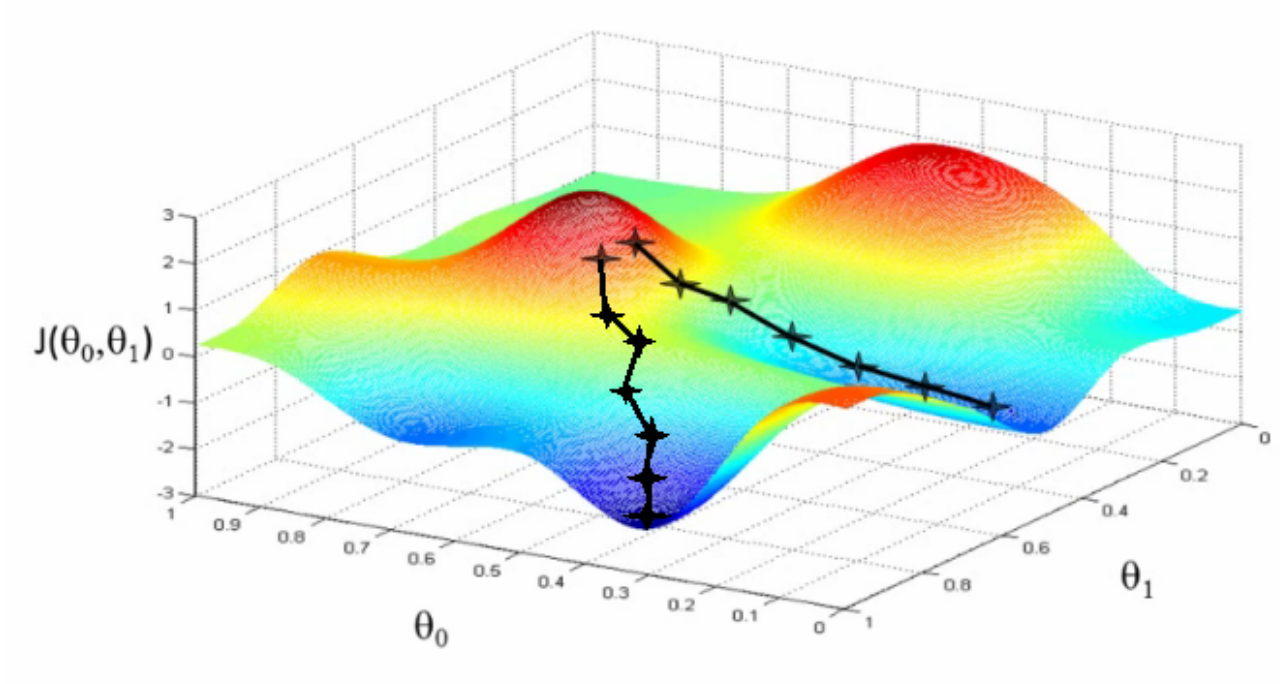
Hello, Gradient Descent

Juan Camilo Bages Prada Follow

Feb 16, 2017 · 7 min read



Gradient Descent. Image taken from http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

Hi there! This article is part of a series called ***"Hello, <algorithm>"***. In this series we will give some insights about how different AI algorithms work, and we will have fun by implementing them. Today we are gonna talk about *Gradient Descent*, a simple yet powerful optimization algorithm for finding the (local) minimum of a given function.

Getting some insight

The inspiration behind *Gradient Descent* comes directly from calculus. Basically, it states that if we've a *differentiable function*, the fastest way to decrease is by taking steps

proportional to the opposite direction of the function's *gradient* at any given point. This happens because the gradient points to the steepest direction of the function's generated surface at the current point.

In other words, think about the function's surface as a mountain that you are hiking down. You know that your goal is to reach the bottom, and you may think that the fastest way to accomplish this is by proceeding through the path that makes you descend the most. In this case, that path points to the opposite of the steepest mountain direction upwards.

With this in mind, we can repeatedly perform these steps in the appropriate direction and we should eventually converge into the (local) minimum. Following our analogy, this is the equivalent of arriving to the bottom of our mountain.

Hiking down a mountain. Image taken from https://raftrek.com/wp-content/uploads/2015/10/Hiking-down-mountain-ridge.jpg

Calculating the next step

So we've been talking about taking steps in the right direction, but how can we calculate them? Well, the answer is in the following equation:

$$\Theta^1 = \Theta^0 - \alpha \nabla J(\Theta) \text{ evaluated at } \Theta^0$$

Gradient Descent Step.

This formula needs some clarification. Let's say we are currently in a position Θ^0, and we want to get to a position Θ^1. As we said previously, every step is gonna be proportional to

the opposite direction of the function's gradient at any given point. So this definition of step for a given function $J(\Theta)$ will be equal to $-\alpha\nabla J(\Theta)$.

Why minus and not plus?

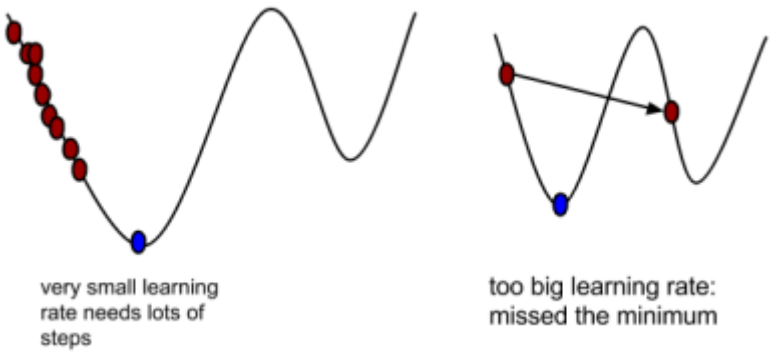
Remember that we take steps in the ***opposite*** direction of the gradient. So in order to achieve this, we subtract the step value to our current position.

Okay that sounds good, but what do you mean with α?

The symbol α is called the ***Learning Rate***. This is a value that will force us to take little steps so we don't overshoot the (local) minimum. A bad choice for α would trap us into one of the following possibilities:

- If α is too small, our learning algorithm is gonna take too much time to converge.
- If α is too large, our learning algorithm might overshoot the bottom, and even diverge because of an infinite loop.

Take a look at the following examples to see what happens when we make a bad choice for the ***Learning Rate*** α:



Bad choices for Learning Rate α. Image taken from https://storage.googleapis.com/supplemental_media/udacityu/315142919/Gradient%20Descent.pdf

Calculating the gradient

The gradient of a function $J(\Theta)$ (denoted by $\nabla J(\Theta)$) is a vector of *partial derivatives* with respect to each dimension or parameter Θ_i. Notational details are given in the equation below:

$$\nabla J(\Theta) = \left\langle \frac{\partial J}{\partial \Theta_0}, \frac{\partial J}{\partial \Theta_1}, \ldots, \frac{\partial J}{\partial \Theta_n} \right\rangle$$

$$\backslash \partial\Theta_1 \quad \partial\Theta_2 \quad \partial\Theta_n /$$

A little example

To make this definition of gradient clearer, let's calculate the gradient of the following function:

$$J(\Theta_1, \Theta_2, \Theta_3) = 2\Theta_1 + 10\Theta_1\Theta_3 - 8\Theta_2$$

As we can see, this function contains three parameters or dimensions. Thus the appropriate way to proceed is by calculating the partial derivative with respect to each param:

$$\frac{\delta J}{\delta \Theta_1} = 2 + 10\Theta_3$$

$$\frac{\delta J}{\delta \Theta_2} = -8$$

$$\frac{\delta J}{\delta \Theta_3} = 10\Theta_1$$

Now we can group those values and that will give us the function's gradient:

$$\nabla J(\Theta_1, \Theta_2, \Theta_3) = \langle 2 + 10\Theta_3, -8, 10\Theta_1 \rangle$$

And that's it! With this vector, we can get the steepest direction at any given point simply by replacing each parameter with its corresponding value:

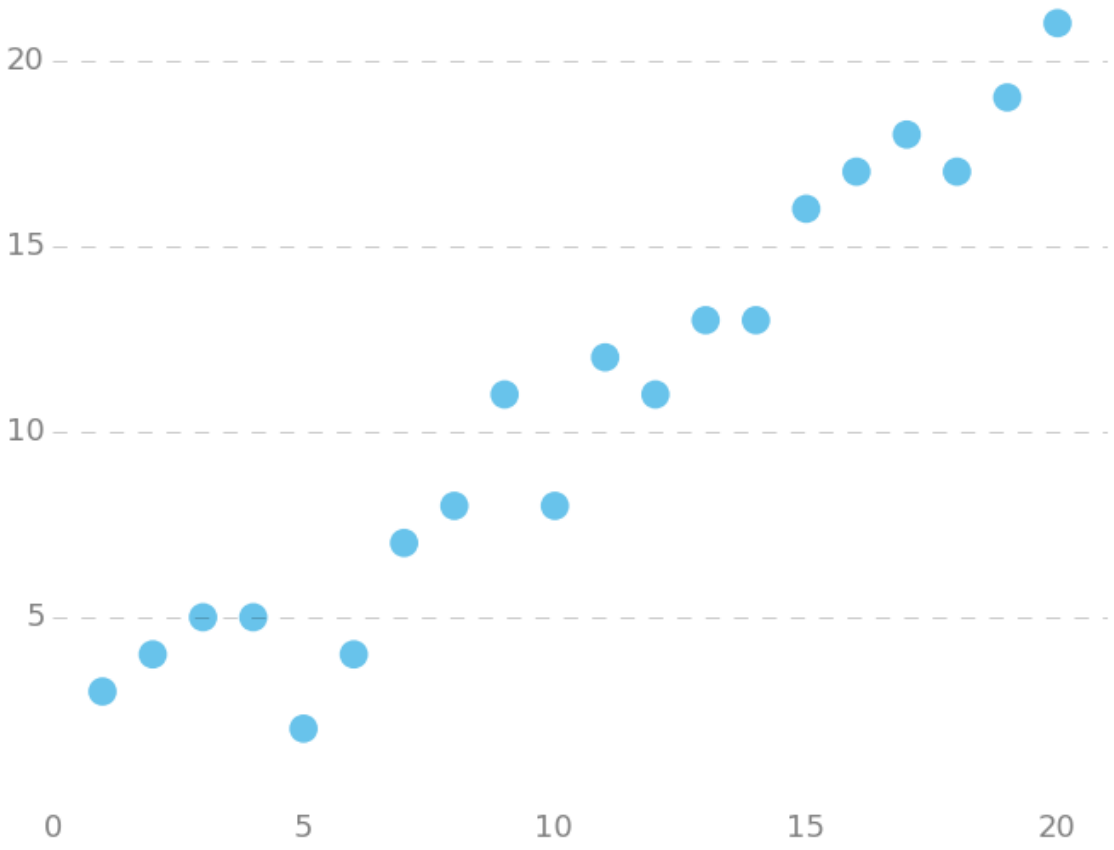
$$\nabla J(12, 9, -2) = \langle -18, -8, 120 \rangle$$

Hacking Time

And now, for the grand finale, we will go through a full example and we will code our own algorithm for gradient descent.

Defining the example

In this section, we will apply *linear regression* in order to find the correct function approximation for a given set of points in a plane. The set of points we are trying to predict looks as follows:



As it's common, the choice for $J(\Theta)$ will be the *least-squares cost function* for measuring the error of an approximation:

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)})^2$$

In the equation above:

- *m* is the amount of points in the set.
- ½ is a convenient constant that will cancel out when we take the gradient of $J(\Theta)$. This makes maths nicer and doesn't affect the result.
- *y* is the real value of the y-coordinate for the ith point.
- *h* is our function approximation. It will give us the predicted y-coordinate for the ith-point using parameters Θ and input *x*.

$$h_\Theta(x^{(i)}) = \Theta_0 + \Theta_1 x_1^{(i)}$$

Finally, before beginning to code let's calculate the gradient vector of our function. You can see that as we've got two parameters for Θ, we will need to calculate two partial derivatives.

$$\nabla J(\Theta) = \left\langle \frac{\delta J}{\delta \Theta_0}, \frac{\delta J}{\delta \Theta_1} \right\rangle$$

$$\frac{\delta J}{\delta \Theta_0} = \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)})$$

$$\frac{\delta J}{\delta \Theta_1} = \frac{1}{m} \sum_{i=1}^{m} (h_\Theta(x^{(i)}) - y^{(i)}) x_1^{(i)}$$

Okay, time to proceed. It's important to mention that our implementation will be in a vectorized form. This means that we will transform all the formulas mentioned above into matrices operations. The advantages of this implementation are that code will be more concise, and with this our computer can take advantage of advanced underlying matrix algorithms.

To work with the vectorized form, we need to add a dummy variable $x0$ to each point with a value equal to 1. The reason for this is that when we perform matrix multiplication, the intercept parameter $\Theta 0$ will be multiplied with that 1 and it will maintain its value as the defined equations establishes.

$$(x_1^{(i)}, y^{(i)}) \rightarrow (x_0^{(i)}, x_1^{(i)}, y^{(i)}) \text{ with } x_0^{(i)} = 1 \ \forall i$$

Below you can see the vectorized form of the error function $J(\Theta)$ and its gradient $\nabla J(\Theta)$:

$$J(\Theta) = \frac{1}{2m}(X\Theta - \overrightarrow{y})^T(X\Theta - \overrightarrow{y})$$

$$\nabla J(\Theta) = \frac{1}{m}X^T(X\Theta - \overrightarrow{y})$$

Coding time

With every function defined, we can proceed to code our algorithm. The first thing we should do is to declare the points dataset and the Learning Rate α.

```
import numpy as np

# Size of the points dataset.
m = 20

# Points x-coordinate and dummy value (x0, x1).
X0 = np.ones((m, 1))
X1 = np.arange(1, m+1).reshape(m, 1)
X = np.hstack((X0, X1))

# Points y-coordinate
y = np.array([
        3, 4, 5, 5, 2, 4, 7, 8, 11, 8, 12,
        11, 13, 13, 16, 17, 18, 17, 19, 21
]).reshape(m, 1)

# The Learning Rate alpha.
alpha = 0.01
```

hello_gradient_descent_dataset.py hosted with ❤ by GitHub view raw

Now we can proceed by defining the error function $J(\Theta)$ and its gradient $\nabla J(\Theta)$. Remember everything will be defined in a vectorized way.

```
def error_function(theta, X, y):
        '''Error function J definition.'''
        diff = np.dot(X, theta) - y
        return (1./2*m) * np.dot(np.transpose(diff), diff)

def gradient_function(theta, X, y):
        '''Gradient of the function J definition.'''
        diff = np.dot(X, theta) - y
        return (1./m) * np.dot(np.transpose(X), diff)
```

hello_gradient_descent_functions.py hosted with ❤ by GitHub view raw

This is the heart of our code. Here we will perform steps that update Θ until we reach the (local) minimum. That is, when all the values of the gradient vector are less than or equal to some specified threshold ($1/e^5$ in this case).

```
def gradient_descent(X, y, alpha):
        '''Perform gradient descent.'''
        theta = np.array([1, 1]).reshape(2, 1)
        gradient = gradient_function(theta, X, y)
        while not np.all(np.absolute(gradient) <= 1e-5):
                theta = theta - alpha * gradient
                gradient = gradient_function(theta, X, y)
        return theta
```

hello_gradient_descent_algorithm.py hosted with ❤ by GitHub view raw

And we're done! You can see the complete code in the snippet below:

```
import numpy as np

# Size of the points dataset.
m = 20

# Points x-coordinate and dummy value (x0, x1).
X0 = np.ones((m, 1))
```

```
X0 = np.ones((m, 1))
X1 = np.arange(1, m+1).reshape(m, 1)
X = np.hstack((X0, X1))

# Points y-coordinate
y = np.array([
        3, 4, 5, 5, 2, 4, 7, 8, 11, 8, 12,
        11, 13, 13, 16, 17, 18, 17, 19, 21
]).reshape(m, 1)

# The Learning Rate alpha.
alpha = 0.01

def error_function(theta, X, y):
        '''Error function J definition.'''
        diff = np.dot(X, theta) - y
        return (1./2*m) * np.dot(np.transpose(diff), diff)

def gradient_function(theta, X, y):
        '''Gradient of the function J definition.'''
        diff = np.dot(X, theta) - y
        return (1./m) * np.dot(np.transpose(X), diff)

def gradient_descent(X, y, alpha):
        '''Perform gradient descent.'''
        theta = np.array([1, 1]).reshape(2, 1)
        gradient = gradient_function(theta, X, y)
        while not np.all(np.absolute(gradient) <= 1e-5):
                theta = theta - alpha * gradient
                gradient = gradient_function(theta, X, y)
        return theta

optimal = gradient_descent(X, y, alpha)
print 'optimal:', optimal
print 'error function:', error_function(optimal, X, y)[0,0]
```

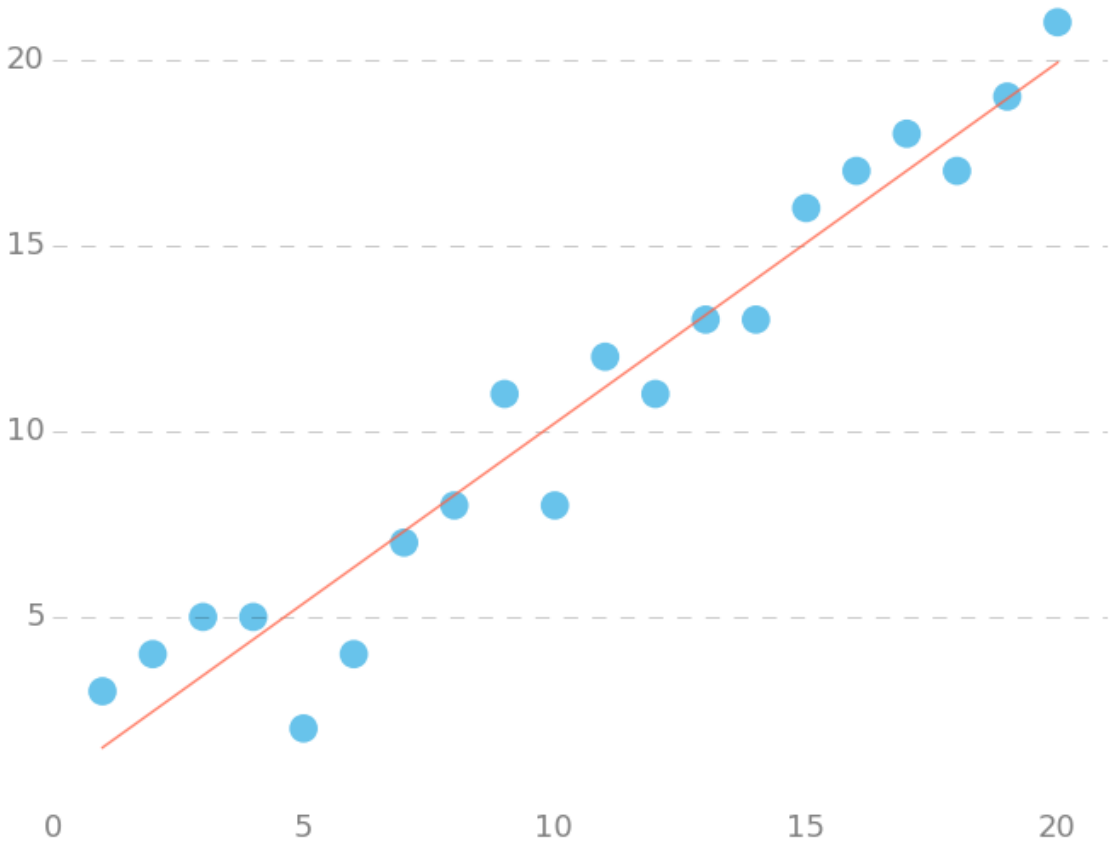
hello_gradient_descent_complete.py hosted with ❤ by GitHub view raw

Now we can run our algorithm and it will give us the optimal values for Θ that minimize the error. Below you can see the answers I obtained after running it on my computer:

$$\Theta = \langle 0.51583286, 0.96992163 \rangle$$

$$J(\Theta) = 405.984962493$$

This is the scatter plot we showed before with the line corresponding to the optimal Θ:



Well, we've finished our code and our article. I hope that you'd learned one thing or two about Gradient Descent, and more importantly, that you are now really excited about learning by taking a look at the further reading list.

Further reading

- Gradient Descent lecture notes from UD262 Udacity Georgia Tech ML Course. I was inspired a lot by this article, and based mine on it.

- Supervised Learning lecture notes from CS229 Stanford University ML Course. This article helped me a lot to understand Linear Regression.

- An overview of gradient descent optimization algorithms. This paper provides you with a lot of information about implementing production-ready Gradient Descent algorithms.

Thanks to Esteban Vargas.

Machine Learning Artificial Intelligence Gradient Descent