Gradient Descent
Mathematical Programming with Python

https://people.sc.fsu.edu/∼jburkardt/classes/mpp_2023/gradient_descent/gradient_descent.pdf

Gradient descent goes downhill to reduce error.

optimization; least squares error; the basis gradient descent method; stochastic gradient descent. Gradient descent in higher dimensions; the learning rate; stepsize control.

Gradient Descent

- *Optimization of a function seeks its extreme values (minimums or maximums);*
- *The least squares approach minimizes the norm of a vector of function values;*
- *Gradient descent updates the function arguments "down" the gradient of the least squares norm ;*
- *A stochastic version of gradient descent greatly decreases the storage and complexity;*
- *Gradient descent works in higher dimensions as well.*
- *Stepsize control is a significant problem.*
- *Very small steps converge slowly; steps too big may cause the function to increase.*
- *The user can vary the iteration limit, learning rate, and stepsize control.*

1 Optimization

The mathematical and computational field of the optimization studies scalar-valued functions $h(x)$, whose argument x might be either a scalar or a vector. Optimization tries to determine the existence and location of the arguments x which result in maximum or minimum values of $h(x)$. Often, the function is a measurement of reward, cost, or error, quantities which it is natural to try to maximize or minimize.

If the function is a polynomial, for which we have a formula, we might start by computing its derivative $h'(x)$, but then, unless we have a very simple polynomial, we are faced with the not much simpler task of finding the roots of $h'(x)$. And if our function has some more complicated structure, we will want a general algorithmic approach to optimization.

2 The method of least squares

An interesting class of optimization problems actually starts by considering the solution of equations involving a vector-valued function. Such equations might come from a linear algebra problem $Ax = b$, or have the

general nonlinear form $f(x) = b$. In either case, it is customary to rewrite these equations as an expression for the "residual function", $r(x)$, so that the right hand side is zero if x is an exact solution:

$$r(x) = Ax - b \quad \text{the linear case}$$
$$r(x) = f(x) - b \quad \text{the nonlinear case}$$

Now both the argument x and function $f(x)$ represent vector quantities.

In order to perform optimization, we need a scalar function. The natural choice is to define a new function $s(x)$, which is the sum of the squares of the components of $r(x)$, which is actually the square of the Euclidean norm of $r(x)$:

$$s(x) = \frac{1}{2}(\sum_i(\sum_j A_{i,j}x_j) - b_i)^2 = \frac{1}{2}||Ax - b||_2^2 \quad \text{the linear case}$$
$$s(x) = \frac{1}{2}(\sum f_i(x) - b_i)^2 \qquad = \frac{1}{2}||f(x) - b||_2^2 \quad \text{the nonlinear case}$$

where we include the $\frac{1}{2}$ multiplier to make our next calculation convenient.

We now plan to determine our solution x to the original problem by trying to minimize the value of $s(x)$. For any x, it must be the case that $0 \leq s(x)$. Let us suppose that x^* is a *local minimizer* of $s'(x^*)$, so that $s'(x^*) = 0$. It is natural to hope that this x^* is a solution to our original problem, and that it is unique. Neither of these hopes is quite certain!

The first question is one of existence. Actually, we can assert that there is always at least one argument x^* for which $s'(x^*) = 0$, so we have existence ... of a solution to the least squares minimization. But if x^* minimizes $s(x)$, that doesn't mean that $s(x^*) = 0$. In particular, the original set of equations might have no solution, in which case it is impossible for the norm of the error to be zero. So when we apply least squares to our problem, we are going to get a result, but it is up to us to figure out what this result is. Usually, if our problem does not have a solution, we will still be very happy to have a result that minimizes the error.

The question of uniqueness is also a bit complicated. For the linear case, as long as the matrix A has full column rank, x^* will be the only local minimizer of $s(x)$ (and hence it is called the *global* minimizer, or simply the minimizer). For the nonlinear case, there may be several, or even infinitely many local minimizers. Thus, the least squares approach doesn't by itself answer the question of uniqueness. Having found x^* through a least squares approach, there may actually be other values x_1^*, x_2^*... which also are local minimizers, and it may be the case that $s(x_1^*) < s(x^*)$, so that we could find better minimizers of our error if only we knew to keep on looking!

3 Going downhill with gradient descent

Now we know that our scalar-valued function $s(x)$ to be minimized might come from a vector-valued function to which we have applied the least squares approach. Thus, our task becomes

Find x^* which minimizes the scalar function $s(x)$.

As an example, let us start by considering the function named *quartic()* whose formula is:

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

Suppose $f(x)$ measures a cost and we seek a value x which minimizes it. Suppose we suspect that this minimizing value x is somewhere between -2 and +2. How do we proceed?

Because of the special fact that *quartic()* is a polynomial, we might suppose that there are methods available to determine the local maximum, or to report the location of zeros of the derivative $f'(x)$. However, because we will soon see more difficult and complicated problems, let's try to solve this on our own.

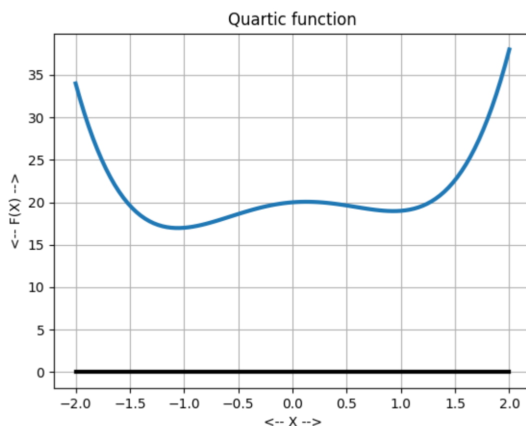
We might begin by simply picking a starting point x_0 somewhere in $[-2, +2]$, that we suspect is close to the minimizer. Most likely, our guess is not perfect, but now we have an interesting piece of information that suggests what we could do next. Our new clue is the value $f'(x_0)$, the derivative at x_0. From its value, we know:

- if $0 < f'(x_0)$, then $f(x)$ decreases to the left: $\swarrow$;
- if $f'(x_0) < 0$, 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 $f(1.5) = 22.625$, and $f'(1.5) = 16.0$. Figuratively speaking, we are standing on a mountain at a point where it slopes up to the right, and down to the left. So our next guess should be somewhere to the left of our current position.



The quartic() function

The derivative tells us which direction to move, but not how far. It is natural to base our stepsize as some multiple α of the derivative. Typically, α is chosen as a small value, like 0.005. If the method moves too slowly, we can try increasing α, but if it misbehaves (typically, by producing values that are too large, or that oscillate) we can try decreasing α.
So, starting at x_0, our proposed method will compute the next iterate x_1 by

$$x_1 = x_0 - \alpha s'(x_0)$$

and then keep doing this until we are satisfied. We might simply run the iteration for `it_max` iterations. A more sophisticated method would watch the size of $s'(x)$, which should go to zero as we get very close to a minimizer. For now, we will stick with just running for a fixed number of iterations.

4 Gradient descent pseudocode for 1D case

Pseudocode for a gradient descent method might look like this,

```
gradient_descent_1d ( f, df, x, itmax, alpha )

  for it from 1 to it_max
    x = x − alpha ∗ df ( x ) ∗ f ( x )
  end

  return x
```

Listing 1: Pseudocode for gradient descent in 1D.

5 Minimizing the quartic() function

For our quartic function, we define the functions `f()` and `df()` as follows:

```
def quartic ( x ):
  value = 2.0 ∗ x∗∗4 − 4.0 ∗ x∗∗2 + x + 20.0
  return value
```

Listing 2: quartic.py defines f(x).

and

```
def quartic_df ( x ):
  value = 8.0 ∗ x∗∗3 − 8.0 ∗ x + 1.0
  return value
```

Listing 3: quartic_df.m defines f'(x)

We call *gradient_descent_1d()* with a starting guess `x0` in our chosen interval of $[-2, +2]$, and we try `it_max=40` and `alpha=0.0`.

```
  x0 = 1.5;
  itmax = 40;
  alpha = 0.005;
  x = gradient_descent_1d ( quartic, quartic_df, x0, it_max, alpha )
```

Listing 4: Calling gradient descent for the quartic function.

Our first 10 results are:

it	x	f(x)	f'(x)
0	1.5	22.625	16
1	-0.31	19.32407	3.241672
2	-0.62321149	18.124916	4.0492863
3	-0.99017635	17.010588	1.1548698
4	-1.0884014	16.979766	-0.60750508
5	-1.036825	16.974427	0.37785531
6	-1.0688943	16.971738	-0.21883987
7	-1.0503239	16.970968	0.13301854
8	-1.0616111	16.970656	-0.078751722
9	-1.0549288	16.970553	0.047401121
10	-1.0589509	16.970514	-0.028258733

It is interesting to observe in the last column of this table how the derivative keeps changing sign. That's because the iteration is near the minimizer, and keeps moving back and forth over it, with smaller and smaller steps. After the full 40 iterations that we allow, the derivative has decreased to about 10^{-10} and we can assume we have got a good approximation to a local minimizer.

6 Gradient descent for a vector argument

Now let's consider what happens if we have a scalar function $f(x)$ of multiple variables, and again, for simplicity, we will assume we have just 2 variables, and write $f(x, y)$ as our function. Now when we differentiate, our result is called the gradient vector, and in this case contains the components $[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}]$. When we evaluate the gradient vector at a particular point (x, y), it again points in the direction in which the function increases most strongly. Because we are interested in minimizing the function value, we will want to move in the direction of the negative gradient.

To implement the gradient descent method for this case, we just have to keep in mind that x is a vector, $f(x)$ is a scalar, and $f'(x)$ is a vector. We end up with code that looks the same as before (assuming our computing language supports vector operations in a natural way), but actually there is much more going on now when we execute out commands.

```
gradient_descent_vector_x ( f, df, x, it_max, alpha )

  for it from 1 to it_max
    x = x - alpha * df ( x ) * f ( x )
  end

  return x
```

Listing 5: Pseudocode for gradient descent with vector x.

7 Gradient descent for the hex2(x,y) function

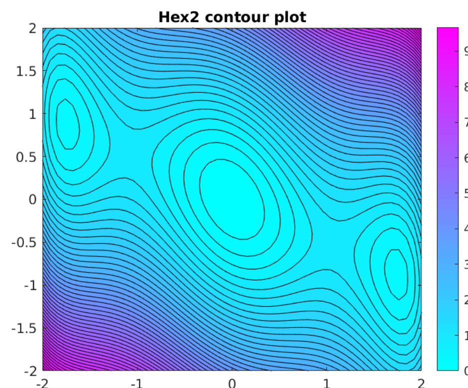
An example of a scalar function of a vector argument is named *hex2()*, and has the form

$$f(x, y) = 2x^2 - 1.05x^4 + x^6/6 + xy + y^2$$

We can ask for a minimizer of this function, which we suspect is somewhere in the range $-3 \leq x, y \leq +3$. As before, we will use derivative information, known as the *gradient*, symbolized by ∇f. For our problem, this information is $\left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$ with values:

$$\frac{\partial f}{\partial x} = 4x - 4.2x^3 + x^5 + y$$
$$\frac{\partial f}{\partial y} = x + 2y$$

A contour plot of the *hex2(x,y)* function suggests that there are three minimizers in the range $-2 \leq x, y \leq +2$.



Contour plot of the hex2() function

We define the functions *hex2(xy)* and *hex2_df(xy)*.

```
def hex2 ( xy ):
  x = xy[0]
  y = xy[1]
  value = 2.0 * x**2 - 1.05 * x**4 + x**6 / 6.0 + x * y + y**2
  return value
```

Listing 6: Definitions of f(x) for hex2().

```
def hex2_df ( xy ):
  import numpy as np
  x = xy[0]
  y = xy[1]
  value = np,array ( [ 4.0 * x - 4.2 * x**3 + x**5 + y, x + 2.0 * xy ] )
  return value
```

Listing 7: Definitions of f'(x) for hex2().

We call `gradient_descent_vector_x()` with a starting point `x0` inside our interval, and tentative values for `it_max` and `alpha`:

```
  x0 = np.array ( [ 1.0, 1.5 ] )
  it_max = 10000
  alpha = 0.15

  x = gradient_descent_vector_x ( hex2, hex2_df, x0, it_max, alpha )
```

Listing 8: Calling gradient_descent_vector_x for hex2().

It takes a long time for this problem to converge, and so we only give the first and last results:

it	[x, y]	f(x)	[dfdx,dfdy]
0	[1.0 1.5]	4.866666666666667	[2.3 4.0]
10000	[0.00634006, -0.01481108]	0.00020585590545710 92	[0.0105481 -0.02328209]

Here the results look pretty unsatisfactory. It's tempting to try a larger learning rate; however, increasing `alpha` to 0.20 causes the iteration to blow up. The learning rate is the problem, however, and we can cure it if we are willing to adjust it as we go along. One of the exercises suggests a way to get the *hex2()* function minimized faster and better by an intelligent control of the value of `alpha`.

8 Gradient descent for a vector function

Up until now, we have been interested in finding the minimizer of a scalar function $f(x)$, with x allowed to be a scalar or vector. Now we are interested in cases in which $f(x)$ represents a vector of m functions. We will assume that x is an n vector, and the values m and n might not be equal.

We express our goal as the minimization of the sum of the squares of the components of $f(x)$:

$$\text{Minimize } \frac{1}{2}||f||_2^2 = \frac{1}{2}f' * f$$

Once again, we will need to rely on derivative information, but now, our information will come to us in the form of the *Jacobian matrix*, for which a typical entry is

$$J_{i,j}(x) = \frac{\partial f_i(x)}{\partial x_j}$$

and now the gradient descent method becomes

$$x = x - \alpha \cdot J'(x) \cdot f(x)$$

We see that the algorithm has the same form as the previous versions, although the derivative information is now a matrix, and the function is a vector.

9 Minimize the wiki3() function

The Wikipedia article on gradient descent considers the following function, which we will call *wiki3()*:

$$f(x) = \begin{bmatrix} 3x_0 - \cos(x_1 x_2) - \frac{3}{2} \\ 4x_0^2 - 625x_1^2 + 2x_1 - 1 \\ \exp(-x_0 x_1) + 20x_2 + \frac{10\pi - 3}{3} \end{bmatrix}$$

The Jacobian matrix of $f(x)$ is

$$J(x) = \begin{bmatrix} 3 & \sin(x_1 x_2) x_2 & \sin(x_1 x_2) x_1 \\ 8x_0 & -1250x_1 + 2 & 0 \\ -x_1 \exp(-x_0 x_1) & -x_0 \exp(x_0 x_1) & 20 \end{bmatrix}$$

The article uses a starting point `x=[0,0,0]`, a value of `it_max=83`, and a small learning rate of `alpha=0.001`, which means that convergence will be slow. Here are the results for the first and last few steps of the iteration:

it	||x||	||f(x)||	||J(x)||
0	0	58.456136	20.322401
1	0.2095833	23.306394	20.230019
2	0.33545194	10.616628	20.224202
3	0.41113662	6.013408	20.227599
4	0.45674602	4.3216981	20.230302
5	0.48434855	3.6785445	20.234899
...	...	...	...
80	0.71486413	0.5418691	20.600946
81	0.7171265	0.53099798	20.606056
82	0.71934267	0.52064216	20.611077
83	0.7215126	0.51078342	20.616007

The 83rd x iterate is (0.496451,0.001604,-0.52356) but we are still far from a minimizer. We can run the iteration longer, or try to increase the value of `alpha`, to see how much lower we can go.

10 Stochastic gradient descent

When our function $f(x)$ involves multiple independent variables x, the standard procedure is to use the derivative information to compute an update for all components. Sometimes, for reasons of storage or speed, it can be more convenient to only update one component of the current solution. This is somewhat like a skier wanting to go downhill, whose steps can only go North/South or East/West. It is most likely that the skier can still descend, although at a slower or less efficient rate.

The method of *stochastic gradient descent* is applied to a problem with m independent variables x. It randomly chooses a component index $0 \le j < m$, and then updates only that component, as follows:

$$x_j = x_j - \alpha \cdot J[:, j]' \cdot f(x)$$

It seems natural to assume that, to get a similar reduction in function value with stochastic gradient descent, we would need to take roughly m times as many steps that update 1 component at a time, as when we update all m components of x at a time.

11 Minimize the wiki3() function with stochastic gradient descent

Using the same input as we did for the full gradient descent method, here is what we see when using stochastic gradient method on the *wiki3()* function:

it	j	||x||	||f(x)||	||J(x)||
0		0	58.456136	20.322401
1	2	0.002	58.454637	20.229928
2	3	0.20944906	23.36271	20.229928
3	1	0.20958375	23.3063	20.230019
4	1	0.20998675	23.250004	20.230291
5	1	0.2106557	23.193825	20.230743
...	...	...	...	...
80	3	0.56241181	2.1187147	20.291414
81	2	0.56241181	2.1187147	20.291414
82	2	0.56241181	2.1187147	20.291414
83	1	0.56501108	2.0696339	20.296107

The encouraging thing is that the function norm is decreasing. However, as we suspected, the rate of decrease is noticeably slower than before. In fact, for small problems like this, stochastic gradient descent doesn't really offer much of an advantage. Its payoff occurs when the number of variables x is very large, and the Jacobian is a constant, which can occur when we try to find a linear function that approximately fits a very large dataset. Such problems arise very frequently in machine learning.

12 Exercises

1. For the quartic function, there are two local mimimums and one local maximum, which occur in the interval $[-2, +2]$. Each of these represents a zero of the derivative function that we defined with *quartic_df()*. From the plot, you can roughly estimate the location of each of these values. For instance, the leftmost minimum is near -1, somewhere in the interval [-1.1, -0.9]. Use the `fsolve()` function to try to get an accurate estimate of this root with commands like:

```
from scipy.optimize import fsolve
root = fsolve ( quartic_df, [ -1.1, -0.9] )
```

   Use similar commands to find the local maximizer, and the rightmost local minimizer.

2. If you use a value of `alpha` that is too large, the iteration can sometimes rapidly diverge to infinity, so that all your subsequent results are printed as `NaN`, or "Not A Number". So we could be cautious and just keep making `alpha` smaller and smaller until we get a result. But instead of finding a single tiny value of `alpha` that works for every step, we could instead start every step with an ambitious value of `alpha`, and reduce it if necessary until our new iterate has a lower function value than the previous iterate. Do you see how this would work? The iteration starts with x0 and alpha. We compute f(x0), and we double alpha and compute x1. If f(x1) ¡ f(x0), then we finish the first step. Otherwise, we divide alpha by 2 and try again. We keep dividing alpha until we satisfy f(x1) ¡ f(x0). Thus, at the end of step 1, we might have doubled alpha (if we were lucky), but in any case we reduced f. And on every subsequent step, we try to increase alpha in this way, but always back off if it causes the function

value to rise instead of fall. Try this approach on the `quartic()` function, starting with $x0 = 2.0$ and $alpha = 0.1$. See if you can print the value of `alpha` that ends up being used on each step.

3. The function *sag(x)* has the definition:

$$f(x) = x^2 \cos(\frac{x + 3\pi}{4})$$

   A plot suggests that there is a minimizer between 0 and 10. Use the *gradient_descent_1d()* function to estimate the location of the minimizer, and the value of $f(x)$ there. Make a plot to convince yourself of your results.

4. The minimization of the *hex2(x,y)* function takes a long time and returns a not very satisfactory result. It is suggested that the value of `alpha` is too small, at least once the iteration gets near the minimizer. A possible approach is to double alpha whenever it seems safe. Here is some pseudocode with this idea:

```
x1 = x - alpha * df(x) * f(x)
x2 = x - 2.0 * alpha * df(x) * f(x)
if f(x2) < f(x1) then
  x = x2
  alpha = 2.0 * alpha
else
  x = x1
end
```

   Rerun the *hex2()* minimization, using the same starting values of `x0`, `it_max` and *alpha*, and decide whether this approach works.

5. The Rosenbrock function is defined by:

$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2$$

   A contour plot suggests that there is a minimizer in the range $0.0 \leq x, y \leq 3.0$. However, the contour plot also suggests that this function will be difficult to minimize. The lines of descent defined by the gradient vectors twist very sharply, and so only small steps can be taken reliably. Try starting at `x0=[1.50,1.25]`, `it_max=100` and `alpha=0.0025`. We want to try to find (x, y) for which $f(x, y) < 0.00001$. If your first 100 iterations don't satisfy this goal, use the output of the computation as the starting point for another 100 iterations, and repeat this process (no more than 10 times) and see if you can eventually come close enough to the minimizer.

6. Minimize the norm of *mini2()*, a vector function of vector arguments, using gradient descent, as previously applied to the *wiki3()* function. This function has the form

$$f(x, y) = \begin{bmatrix} x^2 - 10x + y^2 + 8 \\ xy^2 + x - 10y + 8 \end{bmatrix}$$

   Use the starting point $(3, -2)$. Experiment with choices for `it_max` and `alpha` until you can drive the value of $0.5||f||_2^2$ to less than 0.00001.