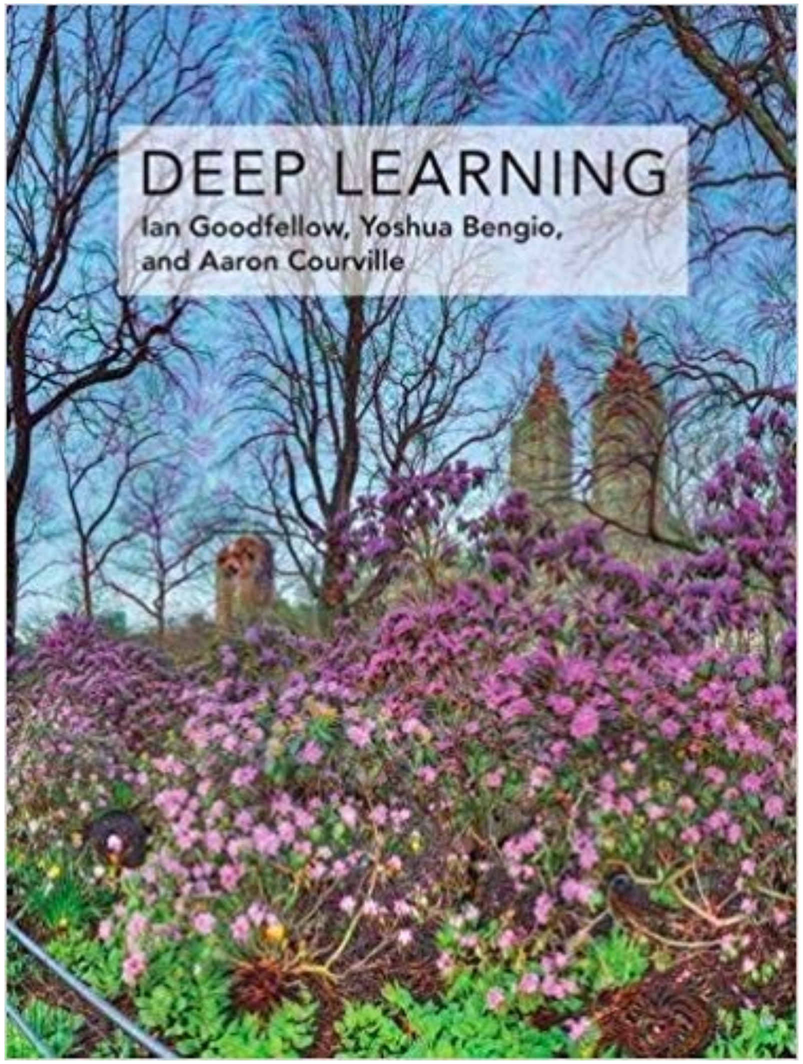
國立陽明交通大學
NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Deep Learning
深度學習
Fall 2023

Optimization for Training
(Chapter 8.1-8.2)

Prof. Chia-Han Lee
李佳翰 教授



- Figure source: Textbook and Internet
- You are encouraged to buy the textbook.
- Please respect the copyright of the textbook. Do not distribute the materials to other people.

Empirical risk minimization

- The goal of a machine learning algorithm is to reduce the expected generalization error given by

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

  This quantity is known as the risk. The expectation is taken over the true underlying distribution p_{data}, instead of empirical distribution, $\hat{p}_{\text{data}}$.

- If we knew the true distribution $p_{\text{data}}(\boldsymbol{x}, y)$, risk minimization would be an optimization task solvable by an optimization algorithm. When we do not know $p_{\text{data}}(\boldsymbol{x}, y)$ but only have a training set of samples, however, we have a machine learning problem.

Empirical risk minimization

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

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

  where m is the number of training examples.
- The training process based on minimizing this average training error is known as empirical risk minimization. Rather than optimizing the risk directly, we optimize the empirical risk and hope that the risk decreases significantly as well.

Empirical risk minimization

- Nonetheless, 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.

Surrogate loss functions and early stopping

- Sometimes, the loss function we actually care about (say, classification error) is not one that can be optimized efficiently. For example, exactly minimizing expected 0-1 loss is typically intractable (exponential in the input dimension), even for a linear classifier.

- In such situations, one typically optimizes a surrogate loss function instead, which acts as a proxy but has advantages. For example, the negative log-likelihood of the correct class is typically used as a surrogate for the 0-1 loss.

- The negative log-likelihood allows the model to estimate the conditional probability of the classes, and if the model can do that well, then it can pick the classes that yield the least expected classification error.

Surrogate loss functions and early stopping

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

- This is because even when the expected 0-1 loss is zero, one can improve the robustness of the classifier by further pushing the classes apart from each other, obtaining a more confident and reliable classifier, thus extracting more information from the training data than would have been possible by simply minimizing the average 0-1 loss on the training set.

Surrogate loss functions and early stopping

- The difference between optimization in general and optimization used for training algorithms is that training algorithms do not usually halt at a local minimum.

- Instead, a machine learning algorithm usually minimizes a surrogate loss function but halts when a convergence criterion based on early stopping is satisfied.

- Typically the early stopping criterion is based on the true underlying loss function, such as 0-1 loss measured on a validation set.

- Training often halts while the surrogate loss function still has large derivatives, which is very different from the pure optimization, where the algorithm is converged when the gradient becomes very small.

Batch and minibatch algorithms

- One aspect of machine learning algorithms that separates them from general optimization algorithms is that the objective function usually decomposes as a sum over the training examples.

- Optimization algorithms for machine learning typically compute each update to the parameters based on an expected value of the cost function estimated using only a subset of the terms of the full cost function.

Batch and minibatch algorithms

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

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

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

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

- Most of the properties of the objective function *J* used by most of our optimization algorithms are also expectations over the training set. For example, the most commonly used property is the gradient:

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

Batch and minibatch algorithms

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

- Recall that the standard error of the mean (equation 5.46) estimated from n samples is given by $\sigma/\sqrt{n}$, where σ is the true standard deviation of the value of the samples. The denominator of $\sqrt{n}$ shows that there are less than linear returns to using more examples to estimate the gradient.

Batch and minibatch algorithms

- 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.

Batch and minibatch algorithms

- Another consideration motivating statistical estimation of the gradient from a small number of samples is redundancy in the training set.

- In the worst case, all m samples in the training set could be identical copies of each other. A sampling-based estimate of the gradient could compute the correct gradient with a single sample, using m times less computation than the naive approach.

- In practice, we are unlikely to encounter this worst-case situation, but we may find large numbers of examples that all make very similar contributions to the gradient.

Batch and minibatch algorithms

- Optimization algorithms that use the entire training set are called batch or deterministic gradient methods, because they process all the training examples simultaneously in a large batch.

- This terminology can be somewhat confusing because the word "batch" is also often used to describe the minibatch used by minibatch stochastic gradient descent. Typically the term "batch gradient descent" implies the use of the full training set, while the use of the term "batch" to describe a group of examples does not. For example, it is common to use the term "batch size" to describe the size of a minibatch.

Batch and minibatch algorithms

- Optimization algorithms that use only a single example at a time are sometimes called stochastic and sometimes online methods.
- The term "online" is usually reserved for when the examples are drawn from a stream of continually created examples rather than from a fixed-size training set over which several passes are made.
- Most algorithms used for deep learning fall somewhere in between, using more than one but fewer than all the training examples. These were traditionally called minibatch or minibatch stochastic methods, and it is now common to call them simply stochastic methods.

Batch and minibatch algorithms

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.

Batch and minibatch algorithms

- Some kinds of hardware achieve better runtime with specific sizes of arrays. When using GPUs, it is common for power of 2 batch sizes to offer better runtime. Typical power of 2 batch sizes range from 32 to 256, with 16 sometimes being attempted for large models.

- Small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1.

- Training with small batch size might require a small learning rate to maintain stability because of the high variance in the estimate of the gradient. The total runtime can be very high as a result of the need to make more steps.

Batch and minibatch algorithms

- Different algorithms use different kinds of information from the mini-batch in various ways. Some algorithms 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, which also use the Hessian matrix $\boldsymbol{H}$ and compute updates such as $\boldsymbol{H}^{-1}\boldsymbol{g}$, typically require much larger batch sizes, like 10000, to minimize fluctuations in the estimates of $\boldsymbol{H}^{-1}\boldsymbol{g}$. Suppose that $\boldsymbol{H}$ is estimated perfectly but has a poor condition number. Multiplication by $\boldsymbol{H}$ or its inverse amplifies estimation errors in $\boldsymbol{g}$.

Batch and minibatch algorithms

- It is also crucial that the minibatches be selected randomly.
- Computing an unbiased estimate of the expected gradient from a set of samples requires that those samples be independent.
- We also wish for two subsequent gradient estimates to be independent from each other, so two subsequent minibatches of examples should also be independent from each other.

Batch and minibatch algorithms

- Many datasets are most naturally arranged in a way where successive examples are highly correlated. In cases where the order of the dataset holds some significance, it is necessary to shuffle the examples before selecting minibatches.

- For very large datasets, for example, datasets containing billions of examples in a data center, it can be impractical to sample examples truly uniformly at random every time we want to construct a minibatch.

- Fortunately, in practice it is usually sufficient to shuffle the order of the dataset once and then store it in shuffled fashion.

Batch and minibatch algorithms

- Many optimization problems in machine learning decompose over examples well enough that we can compute entire separate updates over different examples in parallel. 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 other minibatches.

- An interesting motivation for minibatch stochastic gradient descent is that it follows the gradient of the true generalization error as long as no examples are repeated.

Batch and minibatch algorithms

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

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

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

Batch and minibatch algorithms

- Of course, this interpretation applies only when examples are not reused.
- Nonetheless, it is usually best to make several passes through the training set, unless the training set is extremely large.
- When multiple such epochs are used, only the first epoch follows the unbiased gradient of the generalization error, but of course, the additional epochs usually provide enough benefit due to decreased training error to offset the increased gap between training error and test error.

Batch and minibatch algorithms

- With some datasets growing rapidly in size, faster than computing power, it is becoming more common for machine learning applications to use each training example only once or even to make an incomplete pass through the training set.

- When using an extremely large training set, overfitting is not an issue, so underfitting and computational efficiency become the predominant concerns.

Challenges in neural network optimization

- Optimization in general is an extremely difficult task.
- Traditionally, machine learning has avoided the difficulty of general optimization by carefully designing the objective function and constraints to ensure that the optimization problem is convex.
- When training neural networks, we must confront the general nonconvex case.

Ill-conditioning

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

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

Ill-conditioning

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

$$\frac{1}{2}\epsilon^2 \boldsymbol{g}^\top \boldsymbol{H}\boldsymbol{g} - \epsilon \boldsymbol{g}^\top \boldsymbol{g} \qquad (8.10)$$

  to the cost.

- Ill-conditioning of the gradient becomes a problem when $\frac{1}{2}\epsilon^2 \boldsymbol{g}^{\mathrm{T}} \boldsymbol{H}\boldsymbol{g}$ exceeds $\epsilon \boldsymbol{g}^{\mathrm{T}} \boldsymbol{g}$.

- In many cases, the gradient norm $\boldsymbol{g}^{\mathrm{T}} \boldsymbol{g}$ does not shrink significantly throughout learning, but the $\boldsymbol{g}^{\mathrm{T}} \boldsymbol{H}\boldsymbol{g}$ term grows by more than an order of magnitude.

Ill-conditioning

- The result is that learning becomes very slow despite the presence of a strong gradient because the learning rate must be shrunk to compensate for even stronger curvature.

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

Local minima

- One of the most prominent features of a convex optimization problem is that it can be reduced to the problem of finding a local minimum. Any local minimum is guaranteed to be a global minimum. Some convex functions have a flat region at the bottom rather than a single global minimum point, but any point within such a flat region is an acceptable solution.

- With nonconvex functions, such as neural nets, it is possible to have many local minima. Indeed, nearly any deep model is essentially guaranteed to have an extremely large number of local minima.

- As we will see, however, this is not necessarily a major problem.

Local minima

- Neural networks and any models with multiple equivalently parametrized latent variables have multiple local minima because of the model identifiability problem.

- A model is said to be identifiable if a sufficiently large training set can rule out all but one setting of the model's parameters. Models with latent variables are often not identifiable because we can obtain equivalent models by exchanging latent variables with each other.

- We could modify layer 1 of a neural network by swapping the incoming weight vector for unit i with the incoming weight vector for unit j, then do the same for the outgoing weight vectors. If we have m layers with n units each, then there are $n!^m$ ways of arranging the hidden units. This is known as weight space symmetry.

Local minima

- In addition to weight space symmetry, many kinds of neural networks have additional causes of nonidentifiability.

- For example, in any rectified linear or maxout network, we can scale all the incoming weights and biases of a unit by α if we also scale all its outgoing weights by $\frac{1}{\alpha}$.

- This means that—if the cost function does not include terms such as weight decay that depend directly on the weights rather than the models' outputs—every local minimum of a rectified linear or maxout network lies on an $(m \times n)$-dimensional hyperbola of equivalent local minima.

Local minima

- These model identifiability issues mean that a neural network cost function can have an extremely large or even uncountably infinite amount of local minima.

- However, all these local minima arising from nonidentifiability are equivalent to each other in cost function value. As a result, these local minima are not a problematic form of nonconvexity.

- Local minima can be problematic if they have high cost in comparison to the global minimum. If local minima with high cost are common, this could pose a serious problem for gradient-based optimization algorithms.

Local minima

- For many years, some practitioners believed that local minima were a common problem plaguing neural network optimization. Today, that does not appear to be the case.

- Experts now suspect that, for sufficiently large neural networks, most local minima have a low cost function value, and that it is not important to find a true global minimum rather than to find a point in parameter space that has low but not minimal cost.

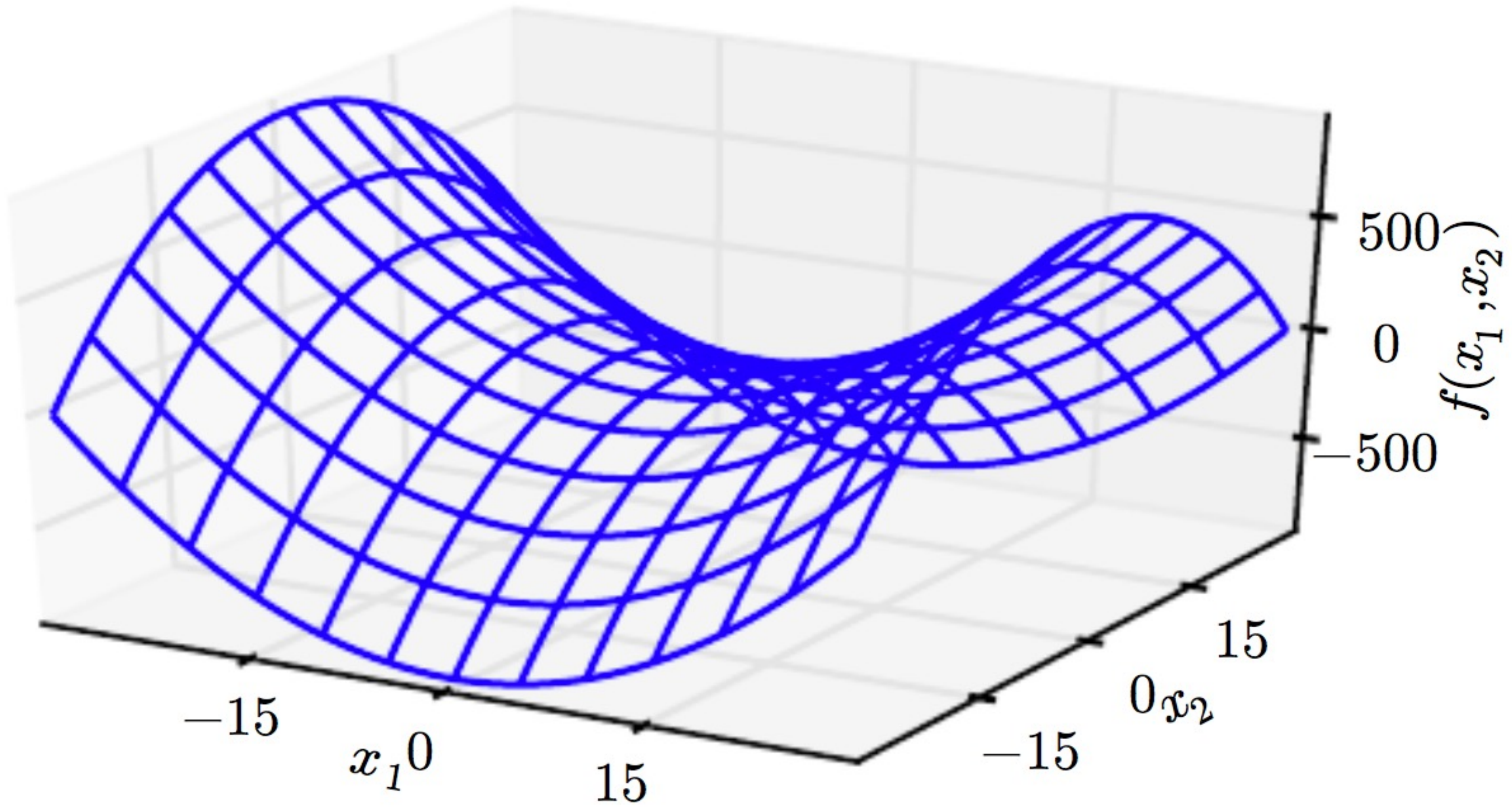
Plateaus, saddle points and other flat regions

- For many high-dimensional nonconvex functions, local minima (and maxima) are in fact rare compared to another kind of point with zero gradient: a saddle point.

- At a saddle point, the Hessian matrix has both positive and negative eigenvalues. Points lying along eigenvectors associated with positive eigenvalues have greater cost than the saddle point, while points lying along negative eigenvalues have lower value.

- We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section.

Plateaus, saddle points and other flat regions



Plateaus, saddle points and other flat regions

- Many classes of random functions exhibit the following behavior: in low-dimensional spaces, local minima are common. In higher-dimensional spaces, local minima are rare, and saddle points are more common.

- For a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ of this type,. the expected ratio of the number of saddle points to local minima grows exponentially with n

- To understand the intuition behind this behavior, observe that the Hessian matrix at a local minimum has only positive eigenvalues. The Hessian matrix at a saddle point has a mixture of positive and negative eigenvalues. In n-dimensional space, it is exponentially unlikely that all eigenvalues are positive or negative.

Plateaus, saddle points and other flat regions

- An amazing property of many random functions is that the eigenvalues of the Hessian become more likely to be positive as we reach regions of lower cost. It also means that local minima are much more likely to have low cost than high cost.
- Critical points with high cost are far more likely to be saddle points.
- Critical points with extremely high cost are more likely to be local maxima.

Plateaus, saddle points and other flat regions

- What are the implications of the proliferation of saddle points for training algorithms? For algorithms that use only gradient information, the situation is unclear. The gradient can often become very small near a saddle point. On the other hand, gradient descent empirically seems able to escape saddle points in many cases.

- For Newton's method, saddle points clearly constitute a problem. Newton's method is designed to solve for a point where the gradient is zero. Without appropriate modification, it can jump to a saddle point.

- This explains why second-order methods have not succeeded in replacing gradient descent for neural network training in high-dimensional spaces.

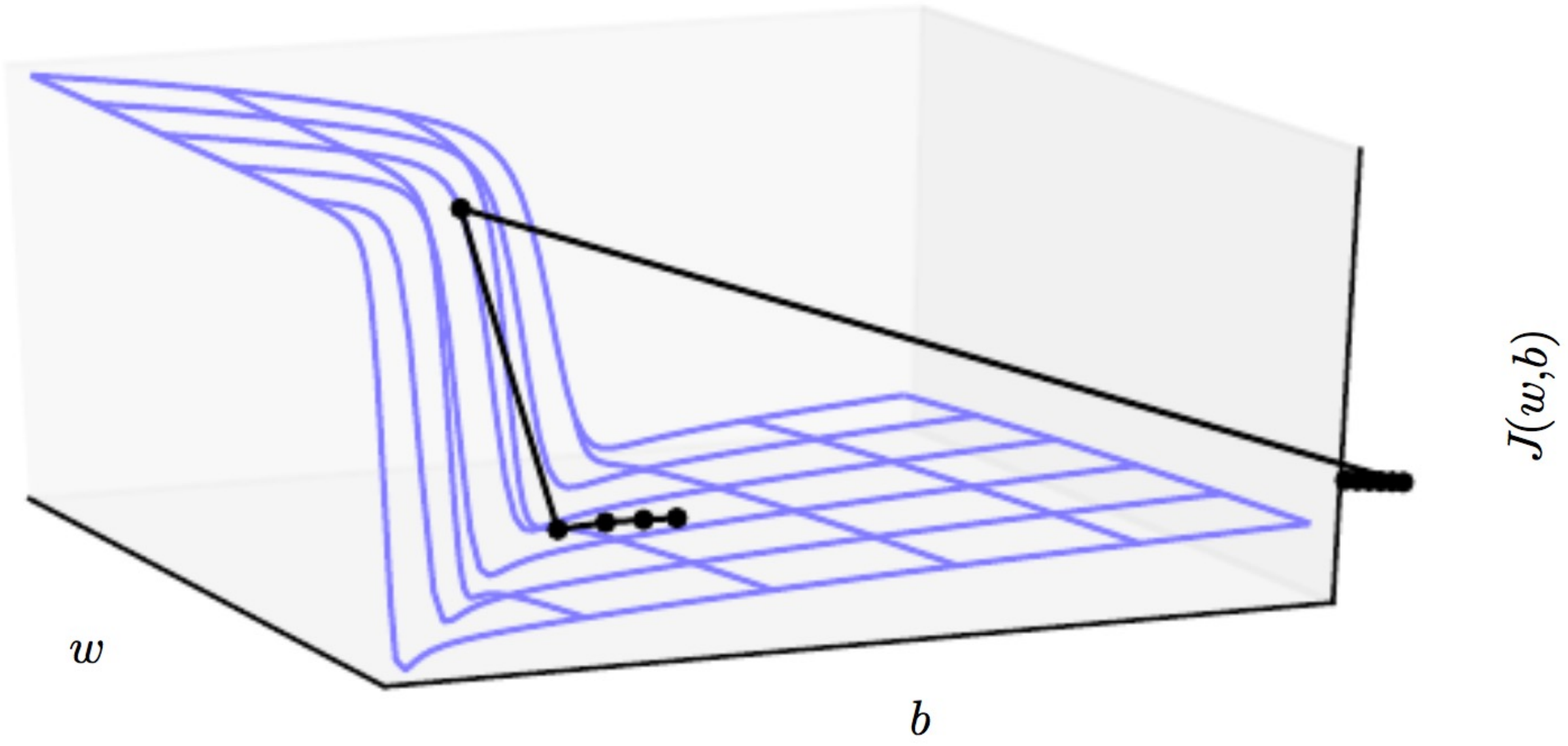
Cliffs and exploding gradients

- Neural networks with many layers often have extremely steep regions resembling cliffs. These result from the multiplication of several large weights together.

- On the face of an extremely steep cliff structure, the gradient update step can move the parameters extremely far, usually jumping off the cliff structure altogether.

- The cliff can be dangerous whether we approach it from above or from below, but fortunately its most serious consequences can be avoided using the gradient clipping heuristic.

Cliffs and exploding gradients



Cliffs and exploding gradients

- When the traditional gradient descent algorithm proposes making a very large step, the gradient clipping heuristic intervenes to reduce the step size, making it less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.

- Cliff structures are most common in the cost functions for recurrent neural networks, because such models involve a multiplication of many factors, with one factor for each time step. Long temporal sequences thus incur an extreme amount of multiplication.

Long-term dependencies

- Another difficulty that neural network optimization algorithms must overcome arises when the computational graph becomes extremely deep.

- Feedforward networks with many layers have such deep computational graphs. So do recurrent networks, in which repeated application of the same parameters gives rise to especially pronounced difficulties.

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

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

Long-term dependencies

- Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude or vanish if they are less than 1 in magnitude. The vanishing and exploding gradient problem refers to the fact that gradients through such a graph are also scaled according to $\mathrm{diag}(\lambda)^t$.

- Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function, while exploding gradients can make learning unstable.

Long-term dependencies

- The cliff structures described earlier that motivate gradient clipping are an example of the exploding gradient phenomenon.

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

Inexact gradients

- Most optimization algorithms are designed with the assumption that we have access to the exact gradient or Hessian matrix. In practice, we usually have only a noisy or even biased estimate of these quantities.

- In other cases, the objective function we want to minimize is actually intractable. Typically its gradient is intractable as well. In such cases we can only approximate the gradient.

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

Poor correspondence between local/global structure

- Many of the problems we have discussed so far correspond to properties of the loss function at a single point.
- It is possible to overcome all these problems at a single point and still perform poorly if the direction that results in the most improvement locally does not point toward distant regions of much lower cost.
- It is argued that much of the runtime of training is due to the length of the trajectory needed to arrive at the solution.

Poor correspondence between local/global structure

- Much of research into the difficulties of optimization has focused on whether training arrives at a global minimum, a local minimum, or a saddle point, but in practice, neural networks do not arrive at a critical point of any kind.

- Neural networks often do not arrive at a region of small gradient. Indeed, such critical points do not even necessarily exist.

- For example, the loss function $-\log p(y|\boldsymbol{x};\boldsymbol{\theta})$ can lack a global minimum point and instead asymptotically approach some value as the model becomes more confident.

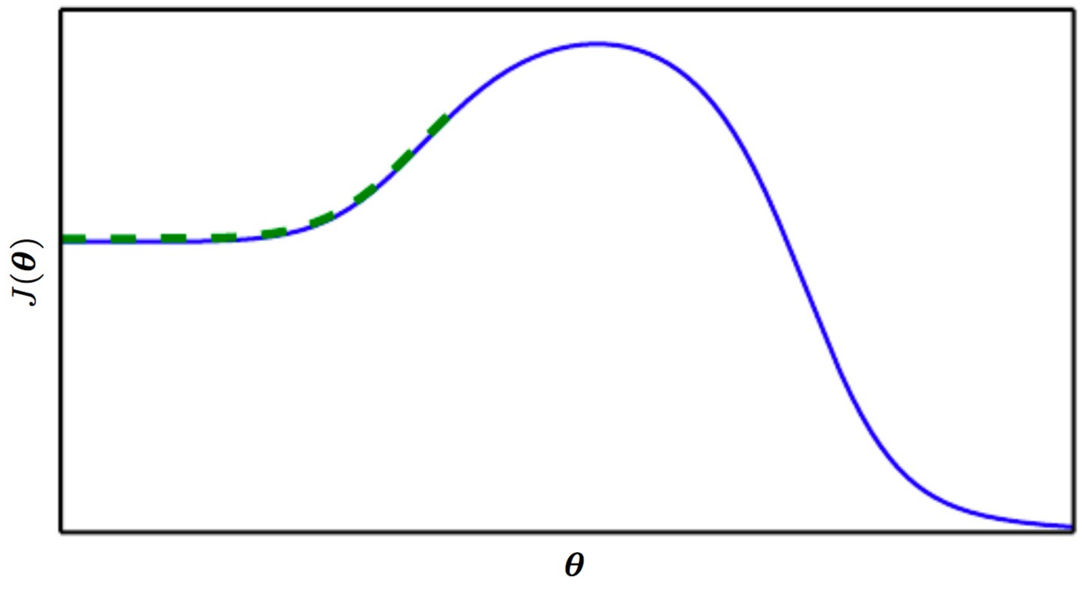
Poor correspondence between local/global structure

- For a classifier with discrete y and $p(y|\boldsymbol{x})$ provided by a softmax, the negative log-likelihood can become arbitrarily close to zero if the model is able to correctly classify every example in the training set, but it is impossible to actually reach the value of zero.

- Likewise, a model of real values $p(y|\boldsymbol{x}) = \mathcal{N}(y; f(\boldsymbol{\theta}), \beta^{-1})$ can have negative log-likelihood that asymptotes to negative infinity—if $f(\boldsymbol{\theta})$ is able to correctly predict the value of all training set y targets, the learning algorithm will increase β without bound.

Poor correspondence between local/global structure

An example of a failure of local optimization to find a good cost function value even in the absence of any local minima or saddle points. The main cause of difficulty in this case is being initialized on the wrong side of the "mountain" and not being able to traverse it. In higher-dimensional space, learning algorithms can often circumnavigate such mountains, but the trajectory associated with doing so may be long and result in excessive training time.



Poor correspondence between local/global structure

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

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

Poor correspondence between local/global structure

- Local descent may or may not define a reasonably short path to a valid solution, but we are not actually able to follow the local descent path. We may be able to compute some properties of the objective function, such as its gradient, only approximately, with bias or variance in our estimate of the correct direction.

- The objective function may have issues, such as poor conditioning or discontinuous gradients, causing the region where the gradient provides a good model of the objective function to be very small. Local descent with steps of size ϵ may define a reasonably short path to the solution, but we are only able to compute the local descent direction with steps of size $\delta \ll \epsilon$. Local descent may define a path to the solution, but the path contains many steps, so it incurs a high computation.

Poor correspondence between local/global structure

- Sometimes local information provides us no guide, such as when the function has a wide flat region, or if we manage to land exactly on a critical point (such as when using Newton's method). In these cases, local descent does not define a path to a solution at all.

- In other cases, local moves can be too greedy and lead us along a path that moves downhill but away from any solution, or along an unnecessarily long trajectory to the solution.

Poor correspondence between local/global structure

- Currently, we do not understand which of these problems are most relevant to making neural network optimization difficult, and this is an active area of research.

- Regardless of which of these problems are most significant, all of them might be avoided if there exists a region of space connected reasonably directly to a solution by a path that local descent can follow, and if we are able to initialize learning within that well-behaved region.

- This last view suggests research into choosing good initial points for traditional optimization algorithms to use.

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. Typically these results have little bearing on the use of neural networks in practice.

- Some theoretical results apply only when the units of a neural network output discrete values. 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.

Theoretical limits of optimization

- Other results show that finding a solution for a network of a given size is intractable, but in practice we can find a solution easily by using a larger network for which many more parameter settings correspond to an acceptable solution. Moreover, in the context of neural network training, we usually do not care about finding the exact minimum of a function, but seek only to reduce its value sufficiently to obtain good generalization error.

- Theoretical analysis of whether an optimization algorithm can accomplish this goal is extremely difficult. Developing more realistic bounds on the performance of optimization algorithms therefore remains an important goal for machine learning research.