6 Gradient Descent

In this topic we will discuss optimizing over general functions f. Typically the function is defined $f : \mathbb{R}^d \to \mathbb{R}$; that is its domain is multi-dimensional (in this case d-dimensional) 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 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 $f : \mathbb{R}^d \to \mathbb{R}$. 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 calculous.

Let $B_r(x)$ define a Euclidean ball around a point x of radius r. That is, it includes all points $\{y \in \mathbb{R}^d \mid \|x - y\| \leq r\}$, within a Euclidean distance of r from x. We will use $B_r(x)$ to define a *local neighborhood* around a point x. 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 $x \in \mathbb{R}^d$ so for some neighborhood $B_r(x)$, all points $y \in B_r(x)$ have smaller (or equal) function value than at x: $f(y) \leq f(x)$. A *local maximum* of f is a point $x \in \mathbb{R}^d$ so for some neighborhood $B_r(x)$, all points $y \in B_r(x)$ have larger (or equal) function value than at x: $f(y) \geq f(x)$. If we remove the "or equal" condition for both definitions for $y \in B_r(x), y \neq x$, we say the maximum or minimum points are *strict*.

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

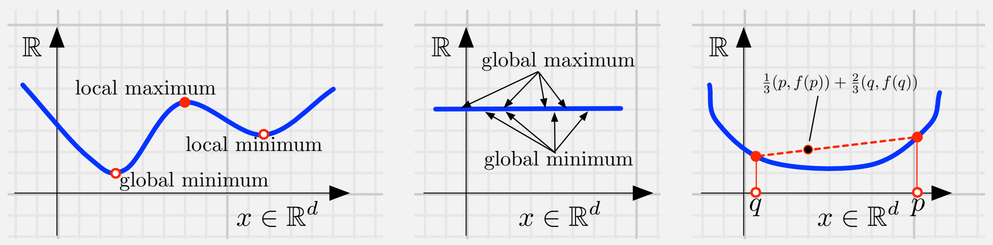
When we just use the term *minimum* or *maximum* (without local or global) it implies a local minimum or maximum.

Focusing on a function restricted to a closed and bounded subset $S \subset \mathbb{R}^d$, if the function is continuous (we won't formally define this, but it likely means what you think it does), then the function must have a global minimum and a global maximum. It may occur on the boundary of S.

A *saddle* point is a type of point $x \in \mathbb{R}^d$ so that within any neighborhood $B_r(x)$, it has points $y \in B_r(x)$ with $f(y) < f(x)$ (the lower points) and $y' \in B_r(x)$ with $f(y') > f(x)$ (the upper points). In particular, it is a saddle if there are disconnected regions of upper points (and of lower points). For $d = 1$, then there can be no saddle points. 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 x, 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 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 for any scalar $\alpha \in \mathbb{R}$, a line ℓ is the set of points

$$\ell = \{x = \alpha p + (1 - \alpha)q \mid \alpha \in \mathbb{R} \text{ and } p, q \in \mathbb{R}^d\}.$$

When $\alpha \in [0, 1]$, 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

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

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 $h(x) = f(x) + g(x)$ is convex and so is $h(x) = \max\{f(x), g(x)\}$. But one will be most important for us:

- 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 $f(x) = f(x_1, x_2, \ldots, x_d)$, and a unit vector $u = (u_1, u_2, \ldots, u_d)$, then the *directional derivative* is defined

$$\nabla_u f(x) = \lim_{h \to 0} \frac{f(x + hu) - f(x)}{h}.$$

We are interested in functions f which are *differentiable*; this implies that $\nabla_u f(x)$ is well-defined for all x 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 ith coordinate.

Then define

$$\nabla_i f(x) = \nabla_{e_i} f(x) = \frac{\mathrm{d}}{\mathrm{d}x_i} f(x).$$

It is the derivative in the ith coordinate, treating all other coordinates as constants.

We can now, for a differentiable function f, define the *gradient of f* as

$$\nabla f = \frac{\mathrm{d}f}{\mathrm{d}x_1} e_1 + \frac{\mathrm{d}f}{\mathrm{d}x_2} e_2 + \ldots + \frac{\mathrm{d}f}{\mathrm{d}x_d} e_d = \left(\frac{\mathrm{d}f}{\mathrm{d}x_1}, \frac{\mathrm{d}f}{\mathrm{d}x_2}, \ldots, \frac{\mathrm{d}f}{\mathrm{d}x_d} \right).$$

Note that ∇f is a function from $\mathbb{R}^d \to \mathbb{R}^d$, which we can evaluate at any point $x \in \mathbb{R}^d$.

Example: Gradient
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 x, for any direction (unit vector) u as

$$\nabla_u f(x) = \langle \nabla f(x), u \rangle.$$

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

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

To find the minimum of a function f, we then typically want to move from any point x in the direction $-\overline{\nabla f(x)}$; 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

$$\min_{x \in \mathbb{R}^d} f(x) \qquad \text{and/or} \qquad x^* = \arg\min_{x \in \mathbb{R}^d} f(x).$$

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

Algorithm 6.3.1 Gradient Descent($f, x_{\textsf{start}}$)

```
initialize $x^{(0)} = x_{\textsf{start}} \in \mathbb{R}^d$.
repeat
    $x^{(k+1)} := x^{(k)} - \gamma_k \nabla f(x^{(k)})$
until ($\|\nabla f(x^{(k)})\| \leq \tau$)
return $x^{(k)}$
```

Basically, for any starting point $x^{(0)}$ the algorithm moves to another point in the direction opposite to the gradient – in the direction that locally decreases f the fastest.

Stopping condition. The parameter τ is the tolerance of the algorithm. If we assume the function is differentiable, then at the minimum x^*, we must have that $\nabla f(x) = (0, 0, \ldots, 0)$. So close to the minimum, it should also have a small norm. The algorithm may never reach the true minimum (and we don't 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.

6.3.1 Learning Rate

The most critical parameter of gradient descent is γ, 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 γ?

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

$$\|g(x) - g(y)\| \leq L\|x - y\|.$$

If ∇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 x^*, then after $k = O(1/\varepsilon)$ steps we can guarantee that

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

A function $f : \mathbb{R}^d \to \mathbb{R}$ is *η-strongly convex* with parameter $\eta > 0$ if for all $x \in \mathbb{R}^d$ and any unit vector $u \in \mathbb{R}^d$ (that is $\|u\| = 1$) then

$$\langle \nabla f(x), u \rangle \geq \eta.$$

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

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

The constant in $k = O(\log(1/\varepsilon))$ depends on the condition number L/η; recall that for any unit vector u we have $L \geq \langle \nabla f(x), u \rangle \geq \eta$. When an algorithm converges at such a rate, it is known as *linear convergence* since the log-error $\log(f(x^{(k)}) - f(x^*))$ looks like a linear function of k.

Line search. In other cases, we can search for γ_k at each step. Once we have computed the gradient $\nabla f(x^{(k)})$ 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 γ' such that

$$f\left(x^{(k)} - \gamma' \nabla f(x^{(k)})\right) > f(x^{(k)})$$

then we keep subdividing the region $[0, \gamma']$ into pieces which must contain the minimum (for instance the *golden section search* keeps dividing by the golden ratio).

In other situations, we can solve for the optimal γ_k exactly at each step. This is the case if we can again analytically take the derivative $\frac{\mathrm{d}}{\mathrm{d}\gamma}\left(f(x^{(k)}) - \gamma \nabla f(x^{(k)})\right)$ and solve for the γ 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 γ values, probably being a bit conservative. As long as $f(x^{(k)})$ keep decreasing, it works well. This also may alert us if there there is more than one local minimum if the algorithm converges to different locations.

An algorithm called "backtracking line search" automatically tunes the parameter γ. 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 γ (e.g., $\gamma = 1$). Then at each step of gradient descent at location x, if

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

then update $\gamma = \beta\gamma$. This shrinks γ over the course of the algorithm to ensure it will satisfy the condition for linear convergence.

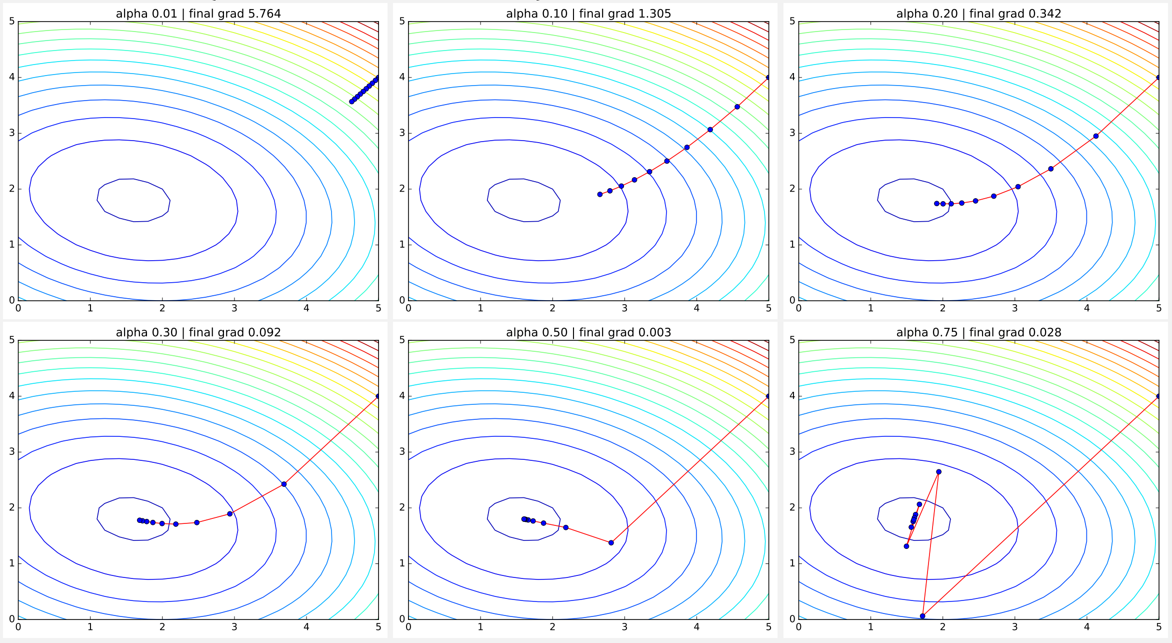
Example: Gradient Descent with Fixed Learning Rate
Consider the function

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

with gradient

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

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 γ very small, the algorithm does not get close to the minimum. When γ is too large, then the algorithm jumps around a lot, and is in danger of not converging. But at a learning rate of $\gamma = 0.5$ 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 its 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 alpha 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 - alpha * 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 = "alpha %0.2f | final grad %0.3f" % (alpha,grad_norm)
  plt.title(title)
  file = "gd-%2.0f.pdf" % (alpha*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 $P = \{p_1, p_2, \ldots, p_n\}$ and a family of models $\mathcal{M}$ so each possible model M_α is defined by a d-dimensional vector $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_d\}$ for p parameters.

Next we define a *loss function* $L(P, M_\alpha)$ 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 $f(\alpha) = L(P, M_\alpha)$. Then we can run gradient descent to find our model M_{α^*}. For instance we can set

$$f(\alpha) = L(P, M_\alpha) = \mathsf{SSE}(P, M_\alpha) = \sum_{p \in P} (p_y - M_\alpha(p_x))^2. \tag{6.1}$$

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 Linear Regression

Now we will work through how to use gradient descent for simple quadratic regression. It should be straightforward to generalize to linear regression, multiple-explanatory variable linear regression, or general polynomial regression from here. This will specify the function $f(\alpha)$ in equation (6.1) to where $d = 3$, $\alpha = (\alpha_0, \alpha_1, \alpha_2)$, for each $p \in P$ we have $p = (p_x, p_y) \in \mathbb{R}^2$, but we consider a vector $q = (q_0 = p_x^0, q_1 = p_x^1, q_2 = p_x^2)$ as the set of explanatory variables. Finally,

$$M_\alpha(p_x) = \alpha_0 + \alpha_1 p_x + \alpha_2 p_x^2 = \alpha_0 q_0 + \alpha_1 q_1 + \alpha_2 q_2.$$

To specify the gradient descent step:

$$\alpha := \alpha - \gamma \nabla f(\alpha)$$

we 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 $p = (p_x, p_y) = (x_1, y_1)$. It should now be easy to verify that the cost function $f_1(\alpha) = (\alpha_0 + \alpha_1 x_1 + \alpha_2 x_1^2 - y_1)^2$ is convex. Next, derive

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\alpha_j} f(\alpha) &= \frac{\mathrm{d}}{\mathrm{d}\alpha_j}(M_\alpha(x_1) - y_1)^2 \\
&= 2(M_\alpha(x_1) - y_1)\frac{\mathrm{d}}{\mathrm{d}\alpha_j}(M_\alpha(x_1) - y_1) \\
&= 2(M_\alpha(x_1) - y_1)\frac{\mathrm{d}}{\mathrm{d}\alpha_j}(\sum_{j=0}^{2} \alpha_j x_1^j - y_1) \\
&= 2(M_\alpha(x_1) - y_1)x_1^j
\end{aligned}$$

Thus, we define

$$\begin{aligned}
\nabla f(\alpha) &= \left(\frac{\mathrm{d}}{\mathrm{d}\alpha_0} f(\alpha), \frac{\mathrm{d}}{\mathrm{d}\alpha_1} f(\alpha), \frac{\mathrm{d}}{\mathrm{d}\alpha_2} f(\alpha)\right) \\
&= 2\left((M_\alpha(x_1) - y_1),\ \ (M_\alpha(x_1) - y_1)x_1,\ \ (M_\alpha(x_1) - y_1)x_1^2\right)
\end{aligned}$$

Applying $\alpha := \alpha - \gamma \nabla f(\alpha)$ according to this specification is known as the *LMS (least mean squares)* update rule or the *Widrow-Huff 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 α, then we take a large step; if we do not have a lot of error, we take a small step.

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 ith data point $p_i \in P$. In particular, where $p_i = (x_i, y_i)$, then

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

First notice that since f is the sum of f_is, where each is convex, then f must also be convex; in fact the sum of these usually becomes strongly convex. 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

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

and thus

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

That is, the step is now just the sum of the terms from each data point. Since f is 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.

Stochastic gradient descent. The second technique is called *incremental gradient descent*. It avoids computing the full gradient each step, and only computes it for f_i for a single data point $p_i \in P$; see algorithm 6.4.1.

Algorithm 6.4.1 Incremental Gradient Descent$(f, x_{\textsf{start}})$

```
initialize $x^{(0)} = x_{\textsf{start}} \in \mathbb{R}^d$; $i = 1$.
repeat
    $x^{(k+1)} := x^{(k)} - \gamma_k \nabla f_i(x^{(k)})$
    $i := (i+1) \mod n$
until $(\|\nabla f(x^{(k)})\| \leq \tau)$ (or perhaps average of a few steps)
return $x^{(k)}$
```

A more common variant of this is called *stochastic gradient descent*. 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).

These algorithms are often much faster than the batch version since each iteration now takes $O(1)$ time. However, it does not automatically inherent all of the nice convergence results from what is known about (strongly) convex functions. Yet, since in most settings we are interested in, there is an abundance of data points from the same model. This implies 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). When one is close to the optimal model, then the incremental / stochastic variants may not exactly converge. However, if one is willing to reach a point that is not too optimal, there are some randomized (PAC-style) guarantees possible for the stochastic variant. And in fact, for very large data sets (n is very big) they typically converge before the algorithm even uses all (or even most) of the data points.