Gradient Descent in MATLAB

Storm Frazier

Numerical Analysis

Abstract

Gradient descent is a first-order optimization algorithm that runs iteratively to find the minimum of a (cost) function. In this algorithm we will begin with a strictly convex function of two variables. In order to find a local minimum of our function using gradient descent, we need an intial guess. Then, one takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the initial guess. The new current point will be our next guess and we continue this process iteratively until we (hopefully) reach our local minimum. Imagine standing near the top of a mountain and looking down at a lake at the base of the mountain. In order to find the shortest path down to this lake, we should walk opposite of the gradient at each step we take down the mountain.

1 About the gradient descent method

It has long been observed that if we have some multivariable function F, and this function is well defined and differentiable in an open neighborhood of some point x. Then our function F decreases the fastest if one travels in the direction opposite of the gradient. So it follows that given a convex function F, we can define a recursive sequence that will be our gradient descent method:

$$\alpha_{n+1} = \alpha_n - \gamma * \nabla F(\alpha_n)$$

Where α_n is our guess, γ is defined to be our step size(usually very small), and finally $\nabla F(\alpha_n)$ is our gradient at our guess. The algorithm boils down to taking a point and taking a small step in the direction opposite of the gradient. Next we need to define some stopping criteria. This can be done in a multitude of ways. The first method we use is to define some error tolerance ϵ, and stop when $\|\nabla F(\alpha_n)\|_2 < \epsilon$. We know the norm of the gradient is zero at our solution, but because this is a numerical approximation, we just need to find when we are very close to zero. So we define ϵ to be small and stop our algorithm when we are sufficiently close. But what happens if our function diverges? Then our norm will never get below our desired tolerance, so we must define some number of maximum allowed iterations. Now we can guarantee our algorithm won't get stuck in an infinite loop.

2 Real examples

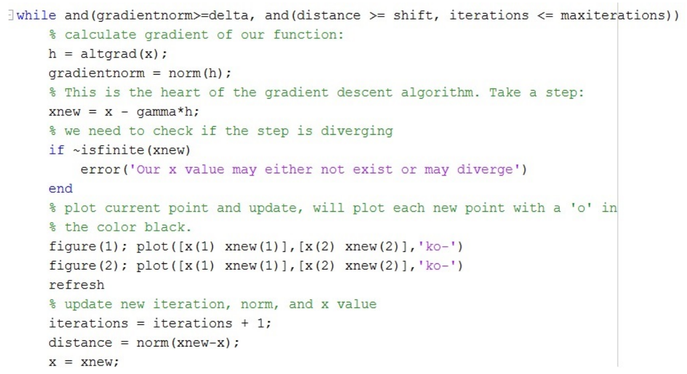
Lets begin with an easy convex multivariable function $F(x,y) = 2x^2 + xy + 3y^2$. Then $\nabla F(x,y) = (4x + y, x + 6y)$ and it is clear that the minimum of this function occurs at (0,0). So lets begin with an initial guess x_0= (5,5) and with step size $\gamma = 0.1$. Then we can calculate the first few terms using our recursive alogorithm: $x_1 = (2.5, 1.5)$: $x_2 = (1.35, 0.35)$: $x_3 = (.775, .005)$: $x_4 = (.4645, -.0755)$: $x_5 = (.2862, -.0767)$: ... $x_{20} = (0.000542, 0.000224)$: Now it is clear that we are steadily moving toward our minimum at (0,0) and after only 20 iterations we are already within $1e^{-3}$ of the true value.

Now lets see what happens if we try a non-convex function. Consider $F(x,y) = 5x^2 + xy - 10y^2$. Then $\nabla F(x,y) = (10x + y, x - 20y)$. Now it happens that we have a saddle point(both a max and a min) at (0,0), so we will see what happens when our algorithm is told to both run toward and away from the origin. Consider the initial guess x_0= (1,1) with $\gamma = 0.1$. Then using our recursive algorithm we can calulate the first few terms:

$x_1 = (-0.1, 2.9)$: $x_2 = (-0.29, 8.71)$: $x_3 = (-2.616, 78.56)$: $x_4 = (-7.856, 236)$: Well it looks like the alogirthm wants to run away from the origin(and quickly). So it is clear that the method does not work well when our function is not strictly convex. So we really need a nice function to work with, which is not usually what we are given in the real world!

3 Gradient descent in MATLAB

Now that we know the ins and outs of how gradient descent works(or doesn't work), lets see how we can create a MATLAB program to run our algorithm.



```
while and(gradientnorm>=delta, and(distance >= shift, iterations <= maxiterations))
    % calculate gradient of our function:
    h = altgrad(x);
    gradientnorm = norm(h);
    % This is the heart of the gradient descent algorithm. Take a step:
    xnew = x - gamma*h;
    % we need to check if the step is diverging
    if ~isfinite(xnew)
        error('Our x value may either not exist or may diverge')
    end
    % plot current point and update, will plot each new point with a 'o' in
    % the color black.
    figure(1); plot([x(1) xnew(1)],[x(2) xnew(2)],'ko-')
    figure(2); plot([x(1) xnew(1)],[x(2) xnew(2)],'ko-')
    refresh
    % update new iteration, norm, and x value
    iterations = iterations + 1;
    distance = norm(xnew-x);
    x = xnew;
```

We begin by creating a while loop with our stopping criteria. We check the norm of our gradient, the shift between terms, and the number of iterations. If any of these three criteria are not met, the algorithm stops and returns the last value that was calculated. The function follows our method from earlier, namely: $x_{new} = x - gamma * h$, where x is our guess, gamma is our step size, and h is the gradient at our guess. This process repeats iteratively until one of the criteria is breached. Each point is plotted on our maps in order to get a sense of how our algorithm is moving.

4 How to use the gradient descent method in MATLAB

Now we can begin with the gradient descent function in MATLAB. Begin by choosing a nice convex function. Next, the user **MUST** hardcode a gradient matrix in the altgrad.m file before using this method. In an attempt to allow anyone with MATLAB to use this method, we needed to avoid using any packages that did not come with MATLAB. So without the symbolic toolbox, the user must define a gradient matrix themselves. A very quick and easy process and after creating and saving the gradient matrix, the user can now run the alogorithm. To call our function, we simply call **gradient_descent**(*func*, *gamma*, *delta*, *maxiterations*, *shift*, *varargin*). Where:

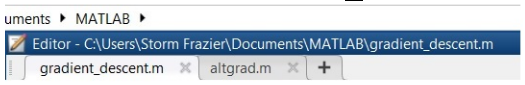
func - An anonymous convex function of two variables x_1 and x_2.
gamma - Step size.
delta - Our stopping criteria or tolerance.
maxiterations - The maximum allowed iterations (to avoid infinite loops).
shift - A shift value to break out of loops.
varargin - Our initial guess, must be in the form [x y]' (Note the ').

The function will then spit out a two variable answer that is an approximation of a local minimum, as well as two plots to help visualize the algorithm.

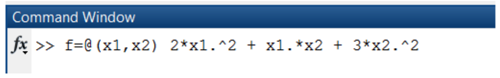
5 Step by step example

Instructions:

Step 1: Open both **gradient_descent**.m and **altgrad**.m

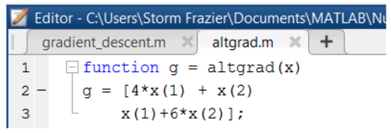

uments ▸ MATLAB ▸
Editor - C:\Users\Storm Frazier\Documents\MATLAB\gradient_descent.m
gradient_descent.m
altgrad.m

Step 2: Define a two variable anonymous convex function.



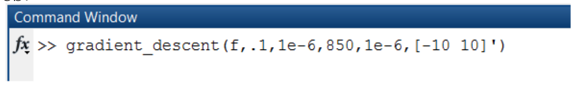
```
>> f=@(x1,x2) 2*x1.^2 + x1.*x2 + 3*x2.^2
```

Step 3: Define your gradient matrix in altgrad.m and save it.



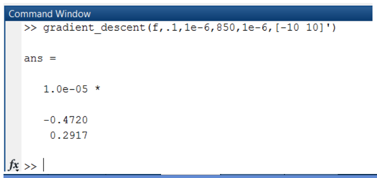
```
function g = altgrad(x)
g = [4*x(1) + x(2)
    x(1)+6*x(2)];
```

Step 4: Call the gradient descent function with the appropriate input variables.

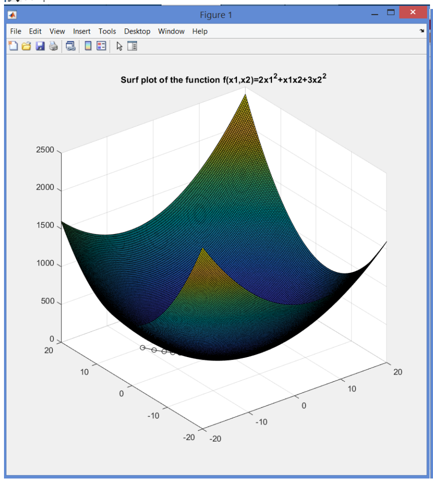


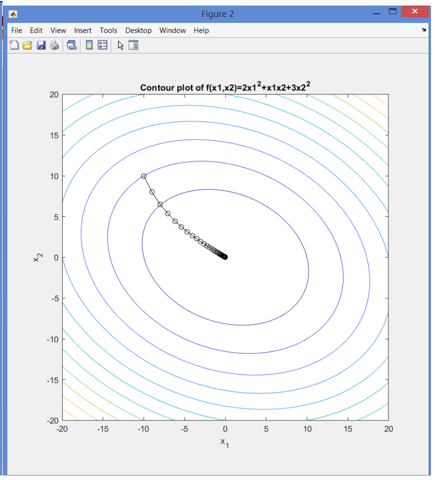
```
>> gradient_descent(f,.1,1e-6,850,1e-6,[-10 10]')
```

Step 5: Hit Enter and wait for results.


Command Window

```
>> gradient_descent(f,.1,1e-6,850,1e-6,[-10 10]')

ans =

   1.0e-05 *

   -0.4720
    0.2917

>> |
```





We get our numerical approximation to a local minimum as an output as well as two maps to help visualize the algorithm. The first map is a 3-D plot that can help visualize where the minimum is and our second plot is a contour map that helps visualize each step. Both plots include the path of the algorithm. This is the extent of the algorithm. The user can go in and adjust some of the bounds on the maps for other functions if they are unhappy with the current plots.

6 More examples

Here is a list of a few examples that are easy to plug in and run:

Example 1:
f=@(x1,x2) 5*x1.^2 + x1.*x2 + 10*x2.^2
g = [10*x(1) + x(2) x(1)+20*x(2)];
Example 2(divergent):
f=@(x1,x2)x1.^2 + x1.*x2 - 3*x2.^2
g = [2*x(1) + x(2) x(1)-6*x(2)];

7 Conclusion

Gradient descent is one of the most popular optimization methods, used in both machine learning and deep learning today. Though quite easy to understand with only two variables, things can get quite tricky when we introduce a function with thousands of variables. Hardly any real life scenario can be mapped using only two variables, so the challeges facing gradient descent today arise when we have functions that are incomprehensibly large. Gradient descent comes with two primary downfalls. The first being that the algorithm is great at finding local minimum, and terrible at finding global minimum. The next is that we are restricted to convex functions. This really limits the range of functions we can actually use gradient descent on, so being able to resolve these two issues would do wonders for the machine learning community.