A2Q1

November 2, 2021

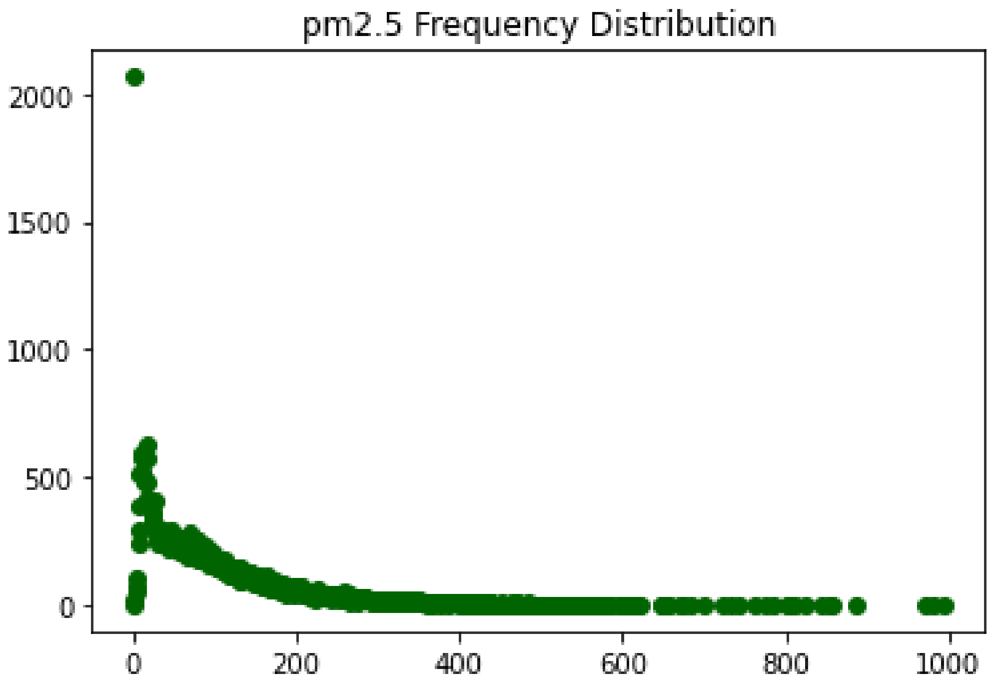
[4]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
from sklearn.tree import DecisionTreeClassifier
import random
```

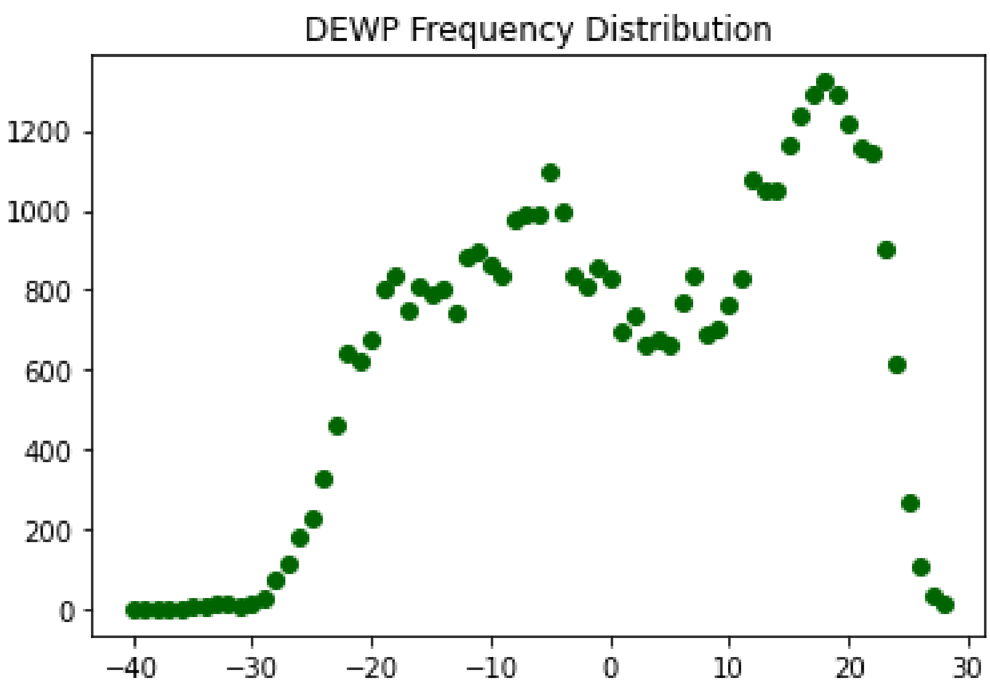
0.0.1 Data Pre - Processing and EDA

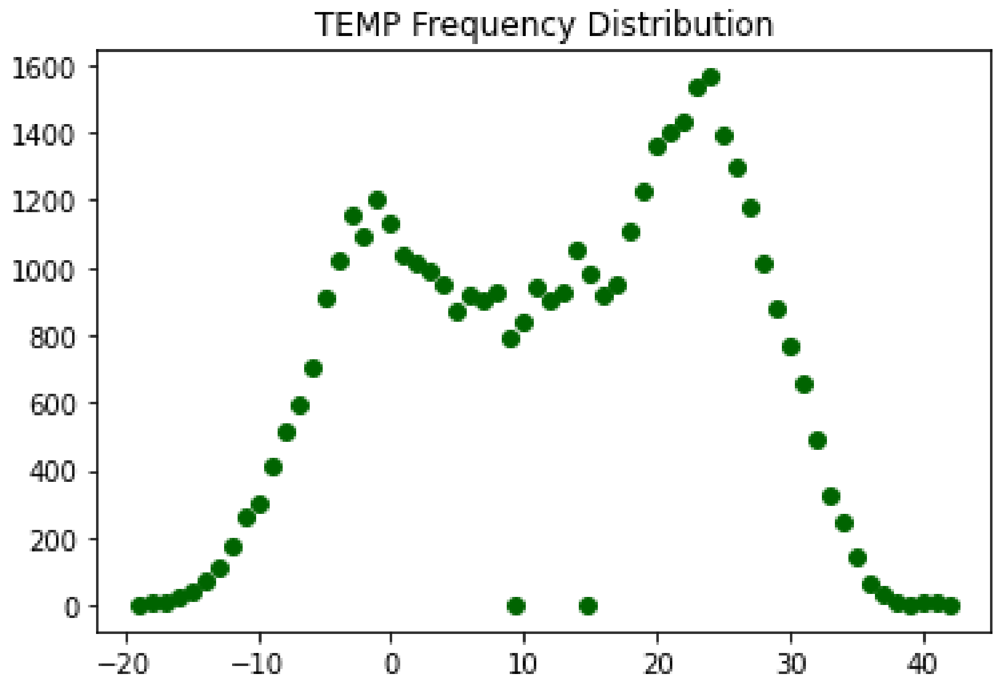
1. Replacing NaN values with 0
2. Drop the No. column as described in question
3. One Hot Encoding has been used for the 'cbwd' column
4. Plotted Frequency Distribution of Various Attributes
5. Performed Gaussian Normalization for ['DEWP', 'PRES', 'TEMP']
6. Performed Min-Max Normalization for ['pm2.5','Iws', 'Is', 'Ir']

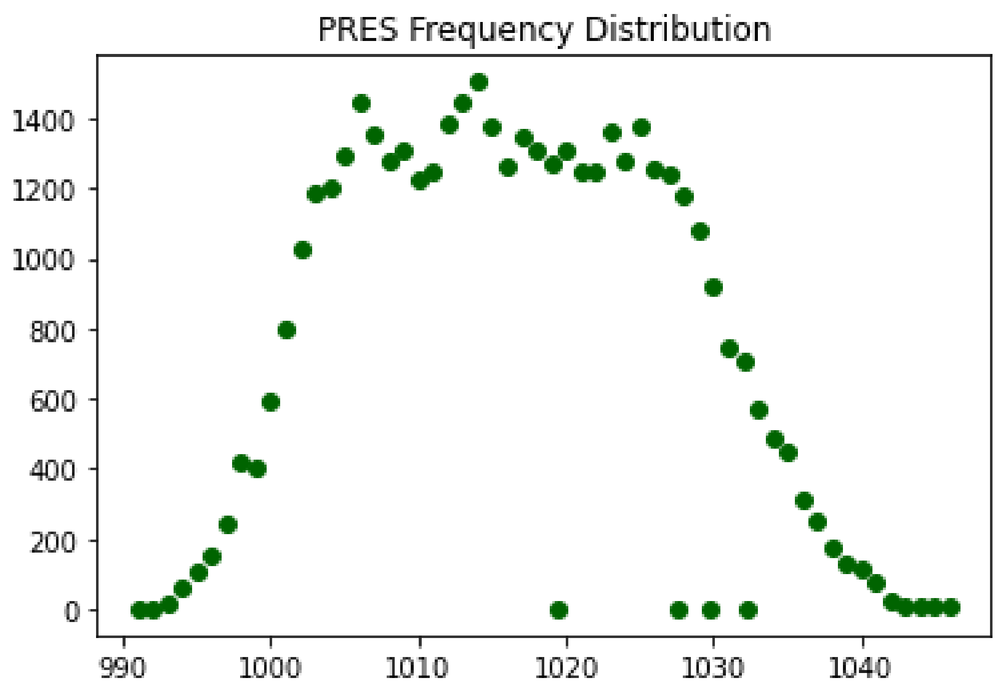
[6]:
```python
def eda(df):
  for column in ['pm2.5', 'DEWP', 'TEMP', 'PRES', 'Iws', 'Is', 'Ir']:
    freq = {}
    for val in df[column]:
        if not np.isnan(val):
            freq[val] = freq.get(val, 0) + 1
    x = freq.keys()
    y = freq.values()
    plt.scatter(x, y, color='darkgreen')
    plt.title(f"{column} Frequency Distribution")
    plt.show()
```

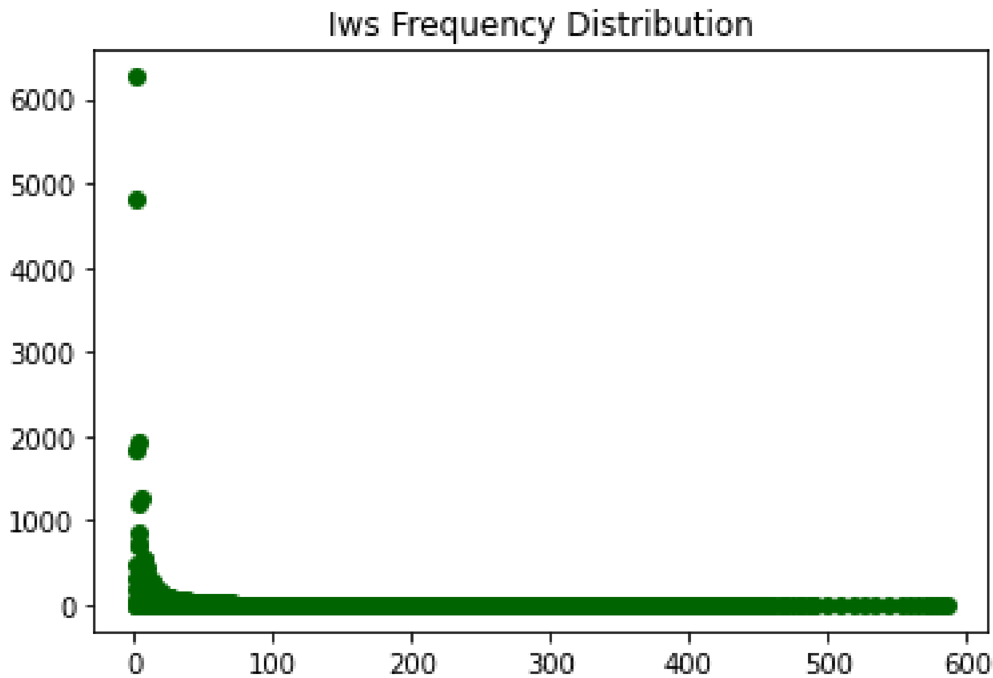
[7]:
```python
dir="datasetq1.csv"
df=pd.read_csv(dir)
df = df.sample(frac=1, random_state=0).reset_index(drop=True)
df.replace({np.nan:0}, inplace=True)
df=df.drop('No',axis=1)
df = pd.get_dummies(df, columns = ['cbwd'])
eda(df)
y=df["month"]
X=df.drop("month",axis=1)
```

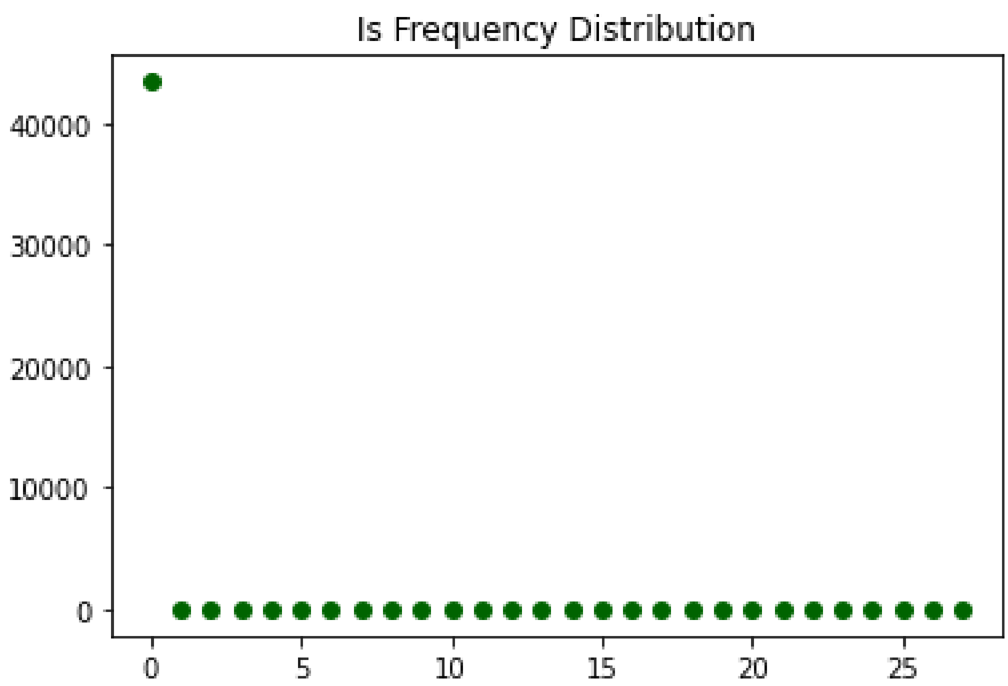

pm2.5 Frequency Distribution
2000
1500
1000
500
0
0
200
400
600
800
1000

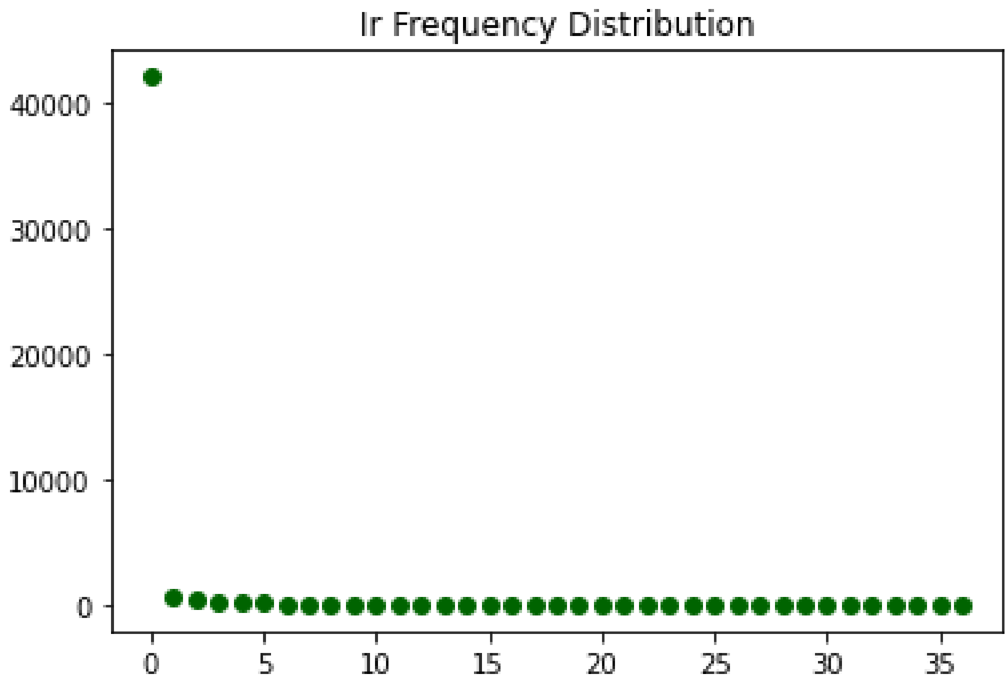

DEWP Frequency Distribution
1200
1000
800
600
400
200
0
−40
−30
−20
−10
0
10
20
30


TEMP Frequency Distribution
1600
1400
1200
1000
800
600
400
200
0
−20
−10
0
10
20
30
40


PRES Frequency Distribution
1400
1200
1000
800
600
400
200
0
990
1000
1010
1020
1030
1040


Iws Frequency Distribution
6000
5000
4000
3000
2000
1000
0
0
100
200
300
400
500
600


Is Frequency Distribution
40000
30000
20000
10000
0
0
5
10
15
20
25



[8]:
```
gaussian_cols=['DEWP', 'PRES', 'TEMP']
minmax_cols=['pm2.5','Iws', 'Is', 'Ir']
for i in minmax_cols:
  X[i]=(X[i]-X[i].min())/(X[i].max()-X[i].min())
for i in gaussian_cols:
  X[i]=(X[i]-X[i].mean())/(X[i].std())
```

[9]:
```
def accuracy_score(y_pred, y):
  counter=0
  for i in range(len(y_pred)):
    if(y_pred[i]==y[i]):
      counter+=1
  return counter/len(y)

def mytraintestvalsplit(total,valsize,testsize):
    notestrows=int(testsize*total.shape[0])
    novalrows=int(valsize*total.shape[0])
    notrainrows=total.shape[0]-notestrows-novalrows
    trainrows=total[:notrainrows]
    valrows=total[notrainrows:notrainrows+novalrows]
    testrows=total[notrainrows+novalrows:]
    return trainrows,valrows,testrows
X=X.to_numpy()
y=y.to_numpy()
```

```
X_train, X_val,X_test = mytraintestvalsplit(X, 0.15,0.15)
Y_train, Y_val,Y_test = mytraintestvalsplit(y, 0.15,0.15)
```

[10]:
```
clf = DecisionTreeClassifier(criterion='gini')
clf = clf.fit(X_train, Y_train)
pred = clf.predict(X_test)
score = accuracy_score(pred, Y_test)
print(f"Accuracy with Gini comes out to be {score}")
```

```
Accuracy with Gini comes out to be 0.8209341244485014
```

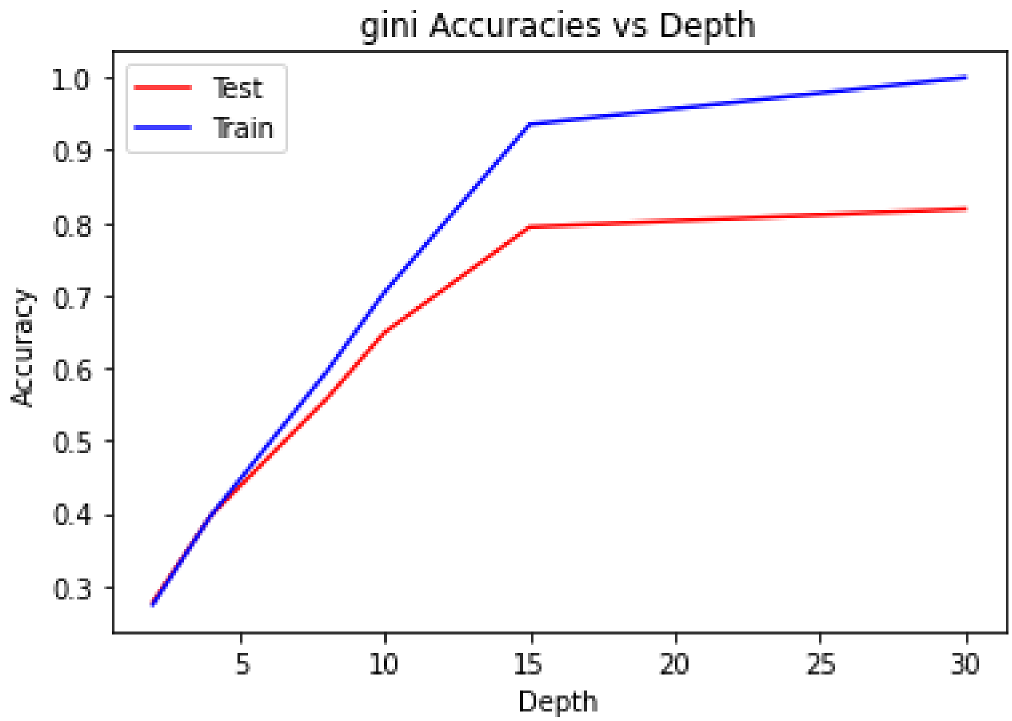
[11]:
```
clf = DecisionTreeClassifier(criterion='entropy')
clf = clf.fit(X_train, Y_train)
pred = clf.predict(X_test)
score = accuracy_score(pred, Y_test)
print(f"Accuracy with Entropy comes out to be {score}")
```

```
Accuracy with Entropy comes out to be 0.8323444393731934
```

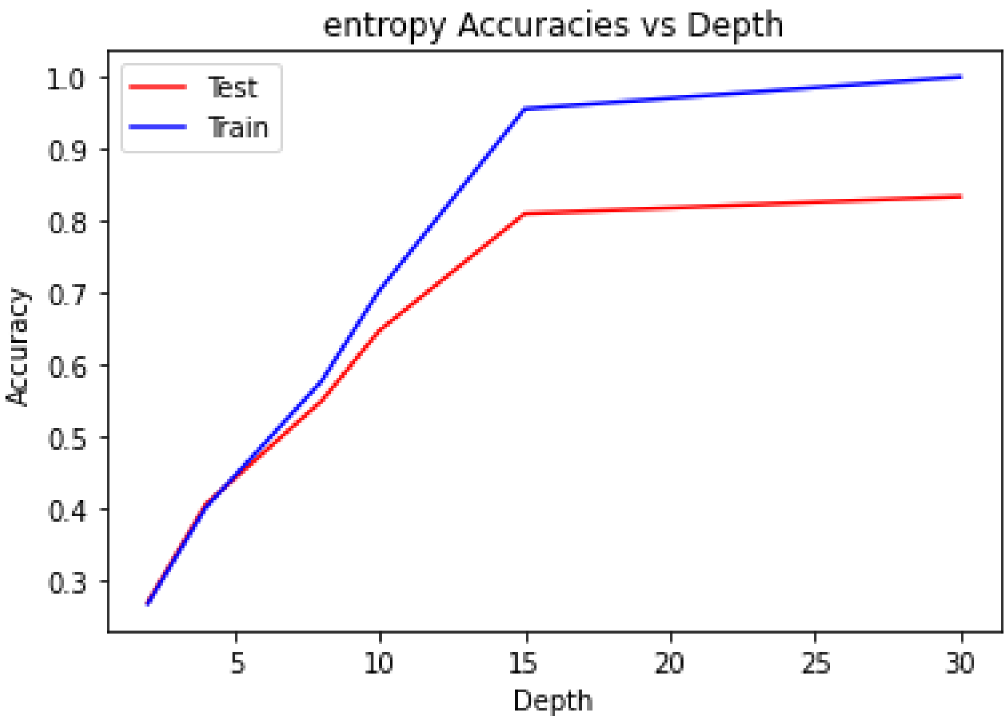
Q1 (a) Decision Tree with Entropy Gives a higher Accuracy.

[8]:
```
def q1b(technique):
  max_dep=[2,4,8,10,15,30]
  score_test=[]
  score_train=[]
  for i in max_dep:
    clf = DecisionTreeClassifier(criterion=technique,max_depth=i)
    clf = clf.fit(X_train, Y_train)
    pred = clf.predict(X_test)
    score = accuracy_score(pred, Y_test)
    score_test.append(score)
    pred = clf.predict(X_train)
    score = accuracy_score(pred, Y_train)
    score_train.append(score)
  plt.plot(max_dep, score_test, label = "Test",color="red")
  plt.plot(max_dep, score_train, label = "Train",color="blue")
  plt.xlabel('Depth')
  plt.ylabel('Accuracy')
  plt.title(f'{technique} Accuracies vs Depth')

  plt.legend()
  plt.show()
  print()

  for x in range(len(score_test)):
    print(f"The accuracy score with {technique} on the test set with depth␣
↪{max_dep[x]} is {score_test[x]}")
```

```
q1b('gini')
q1b('entropy')
```




```
The accuracy score with gini on the test set with depth 2 is 0.2771945839038491
The accuracy score with gini on the test set with depth 4 is 0.3966225467822912
The accuracy score with gini on the test set with depth 8 is 0.5575840559866119
The accuracy score with gini on the test set with depth 10 is 0.6490187129164765
The accuracy score with gini on the test set with depth 15 is 0.7941579187585578
The accuracy score with gini on the test set with depth 30 is 0.8189563365282215
```




```
The accuracy score with entropy on the test set with depth 2 is
0.27065267001369236
The accuracy score with entropy on the test set with depth 4 is
0.40696789898067853
The accuracy score with entropy on the test set with depth 8 is
0.5505857294994675
The accuracy score with entropy on the test set with depth 10 is
0.6482580252548303
The accuracy score with entropy on the test set with depth 15 is
0.8101323596531265
The accuracy score with entropy on the test set with depth 30 is
0.8338658146964856
```

Q1 (b) The best value of depth for both is 30. Entropy performs slightly better than Gini. (Plots shown above)

[9]:
```
def q1c(nooftrees,depth,X_train,Y_train,X_test,Y_test):
  list_clf=[]

  for i in range(nooftrees):
    clf=DecisionTreeClassifier(criterion='entropy',max_depth=depth)
    temp_xtrain,_,temp_ytrain,__=train_test_split(X_train,Y_train,test_size=.
↪50,shuffle=True)
```

```
    clf = clf.fit(temp_xtrain, temp_ytrain)
    temp_pred = clf.predict(X_test)
    list_clf.append(temp_pred)


  dic_pred=[]
  for i in range(len(list_clf[0])):
    new_list=[]
    for j in range(len(list_clf)):
      new_list.append(list_clf[j][i])
    dic_pred.append(new_list)

  def most_frequent(List):
      return max(set(List), key = List.count)

  prediction=[]
  for i in range(len(dic_pred)):
    prediction.append(most_frequent(dic_pred[i]))

  score = accuracy_score(prediction, Y_test)
  return (score)
```

[10]:
```
print(f"The accuracy score for our experiment comes out to be␣
 ↪{q1c(100,3,X_train,Y_train,X_test,Y_test)}")
```

```
The accuracy score for our experiment comes out to be 0.34215731020842843
```

This accuracy is way less than 1 (a) and (b)

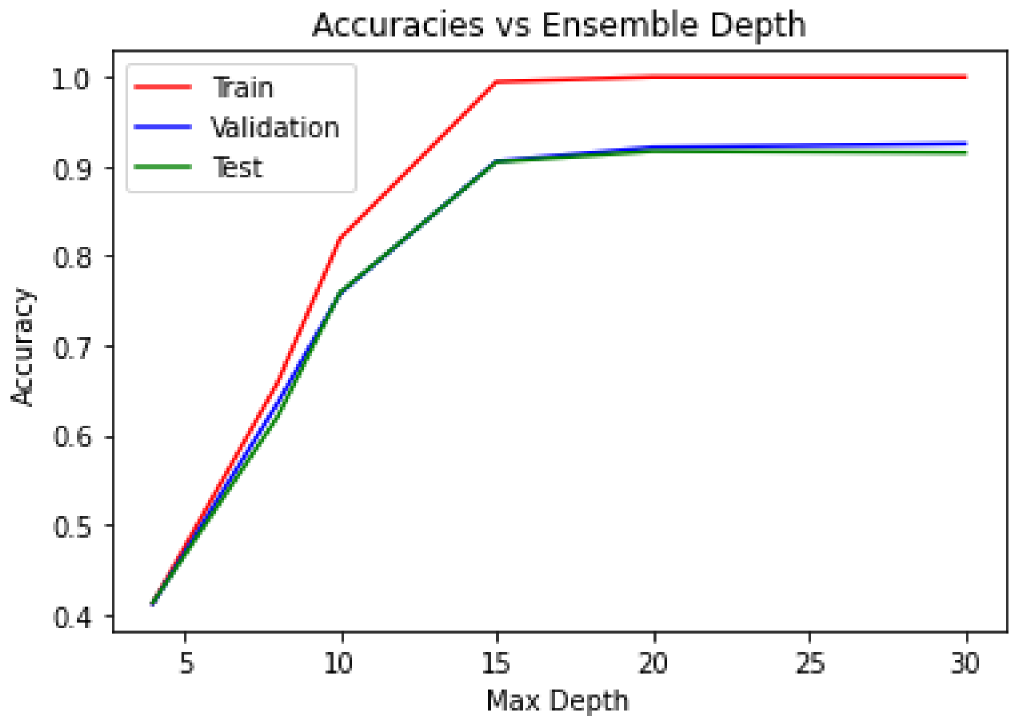
This happened because the max depth of trees in the random forest were taken to be 3 which makes a decision tree with very low depth and the model is under-fitting, i.e. it is not able to use all the input features.

Q1 (d) Trying out various depths [4,8,10,15,20,30] and plotting to find the best depth.

[11]:
```
max_depths=[4,8,10,15,20,30]
num_of_trees=100
training_accuracy=[]
validation_accuracy=[]
testing_accuracy=[]
for j in max_depths:
  training_accuracy.append(q1c(100,j,X_train,Y_train,X_train,Y_train))
  validation_accuracy.append(q1c(100,j,X_train,Y_train,X_val,Y_val))
  testing_accuracy.append(q1c(100,j,X_train,Y_train,X_test,Y_test))

print("Test Set Accuracies")
for x in range(len(testing_accuracy)):
  print(f"The accuracy with depth {max_depths[x]} is : {testing_accuracy[x]}")
```

```python
print("Validation Set Accuracies")
for x in range(len(validation_accuracy)):
  print(f"The accuracy with depth {max_depths[x]} is :␣
↪{validation_accuracy[x]}")

print("Training Set Accuracies")
for x in range(len(training_accuracy)):
  print(f"The accuracy with depth {max_depths[x]} is : {training_accuracy[x]}")

plt.plot(max_depths,training_accuracy,label = "Train",color="red")
plt.plot(max_depths,validation_accuracy,label = "Validation",color="blue")
plt.plot(max_depths,testing_accuracy,label = "Test",color="green")
plt.xlabel('Max Depth')
plt.ylabel('Accuracy')
plt.title('Accuracies vs Ensemble Depth')
plt.legend()
plt.show()
print()
```

```
Test Set Accuracies
The accuracy with depth 4 is : 0.41259698767685987
The accuracy with depth 8 is : 0.6205689943709113
The accuracy with depth 10 is : 0.7588620112581774
The accuracy with depth 15 is : 0.9047619047619048
The accuracy with depth 20 is : 0.9167807698159136
The accuracy with depth 30 is : 0.9151072569602922
Validation Set Accuracies
The accuracy with depth 4 is : 0.41137988741822606
The accuracy with depth 8 is : 0.6357827476038339
The accuracy with depth 10 is : 0.7577970485318728
The accuracy with depth 15 is : 0.9055225924235509
The accuracy with depth 20 is : 0.9210406207211319
The accuracy with depth 30 is : 0.925148334094021
Training Set Accuracies
The accuracy with depth 4 is : 0.4119890475259143
The accuracy with depth 8 is : 0.6586478909968055
The accuracy with depth 10 is : 0.8192189842884151
The accuracy with depth 15 is : 0.9942955864137166
The accuracy with depth 20 is : 0.9997718234565487
The accuracy with depth 30 is : 0.9997066301584198
```



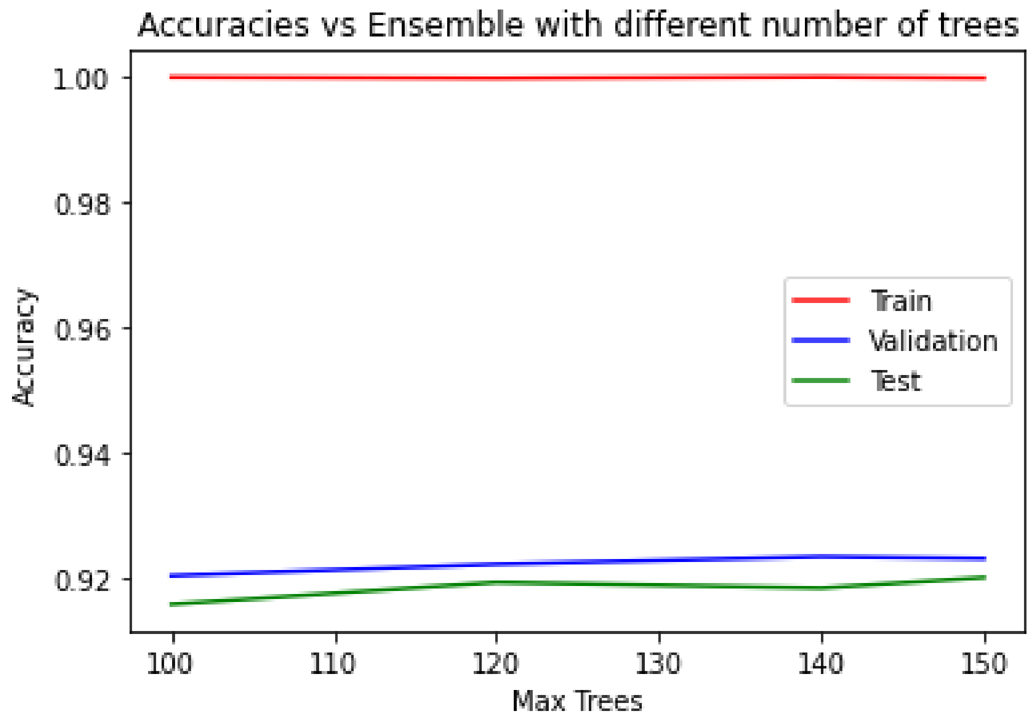
So we can see till depth 20, the accuracy increases on all the sets(training, validation and the test set) and after that it decreases or remains nearly same. So the best depth we can use out of these will be 20. On using a higher depth we are over fitting the model and increasing the computation power required as well, so 20 turns out to be an optimum choice.

0.0.2 Now Tuning for number of Trees [100,120,140,150]

[18]:
```
max_trees=[100,120,140,150]
training_accuracy=[]
validation_accuracy=[]
testing_accuracy=[]
for j in max_trees:
  training_accuracy.append(q1c(j,20,X_train,Y_train,X_train,Y_train))
  validation_accuracy.append(q1c(j,20,X_train,Y_train,X_val,Y_val))
  testing_accuracy.append(q1c(j,20,X_train,Y_train,X_test,Y_test))

print("Test Set Accuracies")
for x in range(len(testing_accuracy)):
  print(f"The accuracy with no of trees {max_trees[x]} is :␣
 ↪{testing_accuracy[x]}")

print("Validation Set Accuracies")
```

```
for x in range(len(validation_accuracy)):
  print(f"The accuracy with no of trees {max_trees[x]} is :␣
↪{validation_accuracy[x]}")

print("Training Set Accuracies")
for x in range(len(training_accuracy)):
  print(f"The accuracy with no of trees {max_trees[x]} is :␣
↪{training_accuracy[x]}")

plt.plot(max_trees,training_accuracy,label = "Train",color="red")
plt.plot(max_trees,validation_accuracy,label = "Validation",color="blue")
plt.plot(max_trees,testing_accuracy,label = "Test",color="green")
plt.xlabel('Max Trees')
plt.ylabel('Accuracy')
plt.title('Accuracies vs Ensemble with different number of trees')
plt.legend()
plt.show()
print()
```

```
Test Set Accuracies
The accuracy with no of trees 100 is : 0.915715807089609
The accuracy with no of trees 120 is : 0.9192149703331812
The accuracy with no of trees 140 is : 0.9183021451392058
The accuracy with no of trees 150 is : 0.9199756579948273
Validation Set Accuracies
The accuracy with no of trees 100 is : 0.9202799330594857
The accuracy with no of trees 120 is : 0.9221055834474364
The accuracy with no of trees 140 is : 0.9233226837060703
The accuracy with no of trees 150 is : 0.9230184086414118
Training Set Accuracies
The accuracy with no of trees 100 is : 0.9997718234565487
The accuracy with no of trees 120 is : 0.9996088402112263
The accuracy with no of trees 140 is : 0.9998044201056131
The accuracy with no of trees 150 is : 0.9996414368602907
```



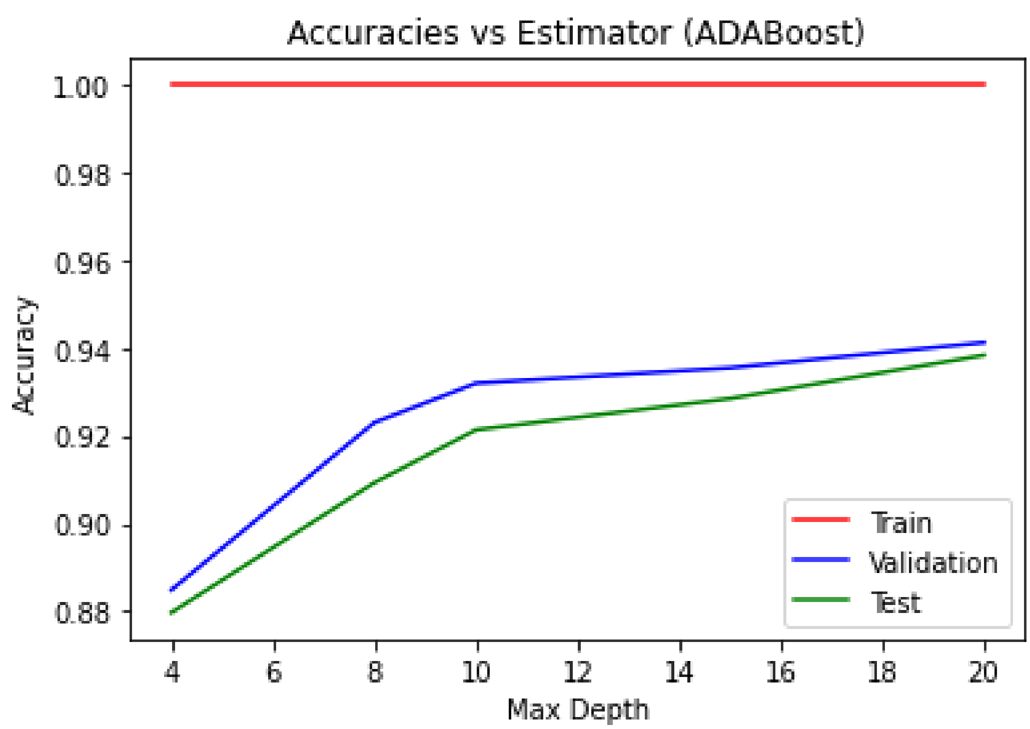
According to the plot we can see, that the best number of trees for the ensemble out of the tested values will be 150.

Analysis : On increasing depth beyond 20, our model is overfitting and hence we see slight decrease in accuracies. On max trees=150, we are getting the best results as both the validation and test sets are giving similar results and overall accuracy is also maximum.

[12]:
```
from sklearn.ensemble import AdaBoostClassifier
est=[4,8,10,15,20]
training_accuracy=[]
validation_accuracy=[]
testing_accuracy=[]
for j in est:
  clf = DecisionTreeClassifier(criterion = 'entropy',max_depth=20)
  abc = AdaBoostClassifier(base_estimator=clf, n_estimators=j, learning_rate=1,␣
↪random_state=0)
  model = abc.fit(X_train, Y_train)
  train_pred = model.predict(X_train)
  validation_pred = model.predict(X_val)
  test_pred = model.predict(X_test)
  training_accuracy.append(accuracy_score(Y_train, train_pred))
  validation_accuracy.append(accuracy_score(Y_val, validation_pred))
  testing_accuracy.append(accuracy_score(Y_test, test_pred))
```

```python
print("Test Set Accuracies")
for x in range(len(testing_accuracy)):
  print(f"The accuracy with ADABoost with n_estimator {est[x]} is :␣
→{testing_accuracy[x]}")


plt.plot(est,training_accuracy,label = "Train",color="red")
plt.plot(est,validation_accuracy,label = "Validation",color="blue")
plt.plot(est,testing_accuracy,label = "Test",color="green")
plt.xlabel('Max Depth')
plt.ylabel('Accuracy')
plt.title('Accuracies vs Estimator (ADABoost)')
plt.legend()
plt.show()
```

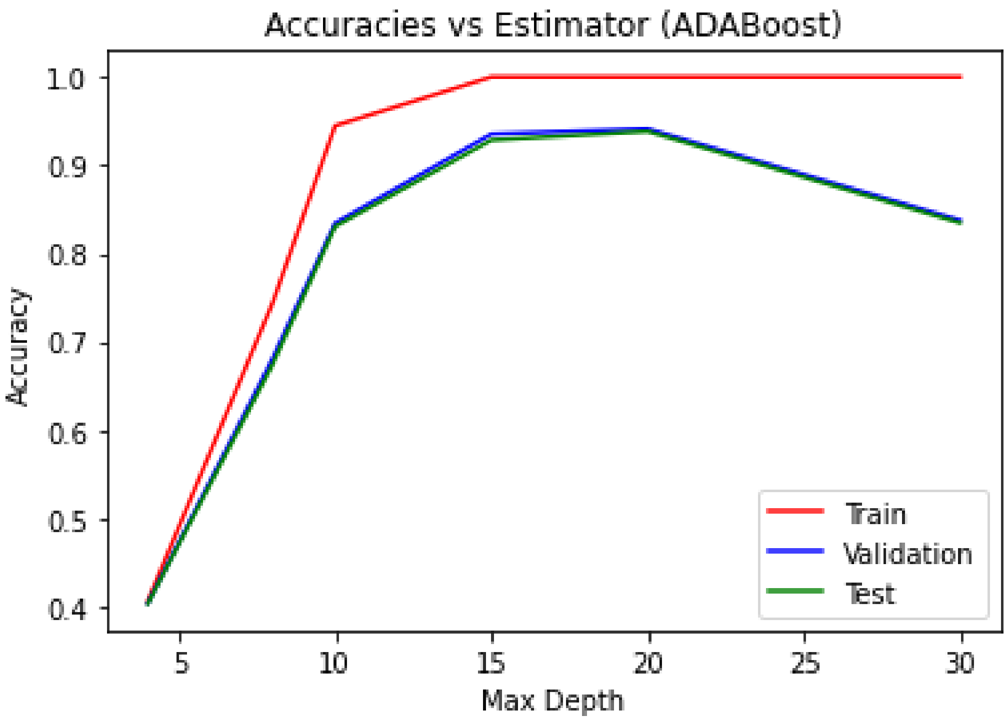
```
Test Set Accuracies
The accuracy with ADABoost with n_estimator 4 is : 0.8796592119275826
The accuracy with ADABoost with n_estimator 8 is : 0.9093260307317815
The accuracy with ADABoost with n_estimator 10 is : 0.9213448957857904
The accuracy with ADABoost with n_estimator 15 is : 0.928495359805264
The accuracy with ADABoost with n_estimator 20 is : 0.9383842994066636
```



0.0.3 ADABoost performs best for n_estimators = 20

[13]:
```
max_depths=[4,8,10,15,20,30]
training_accuracy=[]
validation_accuracy=[]
testing_accuracy=[]
for j in max_depths:
  clf = DecisionTreeClassifier(criterion = 'entropy', max_depth = j)
  abc = AdaBoostClassifier(base_estimator=clf, n_estimators=20,␣
 ↪learning_rate=1, random_state=0)
  model = abc.fit(X_train, Y_train)
  train_pred = model.predict(X_train)
  validation_pred = model.predict(X_val)
  test_pred = model.predict(X_test)
  training_accuracy.append(accuracy_score(Y_train, train_pred))
  validation_accuracy.append(accuracy_score(Y_val, validation_pred))
  testing_accuracy.append(accuracy_score(Y_test, test_pred))

print("Test Set Accuracies")
for x in range(len(testing_accuracy)):
  print(f"The accuracy with ADABoost with max depth of tree {max_depths[x]} is :
 ↪ {testing_accuracy[x]}")


plt.plot(max_depths,training_accuracy,label = "Train",color="red")
plt.plot(max_depths,validation_accuracy,label = "Validation",color="blue")
plt.plot(max_depths,testing_accuracy,label = "Test",color="green")
plt.xlabel('Max Depth')
plt.ylabel('Accuracy')
plt.title('Accuracies vs Estimator (ADABoost)')
plt.legend()
plt.show()
```

```
Test Set Accuracies
The accuracy with ADABoost with max depth of tree 4 is : 0.4033165982047771
The accuracy with ADABoost with max depth of tree 8 is : 0.6744256808154572
The accuracy with ADABoost with max depth of tree 10 is : 0.8305187889852427
The accuracy with ADABoost with max depth of tree 15 is : 0.928495359805264
The accuracy with ADABoost with max depth of tree 20 is : 0.9383842994066636
The accuracy with ADABoost with max depth of tree 30 is : 0.834778639890461
```



Conclusion : ADABoost with n_estimators = 20 is superior to RF on comparing on various decision tree depths. Our custom implementation was able to reach a max accuracy of about 92 % whereas ADABoost with max depth = 20 and n_estimators = 20 has a testing accuracy of 93.8%. On comparing RF with various tree depths with ADABoost we can see ADABoost performs better or very similar in max_depths=[4,8,10,15,20,30]

[ ]:

```
```

Q2

November 2, 2021

[7]:
```
import matplotlib.pyplot as plt
class NN():
  """
  Neural Network Classifier
  """
  def __init__(self, num_layers, layer_size, activation, lr, weightinit,␣
↪batch_size, epochs):
    self.num_layers, self.layer_size, self.activation, self.lr, self.
↪weightinit, self.batch_size, self.epochs = num_layers, layer_size,␣
↪activation, lr, weightinit, batch_size, epochs

  def activationfn(self,X):
    string=f"self.{self.activation}(X)"
    return eval(string)

  def gradfn(self,X):
    string=f"self.{self.activation}_grad(X)"
    return eval(string)

  def score(self, X, y):
    y_pred = self.predict(X)
    counter=0
    for i in range(len(y_pred)):
      if(y_pred[i]==y):
        counter+=1
    return counter/len(y)

  def initialization(self):
    """
    We make matrices of (layer_size[i], layer_size[i+1]) as we need
    Wji as the params, all combinations that are possible.
    """


    params = {}
    mylayers = self.layer_size
```

```
   for i in range(0,self.num_layers-1):
       params["b" + str(i+1)] = np.zeros((1,mylayers[i+1]))


       if(self.weightinit == 'normal'):
         thislayer = np.random.normal(size = (mylayers[i],mylayers[i+1]))*0.01

       elif(self.weightinit == 'zero'):
         thislayer = np.zeros((mylayers[i],mylayers[i+1]))

       else:
         thislayer = np.random.rand(mylayers[i],mylayers[i+1])*0.01

       params["W" + str(i+1)] = thislayer


   self.params = params

 def crossentropyloss(self, Amatrix, Y):
     temp=Amatrix[np.arange(len(Y)), Y.argmax(axis=1)]
     temp=np.where(temp>0.000000000000001,temp,0.000000000000001)
     logp = - np.log(temp)
     celoss = np.sum(logp)/len(Y)
     return celoss

 def forward_prop(self,X,params):

     A = X
     myactivations = {}
     before_activation = {}
     numhiddenlayers=self.num_layers-2
     for i in range(numhiddenlayers):
         A_prev = A
         Z = np.dot(A_prev, params["W" + str(i+1)]) + params["b" + str(i+1)]
         before_activation["Z" + str(i+1)] = Z
         A=self.activationfn(Z)
         myactivations["A" + str(i+1)] = A
         A_prev = A


     ZL = np.dot(A_prev, params["W" + str(numhiddenlayers+1)]) + params["b" + str(numhiddenlayers+1)]
     AL = (np.exp(ZL)/(np.sum(np.exp(ZL),axis = 1, keepdims = True)))   # SoftMax
     myactivations["A" + str(numhiddenlayers+1)] = AL
     before_activation["Z" + str(numhiddenlayers+1)] = ZL
```

```
        return AL, myactivations, before_activation
    def backward_prop(self, X, Y, before_activation, myactivations):
        """
        Complete Backward Prop
        """
        gradients = {}
        Lay = self.num_layers-1
        myactivations["A0"] = X


        A = myactivations["A" + str(Lay)]
        dZ = A - Y

        dW = np.dot(myactivations["A" + str(Lay-1) ].T, dZ)/len(X)
        db = np.sum(dZ, axis=0, keepdims=True) / len(X)
        dAp = np.dot(dZ, self.params["W" + str(Lay)].T)

        gradients["db" + str(Lay)] = db
        gradients["dW" + str(Lay)] = dW

        for l in range(Lay - 1, 0, -1):
            dGrad=self.gradfn(before_activation["Z" + str(l)])
            dZ = dAp * dGrad
            dW = (1/len(X)) * np.dot(myactivations["A" + str(l - 1)].T, dZ)
            db = (1/len(X)) * np.sum(dZ, axis=0, keepdims=True)
            if l > 1:
              dAp = np.dot(dZ,self.params["W" + str(l)].T)
            gradients["dW" + str(l)] = dW
            gradients["db" + str(l)] = db

        #Update the params
        for i in range(Lay):
            self.params["W" + str(i+1)] -= self.lr*gradients["dW" + str(i+1)]
            self.params["b" + str(i+1)] -= self.lr*gradients["db" + str(i+1)]





    def fit(self, X, y, x_val, y_val):
        """
        Train the model.
        """
        self.initialization()
        self.classes=int(np.max(y))
        m=X.shape[0]
        y = self.convertoprobvsclassmatrix(y)
```

```
    train_losses = []
    val_losses = []


    noofbatches=m//self.batch_size
    combined_train_data=[]

    for i in range(0,noofbatches):
      myX=X[self.batch_size*i:self.batch_size*(i+1),:]
      myY=y[self.batch_size*i:self.batch_size*(i+1),:]
      combined_train_data.append((myX,myY))



    for epoch in range(self.epochs):
        print(f"Epoch: {epoch+1} ", end='' )
        trainbatchloss = []
        valbatchloss = []

        for batch_x, batch_y in combined_train_data:
            A, activations, preactivations = self.forward_prop(batch_x,self.
↪params)
            train_cost = self.crossentropyloss(A,batch_y)
            trainbatchloss.append(train_cost)
            self.backward_prop(batch_x,batch_y,preactivations, activations )
            proba = self.predict_proba(x_val)
            valloss = self.crossentropyloss(proba, self.
↪converttoprobvsclassmatrix(y_val))
            valbatchloss.append(valloss)



        train_losses.append(np.array(trainbatchloss).mean())
        val_losses.append(np.array(valbatchloss).mean())



    self.train_losses = train_losses
    self.val_losses = val_losses


def converttoprobvsclassmatrix(self, y):
    m = len(y)
    myy = np.zeros((int(m),self.classes+1))
    for i in range(m):
        l = int(y[i])
```

```
        myy[i,l] = 1
    return myy
def predict_proba(self, inp_X):
    proba,temp,temp2 = self.forward_prop(inp_X,self.params)
    return proba
def predict(self, X):
    proba = self.predict_proba(X)
    y_pred = np.argmax(proba, axis = 1)
    return y_pred

def score(self, X, y):
    y_pred = self.predict(X)
    counter=0

    for i in range(len(y_pred)):
      if(y_pred[i]==y[i]):
        counter+=1
    return counter/len(y)

def relu(self, X):
    return X * (X>=0)

def relu_grad(self, X):
    return 1*(X>=0)

def leakyrelu(self, X):
    return np.where(X > 0, X, X * 0.01)
def leakyrelu_grad(self,z):
    f = np.maximum(z, 0.01*z)
    grad = np.where(f>0, 1, 0.01)
    return grad


def sigmoid(self, X):
    return 1/(1+np.exp(-X))

def sigmoid_grad(self, X):
    return self.sigmoid(X) *(1-self.sigmoid (X))

def linear(self, X):
    return X

def linear_grad(self, X):
    return np.ones(X.shape)

def tanh(self, X):
    return np.tanh(X)
```

```
  def tanh_grad(self, X):
    return 1-self.tanh(X)*self.tanh(X)

  def softmax(self, X):
    exp = np.exp(X)
    return exp/(np.sum(exp,axis = 1, keepdims = True))

  def softmax_grad(self, X):
    return self.softmax(X) *(1-self.softmax(X))
```

Data Preprocessing

1. Converted the 28x28 images to 784 features for every image.
2. Tried Normalization - Both Min-Max and Gaussian Normalization i tried but they gave worse results.

[2]:
```
import numpy as np
import idx2numpy

dir="train-images.idx3-ubyte"
train_X = idx2numpy.convert_from_file(dir)
dir="train-labels.idx1-ubyte"
train_Y = idx2numpy.convert_from_file(dir)
dir="t10k-images.idx3-ubyte"
test_X = idx2numpy.convert_from_file(dir)
dir="t10k-labels.idx1-ubyte"
test_Y = idx2numpy.convert_from_file(dir)
```

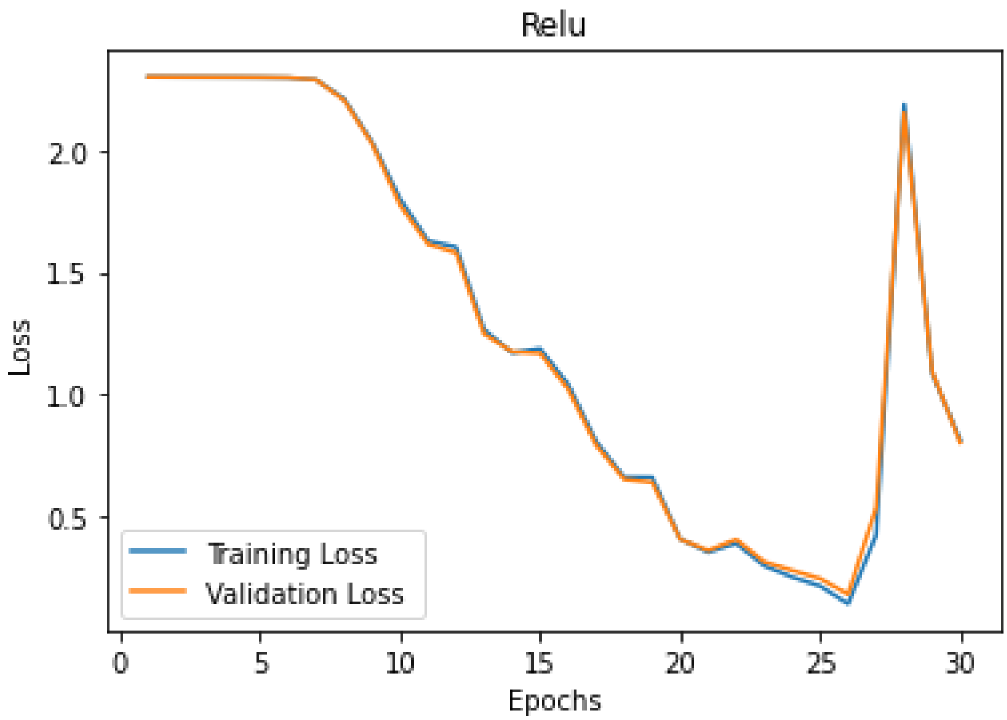
[3]:
```
import pandas as pd
total_data_X=[]
total_data_Y=[]
for i in range(len(train_Y)):
  mylist=list(train_X[i].ravel())
  total_data_X.append(mylist)
  total_data_Y.append(train_Y[i])
for i in range(len(test_Y)):
  mylist=list(test_X[i].ravel())
  total_data_X.append(mylist)
  total_data_Y.append(test_Y[i])

total_data_X=np.array(total_data_X)
#total_data_X = total_data_X/255
total_data_Y=np.array(total_data_Y)
```

```
def mytraintestvalsplit(total,valsize,testsize):
    notestrows=int(testsize*total.shape[0])
    novalrows=int(valsize*total.shape[0])
    notrainrows=total.shape[0]-notestrows-novalrows
    trainrows=total[:notrainrows]
    valrows=total[notrainrows:notrainrows+novalrows]
    testrows=total[notrainrows+novalrows:]
    return trainrows,valrows,testrows

X_train, X_val,X_test = mytraintestvalsplit(total_data_X, 0.2,0.1)
Y_train, Y_val,Y_test = mytraintestvalsplit(total_data_Y, 0.2,0.1)







#Min Max Scaling#
```

[4]:
```
nn = NN(6, [784,256, 128, 64, 32,10], 'relu', 0.08, 'normal', len(X_train)//20,␣
 ↪30)
nn.fit(X_train,Y_train,X_val,Y_val)
print()
print("The Accuracy Score for Relu Activation is with normal Initialization is")
print(nn.score(X_test,Y_test))
```

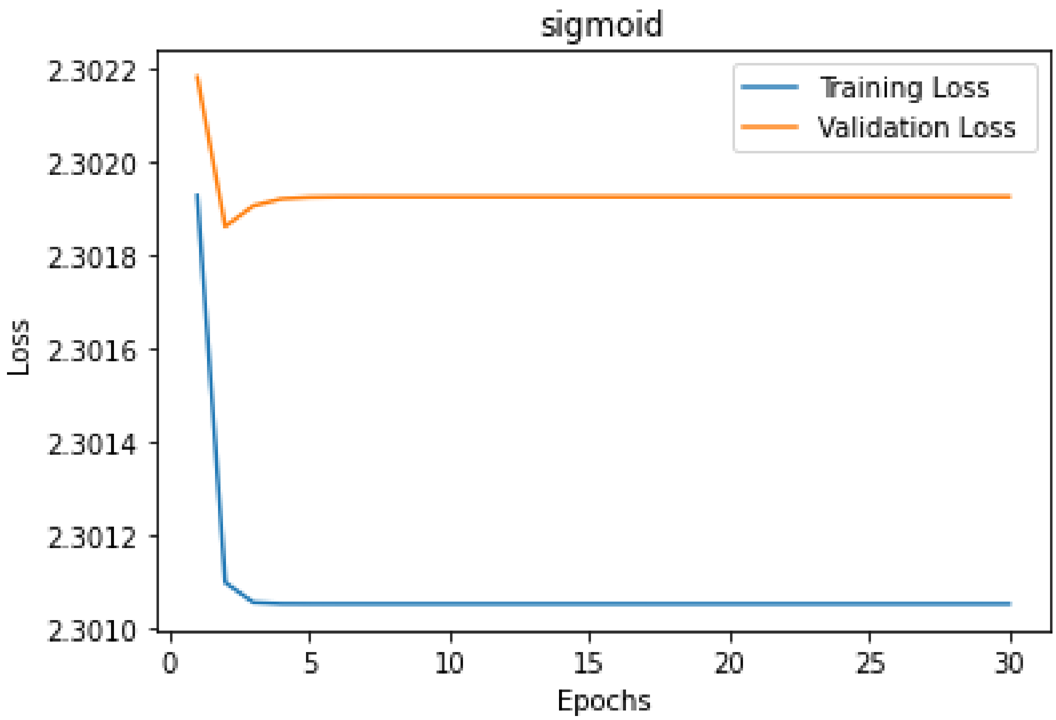
```
Epoch: 1 Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9
Epoch: 10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17
Epoch: 18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25
Epoch: 26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30
The Accuracy Score for Relu Activation is with normal Initialization is
0.8475714285714285
```

[5]:
```
alltheweights={}
alltheweights['relu'] = nn.params
plt.plot(list(range(1,len(nn.train_losses) + 1)),nn.train_losses, label =␣
 ↪"Training Loss " )
plt.plot(list(range(1,len(nn.val_losses) + 1)),nn.val_losses, label =␣
 ↪"Validation Loss " )
plt.ylabel('Loss')
plt.legend()
plt.xlabel('Epochs')
plt.title("Relu")
plt.show()
```

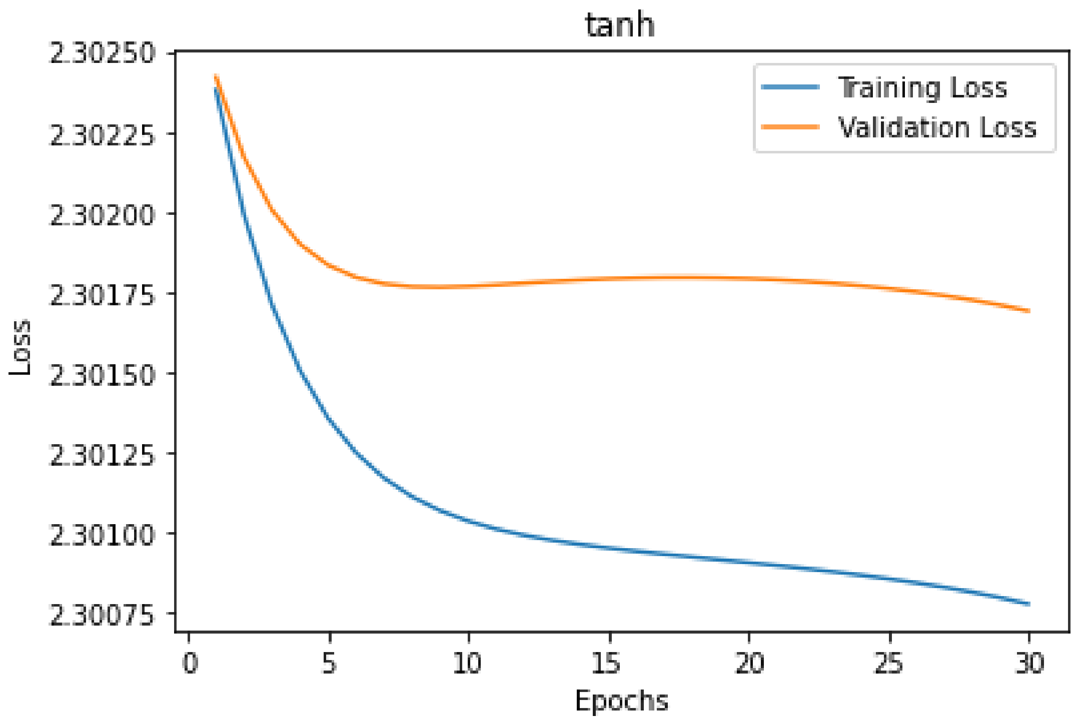


[6]:
```
alltheactivations=['sigmoid','tanh','linear','leakyrelu', 'softmax']

for x in alltheactivations:
  nn = NN(6, [784,256, 128, 64, 32,10], x, 0.08, 'normal', len(X_train)//20, 30)
  nn.fit(X_train,Y_train,X_val,Y_val)
  print(f"The Accuracy Score for {x} activation fn is :")
  print(nn.score(X_test,Y_test))
  alltheweights[x] = nn.params
  plt.plot(list(range(1,len(nn.train_losses) + 1)),nn.train_losses, label =␣
↪"Training Loss " )
  plt.plot(list(range(1,len(nn.val_losses) + 1)),nn.val_losses, label =␣
↪"Validation Loss " )
  plt.ylabel('Loss')
  plt.legend()
  plt.xlabel('Epochs')
  plt.title(x)
  plt.show()
```

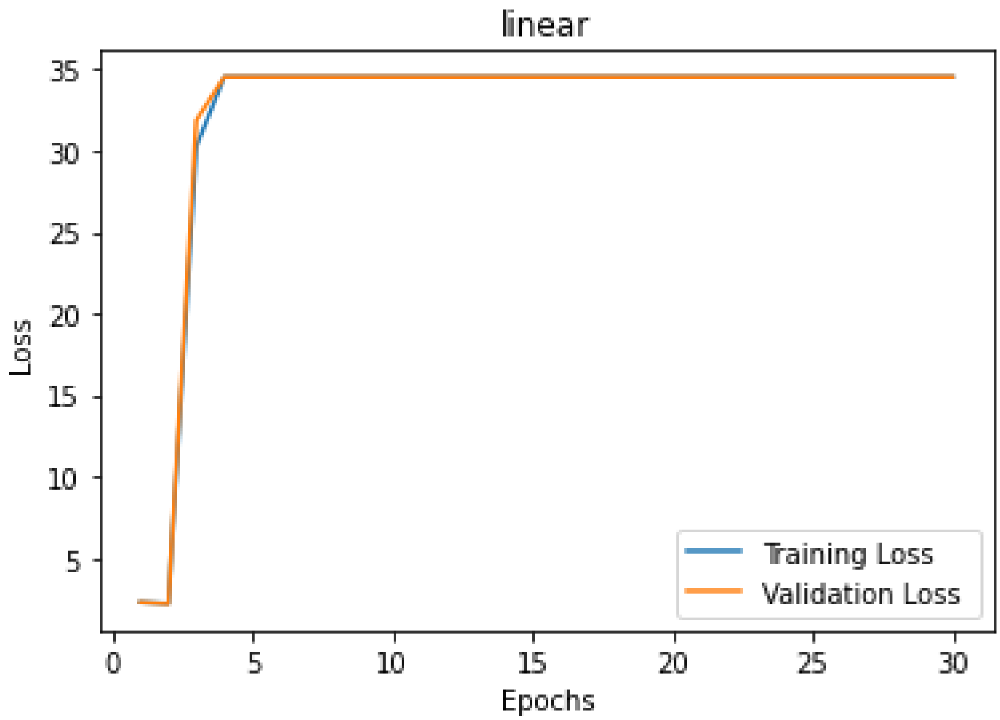
```
Epoch: 1 Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9
Epoch: 10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17
Epoch: 18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25
Epoch: 26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30 The Accuracy Score for sigmoid
activation fn is :
0.11357142857142857
```




```
Epoch: 1 Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9
Epoch: 10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17
Epoch: 18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25
Epoch: 26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30 The Accuracy Score for tanh
activation fn is :
0.11357142857142857
```




```
Epoch: 1 Epoch: 2 Epoch: 3

C:\Users\Pritish\AppData\Local\Temp/ipykernel_14372/609547596.py:76:
RuntimeWarning: overflow encountered in exp
  AL = (np.exp(ZL)/(np.sum(np.exp(ZL),axis = 1, keepdims = True)))  # SoftMax
C:\Users\Pritish\AppData\Local\Temp/ipykernel_14372/609547596.py:76:
RuntimeWarning: invalid value encountered in true_divide
  AL = (np.exp(ZL)/(np.sum(np.exp(ZL),axis = 1, keepdims = True)))  # SoftMax

Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9 Epoch: 10 Epoch: 11 Epoch:
12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17 Epoch: 18 Epoch: 19 Epoch:
20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25 Epoch: 26 Epoch: 27 Epoch:
28 Epoch: 29 Epoch: 30 The Accuracy Score for linear activation fn is :
0.10128571428571428
```




```
Epoch: 1
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_14372/252400328.py in <module>
      3 for x in alltheactivations:
      4     nn = NN(6, [784,256, 128, 64, 32,10], x, 0.08, 'normal', len(X_train)/
 ↪20, 30)
----> 5     nn.fit(X_train,Y_train,X_val,Y_val)
      6     print(f"The Accuracy Score for {x} activation fn is :")
      7     print(nn.score(X_test,Y_test))

~\AppData\Local\Temp/ipykernel_14372/609547596.py in fit(self, X, y, x_val,␣
 ↪y_val)
    149                 train_cost = self.crossentropyloss(A,batch_y)
    150                 trainbatchloss.append(train_cost)
--> 151                 self.backward_prop(batch_x,batch_y,preactivations,␣
 ↪activations )
    152                 proba = self.predict_proba(x_val)
    153                 valloss = self.crossentropyloss(proba, self.
 ↪converttoprobvsclassmatrix(y_val))

~\AppData\Local\Temp/ipykernel_14372/609547596.py in backward_prop(self, X, Y,␣
 ↪before_activation, myactivations)
```

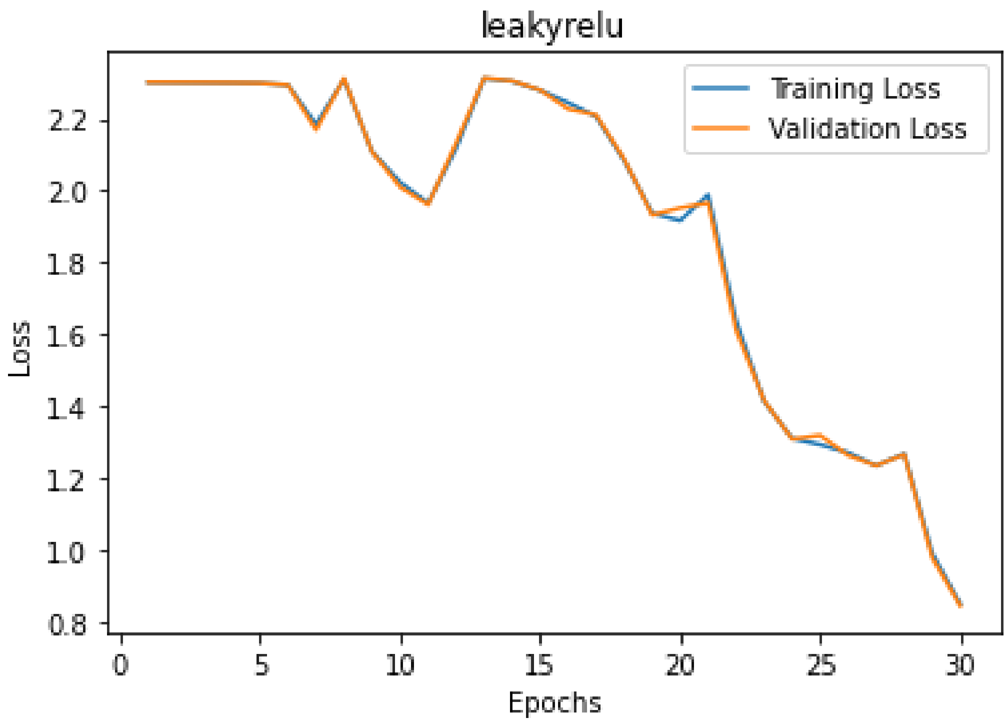
```
     99
    100         for l in range(Lay - 1, 0, -1):
--> 101             dGrad=self.gradfn(before_activation["Z" + str(l)])
    102             dZ = dAp * dGrad
    103             dW = (1/len(X)) * np.dot(myactivations["A" + str(l - 1)].T, d )

~\AppData\Local\Temp/ipykernel_14372/609547596.py in gradfn(self, X)
     13     def gradfn(self,X):
     14       string=f"self.{self.activation}_grad(X)"
---> 15       return eval(string)
     16
     17     def score(self, X, y):

<string> in <module>

~\AppData\Local\Temp/ipykernel_14372/609547596.py in leakyrelu_grad(self, X)
    200     def leakyrelu_grad(self,X):
    201       dx = np.ones_like(x)
--> 202       dx[X < 0] = 0.01
    203       return dx
    204

IndexError: too many indices for array: array is 0-dimensional, but 2 were␣
 ↪indexed
```

[8]:

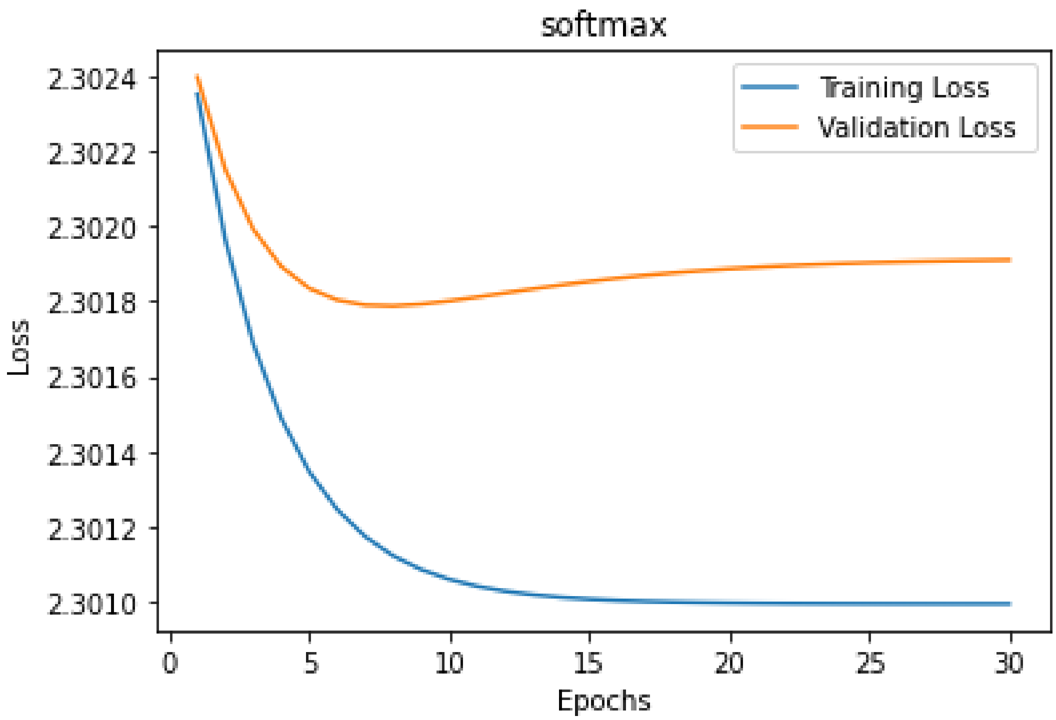
```python
#Re ran leaky relu because of error

alltheactivations=['leakyrelu', 'softmax']

for x in alltheactivations:
  nn = NN(6, [784,256, 128, 64, 32,10], x, 0.08, 'normal', len(X_train)//20, 30)
  nn.fit(X_train,Y_train,X_val,Y_val)
  print(f"The Accuracy Score for {x} activation fn is :")
  print(nn.score(X_test,Y_test))
  alltheweights[x] = nn.params
  plt.plot(list(range(1,len(nn.train_losses) + 1)),nn.train_losses, label =␣
 ↪"Training Loss " )
  plt.plot(list(range(1,len(nn.val_losses) + 1)),nn.val_losses, label =␣
 ↪"Validation Loss " )
  plt.ylabel('Loss')
  plt.legend()
  plt.xlabel('Epochs')
  plt.title(x)
  plt.show()
```

```
Epoch: 1 Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9
Epoch: 10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17
```

```
Epoch: 18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25
Epoch: 26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30 The Accuracy Score for
leakyrelu activation fn is :
0.7015714285714286
```




```
Epoch: 1 Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9
Epoch: 10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17
Epoch: 18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25
Epoch: 26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30 The Accuracy Score for softmax
activation fn is :
0.11357142857142857
```



(b) From the above test accuracies, it is quite evident that “ReLU” works best. It has the best test accuracy.

The worst performance is shown in the case of the “linear” activation function as the accuracy is least in that case. It does not perform any activation and is suitable for single layers only, in our multi-layer case it does not perform well.

For “tanh”, also we weren’t able to make a good model. Maybe with more epochs we could get a desired result, but since i was low on computation power. I used only 30 epochs for these experiments.

For “relu”, it shows the best behaviour it reacts very fast to the changes and to the data. The only noticeable problem is that after certain number of epochs we tend to go away from the global

For “linear”, the training loss keeps in decreasing while validation loss fluctuates a bit and then stablizes.

For “sigmoid”, it was stuck at a local minima for a long time and the moment it got out, the training loss started decreasing while validation loss shot and then decreased to a nominal value.

(c) In all cases the output layer should have an activation function of “softmax” with the number of nodes = the number of classes i.e. 10 (0-9), as it converts the outputs to probabilities out of which the highest is taken to get the predicted label.

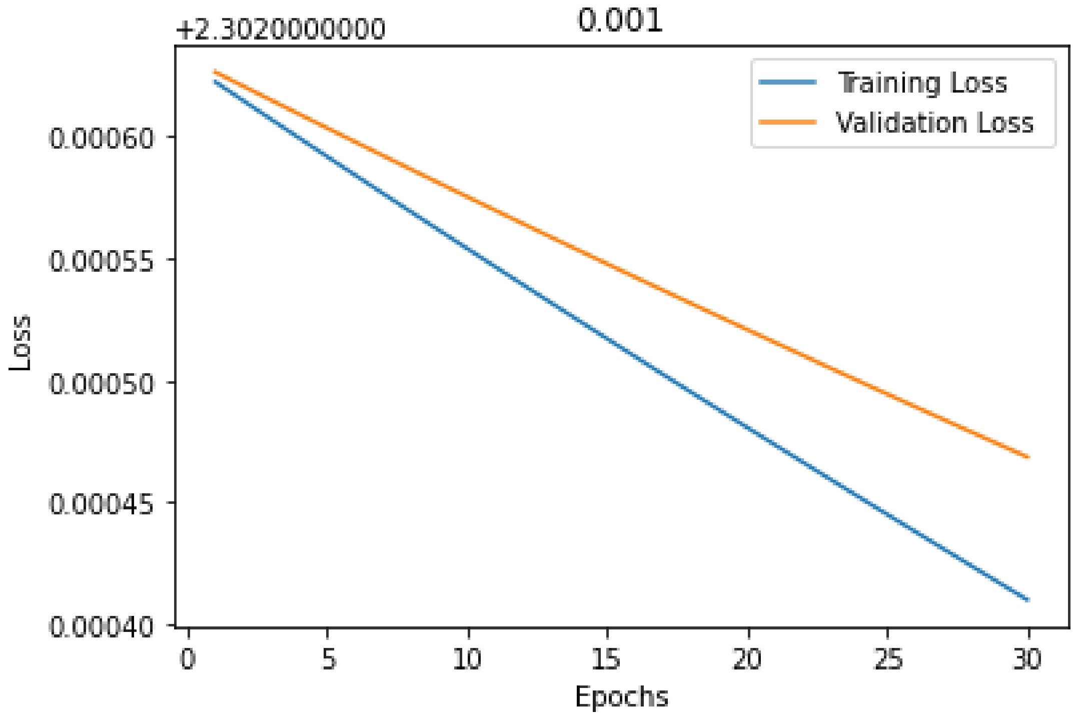
[ ]:
```
for activation in ["logistic", "tanh", "identity", "relu"]:
    nn = MLPClassifier(activation=activation, hidden_layer_sizes=[256, 128, 64,␣
 ↪32], learning_rate_init=0.08, max_iter=30, solver="sgd", alpha = 0)
    nn.fit(X_train, Y_train)
    print(f'Test accuracy for {activation} = {nn.score(X_test, Y_test)}')
```

Test accuracy for logistic = 0.814 Test accuracy for tanh = 0.103 Test accuracy for identity = 0 Test accuracy for relu = 0.869523809523809

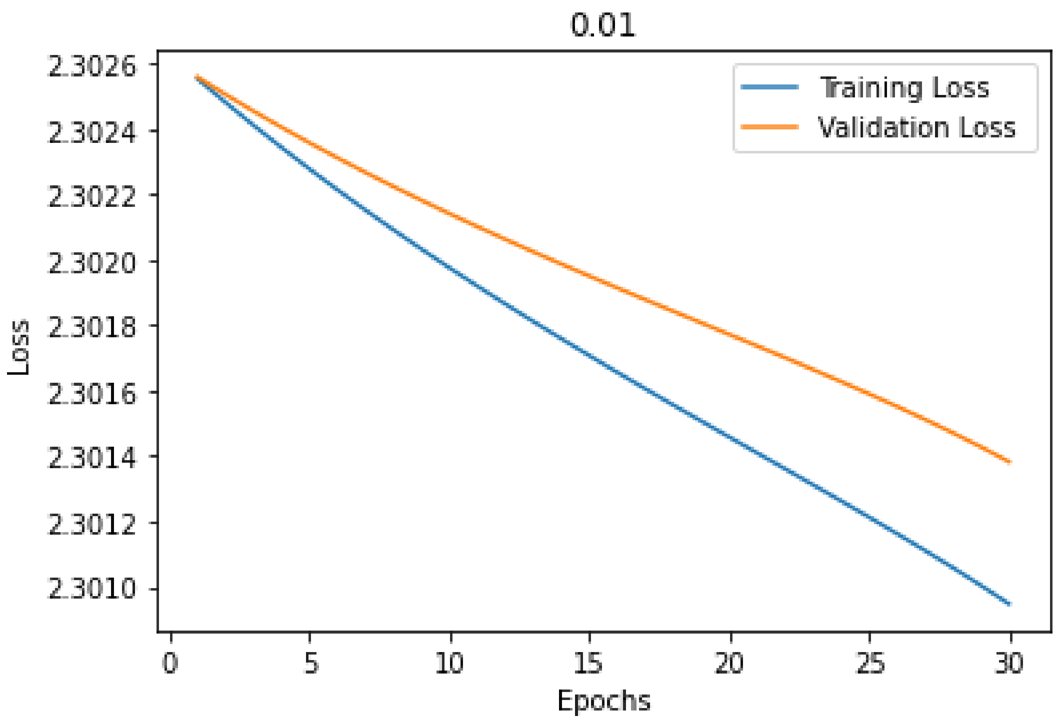
(d) In case of sigmoid (logistic) sklearn implementation is far superior. In the case of "linear", the accuracy on test in custom implementation comes out to be better than the one in sklearn's as it did not converge with the given parameters. In case of tanh we get similar results as 30 epochs are not sufficient which is what we have taken for our experiments. Sklearn also performs best for relu just like ours and the accuracies are also closeby.

[9]:
```
#Bonus
lrs=[0.001,0.01,0.1,1]

for x in lrs:
  nn = NN(6, [784,256, 128, 64, 32,10], 'relu', x, 'normal', len(X_train)//20,␣
 ↪30)
  nn.fit(X_train,Y_train,X_val,Y_val)
  print()
  print(f"The Accuracy Score for {x} learning rate is :")
  print(nn.score(X_test,Y_test))
  alltheweights[x] = nn.params
  plt.plot(list(range(1,len(nn.train_losses) + 1)),nn.train_losses, label =␣
 ↪"Training Loss " )
  plt.plot(list(range(1,len(nn.val_losses) + 1)),nn.val_losses, label =␣
 ↪"Validation Loss " )
  plt.ylabel('Loss')
  plt.legend()
  plt.xlabel('Epochs')
  plt.title(x)
  plt.show()
```

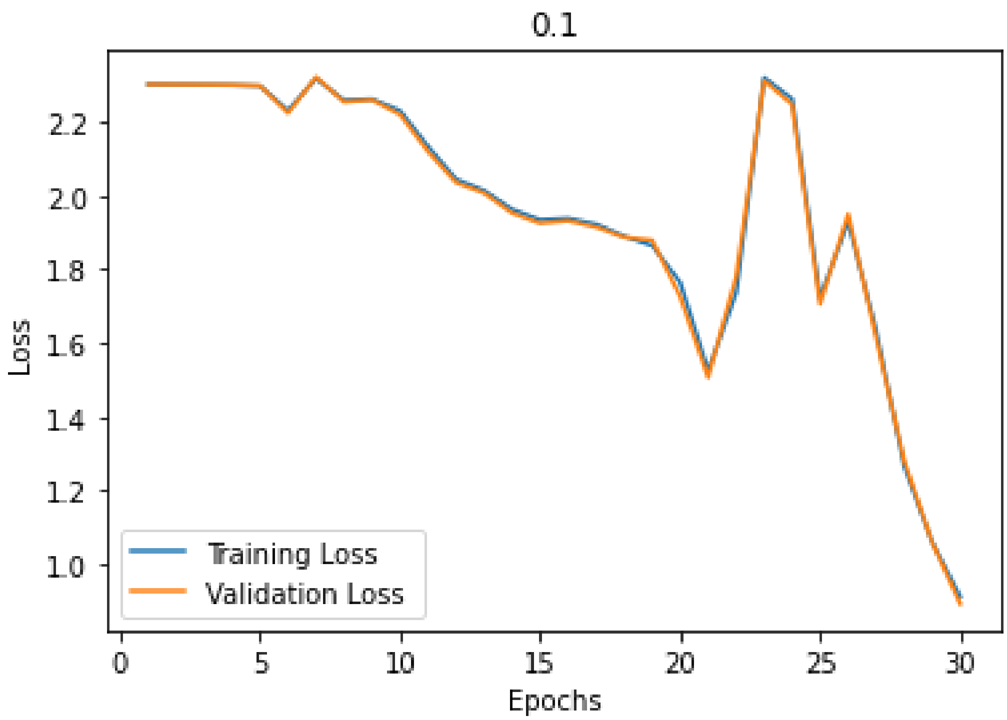
```
Epoch: 1 Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9
Epoch: 10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17
Epoch: 18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25
Epoch: 26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30
The Accuracy Score for 0.001 learning rate is :
0.11357142857142857
```



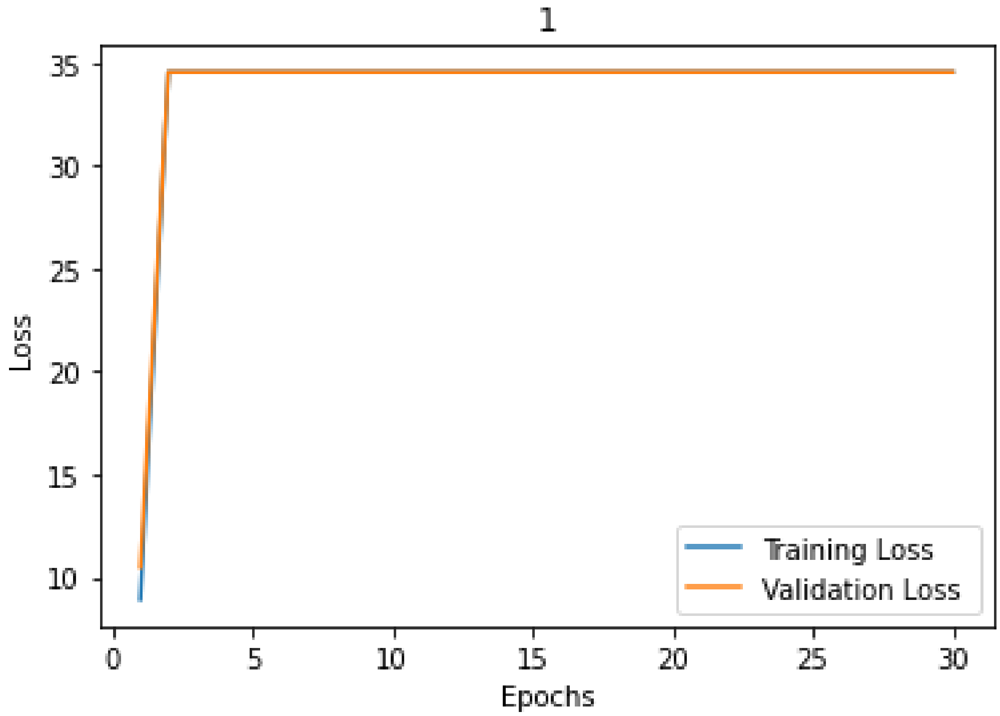

```
Epoch: 1 Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9
Epoch: 10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17
Epoch: 18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25
Epoch: 26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30
The Accuracy Score for 0.01 learning rate is :
0.11357142857142857
```




```
Epoch: 1 Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9
Epoch: 10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17
Epoch: 18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25
Epoch: 26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30
The Accuracy Score for 0.1 learning rate is :
0.6661428571428571
```




```
Epoch: 1

C:\Users\Pritish\AppData\Local\Temp/ipykernel_14372/692279156.py:76:
RuntimeWarning: overflow encountered in exp
  AL = (np.exp(ZL)/(np.sum(np.exp(ZL),axis = 1, keepdims = True)))  # SoftMax
C:\Users\Pritish\AppData\Local\Temp/ipykernel_14372/692279156.py:76:
RuntimeWarning: invalid value encountered in true_divide
  AL = (np.exp(ZL)/(np.sum(np.exp(ZL),axis = 1, keepdims = True)))  # SoftMax

Epoch: 2 Epoch: 3 Epoch: 4 Epoch: 5 Epoch: 6 Epoch: 7 Epoch: 8 Epoch: 9 Epoch:
10 Epoch: 11 Epoch: 12 Epoch: 13 Epoch: 14 Epoch: 15 Epoch: 16 Epoch: 17 Epoch:
18 Epoch: 19 Epoch: 20 Epoch: 21 Epoch: 22 Epoch: 23 Epoch: 24 Epoch: 25 Epoch:
26 Epoch: 27 Epoch: 28 Epoch: 29 Epoch: 30
The Accuracy Score for 1 learning rate is :
0.10128571428571428
```

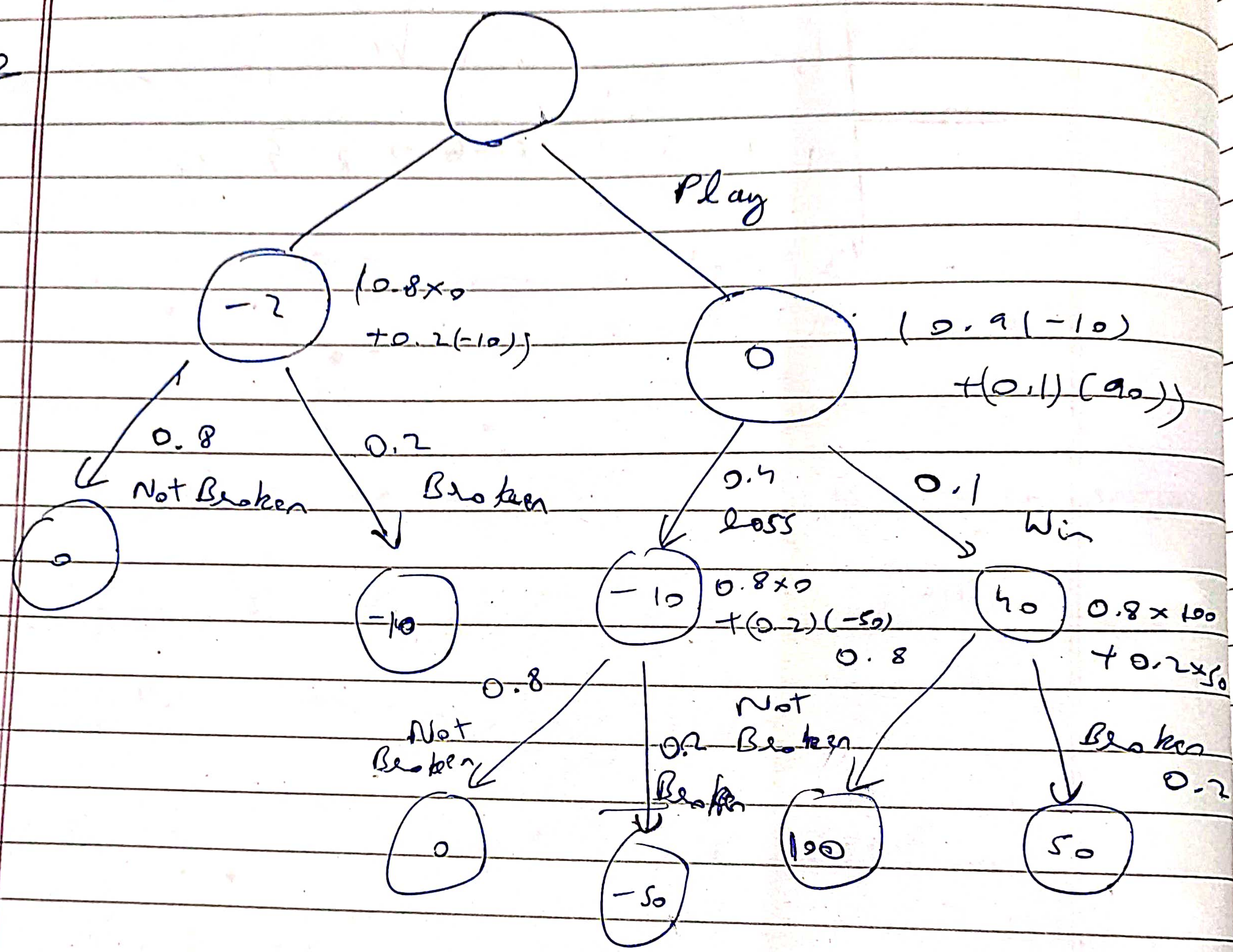


0.1 learning rate works best with 30 epochs, with hidden layers as [256, 128, 64, 32], batch size = len(X_train)//20, activation = ReLU. This worked best, as it is the optimum step the model takes after every step. So, it is able to reach the global minima faster and efficiently. It does not get stuck at the local minima like 0.001 and does not pass fast like 1.

```
[ ]:
```

Q3

1,2

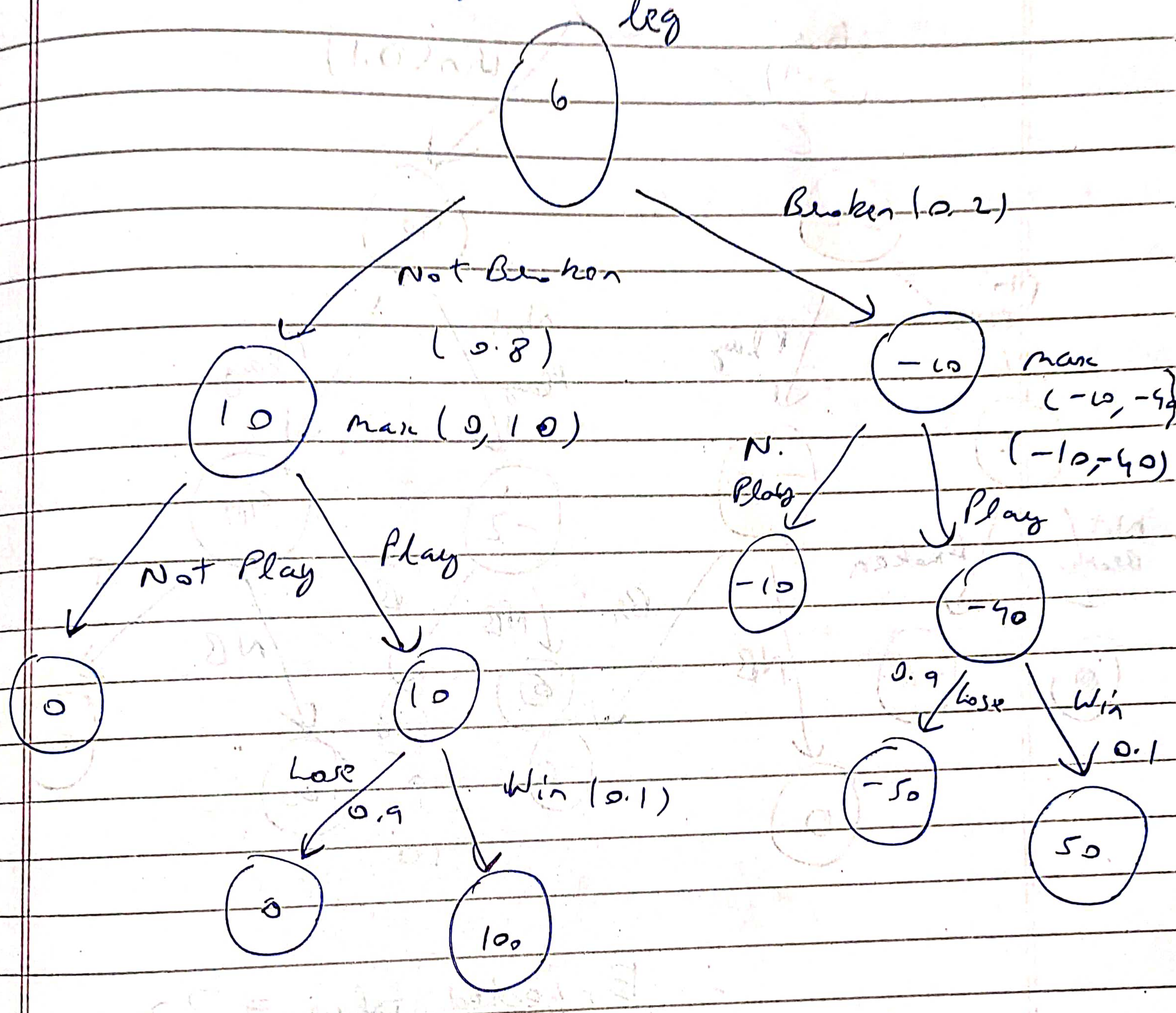


E(play) = 0
E(not play) = -2

∴ We should play

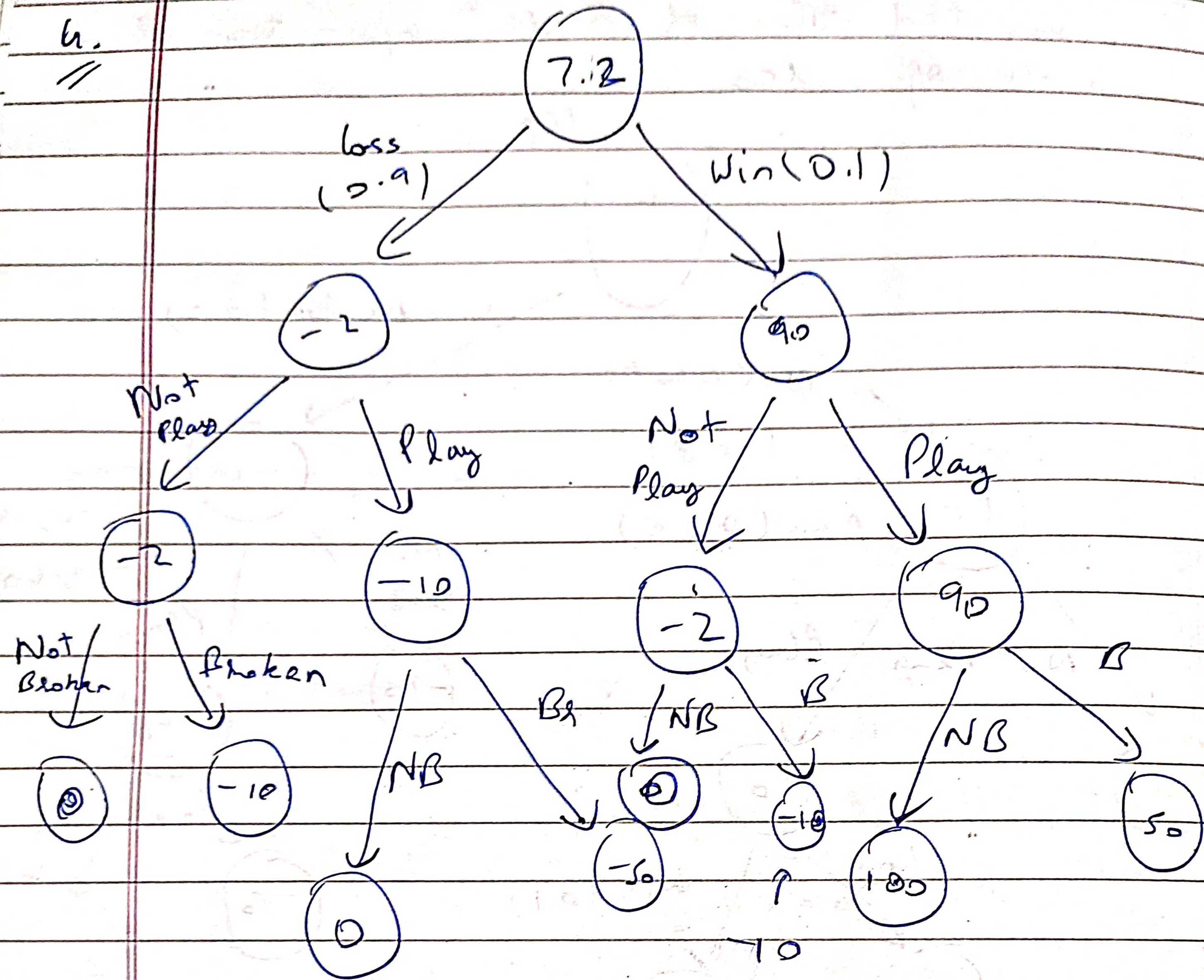
Date : / /
Page No.

3. Expected value of perfect information about state of leg



∴ Expected value of perfect information about state of leg = 6

4.



$\therefore$ Expected value = 72

Q5 Yes, on putting the win branch after the Broken branch.

Date : / /
Page No.

Q2 Given $x \in \{0,1\}^d$

We have $x = [x_1, x_2 \ldots x_d]$
where $x_i \in \{0,1\}$

Now, we know that the conjunction operator can be modelled by a single neuron.

$\Rightarrow$ We can model all possible 2^d conjunctions as "possible" inputs thus making size of hidden layer as 2^d.

Now, since we have all possible conjunctions for a certain d-dimensional input we can simply ~~taken~~ take the disjunction of all the cases we want to include in the final function {Inverse K-Map}

~~So~~ Since, a disjunction can be modelled by a single neuron,

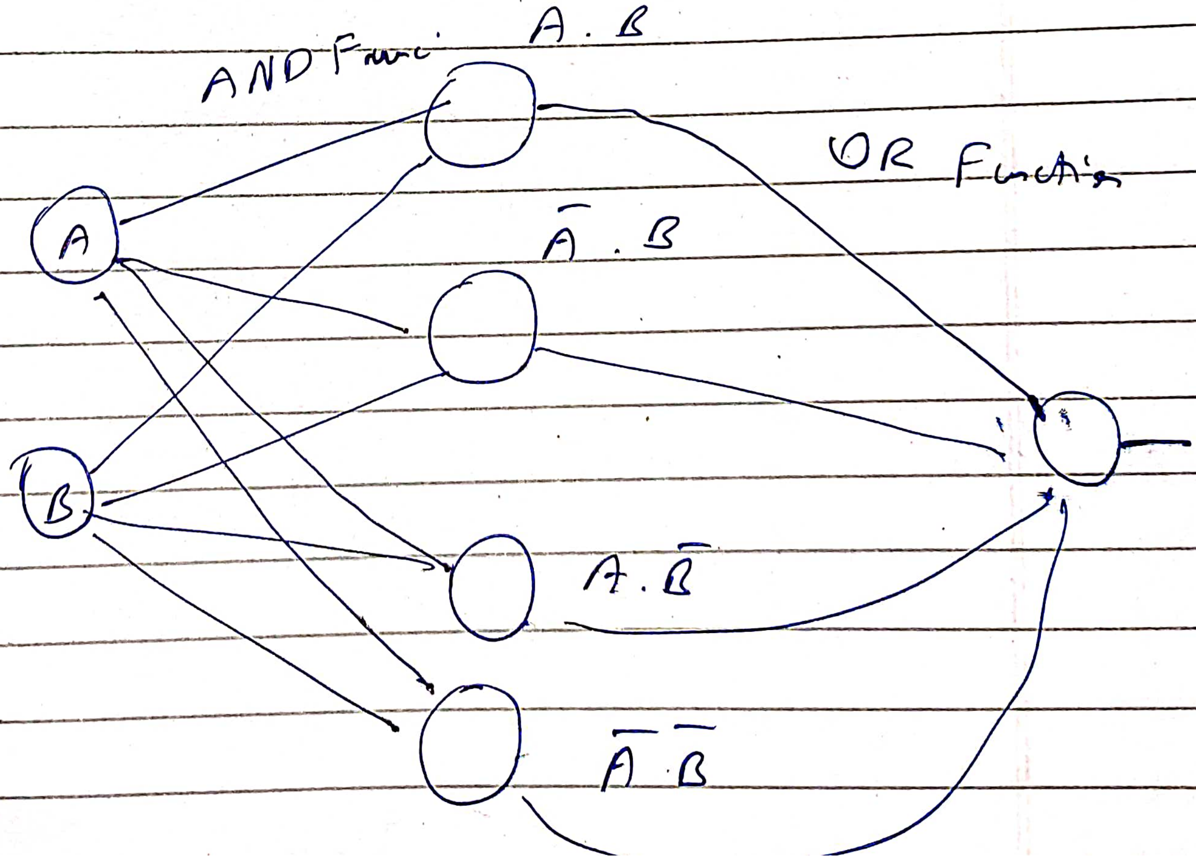
$\therefore$ The second layer can act as a selective disjunction

Hence, we split the listing of all possible functions in two phases, each of which use a single layer

x^i (Input) → Permutational conjunction (Phase 1) 2^d nodes in this layer → Selective Disjunction (Phase 2) 1 node in this layer (output)

Hence, a 2-layered NN can approximately model any function. When dealing with ~~to~~ boolean values, a hard ~~thresh~~ threshhold activation can be used.

Example



Q3

$$P(Y=1 \mid X) = \frac{e^{\beta_1 x_1 + \beta_2 x_2}}{1 + e^{\beta_1 x_1 + \beta_2 x_2}}$$

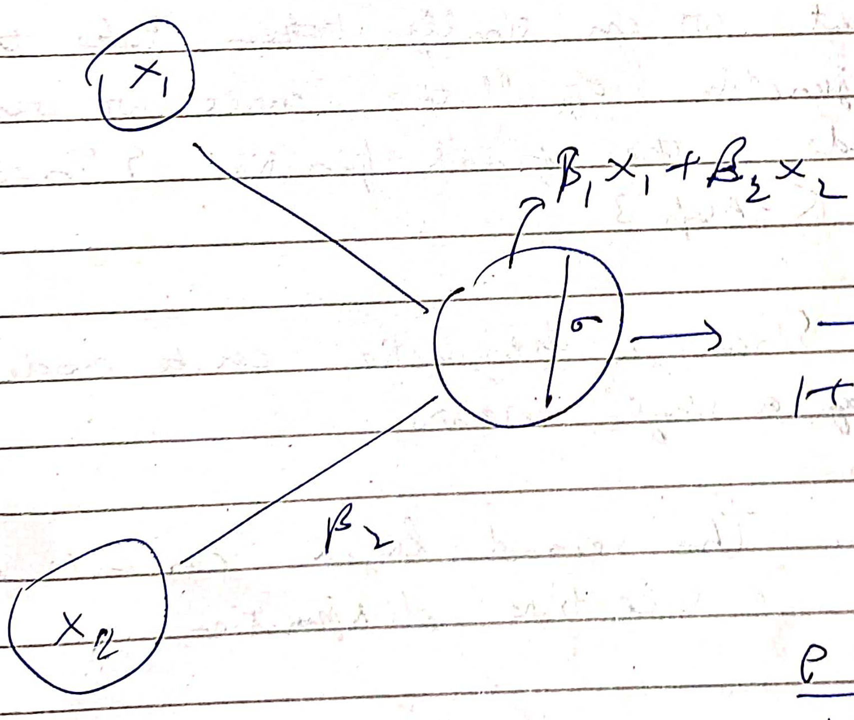
$$= \frac{1}{1 + \exp(-\beta_1 x_1 - \beta_2 x_2)}$$

$$\downarrow$$

$$\sigma(\beta_1 x_1 + \beta_2 x_2)$$

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \qquad W = [\beta_1 \ \ \beta_2]$$

$$z = WX = [\beta_1 x_1 + \beta_2 x_2]$$



$$\rightarrow \frac{1}{1 + \exp(-\beta_1 x_1 - \beta_2 x_2)}$$

$$\downarrow$$

$$\frac{e^{\beta_1 x_1 + \beta_2 x_2}}{1 + e^{\beta_1 x_1 + \beta_2 x_2}}$$