CS168: The Modern Algorithmic Toolbox
Lecture #5: Gradient Descent Basics

Tim Roughgarden & Gregory Valiant[*]

April 11, 2016

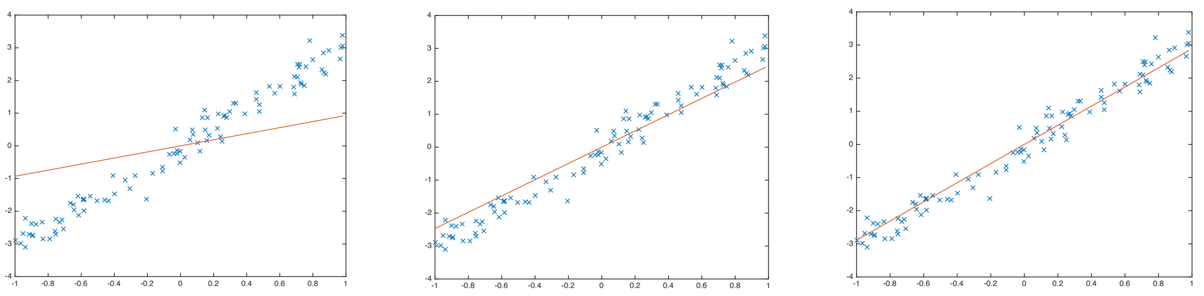
1 Preamble

Gradient descent is an extremely simple algorithm — simpler than most of the algorithms you studied in CS161 — that 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.

The first goal of this lecture is to develop the geometry and intuition behind gradient descent, to the point that the algorithm seems totally obvious. The second goal is to make the general method concrete with a case study on linear regression. (The method is also very useful for many other problems.) Next lecture we'll talk about extensions to the basic method and the basic problem formulation that will bring you a step closer to the state-of-the-art in modern machine learning.

Throughout this lecture, you might want to keep Figure 1 in mind. The figures show a succession of lines that are an increasingly good fit for a collection of points in the plane. "Linear regression" just means computing the best-fitting line, and this succession of lines is generated by successive iterations of gradient descent. This is not all supposed to make 100% sense yet, of course — the rest of the lecture explains what's going on — but this example should provide you with a concrete picture to refer back to as the lecture proceeds.

[*]©2015–2016, Tim Roughgarden and Gregory Valiant. Not to be sold, published, or distributed without the authors' consent.



(a) 0 Iterations (b) 1 Iteration (c) 2 Iterations

Figure 1: Using gradient descent to solve a linear regression problem. The goal is to compute the "best-fit" line to the given data points, and each iteration of gradient descent computes a successively better line.

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

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

subject to

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

Note that maximizing a function falls into this problem definition, since maximizing f is the same as minimizing $-f$. In this lecture, we'll always assume that f is differentiable and hence also continuous.[1] For example, in our linear regression case study (Section 3), $\mathbf{w}$ will encode a linear prediction function and f the mean-squared error of the function.

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 Figures 2 and 3. The intuition we'll develop for gradient descent in this simple case gives a surprisingly accurate picture of what's going on in the general case (for any number of dimensions).

What would it mean to try to minimize f via greedy local search? For example, in the parabola in Figure 2, if we start at the 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

[1]Gradient descent requires gradients! Well, actually it doesn't — there are extensions of gradient descent that relax the differentiability assumption, but we won't have time to discuss them.

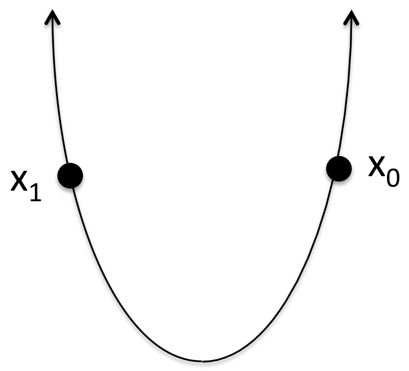


Figure 2: Minimizing a univariate objective function f via gradient descent.

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 either case, the algorithm terminates at the bottom of the basin.

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

Not all functions are as nice as the parabola in Figure 2, however. Consider the function shown in Figure 3. If we start at point x_0, then we go left and wind up at the bottom of the left basin. If we start at x_1, then we go right and the algorithm terminates at the bottom of the right basin.

Our two examples exhibit two obvious differences.

1. In Figure 2, no matter which starting point is chosen, the termination point of the algorithm remains the same. In Figure 3, where you end up depends on where you start.

2. In Figure 2, the algorithm always terminates at the global minimum. In Figure 3, the algorithm might terminate at a local minimum — meaning there's no way to improve f by moving a little bit in either direction — that is not a global minimum.

What is it about the functions that lead to these differences? The answer is *convexity*.[2] A function is convex if all chords of its graph only lie above the graph. It is visually clear

[2]Your math classes may or may not have emphasized the central importance of convexity. But a good rule of thumb, especially in optimization, is to equate convexity with "niceness," including efficient solvability.

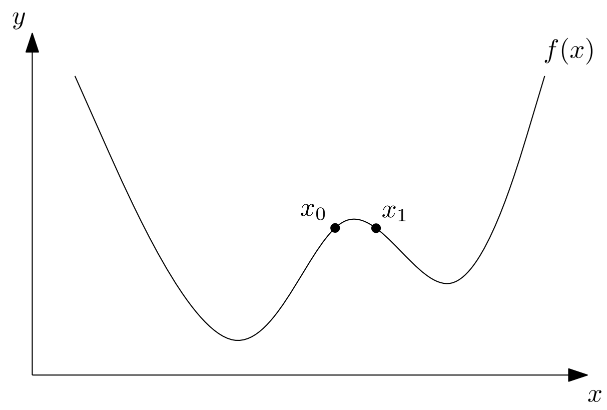


Figure 3: With a non-convex objective function, gradient descent can yield different local minima with different start states.

that the function in Figure 2 is convex while the function in Figure 3 is not. Mathematically, a function is convex if and only if

$$f(\tfrac{1}{2}\mathbf{w} + \tfrac{1}{2}\mathbf{z}) \leq \underbrace{\tfrac{1}{2}f(\mathbf{w}) + \tfrac{1}{2}f(\mathbf{z})}_{\text{midpoint of chord between } f(\mathbf{w}) \text{ and } f(\mathbf{z})}$$

for every $\mathbf{w}, \mathbf{z} \in \mathbb{R}^n$. That is, for points $\mathbf{w}$ and $\mathbf{z}$, if you take the average of $\mathbf{w}$ and $\mathbf{z}$ and then apply f, you'll get a smaller number than if you first apply f to $\mathbf{w}$ and $\mathbf{z}$ and then average the results. A seemingly stronger but in fact equivalent condition is

$$\underbrace{f(\lambda\mathbf{w} + (1-\lambda)\mathbf{z})}_{\text{graph}} \leq \underbrace{\lambda f(\mathbf{w}) + (1-\lambda)f(\mathbf{z})}_{\text{point on chord}} \tag{1}$$

for all $\mathbf{w}, \mathbf{z} \in \mathbb{R}^n$ and $\lambda \in [0,1]$. Note that these definitions makes sense for any function $f : \mathbb{R}^n \to \mathbb{R}$, not just in the $n = 1$ case. It's not always easy to check whether or not a given function is convex, but there is a mature analytical toolbox for this purpose (taught in EE364, for example).

Already with $n = 1$ and in Figure 3, we observed that with a non-convex function f, gradient descent can compute a local minimum that is worse (i.e., larger) than a global minimum. The converse also holds: if f is convex, then gradient descent can only terminate at a global minimum.[3] 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 $f(x^*)$. Inequality (1) then implies that moving toward x^* from x can only decrease f. Thus gradient descent will not get stuck at x.

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

2.3 Warm-Up #2: Linear Functions

In almost all of the 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 $\mathbf{w} \in \mathbb{R}^n$, there's an infinite number of directions in which we could move, not just 2.

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

$$f(\mathbf{w}) = \mathbf{c}^T\mathbf{w} + b, \tag{2}$$

where $\mathbf{c} \in \mathbb{R}^n$ is an n-vector and $b \in \mathbb{R}$ is a scalar.

Unconstrained minimization of a linear function is a trivial problem, because (assuming $\mathbf{c} \neq 0$) it is possible to make the objective function arbitrarily negative. To see this, take any vector $\mathbf{w}$ with negative inner product $\mathbf{c}^T\mathbf{w} < 0$ with $\mathbf{c}$ (such as $-\mathbf{c}$) and consider points of the form $\beta\mathbf{w}$ for β arbitrarily large.

Suppose you are currently at a point $\mathbf{w} \in \mathbb{R}^n$, and you are allowed to move at most one unit of Euclidean distance in whatever direction you want. Where should you go to decrease the function f in (2) 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 $\mathbf{w}$ one unit of distance in the direction $\mathbf{u}$ changes the objective function as follows:

$$\begin{aligned}
\mathbf{c}^T\mathbf{w} + b \quad &\mapsto \quad \mathbf{c}^T(\mathbf{w} + \mathbf{u}) + b && (3)\\
&= \quad \mathbf{c}^T\mathbf{w} + b + \mathbf{c}^T\mathbf{u} && (4)\\
&= \quad \underbrace{\mathbf{c}^T\mathbf{w} + b}_{\text{independent of } \mathbf{u}} + \|\mathbf{c}\|_2 \underbrace{\|\mathbf{u}\|_2}_{=1} \cos\theta, && (5)
\end{aligned}$$

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 (3)–(5) 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.

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 in Figure 3, 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

$$f(x+\epsilon) = \underbrace{f(x) + \epsilon \cdot f'(x)}_{\text{linear approx.}} + \tfrac{\epsilon^2}{2!} \cdot f''(x) + \tfrac{\epsilon^3}{3!} \cdot f''(x) + \cdots . \tag{6}$$

So what? The point is that with the first two terms on the right-hand side of (6), 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 $y - x$.[4]

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 (6) remains valid in higher dimensions, just with the derivatives replaced by their higher dimensional analogs. Since we'll use only linear approximations, 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* $\nabla f(\mathbf{w})$ of f at $\mathbf{w}$ is the real-valued n-vector

$$\nabla f(\mathbf{w}) = \left(\frac{\partial f}{\partial w_1}(\mathbf{w}), \frac{\partial f}{\partial w_2}(\mathbf{w}), \ldots, \frac{\partial f}{\partial w_n}(\mathbf{w})\right) \tag{7}$$

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

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

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

[4]Note the analogy with some of our "lossy compression" solutions — we're throwing out as much information as possible subject to some type of approximation guarantee.

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

$$f(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}w_iw_j - \sum_{i=1}^{n} b_iw_i,$$

and you should check that

$$\frac{\partial f}{\partial w_i}(\mathbf{w}) = \sum_{j=1}^{n} a_{ij}w_j - b_i$$

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

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

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

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 (7) 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 — $\mathbf{w}_0$, ϵ, and α — which we'll elaborate on shortly.

Gradient Descent

```
initialize $\mathbf{w} := \mathbf{w}_0$
**while** $\|\nabla f(\mathbf{w})\|_2 > \epsilon$ **do**
        $\mathbf{w} := \mathbf{w} - \underbrace{\alpha}_{\text{step size}} \cdot \nabla f(\mathbf{w})$        (8)
```

And that's it!

It's also worth zooming in to see what the update rule (8) looks like in some coordinate, say the jth one:

$$w_j := w_j - \alpha \cdot \frac{\partial f}{\partial w_j}(\mathbf{w}). \tag{9}$$

The update (8) can be thought of as n updates of the form (9) being done in parallel (one per coordinate j).

Conceptually, gradient descent enters the following contract with basic calculus:

1. Calculus promises that, for $\mathbf{z}$ close to $\mathbf{w}$, one can pretend that the true function f is just the linear function $f(\mathbf{w}) + \nabla f(\mathbf{w})^T(\mathbf{z} - \mathbf{w})$ (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 $-\nabla f(\mathbf{w})$ — to locally decrease the function at a rate of $\|\nabla f(\mathbf{w})\|_2$ 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 $\mathbf{w}$. Taking a large step would violate the agreement, in that far away from $\mathbf{w}$ the function $f(\mathbf{w})$ need not behave anything like the local linear approximation $f(\mathbf{w}) + \nabla f(\mathbf{w})^T(\mathbf{z} - \mathbf{w})$ (recall the tangent lines in Figure 3).

The starting point $\mathbf{w}_0$ 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 $\mathbf{w}_0$. For convex functions f, gradient descent will converge toward the same point — the global minimum — no matter how the starting state is chosen.[5] The choice of start state can still affect the number of iterations until convergence, however. In practice, one should choose $\mathbf{w}_0$ according to your best guess as to where the global minimum is likely be — generally, the closer $\mathbf{w}_0$ 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."[6] Since the rate of decrease of a given step is $\|\nabla f(\mathbf{w})\|_2$, at least locally, once $\|\nabla f(\mathbf{w})\|_2$ 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.[7] 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 minimizes f over the line $\mathbf{w} - \alpha \cdot \nabla f(\mathbf{w})$. 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.

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

[6]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.

[7]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.

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.

3.1 Linear Regression

In linear regression, the input is m data points $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)} \in \mathbb{R}^n$, each an n-dimensional vector. Also given is a real-valued "label" $y^{(i)} \in \mathbb{R}$ for each data point i.[8] 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 n 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 $h : \mathbb{R}^n \to \mathbb{R}$ such that $h(\mathbf{x}^{(i)}) \approx y^{(i)}$ for every i. Every such linear function h can be written as

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + \sum_{j=1}^{n} w_j x_j$$

for real-valued coefficients $w_0, w_1, \ldots, w_n$. We can simplify the notation by giving every data point a "dummy zeroth coordinate" equal to 1. Then, the coefficient of the dummy coordinate plays the role of the intercept w_0. From now on, we assume that the data points have been preprocessed in this way, and that the coordinates are named $\{1, 2, \ldots, n\}$. We then associate $\mathbf{w} \in \mathbb{R}^n$ with the linear function

$$h_{\mathbf{w}}(\mathbf{x}) = \sum_{j=1}^{n} w_j x_j. \tag{10}$$

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 n features is much more strongly correlated with the label $y^{(i)}$ than any of the others.

[8]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).

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

$$\text{MSE}(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} E_i(\mathbf{w})^2 \tag{11}$$

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

$$E_i(\mathbf{w}) = h_{\mathbf{w}}(\mathbf{x}^{(i)}) - y^{(i)}. \tag{12}$$

There are a couple of reasons to choose the MSE objective function.[9] 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 $h_{\mathbf{w}}$ and then adding independent Gaussian noise to each data point, then minimizing the MSE is equivalent to the problem of maximizing (over linear functions) the likelihood of the data.[10]

Since we want to minimize the mean-squared error, our function $f : \mathbb{R}^n \to \mathbb{R}$ is that in (11).[11] In this minimization problem, the variables are the coefficients $\mathbf{w}$ of the linear function $h_{\mathbf{w}}$ — 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 $\mathbf{w}$. The rough argument is: each function $E_i(\mathbf{w})$ is linear in $\mathbf{w}$, and linear functions are convex; taking the square only makes these functions "more convex;" and the sum (11) 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 (11) — it turns out to be quite nice and interpretable. Recall that derivatives are linear — for example, $(g+h)' = g'+h'$. Since (11) has one term per data point $i = 1, 2, \ldots, m$, the gradient will also have one term per data point. This is a key point: the fact that the gradient separates over the data points is a big reason why gradient descent can scale to very machine learning problems.

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

[10]Even though reality may not conform to the precise assumption of independent Gaussian noise, a result like this provides evidence that this approach should give good results quite generally.

[11]Omitting the normalizing term $1/m$ in (11) results in an equivalent optimization problem. Note that the best step size to use in gradient descent will depend on whether or not this term in included.

The term for the ith data point is, by the chain rule of calculus,

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

Inspecting (10) and (12), we have

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

for $j = 1, 2, \ldots, n$, and hence $\nabla E_i(\mathbf{w}) = \mathbf{x}$. Putting it all together, we have

$$\nabla f(\mathbf{w}) = \frac{2}{m} \sum_{i=1}^{m} \left(\underbrace{E_i(\mathbf{w})}_{\text{scalar}} \cdot \underbrace{\mathbf{x}^{(i)}}_{n\text{-vector}} \right), \tag{13}$$

where $f(\mathbf{w})$ denotes the MSE of the linear function $h_{\mathbf{w}}$.

The gradient (13) has a natural interpretation. The ith term $E_i(\mathbf{w}) \cdot \mathbf{x}^{(i)}$ can be thought of as the ith data point's "opinion" as to how the coefficients $\mathbf{w}$ should be updated. To see this, first note that if we move from $\mathbf{w}$ in the direction of $\mathbf{x}^{(i)}$, then the prediction of the current linear function

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

increases at rate $\|\mathbf{x}^{(i)}\|_2^2$.[12] Similarly, if we move in the direction of $-\mathbf{x}^{(i)}$, then the prediction of the current linear function on $\mathbf{x}^{(i)}$ decreases at this rate. Thus, if $E_i(\mathbf{w}) < 0$, so that the prediction $h_{\mathbf{w}}(\mathbf{x}^{(i)})$ 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 $h_{\mathbf{w}}$'s prediction), at a rate proportional to the magnitude $|E_i(\mathbf{w})|$ of the current error. Similarly, if $E_i(\mathbf{w}) > 0$, then data point i votes for changing $\mathbf{w}$ in the direction that would decrease $h_{\mathbf{w}}$'s prediction for $\mathbf{x}^{(i)}$ as rapidly as possible. Every data point has its own opinion, and the gradient (13) just averages these opinions. In addition to be conceptually transparent, computing this gradient is straightforward, requiring $O(mn)$ time. And since (13) is a just a sum of terms, one per data point, the computation is easy to parallelize. 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.

4 Lecture Take-Aways

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.

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

2. Gradient descent scales well to large data sets, especially with some tweaks (covered next lecture) 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, like linear regression, 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 #3.