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 $J(\boldsymbol{X}^{(\text{train})}, \boldsymbol{\theta})$ with respect to the model parameters $\boldsymbol{\theta}$.

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

TODO: be more clear about what p is here, and also emphasize it. Suppose that we have input feature $\boldsymbol{x}$, targets y, and 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*. If we knew the true distribution $p(\boldsymbol{x}, y)$, this would be an optimization task solveable 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*

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

where m is the number of training examples.

This process 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.

TODO– make sure 0-1 loss is defined and in the index

8.1.2 Surrogate Loss Functions

TODO– coordinate with regularization.tex, surrogate loss functions now appear to be first introduced in the context of early stopping

TODO– in some cases, surrogate loss function actually results in being able to learn more. for example, test 0-1 loss continues to decrease for a long time after train 0-1 loss has reached zero when training using log-likelihood surrogate

In some cases, using a surrogate loss function allows us to extract more information

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 a regularization method known as early stopping (see Sec. 7.7), 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:

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

TODO: after agreeing on probability distribution format, make this consistent with eq:max-log-lik

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

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{8.1}$$

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:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{\mathrm{data}}} \nabla_{\boldsymbol{\theta}} \log p_{\mathrm{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{8.2}$$

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.

TODO: make sure standard error of the mean is covered in statistical estimators section of chap 5

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* 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 Section 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, and some are more sensitive to sampling error than others. 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 datacenter, 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 Generalization and Early Stopping

In machine learning, typically we minimize a objective function defined as an expectation of some per-example loss across the training set:

$$\hat{J}(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{\text{data}}} L(\boldsymbol{x}, \boldsymbol{\theta}).$$

However, we would usually prefer to minimize the corresponding objective function where the expectation is taken across *the data generating distribution* rather than just the finite training set:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} L(\boldsymbol{x}, \boldsymbol{\theta}).$$

In other words, we care about generalization error, not training error.

Usually, we use an optimization algorithm based on minibatch estimates of the gradient. During the first stages of learning, this is equivalent to minimizing the generalization error directly. After we have used up the training data and begin to repeat minibatches, the two criteria are different.

This is the main way in which optimization for machine learning is actually different from traditional optimization, rather than just a special case of optimization. Many neural network optimization algorithms are implicitly designed in ways that are intended to yield better results in terms of generalization error, even if they perform worse as an optimization algorithm (yield worse training error or minimize the training error more slowly).

Some neural network algorithms actually change all of their updates to account for the uncertainty in the true generalization loss. For example, an algorithm called TONGA uses the covariance between gradients across different examples to approximately minimize the generalization loss (Le Roux *et al.*, 2008).

Most neural network training algorithms are actually designed to be traditional optimization algorithms that minimize the training loss, apart from one change: the convergence criterion.

8.2 Challenges in Optimization

TODO: write this section. make title more specific to neural nets?

8.2.1 Local Minima

TODO check whether this is already covered in numerical.tex

8.2.2 Ill-Conditioning

TODO this is definitely already covered in numerical.tex

8.2.3 Plateaus, Saddle Points, and Other Flat Regions

The long-held belief that neural networks are hopeless to train because they are fraught with local minima has been one of the reasons for the "neural networks winter" in the 1995-2005 decade. Indeed, one can show that there may be an exponentially large number of local minima, even in the simplest neural network optimization problems (Sontag and Sussman, 1989; Brady *et al.*, 1989; Gori and Tesi, 1992).

Theoretical work has shown that saddle points (and the flat regions surrounding them) are important barriers to training neural networks, and may be more important than local minima.

Explain (Dauphin *et al.*, 2014; Choromanska *et al.*, 2014)

8.2.4 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. (TODO: REFER TO A PLOT FROM THE ILL-CONDITIONING SECTION WITH COUNTOURS OF VALLEY). Second-order methods and momentum or gradient-averaging methods introduced in Section 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).

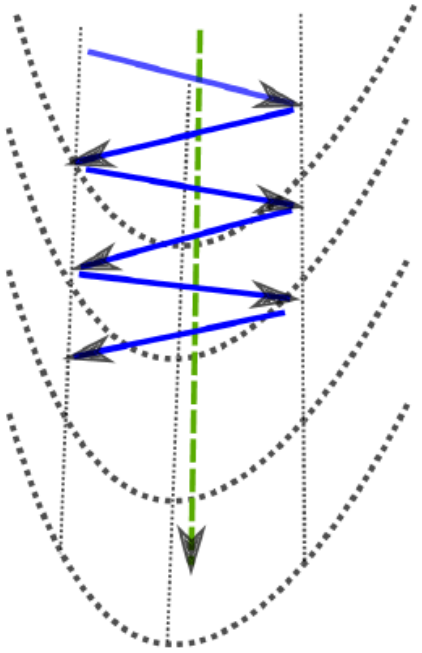

Figure 8.1: One theory about the neural network optimization is that poorly conditioned Hessian matrices cause much of the difficulty in training. In this view, some directions have a high curvature (second derivative), corresponding to the rising sides of the valley, and other directions have a low curvature, corresponding to the smooth slope of the valley. 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's most paying in the long run to go) and decreasing it in the directions of steep rise, which would otherwise lead to oscillations. The objective is to smoothly go down, staying at the bottom of the valley.

However, although classical second order methods can help, as shown in Figure 8.2, due to higher order derivatives, the objective function may have a lot more non-linearity, which 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.

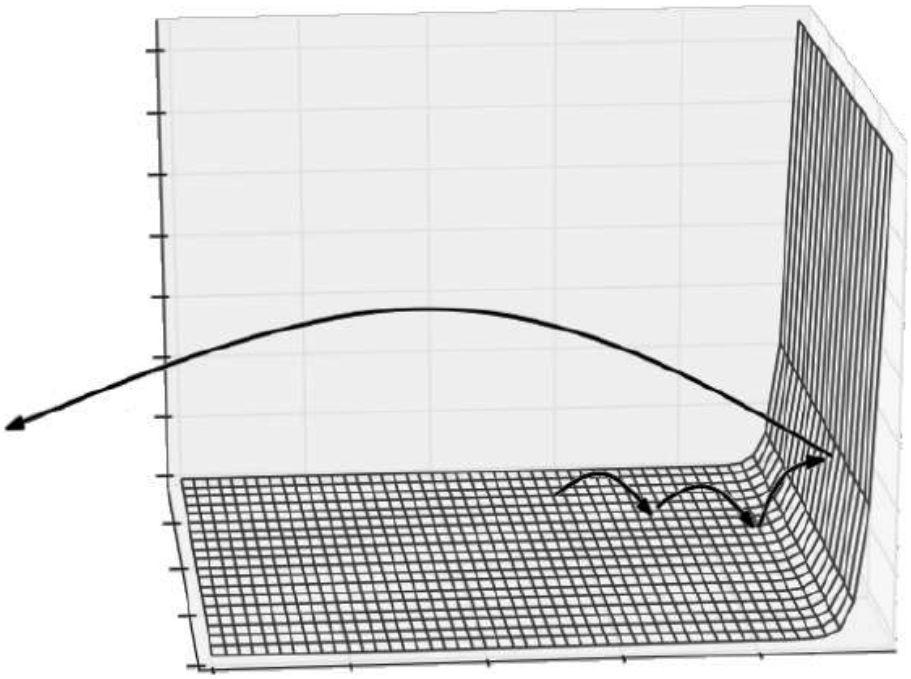

Figure 8.2: Contrary to what is shown in Figure 8.1, the objective function for highly non-linear deep neural networks or for recurrent neural networks is typically not made of symmetrical sides. As shown in the figure, there are sharp non-linearities that give rise to very high derivatives in some places. 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.

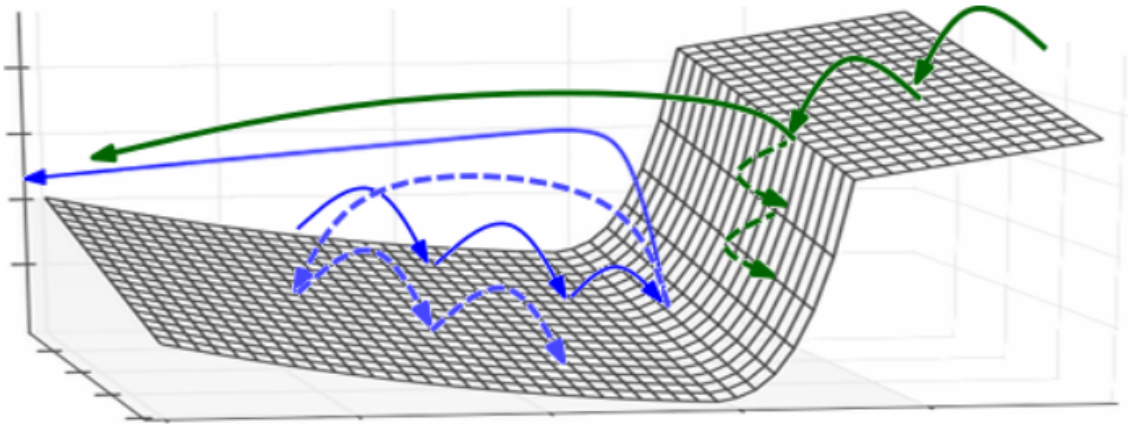

Figure 8.3: To address the presence of cliffs such as shown in Figure 8.2, a useful heuristic is to clip the magnitude of the gradient, only keeping its direction if its magnitude is above a threshold (which is a hyperparameter, although not a very critical one). This helps to avoid the destructive big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

The gradient clipping heuristics are described in more detail in Section 10.8.7. 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.

8.2.5 Vanishing and Exploding Gradients - An Introduction to the Issue of 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 Section 10.8 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 more ups and downs. TODO: the phrase "ups and downs" has a connotation of specifically good things and bad things happening over time, use a different phrase. this section is also sloppily conflating many different ideas, just having both areas of large derivatives and small derivatives does not mean there are lots of ups and downs, consider the function $(1 - x * y)^2$, which has small derivatives near the origin and the global minima, much larger derivatives in between This arises simply because the Jacobian (matrix of derivatives) of a composition is the product of the Jacobians of each stage, i.e., 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 x is the product

$$f' = f'_T f'_{T-1} \ldots, f'_2 f_1 \tag{8.3}$$

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 $\boldsymbol{a}_t = f_{t-1}(f_{t-2}(\ldots, f_2(f_1(\boldsymbol{x}))))$, i.e. composition has been replaced by matrix multiplication. This is illustrated in Figure 8.4. TODO: the above sentence is incredibly long, split it up and probably put the definitions in the opposite order. It also seems strange to say composition is "replaced" by matrix multiplication, more like composition in forward prop implies matrix multiplication in backprop

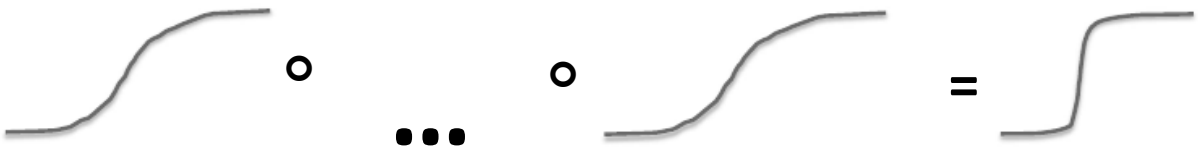

Figure 8.4: When composing many non-linearities (like the activation non-linearity in a deep or recurrent neural network), the result is highly non-linear, typically with most of the values associated with a tiny derivative, some values with a large derivative, and many ups and downs (not shown here).

TODO: can we avoid using capital letters for scalars here? (T) 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.

Consequence for Recurrent Networks: 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.

TODO: why the capital F? Can we use lowercase instead? This also appears in rnn.tex 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:

$$s_t = F_{\boldsymbol{\theta}}(s_{t-1}, x_t) \tag{8.4}$$

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,

$$o_t = g_\omega(s_t), \tag{8.5}$$

TODO: could we avoid the capital T? 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:

$$\begin{aligned}\frac{\partial L_T}{\partial \boldsymbol{\theta}} &= \sum_{t \le T} \frac{\partial L_T}{\partial s_t} \frac{\partial s_t}{\partial \boldsymbol{\theta}} \\ \frac{\partial L_T}{\partial \boldsymbol{\theta}} &= \sum_{t \le T} \frac{\partial L_T}{\partial s_T} \frac{\partial s_T}{\partial s_t} \frac{\partial F_{\boldsymbol{\theta}}(s_{t-1}, x_t)}{\partial \boldsymbol{\theta}}\end{aligned} \tag{8.6}$$

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 Section 6.4, 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:

$$\frac{\partial s_T}{\partial s_t} = \frac{\partial s_T}{\partial s_{T-1}} \frac{\partial s_{T-1}}{\partial s_{T-2}} \cdots \frac{\partial s_{t+1}}{\partial s_t} \tag{8.7}$$

which is of the same form as Eq. 8.3 discussed above, i.e., which tends to either vanish or explode.

As a consequence, we see from Eq. 8.6 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 vanilla recurrent net and stochastic gradient descent (Section 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). Section 10.8 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.

8.2.6 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, and we can only approximate its 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.

8.2.7 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, 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* in neural network papers because it updates the parameters only after having seen a batch of all the training examples. It involves updating the model parameters $\boldsymbol{\theta}$ (in the case of a deep neural network, these parameters would include the weights and biases associated with each layer) 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:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \epsilon \nabla_{\boldsymbol{\theta}} \sum_t L(f(\boldsymbol{x}^{(t)}; \boldsymbol{\theta}), \boldsymbol{y}^{(t)}; \boldsymbol{\theta}), \tag{8.8}$$

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. Of course, following the gradient in this way is only guaranteed to reduce the loss in the limit as $\epsilon \to 0$.

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 in $o(1/k)$ after k steps (Bertsekas, 2004). In addition, the number of training iteration to reach a particular error level is proportional to the ratio of the largest to the smallest eigenvalue of the Hessian matrix (second derivatives of the objective

function with respect to the parameters). Very slow convergence can thus occur when the Hessian is ill-conditioned.

There exists a learning rate value $\epsilon_{\max}$ such that if $\epsilon < \epsilon_{\max}$ and the objective function is locally convex (around the minimum to which we converge), gradient descent is guaranteed to converge to a minimum (local or global) However, 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 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 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 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, i.e., the gradient estimator is unbiased:

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

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

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

Require: Learning rate η.
Require: Initial parameter $\boldsymbol{\theta}$
  while Stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Set $\hat{\boldsymbol{g}} = \mathbf{0}$
    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

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 Section 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 convergence is that

$$\sum_{k=1}^{\infty} \eta_k = \infty, \quad \text{and}$$
$$\sum_{k=1}^{\infty} \eta_k^2 < \infty. \tag{8.9}$$

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

8.3.3 Online Gradient Descent Minimizes Generalization Error

TODO: move this subsection elsewhere (IG votes to merge it into Batch and Minibatch Algorithms and/or Generalization and Early Stopping) see e-mail thread "Feedback on recent commits"

Let us consider the sense of the term "online" referring to 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 which sees new a 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 of the current predictor $f(\cdot; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$ is

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

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. 8.1 and 8.2 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 (the usual scenario for many machine learning applications). 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.

8.3.4 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, but consistent across minibatches. 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.

Formally, we introduce a variable $\boldsymbol{v}$ that plays the role of velocity (or momen-

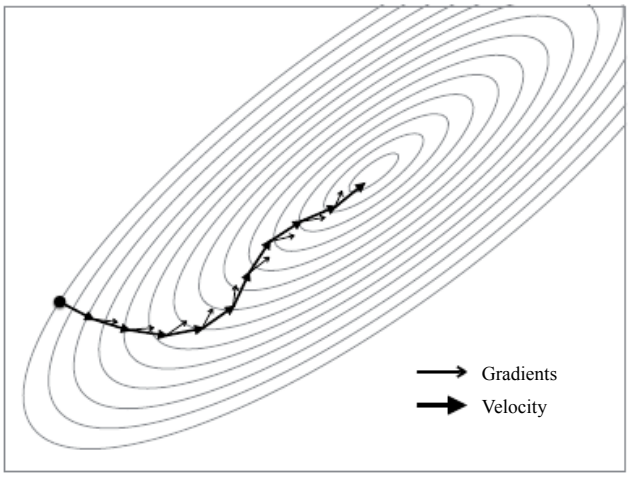

Figure 8.5: TODO: write this caption. TODO: resolve redundancy between this and fig:valley (right now the images are esentially the same thing)

tum) that accumulates gradient. The update rule is given by:

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

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

The velocity $\boldsymbol{v}$ accumulates the gradient elements $\nabla_{\boldsymbol{\theta}}\big(\frac{1}{n}\sum_{t=1}^{n} L(\boldsymbol{f}(\boldsymbol{x}^{(t)};\boldsymbol{\theta}), \boldsymbol{y}^{(t)})\big)$. 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 η, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while Stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \boldsymbol{0}$
    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

8.3.5 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.

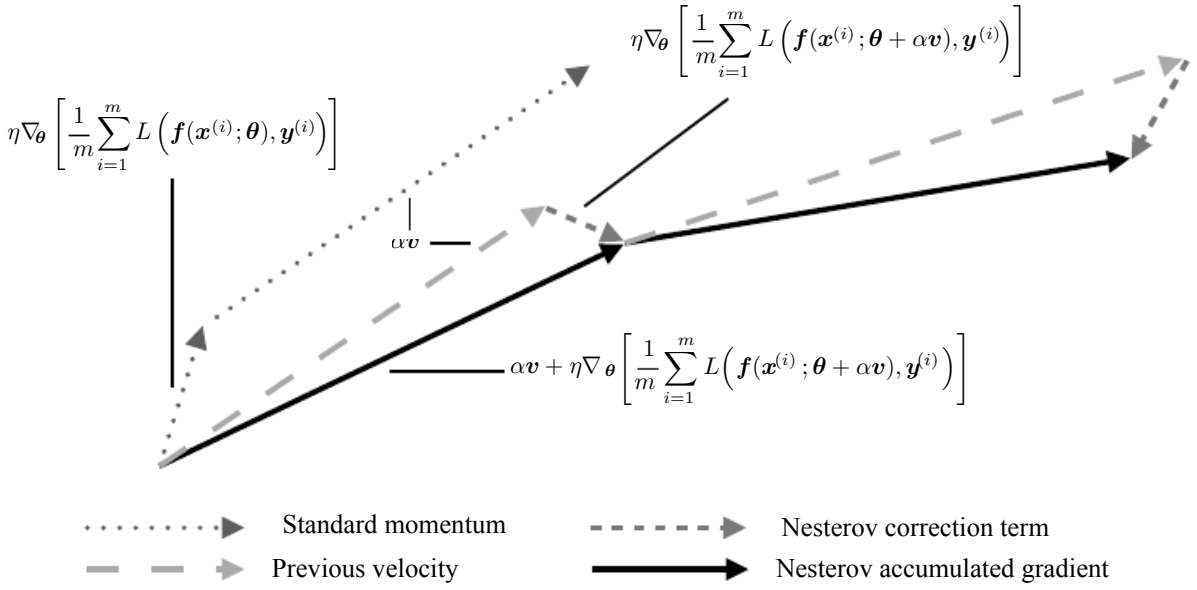

Figure 8.6: An illustration of the difference between Nesterov momentum and standard momemtum. Nesterov momentum incorporates the gradient after the velocity is already applied. This figure is derived from a similar image in Geoff Hinton's Coursera lectures.

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate η, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  while Stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    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

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 Sections 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 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.

Algorithm 8.4 The Adagrad algorithm

Require: Global learning rate η,
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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    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

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 effective 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.

Algorithm 8.5 The RMSprop algorithm

Require: Global learning rate η, decay rate ρ.
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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    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 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

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 estimate may have high bias early in training.

Algorithm 8.6 RMSprop algorithm with Nesterov momentum

Require: Global learning rate η, decay rate ρ, momentum para α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while Stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Compute interim update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    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 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

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 $\{\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\}$.[1]

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

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

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 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

[1] 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 8.7 The Adam algorithm

Require: Stepsize α
Require: Decay rates ρ_1 and ρ_2, constant ϵ
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 $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    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
    $t \leftarrow t + 1$
    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

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 8.8 The Adadelta algorithm

Require: Decay rate ρ, constant ϵ
Require: Initial parameter $\boldsymbol{\theta}$
  Initialize accumulation variables $\boldsymbol{r} = \mathbf{0}$, $\boldsymbol{s} = \mathbf{0}$,
  while Stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$.
    Set $\boldsymbol{g} = \mathbf{0}$
    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 update: $\Delta\boldsymbol{\theta} = -\frac{\sqrt{\boldsymbol{s} + \epsilon}}{\sqrt{\boldsymbol{r} + \epsilon}} \boldsymbol{g}$ $\quad$ % (operations applied element-wise)
    Accumulate update: $\boldsymbol{s} \leftarrow \rho \boldsymbol{s} + (1 - \rho) \left[\Delta\boldsymbol{\theta}\right]^2$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

8.5 Optimization Algorithms III: Approximate Second-Order Methods

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 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 $\boldsymbol{H}(f)(\boldsymbol{x})$ is defined such that

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i \, \partial x_j} f(\boldsymbol{x}).$$

Equivalently, the Hessian is the Jacobian of the gradient.

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

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_0) + (\boldsymbol{x} - \boldsymbol{x}_0)^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}_0) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_0)^\top \boldsymbol{H}(f)(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0).$$

If we then solve for the critical point of this function, we obtain:

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

When the function can be locally approximated as quadratic, iteratively updating the approximation and jumping to the minimum of the approximation can reach the critical point much faster than gradient descent would. This is a useful property near a local minimum, but it can be a harmful property near a saddle point. As discussed in Section 8.2.3, Newton's method is only appropriate when the nearby critical point is a minimum (all the eigenvalues of the Hessian are positive), whereas gradient descent can in principle escape a saddle point, although it may take a lot of time if the negative eigenvalues are very small in magnitude, producing a kind of plateau around the saddle point.

8.5.2 Conjugate Gradients

Algorithm 8.9 Conjugate gradient method

Require: Initial parameters $\boldsymbol{\theta}_0$
  Initialize $\boldsymbol{\rho}_0 = \mathbf{0}$
  while stopping criterion not met **do**
    Initialize the gradient $\boldsymbol{g} = \mathbf{0}$
    for $i = 1$ to n % loop over the training set. **do**
      Compute gradient: $\boldsymbol{g} \leftarrow \boldsymbol{g} + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    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: $\eta^* = \text{argmin}_\eta J(\boldsymbol{\theta}_t + \eta \boldsymbol{\rho}_t)$
    Apply update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta^* \boldsymbol{\rho}_t$
  end while

8.5.3 BFGS

8.6 Optimization Algorithms IV: Natural Gradient Methods

TODO brief descriptions of natural gradient methods.

8.7 Optimization Strategies and Meta-Algorithms

8.7.1 Batch Normalization

TODO: actually write this section

Algorithm 8.10 BFGS method

Require: Initial parameters $\boldsymbol{\theta}_0$
  Initialize inverse Hessian $\boldsymbol{M}_0 = \boldsymbol{I}$
  while stopping criterion not met **do**
    Compute gradient: $\boldsymbol{g}_t = \nabla J(\boldsymbol{\theta}_t)$ (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

Batch normalization (Ioffe and Szegedy, 2015) is one of the most exciting recent innovations in optimizing deep neural networks and it is actually not an optimization algorithm at all. Instead, it is a method of adaptive reparametrization, motivated by the observed difficulty in training very deep models (e.g. ¿10 layers). One major reason for this is that the distribution of the inputs to each layer changes throughout learning – as the parameters of all lower layers adapt.

8.7.2 Coordinate Descent

In some cases, it may be possible to solve an optimization problem quickly by breaking it into separate pieces. If we minimize $f(\boldsymbol{x})$ with respect to a single variable x_i, then minimize it with respect to another variable x_j and so on, 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.

8.7.3 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 diferent 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 Section 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 feed-forward 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.

8.7.4 Greedy Supervised Pre-training

TODO: write this section on greedy supervised pretraining

8.7.5 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.

KEY IDEA

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, 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 (TODO cite a skip connection paper). A related idea to skip connections is adding extra copies of the output that are attached to the intermediate hidden layers of the network (Szegedy *et al.*, 2014a) TODO also cite DSNs. 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.

8.8 Hints, Global Optimization and Curriculum Learning

Most of the work on numerical optimization for machine learning and deep learning in particular is focused on local descent, i.e., on how to locally improve the objective function efficiently. What the experiments with different initialization strategies tell us is that local descent methods can get stuck, presumably near a local minimum or near a saddle point, i.e., where gradients are small, so that the initial value of the parameters can matter a lot.

As an illustration of this issue, consider the experiments reported by Gülçehre and Bengio (2013), where a learning task is setup so that if the lower half of the deep supervised network is pre-trained with respect to an appropriate sub-task, the whole network can learn to solve the overall task, whereas random initialization almost always fails. In these experiments, we know that the overall task can be decomposed into two tasks (1) (identifying the presence of different objects in an input image and (2) verifying whether the different objects detected are of the same class or not. Each of these two tasks (object recognition, exclusive-or)

are known to be learnable, but when we compose them, it is much more difficult to optimize the neural network (including a large variety of architectures), while other methods such as SVMs, boosting and decision trees also fail. This is an instance where the optimization difficulty was solved by introducing prior knowledge in the form of *hints*, specifically *hints about what the intermediate layer in a deep net should be doing*. We have already seen in Section 8.7.4 that a useful strategy is to ask the hidden units to extract features that are useful to the supervised task at hand, with greedy supervised pre-training. In section 16.1 we will discuss an unsupervised version of this idea, where we ask the intermediate layers to extract features that are good explaining the variations in the input, without reference to a specific supervised task. Another related line of work is the *FitNets* (Romero *et al.*, 2015), where the middle layer of 5-layer supervised teacher network is used as a hint to be predicted by the middle layer of a much deeper student network (11 to 19 layers). In that case, additional parameters are introduced to regress the middle layer of the 5-layer teacher network from the middle layer of the deeper student network. The lower layers of the student networks thus get two objectives: help the outputs of the student network accomplish their task, as well as predict the intermediate layer of the teacher network. Although a deeper network is usually more difficult to optimize, it can generalize better (it has to extract these more abstract and non-linear features). Romero *et al.* (2015) were motivated by the fact that a deep student network with a smaller number of hidden units per layer can have a lot less parameters (and faster computation) than a fatter shallower network and yet achieve the same or better generalization, thus allowing a trade-off between better generalization (with 3 times fewer parameters) and faster test-time computation (up to 10 fold, in the paper, using a very thin and deep network with 35 times less parameters). Without the hints on the hidden layer, the student network performed very poorly in the experiments, both on the training and test set.

These drastic effects of initialization and hints to middle layers bring forth the question of what is sometimes called *global optimization* (Horst *et al.*, 2000), the main subject of this section. The objective of global optimization methods is to find better solutions than local descent minimizers, i.e., ideally find a global minimum of the objective function and not simply a local minimum. If one could restart a local optimization method from a very large number of initial conditions, one could imagine that the global minimum could be found, but there are more efficient approaches.

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

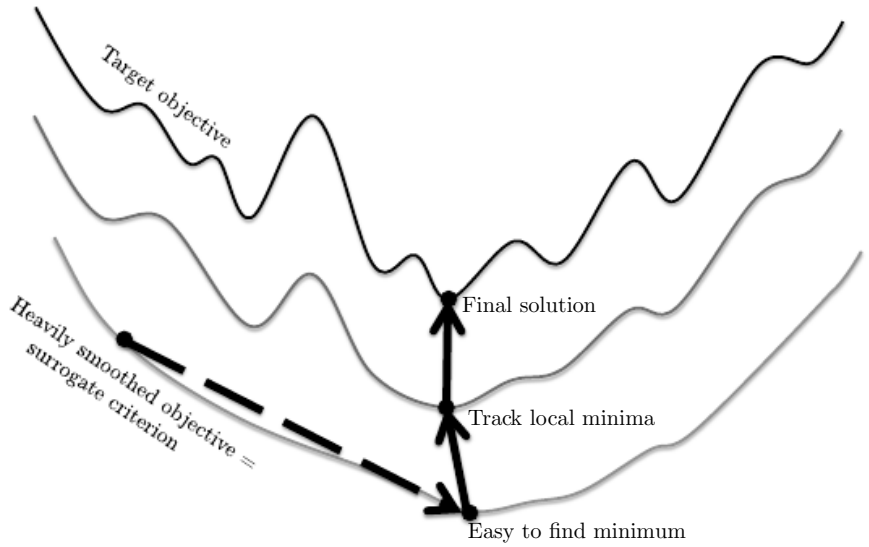

Figure 8.7: Optimization based on continuation methods: start by optimizing a smoothed out version of the target objective function (possibly convex), then gradually reduce the amount of smoothing while tracking the local optimum. This approach tends to find better local minima than a straight local descent approach on the target objective function. Curriculum learning (starting from easy examples and gradually introducing with higher probability more difficult examples) can be justified under that light (Bengio *et al.*, 2009).

Gaussian) whose global minima arae 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 8.7, 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.

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). Continuation methods define a *family of objective functions*, indexed by a single scalar index λ, with an easy to optimize objective function at one end (usually convex, say $\lambda = 1$) and the target objective at the other end (say

$\lambda = 0$). The idea is to first find the solution for the easy problem ($\lambda = 1$) and then gradually decrease λ towards the more difficult objectives, *while tracking the minimum*.

Curriculum learning (Bengio *et al.*, 2009) was introduced as a general strategy for machine learning that is inspired by how humans learn, starting by learning to solve simple tasks, and then exploiting what has been learned to learn slightly more difficult and abstract tasks, etc. It was justified as a continuation method (Bengio *et al.*, 2009) in the context of deep learning, where it was previously observed that the optimization problem can be challenging. Experiments showed that better results could be obtained by following a curriculum, in particular on a large-scale neural language modeling task. One view on curriculum learning introduced in that paper is that a particular intermediate objective function corresponds to a reweighing on the examples: initially the easy to learn examples are given more weights or a higher probability, and harder examples see their weight or probability gradually increased as the learner gets sufficiently ready to learn them. The idea of curriculum learning to help train difficult to optimize models has been taken up successfully not only in natural language tasks (Spitkovsky *et al.*, 2010; Collobert *et al.*, 2011a; Mikolov *et al.*, 2011b; Tu and Honavar, 2011) but also in computer vision (Kumar *et al.*, 2010; Lee and Grauman, 2011; Supancic and Ramanan, 2013). It was also found to be consistent with the way in which humans *teach* (Khan *et al.*, 2011): they start by showing easier and more prototypical examples and then help the learner refine the decision surface with the less obvious cases. In agreement with this, it was found that such strategies are *more effective* when teaching to humans (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.