Chapter 8

Optimization for Training Deep Models

Deep learning algorithms involve optimization in many contexts. For example, we often solve optimization problems analytically in order to prove that an algorithm has a certain property. Inference in a probabilistic model can be cast as an optimization problem. 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 on 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're 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: minimizing an objective function $J(\boldsymbol{\theta})$ with respect to the model parameters $\boldsymbol{\theta}$, which is also implicitly a function of the training data. Typically, that objective function can be rewritten as an average over the training set, such as

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

where L is the per-example loss function, $f(\boldsymbol{x};\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$, y is the target output and $\hat{p}_{\text{data}}$ is the empirical distribution, in the supervised learning case. However, 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:

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

8.1 Optimization for Model Training

Optimization algorithms used for training of deep models differ from traditional optimization algorithms in several ways. Machine learning usually acts indirectly—we care about some performance measure P that we do not know how to directly influence, so instead we reduce some objective function $J(\boldsymbol{\theta})$ in hope that it 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.

8.1.1 Empirical Risk Minimization

Suppose that we have input feature vector $\boldsymbol{x}$ and targets y, sampled from some unknown joint distribution $p(\boldsymbol{x}, y)$, as well as some loss function $L(\boldsymbol{x}, y)$. Our ultimate goal is to minimize $\mathbb{E}_{\boldsymbol{x},y\sim p(\boldsymbol{x},y)}[L(\boldsymbol{x}, y)]$. This quantity is known as the *risk*. We emphasize here that the expectation is taken over the true underlying distribution p, so the risk is a form of generalization error. If we knew the true distribution $p(\boldsymbol{x}, y)$, this would be an optimization task solvable by an optimization algorithm. However, when we do not know $p(\boldsymbol{x}, y)$ but only have a training set of samples from it, 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*

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

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

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). Practically, note that we cannot use gradient-based optimization on the 0-1 loss because its derivative is 0 almost everywhere. 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 with neural networks. It allows the model to estimate the conditional probability of the classes, given the input, and if we can do that well, then we can pick the classes that yield the less classification error in expectation. In other cases, the actual loss function is one that is very expensive to obtain. Note that 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 0, 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.

As discussed in Section 7.7, although one uses a surrogate loss function for the optimization objective, one can still use the actual loss of interest (if it is computable) to monitor progress and perform early stopping, a form of regularization. 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, using early stopping, they halt whenever overfitting begins to occur. This is often in the middle of a wide, flat region, but it can also occur on a steep part of the surrogate loss function. This is in contrast to general optimization, where converge is usually defined by arriving at a point that is very near a (local) minimum.

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 a subset of the terms of the objective function, not based on the complete objective function itself.

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

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

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

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{8.3}$$

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:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x}\sim\hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{8.4}$$

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 estimated from n samples is given by $\hat{\sigma}/\sqrt{n}$, where $\hat{\sigma}$ is the estimated standard deviation. This means 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 way to intuitively understand the appeal of statistically estimating the gradient from a small number of samples is to consider that there may be 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 1 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 functionally 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. 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 will be 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 Sec. 8.3.2.

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 GPU, 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. Generalization error is often best for a batch size of 1, though this might take a very long time to train and require a small learning rate to maintain stability.

Different kinds of algorithms use different kinds of information 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, that 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. To see this, consider that when $\boldsymbol{H}$ and its inverse are poorly conditioned, then very small changes in the estimate of

$\boldsymbol{g}$ can cause large changes in the update $\boldsymbol{H}^{-1}\boldsymbol{g}$. This is further compounded by estimation error in $\boldsymbol{H}$ itself.

Computing the expected gradient from a set of samples requires that those samples be independent. 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 on 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 re-use 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.

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. This is discussed further in Chapter 12.1.3.

8.1.4 Online Gradient Descent Minimizes Generalization Error

In machine learning, typically we minimize a objective function defined as an expectation of some per-example loss across the training set, such as in Eq. 8.1, but we really care about minimizing the corresponding generalization error, as in Eq. 8.2.

Usually, we use an optimization algorithm based on minibatch estimates of the gradient. As we argue below, during the first stages of learning, this is equivalent to minimizing the generalization error directly. However, after we have pass through the training data once and begin to repeat minibatches, the two criteria diverge.

Let us consider the "online" learning case, where examples or minibatches are drawn from a *stream* of data. In other words, instead of a fixed-size training set, we are in the situation 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 $p(\boldsymbol{x}, y)$ (the same argument could be made in the unsupervised case, where there is no y).

Consider a loss function $L(f(\boldsymbol{x}; \boldsymbol{\theta}), y)$ whose expected value over $p(\boldsymbol{x}, y)$ we would like to minimize with respect to the parameters $\boldsymbol{\theta}$. In other words, the generalization error (Eq. 8.2) of the current predictor $f(\cdot; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$ can be rewritten as

$$J^*(\boldsymbol{\theta}) = \int L(f(\boldsymbol{x}; \boldsymbol{\theta}), y) dp(\boldsymbol{x}, y)$$

and under continuity assumptions of p, its exact gradient is

$$\boldsymbol{g} = \frac{\partial J^*(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \int \frac{\partial L(f(\boldsymbol{x}; \boldsymbol{\theta}), y)}{\partial \boldsymbol{\theta}} dp(\boldsymbol{x}, y),$$

similarly to what we have seen in Eqs. 8.3 and 8.4 for the log-likelihood. Hence, we can obtain an unbiased estimator of the exact gradient of generalization error by sampling one example $(\boldsymbol{x}, y)$ (or equivalently a minibatch) from the data generating process p, and computing the gradient of the loss with respect to the parameters for that example (or that minibatch),

$$\hat{\boldsymbol{g}} = \frac{\partial L(f(\boldsymbol{x}; \boldsymbol{\theta}), y)}{\partial \boldsymbol{\theta}}.$$

It should be clear that this stochastic gradient estimator $\hat{\boldsymbol{g}}$ is a noisy but unbiased estimator of the *exact gradient of generalization error*, $\boldsymbol{g}$. Hence, updating $\boldsymbol{\theta}$ in the direction of $\hat{\boldsymbol{g}}$ performs SGD on the generalization error. Of course, this is only possible if the examples are not repeated (unlike in the usual scenario for many machine learning applications, where several epochs through the training set are performed). With some datasets growing rapidly in size, faster than computing power, this online scenario is actually quite plausible. In that setting, overfitting is not an issue, while underfitting and computational efficiency matter a lot. 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 this difficulty 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, which introduces many difficulties. In this section, we summarize several of the most prominent challenges.

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. Suppose we update our parameters using a gradient descent step $\boldsymbol{\theta}' = \boldsymbol{\theta} - \alpha \boldsymbol{g}$ where α is a learning rate and $\boldsymbol{g} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. A second-order Taylor series expansion predicts that the value of the cost function at the new point is given by

$$J(\boldsymbol{\theta}') \approx J(\boldsymbol{\theta}) - \alpha \boldsymbol{g}^\top \boldsymbol{g} + \frac{1}{2}\alpha^2 \boldsymbol{g}\boldsymbol{H}\boldsymbol{g}$$

where $\boldsymbol{H}$ is the Hessian of J with respect to $\boldsymbol{\theta}$. The $-\alpha \boldsymbol{g}^\top \boldsymbol{g}$ term is always negative—if the cost function were a linear function of the parameters, gradient descent would always move downhill. However, the second-order term $\frac{1}{2}\alpha^2 \boldsymbol{g}\boldsymbol{H}\boldsymbol{g}$ can be negative or positive depending on the eigenvalues of $\boldsymbol{H}$ and the alignment of the corresponding eigenvectors with $\boldsymbol{g}$. On steps where $\boldsymbol{g}$ aligns closely with large, positive eigenvalues of $\boldsymbol{H}$, the learning rate must be very small, or the second-order term will result in gradient descent accidentally moving uphill.

This is a very general problem in most numerical optimization, convex or otherwise, and is described in more detail in Sec. 4.2.

The ill-conditioning problem is generally believed to be present in neural networks. It can manifest by causing SGD to get "stuck" in the sense that even very small steps increase the cost function. We 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 $\boldsymbol{g}\boldsymbol{H}\boldsymbol{g}$ term grows by more than order of magnitude.

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 a strictly convex function contains only one local minimum, which is also the global minimum. A weakly convex function may contain multiple global minima, but they are conveniently grouped into a single convex set in which all solutions are equivalent. 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.

Neural networks and any models with multiple equivalently parameterized 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 income 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-identifability 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 $1/\alpha$. This means that—if the cost function does not include terms such of 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 $(m \times n)$-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. 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 a local minimum that also has very low value (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. To verify that a problem arises from local minima, plot the norm of the gradient over time. If the norm of the gradient does not shrink to insignificant size, the problem is not local minima.

8.2.3 Ill-Conditioning

In Sec. 4.2 we discussed the general notion of poor conditioning in numerical computation. We defined the condition number of a matrix as the ratio of its largest to its smallest eigenvalue (in magnitude). In optimization, ill-conditioning refers to the difficulty that arises when the objective function to be minimized has eigenvalues that are tiny or even 0. In that case, inverting the Hessian matrix or optimizing the objective function both become numerically difficult. The condition number of the Hessian is directly linked to the number of training iterations needed by gradient descent, for example.

8.2.4 Plateaus, Saddle Points and Other Flat Regions

In a convex problem, any point with zero gradient is a global minimum. In a non-convex problem, a point with zero gradient may be a global minimum. It may also be a local minimum, as discussed above. Traditionally, local minima received much attention and fear of them was one of the reasons for the "neural networks winter" of 1995-2005.

However, local minima and maxima may in fact be 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.

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 exponentially more common. To understand the intution behind this, observe that a local minimum has only positive eigenvalues. 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 1-dimensional space, 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 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.

This happens for many classes of random functions. Does it happen for neural networks? Baldi and Hornik (1989) showed theoretically that shallow autoencoders with no non-linearities have global minima and saddle points but no local minima with higher cost than the global minimum. Saxe *et al.* (2013) showed theoretically the same result for deep networks without non-linearities. Such networks are essentially just multiple matrices composed together. Their output is a linear function of their input, but they are useful to study as a model of non-linear neural networks because their loss function is a non-convex function of their parameters. Dauphin *et al.* (2014) showed experimentally that real neural networks also have this behavior. 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? This remains unclear. Some practitioners believe that saddle points could cause a serious problem. Currently, no evidence exists to suggest that first-order methods are hindered by saddle points. Goodfellow *et al.* (2015) provided visualizations of several learning trajectories of state-of-the-art neural networks and found no evidence of gradient descent slowing down near saddle points. 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.

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 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.

8.2.5 Cliffs and Exploding Gradients

Whereas the issues of ill-conditioning and saddle points discussed in the previous sections arise because of the second-order structure of the objective function (as a function of the parameters), neural networks involve stronger non-linearities which do not fit well with this picture. In particular, the second-order Taylor series approximation of the objective function yields a symmetric view of the landscape around the minimum, oriented according to the axes defined by the principal eigenvectors of the Hessian matrix. See Fig. 8.7 for an illustration of the Hessian matrix eigenvector directions around a local minimum. Second-order methods and momentum or gradient-averaging methods introduced in Sec. 8.5 are able to reduce the difficulty due to ill-conditioning by increasing the size of the steps in the low-curvature directions (the "valley", in Figure 8.1) and decreasing the size of the steps in the high-curvature directions (the steep sides of the valley, in the figure).

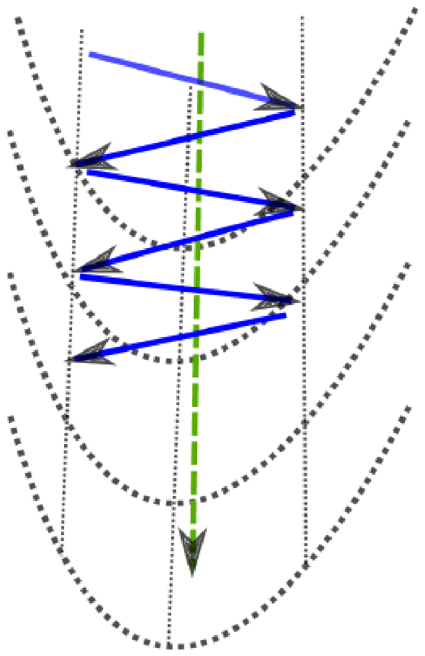

Figure 8.1: One theory about the neural network optimization is that poorly conditioned Hessian matrices cause much of the difficulty in training. In this view, some directions have a high curvature (second derivative), corresponding to the quickly rising sides of the valley (going left or right), and other directions have a low curvature, corresponding to the smooth slope of the valley (going down, dashed arrow). Most second-order methods, as well as momentum or gradient averaging methods are meant to address that problem, by increasing the step size in the direction of the valley (where it is most paying in the long run to go) and decreasing it in the directions of steep rise, which would otherwise lead to oscillations (blue full arrows). The objective is to smoothly go down, staying at the bottom of the valley (green dashed arrow).

However, although classical second order methods can help, due to higher order derivatives, the objective function may have a lot more non-linearity, as shown in Figure 8.2. The objective function often does not have the nice symmetrical shapes that the second-order “valley” picture builds in our mind. Instead, there are cliffs where the gradient rises sharply. When the parameters approach a cliff region, the gradient update step can move the learner towards a very bad configuration, ruining much of the progress made during recent training iterations.

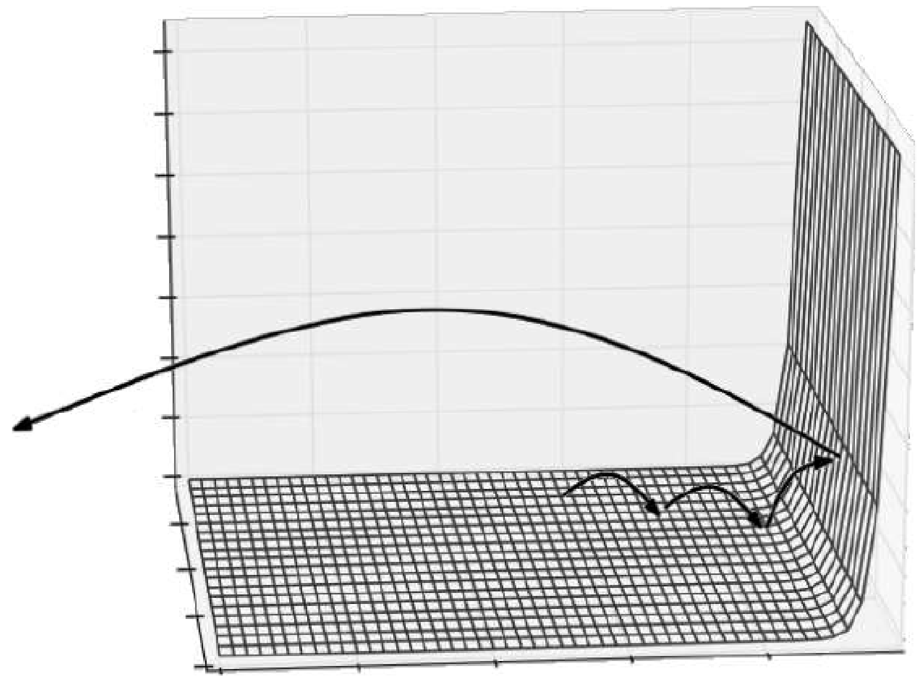

Figure 8.2: Contrary to what is shown in Figure 8.1, the objective function for highly non-linear deep neural networks or for recurrent neural networks is typically not made of symmetrical sides. As shown in the figure, there are sharp non-linearities that give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly ruining a lot of the optimization work that had been done. Figure graciously provided by Razvan Pascanu (Pascanu, 2014).

As illustrated in Figure 8.3, the cliff can be dangerous whether we approach it from above or from below, but fortunately there are some fairly straightforward heuristics that allow one to avoid its most serious consequences. The basic idea is to limit the size of the jumps that one would make. Indeed, one should keep in mind that when we use the gradient to make an update of the parameters, we are relying on the assumption of *infinitesimal moves*. There is no guarantee that making a finite step of the parameters $\boldsymbol{\theta}$ in the direction of the gradient will yield an improvement. The only thing that is guaranteed is that a *small enough* step in that direction will be helpful. As we can see from Figure 8.3, in the presence of a cliff (and in general in the presence of very large gradients), the decrease in the objective function expected from going in the direction of the gradient is only valid for a very small step. In fact, because the objective function is usually bounded in its actual value (within a finite domain), when the gradient is large at $\boldsymbol{\theta}$, it typically only remains like this (especially, keeping its sign) in a small region around $\boldsymbol{\theta}$. Otherwise, the value of the objective function would have to change a lot: if the slope was consistently large in some direction as we would move in that direction, we would be able to decrease the objective function value by a very large amount by following it, simply because the total change is the integral over some path of the directional derivatives along that path.

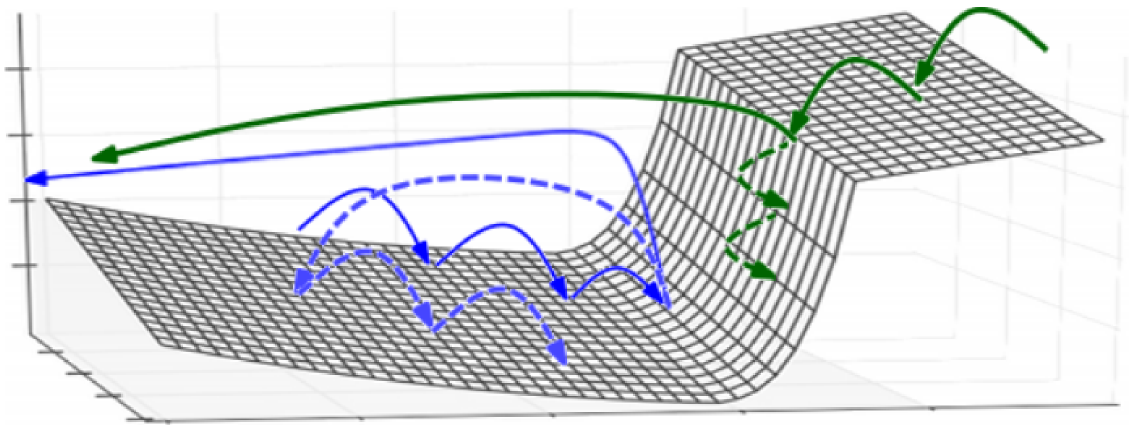

Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyperparameter, although not a very critical one). Using such a gradient clipping heuristic (dotted arrows trajectories) helps to avoid the destructive big moves which would happen when approaching the cliff, either from above or from below (bold arrows trajectories). Figure graciously provided by Razvan Pascanu (Pascanu, 2014).

The gradient clipping heuristics are described in more detail in Sec. 10.7.7. The basic idea is to bound the magnitude of the update step, i.e., not trust the gradient too much when it is very large in magnitude. The context in which such cliffs have been shown to arise in particular is that of recurrent neural networks, when considering long sequences, as discussed in the next section.

8.2.6 An Introduction to Learning Long-Term Dependencies

Parametrized dynamical systems such as recurrent neural networks (Chapter 10) face a particular optimization problem which is different but related to that of training very deep networks. We introduce this issue here and refer to reader to Sec. 10.7 for a deeper treatment along with a discussion of approaches that have been proposed to reduce this difficulty.

Exploding or Vanishing Product of Jacobians

The simplest explanation of the problem, which is shared among very deep nets and recurrent nets, is that in both cases the final output is the composition of a large number of non-linear transformations. Even though each of these non-linear stages may be relatively smooth (e.g. the composition of an affine transformation with a hyperbolic tangent or sigmoid), their composition is going to be much "more non-linear", in the sense that derivatives through the whole composition will tend to be either very small or very large, with many strong variations. This arises simply because the Jacobian (matrix of derivatives) of a composition is the product of the Jacobians of each stage. If

$$f = f_T \circ f_{T-1} \circ \ldots, f_2 \circ f_1$$

then the Jacobian matrix of derivatives of $f(x)$ with respect to its input vector $\boldsymbol{x}$ is the product

$$f' = f'_T f'_{T-1} \ldots, f'_2 f_1 \tag{8.5}$$

where

$$f' = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}$$

and

$$f'_t = \frac{\partial f_t(\boldsymbol{a}_t)}{\partial \boldsymbol{a}_t},$$

where $\boldsymbol{a}_t = f_{t-1}(f_{t-2}(\ldots, f_2(f_1(\boldsymbol{x}))))$. When taking derivatives through compositions, we obtain matrix multiplication of the corresponding Jacobians. This is illustrated in Figure 8.4.

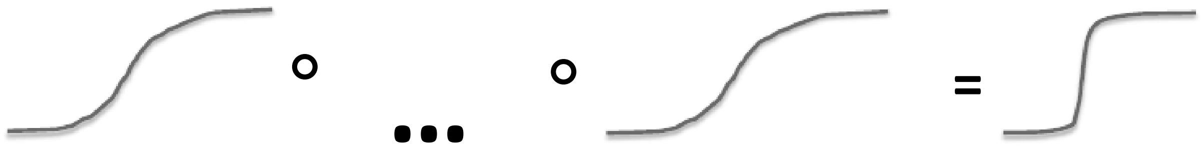

Figure 8.4: When composing many non-linearities (like the activation non-linearity in a deep or recurrent neural network), the result is highly non-linear, typically with most of the values associated with a tiny derivative, some values with a large derivative, and many ups and downs (not shown here).

In the scalar case, we can imagine that multiplying many numbers together tends to be either very large or very small. In the special case where all the numbers in the product have the same value α, this is obvious, since α^T goes to 0 if $\alpha < 1$ and goes to ∞ if $\alpha > 1$, as T increases. The more general case of non-identical numbers be understood by taking the logarithm of these numbers, considering them to be random, and computing the variance of the sum of these logarithms. Clearly, although some cancellation can happen, the variance grows with T, and in fact if those numbers are independent, the variance grows linearly with T, i.e., the size of the sum (which is the standard deviation) grows as $\sqrt{T}$, which means that the product grows roughly as e^T (consider the variance of log-normal variate X if $\log X$ is normal with mean 0 and variance T).

It would be interesting to push this analysis to the case of multiplying square matrices instead of multiplying numbers, but one might expect qualitatively similar conclusions, i.e., the size of the product somehow grows with the number of matrices, and that it grows exponentially. In the case of matrices, one can get a new form of cancellation due to leading eigenvectors being well aligned or not. The product of matrices will blow up only if, among their leading eigenvectors with eigenvalue greater than 1, there is enough "in common" (in the sense of the appropriate dot products of leading eigenvectors of one matrix and another).

However, this analysis was for the case where these numbers are independent. In the case of an ordinary recurrent neural network (developed in more detail in Chapter 10), these Jacobian matrices are highly related to each other. Each layer-wise Jacobian is actually the product of two matrices: (a) the recurrent matrix $\boldsymbol{W}$ and (b) the diagonal matrix whose entries are the derivatives of the non-linearities associated with the hidden units, which vary depending on the time step. This makes it likely that successive Jacobians have similar eigenvectors, making the product of these Jacobians explode or vanish even faster.

Recurrent Networks and the Difficulty of Learning Long-Term Dependencies

The consequence of the exponential convergence of these products of Jacobians towards either very small or very large values is that it makes the learning of *long-term dependencies* particularly difficult, as we explain below and was independently introduced in Hochreiter (1991) and Bengio *et al.* (1993, 1994) for the first time.

Consider a fairly general parametrized dynamical system (which includes classical recurrent networks as a special case, as well as all their known variants), processing a sequence of inputs, $x_1, \ldots, x_t, \ldots$, involving iterating over the transition operator:

$$s_t = F_{\boldsymbol{\theta}}(s_{t-1}, x_t) \tag{8.6}$$

where s_t is called the state of the system and $F_{\boldsymbol{\theta}}$ is the recurrent function that maps the previous state and current input to the next state. The state can be used to produce an output via an output function,

$$o_t = g_{\omega}(s_t), \tag{8.7}$$

and a loss L_t is computed at each time step t as a function of o_t and possibly of some targets y_t. Let us consider the gradient of a loss L_T at time T with respect to the parameters $\boldsymbol{\theta}$ of the recurrent function $F_{\boldsymbol{\theta}}$. One particular way to decompose the gradient $\frac{\partial L_T}{\partial \boldsymbol{\theta}}$ using the chain rule is the following:

$$\frac{\partial L_T}{\partial \boldsymbol{\theta}} = \sum_{t \leq T} \frac{\partial L_T}{\partial s_t} \frac{\partial s_t}{\partial \boldsymbol{\theta}}$$
$$\frac{\partial L_T}{\partial \boldsymbol{\theta}} = \sum_{t \leq T} \frac{\partial L_T}{\partial s_T} \frac{\partial s_T}{\partial s_t} \frac{\partial F_{\boldsymbol{\theta}}(s_{t-1}, x_t)}{\partial \boldsymbol{\theta}} \tag{8.8}$$

where the last Jacobian matrix only accounts for the immediate effect of $\boldsymbol{\theta}$ as a parameter of $F_{\boldsymbol{\theta}}$ when computing $s_t = F_{\boldsymbol{\theta}}(s_{t-1}, x_t)$, i.e., not taking into account the indirect effect of $\boldsymbol{\theta}$ via s_{t-1} (otherwise there would be double counting and the result would be incorrect). To see that this decomposition is correct, please refer to the notions of gradient computation in a flow graph introduced in Sec. 6.4.3, and note that we can construct a graph in which $\boldsymbol{\theta}$ influences each s_t, each of which influences L_T via s_T. Now let us note that each Jacobian matrix $\frac{\partial s_T}{\partial s_t}$ can be decomposed as follows:

$$\frac{\partial s_T}{\partial s_t} = \frac{\partial s_T}{\partial s_{T-1}} \frac{\partial s_{T-1}}{\partial s_{T-2}} \cdots \frac{\partial s_{t+1}}{\partial s_t} \tag{8.9}$$

which is of the same form as Eq. 8.5 discussed above, i.e., which tends to either vanish or explode.

As a consequence, we see from Eq. 8.8 that $\frac{\partial L_T}{\partial \boldsymbol{\theta}}$ is a weighted sum of terms over spans $T - t$, *with weights that are exponentially smaller (or larger)* for longer-term dependencies relating the state at t to the state at T. As shown in Bengio *et al.* (1994), in order for a recurrent network to *reliably store memories*, the Jacobians $\frac{\partial s_t}{\partial s_{t-1}}$ relating each state to the next must have a determinant that is less than 1 (i.e., yielding to the formation of *attractors* in the corresponding dynamical system). Hence, *when the model is able to capture long-term dependencies it is also in a situation where gradients vanish and long-term dependencies have an exponentially smaller weight than short-term dependencies in the total gradient.* It does not mean that it is impossible to learn, but that it might take a very long time to learn long-term dependencies, because the signal about these dependencies will tend to be hidden by the smallest fluctuations arising from short-term dependencies. In practice, the experiments in Bengio *et al.* (1994) show that as we increase the span of the dependencies that need to be captured, gradient-based optimization becomes increasingly difficult, with the probability of successful learning rapidly reaching 0 after only 10 or 20 steps in the case of the ordinary recurrent net and stochastic gradient descent (Sec. 8.3.2).

For a deeper treatment of the dynamical systems view of recurrent networks, consider Doya (1993); Bengio *et al.* (1994); Siegelmann and Sontag (1995), with a review in Pascanu *et al.* (2013a). Sec. 10.7 discusses various approaches that have been proposed to reduce the difficulty of learning long-term dependencies (in some cases allowing one to reach to hundreds of steps), but it remains one of the main challenges in deep learning.

8.2.7 Inexact Gradients

Most optimization algorithms are primarily motivated by the case where we have exact knowledge of the 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 best 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 of this book. For example, persistent 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 these 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.

8.2.8 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 only in reducing 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 Optimization Algorithms I: Basic Algorithms

In Sec. 6.4.3, we discussed the backpropagation algorithm (backprop): that is, how to efficiently compute the gradient of the loss with respect to the model parameters. The backpropagation algorithm does *not* specify how we use this gradient to update the weights of the model.

In this section we introduce a number of gradient-based *learning algorithms* that have been proposed to optimize the parameters of deep learning models.

8.3.1 Gradient Descent

Gradient descent is the most basic gradient-based algorithm one might apply to train a deep model. The algorithm is also sometimes called *batch gradient descent* or *deterministic gradient descent* in neural network papers because it updates the parameters only after having seen a batch of all the training examples and the gradient is computed exactly and deterministically. This method involves updating the model parameters $\boldsymbol{\theta}$ [1] with a small step in the direction of the gradient of the objective function, i.e., for neural networks that includes the terms for all the training examples as well as any regularization terms. For the case of supervised learning with data pairs $[\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)}]$ we have:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \nabla_{\boldsymbol{\theta}} \sum_t L(f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}), \tag{8.10}$$

where ϵ is the *learning rate*, an optimization hyperparameter that controls the size of the step the the parameters take in the direction of the gradient. Following the gradient in this way is only guaranteed to reduce the loss if ϵ is smaller than a threshold value[2]. Note that the learning rate ϵ does not need to be decreased towards 0, in the batch gradient case. We do get convergence with a fixed ϵ because the gradients get smaller and smaller (approaching 0) as we approach the local minimum of a function whose gradients are Lipschitz continuous.

In fact, once the algorithm has reached a convex basin of attraction, convergence is fast, with the magnitude of the difference to the minimum decreasing quickly. If the smallest second derivative in any direction (the smallest eigenvalue of the Hessian) is at least μ and the largest second derivative is at most L, then *linear convergence* is achieved: the *excess error*, or difference between the current value of the objective function and the value at the local minimum, is at least reduced by a fixed factor $(1 - \frac{\mu}{\mathcal{L}})$ after each update. The excess error goes down

[1] in the case of a deep neural network, these parameters would include the weights and biases associated with each layer

[2] greater than $1/\mathcal{L}$, where $\mathcal{L}$ is the Lipschitz constant or the largest second derivative in any direction, and this statement is true only if the gradient is Lipschitz-continuous

exponentially towards zero, or in other words, the error decreases exponentially fast towards its (locally) minimum value. If there is no lower bound on the smallest second derivative μ, then convergence is sub-linear and the error decreases in $O(1/k)$ after k steps (Bertsekas, 2004). In addition, the number of training iterations to reach a particular error level is proportional to $\frac{\mathcal{L}}{\mu}$. Very slow convergence can thus occur when the Hessian is ill-conditioned (μ is tiny or 0).

In spite of its impressive convergence properties (when the Hessian is not ill-conditioned), batch gradient descent is rarely used in machine learning because it does not exploit the particular structure of the objective function, which is written as a large sum of generally i.i.d. terms associated with each training example. Exploiting this structure is what allows *stochastic* gradient descent, discussed next, to achieve much faster practical convergence, as analyzed theoretically by Bottou and Bousquet (2008).

8.3.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) and its variants are probably the most used optimization algorithm for machine learning in general and for deep learning in particular. It is very similar to (batch) gradient descent except that it uses a stochastic (i.e., noisy) estimator of the gradient to perform its update. With machine learning, this is typically obtained by sampling one or a small subset of m of the training examples and computing their gradient, as shown in Algorithm 8.1. When the examples are i.i.d., it means that the expected value $E[\hat{\boldsymbol{g}}]$ of this estimated gradient (averaging over different draws of the examples used to compute the estimated gradient) equals the true gradient. Thus, the gradient estimator is unbiased:

$$E[\hat{\boldsymbol{g}}] = \boldsymbol{g},$$

where $\boldsymbol{g}$ is the total gradient.

When $m = 1$, Algorithm 8.1 is sometimes called *online gradient descent*. When $m > 1$ but m a fraction of the number of training examples, this algorithm is sometimes called *minibatch* SGD. See Sec. 8.1.3 about minibatch optimization algorithms.

A crucial hyper-parameter that is introduced when one applies SGD is the learning rate (η_k in Algorithm 8.1). Whereas ordinary gradient descent can work with a fixed learning rate, it is necessary to allow SGD's learning rate to decrease at an appropriate rate during training, if one wants to converge to a minimum. This is because the SGD gradient estimator introduces a source of noise (the random sampling of m training examples) that does not become 0 even when we arrive at a minimum (whereas the true gradient becomes small and then 0 when we approach and reach a minimum). A sufficient condition to guarantee

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

```
Require: Learning rate $\eta$.
Require: Initial parameter $\boldsymbol{\theta}$
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$.
    Set $\hat{\boldsymbol{g}} = \boldsymbol{0}$
    for $i = 1$ to $m$ do
      Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})/m$
    end for
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_k - \eta \hat{\boldsymbol{g}}$
  end while
```

convergence is that

$$\sum_{k=1}^{\infty} \eta_k = \infty, \quad \text{and}$$
$$\sum_{k=1}^{\infty} \eta_k^2 < \infty. \tag{8.11}$$

For a deeper treatment of SGD, see Bottou (1998), which covers the case when the objective function is not convex in the parameters.

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, reaching to the online or streaming limit, where each example is only seen once.

Again, let us use k to denote the number of iterations, μ the smallest eigenvalue of the Hessian and $\mathcal{L}$ its largest eigenvalue. We have seen that batch gradient excess error converges at a rate $O((1 - \frac{\mu}{\mathcal{L}})^k)$ in the strongly convex case (in a convex basin of attraction where the smallest second derivatives are lower bounded by μ) and in $O(1/k)$ in the convex case (or $O(1/k^2)$ with accelerated convergence or Nesterov momentum (Nesterov, 1983), discussed below, Sec. 8.3.4). Unfortunately, the linear convergence (the error going down towards its minimum exponentially fast) of the strongly convex case is lost in the stochastic setup. With SGD, error converges in $O(1/\sqrt{k})$ in the convex case and in $O(1/k)$ in the strongly convex case, and these bounds cannot be improved unless extra conditions are assumed.

Because many more stochastic updates can be performed for the price of one deterministic update, stochastic gradient converges initially much faster than deterministic (or batch) gradient descent. On the other hand, after some number of iterations, deterministic gradient descent (or equivalently, using larger and larger minibatches) will converge to lower values of the objective function, because of its faster rate. However, in actual applications, for the conditions in which large neural networks are trained on large datasets, SGD variants remain the choice of practitioners. Training error can in principle be greatly improved by following SGD by a deterministic gradient-based optimization method, but note that it may be at the cost of worse generalization error. Indeed, generalization error cannot go down faster than the $O(1/k)$ rate, which corresponds to the statistical rate of convergence (in the best possible case, which is the online case, when every example is new), or the Cramér-Rao bound (Cramér, 1946; Rao, 1945). As pointed out by Bottou and Bousquet (2008), this makes it unclear whether it is worthwhile pursuing a faster rate by optimization techniques that would make training error converge at a faster asymptotic rate. Without a faster convergence on the test set, the faster training set convergence is likely to correspond to overfitting.

However, it is not just the asymptotic rate of convergence that matters. The "constants" hidden by the $O()$ notation matter, in this case, and the speed of convergence early in the process also matters. Note that second order methods and adaptive learning rate methods described below can have an important effect on these constants, even though they do not change the asymptotic worst-case asymptotic rate of convergence. Although learning theory usually considers estimation error (variance) and approximation error (bias), one should also consider the effect of finite computational resources (training time), as in Bottou and Bousquet (2008). Although SGD approaches the minimum slowly in an asymptotic sense (when we consider what happens close to the minimum, because of the added noise), it approaches the region of the minimum exponentially fast! This was shown by Nedic and Bertsekas (2000): constant step-size SGD approaches some error level (higher than at the minimum) exponentially fast (multiplying the excess error by a fixed factor after each iteration, in average).

8.3.3 Momentum

While stochastic gradient descent remains a very popular optimization strategy, learning with it can sometimes be slow. This is especially true in situations where the gradient is small. When the gradient is consistent across consecutive minibatches, we know that we can afford to take larger steps in this direction.

The method of Momentum Polyak (1964) is designed to accelerate learning, especially in the face of small and consistent gradients. The intuition behind momentum, as the name suggests, is derived from a physical interpretation of the optimization process. Imagine you have a small ball (think of a marble) that represents the current position in parameter space (for our purposes here we can imagine a 2-D parameter space). Now consider that the ball is on a gentle slope, while the instantaneous force pulling the ball down hill is relatively small, their contributions combine and the downhill velocity of the ball gradually begins to increase over time. The momentum method is designed to inject this kind of downhill acceleration into gradient-based optimization. The effect of momentum is illustrated in Fig. 8.5.

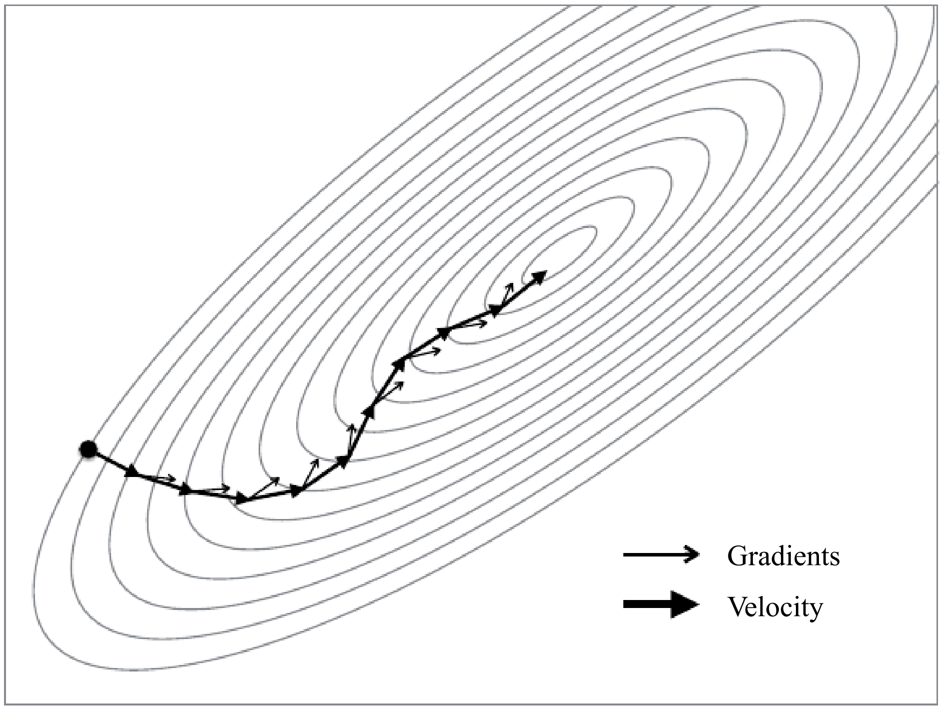

Figure 8.5: The effect of momentum on the progress of learning. Momentum acts to accumulate gradient contributions over training iterations. Directions that consistently have positive contributions to the gradient will be augmented.

Formally, we introduce a variable $\boldsymbol{v}$ that plays the role of velocity (or momentum) that accumulates gradient. The update rule is given by:

$$\boldsymbol{v} \leftarrow +\alpha \boldsymbol{v} + \eta \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{t=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \boldsymbol{y}^{(t)}) \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}} \left(\frac{1}{n} \sum_{t=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \boldsymbol{y}^{(t)}) \right)$. The larger α is relative to η, the more previous gradients affect the current direction. The overall learning rate, which in the case of SGD, was a simple hyperparameter, is here a relatively complicated function of α and η. The SGD+momentum algorithm is given in Algorithm 8.2.

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

```
Require: Learning rate $\eta$, momentum parameter $\alpha$.
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 $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \mathbf{0}$
    for $i = 1$ to $m$ do
      Compute gradient estimate: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \eta \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

8.3.4 Nesterov Momentum

Sutskever *et al.* (2013) introduced a variant of the momentum algorithm that was inspired by Nesterov.

$$\boldsymbol{v} \leftarrow +\alpha \boldsymbol{v} + \eta \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{t=1}^{m} L\Big(\boldsymbol{f}(\boldsymbol{x}^{(t)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(t)}\Big) \right],$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v},$$

where the parameters α and η 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. Figure 8.6 illustrates the difference between Nesterov momentum and standard momentum. The complete Nesterov momentum algorithm is presented in Algorithm 8.3.

In the batch gradient case, Nesterov momentum in the convex basin of attraction (not necessarily strictly convex[3]) brings the rate of convergence of the excess error from $O(1/k)$ (after k steps, in the batch gradient case) to $O(1/k^2)$ as shown by Nesterov (1983). In the strictly convex case, convergence goes from from $O(1 - \frac{\mu}{L})$ (batch gradient) to $O(1 - \sqrt{\frac{\mu}{L}})$. Unfortunately, in the stochastic gradient case, Nesterov momentum does not improve the rate of convergence.

[3] which means that the Hessian can be ill-conditioned

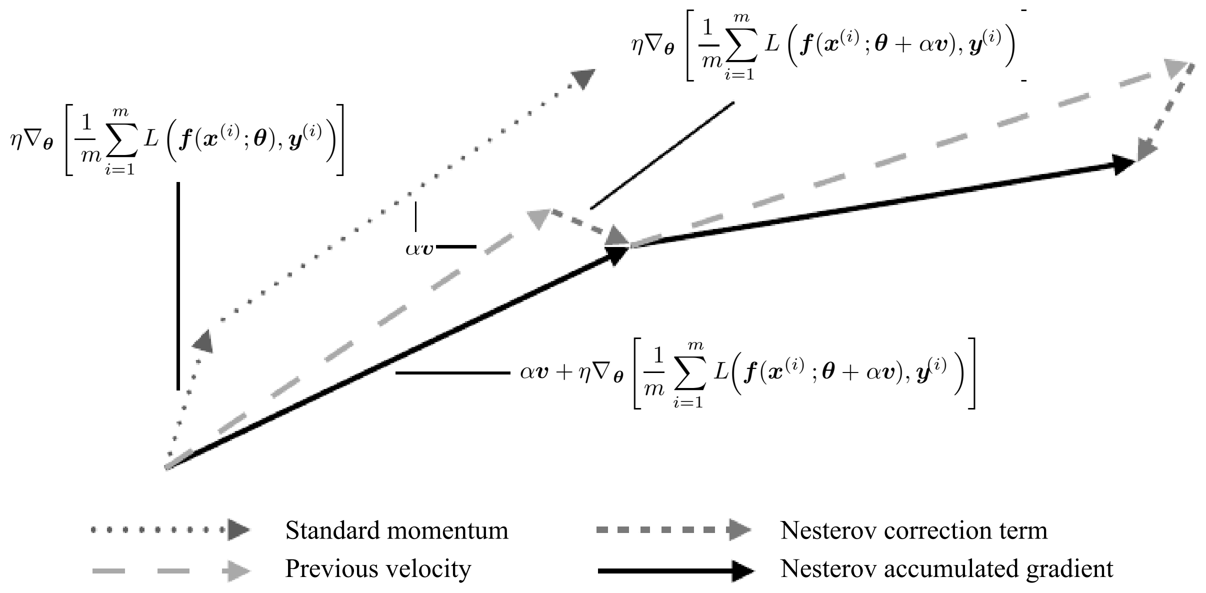
$\eta\nabla_{\boldsymbol{\theta}}\left[\frac{1}{m}\sum_{i=1}^{m}L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}+\alpha\boldsymbol{v}),\boldsymbol{y}^{(i)}\right)\right]$
$\eta\nabla_{\boldsymbol{\theta}}\left[\frac{1}{m}\sum_{i=1}^{m}L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}),\boldsymbol{y}^{(i)}\right)\right]$
$\alpha\boldsymbol{v}$
$\alpha\boldsymbol{v}+\eta\nabla_{\boldsymbol{\theta}}\left[\frac{1}{m}\sum_{i=1}^{m}L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}+\alpha\boldsymbol{v}),\boldsymbol{y}^{(i)}\right)\right]$
Standard momentum
Nesterov correction term
Previous velocity
Nesterov accumulated gradient

Figure 8.6: An illustration of the difference between Nesterov momentum and standard momentum. Nesterov momentum incorporates the gradient after the velocity is already applied. This figure is derived from a similar image in Geoff Hinton's Coursera lectures.

8.4 Optimization Algorithms II: 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. In reality, as we've discussed in Sec.s 4.3 and 8.2, we often have a subset of parameters to which the cost is much more sensitive. These directions in parameter space will limit the size of the SGD learning rate and consequently limit the progress that can be made in the other, less sensitive directions. While the use of momentum can go some way to alleviate these issues, it does so by introducing another hyperparameter that may be just as difficult to set as the original learning rate. In the face of this, it is natural to ask if there is another way. Can learning rates be set automatically and independently for each parameter?

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 it 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

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

```
Require: Learning rate $\eta$, momentum parameter $\alpha$.
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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Apply interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Set $\boldsymbol{g} = \boldsymbol{0}$
    for $i = 1$ to $m$ do
      Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \eta \boldsymbol{g}$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

been introduced that adapt the learning rates of model parameters. This section will briefly review a few of these algorithms.

8.4.1 AdaGrad

The AdaGrad algorithm, shown in Algorithm 8.4, individually adapts the learning rates of all model parameters by scaling them inversely proportional to an accumulated sum of squared partial derivatives over all training iterations. 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 convex optimization context, 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* results in a premature and excessive decrease in the effective learning rate.

8.4.2 RMSprop

The RMSprop algorithm (Hinton, 2012) addresses the deficiency of AdaGrad by changing the gradient accumulation into an exponentially weighted moving average. As we have previously discussed (especially in Sec. 8.2), in deep networks, the optimization surface is far from convex. Directions in parameter space with strong partial derivatives early in training may flatten out as training progresses. The introduction of the exponentially weighted moving average allows the effec-

Algorithm 8.4 The Adagrad algorithm

```
Require: Global learning rate $\eta$,
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$,
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \mathbf{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Accumulate gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g}^2$ (square is applied element-wise)
    Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\eta}{\sqrt{\boldsymbol{r}}}\boldsymbol{g}$.   % ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}_t$
  end while
```

tive learning rates to adapt to the changing local topology of the loss surface.

RMSprop is shown in its standard form in Algorithm 8.5 and combined with Nesterov momentum in Algorithm 8.6. Note that compared to AdaGrad, the use of the moving average does introduce a new hyperparameter, ρ, that controls the length scale of the moving average.

Empirically RMSprop has shown to be an effective and practical optimization algorithm for deep neural networks. It is easy to implement and relatively simple to use (i.e. there does not appear to be a great sensitivity to the algorithm's hyperparameters). It is currently one of the "go to" optimization methods being employed routinely by deep learning researchers.

8.4.3 Adam

Adam (Kingma and Ba, 2014) is yet another adaptive learning rate optimization algorithm and is presented in Algorithm 8.7. In the context of the earlier algorithms, it is perhaps best seen as a variant on RMSprop+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 which is not particularly well motivated. Second, Adam includes 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 term. Thus, unlike in Adam, the RMSprop second-order moment

Algorithm 8.5 The RMSprop algorithm

```
Require: Global learning rate $\eta$, decay rate $\rho$.
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize accumulation variables $\boldsymbol{r} = 0$
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \boldsymbol{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g}^2$

    Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\eta}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   % ( $\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

estimate may have high bias early in training.

8.4.4 AdaDelta

AdaDelta is another recently introduced optimization algorithm that seeks to directly address the issues with AdaGrad. AdaDelta starts from an attempt to incorporate some second-order gradient information (see 4.3) into the optimization algorithm. In particular, consider the Newton's step for a single parameter $\boldsymbol{\theta}_j$ on the loss for a single example $\{\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\}$.[4]

$$\Delta\theta_j = -\frac{1}{\frac{\partial^2}{\partial\boldsymbol{\theta}_j^2} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})} \frac{\partial}{\partial\boldsymbol{\theta}_j} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})$$

$$\frac{1}{\frac{\partial^2}{\partial\boldsymbol{\theta}_j^2} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})} = \frac{\Delta\theta_j}{\frac{\partial}{\partial\boldsymbol{\theta}_j} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})}$$

Thus, assuming a diagonal Hessian and a Newton update (which we do not have), its inverse could be estimated as the ratio of the increment $\Delta\boldsymbol{\theta}_j$ over the first

[4]Recall, from Chapter 4, that Newton's method — in the single dimension of $\boldsymbol{\theta}_j$ — can be motivated by looking at the Taylor series expansion of the loss around the current point θ^0: $L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0 + \boldsymbol{e}_j \Delta\boldsymbol{\theta}_j), \boldsymbol{y}^{(i)}) \approx L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)}) + \boldsymbol{e}_j \frac{\partial}{\partial\boldsymbol{\theta}_j} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})\ \Delta\boldsymbol{\theta}_j + \boldsymbol{e}_j \frac{1}{2} \frac{\partial^2}{\partial\boldsymbol{\theta}_j^2} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})\ \Delta\boldsymbol{\theta}_j^2$. This expression reaches its extremum, with respect to $\Delta\theta_j$ when its derivative (w.r.t. $\Delta\boldsymbol{\theta}_j$) is equal to zero. Using this, we can solve for the optimal step $\Delta\theta_j = -\frac{\frac{\partial}{\partial\boldsymbol{\theta}_j} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})}{\frac{\partial^2}{\partial\boldsymbol{\theta}_j^2} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^0), \boldsymbol{y}^{(i)})}$.

Algorithm 8.6 RMSprop algorithm with Nesterov momentum

```
Require: Global learning rate $\eta$, decay rate $\rho$, momentum coefficient $\alpha$.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  Initialize accumulation variable $\boldsymbol{r} = \mathbf{0}$
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$.
    Compute interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Set $\boldsymbol{g} = \mathbf{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho)\boldsymbol{g}^2$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\eta}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$.   % ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while
```

partial derivative of the loss. AdaDelta separately estimates this ratio as the ratio of RMS estimates, using the square-roots of an exponentially weighted moving average over squares of increments (in the numerator) and partial derivatives (in the denominator). The complete AdaDelta algorithm is shown in Fig. 8.8.

8.4.5 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. Tom Schaul (2014) presented a valuable comparison of a large number of optimization algorithms across a wide range of learning tasks. While the results suggest that this family of algorithms (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+momentum, RMSprop, RMSprop+momentum, AdaDelta and Adam. The choice of which algorithm to use, as this point, seems to depend as much on the users familiarity with the algorithm (for ease of hyperparameter tuning) as it does on any established notion of superior performance.

Algorithm 8.7 The Adam algorithm

```
Require: Step-size $\alpha$
Require: Decay rates $\rho_1$ and $\rho_2$, constant $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$,
  Initialize timestep $t = 0$
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \boldsymbol{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    $t \leftarrow t + 1$
    Get biased first moment: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Get biased second moment: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g}^2$
    Compute bias-corrected first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
    Compute bias-corrected second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\alpha \frac{\boldsymbol{s}}{\sqrt{\boldsymbol{r}}+\epsilon}\boldsymbol{g}$   % (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

8.5 Optimization Algorithms III: Approximate Second-Order Methods

In this section we discuss the application of second-order methods to the training of deep networks. For simplicity of exposition, the only objective function we will consider is the empirical risk:

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

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.

8.5.1 Newton's Method

In section 4.3, we discussed the difference between first-order gradient methods and second-order gradient methods. Namely, that second-order gradient methods

Algorithm 8.8 The Adadelta algorithm

```
Require: Decay rate $\rho$, constant $\epsilon$
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize accumulation variables $\boldsymbol{r} = \boldsymbol{0}$, $\boldsymbol{s} = \boldsymbol{0}$,
  while Stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \boldsymbol{0}$
    for $i = 1$ to $m$ do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g}^2$
    Compute update: $\Delta\boldsymbol{\theta} = -\frac{\sqrt{\boldsymbol{s}+\epsilon}}{\sqrt{\boldsymbol{r}+\epsilon}} \boldsymbol{g}$   % (operations applied element-wise)
    Accumulate update: $\boldsymbol{s} \leftarrow \rho \boldsymbol{s} + (1 - \rho) [\Delta\boldsymbol{\theta}]^2$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while
```

use information about the partial derivatives of the partial derivatives of the loss - i.e. second-order gradient information.

In multiple dimensions, we may need to examine all of the second derivatives of the function. These derivatives can be collected together into a matrix called the *Hessian matrix*. The Hessian matrix of a function $J(\boldsymbol{\theta})$, denoted $\boldsymbol{H}(J)(\boldsymbol{\theta})$, or simply as $\boldsymbol{H}$, is defined as

$$\boldsymbol{H}(J)(\boldsymbol{\theta})_{i,j} = \frac{\partial^2}{\partial\theta_i \partial\theta_j} J(\boldsymbol{x}; \boldsymbol{\theta}). \tag{8.12}$$

Here the Hessian is computed with respect to the parameters $\boldsymbol{\theta}$ of J. Equivalently, the Hessian is the Jacobian of the gradient.

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:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(J)(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0). \tag{8.13}$$

If we then solve for the critical point of this function, we obtain the Newton parameter update rule:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - [H(J(\boldsymbol{\theta}_0))]^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) \tag{8.14}$$

Thus for a locally quadratic function (with positive definite H), by rescaling the gradient by 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 8.9.

Algorithm 8.9 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 $\boldsymbol{\theta}_0$
  while stopping criterion not met do
    Initialize the gradient $\boldsymbol{g} = \boldsymbol{0}$
    Initialize the gradient $\boldsymbol{H} = \boldsymbol{0}$
    for $i = 1$ to $m$ % loop over the training set. do
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \frac{1}{m}\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
      Compute gradient: $\boldsymbol{H} \leftarrow \boldsymbol{H} + \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end for
    Compute Hessian inverse: $\boldsymbol{H}^{-1}$
    Compute update: $\Delta\boldsymbol{\theta}_t = \boldsymbol{H}^{-1}\boldsymbol{g}$
    Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t$
  end while
```

One way to understand how Newton's method works is to consider the Hessian $\boldsymbol{H}$ as a transformation from the Euclidean space where the problem is defined, to a space where gradient optimization is simplified (at least under the quadratic assumption). For real, symmetric H, its eigendecomposition is given by $\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top$, where $\boldsymbol{Q}$ is an orthogonal matrix[5].

We can now consider a change of variables from the original parameters $\boldsymbol{\theta}$ to an alternative parametrization $\boldsymbol{\phi}$ via the transformation $\boldsymbol{\phi} = \boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top\boldsymbol{\theta}$. Our goal is to re-express the quadratic approximation in $\boldsymbol{\phi}$-space. For this, we need to know how the gradient $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0)$ changes under this transformation. For that purpose, we need to compute $\nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0)$, which, by the chain rule of calculus, is given by:

$$\begin{aligned}\nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0) &= \left[\frac{\partial\boldsymbol{\theta}}{\partial\boldsymbol{\phi}}\right]^\top \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0) \\ &= \boldsymbol{Q}\boldsymbol{\Lambda}^{\frac{1}{2}}\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0) \qquad (8.15)\end{aligned}$$

Or equivalently, $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0) = \boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top\nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0)$. Using the eigendecomposition of $\boldsymbol{H}$ and the change in variables to $\boldsymbol{\phi}$, we can re-express the quadratic approximation

[5]Recall that an orthogonal matrix is square matrix with real-valued elements and whose columns and rows are orthogonal unit vectors, i.e. for orthogonal matrix $\boldsymbol{Q}$, $\forall i$, $\boldsymbol{Q}_{:,i}^\top\boldsymbol{Q}_{:,i} = 1$, $\boldsymbol{Q}_{i,:}\boldsymbol{Q}_{i,:}^\top = 1$ and for $i \neq j$, $\boldsymbol{Q}_{:,i}^\top\boldsymbol{Q}_{:,j} = 0$ and $\boldsymbol{Q}_{i,:}\boldsymbol{Q}_{j,:}^\top = 0$.

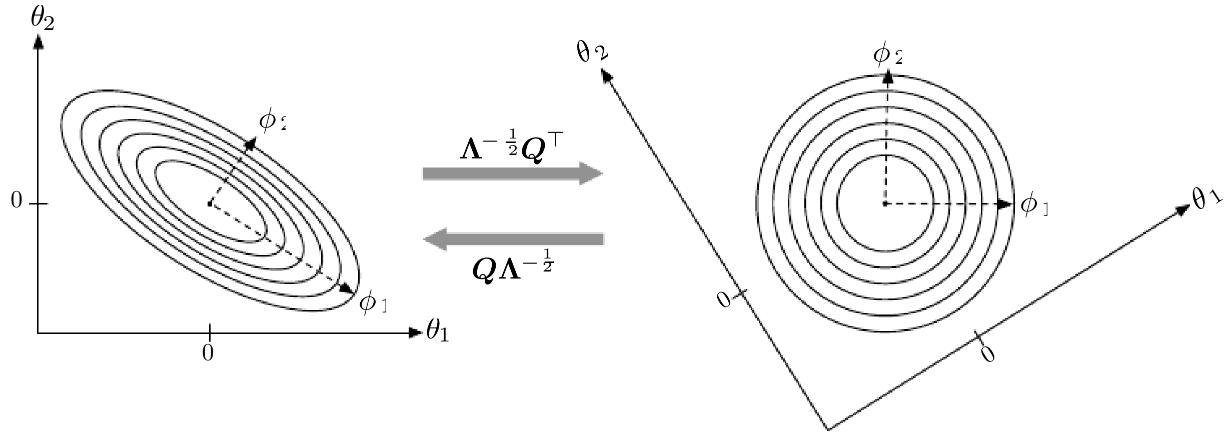

Figure 8.7: Newton's method maps an arbitrary and possibly ill-conditioned (see Sec.4.2) quadratic objective function in parameter space $\boldsymbol{\theta}$ into an alternative parameter space $\boldsymbol{\phi}$ where the quadratic objective function is isometric.

in Eq. 8.13.

$$\begin{aligned} f(\boldsymbol{\theta}) &\approx f(\boldsymbol{\theta}_0) + (\boldsymbol{\theta}-\boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta}-\boldsymbol{\theta}_0) \\ &= f(\boldsymbol{\theta}_0) + (\boldsymbol{\theta}-\boldsymbol{\theta}_0)^\top \boldsymbol{Q}\boldsymbol{\Lambda}^{\frac{1}{2}} \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta}-\boldsymbol{\theta}_0)^\top \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top(\boldsymbol{\theta}-\boldsymbol{\theta}_0) \\ &= f(\boldsymbol{\theta}_0) + \left(\boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top\boldsymbol{\theta} - \boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top\boldsymbol{\theta}_0\right)^\top \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0) \\ &\qquad + \frac{1}{2}\left(\boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top\boldsymbol{\theta} - \boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top\boldsymbol{\theta}_0\right)^\top \left(\boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top\boldsymbol{\theta} - \boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top\boldsymbol{\theta}_0\right) \\ &= f(\boldsymbol{\theta}_0) + (\boldsymbol{\phi}-\boldsymbol{\phi}_0)^\top \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\phi}-\boldsymbol{\phi}_0)^\top(\boldsymbol{\phi}-\boldsymbol{\phi}_0). \end{aligned} \tag{8.16}$$

Computing the gradient with respect to $\boldsymbol{\phi}$ and setting this to zero yields the Newton update in $\boldsymbol{\phi}$-space.

$$\boldsymbol{\phi}^* = \boldsymbol{\phi}_0 - \nabla_{\boldsymbol{\phi}} f(\boldsymbol{\theta}_0), \tag{8.17}$$

That is, in $\boldsymbol{\phi}$-space, the Newton update is transformed into a standard gradient step with unit learning rate. Figure 8.7 illustrates the effect of the transformation from $\boldsymbol{\theta}$-space to $\boldsymbol{\phi}$-space on the newton update.

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 Eq. 8.14.

In Sec. 8.2.4, we discussed how Newton's method is only appropriate when the local quadratic approximation holds. 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

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[H\left(f(\boldsymbol{\theta}_0)\right) + \alpha \boldsymbol{I}\right]^{-1} \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_0).$$

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.

Beyond the challenges created by certain topological 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 moderately sized networks the number of parameters K can be in the millions), Newton's method would require the inversion of a $K \times K$ 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 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.

8.5.2 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 Sec. 4.3 for details.), where line searches are applied iteratively in the direction associated with the gradient. Figure 8.8 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, i.e. the gradient, is guaranteed to be orthogonal to the previous line search direction.

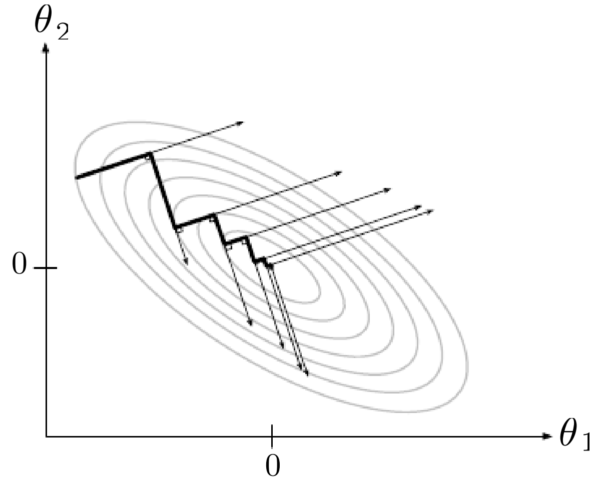

Figure 8.8: The method of steepest descent applied to a quadratic cost surface. The arrows show directions of the gradient. Note the back-and-forth progress made toward the optimum. By definition, at the minimum of the objective along a given direction, the gradient is orthogonal to that direction.

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 $\boldsymbol{d}_{t-1}$: $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \cdot \boldsymbol{d}_{t-1} = 0$. Since the gradient at this point defines the current search direction, $\boldsymbol{d}_t = \nabla_{\theta} J(\boldsymbol{\theta})$ 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 Fig. 8.8 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've already made in the direction of the previous line search. The method of conjugate gradients seeks to address this problem with steepest descent.

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:

$$\boldsymbol{d}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \beta_t \boldsymbol{d}_{t-1} \tag{8.18}$$

were 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 $\boldsymbol{d}_t^{\top} \boldsymbol{H}(J) \boldsymbol{d}_{t-1} = 0$, that is, if they are orthogonal in the ϕ-space defined above in Sec. 8.5.1:

$$\begin{aligned}
\boldsymbol{d}_t^\top \boldsymbol{H} \boldsymbol{d}_{t-1} &= 0 \\
\boldsymbol{d}_t^\top \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^\top \boldsymbol{d}_{t-1} &= 0 \\
\left(\boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top \boldsymbol{d}_{t-1}\right)^\top \left(\boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top \boldsymbol{d}_{t-1}\right) &= 0 \\
\boldsymbol{d}_t^{(\phi)\top} \boldsymbol{d}_{t-1}^{(\phi)} &= 0,
\end{aligned}$$

where we have defined $\boldsymbol{d}_{t-1}^{(\phi)} = \boldsymbol{\Lambda}^{\frac{1}{2}}\boldsymbol{Q}^\top \boldsymbol{d}_t$ as the direction $\boldsymbol{d}_{t-1}$ transformed to ϕ-space. The method of conjugate gradients can be interpreted as the application of the method of steepest descent in ϕ-space.

We motivated the method of conjugate gradients by arguing that it was more computationally viable for large problems that Newton's method. However, so far, we have only shown that the conjugate directions can be computed using the eigendecomposition of the Hessian, $\boldsymbol{H}$. Computing the Hessian, its inverse or its decomposition are exactly the calculations we are hoping to avoid. 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:
$$\beta_t = \frac{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.19}$$

2. Polak-Ribière:
$$\beta_t = \frac{\left(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})\right)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)}{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1})} \tag{8.20}$$

As illustrated in Fig. 8.9, for a quadratic surface, the conjugate directions ensure that the gradient along the previous direction does not increase in magnitude, i.e., we stay at the minimum along the previous directions. As a consequence, in a k-dimensional parameter space, conjugate gradients only requires k line searches to achieve the minimum. The conjugate gradient algorithm is given in Algorithm 8.10.

Nonlinear Conjugate Gradients: So far we've discussed the method of conjugate gradients as its 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 correspondiing objective function is far from quadratic. Perhaps surprisingly, the method of conjugate gradients is still applicable in this setting, though with some modification.

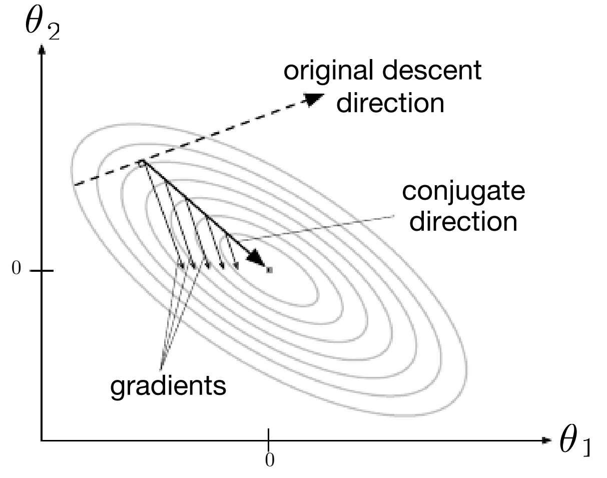

Figure 8.9: The conjugate direction ensures that the contribution to gradient along the original descent direction remains close to zero. As a result, the gradients evaluated along the conjugate direction are all orthogonal to the original descent direction. Figure adapted from (LeCun *et al.*, 1998a).

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 so-called *nonlinear conjugate gradients* algorithm includes occasional resets where the method of conjugate gradients is restarted with line search along the unaltered gradient.

Practionners 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).

8.5.3 BFGS

Like the method of conjugate gradients, the BFGS algorithm (the Broyden – Fletcher – Goldfarb – Shanno algorithm) attempts to bring some of the advantages of Newton's method without the computational burden. However, BFGS takes a more direct approach to the approximation of Newton's update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \left[\boldsymbol{H}\left(J(\boldsymbol{\theta}_0)\right)\right]^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0). \tag{8.21}$$

The primary computational difficulty in applying Newton's update is the calculation of the inverse Hessian $\boldsymbol{H}(J)(\boldsymbol{\theta}_0)$. The approach adopted by Quasi-Newton

Algorithm 8.10 Conjugate gradient method

```
Require: Initial parameters $\boldsymbol{\theta}_0$
  Initialize $\boldsymbol{\rho}_0 = \mathbf{0}$
  while stopping criterion not met do
    Initialize the gradient $\boldsymbol{g}_t = \mathbf{0}$
    for $i = 1$ to $m$ % loop over the training set. do
      Compute gradient: $\boldsymbol{g}_t \leftarrow \boldsymbol{g}_t + \frac{1}{m}\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    end forbackpropagation)
    Compute $\beta_t = \frac{(\boldsymbol{g}_t - \boldsymbol{g}_{t-1})^\top \boldsymbol{g}_t}{\boldsymbol{g}_{t-1}^\top \boldsymbol{g}_{t-1}}$ (Polak — Ribière)
    Compute search direction: $\boldsymbol{\rho}_t = -\boldsymbol{g}_t + \beta_t \boldsymbol{\rho}_{t-1}$
    Perform line search to find: $\eta^* = \text{argmin}_\eta \frac{1}{m}\sum_{i=1}^m L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$
  end while
```

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 $\boldsymbol{H}(J)$.

From Newton's update, in Eq. 8.21, we can see that the parameters at learning steps t are related via the secant condition (also known as the quasi-Newton condition):

$$\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = -\boldsymbol{H}^{-1}\left(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t+1}) - \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)\right) \tag{8.22}$$

The approximation to the Hessian inverse used in the BFGS procedure is constructed so as to satisfy this condition, with $\boldsymbol{M}$ in place of $\boldsymbol{H}^{-1}$. Specifically, $\boldsymbol{M}$ is updated according to:

$$\boldsymbol{M}_t = \boldsymbol{M}_{t-1} + \left(1 + \frac{\boldsymbol{\phi}^\top \boldsymbol{M}_{t-1}\boldsymbol{\phi}}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}}\right)\frac{\boldsymbol{\phi}^\top \boldsymbol{\phi}}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}} - \left(\frac{\boldsymbol{\Delta}\boldsymbol{\phi}^\top \boldsymbol{M}_{t-1} + \boldsymbol{M}_{t-1}\boldsymbol{\phi}\boldsymbol{\Delta}^\top}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}}\right), \tag{8.23}$$

where $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$, $\boldsymbol{\phi} = \boldsymbol{g}_t - \boldsymbol{g}_{t-1}$ and $\boldsymbol{\Delta} = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$. Eq. 8.23 shows that the BFGS procedure iteratively refines the approximation of the Hessian with rank one updates — the last three terms are all rank-1. This mean that if $\boldsymbol{\theta} \in \mathbb{R}^n$, then the computational complexity of the update is $O(n^2)$. The derivation of the BFGS approximation is given in many textbooks on optimization, including Luenberger (1984).

Once the inverse Hessian approximation is updated, that is $\boldsymbol{M}_t$ is calculated, 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, η^*, taken in this direction. The final update to the parameters is given by:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t.$$

Algorithm 8.11 BFGS method

```
Require: Initial parameters $\boldsymbol{\theta}_0$
  Initialize inverse Hessian $\boldsymbol{M}_0 = \boldsymbol{I}$
  while stopping criterion not met do
    Compute gradient: $\boldsymbol{g}_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$ (via batch backpropagation)
    Compute $\boldsymbol{\phi} = \boldsymbol{g}_t - \boldsymbol{g}_{t-1}$, $\boldsymbol{\Delta} = \boldsymbol{\theta}_t - \boldsymbol{\theta}_{t-1}$
    Approx $\boldsymbol{H}^{-1}$: $\boldsymbol{M}_t = \boldsymbol{M}_{t-1} + \left(1 + \frac{\boldsymbol{\phi}^\top \boldsymbol{M}_{t-1}\boldsymbol{\phi}}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}}\right) \frac{\boldsymbol{\phi}^\top \boldsymbol{\phi}}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}} - \left(\frac{\boldsymbol{\Delta}\boldsymbol{\phi}^\top \boldsymbol{M}_{t-1} + \boldsymbol{M}_{t-1}\boldsymbol{\phi}\boldsymbol{\Delta}^\top}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}}\right)$
    Compute search direction: $\boldsymbol{\rho}_t = \boldsymbol{M}_t \boldsymbol{g}_t$
    Perform line search to find: $\eta^* = \text{argmin}_\eta \; J(\boldsymbol{\theta}_t + \eta \boldsymbol{\rho}_t)$
    Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$
  end while*
```

The complete BFGS algorithm is presented in Algorithm 8.11.

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 spending 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 having 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 $\boldsymbol{M}$. Alternatively, by replacing the $\boldsymbol{M}_{t-1}$ in Eq. 8.23 with an identity matrix, the BFGS search direction update formula becomes:

$$\boldsymbol{\rho}_t = -\boldsymbol{g}_t + b\boldsymbol{\Delta} + a\boldsymbol{\phi}, \tag{8.24}$$

where the scalars a and b are given by:

$$a = -\left(1 + \frac{\boldsymbol{\phi}^\top \boldsymbol{\phi}}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}}\right) \frac{\boldsymbol{\Delta}^\top g_t}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}} + \frac{\boldsymbol{\phi}^\top g_t}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}}$$
$$b = \frac{\boldsymbol{\Delta}^\top g_t}{\boldsymbol{\Delta}^\top \boldsymbol{\phi}}$$

with $\boldsymbol{\phi}$ and $\boldsymbol{\Delta}$ as defined above. If used with exact line searches, the directions defined by Eq: 8.24 are mutually conjugate. However, unlike the method of conjugate gradients, this procedure remains well behaved when the the minimum of the line search is reached only approximately. This strategy can be generalized to include more information about the Hessian by storing previous values of $\boldsymbol{\phi}$ and $\boldsymbol{\Delta}$.

8.6 Optimization Algorithms IV: Natural Gradient Methods

Let us take a moment and consider the optimization problem we face when attempting to train a deep learning model. One perspective is that we are trying to adapt the parameters of the model to improve the performance of the network, as measured by the objective function. From this perspective, moving in the direction of steepest descent in parameter space is sensible, as it is in some sense, the direction where we can make the fastest progress. However, as we have seen when considering the Newton's method, the direction of instantaneous steepest descent, i.e. the gradient, in parameter space does not always correspond to the direction of greatest progress when moving in a straight line. One way to view the problem with the direction of steepest descent in parameter space is that it depends on the specific and arbitrary parametrization of the model. For example, replacing θ by a scaled version of θ would yield a different optimization trajectory, with gradient descent. We would like our optimization method to be as much as possible invariant to the specific choice of model parametrization. The method of natural gradients is an attempt to specify such a parametrization invariant optimization scheme.

Natural gradient methods are motivated by the insight that when considering directions of descent of the cost function, it is more 'natural' to consider this in the space of the functions represented by the model, rather than in the space of parameters. Every value of θ corresponds to an input-output function, and the set of functions achievable by a neural network corresponds to a low-dimensional manifold in the space of functions (of dimension no greater than the number of parameters). The natural metric in the space of function is given by how the output changes (at every possible input $\boldsymbol{x}$) as we move on that manifold.

Deep learning approaches to machine learning tasks, such as classification and regression, typically cast the output of the deep learning model as a probability distribution. For example, for a classification task, the model output is cast as the distribution over class labels $\boldsymbol{y}$ given an input $\boldsymbol{x}$. Natural gradient methods aim to find the direction of steepest descent in the space of the probability distribution output by the model. One way to formalize this is to first specify the natural "finite difference", Δ_N as

$$\begin{aligned}\Delta_N \equiv \arg\min_{\Delta\boldsymbol{\theta}} \mathbb{E}_{\hat{p}_{\text{data}}} \left[-\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(x)\right] \\ \text{s.t.} \mathrm{KL}\left(p_{\boldsymbol{\theta}}(x) \| p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}(x)\right) = \Delta\mathrm{KL}.\end{aligned} \tag{8.25}$$

Under the constraint of unit displacement $\Delta\mathrm{KL}$ of the output distribution under the KL divergence, the natural gradient finds the direction that minimizes the objective function, typically the negative log likelihood.

Taking $\Delta\boldsymbol{\theta} \to 0$, and assuming we have a discrete and bounded domain $\mathcal{X}$ over the probability distribution, we can express the Taylor series expansion of the log probability $\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ around $\boldsymbol{\theta}$ as

$$\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}} \approx \log p_{\boldsymbol{\theta}} + \left(\nabla \log p_{\boldsymbol{\theta}}\right)^\top \Delta\boldsymbol{\theta} + \frac{1}{2}\Delta\boldsymbol{\theta}^\top \left(\nabla^2 \log p_{\boldsymbol{\theta}}\right)\Delta\boldsymbol{\theta}. \tag{8.26}$$

First note that

$$\sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}} = \sum_{\mathcal{X}} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{\theta}} = \nabla_{\boldsymbol{\theta}} \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} = \nabla_{\boldsymbol{\theta}} 1 = 0, \tag{8.27}$$

which also makes sense intuitively: when the examples comes from the model distribution, the model is the best possible for fitting them, and the expected log-likelihood gradient is 0. Substituting the expression in Eq. 8.26 into the KL divergence $\mathrm{KL}\left(p_{\boldsymbol{\theta}} \| p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}\right)$ we have:

$$\begin{aligned}\mathrm{KL}\left(p_{\boldsymbol{\theta}} \| p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}\right) &= \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}} - \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}} \\ &\approx \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \log p_{\boldsymbol{\theta}} \\ &\quad - \sum_{\mathcal{X}} p_{\boldsymbol{\theta}} \left[\log p_{\boldsymbol{\theta}} + \left(\nabla \log p_{\boldsymbol{\theta}}\right)^\top \Delta\boldsymbol{\theta} + \frac{1}{2}\Delta\boldsymbol{\theta}^\top \left(\nabla^2 \log p_{\boldsymbol{\theta}}\right)\Delta\boldsymbol{\theta}\right] \\ &= -\frac{1}{2}\Delta\boldsymbol{\theta}^\top \mathbb{E}_{p_{\boldsymbol{\theta}}}\left[\nabla^2 \log p_{\boldsymbol{\theta}}\right]\Delta\boldsymbol{\theta}\end{aligned}$$

where we have used Eq. 8.27 to cancel the $\left(\nabla \log p_{\boldsymbol{\theta}}\right)^\top \Delta\boldsymbol{\theta}$ term and the quantity $\mathbb{E}_{p_{\boldsymbol{\theta}}}\left[-\nabla^2 \log p_{\boldsymbol{\theta}}\right]$ is the negative expected Hessian matrix of $\log p_{\boldsymbol{\theta}}$, which can be equivalently expressed in a form known as the Fisher information matrix (FIM)

$$\mathrm{FIM}: \mathbb{E}_{p_{\boldsymbol{\theta}}} \left(\nabla \log p_{\boldsymbol{\theta}}\right)^\top \left(\nabla \log p_{\boldsymbol{\theta}}\right). \tag{8.28}$$

To see this equivalence, note that, like in Eq. 8.27,

$$0 = \nabla^2_{\boldsymbol{\theta}} \sum_{\mathcal{X}} p_{\theta} = \sum_{\mathcal{X}} \nabla_{\boldsymbol{\theta}} \left(p_{\theta} \nabla_{\boldsymbol{\theta}} \log p_{\theta}\right) = \sum_{\mathcal{X}} p \left(\nabla_{\boldsymbol{\theta}} \log p_{\theta}\right)^\top \nabla_{\boldsymbol{\theta}} \log p_{\theta} + \sum_{\mathcal{X}} p \nabla^2_{\boldsymbol{\theta}} \log p_{\theta} \tag{8.29}$$

where the first term is the FIM and the second one is the expected second derivative.

Using the Taylor series expansion of $\log p_{\boldsymbol{\theta}+\Delta\boldsymbol{\theta}}$ in the log-likelihood objective given in Eq. 8.25 and expressing the constraint as a Lagrangian, we can express the natural gradient objective as:

$$L_N(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) = \mathbb{E}_{\hat{p}_{\text{data}}}[-\log p_{\boldsymbol{\theta}}] + \mathbb{E}_{\hat{p}_{\text{data}}}[-\nabla \log p_{\boldsymbol{\theta}}]^\top + \frac{\lambda}{2}\Delta\boldsymbol{\theta}^\top \mathbb{E}_{p_{\boldsymbol{\theta}}}\left[-\nabla^2 \log p_{\boldsymbol{\theta}}\right]\Delta\boldsymbol{\theta}. \tag{8.30}$$

As we did in deriving Newton's method, we can ask what value of $\Delta\boldsymbol{\theta}$ minimizes this objective function. For this purpose, we can solve for $\nabla_{\Delta\boldsymbol{\theta}} L_N(\boldsymbol{\theta}, \Delta\boldsymbol{\theta}) = 0$ and get the natural gradient update equation (with $\Delta\boldsymbol{\theta} = \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t$)

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \left(\mathbb{E}_{p_{\boldsymbol{\theta}}}\left[-\nabla^2 \log p_{\boldsymbol{\theta}}\right]\right)^{-1} \mathbb{E}_{\hat{p}_{\text{data}}}[-\nabla \log p_{\boldsymbol{\theta}}]\,. \tag{8.31}$$

Comparing Newton method's update (Eq. 8.14) to this one, we see that while Newton's method scaled the gradient by the inverse Hessian matrix, the method of natural gradient scales the gradient with the inverse of the Fisher information matrix.

It is worthwhile reflecting on the difference between the natural gradient and Newton's method. First, they are motivated from two quite different perspectives. Newton's method starts by making a quadratic approximation to the objective function and then attempts to jump directly to the minimum of this approximation. On the other hand, the natural gradient approach starts with the goal of finding the direction of steepest descent in the space of probability distributions. The close correspondence shown here has much more to do with the use of the Taylor series approximation, to 2nd-order, done here in an effort to turn the natural gradient into a practical algorithm.

The main difference between Newton's method and natural gradient is that the expectation of second derivatives (or of squared first derivatives) in the natural gradient is taken with respect to the model distribution $p_{\boldsymbol{\theta}}$, while the Hessian used in Newton's method is computed over the dataset (or perhaps a subsample of the dataset, i.e. a minibatch), which implies that the analogous expectation is taken with respect to the data distribution $\hat{p}_{\text{data}}$.

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 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, 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, the objective function most commonly used for sparse coding is not convex. However, we can divide the inputs to the training algorithm into two sets: the dictionary parameters and the code representations. 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.

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 $f(\boldsymbol{x}) = (x_1 - x_2)^2 + \alpha \left(x_1^2 + y_1^2 \right)$ where α is a positive constant. As α approaches 0, coordinate descent ceases to make any progress at all, while Newton's method could solve the problem in a single step.

8.7.2 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 this luxury. 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 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 backpropagation. This 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 computation 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 a lot, 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 backpropagation through the linear component of each layer—larger values in the matrix result in larger outputs of matrix multiplication. Too large of initial weights may, however, result in exploding values during forward propagation or backpropagation. 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 Sec. 7.7 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 in pre-specified ways.

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 $U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{n}})$, while Glorot and Bengio

(2010b) suggest using the *normalized initialization*

$$W_{i,j} \sim U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}).$$

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 non-linearities.

Saxe *et al.* (2013) recommend initializing to random orthogonal matrices, so that all singular values are 1. This initialization scheme is also motivated by a model of a deep network as a sequence of matrix multiplies without non-linearities.

In order to account for the non-linearity, Saxe *et al.* (2013) recommend rescaling all initial weights by a gain factor g. This can offset the effect of the non-linearities on the eigenvalues of the Jacobian, though the interaction between the non-linearities and the eigenvectors of the Jacobian remains poorly characterized and presumably cannot be accounted for by adjusting the gain. Increasing 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 the 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 backpropagation, 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 avoid the vanishing and exploding gradients problem altogether. Feedforward networks are qualitatively different from recurrent networks. Recurrent networks repeatedly use the same weight matrix for forward propagation and repeatedly use its transpose for backpropagation. If we use the same simplification to analyze recurrent nets as is commonly used to analyze feedforward nets, that is, if we assume that the recurrent net consists only of matrix multiplications composed together, then both forward and back-propagation behave very much like the power method, systematically driving the propagated values toward the principle singular vector of the weight matrix.

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 like $1/\sqrt{m}$ 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 m without making the magnitude of individual weight elements shrink with m. This helps to achieve more diversity among the units at initialization. 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 Chapter 11.2.2, 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.

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 of the book, 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).

- When one unit gates another unit (for example, the forget gate of an LSTM), we may want to set the bias of the gating unit to 1, in order to make the gate initially be open and avoid discarding the gradient through the unit that it gates (Jozefowicz *et al.*, 2015b).

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

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

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 zero, 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 tasks 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.

8.7.3 Greedy Supervised Pre-training

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 *pre-training*.

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.

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

In the original (Bengio *et al.*, 2007b) version of greedy supervised pre-training, each stage consists in a supervised learning training task involving only a subset of the layers in the final neural network. An example of greedy supervised pre-training is illustrated in Figure 8.10, in which each added hidden layer is pre-trained as part of a shallow supervised MLP, taking as input the output of the previously trained hidden layer. Instead of pre-training one layer at a time, Simonyan and Zisserman (2015) pretrain a deep convolutional network (11 weight layers) and then use the first 4 and last 3 layers from this network to initialize even deeper networks (with up to 19 layers of weights). The middle layers are then initialized randomly and all the layers of the very deep network and 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.

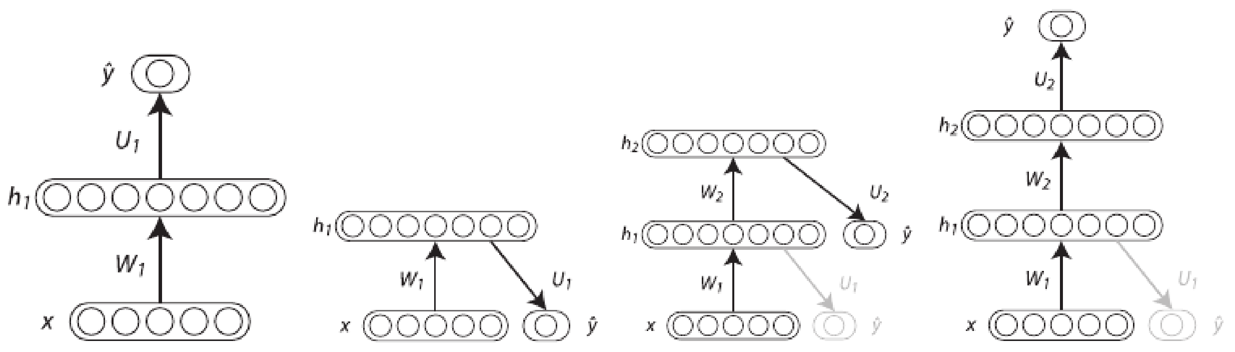

Figure 8.10: Illustration of one form of greedy supervised pre-training (Bengio *et al.*, 2007b). We start by training a sufficiently shallow architecture, like the first one on the left. For the sake of illustration, we redraw that architecture in the second figure from the left, because we are going to only keep the input-to-hidden layer and discard the hidden-to-output layer: we send the output of the first hidden layer as input to another supervised single hidden layer MLP that is trained with the same objective, thus adding a second hidden layer (third figure). This can be repeated for as many layers as desired. We can then redraw the figure (last on the right) to view it as an ordinary feedforward network. To further improve the optimization, jointly fine-tuning all the already pre-trained layers can be done, either only at the end or at each stage of this process.

Why would greedy supervised pre-training help? The hypothesis initially discussed by Bengio *et al.* (2007b) is that it *helps to provide better guidance to the intermediate levels of a deep hierarchy.* In general, pre-training may help both in terms of optimization and in terms of generalization.

An approach related to supervised pre-training extends the idea to the context of transfer learning: Yosinski *et al.* (2014) pre-train 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 towards a different set of tasks (another subset of the 1000 ImageNet object categories), with fewer training examples then for the first set of tasks. See Sec. 16.2 for a deeper treatment of transfer learning with neural networks.

8.7.4 Designing Models to Aid Optimization

Most of the model families we study for deep learning are incredibly broad. While most of these model families result in objective functions that are non-convex any time we include at least one hidden layer, there are many other difficulties for optimization besides just non-convexity.

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 and have significant slope. 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 remove 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 parameter 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), deeply-supervised nets (Lee *et al.*, 2014) and FitNets (Romero *et al.*, 2015). With GoogLeNet and deeply-supervised nets, these "auxiliary heads" are trained to perform the same task as the primary output as the top of the network in order to insure that the lower layers receive a large gradient. When training is complete the auxiliary heads may be discarded. This is an alternative to the pre-training 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 objective function and 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.

8.7.5 Continuation Methods and Curriculum Learning

Many optimization algorithms depend strongly on a good choice of initial parameters, for a variety of reasons. Most of our learning algorithms are based on making small, local moves. Sometimes 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 $\delta \ll \epsilon$. 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. This can happen even without local minima. Imagine a mountain standing in a plain, with a deep valley on the south side of the mountain. If we begin our descent from the north face of the mountain, we will begin by traveling north, rather than south, even though the only minimum lies to our south. After descending the mountain, we may become stuck on the plain. If the plain has sufficient slope, we will still waste a lot of time moving around the perimeter of the mountain before finally descending into the valley. 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.

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 $\{J^{(0)}, \ldots, J^{(n)}\}$. 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 a recent overview of recent literature, especially for AI applications in Mobahi and Fisher III (2015).

33Continuation 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

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

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. See Fig. 8.11 for an illustration. 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

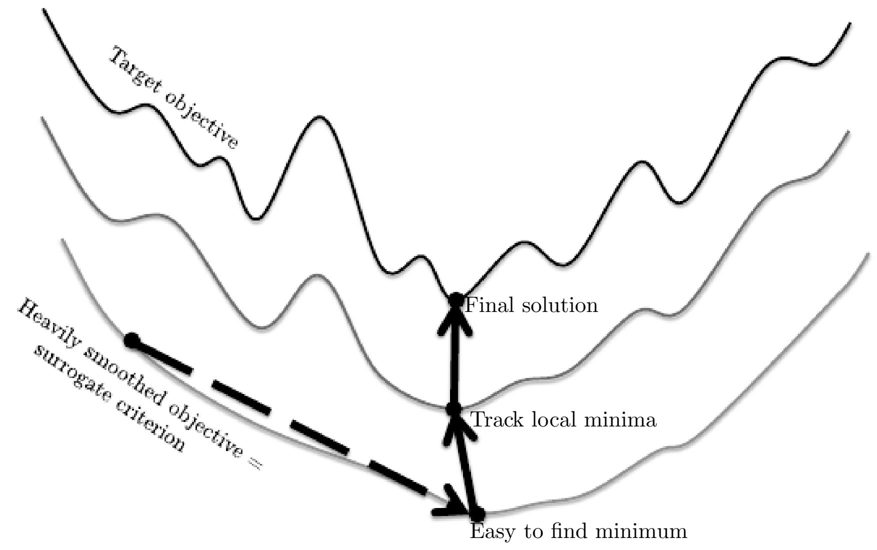

Figure 8.11: Continuation methods start by optimizing a smoothed version of the target objective function (possibly convex), then gradually reduce the amount of smoothing while tracking the local optimum. This approach tends to find better local minima than local descent on the target objective function starting from random initializations.

updates easier to compute or improve the correspondence between local update directions and progress toward a global solution.

Two fairly general approaches to global optimization are *continuation methods* (Wu, 1997), a deterministic approach and *simulated annealing* (Kirkpatrick *et al.*, 1983), a stochastic approach. They both proceed from the intuition that if we sufficiently blur a non-convex objective function (e.g. convolve it with a Gaussian) whose global minima are not at infinite values, then it becomes convex and finding the global optimum of that blurred objective function should be much easier. As illustrated in Figure 8.11, by gradually changing the objective function from a very blurred easy to optimize version to the original crisp and difficult objective function, we are actually likely to find better local minima. In the case of simulated annealing, the blurring occurs because of injecting noise. With injected noise, the state of the system can sometimes go uphill, and thus does not necessarily get stuck in a local minimum. With a lot of noise, the effective objective function (averaged over the noise) is flatter and convex, and if the amount of noise is reduced sufficiently slowly, then one can show convergence to the global minimum. However, the annealing schedule (the rate at which the noise level is decreased, or equivalently the temperature is decreased when you think of the physical annealing analogy) might need to be extremely slow, so an NP-hard optimization problem remains NP-hard.

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): it was 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. Instead, with a deterministic curriculum, no improvement over the baseline (ordinary training from the fully training set) was observed.