Chapter 9

Optimization

9.1 Introduction: Going from Data to Functions

In this chapter, we take a turn into the heart of function building (which is, in turn, the primary topic of machine learning and model building as well).

In many cases, we'll have some kind of proposed form for a function. For example, in the line of best fit problem, the proposed form for the line was: $y = mx + b$, or $F(x) = mx + b$, where we needed to determine the model parameters m and b. Given data points (x_i, t_i) where t is for targets, this translated to our goal of finding m, b that satisfied some equation or system of equations:

$$\begin{aligned} F(x_1) &= t_1 \\ F(x_2) &= t_2 \\ &\vdots \\ F(x_p) &= t_p \end{aligned}$$

There was no exact solution to this system, which gives us error, which is typically a sum-of-squares:

$$E(m, b) = (t_1 - y_1)^2 + (t_2 - y_2)^2 + \cdots + (t_p - y_p)^2 = \sum_{k=1}^{p} (t_k - y_k)^2$$

where y_i is our model output, $F(x_i)$. And here we go- We need to minimize the error. In fact, most problems in machine learning (and model building) are cast as optimization problems.

How do we optimize a function? We learned the steps back in Calculus, and the algorithm depended on whether or not we had a bounded domain. We'll assume a domain that is not bounded for the remainder of this chapter, unless specified otherwise.

In the unbounded case, we know that the candidates for the maximum and minimum are found among the critical points of f (where $f'(x) = 0$ or where the derivative does not exist). Therefore, we need to have an algorithm that can seek out the roots (or the zeros) of a function.

9.2 Optimization and Calculus

As discussed, there is a close relationship between finding the roots to a function and optimizing a function. For the roots, we solve for where $g(x) = 0$, and in the optimization problem, we solve for where $f'(x) = 0$.

Therefore, discussions about optimization often turn out to be discussions about finding roots. In that spirit, we look at a very basic algorithm for finding roots- An algorithm called "The Bisection Method".

9.2.1 The Bisection Method

The motivation for this method comes from the Intermediate Value Theorem from Calculus- In this case, suppose that our function g is continuous on $[a, b]$, and further $g(a)$ and $g(b)$ have different signs (more compactly $g(a)g(b) < 0$), then there is a c in the interval such that $g(c) = 0$.

Now, given such an interval $[a, b]$, we can get a better approximation by breaking the interval in half. Here is the algorithm:

The Bisection Algorithm to Solve $f(x) = 0$

- Initialize with function $f(x)$, values of a, b and tol, so that the root of f is inside the interval $[a, b]$ (we should have $f(a)f(b) < 0$).
- Set $f_a = f(a), f_b = f(b)$.
- Main loop:
  - Set $c = (a + b)/2$, and compute $f_c = f(c)$.
  - If $f_a f_c < 0$, then the new interval is $[a, c]$. Set $b = c$, $f_b = f_c$.
    Otherwise, the new interval is $[c, b]$. Set $a = c$, $f_a = f_c$
  - Stopping criteria (one or more):
    - $(b - a)/2 < tol$, or
    - $|f_c| < tol$ (or $f(c)$ is actually 0)
  - At the end, output the midpoint of the current interval as the solution: $(a + b)/2$.

The nice thing about the bisection method is that it is easy to implement. It takes a lot of iterations to converge, however.

There are examples of this algorithm available to you in Matlab and Python that we wrote ourselves directly from our pseudo-code. At the end of this section, we'll discuss the built-in code.

When using the Bisection Method in either Matlab or Python, we need to be able to pass in the name of the function to our algorithm. Here is an example, where we solve $x^3 + x - 1 = 0$ with a starting interval of $[0, 1]$ and a tolerance of "5e-5" (which is short for scientific notation, 5×10^{-5}).

In Matlab, we'll assume that `bisect.m` is in the folder or search path, and in Python, we'll assume that the **def**inition of the function **bisection** has been run (again, as a reminder, these are functions that we defined and are available on our class website).

Below we show how to create what is referred to as an **anonymous function**, and we'll use that to pass in the function $g(x)$ into our algorithm. The nice thing about these kinds of functions is that they do not need a separate file to define them (like Matlab), nor do the need a whole **def** section in Python. Of course, if your function is more complicated, you may need to go that route, and in the next section we'll have an example of creating that kind of function.

Matlab Code Example	Python Code Example
`>>f= @(x) x.^3+x-1;` `>>x=bisect(f,0,1,5e-5);`	`>>> f = lambda x: x**3 + x - 1` `>>> x=bisection(f,0,1,5e-5)`

9.2.2 Newton's Method

Another method we have from Calculus is Newton's Method. Newton's Method is a lot faster, but it also requires the computation of the derivative. The formula is given below. If you're not sure where the formula comes from, we'll review it in class (you can also look it up online or in a calculus text).

Newton's Method

Newton's method is used to locate a root to a function g. It uses an initial guess and produces a sequence of values that (hopefully) converge to a root.

Given $g(x)$, and an initial guess of the root, x_0, we iterate the following:

$$x_{i+1} = x_i - \frac{g(x_i)}{g'(x_i)}$$

Coding this is straightforward; the biggest consideration is how we'll pass in the function. It might be easiest to have the function compute both the value of the function *and* the value of the derivative at a given point. That is currently the way the code is written.

We might mention that the code might be more streamlined if you have one (anonymous) function to handle the computation of g, and a second to compute the computation of the derivative, g'. Feel free to make those changes.

To use either the bisection or Newton's method Matlab functions, we'll need to define the function (that we're finding the roots to). You can either use an anonymous function handle (defined in-line) or use an $m-$file- It is useful to go ahead and program in both f and the derivative so you can use either bisection or Newton's method. Here's an example finding a root to $e^x = 3x$.

First, we'll write a function file. To the left is Matlab, to the right is Python:

```
function [y,dy]=testfunc01(x)
% Test function e^x-3x
y=exp(x)-3*x;  % Output y
dy=exp(x)-3;   % and dy
```

```
import math

def testfunc01(x):
y=math.exp(x)-3*x
dy=math.exp(x)-3
return y,dy
```

We'll put together examples in Matlab and Python. To keep things tidy, we'll have the function definitions in the same file as the script. (This is available as `ExScript1.m` or `ExScript1.py` on our class website).

Built-in Methods

I don't believe that Matlab has the Bisection Method or Newton's Method specifically built-in on their own. Rather, it uses `fzero` as a general solver for the roots of a function, but it uses methods that don't rely on the derivative (bisection is one option), so it is slower. It also provides `fminbnd, fminunc, fmincon` and others in the Optimization toolbox. For now, we'll just stick to our two algorithms.

In contract, in Python we have `scipy.optimize.newton()`, where the inputs vary depending on the information available.

9.3 Homework

1. It's always good to be able to do a simple example of these algorithms "by hand". Try these out (you may use a hand calculator):

   (a) Estimate the first root of $e^x - 5x$ by computing three iterations of the bisection method on the interval $[0, 1]$.

   (b) Estimate the second root of $e^x - 5x$ by computing Newton's Method twice, with a starting guess of $x_0 = 2$.

2. Use the example script file in Matlab or Python to solve the next two problems. You can put them both on the same script, both functions can be anonymous.

   (a) Use the bisection method to find the root correct to 6 decimal places: $3x^3 + x^2 = x + 5$.

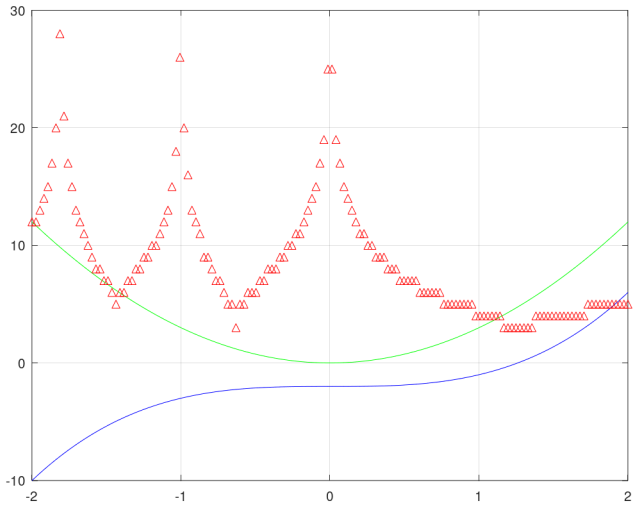
(b) Use Newton's Method to find the same root.

3. In this exercise, we want to look at how fast Newton's Method will converge as a function of the initial guess. We'll use $f(x) = x^3 - 2$ (so we'll be estimating $\sqrt[3]{2}$). We will choose the initial guess x_0 from the interval $[-2, 2]$, and run Newton's Method- Keeping track of how many iterations it takes to converge.

   We then repeat this for another x_0, and another x_0- In fact, we take 150 values between $[-2, 2]$ for the initial condition, and plot the number of iterations for convergence (in red triangles below).

   We'll then plot the curve $f(x) = x^3 - 2$, and the derivative $y = 3x^2$ in the same graph.

   See if you can examine the curve and explain the shape you see.



(Extra for students with coding experience: Reproduce the graph for yourself!)

9.4 Linearization

Before continuing with these algorithms, it will be helpful to review the Taylor series for a function of one variable, and see how it extends to functions of more than one variable.

Recall that the Taylor series for a function $f(x)$, based at a point $x = a$, is given by the following, where we assume that f is analytic:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Therefore, we can *approximate* f using a constant:

$$f(x) \approx f(a)$$

or using a linear approximation (which is the tangent line to f at a):

$$f(x) \approx f(a) + f'(a)(x-a)$$

or using a quadratic approximation:

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2$$

We can do something similar if f depends on more than one variable. For example, in Calculus III we look at functions like

$$z = f(x, y)$$

In this case, the linearization of f at $x = a$, $y = b$ is given by the tangent plane:

$$z = f(a, b) + f_x(a, b)(x - a) + f_y(a, b)(y - b)$$

Similarly, if $y = f(x_1, x_2, \ldots, x_n)$, then the tangent plane at $x_1 = a_1, \ldots, x_n = a_n$ is given by:

$$\begin{aligned} z =& f(a_1, a_2, \ldots, a_n) + f_{x_1}(a_1, \ldots, a_n)(x_1 - a_1) + f_{x_2}(a_1, \ldots, a_n)(x_2 - a_2) + \ldots \\ & f_{x_n}(a_1, \ldots, a_n)(x_n - a_n) \end{aligned}$$

If we want to go further with a second order (quadratic) approximation, it looks very similar. First, if $z = f(x, y)$ at (a, b), the quadratic approximation looks like this:

$$\begin{aligned} z = f(a, b) + f_x(a, b)(x - a) + & f_y(a, b)(y - b) + \frac{1}{2}[f_{xx}(a, b)(x - a)^2 + \\ & 2f_{xy}(a, b)(x - a)(y - b) + f_{yy}(a, b)(y - b)^2] \end{aligned}$$

where we assume that $f_{xy}(a, b) = f_{yx}(a, b)$. For functions of n variables, the notation gets a little awkward. We'll define a new structure that will make the notation work a little better.

Gradient and Hessian

Let $y = f(x_1, \ldots, x_n)$ be a real valued function of n variables. To make the notation a bit easier to read, we'll denote the partial derivatives using subscript notation, where

$$\frac{\partial f}{\partial x_i} \doteq f_i \qquad \frac{\partial^2 f}{\partial x_i \partial x_j} \doteq f_{ji}$$

(Does the order of the differentiation matter? In the last equation, could I have written f_{ij}?)

We'll recall from Calculus III that the gradient of f is usually defined as a row vector of first partial derivatives:

$$\nabla f = [f_1, f_2, \ldots, f_n]$$

The $n \times n$ matrix of second partial derivatives is called the **Hessian matrix**, where the (i, j) element is f_{ij}, or

$$Hf = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & & & \vdots \\ f_{n1} & f_{n2} & \cdots & f_{nn} \end{bmatrix}$$

Using this notation, the **linear approximation** to $f(x_1, \ldots, x_n) = f(\boldsymbol{x})$ at the point $\boldsymbol{x} = \boldsymbol{a}$ is:

$$f(\boldsymbol{a}) + \nabla f(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})$$

(That last term is a dot product, or a row vector times a column vector) The **quadratic approximation** to f is:

$$f(\boldsymbol{a}) + \nabla f(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{a})^T Hf(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})$$

As another example, suppose we want the quadratic approximation to the function

$$w = x \sin(y) + y^2 z + xyz + z$$

Find the quadratic approximation to w at the point $(1, 0, 2)$.

SOLUTION: We need to evaluate f, all the first partials, and all the second partials at the given point:

$$f(1, 0, 2) = 1 \cdot 0 + 0 \cdot 2 + 0 + 2 = 2$$

Now, using our new notation for partial derivatives:

$$f_1 = \sin(y) + yz \quad \Rightarrow \quad f_1(1, 0, 2) = 0$$
$$f_2 = x\cos(y) + 2yz + xz \quad \Rightarrow \quad f_2(1, 0, 2) = 3$$
$$f_3 = y^2 + xy + 1 \quad \Rightarrow \quad f_3(1, 0, 2) = 1$$

Therefore, $\nabla f(1, 0, 2) = [0, 3, 1]$. Now computing the Hessian, we get:

$$\left[\begin{array}{ccc} 0 & \cos(y) + z & y \\ \cos(y) + z & -x\sin(y) + 2z & 2y + x \\ y & 2y + x & 0 \end{array} \right] \Bigg|_{x=1, y=0, z=2} = \left[\begin{array}{ccc} 0 & 3 & 0 \\ 3 & 4 & 1 \\ 0 & 1 & 0 \end{array} \right]$$

Now we can put these into the formula for the quadratic approximation:

$$2 + [0, 3, 1] \left[\begin{array}{c} x - 1 \\ y \\ z - 2 \end{array} \right] + \frac{1}{2}[x - 1, y, z - 2] \left[\begin{array}{ccc} 0 & 3 & 0 \\ 3 & 4 & 1 \\ 0 & 1 & 0 \end{array} \right] \left[\begin{array}{c} x - 1 \\ y \\ z - 2 \end{array} \right]$$

And expanded, we could write this as:

$$2 + 3y + (z - 2) + \frac{1}{2}\left[6(x - 1)y + 4y^2 + 2y(z - 2)\right] = \\ -2y + z + 3xy + 2y^2 + yz$$

This is the quadratic approximation to $x\sin(y) + y^2 z + xyz + z$ at the point $(1, 0, 2)$.

Linearization, Continued

Suppose we have a general function $\boldsymbol{G}$ that inputs n variables and outputs m variables. In that case, we might write $\boldsymbol{G}$ as:

$$\boldsymbol{G}(\boldsymbol{x}) = \left[\begin{array}{c} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_m(x_1, x_2, \ldots, x_n) \end{array} \right]$$

For example, here's a function that inputs two variables and outputs 3 nonlinear functions:

$$\boldsymbol{G}(x, y) = \left[\begin{array}{c} x^2 + xy + y^2 - 3x + 5 \\ \sin(x) + \cos(y) \\ 1 + 3x - 5y + 2xy \end{array} \right] \tag{9.1}$$

We need a new way of getting the derivative- In this case, it is called **the Jacobian matrix**. If $\boldsymbol{G}$ maps $\mathbb{R}^n$ to $\mathbb{R}^m$, then the Jacobian matrix is $m \times n$, where each row is the gradient of the corresponding function:

$$\boldsymbol{G}(\boldsymbol{x}) = \left[\begin{array}{c} g_1(x_1, x_2, \ldots, x_n) \\ g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ g_m(x_1, x_2, \ldots, x_n) \end{array} \right] \quad \Rightarrow \quad JG = \left[\begin{array}{c} \nabla g_1(x_1, x_2, \ldots, x_n) \\ \nabla g_2(x_1, x_2, \ldots, x_n) \\ \vdots \\ \nabla g_m(x_1, x_2, \ldots, x_n) \end{array} \right]$$

Or, written out, the (i, j) element of the Jacobian matrix for $\boldsymbol{G}$ is:

$$(JG)_{ij} = \frac{\partial g_i}{\partial x_j}$$

Continuing with our previous example (Equation 9.1), we'll construct the Jacobian matrix:

$$\boldsymbol{G}(x, y) = \begin{bmatrix} x^2 + xy + y^2 - 3x + 5 \\ \sin(x) + \cos(y) \\ 1 + 3x - 5y + 2xy \end{bmatrix} \quad \Rightarrow \quad JG = \begin{bmatrix} 2x + y - 3 & x + 2y \\ \cos(x) & -\sin(y) \\ 3 + 2y & -5 + 2x \end{bmatrix}$$

As a second example, suppose that we have a function f that maps $\mathbb{R}^n$ to $\mathbb{R}$. Then we could let

$$\boldsymbol{G}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$$

Note that we have to think of the gradient as a column vector instead of a row, and the i^{th} row is:

$$f_{x_i}(x_1, x_2, \ldots, x_n)$$

so that $\boldsymbol{G}$ maps $\mathbb{R}^n$ to $\mathbb{R}^n$. Furthermore, in that case, the **Jacobian of $\boldsymbol{G}$** is the **Hessian of** f:

$$(JG)_{ij} = \frac{\partial f_{x_i}}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_i}\right) = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

(We're using Clairaut's Theorem to simplify our expression).

Now we get to the **linearization** of the function $\boldsymbol{G}$ at a point $\boldsymbol{x} = \boldsymbol{a}$:

$$\boldsymbol{G}(\boldsymbol{a}) + JG(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a})$$

To illustrate this notation, let's linearize Equation 9.1 at the origin $(0, 0)$.

$$\boldsymbol{G}(0, 0) = \begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix} \qquad JG(0, 0) = \begin{bmatrix} -3 & 0 \\ 1 & 0 \\ 3 & -5 \end{bmatrix}$$

Therefore, the linearization at the origin is:

$$\begin{bmatrix} 5 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -3 & 0 \\ 1 & 0 \\ 3 & -5 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 - 3x \\ 1 + x \\ 1 + 3x - 5y \end{bmatrix}$$

You might notice that each row is the linearization of the corresponding function from $\boldsymbol{G}$- For example, $1 + x$ is the linearization of $\sin(x) + \cos(y)$ at the point $(0, 0)$.

Now we can get back to Newton's Method.

Multivariate Newton's Method

In the general case, we are solving for the critical points of some function f,

$$\nabla f(\boldsymbol{x}) = \boldsymbol{0}$$

That is, if f depends on n variables, then this is a system of n equations (in n unknowns).

As we did in the previous section, we can think of the gradient itself as a function:

$$\boldsymbol{G}(\boldsymbol{x}) = \nabla f(\boldsymbol{x})$$

and think about how Newton's Method will apply in this case.

Recall that in Newton's Method in one dimension, we begin with a guess x_0. We then solve for where the *linearization* of f crosses the $x-$axis:

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

From which we get the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Now we do the same thing with the vector-valued function $\boldsymbol{G}$, where $\boldsymbol{G} : \mathbb{R}^n \to \mathbb{R}^n$. Start with an initial guess, $\boldsymbol{x}_0$, then we linearize $\boldsymbol{G}$ and solve the following for $\boldsymbol{x}$:

$$\boldsymbol{G}(\boldsymbol{x}_0) + JG(\boldsymbol{x}_0)(\boldsymbol{x} - \boldsymbol{x}_0) = 0$$

Remember that JG, the Jacobian of $\boldsymbol{G}$, is now an $n \times n$ matrix, so this is a system of n equations in n unknowns. Using the recursive notation, and solving for $\boldsymbol{x}$, we now get the formula for **multivariate Newton's Method**:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - JG^{-1}(\boldsymbol{x}_i)G(\boldsymbol{x}_i)$$

We should notice the beautiful way that method generalizes to multiple dimensions- The reciprocal $1/f'(x_i)$ becomes the matrix inverse: $JG^{-1}(\boldsymbol{x}_0)$.

9.5 Nonlinear Optimization with Newton

Now we go back to our original question: We want to optimize a function whose domain is multidimensional:

$$\min_{\mathbf{x}} f(\mathbf{x}) \qquad \text{or} \qquad \max_{\mathbf{x}} f(\mathbf{x})$$

where it is assumed that $f : \mathbb{R}^n \to \mathbb{R}$. To use Newton's Method, we let $\boldsymbol{G} = \nabla f$ for the formulas in the previous section. Here's a summary:

Multivariate Newton's Method to Solve $\nabla f(\mathbf{x}) = 0$

Given an initial $\mathbf{x}_0$, compute a sequence of better approximations to the solution:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - Hf^{-1}(\boldsymbol{x}_i)\nabla f(\boldsymbol{x}_i)$$

where Hf is the Hessian matrix ($n \times n$ matrix of the second partial derivatives of f).

Example: Minimize $f(x, y) = \frac{1}{4}x^4 - \frac{1}{2}x^2 + \frac{1}{2}y^2$.

SOLUTION: First, we can use Calculus to check the computer output. The gradient is:

$$\nabla f = [x^3 - x, \quad y]$$

so that the critical points are $(-1, 0), (1, 0)$ and $(0, 0)$.

9.5.1 Issue: Local Extrema

As in calculus, once we find the critical points, we need to determine if they represent a local max, a local min, or something else (like a saddle).

We may recall the "second derivatives test" from Calculus III- It actually uses the Hessian:

The Second Derivatives Test for $z = f(x, y)$ at critical point (a, b)

Let $D = f_{xx}(a, b)f_{yy}(a, b) - f_{xy}^2(a, b)$. If

- If $D > 0$ and $f_{xx}(a, b) > 0$, then $f(a, b)$ is a local min.
- If $D > 0$ and $f_{xx}(a, b) < 0$, then $f(a, b)$ is a local max.
- If $D < 0$, then $f(a, b)$ is a saddle point.

Returning to our example, the Hessian of f:

$$Hf = \begin{bmatrix} f_{xx}(x, y) & f_{xy}(x, y) \\ f_{yx}(x, y) & f_{yy}(x, y) \end{bmatrix} = \begin{bmatrix} 3x^2 - 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Therefore, the second derivatives test is a test using the determinant of the Hessian at the critical point. In this example, at $(\pm 1, 0)$, the determinant is positive and $f_{xx}(\pm 1, 0)$ is positive. Therefore, $f(\pm 1, 0) = 1/4$ is the local minimum. We have a saddle point at the origin.

9.5.2 Issue: Matrix Inversion

Since the multivariate Newton's method requires us to invert an $n \times n$ matrix, we need to be careful. In fact, many algorithms will watch the eigenvalues of the Hessian to make sure that the matrix doesn't get too close to being non-invertible. One way to do this is to track what is called **the condition number** of the matrix. The condition number of the matrix is defined as:

$$k = \frac{|\lambda_{\max}|}{|\lambda_{\min}|}.$$

If the matrix is not invertible, or very close to being not invertible, then $\lambda_{\min} \approx 0$, which makes the condition number blow up to a very large number.

One way we may avoid this problem is through the use of the pseudo-inverse, but one need to be careful about when to employ this technique.

Multivariate Newton's Method, in Matlab and Python

These will be provided in a separate handout.

9.6 Gradient Descent

The method of gradient descent is an algorithm that will search for local extrema by taking advantage of the fact about gradients:

> The gradient of a function at a given point will point in the direction of fastest increase (of the function).

Therefore, if we started at a point (or vector) $\boldsymbol{a}$, if we want to find a local maximum, we should move in the direction of the gradient. To locate a local minimum, we should move in the direction negative to the gradient.

Therefore, the method of gradient descent proceeds as follows: Given a starting point $\boldsymbol{a}$ and a scalar α (which is referred to as the step size), we iterate the following:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \alpha \nabla f(\boldsymbol{x}_i)$$

The method is fairly quick, but can have bad convergence properties- Mainly because we have that step size. There is a way to get "best" step size α. In the following, we'll continue with finding the local **minimum**.

Once we choose a direction of travel, the idea will be to follow that line until we find the minimum of the function (along the line). That is, we have a small optimization problem. Find the value of t that minimizes the expression:

$$f(\boldsymbol{a} - t\nabla f(\boldsymbol{a}))$$

This is called a *line search*, and can be time consuming. Before continuing, let's look at this more closely. Define a new function $\phi(t)$ as the output along the line:

$$\phi(t) = f(\boldsymbol{a} - t\boldsymbol{u}) = f\begin{pmatrix} a_1 - tu_1 \\ a_2 - tu_2 \\ \vdots \\ a_n - tu_n \end{pmatrix}$$

where $\boldsymbol{u}$ is a vector and f depends on $x_1, \ldots, x_n$.

Then optimizing this means to find where the derivative of ϕ is zero. Let's compute the derivative using the chain rule:

$$\phi'(t) = \frac{\partial f}{\partial x_1}\frac{\partial x_1}{\partial t} + \cdots + \frac{\partial f}{\partial x_n}\frac{\partial x_n}{\partial t} = \frac{\partial f}{\partial x_1}u_1 + \cdots + \frac{\partial f}{\partial x_n}u_n = -\nabla f(\boldsymbol{a} - t\boldsymbol{u}) \cdot \boldsymbol{u}$$

(Note the minus sign from differentiating $-t\boldsymbol{u}$). Thus, setting this to zero and solving should give us an optimal step size.

Example: By hand, compute one step of gradient descent to the following (with optimal step size).

$$f(x, y) = 4x^2 - 4xy + 2y^2 \qquad (x_0, y_0) = (2, 3).$$

SOLUTION:

Step 1: Compute the gradient and evaluate it at $(2, 3)$:

$$\nabla f = [8x - 4y, -4x + 4y] \quad \Rightarrow \quad \nabla f(2, 3) = [4, 4]$$

Step 2A: Find the parametric equations of the line along which we will travel (h is the stepsize to be found later):

$$\boldsymbol{x}_0 - h\nabla f(\boldsymbol{x}_0) = \begin{bmatrix} 2 \\ 3 \end{bmatrix} - h\begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 - 4h \\ 3 - 4h \end{bmatrix}$$

Step 2B: Apply f to the path found in part $2A$. This is the expression we want to minimize.

$$\phi(h) = f(\boldsymbol{x}_0 - h\nabla f(\boldsymbol{x}_0) = f(2 - 4h, 3 - 4h)$$

We could write out this expression, but it is not needed at the moment.

Step 2C: To find the optimal path, we define $\phi(h)$ to be the function restricted to the line:

$$\phi(h) = f(2 - 4h, 3 - 4h)$$

We'll need to evalate the gradient along the line, so let's go ahead and compute that:

$$f_x(2 - 4h, 3 - 4h) = 8(2 - 4h) - 4(3 - 4h) = 16 - 32h + 12 + 16h = 4 - 16h$$

$$f_y(2 - 4h, 3 - 4h) = -4(2 - 4h) + 4(3 - 4h) = -8 + 16h + 12 - 16h = 4$$