Chapter 8

Optimization for Training Deep Models

Deep learning algorithms involve optimization in many contexts. For example, performing inference in models such as PCA involves solving an optimization problem. We often use analytical optimization to write proofs or design algorithms. Of all of the many optimization problems involved in deep learning, the most difficult is neural network training. It is quite common to invest days to months of time on hundreds of machines in order to solve even a single instance of the neural network training problem. Because this problem is so important and so expensive, a specialized set of optimization techniques have been developed for solving it. This chapter presents these optimization techniques for neural network training.

If you are unfamiliar with the basic principles of gradient-based optimization, we suggest reviewing chapter 4. That chapter includes a brief overview of numerical optimization in general.

This chapter focuses on one particular case of optimization: finding the parameters $\boldsymbol{\theta}$ of a neural network that significantly reduce a cost function $J(\boldsymbol{\theta})$, which typically includes a performance measure evaluated on the entire training set as well as additional regularization terms.

We begin with a description of how optimization used as a training algorithm for a machine learning task differs from pure optimization. Next, we present several of the concrete challenges that make optimization of neural networks difficult. We then define several practical algorithms, including both optimization algorithms themselves and strategies for initializing the parameters. More advanced algorithms adapt their learning rates during training or leverage information contained in

the second derivatives of the cost function. Finally, we conclude with a review of several optimization strategies that are formed by combining simple optimization algorithms into higher-level procedures.

8.1 How Learning Differs from Pure Optimization

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly. In most machine learning scenarios, we care about some performance measure P, that is defined with respect to the test set and may also be intractable. We therefore optimize P only indirectly. We reduce a different cost function $J(\boldsymbol{\theta})$ in the hope that doing so will improve P. This is in contrast to pure optimization, where minimizing J is a goal in and of itself. Optimization algorithms for training deep models also typically include some specialization on the specific structure of machine learning objective functions.

Typically, the cost function can be written as an average over the training set, such as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y), \tag{8.1}$$

where L is the per-example loss function, $f(\boldsymbol{x};\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$, $\hat{p}_{\mathrm{data}}$ is the empirical distribution. In the supervised learning case, y is the target output. Throughout this chapter, we develop the unregularized supervised case, where the arguments to L are $f(\boldsymbol{x};\boldsymbol{\theta})$ and y. However, it is trivial to extend this development, for example, to include $\boldsymbol{\theta}$ or $\boldsymbol{x}$ as arguments, or to exclude y as arguments, in order to develop various forms of regularization or unsupervised learning.

Equation 8.1 defines an objective function with respect to the training set. We would usually prefer to minimize the corresponding objective function where the expectation is taken across *the data generating distribution* p_{data} rather than just over the finite training set:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y). \tag{8.2}$$

8.1.1 Empirical Risk Minimization

The goal of a machine learning algorithm is to reduce the expected generalization error given by equation 8.2. This quantity is known as the **risk**. We emphasize here that the expectation is taken over the true underlying distribution p_{data}. If we knew the true distribution $p_{\mathrm{data}}(\boldsymbol{x}, y)$, risk minimization would be an optimization task

solvable by an optimization algorithm. However, when we do not know $p_{\text{data}}(\boldsymbol{x}, y)$ but only have a training set of samples, we have a machine learning problem.

The simplest way to convert a machine learning problem back into an optimization problem is to minimize the expected loss on the training set. This means replacing the true distribution $p(\boldsymbol{x}, y)$ with the empirical distribution $\hat{p}(\boldsymbol{x}, y)$ defined by the training set. We now minimize the **empirical risk**

$$\mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{\text{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}) \tag{8.3}$$

where m is the number of training examples.

The training process based on minimizing this average training error is known as **empirical risk minimization**. In this setting, machine learning is still very similar to straightforward optimization. Rather than optimizing the risk directly, we optimize the empirical risk, and hope that the risk decreases significantly as well. A variety of theoretical results establish conditions under which the true risk can be expected to decrease by various amounts.

However, empirical risk minimization is prone to overfitting. Models with high capacity can simply memorize the training set. In many cases, empirical risk minimization is not really feasible. The most effective modern optimization algorithms are based on gradient descent, but many useful loss functions, such as 0-1 loss, have no useful derivatives (the derivative is either zero or undefined everywhere). These two problems mean that, in the context of deep learning, we rarely use empirical risk minimization. Instead, we must use a slightly different approach, in which the quantity that we actually optimize is even more different from the quantity that we truly want to optimize.

8.1.2 Surrogate Loss Functions and Early Stopping

Sometimes, the loss function we actually care about (say classification error) is not one that can be optimized efficiently. For example, exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifier (Marcotte and Savard, 1992). In such situations, one typically optimizes a **surrogate loss function** instead, which acts as a proxy but has advantages. For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss. The negative log-likelihood allows the model to estimate the conditional probability of the classes, given the input, and if the model can do that well, then it can pick the classes that yield the least classification error in expectation.

In some cases, a surrogate loss function actually results in being able to learn more. For example, the test set 0-1 loss often continues to decrease for a long time after the training set 0-1 loss has reached zero, when training using the log-likelihood surrogate. This is because even when the expected 0-1 loss is zero, one can improve the robustness of the classifier by further pushing the classes apart from each other, obtaining a more confident and reliable classifier, thus extracting more information from the training data than would have been possible by simply minimizing the average 0-1 loss on the training set.

A very important difference between optimization in general and optimization as we use it for training algorithms is that training algorithms do not usually halt at a local minimum. Instead, a machine learning algorithm usually minimizes a surrogate loss function but halts when a convergence criterion based on early stopping (section 7.8) is satisfied. Typically the early stopping criterion is based on the true underlying loss function, such as 0-1 loss measured on a validation set, and is designed to cause the algorithm to halt whenever overfitting begins to occur. Training often halts while the surrogate loss function still has large derivatives, which is very different from the pure optimization setting, where an optimization algorithm is considered to have converged when the gradient becomes very small.

8.1.3 Batch and Minibatch Algorithms

One aspect of machine learning algorithms that separates them from general optimization algorithms is that the objective function usually decomposes as a sum over the training examples. Optimization algorithms for machine learning typically compute each update to the parameters based on an expected value of the cost function estimated using only a subset of the terms of the full cost function.

For example, maximum likelihood estimation problems, when viewed in log space, decompose into a sum over each example:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}, y^{(i)}; \boldsymbol{\theta}). \tag{8.4}$$

Maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\,\mathrm{,y}\sim\hat{p}_{\,\mathrm{data}}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{8.5}$$

Most of the properties of the objective function J used by most of our optimization algorithms are also expectations over the training set. For example, the

most commonly used property is the gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathrm{y} \sim \hat{p}_{\mathrm{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\mathrm{model}}(\boldsymbol{x}, y; \boldsymbol{\theta}). \tag{8.6}$$

Computing this expectation exactly is very expensive because it requires evaluating the model on every example in the entire dataset. In practice, we can compute these expectations by randomly sampling a small number of examples from the dataset, then taking the average over only those examples.

Recall that the standard error of the mean (equation 5.46) estimated from n samples is given by $\sigma / \sqrt{n}$, where σ is the true standard deviation of the value of the samples. The denominator of $\sqrt{n}$ shows that there are less than linear returns to using more examples to estimate the gradient. Compare two hypothetical estimates of the gradient, one based on 100 examples and another based on 10,000 examples. The latter requires 100 times more computation than the former, but reduces the standard error of the mean only by a factor of 10. Most optimization algorithms converge much faster (in terms of total computation, not in terms of number of updates) if they are allowed to rapidly compute approximate estimates of the gradient rather than slowly computing the exact gradient.

Another consideration motivating statistical estimation of the gradient from a small number of samples is redundancy in the training set. In the worst case, all m samples in the training set could be identical copies of each other. A sampling-based estimate of the gradient could compute the correct gradient with a single sample, using m times less computation than the naive approach. In practice, we are unlikely to truly encounter this worst-case situation, but we may find large numbers of examples that all make very similar contributions to the gradient.

Optimization algorithms that use the entire training set are called **batch** or **deterministic** gradient methods, because they process all of the training examples simultaneously in a large batch. This terminology can be somewhat confusing because the word "batch" is also often used to describe the minibatch used by minibatch stochastic gradient descent. Typically the term "batch gradient descent" implies the use of the full training set, while the use of the term "batch" to describe a group of examples does not. For example, it is very common to use the term "batch size" to describe the size of a minibatch.

Optimization algorithms that use only a single example at a time are sometimes called **stochastic** or sometimes **online** methods. The term online is usually reserved for the case where the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made.

Most algorithms used for deep learning fall somewhere in between, using more

than one but less than all of the training examples. These were traditionally called **minibatch** or **minibatch stochastic** methods and it is now common to simply call them **stochastic** methods.

The canonical example of a stochastic method is stochastic gradient descent, presented in detail in section 8.3.1.

Minibatch sizes are generally driven by the following factors:

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.
- If all examples in the batch are to be processed in parallel (as is typically the case), then the amount of memory scales with the batch size. For many hardware setups this is the limiting factor in batch size.
- Some kinds of hardware achieve better runtime with specific sizes of arrays. Especially when using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.
- Small batches can offer a regularizing effect (Wilson and Martinez, 2003), perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1. Training with such a small batch size might require a small learning rate to maintain stability due to the high variance in the estimate of the gradient. The total runtime can be very high due to the need to make more steps, both because of the reduced learning rate and because it takes more steps to observe the entire training set.

Different kinds of algorithms use different kinds of information from the minibatch in different ways. Some algorithms are more sensitive to sampling error than others, either because they use information that is difficult to estimate accurately with few samples, or because they use information in ways that amplify sampling errors more. Methods that compute updates based only on the gradient $\boldsymbol{g}$ are usually relatively robust and can handle smaller batch sizes like 100. Second-order methods, which use also the Hessian matrix $\boldsymbol{H}$ and compute updates such as $\boldsymbol{H}^{-1}\boldsymbol{g}$, typically require much larger batch sizes like 10,000. These large batch sizes are required to minimize fluctuations in the estimates of $\boldsymbol{H}^{-1}\boldsymbol{g}$. Suppose that $\boldsymbol{H}$ is estimated perfectly but has a poor condition number. Multiplication by

$\boldsymbol{H}$ or its inverse amplifies pre-existing errors, in this case, estimation errors in $\boldsymbol{g}$. Very small changes in the estimate of $\boldsymbol{g}$ can thus cause large changes in the update $\boldsymbol{H}^{-1}\boldsymbol{g}$, even if $\boldsymbol{H}$ were estimated perfectly. Of course, $\boldsymbol{H}$ will be estimated only approximately, so the update $\boldsymbol{H}^{-1}\boldsymbol{g}$ will contain even more error than we would predict from applying a poorly conditioned operation to the estimate of $\boldsymbol{g}$.

It is also crucial that the minibatches be selected randomly. Computing an unbiased estimate of the expected gradient from a set of samples requires that those samples be independent. We also wish for two subsequent gradient estimates to be independent from each other, so two subsequent minibatches of examples should also be independent from each other. Many datasets are most naturally arranged in a way where successive examples are highly correlated. For example, we might have a dataset of medical data with a long list of blood sample test results. This list might be arranged so that first we have five blood samples taken at different times from the first patient, then we have three blood samples taken from the second patient, then the blood samples from the third patient, and so on. If we were to draw examples in order from this list, then each of our minibatches would be extremely biased, because it would represent primarily one patient out of the many patients in the dataset. In cases such as these where the order of the dataset holds some significance, it is necessary to shuffle the examples before selecting minibatches. For very large datasets, for example datasets containing billions of examples in a data center, it can be impractical to sample examples truly uniformly at random every time we want to construct a minibatch. Fortunately, in practice it is usually sufficient to shuffle the order of the dataset once and then store it in shuffled fashion. This will impose a fixed set of possible minibatches of consecutive examples that all models trained thereafter will use, and each individual model will be forced to reuse this ordering every time it passes through the training data. However, this deviation from true random selection does not seem to have a significant detrimental effect. Failing to ever shuffle the examples in any way can seriously reduce the effectiveness of the algorithm.

Many optimization problems in machine learning decompose over examples well enough that we can compute entire separate updates over different examples in parallel. In other words, we can compute the update that minimizes $J(\boldsymbol{X})$ for one minibatch of examples $\boldsymbol{X}$ at the same time that we compute the update for several other minibatches. Such asynchronous parallel distributed approaches are discussed further in section 12.1.3.

An interesting motivation for minibatch stochastic gradient descent is that it follows the gradient of the true *generalization error* (equation 8.2) so long as no examples are repeated. Most implementations of minibatch stochastic gradient

descent shuffle the dataset once and then pass through it multiple times. On the first pass, each minibatch is used to compute an unbiased estimate of the true generalization error. On the second pass, the estimate becomes biased because it is formed by re-sampling values that have already been used, rather than obtaining new fair samples from the data generating distribution.

The fact that stochastic gradient descent minimizes generalization error is easiest to see in the online learning case, where examples or minibatches are drawn from a **stream** of data. In other words, instead of receiving a fixed-size training set, the learner is similar to a living being who sees a new example at each instant, with every example $(\boldsymbol{x}, y)$ coming from the data generating distribution $p_{\text{data}}(\boldsymbol{x}, y)$. In this scenario, examples are never repeated; every experience is a fair sample from p_{data}.

The equivalence is easiest to derive when both $\boldsymbol{x}$ and y are discrete. In this case, the generalization error (equation 8.2) can be written as a sum

$$J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) L(f(\boldsymbol{x}; \boldsymbol{\theta}), y), \tag{8.7}$$

with the exact gradient

$$\boldsymbol{g} = \quad _{\boldsymbol{\theta}} J^*(\boldsymbol{\theta}) = \sum_{\boldsymbol{x}} \sum_{y} p_{\text{data}}(\boldsymbol{x}, y) \quad _{\boldsymbol{\theta}} L(f(\boldsymbol{x}; \boldsymbol{\theta}), y). \tag{8.8}$$

We have already seen the same fact demonstrated for the log-likelihood in equation 8.5 and equation 8.6; we observe now that this holds for other functions L besides the likelihood. A similar result can be derived when $\boldsymbol{x}$ and y are continuous, under mild assumptions regarding p_{data} and L.

Hence, we can obtain an unbiased estimator of the exact gradient of the generalization error by sampling a minibatch of examples $\boldsymbol{x}^{(1)}, \ldots \boldsymbol{x}^{(m)}$ with corresponding targets $y^{(i)}$ from the data generating distribution p_{data}, and computing the gradient of the loss with respect to the parameters for that minibatch:

$$\hat{\boldsymbol{g}} = \frac{1}{m} \quad _{\boldsymbol{\theta}} \sum_{i} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \tag{8.9}$$

Updating $\boldsymbol{\theta}$ in the direction of $\hat{\boldsymbol{g}}$ performs SGD on the generalization error.

Of course, this interpretation only applies when examples are not reused. Nonetheless, it is usually best to make several passes through the training set, unless the training set is extremely large. When multiple such epochs are used, only the first epoch follows the unbiased gradient of the generalization error, but

of course, the additional epochs usually provide enough benefit due to decreased training error to offset the harm they cause by increasing the gap between training error and test error.

With some datasets growing rapidly in size, faster than computing power, it is becoming more common for machine learning applications to use each training example only once or even to make an incomplete pass through the training set. When using an extremely large training set, overfitting is not an issue, so underfitting and computational efficiency become the predominant concerns. See also Bottou and Bousquet (2008) for a discussion of the effect of computational bottlenecks on generalization error, as the number of training examples grows.

8.2 Challenges in Neural Network Optimization

Optimization in general is an extremely difficult task. Traditionally, machine learning has avoided the difficulty of general optimization by carefully designing the objective function and constraints to ensure that the optimization problem is convex. When training neural networks, we must confront the general non-convex case. Even convex optimization is not without its complications. In this section, we summarize several of the most prominent challenges involved in optimization for training deep models.

8.2.1 Ill-Conditioning

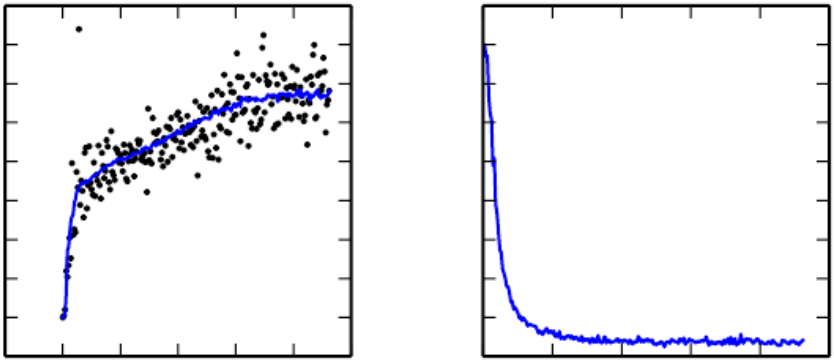
Some challenges arise even when optimizing convex functions. Of these, the most prominent is ill-conditioning of the Hessian matrix $\boldsymbol{H}$. This is a very general problem in most numerical optimization, convex or otherwise, and is described in more detail in section 4.3.1.

The ill-conditioning problem is generally believed to be present in neural network training problems. Ill-conditioning can manifest by causing SGD to get "stuck" in the sense that even very small steps increase the cost function.

Recall from equation 4.9 that a second-order Taylor series expansion of the cost function predicts that a gradient descent step of $\epsilon \boldsymbol{g}$ will add

$$\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g} \quad \epsilon \boldsymbol{g}^\top \boldsymbol{g} \tag{8.10}$$

to the cost. Ill-conditioning of the gradient becomes a problem when $\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ exceeds $\epsilon \boldsymbol{g}^\top \boldsymbol{g}$. To determine whether ill-conditioning is detrimental to a neural network training task, one can monitor the squared gradient norm $\boldsymbol{g}^\top \boldsymbol{g}$ and



the $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ term. In many cases, the gradient norm does not shrink significantly throughout learning, but the $\boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g}$ term grows by more than an order of magnitude. The result is that learning becomes very slow despite the presence of a strong gradient because the learning rate must be shrunk to compensate for even stronger curvature. Figure 8.1 shows an example of the gradient increasing significantly during the successful training of a neural network.

Though ill-conditioning is present in other settings besides neural network training, some of the techniques used to combat it in other contexts are less applicable to neural networks. For example, Newton's method is an excellent tool for minimizing convex functions with poorly conditioned Hessian matrices, but in the subsequent sections we will argue that Newton's method requires significant modification before it can be applied to neural networks.

8.2.2 Local Minima

One of the most prominent features of a convex optimization problem is that it can be reduced to the problem of finding a local minimum. Any local minimum is

guaranteed to be a global minimum. Some convex functions have a flat region at the bottom rather than a single global minimum point, but any point within such a flat region is an acceptable solution. When optimizing a convex function, we know that we have reached a good solution if we find a critical point of any kind.

With non-convex functions, such as neural nets, it is possible to have many local minima. Indeed, nearly any deep model is essentially guaranteed to have an extremely large number of local minima. However, as we will see, this is not necessarily a major problem.

Neural networks and any models with multiple equivalently parametrized latent variables all have multiple local minima because of the **model identifiability** problem. A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters. Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other. For example, we could take a neural network and modify layer 1 by swapping the incoming weight vector for unit i with the incoming weight vector for unit j, then doing the same for the outgoing weight vectors. If we have m layers with n units each, then there are $n!^m$ ways of arranging the hidden units. This kind of non-identifiability is known as **weight space symmetry**.

In addition to weight space symmetry, many kinds of neural networks have additional causes of non-identifiability. For example, in any rectified linear or maxout network, we can scale all of the incoming weights and biases of a unit by α if we also scale all of its outgoing weights by $\frac{1}{\alpha}$. This means that—if the cost function does not include terms such as weight decay that depend directly on the weights rather than the models' outputs—every local minimum of a rectified linear or maxout network lies on an $(m \quad n)$-dimensional hyperbola of equivalent local minima.

These model identifiability issues mean that there can be an extremely large or even uncountably infinite amount of local minima in a neural network cost function. However, all of these local minima arising from non-identifiability are equivalent to each other in cost function value. As a result, these local minima are not a problematic form of non-convexity.

Local minima can be problematic if they have high cost in comparison to the global minimum. One can construct small neural networks, even without hidden units, that have local minima with higher cost than the global minimum (Sontag and Sussman, 1989; Brady *et al.*, 1989; Gori and Tesi, 1992). If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms.

It remains an open question whether there are many local minima of high cost

for networks of practical interest and whether optimization algorithms encounter them. For many years, most practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case. The problem remains an active area of research, but experts now suspect that, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than to find a point in parameter space that has low but not minimal cost (Saxe *et al.*, 2013; Dauphin *et al.*, 2014; Goodfellow *et al.*, 2015; Choromanska *et al.*, 2014).

Many practitioners attribute nearly all difficulty with neural network optimization to local minima. We encourage practitioners to carefully test for specific problems. A test that can rule out local minima as the problem is to plot the norm of the gradient over time. If the norm of the gradient does not shrink to insignificant size, the problem is neither local minima nor any other kind of critical point. This kind of negative test can rule out local minima. In high dimensional spaces, it can be very difficult to positively establish that local minima are the problem. Many structures other than local minima also have small gradients.

8.2.3 Plateaus, Saddle Points and Other Flat Regions

For many high-dimensional non-convex functions, local minima (and maxima) are in fact rare compared to another kind of point with zero gradient: a saddle point. Some points around a saddle point have greater cost than the saddle point, while others have a lower cost. At a saddle point, the Hessian matrix has both positive and negative eigenvalues. Points lying along eigenvectors associated with positive eigenvalues have greater cost than the saddle point, while points lying along negative eigenvalues have lower value. We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section. See figure 4.5 for an illustration.

Many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher dimensional spaces, local minima are rare and saddle points are more common. For a function $f : \mathbb{R}^n \quad \mathbb{R}$ of this type, the expected ratio of the number of saddle points to local minima grows exponentially with n. To understand the intuition behind this behavior, observe that the Hessian matrix at a local minimum has only positive eigenvalues. The Hessian matrix at a saddle point has a mixture of positive and negative eigenvalues. Imagine that the sign of each eigenvalue is generated by flipping a coin. In a single dimension, it is easy to obtain a local minimum by tossing a coin and getting heads once. In n-dimensional space, it is exponentially unlikely that all n coin tosses will

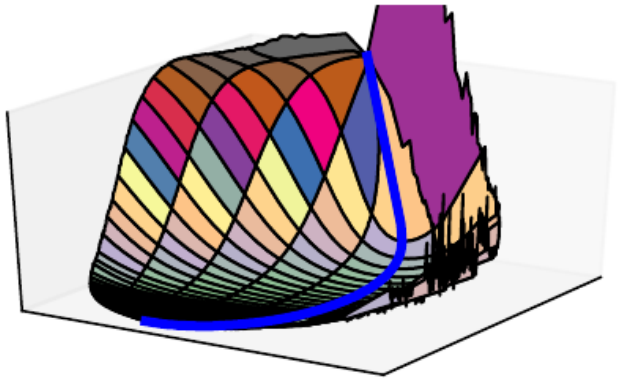
be heads. See Dauphin *et al.* (2014) for a review of the relevant theoretical work.

An amazing property of many random functions is that the eigenvalues of the Hessian become more likely to be positive as we reach regions of lower cost. In our coin tossing analogy, this means we are more likely to have our coin come up heads n times if we are at a critical point with low cost. This means that local minima are much more likely to have low cost than high cost. Critical points with high cost are far more likely to be saddle points. Critical points with extremely high cost are more likely to be local maxima.

This happens for many classes of random functions. Does it happen for neural networks? Baldi and Hornik (1989) showed theoretically that shallow autoencoders (feedforward networks trained to copy their input to their output, described in chapter 14) with no nonlinearities have global minima and saddle points but no local minima with higher cost than the global minimum. They observed without proof that these results extend to deeper networks without nonlinearities. The output of such networks is a linear function of their input, but they are useful to study as a model of nonlinear neural networks because their loss function is a non-convex function of their parameters. Such networks are essentially just multiple matrices composed together. Saxe *et al.* (2013) provided exact solutions to the complete learning dynamics in such networks and showed that learning in these models captures many of the qualitative features observed in the training of deep models with nonlinear activation functions. Dauphin *et al.* (2014) showed experimentally that real neural networks also have loss functions that contain very many high-cost saddle points. Choromanska *et al.* (2014) provided additional theoretical arguments, showing that another class of high-dimensional random functions related to neural networks does so as well.

What are the implications of the proliferation of saddle points for training algorithms? For first-order optimization algorithms that use only gradient information, the situation is unclear. The gradient can often become very small near a saddle point. On the other hand, gradient descent empirically seems to be able to escape saddle points in many cases. Goodfellow *et al.* (2015) provided visualizations of several learning trajectories of state-of-the-art neural networks, with an example given in figure 8.2. These visualizations show a flattening of the cost function near a prominent saddle point where the weights are all zero, but they also show the gradient descent trajectory rapidly escaping this region. Goodfellow *et al.* (2015) also argue that continuous-time gradient descent may be shown analytically to be repelled from, rather than attracted to, a nearby saddle point, but the situation may be different for more realistic uses of gradient descent.

For Newton's method, it is clear that saddle points constitute a problem.



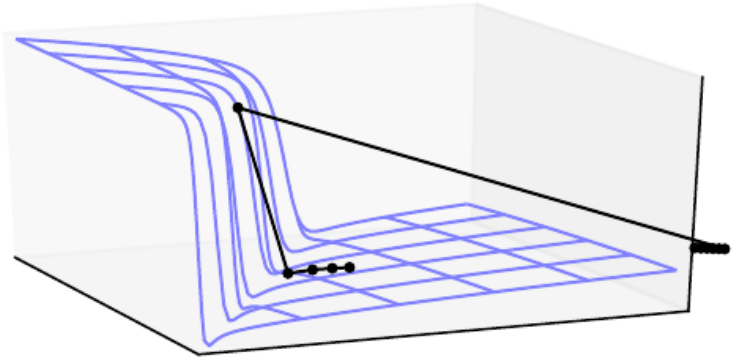
Gradient descent is designed to move "downhill" and is not explicitly designed to seek a critical point. Newton's method, however, is designed to solve for a point where the gradient is zero. Without appropriate modification, it can jump to a saddle point. The proliferation of saddle points in high dimensional spaces presumably explains why second-order methods have not succeeded in replacing gradient descent for neural network training. Dauphin *et al.* (2014) introduced a **saddle-free Newton method** for second-order optimization and showed that it improves significantly over the traditional version. Second-order methods remain difficult to scale to large neural networks, but this saddle-free approach holds promise if it could be scaled.

There are other kinds of points with zero gradient besides minima and saddle points. There are also maxima, which are much like saddle points from the perspective of optimization—many algorithms are not attracted to them, but unmodified Newton's method is. Maxima of many classes of random functions become exponentially rare in high dimensional space, just like minima do.

There may also be wide, flat regions of constant value. In these locations, the gradient and also the Hessian are all zero. Such degenerate locations pose major problems for all numerical optimization algorithms. In a convex problem, a wide, flat region must consist entirely of global minima, but in a general optimization problem, such a region could correspond to a high value of the objective function.

8.2.4 Cliffs and Exploding Gradients

Neural networks with many layers often have extremely steep regions resembling cliffs, as illustrated in figure 8.3. These result from the multiplication of several large weights together. On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off of the cliff structure altogether.



The cliff can be dangerous whether we approach it from above or from below, but fortunately its most serious consequences can be avoided using the **gradient clipping** heuristic described in section 10.11.1. The basic idea is to recall that the gradient does not specify the optimal step size, but only the optimal direction within an infinitesimal region. When the traditional gradient descent algorithm proposes to make a very large step, the gradient clipping heuristic intervenes to reduce the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent. Cliff structures are most common in the cost functions for recurrent neural networks, because such models involve a multiplication of many factors, with one factor for each time step. Long temporal sequences thus incur an extreme amount of multiplication.

8.2.5 Long-Term Dependencies

Another difficulty that neural network optimization algorithms must overcome arises when the computational graph becomes extremely deep. Feedforward networks with many layers have such deep computational graphs. So do recurrent networks, described in chapter 10, which construct very deep computational graphs

by repeatedly applying the same operation at each time step of a long temporal sequence. Repeated application of the same parameters gives rise to especially pronounced difficulties.

For example, suppose that a computational graph contains a path that consists of repeatedly multiplying by a matrix $\boldsymbol{W}$. After t steps, this is equivalent to multiplying by $\boldsymbol{W}^t$. Suppose that $\boldsymbol{W}$ has an eigendecomposition $\boldsymbol{W} = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$. In this simple case, it is straightforward to see that

$$\boldsymbol{W}^t = \left(\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^{-1}\right)^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^{-1}. \tag{8.11}$$

Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude or vanish if they are less than 1 in magnitude. The **vanishing and exploding gradient problem** refers to the fact that gradients through such a graph are also scaled according to $\text{diag}(\boldsymbol{\lambda})^t$. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable. The cliff structures described earlier that motivate gradient clipping are an example of the exploding gradient phenomenon.

The repeated multiplication by $\boldsymbol{W}$ at each time step described here is very similar to the **power method** algorithm used to find the largest eigenvalue of a matrix $\boldsymbol{W}$ and the corresponding eigenvector. From this point of view it is not surprising that $\boldsymbol{x}^\top\boldsymbol{W}^t$ will eventually discard all components of $\boldsymbol{x}$ that are orthogonal to the principal eigenvector of $\boldsymbol{W}$.

Recurrent networks use the same matrix $\boldsymbol{W}$ at each time step, but feedforward networks do not, so even very deep feedforward networks can largely avoid the vanishing and exploding gradient problem (Sussillo, 2014).

We defer a further discussion of the challenges of training recurrent networks until section 10.7, after recurrent networks have been described in more detail.

8.2.6 Inexact Gradients

Most optimization algorithms are designed with the assumption that we have access to the exact gradient or Hessian matrix. In practice, we usually only have a noisy or even biased estimate of these quantities. Nearly every deep learning algorithm relies on sampling-based estimates at least insofar as using a minibatch of training examples to compute the gradient.

In other cases, the objective function we want to minimize is actually intractable. When the objective function is intractable, typically its gradient is intractable as well. In such cases we can only approximate the gradient. These issues mostly arise

with the more advanced models in part III. For example, contrastive divergence gives a technique for approximating the gradient of the intractable log-likelihood of a Boltzmann machine.

Various neural network optimization algorithms are designed to account for imperfections in the gradient estimate. One can also avoid the problem by choosing a surrogate loss function that is easier to approximate than the true loss.

8.2.7 Poor Correspondence between Local and Global Structure

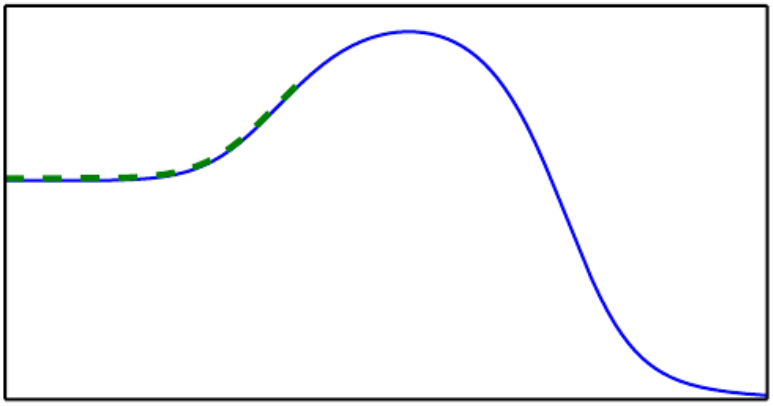
Many of the problems we have discussed so far correspond to properties of the loss function at a single point—it can be difficult to make a single step if $J(\boldsymbol{\theta})$ is poorly conditioned at the current point $\boldsymbol{\theta}$, or if $\boldsymbol{\theta}$ lies on a cliff, or if $\boldsymbol{\theta}$ is a saddle point hiding the opportunity to make progress downhill from the gradient.

It is possible to overcome all of these problems at a single point and still perform poorly if the direction that results in the most improvement locally does not point toward distant regions of much lower cost.

Goodfellow *et al.* (2015) argue that much of the runtime of training is due to the length of the trajectory needed to arrive at the solution. Figure 8.2 shows that the learning trajectory spends most of its time tracing out a wide arc around a mountain-shaped structure.

Much of research into the difficulties of optimization has focused on whether training arrives at a global minimum, a local minimum, or a saddle point, but in practice neural networks do not arrive at a critical point of any kind. Figure 8.1 shows that neural networks often do not arrive at a region of small gradient. Indeed, such critical points do not even necessarily exist. For example, the loss function $\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$ can lack a global minimum point and instead asymptotically approach some value as the model becomes more confident. For a classifier with discrete y and $p(y \mid \boldsymbol{x})$ provided by a softmax, the negative log-likelihood can become arbitrarily close to zero if the model is able to correctly classify every example in the training set, but it is impossible to actually reach the value of zero. Likewise, a model of real values $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$ can have negative log-likelihood that asymptotes to negative infinity—if $f(\boldsymbol{\theta})$ is able to correctly predict the value of all training set y targets, the learning algorithm will increase β without bound. See figure 8.4 for an example of a failure of local optimization to find a good cost function value even in the absence of any local minima or saddle points.

Future research will need to develop further understanding of the factors that influence the length of the learning trajectory and better characterize the outcome



of the process.

Many existing research directions are aimed at finding good initial points for problems that have difficult global structure, rather than developing algorithms that use non-local moves.

Gradient descent and essentially all learning algorithms that are effective for training neural networks are based on making small, local moves. The previous sections have primarily focused on how the correct direction of these local moves can be difficult to compute. We may be able to compute some properties of the objective function, such as its gradient, only approximately, with bias or variance in our estimate of the correct direction. In these cases, local descent may or may not define a reasonably short path to a valid solution, but we are not actually able to follow the local descent path. The objective function may have issues such as poor conditioning or discontinuous gradients, causing the region where the gradient provides a good model of the objective function to be very small. In these cases, local descent with steps of size ϵ may define a reasonably short path to the solution, but we are only able to compute the local descent direction with steps of size $\delta < \epsilon$. In these cases, local descent may or may not define a path to the solution, but the path contains many steps, so following the path incurs a

high computational cost. Sometimes local information provides us no guide, when the function has a wide flat region, or if we manage to land exactly on a critical point (usually this latter scenario only happens to methods that solve explicitly for critical points, such as Newton's method). In these cases, local descent does not define a path to a solution at all. In other cases, local moves can be too greedy and lead us along a path that moves downhill but away from any solution, as in figure 8.4, or along an unnecessarily long trajectory to the solution, as in figure 8.2. Currently, we do not understand which of these problems are most relevant to making neural network optimization difficult, and this is an active area of research.

Regardless of which of these problems are most significant, all of them might be avoided if there exists a region of space connected reasonably directly to a solution by a path that local descent can follow, and if we are able to initialize learning within that well-behaved region. This last view suggests research into choosing good initial points for traditional optimization algorithms to use.

8.2.8 Theoretical Limits of Optimization

Several theoretical results show that there are limits on the performance of any optimization algorithm we might design for neural networks (Blum and Rivest, 1992; Judd, 1989; Wolpert and MacReady, 1997). Typically these results have little bearing on the use of neural networks in practice.

Some theoretical results apply only to the case where the units of a neural network output discrete values. However, most neural network units output smoothly increasing values that make optimization via local search feasible. Some theoretical results show that there exist problem classes that are intractable, but it can be difficult to tell whether a particular problem falls into that class. Other results show that finding a solution for a network of a given size is intractable, but in practice we can find a solution easily by using a larger network for which many more parameter settings correspond to an acceptable solution. Moreover, in the context of neural network training, we usually do not care about finding the exact minimum of a function, but seek only to reduce its value sufficiently to obtain good generalization error. Theoretical analysis of whether an optimization algorithm can accomplish this goal is extremely difficult. Developing more realistic bounds on the performance of optimization algorithms therefore remains an important goal for machine learning research.

8.3 Basic Algorithms

We have previously introduced the gradient descent (section 4.3) algorithm that follows the gradient of an entire training set downhill. This may be accelerated considerably by using stochastic gradient descent to follow the gradient of randomly selected minibatches downhill, as discussed in section 5.9 and section 8.1.3.

8.3.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) and its variants are probably the most used optimization algorithms for machine learning in general and for deep learning in particular. As discussed in section 8.1.3, it is possible to obtain an unbiased estimate of the gradient by taking the average gradient on a minibatch of m examples drawn i.i.d from the data generating distribution.

Algorithm 8.1 shows how to follow this estimate of the gradient downhill.

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient estimate: $\hat{\boldsymbol{g}}$ $+\frac{1}{m}$ $\boldsymbol{\theta} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Apply update: $\boldsymbol{\theta}$ $\boldsymbol{\theta}$ $\epsilon \hat{\boldsymbol{g}}$
end while

A crucial parameter for the SGD algorithm is the learning rate. Previously, we have described SGD as using a fixed learning rate ϵ. In practice, it is necessary to gradually decrease the learning rate over time, so we now denote the learning rate at iteration k as ϵ_k.

This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not vanish even when we arrive at a minimum. By comparison, the true gradient of the total cost function becomes small and then when we approach and reach a minimum using batch gradient descent, so batch gradient descent can use a fixed learning rate. A sufficient condition to guarantee convergence of SGD is that

$$\sum_{k=1}^{\infty} \epsilon_k = \quad , \quad \text{and} \tag{8.12}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \quad . \tag{8.13}$$

In practice, it is common to decay the learning rate linearly until iteration τ:

$$\epsilon_k = (1 \quad \alpha)\epsilon^{\,0} + \alpha\epsilon_\tau \tag{8.14}$$

with $\alpha = \frac{k}{\tau}$. After iteration τ, it is common to leave ϵ constant.

The learning rate may be chosen by trial and error, but it is usually best to choose it by monitoring learning curves that plot the objective function as a function of time. This is more of an art than a science, and most guidance on this subject should be regarded with some skepticism. When using the linear schedule, the parameters to choose are ϵ_0, ϵ_τ, and τ. Usually τ may be set to the number of iterations required to make a few hundred passes through the training set. Usually ϵ_τ should be set to roughly 1% the value of ϵ_0. The main question is how to set ϵ_0. If it is too large, the learning curve will show violent oscillations, with the cost function often increasing significantly. Gentle oscillations are fine, especially if training with a stochastic cost function such as the cost function arising from the use of dropout. If the learning rate is too low, learning proceeds slowly, and if the initial learning rate is too low, learning may become stuck with a high cost value. Typically, the optimal initial learning rate, in terms of total training time and the final cost value, is higher than the learning rate that yields the best performance after the first 100 iterations or so. Therefore, it is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing learning rate at this time, but not so high that it causes severe instability.

The most important property of SGD and related minibatch or online gradient-based optimization is that computation time per update does not grow with the number of training examples. This allows convergence even when the number of training examples becomes very large. For a large enough dataset, SGD may converge to within some fixed tolerance of its final test set error before it has processed the entire training set.

To study the convergence rate of an optimization algorithm it is common to measure the **excess error** $J(\boldsymbol{\theta}) \quad \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, which is the amount that the current cost function exceeds the minimum possible cost. When SGD is applied to a convex problem, the excess error is $O(\frac{1}{\sqrt{k}})$ after k iterations, while in the strongly convex case it is $O(\frac{1}{k})$. These bounds cannot be improved unless extra conditions are assumed. Batch gradient descent enjoys better convergence rates than stochastic gradient descent in theory. However, the Cramér-Rao bound (Cramér, 1946; Rao, 1945) states that generalization error cannot decrease faster than $O(\frac{1}{k})$. Bottou

and Bousquet (2008) argue that it therefore may not be worthwhile to pursue an optimization algorithm that converges faster than $O(\frac{1}{k})$ for machine learning tasks—faster convergence presumably corresponds to overfitting. Moreover, the asymptotic analysis obscures many advantages that stochastic gradient descent has after a small number of steps. With large datasets, the ability of SGD to make rapid initial progress while evaluating the gradient for only very few examples outweighs its slow asymptotic convergence. Most of the algorithms described in the remainder of this chapter achieve benefits that matter in practice but are lost in the constant factors obscured by the $O(\frac{1}{k})$ asymptotic analysis. One can also trade off the benefits of both batch and stochastic gradient descent by gradually increasing the minibatch size during the course of learning.

For more information on SGD, see Bottou (1998).

8.3.2 Momentum

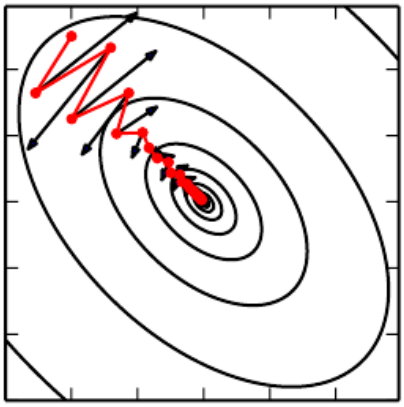
While stochastic gradient descent remains a very popular optimization strategy, learning with it can sometimes be slow. The method of momentum (Polyak, 1964) is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. The effect of momentum is illustrated in figure 8.5.

Formally, the momentum algorithm introduces a variable $\boldsymbol{v}$ that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space. The velocity is set to an exponentially decaying average of the negative gradient. The name **momentum** derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space, according to Newton's laws of motion. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector $\boldsymbol{v}$ may also be regarded as the momentum of the particle. A hyperparameter α $[0, 1)$ determines how quickly the contributions of previous gradients exponentially decay. The update rule is given by:

$$\boldsymbol{v} \quad \alpha\boldsymbol{v} \quad \epsilon \quad \boldsymbol{\theta} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right), \tag{8.15}$$

$$\boldsymbol{\theta} \quad \boldsymbol{\theta} + \boldsymbol{v}. \tag{8.16}$$

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\boldsymbol{\theta} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$. The larger α is relative to ϵ, the more previous gradients affect the current direction. The SGD algorithm with momentum is given in algorithm 8.2.



Previously, the size of the step was simply the norm of the gradient multiplied by the learning rate. Now, the size of the step depends on how large and how aligned a *sequence* of gradients are. The step size is largest when many successive gradients point in exactly the same direction. If the momentum algorithm always observes gradient $\boldsymbol{g}$, then it will accelerate in the direction of $-\boldsymbol{g}$, until reaching a terminal velocity where the size of each step is

$$\frac{\epsilon \|\boldsymbol{g}\|}{1 - \alpha}. \tag{8.17}$$

It is thus helpful to think of the momentum hyperparameter in terms of $\frac{1}{1-\alpha}$. For example, $\alpha = .9$ corresponds to multiplying the maximum speed by 10 relative to the gradient descent algorithm.

Common values of α used in practice include .5, .9, and .99. Like the learning rate, α may also be adapted over time. Typically it begins with a small value and is later raised. It is less important to adapt α over time than to shrink ϵ over time.

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

We can view the momentum algorithm as simulating a particle subject to continuous-time Newtonian dynamics. The physical analogy can help to build intuition for how the momentum and gradient descent algorithms behave.

The position of the particle at any point in time is given by $\boldsymbol{\theta}(t)$. The particle experiences net force $\boldsymbol{f}(t)$. This force causes the particle to accelerate:

$$\boldsymbol{f}(t) = \frac{\partial^2}{\partial t^2} \boldsymbol{\theta}(t). \tag{8.18}$$

Rather than viewing this as a second-order differential equation of the position, we can introduce the variable $\boldsymbol{v}(t)$ representing the velocity of the particle at time t and rewrite the Newtonian dynamics as a first-order differential equation:

$$\boldsymbol{v}(t) = \frac{\partial}{\partial t} \boldsymbol{\theta}(t), \tag{8.19}$$

$$\boldsymbol{f}(t) = \frac{\partial}{\partial t}\boldsymbol{v}(t). \tag{8.20}$$

The momentum algorithm then consists of solving the differential equations via numerical simulation. A simple numerical method for solving differential equations is Euler's method, which simply consists of simulating the dynamics defined by the equation by taking small, finite steps in the direction of each gradient.

This explains the basic form of the momentum update, but what specifically are the forces? One force is proportional to the negative gradient of the cost function: $_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. This force pushes the particle downhill along the cost function surface. The gradient descent algorithm would simply take a single step based on each gradient, but the Newtonian scenario used by the momentum algorithm instead uses this force to alter the velocity of the particle. We can think of the particle as being like a hockey puck sliding down an icy surface. Whenever it descends a steep part of the surface, it gathers speed and continues sliding in that direction until it begins to go uphill again.

One other force is necessary. If the only force is the gradient of the cost function, then the particle might never come to rest. Imagine a hockey puck sliding down one side of a valley and straight up the other side, oscillating back and forth forever, assuming the ice is perfectly frictionless. To resolve this problem, we add one other force, proportional to $_{\boldsymbol{v}}(t)$. In physics terminology, this force corresponds to viscous drag, as if the particle must push through a resistant medium such as syrup. This causes the particle to gradually lose energy over time and eventually converge to a local minimum.

Why do we use $_{\boldsymbol{v}}(t)$ and viscous drag in particular? Part of the reason to use $_{\boldsymbol{v}}(t)$ is mathematical convenience—an integer power of the velocity is easy to work with. However, other physical systems have other kinds of drag based on other integer powers of the velocity. For example, a particle traveling through the air experiences turbulent drag, with force proportional to the square of the velocity, while a particle moving along the ground experiences dry friction, with a force of constant magnitude. We can reject each of these options. Turbulent drag, proportional to the square of the velocity, becomes very weak when the velocity is small. It is not powerful enough to force the particle to come to rest. A particle with a non-zero initial velocity that experiences only the force of turbulent drag will move away from its initial position forever, with the distance from the starting point growing like $O(\log t)$. We must therefore use a lower power of the velocity. If we use a power of zero, representing dry friction, then the force is too strong. When the force due to the gradient of the cost function is small but non-zero, the constant force due to friction can cause the particle to come to rest before reaching a local minimum. Viscous drag avoids both of these problems—it is weak enough

that the gradient can continue to cause motion until a minimum is reached, but strong enough to prevent motion if the gradient does not justify moving.

8.3.3 Nesterov Momentum

Sutskever *et al.* (2013) introduced a variant of the momentum algorithm that was inspired by Nesterov's accelerated gradient method (Nesterov, 1983, 2004). The update rules in this case are given by:

$$\boldsymbol{v} \quad \alpha\boldsymbol{v} \quad \epsilon \quad \boldsymbol{\theta} \left[\frac{1}{m} \sum_{i=1}^{m} L\Big(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha\boldsymbol{v}), \boldsymbol{y}^{(i)} \Big) \right], \tag{8.21}$$

$$\boldsymbol{\theta} \quad \boldsymbol{\theta} + \boldsymbol{v}, \tag{8.22}$$

where the parameters α and ϵ play a similar role as in the standard momentum method. The difference between Nesterov momentum and standard momentum is where the gradient is evaluated. With Nesterov momentum the gradient is evaluated after the current velocity is applied. Thus one can interpret Nesterov momentum as attempting to add a *correction factor* to the standard method of momentum. The complete Nesterov momentum algorithm is presented in algorithm 8.3.

In the convex batch gradient case, Nesterov momentum brings the rate of convergence of the excess error from $O(1/k)$ (after k steps) to $O(1/k^2)$ as shown by Nesterov (1983). Unfortunately, in the stochastic gradient case, Nesterov momentum does not improve the rate of convergence.

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
  Apply interim update: $\tilde{\boldsymbol{\theta}} \quad \boldsymbol{\theta} + \alpha\boldsymbol{v}$
  Compute gradient (at interim point): $\boldsymbol{g} \quad \frac{1}{m} \quad \tilde{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
  Compute velocity update: $\boldsymbol{v} \quad \alpha\boldsymbol{v} \quad \epsilon\boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} \quad \boldsymbol{\theta} + \boldsymbol{v}$
end while

8.4 Parameter Initialization Strategies

Some optimization algorithms are not iterative by nature and simply solve for a solution point. Other optimization algorithms are iterative by nature but, when applied to the right class of optimization problems, converge to acceptable solutions in an acceptable amount of time regardless of initialization. Deep learning training algorithms usually do not have either of these luxuries. Training algorithms for deep learning models are usually iterative in nature and thus require the user to specify some initial point from which to begin the iterations. Moreover, training deep models is a sufficiently difficult task that most algorithms are strongly affected by the choice of initialization. The initial point can determine whether the algorithm converges at all, with some initial points being so unstable that the algorithm encounters numerical difficulties and fails altogether. When learning does converge, the initial point can determine how quickly learning converges and whether it converges to a point with high or low cost. Also, points of comparable cost can have wildly varying generalization error, and the initial point can affect the generalization as well.

Modern initialization strategies are simple and heuristic. Designing improved initialization strategies is a difficult task because neural network optimization is not yet well understood. Most initialization strategies are based on achieving some nice properties when the network is initialized. However, we do not have a good understanding of which of these properties are preserved under which circumstances after learning begins to proceed. A further difficulty is that some initial points may be beneficial from the viewpoint of optimization but detrimental from the viewpoint of generalization. Our understanding of how the initial point affects generalization is especially primitive, offering little to no guidance for how to select the initial point.

Perhaps the only property known with complete certainty is that the initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters. If they have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way. Even if the model or training algorithm is capable of using stochasticity to compute different updates for different units (for example, if one trains with dropout), it is usually best to initialize each unit to compute a different function from all of the other units. This may help to make sure that no input patterns are lost in the null space of forward propagation and no gradient patterns are lost in the null space of back-propagation. The goal of having each unit compute a different function

motivates random initialization of the parameters. We could explicitly search for a large set of basis functions that are all mutually different from each other, but this often incurs a noticeable computational cost. For example, if we have at most as many outputs as inputs, we could use Gram-Schmidt orthogonalization on an initial weight matrix, and be guaranteed that each unit computes a very different function from each other unit. Random initialization from a high-entropy distribution over a high-dimensional space is computationally cheaper and unlikely to assign any units to compute the same function as each other.

Typically, we set the biases for each unit to heuristically chosen constants, and initialize only the weights randomly. Extra parameters, for example, parameters encoding the conditional variance of a prediction, are usually set to heuristically chosen constants much like the biases are.

We almost always initialize all the weights in the model to values drawn randomly from a Gaussian or uniform distribution. The choice of Gaussian or uniform distribution does not seem to matter very much, but has not been exhaustively studied. The scale of the initial distribution, however, does have a large effect on both the outcome of the optimization procedure and on the ability of the network to generalize.

Larger initial weights will yield a stronger symmetry breaking effect, helping to avoid redundant units. They also help to avoid losing signal during forward or back-propagation through the linear component of each layer—larger values in the matrix result in larger outputs of matrix multiplication. Initial weights that are too large may, however, result in exploding values during forward propagation or back-propagation. In recurrent networks, large weights can also result in **chaos** (such extreme sensitivity to small perturbations of the input that the behavior of the deterministic forward propagation procedure appears random). To some extent, the exploding gradient problem can be mitigated by gradient clipping (thresholding the values of the gradients before performing a gradient descent step). Large weights may also result in extreme values that cause the activation function to saturate, causing complete loss of gradient through saturated units. These competing factors determine the ideal initial scale of the weights.

The perspectives of regularization and optimization can give very different insights into how we should initialize a network. The optimization perspective suggests that the weights should be large enough to propagate information successfully, but some regularization concerns encourage making them smaller. The use of an optimization algorithm such as stochastic gradient descent that makes small incremental changes to the weights and tends to halt in areas that are nearer to the initial parameters (whether due to getting stuck in a region of low gradient, or

due to triggering some early stopping criterion based on overfitting) expresses a prior that the final parameters should be close to the initial parameters. Recall from section 7.8 that gradient descent with early stopping is equivalent to weight decay for some models. In the general case, gradient descent with early stopping is not the same as weight decay, but does provide a loose analogy for thinking about the effect of initialization. We can think of initializing the parameters $\boldsymbol{\theta}$ to $\boldsymbol{\theta}_0$ as being similar to imposing a Gaussian prior $p(\boldsymbol{\theta})$ with mean $\boldsymbol{\theta}_0$. From this point of view, it makes sense to choose $\boldsymbol{\theta}_0$ to be near 0. This prior says that it is more likely that units do not interact with each other than that they do interact. Units interact only if the likelihood term of the objective function expresses a strong preference for them to interact. On the other hand, if we initialize $\boldsymbol{\theta}_0$ to large values, then our prior specifies which units should interact with each other, and how they should interact.

Some heuristics are available for choosing the initial scale of the weights. One heuristic is to initialize the weights of a fully connected layer with m inputs and n outputs by sampling each weight from $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$, while Glorot and Bengio (2010) suggest using the **normalized initialization**

$$\mathrm{W}_{i,j} \quad U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right). \tag{8.23}$$

This latter heuristic is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance. The formula is derived using the assumption that the network consists only of a chain of matrix multiplications, with no nonlinearities. Real neural networks obviously violate this assumption, but many strategies designed for the linear model perform reasonably well on its nonlinear counterparts.

Saxe *et al.* (2013) recommend initializing to random orthogonal matrices, with a carefully chosen scaling or **gain** factor g that accounts for the nonlinearity applied at each layer. They derive specific values of the scaling factor for different types of nonlinear activation functions. This initialization scheme is also motivated by a model of a deep network as a sequence of matrix multiplies without nonlinearities. Under such a model, this initialization scheme guarantees that the total number of training iterations required to reach convergence is independent of depth.

Increasing the scaling factor g pushes the network toward the regime where activations increase in norm as they propagate forward through the network and gradients increase in norm as they propagate backward. Sussillo (2014) showed that setting the gain factor correctly is sufficient to train networks as deep as

1,000 layers, without needing to use orthogonal initializations. A key insight of this approach is that in feedforward networks, activations and gradients can grow or shrink on each step of forward or back-propagation, following a random walk behavior. This is because feedforward networks use a different weight matrix at each layer. If this random walk is tuned to preserve norms, then feedforward networks can mostly avoid the vanishing and exploding gradients problem that arises when the same weight matrix is used at each step, described in section 8.2.5.

Unfortunately, these optimal criteria for initial weights often do not lead to optimal performance. This may be for three different reasons. First, we may be using the wrong criteria—it may not actually be beneficial to preserve the norm of a signal throughout the entire network. Second, the properties imposed at initialization may not persist after learning has begun to proceed. Third, the criteria might succeed at improving the speed of optimization but inadvertently increase generalization error. In practice, we usually need to treat the scale of the weights as a hyperparameter whose optimal value lies somewhere roughly near but not exactly equal to the theoretical predictions.

One drawback to scaling rules that set all of the initial weights to have the same standard deviation, such as $\frac{1}{\sqrt{m}}$, is that every individual weight becomes extremely small when the layers become large. Martens (2010) introduced an alternative initialization scheme called **sparse initialization** in which each unit is initialized to have exactly k non-zero weights. The idea is to keep the total amount of input to the unit independent from the number of inputs m without making the magnitude of individual weight elements shrink with m. Sparse initialization helps to achieve more diversity among the units at initialization time. However, it also imposes a very strong prior on the weights that are chosen to have large Gaussian values. Because it takes a long time for gradient descent to shrink "incorrect" large values, this initialization scheme can cause problems for units such as maxout units that have several filters that must be carefully coordinated with each other.

When computational resources allow it, it is usually a good idea to treat the initial scale of the weights for each layer as a hyperparameter, and to choose these scales using a hyperparameter search algorithm described in section 11.4.2, such as random search. The choice of whether to use dense or sparse initialization can also be made a hyperparameter. Alternately, one can manually search for the best initial scales. A good rule of thumb for choosing the initial scales is to look at the range or standard deviation of activations or gradients on a single minibatch of data. If the weights are too small, the range of activations across the minibatch will shrink as the activations propagate forward through the network. By repeatedly identifying the first layer with unacceptably small activations and

increasing its weights, it is possible to eventually obtain a network with reasonable initial activations throughout. If learning is still too slow at this point, it can be useful to look at the range or standard deviation of the gradients as well as the activations. This procedure can in principle be automated and is generally less computationally costly than hyperparameter optimization based on validation set error because it is based on feedback from the behavior of the initial model on a single batch of data, rather than on feedback from a trained model on the validation set. While long used heuristically, this protocol has recently been specified more formally and studied by Mishkin and Matas (2015).

So far we have focused on the initialization of the weights. Fortunately, initialization of other parameters is typically easier.

The approach for setting the biases must be coordinated with the approach for settings the weights. Setting the biases to zero is compatible with most weight initialization schemes. There are a few situations where we may set some biases to non-zero values:

If a bias is for an output unit, then it is often beneficial to initialize the bias to obtain the right marginal statistics of the output. To do this, we assume that the initial weights are small enough that the output of the unit is determined only by the bias. This justifies setting the bias to the inverse of the activation function applied to the marginal statistics of the output in the training set. For example, if the output is a distribution over classes and this distribution is a highly skewed distribution with the marginal probability of class i given by element c_i of some vector $\boldsymbol{c}$, then we can set the bias vector $\boldsymbol{b}$ by solving the equation $\text{softmax}(\boldsymbol{b}) = \boldsymbol{c}$. This applies not only to classifiers but also to models we will encounter in Part III, such as autoencoders and Boltzmann machines. These models have layers whose output should resemble the input data $\boldsymbol{x}$, and it can be very helpful to initialize the biases of such layers to match the marginal distribution over $\boldsymbol{x}$.

Sometimes we may want to choose the bias to avoid causing too much saturation at initialization. For example, we may set the bias of a ReLU hidden unit to 0.1 rather than 0 to avoid saturating the ReLU at initialization. This approach is not compatible with weight initialization schemes that do not expect strong input from the biases though. For example, it is not recommended for use with random walk initialization (Sussillo, 2014).

Sometimes a unit controls whether other units are able to participate in a function. In such situations, we have a unit with output u and another unit h $[0, 1]$, and they are multiplied together to produce an output uh. We

can view h as a gate that determines whether $uh \approx u$ or $uh \approx 0$. In these situations, we want to set the bias for h so that $h \approx 1$ most of the time at initialization. Otherwise u does not have a chance to learn. For example, Jozefowicz *et al.* (2015) advocate setting the bias to 1 for the forget gate of the LSTM model, described in section 10.10.

Another common type of parameter is a variance or precision parameter. For example, we can perform linear regression with a conditional variance estimate using the model

$$p(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{w}^T \boldsymbol{x} + b, 1/\beta) \tag{8.24}$$

where β is a precision parameter. We can usually initialize variance or precision parameters to 1 safely. Another approach is to assume the initial weights are close enough to zero that the biases may be set while ignoring the effect of the weights, then set the biases to produce the correct marginal mean of the output, and set the variance parameters to the marginal variance of the output in the training set.

Besides these simple constant or random methods of initializing model parameters, it is possible to initialize model parameters using machine learning. A common strategy discussed in part III of this book is to initialize a supervised model with the parameters learned by an unsupervised model trained on the same inputs. One can also perform supervised training on a related task. Even performing supervised training on an unrelated task can sometimes yield an initialization that offers faster convergence than a random initialization. Some of these initialization strategies may yield faster convergence and better generalization because they encode information about the distribution in the initial parameters of the model. Others apparently perform well primarily because they set the parameters to have the right scale or set different units to compute different functions from each other.

8.5 Algorithms with Adaptive Learning Rates

Neural network researchers have long realized that the learning rate was reliably one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance. As we have discussed in sections 4.3 and 8.2, the cost is often highly sensitive to some directions in parameter space and insensitive to others. The momentum algorithm can mitigate these issues somewhat, but does so at the expense of introducing another hyperparameter. In the face of this, it is natural to ask if there is another way. If we believe that the directions of sensitivity are somewhat axis-aligned, it can make sense to use a separate learning

rate for each parameter, and automatically adapt these learning rates throughout the course of learning.

The **delta-bar-delta** algorithm (Jacobs, 1988) is an early heuristic approach to adapting individual learning rates for model parameters during training. The approach is based on a simple idea: if the partial derivative of the loss, with respect to a given model parameter, remains the same sign, then the learning rate should increase. If the partial derivative with respect to that parameter changes sign, then the learning rate should decrease. Of course, this kind of rule can only be applied to full batch optimization.

More recently, a number of incremental (or mini-batch-based) methods have been introduced that adapt the learning rates of model parameters. This section will briefly review a few of these algorithms.

8.5.1 AdaGrad

The **AdaGrad** algorithm, shown in algorithm 8.4, individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical squared values (Duchi *et al.*, 2011). The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. The net effect is greater progress in the more gently sloped directions of parameter space.

In the context of convex optimization, the AdaGrad algorithm enjoys some desirable theoretical properties. However, empirically it has been found that—for training deep neural network models—the accumulation of squared gradients *from the beginning of training* can result in a premature and excessive decrease in the effective learning rate. AdaGrad performs well for some but not all deep learning models.

8.5.2 RMSProp

The **RMSProp** algorithm (Hinton, 2012) modifies AdaGrad to perform better in the non-convex setting by changing the gradient accumulation into an exponentially weighted moving average. AdaGrad is designed to converge rapidly when applied to a convex function. When applied to a non-convex function to train a neural network, the learning trajectory may pass through many different structures and eventually arrive at a region that is a locally convex bowl. AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} =$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \quad \frac{1}{m} \quad \boldsymbol{\theta} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \quad \boldsymbol{r} + \boldsymbol{g} \quad \boldsymbol{g}$
Compute update: $\Delta_{\boldsymbol{\theta}} \quad \frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \quad \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \quad \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

have made the learning rate too small before arriving at such a convex structure. RMSProp uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of the AdaGrad algorithm initialized within that bowl.

RMSProp is shown in its standard form in algorithm 8.5 and combined with Nesterov momentum in algorithm 8.6. Compared to AdaGrad, the use of the moving average introduces a new hyperparameter, ρ, that controls the length scale of the moving average.

Empirically, RMSProp has been shown to be an effective and practical optimization algorithm for deep neural networks. It is currently one of the go-to optimization methods being employed routinely by deep learning practitioners.

8.5.3 Adam

Adam (Kingma and Ba, 2014) is yet another adaptive learning rate optimization algorithm and is presented in algorithm 8.7. The name "Adam" derives from the phrase "adaptive moments." In the context of the earlier algorithms, it is perhaps best seen as a variant on the combination of RMSProp and momentum with a few important distinctions. First, in Adam, momentum is incorporated directly as an estimate of the first order moment (with exponential weighting) of the gradient. The most straightforward way to add momentum to RMSProp is to apply momentum to the rescaled gradients. The use of momentum in combination with rescaling does not have a clear theoretical motivation. Second, Adam includes

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \quad \frac{1}{m} \quad \boldsymbol{\theta} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \quad \rho\boldsymbol{r} + (1 \quad \rho)\boldsymbol{g}$
Compute parameter update: $\Delta\boldsymbol{\theta} = \frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \quad \boldsymbol{g}. \; (\frac{\boldsymbol{g}}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \quad \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

bias corrections to the estimates of both the first-order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin (see algorithm 8.7). RMSProp also incorporates an estimate of the (uncentered) second-order moment, however it lacks the correction factor. Thus, unlike in Adam, the RMSProp second-order moment estimate may have high bias early in training. Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs to be changed from the suggested default.

8.5.4 Choosing the Right Optimization Algorithm

In this section, we discussed a series of related algorithms that each seek to address the challenge of optimizing deep models by adapting the learning rate for each model parameter. At this point, a natural question is: which algorithm should one choose?

Unfortunately, there is currently no consensus on this point. Schaul *et al.* (2014) presented a valuable comparison of a large number of optimization algorithms across a wide range of learning tasks. While the results suggest that the family of algorithms with adaptive learning rates (represented by RMSProp and AdaDelta) performed fairly robustly, no single best algorithm has emerged.

Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam. The choice of which algorithm to use, at this point, seems to depend

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
Initialize accumulation variable $\boldsymbol{r} =$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute interim update: $\tilde{\boldsymbol{\theta}}$ $\boldsymbol{\theta} + \alpha\boldsymbol{v}$
Compute gradient: $\boldsymbol{g}$ $\frac{1}{m}$ $\tilde{\boldsymbol{\theta}}$ $\sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Accumulate gradient: $\boldsymbol{r}$ $\rho\boldsymbol{r} + (1$ $\rho)\boldsymbol{g}$ $\boldsymbol{g}$
Compute velocity update: $\boldsymbol{v}$ $\alpha\boldsymbol{v}$ $\frac{\epsilon}{\sqrt{\boldsymbol{r}}}$ $\boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta}$ $\boldsymbol{\theta} + \boldsymbol{v}$
end while

largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning).

8.6 Approximate Second-Order Methods

In this section we discuss the application of second-order methods to the training of deep networks. See LeCun *et al.* (1998a) for an earlier treatment of this subject. For simplicity of exposition, the only objective function we examine is the empirical risk:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{,\mathrm{y}\sim\hat{p}_{\mathrm{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}). \tag{8.25}$$

However the methods we discuss here extend readily to more general objective functions that, for instance, include parameter regularization terms such as those discussed in chapter 7.

8.6.1 Newton's Method

In section 4.3, we introduced second-order gradient methods. In contrast to first-order methods, second-order methods make use of second derivatives to improve optimization. The most widely used second-order method is Newton's method. We now describe Newton's method in more detail, with emphasis on its application to neural network training.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \quad , \boldsymbol{r} =$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \quad \frac{1}{m} \quad \boldsymbol{\theta} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \quad t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \quad \rho^1 \boldsymbol{s} + (1 \quad \rho^1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \quad \rho^2 \boldsymbol{r} + (1 \quad \rho^2)\boldsymbol{g} \quad \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \quad \frac{\boldsymbol{s}}{1-\rho_1}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \quad \frac{\boldsymbol{r}}{1-\rho_2}$
    Compute update: $\Delta\boldsymbol{\theta} = \quad \epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}}+\delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \quad \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

Newton's method is an optimization scheme based on using a second-order Taylor series expansion to approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$, ignoring derivatives of higher order:

$$J(\boldsymbol{\theta}) \quad J(\boldsymbol{\theta}^0) + (\boldsymbol{\theta} \quad \boldsymbol{\theta}^0)^\top \quad \boldsymbol{\theta} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} \quad \boldsymbol{\theta}^0)^\top \boldsymbol{H}(\boldsymbol{\theta} \quad \boldsymbol{\theta}^0), \tag{8.26}$$

where $\boldsymbol{H}$ is the Hessian of J with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_0$. If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 \quad \boldsymbol{H}^{-1} \quad \boldsymbol{\theta} J(\boldsymbol{\theta}_0) \tag{8.27}$$

Thus for a locally quadratic function (with positive definite $\boldsymbol{H}$), by rescaling the gradient by $\boldsymbol{H}^{-1}$, Newton's method jumps directly to the minimum. If the objective function is convex but not quadratic (there are higher-order terms), this update can be iterated, yielding the training algorithm associated with Newton's method, given in algorithm 8.8.

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)})$.

Require: Initial parameter $\boldsymbol{\theta}_0$
Require: Training set of m examples
while stopping criterion not met **do**
  Compute gradient: $\boldsymbol{g} \quad \frac{1}{m} \quad \boldsymbol{\theta} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Compute Hessian: $\boldsymbol{H} \quad \frac{1}{m} \quad \overset{2}{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Compute Hessian inverse: $\boldsymbol{H}^{-1}$
  Compute update: $\Delta\boldsymbol{\theta} = \quad \boldsymbol{H}^{-1}\boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

For surfaces that are not quadratic, as long as the Hessian remains positive definite, Newton's method can be applied iteratively. This implies a two-step iterative procedure. First, update or compute the inverse Hessian (i.e. by updating the quadratic approximation). Second, update the parameters according to equation 8.27.

In section 8.2.3, we discussed how Newton's method is appropriate only when the Hessian is positive definite. In deep learning, the surface of the objective function is typically non-convex with many features, such as saddle points, that are problematic for Newton's method. If the eigenvalues of the Hessian are not all positive, for example, near a saddle point, then Newton's method can actually cause updates to move in the wrong direction. This situation can be avoided by regularizing the Hessian. Common regularization strategies include adding a constant, α, along the diagonal of the Hessian. The regularized update becomes

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 \quad [H\,(f(\boldsymbol{\theta}\,0)) + \alpha\boldsymbol{I}]^{-1} \quad \boldsymbol{\theta} f(\boldsymbol{\theta}_0). \tag{8.28}$$

This regularization strategy is used in approximations to Newton's method, such as the Levenberg–Marquardt algorithm (Levenberg, 1944; Marquardt, 1963), and works fairly well as long as the negative eigenvalues of the Hessian are still relatively close to zero. In cases where there are more extreme directions of curvature, the value of α would have to be sufficiently large to offset the negative eigenvalues. However, as α increases in size, the Hessian becomes dominated by the $\alpha\boldsymbol{I}$ diagonal and the direction chosen by Newton's method converges to the standard gradient divided by α. When strong negative curvature is present, α may need to be so large that Newton's method would make smaller steps than gradient descent with a properly chosen learning rate.

Beyond the challenges created by certain features of the objective function,

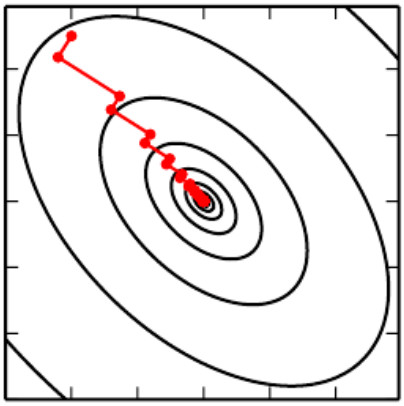
such as saddle points, the application of Newton's method for training large neural networks is limited by the significant computational burden it imposes. The number of elements in the Hessian is squared in the number of parameters, so with k parameters (and for even very small neural networks the number of parameters k can be in the millions), Newton's method would require the inversion of a $k \quad k$ matrix—with computational complexity of $O(k^3)$. Also, since the parameters will change with every update, the inverse Hessian has to be computed *at every training iteration*. As a consequence, only networks with a very small number of parameters can be practically trained via Newton's method. In the remainder of this section, we will discuss alternatives that attempt to gain some of the advantages of Newton's method while side-stepping the computational hurdles.

8.6.2 Conjugate Gradients

Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending **conjugate directions**. The inspiration for this approach follows from a careful study of the weakness of the method of steepest descent (see section 4.3 for details), where line searches are applied iteratively in the direction associated with the gradient. Figure 8.6 illustrates how the method of steepest descent, when applied in a quadratic bowl, progresses in a rather ineffective back-and-forth, zig-zag pattern. This happens because each line search direction, when given by the gradient, is guaranteed to be orthogonal to the previous line search direction.

Let the previous search direction be $\boldsymbol{d}_{t-1}$. At the minimum, where the line search terminates, the directional derivative is zero in direction $\boldsymbol{d}_{t-1}$: $\quad _{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ $\boldsymbol{d}_{t-1} = 0$. Since the gradient at this point defines the current search direction, $\boldsymbol{d}_t = \quad _{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ will have no contribution in the direction $\boldsymbol{d}_{t-1}$. Thus $\boldsymbol{d}_t$ is orthogonal to $\boldsymbol{d}_{t-1}$. This relationship between $\boldsymbol{d}_{t-1}$ and $\boldsymbol{d}_t$ is illustrated in figure 8.6 for multiple iterations of steepest descent. As demonstrated in the figure, the choice of orthogonal directions of descent do not preserve the minimum along the previous search directions. This gives rise to the zig-zag pattern of progress, where by descending to the minimum in the current gradient direction, we must re-minimize the objective in the previous gradient direction. Thus, by following the gradient at the end of each line search we are, in a sense, undoing progress we have already made in the direction of the previous line search. The method of conjugate gradients seeks to address this problem.

In the method of conjugate gradients, we seek to find a search direction that is **conjugate** to the previous line search direction, i.e. it will not undo progress made in that direction. At training iteration t, the next search direction $\boldsymbol{d}_t$ takes



the form:

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1} \tag{8.29}$$

where β_t is a coefficient whose magnitude controls how much of the direction, $\boldsymbol{d}_{t-1}$, we should add back to the current search direction.

Two directions, $\boldsymbol{d}_t$ and $\boldsymbol{d}_{t-1}$, are defined as conjugate if $\boldsymbol{d}_t^\top \boldsymbol{H} \boldsymbol{d}_{t-1} = 0$, where $\boldsymbol{H}$ is the Hessian matrix.

The straightforward way to impose conjugacy would involve calculation of the eigenvectors of $\boldsymbol{H}$ to choose β_t, which would not satisfy our goal of developing a method that is more computationally viable than Newton's method for large problems. Can we calculate the conjugate directions without resorting to these calculations? Fortunately the answer to that is yes.

Two popular methods for computing the β_t are:

1. Fletcher-Reeves:

$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.30}$$

2. Polak-Ribière:

$$\beta_t = \frac{(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1}))^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.31}$$

For a quadratic surface, the conjugate directions ensure that the gradient along the previous direction does not increase in magnitude. We therefore stay at the minimum along the previous directions. As a consequence, in a k-dimensional parameter space, the conjugate gradient method requires at most k line searches to achieve the minimum. The conjugate gradient algorithm is given in algorithm 8.9.

Algorithm 8.9 The conjugate gradient method

Require: Initial parameters $\boldsymbol{\theta}_0$
Require: Training set of m examples
  Initialize $\boldsymbol{\rho}_0 =$
  Initialize $g_0 = 0$
  Initialize $t = 1$
  while stopping criterion not met **do**
    Initialize the gradient $\boldsymbol{g}_t =$
    Compute gradient: $\boldsymbol{g}_t \quad \frac{1}{m} \quad \boldsymbol{\theta} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Compute $\beta_t = \frac{(\boldsymbol{g} - \boldsymbol{g}_{\ 1}) \ \boldsymbol{g}}{\boldsymbol{g}_{\ 1} \boldsymbol{g}_{\ 1}}$ (Polak-Ribière)
    (Nonlinear conjugate gradient: optionally reset β_t to zero, for example if t is a multiple of some constant k, such as $k = 5$)
    Compute search direction: $\boldsymbol{\rho}_t = \boldsymbol{g}^{\ t} + \beta_t \boldsymbol{\rho}_{t-1}$
    Perform line search to find: $\epsilon^* = \text{argmin}_\epsilon \frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}_t + \epsilon \boldsymbol{\rho}_t), \boldsymbol{y}^{(i)})$
    (On a truly quadratic cost function, analytically solve for ϵ^* rather than explicitly searching for it)
    Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t$

    $t \quad t + 1$
  end while

Nonlinear Conjugate Gradients: So far we have discussed the method of conjugate gradients as it is applied to quadratic objective functions. Of course, our primary interest in this chapter is to explore optimization methods for training neural networks and other related deep learning models where the corresponding objective function is far from quadratic. Perhaps surprisingly, the method of conjugate gradients is still applicable in this setting, though with some modification. Without any assurance that the objective is quadratic, the conjugate directions

are no longer assured to remain at the minimum of the objective for previous directions. As a result, the **nonlinear conjugate gradients** algorithm includes occasional resets where the method of conjugate gradients is restarted with line search along the unaltered gradient.

Practitioners report reasonable results in applications of the nonlinear conjugate gradients algorithm to training neural networks, though it is often beneficial to initialize the optimization with a few iterations of stochastic gradient descent before commencing nonlinear conjugate gradients. Also, while the (nonlinear) conjugate gradients algorithm has traditionally been cast as a batch method, minibatch versions have been used successfully for the training of neural networks (Le *et al.*, 2011). Adaptations of conjugate gradients specifically for neural networks have been proposed earlier, such as the scaled conjugate gradients algorithm (Moller, 1993).

8.6.3 BFGS

The **Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm** attempts to bring some of the advantages of Newton's method without the computational burden. In that respect, BFGS is similar to the conjugate gradient method. However, BFGS takes a more direct approach to the approximation of Newton's update. Recall that Newton's update is given by

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 \quad \boldsymbol{H}^{-1} \quad {}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \tag{8.32}$$

where $\boldsymbol{H}$ is the Hessian of J with respect to $\boldsymbol{\theta}$ evaluated at $\boldsymbol{\theta}_0$. The primary computational difficulty in applying Newton's update is the calculation of the inverse Hessian $\boldsymbol{H}^{-1}$. The approach adopted by quasi-Newton methods (of which the BFGS algorithm is the most prominent) is to approximate the inverse with a matrix $\boldsymbol{M}_t$ that is iteratively refined by low rank updates to become a better approximation of $\boldsymbol{H}^{-1}$.

The specification and derivation of the BFGS approximation is given in many textbooks on optimization, including Luenberger (1984).

Once the inverse Hessian approximation $\boldsymbol{M}_t$ is updated, the direction of descent $\boldsymbol{\rho}_t$ is determined by $\boldsymbol{\rho}_t = \boldsymbol{M}_t \boldsymbol{g}_t$. A line search is performed in this direction to determine the size of the step, ϵ^*, taken in this direction. The final update to the parameters is given by:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \epsilon^* \boldsymbol{\rho}_t \,. \tag{8.33}$$

Like the method of conjugate gradients, the BFGS algorithm iterates a series of line searches with the direction incorporating second-order information. However

unlike conjugate gradients, the success of the approach is not heavily dependent on the line search finding a point very close to the true minimum along the line. Thus, relative to conjugate gradients, BFGS has the advantage that it can spend less time refining each line search. On the other hand, the BFGS algorithm must store the inverse Hessian matrix, $\boldsymbol{M}$, that requires $O(n^2)$ memory, making BFGS impractical for most modern deep learning models that typically have millions of parameters.

Limited Memory BFGS (or L-BFGS) The memory costs of the BFGS algorithm can be significantly decreased by avoiding storing the complete inverse Hessian approximation $\boldsymbol{M}$. The L-BFGS algorithm computes the approximation $\boldsymbol{M}$ using the same method as the BFGS algorithm, but beginning with the assumption that $\boldsymbol{M}^{(t-1)}$ is the identity matrix, rather than storing the approximation from one step to the next. If used with exact line searches, the directions defined by L-BFGS are mutually conjugate. However, unlike the method of conjugate gradients, this procedure remains well behaved when the minimum of the line search is reached only approximately. The L-BFGS strategy with no storage described here can be generalized to include more information about the Hessian by storing some of the vectors used to update $\boldsymbol{M}$ at each time step, which costs only $O(n)$ per step.

8.7 Optimization Strategies and Meta-Algorithms

Many optimization techniques are not exactly algorithms, but rather general templates that can be specialized to yield algorithms, or subroutines that can be incorporated into many different algorithms.

8.7.1 Batch Normalization

Batch normalization (Ioffe and Szegedy, 2015) is one of the most exciting recent innovations in optimizing deep neural networks and it is actually not an optimization algorithm at all. Instead, it is a method of adaptive reparametrization, motivated by the difficulty of training very deep models.

Very deep models involve the composition of several functions or layers. The gradient tells how to update each parameter, under the assumption that the other layers do not change. In practice, we update all of the layers simultaneously. When we make the update, unexpected results can happen because many functions composed together are changed simultaneously, using updates that were computed under the assumption that the other functions remain constant. As a simple

example, suppose we have a deep neural network that has only one unit per layer and does not use an activation function at each hidden layer: $\hat{y} = xw_1 w_2 w_3 \dots w_l$. Here, w_i provides the weight used by layer i. The output of layer i is $h_i = h_{i-1} w_i$. The output $\hat{y}$ is a linear function of the input x, but a nonlinear function of the weights w_i. Suppose our cost function has put a gradient of 1 on $\hat{y}$, so we wish to decrease $\hat{y}$ slightly. The back-propagation algorithm can then compute a gradient $\boldsymbol{g} = \nabla_{\boldsymbol{w}} \hat{y}$. Consider what happens when we make an update $\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \boldsymbol{g}$. The first-order Taylor series approximation of $\hat{y}$ predicts that the value of $\hat{y}$ will decrease by $\epsilon \boldsymbol{g}^\top \boldsymbol{g}$. If we wanted to decrease $\hat{y}$ by .1, this first-order information available in the gradient suggests we could set the learning rate ϵ to $\frac{.1}{\boldsymbol{g}^\top \boldsymbol{g}}$. However, the actual update will include second-order and third-order effects, on up to effects of order l. The new value of $\hat{y}$ is given by

$$x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l). \tag{8.34}$$

An example of one second-order term arising from this update is $\epsilon^2 g_1 g_2 \prod_{i=3}^{l} w_i$. This term might be negligible if $\prod_{i=3}^{l} w_i$ is small, or might be exponentially large if the weights on layers 3 through l are greater than 1. This makes it very hard to choose an appropriate learning rate, because the effects of an update to the parameters for one layer depends so strongly on all of the other layers. Second-order optimization algorithms address this issue by computing an update that takes these second-order interactions into account, but we can see that in very deep networks, even higher-order interactions can be significant. Even second-order optimization algorithms are expensive and usually require numerous approximations that prevent them from truly accounting for all significant second-order interactions. Building an n-th order optimization algorithm for $n > 2$ thus seems hopeless. What can we do instead?

Batch normalization provides an elegant way of reparametrizing almost any deep network. The reparametrization significantly reduces the problem of coordinating updates across many layers. Batch normalization can be applied to any input or hidden layer in a network. Let $\boldsymbol{H}$ be a minibatch of activations of the layer to normalize, arranged as a design matrix, with the activations for each example appearing in a row of the matrix. To normalize $\boldsymbol{H}$, we replace it with

$$\boldsymbol{H}' = \frac{\boldsymbol{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \tag{8.35}$$

where $\boldsymbol{\mu}$ is a vector containing the mean of each unit and $\boldsymbol{\sigma}$ is a vector containing the standard deviation of each unit. The arithmetic here is based on broadcasting the vector $\boldsymbol{\mu}$ and the vector $\boldsymbol{\sigma}$ to be applied to every row of the matrix $\boldsymbol{H}$. Within each row, the arithmetic is element-wise, so $H_{i,j}$ is normalized by subtracting μ_j

and dividing by σ_j. The rest of the network then operates on $\boldsymbol{H}'$ in exactly the same way that the original network operated on $\boldsymbol{H}$.

At training time,

$$\boldsymbol{\mu} = \frac{1}{m} \sum_i \boldsymbol{H}_{i,:} \tag{8.36}$$

and

$$\boldsymbol{\sigma} = \sqrt{\delta + \frac{1}{m} \sum_i (\boldsymbol{H} \quad \boldsymbol{\mu})_i^2}, \tag{8.37}$$

where δ is a small positive value such as 10^{-8} imposed to avoid encountering the undefined gradient of $\overline{z}$ at $z = 0$. Crucially, *we back-propagate through these operations* for computing the mean and the standard deviation, and for applying them to normalize $\boldsymbol{H}$. This means that the gradient will never propose an operation that acts simply to increase the standard deviation or mean of h_i; the normalization operations remove the effect of such an action and zero out its component in the gradient. This was a major innovation of the batch normalization approach. Previous approaches had involved adding penalties to the cost function to encourage units to have normalized activation statistics or involved intervening to renormalize unit statistics after each gradient descent step. The former approach usually resulted in imperfect normalization and the latter usually resulted in significant wasted time as the learning algorithm repeatedly proposed changing the mean and variance and the normalization step repeatedly undid this change. Batch normalization reparametrizes the model to make some units always be standardized by definition, deftly sidestepping both problems.

At test time, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ may be replaced by running averages that were collected during training time. This allows the model to be evaluated on a single example, without needing to use definitions of $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ that depend on an entire minibatch.

Revisiting the $\hat{y} = xw_1w_2 \dots w_l$ example, we see that we can mostly resolve the difficulties in learning this model by normalizing h_{l-1}. Suppose that x is drawn from a unit Gaussian. Then h_{l-1} will also come from a Gaussian, because the transformation from x to h_l is linear. However, h_{l-1} will no longer have zero mean and unit variance. After applying batch normalization, we obtain the normalized $\hat{h}_{l-1}$ that restores the zero mean and unit variance properties. For almost any update to the lower layers, $\hat{h}_{l-1}$ will remain a unit Gaussian. The output $\hat{y}$ may then be learned as a simple linear function $\hat{y} = w_l\hat{h}_{l-1}$. Learning in this model is now very simple because the parameters at the lower layers simply do not have an effect in most cases; their output is always renormalized to a unit Gaussian. In some corner cases, the lower layers can have an effect. Changing one of the lower layer weights to 0 can make the output become degenerate, and changing the sign

of one of the lower weights can flip the relationship between $\hat{h}_{l-1}$ and y. These situations are very rare. Without normalization, nearly every update would have an extreme effect on the statistics of h_{l-1}. Batch normalization has thus made this model significantly easier to learn. In this example, the ease of learning of course came at the cost of making the lower layers useless. In our linear example, the lower layers no longer have any harmful effect, but they also no longer have any beneficial effect. This is because we have normalized out the first and second order statistics, which is all that a linear network can influence. In a deep neural network with nonlinear activation functions, the lower layers can perform nonlinear transformations of the data, so they remain useful. Batch normalization acts to standardize only the mean and variance of each unit in order to stabilize learning, but allows the relationships between units and the nonlinear statistics of a single unit to change.

Because the final layer of the network is able to learn a linear transformation, we may actually wish to remove all linear relationships between units within a layer. Indeed, this is the approach taken by Desjardins *et al.* (2015), who provided the inspiration for batch normalization. Unfortunately, eliminating all linear interactions is much more expensive than standardizing the mean and standard deviation of each individual unit, and so far batch normalization remains the most practical approach.

Normalizing the mean and standard deviation of a unit can reduce the expressive power of the neural network containing that unit. In order to maintain the expressive power of the network, it is common to replace the batch of hidden unit activations $\boldsymbol{H}$ with $\boldsymbol{\gamma}\boldsymbol{H}' + \boldsymbol{\beta}$ rather than simply the normalized $\boldsymbol{H}'$. The variables $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are learned parameters that allow the new variable to have any mean and standard deviation. At first glance, this may seem useless—why did we set the mean to , and then introduce a parameter that allows it to be set back to any arbitrary value $\boldsymbol{\beta}$? The answer is that the new parametrization can represent the same family of functions of the input as the old parametrization, but the new parametrization has different learning dynamics. In the old parametrization, the mean of $\boldsymbol{H}$ was determined by a complicated interaction between the parameters in the layers below $\boldsymbol{H}$. In the new parametrization, the mean of $\boldsymbol{\gamma}\boldsymbol{H}' + \boldsymbol{\beta}$ is determined solely by $\boldsymbol{\beta}$. The new parametrization is much easier to learn with gradient descent.

Most neural network layers take the form of $\phi(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b})$ where ϕ is some fixed nonlinear activation function such as the rectified linear transformation. It is natural to wonder whether we should apply batch normalization to the input $\boldsymbol{X}$, or to the transformed value $\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}$. Ioffe and Szegedy (2015) recommend

the latter. More specifically, $\boldsymbol{XW} + \boldsymbol{b}$ should be replaced by a normalized version of $\boldsymbol{XW}$. The bias term should be omitted because it becomes redundant with the β parameter applied by the batch normalization reparametrization. The input to a layer is usually the output of a nonlinear activation function such as the rectified linear function in a previous layer. The statistics of the input are thus more non-Gaussian and less amenable to standardization by linear operations.

In convolutional networks, described in chapter 9, it is important to apply the same normalizing μ and σ at every spatial location within a feature map, so that the statistics of the feature map remain the same regardless of spatial location.

8.7.2 Coordinate Descent

In some cases, it may be possible to solve an optimization problem quickly by breaking it into separate pieces. If we minimize $f(\boldsymbol{x})$ with respect to a single variable x_i, then minimize it with respect to another variable x_j and so on, repeatedly cycling through all variables, we are guaranteed to arrive at a (local) minimum. This practice is known as **coordinate descent**, because we optimize one coordinate at a time. More generally, **block coordinate descent** refers to minimizing with respect to a subset of the variables simultaneously. The term "coordinate descent" is often used to refer to block coordinate descent as well as the strictly individual coordinate descent.

Coordinate descent makes the most sense when the different variables in the optimization problem can be clearly separated into groups that play relatively isolated roles, or when optimization with respect to one group of variables is significantly more efficient than optimization with respect to all of the variables. For example, consider the cost function

$$J(\boldsymbol{H}, \boldsymbol{W}) = \sum_{i,j} H^{i,j} + \sum_{i,j} \left(\boldsymbol{X} \quad \boldsymbol{W}^{\top} \boldsymbol{H} \right)^2_{i,j}. \tag{8.38}$$

This function describes a learning problem called sparse coding, where the goal is to find a weight matrix $\boldsymbol{W}$ that can linearly decode a matrix of activation values $\boldsymbol{H}$ to reconstruct the training set $\boldsymbol{X}$. Most applications of sparse coding also involve weight decay or a constraint on the norms of the columns of $\boldsymbol{W}$, in order to prevent the pathological solution with extremely small $\boldsymbol{H}$ and large $\boldsymbol{W}$.

The function J is not convex. However, we can divide the inputs to the training algorithm into two sets: the dictionary parameters $\boldsymbol{W}$ and the code representations $\boldsymbol{H}$. Minimizing the objective function with respect to either one of these sets of variables is a convex problem. Block coordinate descent thus gives

us an optimization strategy that allows us to use efficient convex optimization algorithms, by alternating between optimizing $\boldsymbol{W}$ with $\boldsymbol{H}$ fixed, then optimizing $\boldsymbol{H}$ with $\boldsymbol{W}$ fixed.

Coordinate descent is not a very good strategy when the value of one variable strongly influences the optimal value of another variable, as in the function $f(\boldsymbol{x}) = (x_1 \quad x_2)^2 + \alpha \left(x_1^2 + x_2^2\right)$ where α is a positive constant. The first term encourages the two variables to have similar value, while the second term encourages them to be near zero. The solution is to set both to zero. Newton's method can solve the problem in a single step because it is a positive definite quadratic problem. However, for small α, coordinate descent will make very slow progress because the first term does not allow a single variable to be changed to a value that differs significantly from the current value of the other variable.

8.7.3 Polyak Averaging

Polyak averaging (Polyak and Juditsky, 1992) consists of averaging together several points in the trajectory through parameter space visited by an optimization algorithm. If t iterations of gradient descent visit points $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(t)}$, then the output of the Polyak averaging algorithm is $\hat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t} \sum_i \boldsymbol{\theta}^{(i)}$. On some problem classes, such as gradient descent applied to convex problems, this approach has strong convergence guarantees. When applied to neural networks, its justification is more heuristic, but it performs well in practice. The basic idea is that the optimization algorithm may leap back and forth across a valley several times without ever visiting a point near the bottom of the valley. The average of all of the locations on either side should be close to the bottom of the valley though.

In non-convex problems, the path taken by the optimization trajectory can be very complicated and visit many different regions. Including points in parameter space from the distant past that may be separated from the current point by large barriers in the cost function does not seem like a useful behavior. As a result, when applying Polyak averaging to non-convex problems, it is typical to use an exponentially decaying running average:

$$\hat{\boldsymbol{\theta}}^{(t)} = \alpha \hat{\boldsymbol{\theta}}^{(t-1)} + (1 \quad \alpha) \boldsymbol{\theta}^{(t)}. \tag{8.39}$$

The running average approach is used in numerous applications. See Szegedy *et al.* (2015) for a recent example.

8.7.4 Supervised Pretraining

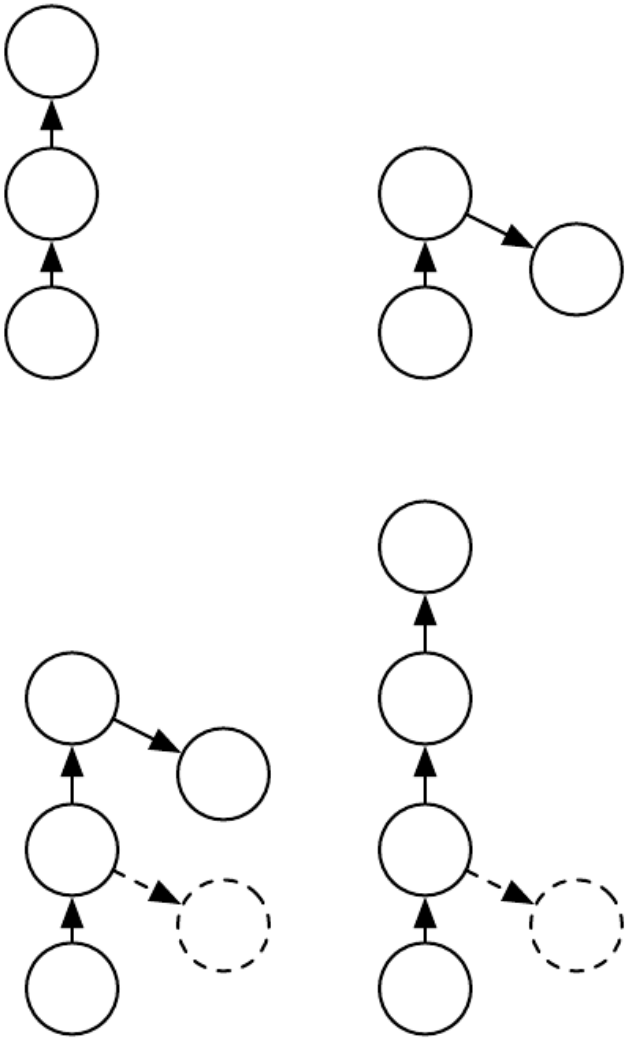
Sometimes, directly training a model to solve a specific task can be too ambitious if the model is complex and hard to optimize or if the task is very difficult. It is sometimes more effective to train a simpler model to solve the task, then make the model more complex. It can also be more effective to train the model to solve a simpler task, then move on to confront the final task. These strategies that involve training simple models on simple tasks before confronting the challenge of training the desired model to perform the desired task are collectively known as **pretraining**.

Greedy algorithms break a problem into many components, then solve for the optimal version of each component in isolation. Unfortunately, combining the individually optimal components is not guaranteed to yield an optimal complete solution. However, greedy algorithms can be computationally much cheaper than algorithms that solve for the best joint solution, and the quality of a greedy solution is often acceptable if not optimal. Greedy algorithms may also be followed by a **fine-tuning** stage in which a joint optimization algorithm searches for an optimal solution to the full problem. Initializing the joint optimization algorithm with a greedy solution can greatly speed it up and improve the quality of the solution it finds.

Pretraining, and especially greedy pretraining, algorithms are ubiquitous in deep learning. In this section, we describe specifically those pretraining algorithms that break supervised learning problems into other simpler supervised learning problems. This approach is known as **greedy supervised pretraining**.

In the original (Bengio *et al.*, 2007) version of greedy supervised pretraining, each stage consists of a supervised learning training task involving only a subset of the layers in the final neural network. An example of greedy supervised pretraining is illustrated in figure 8.7, in which each added hidden layer is pretrained as part of a shallow supervised MLP, taking as input the output of the previously trained hidden layer. Instead of pretraining one layer at a time, Simonyan and Zisserman (2015) pretrain a deep convolutional network (eleven weight layers) and then use the first four and last three layers from this network to initialize even deeper networks (with up to nineteen layers of weights). The middle layers of the new, very deep network are initialized randomly. The new network is then jointly trained. Another option, explored by Yu *et al.* (2010) is to use the *outputs* of the previously trained MLPs, as well as the raw input, as inputs for each added stage.

Why would greedy supervised pretraining help? The hypothesis initially discussed by Bengio *et al.* (2007) is that it helps to provide better guidance to the



intermediate levels of a deep hierarchy. In general, pretraining may help both in terms of optimization and in terms of generalization.

An approach related to supervised pretraining extends the idea to the context of transfer learning: Yosinski *et al.* (2014) pretrain a deep convolutional net with 8 layers of weights on a set of tasks (a subset of the 1000 ImageNet object categories) and then initialize a same-size network with the first k layers of the first net. All the layers of the second network (with the upper layers initialized randomly) are then jointly trained to perform a different set of tasks (another subset of the 1000 ImageNet object categories), with fewer training examples than for the first set of tasks. Other approaches to transfer learning with neural networks are discussed in section 15.2.

Another related line of work is the **FitNets** (Romero *et al.*, 2015) approach. This approach begins by training a network that has low enough depth and great enough width (number of units per layer) to be easy to train. This network then becomes a **teacher** for a second network, designated the **student**. The student network is much deeper and thinner (eleven to nineteen layers) and would be difficult to train with SGD under normal circumstances. The training of the student network is made easier by training the student network not only to predict the output for the original task, but also to predict the value of the middle layer of the teacher network. This extra task provides a set of hints about how the hidden layers should be used and can simplify the optimization problem. Additional parameters are introduced to regress the middle layer of the 5-layer teacher network from the middle layer of the deeper student network. However, instead of predicting the final classification target, the objective is to predict the middle hidden layer of the teacher network. The lower layers of the student networks thus have two objectives: to help the outputs of the student network accomplish their task, as well as to predict the intermediate layer of the teacher network. Although a thin and deep network appears to be more difficult to train than a wide and shallow network, the thin and deep network may generalize better and certainly has lower computational cost if it is thin enough to have far fewer parameters. Without the hints on the hidden layer, the student network performs very poorly in the experiments, both on the training and test set. Hints on middle layers may thus be one of the tools to help train neural networks that otherwise seem difficult to train, but other optimization techniques or changes in the architecture may also solve the problem.

8.7.5 Designing Models to Aid Optimization

To improve optimization, the best strategy is not always to improve the optimization algorithm. Instead, many improvements in the optimization of deep models have come from designing the models to be easier to optimize.

In principle, we could use activation functions that increase and decrease in jagged non-monotonic patterns. However, this would make optimization extremely difficult. In practice, *it is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm.* Most of the advances in neural network learning over the past 30 years have been obtained by changing the model family rather than changing the optimization procedure. Stochastic gradient descent with momentum, which was used to train neural networks in the 1980s, remains in use in modern state of the art neural network applications.

Specifically, modern neural networks reflect a *design choice* to use linear transformations between layers and activation functions that are differentiable almost everywhere and have significant slope in large portions of their domain. In particular, model innovations like the LSTM, rectified linear units and maxout units have all moved toward using more linear functions than previous models like deep networks based on sigmoidal units. These models have nice properties that make optimization easier. The gradient flows through many layers provided that the Jacobian of the linear transformation has reasonable singular values. Moreover, linear functions consistently increase in a single direction, so even if the model's output is very far from correct, it is clear simply from computing the gradient which direction its output should move to reduce the loss function. In other words, modern neural nets have been designed so that their *local* gradient information corresponds reasonably well to moving toward a distant solution.

Other model design strategies can help to make optimization easier. For example, linear paths or skip connections between layers reduce the length of the shortest path from the lower layer's parameters to the output, and thus mitigate the vanishing gradient problem (Srivastava *et al.*, 2015). A related idea to skip connections is adding extra copies of the output that are attached to the intermediate hidden layers of the network, as in GoogLeNet (Szegedy *et al.*, 2014a) and deeply-supervised nets (Lee *et al.*, 2014). These "auxiliary heads" are trained to perform the same task as the primary output at the top of the network in order to ensure that the lower layers receive a large gradient. When training is complete the auxiliary heads may be discarded. This is an alternative to the pretraining strategies, which were introduced in the previous section. In this way, one can train jointly all the layers in a single phase but change the architecture, so that intermediate layers (especially the lower ones) can get some hints about what they

should do, via a shorter path. These hints provide an error signal to lower layers.

8.7.6 Continuation Methods and Curriculum Learning

As argued in section 8.2.7, many of the challenges in optimization arise from the global structure of the cost function and cannot be resolved merely by making better estimates of local update directions. The predominant strategy for overcoming this problem is to attempt to initialize the parameters in a region that is connected to the solution by a short path through parameter space that local descent can discover.

Continuation methods are a family of strategies that can make optimization easier by choosing initial points to ensure that local optimization spends most of its time in well-behaved regions of space. The idea behind continuation methods is to construct a series of objective functions over the same parameters. In order to minimize a cost function $J(\boldsymbol{\theta})$, we will construct new cost functions $J^{(0)}, \dots, J^{(n)}$. These cost functions are designed to be increasingly difficult, with $J^{(0)}$ being fairly easy to minimize, and $J^{(n)}$, the most difficult, being $J(\boldsymbol{\theta})$, the true cost function motivating the entire process. When we say that $J^{(i)}$ is easier than $J^{(i+1)}$, we mean that it is well behaved over more of $\boldsymbol{\theta}$ space. A random initialization is more likely to land in the region where local descent can minimize the cost function successfully because this region is larger. The series of cost functions are designed so that a solution to one is a good initial point of the next. We thus begin by solving an easy problem then refine the solution to solve incrementally harder problems until we arrive at a solution to the true underlying problem.

Traditional continuation methods (predating the use of continuation methods for neural network training) are usually based on smoothing the objective function. See Wu (1997) for an example of such a method and a review of some related methods. Continuation methods are also closely related to simulated annealing, which adds noise to the parameters (Kirkpatrick *et al.*, 1983). Continuation methods have been extremely successful in recent years. See Mobahi and Fisher (2015) for an overview of recent literature, especially for AI applications.

Continuation methods traditionally were mostly designed with the goal of overcoming the challenge of local minima. Specifically, they were designed to reach a global minimum despite the presence of many local minima. To do so, these continuation methods would construct easier cost functions by "blurring" the original cost function. This blurring operation can be done by approximating

$$J^{(i)}(\boldsymbol{\theta}) = \mathbb{E}_{\theta \sim \mathcal{N}(\boldsymbol{\theta}; \boldsymbol{\theta}, \sigma^2)} J(\boldsymbol{\theta}') \tag{8.40}$$

via sampling. The intuition for this approach is that some non-convex functions

become approximately convex when blurred. In many cases, this blurring preserves enough information about the location of a global minimum that we can find the global minimum by solving progressively less blurred versions of the problem. This approach can break down in three different ways. First, it might successfully define a series of cost functions where the first is convex and the optimum tracks from one function to the next arriving at the global minimum, but it might require so many incremental cost functions that the cost of the entire procedure remains high. NP-hard optimization problems remain NP-hard, even when continuation methods are applicable. The other two ways that continuation methods fail both correspond to the method not being applicable. First, the function might not become convex, no matter how much it is blurred. Consider for example the function $J(\boldsymbol{\theta}) = -\boldsymbol{\theta}^\top \boldsymbol{\theta}$. Second, the function may become convex as a result of blurring, but the minimum of this blurred function may track to a local rather than a global minimum of the original cost function.

Though continuation methods were mostly originally designed to deal with the problem of local minima, local minima are no longer believed to be the primary problem for neural network optimization. Fortunately, continuation methods can still help. The easier objective functions introduced by the continuation method can eliminate flat regions, decrease variance in gradient estimates, improve conditioning of the Hessian matrix, or do anything else that will either make local updates easier to compute or improve the correspondence between local update directions and progress toward a global solution.

Bengio *et al.* (2009) observed that an approach called **curriculum learning** or **shaping** can be interpreted as a continuation method. Curriculum learning is based on the idea of planning a learning process to begin by learning simple concepts and progress to learning more complex concepts that depend on these simpler concepts. This basic strategy was previously known to accelerate progress in animal training (Skinner, 1958; Peterson, 2004; Krueger and Dayan, 2009) and machine learning (Solomonoff, 1989; Elman, 1993; Sanger, 1994). Bengio *et al.* (2009) justified this strategy as a continuation method, where earlier $J^{(i)}$ are made easier by increasing the influence of simpler examples (either by assigning their contributions to the cost function larger coefficients, or by sampling them more frequently), and experimentally demonstrated that better results could be obtained by following a curriculum on a large-scale neural language modeling task. Curriculum learning has been successful on a wide range of natural language (Spitkovsky *et al.*, 2010; Collobert *et al.*, 2011a; Mikolov *et al.*, 2011b; Tu and Honavar, 2011) and computer vision (Kumar *et al.*, 2010; Lee and Grauman, 2011; Supancic and Ramanan, 2013) tasks. Curriculum learning was also verified as being consistent with the way in which humans *teach* (Khan *et al.*, 2011): teachers start by showing easier and

more prototypical examples and then help the learner refine the decision surface with the less obvious cases. Curriculum-based strategies are *more effective* for teaching humans than strategies based on uniform sampling of examples, and can also increase the effectiveness of other teaching strategies (Basu and Christensen, 2013).

Another important contribution to research on curriculum learning arose in the context of training recurrent neural networks to capture long-term dependencies: Zaremba and Sutskever (2014) found that much better results were obtained with a *stochastic curriculum*, in which a random mix of easy and difficult examples is always presented to the learner, but where the average proportion of the more difficult examples (here, those with longer-term dependencies) is gradually increased. With a deterministic curriculum, no improvement over the baseline (ordinary training from the full training set) was observed.

We have now described the basic family of neural network models and how to regularize and optimize them. In the chapters ahead, we turn to specializations of the neural network family, that allow neural networks to scale to very large sizes and process input data that has special structure. The optimization methods discussed in this chapter are often directly applicable to these specialized architectures with little or no modification.