Gradient Descent From Scratch

Arseny Turin Feb 3, 2020 · 6 min read

In this post, I'm going to explain what is the Gradient Descent and how to implement it from scratch in Python. To understand how it works you will need some basic math and logical thinking. Though a stronger math background would be preferable to understand derivatives, I will try to explain them as simple as possible.

Gradient Descent can be used in different machine learning algorithms, including neural networks. For this tutorial, we are going to build it for a linear regression problem, because it's easy to understand and visualize.

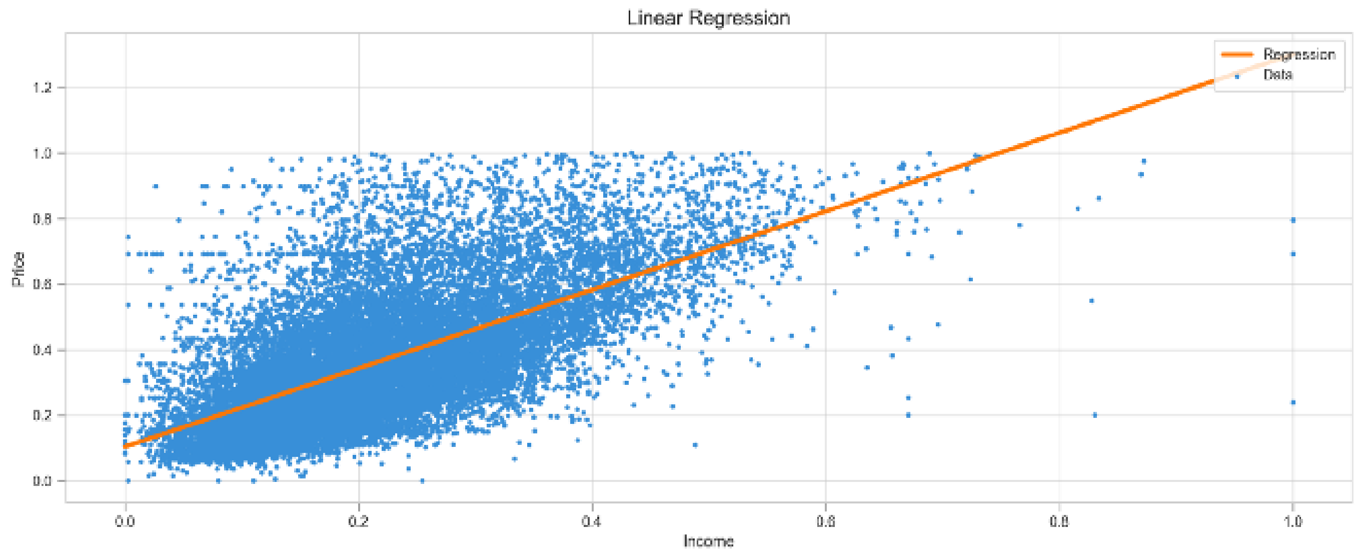
I also created GitHub repo with all explanations. Let's get started.

Linear Regression

$$\hat{y} = mx + b$$

Linear Regression

In order to fit the regression line, we tune two parameters: slope (`m`) and intercept (`b`). Once optimal parameters are found, we usually evaluate results with a mean squared error (`MSE`). We remember that smaller MSE — better. In other words, we are trying to minimize it.



Minimization of the function is the exact task of the Gradient Descent algorithm. It takes parameters and tunes them till the local minimum is reached.

Let's break down the process in steps and explain what is actually going on under the hood:

1. First, we take a function we would like to minimize, and very frequently it will be Mean Squared Errors function.

2. We identify parameters, such as `m` and `b` in the regression function and we take partial derivatives of MSE with respect to these parameters. This is the most crucial and hardest part. Each derived function can tell which way we should tune parameters and by how much.

3. We update parameters by iterating through our derived functions and gradually minimizing MSE. In this process, we use an additional parameter `learning rate` which helps us define the step we take towards updating parameters with each iteration. By setting a smaller learning rate we make sure our model wouldn't jump over a minimum point of MSE and converge nicely.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \quad \text{where} \quad \hat{y}_i = mx_i + b$$

Mean Squared Error

Derivatives

We use partial derivatives to find how each individual parameter affects MSE, so that's where word *partial* comes from. We take these derivatives with respect to `m` and `b` **separately**. Take a look at the formula below. It is almost the same as MSE, but this time we added f(m,b) to it. It essentially changes nothing, except now we can plug `m` and `b` numbers into it and calculate the result.

$$f(m, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

MSE with input parameters

With respect to `m` means we derive parameter `m` and basically, ignore what is going on with `b` , or we can say its 0 and vice versa. To take partial derivatives we are going to use a chain rule. We use it when we need to take a derivative of a function that contains another function inside.

$$[f(g(x))]' = f'(g(x)) * g(x)' \ \text{– chain rule}$$

The chain rule says that we should take a derivative of an outside function, keep an inside function untouched and then multiply everything by the derivative of the inside function. All we need to do now is to take a partial derivative with respect to **m** and **b** of the function below:

$$(y - (mx + b))^2$$

Squared Error

If you're new to this you'd be surprised that `()²` is an outside function that contains `y-(mx+b)`. Now, let's break down each step according to the chain rule:

1. Take a derivative of an outside function: `()²` becomes `2()`
2. Keep an inside function as-is: `y-(mx+b)`
3. Take a partial derivative with respect to m: `0-(x+0)` or `-x`. Lets elaborate on how we get this result: we treat anything that is not **m** as a constant. Constants are always equal to 0. The derivative of `mx` is `x`, because the derivative of `m` is 1, and any number or a variable attached to `m` stays in place, meaning `1*x`, or just `x`.

To put everything together we get the following: `2(y-(mx+b))*-x`. This can be rewritten as `-2x(y-(mx+b))`. Great! We can rewrite this again to follow proper notation:

$$\frac{\partial f}{\partial m} = \frac{1}{n}\sum_{i=1}^{n} -2x_i(y_i - (mx_i + b))$$

Partial derivative with respect to **m**

To derive with respect to `b`, we follow the same chain rule steps:

1. `()²` becomes `2()`

2. `y-(mx+b)` stays the same

3. Here derivative of `y-(mx+b)` turns into this: `(0-(0+1))` or `-1` . Here is why: we treat `y` and `mx` as constants again, so they become 0. `b` becomes 1.

Put everything together: `2(y-(mx+b))*-1` , or - `2(y-(mx+b))` . Once again the proper notation would look like this:

$$\frac{\partial f}{\partial b} = \frac{1}{n}\sum_{i=1}^{n} -2(y_i - (mx_i + b))$$

Partial derivative with respect to **b**

Phew! The hardest part behind us, now we can dive into the Python environment.

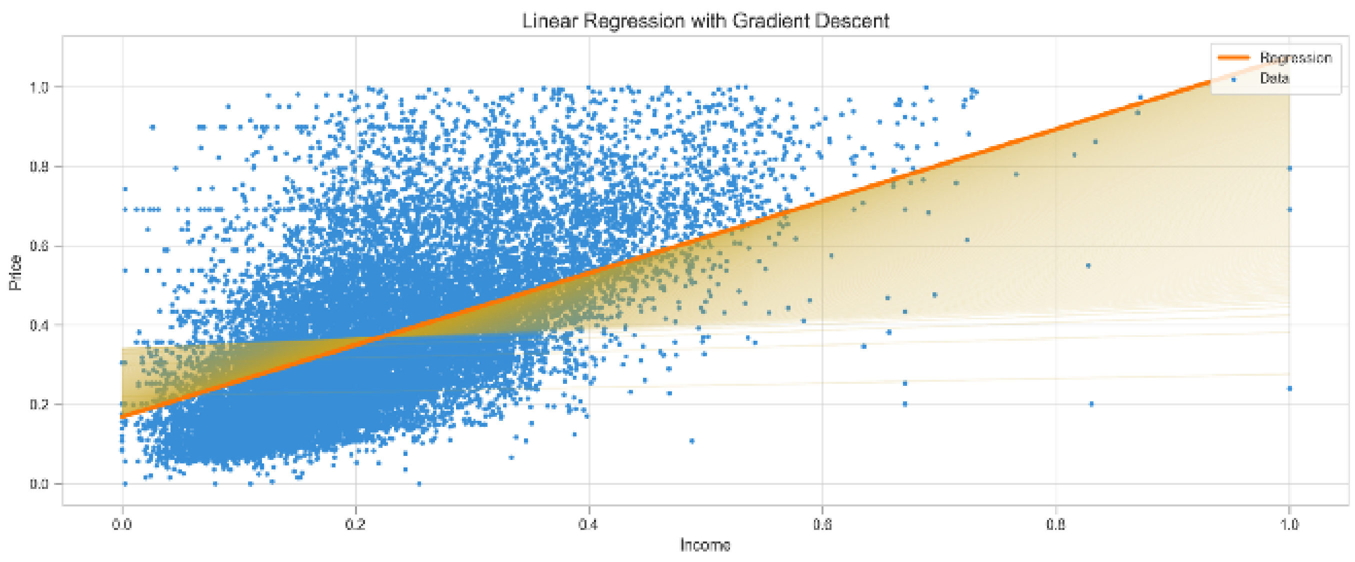
Gradient Descent

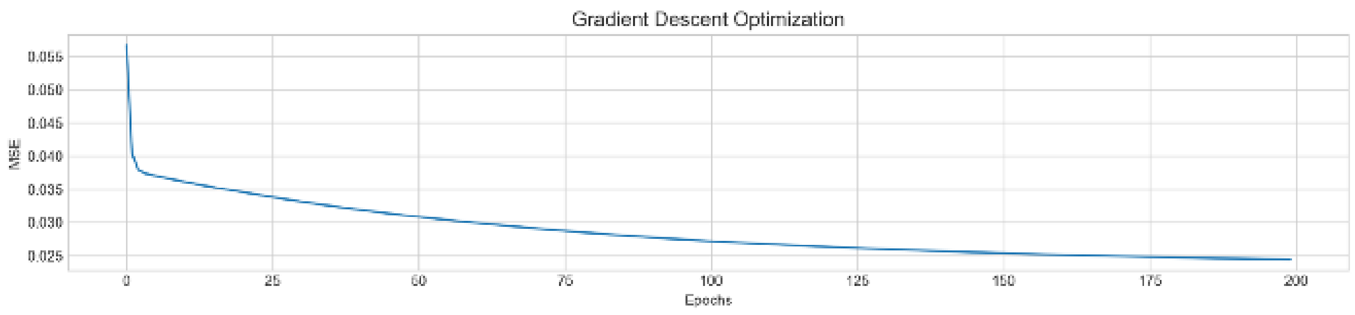
Because we have only one input feature, X must be a NumPy vector, which is a list of values. We can easily extend it to multiple features by taking derivatives of weights `m1` , `m2` .. etc, but this time we do it for simple linear regression.

Take a look at the code. We start by defining `m` and `b` with random values, then we have a for loop to iterate through derived functions and each step is controlled by a learning rate (`lr`). We use `log` and `mse` lists to track our progress.

```
import numpy as np
from sklearn.metrics import mean_squared_error

def gradient_descent(X, y, lr=0.05, epoch=10):
    
    '''
    Gradient Descent for a single feature
    '''
    
    m, b = 0.33, 0.48 # parameters
    log, mse = [], [] # lists to store learning process
    N = len(X) # number of samples

    for _ in range(epoch):

        f = y - (m*X + b)

        # Updating m and b
        m -= lr * (-2 * X.dot(f).sum() / N)
        b -= lr * (-2 * f.sum() / N)

        log.append((m, b))
        mse.append(mean_squared_error(y, (m*X + b)))

    return m, b, log, mse
```

We can track visually how the algorithm was approaching a local minimum. To the naked eye, it seems like it didn't converge completely. To solve that we can increase epochs and learning rate parameters.



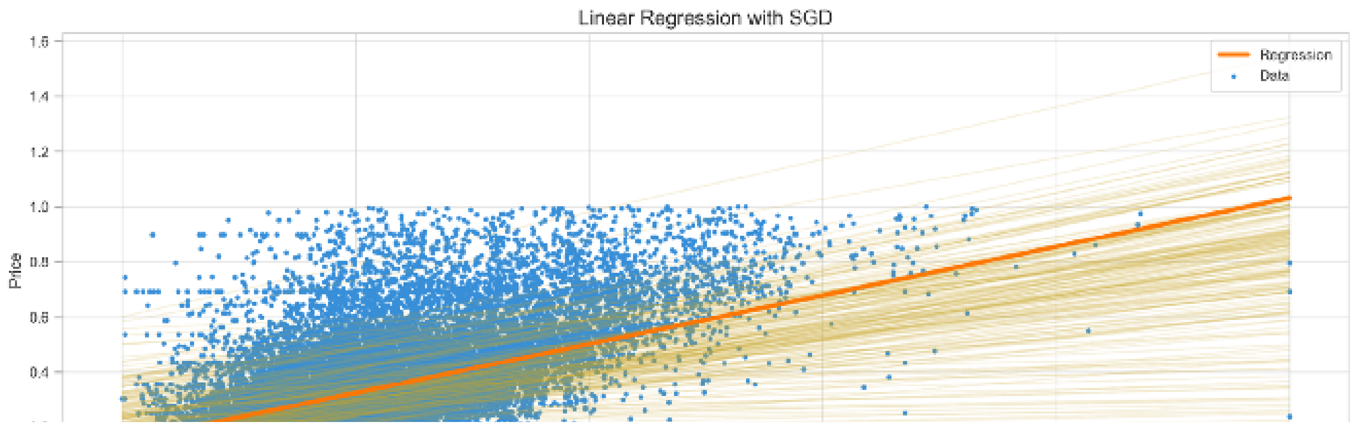


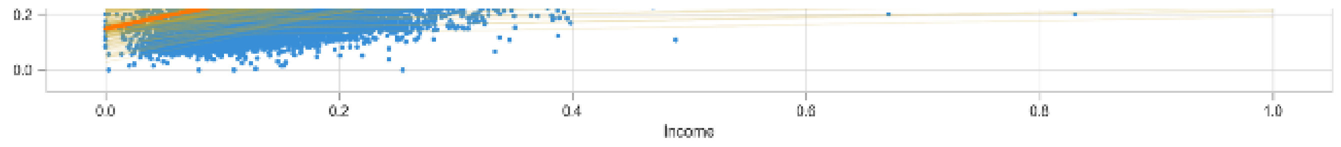
BONUS: Stochastic Gradient Descent

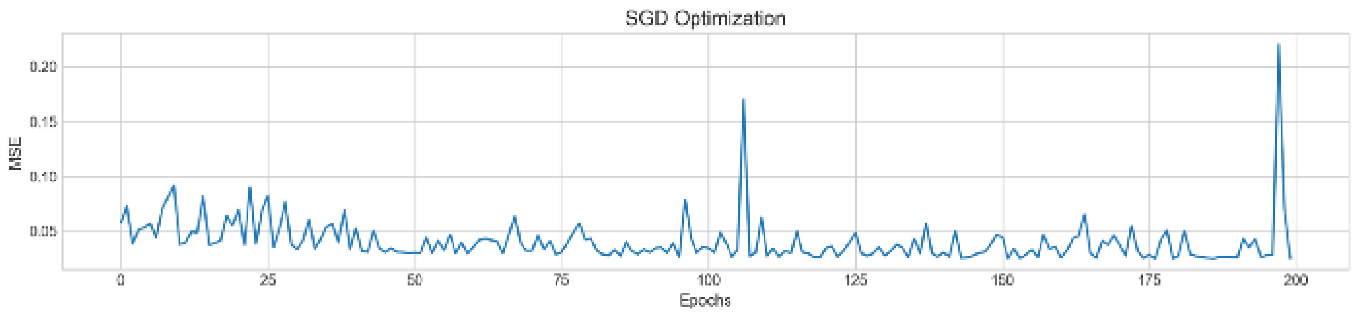
Stochastic Gradient Descent uses only one sample to update parameters, which makes it faster. We can make small corrections to the previous version and see how it performs.

```
def SGD(X, y, lr=0.05, epoch=10, batch_size=1):

    '''
    Stochastic Gradient Descent for a single feature
    '''

    m, b = 0.33, 0.48 # initial parameters
    log, mse = [], [] # lists to store learning process

    for _ in range(epoch):

        indexes = np.random.randint(0, len(X), batch_size) # random
sample

        Xs = np.take(X, indexes)
        ys = np.take(y, indexes)
        N = len(Xs)

        f = ys - (m*Xs + b)

        # Updating parameters m and b
        m -= lr * (-2 * Xs.dot(f).sum() / N)
        b -= lr * (-2 * f.sum() / N)

        log.append((m, b))
        mse.append(mean_squared_error(y, m*X+b))

    return m, b, log, mse
```

We can observe the stochastic nature of the process. The regression line was jumping all over the place, trying to find a minimum based on just one sample. Poor thing.







Conclusion

We've learned how to implement Gradient Descent and SGD from scratch. The same approach we take when we do backpropagation when we train neural networks. If you're interested in implementing Deep Neural Networks from scratch let me know in the comments. Thank you for reading!

Arseniy.

Gradient Descent Linear Regression Machine Learning