CHAPTER 8
Gradient Descent

Those who boast of their descent, brag on what they owe to others.
—Seneca

Frequently when doing data science, we'll be trying to the find the best model for a certain situation. And usually "best" will mean something like "minimizes the error of its predictions" or "maximizes the likelihood of the data." In other words, it will represent the solution to some sort of optimization problem.

This means we'll need to solve a number of optimization problems. And in particular, we'll need to solve them from scratch. Our approach will be a technique called *gradient descent*, which lends itself pretty well to a from-scratch treatment. You might not find it super-exciting in and of itself, but it will enable us to do exciting things throughout the book, so bear with me.

The Idea Behind Gradient Descent

Suppose we have some function `f` that takes as input a vector of real numbers and outputs a single real number. One simple such function is:

```
from scratch.linear_algebra import Vector, dot

def sum_of_squares(v: Vector) -> float:
    """Computes the sum of squared elements in v"""
    return dot(v, v)
```

We'll frequently need to maximize or minimize such functions. That is, we need to find the input `v` that produces the largest (or smallest) possible value.

For functions like ours, the *gradient* (if you remember your calculus, this is the vector of partial derivatives) gives the input direction in which the function most quickly

increases. (If you don't remember your calculus, take my word for it or look it up on the internet.)

Accordingly, one approach to maximizing a function is to pick a random starting point, compute the gradient, take a small step in the direction of the gradient (i.e., the direction that causes the function to increase the most), and repeat with the new starting point. Similarly, you can try to minimize a function by taking small steps in the *opposite* direction, as shown in Figure 8-1.

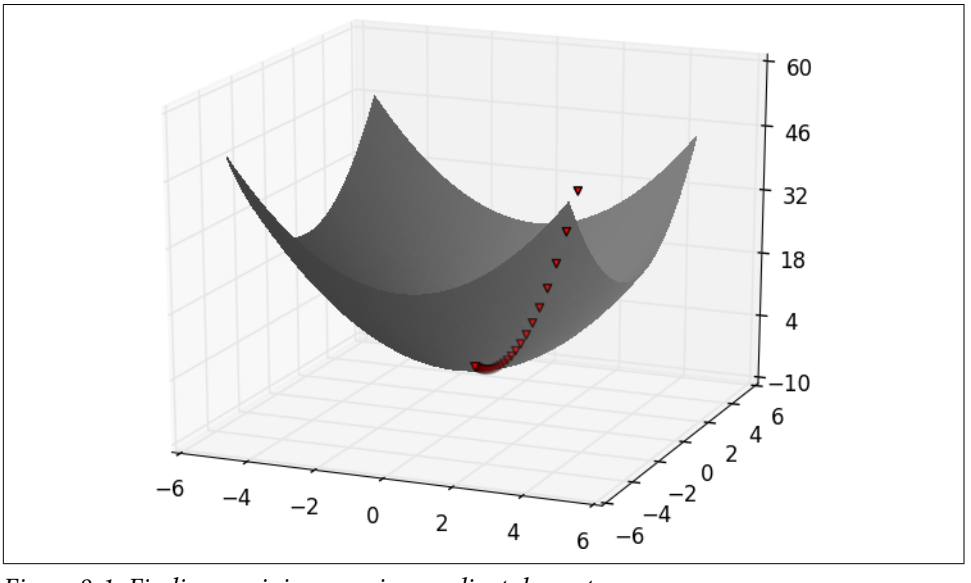

Figure 8-1. Finding a minimum using gradient descent

If a function has a unique global minimum, this procedure is likely to find it. If a function has multiple (local) minima, this procedure might "find" the wrong one of them, in which case you might rerun the procedure from different starting points. If a function has no minimum, then it's possible the procedure might go on forever.

Estimating the Gradient

If `f` is a function of one variable, its derivative at a point `x` measures how `f(x)` changes when we make a very small change to `x`. The derivative is defined as the limit of the difference quotients:

```
from typing import Callable

def difference_quotient(f: Callable[[float], float],
                        x: float,
```

```
                        h: float) -> float:
    return (f(x + h) - f(x)) / h
```

as h approaches zero.

(Many a would-be calculus student has been stymied by the mathematical definition of limit, which is beautiful but can seem somewhat forbidding. Here we'll cheat and simply say that "limit" means what you think it means.)

The derivative is the slope of the tangent line at $(x, f(x))$, while the difference quotient is the slope of the not-quite-tangent line that runs through $(x + h, f(x + h))$. As h gets smaller and smaller, the not-quite-tangent line gets closer and closer to the tangent line (Figure 8-2).

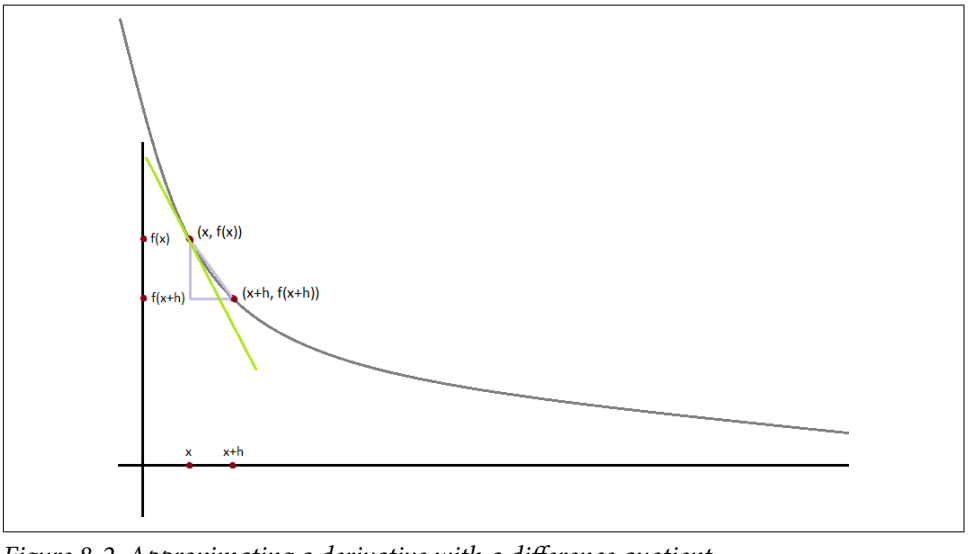

Figure 8-2. Approximating a derivative with a difference quotient

For many functions it's easy to exactly calculate derivatives. For example, the `square` function:

```
def square(x: float) -> float:
    return x * x
```

has the derivative:

```
def derivative(x: float) -> float:
    return 2 * x
```

which is easy for us to check by explicitly computing the difference quotient and taking the limit. (Doing so requires nothing more than high school algebra.)

What if you couldn't (or didn't want to) find the gradient? Although we can't take limits in Python, we can estimate derivatives by evaluating the difference quotient for a very small `e`. Figure 8-3 shows the results of one such estimation:

```
xs = range(-10, 11)
actuals = [derivative(x) for x in xs]
estimates = [difference_quotient(square, x, h=0.001) for x in xs]

# plot to show they're basically the same
import matplotlib.pyplot as plt
plt.title("Actual Derivatives vs. Estimates")
plt.plot(xs, actuals, 'rx', label='Actual')       # red  x
plt.plot(xs, estimates, 'b+', label='Estimate')   # blue +
plt.legend(loc=9)
plt.show()
```

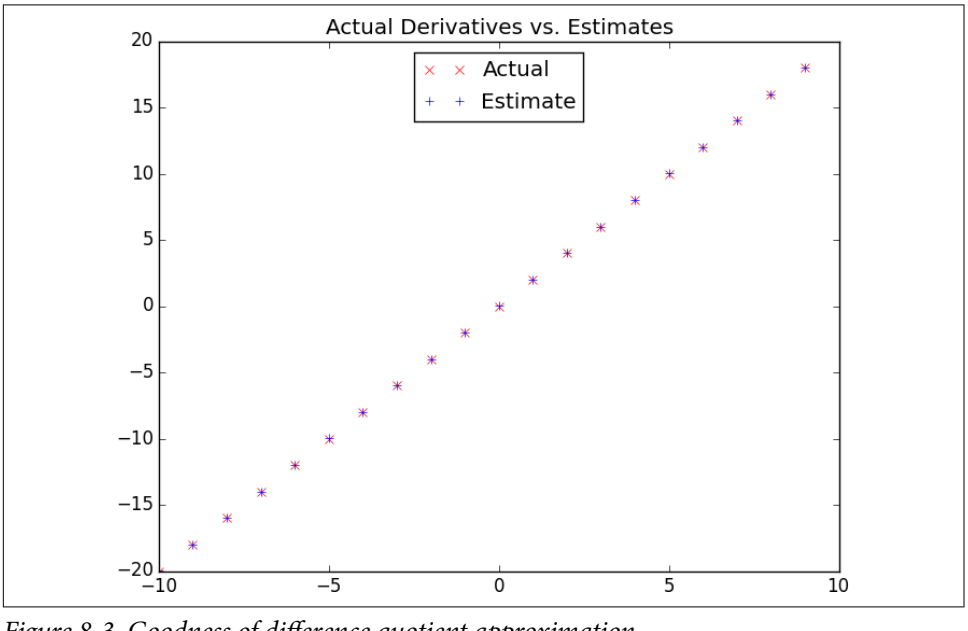

Figure 8-3. Goodness of difference quotient approximation

When `f` is a function of many variables, it has multiple *partial derivatives*, each indicating how `f` changes when we make small changes in just one of the input variables.

We calculate its *i*th partial derivative by treating it as a function of just its *i*th variable, holding the other variables fixed:

```
def partial_difference_quotient(f: Callable[[Vector], float],
                                v: Vector,
                                i: int,
                                h: float) -> float:
    """Returns the i-th partial difference quotient of f at v"""
```

```
    w = [v_j + (h if j == i else 0)    # add h to just the ith element of v
         for j, v_j in enumerate(v)]

    return (f(w) - f(v)) / h
```

after which we can estimate the gradient the same way:

```
def estimate_gradient(f: Callable[[Vector], float],
                      v: Vector,
                      h: float = 0.0001):
    return [partial_difference_quotient(f, v, i, h)
            for i in range(len(v))]
```

A major drawback to this "estimate using difference quotients" approach is that it's computationally expensive. If `v` has length n, `estimate_gradient` has to evaluate `f` on $2n$ different inputs. If you're repeatedly estimating gradients, you're doing a whole lot of extra work. In everything we do, we'll use math to calculate our gradient functions explicitly.

Using the Gradient

It's easy to see that the `sum_of_squares` function is smallest when its input `v` is a vector of zeros. But imagine we didn't know that. Let's use gradients to find the minimum among all three-dimensional vectors. We'll just pick a random starting point and then take tiny steps in the opposite direction of the gradient until we reach a point where the gradient is very small:

```
import random
from scratch.linear_algebra import distance, add, scalar_multiply

def gradient_step(v: Vector, gradient: Vector, step_size: float) -> Vector:
    """Moves `step_size` in the `gradient` direction from `v`"""
    assert len(v) == len(gradient)
    step = scalar_multiply(step_size, gradient)
    return add(v, step)

def sum_of_squares_gradient(v: Vector) -> Vector:
    return [2 * v_i for v_i in v]

# pick a random starting point
v = [random.uniform(-10, 10) for i in range(3)]

for epoch in range(1000):
    grad = sum_of_squares_gradient(v)    # compute the gradient at v
    v = gradient_step(v, grad, -0.01)    # take a negative gradient step
    print(epoch, v)

assert distance(v, [0, 0, 0]) < 0.001    # v should be close to 0
```

If you run this, you'll find that it always ends up with a `v` that's very close to `[0,0,0]`. The more epochs you run it for, the closer it will get.

Choosing the Right Step Size

Although the rationale for moving against the gradient is clear, how far to move is not. Indeed, choosing the right step size is more of an art than a science. Popular options include:

- Using a fixed step size
- Gradually shrinking the step size over time
- At each step, choosing the step size that minimizes the value of the objective function

The last approach sounds great but is, in practice, a costly computation. To keep things simple, we'll mostly just use a fixed step size. The step size that "works" depends on the problem—too small, and your gradient descent will take forever; too big, and you'll take giant steps that might make the function you care about get larger or even be undefined. So we'll need to experiment.

Using Gradient Descent to Fit Models

In this book, we'll be using gradient descent to fit parameterized models to data. In the usual case, we'll have some dataset and some (hypothesized) model for the data that depends (in a differentiable way) on one or more parameters. We'll also have a *loss* function that measures how well the model fits our data. (Smaller is better.)

If we think of our data as being fixed, then our loss function tells us how good or bad any particular model parameters are. This means we can use gradient descent to find the model parameters that make the loss as small as possible. Let's look at a simple example:

```
# x ranges from -50 to 49, y is always 20 * x + 5
inputs = [(x, 20 * x + 5) for x in range(-50, 50)]
```

In this case we *know* the parameters of the linear relationship between `x` and `y`, but imagine we'd like to learn them from the data. We'll use gradient descent to find the slope and intercept that minimize the average squared error.

We'll start off with a function that determines the gradient based on the error from a single data point:

```
def linear_gradient(x: float, y: float, theta: Vector) -> Vector:
    slope, intercept = theta
    predicted = slope * x + intercept    # The prediction of the model.
    error = (predicted - y)              # error is (predicted - actual).
```

```
    squared_error = error ** 2          # We'll minimize squared error
    grad = [2 * error * x, 2 * error]   # using its gradient.
    return grad
```

Let's think about what that gradient means. Imagine for some `x` our prediction is too large. In that case the `error` is positive. The second gradient term, `2 * error`, is positive, which reflects the fact that small increases in the intercept will make the (already too large) prediction even larger, which will cause the squared error (for this `x`) to get even bigger.

The first gradient term, `2 * error * x`, has the same sign as `x`. Sure enough, if `x` is positive, small increases in the slope will again make the prediction (and hence the error) larger. If `x` is negative, though, small increases in the slope will make the prediction (and hence the error) smaller.

Now, that computation was for a single data point. For the whole dataset we'll look at the *mean squared error*. And the gradient of the mean squared error is just the mean of the individual gradients.

So, here's what we're going to do:

1. Start with a random value for `theta`.
2. Compute the mean of the gradients.
3. Adjust `theta` in that direction.
4. Repeat.

After a lot of *epochs* (what we call each pass through the dataset), we should learn something like the correct parameters:

```
from scratch.linear_algebra import vector_mean

# Start with random values for slope and intercept
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

learning_rate = 0.001

for epoch in range(5000):
    # Compute the mean of the gradients
    grad = vector_mean([linear_gradient(x, y, theta) for x, y in inputs])
    # Take a step in that direction
    theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1,   "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Minibatch and Stochastic Gradient Descent

One drawback of the preceding approach is that we had to evaluate the gradients on the entire dataset before we could take a gradient step and update our parameters. In this case it was fine, because our dataset was only 100 pairs and the gradient computation was cheap.

Your models, however, will frequently have large datasets and expensive gradient computations. In that case you'll want to take gradient steps more often.

We can do this using a technique called *minibatch gradient descent*, in which we compute the gradient (and take a gradient step) based on a "minibatch" sampled from the larger dataset:

```
from typing import TypeVar, List, Iterator

T = TypeVar('T')  # this allows us to type "generic" functions

def minibatches(dataset: List[T],
                batch_size: int,
                shuffle: bool = True) -> Iterator[List[T]]:
    """Generates `batch_size`-sized minibatches from the dataset"""
    # start indexes 0, batch_size, 2 * batch_size, ...
    batch_starts = [start for start in range(0, len(dataset), batch_size)]

    if shuffle: random.shuffle(batch_starts)  # shuffle the batches

    for start in batch_starts:
        end = start + batch_size
        yield dataset[start:end]
```

The `TypeVar(T)` allows us to create a "generic" function. It says that our `dataset` can be a list of any single type—`str`s, `int`s, `list`s, whatever—but whatever that type is, the outputs will be batches of it.

Now we can solve our problem again using minibatches:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

for epoch in range(1000):
    for batch in minibatches(inputs, batch_size=20):
        grad = vector_mean([linear_gradient(x, y, theta) for x, y in batch])
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1,   "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

Another variation is *stochastic gradient descent*, in which you take gradient steps based on one training example at a time:

```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

for epoch in range(100):
    for x, y in inputs:
        grad = linear_gradient(x, y, theta)
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1,   "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

On this problem, stochastic gradient descent finds the optimal parameters in a much smaller number of epochs. But there are always tradeoffs. Basing gradient steps on small minibatches (or on single data points) allows you to take more of them, but the gradient for a single point might lie in a very different direction from the gradient for the dataset as a whole.

In addition, if we weren't doing our linear algebra from scratch, there would be performance gains from "vectorizing" our computations across batches rather than computing the gradient one point at a time.

Throughout the book, we'll play around to find optimal batch sizes and step sizes.

The terminology for the various flavors of gradient descent is not uniform. The "compute the gradient for the whole dataset" approach is often called *batch gradient descent*, and some people say *stochastic gradient descent* when referring to the minibatch version (of which the one-point-at-a-time version is a special case).

For Further Exploration

- Keep reading! We'll be using gradient descent to solve problems throughout the rest of the book.
- At this point, you're undoubtedly sick of me recommending that you read textbooks. If it's any consolation, *Active Calculus 1.0*, by Matthew Boelkins, David Austin, and Steven Schlicker (Grand Valley State University Libraries), seems nicer than the calculus textbooks I learned from.
- Sebastian Ruder has an epic blog post comparing gradient descent and its many variants.