CS168: The Modern Algorithmic Toolbox
Lecture #15: Gradient Descent Basics

Tim Roughgarden & Gregory Valiant

May 18, 2015

1 Executive Summary

Over the past few weeks, we've discussed several fundamental algorithms that have numerous applications — the singular value decomposition (SVD), linear and convex programming, etc. This week we'll talk about *gradient descent*, another justly famous algorithm (or really, family of algorithms). Unlike the past few weeks, where we treated the algorithms as "black boxes," we'll also talk about how to implement gradient descent. One reason for this is that the algorithm is extremely simple — simpler than most of the algorithms you studied in CS161. Another reason is that to apply the algorithm effectively in practice, it's important to be familiar with the various ways one can tweak the basic algorithm.

After the course, the following would be a good list of things to remember about gradient descent.

1. The goal of gradient descent is to minimize a function via greedy local search.
2. Gradient descent scales well to large data sets, especially with some tweaks and if an approximately optimal solution is good enough. For example, the algorithm doesn't even need to multiply matrices. This is the primary reason for the algorithm's renaissance in the 21st century, driven by large-scale machine learning applications.
3. Gradient descent provably solves many convex problems. (Some problems of interest are convex, as discussed last lecture, while others are not.)
4. Gradient descent can be an unreasonably good heuristic for the approximate solution of non-convex problems; this is one of the main points of Mini-Project #8.

Gradient descent has been around for centuries. These days, the main "killer app" is machine learning. Model-fitting often reduces to optimization — for example, maximizing the likelihood of observed data over a family of generative models. A remarkably large fraction of modern machine learning research, including some of the much-hyped recent work on "deep learning," boils down to implementing variants of gradient descent on a very large scale (i.e., for huge training sets). Indeed, the choice of models in many machine learning applications is driven as much by computational considerations — whether or not gradient descent can be implemented quickly — as by any other criteria.

2 How to Think About Gradient Descent

2.1 Unconstrained Optimization

Viewed the right way, gradient descent is really easy to understand and remember — more so than most algorithms. First off, what problem is gradient descent trying to solve? *Unconstrained optimization*, meaning that for a given real-valued function $f : \mathbb{R}^n \to \mathbb{R}$ defined on n-dimensional Euclidean space, the goal is

$$\min f(\mathbf{x})$$

subject to

$$\mathbf{x} \in \mathbb{R}^n.$$

Note that maximizing a function falls into this problem definition, since maximizing f is the same as minimizing $-f$.[1] In this lecture, we'll always assume that f is differentiable and hence also continuous.[2]

2.2 Warm-Up #1: $n = 1$

Suppose first that $n = 1$, so $f : \mathbb{R} \to \mathbb{R}$ is a univariate real-valued function. We can visualize the graph of f in the usual way; see Figure 1.

Intuitively, what would it mean to try to minimize f via greedy local search? For example, if we start at point x_0, then we look to the right (f goes up) and to the left (f goes down), and go further to the left. (Recall we want to make f as small as possible.) If we start at x_1, then f is decreasing to the right and increasing to the left, so we'd move further to the right. In the first case, the algorithm terminates at the bottom of the left basin; in the second case, at the bottom of the right basin. Intuitively, we're releasing a ball at some point of the graph of the function of f, and the algorithm terminates at the final resting point of this ball (after gravity has done its work).

A little more formally — we'll be precise when we discuss the general case — the basic algorithm in the $n = 1$ case is the following:

1. while $f'(x) \neq 0$

[1]A key ingredient of linear and convex programs (discussed last lecture) is their constraints, which describe which solutions are allowed. In unconstrained optimization, there are no constraints, and the only consideration is the objective function. There are various ways to transform constrained problems into unconstrained ones (like Lagrangian relaxation and barrier functions), and various ways to extend gradient descent to handle constraints (like projecting back to the feasible region). We won't have time for any of these, but you'll see them if you study convex optimization or machine learning more deeply.

[2]There are extensions of gradient descent that relax this assumption, but we won't have time to discuss them.

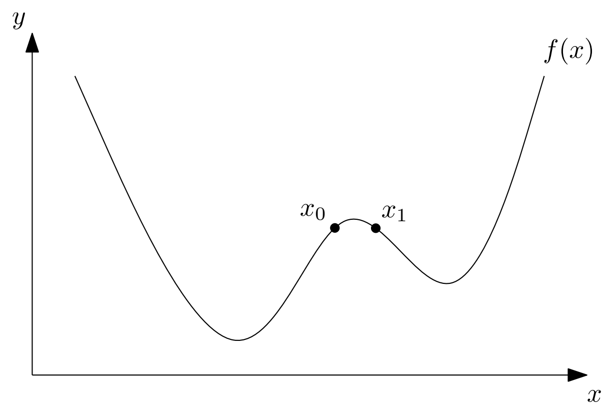

Figure 1: A univariate objective function f. Minimizing f using gradient descent may yield a different local minimum depending on the start position.

(a) if $f'(x) > 0$ — so f is increasing — then move x a little to the left;

(b) if $f'(x) < 0$ then move x a little to the right.

Note that at each step, the derivative of f is used to decide which direction to move in.

Already with $n = 1$ and in Figure 1, it is clear that the outcome of gradient descent depends on the starting point. This example also shows how, with a non-convex function f, gradient descent can compute a local minimum — meaning there's no way to improve f by moving a little bit in either direction — that is worse (i.e., larger) than a global minimum.[3] The converse also holds: if f is convex, then gradient descent can only terminate at a global minimum.[4] To see this, suppose x is sub-optimal and x^* is optimal. As λ goes from 0 to 1, the expression $\lambda f(x^*) + (1-\lambda)f(x)$ changes linearly from $f(x)$ to $f(x^*)$. Since all chords of the graph of a convex function lie above the graph, meaning

$$f(\lambda x^* + (1-\lambda)x) \leq \lambda f(x^*) + (1-\lambda)f(x),$$

it follows that moving toward x^* from x can only decrease f. Thus gradient descent will not get stuck at x.[5]

2.3 Warm-Up #2: Linear Functions

In almost all of the real applications of gradient descent, the number n of dimensions is much larger than 1. Already with $n = 2$ we see an immediate complication: from a point $\mathbf{x} \in \mathbb{R}^n$, there's an infinite number of directions in which we could move, not just 2.

[3]Recall from last lecture that a function is convex if the region above its graph is a convex set. Equivalently, given two points on or above its graph, the entire line segment between the two points should also be on or above its graph. It's clear that the function shown in Figure 1 does not have this property.

[4]Modulo any approximation error from stopping before the derivative is exactly zero; see Section 2.5 for details.

[5]The same argument works for any number n of dimensions.

To develop our intuition, we first consider the rather silly case of *linear functions*, meaning functions of the form

$$f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} + b, \tag{1}$$

where $\mathbf{c} \in \mathbb{R}^n$ is an n-vector and $b \in \mathbb{R}$ is a scalar. There are the same types of objective functions that we were using last lecture in linear programs.

Unconstrained minimization of a linear function is a trivial problem, because (assuming $\mathbf{c} \neq 0$) it is possible to make the objective function arbitrary negative. To see this, take any vector $\mathbf{x}$ with negative inner product $\mathbf{c}^T\mathbf{x} < 0$ with $\mathbf{c}$ (such as $-\mathbf{c}$) and consider points of the form $\beta\mathbf{x}$ for β arbitrarily large. This issue does not arise with linear programs that have bounded feasible regions — recall that geometrically, a linear program is asking for the feasible point that is furthest in the direction of maximization or minimization. Here, the feasible region is all of $\mathbb{R}^n$, so there is no optimal point.

Let's instead ask: suppose you are currently at a point $\mathbf{x} \in \mathbb{R}^n$, and you are allowed to move at most one unit of Euclidean distance in whatever direction you want. Where you should go to decrease the function f in (1) as much as possible, and how much will the function decrease? To answer this, let $\mathbf{u} \in \mathbb{R}^n$ be a unit vector; moving from $\mathbf{x}$ one unit of distance in the direction $\mathbf{u}$ changes the objective function as follows:

$$\begin{aligned}
\mathbf{c}^T\mathbf{x} + b &\mapsto \mathbf{c}^T(\mathbf{x}+\mathbf{u}) + b && (2)\\
&= \mathbf{c}^T\mathbf{x} + b + \mathbf{c}^T\mathbf{u} && (3)\\
&= \underbrace{\mathbf{c}^T\mathbf{x} + b}_{\text{independent of } \mathbf{u}} + \|\mathbf{c}\|_2 \underbrace{\|\mathbf{u}\|_2}_{=1} \cos\theta, && (4)
\end{aligned}$$

where θ denotes the angle between the vectors $\mathbf{c}$ and $\mathbf{u}$. To decrease f as much as possible, we see that we should make $\cos\theta$ as small as possible (i.e., -1), which we do by choosing $\mathbf{u}$ to point in the opposite direction of $\mathbf{c}$ (i.e., $\mathbf{u} = -\mathbf{c}/\|\mathbf{c}\|_2$). The derivation (2)–(4) shows that moving one unit in this direction causes f to decrease by $\|\mathbf{c}\|_2$, so $\|\mathbf{c}\|_2$ is also the rate of decrease (per unit moved) in the direction $-\|\mathbf{c}\|_2$. These are the things to remember about this warm-up example: *the direction of steepest descent is that of $-\mathbf{c}$, for a rate of decrease of $\|\mathbf{c}\|_2$*.

2.4 Some Calculus, Revisited

What about general (differentiable) functions, the ones we really care about? The idea is to *reduce general functions to linear functions.* This might sound ridiculous, given how simple linear functions are and how weird general functions can be, but basic calculus already gives a method for doing this.[6]

What it really means for a function to be differentiable at a point is that it can be locally approximated at that point by a linear function. For a univariate differentiable function, like

[6]Note the parallel with our approach to the optimization problem underlying PCA: solving $\max \mathbf{x}^T\mathbf{B}\mathbf{x}$ is easy when $\mathbf{B}$ is a diagonal, and basic linear algebra reduces the case of general matrices of the form $\mathbf{B} = \mathbf{A}^T\mathbf{A}$ to diagonal matrices.

in Figure 1, it's clear what the linear approximation is — just use the tangent line. That is, at the point x, approximate the function f for y near x by the linear function

$$f(y) \approx f(x) + (y-x)f'(x) = \underbrace{f(x) - xf'(x)}_{y\text{-intercept}} + y\underbrace{f'(x)}_{\text{slope}},$$

where x is fixed and y is the variable. It's also clear that the tangent line is only a good approximation of f locally — far away from x, the value of f and this linear function have nothing to do with each other. Thus being differentiable means that at each point there exists a good local approximation by a linear function, with the specific linear function depending on the choice of point.

Another way to think about this, which has the benefit of extending to better approximations via higher-degree polynomials, is through Taylor expansions. Recall what Taylor's Theorem says (for $n = 1$): if all of the derivatives of a function f exist at a point x, then for all sufficiently small $\epsilon > 0$ we can write

$$f(x+\epsilon) = \underbrace{f(x) + \epsilon \cdot f'(x)}_{\text{linear approx.}} + \tfrac{\epsilon^2}{2!} \cdot f''(x) + \tfrac{\epsilon^3}{3!} \cdot f''(x) + \cdots . \tag{5}$$

So what? The point is that with the first two terms on the right-hand side of (5), we have a linear approximation of f around x staring us in the face (in the variable ϵ). This is the same as the tangent line approximation, with ϵ playing the role of $y - x$.[7]

The discussion so far has been for the $n = 1$ case for simplicity, but everything we've said extends to an arbitrary number n of dimensions. For example, the Taylor expansion (5) remains valid in higher dimensions, just with the derivatives replaced by their higher dimensional analogs. Since we'll use only linear approximation, we only need to care about the higher-dimensional analog of the first derivative $f'(x)$, which is the gradient.

Recall that for a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$, the *gradient* $\nabla f(\mathbf{x})$ of f at $\mathbf{x}$ is the real-valued n-vector

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \ldots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right) \tag{6}$$

in which the ith component specifies the rate of change of f as a function of x_i, holding the other $n - 1$ components of $\mathbf{x}$ fixed.

To relate this definition to our two-warm ups, note that if $n = 1$, then the gradient becomes the scalar $f'(x)$. If $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} + b$ is linear, then $\partial f / \partial x_i = c_i$ for every i (no matter $\mathbf{x}$ is), so ∇f is just the constant function everywhere equal to $\mathbf{c}^T$.

For a simple but slightly less trivial example, we can consider a quadratic function $f : \mathbb{R}^n \to \mathbb{R}$ of the form

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x},$$

[7]Note the analogy with some of our "lossy compression" solutions, such as low-rank matrix approximations via the SVD: the equation (5) decomposes $f(x+\epsilon)$ into the sum of a bunch of terms; for sufficiently small ϵ, the first two terms on the right-hand size of (5) dwarf the sum of the rest of them, yielding a good local approximation by a linear function.

where $\mathbf{A}$ is a symmetric $n \times n$ matrix and $\mathbf{b}$ is an n-vector. Expanding, we have

$$f(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_ix_j - \sum_{i=1}^{n} b_ix_i,$$

and you should check that

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) = \sum_{j=1}^{n} a_{ij}x_j - b_i$$

for each $i = 1, 2, \ldots, n$. We can therefore express the gradient succinctly as

$$\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$$

at each point $\mathbf{x} \in \mathbb{R}^n$.

We'll see another explicit gradient computation below, when we apply gradient descent to a linear regression problem. For more complex functions f, it's not always clear how to compute the gradient of f. But as long as one can evaluate f, one can estimate ∇f by estimating each partial derivative in the definition (6) in the obvious way — changing one coordinate a little bit and seeing how much f changes.

2.5 Gradient Descent: The General Case

Here is the general gradient descent algorithm. It has three parameters — $\mathbf{x}_0$, ϵ, and α — which we'll elaborate on shortly.

1. Let $\mathbf{x} := \mathbf{x}_0 \in \mathbb{R}^n$ be an initial point.

2. While $\|\nabla f(\mathbf{x})\|_2 > \epsilon$:

   (a) $\mathbf{x} := \mathbf{x} - \underbrace{\alpha}_{\text{step size}} \cdot \nabla f(\mathbf{x})$.

And that's it!

Conceptually, gradient descent enters the following gentleman's agreement with basic calculus:

1. Calculus promises that, for $\mathbf{y}$ close to $\mathbf{x}$, one can pretend that the true function f is just the linear function $f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x})$ (Section 2.4). We know what to do with linear functions: move in the opposite direction of the coefficient vector — that is, in the direction $-\nabla f(\mathbf{x})$ — to decrease the function at a rate of $\|\nabla f(\mathbf{x})\|_2$ per unit distance (Section 2.3).

2. In exchange, gradient descent promises to only take a small step (parameterized by the step size α) away from $\mathbf{x}$. Taking a large step would violate the agreement, in that far away from $\mathbf{x}$ the function $f(\mathbf{x})$ need not behave anything like the local linear approximation $f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x})$ (recall the tangent lines in Figure 1).

The starting point $\mathbf{x}_0$ can be chosen arbitrarily, though as we saw in Section 2.2, for non-convex f the output of gradient descent can vary with the choice of $\mathbf{x}_0$. For convex functions f, gradient descent will converge toward the same point — the global minimum — no matter how the starting state is chosen.[8] The choice of start state can still affect the number of iterations until convergence, however. In practice, one should choose $\mathbf{x}_0$ according to your best guess as to where the global minimum is likely be — generally, the closer $\mathbf{x}_0$ is to the global minimum of f, the faster the convergence of gradient descent.

The parameter ϵ determines the stopping rule. Note that because $\epsilon > 0$, gradient descent generally does not halt at an actual local minimum, but rather at some kind of "approximate local minimum."[9] Since the rate of decrease of a given step is $\|\nabla f(\mathbf{x})\|_2$, at least locally, once $\|\nabla f(\mathbf{x})\|_2$ gets close to 0 each iteration of gradient descent makes very little progress; this is an obvious time to quit. Smaller values of ϵ mean more iterations before stopping but a higher-quality solution at termination. In practice, one tries various values of ϵ to achieve the right balance between computation time and solution quality. Alternatively, one can just run gradient descent for a fixed amount of time and use whatever point was computed in the final iteration.

The final parameter α, the "step size," is perhaps the most important. While gradient descent is flexible enough that different α's can be used in different iterations, in practice one typically uses a fixed value of α over all iterations.[10] While there is some nice theory that gives advice on how to choose α as a function of the "niceness" of f, in practice the "best" value of α is typically chosen by experimentation. The very first time you're exploring some function f, one option is a "line search," which just means identifying by binary search the value of α that minimizing f over the line $\mathbf{x} - \alpha \cdot \nabla f(\mathbf{x})$. After a few line searches, you should have a decent guess as to a good value of α. Alternatively, you can run the entire gradient descent algorithm with a few different choices of α to see which run gives you the best results.

3 Application: Linear Regression

A remarkable amount of modern machine learning boils down to variants of gradient descent. This section illustrates how to apply gradient descent to one of the simplest non-trivial machine learning problems, namely linear regression.[11]

[8]Strictly speaking, this is true only for functions that are "strictly convex" is some sense. We'll gloss over this distinction in this lecture.

[9]If the function f is sufficiently nice — "strongly convex" is most common sufficient condition — then gradient descent provably terminates at a point very close to a global minimum.

[10]For example, one could imagine decreasing α over the course of the algorithm, a la simulated annealing. But in the common case where the norm $\|\nabla f\|_2$ of f decreases each iteration, then even with a fixed α, the distance traveled by gradient descent each iteration is already decreasing.

[11]You briefly saw linear regression on Mini-Project #4.

3.1 Linear Regression

In linear regression, the input is m data points $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)} \in \mathbb{R}^d$, each a d-dimensional vector. Also given is a real-valued "label" $y^{(i)} \in \mathbb{R}$ for each data point i.[12] For example, each data point i could correspond to a 5th-grade student, $y^{(i)}$ could correspond to the score earned by that student on some standardized test, and $\mathbf{x}^{(i)}$ could represent the values of d different "features" of student i — the average household income in his/her neighborhood, the number of years of education earned by his/her parents, etc.

The goal is to compute the "best" linear relationship between the $\mathbf{x}^{(i)}$'s and the $y^{(i)}$'s. That is, we want to compute a linear function $h : \mathbb{R}^d \to \mathbb{R}$ such that $h(\mathbf{x}^{(i)}) \approx y^{(i)}$ for every i. Every such linear function h can be written as

$$h_{\mathbf{a}}(\mathbf{x}) = a_0 + \sum_{j=1}^{d} a_j x_j$$

for a $(d+1)$-dimensional coefficient vector $\mathbf{a}$. To keep the notation simple, let's agree that every data point $\mathbf{x}^{(i)}$ has a "phantom" zero-coordinate equal to 1 (i.e., $x_0^{(i)} = 1$ for every i), so that we can just write

$$h_{\mathbf{a}}(\mathbf{x}) = \sum_{j=0}^{d} a_j x_j. \tag{7}$$

The two most common motivations for computing a "best-fit" linear function are prediction and data analysis. In the first scenario, one uses the given "labeled data" (the $\mathbf{x}^{(i)}$'s and $y^{(i)}$'s) to identify a linear function h that, at least for these data points, does a good job of predicting the label $y^{(i)}$ from the feature values $\mathbf{x}^{(i)}$. The hope is that this linear function "generalizes," meaning that it also makes accurate predictions for other data points for which the label is not already known. There is a lot of beautiful and useful theory in statistics and machine learning about when one can and cannot expect a hypothesis to generalize, which you'll learn about if you take courses in those areas. In the second scenario, the goal is to understand the relationship between each feature of the data points and the labels, and also the relationships between the different features. As a simple example, it's clearly interesting to know when one of the d features is much more strongly correlated with the label $y^{(i)}$ than any of the others.

3.2 Mean Squared Error (MSE)

To complete the formal problem description, we need to choose a notion of "best fit." We'll use the most common one, that of minimizing the mean squared error (MSE) of a linear

[12] This is an example of *supervised* learning, in that the input includes the "correct answers" $y^{(1)}, \ldots, y^{(m)}$ for the data points $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$, as opposed to just the data points alone (which would be an *unsupervised* learning problem).

function. For a linear function $h_{\mathbf{a}}$ with coefficient vector $\mathbf{a} \in \mathbb{R}^{d+1}$, this is defined as

$$\text{MSE}(\mathbf{a}) = \frac{1}{m}\sum_{i=1}^{m} E_i(\mathbf{a})^2 \tag{8}$$

where the "error" or "residual" $E_i(\mathbf{a})$ is the difference between $h_{\mathbf{a}}$'s "prediction" $h_{\mathbf{a}}(\mathbf{x}^{(i)})$ for the ith data point and the "correct answer" $y^{(i)}$:

$$E_i(\mathbf{a}) = h_{\mathbf{a}}(\mathbf{x}^{(i)}) - y^{(i)}. \tag{9}$$

There are a couple of reasons to choose the MSE objective function.[13] One is that, as we'll see, the function has several nice mathematical and computational properties. The function also has a satisfying Bayesian justification: if the data is such that each label $y^{(i)}$ is generated from $\mathbf{x}^{(i)}$ by applying a linear function $h_{\mathbf{a}}$ and then adding independent Gaussian noise to each data points, then minimizing the MSE is equivalent to the problem of maximizing (over linear functions) the likelihood of the data.

Since we want to minimize the mean-squared error, our function $f : \mathbb{R}^{d+1} \to \mathbb{R}$ is that in (8). In this minimization problem, the variables are the coefficients $\mathbf{a}$ of the linear function $h_{\mathbf{a}}$ — all of the data points (the $\mathbf{x}^{(i)}$'s) and labels (the $y^{(i)}$'s) are given as input and fixed forevermore.

One nice property of the MSE is that it is a convex function of its variables $\mathbf{a}$. The rough argument is: each function $E_i(\mathbf{a})$ is linear in $\mathbf{a}$, and linear functions are convex; taking the square only makes these functions "more convex;" and the sum (8) of convex functions is again convex. In particular, the only local minimum of the MSE function is the global minimum.

3.3 The Gradient of the MSE Function

One approach to computing the linear function with minimum-possible MSE is to apply gradient descent. To see what this would look like, let's compute the gradient of the MSE function (8). Recall that derivatives are linear — for example, $(g+h)' = g' + h'$. Since (8) has one term per data point $i = 1, 2, \ldots, m$, the gradient will also have one term per data point. The term for the ith data point is, by the chain rule of calculus,

$$\nabla(E_i(\mathbf{a}))^2 = 2E_i(\mathbf{a}) \cdot \nabla E_i(\mathbf{a}).$$

Inspecting (7) and (9), we have

$$\frac{\partial E_i}{\partial a_j} = x_j^{(i)}$$

for $j = 0, 1, \ldots, d$, and hence $\nabla E_i(\mathbf{a}) = \mathbf{x}$. Putting it all together, we have

$$\nabla f(\mathbf{a}) = \frac{2}{m}\sum_{i=1}^{m}\left(\underbrace{E_i(\mathbf{a})}_{\text{scalar}} \cdot \underbrace{\mathbf{x}^{(i)}}_{(d+1)\text{-vector}}\right), \tag{10}$$

[13]Next lecture we'll look at some important variations.

where $f(\mathbf{a})$ denotes the MSE of the linear function $h_{\mathbf{a}}$. Also, recall our convention that $x_0^{(i)} = 1$ for every i.

The gradient (10) has a natural interpretation. The ith term $E_i(\mathbf{a}) \cdot \mathbf{x}^{(i)}$ can be thought of as the ith data point's "opinion" as to how the coefficients $\mathbf{a}$ should be updated. To see this, first note that if we move from $\mathbf{a}$ in the direction of $\mathbf{x}^{(i)}$, then the prediction of the current linear function

$$h_{\mathbf{a}}(\mathbf{x}^{(i)}) = \mathbf{a}^T \mathbf{x}^{(i)}$$

increases at rate $\|\mathbf{x}^{(i)}\|_2^2$.[14] Similarly, if we move in direction of $-\mathbf{x}^{(i)}$, then the prediction of the current linear function on $\mathbf{x}^{(i)}$ decreases at this rate. Thus, if $E_i(\mathbf{a}) < 0$, so that the prediction $h_{\mathbf{a}}(\mathbf{x}^{(i)})$ of the current linear function underestimates the correct value $y^{(i)}$, then data point i's "vote" is to move in the direction of $\mathbf{x}^{(i)}$ (increasing $h_{\mathbf{a}}$'s prediction), at a rate proportional to the magnitude $|E_i(\mathbf{a})|$ of the current error. Similarly, if $E_i(\mathbf{a}) > 0$, then data point i votes for changing $\mathbf{a}$ in the direction that would decrease $h_{\mathbf{a}}$'s prediction for $\mathbf{x}^{(i)}$ as rapidly as possible. Every data point has its own opinion, and the gradient (10) just averages these opinions. In addition to be conceptually transparent, computing this gradient is straightforward, requiring $O(md)$ time.

3.4 Optimizations for Large m

In many machine learning applications, one wants to use as much data as possible — as m grows larger, it's possible to get more and more accuracy and/or use richer and richer models. For very large m, it can be problematic to spend $O(md)$ time computing the gradient (10) in every single iteration of gradient descent. One simple approach to speeding up the gradient computation is to exploit the fact that (10) is a just a sum of terms, one per data point, enabling easy parallelization. The data set can be spread over however many machines or cores are available, the summands can be computed independently, and then the results are aggregated together.

Another idea, which works surprisingly well in many applications, is to use *stochastic gradient descent*. This optimization is relevant for objective functions f that are an average of many terms, like MSE and many other objective functions relevant to machine learning. The idea is super-simple: instead of asking all data points for their votes and averaging as in (10), we only ask a *single* randomly chosen data point for its opinion and use that as a rough proxy for the gradient. So instead of the update rule used in the algorithm in Section 2.5 — called *batch* gradient descent in this context (since the whole batch of data points is used for every update) — we do the following each iteration:

- Choose $i \in \{1, 2, \ldots, m\}$ uniformly at random.[15]

[14] In more detail, going from $\mathbf{a}$ to $\mathbf{a} + \gamma\mathbf{x}^{(i)}$ means we go from $h_{\mathbf{a}}(\mathbf{x}^{(i)}) = \mathbf{a}^T\mathbf{x}^{(i)}$ to $h_{(\mathbf{a}+\gamma\mathbf{x}^{(i)})}(\mathbf{x}^{(i)}) = (\mathbf{a} + \gamma\mathbf{x}^{(i)})^T\mathbf{x}^{(i)} = \mathbf{a}^T\mathbf{x}^{(i)} + \gamma(\mathbf{x}^{(i)})^T\mathbf{x}^{(i)} = \mathbf{a}^T\mathbf{x}^{(i)} + \gamma\|\mathbf{x}^{(i)}\|_2^2$.

[15] In practice, it's usually easier to just order the data points, randomly or arbitrarily, and go over them repeatedly in this order, using only one data point per iteration.

- $\mathbf{a} := \mathbf{a} - \underbrace{\alpha}_{\text{step size}} \cdot E_i(\mathbf{a}) \cdot \mathbf{x}^{(i)}$.

Switching from batch to stochastic gradient descent reduces the per-iteration time from $O(md)$ to $O(d)$. Of course, since we're now using only a very rough approximation of the true gradient each iteration, we might expect the number of iterations needed to converge to a good solution to be larger than before (if we converge at all). In many cases, especially when m is very large, this trade-off winds up being a big win.

3.5 Optimizations for Small m

The problem of minimizing the MSE over all linear functions $h_{\mathbf{a}}$ actually has a closed-form solution, namely

$$\mathbf{a} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}, \tag{11}$$

where $\mathbf{X}$ is the $m \times (d+1)$ matrix of the $\mathbf{x}^{(i)}$'s (with the "phantom coordinate" $x_0^{(i)} = 1$ for all i), and $\mathbf{y}$ denotes the m-vector of $y^{(i)}$'s. The $(d+1)$ equations in (11) are called the *normal equations*; the derivation is simple calculus (setting the derivative of MSE($\mathbf{a}$) to 0 and solving).

Given that one can solve exactly for the optimal coefficients $\mathbf{a}$ using basic linear algebra, why would you ever compute an approximate solution using gradient descent? The answer depends on the problem size. If d and m are small enough that the computations required for (11) — which require time something like $\Theta(md^2 + d^3)$ — can be done quickly, then there's no reason not to do it. For larger d and m, linear-algebraic operations like matrix inversion are no longer tractable, while gradient descent — at least the stochastic version — remains fast. Another reason to keep gradient descent in your toolbox is that many other important machine learning and regression problems, such as logistic regression, do not admit closed-form solutions at all.