EAI 320

Lefa Raleting

April 2020

1 Introduction

Neural networks is a sub division of artificial intelligence, it was first introduced in 1943, but the first real world application came in 1959, for Multiple Adaptive Linear elements(reference 1).

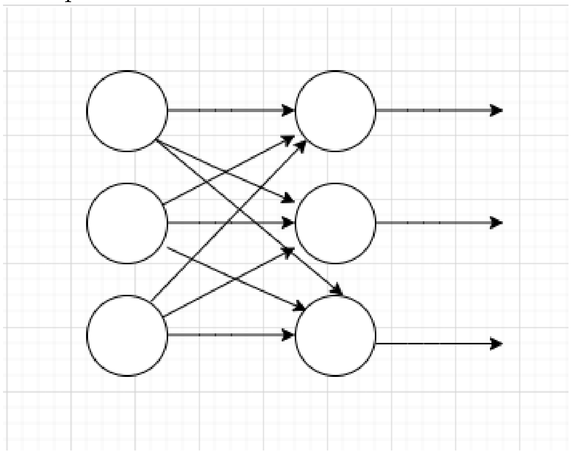
From there on, Neural networks have been widely used for different applications. For this assignment a task was given to develop an Artificial Neural Network(ANN) with back-propagation. This report illustrates the different test cases, results and discussion of the findings.

2 Background

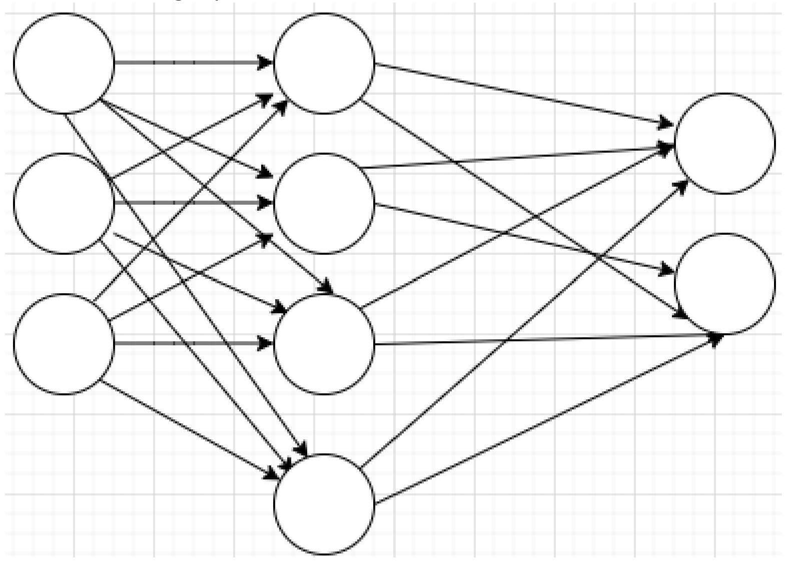
2.1 Artificial Neural Networks

What are artificial neural networks?
Artificial neural networks is the mimicking of the brains, and nervous system. For instance how the brain process images and learns to identify things much better. On example of this would be how the brain is able to identify different hand writings.

Types of ANN?

1. Feed-forward Neural Network
   The data moves in one direction from the first hidden layer till the output. With this type of network there is no back-propagation. Bellow is an example of a feed-forward network
   
2. Radial Basis Function Neural Network
   A radial basis neural network is one which its activation functions uses radial basis functions as activation functions.

3. Multi-layer Perceptron
   A Multilayer Perceptron also known as a fully connected neural network is one which each neuron in the current node is connected to every node in the following layer. It uses a non- linear activation function



Rules of thumb

1. Begin with two hidden layers which don't include the input and output layer
2. For faster training, the number of nodes in the intermediate layers must be half of the previous layers.
3. For a multi- class classifier the number of output nodes is the number of classes.
4. Use Relu for intermediate activation functions
5. For the output layer use softmax for multiclass classifier problem and linear function for regression problem
6. Plot loss vs Epoch

2.2 Bias

What is a Bais and why is it important?

Bias is a constant that helps the data to best fit the model. The reason it is important is that it helps control the activation function, by shifting the intercept where the neuron begins to activate.

How to select a Bias value?

Usually a bias of 1 is selected. A high bias may result in a training error and low bias results in a low training error.

2.3 Learning Rate

What is the learning rate?

Objective during training is to minimise the losses. You start of the learning process with arbitrary weights. You then update this weights to get better results which lead to less losses. The rate at which you update the weights is called the learning rate.

2.4 Activation function

What is the activation function?

Activation function sums up the product of the input and the weight, adds a bias and based on those values it decides where there a neuron should be activated or not. You use it to filter out unlikely options. The aim of the activation function is to introduce non linearity in the learning of complex patterns (Ravichandiran, 2019)

Types of activation functions

1. Linear

$$f(x) = mx \tag{1}$$

2. Step

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & 0 \leq x \end{cases}$$

3. Sigmoid The sigmoid function keeps the value between 0 and 1, this function is differentiable

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

   There are two problem's when it comes to selecting Sigmoid, namely

   (a) Vanishing and exploding gradients
   For instances when a neuron saturates towards zero or 1, the gradient is close to zero. This becomes a problem during Back propagation, This local gradient will be 3 multiplied by the gradient of the gates output, this will in turn make the gate disappear as no signal will flow through the neuron to its weights and its data.

   (b) Output is not zero centred The value is always positive king the gradient of the weights all positive or all negative. This causes over stepping, which make optimization harder.

4. Rectified linear unit (ReLu) This is one of the most commonly used activation function. It has an output of 0 when the input is less than zero and a output of the input when the input is greater than or equal to zero. Since the fact that when the input is negative you have and output of zero, you face a problem called the dying Relu (Ravichandiran, 2019).

$$f(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leq x \end{cases}$$

5. Leaky ReLu Leaky Relu solves the problem with the dying zero by introducing a small gradient for negative values. The function then becomes as bellow:

$$f(x) = \begin{cases} ax & x < 0 \\ x & 0 \leq x \end{cases}$$

6. Softmax It is usually applied to the final layer in multi-class classification problem. It gives the probabilities of each class therefore the probabilities will always equal 1.

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{3}$$

2.5 Propagation

Forward propagation

In this process inputs are propagated from the input layer to the output layer. The values are multiplied by their weights then an activation function is applied.

Weight matrix

It is not known at start which input is more important therefore the weights need to be randomized. The Dimension of the weight matrix must be number of neurons in the current layer by the number of layers in the next layer. This is due to simple matrix multiplication, to multiply matrix's AH then the number of columns in matrix A need to equal the number rows in matrix H.

Loss function

It is of vital importance to be able to check how well the algorithm is working and this is achieved by a loss function. There are many loss function and one example of such is the mean squared error function. Which compare the predicted output to the expected output.

$$L = \frac{1}{n} \sum (y_i - p_i)^2 \tag{4}$$

Where n is the size of the training sample and y is the actual output and p is the predicted output.

What is back-propagation?

Back propagation is uses a learning scheme for the artificial neural network(ANN) . Since the loss function gives us an indication of how well the ANN is performing. The weights and bias need to adjusted to give us the lowest loss possible. One way of achieving this is by the use of gradient decent. This one of many optimization algorithms.

2.6 Weights

After the error has been propagated, the propagated errors are then used to update the weights.

$$NewWeight = weight + LearningRate * Error * Input \tag{5}$$

where inputs references to the input causing the error. This process is repeated fall all the network weights.

3 Methodology

3.1 Input Layer

For the input layer, the ANN takes the previous two moves of both the agent and opponent and then encodes to a decimal number that's between 0 and 0.80. The ANN uses a single neuron input layer.

3.2 Hidden layers

3.2.1 Activation function

Leaky Relu solves the problem with the dying zero by introducing a small gradient for negative values. The function then becomes as bellow:

$$f(x) = \begin{cases} ax & x < 0 \\ x & 0 \leq x \end{cases}$$

This helped when the model was being trained. It enabled the network to function with all the neurons.

3.3 Output layer

Activation Function

For the activation function on the output layer it is important to implement a function which has an output between 0 and 1, Where one denotes a switched on neuron and 0 denotes an off neuron.

At the output neurons a modified soft-max function was utilized instead of the default soft max function, It was found that for high weights, soft-max caused computational errors.

$$f(x) = \begin{cases} ax & x < 0 \\ x & 0 \leq x \end{cases}$$

Representation

Output level has three neurons, of which each represent a move, "R","P","S". The neuron with the highest value is taken as the move to be played.

3.4 Back-propagation

The agents starts of untrained and the method in which is used to train it is called back propagation. For a successful propagation, it is important to firstly insure that the activation function that you are using are none linear, this is done to avoid one of the cases that lead to saturation during back-propagation.

Gradient decent
For back-propagation I utilised a method called gradient decent. What this does is that the takes gradual steps in a particular direction until it find local or global minimum. The step size is known as the learning rate. However there is a trade-off. A high learning rate causes, overstep and leads to saturation. Whereas a low learning rate causes an underestimate. There is not a general perfect learning rate, it depends from network to network.

Error Propagation For the output layer, error is calculated as the difference between the output neuron and the target neuron.

$$Errorx = zx–tx \tag{6}$$

For the links, the error is calculate slightly different. The error is equal to:

$$error = linkweights * Derivativeofactivationfunction * errorneruon \tag{7}$$

The error is propagated from one layer to another until it reaches the input weights.

3.5 Training

For Training There are several inputs that need to be considered, Learning-rate, which has been discussed above, Epoch, error. Training is where you train your ANN(Artificial Neural Network) to predict better.

While the training time could decrease significantly by grouping the training data, this would produce over-fitting of data, therefore alternative methods needed to be considered. By choosing the data structures properly , the time complexity can be reduced significantly. Pandas were considered however the advantage they hold over list of tuples cannot be taken advantage in this application and what is gain in implement using pandas would be lost in the cost of the implementations. List of tuples however has the same time complexity for access time and has a lower implementation cost. There they were used to store the training data.

For easy manipulation of data, matrix's are used. Therefore most of the data such as weights and neuron values would be in matrix form. To convert string values to numerical values, dictionaries are used to encode the data. All of these operations have a time complexity of O(1)

One training epoch has a time complexity of O(n) , each epoch introduces a constant value m, giving the training algorithm time complexity of O(m*n). where m is the number of epochs.

Definitions

Epoch- This term tends to be confusing however in simple terms it means how many times the training set is processed. In the discussion we discuss the impact of this variable to detail.

Batches- Sometimes when a training set is to large, it is split into batches, let's say for instances you had a data set of 1000, and that's too big for you to process. Let's say your computer can handle 200 data sets at a time, which is a batch size of 200, the number of batches that would need to be processed would be 5 bathes per Epoch

Error- This is the term you use as your stop condition, error convergences, that is when your error is less than your previous error the train must stop, or when error is less than a specified cut off arbitrary value.

4 Results and Discussion

4.1 Questions

How does the size of the hidden layers affect the performance of the ANN?

The number of neurons in the hidden layers and number of hidden layers affect the process drastically. When it comes to neurons, to many neurons, with little data points can cause your neurons to not be properly trained and give bad results. Having to little neurons can end up leading bad approximations. The number of layers relate to complexity too, increasing the number of layer may increase the time complexity of the training time. To little layer leads to bad data approximation.

How is the input data represented

For the input layer, the ANN takes the previous two moves of both the agent and opponent and then encodes to a decimal number that's between 0 and 0.80. The ANN uses a single neuron input layer. It is represented by a 1X1 matrix.

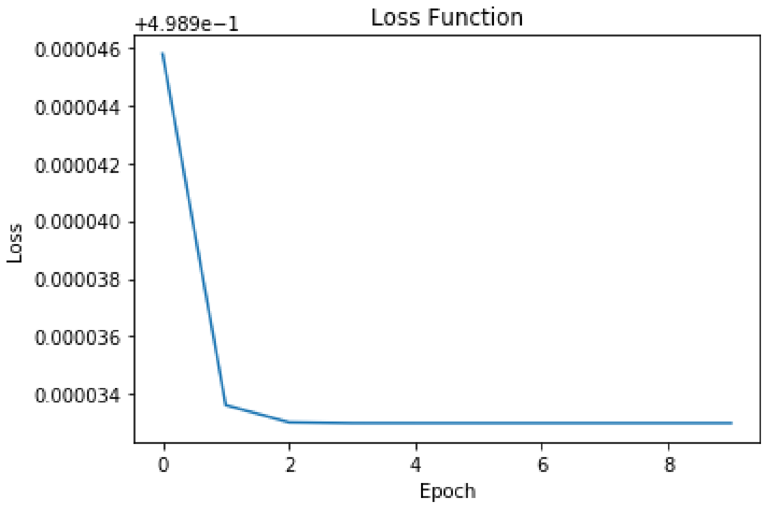
How is the output data represented

Refer to section 3.3

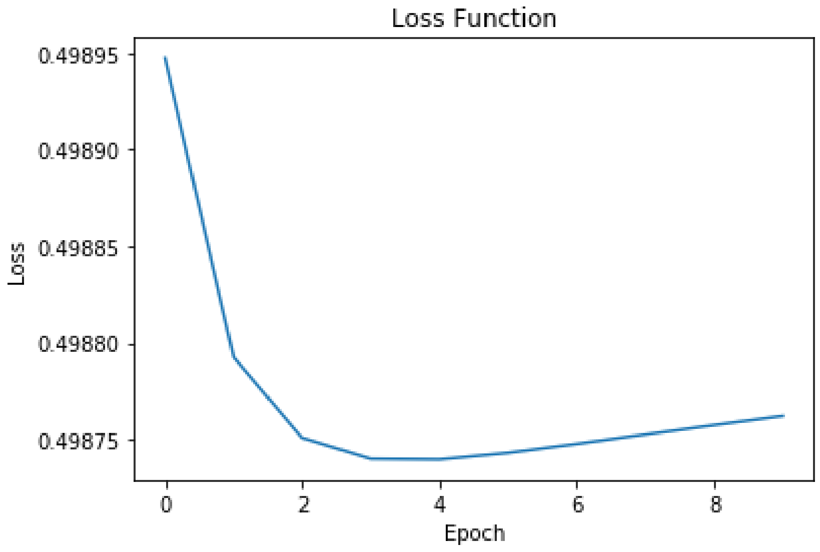
How the learning rate affects the training process

The convergence is manly steered by the error, therefore your learning rate cannot be higher than your error range, and since this error range is between [0,01,1], the learning rate should then be less than 0.01 to avoid saturation. Bellow we have a comparison of the different learning rates

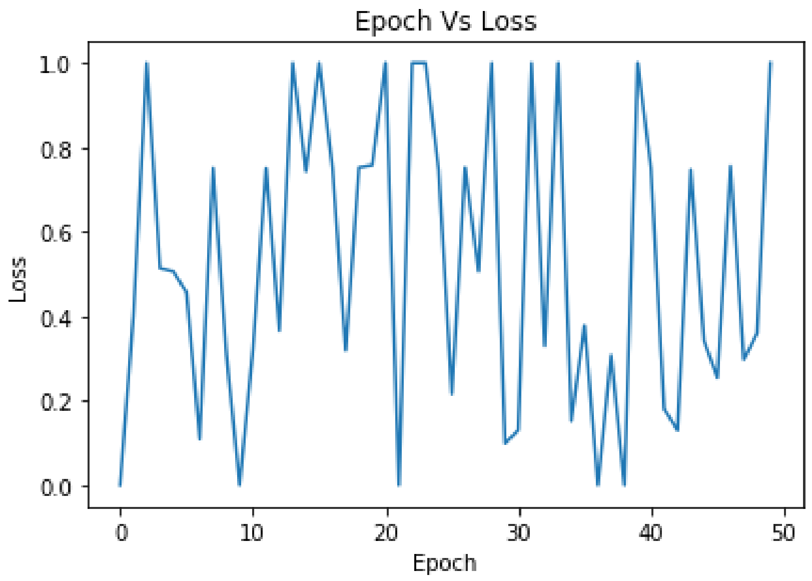
1. Learning rate =0.05



2. Learning rate= 0.005



3. Learning rate =0.25



How did the loss increase or decrease during training

The loss of the ANN decreases or oscillates depending on the learning rate and the number of epochs. It can be seen as the number of epochs increase the better the ANN trains and the less the loss. If the correct Learning rate is used it can be seen in 4.2.1 that with the increase in epoch there is decrease in loss, to a point where the error does not significantly change.

What is the importance of the activation function

Refer to section 2.4

Epoch VS loss

The higher the epoch the better trained the neural networks becomes and the less the error or loss you observe.

5 Conclusion

For a good performing neural network, You need to ensure that your hidden layers are less or equal to your input and your outputs, to stimulate better learning. High learning rate Causes an overshot and low learning rate caused an underestimate. More Epoch increase accuracy, and if there is too much data split it into batches

6 References

1. Raleting,LR.,2019. Practical 4: Artificial Neural Network

2. Davydova, O., 2017. 7 Types Of Artificial Neural Networks For Natural Language Processing. [online] Medium. Available at: ¡https://medium.com/@datamonsters/artificial-neural-networks-for-natural-language-processing-part-1-64ca9ebfa3b2¿ [Accessed 9 April 2020].

3. Ravichandiran, S. (2019) Hands-on deep learning algorithms with python : master deep learning algorithms with extensive math by implementing them using tensorflow. Birmingham: Packt Publishing. Available at: INSERT-MISSING-URL (Accessed: April 13, 2020).

7 Appendix

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 13 13:39:46 2020

@author: Boris
"""

import numpy as np
import matplotlib.pyplot as plt
import time
import csv


import pandas as pd


#This part of the code. opens up a specified csv file, then adds it to a turple

with open('data1.csv', 'r') as csvFile:
    reader = csv.reader(csvFile)
    line= [tuple(row) for row in reader]
csvFile.close()


database=line

#function to split datainto chunks

def chunks(l,n):
    for i in range(0,len(l),n):
        yield l[i:i+n]
#print(list(chunks(database,90))[0][89])
#This part of the code then goes on to create a data base of the the data collected
#counting ever instances and classifying it in a dictionary

#database=collections.Counter()
#database.update(line)




#these are my helper dictionaries to better help navigate my functions

indexmoves= {
           0:'RRRR', 1:'RRRP', 2:'RRRS', 3:'RRPR', 4:'RRPP', 5:'RRPS', 6:'RRSR',
           7:'RRSP', 8:'RRSS', 9: 'RPRR', 10:'RPRP',11:'RPRS', 12:'RPPR', 13:'RPPP',
           14:'RPPS',15:'RPSR',16:'RPSP',17:'RPSS', 18:'RSRR', 19:'RSRP',20: 'RSRS',
           21: 'RSPR', 22: 'RSPP', 23: 'RSPS', 24: 'RSSR', 25: 'RSSP', 26: 'RSSS',
           27: 'PRRR', 28: 'PRRP', 29: 'PRRS', 30: 'PRPR', 31: 'PRPP', 32: 'PRPS',
           33: 'PRSR', 34: 'PRSP', 35: 'PRSS', 36: 'PPRR', 37: 'PPRP', 38: 'PPRS',
           39: 'PPPR', 40: 'PPPP', 41: 'PPPS', 42: 'PPSR', 43: 'PPSP', 44: 'PPSS',
           45: 'PSRR', 46: 'PSRP', 47: 'PSRS', 48: 'PSPR', 49: 'PSPP', 50: 'PSPS',
           51: 'PSSR', 52: 'PSSP', 53: 'PSSS', 54: 'SRRR', 55: 'SRRP', 56: 'SRRS',
           57: 'SRPR', 58: 'SRPP', 59: 'SRPS', 60: 'SRSR', 61: 'SRSP', 62: 'SRSS',
            63: 'SPRR', 64: 'SPRP', 65: 'SPRS', 66: 'SPPR', 67: 'SPPP', 68: 'SPPS',
            69: 'SPSR', 70: 'SPSP', 71: 'SPSS', 72: 'SSRR', 73: 'SSRP', 74: 'SSRS',
            75: 'SSPR', 76: 'SSPP', 77: 'SSPS', 78: 'SSSR', 79: 'SSSP', 80: 'SSSS'}
plays={0:'R',1:'P',2:'S'}
players={'R':0,'P':1,'S':2}
outputEncoder={'R':[1,0,0],'P':[0,1,0],'S':[0,0,1]}



movesIndex={
            "RRRR": 0,"RRRP": 1,"RRRS": 2,"RRPR": 3,"RRPP": 4,"RRPS": 5,"RRSR": 6,
            "RRSP": 7,"RRSS": 8,"RPRR":9,"RPRP": 10,"RPRS":11,"RPPR": 12,"RPPP": 13,
            "RPPS":14,"RPSR":15,"RPSP":16,"RPSS":17,"RSRR":18,"RSRP":19,"RSRS":20,
            "RSPR":21,"RSPP":22,"RSPS":23,"RSSR":24,"RSSP":25,"RSSS":26,"PRRR":27,
            "PRRP":28,"PRRS":29,"PRPR":30,"PRPP":31,"PRPS":32,"PRSR":33,"PRSP":34,
            "PRSS":35,"PPRR":36,"PPRP":37,"PPRS":38,"PPPR":39,"PPPP":40,"PPPS":41,
            "PPSR":42,"PPSP":43,"PPSS":44,"PSRR":45,"PSRP":46,"PSRS":47,"PSPR":48,
            "PSPP":49,"PSPS":50,"PSSR":51,"PSSP":52,"PSSS":53,"SRRR":54,"SRRP":55,
            "SRRS":56,"SRPR":57,"SRPP":58,"SRPS":59,"SRSR":60,"SRSP":61,"SRSS":62,
            "SPRR":63,"SPRP":64,"SPRS":65,"SPPR":66,"SPPP":67,"SPPS":68,"SPSR":69,
            "SPSP":70,"SPSS":71,"SSRR":72,"SSRP":73,"SSRS":74,"SSPR":75,"SSPP":76,
            "SSPS":77,"SSSR":78,"SSSP":79,"SSSS":80
            }


encoder={
            "RRRR": 0.0,"RRRP": 0.01,"RRRS": 0.02,"RRPR": 0.03,"RRPP": 0.04,"RRPS": 0.05,
            "RRSR": 0.06,"RRSP": 0.07,"RRSS": 0.08,"RPRR": 0.09,"RPRP": 0.10,"RPRS":0.11,
            "RPPR": 0.12,"RPPP": 0.13, "RPPS":0.14,"RPSR": 0.15,"RPSP": 0.16,"RPSS":0.17,
            "RSRR": 0.18,"RSRP": 0.19,"RSRS": 0.20,"RSPR": 0.21,"RSPP": 0.22,"RSPS":0.23,
            "RSSR": 0.24,"RSSP": 0.25,"RSSS": 0.26,"PRRR": 0.27,"PRRP": 0.28,"PRRS":0.29,
            "PRPR": 0.30,"PRPP": 0.31,"PRPS": 0.32,"PRSR": 0.33,"PRSP": 0.34, "PRSS":0.35,
            "PPRR": 0.36,"PPRP": 0.37,"PPRS": 0.38,"PPPR": 0.39,"PPPP": 0.40,"PPPS":0.41,
            "PPSR": 0.42,"PPSP": 0.43,"PPSS": 0.44,"PSRR": 0.45,"PSRP": 0.46,"PSRS":0.47,
            "PSPR": 0.48,"PSPP": 0.49,"PSPS": 0.50,"PSSR": 0.51,"PSSP": 0.52,"PSSS":0.53,
            "SRRR": 0.54,"SRRP": 0.55,"SRRS": 0.56,"SRPR": 0.57,"SRPP": 0.58,"SRPS":0.59,
            "SRSR": 0.60,"SRSP": 0.61,"SRSS": 0.62,"SPRR": 0.63,"SPRP": 0.64,"SPRS":0.65,
            "SPPR": 0.66,"SPPP": 0.67,"SPPS": 0.68,"SPSR": 0.69,"SPSP": 0.70,"SPSS":0.71,
            "SSRR": 0.72,"SSRP": 0.73,"SSRS": 0.74,"SSPR": 0.75,"SSPP": 0.76,"SSPS":0.77,
            "SSSR": 0.78,"SSSP": 0.79,"SSSS": 0.80
            }

#print([encoder[indexmoves[80]]])
#print(database[(indexmoves[60],plays[0])])



#_______________Global parameters_____________________________\

Input_number = 1 # This the number of nodes in the input layer
Hidden_layers = 2 #This is the number hidden layers
Hidden_number1= 3 #number of neurons in the first hidden layer
Hidden_number2= 3 #Number of neurons in the second hidden layer
Output_number= 3 #Number of nodes in output layer

#_________________Intializations_____________________________
```

```
#------------------------Matrix's--------------------------------

#Matrix for between input and hidden layer 1
Weight_IH=np.random.randn(Input_number,Hidden_number1)
#matrix for hidden layer 1
Bias_H1 = np.array([[0.65685711, 0.6665443,  0.45332548]])#np.random.rand(1,Hidden_number1)
#Matrix for between input and hidden layer 2
#Weight_HH= np.array([[-0.3016027 , -0.26154309,  0.18243307],[ 0.05816142, -0.86195668 , 0.47
Weight_HH= np.random.randn(Hidden_number1,Hidden_number2)
#matrix for hidden layer 2
Bias_H2= np.array([[0.3889918,  0.13289353, 0.1503658 ]])#np.random.rand(1,Hidden_number2)
#Weight Matrix for hidden layer 2 to output
#Weight_HO= np.array([[ 0.96110777, -0.10597898, -0.18091392],[-0.27495829 ,-0.66875004 , 0.16
Weight_HO=np.random.randn(Hidden_number2,Output_number)
#Bias matrix for output layer
Bias_O =np.array([[0.93448987, 0.47523067, 0.48086404]])#np.random.rand(1,Output_number)


#Activation Functions

#Leaky Relu
def Relu(x,a=0.1):
    #if x is less than zero
    #return the multiplication of the value and small gradient
    #else return the value as is
    x[x<0]=a*x[x<0]

    return x

def DerivativeRelu(x,a=0.1):
    x[x<=0]=a #let the diravitive equal to the slope which is a for x<=0
    x[x>0]=1 #slope equal 1 for x>0
    return x
#Soft max function
def Softmax(x):
#    return np.exp(x)/np.exp(x).sum(axis=0)
    return np.exp(x-np.max(x))/np.sum(np.exp(x-np.max(x)),axis=0)
#softmax derivative
def Softmaxderv(x):
     return (1-Softmax(x[0]))

def Foward_Propagation(Inputs,Weight_IH,Weight_HH,Weight_HO,a=0.1):
    S1= np.dot(Inputs,Weight_IH)+ Bias_H1
    A1= Relu(S1[0],a)
    S2= np.dot(A1,Weight_HH)+ Bias_H2
    A2= Relu(S2[0],a)
    Y=  np.dot(A2,Weight_HO)+ Bias_O
    Out= Softmax(Y[0])

    return S1,A1,S2,A2,Y,Out

#a,b,c,d,e,f=Foward_Propagation(dumpyinputs,Weight_IH,Weight_HH,Weight_HO)
#-------------------------TASK1---------------------------------------

#This function is for backpopagation and it works as explained in the report
def Back_Propagation(Out,S1,A1,S2,A2,Y,LearningRate,Target,inputs,a=0.1):
    global Weight_HO
    global Weight_HH
    global Weight_IH

    Alpha1= np.multiply(-(Target-Out),Softmaxderv(Y)) #3by3 matrix
    Link_H2O = np.dot(np.array([A2]).T, np.array([Alpha1])) #gradient calculation
    Alpha2= np.dot(Alpha1,Weight_HO.T)*DerivativeRelu(S2,a)
    Link_H1H2= np.dot(np.array([A1]).T,Alpha2)
    Alpha3=np.dot(Alpha2,Weight_HH.T)*DerivativeRelu(S1,a)
    Link_IH1=np.dot(np.array([inputs]).T,Alpha3)#

    Weight_HO=Weight_HO-LearningRate*Link_H2O
    Weight_HO=Weight_HO/np.max(Weight_HO)
    Weight_HH=Weight_HH-LearningRate*Link_H1H2
    Weight_HH=Weight_HH/np.max(Weight_HH)
    Weight_IH=Weight_IH-LearningRate*Link_IH1
    Weight_IH=Weight_IH/np.max(Weight_IH)

    return Weight_HO,Weight_HH,Weight_IH #Link_H2O,Link_H1H2,Link_IH1

#loss function to help corret the error in training
def Loss_function(EOut,Out):
    L=0.5*sum(((EOut-Out))**2)
    return L


gama=0.10
number_of_iterations=10 #Epoch
Loss=[]


def train(Inputs,Target,IH,HH,HO,LearningRate,Epoch):
   # DataBase_length= len(database)


    for i in range(Epoch):#[encoder[Inputs[k][0]]] outputEncoder[Target[k][1]]
        for k in Inputs: #range(0,1000000):#DataBase_length
            S1,A1,S2,A2,Y,Out=Foward_Propagation([encoder[k[0]]],IH,HH,HO,0.05)
            HO,HH,IH=Back_Propagation(Out,S1,A1,S2,A2,Y,LearningRate,outputEncoder[k[1]],[enco

        L=Loss_function(outputEncoder[k[1]],Out)
        Loss.append(L)

    plt.plot(range(Epoch),Loss)
    plt.title('Loss_Function')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')



    return HO,HH,IH,Loss




#timea=time.time()
#train(database,database,Weight_IH,Weight_HH,Weight_HO,gama,number_of_iterations)
#timeb=time.time()
```

```
#print("Time:", timeb-timea)
```