Lecture 8: Optimization

Roger Grosse

1 Introduction

Now that we've seen how to compute derivatives of the cost function with respect to model parameters, what do we do with those derivatives? In this lecture, we're going to take a step back and look at optimization problems more generally. We've briefly discussed gradient descent and used it to train some models, but what exactly is the gradient, and why is it a good idea to move opposite it? We also introduce stochastic gradient descent, a way of obtaining noisy gradient estimates from a small subset of the data.

Using modern neural network libraries, it is easy to implement the backprop algorithm so that it correctly computes the gradient. It's not always so easy to get it to work well. In this lecture, we'll make a list of things that can go drastically wrong in neural net training, and talk about how we can spot them. This includes: learning rates that are too large or too small, symmetries, dead or saturated units, and badly conditioned curvature. We discuss tricks to ameliorate all of these problems. In general, debugging a learning algorithm is like debugging any other complex piece of software: if something goes wrong, you need to make hypotheses about what might have happened, and look for evidence or design experiments to test those hypotheses. This requires a thorough understanding of the principles of optimization.

Understanding the principles of neural nets and being able to diagnose failure modes are what distinguishes someone who's finished CSC321 from someone who's merely worked through the TensorFlow tutorial.

Our style of thinking in this lecture will be very different from that in the last several lectures. When we discussed backprop, we looked at the gradient computations *algebraically*: we derived mathematical equations for computing all the derivatives. We also looked at the computations *implementationally*, seeing how to implement them efficiently (e.g. by vectorizing the computations), and designing an automatic differentiation system which separated the backprop algorithm itself from the design of a network architecture. In this lecture, we'll look at gradient descent *geometrically*: we'll reason qualitatively about optimization problems and about the behavior of gradient descent, without thinking about how the gradients are actually computed. I.e., we *abstract away* the gradient computation. One of the most important skills to develop as a computer scientist is the ability to move between different levels of abstraction, and to figure out which level is most appropriate for the problem at hand.

1.1 Learning goals

- Be able to interpret visualizations of cross-sections of an error surface.
- Know what the gradient is and how to draw it geometrically.

- Know why stochastic gradient descent can be faster than batch gradient descent, and understand the tradeoffs in choosing the mini-batch size.
- Know what effect the learning rate has on the training process. Why can it be advantageous to decay the learning rate over time?
- Be aware of various potential failure modes of gradient descent. How might you diagnose each one, and how would you solve the problem if it occurs?
  - slow progress
  - instability
  - fluctuations
  - dead or saturated units
  - symmetries
  - badly conditioned curvature
- Understand why momentum can be advantageous.

2 Visualizing gradient descent

When we train a neural network, we're trying to minimize some cost function $\mathcal{E}$, which is a function of the network's parameters, which we'll denote with the vector $\boldsymbol{\theta}$. In general, $\boldsymbol{\theta}$ would contain all of the network's weights and biases, and perhaps a few other parameters, but for the most part, we're not going to think about what the elements of $\boldsymbol{\theta}$ represent in this lecture. We're going to think about optimization problems in the abstract. In general, the cost function will be the sum of losses over the training examples; it may also include a regularization term (which we'll discuss in the next lecture). But for the most part, we're not going to think about the particulars of the cost function.

In order to think qualitatively about optimization problems, we'll need some ways to visualize them. Suppose $\boldsymbol{\theta}$ consists of two weights, w_1 and w_2. One way to visualize $\mathcal{E}$ is to draw the **cost surface**, as in Figure 1(a); this is an example of a **surface plot**. This particular cost function has two **local minima**, or points which minimize the cost within a small neighborhood. One of these local optima is also a **global optimum**, a point which achieves the minimum cost over all values of $\boldsymbol{\theta}$. In the context of optimization, local and global minima are also referred to as **local and global optima**.

A function can have multiple global minima if there are multiple points that achieve the minimum cost. Technically speaking, global optima are also local optima, but informally when we refer to "local optima," we usually mean the ones which aren't global optima.

Surface plots can be hard to interpret, so we're only going to use them when we absolutely have to. Instead, we'll primarily rely on two other visualizations. First, suppose we have a one-dimensional optimization problem, i.e. $\boldsymbol{\theta}$ consists of a single weight w. We can visualize this by plotting $\mathcal{E}$ as a function of w, as in Figure 1(b). This figure also shows the gradient descent **iterates** (i.e. the points the algorithm visits) starting from two different initializations. One of these sequences converges to the global optimum, and the other one converges to the other local optimum. In general, gradient descent greedily tries to move downhill; by historical accident, it is

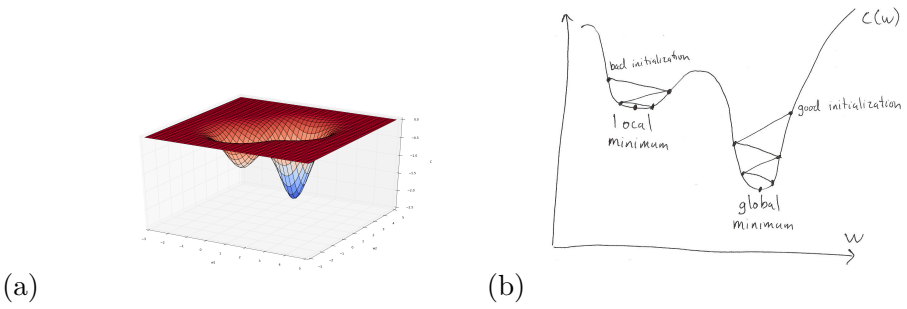


Figure 1: **(a)** Cost surface for an optimization problem with two local minima, one of which is the global minimum. **(b)** Cartoon plot of a one-dimensional optimization problem, and the gradient descent iterates starting from two different initializations, in two different basins of attraction.

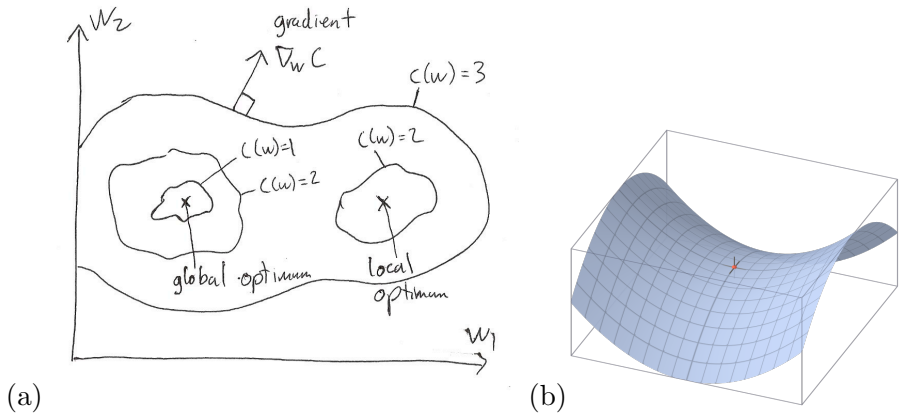


Figure 2: **(a)** Contour plot of a cost function. **(b)** A saddle point.

referred to as a **hill-climbing algorithm**. It's important to choose a good initialization, because we'd like to converge to the global optimum, or at least a good local optimum. The set of weights which lead to a given local optimum are known as a **basin of attraction**.

Figure 1(a) also shows a different feature of the cost function, known as a **plateau** (plural = plateaux). This is a region where the function is flat, or nearly flat, i.e. the derivative is zero or very close to zero. Gradient descent can perform very badly on plateaux, because the parameters change very slowly. In neural net training, plateaux are generally a bigger problem than local optima: while most local optima tend to be good enough in practice, plateaux can cause the training to get stuck on a very bad solution.

Remember when we observed that the gradient of 0–1 loss is zero almost everywhere? That's an example of a plateau.

Figure 2(a) shows a different visualization of a two-dimensional optimization problem: a **contour plot**. Here, the axes correspond to w_1 and w_2; this means we're visualizing weight space (just like we did in our lecture on linear classifiers). Each of the contours represents a **level set**, or set of parameters where the cost takes a particular value.

Check your understanding: how can you (approximately) see local optima on a countour plot? How do you tell which one is the global optimum?

One of the most important things we can visualize on a contour plot is the **gradient**, or the direction of **steepest ascent**, of the cost function,

denoted $\nabla_{\boldsymbol{\theta}} \mathcal{E}$. This is the direction which goes directly uphill, i.e. the direction which increases the cost the fastest relative to the distance moved. We can't determine the magnitude of the gradient from the contour plot, but it is easy to determine its direction: *the gradient is always orthogonal (perpendicular) to the level sets*. This gives an easy way to draw it on a contour plot (e.g. see Figure 2(a)). Algebraically, the gradient is simply the vector of partial derivatives of the cost function:

$$\nabla_{\boldsymbol{\theta}} \mathcal{E} = \frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = \begin{pmatrix} \partial \mathcal{E} / \partial \theta_1 \\ \vdots \\ \partial \mathcal{E} / \partial \theta_M \end{pmatrix} \tag{1}$$

In this context, $\mathcal{E}$ is taken as a function of the parameters, not of the loss $\mathcal{L}$. Therefore, the partial derivatives correspond to the values $\overline{w_{ij}}$, $\overline{b_i}$, etc., computed from backpropagation.

The fact that the vector of partial derivatives gives the steepest ascent direction is far from obvious; you would see the derivation in a multivariable calculus class, but here we will take it for granted.

The **gradient descent** update rule (which we've already seen multiple times) can be written in terms of the gradient:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{E}, \tag{2}$$

where α is the scalar-valued **learning rate**. This shows directly that gradient descent moves opposite the gradient, or in the direction of **steepest descent**. Too large a learning rate can cause instability, whereas too small a learning rate can cause slow progress. In general, the learning rate is one of the most important hyperparameters of a learning algorithm, so it's very important to **tune** it, i.e. look for a good value. (Most commonly, one tries a bunch of values and picks the one which works the best.)

Recall that hyperparameters are parameters which aren't part of the model and which aren't tuned with gradient descent.

For completeness, it's worth mentioning one more possible feature of a cost function, namely a **saddle point**, shown in Figure 2(b). This is a point where the gradient is zero, but which isn't a local minimum because the cost increases in some directions and decreases in others. If we're exactly on a saddle point, gradient descent won't go anywhere because the gradient is zero.

3 Stochastic gradient descent

In machine learning, our cost function generally consists of the average of costs for individual training examples. By linearity of derivatives, the gradient is the average of the gradients for individual examples:

$$\mathcal{E} = \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}_n \tag{3}$$

$$= \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(y^{(n)}, \hat{y}^{(n)}) \tag{4}$$

$$\nabla_{\boldsymbol{\theta}} \mathcal{E} = \nabla_{\boldsymbol{\theta}} \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}_n \tag{5}$$

$$= \frac{1}{N} \sum_{n=1}^{N} \nabla_{\boldsymbol{\theta}} \mathcal{E}_n \tag{6}$$

If we use this formula directly, we must visit every training example to compute the gradient. This is known as **batch training**, since we're treating the entire training set as a batch. But this can be very time-consuming, and it's also unnecessary: we can get a stochastic estimate of the gradient from a single training example. In **stochastic gradient descent (SGD)**, we pick a training example, and update the parameters opposite the gradient for that example:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{E}_n. \tag{7}$$

This is identical to the gradient descent update rule, except that $\mathcal{E}$ is replaced with $\mathcal{E}_n$.

SGD is able to make a lot of progress even before the whole training set has been visited. A lot of datasets are so large that it can take hours or longer to make a single pass over the training set; in such cases, batch training is impractical, and we need to use a stochastic algorithm.

In practice, we don't compute the gradient on a single example, but rather average it over a batch of B training examples known as a **mini-batch**. Typical mini-batch sizes are on the order of 100. Why mini-batches? Observe that the number of operations required to compute the gradient for a mini-batch is *linear* in the size of the mini-batch (since mathematically, the gradient for each training example is a separate computation). Therefore, if all operations were equally expensive, one would always prefer to use $B = 1$. In practice, there are two important reasons to use $B > 1$:

- Operations on mini-batches can be vectorized by writing them in terms of matrix operations. This reduces the interpreter overhead, and makes use of efficient and carefully tuned linear algebra libraries.

In previous lectures, we already derived vectorized forms of batch gradient descent. The same formulas can be applied in mini-batch mode.

- Most large neural networks are trained on GPUs or some other architecture which enables a high degree of parallelism. There is much more parallelism to exploit when B is large, since the gradients can be computed independently for each training example.

On the flip side, we don't want to make B too large, because then it takes too long to compute the gradients. In the extreme case where $B = N$, we get batch gradient descent. (The activations for large mini-batches may also be too large to store in memory.)

4 Problems, diagnostics, and workarounds

Now we get to the most important part of this lecture: debugging gradient descent training. When you first learned to program, whenever something didn't work, you might have looked through your code line by line to try and spot the mistake. This might have worked for 10-line programs, but it probably became unworkable for more complex programs. Line-by-line inspection doesn't work very well in machine learning either — not just because the programs are complicated, but also because most of the problems we're going to talk about can occur *even for a correctly implemented training algorithm.* E.g., if the problem is that you set the learning rate too small, you're not going to be able to deduce this by looking at your code, since you don't know ahead of time what the right learning rate is.

Let's make a list of various things that can go wrong, and how to diagnose and fix them.

4.1 Incorrect gradient computations

If your computed gradients are wrong, then all bets are off. If you're lucky, the training will fail completely, and you'll notice that something is wrong. If you're unlucky, it will sort of work, but it will also somehow be broken. This is much more common than you might expect: it's not unusual for an incorrectly implemented learning algorithm to perform reasonably well. But it will perform a bit worse than it should; furthermore, it will make it harder to tune, since some of the diagnostics might give misleading results if the gradients are wrong. Therefore, *it's completely useless to do anything else until you're sure the gradients are correct.*

Fortunately, it's possible to be confident in the correctness of the gradients. We've already covered finite difference methods, which are pretty reliable (see the lecture "Training a Classifier"). If you're using one of the major neural net frameworks, you're pretty safe, because the gradients are being computed automatically by a system which has been thoroughly tested. For the rest of this discussion, we'll assume the gradient computation is correctly implemented.

4.2 Local optima

We're trying to minimize the cost function, and one of the ways we can fail to do this is if we get stuck in a local optimum. Actually, that formulation isn't quite precise, since we rarely converge exactly to any optimum (local or global) when training neural nets. A more precise statement would be, we might wind up in a bad basin of attraction, and therefore not achieve as low a cost as we would be able to in the best basin of attraction.

In general, it's very hard to diagnose if you're in a bad basin of attraction. In many areas of machine learning, one tries to ameliorate the issue using **random restarts**: initialize the training from several random locations, run the training procedure from each one, and pick whichever result has the lowest cost. This is sometimes done in neural net training, but more often we just ignore the problem. In practice, the local optima are usually fine, so we think about training in terms of converging faster to a local optimum, rather than finding the global optimum.

4.3 Symmetries

Suppose we initialize all the weights and biases of a neural network to zero. All the hidden activations will be identical, and you can check by inspection (see the lecture on backprop) that all the weights feeding into a given hidden unit will have identical derivatives. Therefore, these weights will have identical values in the next step, and so on. With nothing to distinguish different hidden units, no learning will occur. This phenomenon is perhaps the most important example of a saddle point in neural net training.

Fortunately, the problem is easy to deal with, using any sort of **symmetry breaking**. Once two hidden units compute slightly different things, they will probably get a gradient signal driving them even farther apart. (Think of this in terms of the saddle point picture; if you're exactly on the saddle point, you get zero gradient, but if you're slightly to one side,

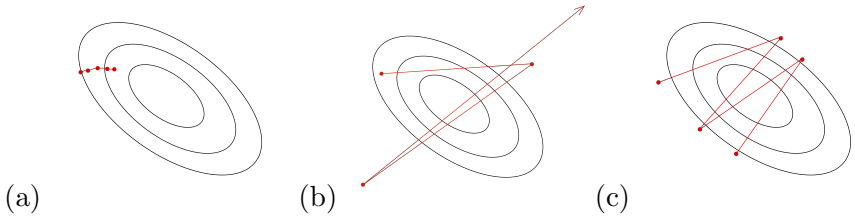


Figure 3: **(a)** Slow progress due to a small learning rate. **(b)** Instability due to a large learning rate. **(c)** Oscillations due to a large learning rate.

you'll move away from it, which gives you a larger gradient, and so on.) In practice, we typically initialize all the weights randomly.

4.4 Slow progress

If the learning rate is too small, gradient descent makes very slow progress, as shown in Figure 3(a). When you plot the training curve, this may show up as a cost which decreases very slowly, but at an approximately linear rate. If you see this happening, then try increasing the learning rate.

4.5 Instability and oscillations

Conversely, if the learning rate is too large, the gradient descent step will overshoot. In some cases, it will overshoot so much that the gradient gets larger, a situation known as **instability**. If this repeats itself, the parameter values and gradient can quickly blow up; this is visualized in Figure 3(b). In the training curve, the cost may appear to suddenly shoot up to infinity. If this is happening, you should decrease the learning rate.

If the learning rate is too large, yet not enough to cause instability, you might get **oscillations**, as shown in Figure 3(c). While the phenomenon might seem easy to spot based on this picture, it's actually pretty hard in practice — keep in mind that weight space is very high-dimensional, and it might not be obvious in which direction to look for oscillations. Also note that oscillations in weight space don't necessarily lead to oscillations in the training curve.

Since we can't detect oscillations, we simply try to tune the learning rate, finding the best value we can. Typically, we do this using a grid search over values spaced approximately by factors of 3, i.e. $\{0.3, 0.1, 0.03, \ldots, 0.0001\}$. The learning rate is one of the most important parameters, and one of the hardest to choose a good value for a priori, so it is usually worth tuning it carefully.

As it happens, there's one more idea which can dampen oscillations while also speeding up training: **momentum**. The physical intuition is as follows: the parameter vector $\boldsymbol{\theta}$ is treated as a particle which is moving through a field whose potential energy function is the cost $\mathcal{E}$. The gradient does not determine the velocity of the particle (as it would in SGD), but rather the acceleration. As a rough intuition, imagine you've built a surface in 3-D corresponding to a 2-D cost function, and you start a frictionless ball rolling from somewhere on that surface. If the surface is sufficiently flat,

the dynamics are essentially those described above. (The potential energy is the height of the surface.)

We can simulate these dynamics with the following update rule, known as **gradient descent with momentum**. (Momentum can be used with either the batch version or with SGD.)

$$\mathbf{p} \leftarrow \mu \mathbf{p} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{E}_n \tag{8}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{p} \tag{9}$$

Just as with ordinary SGD, there is a learning rate α. There is also another parameter μ, called the **momentum parameter**, satisfying $0 \leq \mu \leq 1$. It determines the timescale on which momentum decays. In terms of the physical analogy, it determines the amount of friction (with $\mu = 1$ being frictionless). As usual, it's useful to think about the edge cases:

- $\mu = 0$ yields standard gradient descent.
- $\mu = 1$ is frictionless, so momentum never decays. This is problematic because of conservation of energy. We would like to minimize the cost function, but whenever the particle gets near the optimum, it has low potential energy, and hence high kinetic energy, so it doesn't stay there very long. We need $\mu < 1$ in order for the energy to decay.

In practice, $\mu = 0.9$ is a reasonable value. Momentum sometimes helps a lot, and it hardly ever hurts, so using momentum is standard practice.

4.6 Fluctuations

All of the problems we've discussed so far occur both in batch training and in SGD. But in SGD, we have the further problem that the gradients are stochastic; even if they point in the right direction on average, individual stochastic gradients are noisy and may even increase the cost function. The effect of this noise is to push the parameters in a random direction, causing them to **fluctuate**. Note the difference between oscillations and fluctuations: oscillations are a systematic effect caused by the cost surface itself, whereas fluctuations are an effect of the stochasticity in the gradients.

Fluctuations often show up as fluctuations in the cost function, and can be seen in the training curves. One solution to fluctuations is to decrease the learning rate; however, this can slow down the progress too much. It's actually fine to have fluctuations during training, since the parameters are still moving in the right direction "on average."

A better approach to deal with fluctuations is **learning rate decay**. My favorite approach is to keep the learning rate relatively high throughout training, but then at the very end, to decay it using an exponential schedule, i.e.

$$\alpha_t = \alpha_0 e^{-t/\tau}, \tag{10}$$

where α_0 is the initial learning rate, t is the iteration count, τ is the **decay timescale**, and $t = 0$ corresponds to the start of the decay.

I should emphasize that we *don't begin the decay until late in training*, when the parameters are already pretty good "on average" and we merely have a high cost because of fluctuations. Once you start decaying α, progress

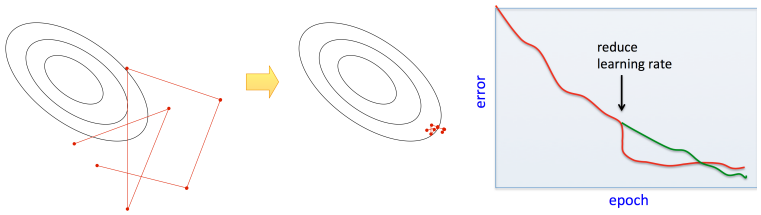


Figure 4: If you decay the learning rate too soon, you'll get a sudden drop in the loss as a result of reducing fluctuations, but the algorithm will stop making progress towards the optimum, leading to slower convergence in the long run. This is a big problem in practice, and we haven't figured out any good ways to detect if this is happening.

slows down drastically. If you decay α too early, you may get a sudden improvement in the cost from reducing fluctuations, at the cost of failure to converge in the long term. This phenomenon is illustrated in Figure 4.

Another neat trick for dealing with fluctuations is **iterate averaging**. Separate from the training process, we keep an **exponential moving average** $\tilde{\boldsymbol{\theta}}$ of the iterates, as follows:

$$\tilde{\boldsymbol{\theta}} \leftarrow \left(1 - \frac{1}{\tau}\right) \tilde{\boldsymbol{\theta}} + \frac{1}{\tau} \boldsymbol{\theta}. \tag{11}$$

τ is a hyperparameter called the **timescale**. Iterate averaging doesn't change the training algorithm itself at all, but when we apply or evaluate the network, we use $\tilde{\boldsymbol{\theta}}$ rather than $\boldsymbol{\theta}$. In practice, iterate averaging can give a huge performance boost by reducing the fluctuations.

4.7 Dead and saturated units

Another tricky problem is that of **saturated units**, i.e. units whose activations are nearly always near the ends of their dynamic range (i.e. the range of possible values). An important special case is that of **dead units**, units whose activations are always very close to zero. To understand why saturated units are problematic, we need to revisit one of the equations we derived for backprop. Suppose $h_i = \phi(z_i)$, where ϕ is a sigmoidal nonlinearity (such as the logistic function). Then:

$$\overline{z_i} = \overline{h_i} \frac{\mathrm{d}h_i}{\mathrm{d}zi} = \overline{h_i}\, \phi'(z_i). \tag{12}$$

If h is near the edge of its dynamic range, then $\phi'(z)$ is very small. (Think about why this is the case.) Therefore, $\overline{z}$ is also very small, and no gradient signal will pass through this node in the computation graph. In particular, all the weights that feed into z_i will get no gradient signal:

$$\overline{w_{ij}} = \overline{z_i}\, x_j \approx 0 \tag{13}$$

$$\overline{b_i} = \overline{z_i} \approx 0. \tag{14}$$

If the incoming weights and bias don't change, then this unit can stay saturated for a long time. In terms of our visualizations, this situation corresponds to a plateau.

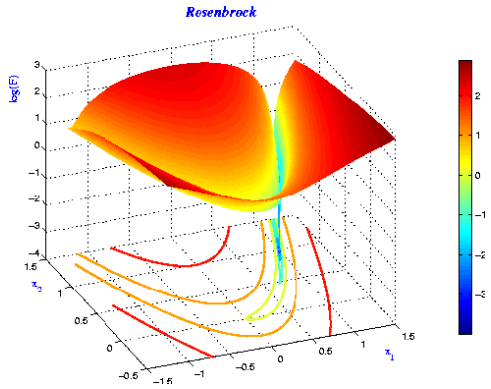


Figure 5: The Rosenbrock function, a function which is commonly used as an optimization benchmark and demonstrates badly conditioned curvature (i.e. a ravine).

Diagnosing saturated units is simple: just look at a histogram of the average activations, and make sure they're not concentrated at the endpoints.

Preventing saturated units is pretty hard, but there are some tricks that help. One trick is to carefully choose the scale of the random initialization of the weights so that the activations are in the middle of their dynamic range. One such trick is the "Xavier initialization", named after one of its inventors[1].

Another way to avoid saturation is to use an activation function which doesn't saturate. Linear activation functions would fit the bill, but unfortunately we saw that deep linear networks aren't any more powerful than shallow ones. Instead, consider **rectified linear units (ReLUs)**, which have the activation function

$$\phi(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0. \end{cases} \tag{15}$$

ReLU units don't saturate for positive z, which is convenient. Unfortunately, they can die if z is consistently negative, so it helps to initialize the biases to a small positive value (such as 0.1).

4.8 Badly conditioned curvature

All of the problems we've discussed so far are fairly specific things that can be attenuated using simple tricks. But there's one more problem that's fundamentally very hard to deal with: badly conditioned curvature. Let's unpack this. Intuitively, **curvature** refers to how fast the function curves upwards when you move in a given direction. In directions of high curvature, you want to take a small step, because you can overshoot very quickly.

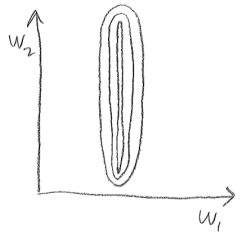
In directions of low curvature, you want to take a large step, because there's a long distance you need to travel. But what actually happens in gradient descent is precisely the opposite: it likes to take large steps in high curvature directions and small steps in low curvature directions. If

The curvature and its conditioning are formalized in terms of the eigenvalues of the matrix of second derivatives of $\mathcal{E}$, but we won't go into that here.

[1] X. Glorot and Y. Bengio, 2010. Understanding the difficulty of training deep feedforward neural networks. AISTATS

(a)

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮



(b)

x_1	x_2	t
1003.2	1005.1	3.3
1001.1	1008.2	4.8
998.3	1003.4	2.9
⋮	⋮	⋮

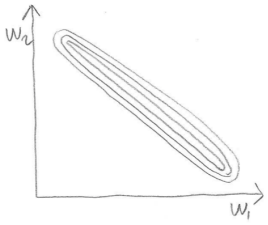


Figure 6: Unnormalized data can lead to badly conditioned curvature. **(a)** The two inputs have vastly different scales. Changing w_1 has a much bigger effect on the model's predictions than changing w_2, so the cost function curves more rapidly along that dimension. **(b)** The two inputs are offset by about the same amount. Changing the weights in a direction that preserves $w_1 + w_2$ has little effect on the predictions, while changing $w_1 + w_2$ has a much larger effect.

the curvature is very different in different directions, we say the curvature is **badly conditioned**. An example is shown in Figure 5. A region with badly conditioned curvature is sometimes called a **ravine**, because of what it looks like in a surface plot of the cost function. Think about the effect this has on optimization. You need to set α small enough that you don't get oscillations or instability in the high curvature directions. But if α is small, then progress will be very slow in the low curvature directions.

In practice, neural network training is very badly conditioned. This is likely a big part of why modern neural nets can take weeks to train. Much effort has been spent researching second-order optimization methods, alternatives to SGD which attempt to correct for the curvature. Unfortunately, these methods are complicated and pretty hard to tune (in the context of neural nets), so SGD is still the go-to algorithm, and we just have to live with badly conditioned curvature.

However, we can at least try to eliminate particular egregious instances of badly conditioned curvature. One way in which badly conditioned curvature can arise is if the inputs have very different scales or are off-center. See Figure 6 for examples of this effect in linear regression problems. Such examples could arise because inputs represent arbitrary units, such as feet or years. This framing almost immediately suggests a workaround: **normalize** the inputs so that they have zero mean and unit variance. I.e., take

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}, \tag{16}$$

where $\mu_j = \mathbb{E}[x_j]$ and $\sigma_j^2 = \mathrm{Var}(x_j)$.

It's worth mentioning two very popular algorithms which help with badly

It is perhaps less intuitive why having the *means* far from zero causes badly conditioned curvature, but rest assured this is an important effect, and worth combating.

conditioned curvature: batch normalization and Adam. We won't cover them properly, but the original papers are very readable, in case you're curious.[2] Batch normalization normalizes the activations of each layer of a network to have zero mean and unit variance. This can help significantly for the reason outlined above. (It can also attenuate the problem of saturated units.) Adam separately adapts the learning rate of each individual parameter, in order to correct for differences in curvature along individual coordinate directions.

4.9 Recap

Here is a table to summarize all the pitfalls, diagnostics, and workarounds that we've covered:

Problem	Diagnostics	Workarounds
incorrect gradients	finite differences	fix them, or use an autodiff package
local optima	(hard)	random restarts
symmetries	visualize $\mathbf{W}$	initialize $\mathbf{W}$ randomly
slow progress	slow, linear training curve	increase α
instability	cost increases	decrease α
oscillations	fluctuations in training curve	decrease α; momentum
fluctuations	fluctuations in training curve	decay α; iterate averaging
dead/saturated units	activation histograms	initial scale of $\mathbf{W}$; ReLU
badly conditioned curvature	(hard)	normalization; momentum; Adam; second-order opt.

[2] D. P. Kingma and J. L. Ba, 2015. Adam: a method for stochastic optimization. ICLR
S. Ioffe and C. Szegedy, 2015. Batch normalization: accelerating deep network training by reducing internal covariate shift.