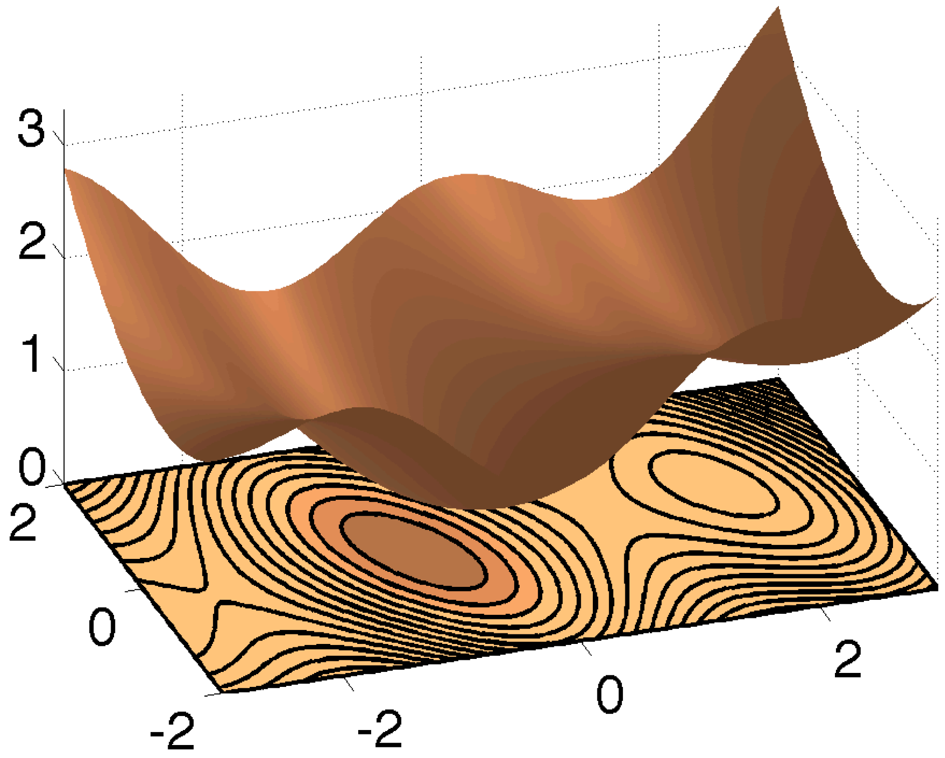
Optimization in ML

As we've seen, a lot of machine learning methods are based on optimizing something: maximizing a likelihood, minimizing an error measure, maximizing information content, etc. Let's say we want to solve

$$\min_\theta L(\theta) \qquad \theta \in \mathbb{R}^d$$

Here $L(\theta)$ is our objective, and θ is a d-dimensional parameter vector.

In our examples so far, the objective has often been simple (e.g., convex and quadratic); but more generally, it could have a nearly arbitrary shape, including multiple local optima:



In this plot, $\theta \in \mathbb{R}^2$. The horizontal axes represent θ_1 and θ_2, and the vertical axis is $L(\theta)$.

Optimization from data

Where does a problem like this come from? Suppose we have a model with parameters θ . This model tells us the probability of an individual data point x_i. Write X for all of our data, $x_1 \ldots x_T$. Then by Bayes' rule,

$$P(\theta \mid X) = P(X \mid \theta)P(\theta)/P(X)$$

In this equation,

- The term $P(X)$ doesn't depend on θ, so it doesn't help us prefer one value of θ over another.
- The term $P(\theta)$ is often *uninformative*: we don't know ahead of time what θ is likely to be, so $P(\theta)$ is either uniform or near-uniform over some large region.

That leaves just one term to help us find a good value of θ: $P(X \mid \theta)$. This term is called the *likelihood* or *evidence* for θ. The most probable θ is the one that maximizes the likelihood.

> *Picking θ this way is called maximum likelihood estimation. It's a common and effective strategy. There are alternatives, though: e.g., we could use a more-informative prior, or we could ask for a sample of likely values of θ instead of a single point estimate.*

If our data points are independent of one another, then

$$P(X \mid \theta) = P(x_1 \mid \theta)P(x_2 \mid \theta)\ldots P(x_T \mid \theta)$$

In this case, since sums can be easier to work with than products, we'll often work with the *log-likelihood*

$$\ln P(X \mid \theta) = \ln P(x_1 \mid \theta) + \ln P(x_2 \mid \theta) + \ldots + \ln P(x_T \mid \theta)$$

We can do something similar if our model tells us the *conditional* probability of a label y_i given input features x_i. A derivation like the one above leads us to maximize the *conditional likelihood* $P(Y \mid X, \theta)$ or the *conditional log-likelihood*

$$\ln P(Y \mid X, \theta) = \ln P(y_1 \mid x_1, \theta) + \ln P(y_2 \mid x_2, \theta) + \ldots + \ln P(y_T \mid x_T, \theta)$$

This sort of structure in our objective function is common in machine learning: a sum over training examples, so that $L(\theta) = \sum_i \ell_i(\theta)$. We'll see more later about how to take advantage of it. As a concrete example, ℓ_i could measure squared error for a training example in a regression problem:

$$\ell_i(\theta) = (x_i \cdot \theta - y_i)^2$$

The first order

We will focus on *first-order* methods for optimization. These are methods that access our loss function only through its value and gradient.

Even though they're simple, first-order methods turn out to be quite effective for a lot of machine learning problems. This happens for a few different reasons:

- Speed: each optimizer step can be fast. If we have big data or big models, we might not be able to afford fancier optimizers.
- Just enough accuracy: first-order methods are not as accurate as some other optimizers. But in ML, we often don't care about that: those last few decimal places of the loss function probably correspond to overfitting, and won't translate to the test set.
- Exploration: the trajectory of a first-order method can skip over small increases in the loss, helping us find a region of better values of θ. It can also wander around the region near a local optimum, and help us characterize the shape of the objective function there.
- Regularization: For reasons we don't completely understand, first-order methods seem to pick values of θ that generalize better — even among those with the same $L(\theta)$.

For these reasons, first-order methods are often the best place to start if we need to pick an optimizer for machine learning.

Gradient descent

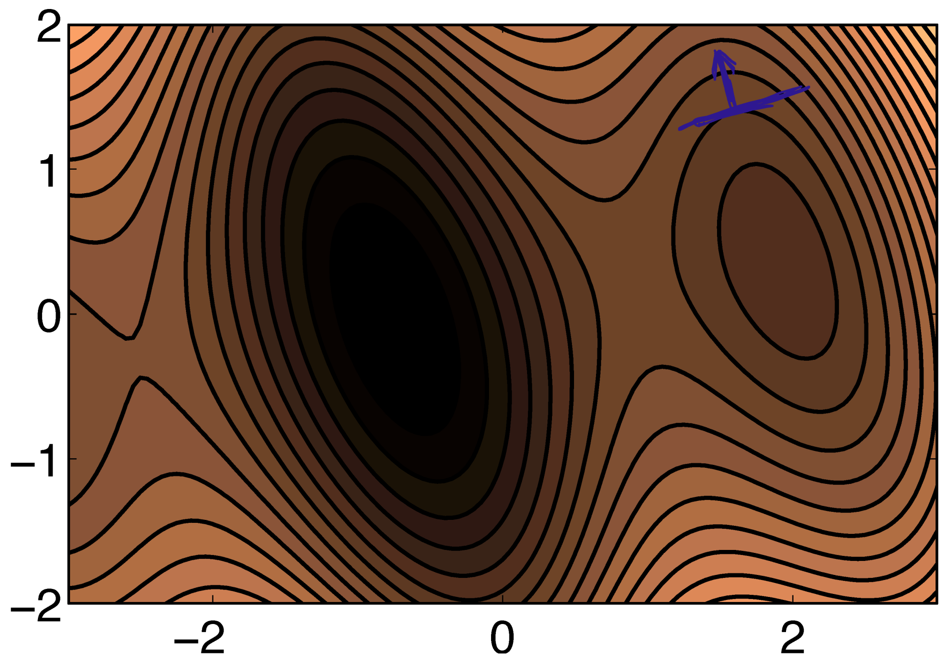
To solve $\min_\theta L(\theta)$, one place to start is the *gradient descent* algorithm:

> **Gradient descent**
>
> ```
> input $\theta_1, \eta, T$
> for $t \leftarrow 1, \ldots, T$:
>     $g_t \leftarrow \frac{dL}{d\theta}\Big|_{\theta_t}$
>     $\theta_{t+1} \leftarrow \theta_t - \eta g_t$
> ```

The vector $g_t \in \mathbb{R}^d$ is the *gradient* of L at θ_t. The parameter η is the *learning rate* or *step size*: we'll see below that it trades off optimization speed against stability.

With the conventions we've been using so far, the gradient is the transpose of the first derivative: if $dL = L'(\theta)\,d\theta$, then $g_t = L'(\theta_t)^T$. But you will commonly see the other convention, where L' stands for the gradient itself rather than its transpose.

Why does gradient descent work? In the neighborhood of θ_t, the function L decreases most rapidly in the negative gradient direction $-g_t$:



So, if we take a small-enough step in this direction, it should reduce L. But if we step too far, the curvature of L can mean that we don't actually reduce L. Strong curvature shows up in the figure as contour lines that bend sharply or change spacing suddenly.

One of the main bottlenecks in using gradient descent is computing the gradients — both in the sense that it takes up a lot of compute time and in the sense that it takes up a lot of of our effort in applying gradient descent to any given problem. To help with the latter difficulty, we can use *automatic differentiation* toolkits; these are commonly included for example in libraries for working with neural networks. However, autodiff toolkits can't always handle every case we need, and may even fail silently, so it's wise to consider them as a potential source of bugs. (Though perhaps fewer bugs than if we tried to differentiate by hand.)

A lot of times, we won't wind up using gradient descent itself, for reasons to be discussed below. But its cousin, *stochastic* gradient descent, can often work quite well. The two methods share a lot of the same intuition and analysis; so we'll start by looking at plain gradient descent, and then cover the stochastic version.

Local quadratic model

Above we gave a handwavy intuition for why a gradient step decreases $L(\theta)$. Can we make this intuition precise? Let's look at a 1d slice of our objective $L(\theta)$ by restricting to the line that goes through θ_t and θ_{t+1}.

We can make a first-order Taylor expansion:

$$L(\theta) = L(\theta_t) + (\theta - \theta_t) \cdot g_t + \frac{1}{2}\|\theta - \theta_t\|^2 R(\theta)$$

Here the residual $R(\theta)$ is an unknown function — but by the mean value theorem, we know that $R(\theta)$ is equal to the second derivative of our slice of L, evaluated at some point along the line segment between θ_t and θ.

Suppose we know that the second derivative is bounded, at least in the neighborhood of θ_t. In particular, let's say $|R(\theta)| \leq H$ for some constant $H \geq 0$. Remembering that our step $(\theta_{t+1} - \theta_t)$ is equal to $-\eta g_t$, we get

$$L(\theta_{t+1}) \leq L(\theta_t) - \eta\|g_t\|^2 + \frac{1}{2}\eta^2\|g_t\|^2 H$$

So, as long as $g_t \neq 0$ and $0 < \eta < 2/H$, we get a decrease in L. If we set $\eta = 1/H$, we will reduce L by at least

$$\frac{\|g_t\|^2}{2H}$$

on each step.

In other words, we make progress as long as we take a small enough step; the best step size depends on the local curvature of L. For fast progress we want the gradient to be big compared to the local curvature. If we know the curvature, the size of the gradients, and the difference between the initial and final values of L, we can predict how long our optimization will take.

Setting the learning rate

Of course, we often have no idea what the local curvature of L is like. So what happens if we set the step size wrong?

If we set η too small, we slow down our progress: as long as η is small compared to the local curvature, the gradient direction g_{t+1} will be more or less the same as g_t. So, if we halve our step size, it will just take us twice as many steps to go the same distance.

On the other hand, if we set η too big, the result can be much more dramatic. Imagine minimizing a simple 1d quadratic:

$$L(\theta) = \frac{1}{2}H\theta^2$$

The gradient descent update is

$$\theta_{t+1} = \theta_t - \eta H \theta_t = (1 - \eta H)\theta_t$$

If we set $\eta = 3/H$, so that $1 - \eta H = -2$, we get

$$\theta_t = (-2)^{t-1}\theta_1$$

so that the sequence θ_t flips back and forth from positive to negative while blowing up exponentially!

Exercise: what happens to gradient descent if we multiply the objective L by a constant α?

Conditioning

It gets worse: if the dimension of θ is higher than 1, we can run into *both of the above problems* at once: the learning rate can simultaneously be too small and too large. If we're lucky, we get a nice objective function like this one:

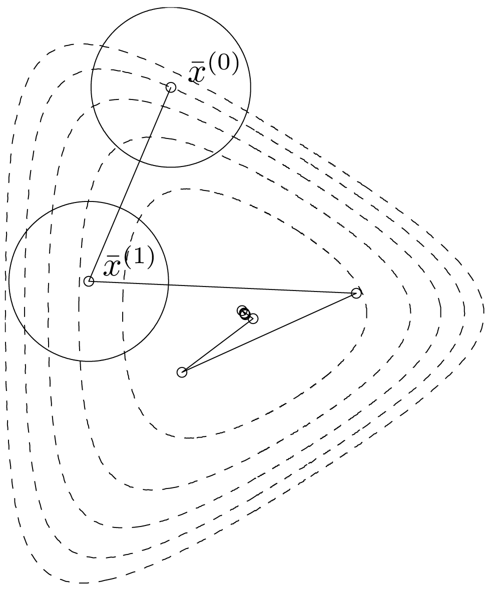


Image credit: Boyd & Vandenberghe

In the above picture, the curvature is similar in all directions, so the same learning rate is appropriate no matter what θ_t is.

But more typically, this happens:

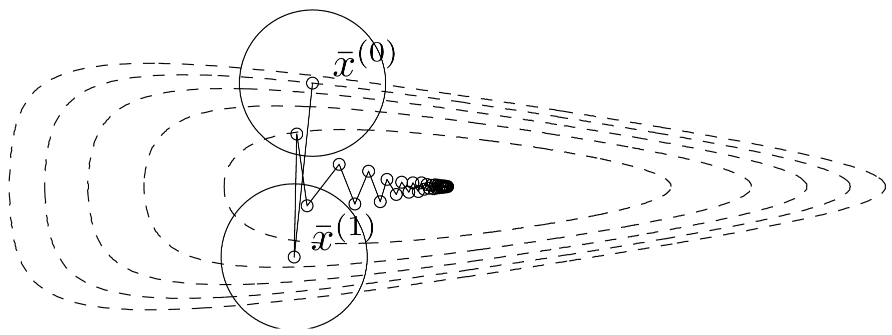


Image credit: Boyd & Vandenberghe

In the above picture, the objective is flat (low second derivative) in the horizontal direction, but sharply curved (high second derivative) in the vertical direction. So, we have to choose a small learning rate to avoid diverging in the vertical direction — which means that we can only make slow progress in the horizontal direction.

This phenomenon, where the curvature is different in different directions, is called the *conditioning* of an optimization problem. In a well conditioned problem, the curvature in all directions is similar, and gradient descent can make fast progress. In a poorly conditioned problem, the curvature varies a lot, and gradient descent can only make slow progress. We can see this difference if we look at how quickly gradient descent reduces the objective function in the above two problems:

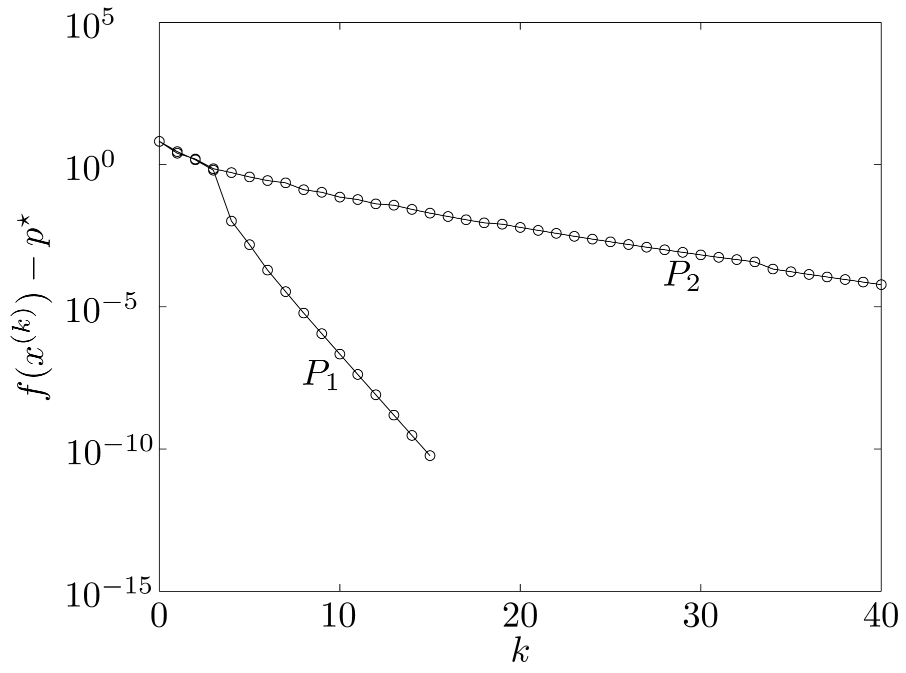


Image credit: Boyd & Vandenberghe

To measure conditioning, it's common to define the *condition number* of a problem to be the ratio between the largest and smallest curvatures that we can find in our objective function. Locally, this is the ratio between the largest and smallest positive eigenvalues of the Hessian (second derivative matrix) of our objective L.

We can relate the condition number to the 1D bound that we computed earlier,

$$L(\theta_{t+1}) \leq L(\theta_t) - \frac{\|g_t\|^2}{2H}$$

Here H is the 1D curvature, which is bounded by the largest curvature in our objective function L.

Preconditioning

A sharp eye might notice that the two contour plots above are strongly related: the second one is just stretched horizontally and squashed vertically compared to the first. We can undo this stretching and squashing by redefining our parameter vector: if $L(\theta)$ is the loss function pictured in the second figure, then we can instead minimize $M(\phi) = L(\theta)$, where ϕ is defined as

$$\phi = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{4}{3} \end{pmatrix} \theta$$

If we know (or can guess) a good transformation, we are rewarded: gradient descent can make progress much faster, even though we are effectively solving the same problem. The cost is that we have to go back and forth between the two different parameter representations on every iteration of gradient descent; if it's expensive to apply or invert our transformation, we may lose more time than we gain.

This strategy, transforming our parameter vector to make an optimization problem easier, is called *preconditioning*. In general we can precondition with any invertible transformation that we like. But, linear preconditioners are by far the most common. And, by far the most common linear preconditioners are diagonal ones like we used above.

Finding a good preconditioner — one that is both efficient and effective — may be difficult. But if we can do it, it's a great way to speed up first-order methods. For this reason, there are a lot of methods that try to discover a reasonable preconditioner as we go along: e.g., one of the most popular is Adam.

Momentum

We can see in the above plots, as well as in our analysis of the quadratic objective, that gradient descent tends to *oscillate* when the learning rate is close to or above its stability limit. That is, the gradient tends to point in opposite directions on adjacent iterations. Maybe if we could get rid of this oscillation, we could push the learning rate higher and converge faster?

In the *momentum* method, we keep a running average of the past gradients, and use this average direction to update θ instead of the raw gradient. Intuitively, if our gradients keep pointing in the same direction, our optimizer builds up momentum and takes larger steps.

> **Gradient descent with momentum**
>
> ```
> input $\theta_1, \eta, \beta, T$
> $m_0 \leftarrow 0$
> for $t \leftarrow 1, \ldots, T$:
>     $g_t \leftarrow \frac{dL}{d\theta}\Big|_{\theta_t}$
>     $m_t \leftarrow (1-\beta) g_t + \beta m_{t-1}$
>     $\theta_{t+1} \leftarrow \theta_t - \frac{\eta}{1-\beta^t} m_t$
> ```

Note that we are weighting recent gradients more heavily in our average; τ-step older gradients are scaled down by a factor of β^τ. This is called an exponentially weighted moving average. Also note the scaling factor $\frac{1}{1-\beta^t}$: this ensures that the weights in our average always sum to 1. Without this factor, in early iterations we'd go too slowly, since it would take a while to build up momentum even if all of our gradients point in the same direction.

Why does momentum help? If the gradients are oscillating, they'll tend to cancel each other out. So, if our learning rate is too high for the curvature in some direction, we'll automatically slow down our progress in that direction. Meanwhile, if our curvature is flatter in another direction, our gradients in that direction will tend to have the same sign, so we'll continue to make fast progress.

It turns out that, if we set the learning rate and momentum carefully, we can under some conditions reduce L at a rate proportional to $\frac{1}{\sqrt{H}}$ without diverging. Compare this to the earlier progress rate for plain gradient descent, which was proportional to $\frac{1}{H}$. For poorly conditioned objectives, the faster rate can make a huge difference. (See the supplemental reading for a full derivation.)

Stochastic gradient descent

Until now we haven't paid attention to the structure of our objective when designing or analyzing our algorithms. Let's look now at what the consequences are if our objective is a sum over training examples:

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell_i(\theta)$$

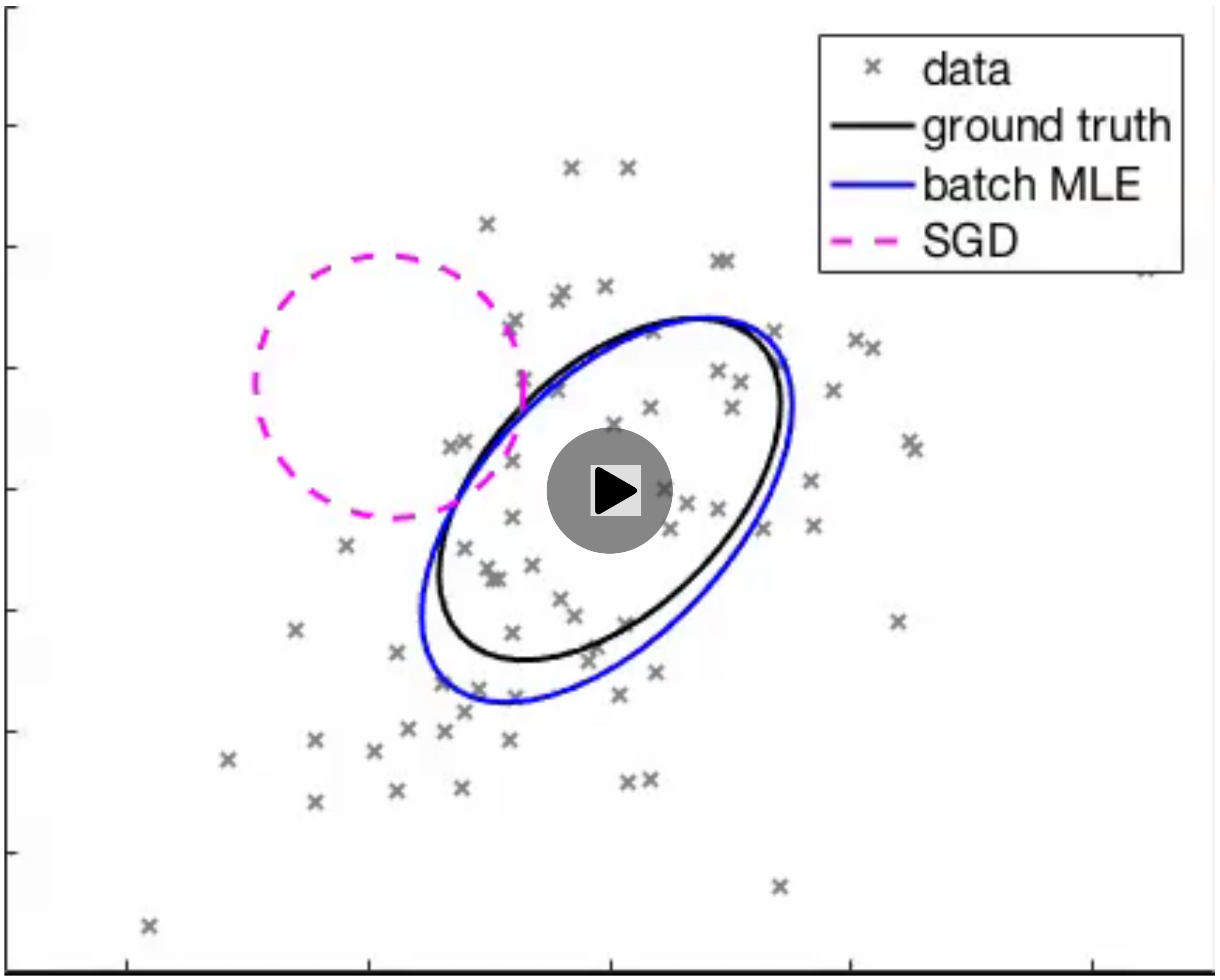
What kind of difficulties might we run into if we use gradient descent on this kind of objective?

If our examples are i.i.d., then every one of the terms in our objective is in some sense equivalent. It seems like a waste to compute all of them on every iteration of gradient descent — especially if we are only going to take a small step in the resulting direction.

This suggests that we could evaluate only a few randomly sampled terms from the objective on each iteration without losing too much information. That is, we will pick B indices uniformly and randomly without replacement, and average together the gradients of only the corresponding terms ℓ_j. If $B = 1$ we call this method *stochastic gradient descent*, while if $B > 1$ we call it *minibatch (stochastic) gradient descent*.

For this purpose, have a look at `numpy.random.choice`.

Intuitively, we might get unlucky and *increase* the value of L on any given iteration: we could pick a non-representative sample and step in the wrong direction. But on average our stochastic gradients will point in the right direction. If our learning rate is small enough, the errors will tend to average out, and we will still tend to decrease L.



```
Stochastic / Minibatch Gradient Descent

input $\theta_1, \eta, \beta, B, T$
$m_0 \leftarrow 0$
for $t \leftarrow 1, \dots, T$:
    for $i \leftarrow 1, \dots, B$:
        $j_{ti} \leftarrow$ random $1 : N$ w/o replacement
        $g_{ti} \leftarrow \frac{d}{d\theta} \ell_{j_{ti}} \Big|_{\theta_t}$

    $g_t \leftarrow \frac{1}{B} \sum_{i=1}^{B} g_{ti}$
    $m_t \leftarrow (1 - \beta) g_t + \beta m_{t-1}$
    $\theta_{t+1} \leftarrow \theta_t - \frac{\eta}{1-\beta^t} m_t$
```

Note that we sample indices without replacement. In fact, it works best to sample without replacement across an entire *epoch* (a pass through our training data). That is, we mark each term that we sample, and never sample it again until we've hit every other term in our objective. When we run out of unmarked terms, we clear the marks, and continue.

Gradient sample variance

Let's first look at how stochastic gradient descent works ($B = 1$). The biggest difference from plain gradient descent is *gradient sample variance*: our stochastic gradient might point in a completely different direction depending on which term ℓ_j we sample. If we're lucky the gradient variance will be small compared to the size of the gradient. If we're unlucky, the reverse might be true.

Variance can be a problem: if L is locally strictly convex (e.g., near a local optimum), Jensen's inequality says that

$$\mathbb{E}(L(\theta_{t+1})) > L(\mathbb{E}(\theta_{t+1}))$$

The RHS is what happens with plain gradient descent: we deterministically set θ_{t+1} using the expected gradient. The LHS is what happens with SGD: we go in the right direction on average, but gradient variance together with the curvature of L mean that we won't make as much progress — or might even tend to increase L.

If our gradient samples have variance bounded by σ^2 in all directions, and our learning rate is η, then our updates to θ_t will have variance bounded by $\eta^2\sigma^2$. Then, if the curvature of L is H, the difference between the two sides of Jensen's inequality will be bounded by $H\eta^2\sigma^2$.

This analysis means that we now have two constraints on learning rate: one due to the way the curvature of L changes the deterministic gradient (as before) and one due to variance (new for SGD). If we didn't have to worry about variance, our analysis of plain gradient descent shows that we could improve L by $\|\mathbb{E}(g_t)\|^2/2H$ per iteration by setting $\eta = 1/H$. But this setting of η results in a penalty of σ^2/H due to gradient variance. So, if σ^2 is comparable to or bigger than $\|E(g_t)\|^2$, we may have to reduce our learning rate to make progress.

Behavior of SGD

Because of the differing effects of curvature and variance, SGD typically has three different phases of convergence:

- Far away from the optimum, the gradient samples all point in about the same direction. So, $\|E(g_t)\|^2$ dominates, and SGD behaves mostly like plain gradient descent.
- A bit nearer to the optimum, the gradient variance and squared gradient norm

become comparable. In this regime our progress depends on moving slowly enough to average out the gradient variance, and SGD will make much less progress per iteration than plain gradient descent. (It may still be faster overall, since the iterations are a lot cheaper.)

- When we get sufficiently close to the optimum, the gradient variance dominates, and we stop reducing L at all. Instead we bounce around at a scale determined by σ^2, H, and η.

We can tune the transitions between these phases by tuning the learning rate: in the range we care about, changing η affects the variance of ηg_t as $O(\eta^2)$, but it affects our reduction in L from the expected update $\mathbb{E}(\eta g_t)$ as $O(\eta)$. So reducing η can mitigate the effect of gradient variance; for example, it can allow the second phase to make more progress, getting us closer to a local optimum before we start bouncing around.

Unfortunately, though, the best value of η might be different during different parts of the overall optimization. For this reason, it's common to alter η during the course of optimization using a *learning rate schedule*. Designing a learning rate schedule is unfortunately somewhat of a black art: at one point, one of my past professors set up his workstation so that his mouse pointer controlled learning rate and momentum, and learned to achieve the fastest possible convergence on the fly by streaming a lot of diagnostics and developing an intuition for what could keep the network on the ragged edge of stability.

Perhaps counterintuitively, once we fix η, the three phases above *do not depend* on the size of our training set: all that matters is the curvature of L and the variance of g_t. SGD might "finish" (i.e., reach the last regime above) before even seeing all of our data.

Instead, the size of our training set influences optimization only indirectly, through our choice of hyperparameters (learning rate schedule and, once we get to it, minibatch size). That is, with a larger training set, we can hope to generalize better, so we may want to select different hyperparameters to make our optimization more accurate. These altered hyperparameters can change our convergence rate.

Behavior of minibatch SGD

What happens when we use minibatches of size $B > 1$? Broadly we get similar behavior to SGD, but there are three interacting effects we need to consider.

The first effect is computational cost: if we double B, we have to compute twice as many gradients for each parameter update. The second effect is variance: if we double B, we halve the variance of our gradient estimates. The third effect is curvature: the gradient of L changes as we update θ_t, and this change might wipe out any benefit of an accurate gradient estimate.

Looking at the first two effects together, it makes sense to compare:

- two steps of SGD with minibatch size B and learning rate η, vs.
- one step of SGD with minibatch size $2B$ and learning rate 2η.

These approaches seem similar at first glance. If the variance of a single gradient sample is bounded by σ^2 in all directions, then either of these approaches yields a total update variance of $2\eta^2\sigma^2/B$. Either of these approaches yields a similar decrease in L, at least to first order. And, either of these approaches has to compute $2B$ gradients in total.

There are some factors that break the apparent equivalence, though. Computing $2B$ gradient samples might take a different amount of wall-clock time depending on whether we do them all at once or split them up. This is particularly true if we can compute the samples in parallel (e.g., on a GPU); in this case the all-at-once approach could be much cheaper.

On the other hand, curvature kicks in if we try to grow B too much. One way to see this is to remember that step size is limited both by the variance of our stochastic gradient estimates and by the effect of the curvature of L on the deterministic gradient direction. For small minibatches, typically the first limit will be tighter; in this case, our two approaches will have similar performance. But bigger minibatches increase the step size while keeping the variance approximately constant. So eventually, the second limit will take over and keep us from increasing B and η too much.

Finally, our minibatch size might be limited by architectural factors. For example, we might want to ensure that each minibatch fits within our GPU memory or our main memory cache.

Empirically, good minibatch sizes can vary greatly depending on details of our loss function and computational architecture. Reasonable values might range anywhere from $B = 10$ to $B = 100,000$.

Over this range of B, we actually have a fair bit of flexibility: we can often achieve good performance with a wide range of different minibatch sizes and learning rates. But there is one important caveat: the analysis above tells us that we have to scale η along with B. If we fail to do this, we can wind up implicitly using a mis-tuned learning rate, and hurting performance.

Suggested reading:

For gradient descent:

- Boyd & Vandenberghe, Convex Optimization, sec. 9.3

For an analysis of momentum:

- http://mitliagkas.github.io/ift6085-2019/ift-6085-lecture-5-notes.pdf