

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

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

8.1.3 Batch and Minibatch Algorithms

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

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

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

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

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

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

most commonly used property is the gradient:

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

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

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

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

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

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

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

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

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

Minibatch sizes are generally driven by the following factors:

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

Different kinds of algorithms use different kinds of information from the mini-batch 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 \mathbf{g} are usually relatively robust and can handle smaller batch sizes like 100. Second-order methods, which use also the Hessian matrix \mathbf{H} and compute updates such as $\mathbf{H}^{-1}\mathbf{g}$, typically require much larger batch sizes like 10,000. These large batch sizes are required to minimize fluctuations in the estimates of $\mathbf{H}^{-1}\mathbf{g}$. Suppose that \mathbf{H} is estimated perfectly but has a poor condition number. Multiplication by

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

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

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

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

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

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

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

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

with the exact gradient

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

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

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

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

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

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

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

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

8.2 Challenges in Neural Network Optimization

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

8.2.1 Ill-Conditioning

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

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

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

$$\frac{1}{2}\epsilon^2\mathbf{g}^\top \mathbf{H}\mathbf{g} - \epsilon\mathbf{g}^\top \mathbf{g} \tag{8.10}$$

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

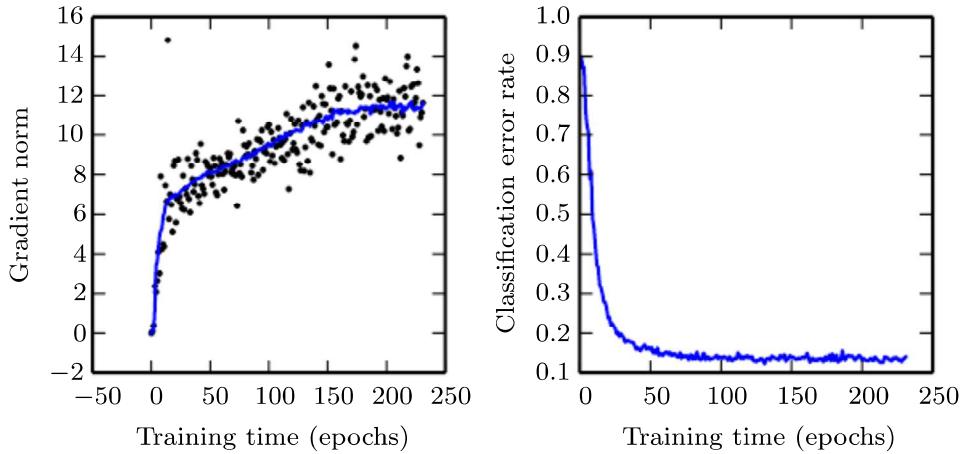


Figure 8.1: Gradient descent often does not arrive at a critical point of any kind. In this example, the gradient norm increases throughout training of a convolutional network used for object detection. (*Left*)A scatterplot showing how the norms of individual gradient evaluations are distributed over time. To improve legibility, only one gradient norm is plotted per epoch. The running average of all gradient norms is plotted as a solid curve. The gradient norm clearly increases over time, rather than decreasing as we would expect if the training process converged to a critical point. (*Right*)Despite the increasing gradient, the training process is reasonably successful. The validation set classification error decreases to a low level.

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

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

8.2.2 Local Minima

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

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

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

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

In addition to weight space symmetry, many kinds of neural networks have additional causes of non-identifiability. For example, in any rectified linear or maxout network, we can scale all of the incoming weights and biases of a unit by α if we also scale all of its outgoing weights by $\frac{1}{\alpha}$. This means that—if the cost function does not include terms such as weight decay that depend directly on the weights rather than the models’ outputs—every local minimum of a rectified linear or maxout network lies on an $(m \times n)$ -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 in comparison to the global minimum. One can construct small neural networks, even without hidden units, that have local minima with higher cost than the global minimum (Sontag and Sussman, 1989; Brady *et al.*, 1989; Gori and Tesi, 1992). If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms.

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

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

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

8.2.3 Plateaus, Saddle Points and Other Flat Regions

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

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

be heads. See [Dauphin et al. \(2014\)](#) for a review of the relevant theoretical work.

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

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

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

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

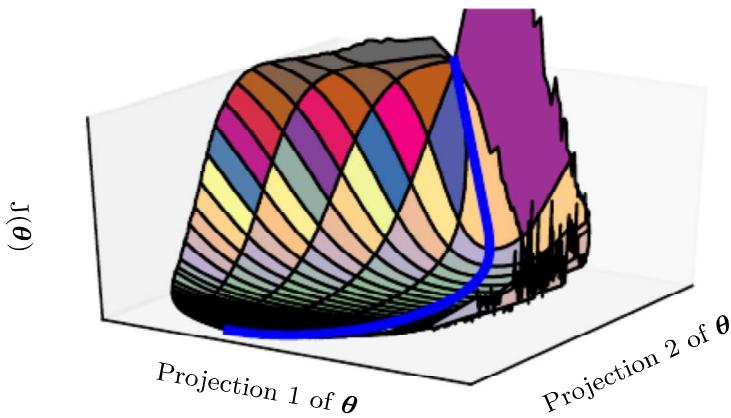


Figure 8.2: A visualization of the cost function of a neural network. Image adapted with permission from [Goodfellow et al. \(2015\)](#). These visualizations appear similar for feedforward neural networks, convolutional networks, and recurrent networks applied to real object recognition and natural language processing tasks. Surprisingly, these visualizations usually do not show many conspicuous obstacles. Prior to the success of stochastic gradient descent for training very large models beginning in roughly 2012, neural net cost function surfaces were generally believed to have much more non-convex structure than is revealed by these projections. The primary obstacle revealed by this projection is a saddle point of high cost near where the parameters are initialized, but, as indicated by the blue path, the SGD training trajectory escapes this saddle point readily. Most of training time is spent traversing the relatively flat valley of the cost function, which may be due to high noise in the gradient, poor conditioning of the Hessian matrix in this region, or simply the need to circumnavigate the tall “mountain” visible in the figure via an indirect arcing path.

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

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

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

8.2.4 Cliffs and Exploding Gradients

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

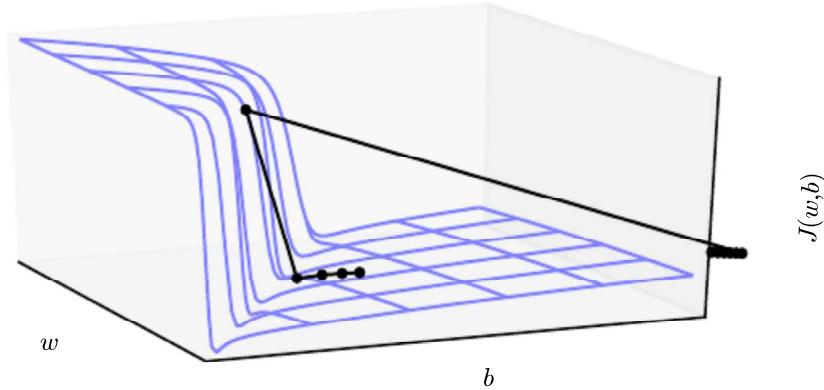


Figure 8.3: The objective function for highly nonlinear deep neural networks or for recurrent neural networks often contains sharp nonlinearities in parameter space resulting from the multiplication of several parameters. These nonlinearities give rise to very high derivatives in some places. When the parameters get close to such a cliff region, a gradient descent update can catapult the parameters very far, possibly losing most of the optimization work that had been done. Figure adapted with permission from [Pascanu et al. \(2013\)](#).

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

8.2.5 Long-Term Dependencies

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