Optimization Theory (DS2) Lecture #12 Basics of Non-Linear Programs: Finding a Min/Max of a Function (Derivatives, Gradient Descent, Newton's Method)

December 26, 2016

Abstract

Introduction to the basic ideas of optimizing non-linear functions, meaning things with more complex objective functions than simple, linear functions where the maximum is along some edge of the space of the possible. This will prepare you to understand deep learning, one of today's trendiest research topics.

1 Derivative of a Function

I hope most of you are familiar with the basic idea of derivatives. For simple functions of a single variable x, the derivative is the *slope* of the function. Take the derivative and evaluate for your chosen value of x, and you'll have the slope of a line tangent to the curve at that point. Take the derivative of the derivative, known as the *second derivative*, and you have the *curvature*.

1.1 Taking the Derivative

For simple polynomials, each term can be differentiated independently,

$$\frac{d}{dx}x^n = nx^{n-1}, \tag{1}$$

so the derivative of

$$f(x) = -x^2 + 3x + 2 \tag{2}$$

is

$$f'(x) = \frac{df}{dx} = -2x + 3. \tag{3}$$

For basic trig functions,

$$\frac{d}{dx}\sin(x) = \cos(x) \tag{4}$$

$$\frac{d}{dx}\cos(x) = -\sin(x) \tag{5}$$

and for exponentionals and logs,

$$\frac{d}{dx}e^x = e^x \tag{6}$$

$$\frac{d}{dx}a^x = a^x\ln(a) \tag{7}$$

$$\frac{d}{dx}\ln(x) = \frac{1}{x}. \tag{8}$$

Finally, we'll need the chain rule

$$h(x) = f(g(x)) \tag{9}$$

$$\frac{d}{dx}h(x) = \frac{d}{dx}f(g(x)) = f'(g(x))g'(x), \tag{10}$$

e.g.

$$\frac{d}{dx}e^{2x} = 2e^{2x} \tag{11}$$

$$\frac{d}{dx}\sin(x^2) = 2x\cos(x^2) \tag{12}$$

and the product rule

$$h(x) = f(x)g(x) \tag{13}$$

$$\frac{d}{dx}h(x) = \frac{d}{dx}f(x)g(x) = f'(x)g(x) + f(x)g'(x). \tag{14}$$

All simple, right? That should be more than you need here.

1.2 Using the Derivative to Find Maxima/Minima/Inflection Points

A positive curvature means that you have a *local minimum* "cup" or "valley" shape, a negative curvature means you have a *local maximum* "cap" or "ridge"/"hill" shape, and a zero tells you nothing; probably it's an *inflection point*, where the direction of the curvature is changing.

1.3 Local v. Global Minima

So far, our work has involved linear programs and integer programs, which have only a single minimum (or maximum, if that's the way you're running) value, though that value might appear for multiple input values; in fact, for linear programs, it might be an edge of the polytope, so that every point along that edge has the same value for the objective function. (We'll learn more about functions with many optimal points in the last week.)

A function may have more than one valley "bottom", known as *minima*. A minimum that is not the minimum across all possible values is a *local minimum*, and one that is minimum for every possible input value is a *global minimum*.

For example, the polynomial $0.07x^4 + x^3 + 3x^2 - 2x + 1$ can be plotted in R over the range $[-10, 2]$ using

```
x <- seq(-10,2, length=100)
plot(x,0.07*x^4+x^3+3*x^2-2*x+1, "l")
```

and you can see that there is a local minimum just above $x = 0$, and a global minimum down around $x = -8$.

Finding those minima for a simple polynomial of a single variable, like this one, is not so hard, but the functions we are dealing with are mostly over many input variables, which may give us too large a space to find solutions analytically (heck, we saw how hard it was for simple linear functions when we did simplex). Generally, we are looking for the global minimum, but in general that is an exponentially hard problem, so we look instead for heuristics that give us an answer that is within some distance of the global optimum. Today we are beginning our investigation of techniques for such non-linear problems.

2 Scalar Fields and Vector Fields

A scalar field is the altitude. A vector field is the slope; the length of the vector is the magnitude of the slope (steepest change in any direction), and the direction is (naturally) the direction in which the change is the steepest (downhill, for our purposes here).

2.1 Scalar Field Examples

Examples of scalar fields:

- altitude on a 2-D map
- temperature in a location (location can be 2-D on a surface or 3-D in space)
- air pressure (naturally it's the pressure in three dimensions, but on weather maps is usually reported as a single value for a latitude-longitude location).

Note that the number of dimensions of a function is the number of *arguments* to the function (or equivalent number of dimensions in the vector); the *value* of the function is another thing altogether. When your function has two arguments, the value of the function often can be expressed as a third dimension, so you see contour maps of temperature or air pressure rendered as 3-D plots or, as I've done in class, 3-D printed examples.

2.2 Vector Field Examples

Examples of vector fields:

- wind speed arrows on a weather map
- more generally, flow in a liquid or gas (you see these often in animations of wings and jet engines and the like)
- electric and magnetic fields
- most generally, the *direction and magnitude of change* of any scalar function, such as the ones above.

2.3 Valleys and Saddles in Two Dimensions

With two variables, we can have a saddle in the value of the function, which looks like a Pringles or Chip Star potato chip. Consider the function $x^2 - y^2$. In R:

```
x <- y <- seq(-1, 1, length= 20)
saddle <- function(x,y){
x^2-y^2
}
z <- outer(x, y, saddle)
persp(x,y,z, theta=30, phi=30)
```

We'll also see shortly that how steep and narrow a valley is causes problems for the gradient descent algorithm:

```
valley <-function(x,y){
0.1*x^2+y^2
}
z <- outer(x, y, valley)
persp(x,y,z, theta=-60, phi=30)
```

(Try varying `theta` and `phi` for different views.)

3 Partial Derivative of a Function

Okay, time to get a little weird: many functions are functions of more than one variable. (In fact, almost everything we are doing in this class is a function of more than one variable.) So how do you take the derivative? Basically, you take the derivative one variable at a time, and treat the rest as constants. We call it a *partial derivative*, and write it with the symbol ∂ (literally, `\partial` in LaTeX).

The partial derivative in x of $x^2 - y^2$ (remembering that the derivative of a constant is zero) is

$$\frac{\partial}{\partial x}(x^2 - y^2) = 2x \tag{15}$$

while the partial derivative in y is

$$\frac{\partial}{\partial y}(x^2 - y^2) = -2y. \tag{16}$$

They are read "partial-partial-x" and "partial-partial-y". We can put these two together into a vector known as the *gradient*, written using the symbol ∇, called "nabla" or "del" (especially by physicists, in three dimensions) [1] and written in LaTeX using (you guessed it) `\nabla`:

$$f(x, y) = x^2 - y^2 \tag{17}$$

$$\operatorname{grad} f = \nabla f(x, y) = (\frac{\partial f}{\partial x}(x, y), \frac{\partial f}{\partial y}(x, y)) = (2x, -2y) \tag{18}$$

(Exercise: Where is this vector completely zero? Where is the x component zero? y component?)

4 Simple Gradient Descent

The basic idea is to go downhill, but we do it in discrete steps, based on the derivative or partial derivative. Recall that our function is a scalar field, but the partial derivative is a vector field.

4.1 Setting Up for Examples in R

An animation in R demonstrates gradient descent (see the link in the Sources at the end of these notes). You'll probably need to install the animation library:

```
library(animation)
Error in library(animation) : there is no package called animation
install.packages('animation')
--- Please select a CRAN mirror for use in this session ---
Fontconfig warning: ignoring UTF-8: not a valid region tag
trying URL 'https://cran.ism.ac.jp/bin/macosx/mavericks/contrib/3.3/animation_2.
Content type 'application/x-gzip' length 466777 bytes (455 KB)
==================================================
downloaded 455 KB


The downloaded binary packages are in
/var/folders/cw/3nx1g5cd2_vc6dn637d1gyl40000gn/T//Rtmpfz3kca/downloaded_packages
library(animation)
```

On a Mac, that pops up a window for you to pick a mirror. I picked Tokyo, of course.

[1] When I was a student at Caltech, a nearby fast food restaurant was named Del Taco, so of course it was *always* written ∇ Taco.

4.2 A Couple of Examples in R

Then this simple set of commands will animate a descent to find the minimum of the function $z = f(x, y) = x^2 + 2y^2$:

```
library(animation)
par(mar = c(4, 4, 2, 0.1))
grad.desc()
```

See how smoothly it converges on the global minimum! Like a ship gliding into a dock.

What happens at that saddle point we looked at above?

```
library(animation)
ani.options(nmax = 15)
x <- seq(-2,2, length=50)
y <- seq(-2,2, length=50)
saddle <- function(x,y) x^2-y^2
grad.desc(saddle, c(-2, -2, 3, 2), c(-1, 0.5), gamma = 0.3, tol = 1e-04)
grad.desc(saddle, c(-2, -2, 3, 2), c(-1, 0.5), gamma = 0.3, tol = 1e-04)
grad.desc(saddle, c(-2, -2, 3, 2), c(-1, 0), gamma = 0.3, tol = 1e-04)
grad.desc(saddle, c(-2, -2, 3, 2), c(-1, 0.01), gamma = 0.3, tol = 1e-04)
```

4.3 Rough Seas

But if the seas are a little rougher, say, the function

$$z = \sin(\frac{x^2}{2} - \frac{y^4}{4} + 3)\cos(2x + 1 - e^y), \tag{19}$$

now we've got a problem:

```
ani.options(nmax = 70)
f2 = function(x, y) sin(1/2 * x^2 - 1/4 * y^2 + 3) * cos(2 * x + 1 -
  exp(y))
x <- seq(-2,3, length=50)
y <- seq(-2,2, length=50)
z <- outer(x, y, f2)
persp(x,y,z, theta=-60, phi=30, expand=0.3)
grad.desc(f2, c(-2, -2, 3, 2), c(-1, 0.5), gamma = 0.3, tol = 1e-04)
```

our algorithm fails to converge! It heads toward a local minimum, then finds itself up high on a valley wall, overcorrects, and flies out of the valley. In the next valley, same thing happens. Then we see it bouncing side to side in a valley, gradually descending, but obviously wildly inefficient. Try it again with different values for gamma, e.g., 0.1, 0.2, 0.4, and see the behavior (0.4 is particularly entertaining).

4.4 So What *Is* Gradient Descent?

Okay, back up. We just saw what happens in gradient descent, but what's really going on here? Well, we are following the *gradient* of the function, looking for a local minimum. (Gradient descent is not really intended to find a global minimum.) The gradient tells us, from our current position, which direction is the steepest downhill. That gives us a vector, our direction to move in; pure gradient descent always moves exactly downhill from the current position, though variants of the algorithm may not.

In gradient descent, you start from some point x_0, then apply the iterative function

$$\vec{x}_{n+1} = \vec{x}_n - \gamma_n \nabla F(\vec{x}_n), n \geq 0. \tag{20}$$

Gamma (γ) is allowed to change at every step, which is why it has the subscript n in the equation, but the implementation we called in R above uses a constant gamma.

4.5 Simple One-Dimensional Example in Python

If you do Python (which is about the easiest possible language to read, when written simply), this example from Wikipedia's gradient descent entry in English (again, the Japanese page is poor; please fix it!) shows the basic idea of doing this in discrete steps in a single dimension. For the function $f(x) = x^4 - 3x^3 + 2$, the derivative is $f'(x) = 4x^3 - 9x^2$. The derivative is just written directly as the function `df(x)` in this code:

```
# From https://en.wikipedia.org/wiki/Gradient_descent
# From calculation, it is expected that the local minimum occurs at x=9/4

x_old = 0 # The value does not matter as long as abs(x_new - x_old) > precision
x_new = 6 # The algorithm starts at x=6
gamma = 0.01 # step size
precision = 0.00001

def df(x):
    y = 4 * x**3 - 9 * x**2
    return y

while abs(x_new - x_old) > precision:
    x_old = x_new
    x_new += -gamma * df(x_old)

print("The local minimum occurs at ", +x_new)
```

Try this with the different starting points 5, 6, and 7 for `x_new` and see what happens.

4.6 Returning to the Above...

Let's look in detail at the behavior of an example. For $z = f(x, y) = x^2 + 2y^2$, fairly obviously,

$$\nabla f(x, y) = (2x, 4y). \tag{21}$$

So if we refer to our current position as $\vec{r}$ and start at the point $\vec{r}_0 = (3, -3)$, and using $\gamma = 0.3$, we get

$$\vec{r}_1 = \vec{r}_0 - \gamma \nabla f(\vec{r}_0) = (-3, 3) - 0.3(-6, 12) = (-1.2, -0.6) \tag{22}$$
$$\vec{r}_2 = (-0.48, 0.12) \tag{23}$$
$$\vec{r}_3 = (-0.192000, -0.024) \tag{24}$$
$$\vec{r}_4 = (-0.0768, 0.0048) \tag{25}$$
$$\vec{r}_5 = (-3.0720 \times 10^{-02}, -9.6000 \times 10^{-04}) \tag{26}$$
$$\vec{r}_6 = (-1.2583\times^{-04}, -6.1440 \times 10^{-08}) \tag{27}$$

after which we're close enough to stop. Try it with

```
> grad.desc( gamma=0.3)
```

in R, or calculate those values in Octave using

```
octave:24> function retval = f(x)
> retval = x-0.3*[2,4].*x
> endfunction
octave:25> x = [-3, 3]
x =

  -3   3

octave:26> for i = 1:6
> x = f(x)
> endfor
retval =

  -1.20000  -0.60000

x =

  -1.20000  -0.60000

retval =

  -0.48000   0.12000

x =
  -0.48000   0.12000

retval =

  -0.192000  -0.024000

x =

  -0.192000  -0.024000

retval =

  -0.0768000   0.0048000

x =

  -0.0768000   0.0048000

retval =

  -3.0720e-02  -9.6000e-04

x =

  -3.0720e-02  -9.6000e-04

retval =

  -1.2583e-04  -6.1440e-08

x =

  -1.2583e-04  -6.1440e-08
```

Try bigger values for γ, and you'll see it fails to converge well. I'm not going to walk through the math for the more complicated example in Eq. 19, but even using the default small value of $\gamma = 0.05$, as you can see from the animation it flails around and never converges. There are constraints on the values γ can take and guarantee convergence, but we are not going to go into them here.

4.7 Variants

There are many variants on simple gradient descent, attempting to make it converge faster and more reliably. See Ruder's descriptions and animations (link below). The principle of each variant is pretty straightforward, but figuring out how to implement and tune it is complicated, and ultimately the size of your code will grow, probably substantially.

5 Newton's Method in Optimization

Newton's method is a famous iterative means for finding the zeros of a one-dimensional polynomial. Begin with some guess x_0 for the zero of the function, then iterate:

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}. \tag{28}$$

Obviously, this method fails if $f'(x_t) = 0$. If it works, though, it is basically taking a tangent to your curve at your current point, and extending that tangent as a line to find where it intersects the X axis.

5.1 Optimization In One Dimension

Here, though, we are looking at the use in optimization, looking for minima and maxima instead of zeros. Basically, we apply the above equation to $f'(x)$ instead of $f(x)$,

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}. \tag{29}$$

Obviously, this means that the function needs to be twice differentiable.

The second order Taylor expansion of a function,

$$f_T(x) = f_T(x_t + \Delta x) \approx f(x_t) + f'(x_t)\Delta x + \frac{1}{2}f''(x_t)\Delta x^2, \tag{30}$$

is a better approximation to the function around the point x than the first order approximation without the last term. We start with that equation, and look for a value of Δx that will give us $f'(x_t + \Delta x) = 0$, which will help us find a *stationary point* of the function. Our best bet is at $\Delta x = -f'(x_t)/f''(x_t)$. If $f()$ is a quadratic function, this one hop will take us to the exact stationary point, otherwise we will probably have to iterate.

5.2 More Than One Dimension

We can write the equivalent of Eq 30 for multiple dimensions. To avoid confusion with the separate dimensions, let's try to be careful and use the vector notation for the tth term, $\vec{x}_t$, n for the number of dimensions, and the subscripted scalar x_i to be the ith element of $\vec{x}$.

$$\begin{aligned} f(\vec{x}) &= f_T(\vec{x}_t + \Delta\vec{x}) \\ &\approx f(\vec{x}_t) + g^{\mathsf{T}}\Delta\vec{x} + \frac{1}{2}\Delta\vec{x}^{\mathsf{T}}H(\Delta\vec{x}), \end{aligned} \tag{31}$$

where

$$g = \nabla f(\vec{x}_t) \tag{32}$$

$$H = \nabla^2 f(\vec{x}_t) = \left(\frac{\partial^2}{\partial x_i \partial x_j} f(\vec{x}_t) \right)_{i,j}. \tag{33}$$

H is an array known as the *Hessian*. For our two-dimensional case with x and y the Hessian is a 2×2 matrix,

$$H = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}. \tag{34}$$

I'm not going to go through the algebra that leads us to showing that the step size ought to be $-H^{-1}g$ (basically, take the derivative again and solve for zero), but the algorithm is pretty simple:

1. Initialize $\vec{x}_0 \in \mathbb{R}^n$
2. Iterate $\vec{x}_{t+1} = \vec{x}_t - H^{-1}g$, where $g = \nabla f(\vec{x}_t), H = \nabla^2 f(\vec{x}_t)$
3. Stop when the difference between $\vec{x}_t$ and $\vec{x}_{t+1}$ is small enough to suit your tastes.

The problem is that calculating $-H^{-1}g$ is a pain, and there is no obvious guarantee that H is even invertible. *Pure speculation:* The reason this wasn't more painful for Newton to do by hand (besides the fact that he was a genius) was that he was primarily working in one to three dimensions, and usually only with quadratic functions, for topics like gravity and orbits.

5.3 Discussion

There are a whole host of ways in which this procedure can fail, and even when it succeeds each iteration is a lot of work, but it can produce much better steps.

Since gradient descent uses the first derivative of the function and Newton's method uses the second derivative of the function, as you might guess, they are called "first order methods" and "second order methods," respectively.

6 Conclusion

Understanding the basics covered in this lecture will prepare you to understand modern deep learning techniques.

7 Sources for this Lecture

Obviously, the basic idea of this goes back at least as far as Newton, and probably all the way back to the ancient Greeks or medieval Arabs. It is getting a lot of attention now because of the connection to deep learning, plus obviously our tremendous computational capabilities.

- Some text and examples adapted from the paperback edition of *A Gentle Introduction to Optimization*, B. Guerin *et al.*
- The English Wikipedia page on gradient descent is decent and leads to other useful pages.
- `http://sebastianruder.com/optimizing-gradient-descent/` has truly enlightening animations.
- An animation in R (including what I call the "rough seas" example `f2()`), `http://vis.supstat.com/2013/03/gradient-descent-algorithm-with-r/`.
- `https://lukaszkujawa.github.io/gradient-descent.html`.
- mathematicalmonk's YouTube video on Newton's method is a very clear explanation of the math, in two parts.
  `https://www.youtube.com/watch?v=28BMpgxn_Ec`
- The Stanford Coursera course on Machine Learning includes a sequence of short (10-15 minute) lectures on gradient descent,
  `https://www.coursera.org/learn/machine-learning/lecture/8SpIM/gradient-desce`
  explained in a slightly different way than I explain it.
- Also, there is an ad hoc list of test functions for global optimizers:
  - There is a convenient summary at Wikipedia, `https://en.wikipedia.org/wiki/Test_functions_for_optimization`.
  - This list comes from Rody Oldenhuis, originating in Matlab, available at MathWorks
    `http://jp.mathworks.com/matlabcentral/fileexchange/23147-test-functions-`
  - Another use of this by Gilberto A. Ortiz, also in Matlab on MathWorks,
    `http://jp.mathworks.com/matlabcentral/fileexchange/35801-evolution-strat`
- And another online library of extremely difficult (rigorous) test functions collected by Sonja Surjanovic and Derek Bingham at Simon Fraser University,
  `https://www.sfu.ca/~ssurjano/optimization.html`