Optimization by Gradient Descent
ML_2022: Machine Learning

https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/gradient_lecture/gradient_lecture.pdf

A series of downhill steps will take us to the bottom, as long as we know how to stop!

Gradient Descent

Gradient descent is an alternative method for minimizing the mean square error MSE(c).

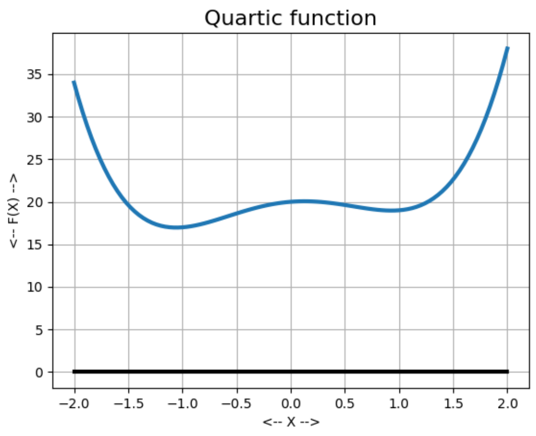
- *The algorithm looks very simple.*
- *In fact, the form of MSE(c) makes the problem even easier than usual.*
- *The simple algorithm usually needs some tuning and parameter settings for best performance;*
- *For our data problems, however, normalization can be very important;*
- *For large problems, even gradient descent may need to be replaced by stochastic or mini-batch versions;*

1 Minimize a quartic function using gradient descent

Gradient descent is an optimization method that can be applied for a wide variety of computational problems, even outside our machine learning applications. Although we want to apply this method to data problems where we minimize MSE(c), we start with an abstract mathematical case that is easy to define and illustrate.

Consider the function we will call *quartic()*, whose formula is:

$$f(x) = 2x^4 - 4x^2 + x + 20$$



The quartic() function

Suppose $f(x)$ measures a cost and we seek a value x which minimizes it. We will suppose we don't know much about the function's behavior, except that the minimizer is probably somewhere in $[-2,+2]$, and we know how to evaluate the derivative $f'(x)$.

We might start at a point x in $[-2,+2]$ that we suspect is close to the minimizer. Our next step could be to move a little to the left or right from our starting point. If we happen to know $f'(x)$, then:

- if $0 < f'(x)$, then $f(x)$ decreases to the left: $\swarrow$;
- if $f'(x) < 0$, then $f(x)$ decreases to the right: $\searrow$;

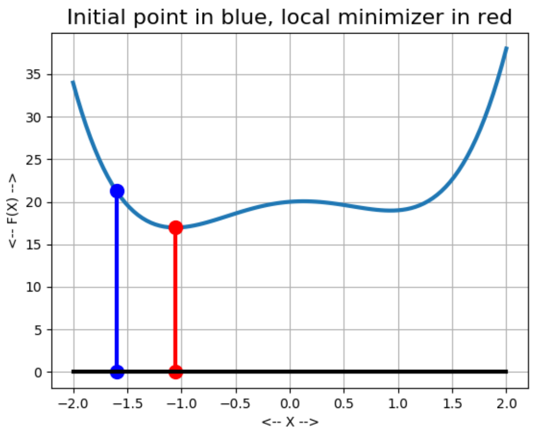
For our quartic function, the derivative is

$$f'(x) = 8x^3 - 8x + 1$$

If we started our investigation at $x = 1.5$, then $f(1.5) = 22.63$, and $f'(1.5) = 16.0$ so for our next estimate of the minimizer, we should move "a little bit" to the left. If we started our investigation at $x = -1.8$, then $f(-1.5) = 26.235$, and $f'(-1.5) = -31.256$ and so we should move "a little bit' to the right. If we take small steps, and enough of them, we gradually come closer and closer to $x \approx -1.057$, where the function reaches its minimum value. One sign of this is that the derivative is nearly zero at our stopping point. The following table suggests the situation:

x	f(x)	f'(x)	
-1.8	26.24	-32.3	go right, increase x.
-1.057	16.97	0.0006	derivative very small, maybe we stop?
1.5	22.63	16.0	go left, decrease x.

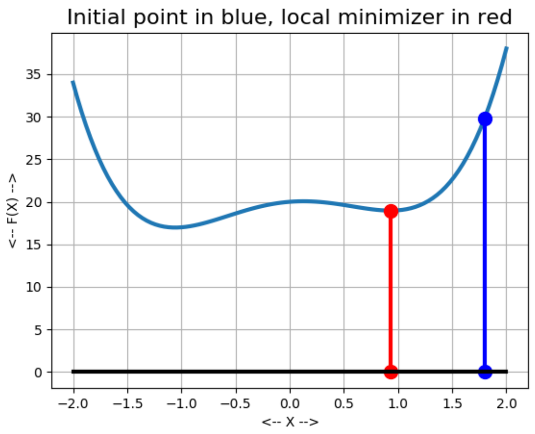
This is the heart of the **gradient descent method**. From your current estimate, take a small step, related to the **size** of the derivative, opposite to the **direction** of the derivative. Keep doing that until the stepsize gets so small that you are evidently near a critical point. If your stepsize causes the function value to increase, back up and try a smaller stepsize.



Starting at $x = -1.6$, gradient descent can approximate the minimizer.

2 The Local Minimum Problem

Calculus tells us that when the derivative of a function $f(x)$ is zero, the function has reached a `critical point`, either a minimum, a maximum, or an inflection point. During gradient descent, we are moving downhill, so let's assume that we are very unlikely to end up at a maximum value, and we are also pretty unlikely to get trapped by an inflection point. But unfortunately, calculus can only promise that we are likely to find a *local minimum* by this method, that is, a value of x for which $f(x)$ is the smallest in some "neighborhood". Any little dent in the curve, as long as it would hold water in a rainstorm, counts as a local minimum. Over the range that we have plotted the quartic function, you can see two local minimizers, but one of them is much lower than the other, and presumably, that's the one we want. But watch what happens when we pick a starting point on the right hand side of the range:



Starting at $x = 1.8$, gradient descent may be attracted to the unsatisfactory local minimizer.

If you don't know any special properties of your function $f(x)$, and you are trying to find its minimizer, then when your algorithm returns a result, it will be difficult to know whether you have actually found the very lowest value of $f(x)$ over the region, or have simply fallen into a local minimum "trap". This is a real problem, and a number of procedures and tricks have been devised to overcome it.

Luckily, our linear least squares problem enjoys a special property: it always has one and only one minimizer. So any procedure that can find a minimizer will find the answer we want.

Later, when we look at other kinds of minimization problems, our local minimum problem will pop up again, but for now, we can stop worrying about it!

3 But didn't we already solve the linear least squares problem?

Recall, from our previous discussions, how we started with n samples of d independent variables x and a dependent variable y. We constructed a linear system of equations

$$Xc = y \tag{1}$$

and then defined the `mean square error` or MSE to be

$$\text{MSE}(c) = \frac{1}{n}\sum_{i=0}^{i<n}\sum_{j=0}^{j<d}(X_{i,j}c_j - y_i)^2$$

To focus our solution effort, we defined the **least squares problem**:

Problem 1 *Given the $n \times d$ matrix X and the n vector y, determine the d vector c of coefficients which achieves the minimum value of MSE(c).*

We came up with a direct approach to solving this problem, using three different algorithms from linear algebra, which were implemented in five different Python procedures.

In data science, however, some problems may have n taking on enormous values, in the hundred thousands or higher. This means that any linear algebraic approach that requires the formation of the corresponding matrix, (especially $X'X$!) may be more expensive than we would like. In such cases, methods based on gradient descent can be compared.

We can compute the i-th component of the gradient as:

$$\frac{\partial\ \text{MSE}}{\partial c_i} = \frac{2}{n}\sum_{j=0}^{j<d} X_{i,j}(X_{i,j}c_j - y_i)$$

or, using linear algebra:

$$\frac{\partial\ \text{MSE}}{\partial c} = \frac{2}{n}X'(Xc - y)$$

Thus, we can find an alternative method of function minimization, using the gradient as a descent direction. Now we need to start thinking in terms of an algorithm.

4 Gradient descent can look very simple

The heart of the gradient descent method is a simple iteration in which the current solution is used to generate a gradient, and a multiple of the gradient is used to update the solution. Given independent data x and dependent data y, we might be seeking a linear model of the form:

$$y \approx \theta_0 + \theta_1 * x$$

Of course, we need to gather the data, and set some parameters as well, but here is what a bare bones version of the code might look like, after we generate some artificial data X and y:

```
learning_rate = 0.5
iterations = 1000
n = 100
theta = np.random.rand(2,1)

for it in range(iterations):
  gradient_vector = (2/n) * X.T.dot(X.dot(theta) - y)  # compare X'(X c-y) above
  theta = theta - learning_rate * gradient_vector
```

Listing 1: Gradient descent in 2 lines.

5 Gradient descent can blow up

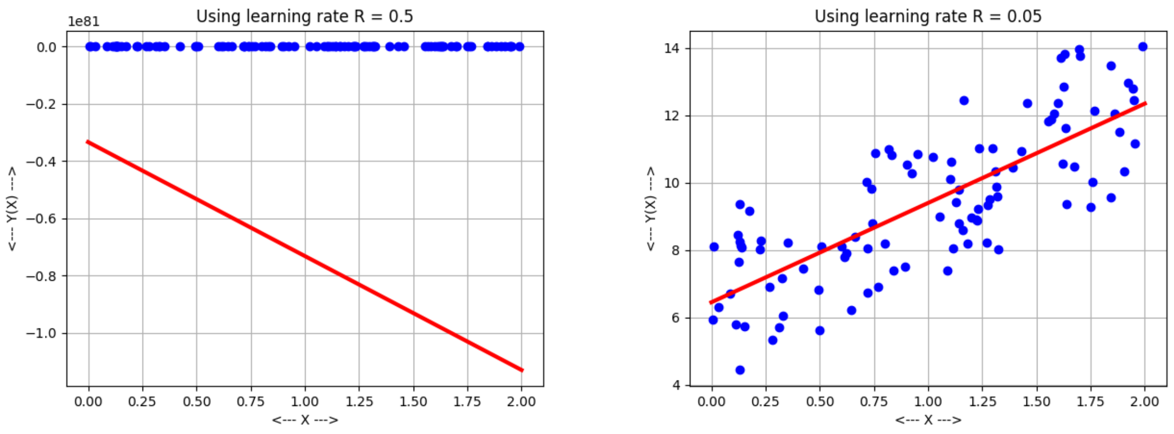
The version of gradient descent shown above is taken directly from a text book. So let's assume it's written correctly, and we'll try it out on a version of our problem in which we start with a formula y = b + a*x, perturb each data point by a small random amount, and then ask gradient descent to give us a linear approximation to the result.

```
  import numpy as np

  n = 100
  X = 2 * np.random.rand(n,1)
  y = 4 + 3*X + 5 * np.random.rand(n,1)
  X = np.c_ [ np.ones(n), X ]

  for r in [ 0.5, 0.05 ]:  # Try the book's learning rate first!

    theta = np.random.rand(2,1)
    iterations = 1000

    for it in range(iterations):
      gradient_vector = (2/n) * X.T.dot(X.dot(theta) - y)
      theta = theta - r * gradient_vector

    r6 = np.dot ( X, theta ) - y
    mse6 = 1 / n * sum ( r6**2 )
```

Listing 2: Gradient descent test.

The reason we use two different learning rates is because the first value is a disaster! The values of `theta` simply blow up. Using the second, smaller, learning rate seems to have found the solution.

```
Using learning rate r =  0.5
theta =  [[-3.33594517e+80][-3.98297312e+80]]  mse6 =  [5.94434886e+161]

Using learning rate r =  0.05
theta =  [[6.44430785][2.95246948]]            mse6 =  [1.92383461]
```



Gradient descent fails with a bad learning rate.
Look carefully at the vertical scale in the left plot!

How could we get such a bad result the first time? Why did the second result come out OK?

Another question we need to ask ourselves is whether we really had to take 1000 iterations. What happens if the process converged much sooner? And how would we know if the process had not yet converged, and needed to take more steps?

6 Taming the gradient descent method

Calculus guarantees that if the function $f(x)$ has a derivative, then, at least for a small enough stepsize Δx, the function must decrease in the direction $-\frac{df}{dx}\Delta x$. The fact that, using a learning rate `r = 0.5` allowed our gradient descent method to fail, with the function value (MSE) steadily increasing, can be explained by assuming that the resulting stepsize we used was too large. The learning rate is a multiplier used to control the stepsize, and so, apparently, reducing it to `r = 0.05` meant our steps were small enough to guarantee that MSE would decrease, and so with small steps we were able to shuffle towards the minimizer.

A reasonable procedure would be to start with a default learning rate, perhaps simply `r = 1`. Compute the gradient vector as usual. Now we begin a second loop. Using the current learning rate, compute a new tentative value `theta_next`. Evaluate the corresponding MSE. If it increases, then cut `r` in half and repeat the loop. Otherwise, accept `theta_next` as the next value of `theta`.

Note that, at some point, no matter how much you decrease the learning rate, MSE might not go down. This can happen for two reasons:

1. the stepsize is so small that adding it to `theta` makes no difference;
2. or the current `theta` might be so close to the minimizer that no further step can be taken.

Remember that, near the minimizer, the gradient vector must be approaching zero.

For now, let's simply try to sketch a safer gradient descent code that

- starts with learning rate `r = 1`;
- computes the next iterate as `theta_next`;
- computes `MSE_next` for `theta_next`;
- if `MSE_next < MSE`, accept `theta_next`;
- Otherwise, multiplies `r` by 1/2 and tries again to compute `theta_next`;

Note that we still haven't accounted for the fact that if we get close to the minimizer, we may never find a value of `r` small enough to allow a further step!

7 "Safer" gradient descent pseudocode

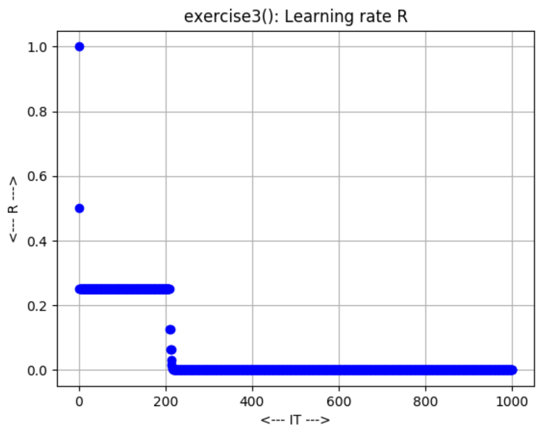
Here is a sketch of what our gradient descent code might look like, if we want to be sure to avoid the problem of the MSE of our iterates blowing up.

```
initial theta is given
mse of theta is computed

it = 0
r = 1.0

while ( not done )

  if ( itmax <= it )
    done = True
    break

  gradient_vector = (2/n) * X.T.dot(X.dot(theta) - y)

  theta_next = theta - r * gradient_vector
  mse_next = ...

  if ( mse < mse_next )
    r = r / 2
  else
    theta = theta_next
    mse = mse_next
```

Listing 3: Pseudocode for gradient descent in 1D.

8 "Is it safe?"

Let's go back to our example that blew up before. We'll start with $r = 1$, but now we will use the "safer" code, and let the learning rate decrease as needed. The code will get the correct linear formula to approximate the data, but the interesting thing is to watch what happens to the learning rate r over the 1000 iterations:



How the learning rate decreases over the iteration.

The initial learning rate of $r = 1$ seems to have been rejected immediately, as was the rate of $r = 0.5$. It was only by taking a step of $r = 0.25$ that the code was able to get started. That value was good until about step 200, when it had to be decreased 4 times, presumably to $r = 0.015625$, after which it's hard to tell from the plot whether there were any further reductions.

However, the important thing to realize is that, with this approach, every step we took resulted in the function going down. By rejecting any step that would increase the MSE function, we have made our algorithm much more reliable. Even if, for some reason, we don't take enough steps to get near to the minimizer, we can expect the value of MSE we reach at the end of the iteration to have decreased from where we started.

9 Knowing when to stop

The other thing that might occur to you from looking at the plot is that we seem to be taking an awful lot of tiny steps at the end. These steps are not of size $r = 0.015625$; their size is the `product` of that small value and the value of the gradient, which we know is shrinking to zero as we approach the minimizer.

Mathematically, if x and y are nonzero, then $x + y$ is NEVER equal to x. But on a computer, this can happen, for instance, if y is very small compare to x. Suppose this is happening in our gradient descent algorithm for, say, the last 500 or 600 of our 1000 iterations. Then we are really wasting our time; our steps are too small to have any effect on our iterate. So our wheels are spinning, but we're going nowhere.

This doesn't make our answer wrong, but it does mean we are wasting time. Moreover, it means that we might foolishly imagine that, given the answer we got after 1000 iterations, we could get an even better answer after 10,000 iterations. (Your boss or manager or advisor might not be satisfied with your result, and so you want to try harder...) So the right thing to do is to be aware that the step size can become so small that the iteration should stop...and stop it right then!

Doing this is actually fairly simple. We have the old value `theta` and the new value theta_next which could be our next iterate. But before we accept it, let's just check whether, in fact, the step size has become so small that `theta` and theta_next are identical. As soon as we calculate `theta_next` then, we can make a check. If our iterate was a scalar value, then the check would be especially simple, and would appear something like this:

```
    theta_next = theta - r * gradient_vector
    if ( theta_next == theta ):
      print ( "Stopping early on iteration ", it, "r = ", r )
      break
```

For us, `theta` will almost always be a vector quantity, and so the check for equality requires a little more care to write out. You will try this in the lab.

Now, if we add this improvement to our gradient descent algorithm, we can run our code for 1000 iterations, but get the result much faster:

```
gradient_descent_version3:
  Stopping early on iteration  270 r =  7.450580596923828e-09
  theta =  [[6.62337736]
 [3.00053278]]
  mse6 =  [2.29298223]
```

10 The stochastic Gauss-Seidel method

Our next topic, stochastic gradient descent, may seem a little bizarre, and hard to justify. In order to give you some context, we will start by inventing a stochastic version of a more familiar algorithm, the Gauss-Seidel method for solving a system of linear equations, $Ax = b$.

Recall that, in the standard version of this algorithm, we start with some guess for the solution vector x. Then, we think of the linear system $Ax = b$ as a sequence of n separate linear equations. We simply march through the list of equations, and when we reach equation i, we "solve" it for variable x_i. In other words, we replace the current value by whatever value makes the i-th equation an exact equality. It turns out that this is simply

$$x_i \leftarrow x_i + \frac{b_i - \sum_j A_{i,j} x_j}{A_{i,i}}$$

You should convince yourself that after this update, equation i really is exactly satisfied.

If we count every time we change a value x_i as a single "update", here are the results of a typical sequence of applications of the Gauss-Seidel method, for a second-difference matrix of order $n = 20$, an exact solution of $x = (1, 2, ..., 20)$, and a zero starting guess. As we repeatedly update individual solution entries, the norm of the error goes down in a regular way:

```
Updates   ||Ax-b||

  10        21
  20        10.5
  40        5.86968
  80        2.99386
 160        1.63108
 320        0.922066
 640        0.529604
1280        0.259305
2560        0.0620072
```

It might seem important that we address the variables in order, that is, we update x_0, then x_1, up to x_{n-1}, then cycle back to x_0. But it turns out that, instead, we could simply choose a random component of x to update on each step. Doing this gives us the stochastic Gauss-Seidel method. We apply the exact same update formula, but now we choose the component i randomly. With this change, here is a sample computation:

```
Iterations  ||Ax-b||

  10        21
  20        21
  40        4.91093
  80        5.86968
 160        3.19846
 320        2.28536
 640        1.08276
1280        0.790858
2560        0.31795
```

In comparison, the error reduction seems slower, and less regular. The crucial point to note, however, is that apparently, we are free to randomly choose the next component to update. The procedure will still converge.

Now we are ready to discuss the stochastic version of the gradient descent method.

11 Stochastic gradient descent

Mathematically, it might seem that we have squeezed all the juice out of this minimization problem. But once again, things that happen in real life raise new issues that we may have to address. One issue is that the data set may be extremely large, or it may be impractical to compute and store the entire gradient vector for each iteration.

What if we only had one component of the gradient vector? Even this tiny bit of information suggests how to modify our current iterate so as to decrease the function value. Essentially, instead of relying on all n items in our data, we are picking one item. If we are doing a linear fit, then we will adjust our linear fit to better approximate that single item.

Let's forget about all the improvements we made to gradient descent to deal with the size of the learning rate, and terminating the loop early if possible. For simplicity, let's go back to the basic two-line version of gradient descent:

```
for it in range(iterations):
  gradient_vector = (2/set_size) * X.T.dot(X.dot(theta) - y)  # compare X'(X c-y) above
  theta = theta - learning_rate * gradient_vector
```

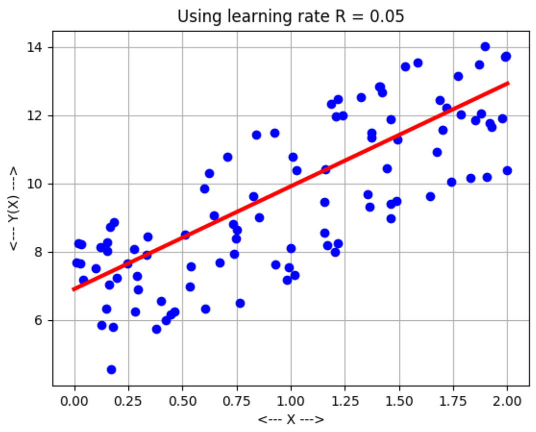
Listing 4: Gradient descent in 2 lines.

Now, if we rethink the algorithm to employ a stochastic approach, this becomes something like:

```
  for k in range ( kits ):
    i = np.random.randint ( n )
    xi = X[i:i+1]  # The i-th row of matrix X
    yi = y[i:i+1]                                    # The i-th entry of vector y
    gradient_vector = 2 * xi.T.dot(xi.dot(theta) - yi)  # no division by n!
    theta = theta - r * gradient_vector
```

Listing 5: Stochastic gradient descent in 5 lines.

Here is the plot of our linear fit after using just 1000 iterations of stochastic gradient descent:



Linear fit after 1000 stochastic gradient descent steps.

You should see the analogy between the stochastic versions of Gauss-Seidel and gradient descent. Of course, the analogy is not perfect: in stochastic Gauss-Seidel, we actually solved the i-th equation exactly, but in stochastic gradient descent, we merely "adjust" our model to better fit the i-th data instance. However, in both cases, we are trying to solve a complex problem, but we do so by repeatedly making adjustments to each single component separately. While we could treat the components in order, in machine learning it is customary to choose them randomly.

Of course, in applications, the stochastic gradient descent method faces the same problems that gradient descent did. In particular: if the learning rate is set badly, the iteration is likely to diverge. As the iteration begins to converge, the learning rate may need to be reduced to avoid jumping past the minimizer; once the stepsize becomes too small to make a difference, the iteration should be terminated.

Thus, practical implementations of stochastic gradient descent involve much more than 5 simple lines! In particular, Scikit-learn includes `SGDRegressor()` and `SGDClassifier()` which use this method for regression (including the linear fits to data we have seen today) and classification (a topic we will come to shortly).

In the special field of neural networks, (a topic we will get to much later), stochastic gradient descent is a vital tool for improving the performance of a network.