Gradient Descent

Eckel, TJHSST AI2, Spring 2022

Background & Explanation

We're going to take just a moment to talk about a concept from multivariable calculus.

It's hard to explain why this is important without just going through the whole algorithm, but trust me, this is an essential component of understanding back propagation!

The Idea of a Gradient

Some of you are in multivariable calculus. If so, you probably understand this sentence: "the opposite of the gradient of a function is the direction of steepest descent". If you aren't in multivariable calculus, or you don't understand that sentence, go watch the linked video on the course website!

Required Task

I want you to write code to find minimum values of certain functions.

The algorithm here is to choose a learning rate, λ, and then:

- Calculate the opposite of the gradient at the current location. This vector represents the direction to move to achieve the steepest descent from the current location.
- Take the current position and add λ times the negative gradient. In other words, if we represent the position after *n* steps as the vector x_n, we are saying $x_n = x_{n-1} - \lambda \nabla f(x_{n-1})$ This represents making a small step in the direction of the negative gradient vector.
- Repeat until you get acceptably close to a local minimum.

How do we know we're acceptably close to a local minimum? Well, as with any local minimum or maximum, the derivative will get close to zero. Let's say less than 10^{-8}. When magnitude of the gradient function is less than 10^{-8}, we'll stop. (Refer back to the numpy demo file for calculating magnitude.)

How do we choose lambda? This is an excellent question. For the required lab, experiment and come up with something that works. For the extension, on the back, I have a better answer for you!

The functions I want you to minimize are:

- Function A: $f(x, y) = 4x^2 - 3xy + 2y^2 + 24x - 20y$
- Function B: $f(x, y) = (1 - y)^2 + (x - y^2)^2$

Please, by all means, hardcode these functions and their partial derivatives! You don't even have to figure out their partial derivatives by hand; just copy them from Wolfram Alpha! I am *definitely not* asking you to write code that will calculate partial derivative functions from scratch.

Your code should:

- Take *a single command line argument*, either "A" or "B". It should minimize the respective function shown above.
- Start at (0,0).
- Print out, at every step in the algorithm, the current location and the current gradient vector.

Specification for GREEN credit

Submit **a single python script** to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code does **exactly as specified in the previous section; please read carefully!**

RED Extension: 1-d Line Optimization

If you know the derivative of a one-variable function, finding a minimum is easy; you've done it in calculus a lot. As it happens, though, there is a lovely algorithm for finding the minimum on a one-variable function, the derivative of which is *not* known. We must assume that there is *one* local minimum value on a particular interval, and then the algorithm goes like this:

- Calculate output values 1/3 and 2/3 of the way across the interval.
- If the value at 1/3 is greater than the value at 2/3, then the minimum can't be between the left endpoint and the 1/3 mark. (Do you see why?) So: the 1/3 mark is the new left endpoint.
- Alternately, if the value at 2/3 is greater than the value at 1/3, then the minimum can't be between the 2/3 mark and the right endpoint. So: the 2/3 mark is the new right endpoint.

In this way, keep making the interval smaller until you arrive at a small enough difference between the left and right endpoints (again, 10^{-8} will work for our purposes).

Write a function `one_d_minimize(f, left, right, tolerance)` that takes a function as an argument along with left and right bounds and a tolerance. This method is recursive. For the base case, check if `right - left < tolerance`, at which point it should simply return the average of `right` and `left`. Otherwise, it should find new points 1/3 and 2/3 of the way between `left` and `right`, and based on which of those points has a larger output value on `f`, either recursively call from `left` to the 2/3 point or recursively call from the 1/3 point to `right`.

Test this by minimizing the function $f(x) = \sin(x) + \sin(3x) + \sin(4x)$ on the interval [-1,0]. It should find the minimum at approximately -.476.

Note again that we are making a big assumption here – that there is only *one* local minimum on the interval in question. (A quick sketch can probably convince you that if the function has more than one local minimum on an interval, this algorithm will find one of them but not necessarily the lowest.) In practice, this is not much of a hurdle; often, finding only one of the minima will suffice anyway, and/or we can restrict where we're searching so that the assumption that only one minimum is present can be made without difficulty. Regardless, it is an important idea to keep in mind!

In any case, now we can use this idea to determine the best λ for each step in the minimization algorithm!

Imagine the x/y plane is underneath your feet, and the value of a function we wish to minimize forms a curved surface above it. The idea here is that we get the negative gradient, and that is the *direction* in which we'll make the next step. So imagine turning to face that direction, and then drawing a line along the x/y plane in the direction you're facing, and considering only the values of the function along that line. This makes a curve that can be minimized using the above

algorithm! Specifically, pick a point somewhere out on that line, and then run this line optimization algorithm *on that line* until we find the minimum value that our function has along it. That is how far we should step in this direction.

Accomplishing this in Python involves one of my favorite tricks, called a closure. Consider this code:

```
def make_funct(a, b):
    def funct(x):
        return a*x + b
    return funct

f = make_funct(2, 3)
g = make_funct(4, 5)

print(f(7), g(7))
```

This is called a closure; we pass in variables that become "closed" in a new function definition *and then we return the function.* Use this technique to create a method that "closes" the values of ∇f and x_0 on a given f, making a new *one-variable function on lambda* to represent $f\left(x_0 - \lambda \nabla f(x_0)\right)$.

If that doesn't immediately make sense, try this explanation on for size. Basically, once we've decided the direction we want to point, we are no longer varying where we are or which direction we're moving; now, the only variable is how far we want to move in that direction. So we're building a new custom-made function that will store our location and the direction we're pointing, then take the one variable, lambda, and return the output value of the original function at that distance from our current location in the direction it has stored.

Now: we have a one-variable function, and we can pass this to the line optimization function we wrote before!

Your task: match the assignment's standard specification, but include in your gradient descent function the line optimization described above. It should execute in many fewer steps than the original code! Put a comment where you've added the line optimization so I can see that it's there.

Specification for RED credit for 1-d Line Optimization

Submit **a single python script** to the link on the course website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- Your code does **exactly as specified in the previous section; please read carefully!**