ECE 657

ASSIGNMENT 1

Question 1:

Restated proof for the perceptron Convergence theorem:

A perceptron, invented by Rosenblatt was inspired by the McCulloch-Pitts model and is like a single neuron that is helpful in the classification problem of machine learning. It can be explained as a function that is a combination of input and weights ultimately giving the output function. The perceptron model can identify inputs and can perform binary classification by putting the inputs in the right classes. The perceptron algorithm can run a loop till we can say the new assigned weight is not misclassified. The perceptron convergence theorem is based on the error correction rule where we can change its weights in iterations. It is more like finding a hyperplane for linearly separable data that separate two classes. It is mostly for high- dimensional data.

Let us assume the equation of hyperplane to be H= {x: $W^T x + b = 0$} and output(Y) can be either class 1 or 2. It is difficult to update W and b so we eliminate b by saying that there is no offset and the line is from the origin, but we add another dimension to our solution and now we have a two-dimensional hyperplane.

Previously the theorem was used for signals that can have either the response +1 or -1. Now, for the classification, if input belongs to class 1 then, $W^T x>0$ and $W^T x<=0$ if it belongs to class 2. The conclusion is that $YW^T x>0$ if we consider Y to be the output. This leads to the first equation that can be written as

$(\boldsymbol{wi}.\ \boldsymbol{y}) > \boldsymbol{\theta} > \mathbf{0}\ \boldsymbol{i}=\mathbf{1},.....,\boldsymbol{N}$ ----------(1)

We can remove $\boldsymbol{\theta}$ which is the threshold if there is no loss of generality. It is a positive number, and its limit should be increased denoting that the output of the perceptron is correct. If the first equation is true, that means the classification is correct.
Now according to the weight updating algorithm, there will be two cases either the data will be correctly classified, or it will be misclassified.

For the first case if it is correctly classified then we will keep the old weight vector. Let us consider it to be the initial vector that was separating the data into the two classes. In the case of correct classification, the new weight vector will be the same as the old weight vector and so

$\boldsymbol{vn}+1 = \boldsymbol{vn}\ \boldsymbol{if}\ (\boldsymbol{win}\ .\ \boldsymbol{vn}) > \boldsymbol{\theta}$ —----------(2)

This is the case when $W^T x>0$ and the class assigned to the input will be 1 or the case when $W^T x<= 0$ and it belongs to class 2. This can also be seen as if the W_{in} it's on the correct side of the hyperplane.

for the second case let's say that the data is misclassified, or the perceptron has abstained from devotion.
$\boldsymbol{vn}+1 = \boldsymbol{vn} + \boldsymbol{win}\ \boldsymbol{if}\ (\boldsymbol{win}.\ \boldsymbol{vn}) \leq \boldsymbol{\theta}$ ----------(3)

Here we take the old weight and add a weight vector value corresponding to the data point to correct misclassification. This is called error correction for the perceptron. These are the cases when $W^T x>0$ and the class assigned to the input will be class 2 or the case when $W^T x<=0$ and it belongs to class 1. This can also be seen when W_{in} will be on the wrong side of the hyperplane.

As we don't have to update weights for the correctly classified data, we will continue with the third equation we have to update the weights for the N iterations. We take N to be a finite value and here N= 1,2,3,4, 5.

The initial assumptions for the convergence to stabilize the weight correction procedure have some initial assumptions which are that the classes are linearly separable and we initially start with putting the weights to be zero vector.

Now if we have to define the value of the magnitude of $\boldsymbol{vn}+1$ in terms of the N looping variable we can write it as :

$\| \boldsymbol{vn}+1 \| = CN^2$ ----------(4)

We have to choose the constant C such that it is positive. We must choose a large value for N.

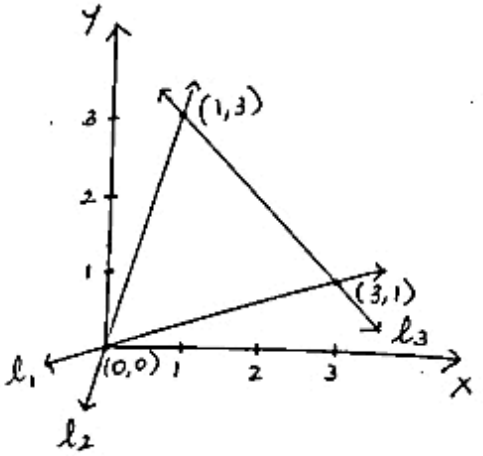
We define the Cauchy -Schwartz inequality such that if there are two vectors u and v such that $|(u,v)|^2 <= (u,u)\ .\ (v,v)$ where the operations are defined as the inner product
And so in a more convenient form the equation is written as $|(u,v)|^2 <= ||u||||v||$
Using the same on our hyperplane vector v,

$\| vn+1 \|^2 \leq \| vn+1 \|^2 + (2\theta + M)\ n$----------(5)
Where $M = \max \| vi \|^2$ where i = 1, ..., N ---------- (6)

Now if for a value of v_o where the data belongs to class one, we may choose $C = \boldsymbol{\theta}^2/\|y\|^2$ which makes equation 4 relevant and can be for all values of N.
Now if for a value of v_o where the data belongs to class two, we can choose $C= (1/4) (\boldsymbol{\theta}^2/\|y\|^2)$, but here we have a limitation that n is greater than $-2[(v_o, y)/ \boldsymbol{\theta}]$.

Here in the inequality $\| v_{k+1} \|^2 - \| v_k \|^2 = 2 (v_k , w_{ik}) + \| w_{ik}\|^2 \leq 2\theta + M$ if add all inequalities for all values of k from 1 to n, we can obtain the equation 5. But for this condition to be true the value of n should not be very large.

Problem 2



Equation of the lines

Equation of l_1 $(0,0)$ $(3,1)$
(x_1,y_1) (x_2,y_2)

$$y-y_1=\left[\frac{y_2-y_1}{x_2-x_1}\right](x-x_1)$$

$$y-0=\left[\frac{1-0}{3-0}\right](x-0)$$

$$y=\frac{1}{3}x$$

$$3y=x$$

Equation of l_2 $(0,0)$ $(1,3)$
(x_1,y_1) (x_2,y_2)

$$y-0=\left[\frac{3-0}{1-0}\right](x-0)$$

$$y=3x$$

Equation of l_3 $(1,3)$ $(3,1)$
(x_1,y_1) (x_2,y_2)

$$y-3=\left[\frac{1-3}{3-1}\right](x-1)$$

$$y-3=\left[\frac{-2}{2}\right](x-1)$$

$$y-3=-x+1$$

$$y=-x+4$$

Considering each perception as a linear seperator, representing one of the decision boundary each

$$x w_{13} + y w_{23} + b_1 = 0 \quad \text{---} (1)$$

$$x w_{14} + y w_{24} + b_2 = 0 \quad \text{---} (2)$$

$$x w_{15} + y w_{25} + b_3 = 0 \quad \text{---} (3)$$

Solving the equations (1), (2) & (3) for getting the weights

$$w_{13} + 3w_{23} = 0$$

$$\boxed{w_{13} = -3w_{23}}$$

$$3w_{14} + w_{24} = 0$$

$$\boxed{3w_{14} = -w_{24}}$$

$$w_{15} + 3w_{25} + b_3 = 0$$

$$\overset{(-)}{-3w_{15}} \overset{(-)}{+ w_{25}} \overset{(-)}{+ b_3} = 0$$

$$-2w_{15} + 2w_{25} = 0$$

$$\boxed{w_{15} = w_{25}}$$

we get,

$w_{13} = -3$	$w_{14} = 1$	$w_{15} = 1$
$w_{23} = 1$	$w_{24} = -3$	$w_{25} = 1$
$b_1 = 0$	$b_2 = 0$	$b_3 = -4$

Considering the signum function in transfer function

$$f(n) = \begin{cases} 1 & n > 0 \\ -1 & n \leq 0 \end{cases}$$

As per the function we categorize the points inside and on the triangle as one class and the points outside the triangle as the other class.

We are assuming the weights $w_{36} = 1$, $w_{46} = 1$, $w_{56} = 1$ and Threshold $= -2$

consider a point (1, 1)

$O_3 = sgn(X w_{13} + Y w_{23}) = sgn(1(-3) + 1(1))$
$= sgn(-2)$
$= -1$

$O_4 = sgn(X w_{14} + Y w_{24}) = sgn(1(1) + 1(-3))$
$= sgn(-2)$
$= -1$

$O_5 = sgn(X w_{15} + Y w_{25}) = sgn(1(1) + 1(1) - 4)$
$= sgn(-2)$
$= -1$

$Z = sgn((O_3 w_{36} + O_4 w_{46} + O_5 w_{56}) + 2) = sgn((O_3 w_{36} + O_4 w_{46} + O_5 w_{56}) - (-2))$
$= sgn(((-1)1 + (-1)1 + (-1)(1)) + 2)$
$= -1$

As we get $Z = -1$, Hence the point is inside the triangle.

consider the point (2,2)

$O_3 = sgn(X w_{13} + Y w_{23}) = sgn(2(-3) + 2(1))$
$= sgn(-4)$
$= -1$

$O_4 = sgn(X w_{14} + Y w_{24}) = sgn(2(1) + 2(-3))$
$= sgn(-4)$
$= -1$

$O_5 = sgn(X w_{15} + Y w_{25}) = sgn(2(1) + 2(1) - 4)$
$= sgn(0)$
$= -1$

$Z = sgn((O_3 w_{36} + O_4 w_{46} + O_5 w_{56}) + 2)$
$= sgn(((-1)(1) + (-1)(1) + (-1)(1)) - (-2))$
$= sgn(-1)$
$= -1$

The point (2,2) lies on the triangle. So I have grouped it with the points inside the triangle

Consider the point (1, 4)

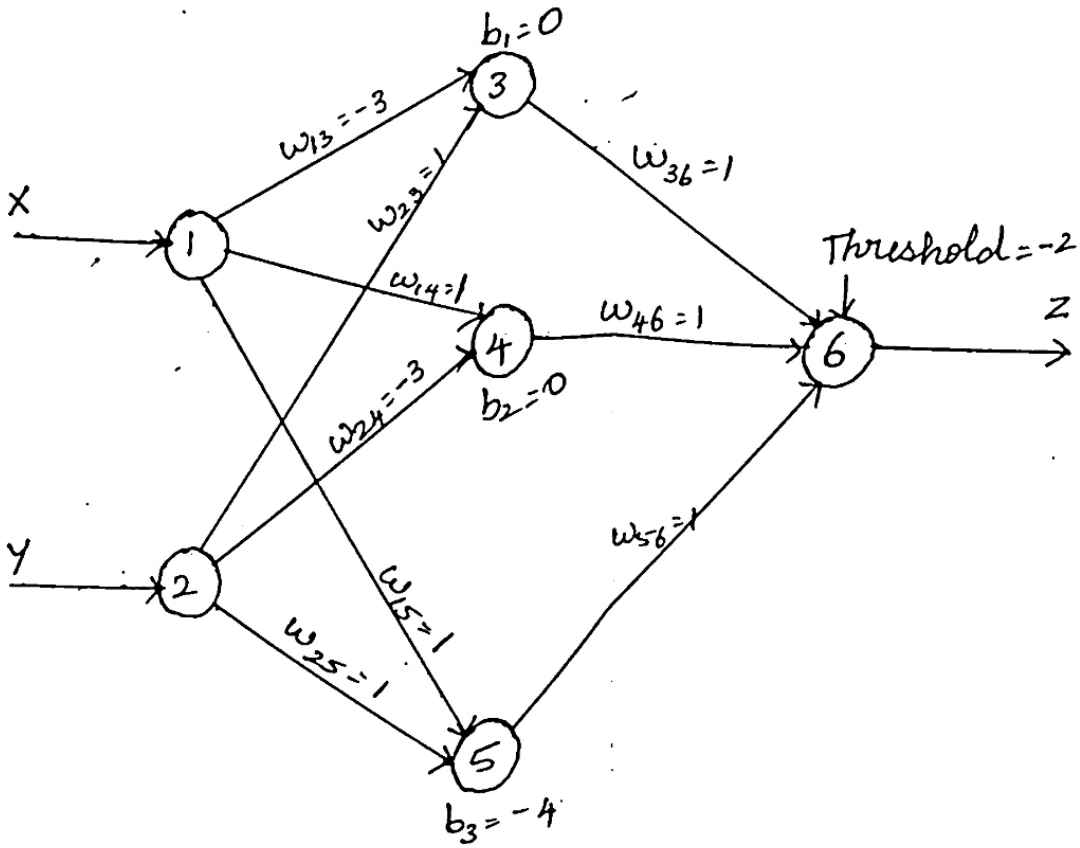
$$O_3 = sgn(xw_{13} + yw_{23}) = sgn(1(-3) + 4(1))$$
$$= sgn(1)$$
$$= 1$$

$$O_4 = sgn(xw_{14} + yw_{24}) = sgn(1(1) + 4(-3))$$
$$= sgn(-11)$$
$$= -1$$

$$O_5 = sgn(xw_{15} + yw_{25}) = sgn(1(1) + 1(4) - 4)$$
$$= sgn(1)$$
$$= 1$$

$$I = sgn((O_3 w_{36} + O_4 w_{46} + O_5 w_{56}) + 2)$$
$$= sgn((1(1) + (-1)(1) + 1(1)) - (-2))$$
$$= sgn(3)$$
$$= 1$$

The point (1, 4) lies outside the triangle



Q3

Given: The normalized widrow Hoff learning rule

$$\Delta w^{(k)} = \eta \left(t^{(k)} - w^{(k)} x^{(k)} \right) \frac{x^{(k)}}{\|x^{(k)}\|^2} \quad -(1)$$

$\Delta w^{(k)} = w^{(k+1)} - w^{(k)}$ —(2)

$x^{(k)}$ - input vector

$\|x^{(k)}\|$ - norm of input vector

η - positive number $(0 < \eta < 1)$

$t^{(k)}$ - target

Solution:

To prove: If $x^{(k)} = x^{(k+1)} \Rightarrow t^{(k+1)} = t^{(k)}$ —(3)

then $\Delta w^{(k+1)} = (1-\eta) \Delta w^{(k)}$

Assuming the operation between $(t^{(k)} - w^{(k)} x^{(k)})$ and $\frac{x^{(k)}}{\|x^{(k)}\|^2}$ is a dot product

At k+1 iteration eq (1) becomes

$$\Delta w^{(k+1)} = \eta \left(t^{(k+1)} - w^{(k+1)} x^{(k+1)} \right) \frac{x^{(k+1)}}{\|x^{(k+1)}\|^2}$$

using (3)

$$\Delta w^{(k+1)} = \eta \left(t^{(k)} - w^{(k+1)} x^{(k)} \right) \frac{x^{(k)}}{\|x^{(k)}\|^2} \quad -(4)$$

Divide eq (4) by eq (1), we get

$$\frac{\Delta w^{(k+1)}}{\Delta w^{(k)}} = \frac{\eta \left(t^{(k)} - w^{(k+1)} x^{(k)} \right) \frac{x^{(k)}}{\|x^{(k)}\|^2}}{\eta \left(t^{(k)} - w^{(k)} x^{(k)} \right) \frac{x^{(k)}}{\|x^{(k)}\|^2}} \quad -(5)$$

From eq (2), we know $w^{(k+1)} = \Delta w^{(k)} + w^{(k)}$

Substituting this in eq (5),

$$\frac{\Delta w^{(k+1)}}{\Delta w^{(k)}} = \frac{\left(t^{(k)} - \left[\Delta w^{(k)} + w^{(k)}\right] x^{(k)}\right)}{t^{(k)} - w^{(k)} x^{(k)}}$$

$$= \frac{t^{(k)} - \Delta w^{(k)} x^{(k)} - w^{(k)} x^{(k)}}{t^{(k)} - w^{(k)} x^{(k)}}$$

$$\frac{\Delta w^{(k+1)}}{\Delta w^{(k)}} = \frac{t^{(k)} - \Delta w^{(k)} x^{(k)}}{t^{(k)} - w^{(k)} x^{(k)}} - \frac{\Delta w^{(k)} x^{(k)}}{t^{(k)} - w^{(k)} x^{(k)}}$$

$$\frac{\Delta w^{(k+1)}}{\Delta w^{(k)}} = 1 - \frac{\Delta w^{(k)} x^{(k)}}{t^{(k)} - w^{(k)} x^{(k)}} \quad \text{— (6)}$$

Substituting eq (1) in eq (6),

$$\frac{\Delta w^{(k+1)}}{\Delta w^{(k)}} = 1 - \eta \frac{\left(t^{(k)} - w^{(k)} x^{(k)}\right)}{t^{(k)} - w^{(k)} x^{(k)}} \frac{\langle x^{(k)} \cdot x^{(k)} \rangle}{\| x^{(k)} \|^2}$$

$$= 1 - \eta \frac{\langle x^{(k)} \cdot x^{(k)} \rangle}{\| x^{(k)} \|^2}$$

we know that $\| x^{(k)} \|^2 = \sum_i \left(x_i^{(k)}\right)^2 = \langle x^{(k)} \cdot x^{(k)} \rangle$

$$\therefore \frac{\Delta w^{(k+1)}}{\Delta w^{(k)}} = 1 - \eta \frac{\sum_i \left(x_i^{(k)}\right)^2}{\sum_i \left(x_i^{(k)}\right)^2}$$

$$\boxed{\Delta w^{(k+1)} = (1 - \eta) \Delta w^{(k)}}$$

Hence proved.

ECE 657 Assignment 1 : Problem 4

In [1]:
```
# Importing the libraries
import pandas as pd
import numpy as np
import random
import math
import pickle
import matplotlib.pyplot as plt
```

In [2]:
```
# Loading the dataset
X = pd.read_csv('train_data.csv',header=None)
y = pd.read_csv('train_labels.csv',header=None)
```

In [3]:
```
# checking the number of features and samples in X and y
X.shape, y.shape
```

Out[3]: ((24754, 784), (24754, 4))

In [4]:
```
y.head()
```

Out[4]:

	0	1	2	3
0	1.0	0.0	0.0	0.0
1	0.0	1.0	0.0	0.0
2	0.0	0.0	1.0	0.0
3	0.0	1.0	0.0	0.0
4	0.0	0.0	0.0	1.0

In [5]:
```
# Checking how may data points are there in each class [0,1,2,3]
print('Number of data points in each class\n', y.sum(axis=0))
```

```
Number of data points in each class
 0    5923.0
1    6742.0
2    5958.0
3    6131.0
dtype: float64
```

In [6]:
```
# function to split the data into train and validation dataset
def train_val_split(X,y,train_size):
    train_size = int(train_size*len(X))
    # Split the X dataset
    train_x = X[:train_size]
    val_x = X[train_size:]
    train_y = y[:train_size]
    val_y = y[train_size:]
    return train_x,val_x,train_y,val_y
```

In [7]:
```
#train-validation test split (80-20%)
X_train,X_val,y_train,y_val = train_val_split(X, y,train_size = 0.8)
```

In [8]:
```
# checking the number of features and samples in X_train, X_val, y_train and y_val
X_train.shape, X_val.shape, y_train.shape ,y_val.shape
```

Out[8]: ((19803, 784), (4951, 784), (19803, 4), (4951, 4))

In [9]:
```
# changing X_train, X_val, y_train and y_val into numpy array
train_x =  X_train.to_numpy()
train_y = y_train.to_numpy()
val_x = X_val.to_numpy()
val_y = y_val.to_numpy()
```

In [18]:
```
class Neural_Net:

# We initialize the list of weights matrices, then store the network architecture and learning rate.
# The "__init__" functions acts as a constructor to Neural_Net class and we are initializing the layer
#parameters using it. We pass the parameter self and later will set the number of nodes to 20 in
#the hidden layer.

    def __init__(self, hidden_layer):
        self.hidden_layer = hidden_layer
        self.parameters = {}
        self.n = 0
        self.train_cost = []
        self.train_acc = []
        self.val_acc = []

    # compute and return the sigmoid activation value for X
    def sigmoid_ftn(self, X):
        return 1 / (1 + np.exp(-X))

    # compute the derivative of the sigmoid function
```

```
    # for this function we are considering that sigmoid_ftn has been called with X
    def sigmoid_der(self, X):
        s = self.sigmoid_ftn(X)
        return s * (1 - s)

    # compute and return the softmax activation value for a given input value
    def softmax_act(self, X):
        expZ = np.exp(X - np.max(X))
        return expZ / expZ.sum(axis=0, keepdims=True)

    def initialize_parameters(self):
        np.random.seed(1)
        # We are creating a weight matrix by randomly linking the number of
        # nodes in each layer and adding an additional node for the bias.
        self.parameters["Wt1"] = np.random.randn(self.hidden_layer, 784) / np.sqrt(784)
        self.parameters["bias1"] = np.zeros((self.hidden_layer, 1))
        self.parameters["Wt2"] = np.random.randn(4, self.hidden_layer) / np.sqrt(self.hidden_layer)
        self.parameters["bias2"] = np.zeros((4, 1))


    # FEEDFORWARD:

        # This function takes data in terms of batch and input units and return output values.
        # The fwd_propagation() function computes the output value for the supplied input observation.


        def fwd_propagation(self, X):
            store = {}

    # While our data point travels through the network, we compile a list of output activations for each layer.
            inp_t = X.T

            # Linear transformation between IP and HIDDEN LAYER
            Wt1_hl = self.parameters["Wt1"].dot(inp_t) + self.parameters["bias1"]
            # Applying sigmoid
            act_hl = self.sigmoid_ftn(Wt1_hl)
            # Linear transformation between HIDDEN LAYER and previous layer
            Wt2_op = self.parameters["Wt2"].dot(act_hl) + self.parameters["bias2"]
            # Softmax is the activation function for last layer
            act_op = self.softmax_act(Wt2_op)

            # Storing the values for use later on in back propagation
            store["act_hl"] = act_hl
            store["act_op"] = act_op
            store["Wt1_hl"] = Wt1_hl
            store["Wt2_op"] = Wt2_op
            store["Wt1"] = self.parameters["Wt1"]
            store["Wt2"] = self.parameters["Wt2"]
            store["bias1"] = self.parameters["bias1"]
            store["bias2"] = self.parameters["bias2"]
            return act_op, store

        def back_propagation(self, X, Y, store):
            derivatives = {}
            store["A0"] = X.T
            act_op = store["act_op"]

            # Since we had already used softmax in the last layer, we get probabilistic output from the network.
            #This will have an exponent component. We choose Cross Entropy as our loss function because the log
            #term in cross entropy and the nature of derivative of softmax function, during back propagation
            #calculation,we can directly put dL\dY as A - Y. Hence, as per the derivation, we tried to directly
            #calculate the same in our code.

            dY = act_op - Y.T
            dWeight = dY.dot(store["act_hl"].T)/ self.n
            loss_db = np.sum(dY, axis=1, keepdims=True)/ self.n

            #backward function will compute and return the delta

            delta = store["Wt2"].T.dot(dY)
            derivatives["dW2"] = dWeight
            derivatives["db2"] = loss_db

            #Applying chain rule to previous layers, we get the following equations for calculation of derivatives

            dY = delta * self.sigmoid_der(store["Wt1_hl"])
            dWeight = (1. / self.n) * dY.dot(store["A0"].T)
            loss_db = (1. / self.n) * np.sum(dY, axis=1, keepdims=True)
            #Storing the derivatives here
            derivatives["dW1"] = dWeight
            derivatives["db1"] = loss_db
            return derivatives

        # Function for training our data
        def train(self, X, Y, X_val, Y_val, learning_rate=0.05, n_iterations=100):
            np.random.seed(22)
            self.n = X.shape[0]
            self.initialize_parameters()
            # creating a for loop for interation for 100 times
            for loop in range(n_iterations):
                #Forward propagation
                inp_t, store = self.fwd_propagation(X)
                #Cross entropy loss function
```

```
            cost = -np.mean(Y * np.log(inp_t.T))
            #Backward propagation
            derivatives = self.back_propagation(X, Y, store)

#Using the derivatives from backprop, we are updating the weights. Here is where the gradient descent happens.
#We take a small step as defined by the learning rate towards the direction of the negative of the gradient.

            self.parameters["Wt1"] = self.parameters["Wt1"] - learning_rate * derivatives["dW1"]
            self.parameters["bias1"] = self.parameters["bias1"] - learning_rate * derivatives["db1"]
            self.parameters["Wt2"] = self.parameters["Wt2"] - learning_rate * derivatives["dW2"]
            self.parameters["bias2"] = self.parameters["bias2"] - learning_rate * derivatives["db2"]

            # Printing during training
            if loop % 5 == 0:
                print("Cost: ", cost, "Train Accuracy:", self.pred_acc(X, Y))
            # Saving values for printing
            if loop % 5 == 0:
                self.train_cost.append(cost)
                self.train_acc.append(self.pred_acc(X, Y))
                self.val_acc.append(self.pred_acc(X_val, Y_val))

    # Predicting accuracy to be used in recording training accuracy and validation accuracy
    def pred_acc(self, X, Y):
        A, cache = self.fwd_propagation(X)
        y_hat = np.argmax(A, axis=0)
        Y = np.argmax(Y, axis=1)
        accuracy = (y_hat == Y).mean()
        return accuracy * 100

    # Predicts the label of the given input data
    def predict(self,X):
        y_hat,_ = self.fwd_propagation(X)
        y_hat = y_hat.T
        y_hat = (y_hat/np.reshape(np.max(y_hat,axis=1),(-1,1))).astype(int)
        return y_hat

    #Saves the weights for later use
    def save(self):
        for name,value in zip(list(self.parameters.keys()),list(self.parameters.values())):
            np.save(name,value)

    #Loads saved weights
    def load(self):
        for name in ['Wt1','Wt2','bias1','bias2']:
            try:
                self.parameters[name] = np.load(name+".npy")
            except:
                raise(name+".npy file not found...")


#In the class, each layer is trying to do two things:
#Processing the input to get the output using forward propagation
#and itself propagation of gradients i.e. back propagation.
#For activation, we're utilising the logistic sigmoid function.
#The logistic function itself is necessary for calculating later activated values,
#while the logistic function's derivative is required for backpropagation.
```

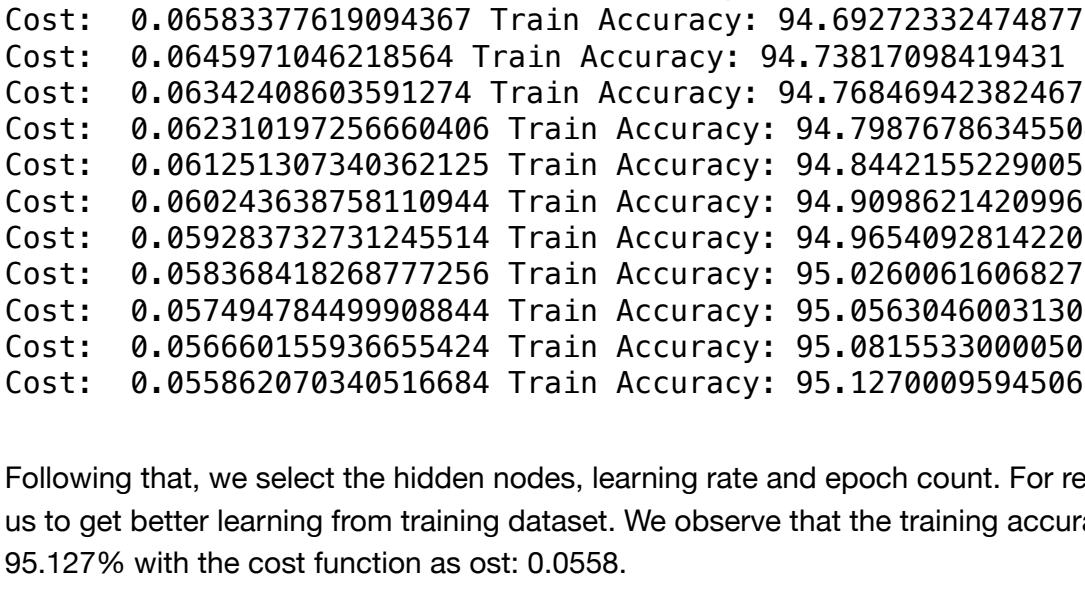
In [19]:
```
# Defining the model with 20 hidden layers
hidden_nodes = 20

# define our multi-layer perceptron
nn_1 = Neural_Net(hidden_nodes)
# Training it with a learning rate of 0.1
nn_1.train(train_x, train_y, val_x, val_y, learning_rate=0.1, n_iterations=250)
nn_1.save()
```

```
Cost:  0.3781774430729633 Train Accuracy: 23.84487198909256
Cost:  0.32567511705775104 Train Accuracy: 55.44614452355704
Cost:  0.2992734965165988 Train Accuracy: 70.31257890218654
Cost:  0.27826339494561675 Train Accuracy: 78.85168913800939
Cost:  0.2599641315963605 Train Accuracy: 82.75513811038732
Cost:  0.24365256724340795 Train Accuracy: 84.72453668636065
Cost:  0.2289141553537713 Train Accuracy: 86.07281724991162
Cost:  0.21547587844809163 Train Accuracy: 87.10296419734384
Cost:  0.20315303860485998 Train Accuracy: 87.98666868656264
Cost:  0.19181485237883944 Train Accuracy: 88.63808513861537
Cost:  0.18136339855603728 Train Accuracy: 89.28950159066808
Cost:  0.17172104814749725 Train Accuracy: 89.90556986315205
Cost:  0.1628228582562557 Train Accuracy: 90.50648891582084
Cost:  0.1546119372774261 Train Accuracy: 90.92056759076908
Cost:  0.14703667053619301 Train Accuracy: 91.30434782608695
Cost:  0.14004913706457608 Train Accuracy: 91.62248144220572
Cost:  0.13360427993227786 Train Accuracy: 91.91031661869414
Cost:  0.12765953228215826 Train Accuracy: 92.19310205524415
Cost:  0.12217469682647443 Train Accuracy: 92.44053931222543
Cost:  0.11711194594348326 Train Accuracy: 92.6778770893299
Cost:  0.11243586004839055 Train Accuracy: 92.89501590668081
Cost:  0.10811345770760816 Train Accuracy: 93.0414583648942
Cost:  0.10411419515687544 Train Accuracy: 93.21819926273797
Cost:  0.10040992821384372 Train Accuracy: 93.38989042064334
Cost:  0.09697483845171892 Train Accuracy: 93.5666313184871
Cost:  0.09378532993941775 Train Accuracy: 93.70297429682371
Cost:  0.09081990444353945 Train Accuracy: 93.81406857546837
Cost:  0.08805902289988059 Train Accuracy: 93.91506337423623
Cost:  0.0854849600061253 Train Accuracy: 94.05140635257284
Cost:  0.08308165748335074 Train Accuracy: 94.16755037115588
Cost:  0.08083458021077039 Train Accuracy: 94.22814725041661
Cost:  0.07873057821908348 Train Accuracy: 94.33924152906125
Cost:  0.0767577565033743 Train Accuracy: 94.39478866838358
Cost:  0.07490535380208939 Train Accuracy: 94.4553855476443
Cost:  0.07316363087010275 Train Accuracy: 94.49578346715144
Cost:  0.07152376832379957 Train Accuracy: 94.51598242690501
Cost:  0.06997777382321574 Train Accuracy: 94.58162904610413
Cost:  0.0685183981506476 Train Accuracy: 94.63717618542645
Cost:  0.06713905962061392 Train Accuracy: 94.6725243649952
Cost:  0.06583377619094367 Train Accuracy: 94.69272332474877
Cost:  0.0645971046218564 Train Accuracy: 94.73817098419431
Cost:  0.06342408603591274 Train Accuracy: 94.76846942382467
Cost:  0.062310197256660406 Train Accuracy: 94.79876786345504
Cost:  0.061251307340362125 Train Accuracy: 94.84421552290057
Cost:  0.060243638758110944 Train Accuracy: 94.90986214209968
Cost:  0.059283732731245514 Train Accuracy: 94.96540928142201
Cost:  0.058368418268777256 Train Accuracy: 95.02600616068273
Cost:  0.057494784499908844 Train Accuracy: 95.05630460031308
Cost:  0.056660155936655424 Train Accuracy: 95.08155330000506
Cost:  0.055862070340516684 Train Accuracy: 95.1270009594506
```

Following that, we select the hidden nodes, learning rate and epoch count. For real neural networks, training over numerous epochs is critical since it helps us to get better learning from training dataset. We observe that the training accuracy keeps on increasing with the number of iterations and finally we get it as 95.127% with the cost function as ost: 0.0558.

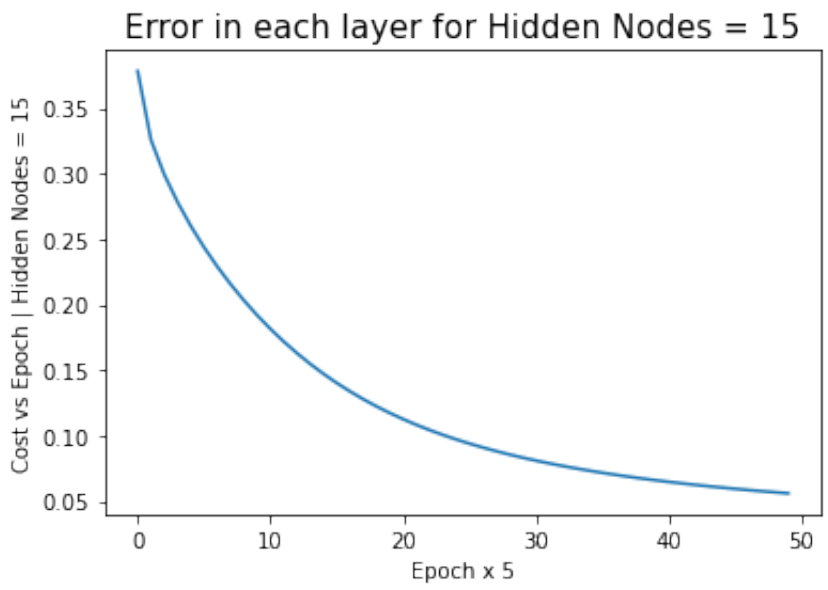
In [22]:
```
plt.plot(range(len(nn_1.train_acc)),nn_1.train_acc,nn_1.val_acc)
plt.title("Training Accuracy and validation accuracy for Hidden Nodes = 15",fontsize=15)
plt.legend(["Train Acc", "Validation Acc"])
plt.xlabel("Epoch x 5")
plt.ylabel("Accuracy vs Epoch | Hidden Nodes = 15")
```

Out[22]: Text(0, 0.5, 'Accuracy vs Epoch | Hidden Nodes = 15')



In [23]:
```
plt.plot(range(len(nn_1.train_acc)),nn_1.train_cost)
plt.title("Error in each layer for Hidden Nodes = 15",fontsize=15)
plt.xlabel("Epoch x 5")
plt.ylabel("Cost vs Epoch | Hidden Nodes = 15")
```

Out[23]: Text(0, 0.5, 'Cost vs Epoch | Hidden Nodes = 15')



We can observe from the accuracy graphs that the training and validation accuracies are increasing with the increase in the number of epochs and the error is decreasing with the number of epochs.