In this topic we will discuss optimizing over general functions \( f \). Typically the function is defined \( f : \mathbb{R}^d \rightarrow \mathbb{R} \); that is its domain is multi-dimensional (in this case \( d \)-dimensional point \( \alpha \)) and output is a real scalar (\( \mathbb{R} \)). This often arises to describe the “cost” of a model which has \( d \) parameters which describe the model (e.g., degree \((d-1)\)-polynomial regression) and the goal is to find the parameters \( \alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d) \) with minimum cost. Although there are special cases where we can solve for these optimal parameters exactly, there are many cases where we cannot. What remains in these cases is to analyze the function \( f \), and try to find its minimum point. The most common solution for this is gradient descent where we try to “walk” in a direction so the function decreases until we no longer can.

### 6.1 Functions

We review some basic properties of a function \( f : \mathbb{R}^d \rightarrow \mathbb{R} \). Again, the goal will be to unveil abstract tools that are often easy to imagine in low dimensions, but automatically generalize to high-dimensional data. We will first provide definitions without any calculus.

Let \( B_r(\alpha) \) define a Euclidean ball around a point \( \alpha \in \mathbb{R}^d \) of radius \( r \). That is, it includes all points \( \{ p \in \mathbb{R}^d \mid \|\alpha - p\| \leq r \} \), within a Euclidean distance of \( r \) from \( \alpha \). We will use \( B_r(\alpha) \) to define a local neighborhood around a point \( \alpha \). The idea of “local” is quite flexible, and we can use any value of \( r > 0 \), basically it can be as small as we need it to be, as long as it is strictly greater than 0.

**Minima and maxima.** A local maximum of \( f \) is a point \( \alpha \in \mathbb{R}^d \) so for some neighborhood \( B_r(\alpha) \), all points \( p \in B_r(\alpha) \) have smaller (or equal) function value than at \( \alpha \): \( f(p) \leq f(\alpha) \). A local minimum of \( f \) is a point \( \alpha \in \mathbb{R}^d \) so for some neighborhood \( B_r(\alpha) \), all points \( p \in B_r(\alpha) \) have larger (or equal) function value than at \( \alpha \): \( f(p) \geq f(\alpha) \). If we remove the “or equal” condition for both definitions for \( p \in B_r(\alpha), p \neq \alpha \), we say the maximum or minimum points are strict.

A point \( \alpha \in \mathbb{R}^d \) is a global maximum of \( f \) if for all \( p \in \mathbb{R}^d \), then \( f(p) \leq f(\alpha) \). Likewise, a point \( \alpha \in \mathbb{R}^d \) is a global minimum if for all \( p \in \mathbb{R}^d \), then \( f(p) \geq f(\alpha) \). There may be multiple global minimum and maximum. If there is exactly one point \( \alpha \in \mathbb{R}^d \) that is a global minimum or global maximum, we again say it is strict.

When we just use the term minimum or maximum (without local or global) it implies a local minimum or maximum.

Focusing on a function restricted to a closed and bounded subset \( S \subset \mathbb{R}^d \), if the function is continuous (that is, there exists a \( \delta \) such that for all \( \alpha \in \mathbb{R}^d \), there exists a radius \( r_\delta \) such that if \( p \in B_{r_\delta}(\alpha) \), then \( |f(\alpha) - f(p)| \leq \delta \)), then the function must have a global minimum and a global maximum. It may occur on the boundary of \( S \).

A saddle point \( \alpha \in \mathbb{R}^d \) (for \( d > 1 \)) has within any neighborhood \( B_r(\alpha) \) a points \( p \in B_r(\alpha) \) with \( f(p) < f(\alpha) \) (the lower points) and \( p' \in B_r(\alpha) \) with \( f(p') > f(\alpha) \) (the upper points). In particular, it is a saddle if within \( B_r(\alpha) \) there are disconnected regions of upper points (and of lower points). The notion of saddle point is defined differently for \( d = 1 \). If these regions are connected, and it is not a minimum or maximum, then it is a regular point.

For an arbitrary (or randomly) chosen point \( \alpha \), it is usually a regular point (except for examples you are unlikely to encounter, the set of minimum, maximum, and saddle points are finite, while the set of regular points is infinite).
Example: Continuous Functions

Here we show some example functions, where the \( \alpha \)-value on the \( x \)-axis represents a \( d \)-dimensional space. The first function has local minimum and maximum. The second function has a constant value, so every point is a global minimum and a global maximum. The third function \( f \) is convex, which is demonstrated with the line segment between points \((p, f(p))\) and \((q, f(q))\) is always above the function \( f \).

Convex functions. In many cases we will assume (or at least desire) that our function is convex.

To define this it will be useful to define a line \( \ell \subset \mathbb{R}^d \) as follows with any two points \( p, q \in \mathbb{R}^d \). Then a line \( \ell_{p,q} \) is the set of all points defined by any scalar \( \lambda \in \mathbb{R} \) as

\[
\ell_{p,q} = \{ x = \lambda p + (1 - \lambda)q \mid \lambda \in \mathbb{R} \}.
\]

When \( \lambda \in [0, 1] \), then this defines the line segment between \( p \) and \( q \).

A function is convex if for any two points \( p, q \in \mathbb{R}^d \), on the line segment between them has value less than (or equal) to the values at the weighted average of \( p \) and \( q \). That is, it is convex if

\[
\text{For all } p, q \in \mathbb{R} \text{ and for all } \lambda \in [0, 1] \quad f(\lambda p + (1 - \lambda)q) \leq \lambda f(p) + (1 - \lambda)f(q).
\]

Removing the “or equal” condition, the function becomes strictly convex.

There are many very cool properties of convex functions. For instance, for two convex functions \( f \) and \( g \), then \( h(\alpha) = f(\alpha) + g(\alpha) \) is convex and so is \( h(\alpha) = \max\{f(\alpha), g(\alpha)\} \). But one will be most important for us:

**Convexity and Global Minimums:** Any local minimum of a convex function will also be a global minimum. A strictly convex function will have at most a single minimum: the global minimum.

This means if we find a minimum, then we must have also found a global minimum (our goal).

6.2 Gradients

For a function \( f(\alpha) = f(\alpha_1, \alpha_2, \ldots, \alpha_d) \), and a unit vector \( u = (u_1, u_2, \ldots, u_d) \) which represents a direction, then the directional derivative is defined

\[
\nabla_u f(\alpha) = \lim_{h \to 0} \frac{f(\alpha + hu) - f(\alpha)}{h}.
\]

We are interested in functions \( f \) which are differentiable; this implies that \( \nabla_u f(\alpha) \) is well-defined for all \( \alpha \) and \( u \). The converse is not necessarily true.

Let \( e_1, e_2, \ldots, e_d \in \mathbb{R}^d \) be a specific set of unit vectors so that \( e_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0) \) where for \( e_i \) the 1 is in the \( i \)-th coordinate.
Then define
\[ \nabla_i f(\alpha) = \nabla_{e_i} f(\alpha) = \frac{d}{d\alpha_i} f(\alpha). \]

It is the derivative in the \( i \)th coordinate, treating all other coordinates as constants.

We can now, for a differentiable function \( f \), define the **gradient of \( f \)** as
\[ \nabla f = \frac{df}{d\alpha_1} e_1 + \frac{df}{d\alpha_2} e_2 + \ldots + \frac{df}{d\alpha_d} e_d = \left( \frac{df}{d\alpha_1}, \frac{df}{d\alpha_2}, \ldots, \frac{df}{d\alpha_d} \right). \]

Note that \( \nabla f \) is a function from \( \mathbb{R}^d \to \mathbb{R}^d \), which we can evaluate at any point \( \alpha \in \mathbb{R}^d \).

**Example: Gradient**

For \( \alpha = (x, y, z) \in \mathbb{R}^3 \), consider the function \( f(x, y, z) = 3x^2 - 2y^3 - 2xe^z \). Then \( \nabla f = (6x - 2e^z, -6y^2, -2xe^z) \) and \( \nabla f(3, -2, 1) = (18 - 2e, 24, -6e) \).

**Linear approximation.** From the gradient we can easily recover the directional derivative of \( f \) at point \( \alpha \), for any direction (unit vector) \( u \) as
\[ \nabla_u f(\alpha) = \langle \nabla f(\alpha), u \rangle. \]

This implies the gradient describes the linear approximation of \( f \) at a point \( \alpha \). The slope of the tangent plane of \( f \) at \( \alpha \) in any direction \( u \) is provided by \( \nabla_u f(\alpha) \).

Hence, the direction which \( f \) is increasing the most at a point \( \alpha \) is the unit vector \( u \) where \( \nabla_u f(\alpha) = \langle \nabla f(\alpha), u \rangle \) is the largest. This occurs at \( \nabla f(\alpha) = \nabla f(\alpha)/\|\nabla f(\alpha)\| \), the normalized gradient vector.

To find the minimum of a function \( f \), we then typically want to move from any point \( \alpha \) in the direction \( -\nabla f(\alpha) \); at regular points this is the direction of steepest descent.

### 6.3 Gradient Descent

Gradient descent is a family of techniques that, for a differentiable function \( f : \mathbb{R}^d \to \mathbb{R} \), try to identify either
\[ \min_{\alpha \in \mathbb{R}^d} f(\alpha) \] and/or \[ \alpha^* = \arg \min_{\alpha \in \mathbb{R}^d} f(\alpha). \]

This is effective when \( f \) is convex and we do not have a “closed form” solution \( \alpha^* \). The algorithm is iterative, in that it may never reach the completely optimal \( \alpha^* \), but it keeps getting closer and closer.

**Algorithm 6.3.1 Gradient Descent**

\[
\begin{align*}
\text{initialize } & \alpha^{(0)} = \alpha_{\text{start}} \in \mathbb{R}^d. \\
\text{repeat } & \alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f(\alpha^{(k)}) \\
\text{until } & (\|\nabla f(\alpha^{(k)})\| \leq \tau) \\
\text{return } & \alpha^{(k)} 
\end{align*}
\]

Basically, for any starting point \( \alpha^{(0)} \) the algorithm moves to another point in the direction opposite to the gradient, in the direction that locally decreases \( f \) the fastest. How fast it moves depends on the scalar learning rate \( \gamma_k \) and the magnitude of the gradient vector \( \nabla f(\alpha^{(k)}) \).
Stopping condition. The parameter $\tau$ is the tolerance of the algorithm. If we assume the function is differentiable, then at the minimum $\alpha^*$, we must have that $\nabla f(\alpha) = (0, 0, \ldots, 0)$. So for $\alpha$ close to the minimum, $\nabla f(\alpha)$ should also have a small norm. The algorithm may never reach the true minimum (and we do not know what it is, so we cannot directly compare against the function value). So we use $\|\nabla f\|$ as a proxy.

In other settings, we may run for a fixed number $T$ steps. Although this does not automatically tune the algorithm to the input, as using a tolerance $\tau$ may, it is easier to describe and compare. Hence, most examples in this text will use this method.

6.3.1 Learning Rate

The most critical parameter of gradient descent is $\gamma$, the learning rate. In many cases the algorithm will keep $\gamma_k = \gamma$ fixed for all $k$. It controls how fast the algorithm works. But if it is too large, when we approach the minimum, then the algorithm may go too far, and overshoot it.

How should we choose $\gamma$? There is no consensus to this answer, and often in practice it is tuned in ad-hoc ways. In the following, we will describe some mathematically described scenarios where something formal can be said about how to choose $\gamma$. In some cases, these analysis show that if the function satisfies a mild property, then many fixed choices of $\gamma$ will result in accuracy guarantees. We will also show methods where it helps to adjust the learning parameter $\gamma_k$ adaptively.

Lipschitz bound. We say a function $g : \mathbb{R}^d \to \mathbb{R}^k$ is $L$-Lipschitz if for all $p, q \in \mathbb{R}^d$ that

$$\|g(p) - g(q)\| \leq L \|p - q\|.$$

This property is useful when $g = \nabla f$ is describing the gradient of a cost function $f$. If $\nabla f$ is $L$-Lipschitz, and we set $\gamma \leq \frac{1}{L}$, then gradient descent will converge to a stationary point. Moreover, if $f$ is convex with global minimum $\alpha^*$, then after $k = O(1/\varepsilon)$ steps we can guarantee that

$$f(\alpha^{(k)}) - f(\alpha^*) \leq \varepsilon.$$

For the $k = O(1/\varepsilon)$ claim (and others stated below), we assume that $f(\alpha^{(0)}) - f(\alpha^*)$ is less than some absolute constant. Intuitively, the closer we start to the optimum, the fewer steps it will take.

For a convex quadratic function $f$ (e.g., most cost functions derived by sum of squared errors), then the gradient $\nabla f$ is $L$-Lipschitz.

Example: Strongly Convex Function

We show an example $\eta$-strongly convex function $f$, in blue. At any point $p$, it is sandwiched between two convex quadratic functions in green. The convex quadratic function which lower bounds $f$ has an $L$-Lipschitz gradient.
**Strongly Convex Functions.** A function \( f : \mathbb{R}^d \to \mathbb{R} \) is \( \eta \)-strongly convex with parameter \( \eta > 0 \) if for all \( \alpha, p \in \mathbb{R}^d \) then

\[
f(p) \leq f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{\eta}{2} \|p - \alpha\|^2.
\]

Intuitively, this implies that \( f \) is at least quadratic. That is, along any direction \( u = \frac{p - \alpha}{\|p - \alpha\|} \) the function \( \langle f, u \rangle \) is 1-dimensional, and then its second derivative \( \frac{d^2}{du^2} \langle f, u \rangle \) is strictly positive; it is at least \( \eta > 0 \). Similarly, saying a function \( f \) has an \( L \)-Lipschitz gradient is equivalent to the condition that \( \frac{d^2}{du^2} \langle f(\alpha), u \rangle \leq L \) for all \( \alpha, p \in \mathbb{R}^d \) where \( u = \frac{p - \alpha}{\|p - \alpha\|} \).

For an \( \eta \)-strongly convex function \( f \), that has an \( L \)-Lipschitz gradient, with global minimum \( \alpha^* \), then gradient descent with learning rate \( \gamma \leq 2/(\eta + L) \) after only \( k = O(\log(1/\varepsilon)) \) steps will achieve

\[
f(\alpha^{(k)}) - f(\alpha^*) \leq \varepsilon.
\]

The constant in \( k = O(\log(1/\varepsilon)) \) depends on the condition number \( L/\eta \). The conditions of this bound, imply that \( f \) is sandwiched between two convex quadratic functions; specifically for any \( \alpha, p \in \mathbb{R}^d \) that we can bound

\[
f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{L}{2} \|p - \alpha\|^2 \leq f(p) \leq f(\alpha) + \langle \nabla f(\alpha), p - \alpha \rangle + \frac{\eta}{2} \|p - \alpha\|^2.
\]

When an algorithm converges at such a rate (takes \( O(\log(1/\varepsilon)) \) steps to obtain \( \varepsilon \) error), it is known as linear convergence since the log-error \( \log(f(\alpha^{(k)}) - f(\alpha^*)) \) looks like a linear function of \( k \).

In practice, since many functions we consider will be convex quadratic functions (e.g., are derived from sum of squared error cost functions), then the error will decreases exponentially fast in terms of the number of steps of gradient descent, if the learning rate is set sufficiently small. That is, only a constant number of steps are required to resolve each bit of precision in the function value at the optimum! However, if the learning rate is set too small, then the constant (number of steps to resolve one bit) will increase.

So at this point, we have explained that for many situations there is a learning rate for which gradient descent will work extremely well. If we can analytically bound various properties of the second derivative of the function, then we can use these bounds to choose such a rate. However, we have not yet explained a formal way to find such a rate in general where we can only evaluate the gradient at any point \( \alpha \in \mathbb{R}^d \).

**Line search.** An alternative, referred to as “line search” is to solve for the (approximately) optimal \( \gamma_k \) at each step. Once we have computed the gradient \( \nabla f(\alpha^{(k)}) \) then we have reduced the high-dimensional minimization problem to a one-dimensional problem. Note if \( f \) is convex, then \( f \) restricted to this one-dimensional search is also convex. We still need to find the minimum of an unknown function, but we can perform some procedure akin to binary search. We first find a value \( \gamma' \) such that

\[
f(\alpha^{(k)} - \gamma' \nabla f(\alpha^{(k)})) > f(\alpha^{(k)})
\]

then we keep subdividing the region \([0, \gamma']\) into pieces, and excluding ones which cannot contain the minimum.

For instance the golden section search divides a range \([b, t]\) containing the optimal \( \gamma_k \) into three sub-intervals (based on the golden ratio) so \([b, t] = [b, b'] \cup [b', t'] \cup (t', t]\). And each step, we can determine that either \( \gamma_k \notin [b, b'] \) if \( f(t') < f(b') \), or \( \gamma_k \notin (t', t] \) if \( f(b') < f(t') \). This reduces the range to \([b', t]\) or \([b, t']\), respectively, and we recurse.

In other situations, we can solve for the optimal \( \gamma_k \) exactly at each step. This is the case if we can again analytically take the derivative \( \frac{d}{d\gamma} (f(\alpha^{(k)}) - \gamma \nabla f(\alpha^{(k)})) \) and solve for the \( \gamma \) where it is equal to 0.
Adjustable rate. In practice, line search is often slow. Also, we may not have a Lipschitz bound. It is often better to try a few fixed $\gamma$ values, probably being a bit conservative. As long as $f(\alpha^{(k)})$ keep decreasing, it works well. This also may alert us if there is more than one local minimum if the algorithm converges to different locations.

An algorithm called “backtracking line search” automatically tunes the parameter $\gamma$. It uses a fixed parameter $\beta \in (0, 1)$ (preferably in $(0.1, 0.8)$; for instance use $\beta = 3/4$). Start with a large step size $\gamma$ (e.g., $\gamma = 1$). Then at each step of gradient descent at location $\alpha$, if

$$f(\alpha - \gamma \nabla f(\alpha)) > f(\alpha) - \frac{\gamma}{2} ||\nabla f(\alpha)||^2$$

then update $\gamma = \beta \gamma$. This shrinks $\gamma$ over the course of the algorithm, and if $f$ is strongly convex, it will eventually decrease $\gamma$ until it satisfies the condition for linear convergence.

---

**Example: Gradient Descent with Fixed Learning Rate**

Consider the function $f$ where $\alpha = (x, y) \in \mathbb{R}^2$ is defined

$$f(x, y) = \left(\frac{3}{4} x - \frac{3}{2}\right)^2 + \left(y - 2\right)^2 + \frac{1}{4} xy$$

and has gradient

$$\nabla f(x, y) = \left(\frac{9}{8} x - \frac{1}{4} y + \frac{1}{4} y, \ 2y - 4 + \frac{1}{4} x\right).$$

We run gradient descent for 10 iterations within initial position $(5, 4)$, while varying the learning rate in the range $\gamma = \{0.01, 0.1, 0.2, 0.3, 0.5, 0.75\}$.

We see that with $\gamma$ very small, the algorithm does not get close to the minimum. When $\gamma$ is too large, then the algorithm jumps around a lot, and is in danger of not converging. But at a learning rate of $\gamma = 0.3$ it converges fairly smoothly and reaches a point where $||\nabla f(x, y)||$ is very small. Using $\gamma = 0.5$ almost overshoots in the first step; $\gamma = 0.3$ is smoother, and it is probably best to use a curve that looks smooth like that one, but with a few more iterations.
import matplotlib as mpl
mpl.use('PDF')
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as LA

def func(x, y):
    return (0.75*x-1.5)**2 + (y-2.0)**2 + 0.25*x*y

def func_grad(vx, vy):
    dfdx = 1.125*vx - 2.25 + 0.25*vy
    dfdy = 2.0*vy - 4.0 + 0.25*vx
    return np.array([dfdx, dfdy])

# prepare for contour plot
xlist = np.linspace(0, 5, 26)
ylist = np.linspace(0, 5, 26)
x, y = np.meshgrid(xlist, ylist)
z = func(x, y)
lev = np.linspace(0, 20, 21)

# iterate location
v_init = np.array([5, 4])
num_iter = 10
values = np.zeros([num_iter, 2])

for gamma in [0.01, 0.1, 0.2, 0.3, 0.5, 0.75]:
    values[0, :] = v_init
    v = v_init

    # actual gradient descent algorithm
    for i in range(1, num_iter):
        v = v - gamma * func_grad(v[0], v[1])
        values[i, :] = v

    # plotting
    plt.contour(x, y, z, levels=lev)
    plt.plot(values[:, 0], values[:, 1], 'r-')
    plt.plot(values[:, 0], values[:, 1], 'bo')
    grad_norm = LA.norm(func_grad(v[0], v[1]))
title = "gamma=%0.2f|final_grad|%0.3f" % (gamma, grad_norm)
plt.title(title)
file = "gd-%2.0f.pdf" % (gamma*100)
plt.savefig(file, bbox_inches='tight')
plt.clf()
plt.cla()

6.4 Fitting a Model to Data

For data analysis, the most common use of gradient descent is to fit a model to data. In this setting we have a data set \( (X, y) = \{ (x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \} \in \mathbb{R}^d \times \mathbb{R} \) and a family of models \( M \) so each possible model \( M_\alpha \) is defined by a \( d \)-dimensional vector \( \alpha = \{ \alpha_1, \alpha_2, \ldots, \alpha_k \} \) for \( k \) parameters.

Next we define a *loss function* \( L((X, y), M_\alpha) \) which measures the difference between what the model predicts and what the data values are. To choose which parameters generate the best model, we let \( f(\alpha) : \mathbb{R}^d \to \mathbb{R} \) be our function of interest, defined \( f(\alpha) = L((X, y), M_\alpha) \). Then we can run gradient descent to
find our model $M_{\alpha^*}$. For instance we can set
\[
f(\alpha) = L((X, y), M_{\alpha}) = \text{SSE}(P, M_{\alpha}) = \sum_{(x_i, y_i) \in (X, y)} (y_i - M_{\alpha}(x_i))^2.
\] (6.1)

This is used for examples including maximum likelihood (or maximum log-likelihood) estimators from Bayesian inference. This includes finding a single point estimator with Gaussian (where we had a closed-form solution), but also many other variants (often where there is no known closed-form solution). It also includes least squares regression and its many variants; we will see this in much more detail next. And will include other topics (including clustering, PCA, classification) we will see later in class.

### 6.4.1 Least Mean Squares Updates for Regression

Now we will work through how to use gradient descent for simple quadratic regression on 1-dimensional explanatory variables. That is, this will specify the function $f(\alpha)$ in equation (6.1) to have $k = 3$ parameters as $\alpha = (\alpha_0, \alpha_1, \alpha_2)$, and so for each $(x_i, y_i) \in (X, y)$ we have $x_i \in \mathbb{R}$. Then we can write the model again as a dot product
\[
M_{\alpha}(x_i) = \langle \alpha, (1, x_i, x_i^2) \rangle = \alpha_0 + \alpha_1 x_i + \alpha_2 x_i^2
\]

It is straightforward to generalize to linear regression, multiple-explanatory variable linear regression, or general polynomial regression from here. For instance, in this case, we can represent $x_i$ as the 3-dimensional feature vector $q = (g_0 = x_i^0 = 1, g_1 = x_i^1, g_2 = x_i^2)$ of explanatory variables. Then the model is $M_{\alpha}(x_i) = \langle \alpha, q \rangle$. For simple linear regression, we simply omit the quadratic term $\alpha_2 x_i^2$ from the above model. For other polynomial forms of regression, the expansion simply includes more terms and powers. And for multiple explanatory variables they are each given a corresponding feature coordinate, possible to some power, or multiplied with other explanatory variables in the polynomial models. We will continue using 1-dimensional quadratic regression as an example.

Now to specify the gradient descent step
\[
\alpha := \alpha - \gamma \nabla f(\alpha)
\]
we only need to define $\nabla f(\alpha)$. We will first show this for the case where $n = 1$, that is when there is a single data point $(x_1, y_1)$. For quadratic regression, the cost function $f_1(\alpha) = (\alpha_0 + \alpha_1 x_1 + \alpha_2 x_1^2 - y_1)^2$ is convex. Next derive
\[
\begin{align*}
\frac{d}{d\alpha_j} f(\alpha) &= \frac{d}{d\alpha_j} (M_{\alpha}(x_1) - y_1)^2 \\
&= 2(M_{\alpha}(x_1) - y_1) \frac{d}{d\alpha_j} (M_{\alpha}(x_1) - y_1) \\
&= 2(M_{\alpha}(x_1) - y_1) \frac{d}{d\alpha_j} \left( \sum_{j=0}^{2} \alpha_j x_1^j - y_1 \right) \\
&= 2(M_{\alpha}(x_1) - y_1) x_1^j
\end{align*}
\]

Using this convenient form (which generalizes to any polynomial model), we define
\[
\nabla f(\alpha) = \left( \frac{d}{d\alpha_0} f(\alpha), \frac{d}{d\alpha_1} f(\alpha), \frac{d}{d\alpha_2} f(\alpha) \right) = 2 \left( (M_{\alpha}(x_1) - y_1), (M_{\alpha}(x_1) - y_1)x_1, (M_{\alpha}(x_1) - y_1)x_1^2 \right).
\]

Applying $\alpha := \alpha - \gamma \nabla f(\alpha)$ according to this specification is known as the LMS (least mean squares) update rule or the Widrow-Hoff learning rule. Quite intuitively, the magnitude the update is proportional to the residual norm $(M_{\alpha}(x_1) - y_1)$. So if we have a lot of error in our guess of $\alpha$, then we take a large step; if we do not have a lot of error, we take a small step.
6.4.2 Decomposable Functions

To generalize this to multiple data points \((n > 1)\), there are two standard ways. Both of these take strong advantage of the cost function \(f(\alpha)\) being decomposable. That is, we can write

\[
f(\alpha) = \sum_{i=1}^{n} f_i(\alpha),
\]

where each \(f_i\) depends only on the \(i\)th data point \(p_i \in P\). In particular, where \(p_i = (x_i, y_i)\), then for quadratic regression

\[
f_i(\alpha) = (M_\alpha(x_i) - y_i)^2 = (\alpha_0 + \alpha_1 x_i + \alpha_2 x_i^2 - y_i)^2.
\]

First notice that since \(f\) is the sum of \(f_i\)'s, where each is convex, then \(f\) must also be convex; in fact the sum of these usually becomes strongly convex (as long as the corresponding feature vectors are full rank). Also two approaches towards gradient descent will take advantage of this decomposition in slightly different ways. This decomposable property holds for most loss functions for fitting a model to data.

**Batch gradient descent.** The first technique, called batch gradient descent, simply extends the definition of \(\nabla f(\alpha)\) to the case with multiple data points. Since \(f\) is decomposable, then we use the linearity of the derivative to define

\[
\frac{d}{d\alpha_j} f(\alpha) = \sum_{i=1}^{n} \frac{d}{d\alpha_j} f_i(\alpha) = \sum_{i=1}^{n} 2(M_\alpha(x_i) - y_i)x_i^j
\]

and thus

\[
\nabla f(\alpha) = \left(2 \sum_{i=1}^{n} (M_\alpha(x_i) - y_i), 2 \sum_{i=1}^{n} (M_\alpha(x_i) - y_i)x_i, 2 \sum_{i=1}^{n} (M_\alpha(x_i) - y_i)x_i^2\right)
\]

\[
= \sum_{i=1}^{n} (2(M_\alpha(x_i) - y_i), 2(M_\alpha(x_i) - y_i)x_i, 2(M_\alpha(x_i) - y_i)x_i^2)
\]

\[
= 2 \sum_{i=1}^{n} (M_\alpha(x_i) - y_i)(1, x_i, x_i^2).
\]

That is, the step is now just the sum of the terms from each data point. Since \(f\) is (strongly) convex, then we can apply all of the nice convergence results discussed about (strongly) convex \(f\) before. However, computing \(\nabla f(\alpha)\) each step takes \(O(n)\) time, which can be slow for large \(n\) (i.e., for large data sets).

**Algorithm 6.4.1 Incremental Gradient Descent**

\[
\text{Algorithm 6.4.1 Incremental Gradient Descent}(f, \alpha_{\text{start}})
\]

\[
\text{initialize } \alpha^{(0)} = \alpha_{\text{start}} \in \mathbb{R}^d, i = 1.
\]

\[
\text{repeat}
\]

\[
\alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f_i(\alpha^{(k)})
\]

\[
i := (i + 1) \mod n
\]

\[
\text{until } (\|\nabla f(\alpha^{(k)})\| \leq \tau) \quad (\star)
\]

\[
\text{return } \alpha^{(k)}
\]

**Stochastic gradient descent.** The second technique is called incremental gradient descent; see Algorithm 6.4.1. It avoids computing the full gradient each step, and only computes \(\nabla f_i(\alpha)\) for a single data point \(p_i \in P\). For quadratic regression with \(M_\alpha(x) = \langle \alpha, (1, x, x^2) \rangle\) it is

\[
\nabla f_i(\alpha) = 2(M_\alpha(x_i) - y_i)(1, x_i, x_i^2).
\]
Implementation Hints:

(*) The norm of a single gradient is not stable. Instead a better stopping condition averages the gradient norm over several (let's say $B$ steps). The condition may then be $\frac{1}{B} \sum_{b=0}^{B-1} \| \nabla f(\alpha^{(k-b)}) \| \leq \tau$, and is only checked after $B$ steps are taken.

A more common variant of this is called stochastic gradient descent; see Algorithm 6.4.2. Instead of choosing the data points in order, it selects a data point $p_i$ at random each iteration (the term “stochastic” refers to this randomness).

Algorithm 6.4.2 Stochastic Gradient Descent ($f, \alpha_{\text{start}}$)

initialize $\alpha^{(0)} = \alpha_{\text{start}} \in \mathbb{R}^d$

repeat

Randomly choose $i \in \{1, 2, \ldots, n\}$

$\alpha^{(k+1)} := \alpha^{(k)} - \gamma_k \nabla f_i(\alpha^{(k)})$

until $(\| \nabla f(\alpha^{(k)}) \| \leq \tau)$ (*)&

return $\alpha^{(k)}$

On very large data sets (i.e., big data!), these algorithms are often much faster than the batch version since each iteration now takes $O(1)$ time. However, it does not automatically inherit all of the nice convergence results from what is known about (strongly) convex functions. Yet in many settings, there is an abundance of data points described by the same model. They should have a roughly similar effect. In practice when one is far from the optimal model, these steps converge about as well as the batch version (but much much faster in runtime). When one is close to the optimal model, then the incremental / stochastic variants may not exactly converge. However, if one is satisfied to reach a point that is close enough to optimal, there are some randomized (PAC-style) guarantees possible for the stochastic variant. And in fact, for very large data sets (i.e., $n$ is very big) they typically converge before the algorithm even uses all (or even most) of the data points.
Exercises

We will use a dataset [http://www.cs.utah.edu/~jeffp/teaching/FoDA/D4.csv](http://www.cs.utah.edu/~jeffp/teaching/FoDA/D4.csv)

**Q6.1:** Consider a function \( f(x, y) \) with gradient \( \nabla f(x, y) = (x - 1, 2y + x) \). Starting at a value \((x = 1, y = 2)\), and a learning rate of \( \gamma = 1 \), execute one step of gradient descent.

**Q6.2:** Consider running gradient descent with a fixed learning rate \( \gamma \). For each of the following, we plot the function value over 10 steps (the function is different each time). Decide whether the learning rate is probably too high, too low, or about right.

1. \( f_1 \): 100, 99, 98, 97, 96, 95, 94, 93, 92, 91
2. \( f_2 \): 100, 50, 75, 60, 65, 45, 75, 110, 90, 85
3. \( f_3 \): 100, 80, 65, 50, 40, 35, 31, 29, 28, 27.5, 27.3
4. \( f_4 \): 100, 80, 60, 40, 20, 0, -20, -40, -60, -80, -100

**Q6.3:** Consider two functions

\[
\begin{align*}
 f_1(x, y) &= (x - 5)^2 + (y + 2)^2 - 2xy \\
 f_2(x, y) &= (1 - (y - 4))^2 + 20((x + 6) - (y - 4)^2)^2
\end{align*}
\]

Starting with \((x, y) = (0, 2)\) run the gradient descent algorithm for each function. Run for \( T \) iterations, and report the function value at the end of each step.

1. First, run with a fixed learning rate of \( \gamma = 0.05 \) for \( f_1 \) and \( \gamma = 0.0015 \) for \( f_2 \).
2. Second, run with any variant of gradient descent you want. Try to get the smallest function value after \( T \) steps.

For \( f_1 \) you are allowed only \( T = 10 \) steps. For \( f_2 \) you are allowed \( T = 100 \) steps.

**Q6.4:** In the first \( D4.csv \) dataset provided, use the first three columns as explanatory variables \( x_1, x_2, x_3 \), and the fourth as the dependent variable \( y \). Run gradient descent on \( \alpha \in \mathbb{R}^4 \), using the dataset provided to find a linear model

\[
\hat{y} = \alpha_0 + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3
\]

minimizing the sum of squared errors. Run for as many steps as you feel necessary. On each step of your run, print on a single line: (1) the model parameters \( \alpha^{(i)} = [\alpha_0^{(i)}, \alpha_1^{(i)}, \alpha_2^{(i)}, \alpha_3^{(i)}] \), (2) the value of a function \( f(\alpha^{(i)}) \), estimating the sum of squared errors, and (3) the gradient \( \nabla f(\alpha^{(i)}) \). (These are the sort of things you would do to check/debug a gradient descent algorithm; you may also want to plot some of these.)

1. First run batch gradient descent.
2. Second run incremental gradient descent.

Choose one method which you preferred (either is ok to choose), and explain why you preferred it to the other method.

**Q6.5:** Explain what parts of the above procedures would change if you instead are minimizing the sum of residuals, not the sum of squared residuals?

- Is the function still convex?
- Does the gradient always exist?