perceptron and the Widrow-Hoff algorithms use the *magnitudes* of the mistakes to make updates, the logistic regression method uses the *probabilities* of the mistakes to make updates. This is a natural extension of the probabilistic nature of the loss function to the update.

2.2.3.1 Alternative Choices of Activation and Loss

It is possible to implement the same model by using different choices of activation and loss in the output node as long as they combine to yield the same result. Instead of using sigmoid activation to create the output $\hat{y}_i \in (0, 1)$, it is also possible to use identity activation to create the output $\hat{y}_i \in (-\infty, +\infty)$, and then apply the following loss function:

$$L_i = \log(1 + \exp(-y_i \cdot \hat{y}_i)) \tag{2.18}$$

The alternative architecture for logistic regression is shown in Figure 2.3(e). For the final prediction of the test instance, the sign function can be applied to $\hat{y}_i$, which is equivalent to predicting it to the class for which its probability is greater than 0.5. This example shows that it is possible to implement the same model using different combinations of activation and loss functions, as long as they combine to yield the same result.

One desirable property of using the identity activation to define $\hat{y}_i$ is that it is consistent with how the loss functions of other models like the perceptron and Widrow-Hoff learning are defined. Furthermore, the loss function of Equation 2.18 contains the product of y_i and $\hat{y}_i$ as in other models. This makes it possible to directly compare the loss functions of various models, which will be explored later in this chapter.

2.2.4 Support Vector Machines

The loss function in support vector machines is closely related to that in logistic regression. However, instead of using a smooth loss function (like that in Equation 2.18), the *hinge-loss* is used instead.

Consider the training data set of n instances denoted by $(\overline{X_1}, y_1)$, $(\overline{X_2}, y_2)$, $\ldots (\overline{X_n}, y_n)$. The neural architecture of the support-vector machine is identical to that of least-squares classification (Widrow-Hoff). The main difference is in the choice of loss function. As in the case of least-squares classification, the prediction $\hat{y}_i$ for the training point $\overline{X_i}$ is obtained by applying the identity activation function on $\overline{W} \cdot \overline{X_i}$. Here, $\overline{W} = (w_1, \ldots w_d)$ contains the vector of d weights for the d different inputs into the single-layer network. Therefore, the output of the neural network is $\hat{y}_i = \overline{W} \cdot \overline{X_i}$ for computing the loss function, although a test instance is predicted by applying the sign function to the output.

The loss function L_i for the ith training instance in the support-vector machine is defined as follows:

$$L_i = \max\{0, 1 - y_i \hat{y}_i\} \tag{2.19}$$

This loss is referred to as the *hinge-loss*, and the corresponding neural architecture is illustrated in Figure 2.3(f). The overall idea behind this loss function is that a positive training instance is only penalized for being less than 1, and a negative training instance is only penalized for being greater than -1. In both cases, the penalty is linear, and abruptly flattens out at the aforementioned thresholds. It is helpful to compare this loss function with the Widrow-Hoff loss value of $(1 - y_i \hat{y}_i)^2$, in which predictions are penalized for being *different* from the target values. As we will see later, this difference is an important advantage for the support vector machine over the Widrow-Hoff loss function.

In order to explain the difference in loss functions between the perceptron, Widrow-Hoff, logistic regression, and the support vector machine, we have shown the loss for a single *positive* training instance at different values of $\hat{y}_i = \overline{W} \cdot \overline{X_i}$ in Figure 2.4. In the case of the perceptron, only the smoothed *surrogate* loss function (cf. Section 1.2.1.1 of Chapter 1) is shown. Since the target value is +1, the loss function shows diminishing improvement by increasing $\overline{W} \cdot \overline{X_i}$ beyond +1 in the case of logistic regression. In the case of the support-vector machine the hinge-loss function flattens out beyond this point. In other words, only misclassified points or points that are too close to the decision boundary $\overline{W} \cdot \overline{X} = 0$ are penalized. The perceptron criterion is identical in shape to the hinge loss, except that it is shifted by one unit to the left. The Widrow-Hoff method is the only case in which a positive training point is penalized for having too large a positive value of $\overline{W} \cdot \overline{X_i}$. In other words, the Widrow-Hoff method penalizes points for being properly classified in a very strong way. This is a potential problem with the Widrow-Hoff objective function, in which well-separated points cause problems in training.

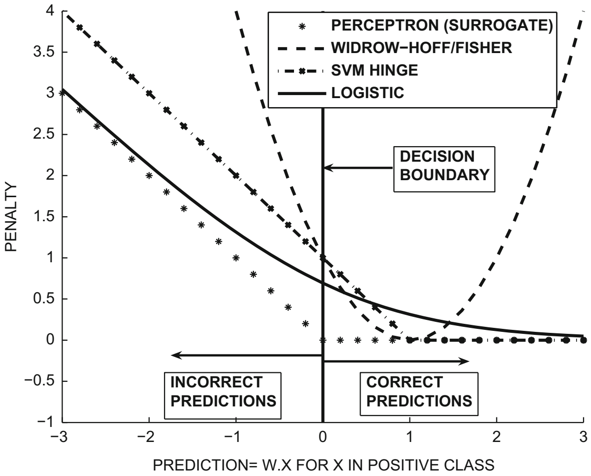


Figure 2.4: The loss functions of different variants of the perceptron. Key observations: (i) The SVM loss is shifted from the perceptron (surrogate) loss by exactly one unit to the right; (ii) the logistic loss is a smooth variant of the SVM loss; (iii) the Widrow-Hoff/Fisher loss is the only case in which points are increasingly penalized for classifying points "too correctly" (i.e., increasing $\overline{W} \cdot \overline{X}$ beyond +1 for $\overline{X}$ in positive class). Repairing the Widrow-Hoff loss function by setting it to 0 for $\overline{W} \cdot \overline{X} > 1$ yields the *quadratic loss* SVM [190].

The stochastic gradient-descent method computes the partial derivative of the point-wise loss function L_i with respect to the elements in $\overline{W}$. The gradient is computed as follows:

$$\frac{\partial L_i}{\partial \overline{W}} = \begin{cases} -y_i \overline{X_i} & \text{if } y_i \hat{y}_i < 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.20}$$

Therefore, the stochastic gradient method samples a point and checks whether $y_i \hat{y}_i < 1$. If this is the case, an update is performed that is proportional to $y_i \overline{X_i}$:

$$\overline{W} \Leftarrow \overline{W}(1 - \alpha\lambda) + \alpha y_i \overline{X_i} \, [I(y_i \hat{y}_i < 1)] \tag{2.21}$$

Here, $I(\cdot) \in \{0, 1\}$ is the indicator function that takes on the value of 1 when the condition in its argument is satisfied. This approach is the simplest version of the primal update for

SVMs [448]. The reader should also convince herself is that this update is *identical* to that of a (regularized) perceptron (cf. Equation 2.3), except that the condition for making this update in the perceptron is $y_i \hat{y}_i < 0$. Therefore, a perceptron makes the update only when a point is misclassified, whereas the support vector machine also makes updates for points that are classified correctly, albeit not very confidently. This neat relationship is because the loss function of the perceptron criterion shown in Figure 2.4 is shifted from the hinge-loss in the SVM.

To emphasize the similarities and differences in the loss functions used by the different methods, we tabulate the loss functions below:

Model	**Loss function** L_i for $(\overline{X_i}, y_i)$
Perceptron (Smoothed surrogate)	$\max\{0, -y_i \cdot (\overline{W} \cdot \overline{X_i})\}$
Widrow-Hoff/Fisher	$(y_i - \overline{W} \cdot \overline{X_i})^2 = \{1 - y_i \cdot (\overline{W} \cdot \overline{X_i})\}^2$
Logistic Regression	$\log(1 + \exp[-y_i(\overline{W} \cdot \overline{X_i})])$
Support vector machine (Hinge)	$\max\{0, 1 - y_i \cdot (\overline{W} \cdot \overline{X_i})\}$
Support vector machine (Hinton's L_2-Loss) [190]	$[\max\{0, 1 - y_i \cdot (\overline{W} \cdot \overline{X_i})\}]^2$

It is noteworthy that all the derived updates in this section typically correspond to stochastic gradient-descent updates that are encountered both in traditional machine learning and in neural networks. The updates are the same whether or not we use a neural architecture to represent the models for these algorithms. Our main point in going through this exercise is to show that rudimentary special cases of neural networks are instantiations of well-known algorithms in the machine learning literature. The key point is that with greater availability of data one can incorporate additional nodes and depth to increase the model's capacity, explaining the superior behavior of neural networks with larger data sets (cf. Figure 2.1).

2.3 Neural Architectures for Multiclass Models

All the models discussed so far in this chapter are designed for binary classification. In this section, we will discuss how one can design multiway classification models by changing the architecture of the perceptron slightly, and allowing multiple output nodes.

2.3.1 Multiclass Perceptron

Consider a setting with k different classes. Each training instance $(\overline{X_i}, c(i))$ contains a d-dimensional feature vector $\overline{X_i}$ and the index $c(i) \in \{1 \dots k\}$ of its observed class. In such a case, we would like to find k different linear separators $\overline{W_1} \dots \overline{W_k}$ simultaneously so that the value of $\overline{W}_{c(i)} \cdot \overline{X_i}$ is larger than $\overline{W_r} \cdot \overline{X_i}$ for each $r \neq c(i)$. This is because one always predicts a data instance $\overline{X_i}$ to the class r with the largest value of $\overline{W_r} \cdot \overline{X_i}$. Therefore, the loss function for the ith training instance in the case of the multiclass perceptron is defined as follows:

$$L_i = \max_{r:r \neq c(i)} \max(\overline{W_r} \cdot \overline{X_i} - \overline{W}_{c(i)} \cdot \overline{X_i}, 0) \tag{2.22}$$

The multiclass perceptron is illustrated in Figure 2.5(a). As in all neural network models, one can use gradient-descent in order to determine the updates. For a correctly classified instance, the gradient is always 0, and there are no updates. For a misclassified instance, the gradients are as follows: