Introduction to the Math of Gradient Descent
Andrew Eaton

The purpose of this post is to provide an understanding of the math behind gradient descent – a key concept in the field of deep learning. While this post will not cover deep learning directly, the concepts discussed will provide a good background for understanding deep learning going forward. This is catered towards those with a basic understanding of calculus and Python.

We will learn about gradient descent with reference to a diabetes data set, which contains data from patients who have and do not have diabetes. Our goal is to use this data to build a classifier that can accurately predict whether patients have or do not have diabetes based on their data. The values of the diabetes label (what we are trying to predict) will either be 0 or 1 (0 = does not have diabetes, 1 = has diabetes). This type of problem is known as a "binary classification" problem.

The dataset at a glance looks like this

	0	1	2	3	4	5	6	7	8
0	-0.294118	0.487437	0.180328	-0.292929	0.000000	0.001490	-0.531170	-0.033333	0
1	-0.882353	-0.145729	0.081967	-0.414141	0.000000	-0.207153	-0.766866	-0.666667	1
2	-0.058824	0.839196	0.049180	0.000000	0.000000	-0.305514	-0.492741	-0.633333	0
3	-0.882353	-0.105528	0.081967	-0.535354	-0.777778	-0.162444	-0.923997	0.000000	1
4	0.000000	0.376884	-0.344262	-0.292929	-0.602837	0.284650	0.887276	-0.600000	0

The diabetes data set has 759 rows. Each row is a different patient containing data about them and whether or not they have diabetes. We will define "Y" as column "8", which is the label we are trying to predict – diabetes or no diabetes. Y is a vector with length 759 consisting of y_1, y_2, y_3, … , y_{759}.

The 8 features are columns "0" – "7". We will define "X" as a two-dimensional vector with dimensions 8 * 759 (8 columns by 759 rows). We define the 8 features as x_1, x_2, x_3, … , x_9. The details of the information in the columns is not too important for this lesson, but we can assume they are the results of different medical tests.

Our goal is to take these 8 features and predict the probability someone has diabetes based on the values of the features. How do we make a prediction for whether someone has diabetes based on these features? It all comes down to math. Our predicted probability is a number, each label is a number, and every feature has number values. We can think of predicting this probability of having diabetes as being similar to solving the equation for any line that has the form "y = mx" (in this analogy, we are ignoring "b"). "y" is the probability of having diabetes and "x" is the value of a feature. What is "m" in this analogy? m is the "weight" of the feature - this is one of the key concepts in deep learning.

We will define "W" as a one-dimensional vector with length 8. We will define these weights as $w_1, w_2, w_3, \dots, w_9$. Each value in the weights vector is a list of values where each weight corresponds to a feature, for example w_1 is linked to x_1. This weight vector is what we fine-tune to give us accurate predictions. Going back to the line analogy – just as adjusting the value for slope m changes the shape of the line, adjusting the weights will change our prediction.

So, now that we have a list of features that are numbers, weights that are numbers, labels that are numbers, and we are trying to predict a probability (a number), we can utilize math and computational power to generate predictions.

The first equation

We will now go through an equation for $\hat{y}$.
We define $\hat{y}$ as the predicted probability that a patient has diabetes (that $y = 1$).
If $y = 1$, we want $\hat{y}$ to be large. If $y = 0$, we want $\hat{y}$ to be small.

$$(1) \quad \hat{y} = W^T X$$

Where W is the weight matrix, X is the feature vector, and superscript T means transpose.

Why do we transpose W?

We transpose W because we are dealing with multiplying two matrices. We can't use regular algebra here! The number of the columns of the X matrix must be the same as the number of the rows of the W matrix. This formats the W matrix to be used for matrix multiplication.

Next, for binary classification using logistic regression, we must modify $\hat{y} = W^T X$ to only produce values between 0 and 1, because we are trying to predict a probability. Probabilities can only take on values between 0 and 1. The transformation we will see is called a sigmoid function.

We do this by modifying the previous equation $\hat{y} = W^T X$:

$$\hat{y} = \sigma(W^T X) \text{ where } \sigma(z) = \frac{1}{1 + e^{-z}}$$

Why do we transform the ŷ equation?

This will force ŷ to stay in the bounds between values of 0, and 1. In the equation $\sigma(W^TX)$: if W^TX is a large value, ŷ will approximate to 1, because $e^{-(\text{large number})}$ will cause $\frac{1}{1+e^{-z}}$ to be approximately equal to $\frac{1}{1} \approx 1$. If z is a large negative number, then ŷ will approximate to 0, because $e^{-(\text{large negative number})}$ will cause $\frac{1}{1+e^{-z}}$ to be approximately equal to $\frac{1}{1+\text{large number}} \approx 0$.

At this point, we have an equation to generate a value between 0 and 1 for ŷ and we have values of either 0 or 1 for y. How do we generate accurate predictions of ŷ that are closer to y? This is where the "loss function" comes in. Essentially, the loss function will keep track of the error in the model by tracking the difference between ŷ and y. We want to minimize the value of the loss function. By doing this, the model will become more accurate, since ŷ will be closer to y. The loss function takes into account one of the samples of the dataset. Let's look at the equation.

(2) $L(\hat{y}, y) = -(y \log \hat{y} + (1 - y)(1 - \log \hat{y}))$

This specific type of loss function is often referred to as "log loss" or "cross-entropy loss". The equation looks intimidating, but thanks to algebra it can be simplified. Depending on whether y = 1 or y = 0, the loss function takes on two simpler forms.

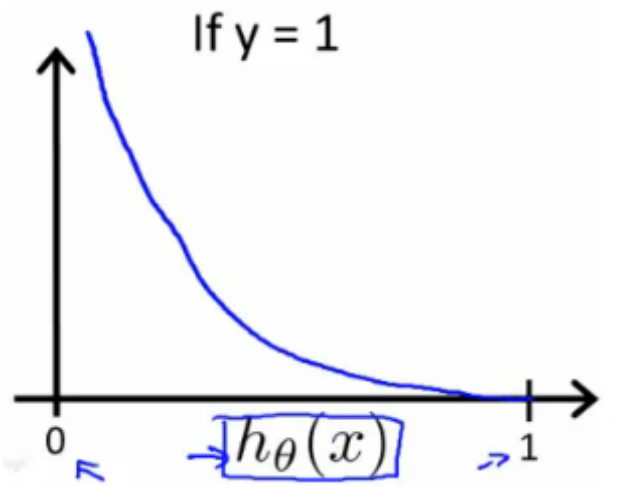
If y = 1: $L(\hat{y}, y) = -(\log \hat{y})$

If y = 0: $L(\hat{y}, y) = -(1 - \log \hat{y})$

If ŷ does not match the value of y, then the loss function will take on a large value because of these two equations. To get a better understanding of why this is, let's graph the loss function as a function of ŷ when y = 1 and then when y = 0.

Let's look at what happens when y = 1. Below is the graph the loss function as a function of ŷ.

If y = 1: L(ŷ, y) = - (log ŷ):

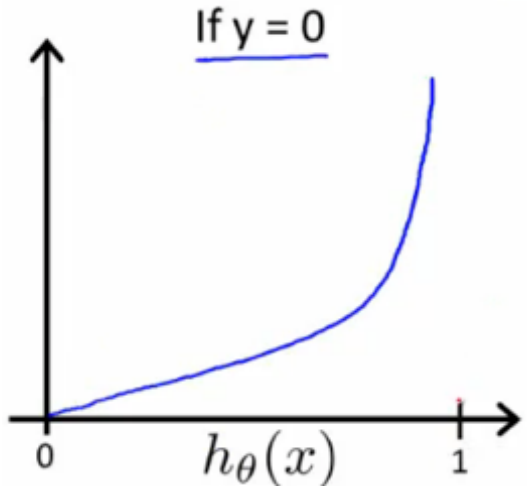


In the diagram above, Andrew Ng has designated ŷ as $h_0(x)$.

The above graph is the equation the loss function takes on if y = 1. In this scenario, an accurate model should predict ŷ to be close to a value of 1. The value of the loss function is on the y-axis and ŷ (here denoted as $h_0(x)$) is on the x-axis. We can see that the closer ŷ is to 0, the loss function becomes equal to a larger value. This is because the true value is 1 and our prediction is incorrect because it is close to 0. As the value of ŷ approaches 1, the value of the loss function approaches 0. Soon, we will see how to update our predictions to minimize the loss function. This will fine-tune the model to more accurately choosing a value for ŷ that matches the corresponding value for y.

Now, let's look at what happens when y = 0. We will graph the loss function as a function of ŷ.

If y = 0: $L(\hat{y}, y) = -(1 - \log \hat{y})$



In the diagram above, Andrew Ng has designated ŷ as $h_0(x)$.

The above graph is the equation the loss function takes on if y = 0. In this scenario, an accurate model should predict ŷ to be close to a value of 0. The value of the loss function is on the y-axis and ŷ (here denoted as $h_0(x)$) is on the x-axis. We can see that the closer ŷ is to 1, the loss function becomes equal to a larger value. This is because the true value is 0 and our prediction is incorrect because it is close to 1. As the value of ŷ approaches 0, the value of the loss function approaches 0. Soon, we will see how to update our predictions to minimize the loss function. This will fine-tune the model to more accurately choosing a value for ŷ that matches the corresponding value for y.

Since the loss function describes one sample, we want to look at the average loss function for all of the training examples. This brings us to equation 3, which is essentially just the mean of all of the loss functions.

$$(3) \quad J(w) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}_i, y_i)$$

Where i refers to the i^{th} sample. Now that we have our loss function, our objective is to tweak our predictions to make them more accurate. How do we adjust our predictions? We do this by adjusting the weight vector W to minimize the value of the loss function, since this translates to a minimized difference between ŷ and y, which makes our classifier more accurate.

So, how do we adjust the weights?

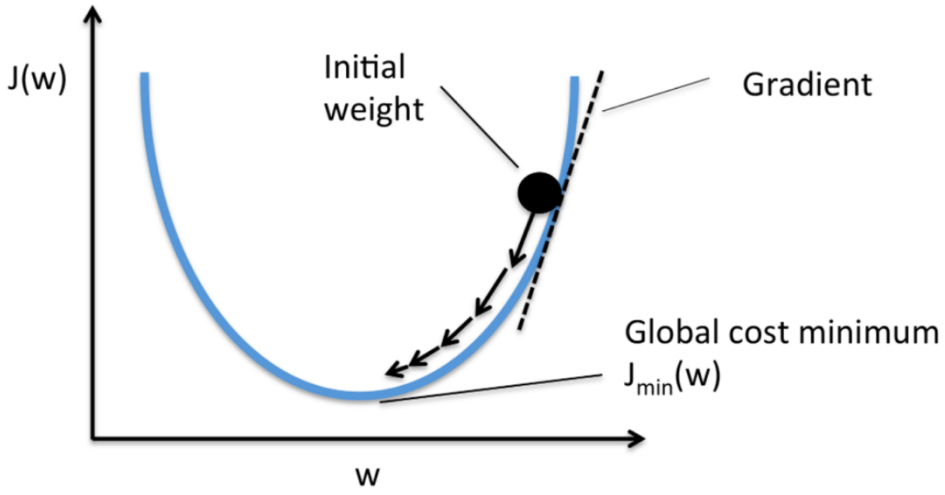
This is where "gradient descent" comes in. We initialize our weights to random values. Based on the initial weight values, we calculate the lost function. We then update the weights to produce a lower loss function. This is repeated until the model becomes accurate enough to serve what specific purpose you are looking for.

This brings us to our next equation:

(4) $w_1 := w_1 - \alpha(\frac{dL}{dw1})$

Here, w_1 is the first weight value in the weight vector. the ":=" symbol means to update. So, w_1 becomes equal to a new value based on the equation. α is the learning rate (a constant – the details of which are not too important for the context of this post) and $\frac{dL}{dw1}$ is the derivative of the loss function with respect to w_1. We will go through the derivative aspect in more detail later, but let's stick to the basics for now.

This equation updates our weights. While this equation shows one specific weight, it applies to the entire weight vector. Our goal is to continually update our weights until we have an accurate classifier. Why do we subtract the derivative of the loss function with respect to the weight? This is best explained graphically.



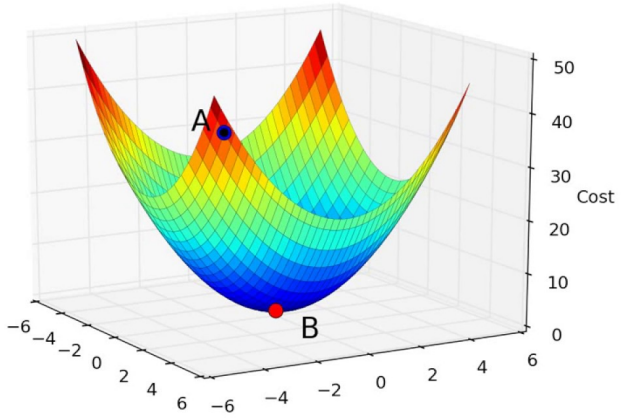
Gradient Descend Visualization. Credit: rasbt.github.io

Here, the y-axis is J(w) – the value of the loss function. On the x-axis, w is one of the weights and the different values it can take on. There is a certain weight value that equates to a minimum loss function value. The black dot on the graph is the value of the loss function based on a randomly initialized value of w. Our goal is too head towards the minimum.

By subtracting the derivative of J(w) with respect to w from the original w, we can move the value of J(w) towards a minimum because the initial point will follow the black arrows.

Specifically, if the black dot is on the right side of the minimum, the derivative of the loss function with respect to the weight will be positive. So, subtracting the derivative will move the dot towards the minimum. If the black dot happened to be on the left side of the minimum, then the derivative would be negative. Subtracting the negative derivative would be the same as adding a positive value, so the dot would move towards the right and eventually the minimum.

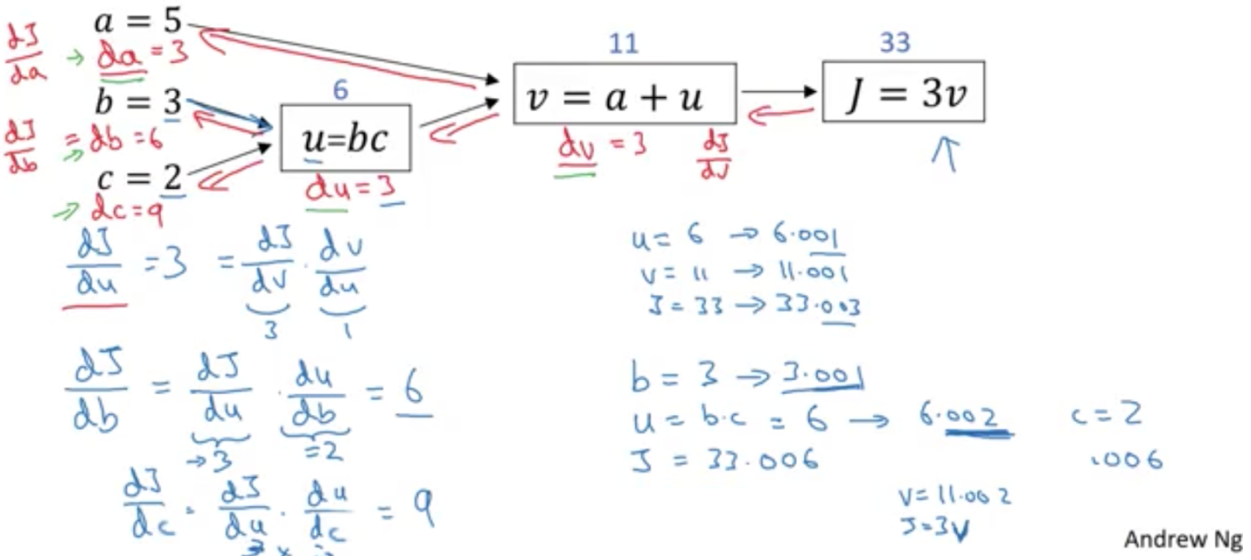
If you have two weights, this would take place in three dimensions. Both of the weights are updated simultaneously to reduce the value of the loss function. This also applies to three or more weights. We can see the example of finding a minimum value when there are two weights by looking at the below graph, where we want to move from A to B (minimize the loss function):



So, how do we calculate the derivative of the loss function with respect to the different weights?

A good way to explain this in more detail is through a “computation graph”.

The below example will be used to explain computation graphs in general and then we will apply this theory to our classification problem.



Above is an example of a computation graph with three inputs: a, b, and c. We can think of these as three different weights.

There are two intermediate equations u(b,c) = bc and v(a,u) = a + u.
The final output is the equation J(v) = 3v.

We can see how the initial three inputs and two intermediate equations impact the final function J(v) by finding the derivative of J(v) with respect to the intermediate equations and inputs.

To start, let's find the derivate of J(v) with respect to v. The derivative is the ratio of change of the dependent variable to that of the independent variable. So, as the independent variable increases by a certain amount, we can measure how much the dependent variable increases. This can be used to calculate the derivative.

In our above computation graph, v will be the independent variable and J the dependent variable. As v increases by a very tiny amount (let's say from a value of 11 to 11.001), J increases by the same tiny amount multiplied by 3 (33.003). Therefore, $\frac{dJ}{dv} = 3$. Where $\frac{dy}{dx}$ represents how the dependent variable y changes based on the independent variable x.

We can keep going backwards to find derivatives with respect to earlier equations and eventually the initial inputs. To find how the function u(b,c) impacts J(v), we will use the "chain rule". The chain rule states that $\frac{dJ}{du} = \frac{dJ}{dv} * \frac{dv}{du}$. We already know that $\frac{dJ}{dv} = 3$.

We can find the value of $\frac{dv}{du}$ by seeing how much a tiny change in the value of u impacts the value of v. If u = 6 and we change its value to 6.001, then the value of v goes from 11 to 11.001. Therefore, $\frac{dv}{du} = 1$.

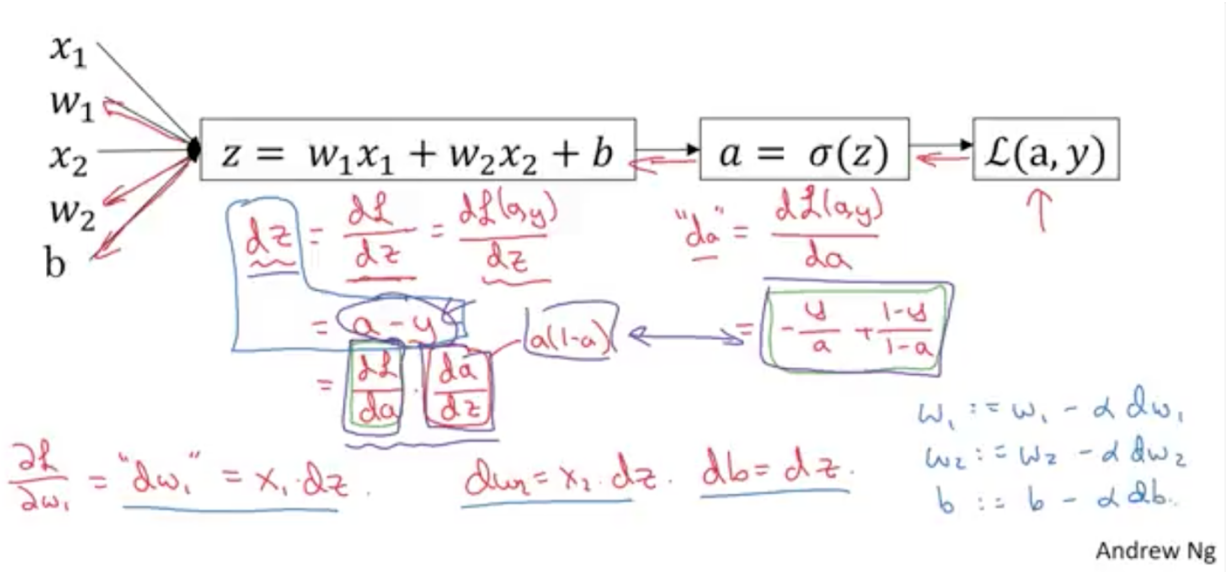
So, $\frac{dJ}{du} = \frac{dJ}{dv} * \frac{dv}{du} = 3 * 1 = 3$

We have now found the derivative of J(v) with respect to the functions v and u. Next, let's see the derivative J(v) with respect to the initial inputs a, b, and c. For input b, we can use the chain rule to state that $\frac{dJ}{db} = \frac{dJ}{du} * \frac{du}{db}$. We already know that $\frac{dJ}{du} = 3$, so we just need to find the value of $\frac{du}{db}$. So how does a small change in b affect u(b,c)?

If we increase b from 3 to 3.001, the u(b,c) will change from 6 to 6.002. Thus, $\frac{du}{db} = 2$.

So, $\frac{dJ}{db} = \frac{dJ}{du} * \frac{du}{db} = 3 * 2 = 6$. We can use the same procedure to find $\frac{dJ}{da}$ and $\frac{dJ}{db}$.

To extrapolate this demonstration to our actual dataset, let's say that a, b, and c were three of the initial weights. The computer will use the loss function to update these weights by finding their derivatives with respect to the loss function.
Let's apply what we learned in the above example to logistic regression. This is an example for the loss function, but the same idea is applied to the loss function.



Here, we have a computation graph of logistic regression. Instead of the previous graph with random equations, our final output equation is our loss function. The intermediate equations are $\hat{y}$ and its wrapping in the sigmoid function. In the above diagram, Andrew Ng depicts $\hat{y}$ as "z" and the wrapping of $\hat{y}$ in the sigmoid function as "a".

In order to see how the weights impact the loss function, the chain rule is applied in a similar manner as before. We won't go through the calculus proofs here, but rather just show the solutions. $\frac{dL}{dx}$ denotes the derivative of the loss function with respect to "x".

$$\frac{dL}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

$$\frac{dL}{dz} = a - y$$

$$\frac{dL}{dw1} = x_1 (a - y)$$

The computer is able to automatically calculate these derivatives for us. The computer uses these derivatives, which we call "gradients" in machine learning, to update the weights. This is repeated until the loss function decreases to a certain value.

Let's summarize what we have gone over.

Let's quickly review gradient descent through logistic regression for a binary classification problem.

1.) Randomize values weight vector "W" of length three equal to w_1, w_2, w_3. Each weight w corresponds to a feature x.

2.) Calculate a value for $\hat{y}$ using the following equations:

$$\hat{y} = W^T X$$

$$\hat{y} = \sigma(W^T X) \text{ where } \sigma(z) = \frac{1}{1 + e^{-z}}$$

3.) Calculate the value of the loss function using the following formula:

$$L(\hat{y}, y) = -(y \log \hat{y} + (1 - y)(1 - \log \hat{y}))$$

4.) Find the average value of the loss function for all samples:

$$J(w) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}_i, y_i)$$

5.) Calculate the derivative of the loss function with respect to each weight. This is our "gradient". We then update our weights using our calculated gradients:

$$w_1 := w_1 - \alpha\left(\frac{dL}{dw1}\right)$$

6.) Repeat steps 2 – 5 until your model is accurate enough for your liking.

Luckily, computers are able to manage all of the above procedure relatively easily for us. They are able to calculate the loss function, or the "forward propagation step", and find its derivatives with respect to the weights, or "backpropagation". It can then update the weights for us.

To end this post, I will leave code for a simple gradient descent in numpy. It will have one feature x, one weight m, and one value y. These values are initialized randomly. Gradient descent will optimize m until the prediction is close to the real value for y. Instead of using the log-loss function, we will use the more simple squared error loss function.

```python
#Let's make a simple gradient descent example from scratch
import numpy as np

#Create a value for y, x, and m
y = np.random.randn(1)
x = np.random.randn(1)
m = np.random.randn(1)

#We are trying to predict y by tweaking the value of m
#Let's calculate an initial prediction
y_pred = x * m

#Next let's calculate the loss through root squared error (forward propagation)
def calculate_loss(y_pred, y):
    loss = (y - y_pred)**2
    return loss

#Calculate the derivative/gradient of this simple loss function (back propagation)
def calculate_gradient(y_pred, y):
    gradient = 2 * (y - y_pred)
    return gradient

#Update step
def update_weight(m, gradient):
    m -= 0.01 * (gradient)
    return m
```

```python
#Now, we need to write a loop for this and print out the loss function after every iteration
print("Original x, m, and y: ", x, m, y)

for epoch in range(1000):

    #Make a prediction for y
    y_pred = m * x

    #Calculate the value of the loss function
    loss = calculate_loss(y_pred, y)
    loss = loss[0]

    #Calculate the gradient
    gradient = calculate_gradient(y_pred, y)
    gradient = gradient[0]

    #Update the weight
    m = update_weight(m, gradient)
    m = m

    #Print out results
    if epoch % 100 == 0:
        print("-----------------------------------------")
        print("Epoch: ", epoch)
        print(f"y_pred: {y_pred}, y: {y}")
        print("Loss: ", loss)
        print("Value of weight: ", m)
print("-----------------------------------------")
print("Final x, m, and y: ", x, m, y)
```

```
Original x, m, and y:  [-1.12242755] [1.06498435] [-0.290105]
-----------------------------------------
Epoch:  0
y_pred: [-1.19536778], y: [-0.290105]
Loss:  0.8195006880017411
Value of weight:  [1.0468791]
-----------------------------------------
Epoch:  100
y_pred: [-0.38358895], y: [-0.290105]
Loss:  0.008739248374301
Value of weight:  [0.33987973]
-----------------------------------------
Epoch:  200
y_pred: [-0.29975883], y: [-0.290105]
Loss:  9.319633682547794e-05
Value of weight:  [0.26686989]
-----------------------------------------
Epoch:  300
y_pred: [-0.29110193], y: [-0.290105]
Loss:  9.938563164344092e-07
Value of weight:  [0.25933037]
-----------------------------------------
Epoch:  400
y_pred: [-0.29020795], y: [-0.290105]
Loss:  1.0598596590384224e-08
Value of weight:  [0.25855178]
-----------------------------------------
Epoch:  500
y_pred: [-0.29011564], y: [-0.290105]
Loss:  1.1302463728896095e-10
Value of weight:  [0.25847138]
-----------------------------------------
Epoch:  600
y_pred: [-0.2901061], y: [-0.290105]
Loss:  1.205307563391514e-12
Value of weight:  [0.25846308]
-----------------------------------------
Epoch:  700
y_pred: [-0.29010512], y: [-0.290105]
Loss:  1.2853536663934478e-14
Value of weight:  [0.25846222]
-----------------------------------------
Epoch:  800
y_pred: [-0.29010502], y: [-0.290105]
Loss:  1.3707157234604092e-16
Value of weight:  [0.25846213]
-----------------------------------------
Epoch:  900
y_pred: [-0.29010501], y: [-0.290105]
Loss:  1.4617465301232346e-18
Value of weight:  [0.25846212]
-----------------------------------------
Final x, m, and y:  [-1.12242755] [0.25846212] [-0.290105]
```

We can see as the gradient descent runs and the number of epochs increases, the loss function decreases and the predicted y value becomes closer to the real y value. m has changed in value to make our model more accurate.

Sources:

Andrew Ng's deep learning course through the Deep Learning Boston YouTube channel:

https://www.youtube.com/watch?v=7PiK4wtfvbA&list=PLBAGcD3siRDguyYYzhVwZ3tLvOyyG5k6K

Sung Kim's YouTube series on pyTorch:

https://www.youtube.com/watch?v=SKq-pmkekTk

http://ml-cheatsheet.readthedocs.io/en/latest/logistic_regression.html

https://stats.stackexchange.com/questions/130983/why-take-transpose-of-regressor-variable-in-linear-regression

https://medium.com/@lachlanmiller_52885/machine-learning-week-1-loss-function-gradient-descent-and-univariate-linear-regression-8f5fe69815fd