Chapter 3

Training Deep Neural Networks

"I hated every minute of training, but I said, 'Don't quit. Suffer now and live the rest of your life as a champion."—Muhammad Ali

3.1 Introduction

The procedure for training neural networks with backpropagation is briefly introduced in Chapter 1. This chapter will expand on the description on Chapter 1 in several ways:

1. The backpropagation algorithm is presented in greater detail together with implementation details. Some details from Chapter 1 are repeated for completeness of the presentation, and so that readers do not have to frequently refer back to the earlier text.

2. Important issues related to feature preprocessing and initialization will be studied in the chapter.

3. The computational procedures that are paired with gradient descent will be introduced. The effect of network depth on training stability will be studied, and methods will be presented for addressing these issues.

4. The efficiency issues associated with training will be discussed. Methods for compressing trained models of neural networks will be presented. Such methods are useful for deploying pretrained networks on mobile devices.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_3

In the early years, methods for training multilayer networks were not known. In their influential book, Minsky and Papert [330] strongly argued against the prospects of neural networks because of the inability to train multilayer networks. Therefore, neural networks stayed out of favor as a general area of research till the eighties. The first significant breakthrough in this respect was proposed[1] by Rumelhart *et al.* [408, 409] in the form of the backpropagation algorithm. The proposal of this algorithm rekindled an interest in neural networks. However, several computational, stability, and overfitting challenges were found in the use of this algorithm. As a result, research in the field of neural networks again fell from favor.

At the turn of the century, several advances again brought popularity to neural networks. Not all of these advances were algorithm-centric. For example, increased data availability and computational power have played the primary role in this resurrection. However, some changes to the basic backpropagation algorithm and clever methods for initialization, such as *pretraining*, have also helped. It has also become easier in recent years to perform the intensive experimentation required for making algorithmic adjustments due to the reduced testing cycle times (caused by improved computational hardware). Therefore, increased data, computational power, and reduced experimentation time (for algorithmic tweaking) went hand-in-hand. These so-called "tweaks" are, nevertheless, very important; this chapter and the next will discuss most of these important algorithmic advancements.

One key point is that the backpropagation algorithm is rather *unstable* to minor changes in the algorithmic setting, such as the initialization point used by the approach. This instability is particularly significant when one is working with very deep networks. A point to note is that neural network optimization is a *multivariable optimization problem.* These variables correspond to the weights of the connections in various layers. Multivariable optimization problems often face stability challenges because one must perform the steps along each direction in the "right" proportion. This turns out to be particularly hard in the neural network domain, and the effect of a gradient-descent step might be somewhat unpredictable. One issue is that *a gradient only provides a rate of change over an infinitesimal horizon in each direction*, whereas an actual step has a finite length. One needs to choose steps of reasonable size in order to make any real progress in optimization. The problem is that the gradients do change over a step of finite length, and in some cases they change drastically. The complex optimization surfaces presented by neural network optimization are particularly treacherous in this respect, and the problem is exacerbated with poorly chosen settings (such as the initialization point or the normalization of the input features). As a result, the (easily computable) steepest-descent direction is often not the best direction to use for retaining the ability to use large steps. Small step sizes lead to slow progress, whereas the optimization surface might change in unpredictable ways with the use of large step sizes. All these issues make neural network optimization more difficult than would seem at first sight. However, many of these problems can be avoided by carefully tailoring the gradient-descent steps to be more robust to the nature of the optimization surface. This chapter will discuss algorithms that leverage some of this understanding.

[1] Although the backpropagation algorithm was popularized by the Rumelhart *et al.* papers [408, 409], it had been studied earlier in the context of control theory. Crucially, Paul Werbos's forgotten (and eventually rediscovered) thesis in 1974 discussed how these backpropagation methods could be used in neural networks. This was well before Rumelhart *et al.*'s papers in 1986, which were nevertheless significant because the style of presentation contributed to a better understanding of why backpropagation might work.

Chapter Organization

This chapter is organized as follows. The next section reviews the backpropagation algorithm initially discussed in Chapter 1. The discussion in this chapter is more detailed, and several variants of the algorithm are discussed. Some parts of the backpropagation algorithm that were already discussed in Chapter 1 are repeated so that this chapter is self-contained. Feature preprocessing and initialization issues are discussed in Section 3.3. The vanishing and exploding gradient problem, which is common in deep networks, is discussed in Section 3.4, with common solutions for dealing with this issue presented. Gradient-descent strategies for deep learning are discussed in Section 3.5. Batch normalization methods are introduced in Section 3.6. A discussion of accelerated implementations of neural networks is found in Section 3.7. The summary is presented in Section 3.8.

3.2 Backpropagation: The Gory Details

In this section, the backpropagation algorithm from Chapter 1 is reviewed again in considerably more detail. The goal of this more-detailed review is to show that the chain rule can be used in multiple ways. To this end, we first explore the standard backpropagation update as it is commonly presented in most textbooks (and Chapter 1). Second, a simplified and decoupled view of backpropagation is examined in which the linear matrix multiplications are decoupled from the activation layers. This decoupled view of backpropagation is what most off-the-shelf systems implement.

3.2.1 Backpropagation with the Computational Graph Abstraction

A neural network is a *computational graph*, in which a unit of computation is the neuron. 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. Furthermore, the nonlinear activations between the different layers add to the expressive power of the network.

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 $f(g(\cdot))$. Just to provide an idea, let us look at a trivial computational graph with two nodes, in which the sigmoid function is applied at each node to the input weight w. In such a case, the computed function appears as follows:

$$f(g(w)) = \frac{1}{1 + \exp\left[-\frac{1}{1+\exp(-w)}\right]} \tag{3.1}$$

We can already see how awkward it would be to compute the derivative of this function with respect to w. Furthermore, consider the case in which $g_1(\cdot)$, $g_2(\cdot) \ldots g_k(\cdot)$ are the functions computed in layer m, and they feed into a particular layer-$(m+1)$ node that computes $f(\cdot)$. In such a case, the composition function computed by the layer-$(m+1)$ node in terms of the layer-m inputs is $f(g_1(\cdot), \ldots g_k(\cdot))$. As we can see, this is a multivariate composition function, which looks rather ugly. Since the loss function uses the output(s) as its argument(s), it may typically be expressed a recursively nested function in terms of the weights in earlier layers. For a neural network with 10 layers and only 2 nodes in each layer, a recursively nested function of depth 10 will result in a summation of 2^{10} recursively nested terms, which appear forbidding from the perspective of computing partial derivatives. Therefore, we need some kind of iterative approach to compute these derivatives. The resulting iterative approach is *dynamic programming*, and the corresponding update is really the *chain rule of differential calculus*.

In order to understand how the chain rule works in a computational graph, we will discuss the two basic variants of the rule that one needs to keep in mind. The simplest version of the chain rule works for a straightforward composition of the functions:

$$\frac{\partial f(g(w))}{\partial w} = \frac{\partial f(g(w))}{\partial g(w)} \cdot \frac{\partial g(w)}{\partial w} \tag{3.2}$$

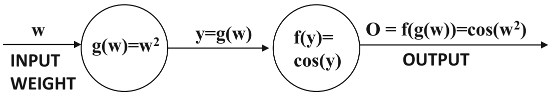
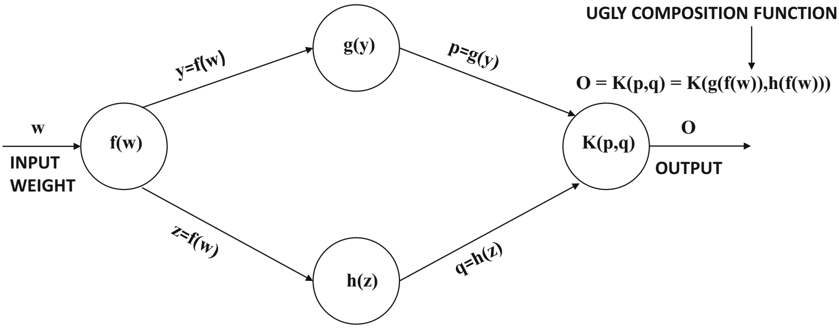


Figure 3.1: A simple computational graph with two nodes



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

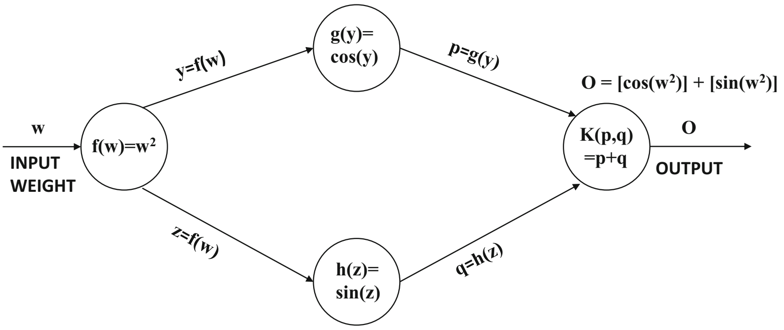
Figure 3.2: **Revisiting Figure 1.13 on chain rule in computational graphs:** The products of node-specific partial derivatives along paths from weight w to output o are aggregated. The resulting value yields the derivative of output O with respect to weight w. Only two paths between input and output exist in this simplified example.

This variant is referred to as the *univariate chain rule*. Note that each term on the right-hand side is a *local gradient* because it computes the derivative of a function with respect to its immediate argument rather than a recursively derived argument. The basic idea is that a composition of functions is applied on the weight w to yield the final output, and the gradient of the final output is given by the product of the local gradients along that path.

Each local gradient only needs to worry about its specific input and output, which simplifies the computation. An example is shown in Figure 3.1 in which the function $f(y)$ is $\cos(y)$ and $g(w) = w^2$. Therefore, the composition function is $\cos(w^2)$. On using the univariate chain rule, we obtain the following:

$$\frac{\partial f(g(w))}{\partial w} = \underbrace{\frac{\partial f(g(w))}{\partial g(w)}}_{-sin(g(w))} \cdot \underbrace{\frac{\partial g(w)}{\partial w}}_{2w} = -2w \cdot \sin(w^2)$$

The computational graphs in neural networks are not paths, which is the main reason that backpropagation is needed. A hidden layer often gets its input from multiple units, which results in multiple paths from a variable w to an output. Consider the function $f(g_1(w), \ldots g_k(w))$, in which a unit computing the *multivariate* function $f(\cdot)$ gets its inputs from k units computing $g_1(w) \ldots g_k(w)$. In such cases, the *multivariable chain rule* needs to be used. The multivariable chain rule is defined as follows:



$$\begin{aligned}\frac{\partial o}{\partial w} &= \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{\text{-sin}(y)} \cdot \underbrace{f'(w)}_{2w} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(w)}_{2w} \\ &= -2w \cdot \sin(y) + 2w \cdot \cos(z) \\ &= -2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)\end{aligned}$$

Figure 3.3: An example of the chain rule in action based on the computational graph of Figure 3.2.

$$\frac{\partial f(g_1(w), \ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w), \ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w} \tag{3.3}$$

It is easy to see that the multivariable chain rule of Equation 3.3 is a simple generalization of that in Equation 3.2. An important consequence of the multivariable chain rule is as follows:

Lemma 3.2.1 (Pathwise Aggregation Lemma) *Consider a directed acyclic computational graph in which the* i*th node contains variable* $y(i)$*. The local derivative* $z(i, j)$ *of the directed edge* (i, j) *in the graph is defined as* $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$*. Let a non-null set of paths* $\mathcal{P}$ *exist from variable* w *in the graph to output node containing variable* o*. Then, the value of* $\frac{\partial o}{\partial w}$ *is given by computing the product of the local gradients along each path in* $\mathcal{P}$*, and* *summing these products over all paths.*

$$\frac{\partial o}{\partial w} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i,j) \tag{3.4}$$

This lemma can be easily shown by applying Equation 3.3 recursively. Although Lemma 3.2.1 is not used anywhere in the backpropagation algorithm, it helps us develop another exponential-time algorithm that computes the derivatives explicitly. This point of view helps us interpret the multivariable chain rule as a dynamic programming recursion to compute a quantity that would otherwise be computationally too expensive to evaluate. Consider the example shown in Figure 3.2. There are two paths in this particular case. The recursive application of the chain rule is also shown in this example. It is evident that the final result is obtained by computing the product of the local gradients along each of the two paths and then adding them. In Figure 3.3, we have shown a more concrete example of a function that is evaluated by the same computational graph.

$$o = \sin(w^2) + \cos(w^2) \tag{3.5}$$

We have also shown in Figure 3.3 that the application of the chain rule on the computational graph correctly evaluates the derivative, which is $-2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)$.

An Exponential-Time Algorithm

The fact that we can compute the composite derivative as an aggregation of the products of local derivatives along all paths in the computational graph leads to the following exponential-time algorithm:

1. Use computational graph to compute the value $y(i)$ of each nodes i in a forward phase.
2. Compute the local partial derivatives $z(i,j) = \frac{\partial y(j)}{\partial y(i)}$ on each edge in the computational graph.
3. Let $\mathcal{P}$ be the set of all paths from an input node with value w to the output. For each path $P \in \mathcal{P}$ compute the product of each local derivative $z(i,j)$ on that path.
4. Add up these values over all paths in $\mathcal{P}$.

In general, a computational graph will have an exponentially increasing number of paths with depth and one must add the product of the local derivatives over all paths. An example is shown in Figure 3.4, in which we have five layers, each of which has only two units. Therefore, the number of paths between the input and output is $2^5 = 32$. The jth hidden unit of the ith layer is denoted by $h(i,j)$. Each hidden unit is defined as the product of its inputs:

$$h(i,j) = h(i-1,1) \cdot h(i-1,2) \quad \forall j \in \{1,2\} \tag{3.6}$$

In this case, the output is w^{32}, which is expressible in closed form, and can be differentiated easily with respect to w. However, we will use the exponential time algorithm to elucidate the workings of the exponential time algorithm. The derivative of each $h(i,j)$ with respect to each of its two inputs are the values of the complementary inputs:

$$\frac{\partial h(i,j)}{\partial h(i-1,1)} = h(i-1,2), \quad \frac{\partial h(i,j)}{\partial h(i-1,2)} = h(i-1,1)$$

The pathwise aggregation lemma implies that the value of $\frac{\partial o}{\partial w}$ is the product of the local derivatives (which are the complementary input values in this particular case) along all 32 paths from the input to the output:

$$\begin{aligned}\frac{\partial o}{\partial w} &= \sum_{j_1,j_2,j_3,j_4,j_5 \in \{1,2\}^5} \prod \underbrace{h(1,j_1)}_{w} \underbrace{h(2,j_2)}_{w^2} \underbrace{h(3,j_3)}_{w^4} \underbrace{h(4,j_4)}_{w^8} \underbrace{h(5,j_5)}_{w^{16}} \\ &= \sum_{\text{All 32 paths}} w^{31} = 32w^{31}\end{aligned}$$

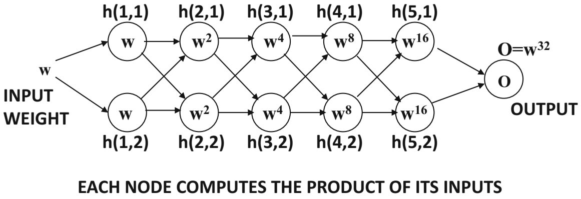


Figure 3.4: The number of paths in a computational graph increases exponentially with depth. In this case, the chain rule will aggregate the product of local derivatives along $2^5 = 32$ paths.

This result is, of course, consistent with what one would obtain on differentiating w^{32} directly with respect to w. However, an important observation is that it requires 2^5 aggregations to compute the derivative in this way for a relatively simple graph. More importantly, *we repeatedly differentiate the same function computed in a node* for aggregation.

Obviously, this is an inefficient approach to compute gradients. For a network with 100 nodes in each layer and three layers, we will have a million paths. *Nevertheless, this is exactly what we do in traditional machine learning when our prediction function is a complex composition function.* This also explains why most of traditional machine learning is a shallow neural model (cf. Chapter 2). Manually working out the details of a complex composition function is tedious and impractical beyond a certain level of complexity. It is here that the beautiful dynamic programming idea of backpropagation brings order to chaos, and enables models that would otherwise have been impossible.

3.2.2 Dynamic Programming to the Rescue

Although the summation discussed above has an exponential number of components (paths), one can compute it efficiently using dynamic programming. In graph theory, computing all types of path-aggregative values over directed acyclic graphs is done using dynamic programming. Consider a directed acyclic graph in which the value $z(i,j)$ (interpreted as local partial derivative of variable in node j with respect to variable in node i) is associated with edge (i,j). An example of such a computational graph is shown in Figure 3.5. We would like to compute the product of $z(i,j)$ over each path $P \in \mathcal{P}$ from source node w to output o and then add them.

$$S(w,o) = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i,j) \tag{3.7}$$

Let $A(i)$ be the set of nodes at the end points of outgoing edges from node i. We can compute the aggregated value $S(i, o)$ for each intermediate node i (between w and o) using the following well-known dynamic programming update:

$$S(i, o) \Leftarrow \sum_{j \in A(i)} S(j, o) z(i, j) \tag{3.8}$$

This computation can be performed backwards starting from the nodes directly incident on o, since $S(o, o)$ is already known to be 1. The algorithm discussed above is among the most widely used methods for computing all types of path-centric functions on directed acyclic graphs, which would otherwise require exponential time. For example, one can even use a variation of the above algorithm to find the longest path in a directed acyclic graph (which is known to be NP-hard for general graphs with cycles) [7]. This generic dynamic programming approach is used extensively in directed acyclic graphs.

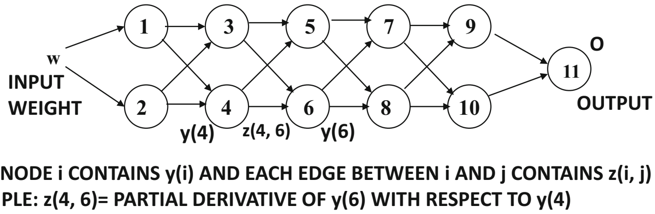

w
INPUT
WEIGHT
1
2
3
4
5
6
7
8
9
10
11
O
OUTPUT
y(4)
z(4, 6)
y(6)
EACH NODE i CONTAINS y(i) AND EACH EDGE BETWEEN i AND j CONTAINS z(i, j)
EXAMPLE: z(4, 6)= PARTIAL DERIVATIVE OF y(6) WITH RESPECT TO y(4)

Figure 3.5: Example of computational graph with edges corresponding to local partial derivatives

In fact, *the aforementioned dynamic programming update is exactly the multivariable chain rule of Equation 3.3, which is repeated in the backwards direction starting at the output node where the local gradient is known.* This is because we derived the path-aggregative form of the loss gradient (Lemma 3.2.1) using this chain rule in the first place. The main difference is that we apply the rule in a particular order in order to minimize computations. We summarize this point below:

> Using dynamic programming to efficiently aggregate the product of local gradients along the exponentially many paths in a computational graph results in a dynamic programming update that is identical to the multivariable chain rule of differential calculus.

The above discussion is for the case of generic computational graphs. How do we apply these ideas to neural networks? In the case of neural networks, one can easily compute $\frac{\partial L}{\partial o}$ in terms of the known value of o (by running the input through the network). This derivative is propagated backwards using the local partial derivatives $z(i, j)$, depending on which variables in the neural network are used as intermediate variables. For example, when the post-activation values inside nodes are treated as nodes of the computational graph, the value of $z(i, j)$ is the product of the weight of edge (i, j) and the local derivative of the activation at node j. On the other hand, if we use the pre-activation variables as the nodes of the computational graph, the value of $z(i, j)$ is product of the local derivative of the activation at node i and the weight of the edge (i, j). We will discuss more about the notion of pre-activation variables and post-activation variables in a neural network with the use of an example slightly later (Figure 3.6). We can even create computational graphs containing both pre-activation and post-activation variables to *decouple* linear operations from activation functions. All these methods are equivalent, and will be discussed in the upcoming sections.

3.2.3 Backpropagation with Post-Activation Variables

In this section, we show how to instantiate the aforementioned approach by considering a computational graph in which the nodes contain the post-activation variables in a neural network. These are the same as the hidden variables of different layers.

The backpropagation algorithm first uses a *forward phase* in order to compute the output and the loss. Therefore, the forward phase sets up the initialization for the dynamic programming recurrence, and also the intermediate variables that will be needed in the backwards phase. As discussed in the previous section, the backwards phase uses the dynamic programming recurrence based on the multivariable chain rule of differential calculus. We describe the forward and backward phases as follows:

Forward phase: In the forward phase, a particular input vector is used to compute the values of each hidden layer based on the current values of the weights; the name "forward phase" is used because such computations naturally cascade forward across the layers. The goal of the forward phase is to compute all the intermediate hidden and output variables for a given input. These values will be required during the backward phase. At the point at which the computation is completed, the value of the output o is computed, as is the derivative of the loss function L with respect to this output. The loss is typically a function of all the outputs in the presence of multiple nodes; therefore, the derivatives with respect to all outputs are computed. For now, we will consider the case of a single output node o for simplicity, and then discuss the straightforward generalization to multiple outputs.

Backward phase: The backward phase computes the gradient of the loss function with respect to various weights. The first step is to compute the derivative $\frac{\partial L}{\partial o}$. If the network has multiple outputs, then this value is computed for each output. This sets up the initialization of the gradient computation. Subsequently, the derivatives are propagated in the backwards direction using the multivariable chain rule of Equation 3.3.

Consider a path is denoted by the sequence of hidden units $h_1, h_2, \ldots, h_k$ followed by output o. The weight of the connection from hidden unit h_r to h_{r+1} is denoted by $w_{(h_r, h_{r+1})}$. If a single path exists in the network, it would be a simple matter to backpropagate the derivative of the loss function L with respect to the weights along this path. In most cases, an exponentially large number of paths will exist in the network from any node h_r to the output node o. As shown in Lemma 3.2.1, the partial derivative can be computed by aggregating the products of partial derivatives over all paths from h_r to o. When a set $\mathcal{P}$ of paths exist from h_r to o, one can write the loss derivative as follows:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial L}{\partial o} \cdot \left[\sum_{[h_r, h_{r+1}, \ldots h_k, o] \in \mathcal{P}} \frac{\partial o}{\partial h_k} \prod_{i=r}^{k-1} \frac{\partial h_{i+1}}{\partial h_i} \right]}_{\text{Backpropagation computes } \Delta(h_r, o) = \frac{\partial L}{\partial h_r}} \frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}} \tag{3.9}$$

The computation of $\frac{\partial h_r}{\partial w_{(h_{r-1},h_r)}}$ on the right-hand side is useful in converting a recursively computed partial derivative with respect to *layer activations* into a partial derivative with respect to the *weights*. The path-aggregated term above [annotated by $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$] is very similar to the quantity $S(i, o) = \frac{\partial o}{\partial y_i}$ discussed in Section 3.2.2. As in that section, the idea is to first compute $\Delta(h_k, o)$ for nodes h_k closest to o, and then recursively compute these values for nodes in earlier layers in terms of nodes in later layers. The value of $\Delta(o, o) = \frac{\partial L}{\partial o}$ is computed as the initial point of the recursion. Subsequently, this computation is propagated in the backwards direction with dynamic programming updates (similar to Equation 3.8). The multivariable chain rule directly provides the recursion for $\Delta(h_r, o)$:

$$\Delta(h_r, o) = \frac{\partial L}{\partial h_r} = \sum_{h: h_r \Rightarrow h} \frac{\partial L}{\partial h} \frac{\partial h}{\partial h_r} = \sum_{h: h_r \Rightarrow h} \frac{\partial h}{\partial h_r} \Delta(h, o) \tag{3.10}$$

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 3.10. 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:

$$\frac{\partial h}{\partial h_r} = \frac{\partial h}{\partial a_h} \cdot \frac{\partial a_h}{\partial h_r} = \frac{\partial \Phi(a_h)}{\partial a_h} \cdot w_{(h_r,h)} = \Phi'(a_h) \cdot w_{(h_r,h)} \tag{3.11}$$

This value of $\frac{\partial h}{\partial h_r}$ is used in Equation 3.10 to obtain the following:

$$\Delta(h_r, o) = \sum_{h: h_r \Rightarrow h} \Phi'(a_h) \cdot w_{(h_r,h)} \cdot \Delta(h, o) \tag{3.12}$$

This recursion is repeated in the backwards direction, starting with the output node. The entire process is linear in the number of edges in the network. Note that one could also have derived Equation 3.12 by using the generic computational graph algorithm in Section 3.2.2 with respect to post-activation variables. One simply needs to set $z(i, j)$ in Equation 3.8 to the product of the weight between nodes i and j, and the activation derivative at node j.

Backpropagation can be summarized in the following steps:

1. Use a forward-pass to compute the values of all hidden units, output o, and loss L for a particular input-output pattern $(\overline{X}, y)$.

2. Initialize $\Delta(o, o)$ to $\frac{\partial L}{\partial o}$.

3. Use the recurrence of Equation 3.12 to compute each $\Delta(h_r, o)$ in the backwards direction. After each such computation, compute the gradients with respect to incident weights as follows:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \Delta(h_r, o) \cdot h_{r-1} \cdot \Phi'(a_{h_r}) \tag{3.13}$$

   The partial derivatives with respect to incident biases can be computed by using the fact that bias neurons are always activated at a value of +1. Therefore, to compute the partial derivative of the loss with respect to the bias of node h_r, we simply set h_{r-1} to 1 in the right-hand side of Equation 3.13.

4. Use the computed partial derivatives of loss function with respect to weights in order to perform stochastic gradient descent for input-output pattern $(\overline{X}, y)$.

This description of backpropagation is greatly simplified, and actual implementations have to incorporate numerous changes for efficiency and stability. For example, the gradients are computed with respect to multiple training instances at one time, and these multiple instances are referred to as a *mini-batch*. These are all backpropagated simultaneously in order to add up their local gradients and execute mini-batch stochastic gradient descent. This enhancement will be discussed in Section 3.2.8. Another difference is that we have assumed a single output. However, in many types of neural networks (e.g., multiclass perceptrons), multiple outputs exist. The description of this section can easily be generalized to multiple outputs by adding the contributions of different outputs to the loss derivatives (see Section 3.2.7).

A few observations are noteworthy. Equation 3.13 shows that the partial derivative of the loss with respect to an edge from h_{r-1} to h_r always contains h_{r-1} as a multiplicative term. The remaining portion of the multiplicative factor in Equation 3.13 is seen as a backpropagated "error." In a sense, the algorithm recursively backpropagates the errors and multiplies them with the values in the hidden layer just before the weight matrix to be updated. This is why backpropagation is sometimes understood as error propagation.

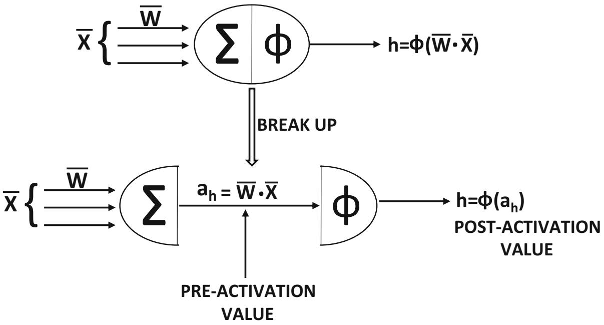


Figure 3.6: Pre- and post-activation values within a neuron

3.2.4 Backpropagation with Pre-activation Variables

In the aforementioned discussion, the values $h_1 \ldots h_k$ along a path are used to compute the chain rule. However, one can also use the values *before* computing the activation function $\Phi(\cdot)$ in order to define the chain rule. In other words, the gradients are computed with respect to the pre-activation values of the hidden variables, which are then propagated backwards. This alternative approach to backpropagation is how it is presented in most textbooks.

The pre-activation value of the hidden variable h_r is denoted by a_{h_r}, where:

$$h_r = \Phi(a_{h_r}) \tag{3.14}$$

Figure 3.6 shows the distinction between pre- and post-activation values. In such a case, we can rewrite Equation 3.9 as follows:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \underbrace{\frac{\partial L}{\partial o} \cdot \Phi'(a_o) \cdot \left[\sum_{[h_r,h_{r+1},\ldots h_k,o] \in \mathcal{P}} \frac{\partial a_o}{\partial a_{h_k}} \prod_{i=r}^{k-1} \frac{\partial a_{h_{i+1}}}{\partial a_{h_i}} \right]}_{\text{Backpropagation computes } \delta(h_r,o) = \frac{\partial L}{\partial a_{h_r}}} h_{r-1} \tag{3.15}$$

We have introduced the notation $\delta()$ to enable recurrence in this case. Note that the recurrence for $\Delta(h_r, o) = \frac{\partial L}{\partial h_r}$ uses the hidden values *after* each activation as intermediate variables in the chain rule, whereas the recurrence for $\delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}}$ uses the hidden values *before* activation. Like Equation 3.10, we can obtain the following recurrence equations:

$$\delta(h_r, o) = \frac{\partial L}{\partial a_{h_r}} = \sum_{h:h_r \Rightarrow h} \frac{\partial L}{\partial a_h} \frac{\partial a_h}{\partial a_{h_r}} = \sum_{h:h_r \Rightarrow h} \frac{\partial a_h}{\partial a_{h_r}} \delta(h, o) \tag{3.16}$$

One can use the chain rule to compute the expression for $\frac{\partial a_h}{\partial a_{h_r}}$ on the right-hand side of Equation 3.16:

$$\frac{\partial a_h}{\partial a_{h_r}} = \frac{\partial a_h}{\partial h_r} \cdot \frac{\partial h_r}{\partial a_{h_r}} = w_{(h_r,h)} \cdot \frac{\partial \Phi(a_{h_r})}{\partial a_{h_r}} = \Phi'(a_{h_r}) \cdot w_{(h_r,h)} \tag{3.17}$$

By substituting the computed expression for $\frac{\partial a_h}{\partial a_{h_r}}$ in the right-hand side of Equation 3.16, we obtain the following:

$$\delta(h_r, o) = \Phi'(a_{h_r}) \sum_{h:h_r \Rightarrow h} w_{(h_r,h)} \cdot \delta(h, o) \tag{3.18}$$

Equation 3.18 can also be derived by using pre-activation variables in the generic computational graph algorithm of Section 3.2.2. One simply needs to set $z(i, j)$ in Equation 3.8 to the product of the weight between nodes i and j, and the activation derivative at node i.

One advantage of this recurrence condition over the one obtained using post-activation variables is that the activation gradient is outside the summation, and therefore we can easily compute the specific form of the recurrence for each type of activation function at node h_r. Furthermore, since the activation gradient is outside the summation, one can simplify the backpropagation computation by decoupling the effect of the activation function and that of the linear transformation in backpropagation updates. The simplified and decoupled view will be discussed in more detail in Section 3.2.6, and it uses *both* pre-activation and post-activation variables for the dynamic programming recursion. This simplified approach represents how backpropagation is actually implemented in real systems. From an implementation point of view, decoupling the linear transformation from the activation function is helpful, because the linear portion is a simple matrix multiplication and the activation portion is an elementwise multiplication. Both can be implemented efficiently on all types of matrix-friendly hardware (such as graphics processor units).

The backpropagation process can now be described as follows:

1. Use a forward-pass to compute the values of all hidden units, output o, and loss L for a particular input-output pattern $(\overline{X}, y)$.

2. Initialize $\frac{\partial L}{\partial a_o} = \delta(o, o)$ to $\frac{\partial L}{\partial o} \cdot \Phi'(a_o)$.

3. Use the recurrence of Equation 3.18 to compute each $\delta(h_r, o)$ in the backwards direction. After each such computation, compute the gradients with respect to incident weights as follows:

$$\frac{\partial L}{\partial w_{(h_{r-1},h_r)}} = \delta(h_r, o) \cdot h_{r-1} \tag{3.19}$$

The partial derivatives with respect to incident biases can be computed by using the fact that bias neurons are always activated at a value of +1. Therefore, to compute the partial derivative of the loss with respect to the bias of node h_r, we simply set h_{r-1} to 1 in the right-hand side of Equation 3.19.

4. Use the computed partial derivatives of loss function with respect to weights in order to perform stochastic gradient descent for input-output pattern $(\overline{X}, y)$.

The main difference of this (more common) variant of the backpropagation algorithm is in terms of the way in which the recursion is written, because pre-activation variables have been used for dynamic programming. Both the pre- and post-activation variants of backpropagation are mathematically equivalent (see Exercise 9). We have chosen to show both variations of backpropagation in order to emphasize the fact that one can use dynamic programming in a variety of ways to derive equivalent equations. An even more simplified view of backpropagation, in which *both* pre-activation and post-activation variables are used, is provided in Section 3.2.6.

3.2.5 Examples of Updates for Various Activations

One advantage of Equation 3.18 is that we can compute the specific types of updates for various nodes. In the following, we provide the instantiation of Equation 3.18 for different types of nodes:

$$\delta(h_r, o) = \sum_{h: h_r \Rightarrow h} w_{(h_r, h)} \delta(h, o) \quad \text{[Linear]}$$

$$\delta(h_r, o) = h_r(1 - h_r) \sum_{h: h_r \Rightarrow h} w_{(h_r, h)} \delta(h, o) \quad \text{[Sigmoid]}$$

$$\delta(h_r, o) = (1 - h_r^2) \sum_{h: h_r \Rightarrow h} w_{(h_r, h)} \delta(h, o) \quad \text{[Tanh]}$$

Note that the derivative of the sigmoid can be written in terms of its *output* value h_r as $h_r(1 - h_r)$. Similarly, the tanh derivative can be expressed as $(1 - h_r^2)$. The derivatives of different activation functions are discussed in Section 1.2.1.6 of Chapter 1. For the ReLU function, the value of $\delta(h_r, o)$ can be computed in case-wise fashion:

$$\delta(h_r, o) = \begin{cases} \sum_{h: h_r \Rightarrow h} w_{(h_r, h)} \delta(h, o) & \text{if } 0 < a_{h_r} \\ 0 & \text{otherwise} \end{cases}$$

A similar recurrence can be shown for the hard tanh function except that the update condition is slightly different:

$$\delta(h_r, o) = \begin{cases} \sum_{h: h_r \Rightarrow h} w_{(h_r, h)} \delta(h, o) & \text{if } -1 < a_{h_r} < 1 \\ 0 & \text{otherwise} \end{cases}$$

It is noteworthy that the ReLU and tanh are non-differentiable at exactly the condition boundaries. However, this is rarely a problem in practical settings, in which one works with finite precision.

3.2.5.1 The Special Case of Softmax

Softmax activation is a special case because the function is not computed with respect to one input, but with respect to multiple inputs. Therefore, one cannot use exactly the same type of update, as with other activation functions. As discussed in Equation 1.12 of

Chapter 1, the softmax converts k real-valued predictions $v_1 \dots v_k$ into output probabilities $o_1 \dots o_k$ using the following relationship:

$$o_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \dots, k\} \tag{3.20}$$

Note that if we try to use the chain rule to backpropagate the derivative of the loss L with respect to $v_1 \dots v_k$, then one has to compute each $\frac{\partial L}{\partial o_i}$ and also each $\frac{\partial o_i}{\partial v_j}$. This backpropagation of the softmax is greatly simplified, when we take two facts into account:

1. The softmax is almost always used in the output layer.

2. The softmax is almost always paired with the *cross-entropy loss*. Let $y_1 \dots y_k \in \{0, 1\}$ be the one-hot encoded (observed) outputs for the k mutually exclusive classes. Then, the cross-entropy loss is defined as follows:

$$L = -\sum_{i=1}^{k} y_i \log(o_i) \tag{3.21}$$

The key point is that the value of $\frac{\partial L}{\partial v_i}$ has a particularly simple form in the case of the softmax:

$$\frac{\partial L}{\partial v_i} = \sum_{j=1}^{k} \frac{\partial L}{\partial o_j} \cdot \frac{\partial o_j}{\partial v_i} = o_i - y_i \tag{3.22}$$

The reader is encouraged to work out the detailed derivation of the result above; it is tedious, but relatively straightforward algebra. The derivation is enabled by the fact that the value of $\frac{\partial o_j}{\partial v_i}$ in Equation 3.22 can be shown to be equal to $o_i(1 - o_i)$ when $i = j$ (which is the same as sigmoid), and otherwise can be shown to be equal to $-o_i o_j$(see Exercise 10).

Therefore, in the case of the softmax, one first backpropagates the gradient from the output to the layer containing $v_1 \dots v_k$. Further backpropagation can proceed according to the rules discussed earlier in this section. Note that in this case, we have decoupled the backpropagation update of the softmax activation from the backpropagation in the rest of the network, in which matrix multiplications are always included along with the activation function in the backpropagation update. In general, it is helpful to create a view of backpropagation in which the linear matrix multiplications and activation layers are decoupled because it greatly simplifies the updates. This view will be discussed in the next section.

3.2.6 A Decoupled View of Vector-Centric Backpropagation

In the previous discussion, two equivalent ways of computing the updates based on Equations 3.12 and 3.18 were provided. In each case, *one is really backpropagating through a linear matrix multiplication and an activation computation simultaneously.* The way in which we order these two coupled computations affects whether we obtain Equation 3.12 or 3.18. Unfortunately, this unnecessarily complicated view of backpropagation has proliferated in papers and textbooks since the beginning. This is, in part, because layers are traditionally defined in a neural network by combining the two separate operations of linear transformation and activation function computation.

However, in many real implementations, the linear computations and the activation computations are decoupled as separate "layers," and one separately backpropagates through the two layers. Furthermore, we use a vector-centric representation of the neural network, so that operations on vector representations of layers are vector-to-vector operations such as a matrix multiplication in a linear layer [cf. Figure 1.11(d) in Chapter 1]. This view greatly simplifies the computations. Therefore, one can create a neural network in which activation layers are alternately arranged with linear layers, as shown in Figure 3.7. Note that the activation layers can use identity activation if needed. Activation layers (usually) perform one-to-one, elementwise computations on the vector components with the activation function $\Phi(\cdot)$, whereas linear layers perform all-to-all computations by multiplying with the coefficient matrix W. Then, for each pair of matrix multiplication and activation function layers, the following forward and backward steps need to be performed:

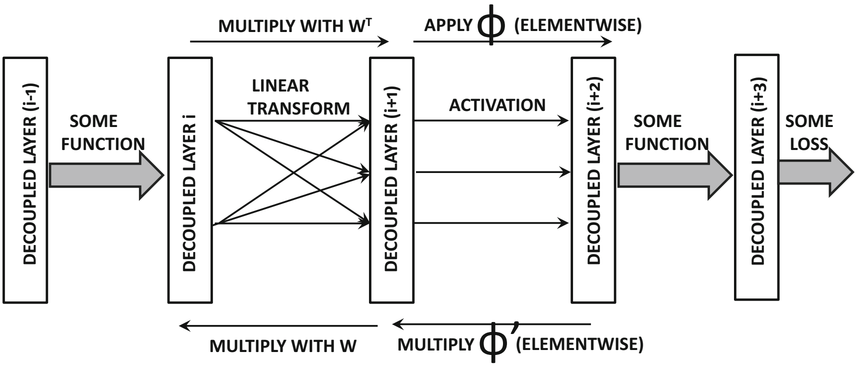


Figure 3.7: A decoupled view of backpropagation

Table 3.1: Examples of different functions and their backpropagation updates between layers i and $(i+1)$. The hidden values and gradients in layer i are denoted by $\overline{z}_i$ and $\overline{g}_i$. Some of these computations use $I(\cdot)$ as the binary indicator function.

Function	Type	Forward	Backward
Linear	Many-Many	$\overline{z}_{i+1} = W^T \overline{z}_i$	$\overline{g}_i = W \overline{g}_{i+1}$
Sigmoid	One-One	$\overline{z}_{i+1}$ =sigmoid$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (1 - \overline{z}_{i+1})$
Tanh	One-One	$\overline{z}_{i+1}$ =tanh$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot (1 - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	One-One	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	One-One	Set to ±1 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Many-One	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)
Arbitrary function $f_k(\cdot)$	Anything	$\overline{z}_{i+1}^{(k)} = f_k(\overline{z}_i)$	$\overline{g}_i = J^T \overline{g}_{i+1}$ J is Jacobian (Equation 3.23)

1. Let $\overline{z}_i$ and $\overline{z}_{i+1}$ be the column vectors of activations in the forward direction when the matrix of linear transformations from the ith to the $(i+1)$th layer is denoted by W. Furthermore, let $\overline{g}_i$ and $\overline{g}_{i+1}$ be the backpropagated vectors of gradients in the two layers. Each element of $\overline{g}_i$ is the partial derivative of the loss function with respect to

a hidden variable in the ith layer. Then, we have the following:

$$\begin{aligned}\overline{z}_{i+1} &= W^T \overline{z}_i \quad \text{[Forward Propagation]}\\ \overline{g}_i &= W \overline{g}_{i+1} \quad \text{[Backward Propagation]}\end{aligned}$$

2. Now consider a situation where the activation function $\Phi(\cdot)$ is applied to each node in layer $(i+1)$ to obtain the activations in layer $(i+2)$. Then, we have the following:

$$\begin{aligned}\overline{z}_{i+2} &= \Phi(\overline{z}_{i+1}) \quad \text{[Forward Propagation]}\\ \overline{g}_{i+1} &= \overline{g}_{i+2} \odot \Phi'(\overline{z}_{i+1}) \quad \text{[Backward Propagation]}\end{aligned}$$

   Here, $\Phi(\cdot)$ and its derivative $\Phi'(\cdot)$ are applied in element-wise fashion to vector arguments. The symbol $\odot$ indicates elementwise multiplication.

Note the extraordinary simplicity once the activation is decoupled from the matrix multiplication in a layer. The forward and backward computations are shown in Figure 3.7. Furthermore, the derivatives of $\Phi(\overline{z}_{i+1})$ can often be computed in terms of the outputs of the next layer. Based on Section 3.2.5, one can show the following for sigmoid activations:

$$\begin{aligned}\Phi'(\overline{z}_{i+1}) &= \Phi(\overline{z}_{i+1}) \odot (1 - \Phi(\overline{z}_{i+1}))\\ &= \overline{z}_{i+2} \odot (1 - \overline{z}_{i+2})\end{aligned}$$

Examples of different types of backpropagation updates for various forward functions are shown in Table 3.1. In this case, we have used layer indices of i and $(i+1)$ for *both* linear transformations and activation functions (rather than using $(i+2)$ for activation function). Note that the second to last entry in the table corresponds to the maximization function. This type of function is useful for *max-pooling* operations in convolutional neural networks. Therefore, the backward propagation operation is just like forward propagation. Given the vector of gradients in a layer, one only has to apply the operations shown in the final column of Table 3.1 to obtain the gradients with respect to the previous layer.

Some neural operations are more complex many-to-many functions than simple matrix multiplications. These cases can be handled by assuming that the kth activation in layer-$(i+1)$ is obtained by applying an arbitrary function $f_k(\cdot)$ on the vector of activations in layer-i. Then, the elements of the Jacobian are defined as follows:

$$J_{kr} = \frac{\partial f_k(\overline{z}_i)}{\partial z_i^{(r)}} \tag{3.23}$$

Here, $z_i^{(r)}$ is the rth element in $\overline{z}_i$. Let J be the matrix whose elements are J_{kr}. Then, it is easy to see that the backpropagation update from layer to layer can be written as follows:

$$\overline{g}_i = J^T \overline{g}_{i+1} \tag{3.24}$$

Writing backpropagation equations as matrix multiplications is often beneficial from an implementation-centric point of view, such as acceleration with Graphics Processor Units (cf. Section 3.7.1). Note that the elements in $\overline{g}_i$ represent gradients of the loss with respect to the *activations* in the ith layer, and therefore an additional step is needed to compute gradients with respect to the *weights*. The gradient of the loss with respect to a weight between the pth unit of the $(i-1)$th layer and the qth unit of ith layer is obtained by multiplying the pth element of $\overline{z}_{i-1}$ with the qth element of $\overline{g}_i$.

3.2.7 Loss Functions on Multiple Output Nodes and Hidden Nodes

For simplicity, the discussion above has used only a single output node at which the loss function is computed. However, in most applications, the loss function is computed over multiple output nodes O. The only difference in this case is that the value of *each* $\frac{\partial L}{\partial a_o} = \delta(o, O)$ for $o \in O$ is initialized to $\frac{\partial L}{\partial o}\Phi'(o)$. Backpropagation is then executed in order to compute $\frac{\partial L}{\partial a_h} = \delta(h, O)$ for each hidden node h.

In some forms of sparse feature learning, even the outputs of the hidden nodes have loss functions associated with them. This occurs frequently in order to encourage solutions with specific properties, such as a hidden layer that is sparse (e.g., sparse autoencoder), or a hidden layer with a specific type of regularization penalty (e.g., contractive autoencoder). The case of sparsity penalties is discussed in Section 4.4.4 of Chapter 4, and the problem of contractive autoencoders is discussed in Section 4.10.3 of Chapter 4. In such cases, the backpropagation algorithm requires only minor modifications in which the gradient flow in the backwards direction is based on all the nodes at which the loss is computed. This can be achieved by simple aggregation of the gradient flows resulting from different losses. One can view this as a special type of network in which the hidden nodes are also output nodes, and the output nodes are not restricted to the final layer of the network. At a fundamental level, the backpropagation methodology remains the same.

Consider the case in which the loss function L_{h_r} is associated with the hidden node h_r, whereas the overall loss over all nodes is L. Furthermore, let $\frac{\partial L}{\partial a_{h_r}} = \delta(h_r, N(h_r))$ denote the gradient flow from all nodes $N(h_r)$ reachable from node h_r, with which some portion of the loss is associated. In this case, the node set $N(h_r)$ might contain both nodes in the output layer as well as nodes in the hidden layer (with which a loss is associated), as long as these nodes are reachable from h_r. Therefore, the set $N(h_r)$ uses h_r as an argument. Note that the set $N(h_r)$ includes the node h_r. Then, the update of Equation 3.18 is first applied as follows:

$$\delta(h_r, N(h_r)) \Leftarrow \Phi'(a_{h_r}) \sum_{h: h_r \Rightarrow h} w_{(h_r,h)} \delta(h, N(h)) \tag{3.25}$$

This is similar to the standard backpropagation update. However, the current value of $\delta(h_r, N(h_r))$ does not yet include the contribution of h_r. Therefore, an *additional step* is executed to adjust $\delta(h_r, N(h_r))$ based on the contribution of h_r to the loss function:

$$\delta(h_r, N(h_r)) \Leftarrow \delta(h_r, N(h_r)) + \Phi'(h_r) \frac{\partial L_{h_r}}{\partial h_r} \tag{3.26}$$

It is important to keep in mind that the overall loss L is different from L_{h_r}, which is the loss specific to node h_r. Furthermore, the addition to the gradient flow in Equation 3.26 has a similar algebraic form to the value of the initialization of the output nodes. In other words, the gradient flows caused by the hidden nodes are similar to those of the output nodes. The only difference is that the computed value is added to the existing gradient flow at the hidden nodes. Therefore, the overall framework of backpropagation remains almost identical, with the main difference being that the backpropagation algorithm picks up additional contributions from the losses at the hidden nodes.

3.2.8 Mini-Batch Stochastic Gradient Descent

From the very first chapter of this book, all updates to the weights are performed in point-specific fashion, which is referred to as *stochastic* gradient descent. Such an approach is common in machine learning algorithms. In this section, we provide a justification for this choice along with related variants like *mini-batch stochastic gradient descent*. We also provide an understanding of the advantages and disadvantages of various choices.

Most machine learning problems can be recast as optimization problems over specific objective functions. For example, the objective function in neural networks can be defined in terms optimizing a loss function L, which is often a *linearly separable sum* of the loss functions on the individual training data points. For example, in a *linear regression application*, one minimizes the sum of the squared prediction errors over the training data points. In a *dimensionality reduction application*, one minimizes the sum of squared representation errors in the reconstructed training data points. One can write the loss function of a neural network in the following form:

$$L = \sum_{i=1}^{n} L_i \tag{3.27}$$

Here, L_i is the loss contributed by the ith training point. For most of the algorithms in Chapter 2, we have worked with training point-specific loss rather than the aggregate loss.

In gradient descent, one tries to minimize the loss function of the neural network by moving the parameters along the negative direction of the gradient. For example, in the case of the perceptron, the parameters correspond to $\overline{W} = (w_1 \ldots w_d)$. Therefore, one would try to compute the loss of the underlying objective function over all points simultaneously and perform gradient descent. Therefore, in traditional gradient descent, one would try to perform gradient-descent steps such as the following:

$$\overline{W} \Leftarrow \overline{W} - \alpha \left(\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2} \ldots \frac{\partial L}{\partial w_d} \right) \tag{3.28}$$

This type of derivative can also be written succinctly in vector notation (i.e., matrix calculus notation):

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L}{\partial \overline{W}} \tag{3.29}$$

For single-layer networks like the perceptron, gradient-descent is done only with respect to $\overline{W}$, whereas for larger networks, all parameters in the network need to be updated with backpropagation. The number of parameters can easily be on the order of millions in large-scale applications, and one needs to *simultaneously* run all examples forwards and backwards through the network in order to compute the backpropagation updates. It is, however, impractical to simultaneously run all examples through the network to compute the gradient with respect to the *entire data set* in one shot. Note that even the memory requirements of all intermediate/final predictions *for each training instance* would need to be maintained by gradient descent. This can be exceedingly large in most practical settings. At the beginning of the learning process, the weights are often incorrect to such a degree that even a small sample of points can be used to create an excellent estimate of the gradient's direction. The additive effect of the updates created from such samples can often provide an accurate direction of movement. This observation provides a practical foundation for the success of the stochastic gradient-descent method and its variants.

Since the loss function of most optimization problems can be expressed as a linear sum of the losses with respect to individual points (cf. Equation 3.27), it is easy to show the following:

$$\frac{\partial L}{\partial \overline{W}} = \sum_{i=1}^{n} \frac{\partial L_i}{\partial \overline{W}} \tag{3.30}$$

In this case, updating the full gradient with respect to all the points sums up the individual point-specific effects. Machine learning problems inherently have a high level of redundancy between the knowledge captured by different training points, and one can often more efficiently undertake the learning process with the point-specific updates of stochastic gradient descent:

$$\overline{W} \Leftarrow \overline{W} - \alpha \frac{\partial L_i}{\partial \overline{W}} \tag{3.31}$$

This type of gradient descent is referred to as *stochastic* because one cycles through the points in some random order. Note that the long-term effect of repeated updates is approximately the same, although each update in stochastic gradient descent can only be viewed as a probabilistic approximation. Each local gradient can be computed efficiently, which makes stochastic gradient descent fast, albeit at the expense of accuracy in gradient computation. However, one interesting property of stochastic gradient descent is that even though it might not perform as well on the training data (compared to gradient descent), it often performs comparably (and sometimes even better) on the test data [171]. As you will learn in Chapter 4, stochastic gradient descent has the indirect effect of regularization. However, it can occasionally provide very poor results with certain orderings of training points.

In mini-batch stochastic descent, one uses a batch $B = \{j_1 \ldots j_m\}$ of training points for the update:

$$\overline{W} \Leftarrow \overline{W} - \alpha \sum_{i \in B} \frac{\partial L_i}{\partial \overline{W}} \tag{3.32}$$

Mini-batch stochastic gradient descent often provides the best trade-off between stability, speed, and memory requirements. When using mini-batch stochastic gradient descent, the outputs of a layer are matrices instead of vectors, and forward propagation requires the multiplication of the weight matrix with the activation matrix. The same is true for backward propagation in which matrices of gradients are maintained. Therefore, the implementation of mini-batch stochastic gradient descent increases the memory requirements, which is a key limiting factor on the size of the mini-batch.

The size of the mini-batch is therefore regulated by the amount of memory available on the particular hardware architecture at hand. Keeping a batch size that is too small also results in constant overheads, which is inefficient even from a computational point of view. Beyond a certain batch size (which is typically of the order of a few hundred points), one does not gain much in terms of the accuracy of gradient computation. It is common to use powers of 2 as the size of the mini-batch, because this choice often provides the best efficiency on most hardware architectures; commonly used values are 32, 64, 128, or 256. Although the use of mini-batch stochastic gradient descent is ubiquitous in neural network learning, most of this book will use a single point for the update (i.e., pure stochastic gradient descent) for simplicity in presentation.

3.2.9 Backpropagation Tricks for Handling Shared Weights

A very common approach for regularizing neural networks is to use *shared weights.* The basic idea is that if one has some semantic insight that a similar function will be computed in different nodes of the network, then the weights associated with those nodes will be constrained to be the same value. Some examples are as follows:

1. In an autoencoder simulating PCA (cf. Section 2.5.1.3 of Chapter 2), the weights in the input layer and the output layer are shared.

2. In a recurrent neural network for text (cf. Chapter 7), the weights in different temporal layers are shared, because it is assumed that the *language model* at each time-stamp is the same.

3. In a convolutional neural network, the same grid of weights (corresponding to a visual field) is used over the entire spatial extent of the neurons (cf. Chapter 8).

Sharing weights in a semantically insightful way is one of the key tricks to successful neural network design. When one can identify the insight that the function computed at two nodes ought to be similar, it makes sense to use the same set of weights in that pair of nodes.

At first sight, it might seem to be an onerous task to compute the gradient of the loss with respect to the shared weights in these different regions of the network, because the different uses of the weights would also influence one another in an unpredictable way in the computational graph. However, backpropagation with respect to shared weights turns out to be mathematically simple.

Let w be a weight, which is shared at T different nodes in the network, and the corresponding copies of the weights at these nodes be denoted by $w_1 \ldots w_T$. Let the loss function be L. Then, it is easy to use the chain rule to show the following:

$$\begin{aligned}\frac{\partial L}{\partial w} &= \sum_{i=1}^{T} \frac{\partial L}{\partial w_i} \cdot \underbrace{\frac{\partial w_i}{\partial w}}_{=1} \\ &= \sum_{i=1}^{T} \frac{\partial L}{\partial w_i}\end{aligned}$$

In other words, all we have to do is to pretend that these weights are independent, compute their derivatives, and add them! Therefore, we simply have to execute the backpropagation algorithm without any change and then sum up the gradients of different copies of the shared weight. This simple observation is used at many places in neural network learning. It also forms the basis of the learning algorithm in recurrent neural networks.

3.2.10 Checking the Correctness of Gradient Computation

The backpropagation algorithm is quite complex, and one might occasionally check the correctness of gradient computation. This can be performed easily with the use of numerical methods. Consider a particular weight w of a randomly selected edge in the network. Let $L(w)$ be the current value of the loss. The weight of this edge is perturbed by adding a small amount $\epsilon > 0$ to it. Then, the forward algorithm is executed with this perturbed weight and the loss $L(w + \epsilon)$ is computed. Then, the partial derivative of the loss with respect to w can be shown to be the following:

$$\frac{\partial L(w)}{\partial w} \approx \frac{L(w+\epsilon) - L(w)}{\epsilon} \tag{3.33}$$

When the partial derivatives do not match closely enough, it is easy to detect that an error must have occurred in computation. One needs to perform the above estimation for only two or three checkpoints in the training process, which is quite efficient. However, it might be advisable to perform the checking over a large subset of the parameters at these checkpoints. One problem is in determining when the gradients are "close enough," especially when one has no idea about the absolute magnitudes of these values. This is achieved by using relative ratios.

Let the backpropagation-determined derivative be denoted by G_e, and the aforementioned estimation be denoted by G_a. Then, the relative ratio ρ is defined as follows:

$$\rho = \frac{|G_e - G_a|}{|G_e + G_a|} \tag{3.34}$$

Typically, the ratio should be less than 10^{-6}, although for some activation functions like the ReLU in which sharp changes in derivatives occur at particular points, it is possible for the numerical gradient to be different from the computed gradient. In such cases, the ratio should still be less than 10^{-3}. One can use this numerical approximation to test various edges and check the correctness of their gradients. If there are millions of parameters, then one can test a sample of the derivatives for a quick check of correctness. It is also advisable to perform this check at two or three points in the training because the checks at initialization might correspond to special cases that do not generalize to arbitrary points in the parameter space.

3.3 Setup and Initialization Issues

There are several important issues associated with the setup of the neural network, preprocessing, and initialization. First, the *hyperparameters* of the neural network (such as the learning rates and regularization parameters) need to be selected. Feature preprocessing and initialization can also be rather important. Neural networks tend to have larger parameter spaces compared to other machine learning algorithms, which magnifies the effect of preprocessing and initialization in many ways. In the following, we will discuss the basic methods used for feature preprocessing and initialization. Strictly speaking, advanced methods like *pretraining* can also be considered initialization techniques. However, these techniques require a deeper understanding of the model generalization issues associated with neural network training. For this reason, discussion on this topic will be deferred to the next chapter.

3.3.1 Tuning Hyperparameters

Neural networks have a large number of *hyperparameters* such as the learning rate, the weight of regularization, and so on. The term “hyperparameter” is used to specifically refer to the parameters regulating the design of the model (like learning rate and regularization), and they are different from the more fundamental parameters representing the weights of connections in the neural network. In Bayesian statistics, the notion of hyperparameter is used to control the prior distribution, although we use this definition in a somewhat loose sense here. In a sense, there is a two-tiered organization of parameters in the neural network, in which primary model parameters like weights are optimized with backpropagation only after fixing the hyperparameters either manually or with the use of a *tuning* phase. As we will discuss in Section 4.3 of Chapter 4, the hyperparameters should not be tuned using the same data used for gradient descent. Rather, a portion of the data is held out as validation data, and the performance of the model is tested on the validation set with various choices of hyperparameters. This type of approach ensures that the tuning process does not overfit to the training data set (while providing poor test data performance).

How should the candidate hyperparameters be selected for testing? The most well-known technique is *grid search*, in which a set of values is selected for each hyperparameter. In the most straightforward implementation of grid search, all combinations of selected values of the hyperparameters are tested in order to determine the optimal choice. One issue with this procedure is that the number of hyperparameters might be large, and the number of points in the grid increases *exponentially* with the number of hyperparameters. For example, if we have 5 hyperparameters, and we test 10 values for each hyperparameter, the training procedure needs to be executed $10^5 = 100000$ times to test its accuracy. Although one does not run such testing procedures to completion, the number of runs is still too large to be reasonably executed for most settings of even modest size. Therefore, a commonly used trick is to first work with coarse grids. Later, when one narrows down to a particular range of interest, finer grids are used. One must be careful when the optimal hyperparameter selected is at the edge of a grid range, because one would need to test beyond the range to see if better values exist.

The testing approach may at times be too expensive even with the coarse-to-fine-grained process. It has been pointed out [37] that grid-based hyperparameter exploration is not necessarily the best choice. In some cases, it makes sense to randomly sample the hyperparameters uniformly within the grid range. As in the case of grid ranges, one can perform multi-resolution sampling, where one first samples in the full grid range. One then creates a new set of grid ranges that are geometrically smaller than the previous grid ranges and centered around the optimal parameters from the previously explored samples. Sampling is repeated on this smaller box and the entire process is iteratively repeated multiple times to refine the parameters.

Another key point about sampling many types of hyperparameters is that the *logarithms* of the hyperparameters are sampled uniformly rather than the hyperparameters themselves. Two examples of such parameters include the regularization rate and the learning rate. For example, instead of sampling the learning rate α between 0.1 and 0.001, we first sample $\log(\alpha)$ uniformly between -1 and -3, and then exponentiate it to the power of 10. It is more common to search for hyperparameters in the logarithmic space, although there are some hyperparameters that should be searched for on a uniform scale.

Finally, a key point about large-scale settings is that it is sometimes impossible to run these algorithms to completion because of the large training times involved. For example, a single run of a convolutional neural network in image processing might take a couple of weeks. Trying to run the algorithm over many different choices of parameter combinations is impractical. However, one can often obtain a reasonable estimate of the broader behavior of the algorithm in a short time. Therefore, the algorithms are often run for a certain number of epochs to test the progress. Runs that are obviously poor or diverge from convergence can be quickly killed. In many cases, multiple threads of the process with different hyperparameters can be run, and one can successively terminate or add new sampled runs. In the end, only one winner is allowed to train to completion. Sometimes a few winners may be allowed to train to completion, and their predictions will be averaged as an ensemble.

A mathematically justified way of choosing for hyperparameters is the use of *Bayesian optimization* [42, 306]. However, these methods are often too slow to practically use in large-scale neural networks and remain an intellectual curiosity for researchers. For smaller networks, it is possible to use libraries such as *Hyperopt* [614], *Spearmint* [616], and *SMAC* [615].

3.3.2 Feature Preprocessing

The feature processing methods used for neural network training are not very different from those in other machine learning algorithms. There are two forms of feature preprocessing used in machine learning algorithms:

1. *Additive preprocessing and mean-centering:* It can be useful to mean-center the data in order to remove certain types of bias effects. Many algorithms in traditional machine learning (such as principal component analysis) also work with the assumption of mean-centered data. In such cases, a vector of column-wise means is subtracted from each data point. Mean-centering is often paired with *standardization*, which is discussed in the section of feature normalization.

   A second type of pre-processing is used when it is desired for all feature values to be non-negative. In such a case, the absolute value of the most negative entry of a feature is added to the corresponding feature value of each data point. The latter is typically combined with min-max normalization, which is discussed below.

2. *Feature normalization:* A common type of normalization is to divide each feature value by its standard deviation. When this type of feature scaling is combined with mean-centering, the data is said to have been *standardized.* The basic idea is that each feature is presumed to have been drawn from a *standard* normal distribution with zero mean and unit variance.

   The other type of feature normalization is useful when the data needs to be scaled in the range $(0, 1)$. Let min_j and max_j be the minimum and maximum values of the jth attribute. Then, each feature value x_{ij} for the jth dimension of the ith point is scaled by min-max normalization as follows:

$$x_{ij} \Leftarrow \frac{x_{ij} - min_j}{max_j - min_j} \tag{3.35}$$

Feature normalization often does ensure better performance, because it is common for the relative values of features to vary by more than an order of magnitude. In such cases, parameter learning faces the problem of ill-conditioning, in which the loss function has an inherent tendency to be more sensitive to some parameters than others. As we will see later in this chapter, this type of ill-conditioning affects the performance of gradient descent. Therefore, it is advisable to perform the feature scaling up front.

Whitening

Another form of feature pre-processing is referred to as *whitening*, in which the axis-system is rotated to create a new set of *de-correlated* features, each of which is scaled to unit variance. Typically, principal component analysis is used to achieve this goal.

Principal component analysis can be viewed as the application of singular value decomposition *after* mean-centering a data matrix (i.e., subtracting the mean from each column). Let D be an $n \times d$ data matrix that has already been mean-centered. Let C be the $d \times d$ co-variance matrix of D in which the (i, j)th entry is the co-variance between the dimensions i and j. Because the matrix D is mean-centered, we have the following:

$$C = \frac{D^T D}{n} \propto D^T D \tag{3.36}$$

The eigenvectors of the co-variance matrix provide the de-correlated directions in the data. Furthermore, the eigenvalues provide the variance along each of the directions. Therefore, if one uses the top-k eigenvectors (i.e., largest k eigenvalues) of the covariance matrix, most of the variance in the data will be retained and the noise will be removed. One can also choose $k = d$, but this will often result in the variances along the near-zero eigenvectors being dominated by numerical errors in the computation. It is a bad idea to include dimensions in which the variance is caused by computational errors, because such dimensions will contain little useful information for learning application-specific knowledge. Furthermore, the whitening process will scale each transformed feature to unit variance, which will blow up the errors along these directions. At the very least, it is advisable to use some threshold like 10^{-5} on the magnitude of the eigenvalues. Therefore, as a practical matter, k will rarely be exactly equal to d. Alternatively, one can add 10^{-5} to each eigenvalue for regularization before scaling each dimension.

Let P be a $d \times k$ matrix in which each column contains one of the top-k eigenvectors. Then, the data matrix D can be transformed into the k-dimensional axis system by post-multiplying with the matrix P. The resulting $n \times k$ matrix U, whose rows contain the transformed k-dimensional data points, is given by the following:

$$U = DP \tag{3.37}$$

Note that the variances of the columns of U are the corresponding eigenvalues, because this is the property of the de-correlating transformation of principal component analysis. In whitening, each column of U is scaled to unit variance by dividing it with its standard deviation (i.e., the square root of the corresponding eigenvalue). The transformed features are fed into the neural network. Since whitening might reduce the number of features, this type of preprocessing might also affect the architecture of the network, because it reduces the number of inputs.

One important aspect of whitening is that one might not want to make a pass through a large data set to estimate its covariance matrix. In such cases, the covariance matrix and columnwise means of the original data matrix can be estimated on a sample of the data. The $d \times k$ eigenvector matrix P is computed in which the columns contain the top-k eigenvectors. Subsequently, the following steps are used for each data point: (i) The mean of each column is subtracted from the corresponding feature; (ii) Each d-dimensional row vector representing a training data point (or test data point) is post-multiplied with P to create a k-dimensional row vector; (iii) Each feature of this k-dimensional representation is divided by the square-root of the corresponding eigenvalue.

The basic idea behind whitening is that data is assumed to be generated from an independent Gaussian distribution along each principal component. By whitening, one assumes that each such distribution is a *standard* normal distribution, and provides equal importance to the different features. Note that after whitening, the scatter plot of the data will roughly have a spherical shape, even if the original data is elliptically elongated with an arbitrary orientation. The idea is that the uncorrelated concepts in the data have now been scaled to equal importance (on an a priori basis), and the neural network can decide which of them to emphasize in the learning process. Another issue is that when different features are scaled very differently, the activations and gradients will be dominated by the "large" features in the initial phase of learning (if the weights are initialized randomly to values of similar magnitude). This might hurt the relative learning rate of some of the important weights in the network. The practical advantages of using different types of feature preprocessing and normalization are discussed in [278, 532].

3.3.3 Initialization

Initialization is particularly important in neural networks because of the stability issues associated with neural network training. As you will learn in Section 3.4, neural networks often exhibit stability problems in the sense that the activations of each layer either become successively weaker or successively stronger. The effect is exponentially related to the depth of the network, and is therefore particularly severe in deep networks. One way of ameliorating this effect to some extent is to choose good initialization points in such a way that the gradients are stable across the different layers.

One possible approach to initialize the weights is to generate random values from a Gaussian distribution with zero mean and a small standard deviation, such as 10^{-2}. Typically, this will result in small random values that are both positive and negative. One problem with this initialization is that it is not sensitive to the number of inputs to a specific neuron. For example, if one neuron has only 2 inputs and another has 100 inputs, the output of the former is far more sensitive to the average weight because of the additive effect of more inputs (which will show up as a much larger gradient). In general, it can be shown that the variance of the outputs linearly scales with the number of inputs, and therefore the standard deviation scales with the square root of the number of inputs. To balance this fact, each weight is initialized to a value drawn from a Gaussian distribution with standard deviation $\sqrt{1/r}$, where r is the number of inputs to that neuron. Bias neurons are always initialized to zero weight. Alternatively, one can initialize the weight to a value that is uniformly distributed in $[-1/\sqrt{r}, 1/\sqrt{r}]$.

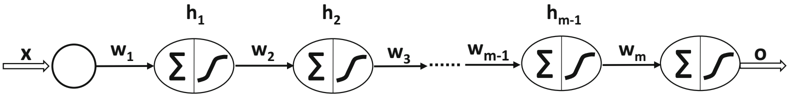


Figure 3.8: The vanishing and exploding gradient problems

More sophisticated rules for initialization consider the fact that the nodes in different layers interact with one another to contribute to output sensitivity. Let r_{in} and r_{out} respectively be the fan-in and fan-out for a particular neuron. One suggested initialization rule, referred to as *Xavier initialization* or *Glorot initialization* is to use a Gaussian distribution with standard deviation of $\sqrt{2/(r_{in} + r_{out})}$.

An important consideration in using randomized methods is that *symmetry breaking* is important. if all weights are initialized to the same value (such as 0), all updates will move in lock-step in a layer. As a result, identical features will be created by the neurons in a layer. It is important to have a source of asymmetry among the neurons to begin with.

3.4 The Vanishing and Exploding Gradient Problems

Deep neural networks have several stability issues associated with training. In particular, networks with many layers may be hard to train because of the way in which the gradients in earlier and later layers are related.

In order to understand this point, let us consider a very deep network that has a single node in each layer. We assume that there are $(m + 1)$ layers, including the non-computational input layer. The weights of the edges between the various layers are denoted by $w_1, w_2, \ldots w_m$. Furthermore, assume that the sigmoid activation function $\Phi(\cdot)$ is applied in each layer. Let x be the input, $h_1 \ldots h_{m-1}$ be the hidden values in the various layers, and o be the final output. Let $\Phi'(h_t)$ be the derivative of the activation function in hidden layer t. Let $\frac{\partial L}{\partial h_t}$ be the derivative of the loss function with respect to the hidden activation h_t. The neural architecture is illustrated in Figure 3.8. It is relatively easy to use the backpropagation update to show the following relationship:

$$\frac{\partial L}{\partial h_t} = \Phi'(h_{t+1}) \cdot w_{t+1} \cdot \frac{\partial L}{\partial h_{t+1}} \tag{3.38}$$

Since the fan-in is 1 of each node, assume that the weights are initialized from a standard normal distribution. Therefore, each w_t has an expected average magnitude of 1.

Let us examine the specific behavior of this recurrence in the case where the sigmoid activation is used. The derivative with a sigmoid with output $f \in (0, 1)$ is given by $f(1-f)$. This value takes on its maximum at $f = 0.5$, and therefore the value of $\Phi'(h_t)$ is no more than 0.25 even at its maximum. Since the absolute value of w_{t+1} is expected to be 1, it follows that each weight update will (typically) cause the value of $\frac{\partial L}{\partial h_t}$ to be less than 0.25 that of $\frac{\partial L}{\partial h_{t+1}}$. Therefore, after moving by about r layers, this value will typically be less than 0.25^r. Just to get an idea of the magnitude of this drop, if we set $r = 10$, then the gradient update magnitudes drop to 10^{-6} of their original values! Therefore, when backpropagation is used, the earlier layers will receive very small updates compared to the later layers. This problem is referred to as the *vanishing gradient problem*. Note that we could try to solve this problem by using an activation function with larger gradients and also initializing the weights to be larger. However, if we go too far in doing this, it is easy to end up in the opposite situation where the gradient *explodes* in the backward direction instead of vanishing. In general, unless we initialize the weight of every edge so that the product of the weight and the derivative of each activation is exactly 1, there will be considerable instability in the magnitudes of the partial derivatives. In practice, this is impossible with most activation functions because the derivative of an activation function will vary from iteration to iteration.

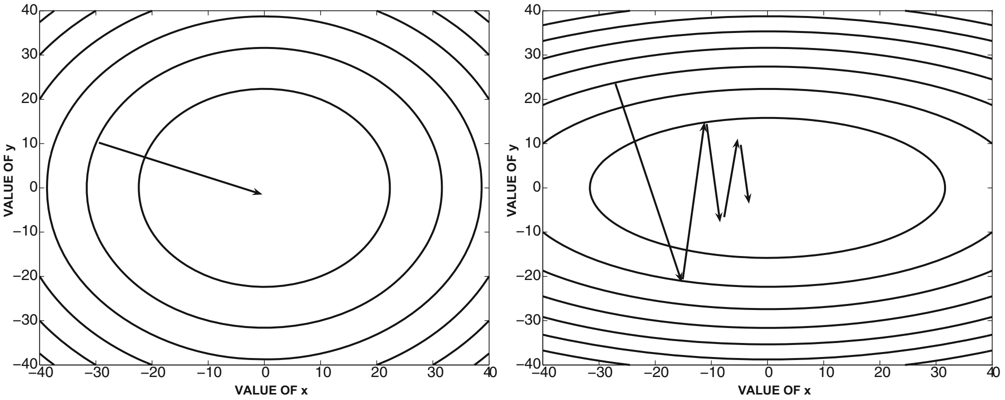
Although we have used an oversimplified example here with only one node in each layer, it is easy to generalize the argument to cases in which multiple nodes are available in each layer. In general, it is possible to show that the layer-to-layer backpropagation update includes a matrix multiplication (rather than a scalar multiplication). Just as repeated scalar multiplication is inherently unstable, so is repeated matrix multiplication. In particular, the loss derivatives in layer-$(i+1)$ are multiplied by a matrix referred to as the Jacobian (cf. Equation 3.23). The Jacobian contains the derivatives of the activations in layer-$(i+1)$ with respect to those in layer i. In certain cases like recurrent neural networks, the Jacobian is a square matrix and one can actually impose stability conditions with respect to the largest eigenvalue of the Jacobian. These stability conditions are rarely satisfied exactly, and therefore the model has an inherent tendency to exhibit the vanishing and exploding gradient problems. Furthermore, the effect of activation functions like the sigmoid tends to encourage the vanishing gradient problem. One can summarize this problem as follows:

Observation 3.4.1 *The relative magnitudes of the partial derivatives with respect to the parameters in different parts of the network tend to be very different, which creates problems for gradient-descent methods.*

In the next section, we will provide a geometric understanding of why it is natural for unstable gradient ratios to cause problems in most multivariate optimization problems, even when working in relatively simple settings.

3.4.1 Geometric Understanding of the Effect of Gradient Ratios

The vanishing and exploding gradient problems are inherent to multivariable optimization, even in cases where there are no local optima. In fact, minor manifestations of this problem are encountered in almost any convex optimization problem. Therefore, in this section, we will consider the simplest possible case of a convex, quadratic objective function with a bowl-like shape and a single global minimum. In a single-variable problem, the path of steepest descent (which is the only path of descent), will always pass through the minimum point of the bowl (i.e., optimum objective function value). However, the moment we increase the number of variables in the optimization problem from 1 to 2, this is no longer the case. The key point to understand is that *with very few exceptions, the path of steepest descent in most loss functions is only an instantaneous direction of best movement, and is not the correct direction of descent in the longer term.* In other words, small steps with "course corrections" are always needed. When an optimization problem exhibits the vanishing gradient problem, it means that the only way to reach the optimum with steepest-descent updates is by using an *extremely large number of tiny updates and course corrections*, which is obviously very inefficient.



(a) Loss function is circular bowl
$L = x^2 + y^2$

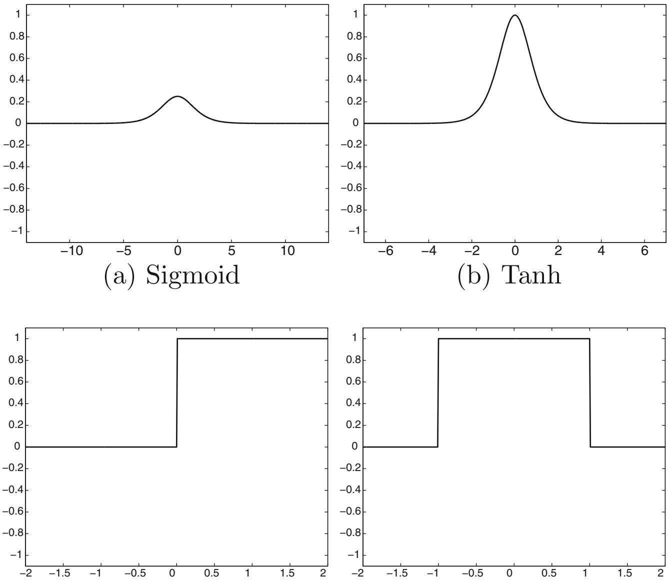
(b) Loss function is elliptical bowl
$L = x^2 + 4y^2$

Figure 3.9: The effect of the shape of the loss function on steepest-gradient descent.

In order to understand this point, we look at two bivariate loss functions in Figure 3.9. In this figure, the contour plots of the loss function are shown, in which each line corresponds to points in the XY-plane where the loss function has the same value. The direction of steepest descent is always perpendicular to this line. The first loss function is of the form $L = x^2 + y^2$, which takes the shape of a perfectly circular bowl, if one were to view the height as the objective function value. This loss function treats x and y in a symmetric way. The second loss function is of the form $L = x^2 + 4y^2$, which is an elliptical bowl. Note that this loss function is more sensitive to changes in the value of y as compared to changes in the value of x, although the specific sensitivity depends on the position of the data point.

In the case of the circular bowl of Figure 3.9(a), the gradient points directly at the optimum solution, and one can reach the optimum in a single step, as long as the correct step-size is used. This is not quite the case in the loss function of Figure 3.9(b), in which the gradients are often more significant in the y-direction compared to the x-direction. Furthermore, the gradient never points to the optimal solution, as a result of which many course corrections are needed over the descent. A salient observation is that the steps along the y-direction are large, but subsequent steps undo the effect of previous steps. On the other hand, the progress along the x-direction is consistent but tiny. Although the situation of

Figure 3.9(b) occurs in almost any optimization problem using steepest descent, the case of the vanishing gradient is an extreme manifestation[2] of this behavior. The fact that a simple quadratic bowl (which is trivial compared to the typical loss function of a deep network) shows so much oscillation with the steepest-descent method is concerning. After all, the repeated composition of functions (as implied by the underlying computational graph) is highly *unstable* in terms of the sensitivity of the output to the parameters in different parts of the network. The problem of differing relative derivatives is extraordinarily large in real neural networks, in which we have millions of parameters and gradient ratios that vary by orders of magnitude. Furthermore, many activation functions have small derivatives, which tends to encourage the vanishing gradient problem during backpropagation. As a result, the parameters in later layers with large descent components are often oscillating with large updates, whereas those in earlier layers make tiny but consistent updates. Therefore, neither the earlier nor the later layers make much progress in getting closer to the optimal solution. As a result, it is possible to get into situations where very little progress is made even after training for a long time.



(a) Sigmoid

(b) Tanh

(c) ReLU

(d) Hard Tanh

Figure 3.10: The derivatives of different activation functions are shown. Piecewise linear activation functions have local gradient values of 1.

[2] A different type of manifestation occurs in cases where the parameters in earlier and later layers are shared. In such cases, the effect of an update can be highly unpredictable because of the combined effect of different layers. Such scenarios occur in recurrent neural networks in which the parameters in later temporal layers are tied to those of earlier temporal layers. In such cases, small changes in the parameters can cause large changes in the loss function in very localized regions without any gradient-based indication in nearby regions. Such topological characteristics of the loss function are referred to as *cliffs* (cf. Section 3.5.4), and they make the problem harder to optimize because the gradient descent tends to either overshoot or undershoot.

3.4.2 A Partial Fix with Activation Function Choice

The specific choice of activation function often has a considerable effect on the severity of the vanishing gradient problem. The derivatives of the sigmoid and the tanh activation functions are illustrated in Figure 3.10(a) and (b), respectively. The sigmoid activation function never has a gradient of more than 0.25, and therefore it is very prone to the vanishing gradient problem. Furthermore, it *saturates* at large absolute values of the argument, which refers to the fact that the gradient is almost 0. In such cases, the weights of the neuron change very slowly. Therefore, a few such activations within the network can significantly affect the gradient computations. The tanh function fares better than the sigmoid function because it has a gradient of 1 near the origin, but the gradient saturates rapidly at increasingly large absolute values of the argument. Therefore, the tanh function will also be susceptible to the vanishing gradient problem.

In recent years, the use of the sigmoid and the tanh activation functions has been increasingly replaced with the ReLU and the hard tanh functions. The ReLU is also faster to train because its gradient is efficient to compute. The derivatives of the ReLU and the hard tanh functions are shown in Figure 3.10(c) and (d), respectively. It is evident that these functions take on the derivative of 1 in certain intervals, although they might have zero gradient in others. As a result, the vanishing gradient problem tends to occur less often, as long as most of these units operate within the intervals where the gradient is 1. In recent years, these piecewise linear variants have become far more popular than their smooth counterparts. Note that the replacement of the activation function is only a partial fix because the matrix multiplication across layers still causes a certain level of instability. Furthermore, the piecewise linear activations introduce the new problem of *dead neurons.*

3.4.3 Dying Neurons and "Brain Damage"

It is evident from Figure 3.10(c) and (d) that the gradient of the ReLU is zero for negative values of its argument. This can occur for a variety of reasons. For example, consider the case where the input into a neuron is always nonnegative, whereas all the weights have somehow been initialized to negative values. Therefore, the output will be 0. Another example is the case where a high learning rate is used. In such a case, the pre-activation values of the ReLU can jump to a range where the gradient is 0 irrespective of the input. In other words, high learning rates can "knock out" ReLU units. In such cases, the ReLU might not fire for any data instance. Once a neuron reaches this point, the gradient of the loss with respect to the weights just before the ReLU will always be zero. In other words, the weights of this neuron will never be updated further during training. Furthermore, its output will not vary across different choices of inputs and therefore will not play a role in discriminating between different instances. Such a neuron can be considered *dead*, which is considered a kind of permanent "brain damage" in biological parlance. The problem of dying neurons can be partially ameliorated by using learning rates that are somewhat modest. Another fix is to use the *leaky ReLU*, which allows the neurons outside the active interval to leak some gradient backwards.

3.4.3.1 Leaky ReLU

The leaky ReLU is defined using an additional parameter $\alpha \in (0, 1)$:

$$\Phi(v) = \begin{cases} \alpha \cdot v & v \leq 0 \\ v & \text{otherwise} \end{cases} \tag{3.39}$$

Although α is a hyperparameter chosen by the user, it is also possible to learn it. Therefore, at negative values of v, the leaky ReLU can still propagate some gradient backwards, albeit at a reduced rate defined by $\alpha < 1$.

The gains with the leaky ReLU are not guaranteed, and therefore this fix is not completely reliable. A key point is that dead neurons are not always a problem, because they represent a kind of pruning to control the precise structure of the neural network. Therefore, a certain level of dropping of neurons can be viewed as a part of the learning process. After all, there are limitations to our ability to tune the number of neurons in each layer. Dying neurons do a part of this tuning for us. Indeed, the intentional pruning of *connections* is sometimes used as a strategy for regularization [282]. Of course, if a very large fraction of the neurons in the network are dead, that can be a problem as well because much of the neural network will be inactive. Furthermore, it is undesirable for too many neurons to be knocked out during the early training phases, when the model is very poor.

3.4.3.2 Maxout

A recently proposed solution is the use of *maxout networks* [148]. The idea in the maxout unit is to have two coefficient vectors $\overline{W_1}$ and $\overline{W_2}$ instead of a single one. Subsequently, the activation used is $\max\{\overline{W_1} \cdot \overline{X}, \overline{W_2} \cdot \overline{X}\}$. In the event that bias neurons are used, the maxout activation is $\max\{\overline{W_1} \cdot \overline{X} + b_1, \overline{W_2} \cdot \overline{X} + b_2\}$. One can view the maxout as a generalization of the ReLU, because the ReLU is obtained by setting one of the coefficient vectors to 0. Even the leaky ReLU can be shown to be a special case of maxout, in which we set $\overline{W_2} = \alpha \overline{W_1}$ for $\alpha \in (0, 1)$. Like the ReLU, the maxout function is piecewise linear. However, it does not saturate at all, and is linear almost everywhere. In spite of its linearity, it has been shown [148] that maxout networks are universal function approximators. Maxout has advantages over the ReLU, and it enhances the performance of ensemble methods like *Dropout* (cf. Section 4.5.4 of Chapter 4). The only drawback with the use of maxout is that it doubles the number of required parameters.

3.5 Gradient-Descent Strategies

The most common method for parameter learning in neural networks is the *steepest-descent method*, in which the gradient of the loss function is used to make parameter updates. In fact, all the discussions in previous chapters are based on this assumption. As discussed in the earlier section, the steepest-gradient method can sometimes behave unexpectedly because it does not always point in the best direction of improvement, when steps of finite size are considered. The steepest-descent direction is the optimal direction only from the perspective of infinitesimal steps. A steepest-descent direction can sometimes become an ascent direction after a small update in parameters. As a result, many course corrections are needed. A specific example of this phenomenon is discussed in Section 3.4.1 in which minor differences in sensitivity to different features can cause a steepest-descent algorithm to have oscillations. The problem of oscillation and zigzagging is quite ubiquitous whenever the steepest-descent direction moves along a direction of *high curvature* in the loss function. The most extreme manifestation of this problem occurs in the case of extreme ill-conditioning, for which the partial derivatives of the loss are wildly different with respect to the different optimization variables. In this section, we will discuss several clever learning strategies that work well in these ill-conditioned settings.

3.5.1 Learning Rate Decay

A constant learning rate is not desirable because it poses a dilemma to the analyst. The dilemma is as follows. A lower learning rate used early on will cause the algorithm to take too long to come even close to an optimal solution. On the other hand, a large initial learning rate will allow the algorithm to come reasonably close to a good solution at first; however, the algorithm will then oscillate around the point for a very long time, or diverge in an unstable way, if the high rate of learning is maintained. In either case, maintaining a constant learning rate is not ideal. Allowing the learning rate to decay over time can naturally achieve the desired learning-rate adjustment to avoid these challenges.

The two most common decay functions are *exponential decay* and *inverse decay*. The learning rate α_t can be expressed in terms of the initial decay rate α_0 and epoch t as follows:

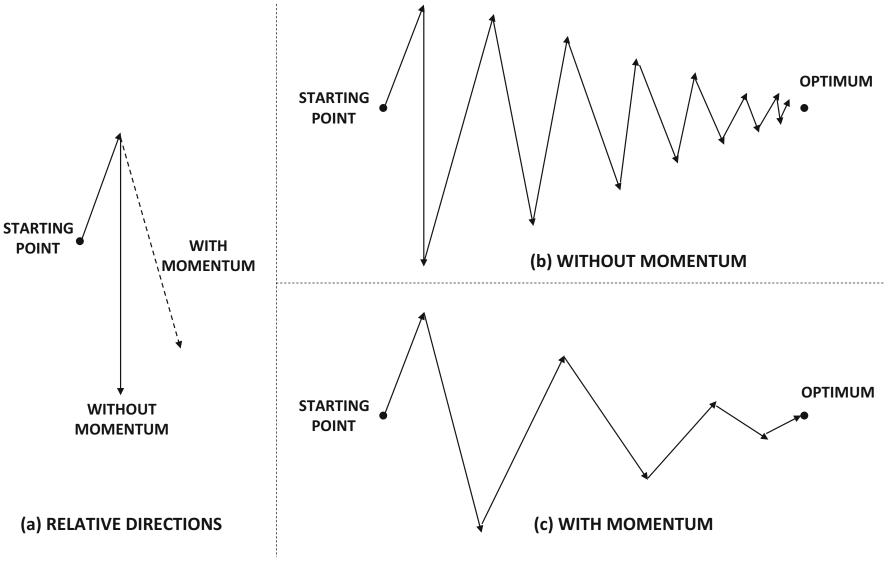


Figure 3.11: Effect of momentum in smoothing zigzag updates

$$\alpha_t = \alpha_0 \exp(-k \cdot t) \quad \text{[Exponential Decay]}$$
$$\alpha_t = \frac{\alpha_0}{1 + k \cdot t} \quad \text{[Inverse Decay]}$$

The parameter k controls the rate of the decay. Another approach is to use step decay in which the learning rate is reduced by a particular factor every few epochs. For example, the learning rate might be multiplied by 0.5 every 5 epochs. A common approach is to track the loss on a held-out portion of the training data set, and reduce the learning rate whenever this loss stops improving. In some cases, the analyst might even babysit the learning process, and use an implementation in which the learning rate can be changed manually depending on the progress. This type of approach can be used with simple implementations of gradient descent, although it does not address many of the other problematic issues.

3.5.2 Momentum-Based Learning

Momentum-based techniques recognize that zigzagging is a result of highly contradictory steps that cancel out one another and reduce the *effective* size of the steps in the correct (long-term) direction. An example of this scenario is illustrated in Figure 3.9(b). Simply attempting to increase the size of the step in order to obtain greater movement in the correct direction might actually move the current solution even further away from the optimum solution. In this point of view, it makes a lot more sense to move in an "averaged" direction of the last few steps, so that the zigzagging is smoothed out.

In order to understand this point, consider a setting in which one is performing gradient-descent with respect to the parameter vector $\overline{W}$. The normal updates for gradient-descent with respect to loss function L (defined over a mini-batch of instances) are as follows:

$$\overline{V} \Leftarrow -\alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

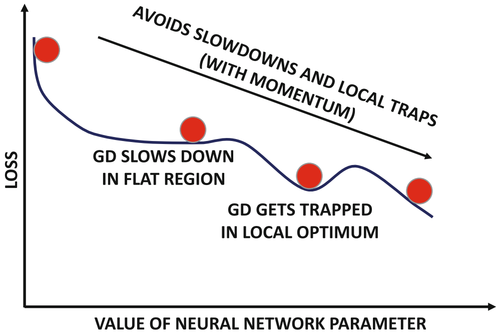


Figure 3.12: Effect of momentum in navigating complex loss surfaces. The annotation "GD" indicates pure gradient descent without momentum. Momentum helps the optimization process retain speed in flat regions of the loss surface and avoid local optima.

Here, α is the learning rate. In momentum-based descent, the vector $\overline{V}$ is modified with exponential smoothing, where $\beta \in (0, 1)$ is a smoothing parameter:

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Larger values of β help the approach pick up a consistent velocity $\overline{V}$ in the correct direction. Setting $\beta = 0$ specializes to straightforward mini-batch gradient-descent. The parameter β is also referred to as the *momentum parameter* or the *friction parameter*. The word "friction" is derived from the fact that small values of β act as "brakes," much like friction.

With momentum-based descent, the learning is accelerated, because one is generally moving in a direction that often points closer to the optimal solution and the useless "sideways" oscillations are muted. The basic idea is to give greater preference to *consistent* directions over multiple steps, which have greater importance in the descent. This allows the use of larger steps in the correct direction without causing overflows or "explosions" in the sideways direction. As a result, learning is accelerated. An example of the use of momentum is illustrated in Figure 3.11. It is evident from Figure 3.11(a) that momentum increases the relative component of the gradient in the correct direction. The corresponding effects on the updates are illustrated in Figure 3.11(b) and (c). It is evident that momentum-based updates can reach the optimal solution in fewer updates.

The use of momentum will often cause the solution to slightly overshoot in the direction where velocity is picked up, just as a marble will overshoot when it is allowed to roll down a bowl. However, with the appropriate choice of β, it will still perform better than a situation in which momentum is not used. The momentum-based method will generally perform better because the marble gains speed as it rolls down the bowl; the quicker arrival at the optimal solution more than compensates for the overshooting of the target. Overshooting is desirable to the extent that it helps avoid local optima. Figure 3.12, which shows a marble rolling down a complex loss surface (picking up speed as it rolls down), illustrates this concept. The marble's gathering of speed helps it efficiently navigate flat regions of the loss surface. The parameter β controls the amount of friction that the marble encounters while rolling down the loss surface. While increased values of β help in avoiding local optima, it might also increase oscillation at the end. In this sense, the momentum-based method has a neat interpretation in terms of the physics of a marble rolling down a complex loss surface.

3.5.2.1 Nesterov Momentum

The Nesterov momentum [353] is a modification of the traditional momentum method in which the gradients are computed at a point that would be reached after executing a β-discounted version of the previous step again (i.e., the momentum portion of the current step). This point is obtained by multiplying the previous update vector $\overline{V}$ with the friction parameter β and then computing the gradient at $\overline{W} + \beta\overline{V}$. The idea is that this corrected gradient uses a better understanding of how the gradients will change because of the momentum portion of the update, and incorporates this information into the gradient portion of the update. Therefore, one is using a certain amount of lookahead in computing the updates. Let us denote the loss function by $L(\overline{W})$ at the current solution $\overline{W}$. In this case, it is important to explicitly denote the argument of the loss function because of the way in which the gradient is computed. Therefore, the update may be computed as follows:

$$\overline{V} \Leftarrow \beta\overline{V} - \alpha\frac{\partial L(\overline{W} + \beta\overline{V})}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Note that the *only* difference from the standard momentum method is in terms of *where* the gradient is computed. Using the value of the gradient a little further along the previous update can lead to faster convergence. In the previous analogy of the rolling marble, such an approach will start applying the "brakes" on the gradient-descent procedure when the marble starts reaching near the bottom of the bowl, because the lookahead will "warn" it about the reversal in gradient direction.

The Nesterov method works only in mini-batch gradient descent with modest batch sizes; using very small batches is a bad idea. In such cases, it can be shown that the Nesterov method reduces the error to $O(1/t^2)$ after t steps, as compared to an error of $O(1/t)$ in the momentum method.

3.5.3 Parameter-Specific Learning Rates

The basic idea in the momentum methods of the previous section is to leverage the *consistency* in the gradient direction of certain parameters in order to speed up the updates. This goal can also be achieved more explicitly by having different learning rates for different parameters. The idea is that parameters with large partial derivatives are often oscillating and zigzagging, whereas parameters with small partial derivatives tend to be more consistent but move in the same direction. An early method, which was proposed in this direction, was the *delta-bar-delta* method [217]. This approach tracks whether the sign of each partial derivative changes or stays the same. If the sign of a partial derivative stays consistent, then it is indicative of the fact that the direction is correct. In such a case, the partial derivative in that direction increases. On the other hand, if the sign of the partial derivative flips all the time, then the partial derivative decreases. However, this kind of approach is designed for gradient descent rather than stochastic gradient descent, because the errors in stochastic gradient descent can get magnified. Therefore, a number of methods have been proposed that can work well even when the mini-batch method is used.

3.5.3.1 AdaGrad

In the AdaGrad algorithm [108], one keeps track of the aggregated squared magnitude of the partial derivative with respect to each parameter over the course of the algorithm. The square-root of this value is *proportional* to the root-mean-square slope for that parameter (although the absolute value will increase with the number of epochs because of successive aggregation).

Let A_i be the aggregate value for the ith parameter. Therefore, in each iteration, the following update is performed:

$$A_i \Leftarrow A_i + \left(\frac{\partial L}{\partial w_i} \right)^2 \quad \forall i \tag{3.40}$$

The update for the ith parameter w_i is as follows:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L}{\partial w_i} \right); \quad \forall i$$

If desired, one can use $\sqrt{A_i + \epsilon}$ in the denominator instead of $\sqrt{A_i}$ to avoid ill-conditioning. Here, ϵ is a small positive value such as 10^{-8}.

Scaling the derivative inversely with $\sqrt{A_i}$ is a kind of "signal-to-noise" normalization because A_i only measures the historical magnitude of the gradient rather than its sign; it encourages faster *relative* movements along gently sloping directions with consistent sign of the gradient. If the gradient component along the ith direction keeps wildly fluctuating between $+100$ and -100, this type of magnitude-centric normalization will penalize that component far more than another gradient component that consistently takes on the value in the vicinity of 0.1 (but with a consistent sign). For example, in Figure 3.11, the movements along the oscillating direction will be de-emphasized, and the movement along the consistent direction will be emphasized. However, absolute movements along all components will tend to slow down over time, which is the main problem with the approach. The slowing down is caused by the fact that A_i is the *aggregate* value of the entire history of partial derivatives. This will lead to diminishing values of the scaled derivative. As a result, the progress of AdaGrad might prematurely become too slow, and it will eventually (almost) stop making progress. Another problem is that the aggregate scaling factors depend on ancient history, which can eventually become stale. The use of stale scaling factors can increase inaccuracy. As we will see later, most of the other methods use exponential averaging, which solves both problems.

3.5.3.2 RMSProp

The RMSProp algorithm [194] uses a similar motivation as AdaGrad for performing the "signal-to-noise" normalization with the absolute magnitude $\sqrt{A_i}$ of the gradients. However, instead of simply adding the squared gradients to estimate A_i, it uses *exponential averaging*. Since one uses *averaging* to normalize rather than *aggregate* values, the progress is not slowed prematurely by a constantly increasing scaling factor A_i. The basic idea is to use a decay factor $\rho \in (0, 1)$, and weight the squared partial derivatives occurring t updates ago by ρ^t. Note that this can be easily achieved by multiplying the current squared aggregate (i.e., *running* estimate) by ρ and then adding $(1 - \rho)$ times the current (squared) partial derivative. The running estimate is initialized to 0. This causes some (undesirable) bias in early iterations, which disappears over the longer term. Therefore, if A_i is the exponentially averaged value of the ith parameter w_i, we have the following way of updating A_i:

$$A_i \Leftarrow \rho A_i + (1 - \rho)\left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{3.41}$$

The square-root of this value for each parameter is used to normalize its gradient. Then, the following update is used for (global) learning rate α:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$$

If desired, one can use $\sqrt{A_i + \epsilon}$ in the denominator instead of $\sqrt{A_i}$ to avoid ill-conditioning. Here, ϵ is a small positive value such as 10^{-8}. Another advantage of RMSProp over AdaGrad is that the importance of ancient (i.e., stale) gradients decays exponentially with time. Furthermore, it can benefit by incorporating concepts of momentum within the computational algorithm (cf. Sections 3.5.3.3 and 3.5.3.5). The drawback of RMSProp is that the running estimate A_i of the second-order moment is biased in early iterations because it is initialized to 0.

3.5.3.3 RMSProp with Nesterov Momentum

RMSProp can also be combined with Nesterov momentum. Let A_i be the squared aggregate of the ith weight. In such cases, we introduce the additional parameter $\beta \in (0, 1)$ and use the following updates:

$$v_i \Leftarrow \beta v_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L(\overline{W} + \beta\overline{V})}{\partial w_i}\right); \quad w_i \Leftarrow w_i + v_i \quad \forall i$$

Note that the partial derivative of the loss function is computed at a shifted point, as is common in the Nesterov method. The weight $\overline{W}$ is shifted with $\beta\overline{V}$ while computing the partial derivative with respect to the loss function. The maintenance of A_i is done using the shifted gradients as well:

$$A_i \Leftarrow \rho A_i + (1 - \rho)\left(\frac{\partial L(\overline{W} + \beta\overline{V})}{\partial w_i}\right)^2 \quad \forall i \tag{3.42}$$

Although this approach benefits from adding momentum to RMSProp, it does not correct for the initialization bias.

3.5.3.4 AdaDelta

The AdaDelta algorithm [553] uses a similar update as RMSProp, except that it eliminates the need for a global learning parameter by computing it as a function of incremental updates in previous iterations. Consider the update of RMSProp, which is repeated below:

$$w_i \Leftarrow w_i - \underbrace{\frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

We will show how α is replaced with a value that depends on the previous incremental updates. In each update, the value of Δw_i is the increment in the value of w_i. As with the exponentially smoothed gradients A_i, we keep an exponentially smoothed value δ_i of the values of Δw_i in previous iterations with the same decay parameter ρ:

$$\delta_i \Leftarrow \rho\delta_i + (1-\rho)(\Delta w_i)^2 \quad \forall i \tag{3.43}$$

For a given iteration, the value of δ_i can be computed using only the iterations before it because the value of Δw_i is not yet available. On the other hand, A_i can be computed using the partial derivative in the current iteration as well. This is a subtle difference between how A_i and δ_i are computed. This results in the following AdaDelta update:

$$w_i \Leftarrow w_i - \underbrace{\sqrt{\frac{\delta_i}{A_i}}\left(\frac{\partial L}{\partial w_i}\right)}_{\Delta w_i}; \quad \forall i$$

It is noteworthy that a parameter α for the learning rate is completely missing from this update. The AdaDelta method shares some similarities with second-order methods because the ratio $\sqrt{\frac{\delta_i}{A_i}}$ in the update is a heuristic surrogate for the inverse of the second derivative of the loss with respect to w_i [553]. As discussed in subsequent sections, many second-order methods like the Newton method also do not use learning rates.

3.5.3.5 Adam

The Adam algorithm uses a similar "signal-to-noise" normalization as AdaGrad and RMSProp; however, it also exponentially smooths the first-order gradient in order to incorporate momentum into the update. It also directly addresses the bias inherent in exponential smoothing when the running estimate of a smoothed value is unrealistically initialized to 0.

As in the case of RMSProp, let A_i be the exponentially averaged value of the ith parameter w_i. This value is updated in the same way as RMSProp with the decay parameter $\rho \in (0,1)$:

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{3.44}$$

At the same time, an exponentially smoothed value of the gradient is maintained for which the ith component is denoted by F_i. This smoothing is performed with a different decay parameter ρ_f:

$$F_i \Leftarrow \rho_f F_i + (1-\rho_f)\left(\frac{\partial L}{\partial w_i}\right) \quad \forall i \tag{3.45}$$

This type of exponentially smoothing of the gradient with ρ_f is a variation of the momentum method discussed in Section 3.5.2 (which is parameterized by a friction parameter β instead of ρ_f). Then, the following update is used at learning rate α_t in the tth iteration:

$$w_i \Leftarrow w_i - \frac{\alpha_t}{\sqrt{A_i}}F_i; \quad \forall i$$

There are two key differences from the RMSProp algorithm. First, the gradient is replaced with its exponentially smoothed value in order to incorporate momentum. Second, the learning rate α_t now depends on the iteration index t, and is defined as follows:

$$\alpha_t = \alpha \underbrace{\left(\frac{\sqrt{1 - \rho^t}}{1 - \rho_f^t} \right)}_{\text{Adjust Bias}} \tag{3.46}$$

Technically, the adjustment to the learning rate is actually a bias correction factor that is applied to account for the unrealistic initialization of the two exponential smoothing mechanisms, and it is particularly important in early iterations. Both F_i and A_i are initialized to 0, which causes bias in early iterations. The two quantities are affected differently by the bias, which accounts for the ratio in Equation 3.46. It is noteworthy that each of ρ^t and ρ_f^t converge to 0 for large t because $\rho, \rho_f \in (0, 1)$. As a result, the initialization bias correction factor of Equation 3.46 converges to 1, and α_t converges to α. The default suggested values of ρ_f and ρ are 0.9 and 0.999, respectively, according to the original Adam paper [241]. Refer to [241] for details of other criteria (such as parameter sparsity) used for selecting ρ and ρ_f. Like other methods, Adam uses $\sqrt{A_i + \epsilon}$ (instead of $\sqrt{A_i}$) in the denominator of the update for better conditioning. The Adam algorithm is extremely popular because it incorporates most of the advantages of other algorithms, and often performs competitively with respect to the best of the other methods [241].

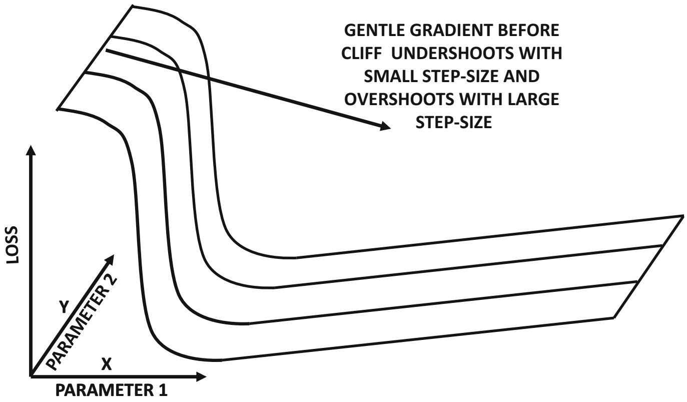

GENTLE GRADIENT BEFORE CLIFF UNDERSHOOTS WITH SMALL STEP-SIZE AND OVERSHOOTS WITH LARGE STEP-SIZE
LOSS
Y
PARAMETER 2
X
PARAMETER 1

Figure 3.13: An example of a cliff in the loss surface

3.5.4 Cliffs and Higher-Order Instability

So far, only the use of first-order derivatives has been discussed in this chapter. The progress with first-order derivatives can be slow with some error surfaces. Part of the problem is that the first-order derivatives provide a limited amount of information about the error surface, which can cause the updates to overshoot. The complexity of the loss surfaces of many neural networks can cause gradient-based updates to perform in an unanticipated way.

An example of a loss surface is shown in Figure 3.13. In this case, there is a gently sloping surface that rapidly changes into a cliff. However, if one computed only the first-order partial derivative with respect to the variable x shown in the figure, one would only see a gentle slope. As a result, a small learning rate will lead to very slow learning, whereas increasing the learning rate can suddenly cause overshooting to a point far from the optimal solution. This problem is caused by the nature of the curvature (i.e., changing gradient), where the first-order gradient does not contain the information needed to control the size of the update. In many cases, the rate of change of gradient can be computed using the second-order derivative, which provides useful (additional) information. In general, second-order methods approximate the local loss surface with a quadratic bowl, which is more accurate than the linear approximation. Some second-order methods like the *Newton method* require exactly one iteration in order to find the local optimal solution for a quadratic surface. Of course, the loss surface of neural models is typically not quadratic. Nevertheless, the approximation is often good enough that gradient-descent methods are greatly accelerated at least in cases where the change in the gradient is not too sudden or drastic.

Cliffs are not desirable because they manifest a certain level of instability in the loss function. This implies that a small change in some of the weights can either change the loss in a tiny way or suddenly change the loss by such a large amount that the resulting solution is even further away from the true optimum. As you will learn in Chapter 7, all temporal layers of a recurrent neural network share the same parameters. In such a case, the vanishing and exploding gradient means that there is varying sensitivity of the loss function with respect to the parameters in earlier and later layers (which are tied anyway). Therefore, a small change in a well-chosen parameter can cascade in an unstable way through the layers and either blow up or have negligible effect on the value of the loss function. Furthermore, it is hard to control the step size in a way that prevents one of these two events. This is the typical behavior one would encounter near a cliff. As a result, it is easy to miss the optimum during a gradient-descent step. One way of understanding this behavior is that sharing parameters across layers naturally leads to higher-order effects of weight perturbations on the loss function. This is because the shared weights of different layers are multiplied during neural network prediction, and a first-order gradient is now insufficient to model the effect of the *curvature* in the loss function, which is a measure of the change in gradient along a particular direction. Such settings are often addressed with techniques that either clip the gradient, or explicitly use the curvature (i.e., second-order derivative) of the loss function.

3.5.5 Gradient Clipping

Gradient clipping is a technique that is used to deal with settings in which the partial derivatives along different directions have exceedingly different magnitudes. Some forms of gradient clipping use a similar principle to that used in adaptive learning rates by trying the make the different components of the partial derivatives more even. However, the clipping is done only on the basis of the current values of the gradients rather than their historical values. Two forms of gradient clipping are most common:

1. *Value-based clipping:* In value-based clipping, a minimum and maximum threshold are set on the gradient values. All partial derivatives that are less than the minimum are set to the minimum threshold. All partial derivatives that are greater than the maximum are set to the maximum threshold.

2. *Norm-based clipping:* In this case, the entire gradient vector is normalized by the L_2-norm of the entire vector. Note that this type of clipping does not change the relative magnitudes of the updates along different directions. However, for neural networks that share parameters across different layers (like *recurrent neural networks*),

the effect of the two types of clipping is very similar. By clipping, one can achieve a better conditioning of the values, so that the updates from mini-batch to mini-batch are roughly similar. Therefore, it would prevent an anomalous gradient explosion in a particular mini-batch from affecting the solution too much.

By and large, the effects of gradient clipping are quite limited compared to many other methods. However, it is particularly effective in avoiding the exploding gradient problem in recurrent neural networks. In recurrent neural networks (cf. Chapter 7), the parameters are shared across different layers, and a derivative is computed with respect to each copy of the shared parameter by treating it as a separate variable. These derivatives are the temporal components of the overall gradient, and the values are clipped before adding them in order to obtain the overall gradient. A geometric interpretation of the exploding gradient problem is provided in [369], and a detailed exploration of why gradient clipping works is provided in [368].

3.5.6 Second-Order Derivatives

A number of methods have been proposed in recent years for using second-order derivatives for optimization. Such methods can partially alleviate some of the problems caused by curvature of the loss function.

Consider the parameter vector $\overline{W} = (w_1 \ldots w_d)^T$, which is expressed[3] as a column vector. The second-order derivatives of the loss function $L(\overline{W})$ are of the following form:

$$H_{ij} = \frac{\partial^2 L(\overline{W})}{\partial w_i \partial w_j}$$

Note that the partial derivatives use all pairwise parameters in the denominator. Therefore, for a neural network with d parameters, we have a $d \times d$ *Hessian matrix* H, for which the (i, j)th entry is H_{ij}. The second-order derivatives of the loss function can be computed with backpropagation [315], although this is rarely done in practice. The Hessian can be viewed as the Jacobian of the gradient.

One can write a quadratic approximation of the loss function in the vicinity of parameter vector $\overline{W}_0$ by using the following Taylor expansion:

$$L(\overline{W}) \approx L(\overline{W}_0) + (\overline{W} - \overline{W}_0)^T[\nabla L(\overline{W}_0)] + \frac{1}{2}(\overline{W} - \overline{W}_0)^T H(\overline{W} - \overline{W}_0) \tag{3.47}$$

Note that the Hessian H is computed at $\overline{W}_0$. Here, the parameter vectors $\overline{W}$ and $\overline{W}_0$ are d-dimensional column vectors, as is the gradient of the loss function. This is a quadratic approximation, and one can simply set the gradient to 0, which results in the following optimality condition for the quadratic approximation:

$$\begin{aligned} &\nabla L(\overline{W}) = 0 \quad \text{[Gradient of Loss Function]} \\ &\nabla L(\overline{W}_0) + H(\overline{W} - \overline{W}_0) = 0 \quad \text{[Gradient of Taylor approximation]} \end{aligned}$$

One can rearrange the above optimality condition to obtain the following Newton update:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla L(\overline{W}_0)] \tag{3.48}$$

[3] In most of this book, we have worked with $\overline{W}$ as a row-vector. However, it is notationally convenient here to work with $\overline{W}$ as a column-vector.

One interesting characteristic of this update is that it is directly obtained from an optimality condition, and therefore there is no learning rate. In other words, this update is approximating the loss function with a quadratic bowl and moving *exactly* to the bottom of the bowl *in a single step*; the learning rate is already incorporated implicitly. Recall from Figure 3.9 that first-order methods bounce along directions of high curvature. Of course, the bottom of the quadratic approximation is not the bottom of the true loss function, and therefore multiple Newton updates will be needed.

The main difference of Equation 3.48 from the update of steepest-gradient descent is pre-multiplication of the steepest direction (which is $[\nabla L(\overline{W}_0)]$) with the inverse of the Hessian. This multiplication with the inverse Hessian plays a key role in changing the direction of the steepest-gradient descent, so that one can take larger steps in that direction (resulting in better improvement of the objective function) even if the *instantaneous* rate of change in that direction is not as large as the steepest-descent direction. This is because the Hessian encodes how fast the gradient is changing in each direction. Changing gradients are bad for larger updates because one might inadvertently worsen the objective function, if the signs of many components of the gradient change during the step. It is profitable to move in directions where the ratio of the gradient to the rate of change of the gradient is large, so

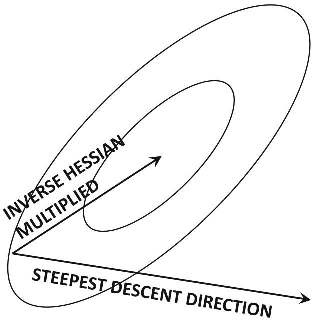


Figure 3.14: The effect of pre-multiplication of steepest-descent direction with the inverse Hessian

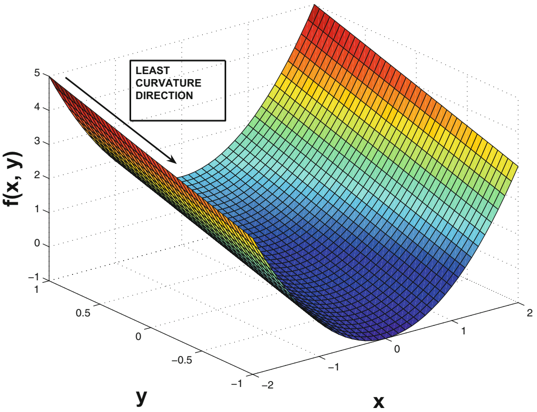


Figure 3.15: The curvature effect in valleys

that one can take larger steps without causing harm to the optimization. Pre-multiplication with the inverse of the Hessian achieves this goal. The effect of the pre-multiplication of the steepest-descent direction with the inverse Hessian is shown in Figure 3.14. It is helpful to reconcile this figure with the example of the quadratic bowl in Figure 3.9. In a sense, pre-multiplication with the inverse Hessian biases the learning steps towards low-curvature directions. In one dimension, the Newton step is simply the ratio of the first derivative (rate of change) to the second derivative (curvature). In multiple dimensions, the low-curvature directions tend to win out because of multiplication by the inverse Hessian.

The specific effect of curvature is particularly evident when one encounters loss functions in the shape of sloping or winding valleys. An example of a sloping valley is shown in Figure 3.15. A valley is a dangerous topography for a gradient-descent method, particularly if the bottom of the valley has a steep and rapidly changing surface (which creates a narrow valley). This is, of course, not the case in Figure 3.15, which is a relatively easier case. However, even in this case, the steepest-descent direction will often bounce along the sides of the valley, and move down the slope relatively slowly if the step-sizes are chosen inaccurately. In narrow valleys, the gradient-descent method will bounce along the steep sides of the valley even more violently without making much progress in the gently sloping direction, where the greatest *long-term* gains are present. In such cases, it is only by normalizing the gradient information with the curvature, that will provide the correct directions of long-term movement. This type of normalization tends to favor low-curvature directions like the ones shown in Figure 3.15. Multiplication of the steepest-descent direction with the inverse Hessian achieves precisely this goal.

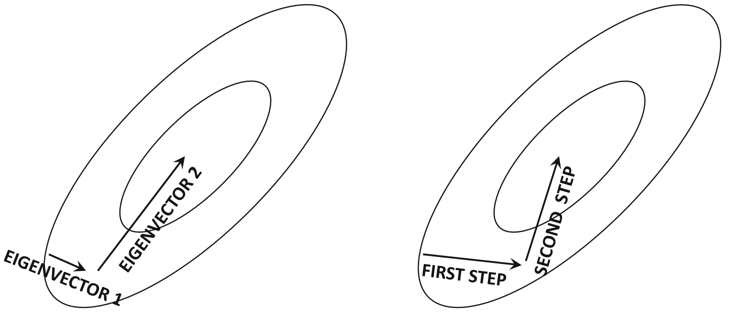
In most large-scale neural network settings, the Hessian is too large to store or compute explicitly. It is not uncommon to have neural networks with millions of parameters. Trying to compute the inverse of a $10^6 \times 10^6$ Hessian matrix is impractical with the computational power available today. In fact, it is difficult to even compute the Hessian, let alone invert it! Therefore, many approximations and variations of the Newton method have been developed. Examples of such methods include *Hessian-free optimization* [41, 189, 313, 314] (or method of *conjugate gradients*) and quasi-Newton methods that approximate the Hessian. The basic goal of these methods to make second-order updates without exactly computing the Hessian.

3.5.6.1 Conjugate Gradients and Hessian-Free Optimization

The *conjugate gradient method* [189] requires d steps to reach the optimal solution of a quadratic loss function (instead of a single Newton step). This approach is well known in the classical literature on neural networks [41, 443], and a variant has recently been reborn under the title of "Hessian-free optimization." This name is motivated by the fact that the search direction can be computed without the explicit computation of the Hessian.

A key problem in first-order methods is the zigzag movement of the optimization process, which undoes much of the work done in previous iterations. In the conjugate gradient method, the directions of movement are related to one another in such a way that the work done in previous iterations is never undone (for a quadratic loss function). This is because the change in gradient in a step, when projected along the vector of any other movement direction, is always 0. Furthermore, one uses *line search* to determine the optimal step size by searching over different step sizes. *Since an optimal step is taken along each direction and the work along that direction is never undone by subsequent steps, d linearly independent steps are needed to reach the optimum of a d-dimensional function.* Since it is possible to find such directions only for quadratic loss functions, we will first discuss the conjugate gradient method under the assumption that the loss function $L(\overline{W})$ is quadratic.

A quadratic and convex loss function $L(\overline{W})$ has an ellipsoidal contour plot of the type shown in Figure 3.16. The orthonormal eigenvectors $\overline{q}_0 \ldots \overline{q}_{d-1}$ of the symmetric Hessian represent the axes directions of the ellipsoidal contour plot. One can rewrite the loss function in a new coordinate space corresponding to the eigenvectors. In the axis system corresponding the eigenvectors, the (transformed) variables do not have interactions with one another because of the alignment of ellipsoidal loss contour with the axis system. *This is because the new Hessian $H_q = Q^T H Q$ obtained by rewriting the loss function in terms of the transformed variables is diagonal*, where Q is a $d \times d$ matrix with columns containing the eigenvectors. Therefore, each transformed variable can be optimized independently of the others. Alternatively, one can work with the original variables by successively making the best (projected) gradient-descent step along each eigenvector so as to minimize the loss function. The best movement along a particular direction is done using line search to select the step size. The nature of the movement is illustrated in Figure 3.16(a). Note that movement along the jth eigenvector does not disturb the work done along earlier eigenvectors and therefore d steps are sufficient to each the optimal solution.



(a) Eigenvectors of Hessian Mutually Orthogonal: $\overline{q}_i^T \overline{q}_j = 0$

(b) Arbitrary conjugate pair Non-orthogonal: $\overline{q}_i^T H \overline{q}_j = 0$

Figure 3.16: The eigenvectors of the Hessian of a quadratic function represent the orthogonal axes of the quadratic ellipsoid and are also mutually orthogonal. The eigenvectors of the Hessian are orthogonal conjugate directions. The generalized definition of conjugacy may result in non-orthogonal directions.

Although it is impractical to compute the eigenvectors of the Hessian, there are other efficiently computable directions satisfying similar properties; this key property is referred to as *mutual conjugacy* of vectors. Note that two eigenvectors $\overline{q}_i$ and $\overline{q}_j$ of the Hessian satisfy $\overline{q}_i^T \overline{q}_j = 0$ because of orthogonality. Furthermore, since $\overline{q}_j$ is an eigenvector of H, we have $H\overline{q}_j = \lambda_j \overline{q}_j$ for some scalar eigenvalue λ_j. Multiplying both sides with $\overline{q}_i^T$, we can easily show that the eigenvectors of the Hessian satisfy $\overline{q}_i^T H \overline{q}_j = 0$ in pairwise fashion. This condition is referred to as the *mutual conjugacy condition*, and it is equivalent to saying that the Hessian $H_q = Q^T H Q$ in the transformed axis-system of directions $\overline{q}_0 \ldots \overline{q}_{d-1}$ is diagonal. In fact, it turns out that *if we select any set of (not necessarily orthogonal) vectors $\overline{q}_0 \ldots \overline{q}_{d-1}$ satisfying the mutual conjugacy condition, then movement along any of these directions does not disturb the projected gradient along other directions.* Conjugate directions other than Hessian eigenvectors, such as those shown in Figure 3.16(b), may not be mutually orthogonal. If we re-write the quadratic loss function in terms of coordinates in a non-orthogonal axis system of conjugate directions, we will get nicely separated variables with a diagonal Hessian $H_q = Q^T H Q$. However, H_q is not a true diagonalization of H because $Q^T Q \neq I$. Nevertheless, such non-interacting directions are crucial to avoid zigzagging.

Let $\overline{W}_t$ and $\overline{W}_{t+1}$ represent the respective parameter vectors before and after movement along $\overline{q}_t$. The change in gradient $\nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)$ caused by movement along the direction $\overline{q}_t$ points in the same direction as $H\overline{q}_t$. This is because the product of the second-derivative (Hessian) matrix with a direction is proportional to the change in the first-derivative (gradient) when moving along that direction. This relationship is a finite-difference approximation for non-quadratic functions and it is exact for quadratic functions. Therefore, the projection (or dot product) of this change vector with respect to any other step vector $(\overline{W}_{i+1} - \overline{W}_i) \propto \overline{q}_i$ is given by the following:

$$\underbrace{[\overline{W}_{i+1} - \overline{W}_i]^T}_{\text{Earlier step}} \underbrace{[\nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)]}_{\text{Current gradient change}} \propto \overline{q}_i^T H \overline{q}_t = 0$$

This means that the only change to the gradient along a particular direction $\overline{q}_i$ (during the entire learning) occurs during the step along that direction. Line search ensures that the final gradient along that direction is 0. Convex loss functions have linearly independent conjugate directions (see Exercise 7). By making the best step along each conjugate direction, the final gradient will have zero dot product with d linearly independent directions; this is possible only when the final gradient is the zero vector (see Exercise 8), which implies optimality for a convex function. In fact, one can often reach a near-optimal solution in far fewer than d updates.

How can one generate conjugate directions iteratively? The obvious approach requires one needs to track $O(d^2)$ vector components of all previous $O(d)$ conjugate directions in order to enforce conjugacy of the next direction with respect to all these previous directions (see Exercise 11). Surprisingly, only the most recent conjugate direction is needed to generate the next direction [359, 443], when steepest decent directions are used for iterative generation. This is not an obvious result (see Exercise 12). The direction $\overline{q}_{t+1}$ is, therefore, defined iteratively as a linear combination of *only* the previous conjugate direction $\overline{q}_t$ and the current steepest descent direction $\nabla L(\overline{W}_{t+1})$ with combination parameter β_t:

$$\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \beta_t \overline{q}_t \tag{3.49}$$

Premultiplying both sides with $\overline{q}_t^T H$ and using the conjugacy condition to set the left-hand side to 0, one can solve for β_t:

$$\beta_t = \frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \tag{3.50}$$

This leads to an iterative update process, which initializes $\overline{q}_0 = -\nabla L(\overline{W}_0)$, and computes $\overline{q}_{t+1}$ iteratively for $t = 0, 1, 2, \ldots T$:

1. Update $\overline{W}_{t+1} \Leftarrow \overline{W}_t + \alpha_t \overline{q}_t$. Here, the step size α_t is computed using line search to minimize the loss function.

2. Set $\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t}\right) \overline{q}_t$. Increment t by 1.

It can be shown [359, 443] that $\overline{q}_{t+1}$ satisfies conjugacy with respect to *all* previous $\overline{q}_i$. A systematic road-map of this proof is provided in Exercise 12.

The above updates do not *seem* to be Hessian-free, because the matrix H is included in the above updates. However, the underlying computations only need the *projection* of the

Hessian along particular directions; we will see that these can be computed indirectly using the method of finite differences without explicitly computing the individual elements of the Hessian. Let $\overline{v}$ be the vector direction for which the projection $H\overline{v}$ needs to be computed. The method of finite differences computes the loss gradient at the current parameter vector $\overline{W}$ and at $\overline{W} + \delta\overline{v}$ for some small value of δ in order to perform the approximation:

$$H\overline{v} \approx \frac{\nabla L(\overline{W} + \delta\overline{v}) - \nabla L(\overline{W})}{\delta} \propto \nabla L(\overline{W} + \delta\overline{v}) - \nabla L(\overline{W}) \tag{3.51}$$

The right-hand side is free of the Hessian. The condition is exact for quadratic functions. Other alternatives for Hessian-free updates are discussed in [41].

So far, we have discussed the simplified case of quadratic loss functions, in which the second-order derivative matrix (i.e., Hessian) is a constant matrix (i.e., independent of the current parameter vector). However, neural loss functions are not quadratic and, therefore, the Hessian matrix is dependent on the current value of $\overline{W}_t$. Do we first create a quadratic approximation at a point and then solve it for a few iterations with the Hessian (quadratic approximation) fixed at that point, or do we change the Hessian every iteration? The former is referred to as the *linear conjugate gradient method*, whereas the latter is referred to as the *nonlinear conjugate gradient method*. The two methods are equivalent for quadratic loss functions, which almost never occur in neural networks.

Classical work in neural networks and machine learning has predominantly explored the use of the nonlinear conjugate gradient method [41], whereas recent work [313, 314] advocates the use of linear conjugate methods. In the nonlinear conjugate gradient method, the mutual conjugacy of the directions will deteriorate over time, which can have an unpredictable effect on overall progress even after a large number of iterations. A part of the problem is that the process of computing conjugate directions needs to be restarted every few steps as the mutual conjugacy deteriorates. If the deterioration occurs too fast, one does not gain much from conjugacy. On the other hand, each quadratic approximation in the linear conjugate gradient method can be solved exactly, and will typically be (almost) solved in much fewer than d iterations. Although multiple such approximations will be needed, there is guaranteed progress within each approximation, and the required number of approximations is often not too large. The work in [313] experimentally shows the superiority of linear conjugate gradient methods.

3.5.6.2 Quasi-Newton Methods and BFGS

The acronym BFGS stands for the Broyden–Fletcher–Goldfarb–Shanno algorithm, and it is derived as an approximation of the Newton method. Let us revisit the updates of the Newton method. A typical update of the Newton method is as follows:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla L(\overline{W}_0)] \tag{3.52}$$

In quasi-Newton methods, a sequence of approximations of the inverse Hessian matrix are used in various steps. Let the approximation of the inverse Hessian matrix in the tth step be denoted by G_t. In the very first iteration, the value of G_t is initialized to the identity matrix, which amounts to moving along the steepest-descent direction. This matrix is continuously updated from G_t to G_{t+1} with low-rank updates. A direct restatement of the Newton update in terms of the inverse Hessian $G_t \approx H_t^{-1}$ is as follows:

$$\overline{W}_{t+1} \Leftarrow \overline{W}_t - G_t[\nabla L(\overline{W}_t)] \tag{3.53}$$

The above update can be improved with an optimized learning rate α_t for non-quadratic loss functions working with (inverse) Hessian approximations like G_t:

$$\overline{W}_{t+1} \Leftarrow \overline{W}_t - \alpha_t G_t[\nabla L(\overline{W}_t)] \tag{3.54}$$

The optimized learning rate α_t is identified with line search. The line search does not need to be performed exactly (like the conjugate gradient method), because maintenance of conjugacy is no longer critical. Nevertheless, approximate conjugacy of the early set of directions is maintained by the method when starting with the identity matrix. One can (optionally) reset G_t to the identity matrix every d iterations (although this is rarely done).

It remains to be discussed how the matrix G_{t+1} is approximated from G_t. For this purpose, the *quasi-Newton condition*, also referred to as the *secant condition*, is needed:

$$\underbrace{\overline{W}_{t+1} - \overline{W}_t}_{\text{Parameter Change}} = G_{t+1} \underbrace{[\nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)]}_{\text{First derivative change}} \tag{3.55}$$

The above formula is simply a finite-difference approximation. Intuitively, multiplication of the second-derivative matrix (i.e., Hessian) with the parameter change (vector) approximately provides the gradient change. Therefore, multiplication of the inverse Hessian approximation G_{t+1} with the gradient change provides the parameter change. The goal is to find a symmetric matrix G_{t+1} satisfying Equation 3.55, but it represents an underdetermined system of equations with an infinite number of solutions. Among these, BFGS chooses the closest symmetric G_{t+1} to the current G_t, and achieves this goal by posing a minimization objective function $||G_{t+1} - G_t||_F$ in the form of a weighted Frobenius norm. The solution is as follows:

$$G_{t+1} \Leftarrow (I - \Delta_t \overline{q}_t \overline{v}_t^T) G_t (I - \Delta_t \overline{v}_t \overline{q}_t^T) + \Delta_t \overline{q}_t \overline{q}_t^T \tag{3.56}$$

Here, the (column) vectors $\overline{q}_t$ and $\overline{v}_t$ represent the parameter change and the gradient change; the scalar $\Delta_t = 1/(\overline{q}_t^T \overline{v}_t)$ is the inverse of the dot product of these two vectors.

$$\overline{q}_t = \overline{W}_{t+1} - \overline{W}_t; \qquad \overline{v}_t = \nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)$$

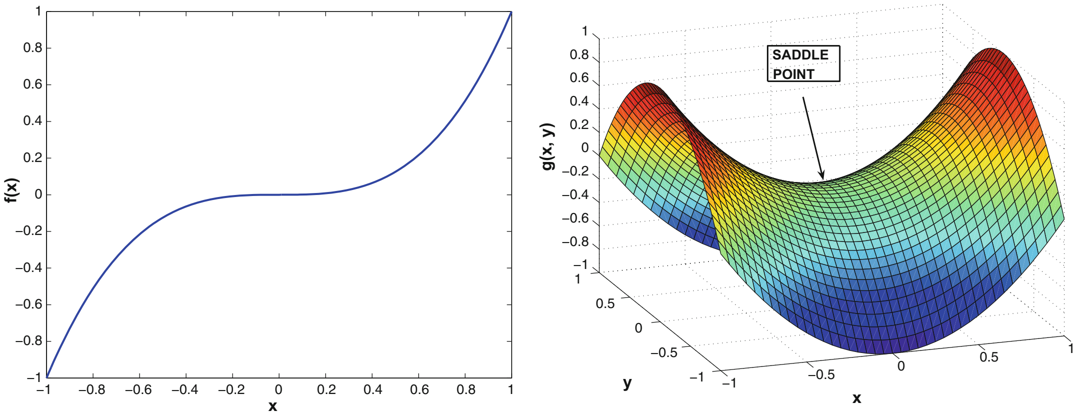
The update in Equation 3.56 can be made more space efficient by expanding it, so that fewer temporary matrices need to be maintained. Interested readers are referred to [300, 359, 376] for implementation details and derivation of these updates.

Even though BFGS benefits from approximating the inverse Hessian, it does need to carry over a matrix G_t of size $O(d^2)$ from one iteration to the next. The *limited memory BFGS* (L-BFGS) reduces the memory requirement drastically from $O(d^2)$ to $O(d)$ by not carrying over the matrix G_t from the previous iteration. In the most basic version of the L-BFGS method, the matrix G_t is replaced with the identity matrix in Equation 3.56 in order to derive G_{t+1}. A more refined choice is to store the $m \approx 30$ most recent vectors $\overline{q}_t$ and $\overline{v}_t$. Then, L-BFGS is equivalent to initializing G_{t-m+1} to the identity matrix and recursively applying Equation 3.56 m times to derive G_{t+1}. In practice, the implementation is optimized to directly compute the direction of movement from the vectors without explicitly storing large intermediate matrices from G_{t-m+1} to G_t. The directions found by L-BFGS roughly satisfy mutual conjugacy even with approximate line search.

3.5.6.3 Problems with Second-Order Methods: Saddle Points

Second-order methods are susceptible to the presence of *saddle points*. A saddle point is a stationary point of a gradient-descent method because its gradient is zero, but it is not a minimum (or maximum). A saddle point is an *inflection point*, which appears to be either a minimum or a maximum depending on which direction we approach it from. Therefore, the quadratic approximation of the Newton method will give vastly different shapes depending on the direction that one approaches the saddle point from. A 1-dimensional function with a saddle point is the following:

$$f(x) = x^3$$



(a) 1-dimensional saddle point

(b) 2-dimensional saddle point

Figure 3.17: Illustrations of saddle points

This function is shown in Figure 3.17(a), and it has an inflection point at $x = 0$. Note that a quadratic approximation at $x > 0$ will look like an upright bowl, whereas a quadratic approximation at $x < 0$ will look like an inverted bowl. Furthermore, even if one reaches $x = 0$ in the optimization process, both the second derivative and the first derivative will be zero. Therefore, a Newton update will take the 0/0 form and become indefinite. Such a point is a degenerate point from the perspective of numerical optimization. Not all saddle points are degenerate points and vice versa. For multivariate problems, such degenerate points are often wide and flat regions that are not minima of the objective function. They do present a significant problem for numerical optimization. An example of such a function is $h(x, y) = x^3 + y^3$, which is degenerate at $(0, 0)$. Furthermore, the region near $(0, 0)$ will appear like a flat plateau. These types of plateaus create problems for learning algorithms, because first-order algorithms slow down in these regions and second-order algorithms also cannot recognize them as spurious regions. It is noteworthy that such saddle points arise only in higher-order algebraic functions (i.e., higher than second order), which are common in neural network optimization.

It is also instructive to examine the case of a saddle point that is not a degenerate point. An example of a 2-dimensional function with a saddle point is as follows:

$$g(x, y) = x^2 - y^2$$

This function is shown in Figure 3.17(b). The saddle point is $(0, 0)$. It is easy to see that the shape of this function resembles a riding saddle. In this case, approaching from the x direction or from the y direction will result in very different quadratic approximations. In one case, the function will appear to be a minimum, and in another case the function will appear to be a maximum. Furthermore, the saddle point $(0, 0)$ will be a stationary point from the perspective of a Newton update, even though it is not an extremum. Saddle points occur frequently in regions between two hills of the loss function, and they present a problematic topography for second-order methods. Interestingly, first-order methods are often able to escape from saddle points [146], because the trajectory of first-order methods is simply not attracted by such points. On the other hand, Newton's method will jump directly to the saddle point.

Unfortunately, some neural-network loss functions seem to contain a large number of saddle points. Second-order methods therefore are not always preferable to first-order methods; the specific topography of a particular loss function may have an important role to play. Second-order methods are advantageous in situations with complex curvatures of the loss function or in the presence of cliffs. In other functions with saddle points, first-order methods are advantageous. Note that the pairing of computational algorithms (like Adam) with first-order gradient-descent methods already incorporates several advantages of second-order methods in an implicit way. Therefore, real-world practitioners often prefer first-order methods in combination with computational algorithms like Adam. Recently, some methods have been proposed [88] to address saddle points in second-order methods.

3.5.7 Polyak Averaging

One of the motivations for second-order methods is to avoid the kind of bouncing behavior caused by high-curvature regions. The example of the bouncing behavior caused in valleys (cf. Figure 3.15) is another example of this setting. One way of achieving some stability with any learning algorithm is to create an exponentially decaying average of the parameters over time, so that the bouncing behavior is avoided. Let $\overline{W}_1 \ldots \overline{W}_T$, be the sequence of parameters found by any learning method over the full sequence of T steps. In the simplest version of Polyak averaging, one simply computes the average of all the parameters as the final set $\overline{W}^f_T$:

$$\overline{W}^f_T = \frac{\sum_{i=1}^T \overline{W}_i}{T} \tag{3.57}$$

For simple averaging, we only need to compute $\overline{W}^f_T$ once at the end of the process, and we do not need to compute the values at $1 \ldots T-1$.

However, for exponential averaging with decay parameter $\beta < 1$, it is helpful to compute these values iteratively and maintain a running average over the course of the algorithm:

$$\overline{W}^f_t = \frac{\sum_{i=1}^t \beta^{t-i}\overline{W}_i}{\sum_{i=1}^t \beta^{t-i}} \qquad \text{[Explicit Formula]}$$

$$\overline{W}^f_t = (1-\beta)\overline{W}_t + \beta\overline{W}^f_{t-1} \quad \text{[Recursive Formula]}$$

The two formulas above are approximately equivalent at large values of t. The second formula is convenient because it enables maintenance over the course of the algorithm, and one does not need to maintain the entire history of parameters. Exponentially decaying averages are more useful than simple averages to avoid the effect of stale points. In simple averaging, the final result may be too heavily influenced by the early points, which are poor approximations to the correct solution.

3.5.8 Local and Spurious Minima

The example of the quadratic bowl given in earlier sections is a relatively simple optimization problem that has a single global optimum. Such problems are referred to as *convex* *optimization problems*, and they represent the simplest case of optimization. In general, however, the objective function of a neural network is not convex, and it is likely to have many local minima. In such cases, it is possible for the learning to converge to a suboptimal solution. In spite of this fact, with reasonably good initialization, the problem of local minima in neural networks causes fewer problems than might be expected.

Local minima are problematic only when their objective function values are significantly larger than that of the global minimum. In practice, however, this does not seem to be the case in neural networks. Many research results [88, 426] have shown that the local minima of real-life networks have very similar objective function values to the global minimum. As a result, their presence does not seem to cause as strong a problem as usually thought.

Local minima often cause problems in the context of *model generalization* with limited data. An important point to keep in mind is that the loss function is always defined on a limited sample of the training data, which is only a rough approximation of what the shape of the loss function looks like on the true distribution of the unseen test data. When the size of the training data is small, a number of spurious global or local minima are created by the paucity of training data. These minima are not seen in the (infinitely large) unseen distribution of test examples, but they appear as random artifacts of the particular choice of the training data set. Such spurious minima are often more prominent and attractive when the loss function is constructed on smaller training samples. In such cases, spurious minima can indeed create a problem, because they do not generalize well to unseen test instances. This problem is slightly different from the usual concept of local minima understood in traditional optimization; the *local minima on the training data do not generalize well to the test data*. In other words, the shape of the loss function is not even the same on the training and on the test data, and therefore the minima in the two cases do not match. Here, it is important to understand that there are fundamental differences between traditional optimization and machine learning methods that attempt to generalize a loss function on a limited data set to the universe of test examples. This is a notion referred to as *empirical risk minimization*, in which one computes the (approximate) *empirical* risk for a learning algorithm because the true distribution of the examples is unknown. When starting with random initialization points, it is often possible to fall into one of these spurious minima, unless one is careful to move the initialization point to a place closer to the basins of true optima (from a model generalization point of view). One such approach is that of *unsupervised pretraining*, which is discussed in Chapter 4.

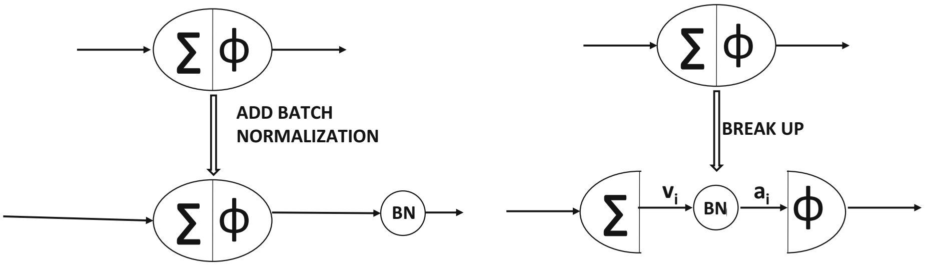
The specific problem of spurious minima (caused by the inability to generalize the results from a limited training data to unseen test data) is a much larger problem in neural network learning than the problem of local minima (from the perspective of traditional optimization). The nature of this problem is different enough from the normal understanding of local minima, so that it discussed in a separate chapter on model generalization (cf. Chapter 4).

3.6 Batch Normalization

Batch normalization is a recent method to address the vanishing and exploding gradient problems, which cause activation gradients in successive layers to either reduce or increase in magnitude. Another important problem in training deep networks is that of *internal covariate shift*. The problem is that the parameters change during training, and therefore the hidden variable activations change as well. In other words, the hidden inputs from early layers to later layers keep changing. Changing inputs from early layers to later layers causes slower convergence during training because the training data for later layers is not stable.

Batch normalization is able to reduce this effect.

In batch normalization, the idea is to add additional "normalization layers" between hidden layers that resist this type of behavior by creating features with somewhat similar variance. Furthermore, each unit in the normalization layers contains two additional parameters β_i and γ_i that regulate the precise level of normalization in the ith unit; these parameters are learned in a data-driven manner. The basic idea is that the output of the ith unit will have a mean of β_i and a standard deviation of γ_i *over each mini-batch of training instances*. One might wonder whether it might make sense to simply set each β_i to 0 and each γ_i to 1, but doing so reduces the representation power of the network. For example, if we make this transformation, then the sigmoid units will be operating within their linear regions, especially if the normalization is performed just before activation (see below for discussion of Figure 3.18). Recall from the discussion in Chapter 1 that multilayer networks do not gain power from depth without nonlinear activations. Therefore, allowing some "wiggle" with these parameters and learning them in a data-driven manner makes sense. Furthermore, the parameter β_i plays the role of a learned bias variable, and therefore we do not need additional bias units in these layers.



(a) Post-activation normalization (b) Pre-activation normalization

Figure 3.18: The different choices in batch normalization

We assume that the ith unit is connected to a special type of node BN_i, where BN stands for batch normalization. This unit contains two parameters β_i and γ_i that need to be learned. Note that BN_i has only one input, and its job is to perform the normalization and scaling. This node is then connected to the next layer of the network in the standard way in which a neural network is connected to future layers. Here, we mention that there are two choices for where the normalization layer can be connected:

1. The normalization can be performed just after applying the activation function to the linearly transformed inputs. This solution is shown in Figure 3.18(a). Therefore, the normalization is performed on *post-activation values.*

2. The normalization can be performed after the linear transformation of the inputs, but before applying the activation function. This situation is shown in Figure 3.18(b). Therefore, the normalization is performed on *pre-activation values.*

It is argued in [214] that the second choice has more advantages. Therefore, we focus on this choice in this exposition. The BN node shown in Figure 3.18(b) is just like any other computational node (albeit with some special properties), and one can perform backpropagation through this node just like any other computational node.

What transformations does BN_i apply? Consider the case in which its input is $v_i^{(r)}$, corresponding to the rth element of the batch feeding into the ith unit. Each $v_i^{(r)}$ is obtained by using the linear transformation defined by the coefficient vector $\overline{W}_i$ (and biases if any). For a particular batch of m instances, let the values of the m activations be denoted by $v_i^{(1)}, v_i^{(2)}, \ldots v_i^{(m)}$. The first step is to compute the mean μ_i and standard deviation σ_i for the ith hidden unit. These are then scaled using the parameters β_i and γ_i to create the outputs for the next layer:

$$\mu_i = \frac{\sum_{r=1}^m v_i^{(r)}}{m} \quad \forall i \tag{3.58}$$

$$\sigma_i^2 = \frac{\sum_{r=1}^m (v_i^{(r)} - \mu_i)^2}{m} + \epsilon \quad \forall i \tag{3.59}$$

$$\hat{v}_i^{(r)} = \frac{v_i^{(r)} - \mu_i}{\sigma_i} \quad \forall i, r \tag{3.60}$$

$$a_i^{(r)} = \gamma_i \cdot \hat{v}_i^{(r)} + \beta_i \quad \forall i, r \tag{3.61}$$

A small value of ϵ is added to σ_i^2 to regularize cases in which all activations are the same, which results in zero variance. Note that $a_i^{(r)}$ is the pre-activation output of the ith node, when the rth batch instance passes through it. This value would otherwise have been set to $v_i^{(r)}$, if we had not applied batch normalization. We conceptually represent this node with a special node BN_i that performs this additional processing. This node is shown in Figure 3.18(b). Therefore, the backpropagation algorithm has to account for this additional node and ensure that the loss derivative of layers earlier than the batch normalization layer accounts for the transformation implied by these new nodes. It is important to note that the function applied at each of these special BN nodes is specific to the *batch* at hand. This type of computation is unusual for a neural network in which the gradients are linearly separable sums of the gradients with respect to individual training examples. This is not quite true in this case because the batch normalization layer computes nonlinear metrics from the batch (such as its standard deviation). Therefore, the activations depend on how the examples in a batch are related to one another, which is not common in most neural computations. However, this special property of the BN node does not prevent us from backpropagating through the computations performed in it.

The following will describe the changes in the backpropagation algorithm caused by the normalization layer. The main point of this change is to show how to backpropagate through the newly added layer of normalization nodes. Another point to be aware of is that we want to optimize the parameters β_i and γ_i. For the gradient-descent steps with respect to each β_i and γ_i, we need the gradients with respect to these parameters. Assume that we have already backpropagated up to the output of the BN node, and therefore we have each $\frac{\partial L}{\partial a_i^{(r)}}$ available. Then, the derivatives with respect to the two parameters can be computed as follows:

$$\frac{\partial L}{\partial \beta_i} = \sum_{r=1}^m \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \beta_i} = \sum_{r=1}^m \frac{\partial L}{\partial a_i^{(r)}}$$

$$\frac{\partial L}{\partial \gamma_i} = \sum_{r=1}^m \frac{\partial L}{\partial a_i^{(r)}} \cdot \frac{\partial a_i^{(r)}}{\partial \gamma_i} = \sum_{r=1}^m \frac{\partial L}{\partial a_i^{(r)}} \cdot \hat{v}_i^{(r)}$$

We also need a way to compute $\frac{\partial L}{\partial v_i^r}$. Once this value is computed, the backpropagation to the pre-activation values $\frac{\partial L}{\partial a_j^r}$ for all nodes j in the previous layer uses the straightforward backpropagation update introduced earlier in this chapter. Therefore, the dynamic programming recursion will be complete because one can then use these values of $\frac{\partial L}{\partial a_j^r}$. One can compute the value of $\frac{\partial L}{\partial v_i^r}$ in terms of $\hat{v}_i^{(r)}$, μ_i, and σ_i, by observing that $v_i^{(r)}$ can be written as a (normalization) function of only $\hat{v}_i^{(r)}$, mean μ_i, and variance σ_i^2. Observe that μ_i and σ_i are not treated as constants, but as variables because they depend on the batch at hand. Therefore, we have the following:

$$\frac{\partial L}{\partial v_i^{(r)}} = \frac{\partial L}{\partial \hat{v}_i^{(r)}} \frac{\partial \hat{v}_i^{(r)}}{\partial v_i^{(r)}} + \frac{\partial L}{\partial \mu_i} \frac{\partial \mu_i}{\partial v_i^{(r)}} + \frac{\partial L}{\partial \sigma_i^2} \frac{\partial \sigma_i^2}{\partial v_i^{(r)}} \tag{3.62}$$

$$= \frac{\partial L}{\partial \hat{v}_i^{(r)}} \left(\frac{1}{\sigma_i}\right) + \frac{\partial L}{\partial \mu_i} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \sigma_i^2} \left(\frac{2(v_i^{(r)} - \mu_i)}{m}\right) \tag{3.63}$$

We need to evaluate each of the three partial derivatives on the right-hand side of the above equation in terms of the quantities that have been computed using the already-executed dynamic programming updates of backpropagation. This allows the creation of the recurrence equation for the batch normalization layer. Among these, the first expression, which is $\frac{\partial L}{\partial \hat{v}_i^{(r)}}$, can be substituted in terms of the loss derivatives of the next layer by observing that $a_i^{(r)}$ is related to $\hat{v}_i^{(r)}$ by a constant of proportionality γ_i:

$$\frac{\partial L}{\partial \hat{v}_i^{(r)}} = \gamma_i \frac{\partial L}{\partial a_i^{(r)}} \quad [\text{Since } a_i^{(r)} = \gamma_i \cdot \hat{v}_i^{(r)} + \beta_i] \tag{3.64}$$

Therefore, by substituting this value of $\frac{\partial L}{\partial \hat{v}_i^{(r)}}$ in Equation 3.63, we have the following:

$$\frac{\partial L}{\partial v_i^{(r)}} = \frac{\partial L}{\partial a_i^{(r)}} \left(\frac{\gamma_i}{\sigma_i}\right) + \frac{\partial L}{\partial \mu_i} \left(\frac{1}{m}\right) + \frac{\partial L}{\partial \sigma_i^2} \left(\frac{2(v_i^{(r)} - \mu_i)}{m}\right) \tag{3.65}$$

It now remains to compute the partial derivative of the loss with respect to the mean and the variance. The partial derivative of the loss with respect to the variance is computed as follows:

$$\frac{\partial L}{\partial \sigma_i^2} = \underbrace{\sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} \cdot \frac{\partial \hat{v}_i^{(q)}}{\partial \sigma_i^2}}_{\text{Chain rule}} = \underbrace{-\frac{1}{2\sigma_i^3} \sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} (v_i^{(q)} - \mu_i)}_{\text{Use Equation 3.60}} = \underbrace{-\frac{1}{2\sigma_i^3} \sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}} \gamma_i \cdot (v_i^{(q)} - \mu_i)}_{\text{Substitution from Equation 3.64}}$$

The partial derivatives of the loss with respect to the mean can be computed as follows:

$$\frac{\partial L}{\partial \mu_i} = \underbrace{\sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} \cdot \frac{\partial \hat{v}_i^{(q)}}{\partial \mu_i} + \frac{\partial L}{\partial \sigma_i^2} \cdot \frac{\partial \sigma_i^2}{\partial \mu_i}}_{\text{Chain rule}} = \underbrace{-\frac{1}{\sigma_i} \sum_{q=1}^{m} \frac{\partial L}{\partial \hat{v}_i^{(q)}} - 2\frac{\partial L}{\partial \sigma_i^2} \cdot \frac{\sum_{q=1}^{m}(v_i^{(q)} - \mu_i)}{m}}_{\text{Use Equations 3.59 and 3.60}}$$

$$= \underbrace{-\frac{\gamma_i}{\sigma_i} \sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}}}_{\text{Eq. 3.64}} + \underbrace{\left(\frac{1}{\sigma_i^3}\right) \cdot \left(\sum_{q=1}^{m} \frac{\partial L}{\partial a_i^{(q)}} \gamma_i \cdot (v_i^{(q)} - \mu_i)\right) \cdot \left(\frac{\sum_{q=1}^{m}(v_i^{(q)} - \mu_i)}{m}\right)}_{\text{Substitution for } \frac{\partial L}{\partial \sigma_i^2}}$$

By plugging in the partial derivatives of the loss with respect to the mean and variance in Equation 3.65, we get a full recursion for $\frac{\partial L}{\partial v_i^{(r)}}$ (value before batch-normalization layer) in terms of $\frac{\partial L}{\partial a_i^{(r)}}$ (value *after* the batch normalization layer). This provides a full view of the backpropagation of the loss through the batch-normalization layer corresponding to the BN node. The other aspects of backpropagation remain similar to the traditional case. Batch normalization enables faster inference because it prevents problems such as the exploding and vanishing gradient (which cause slow learning).

A natural question about batch normalization arises during inference (prediction) time. Since the transformation parameters μ_i and σ_i depend on the batch, how should one compute them during testing when a *single* test instance is available? In this case, the values of μ_i are σ_i are computed up front using the *entire* population (of training data), and then treated as constants during testing time. One can also keep an exponentially weighted average of these values during training. Therefore, the normalization is a simple linear transformation during inference.

An interesting property of batch normalization is that *it also acts as a regularizer*. Note that the same data point can cause somewhat different updates depending on which batch it is included in. One can view this effect as a kind of noise added to the update process. Regularization is often achieved by adding a small amount of noise to the training data. It has been experimentally observed that regularization methods like *Dropout* (cf. Section 4.5.4 of Chapter 4) do not seem to improve performance when batch normalization is used [184], although there is not a complete agreement on this point. A variant of batch normalization, known as *layer normalization*, is known to work well with recurrent networks. This approach is discussed in Section 7.3.1 of Chapter 7.

3.7 Practical Tricks for Acceleration and Compression

Neural network learning algorithms can be extremely expensive, both in terms of the number of parameters in the model and the amount of data that needs to be processed. There are several strategies that are used to accelerate and compress the underlying implementations. Some of the common strategies are as follows:

1. *GPU-acceleration:* Graphics Processor Units (GPUs) have historically been used for rendering video games with intensive graphics because of their efficiency in settings where repeated matrix operations (e.g., on graphics pixels) are required. It was eventually realized by the machine learning community (and GPU hardware companies) that such repetitive operations are also used in settings like neural networks, in which matrix operations are extensively used. Even the use of a single GPU can significantly speed up implementation because of its high memory bandwidth and multithreading within its multicore architecture.

2. *Parallel implementations:* One can parallelize the implementations of neural networks by using multiple GPUs or CPUs. Either the neural network model or the data can be partitioned across different processors. These implementations are referred to as *model-parallel* and *data-parallel* implementations.

3. *Algorithmic tricks for model compression during deployment:* A key point about the practical use of neural networks is that they have different computational requirements during training and deployment. While it is acceptable to train a model for a week with a large amount of memory, the final deployment might be performed on a mobile

phone, which is highly constrained both in terms of memory and computational power. Therefore, numerous tricks are used for model compression during testing time. This type of compression often results in better cache performance and efficiency as well.

In the following, we will discuss some of these acceleration and compression techniques.

3.7.1 GPU Acceleration

GPUs were originally developed for rendering graphics on screens with the use of lists of 3-dimensional coordinates. Therefore, graphics cards were inherently designed to perform many matrix multiplications in parallel to render the graphics rapidly. GPU processors have evolved significantly, moving well beyond their original functionality of graphics rendering. Like graphics applications, neural-network implementations require large matrix multiplications, which is inherently suited to the GPU setting. In a traditional neural network, each forward propagation is a multiplication of a matrix and vector, whereas in a convolutional neural network, two matrices are multiplied. When a mini-batch approach is used, activations become matrices (instead of vectors) in a traditional neural network. Therefore, forward propagations require matrix multiplications. A similar result is true for backpropagation, during which two matrices are multiplied frequently to propagate the derivatives backwards. In other words, most of the intensive computations involve vector, matrix, and tensor operations. Even a single GPU is good at parallelizing these operations in its different cores with multithreading [203], in which some groups of threads sharing the same code are executed concurrently. This principle is referred to as *Single Instruction Multiple Threads (SIMT).* Although CPUs also support short-vector data parallelization via *Single Instruction Multiple Data (SIMD)* instructions, the degree of parallelism is much lower as compared to the GPU. There are different trade-offs when using GPUs as compared to traditional CPUs. GPUs are very good at repetitive operations, but they have difficulty at performing branching operations like *if-then* statements. Most of the intensive operations in neural network learning are repetitive matrix multiplications across different training instances, and therefore this setting is suited to the GPU. Although the clock speed of a single instruction in the GPU is slower than the traditional CPU, the parallelization is so much greater in the GPU that huge advantages are gained.

GPU threads are grouped into small units called *warps.* Each thread in the warp shares the same code in each cycle, and this restriction enables a concurrent execution of the threads. The implementation needs to be carefully tailored to reduce the use of memory bandwidth. This is done by *coalescing* the memory reads and writes from different threads, so that a single memory transaction can be used to read and write values from different threads. Consider a common operation like matrix multiplication in neural network settings. The matrices are multiplied by making each thread responsible for computing a single entry in the product matrix. For example, consider a situation in which a 100×50 matrix is multiplied with a 50×200 matrix. In such a case, a total of $100 \times 200 = 20000$ threads would be launched in order to compute the entries of the matrix. These threads will typically be partitioned into multiple warps, each of which is highly parallelized. Therefore, speedups are achieved. A discussion of matrix multiplication on GPUs is provided in [203].

With high amounts of parallelization, memory bandwidth is often the primary limiting factor. Memory bandwidth refers to the speed at which the processor can access the relevant parameters from their stored locations in memory. GPUs have a high degree of parallelism and high memory bandwidth as compared to traditional CPUs. Note that if one cannot access the relevant parameters from memory fast enough, then faster execution does not help the speed of computation. In such cases, the memory transfer cannot keep up with the speed of the processor whether working with the CPU or the GPU, and the CPU/GPU cores will idle. GPUs have different trade-offs between cache access, computation, and memory access. CPUs have much larger caches than GPUs and they rely on the caches to store an intermediate result, such as the result of multiplying two numbers. Accessing a computed value from a cache is much faster than multiplying them again, which is where the CPU has an advantage over the GPU. However, this advantage is neutralized in neural network settings, where the sizes of the parameter matrices and activations are often too large to fit in the CPU cache. Even though the CPU cache is larger than that of the GPU, it is not large enough to handle the scale at which neural-network operations are performed. In such cases, one has to rely on high memory bandwidth, which is where the GPU has an advantage over the CPU. Furthermore, it is often faster to perform the same computation again rather than accessing it from memory, when working with the GPU (assuming that the result is unavailable in a cache). Therefore, GPU implementations are done somewhat differently from traditional CPU implementations. Furthermore, the advantage gained can be sensitive to the choice of neural network architecture, as the memory bandwidth requirements and multi-threading gains of different architectures can be different.

At first sight, it might seem from the above example that the use of a GPU requires a lot of low-level programming, and it would indeed be a challenge to create custom GPU code for each neural architecture. With this problem in mind, companies like NVIDIA have modularized the interface between the programmer and the GPU implementation. The key point is that the speeding of primitives like matrix multiplication and convolution can be hidden from the user by providing a library of neural network operations that perform these faster operations behind the scenes. The GPU library is tightly integrated with deep learning frameworks like Caffe or Torch to take advantage of the accelerated operations on the GPU. A specific example of such a library is the *NVIDIA CUDA Deep Neural Network Library* [643], which is referred to in short as *cuDNN*. CUDA is a parallel computing platform and programming model that works with CUDA-enabled GPU processors. However, it provides an abstraction and a programming interface that is easy to use with relatively limited rewriting of code. The cuDNN library can be integrated with multiple deep learning frameworks such as Caffe, TensorFlow, Theano, and Torch. The changes required to convert the training code of a particular neural network from its CPU version to a GPU version are often small. For example, in Torch, the CUDA Torch package is incorporated at the beginning of the code, and various data structures (like tensors) are initialized as CUDA tensors (instead of regular tensors). With these types of modest modifications, virtually the same code can run on a GPU instead of a CPU in Torch. A similar situation holds true in other deep learning frameworks. This type of approach shields the developers from the low-level performance tuning required in GPU frameworks, because the primitives in the library already have the code that takes care of all the low-level details of parallelization on the GPU.

3.7.2 Parallel and Distributed Implementations

It is possible to make training even faster by using multiple CPUs or GPUs. Since it is more common to use multiple GPUs, we focus on this setting. Parallelism is not a simple matter when working with GPUs because there are overheads associated with the communication between different processors. The delay caused by these overheads has recently been reduced with specialized network cards for GPU-to-GPU transfer. Furthermore, algorithmic tricks like using 8-bit approximations of the gradients [98] can help in speeding up the communication. There are several ways in which one can partition the work across different processors, namely hyperparameter parallelism, model parallelism, and data parallelism. These methods are discussed below.

Hyperparameter Parallelism

The simplest possible way to achieve parallelism in the training process without much overhead is to train neural networks with different parameter settings on different processors. No communication is required across different executions, and therefore wasteful overhead is avoided. As discussed earlier in this chapter, runs with suboptimal hyperparameters are often terminated long before running them to completion. Nevertheless, a small number of different runs with optimized parameters are often used in order to create an ensemble of models. The training of different ensemble components can be performed independently on different processors.

Model Parallelism

Model parallelism is particularly useful when a single model is too large to fit on a GPU. In such a case, the hidden layer is divided across the different GPUs. The different GPUs work on exactly the same batch of training points, although different GPUs compute different parts of the activations and the gradients. Each GPU only contains the portion of the weight matrix that are multiplied with the hidden activations present in the GPU. However, it would still need to communicate the results of its activations to the other GPUs. Similarly, it would need to receive the derivatives with respect to the hidden units in other GPUs in order to compute the gradients of the weights between its hidden units and those of other GPUs. This is achieved with the use of inter-connections across GPUs, and the computations across these interconnections add to the overhead. In some cases, these interconnections are dropped in a subset of the layers in order to reduce the communication overhead (although the resulting model would not quite be the same as the sequential version). Model parallelism is not helpful in cases where the number of parameters in the neural network is small, and should only be used for large networks. A good practical example of model parallelism is the design of *AlexNet*, which is a convolutional neural network (cf. Section 8.4.1 of Chapter 8). A sequential version of *AlexNet* and a GPU-partitioned version of *AlexNet* are both shown in Figure 8.9 of Chapter 8. Note that the sequential version in Figure 8.9 is not exactly equivalent to the GPU-partitioned version because the interconnections between GPUs have been dropped in some of the layers. A discussion of model parallelism may be found in [74].

Data Parallelism

Data parallelism works best when the model is small enough to fit on each GPU, but the amount of training data is large. In these cases, the parameters are shared across the different GPUs and the goal of the updates is to use the different processors with different training points for faster updates. The problem is that perfect synchronization of the updates can slow down the process, because locking mechanisms would need to be used to synchronize the updates. The key point is that each processor would have to wait for the others to make their updates. As a result, the slowest processor creates a bottleneck. A method that uses *asynchronous* stochastic gradient descent was proposed in [91]. The basic idea is to use a parameter server in order to share the parameters across different GPU processors. The updates are performed without using any locking mechanism. In other words, each GPU can read the shared parameters at any time, perform the computation, and write the parameters to the parameter server without worrying about locks. In this case, inefficiency would still be caused by one GPU processor overwriting the progress made by another, but there would be no waiting times for writes. As a result, the overall progress would still be faster than with a synchronized mechanism. Distributed asynchronous gradient descent is quite popular as a strategy for parallelism in large-scale industrial settings.

Exploiting the Trade-Offs for Hybrid Parallelism

It is evident from the above discussion that model parallelism is well suited to models with a large parameter footprint, whereas data parallelism is well suited to smaller models. It turns out that one can combine the two types of parallelism over different parts of the network. In certain types of convolutional neural networks that have fully connected layers, the vast majority of parameters occur in the fully connected layers, whereas more computations are performed in the earlier layers. In these cases, it makes sense to use data parallelism for the early part of the network, and model parallelism for the later part of the network. This type of approach is referred to as *hybrid parallelism*. A discussion of this type of approach may be found in [254].

3.7.3 Algorithmic Tricks for Model Compression

Training a neural network and deploying it typically have different requirements in terms of memory and efficiency requirements. While it may be acceptable to require a week to train a neural network to recognize faces in images, the end user might wish to use the trained neural network to recognize a face within a matter of a few seconds. Furthermore, the model might be deployed on a mobile device with little memory and computational availability. In such cases, it is crucial to be able to use the trained model efficiently, and also use it with a limited amount of storage. Efficiency is generally not a problem at deployment time, because the prediction of a test instance often requires straightforward matrix multiplications over a few layers. On the other hand, storage requirements are often a problem because of the large number of parameters in multilayer networks. There are several tricks that are used for model compression in such cases. In most of the cases, a larger trained neural network is modified so that it requires less space by approximating some parts of the model. In addition, some efficiency improvements can also be realized at prediction time by model compression because of better cache performance and fewer operations, although this is not the primary goal. Interestingly, this approximation might occasionally *improve* accuracy on out-of-sample predictions because of regularization effects, especially if the original model is unnecessarily large compared to the training data size.

Sparsifying Weights in Training

The links in a neural network are associated with weights. If the absolute value of a particular weight is small, then the model is not strongly influenced by that weight. Such weights can be dropped, and the neural network can be fine-tuned starting with the current weights on links that have not yet been dropped. The level of sparsification will depend on the weight threshold at which links are dropped. By choosing a larger threshold at which weights are dropped, the size of the model will reduce significantly. In such cases, it is particularly important to fine-tune the values of the retained weights with further epochs of training. One can also encourage the dropping of links by using L_1-regularization, which will be discussed in Chapter 4. When L_1-regularization is used during training, many of the weights will have zero values anyway because of the natural mathematical properties of this form of regularization. However, it has been shown in [169] that L_2-regularization has the advantage of higher accuracy. Therefore, the work in [169] uses L_2-regularization and prunes the weights that are below a particular threshold.

Further enhancements were reported in [168], where the approach was combined with Huffman coding and quantization for compression. The goal of quantization is to reduce the number of bits representing each connection. This approach reduced the storage required by *AlexNet* [255] by a factor of 35, or from about 240MB to 6.9MB, with no loss of accuracy. It is now possible as a result of this reduction to fit the model into an on-chip SRAM cache rather than off-chip DRAM memory; this also provide a beneficial effect on prediction times.

Leveraging Redundancies in Weights

It was shown in [94] that the vast majority of the weights in a neural network are redundant. In other words, for any $m \times n$ weight matrix W between a pair of layers with m_1 and m_2 units respectively, one can express this weight matrix as $W \approx UV^T$, where U and V are of sizes $m_1 \times k$ and $m_2 \times k$, respectively. Furthermore, it is assumed that $k \ll \min\{m_1, m_2\}$. This phenomenon occurs because of several peculiarities in the training process. For example, the features and weights in a neural network tend to *co-adapt* because of different parts of the network training at different rates. Therefore, the faster parts of the network often adapt to the slower parts. As a result, there is a lot of redundancy in the network both in terms of the features and the weights, and the full expressivity of the network is never utilized. In such a case, one can replace the pair of layers (containing weight matrix W) with three layers of size m_1, k, and m_2. The weight matrices between the first pair of layers is U and the weight matrix between the second pair of layers is V^T. Even though the new matrix is deeper, it is better regularized as long as $W - UV^T$ only contains noise. Furthermore, the matrices U and V require $(m_1 + m_2) \cdot k$ parameters, which is less than the number of parameters in W as long as k is less than half the harmonic mean of m_1 and m_2:

$$\frac{\text{Parameters in } W}{\text{Parameters in } U,\, V} = \frac{m_1 \cdot m_2}{k(m_1 + m_2)} = \frac{\text{HARMONIC-MEAN}(m_1, m_2)}{2k}$$

As shown in [94], more than 95% of the parameters in the neural network are redundant, and therefore a low value of the rank k suffices for approximation.

An important point is that the replacement of W with U and V must be done *after* completion of the learning of W. For example, if we replaced the pair of layers corresponding to W with the three layers containing the two weight matrices U and V^T and trained from scratch, good results may not be obtained. This is because co-adaptation will occur again during training, and the resulting matrices U and V will have a rank even lower than k. As a result, under-fitting might occur.

Finally, one can compress even further by realizing that both U and V need not be learned because they are redundant with respect to each other. For any rank-k matrix U, one can learn V so that the product UV^T is the same value. Therefore, the work in [94] provides methods to fix U, and then learn V instead.

Hash-Based Compression

One can reduce the number of parameters to be stored by forcing randomly chosen entries of the weight matrix to take on shared values of the parameters. The random choice is achieved with the application of a hash function on the entry position (i, j) in the matrix.

For example, imagine a situation where we have a weight matrix of size 100×100 with 10^4 entries. In such a case, one can hash each weight to a value in the range $\{1, \ldots 1000\}$ to create 1000 groups. Each of these groups will contain an average of 10 connections that will share weights. Backpropagation can handle shared weights using the approach discussed in Section 3.2.9. This approach requires a space requirement of only 1000 for the matrix, which is 10% of the original space requirement. Note that one could instead use a matrix of size 100×10 to achieve the same compression, but the key point is that using shared weights does not hurt the expressivity of the model as much as would reducing the size of the weight matrix *a priori*. More details of this approach are discussed in [66].

Leveraging Mimic Models

Some interesting results in [13, 55] show that it is possible to significantly compress a model by creating a new training data set from a trained model, which is easier to model. This "easier" training data can be used to train a much smaller network without significant loss of accuracy. This smaller model is referred to as a *mimic model*. The following steps are used to create the mimic model:

1. A model is created on the original training data. This model might be very large, and potentially even created out of an ensemble of different models, further increasing the number of parameters; it would not be appropriate to use in space-constrained settings. It is assumed that the model outputs softmax probabilities of the different classes. This model is also referred to as the teacher model.

2. New training data is created by passing unlabeled examples through the trained network. The targets in the newly created training data are set to the softmax probability outputs of the trained model on the unlabeled examples. Since unlabeled data is often copious, it is possible to create a lot of training data in this way. It is noteworthy that the new training data contains soft (probabilistic) targets rather than the discrete targets in the original training data, which significantly contributes to the creation of the compressed model.

3. A much smaller and shallower network is trained using the new training data (with artificially generated labels). The original training data is not used at all. This much smaller and shallower network, which is referred to as the mimic or *student* model, is what is deployed in space-constrained settings. It can be shown that the accuracy of the mimic model does not substantially degrade from the model trained over the original neural network, even though it is much smaller in size.

A natural question arises as to why the mimic model should perform as well as the original model, even though it is much smaller in size both in terms of the depth as well as the number of parameters. Trying to construct a shallow model on the original data cannot match the accuracy of either the shallow model or the mimic model. A number of possible reasons have been hypothesized for the superior performance of the mimic model [13]:

1. If there are errors in the original training data because of mislabeling, it causes unnecessary complexity in the trained model. These types of errors are largely removed in the new training data.

2. If there are complex regions of the decision space, the teacher model simplifies them by providing softer labels in terms of probabilities. Complexity is washed away by filtering targets through the teacher model.

3. The original training data contains targets with 0/1 values, whereas the newly created training contains soft targets, which are more informative. This is particularly useful in one-hot encoded multilabel targets, where there are clear correlations across different classes.

4. The original targets might depend on inputs that are not available in the training data. On the other hand, the teacher-created labels depend on only the available inputs. This makes the model simpler to learn and washes away unexplained complexity. Unexplained complexity often leads to unnecessary parameters and depth.

One can view some of the above benefits as a kind of regularization effect. The results in [13] are stimulating, because they show that deep networks are not *theoretically* necessary, although the regularization effect of depth is practically necessary when working with the original training data. The mimic model enjoys the benefits of this regularization effect by using the artificially created targets instead of depth.

3.8 Summary

This chapter discusses the problem of training deep neural networks. We revisit the backpropagation algorithm in detail along with its challenges. The vanishing and the exploding gradient problems are introduced along with the challenges associated with varying sensitivity of the loss function to different optimization variables. Certain types of activation functions like ReLU are less sensitive to this problem. However, the use of the ReLU can sometimes lead to dead neurons, if one is not careful about the learning rate. The type of gradient descent used to accelerate learning is also important for more efficient executions. Modified stochastic gradient-descent methods include the use of Nesterov momentum, AdaGrad, AdaDelta, RMSProp, and Adam. All these methods encourage gradient-steps that accelerate the learning process.

Numerous methods have been introduced for addressing the problem of cliffs with the use of second-order optimization methods. In particular, Hessian-free optimization is seen as an effective approach for handling many of the underlying optimization issues. An exciting method that has been used recently to improve learning rates is the use of batch normalization. Batch normalization transforms the data layer by layer in order to ensure that the scaling of different variables is done in an optimum way. The use of batch normalization has become extremely common in different types of deep networks. Numerous methods have been proposed for accelerating and compressing neural network algorithms. Acceleration is often achieved via hardware improvements, whereas compression is achieved with algorithmic tricks.

3.9 Bibliographic Notes

The original idea of backpropagation was based on idea of differentiation of composition of functions as developed in control theory [54, 237] under the ambit of *automatic differentiation.* The adaptation of these methods to neural networks was proposed by Paul Werbos in his PhD thesis in 1974 [524], although a more modern form of the algorithm was proposed by Rumelhart *et al.* in 1986 [408]. A discussion of the history of the backpropagation algorithm may be found in the book by Paul Werbos [525].

A discussion of algorithms for hyperparameter optimization in neural networks and other machine learning algorithms may be found in [36, 38, 490]. The random search method for hyperparameter optimization is discussed in [37]. The use of *Bayesian optimization* for hyperparameter tuning is discussed in [42, 306, 458]. Numerous libraries are available for Bayesian tuning such as *Hyperopt* [614], *Spearmint* [616], and *SMAC* [615].

The rule that the initial weights should depend on both the fan-in and fan-out of a node in proportion to $\sqrt{2/(r_{in} + r_{out})}$ is based on [140]. The analysis of initialization methods for rectifier neural networks is provided in [183]. Evaluations and analysis of the effect of feature preprocessing on neural network learning may be found in [278, 532]. The use of rectifier linear units for addressing some of the training challenges is discussed in [141].

Nesterov's algorithm for gradient descent may be found in [353]. The delta-bar-delta method was proposed by [217]. The AdaGrad algorithm was proposed in [108]. The RMSProp algorithm is discussed in [194]. Another adaptive algorithm using stochastic gradient descent, which is *AdaDelta*, is discussed in [553]. This algorithms shares some similarities with second-order methods, and in particular to the method in [429]. The Adam algorithm, which is a further enhancement along this line of ideas, is discussed in [241]. The practical importance of initialization and momentum in deep learning is discussed in [478]. Beyond the use of the stochastic gradient method, the use of coordinate descent has been proposed [273]. The strategy of *Polyak averaging* is discussed in [380].

Several of the challenges associated with the vanishing and exploding gradient problems are discussed in [140, 205, 368]. Ideas for parameter initialization that avoid some of these problems are discussed in [140]. The gradient clipping rule was discussed by Mikolov in his PhD thesis [324]. A discussion of the gradient clipping method in the context of recurrent neural networks is provided in [368]. The ReLU activation function was introduced in [167], and several of its interesting properties are explored in [141, 221].

A description of several second-order gradient optimization methods (such as the Newton method) is provided in [41, 545, 300]. The basic principles of the conjugate gradient method have been described in several classical books and papers [41, 189, 443], and the work in [313, 314] discusses applications to neural networks. The work in [316] leverages a Kronecker-factored curvature matrix for fast gradient descent. Another way of approximating the Newton method is the quasi-Newton method [273, 300], with the simplest approximation being a diagonal Hessian [24]. The acronym BFGS stands for the Broyden-Fletcher-Goldfarb-Shanno algorithm. A variant known as limited memory BFGS or L-BFGS [273, 300] does not require as much memory. Another popular second-order method is the Levenberg–Marquardt algorithm. This approach is, however, defined for squared loss functions and cannot be used with many forms of cross-entropy or log-losses that are common in neural networks. Overviews of the approach may be found in [133, 300]. General discussions of different types of nonlinear programming methods are provided in [23, 39].

The stability of neural networks to local minima is discussed in [88, 426]. Batch normalization methods were introduced recently in [214]. A method that uses whitening for batch normalization is discussed in [96], although the approach seems not to be practical. Batch normalization requires some minor adjustments for recurrent networks [81], although a more effective approach for recurrent networks is that of *layer normalization* [14]. In this method (cf. Section 7.3.1), a single training case is used for normalizing all units in a layer, rather than using mini-batch normalization of a single unit. The approach is useful for recurrent networks. An analogous notion to batch normalization is that of weight normalization [419], in which the magnitudes and directions of the weight vectors are decoupled during the learning process. Related training tricks are discussed in [362].

A broader discussion of accelerating machine learning algorithms with GPUs may be found in [644]. Various types of parallelization tricks for GPUs are discussed in [74, 91, 254], and specific discussions on convolutional neural networks are provided in [541]. Model compression with regularization is discussed in [168, 169]. A related model compression method is proposed in [213]. The use of mimic models for compression is discussed in [55, 13]. A related approach is discussed in [202]. The leveraging of parameter redundancy for compressing neural networks is discussed in [94]. The compression of neural networks with the hashing trick is discussed in [66].

3.9.1 Software Resources

All the training algorithms discussed in this chapter are supported by numerous deep learning frameworks like *Caffe* [571], *Torch* [572], *Theano* [573], and *TensorFlow* [574]. Extensions of *Caffe* to Python and MATLAB are available. All these frameworks provide a variety of training algorithms that are discussed in this chapter. Options for batch normalization are available as separate layers in these frameworks. Several software libraries are available for Bayesian optimization of hyperparameters. These libraries include *Hyperopt* [614], *Spearmint* [616], and *SMAC* [615]. Although these are designed for smaller machine learning problems, they can still be used in some cases. Pointers to the NVIDIA cuDNN may be found in [643]. The different frameworks supported by cuDNN are discussed in [645].

3.10 Exercises

1. Consider the following recurrence:

$$(x_{t+1}, y_{t+1}) = (f(x_t, y_t), g(x_t, y_t)) \tag{3.66}$$

Here, $f()$ and $g()$ are multivariate functions.

(a) Derive an expression for $\frac{\partial x_{t+2}}{\partial x_t}$ in terms of only x_t and y_t.

(b) Can you draw an architecture of a neural network corresponding to the above recursion for t varying from 1 to 5? Assume that the neurons can compute any function you want.

2. Consider a two-input neuron that multiplies its two inputs x_1 and x_2 to obtain the output o. Let L be the loss function that is computed at o. Suppose that you know that $\frac{\partial L}{\partial o} = 5$, $x_1 = 2$, and $x_2 = 3$. Compute the values of $\frac{\partial L}{\partial x_1}$ and $\frac{\partial L}{\partial x_2}$.

3. Consider a neural network with three layers including an input layer. The first (input) layer has four inputs x_1, x_2, x_3, and x_4. The second layer has six hidden units corresponding to all pairwise multiplications. The output node o simply adds the values in the six hidden units. Let L be the loss at the output node. Suppose that you know that $\frac{\partial L}{\partial o} = 2$, and $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, and $x_4 = 4$. Compute $\frac{\partial L}{\partial x_i}$ for each i.

4. How does your answer to the previous question change when the output o is computed as a maximum of its six inputs rather than its sum?

5. The chapter discusses (cf. Table 3.1) how one can perform a backpropagation of an arbitrary function by using the multiplication with the Jacobian matrix. Discuss why one must be careful in using this matrix-centric approach.[Hint: Compute the Jacobian with respect to sigmoid function]

6. Consider the loss function $L = x^2 + y^{10}$. Implement a simple steepest-descent algorithm to plot the coordinates as they vary from the initialization point to the optimal value of 0. Consider two different initialization points of $(0.5, 0.5)$ and $(2, 2)$ and plot the trajectories in the two cases at a constant learning rate. What do you observe about the behavior of the algorithm in the two cases?

7. The Hessian H of a strongly convex quadratic function always satisfies $\overline{x}^T H \overline{x} > 0$ for any nonzero vector $\overline{x}$. For such problems, show that all conjugate directions are linearly independent.

8. Show that if the dot product of a d-dimensional vector $\overline{v}$ with d linearly independent vectors is 0, then $\overline{v}$ must be the zero vector.

9. This chapter discusses two variants of backpropagation, which use the pre-activation and the postactivation variables, respectively, for the dynamic programming recursion. Show that these two variants of backpropagation are mathematically equivalent.

10. Consider the softmax activation function in the output layer, in which real-valued outputs $v_1 \ldots v_k$ are converted into probabilities as follows (according to Equation 3.20):

$$o_i = \frac{\exp(v_i)}{\sum_{j=1}^{k} \exp(v_j)} \quad \forall i \in \{1, \ldots, k\}$$

(a) Show that the value of $\frac{\partial o_i}{\partial v_j}$ is $o_i(1 - o_i)$ when $i = j$. In the case that $i \neq j$, show that this value is $-o_i o_j$.

(b) Use the above result to show the correctness of Equation 3.22:

$$\frac{\partial L}{\partial v_i} = o_i - y_i$$

Assume that we are using the cross-entropy loss $L = -\sum_{i=1}^{k} y_i \log(o_i)$, where $y_i \in \{0, 1\}$ is the one-hot encoded class label over different values of $i \in \{1 \ldots k\}$.

11. The chapter uses steepest descent directions to iteratively generate conjugate directions. Suppose we pick d *arbitrary* directions $\overline{v}_0 \ldots \overline{v}_{d-1}$ that are linearly independent. Show that (with appropriate choice of β_{ti}) we can start with $\overline{q}_0 = \overline{v}_0$ and generate successive conjugate directions in the following form:

$$\overline{q}_{t+1} = \overline{v}_{t+1} + \sum_{i=0}^{t} \beta_{ti} \overline{q}_i$$

Discuss why this approach is more expensive than the one discussed in the chapter.

12. The definition of β_t in Section 3.5.6.1 ensures that $\overline{q}_t$ is conjugate to $\overline{q}_{t+1}$. This exercise systematically shows that *any* direction $\overline{q}_i$ for $i \leq t$ satisfies $\overline{q}_i^T H \overline{q}_{t+1} = 0$.
[Hint: Prove (b), (c), and (d) *jointly* with induction on t while staring at (a).]

(a) Recall from Equation 3.51 that $H\overline{q}_i = [\nabla L(\overline{W}_{i+1}) - \nabla L(\overline{W}_i)]/\delta_i$ for quadratic loss functions, where δ_i depends on ith step-size. Combine this condition with Equation 3.49 to show the following for all $i \leq t$:

$$\delta_i [\overline{q}_i^T H \overline{q}_{t+1}] = -[\nabla L(\overline{W}_{i+1}) - \nabla L(\overline{W}_i)]^T [\nabla L(\overline{W}_{t+1})] + \delta_i \beta_t (\overline{q}_i^T H \overline{q}_t)$$

Also show that $[\nabla L(\overline{W}_{t+1}) - \nabla L(\overline{W}_t)] \cdot \overline{q}_i = \delta_t \overline{q}_i^T H \overline{q}_t$.

(b) Show that $\nabla L(\overline{W}_{t+1})$ is orthogonal to each $\overline{q}_i$ for $i \leq t$. [The proof for the case when $i = t$ is trivial because the gradient at line-search termination is always orthogonal to the search direction.]

(c) Show that the loss gradients at $\overline{W}_0 \ldots \overline{W}_{t+1}$ are mutually orthogonal.

(d) Show that $\overline{q}_i^T H \overline{q}_{t+1} = 0$ for $i \leq t$. [The case for $i = t$ is trivial.]