DSCI6001 – MATH FOR DATA SCIENTISTS
FINAL PROJECT REPORT
On

"Linear regression analysis & Prediction using Artificial Neural Network"

Submitted By
HEMENDRA JAMPALA (00695281)

Under The Guidance of
MINKYU KIM (Assistant Professor)

UNIVERSITY OF NEW HAVEN
Tagliatela College of Engineering
Department of Data Science
300 Boston Post Rd, West Haven, CT 06516

2. Introduction:-

This report is created for explaining about Implementation of Linear Regression and Artificial Neural Network from scratch without using scikit-learn.

First we will be briefly discussing about theory, explanation of Implementation, Approach, Data and Analysis results of Linear Regression Analysis

Then we will be briefly discussing about theory, explanation of Implementation, Approach, Data and Analysis results of Prediction using Artificial Neural Network.

Then I have written conclusion for this project.

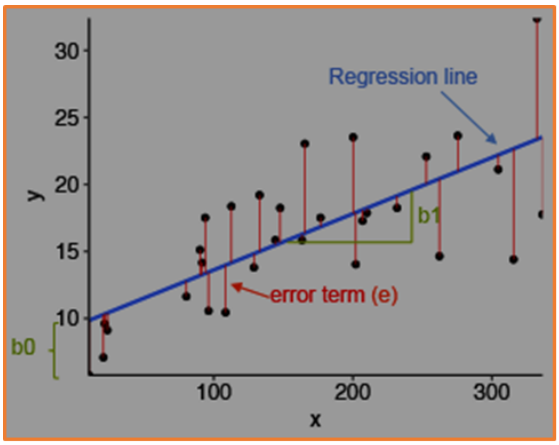
3. Linear Regression Analysis

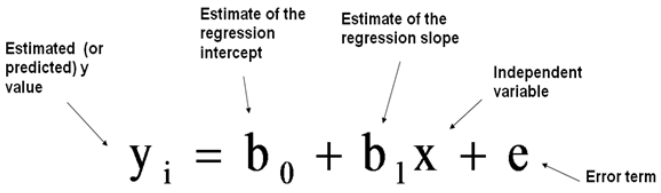
3.1 Linear Regression:-

Linear Regression is a supervised learning algorithm which is both a statistical and a machine learning algorithm. It is used to predict the real-valued output y based on the given input value x. Linear Regression is one of the most basic ways we can model relationships. It depicts the relationship between the dependent variable y and the independent variables xi (or features).

Our model here can be described as y=mx+b, where m is the slope (to change the steepness), b is the bias (to move the line up and down the graph), x is the explanatory variable, and y is the output. We use linear regression if we think there's a linear relationship. The hypothetical function used for prediction is represented by h(x).

For linear regressions we use a cost function known as the mean squared error or MSE. We take the squared value of our real data points minus the approximated values. Our approximated values can be calculated using the current m and b values we have: y_approx = m_current*x + b_current. After that, we add all those values up and divide them by the number of data points we have, effectively just taking the average.





3.2 Brief Explanation of Implementation:-

- Let's try applying gradient descent to m and b and approach it step by step:
- Initially let m = 0 and b = 0. Let L be our learning rate. This controls how much the value of m changes with each step. Learning rate L could be a small value like 0.0001 for good accuracy.
- Calculate the partial derivative of the loss function with respect to m, and plug in the current values of x, y, m and b in it to obtain the derivative value.
- Now we update the current value of m and b.
- We repeat this process until our loss function is a very small value or ideally 0 (which means 0 error or 100% accuracy). The value of m and b that we are left with now will be the optimum values.
- Now with the optimum value of **m** and **b** our model is ready to make predictions.

3.3 Analysis of Implementation Approach:-

As mentioned in above steps, we will be calculating hypothesis, loss, cost and gradient, we will update the m,b values and repeat the process till we get optimal condition or till best fit for our model. Then we predict with the help of optimal parameters.

```
[6]: def gradient_descent_1(a,alpha, X, Y, numIterations):
         m = X.shape[0]
         theta = np.ones(a)
         X_transpose = X.transpose()
         update_J =[]
         for iter in range(0, numIterations):
             hypothesis = np.dot(X, theta)
             loss = hypothesis - y
             J = np.sum(loss ** 2) / (2 * m)  # cost
             #print("Cost for iter %s is %s"%(iter,J))
             update_J.append(J)
             gradient = np.dot(X_transpose, loss) / m
             theta = theta - alpha * gradient  # update
         pylab.plot(update_J)
         pylab.show()
         return theta
```

3.4 Check Data:-

```
Xdf = pd.read_csv(r'C:\Users\madhuyen\Desktop\HEMENDRA J\Fall_2020_UNH\DSCI 6001\Final Project\F20_M4DS_project_LR_X.csv',header=
Y = pd.read_csv(r'C:\Users\madhuyen\Desktop\HEMENDRA J\Fall_2020_UNH\DSCI 6001\Final Project\F20_M4DS_project_LR_Y.csv',header=No

print("X Shape:",Xdf.shape)
print("Y Shape:",Y.shape)
```

```
X Shape: (34, 498)
Y Shape: (34, 3)
```

We have redundant columns in input dataset, we will be dropping all other columns as shown below.

```
In [4]: X= Xdf[[0,1,2,3,4]]
        X.head()
```

Out[4]:

	0	1	2	3	4
0	-0.099362	-0.085577	-0.143910	-0.104020	0.29407
1	-0.166140	0.277080	-0.190330	-0.056387	-0.13422
2	-0.252780	-0.085391	-0.096496	0.422420	-0.27044

The output Y dataset as shown below

```
In [5]: Y.head()
```

Out[5]:

	0	1	2
0	2.72	3.02	31.52
1	2.90	1.98	21.73
2	3.63	2.84	29.20

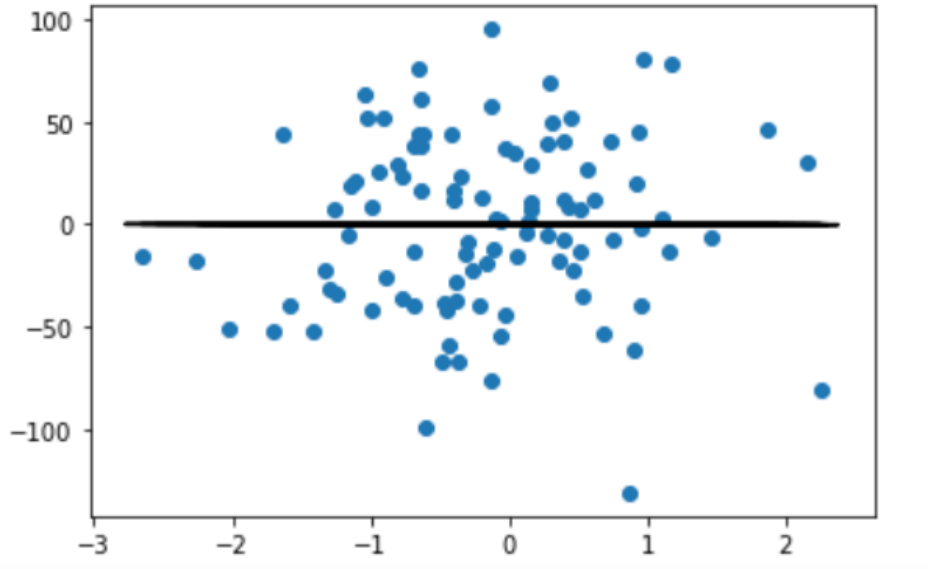
3.5 Analysis Results:-

We predict the optimal values for bj as follows.

```
    #Predictions
    for i in range(X.shape[1]):
        y_predict = theta[0] + theta[1]*X
        print(y_predict)
    pylab.plot(X[:,1],y,'o')
    pylab.plot(X,y_predict,'k-')
    pylab.show()
```

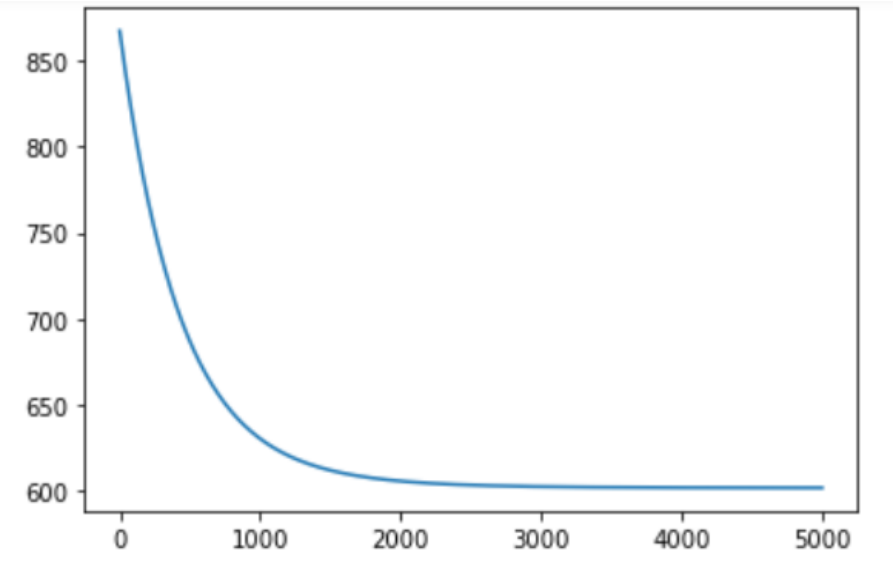
```
[[-0.23390158 -0.11147676 -0.24654032 -0.19118352 -0.21048774]
 [-0.23390158 -0.0524908  -0.29762002 -0.20359796 -0.1670938 ]
 [-0.23390158 -0.24301985 -0.23866739 -0.15554665 -0.07372128]
 [-0.23390158 -0.18341173 -0.18794145 -0.08181449 -0.17783029]
```

Predictions Plot:-



Cost function Plot:-

As shown below after 2000 epochs we get optimal values with low cost function.



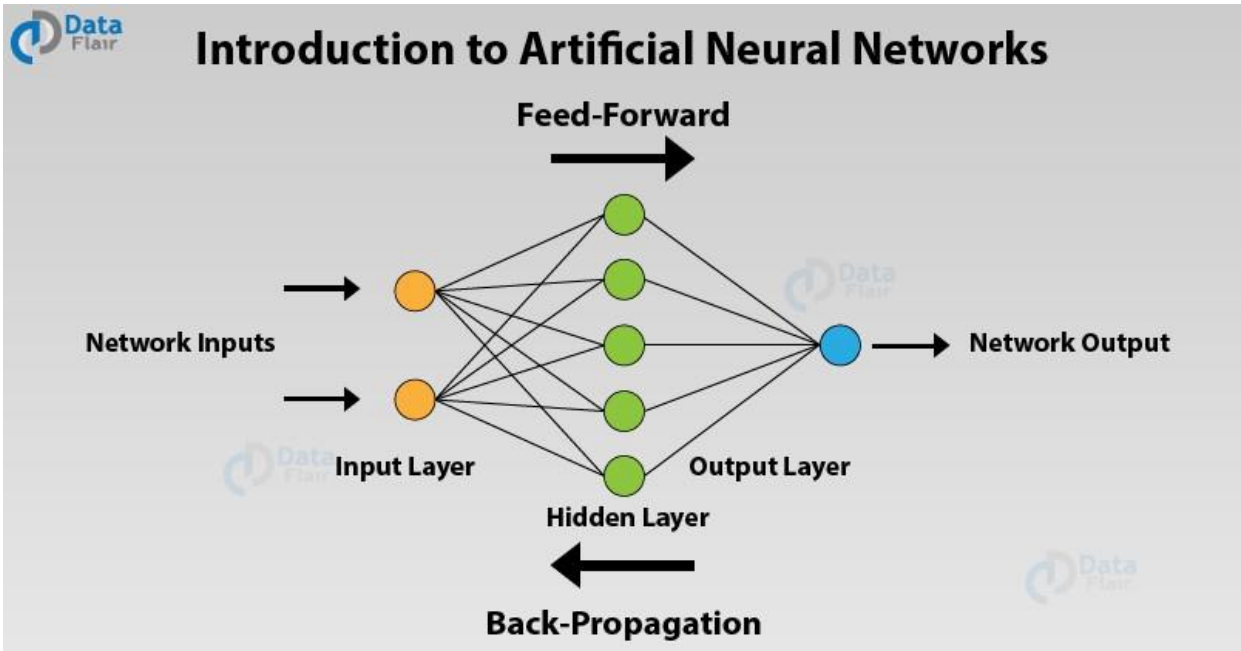
4. Prediction using Artificial Neural Network

4.1 Artificial Neural Network:-

Artificial Neural Networks (ANN) is a supervised learning system built of a large number of simple elements, called neurons or perceptrons. Each neuron can make simple decisions, and feeds those decisions to other neurons, organized in interconnected layers. Together, the neural network can emulate almost any function, and answer practically any question, given enough training samples and computing power.

A “shallow” neural network has only three layers of neurons:

- An input layer that accepts the independent variables or inputs of the model
- Hidden layer
- An output layer that generates predictions



Neuron/perceptron:-

The basic unit of the neural network. Accepts an input and generates a prediction.

Forwardpropagation:-

The forward propagation takes the inputs, passes them through the network and allows each neuron to react to a fraction of the input. Neurons generate their outputs and pass them on to the next layer, until eventually the network generates an output.

Backpropagation:-

In order to discover the optimal weights for the neurons, we perform a backward pass, moving back from the network's prediction to the neurons that generated that prediction. This is called backpropagation.

4.2 Brief Explanation of Implementation:-

We need to do the below steps to build our ANN model.

We need to do the below steps to build our model.

1. Define Network structure
   Number of input units, Number of hidden units, Number of output units.

2. Initialize the model's parameters.

3. Perform the below steps in loop until we get minimum cost/optimal parameters.

- Implement forward propagation
- Compute loss
- Implement backward propagation to get the gradients
- Update parameters

4.3 Analysis of Implementation Approach:-

As mentioned above, we will be implementing each step to achieve the model results.

1. Define Network Structure:-
   I have considered 2 inputs X[0],X[1], 3 hidden layers and 1 output layer Y for this model.

```
In [3]: #Assign size

        n_x = X.shape[1] # size of input layer`
        n_h = 3 #Size of hidden layers
        n_y = Y.shape[1] # size of output layer
        m =len(Y)
        print(n_x,n_h,n_y)
```

```
2 3 1
```

   By choosing more nodes in hidden layer, we can make model learn complex functions. But it comes at a cost of heavy computation to make predictions and learn the network

parameters. More number of hidden layers and nodes could also lead to over-fitting of our data.

2. Initialize the model's parameters:- W1,W2, b1,b2

W1 (weight matrix for hidden layer)
W2 (weight matrix for output layer)
b1, b2 baises are initialized to zeros.

W1, W2 parameters are initialized randomly using the numpy random function. Multiplied by 0.01 as we do not want the initial weights to be large, because it will lead to slower learning.

```
In [4]: W1 = np.random.randn(n_x,n_h) * 0.01
        b1 = np.zeros(shape=(1,n_h))
        W2 = np.random.randn(n_h,n_y) * 0.01
        b2 = np.zeros(shape=(1,n_y))

        print(W1.shape,b1.shape,W2.shape,b2.shape)
```

```
(2, 3) (1, 3) (3, 1) (1, 1)
```

3. Forward Propagation:-

During forward propagation the input features are fed to the every neuron in the hidden layer.
Which will be multiplied by the initial set of weights W1and bias (b1) will be added to form Z1.
Then we apply the non-linearity sigmoid function to Z1.
The output of this activation function/hidden layer will be A1.
For the final output layer, we multiply the inputs from the previous layer (A1) with the initial weights of output layer (W2), add bias (b2) to form Z2.
Then apply the sigmoid activation function on Z2 to produce out final output A2 (which is our predictions).

This is how ANN model makes predictions during forward propagation, which is just a sequence of multiplications and application of activation functions.

```
In [6]: def forwardPropogation(W1,b1,W2,b2):
            Z1 = X.dot(W1) + b1
            A1 = sigmoid_numpy(Z1)
            Z2 = A1.dot(W2) + b2
            A2 = sigmoid_numpy(Z2)
            return Z1,A1,Z2,A2
```

4. Compute Loss/Cost:-

Now that we have our predictions, next step would be to check how much our predictions differ from the actual values, loss/error.

```
Cost in Iteration 97 is 0    0.360048
dtype: float64
Cost in Iteration 98 is 0    0.35993
dtype: float64
Cost in Iteration 99 is 0    0.359818
dtype: float64
Optimized parameters after 100 iterations:
```

5. Back Propagation:-

Back propagation is used to calculate the gradients (slope/derivatives) of the loss function with respect to the model parameters (w1, b1, w2, b2). To minimize our cost we use the Gradient Descent algorithm, which uses the computed gradients to update the parameters so that our cost keeps reducing over iterations, i.e it help move towards global minimum.

In [7]:
```
def backPropagation(Y,Z1,A1,Z2,A2,W2):
    dA2 = A2-Y
    dZ2 = dA2* A2* (1-A2)
    dW2 = A1.T.dot(dZ2)
    db2 = np.sum(dZ2,axis=0)

    dA1 = dZ2.dot(W2.T)
    dZ1 = dA1*A1*(1-A1)
    dW1 = X.T.dot(dZ1)
    db1 = np.sum(dZ1,axis=0)

    db1 = db1.values.reshape(1,n_h)
    db2 = db2.values.reshape(1,n_y)

    return dW1,dW2,db1,db2
```

6. Update the parameters:-
   a. Multiply the gradients by learning rate
   b. Subtract from weights

Once we have computed our gradients, we multiply them with learning-rate and subtract from the initial parameters to get the updated parameters (weights and biases). Learning rate should be minimal so that we will not miss the global minimum point.

```
[8]: def updateParams(learning_rate,m,W1,W2,b1,b2,dW1,dW2,db1,db2):
         W1 = W1 - (learning_rate/m * dW1)
         W2 = W2 - (learning_rate/m * dW2)
         b1 = b1 - (learning_rate/m * db1)
         b2 = b2 - (learning_rate/m * db2)
         return W1,W2,b1,b2
```

Now we have performed one round of forward propagation and backward propagation, i.e. we completed 1 epoch. We need to repeat these steps over multiple epochs.

```
In [9]: def gradientdescentA():

            n_x = X.shape[1] # size of input layer`
            n_h = 3
            n_y = Y.shape[1] # size of output layer
            m =len(Y)
            #print(n_x,n_h,n_y)

            W1 = np.random.randn(n_x,n_h) * 0.01
            b1 = np.zeros(shape=(1,n_h))
            W2 = np.random.randn(n_h,n_y) * 0.01
            b2 = np.zeros(shape=(1,n_y))

            learning_rate = 10
            for i in range(100):
                Z1,A1,Z2,A2 = forwardPropogation(W1,b1,W2,b2)
                dW1,dW2,db1,db2 = backPropagation(Y,Z1,A1,Z2,A2,W2)
                W1,W2,b1,b2 = updateParams(learning_rate,m,W1,W2,b1,b2,dW1,dW2,db1,db2)

            print("Optimized parameters after 100 iterations:")
            print("\nValues of W1:\n",W1)
            print("\nValues of b1:\n",b1)
            print("\nValues of W2:\n",W2)
            print("\nValues of b2:\n",b2)
            return forwardPropogation(W1,b1,W2,b2)
```

4.4 Check Data:-

We have 2 inputs and 1 output datasets, using pandas data frame loaded the data as shown below.

```
In [2]: X = pd.read_csv(r'C:\Users\madhuyen\Desktop\HEMENDRA J\Fall_2020_UNH\DSCI 6001\Final Project\F20_M4DS_project_ANN_X.csv',header=N
        Y = pd.read_csv(r'C:\Users\madhuyen\Desktop\HEMENDRA J\Fall_2020_UNH\DSCI 6001\Final Project\F20_M4DS_project_ANN_Y.csv',header=N

        print("X Shape:",X.shape)
        print("Y Shape:",Y.shape)

        X Shape: (201, 2)
        Y Shape: (201, 1)
```

4.5 Analysis Results:-

After executing above steps, we get optimal parameters to predict the output layer. The optimal parameters and predicted values as follows.

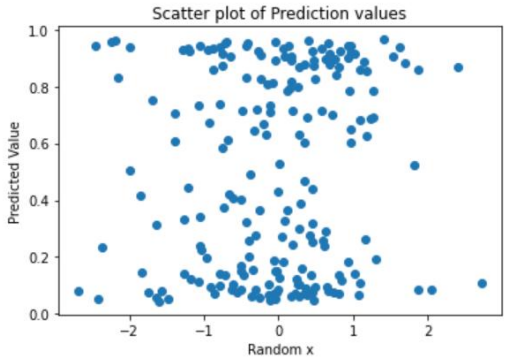
```
#A2 gives the predicted output values, which is the 4th parameter returned by forwardPropogation.
predictions = gradientdescentA()
print("\nA2 gives the predicted output values, which is the 4th parameter returned by forwardPropogation.\n",predictions[3])
```

```
Optimized parameters after 100 iterations:

Values of W1:
          0         1         2
0  0.658162 -0.670411 -0.623095
1 -2.151537  1.886112  1.539287

Values of b1:
 [[ 0.13080525 -0.07872994 -0.06656106]]

Values of W2:
          0
0  2.979438
1 -2.225528
2 -1.730089

Values of b2:
 [[0.61165337]]

A2 gives the predicted output values, which is the 4th parameter returned by forwardPropogation.
            0
0    0.086366
1    0.417740
2    0.959120
3    0.314774
4    0.733372
```

Activate Windows
Go to Settings to activate Win

Plotted predicted values as shown below

```
In [11]: import matplotlib.pyplot as plt
         x = np.random.randn(201,1)

         plt.scatter(x,predictions[3])
         plt.title("Scatter plot of Prediction values")
         plt.xlabel('Random x')
         plt.ylabel('Predicted Value')
         plt.show()
```



5. Conclusions:-

I have learnt a lot of coding my own Linear Regression and Neural Network from scratch.

Although Machine Learning library Scikit-Learn for Linear Regression and Deep Learning libraries such as Tensor Flow and Keras makes it easy to build deep nets without fully understanding the inner workings of a Neural Network, I find that it's beneficial for aspiring data scientist to gain a deeper understanding of Neural Networks.