# What is a Cost Function? — Gradient Descent — Examples with Python

Robert R.F. DeFilippi Follow

May 27, 2018 · 8 min read

First off, you might have seen cost functions referred to by different names: loss function, or error function, or scoring function.

Any of those names will do, and in this article, we'll stick to cost function.

It is a function we can use to evaluate how well our algorithm maps the target estimate, or how well our algorithm performs optimization problems.

Consider linear regression, where we choose mean squared error (MSE) as our cost function. Our goal is to find a way to minimize the MSE.

Or consider a maximum log-likelihood function. Our goal is also to maximize this function.

Our final goal, however, is to use a cost function so we can learn something from our data.

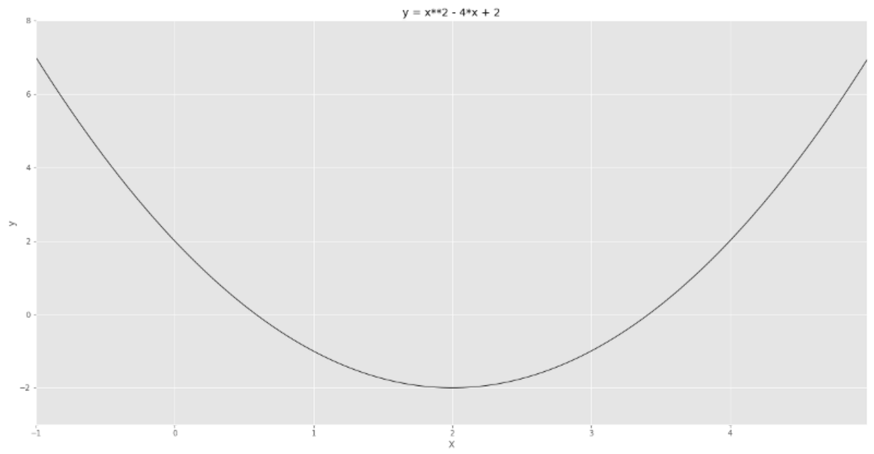
**Cost Functions and Gradient Descent**

Below, we're going to be implementing gradient descent to create a learning process with feedback. Each time—each step really—we receive some new information, we're going to make some updates to our estimated parameter which move towards an optimal combination of parameters. We get these estimates using our cost function from before.

Hence, our algorithm is learning through each step because it now knows something it did not in the previous step.

Let's take this equation below. We want to find the minimum of this function which is quite easy to do. Simply take the first order equation wrt `x`, set it to zero, and compute the value. In fact, our cost function here is simply our first order equation. Nothing too special but we're going to be building off this or the rest of the article.

```
# Finding the minimum of the function


y = x^2 - 4x + 2
dy/dx = 0 = 2*x - 4 # This is our cost function
x = 2
```



Our First Function

How would we find the solution using gradient descent?

Let’s break this down mathematically, as we’re going to be estimating a parameter `θ` which we will substitute for `x`. `θ` is the value we’re going to update after every step and will tell us what the current value of `x` is through minimization process. As `θ` converges to the minimum using our cost function.

However, we don’t always know were to start `θ` on our cost function so we take a guess. It starts at this guessed point somewhere along the cost function, and descends towards the actual value.

That is the descent, in gradient descent.

We are also going to introduce a variable called `α` which is out learning rate.

The learning rate tells our cost function how fast to move toward its goal of minimization, and control steps size taken by each iteration. At every step of the descent, `θi` is updated based on the values provided in the cost function. If `α` is too big, the model may miss the minimum. If it too small, could never get to the minimum.

This is important, as tweaking `α` is just part of applying gradient descent to your problems. It might now work with the first `α` you chose, and that's ok. Just start tweaking it, and when you see the values starting to converge you know you're on the right path.

$$y = x^2 - 4x + 2$$
$$\frac{dy}{dx} = 2x - 4$$
$$\alpha = learning\ rate$$
$$\theta = parameter\ to\ estimate$$
$$\frac{dy}{d\theta} = 2\theta - 4$$
$$\theta_i := \theta_i - \alpha \frac{dy}{d\theta_i}$$

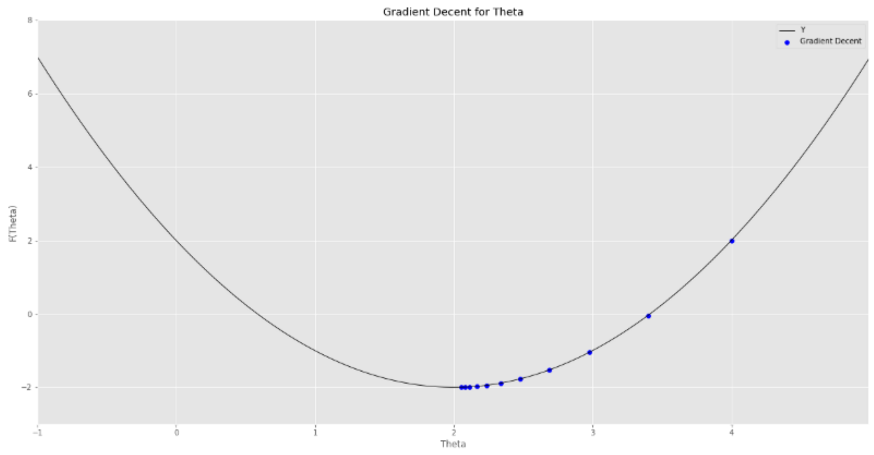
Our Equations

Let’s see how we would code this.

```
def func_y(x):
    y = x**2 - 4*x + 2
    return y


def gradient_descent(previous_x, learning_rate, epoch):

    # To fill with values
    x_gd = []
    y_gd = []

    x_gd.append(previous_x)
    y_gd.append(func_y(previous_x))


    # begin the loops to update x and y with out cost
function
    for i in range(epoch):
        current_x = previous_x - learning_rate *
(2*previous_x - 4)
        x_gd.append(current_x)
        y_gd.append(func_y(current_x))


        # update previous_x
        previous_x = current_x


return x_gd, y_gd


# Initialize x0 and learning rate
x0 = 4 # Our first 'guess' at what theta could be
learning_rate = 0.15 # Alpha
epoch = 10 # Number of tries
```

When we graph our results we can see our initial guess of `4` was not correct.

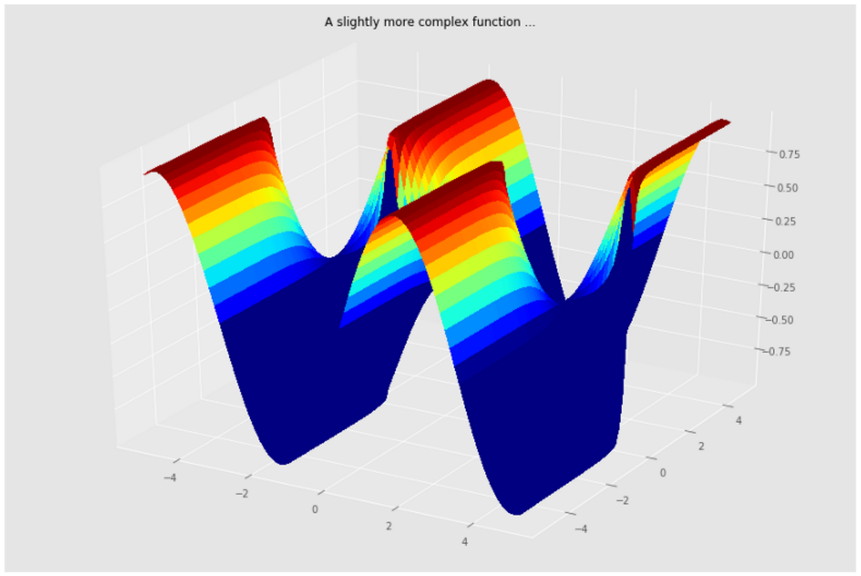
However as we updated our results, our estimate of `θ` became closer and closer to the correct value of `2`. We only iterated 10 times and fell just a little short of the correct value. With more iterations, we would have come much closer.



Simple Gradient Descent

The use of gradient descent here seems trivial, as our function is well behaved. However with more complex functions—such as the one shown below—finding the minimum would be difficult which is why we use this method.

We're not going to go over the more advanced applications of gradient descent in this article, but you should be aware of how to start thinking about this complex problems.

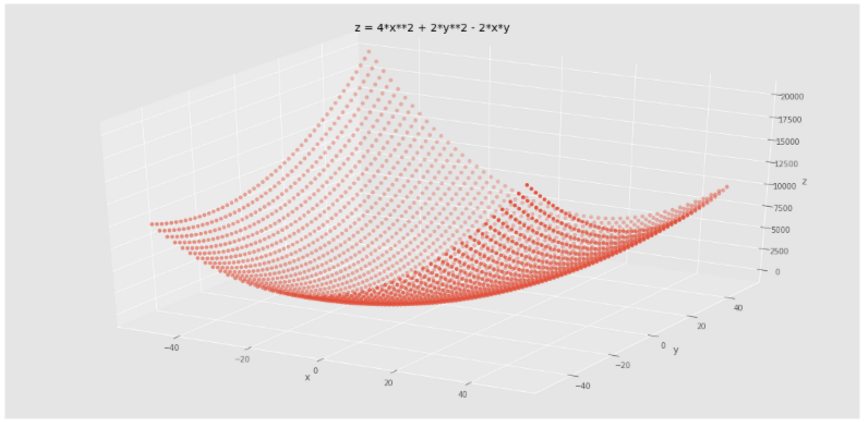


Complex Function

## Estimating Two Parameters

What if we wanted to do the same process as above except we wanted to find two parameters instead of one.

Let's take the function shown below as an example and see if we can find the minimum using gradient descent.



We would go through the same process as before by creating a cost function for each parameter we're estimating—here it is `x` and `y` — set our value for `α`, and run our gradient descent algorithm. However this time, `θ0` and `θ1` will be updated simultaneously as the gradient descends rather than a single value of `θ`.

$$z = 4x^2 + 2y^2 - 2xy$$

$$\frac{dz}{dx} = 8x - 2y$$

$$\frac{dz}{dy} = 4y - 2x$$

$$\theta_i := \theta_i - \alpha \frac{dz}{d\theta_i}$$

Our next set of equations

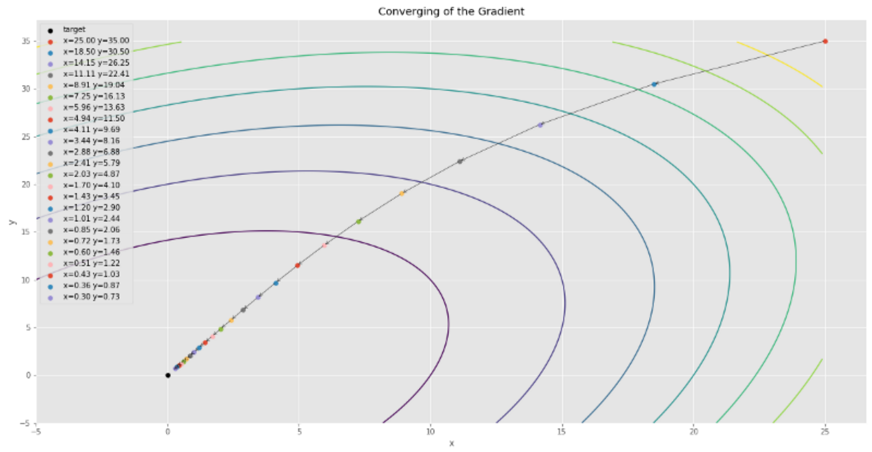
We know the true minimum of the function is `(0, 0)` so our results will be easy to verify.

```
def dx (x, y):
    return 8*x - 2*y
def dy (x, y):
    return 4*y - 2*x


def gradient_descent_2():
```

```
# Create gradient arrays
    grad_x = []
    grad_y = []
    grad_z = []


# Our initinal guess
    theta_0  = 25
    theta_1  = 35


alpha = .05
    epoch = 10000


grad_x.append(theta_0)
    grad_y.append(theta_1)
    grad_z.append(f(theta_0, theta_1))


# Run the gradient
    for i in range(epoch):
        current_theta_0 = theta_0 - alpha * dx(theta_0, 
theta_1)
        current_theta_1 = theta_1 - alpha * dy(theta_0, 
theta_1)
        grad_x.append(current_theta_0)
        grad_y.append(current_theta_1)
        grad_z.append(f(current_theta_0, current_theta_1))


# Update
        theta_0 = current_theta_0
        theta_1 = current_theta_1

    # Return last values
    return theta_0, theta_1


print gradient_descent_2()


# Results for theta_0 and theta_1
# (5e-324, 1e-323)
```

Our results are essentially `(0, 0)` so looks like our algorithm worked. Perfect.

And, the same as last time let's plot our results to see how our gradient descent performed. And, it's right on the mark. We can see our initial guess of `(25, 35)` was nowhere close, but as we went through each step we became closer and closer to the correct value.



Gradient Descends to (0, 0)

## Gradient Descent and Linear Regression

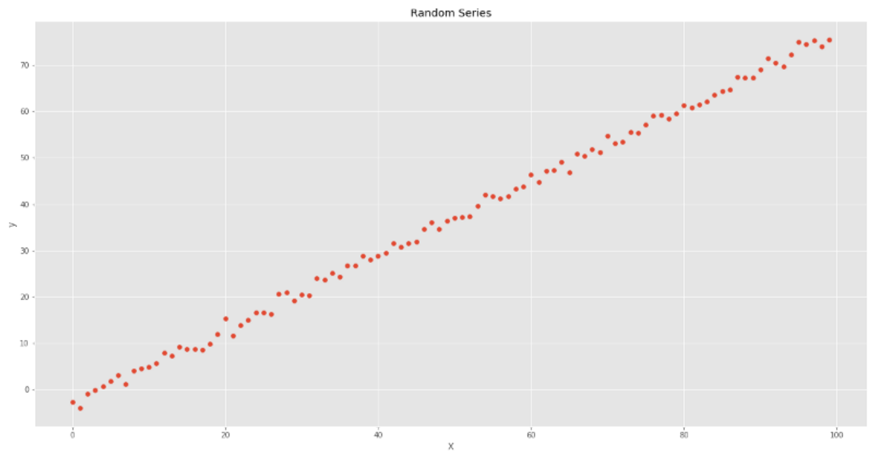
Now let's put everything we've learned together, and show how we estimate the parameters in a linear regression. Just as we built on the above, we'll be estimating two parameters however we'll be using a different cost function.

$$y = \theta_0 + \theta_1 x$$

Our simple linear equation

First we need to plot some random data, with a little noise thrown in for randomness. So we can follow the example the true values of `theta_0` is `-3` and `theta_1` is `.8`.

```
# Set the parameters
n = 100
x = np.arange(n)
y0 = [20] * n


# Our true values
thata_0 = -3
theta_1 = .8


noise = np.random.normal(size=n) + 5
y = theta_0 + theta_1 * x + noise


def plot_linear_data(x, y):
    plt.figure(figsize=(20, 10))
    plt.title("Random Series")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.scatter(x, y);
    plt.legend(['Hypothesis', 'Data'], loc='best')


plot_linear_data(x, y)
```



Random Data We're Going to Fit

Here our cost function is MSE—remember that from the start?—and by minimizing this we can be confident our parameters are correct. We do some simple substitution of our linear equation into `y_hat` for cost function, as those are the parameters we need to estimate.

Just as before our gradient will update simultaneously for both parameters as it gets closer to the true values.

$$\hat{y} = \theta_0 + \theta_1 x$$
$$MSE(\theta_0, \theta_1) = \frac{1}{2n} \sum (y - \hat{y})^2 = \frac{1}{2n} \sum (y - \theta_0 - \theta_1 x)^2$$
$$\frac{MSE(\theta_0, \theta_1)}{d\theta_0} = \frac{1}{n} \sum (\theta_0 + \theta_1 x - y)$$
$$\frac{MSE(\theta_0, \theta_1)}{d\theta_1} = \frac{1}{n} \sum (\theta_0 + \theta_1 x - y) \cdot x$$

Our two cost function for theta_0 and theta_1

We know from above we have to set a guess to start our descent, and we're going to set both values to zero and run our descent.

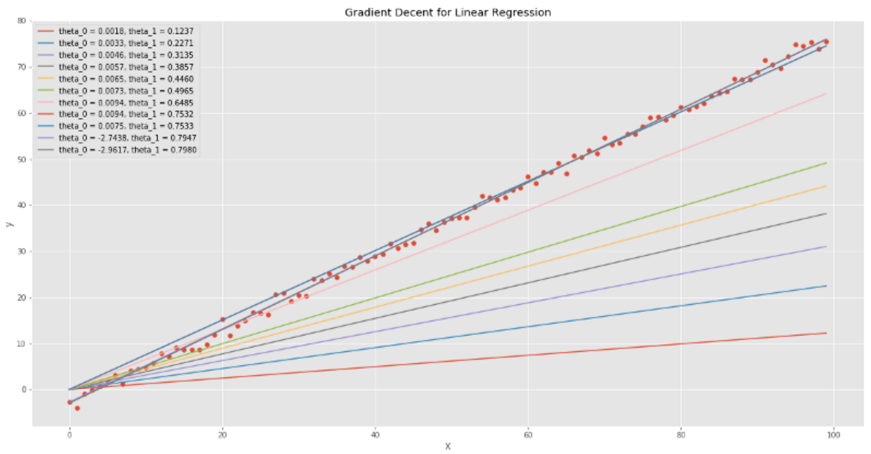
```
grad_theta_0 = []
grad_theta_1 = []


def gradient_descent_reg(x, y):

    epoch = 500000
    alpha = 0.001

    theta_0 = 0
    theta_1 = 0
    cost_0 = 0
    cost_1 = 0


for i in range (0, epoch):


y_hat = theta_0 + theta_1 *  x

        # Get cost functions
        cost_0 = np.sum(y_hat - y) / n
        cost_1 = np.sum( (y_hat - y) * x ) / n
        # Get new theta values
        temp0 = theta_0 - alpha * cost_0
        temp1 = theta_1 - alpha * cost_1

        # Update theta values
        theta_0 = temp0
        theta_1 = temp1

        grad_theta_0.append(theta_0)
        grad_theta_1.append(theta_1)

    return theta_0, theta_1


gradient_descent_regression(x, y)
# theta_0 = -2.975189443909802, and theta_1 =
0.7982329348469989
```

Looks like we came really close to the actual values of `theta_0` and `theta_1`. Just what we expected.

And finally, let’s plot different values from our `grad_theta_0` and `grad_theta_1` arrays to see how well they do estimating the true parameters of `theta_0` and `theta_1` from the initial guess, through various steps, and then the final result.

We can see as we update our values for `theta_0` and `theta_1` our linear regression function is getting closer—descending—to estimating our correct values.

As my friend would say, “This is so dope!”



Linear Regression and Gradient Descent in Action

And, that's all for now. Hopefully, you have an understanding what gradient descent actually is, how cost functions work, and how they can be applied to gradient descent.

What I did not show—for the sake the brevity—was all the alpha tuning to get the gradients to convert during descent. That will be a large part of the time and work you'll do when using this approach. So keep that in mind.

As always, I hope you learned something new. The code for this article can be found here on GitHub.

Cheers,

**Additional Reading**

https://medium.com/@lachlanmiller_52885/machine-learning-week-1-cost-function-gradient-descent-and-univariate-linear-regression-

8f5fe69815fd

https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220

https://scipython.com/blog/visualizing-the-gradient-descent-method/