Ali Gholami
(aligholamee)

Technical Report (**December 2017**)

The Hornburg

Crawling the history of deep learning with Python.

Amirkabir University of Technology, Tehran, Iran.
p. +989396191804
f. +989396191804
aligholamee@yahoo.com
https://aligholamee.com

Table of Contents

I. **Technical Report (December 2017)** 0

II. **Preface** 2

Highlights

Objectives

Machine learning viewpoint

Implementation concerns

III. **Least Squared Error – LSE** 4

LSE explained and coded

LSE results and outputs

IV. **Gradient Descent** 7

Gradient descent explained and coded

Gradient descent results and outputs

V. **Linear Regression** 10

Linear regression explained and coded

Linear regression results and outputs

VI. **Perceptron Learning** 13

Perceptron learning coded and explained

Perceptron results and outputs

VII. **Artificial Neural Network** 15

Artificial neural network explained and coded

Artificial neural network results and outputs

Preface

There is always a beginning to everything. The beginning of the today’s practical and broad usage of deep learning is **Hornburg**. Although it may seem a bit strange at the first glance, there are so many meanings to this word. Derived from The Lord of the Rings by **Peter Jackson**, this movie was my one of the top ten **unforgettable** movies in my life (after **Inception**). I’ve called this project **Hornburg** to emphasize the purpose of that. Keep up the good read to find out more on the project.

***Algorithms** who forget their past are meant to **repeat** it (Same as **humans**)*

Highlights

I'll be going through the basics of deep learning and especially learning itself. The focus is mainly on the implementation and not the same talks about the machine learning. I'll be using some beloved libraries such as **numpy** and **matplotlib** but not too much, just to let the focus to be stayed on the implementation.

Objectives

- *A little bit of talk on my **viewpoint** of machine learning.*
- *Some minor sentences on the **implementation** concern of mine.*
- *Coding and explaining the **Least Square** method.*
- *Coding and explaining the **Gradient Descent** method.*
- *Coding and explaining the **Linear Regression** method.*
- *Coding and explaining the **Perceptron** method for neural networks.*
- *Coding and explaining the **Back Propagation** and **Forward Propagation** in artificial neural networks.*

Machine learning viewpoint

We all know that machine learning was present even before 1980 (When the louie-louie brothers began to shine!), But the practical usage and the capacity of the machines didn't let the machine learning to shine those days. Though we cannot forget the IBM's amazing progress with the assistance of Arthur Samuel – the founder of machine learning in IBM's machines – and the great results on the blue brain computer and other artificial heroes of this field, the most drastic bottleneck of machine learning was its implementation and the practical usage in industries and other world evolving technologies like social networks, cyber security, product sales and other awake computer categories in the world. My overall viewpoint of machine learning is the integration of two quotes by **Geoffrey Hinton** and **Linus Torvalds**.

*"The **AI** coded by a not – cheap – talking programmer will be **better** than us at a lot of things."*

Implementation concerns

Programmers who talk too much instead of coding, are not good ones. Either they are not closely involved into the code or they are not good at the implementation at all. I'm not saying that I've implemented such amazing frameworks and git repositories but, I've always tried to balance the documentation and the code! In this project, there is great chance for beginners like me, to understand what is exactly happening in the creation of such great frameworks like **Tensorflow, Caffe, Theano, etc**. and the basic understanding of working with **numpy, matplotlib** and **python** for implementation of algorithms of machine learning are covered.

Least Squared Error – LSE

While crawling through the machine learning concepts, you will hear about the term **Error** more than any other term! To keep everything simple, I can explain the error as “The offset of being true!” or more scientifically “The error is the difference of known values and the predicted values.”

While doing machine learning stuff, there are two main purpose of using algorithms and they are either **prediction** or **ranking** problems. The first job to do is to let the algorithm (we call it machine) to learn from **data** and then we test or more scientifically, we **validate** the trained algorithm by using some new data. The first task (learning from data) is the main concern in this section. While training the algorithm, we are not 100 percent accurate at the very beginning but, the algorithm will improve itself by converging the data. Each time the algorithm feeds the data, the error of the algorithm will be reduced. This is the main point which **learning** is happening.

The **LSE** or **Least Squared Error** is a method for **minimizing** the error in machine learning. The term LSE is described as the following.

$$Error = (realOutput - predictedOutput)^2$$

To understand the implementation of the LSE let’s assume we have the following training set of some points in the two dimensional space.

$$Training\ Set\ = \begin{matrix} 3 & 5 \\ 5 & 3 \\ 8 & 4 \\ 3 & 1 \\ 6 & 4 \\ 5 & 4 \\ 7 & 5 \\ 8 & 3 \end{matrix}$$

And we want to **fit** a linear function to these points to **predict** the next data. Soon we understand that this is called a **linear regression** but for now, we want to focus on the errors. To understand the concept of error and LSE, I’ll be using some random linear functions as described below, to find the error between these linear functions and the **actual function that fits the above data**.

$$\textit{Random Linear Functions} = \begin{array}{l} y = 5x + 6 \\ y = 3x + 1 \\ y = 6x + 4 \\ y = 6x + 8 \\ y = 3x + 4 \\ y = 4x + 7 \end{array}$$

LSE explained and coded

```
import matplotlib.pyplot as plt
import numpy as numpy

dataset = numpy.array([[3,5],[5,3],[8,4],[3,1],[6,4],[5,4],[7,5],[8,3]])

slope_list = [5, 3, 6, 6, 3, 4]
constant_list = [6, 1, 4, 8, 4, 7]

plot_titles = [
    'y = 5x + 6',
    'y = 3x + 1',
    'y = 6x + 4',
    'y = 6x + 8',
    'y = 3x + 4',
    'y = 4x + 7'
]
```

Some python imports are happening at the beginning. We'll need **matplotlib.pyplot** to visualize the errors and the **numpy** for matrix stuff. The data set given above is defined as an numpy array and the next three arrays are describing the random function we just talked about.

```
def computeErrorForLineGivenPoints(b, m, coordinates):
    totalError = 0

    for i in range(0, len(coordinates)):
        x = coordinates[i][0]
        y = coordinates[i][1]

        # Calcuate the error
        totalError += (y - (m * x + b)) ** 2

    return totalError / float(len(coordinates))
```

The function above is the core logic where error calculation is happening at. It simply grabs the random function bias and slope value, iterates through the points in dataset and finds the total difference between those points and the real value **y** of those points.

```
errorlist = []

for i in range(0, 6):
    errorlist.append(computeErrorForLineGivenPoints(slope_list[i],
constant_list[i], dataset))
    print("Hypothesis " + plot_titles[i] + " error: ")
    print(errorlist[i])
```

This is the place of code which error computing calls are happening. This will be done for all random functions (6 times) and the results are appended to the errorlist array.

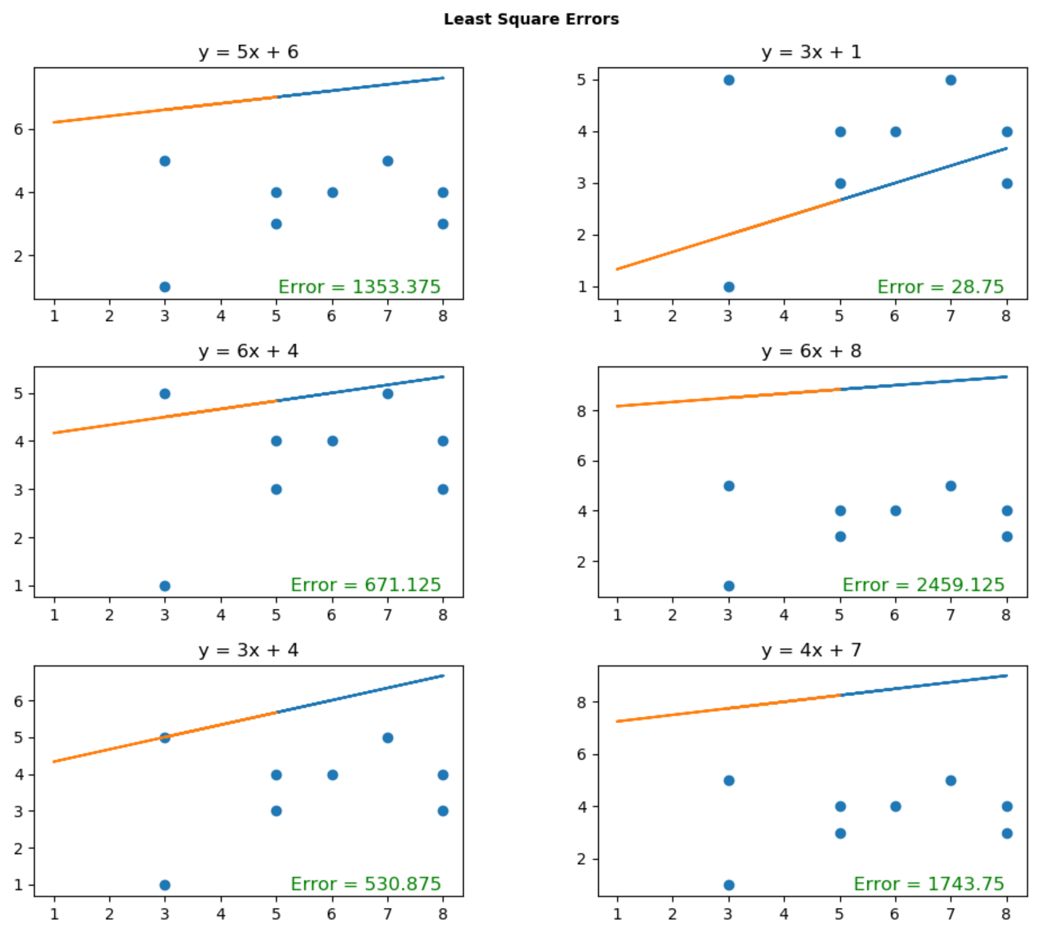
The final section of the code is for the plotting purposes. Make sure you have looked at the documentation of **Matplotlib** to understand what's happening here.

```
fig = plt.figure()
fig.suptitle('Least Square Errors', fontsize=10, fontweight='bold')

for i in range(1, 7):
    ax = fig.add_subplot(3, 2, i)
    ax.title.set_text(plot_titles[i-1])
    ax.scatter(dataset[:,0],dataset[:,1])

    errorLabel = "Error = "
    ax.text(0.95, 0.01, errorLabel + str(errorlist[i-1]),
            verticalalignment='bottom', horizontalalignment='right',
            transform=ax.transAxes,
            color='green', fontsize=12)
    plt.plot(dataset, dataset/slope_list[i-1] + constant_list[i-1])

plt.show()
```

Now it's time to have a look at the result of the above code.

LSE results and outputs

The final result shows that we have a abnormally distributed set of points in the dataset and on each plot, there is the error of each random function is displayed also.



Gradient Descent

Gradient descent is an awesome optimization algorithm which is used to find the minimum of a function. The error formula given in the previous section has to be minimized in real learning. To minimize the error some parameters like **weights** in **neural networks** or **bias** and **slope** in linear functions are configured. To understand the gradient descent, we can think of a person who is going through a diagram to find the minimum. Each position of this person as acquired by taking a step from the previous position. Stepping through the current slope of the given function will give us the minimum value of it (whether it is + or -). For deeper understanding of gradient descent, make sure you read the Wikipedia page of gradient descent.

Gradient descent explained and coded

There are some initial definitions for the parameters that will be used in the code. The starting point in the diagram is the value of **current_x** and the size of each step through the minimum will be called **learning_rate**. There is also another parameter called **num_iteration** which indicates the number of steps we are taking toward the minimum.

```
# Initial Definitions
current_x = 0.5
learning_rate = 0.02
num_iterations = 150

# Goal Function is 5x^4 - 6x^2
def findSlopeAtGivenPoint(x):
    return 5 * x ** 4 - 6 * x ** 2
```

The function we are stepping through is

$$y = x^5 - 2x^3$$

The defined function above, represents the derivative of y with respect to x at each point x.

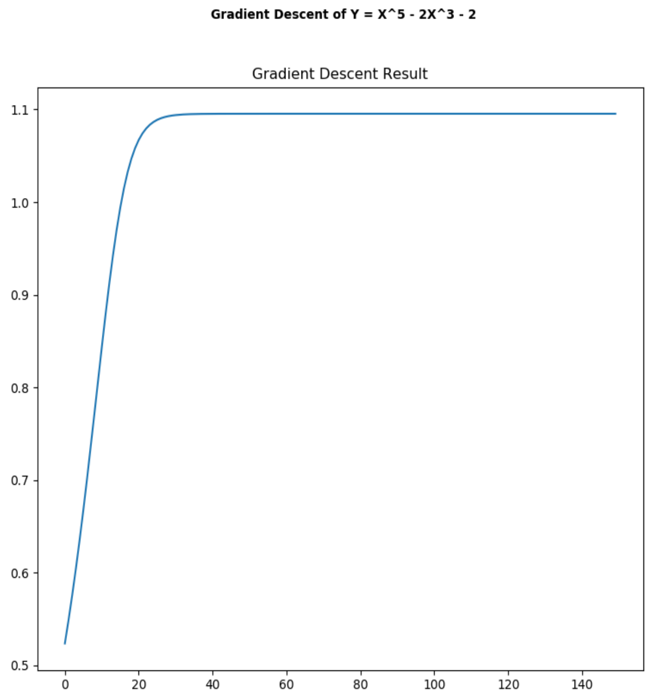
```
def trainGradientDescent(iter_range, current_x):

    # Gradient Descent Result Array
    gd_result = []
    for i in range(iter_range):
        previous_x = current_x
        current_x += -learning_rate * findSlopeAtGivenPoint(current_x)
        gd_result.append(current_x)
        print("X was updated to: ", previous_x)

    return gd_result
```

Gradient descent results and outputs

The function for training the gradient descent algorithm will update the current_x each time through iteration. The results are given below:

```
X was updated to:  0.5
X was updated to:  0.52375
X was updated to:  0.5491428670114746
X was updated to:  0.5762360976132261
X was updated to:  0.6050562723367592
X was updated to:  0.6355850282082159
X was updated to:  0.6677421865687365
X was updated to:  0.7013668776536748
X was updated to:  0.736198651676305
X was updated to:  0.7718621177378439
X was updated to:  0.8078603294026802
X was updated to:  0.8435832502155267
X was updated to:  0.878337064243611
X was updated to:  0.9113966624400573
X was updated to:  0.9420770026722712
X was updated to:  0.9698108556236001
X was updated to:  0.9942145766978934
X was updated to:  1.0151242556331077
X was updated to:  1.0325931889797122
X was updated to:  1.0468544045208217
X was updated to:  1.0582623124909367
X was updated to:  1.067230723144791
X was updated to:  1.0741806264197324
X was updated to:  1.0795040913567826
X was updated to:  1.083544394119272
X was updated to:  1.0865889351283424
X was updated to:  1.0888705298790056
X was updated to:  1.0905732224348028
X was updated to:  1.0918398856667206
X was updated to:  1.0927799469850163
X was updated to:  1.0934763842733253
X was updated to:  1.0939916548274276
X was updated to:  1.09437251325024
X was updated to:  1.094653817832125
X was updated to:  1.0948614797892693
X was updated to:  1.095014717182773
X was updated to:  1.0951277605997813
X was updated to:  1.0952111348267959
X was updated to:  1.095272616972148
X was updated to:  1.0953179500330399
X was updated to:  1.0953513728721442
X was updated to:  1.0953760130481158
X was updated to:  1.095394177560879
X was updated to:  1.0954075678074047
X was updated to:  1.0954174383775594
```

After 150 iterations the value of current_x won't change. Since it has reached the global minimum or maybe stuck at the local minimum. The following diagram shows the changes on the gradient learning:



Linear Regression

While doing prediction, we need to use a function to behave like a predictor with respect to some inputs. To find that function (which is linear in this special case), we have to fit a linear function as much as possible to our training data. This is called a linear regression. The prediction values are continuous obviously. Here I'll describe the linear regression with respect to the gradient descent optimization algorithm to find the best hypothesis matching the current training dataset.

Linear regression explained and coded

Entering the world of machine learning, the first thing to do while coding a model is to define the parameters that are essential to that model. We are starting with a linear function with the **slope** of 2 and the **bias** value of 5. These values will be improved on each iteration to better fit the data.

Here is the first section of the code defining the parameters to be used:

```
dataset = np.array([[1,2.5],[2,3.5],[3,4.6],[4,4.8],[5,5.9],[6,7.1],[6.5,7.5]])

# Define the initial values
learning_rate = 0.02
numOfIterations = 0
initialConstant = 5
initialSlope = 2

# Testing declarations
iterations_array = [10, 30, 60, 150, 500, 1000]
```

I've decided to run this program for 10, 30, 60, 150, 500 and finally for 1000 times to check the difference between the learned linear function. Here is the function containing the core logic of the learning:

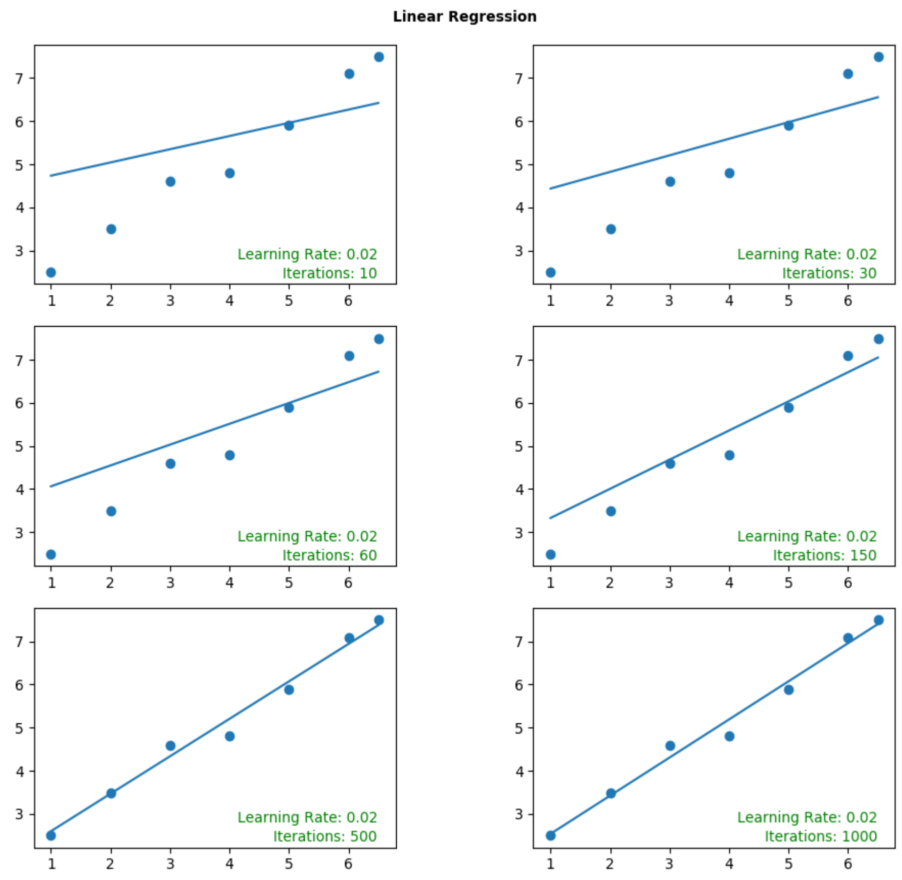
```
def trainWithGradientDescent(coordiantes, h_slope, h_constant, learning_rate):

    # Indicates how much h values must be updated
    slope_gd_rate = 0
    constant_gd_rate = 0

    # Indicates the new slop and constant for each hypothesis
    updated_h_slope = h_slope
    updated_h_constant = h_constant

    # Repeat on each single data
    # This is gradient descent obviously :)
    for i in range(0, len(coordiantes)):

        # Grab the current x and y
        x = coordiantes[i][0]
        y = coordiantes[i][1]

        constant_gd_rate += -2/len(coordiantes) * (y - (h_slope * x +
h_constant))
        slope_gd_rate += -2/len(coordiantes) * x * (y - (h_slope * x +
h_constant))
        print("Constant error: " + str(constant_gd_rate))
        print("Slope error: " + str(slope_gd_rate))

    updated_h_constant = h_constant - (learning_rate * constant_gd_rate)
    updated_h_slope = h_slope - (learning_rate * slope_gd_rate)
    return [updated_h_constant, updated_h_slope]
```

On each step, the algorithm is finding the improved value for the initial slope and bias by minimizing the error between the initial linear function and the actual outputs of the dataset. The next function is for the initialization of the above algorithm.

```python
def gradientDescentInitializer(cooridnates, initial_slope, initial_constant,
learning_rate, numOfIterations):

    # First value for the trained constant
    # and trained slope
    trained_constant = initial_constant
    trained_slope = initial_slope
    # Start training for numOfIterations times
    for i in range(0, numOfIterations):
        trained_constant, trained_slope = trainWithGradientDescent(cooridnates,
trained_slope, trained_constant, learning_rate)

    return [trained_constant, trained_slope]
```

Linear regression results and outputs

The final result shows that after 500 times of iterations, the initial hypothesis is completely fitted to the given data.



Perceptron Learning

This is where human began to find out what is happening to the data transfer inside the brain. There are some awesome material talking about **perceptron learning** algorithm and the basics of neuroscience which is essential to understand this concept. There are some terms called **neuron** in the brain which can be assumed as a **function** which takes some inputs and generates the desired outputs after some proper processing on the inputs. The classification problems which groups the labeled data into some classes with predefined attributes use this technique very often. Perceptron learning at its own is the representation of the **one-layer perceptron** algorithm which is presented by a simple neuron with some inputs and one output. The output can be 0 or 1. So the classification we are looking at is called a **binary classification**. There is a function at the core of the neuron to decide the output based on the summation of its inputs.

Perceptron learning coded and explained

Here is the initial setup for the algorithm. The dataset, parameters and an activation function is described. The activation function works as a decision function on the output with a static threshold.

```
dataset = [          (array([3, 4, 5]), 1),
                     (array([3, 4, 6]), 0),
                     (array([3, 3, 5]), 0),
                     (array([3, 8, 5]), 0),
                     (array([4, 4, 5]), 1),
                     (array([1, 0, 5]), 1),
                     (array([7, 4, 8]), 1),
                     (array([1, 6, 4]), 0),
                     (array([3, 4, 5]), 0),
                     (array([4, 4, 6]), 1),
                     (array([9, 5, 5]), 0)
]

# Define the initial values
learning_rate = 0.2
weights = random.rand(3)
numOfIterations = 150

activationFunction = lambda x: 0 if x < 0 else 1
```

In this case the values 0 and 1 at the end of each row of the dataset shows the true label for the training data. Each input in the perceptron algorithm is multiplied by a value called **weight**. Improving these weights for better prediction is the goal of the perceptron (and other neural networks).

The dot product of input and the weights are going through the activation function to predict the result. The output error is then calculated and the final term for adding to the weights (to make them predict better) is calculated by multiplying the output error and input vector with respect to some learning rate. The less the errors on the output, the less we want to update the internal weights.

```
def trainSingleLayerPerceptron(dataset, weights, numOfIterations, learning_rate):
    for i in range(numOfIterations):
        # Select randomly from dataset
        inputVector, label = choice(dataset)

        # Find the dot product of weights4 and input vector
        result = dot(weights, inputVector)

        # Find the error
        resultError = label - activationFunction(result)

        # Update the weights
        weights += learning_rate * resultError * inputVector

    return weights
```

Perceptron results and outputs

The result shows that the model is not enough accurate to predict the own trained set. The problem maybe because of the learning rate or it may be improved by increasing the number of iterations.

```
Classified  [3 4 5] as  1
Classified  [3 4 6] as  1
Classified  [3 3 5] as  1
Classified  [3 8 5] as  0
Classified  [4 4 5] as  1
Classified  [1 0 5] as  1
Classified  [7 4 8] as  1
Classified  [1 6 4] as  0
Classified  [3 4 5] as  1
Classified  [4 4 6] as  1
Classified  [9 5 5] as  1
```

Artificial Neural Network

The last part of this project will cover the implementation of a 2 layer (some call this a 3-layer network) neural network. There will be 3 input nodes at the very first layer of the network and two other neurons at the center of the network (called hidden layer) and finally, the last single neuron to generate the output. I'll be using sigmoid on the dot product of each layer input and its weights, to find the output of that layer. The error is calculated at the end of the neural network but we have to back propagate the network to update the weights on each single connection in the network. The method of backpropagation needs a deep understanding of the **chain rule** in mathematics and derivatives.

Artificial neural network explained and coded

Here is the core logic of the neural network. We begin by finding the output vector of each layer. Then we have to find the delta terms with respect to the above (right-hand) errors. Finally, we pass the errors from the top (right-hand) layer to the right to perform the backpropagation and update the weights according to what's described here.

```
def trainNeuralNetwork(dataset, flw, slw, numOfIterations):
    updatedFLW = 0
    updatedSLW = 0

    for i in range(numOfIterations):
        firstLayerOutputVector = sigmoid(dot(dataset, flw))
        secondLayerOutputVector = sigmoid(dot(firstLayerOutputVector, slw))

        # Find the error and delta for the final layer
        secondLayerError = secondLayerOutputVector - dataset_labels
        secondLayerDelta = secondLayerError *
sigmoidCurve(secondLayerOutputVector)

        # Find the error and delta for the first layer
        firstLayerError = secondLayerDelta.dot(slw.T)
        firstLayerDelta = firstLayerError * sigmoidCurve(firstLayerOutputVector)

        # Update the layer 1 and layer 2 weights
        flw -= secondLayerOutputVector.T.dot(firstLayerDelta)
        slw -= firstLayerOutputVector.T.dot(secondLayerDelta)

        updatedFLW = flw
        updatedSLW = slw
    return [flw, slw]
```

Artificial neural network results and outputs

The final result will be showing the updated weights for each single connection in the network.

```
Layer 1 weights vector was updated to:
                                [
                                [-0.07681709 -0.45953548]
                                [-0.15554917 -0.1700196 ]
                                [-0.24401318 -0.07086071]
                                                        ]

Layer 2 weights vector was updated to:
                                    [
                                    [-5.02168506]
                                    [ 5.95157547]
                                                ]
```