University of New Brunswick

Programming Report

Students:
Mohammad Meymani 3785776
Michael Odartei Mills 3789215

Instructor:
Professor Huajie Zhang

October 30, 2024

Contents

1 Part 1

1.1 Adaboost (Base-learner: ID3)

1.1.1 Importing Necessary Libraries

```
!pip install ucimlrepo
```

```
Collecting ucimlrepo
  Downloading ucimlrepo-0.0.7-py3-none-any.whl.metadata (5.5 kB)
Requirement already satisfied: pandas>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2.2.2)
Requirement already satisfied: certifi>=2020.12.5 in /usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2024.8.30)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=1.0.0->ucimlrepo) (1.16.0)
Downloading ucimlrepo-0.0.7-py3-none-any.whl (8.0 kB)
Installing collected packages: ucimlrepo
Successfully installed ucimlrepo-0.0.7
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import math
from collections import Counter

from ucimlrepo import fetch_ucirepo
```

At first, we imported ucimlrepo library to fetch the data, sklearn.model_selection.train_test_split to split and shuffle the train and test part.

1.1.2 Loading the Data

```
letter_recognition = fetch_ucirepo(id=59)
dataFrame = pd.DataFrame(data=letter_recognition.data.features, columns=letter_recognition.data.features.columns)
dataFrame['target'] = letter_recognition.data.targets
dataFrame[10:20]
```

	x-box	y-box	width	high	onpix	x-bar	y-bar	x2bar	y2bar	xybar	x2ybr	xy2br	x-ege	xegvy	y-ege	yegvx	target
10	3	9	5	7	4	8	7	3	8	5	6	8	2	8	6	7	X
11	6	13	4	7	4	6	7	6	3	10	7	9	5	9	5	8	O
12	4	9	6	7	6	7	8	6	2	6	5	11	4	8	7	8	G
13	6	9	8	6	9	7	8	6	5	7	5	8	8	9	8	6	M
14	5	9	5	7	6	6	11	7	3	7	3	9	2	7	5	11	R
15	6	9	5	4	3	10	6	3	5	10	5	7	3	9	6	9	F
16	3	4	4	3	2	8	7	7	5	7	6	8	2	8	3	8	O
17	7	10	5	5	2	6	8	6	8	11	7	11	2	8	5	9	C
18	6	11	6	8	5	6	11	5	6	11	9	4	3	12	2	4	T
19	2	2	3	3	1	10	6	3	6	12	4	9	0	7	1	7	J

We loaded the data into letter_recognition variable and created a data frame based on the data, and then printed instances 10 to 19 of the data.

1.1.3 Getting Familiar with Data

```
print(dataFrame.isnull().sum())
print(dataFrame.describe())
dataFrame['target'].value_counts()
```

```
x-box     0
y-box     0
width     0
high      0
onpix     0
x-bar     0
y-bar     0
x2bar     0
y2bar     0
xybar     0
x2ybr     0
xy2br     0
x-ege     0
xegvy     0
y-ege     0
yegvx     0
target    0
dtype: int64
```

	x-box	y-box	width	high	onpix
count	20000.000000	20000.000000	20000.000000	20000.00000	20000.000000
mean	4.023550	7.035500	5.121850	5.37245	3.505850
std	1.913212	3.304555	2.014573	2.26139	2.190458
min	0.000000	0.000000	0.000000	0.00000	0.000000
25%	3.000000	5.000000	4.000000	4.00000	2.000000
50%	4.000000	7.000000	5.000000	6.00000	3.000000
75%	5.000000	9.000000	6.000000	7.00000	5.000000
max	15.000000	15.000000	15.000000	15.00000	15.000000

	x-bar	y-bar	x2bar	y2bar	xybar
count	20000.000000	20000.000000	20000.000000	20000.000000	20000.000000
mean	6.897600	7.500450	4.628600	5.178650	8.282050
std	2.026035	2.325354	2.699968	2.380823	2.488475
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	6.000000	6.000000	3.000000	4.000000	7.000000
50%	7.000000	7.000000	4.000000	5.000000	8.000000
75%	8.000000	9.000000	6.000000	7.000000	10.000000
max	15.000000	15.000000	15.000000	15.000000	15.000000

	x2ybr	xy2br	x-ege	xegvy	y-ege
count	20000.00000	20000.000000	20000.000000	20000.000000	20000.000000
mean	6.45400	7.929000	3.046100	8.338850	3.691750
std	2.63107	2.080619	2.332541	1.546722	2.567073
min	0.00000	0.000000	0.000000	0.000000	0.000000
25%	5.00000	7.000000	1.000000	8.000000	2.000000
50%	6.00000	8.000000	3.000000	8.000000	3.000000
75%	8.00000	9.000000	4.000000	9.000000	5.000000
max	15.00000	15.000000	15.000000	15.000000	15.000000

	yegvx
count	20000.00000
mean	7.80120
std	1.61747
min	0.00000
25%	7.00000
50%	8.00000
75%	9.00000
max	15.00000

	count

target	
U	813
D	805
P	803
T	796
M	792
A	789
X	787
Y	786
N	783
Q	783
F	775
G	773
E	768
B	766
V	764
L	761
R	758
I	755
O	753
W	752

We can see that we do not have any null data, and by using describe() function we can see some statistics regarding our data. Also we can see the occurrence of every label.

1.1.4 Splitting Train & Test

```
x = dataFrame.drop('target', axis=1)
y = dataFrame['target']
xtrain, xtest, ytrain, ytest = train_test_split(x, y, train_size=0.8)
print(xtrain.shape)
print(xtest.shape)
print(ytrain.shape)
print(ytest.shape)
```

```
(16000, 16)
(4000, 16)
(16000,)
(4000,)
```

```
Counter(dataFrame['width'].values)
```

```
Counter({3: 1994,
         6: 3641,
         5: 4262,
         4: 3816,
         13: 6,
         8: 1418,
         2: 1285,
         7: 1946,
         11: 91,
         12: 39,
         9: 679,
         1: 385,
         10: 237,
         0: 195,
         15: 2,
         14: 4})
```

We divided our training and test data with a train ratio of 80% and we can see the size of all training and testing data.

1.1.5 Entropy & Information_gain Functions

```python
#function below calculates entropy for a single attribute on a single node
def entropy(data,attribute):
    res = 0
    frequencyDictionary = Counter(data[attribute].values)
    for key in frequencyDictionary:
        probability = frequencyDictionary[key]/len(data)
        res -= probability * math.log2(probability)
    return res

#function below calculates the information game using the parent and a single child for a single node
def information_gain(data,feature,target_attribute):
    parent_entropy = entropy(data, target_attribute)
    frequencyDictionary = Counter(data[feature].values)
    weighted_entropy = 0
    for key, value in frequencyDictionary.items():
        subset = data[data[feature] == key]
        weighted_entropy += (value/len(data))*entropy(subset, target_attribute)
    return parent_entropy - weighted_entropy
```

In the entropy() function, we calculate a single node entropy. At first, we use Counter() function from collection library to have the frequency of every possible value in a dictionary. Then we calculate the probability for each value using -p*log(p) formula.

In the information_gain() function, we first calculate the entropy of the parent, and for every possible child we calculate the entropy, and after that we get the information gain using weighted entropy. (This function calculate information gain only for a single attribute.)

1.1.6 ID3 Algorithm

```python
#ID3 algorithm to build the tree
def id3(data, features, target_attribute, height=3):
    target_values = data[target_attribute].values
    #If all target values are the same, return the label
    if len(set(target_values)) == 1:
        return target_values[0]
    #At the laef we choose the most repeated value
    if len(features) == 0 or height == 0:
        return Counter(target_values).most_common(1)[0][0]
    #Choosing a feature with the highest information gain
    best_feature = max(features, key=lambda feature: information_gain(data, feature, target_attribute))
    #Our tree is at first has the best feature as the parent and there are no children
    tree = {best_feature: {}}
    #Split the dataset on the best feature and build the subtree
    unique_values = set(data[best_feature])
    remaining_features = [feature for feature in features if feature != best_feature]
    for value in unique_values:
        subset = data[data[best_feature] == value]
        tree[best_feature][value] = id3(subset, remaining_features, target_attribute, height-1)
    return tree

#ID3 classifier to choose instance label
def classify(tree, instance):
    if type(tree) is not dict:
        return tree
    feature = next(iter(tree))
    value = instance[feature]
    if value not in tree[feature]:
        return None
    return classify(tree[feature][value], instance)
```

In id3 function, we build our tree with the height of tree and the nodes are at the zero

level. If a node contains only one label we consider the node as leaf and return the label, if we are at level zero we return the most common label. To decide which feature is the best to split we try every remaining features and retrieve the one with the highest information gain. To find the children for the parents we need to iterate through the features used for splitting and find the next optimal feature to split, then we return a sub-tree with lower height. The classify function returns the label for a single instance.

1.1.7 Adaboost Algorithm

```python
def adaboost(data, features, target_attribute, iterations):
    #at first, every sample has the same weights and the value is 1/n
    weights = [1 / len(data)] * len(data)
    classifiers = []
    alphas = []
    for i in range(iterations):
        # Build a weighted dataset by sampling rows based on their weights
        weighted_data = data.sample(n=len(data), weights=weights, replace=True)
        #creating tree instance
        tree = id3(weighted_data, features, target_attribute, height=5)
        print(tree)
        #error calculation
        error = 0
        for j, row in data.iterrows():
            prediction = classify(tree, row)
            if prediction != row[target_attribute]:
                error += weights[j]
        # Calculate the alpha (influence of this classifier)
        if error == 0:
            error = 1e-10  # Avoid division by zero
        alpha = 0.5 * math.log((1 - error) / error)
        #updating the weights
        for j, row in data.iterrows():
            prediction = classify(tree, row)
            if prediction == row[target_attribute]:
                weights[j] *= math.exp(-alpha)
            else:
                weights[j] *= math.exp(alpha)
        #normalizing the weights
        weights_sum = sum(weights)
        for j in range(len(weights)):
          weights[j] = weights[j]/weights_sum
        classifiers.append(tree)
        alphas.append(alpha)
    return classifiers, alphas

#Adaboost Classification
def adaboost_classify(classifiers, alphas, instances):
    predictions = []
    for index in instances.index:
        instance = instances.loc[index].to_dict()
        result = {}
        for alpha, classifier in zip(alphas, classifiers):
          instance_prediction = classify(classifier,instance)
          if instance_prediction not in result:
            result[instance_prediction] = 0
          result[instance_prediction] += alpha
        prediction = max(result,key=result.get)
        predictions.append(prediction)
    return predictions
```

The Adaboost function assigns the same weight to every instance at first. We also have a list of classifiers and alphas. In every iteration we build a tree and use a weighted data based on our original dataset, then we create our first tree based on the weighted data. For every instance, we predict the label and if the label is wrongly predicted we

increase the error, then based on the error rate we can calculate the alpha. By using alpha, we can change the weights for every instance. Then we normalize the weights and append the tree and the final alpha to the corresponding lists.

In the adaboost_classify function we predict the label for every instance using classify function and receive the voting from the trees and add this to the predictions.

1.1.8 Training the Model

```
#training the model
classifiers, alphas = adaboost(dataFrame, x.columns.tolist(), "target", iterations=3)
```

```
{'x-ege': {0: {'xybar': {0: 'L', 1: 'L', 2: 'L', 3: {'x-bar': {10: 'Y', 3: 'L', 4: 'L', 5: 'L'}}, 4: {'y-bar': {5: 'S', 6: 'S', 7: 'S', 8: 'S', 11: 'Y', 14: 'T', 15: 'T'}}, 5:
{'x-ege': {0: {'xybar': {0: 'L', 1: 'L', 2: 'L', 3: {'x-box': {2: 'L', 3: 'Y'}}, 4: {'y-bar': {5: 'S', 6: 'S', 7: 'S', 8: 'S', 11: 'Y', 12: 'Y', 14: 'T'}}, 5: {'x2ybr': {4: 'E'
{'xy2br': {0: 'P', 1: 'P', 2: {'xegvy': {6: 'L', 7: 'L', 8: 'D', 9: 'P', 10: {'y-box': {2: 'P', 4: 'P', 5: 'P', 6: 'P', 7: 'P', 8: 'P', 9: 'P', 10: 'F', 11: 'P'}}, 11: 'P', 12:
```

We trained the model using id3 algorithm and created 3 trees and 3 alphas, we used voting to decide the classification, then we printed our final trees.

1.1.9 Testing the Model (Accuracy Calculation)

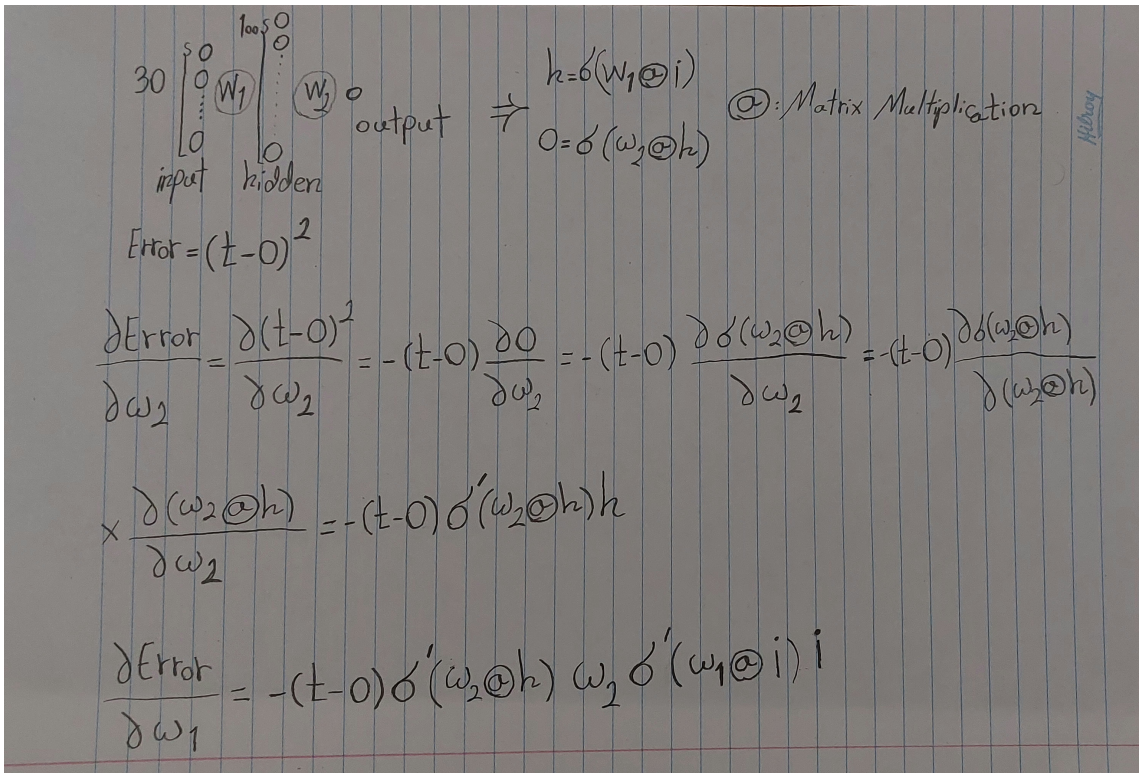
```
#testing the model
yprediction = adaboost_classify(classifiers, alphas, xtest)
print(ytest == yprediction)
print(sum(ytest == yprediction))
print('Accuracy:',sum(ytest == yprediction)/len(ytest))
```

```
5031     True
1884     True
2658     True
5431     True
3091     True
         ...
12540    True
7113     True
6831     True
10192    True
13202    True
Name: target, Length: 4000, dtype: bool
3965
Accuracy: 0.99125
```

We calculated predicted values and compared them to the real values, we got **99%** accuracy.

1.2 ANN

1.2.1 Model Architecture



1.2.2 Importing Necessary Libraries

```
import numpy as np
import pandas as pd
import math
from ucimlrepo import fetch_ucirepo
from sklearn import preprocessing
```

Numpy is used for matrix calculations, Pandas for data manipulation, math for extracting the exact value of e, ucimlrepo for receiving the data set, and sklearn.preprocessing for normalizing our attributes.

1.2.3 Reading the Data

```
BCWD_Dataset = fetch_ucirepo(id = 17)

x = BCWD_Dataset.data.features
y = BCWD_Dataset.data.targets
```

Reading dataset of breast cancer using fetch_ucirepo, x as features and y as classifications ('B','M').

1.2.4 Feature Normalization

```
#normalizing the data
#ss = preprocessing.StandardScaler()
#x_normalized = ss.fit_transform(x)
#print(x_normalized)
x_normalized = preprocessing.normalize(x)
print(x_normalized)
```

We need to normalize x values, so our sigmoid function will not receive large amounts and will not diverge to 1.

1.2.5 Creating Numerical Labels

```
#we need to convert the output to numerical values for classification calculations
y_numerical = pd.DataFrame(data = {'Diagnosis':range(len(y))})
for i in range(len(y)):
    if y['Diagnosis'][i] == 'B':
        y_numerical['Diagnosis'][i] = 0
    else:
        y_numerical['Diagnosis'][i] = 1
```

Because we use raw mathematical calculations and derivations for feedforward and backpropagation steps we need to have our output as numerical numbers, so for B we consider 0 and for M we consider 1.

1.2.6 Sigmoid Function Implementation

```
#sigmoid over a single value of data
def sigmoid(x):
    return 1/(1+(math.e)**-x)
#sigmoid derivation over a single value of data
def sigmoid_derivation(x):
    return sigmoid(x)*(1-sigmoid(x))
#adding the sigmoid function to numpy library
sigmoid = np.frompyfunc(sigmoid,1,1)
sigmoid_derivation = np.frompyfunc(sigmoid_derivation,1,1)
```

The formula for sigmoid function is 1/(1+e^x) and the formula for sigmoid derivation is sigmoid(x) * (1-sigmoid(x)). Then we add these functions to numpy to be able to use them on matrices.

1.2.7 Splitting Train and Test

```
x_train,x_test = x_normalized[:500],x_normalized[500:]
y_train,y_test = y_numerical[:500],y_numerical[500:]
```

We used first 500 data on x_normalized,y_numerical for train and the rest for testing.

1.2.8 Creating the Classifier

```
class BCWDClassifier():
    def __init__(self,inputNumber,hiddenLayerNeuronsNumber,
                 outputNeuronsNumber,learningRate):
        self.learningRate = learningRate
        self.Winput2hidden = np.random.normal(size =
         (inputNumber,hiddenLayerNeuronsNumber))
        self.Whidden2output = np.random.normal(size =
         (hiddenLayerNeuronsNumber,outputNeuronsNumber))
    def forward(self,x):
        hiddenNeurons = sigmoid(x @ self.Winput2hidden)
        outputNeurons = sigmoid(hiddenNeurons @ self.Whidden2output)
        return outputNeurons.reshape(1,len(outputNeurons))
        ,hiddenNeurons.reshape(1,len(hiddenNeurons))
        ,self.Winput2hidden,self.Whidden2output
    def updateWeights(self,derivedWinput2hidden,derivedWhidden2output):
        self.Winput2hidden += self.learningRate * derivedWinput2hidden
        self.Whidden2output += self.learningRate * derivedWhidden2output
```

We get a number of neurons for each layer as our input in the init function and randomly assign the weights. In the forward method, we create hidden neurons and output of our neural network. In updateWeights function, we add the learning rate * gradient for w1 and w2.

1.2.9 Calculating the Gradients for W1 and W2

```
def calculateGradient(x,w1,w2,h,err):
    derivedWinput2hidden = x.reshape(len(x),1) @
     (w2.T * sigmoid_derivation(x @ w1))* sigmoid_derivation(h @ w2) * err
    derivedWhidden2output = err * sigmoid_derivation(h @ w2) * h
    return derivedWinput2hidden.astype('float64'),
    derivedWhidden2output.T.astype('float64')
```

We calculate the gradients for w1 and w2 using the values of input, w1, w2, hidden neurons, and output neurons.

1.2.10 Incremental Mode Training

```
for i in range(500):
    for j in range(len(x_train)):
        o,h,w1,w2 = bcwdc.forward(x_train[i])
        dw1,dw2 = calculateGradient(x_train[j],w1,w2,h,y_train.iloc[j][0]-o)
        bcwdc.updateWeights(dw1,dw2)
```

We train our model in 500 iterations, and after training every instance we update the weights.

1.2.11 Testing the Model (Accuracy Calculation)

```
total = len(y_test)
correct = 0
for i in range(total):
    o,_,_,_ = bcwdc.forward(x_test[i])
    if o < 0.5:
        o = 0
    else:
        o = 1
    if o == y_test.iloc[i][0]:
        correct += 1
print('accuracy:',correct/total)
```

```
accuracy: 0.782608695652174
```

We calculate accuracy by dividing the correct predictions by the total number of tests. If our output is below 0.5 we consider it as 0, else wise the output is 1. The final accuracy is **78%**. Because we have one output for binary class classification we could easily set a boundary of 0.5. If we used two outputs, we would need to use argmax to choose the output neuron.

1.3 Naïve Bayes

1.3.1 Importing Necessary Libraries and Loading the Data

```
!pip install ucimlrepo
```

```
Collecting ucimlrepo
  Downloading ucimlrepo-0.0.7-py3-none-any.whl.metadata (5.5 kB)
Requirement already satisfied: pandas>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2.2.2)
Requirement already satisfied: certifi>=2020.12.5 in /usr/local/lib/python3.10/dist-packages (from ucimlrepo) (2024.8.30)
Requirement already satisfied: numpy>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.0.0->ucimlrepo) (2024.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas>=1.0.0->ucimlrepo) (1.16.0)
Downloading ucimlrepo-0.0.7-py3-none-any.whl (8.0 kB)
Installing collected packages: ucimlrepo
Successfully installed ucimlrepo-0.0.7
```

```
from ucimlrepo import fetch_ucirepo
from sklearn.model_selection import train_test_split
import numpy as np

carEvaluationDataset = fetch_ucirepo(id = 19)

dataFeatures = carEvaluationDataset.data.features
dataTargets = carEvaluationDataset.data.targets
```

We used fetch_ucirepo with id of 19 to load the data. Then we divided targets and features.

1.3.2 Getting Familiar with Data

```
print(dataFeatures)
```

```
     buying  maint  doors persons lug_boot safety
0     vhigh  vhigh      2       2    small    low
1     vhigh  vhigh      2       2    small    med
2     vhigh  vhigh      2       2    small   high
3     vhigh  vhigh      2       2      med    low
4     vhigh  vhigh      2       2      med    med
...     ...    ...    ...     ...      ...    ...
1723    low    low  5more    more      med    med
1724    low    low  5more    more      med   high
1725    low    low  5more    more      big    low
1726    low    low  5more    more      big    med
1727    low    low  5more    more      big   high

[1728 rows x 6 columns]
```

```
print(dataTargets)
```

```
      class
0     unacc
1     unacc
2     unacc
3     unacc
4     unacc
...     ...
1723   good
1724  vgood
1725  unacc
1726   good
1727  vgood

[1728 rows x 1 columns]
```

By printing targets and features, we can see that we have 1728 data with 6 features.

1.3.3 Splitting Train & Test Data

```
print(dataTargets.value_counts())
```

```
class
unacc    1210
acc       384
good       69
vgood      65
Name: count, dtype: int64
```

```
#dividing the train and the test sections
xtrain,xtest,ytrain,ytest = train_test_split(dataFeatures,dataTargets,train_size = 0.8,shuffle = True)
```

```
print(xtrain)
```

```
     buying  maint  doors persons lug_boot safety
1248    med    low      4       2      big    low
760    high    low      2       2      med    med
1055    med   high  5more       2    small   high
1188    med    low      2       2    small    low
63    vhigh  vhigh      4       4    small    low
...     ...    ...    ...     ...      ...    ...
1295    med    low  5more    more      big   high
1095    med    med      2       4      big    low
568    high   high      3       2    small    med
179   vhigh   high      4       4      big   high
318   vhigh    med  5more    more      med    low

[1382 rows x 6 columns]
```

```
print(ytrain)
```

```
      class
1248  unacc
760   unacc
1055  unacc
1188  unacc
63    unacc
...     ...
1295  vgood
1095  unacc
568   unacc
179   unacc
318   unacc

[1382 rows x 1 columns]
```

```
print(xtest)
```

```
     buying  maint  doors persons lug_boot safety
815    high    low      4       2      med   high
291   vhigh    med      4    more      med    low
859    high    low  5more    more      med    med
1597    low    med  5more       2      med    med
1072    med   high  5more    more    small    med
...     ...    ...    ...     ...      ...    ...
833    high    low      4    more      med   high
1537    low    med      2    more      big    med
351   vhigh    low      3       2    small    low
319   vhigh    med  5more    more      med    med
503    high  vhigh      4       4      big   high

[346 rows x 6 columns]
```

```
print(ytest)
```

```
      class
815   unacc
291   unacc
859     acc
1597  unacc
1072  unacc
...     ...
833     acc
1537   good
351   unacc
319     acc
503   unacc

[346 rows x 1 columns]
```

By using value_counts() we can see that we have 4 classes, then we split train and test with train size of 80% of data size.

1.3.4 Class Probability

```
#calculating probability of each class
p_unacc = sum(ytrain['class'] == 'unacc')/len(ytrain)
p_acc = sum(ytrain['class'] == 'acc')/len(ytrain)
p_good = sum(ytrain['class'] == 'good')/len(ytrain)
p_vgood = sum(ytrain['class'] == 'vgood')/len(ytrain)
```

```
print(p_unacc,p_acc,p_good,p_vgood)
```

```
0.6975397973950795 0.223589001447178 0.03907380607814761 0.03979739507959479
```

For the training labels, we need to calculate the probability of each class by calculating sum of each class divided by the size of our test data set.

1.3.5 Posterior Probabilities

```
#function below normalizes word counts to the form of probability distribution
def normalize_counts(myDictionary):
    for key in myDictionary:
        keyNum = 0
        totalSum = 0
        for i in myDictionary[key]:
            keyNum += 1
            totalSum += myDictionary[key][i]
        for i in myDictionary[key]:
            myDictionary[key][i] = ((myDictionary[key][i]+1)/(totalSum+keyNum))
    return myDictionary
```

```
#function below claculates probability of each word according to each class
def calculate_word_probability(data,target,column):
    result = dict()
    for i in range(len(data)):
        if data[column].iloc[i] in result:
            result[data[column].iloc[i]][target['class'].iloc[i]] += 1
        else:
            result[data[column].iloc[i]] = {'unacc':0,'acc':0,'good':0,'vgood':0}
    #we need to make the result form a probability distribution
    result = normalize_counts(result)
    return result
```

```
print(calculate_word_probability(xtrain,ytrain,'buying'))
```

```
{'med': {'unacc': 0.6065573770491803, 'acc': 0.2677595628415301, 'good': 0.06284153005464481, 'vgood': 0.06284153005464481}, 'high': {'unacc': 0.73607038
```

In normalize_count function, we we normalize each word count according to the labels. At first, we have the number of each word with respect to each label, then we use an smoothing algorithm to calculate each word's conditional probability. The smoothing algorithm is as below:
probability = (frequency + 1) / (total + v)

In calculate_word_probability function, we make dictionary of dictionaries, the first key is the words in the data and value is probability of each word regarding all 4 labels.

1.3.6 Training the Model

```
#training the model
probability_dict = {}
for i in xtrain:
    probability_dict[i] = calculate_word_probability(xtrain,ytrain,i)
```

```
print(probability_dict)
```

```
{'buying': {'med': {'unacc': 0.6065573770491803, 'acc': 0.2677595628415301, 'good': 0.06284153005464481, 'vgood': 0.06284153005464481},
```

We trained the model and calculated the priority for each word in the test set, and we saved it in a dictionary with every column header as key.

1.3.7 Testing the Model (Accuracy Calculation)

```
#testing the model
correct = 0
for i in range(len(xtest)):
    prediction = 'unacc'
    unacc_result = p_unacc * calculate_final_probability(xtest,xtest.columns,i,'unacc')
    acc_result = p_acc * calculate_final_probability(xtest,xtest.columns,i,'acc')
    good_result = p_good * calculate_final_probability(xtest,xtest.columns,i,'good')
    vgood_result = p_vgood * calculate_final_probability(xtest,xtest.columns,i,'vgood')
    myList = [unacc_result,acc_result,good_result,vgood_result]
    if np.argmax(myList) == 0:
        prediction = 'unacc'
    elif np.argmax(myList) == 1:
        prediction = 'acc'
    elif np.argmax(myList) == 2:
        prediction = 'good'
    else:
        prediction = 'vgood'
    if prediction == ytest['class'].iloc[i]:
        correct += 1
print('accuracy:',correct/len(xtest))
```

```
accuracy: 0.7109826589595376
```

To calculate the posterior probability, we used the following formula:
P(class|instance) = p(instance|class) * p(class)
p(instance|class) = p(buying=x1, maint=x2, doors=x3, persons=x4, lug_boot=x5, safety=x6)

To relax the problem, we ignored attribute dependencies, so the posterior probability will form like this:
p(instance|class) = p(buying=x1)*p(maint=x2)*p(doors=x3)*p(persons=x4)* p(lug_boot=x5) p(safety=x6)

In the testing part, we calculated the posterior probability of each instance according to the labels and our prediction is the highest value for the labels using the argmax algorithm; then we compared it to the true value and gained the accuracy of **71%**.

2 Part 2

2.1 Deep Fully-Connected Feed-forward ANN

2.1.1 Importing Necessary Libraries

```
import torch
from torch import nn
from torch.utils.data import DataLoader
import numpy as np
import pandas as pd
import torchvision.datasets as datasets
import torchvision.transforms
import matplotlib.pyplot as plt
```

Importing PyTorch to load the data, splitting them, and creating our model. Importing matplotlib for plotting.

2.1.2 Loading and Downloading the Data

```
MNIST_Data_Train = datasets.MNIST(root = '',train = True,transform = torchvision.transforms.ToTensor(),download = True)
MNIST_Data_Test = datasets.MNIST(root = '',train = False,transform = torchvision.transforms.ToTensor(),download = True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to MNIST/raw/train-images-idx3-ubyte.gz
100%|          | 9912422/9912422 [00:00<00:00, 12530368.10it/s]
Extracting MNIST/raw/train-images-idx3-ubyte.gz to MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to MNIST/raw/train-labels-idx1-ubyte.gz
100%|          | 28881/28881 [00:00<00:00, 299523.51it/s]
Extracting MNIST/raw/train-labels-idx1-ubyte.gz to MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to MNIST/raw/t10k-images-idx3-ubyte.gz
100%|          | 1648877/1648877 [00:00<00:00, 2770139.46it/s]
```

By using datasets method from torchvision library we can download MNIST train and test data and also we need to convert the data to matrices using ToTensor().

2.1.3 Getting Familiar with Data

```
print(MNIST_Data_Train)
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location:
    Split: Train
    StandardTransform
Transform: ToTensor()
```

```
print(MNIST_Data_Test)
```

```
Dataset MNIST
    Number of datapoints: 10000
    Root location:
    Split: Test
    StandardTransform
Transform: ToTensor()
```

We can see that we have 60k data for training process and 10k data for testing.

2.1.4 Getting Familiar with Data Instances

```
image0,label0 = MNIST_Data_Train[0]

print(image0)
```

```
          0.4235, 0.0039, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.3176, 0.9412, 0.9922,
          0.9922, 0.4667, 0.0980, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1765, 0.7294,
          0.9922, 0.9922, 0.5882, 0.1059, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0627,
          0.3647, 0.9882, 0.9922, 0.7333, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.9765, 0.9922, 0.9765, 0.2510, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.1804, 0.5098,
          0.7176, 0.9922, 0.9922, 0.8118, 0.0078, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.1529, 0.5804, 0.8980, 0.9922,
          0.9922, 0.9922, 0.9804, 0.7137, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0941, 0.4471, 0.8667, 0.9922, 0.9922, 0.9922,
          0.9922, 0.7882, 0.3059, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000
```

By getting the first image and label and printing the image we can see that our image is in form of 1*28*28 matrix.

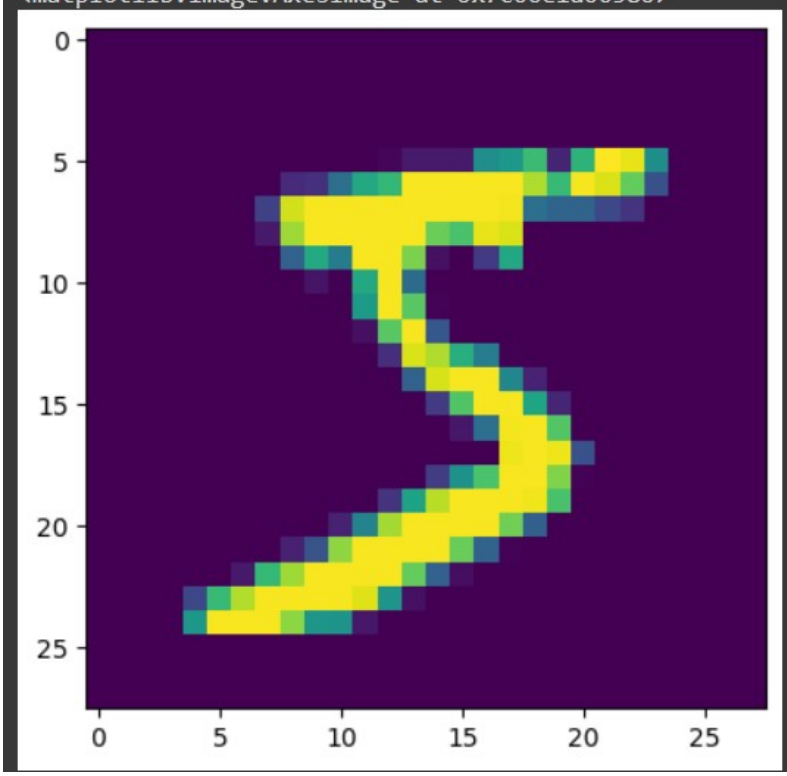
2.1.5 Validating a Label with the Corresponding Image

```
print(label0)
```

```
5
```

```
plt.imshow(image0.reshape(28,28))
```

```
<matplotlib.image.AxesImage at 0x7c06e1a00580>
```



We also printed the label and plotted the corresponding matrix to see if they fit.

2.1.6 Creating Batches

```
trainLoader = DataLoader(MNIST_Data_Train,batch_size = 64,shuffle = True)

testLoader = DataLoader(MNIST_Data_Test,batch_size = 64,shuffle = True)

for i in trainLoader:
    print(i[0])
    print(i[1])
    break
```

```
tensor([[[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          ...,
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.]]],


        [[[0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
          [0., 0., 0.,  ..., 0., 0., 0.],
```

To get a better result, we need to use minibatches instead of using incremental mode or complete batch. By using DataLoader(), we can choose batch size(here is 64) and shuffle the data.

2.1.7 First Model Creation (Sigmoid, 1 hidden layer)

```
#one hidden layered class with sigmoid as activation function
class ArtificialNeuralNetworkV01(nn.Module):
    def __init__(self,inputNeuronsNumber,hiddenNeuronsNumber,outputNeuronsNumber):
        super().__init__()
        self.hiddenLayer = nn.Linear(inputNeuronsNumber,hiddenNeuronsNumber)
        self.activation1 = nn.Sigmoid()
        self.outputLayer = nn.Linear(hiddenNeuronsNumber,outputNeuronsNumber)
        self.activation2 = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        h = self.activation1(self.hiddenLayer(x))
        o = self.activation2(self.outputLayer(h))
        return o
```

```
annv01_TestingFunctionality = ArtificialNeuralNetworkV01(28*28,200,10)
```

```
print(annv01_TestingFunctionality)
```

```
ArtificialNeuralNetworkV01(
  (hiddenLayer): Linear(in_features=784, out_features=200, bias=True)
  (activation1): Sigmoid()
  (outputLayer): Linear(in_features=200, out_features=10, bias=True)
  (activation2): LogSoftmax(dim=1)
)
```

```
print(MNIST_Data_Train[0][0].reshape(-1,28,28))
```

```
tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
```

For PyTorch, it is common to use a class inherited from nn.Module to use the parent features. In the init function, we get the number of input, hidden, and output neurons, then we create class members for hidden layer and output layer. Since our hidden layer and output layer are linear, we define a sigmoid and a log softmax as our activation functions. In the forward function, we get the input and pass it to the layers and the activation functions to receive our output. Our output has 10 members, and each member is a probability for a number between 0 and 9.

2.1.8 Checking Output

```
print(MNIST_Data_Train[0][0].shape)
```

```
torch.Size([1, 28, 28])
```

```
print(annv01_TestingFunctionality.forward(MNIST_Data_Train[0][0].reshape(-1,28*28)))
```

```
tensor([[-2.5202, -2.3548, -2.0328, -1.8324, -2.6401, -2.2774, -2.5847, -2.3908,
         -2.4214, -2.2638]], grad_fn=<LogSoftmaxBackward0>)
```

As mentioned above, the output is a list of 10 numbers.

2.1.9 Training and Testing the First Model

```
#creating our model
annv01 = ArtificialNeuralNetworkV01(28*28,100,10)

#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv01.parameters(),lr = 0.01)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv01.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()

#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv01.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv01:',correctPredictions/total)
```

```
Accuracy for annv01: tensor(0.9708)
```

We created the first model with 100 neurons in the hidden layer and the input is image size which is 784. We used cross entropy loss for multi-class classification and Adam as optimizer with learning rate of 0.01. We also trained our model in 100 epochs and during each epoch we first need to reset the gradients using optimizer.zero_grad() and then calculate the loss between prediction and true values. (please note that cross entropy loss calculates argmax inside the function and we need to give the output value as log softmax) and we backpropagate the error using loss.backward() and finally we update the weights using optimizer.step(). In the testing process, we use the test data set to calculate the accuracy. The final accuracy of the model is **97.08%**.

2.1.10 Second Model (ReLU, 1 hidden layer)

```
#one hidden layered class with relu as activation function
class ArtificialNeuralNetworkV02(nn.Module):
    def __init__(self,inputNeuronsNumber,hiddenNeuronsNumber,outputNeuronsNumber):
        super().__init__()
        self.hiddenLayer = nn.Linear(inputNeuronsNumber,hiddenNeuronsNumber)
        self.activation1 = nn.ReLU()
        self.outputLayer = nn.Linear(hiddenNeuronsNumber,outputNeuronsNumber)
        self.activation2 = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        h = self.activation1(self.hiddenLayer(x))
        o = self.activation2(self.outputLayer(h))
        return o
#creating our model
annv02 = ArtificialNeuralNetworkV02(28*28,100,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv02.parameters(),lr = 0.01)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv02.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv02.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv02:',correctPredictions/total)
```

```
Accuracy for annv02: tensor(0.9728)
```

All details are the same as in the previous section, but instead of sigmoid we used relu and got the accuracy of **97.28%**.

2.1.11 Third Model (Sigmoid, 2 hidden layers)

```
#two hidden layered class with sigmoid as activation function
class ArtificialNeuralNetworkV03(nn.Module):
    def __init__(self,inputNeuronsNumber,hiddenNeurons1Number,hiddenNeurons2Number,outputNeuronsNumber):
        super().__init__()
        self.hiddenLayer1 = nn.Linear(inputNeuronsNumber,hiddenNeurons1Number)
        self.activation1 = nn.Sigmoid()
        self.hiddenLayer2 = nn.Linear(hiddenNeurons1Number,hiddenNeurons2Number)
        self.activation2 = nn.Sigmoid()
        self.outputLayer = nn.Linear(hiddenNeurons2Number,outputNeuronsNumber)
        self.activation3 = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        h1 = self.activation1(self.hiddenLayer1(x))
        h2 = self.activation2(self.hiddenLayer2(h1))
        o = self.activation3(self.outputLayer(h2))
        return o
#creating our model
annv03 = ArtificialNeuralNetworkV03(28*28,100,150,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv03.parameters(),lr = 0.01)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv03.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv03.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv03:',correctPredictions/total)
```

```
Accuracy for annv03: tensor(0.9747)
```

Our third model uses two hidden layers and the first hidden layer includes 100 neurons and the second include 150 neurons. The final accuracy of the model is **97.47%**.

2.1.12 Fourth Model (ReLU, 2 hidden layers)

```
#two hidden layered class with relu as activation function
class ArtificialNeuralNetworkV04(nn.Module):
    def __init__(self,inputNeuronsNumber,hiddenNeurons1Number,hiddenNeurons2Number,outputNeuronsNumber):
        super().__init__()
        self.hiddenLayer1 = nn.Linear(inputNeuronsNumber,hiddenNeurons1Number)
        self.activation1 = nn.ReLU()
        self.hiddenLayer2 = nn.Linear(hiddenNeurons1Number,hiddenNeurons2Number)
        self.activation2 = nn.ReLU()
        self.outputLayer = nn.Linear(hiddenNeurons2Number,outputNeuronsNumber)
        self.activation3 = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        h1 = self.activation1(self.hiddenLayer1(x))
        h2 = self.activation2(self.hiddenLayer2(h1))
        o = self.activation3(self.outputLayer(h2))
        return o
#creating our model
annv04 = ArtificialNeuralNetworkV04(28*28,100,150,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv04.parameters(),lr = 0.01)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv04.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv04.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv04:',correctPredictions/total)
```

```
Accuracy for annv04: tensor(0.9676)
```

The fourth model is the same as the previous one, but uses relu as an activation function. The final accuracy of the model is **96.76%**.

2.1.13 Another Instance of First Model (Different hyper-parameters)

```
#testing one hidden layered model (Sigmoid as Activation) with more neurons and different learning rate
annv01_moreHiddenNeurons = ArtificialNeuralNetworkV01(28*28,300,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv01_moreHiddenNeurons.parameters(),lr = 0.001)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv01_moreHiddenNeurons.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv01_moreHiddenNeurons.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv01_moreHiddenNeurons:',correctPredictions/total)
```

```
Accuracy for annv01_moreHiddenNeurons: tensor(0.9818)
```

By creating an additional instance for the first model and changing the hidden neurons to 300, and learning rate to 0.001 we get the accuracy of **98.18%**.

2.1.14 Another Instance of Second Model (Different hyper-parameters)

```
#testing one hidden layered model (Relu as Activation) with more neurons and different learning rate
annv02_moreHiddenNeurons = ArtificialNeuralNetworkV02(28*28,300,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv02_moreHiddenNeurons.parameters(),lr = 0.001)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv02_moreHiddenNeurons.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv02_moreHiddenNeurons.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv02_moreHiddenNeurons:',correctPredictions/total)
```

```
Accuracy for annv02_moreHiddenNeurons: tensor(0.9822)
```

By creating an additional instance for the second model and changing the hidden neurons to 300, and learning rate to 0.001 we get the accuracy of **98.22**%.

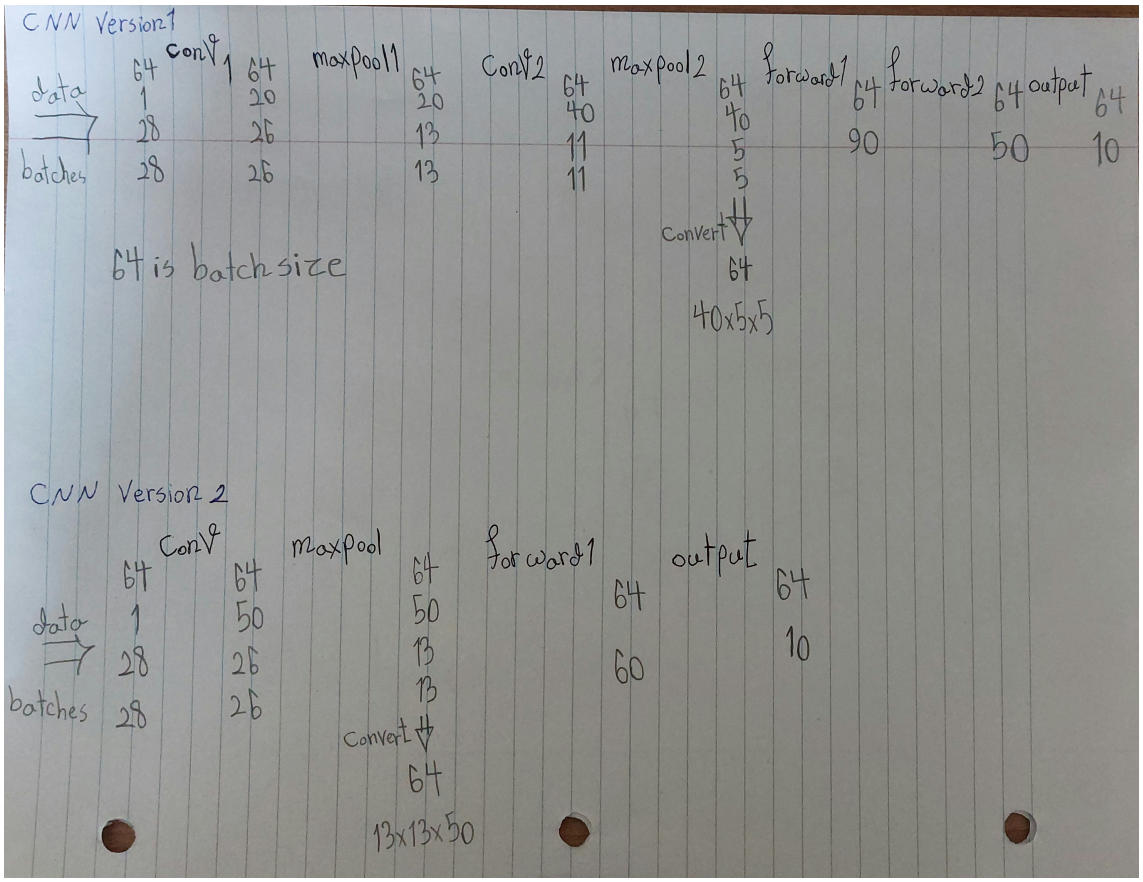
2.1.15 Conclusion

Model Version	Hidden Layer(s)	Hidden Neurons	Activation	LR	Accuracy
1	1	100	Sigmoid	0.01	97.08%
2	1	100	ReLU	0.01	97.28%
3	2	100,150	Sigmoid	0.01	97.47%
4	2	100,150	ReLU	0.01	96.76%
1	1	300	Sigmoid	0.001	98.18%
2	1	300	ReLU	0.001	98.22%

By seeing the results, we understand that ReLU as an activation function is better for Computer Vision. Moreover, having more hidden neurons can lead to higher accuracy rate.

2.2 CNN

2.2.1 Architecture



In version 1, we have two convolutional layers with 3*3 kernel size, which make the dimension of a single image from 28*28 to 26*26. By applying 2*2 max-pooling the dimension of the image is halved and became 13*13. Then we apply another convolutional layer to make the dimensions 11*11 and 5*5 in order. At first, we have only one channel, since the numbers are Black & White. After the first layer of convolution, we have 20 channels and after the second one we have 40 channels. After finishing the convolution layers we have linear layers with 90, 50 and 10 neurons. We should use activation function after each layer. The second version has only one convolutional layer and two linear layers.

2.2.2 Reading Data & Creating Batches

```
import torch
from torch import nn #importing neural network
from torch.utils.data import DataLoader #for making batches and shuffeling the data
from torchvision.datasets import MNIST #MNIST dataset
from torchvision.transforms import ToTensor #transform the data to tensor matrix

trainSet = MNIST(root = '',train = True,download = True, transform = ToTensor())
testSet = MNIST(root = '',train = False,download = True, transform = ToTensor())

print(trainSet)

Dataset MNIST
    Number of datapoints: 60000
    Root location:
    Split: Train
    StandardTransform
Transform: ToTensor()

print(testSet)

Dataset MNIST
    Number of datapoints: 10000
    Root location:
    Split: Test
    StandardTransform
Transform: ToTensor()

trainLoader = DataLoader(trainSet,batch_size = 64,shuffle = True)
testLoader = DataLoader(testSet,batch_size = 64,shuffle = True)
```

After importing the necessary libraries, we load MNIST data and divide train and test data to batches of size 64 and then we shuffle them.

2.2.3 CNN Version 1 - Implementation

```python
#First version of CNN class
class CNN_V01(nn.Module):
    def __init__(self,activation_function = 'sigmoid',learning_rate = 0.01):
        super().__init__()
        self.learning_rate = learning_rate
        self.convolutionalLayer1 = nn.Conv2d(in_channels = 1,out_channels = 20,kernel_size = 3,stride = 1) #balck & white so input channel is one
        self.convolutionalLayer2 = nn.Conv2d(in_channels = 20,out_channels = 40,kernel_size = 3,stride = 1)
        self.forward1 = nn.Linear(40*5*5,90)
        #default activation function is sigmoid
        self.activation1 = nn.Sigmoid()
        #if we set activation function value to relu then we use relu
        if activation_function == 'relu':
            self.activation1 = nn.ReLU()
        self.forward2 = nn.Linear(90,50)
        #default activation function is sigmoid
        self.activation2 = nn.Sigmoid()
        #if we set activation function value to relu then we use relu
        if activation_function == 'relu':
            self.activation2 = nn.ReLU()
        self.outputLayer = nn.Linear(50,10)
        self.log_softmax = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        #convolutional layer1 and max pooling
        x = self.convolutionalLayer1(x)
        x = nn.functional.max_pool2d(x,2,2)
        #convolution layer2 and max pooling
        x = self.convolutionalLayer2(x)
        x = nn.functional.max_pool2d(x,2,2)
        #forward layers
        x = x.reshape(-1,40*5*5)
        x = self.activation1(self.forward1(x))
        x = self.activation2(self.forward2(x))
        x = self.outputLayer(x)
        x = self.log_softmax(x)
        return x
```

In the init function, we get two attributes of learning_rate and activation_function. We also define two convolutional layers and 3 linear layers. According to the activation_function attribute value we decide to choose between relu and sigmoid. After we forwarded the data through the convolutional layers, we need to flatten the data and apply activation functions on each linear layer and the output should be in form of softmax to give us a probability for each of 10 digits.

2.2.4 CNN1

```python
#creating model instance relu as activation function and learning rate of 0.01
cnn1 = CNN_V01('relu',0.01)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn1.parameters(),lr = cnn1.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn1.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn1.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn1:',correct/total)
```

```
Accuracy for cnn1: tensor(0.9642)
```

Activation: ReLU, Learning Rate: 0.01 and version is 1

2.2.5 CNN2

```
#creating model instance relu as activation function and learning rate of 0.001
cnn2 = CNN_V01('relu',0.001)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn2.parameters(),lr = cnn2.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn2.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn2.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn2:',correct/total)
```

```
Accuracy for cnn2: tensor(0.9892)
```

Activation: ReLU, Learning Rate: 0.001 and version is 1

2.2.6 CNN3

```
#creating model instance sigmoid as activation function and learning rate of 0.01
cnn3 = CNN_V01('sigmoid',0.01)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn3.parameters(),lr = cnn3.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn3.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn3.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn3:',correct/total)
```

```
Accuracy for cnn3: tensor(0.9232)
```

Activation: Sigmoid, Learning Rate: 0.01 and version is 1

2.2.7 CNN4

```
#creating model instance sigmoid as activation function and learning rate of 0.001
cnn4 = CNN_V01('sigmoid',0.001)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn4.parameters(),lr = cnn4.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn4.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn4.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn4:',correct/total)
```

```
Accuracy for cnn4: tensor(0.9895)
```

Activation: Sigmoid, Learning Rate: 0.001 and version is 1

2.2.8 CNN Version 2 - Implementation

```
#second version of cnn (simple architecture)
class CNN_V02(nn.Module):
    def __init__(self,activation_function = 'sigmoid',learning_rate = 0.01):
        super().__init__()
        self.learning_rate = learning_rate
        self.convolutionalLayer = nn.Conv2d(in_channels = 1,out_channels = 50,kernel_size = 3,stride = 1)
        self.forward1 = nn.Linear(50*13*13,60)
        #default activation function is sigmoid
        self.activation1 = nn.Sigmoid()
        #if we set activation function value to relu then we use relu
        if activation_function == 'relu':
            self.activation1 = nn.ReLU()
        self.outputLayer = nn.Linear(60,10)
        self.log_softmax = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        x = nn.functional.max_pool2d(self.convolutionalLayer(x),2,2)
        x = x.reshape(-1,50*13*13)
        x = self.activation1(self.forward1(x))
        x = self.log_softmax(x)
        return x
```

Version 2 uses simpler structure only having a convolutional layer and two linear layers.

2.2.9 CNN5

```
#creating model instance relu as activation function and learning rate of 0.01
cnn5 = CNN_V02('relu',0.01)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn5.parameters(),lr = cnn5.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn5.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn5.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn5:',correct/total)
```

```
Accuracy for cnn5: tensor(0.8516)
```

Activation: ReLU, Learning Rate: 0.01 and version is 2

2.2.10 CNN6

```
#creating model instance relu as activation function and learning rate of 0.01
cnn6 = CNN_V02('relu',0.001)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn6.parameters(),lr = cnn6.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn6.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn6.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn6:',correct/total)
```

```
Accuracy for cnn6: tensor(0.8819)
```

Activation: ReLU, Learning Rate: 0.001 and version is 2

2.2.11 CNN7

```
#creating model instance sigmoid as activation function and learning rate of 0.01
cnn7 = CNN_V02('sigmoid',0.01)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn7.parameters(),lr = cnn7.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn7.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn7.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn7:',correct/total)
```

```
Accuracy for cnn7: tensor(0.0980)
```

Activation: Sigmoid, Learning Rate: 0.01 and version is 2

2.2.12 CNN8

```
#creating model instance sigmoid as activation function and learning rate of 0.001
cnn8 = CNN_V02('sigmoid',0.001)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn8.parameters(),lr = cnn8.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn8.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn8.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn8:',correct/total)
```

```
Accuracy for cnn8: tensor(0.9721)
```

Activation: Sigmoid, Learning Rate: 0.001 and version is 2

2.2.13 Conclusion

Model Number	Model Version	Activation Function	Learning Rate	Accuracy
1	1	ReLU	0.01	96.42%
2	1	ReLU	0.001	98.92%
3	1	Sigmoid	0.01	92.32%
4	1	Sigmoid	0.001	98.95%
5	2	ReLU	0.01	85.16%
6	2	ReLU	0.001	88.19%
7	2	Sigmoid	0.01	9.8%
8	2	Sigmoid	0.001	97.21%

We can see that the first structure had higher average accuracy than the second one, due to using more complex structure and more layers. ReLU generally outperforms sigmoid and with the learning rate of 0.001 we achieved better results.

3 Appendix

3.1 Programming Specifications

We did the implementations using Python programming language, we worked on a shared Jupyter Notebook on Google Colab. We also used the PyTorch library to implement the second part. (Link to the project)

3.2 Source Codes

3.2.1 AdaBoost (base learner: ID3)

```
!pip install ucimlrepo
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import math
from collections import Counter
from ucimlrepo import fetch_ucirepo
letter_recognition = fetch_ucirepo(id=59)
dataFrame = pd.DataFrame(data=letter_recognition.data.features, columns=letter_recognition.data.features.columns)
dataFrame['target'] = letter_recognition.data.targets
dataFrame[10:20]
print(dataFrame.isnull().sum())
print(dataFrame.describe())
dataFrame['target'].value_counts()
x = dataFrame.drop('target', axis=1)
y = dataFrame['target']
xtrain, xtest, ytrain, ytest = train_test_split(x, y, train_size=0.8)
print(xtrain.shape)
print(xtest.shape)
print(ytrain.shape)
print(ytest.shape)
Counter(dataFrame['width'].values)
#function below calculates entropy for a single attribute on a single node
def entropy(data,attribute):
    res = 0
    frequencyDictionary = Counter(data[attribute].values)
    for key in frequencyDictionary:
        probability = frequencyDictionary[key]/len(data)
        res -= probability * math.log2(probability)
    return res

#function below calculates the information game using the parent and a single child for a single node
def information_gain(data,feature,target_attribute):
    parent_entropy = entropy(data, target_attribute)
    frequencyDictionary = Counter(data[feature].values)
    weighted_entropy = 0
    for key, value in frequencyDictionary.items():
        subset = data[data[feature] == key]
        weighted_entropy += (value/len(data))*entropy(subset, target_attribute)
    return parent_entropy - weighted_entropy

#ID3 algorithm to build the tree
def id3(data, features, target_attribute, height=3):
    target_values = data[target_attribute].values
    #If all target values are the same, return the label
    if len(set(target_values)) == 1:
        return target_values[0]
    #At the laef we choose the most repeated value
    if len(features) == 0 or height == 0:
        return Counter(target_values).most_common(1)[0][0]
    #Choosing a feature with the highest information gain
    best_feature = max(features, key=lambda feature: information_gain(data, feature, target_attribute))
    #Our tree is at first has the best feature as the parent and there are no children
    tree = {best_feature: {}}
    #Split the dataset on the best feature and build the subtree
    unique_values = set(data[best_feature])
    remaining_features = [feature for feature in features if feature != best_feature]
    for value in unique_values:
        subset = data[data[best_feature] == value]
        tree[best_feature][value] = id3(subset, remaining_features, target_attribute, height-1)
    return tree

#ID3 classifier to choose instance label
def classify(tree, instance):
    if type(tree) is not dict:
        return tree
    feature = next(iter(tree))
    value = instance[feature]
    if value not in tree[feature]:
        return None
    return classify(tree[feature][value], instance)

#Adaboost algorithm
def adaboost(data, features, target_attribute, iterations):
```

```
    #at first, every sample has the same weights and the value is 1/n
    weights = [1 / len(data)] * len(data)
    classifiers = []
    alphas = []
    for i in range(iterations):
        # Build a weighted dataset by sampling rows based on their weights
        weighted_data = data.sample(n=len(data), weights=weights, replace=True)
        #creating tree instance
        tree = id3(weighted_data, features, target_attribute, height=5)
        print(tree)
        #error calculation
        error = 0
        for j, row in data.iterrows():
            prediction = classify(tree, row)
            if prediction != row[target_attribute]:
                error += weights[j]
        # Calculate the alpha (influence of this classifier)
        if error == 0:
            error = 1e-10  # Avoid division by zero
        alpha = 0.5 * math.log((1 - error) / error)
        #updating the weights
        for j, row in data.iterrows():
            prediction = classify(tree, row)
            if prediction == row[target_attribute]:
                weights[j] *= math.exp(-alpha)
            else:
                weights[j] *= math.exp(alpha)
        #normalizing the weights
        weights_sum = sum(weights)
        for j in range(len(weights)):
          weights[j] = weights[j]/weights_sum
        classifiers.append(tree)
        alphas.append(alpha)
    return classifiers, alphas

#Adaboost Classification
def adaboost_classify(classifiers, alphas, instances):
    predictions = []
    for index in instances.index:
        instance = instances.loc[index].to_dict()
        result = {}
        for alpha, classifier in zip(alphas, classifiers):
          instance_prediction = classify(classifier,instance)
          if instance_prediction not in result:
            result[instance_prediction] = 0
          result[instance_prediction] += alpha
        prediction = max(result,key=result.get)
        predictions.append(prediction)
    return predictions
#training the model
classifiers, alphas = adaboost(dataFrame, x.columns.tolist(), "target", iterations=3)
#testing the model
yprediction = adaboost_classify(classifiers, alphas, xtest)
print(ytest == yprediction)
print(sum(ytest == yprediction))
print('Accuracy:',sum(ytest == yprediction)/len(ytest))
```

3.2.2 ANN

```
!pip install ucimlrepo
import numpy as np
import pandas as pd
import math
from ucimlrepo import fetch_ucirepo
from sklearn import preprocessing
BCWD_Dataset = fetch_ucirepo(id = 17)
x = BCWD_Dataset.data.features
y = BCWD_Dataset.data.targets
print(x)
print(y)
#normalizing the data
#ss = preprocessing.StandardScaler()
#x_normalized = ss.fit_transform(x)
#print(x_normalized)
x_normalized = preprocessing.normalize(x)
print(x_normalized)
#we need to convert the output to numerical values for classification calculations
y_numerical = pd.DataFrame(data = {'Diagnosis':range(len(y))})
for i in range(len(y)):
    if y['Diagnosis'][i] == 'B':
        y_numerical['Diagnosis'][i] = 0
    else:
        y_numerical['Diagnosis'][i] = 1
print(y_numerical)
BCWD_Metadata = BCWD_Dataset.metadata
BCWD_Variables = BCWD_Dataset.variables
#sigmoid over a single value of data
def sigmoid(x):
    return 1/(1+(math.e)**-x)
#sigmoid derivation over a single value of data
def sigmoid_derivation(x):
    return sigmoid(x)*(1-sigmoid(x))
#adding the sigmoid function to numpy library
```

```
sigmoid = np.frompyfunc(sigmoid,1,1)
sigmoid_derivation = np.frompyfunc(sigmoid_derivation,1,1)
#Relu over a single value of data
def relu(x):
    if x <= 0:
        return 0
    return x
#relu derivation over a single value of data
def relu_derivation(x):
    if x <= 0:
        return 0
    return 1
#adding the relu function to numpy library
relu = np.frompyfunc(relu,1,1)
relu_derivation = np.frompyfunc(relu_derivation,1,1)
x_train,x_test = x_normalized[:500],x_normalized[500:]
y_train,y_test = y_numerical[:500],y_numerical[500:]
class BCWDClassifier():
    def __init__(self,inputNumber,hiddenLayerNeuronsNumber,outputNeuronsNumber,learningRate):
        self.learningRate = learningRate
        self.Winput2hidden = np.random.normal(size = (inputNumber,hiddenLayerNeuronsNumber))
        self.Whidden2output = np.random.normal(size = (hiddenLayerNeuronsNumber,outputNeuronsNumber))
    def forward(self,x):
        hiddenNeurons = sigmoid(x @ self.Winput2hidden)
        outputNeurons = sigmoid(hiddenNeurons @ self.Whidden2output)
        return outputNeurons.reshape(1,len(outputNeurons)),hiddenNeurons.reshape(1,len(hiddenNeurons)),self.Winput2hidde
    def updateWeights(self,derivedWinput2hidden,derivedWhidden2output):
        self.Winput2hidden += self.learningRate * derivedWinput2hidden
        self.Whidden2output += self.learningRate * derivedWhidden2output
def calculateGradient(x,w1,w2,h,err):
    derivedWinput2hidden = x.reshape(len(x),1) @ (w2.T * sigmoid_derivation(x @ w1))* sigmoid_derivation(h @ w2) * err
    derivedWhidden2output = err * sigmoid_derivation(h @ w2) * h
    return derivedWinput2hidden.astype('float64'),derivedWhidden2output.T.astype('float64')
bcwdc = BCWDClassifier(len(x_train[0]),100,1,0.01)
for i in range(500):
    for j in range(len(x_train)):
        o,h,w1,w2 = bcwdc.forward(x_train[i])
        dw1,dw2 = calculateGradient(x_train[j],w1,w2,h,y_train.iloc[j][0]-o)
        bcwdc.updateWeights(dw1,dw2)
total = len(y_test)
correct = 0
for i in range(total):
    o,_,_,_ = bcwdc.forward(x_test[i])
    if o < 0.5:
        o = 0
    else:
        o = 1
    if o == y_test.iloc[i][0]:
        correct += 1
print('accuracy:',correct/total)
```

3.2.3 Naïve Bayes

```
!pip install ucimlrepo
from ucimlrepo import fetch_ucirepo
from sklearn.model_selection import train_test_split
import numpy as np
carEvaluationDataset = fetch_ucirepo(id = 19)
dataFeatures = carEvaluationDataset.data.features
dataTargets = carEvaluationDataset.data.targets
print(dataFeatures)
print(dataTargets)
print(dataTargets.value_counts())
#dividing the train and the test sections
xtrain,xtest,ytrain,ytest = train_test_split(dataFeatures,dataTargets,train_size = 0.8,shuffle = True)
print(xtrain)
print(ytrain)
print(xtest)
print(ytest)
#calculating probability of each class
p_unacc = sum(ytrain['class'] == 'unacc')/len(ytrain)
p_acc = sum(ytrain['class'] == 'acc')/len(ytrain)
p_good = sum(ytrain['class'] == 'good')/len(ytrain)
p_vgood = sum(ytrain['class'] == 'vgood')/len(ytrain)
print(p_unacc,p_acc,p_good,p_vgood)
#function below normalizes word counts to the form of probability distribution
#function below normalizes word counts to the form of probability distribution
def normalize_counts(myDictionary):
    for key in myDictionary:
        keyNum = 0
        totalSum = 0
        for i in myDictionary[key]:
            keyNum += 1
            totalSum += myDictionary[key][i]
        for i in myDictionary[key]:
            myDictionary[key][i] = ((myDictionary[key][i]+1)/(totalSum+keyNum))
    return myDictionary
#function below claculates probability of each word according to each class
def calculate_word_probability(data,target,column):
    result = dict()
    for i in range(len(data)):
        if data[column].iloc[i] in result:
            result[data[column].iloc[i]][target['class'].iloc[i]] += 1
```

```
            else:
                result[data[column].iloc[i]] = {'unacc':0,'acc':0,'good':0,'vgood':0}
        #we need to make the result form a probability distribution
        result = normalize_counts(result)
        return result
print(calculate_word_probability(xtrain,ytrain,'buying'))
#training the model
probability_dict = {}
for i in xtrain:
    probability_dict[i] = calculate_word_probability(xtrain,ytrain,i)
print(probability_dict)
#calculating post priority probability
def calculate_final_probability(data,columns,index,classification):
    result = 1
    for i in columns:
        result *= probability_dict[i][data[i].iloc[index]][classification]
    return result
#testing the model
correct = 0
for i in range(len(xtest)):
    prediction = 'unacc'
    unacc_result = p_unacc * calculate_final_probability(xtest,xtest.columns,i,'unacc')
    acc_result = p_acc * calculate_final_probability(xtest,xtest.columns,i,'acc')
    good_result = p_good * calculate_final_probability(xtest,xtest.columns,i,'good')
    vgood_result = p_vgood * calculate_final_probability(xtest,xtest.columns,i,'vgood')
    myList = [unacc_result,acc_result,good_result,vgood_result]
    if np.argmax(myList) == 0:
        prediction = 'unacc'
    elif np.argmax(myList) == 1:
        prediction = 'acc'
    elif np.argmax(myList) == 2:
        prediction = 'good'
    else:
        prediction = 'vgood'
    if prediction == ytest['class'].iloc[i]:
        correct += 1
print('accuracy:',correct/len(xtest))
```

3.2.4 Deep Fully-Connected Feed-forward ANN

```
import torch
from torch import nn
from torch.utils.data import DataLoader
import numpy as np
import pandas as pd
import torchvision.datasets as datasets
import torchvision.transforms
import matplotlib.pyplot as plt
MNIST_Data_Train = datasets.MNIST(root = '',train = True,transform = torchvision.transforms.ToTensor(),download = True)
MNIST_Data_Test = datasets.MNIST(root = '',train = False,transform = torchvision.transforms.ToTensor(),download = True)
print(MNIST_Data_Train)
print(MNIST_Data_Test)
image0,label0 = MNIST_Data_Train[0]
print(image0)
print(label0)
plt.imshow(image0.reshape(28,28))
trainLoader = DataLoader(MNIST_Data_Train,batch_size = 64,shuffle = True)
testLoader = DataLoader(MNIST_Data_Test,batch_size = 64,shuffle = True)
for i in trainLoader:
    print(i[0])
    print(i[1])
    break
#one hidden layered class with sigmoid as activation function
class ArtificialNeuralNetworkV01(nn.Module):
    def __init__(self,inputNeuronsNumber,hiddenNeuronsNumber,outputNeuronsNumber):
        super().__init__()
        self.hiddenLayer = nn.Linear(inputNeuronsNumber,hiddenNeuronsNumber)
        self.activation1 = nn.Sigmoid()
        self.outputLayer = nn.Linear(hiddenNeuronsNumber,outputNeuronsNumber)
        self.activation2 = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        h = self.activation1(self.hiddenLayer(x))
        o = self.activation2(self.outputLayer(h))
        return o
annv01_TestingFunctionality = ArtificialNeuralNetworkV01(28*28,200,10)
print(annv01_TestingFunctionality)
print(MNIST_Data_Train[0][0].reshape(-1,28,28))
print(MNIST_Data_Train[0][0].shape)
print(annv01_TestingFunctionality.forward(MNIST_Data_Train[0][0].reshape(-1,28*28)))
#creating our model
annv01 = ArtificialNeuralNetworkV01(28*28,100,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv01.parameters(),lr = 0.01)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv01.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
```

```
total = 0
for x,y in testLoader:
    predictions = annv01.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv01:',correctPredictions/total)
#one hidden layered class with relu as activation function
class ArtificialNeuralNetworkV02(nn.Module):
    def __init__(self,inputNeuronsNumber,hiddenNeuronsNumber,outputNeuronsNumber):
        super().__init__()
        self.hiddenLayer = nn.Linear(inputNeuronsNumber,hiddenNeuronsNumber)
        self.activation1 = nn.ReLU()
        self.outputLayer = nn.Linear(hiddenNeuronsNumber,outputNeuronsNumber)
        self.activation2 = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        h = self.activation1(self.hiddenLayer(x))
        o = self.activation2(self.outputLayer(h))
        return o
#creating our model
annv02 = ArtificialNeuralNetworkV02(28*28,100,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv02.parameters(),lr = 0.01)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv02.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv02.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv02:',correctPredictions/total)
#two hidden layered class with sigmoid as activation function
class ArtificialNeuralNetworkV03(nn.Module):
    def __init__(self,inputNeuronsNumber,hiddenNeurons1Number,hiddenNeurons2Number,outputNeuronsNumber):
        super().__init__()
        self.hiddenLayer1 = nn.Linear(inputNeuronsNumber,hiddenNeurons1Number)
        self.activation1 = nn.Sigmoid()
        self.hiddenLayer2 = nn.Linear(hiddenNeurons1Number,hiddenNeurons2Number)
        self.activation2 = nn.Sigmoid()
        self.outputLayer = nn.Linear(hiddenNeurons2Number,outputNeuronsNumber)
        self.activation3 = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        h1 = self.activation1(self.hiddenLayer1(x))
        h2 = self.activation2(self.hiddenLayer2(h1))
        o = self.activation3(self.outputLayer(h2))
        return o
#creating our model
annv03 = ArtificialNeuralNetworkV03(28*28,100,150,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv03.parameters(),lr = 0.01)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv03.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv03.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv03:',correctPredictions/total)
#two hidden layered class with relu as activation function
class ArtificialNeuralNetworkV04(nn.Module):
    def __init__(self,inputNeuronsNumber,hiddenNeurons1Number,hiddenNeurons2Number,outputNeuronsNumber):
        super().__init__()
        self.hiddenLayer1 = nn.Linear(inputNeuronsNumber,hiddenNeurons1Number)
        self.activation1 = nn.ReLU()
        self.hiddenLayer2 = nn.Linear(hiddenNeurons1Number,hiddenNeurons2Number)
        self.activation2 = nn.ReLU()
        self.outputLayer = nn.Linear(hiddenNeurons2Number,outputNeuronsNumber)
        self.activation3 = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        h1 = self.activation1(self.hiddenLayer1(x))
        h2 = self.activation2(self.hiddenLayer2(h1))
        o = self.activation3(self.outputLayer(h2))
        return o
#creating our model
annv04 = ArtificialNeuralNetworkV04(28*28,100,150,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv04.parameters(),lr = 0.01)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv04.forward(x.reshape(-1,28*28))
```

```
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv04.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv04:',correctPredictions/total)
#testing one hidden layered model (Sigmoid as Activation) with more neurons and different learning rate
annv01_moreHiddenNeurons = ArtificialNeuralNetworkV01(28*28,300,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv01_moreHiddenNeurons.parameters(),lr = 0.001)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv01_moreHiddenNeurons.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv01_moreHiddenNeurons.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv01_moreHiddenNeurons:',correctPredictions/total)
#testing one hidden layered model (Relu as Activation) with more neurons and different learning rate
annv02_moreHiddenNeurons = ArtificialNeuralNetworkV02(28*28,300,10)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(annv02_moreHiddenNeurons.parameters(),lr = 0.001)
for i in range(100):
    for x,y in trainLoader:
        optimizer.zero_grad()
        yPrediction = annv02_moreHiddenNeurons.forward(x.reshape(-1,28*28))
        loss = criterion(yPrediction,y)
        loss.backward()
        optimizer.step()
#testing our model
correctPredictions = 0
total = 0
for x,y in testLoader:
    predictions = annv02_moreHiddenNeurons.forward(x.reshape(-1,28*28))
    correctPredictions += sum(torch.argmax(predictions,dim = 1) == y)
    total += len(y)
print('Accuracy for annv02_moreHiddenNeurons:',correctPredictions/total)
```

3.2.5 CNN

```
import torch
from torch import nn #importing neural network
from torch.utils.data import DataLoader #for making batches and shuffeling the data
from torchvision.datasets import MNIST #MNIST dataset
from torchvision.transforms import ToTensor #transform the data to tensor matrix
trainSet = MNIST(root = '',train = True,download = True, transform = ToTensor())
testSet = MNIST(root = '',train = False,download = True, transform = ToTensor())
print(trainSet)
print(testSet)
trainLoader = DataLoader(trainSet,batch_size = 64,shuffle = True)
testLoader = DataLoader(testSet,batch_size = 64,shuffle = True)
#First version of CNN class
class CNN_V01(nn.Module):
    def __init__(self,activation_function = 'sigmoid',learning_rate = 0.01):
        super().__init__()
        self.learning_rate = learning_rate
        self.convolutionalLayer1 = nn.Conv2d(in_channels = 1,out_channels = 20,kernel_size = 3,stride = 1)
        self.convolutionalLayer2 = nn.Conv2d(in_channels = 20,out_channels = 40,kernel_size = 3,stride = 1)
        self.forward1 = nn.Linear(40*5*5,90)
        #default activation function is sigmoid
        self.activation1 = nn.Sigmoid()
        #if we set activation function value to relu then we use relu
        if activation_function == 'relu':
            self.activation1 = nn.ReLU()
        self.forward2 = nn.Linear(90,50)
        #default activation function is sigmoid
        self.activation2 = nn.Sigmoid()
        #if we set activation function value to relu then we use relu
        if activation_function == 'relu':
            self.activation2 = nn.ReLU()
        self.outputLayer = nn.Linear(50,10)
        self.log_softmax = nn.LogSoftmax(dim = 1)
    def forward(self,x):
        #convolutional layer1 and max pooling
        x = self.convolutionalLayer1(x)
        x = nn.functional.max_pool2d(x,2,2)
        #convolution layer2 and max pooling
        x = self.convolutionalLayer2(x)
        x = nn.functional.max_pool2d(x,2,2)
```

```
        #forward layers
        x = x.reshape(-1,40*5*5)
        x = self.activation1(self.forward1(x))
        x = self.activation2(self.forward2(x))
        x = self.outputLayer(x)
        x = self.log_softmax(x)
        return x
#creating model instance relu as activation function and learning rate of 0.01
cnn1 = CNN_V01('relu',0.01)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn1.parameters(),lr = cnn1.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn1.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn1.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn1:',correct/total)
#creating model instance relu as activation function and learning rate of 0.001
cnn2 = CNN_V01('relu',0.001)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn2.parameters(),lr = cnn2.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn2.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn2.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn2:',correct/total)
#creating model instance sigmoid as activation function and learning rate of 0.01
cnn3 = CNN_V01('sigmoid',0.01)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn3.parameters(),lr = cnn3.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn3.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn3.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn3:',correct/total)
#creating model instance sigmoid as activation function and learning rate of 0.001
cnn4 = CNN_V01('sigmoid',0.001)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn4.parameters(),lr = cnn4.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn4.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn4.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn4:',correct/total)
#second version of cnn (simple architecture)
class CNN_V02(nn.Module):
    def __init__(self,activation_function = 'sigmoid',learning_rate = 0.01):
        super().__init__()
        self.learning_rate = learning_rate
        self.convolutionalLayer = nn.Conv2d(in_channels = 1,out_channels = 50,kernel_size = 3,stride = 1)
        self.forward1 = nn.Linear(50*13*13,60)
        #default activation function is sigmoid
        self.activation1 = nn.Sigmoid()
        #if we set activation function value to relu then we use relu
```

```
            if activation_function == 'relu':
                self.activation1 = nn.ReLU()
            self.outputLayer = nn.Linear(60,10)
            self.log_softmax = nn.LogSoftmax(dim = 1)
        def forward(self,x):
            x = nn.functional.max_pool2d(self.convolutionalLayer(x),2,2)
            x = x.reshape(-1,50*13*13)
            x = self.activation1(self.forward1(x))
            x = self.log_softmax(x)
            return x
#creating model instance relu as activation function and learning rate of 0.01
cnn5 = CNN_V02('relu',0.01)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn5.parameters(),lr = cnn5.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn5.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn5.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn5:',correct/total)
#creating model instance relu as activation function and learning rate of 0.01
cnn6 = CNN_V02('relu',0.001)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn6.parameters(),lr = cnn6.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn6.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn6.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn6:',correct/total)
#creating model instance sigmoid as activation function and learning rate of 0.01
cnn7 = CNN_V02('sigmoid',0.01)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn7.parameters(),lr = cnn7.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn7.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn7.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn7:',correct/total)
#creating model instance sigmoid as activation function and learning rate of 0.001
cnn8 = CNN_V02('sigmoid',0.001)
#training the model
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn8.parameters(),lr = cnn8.learning_rate)
for epoch in range(20):
    for x,y in trainLoader:
        optimizer.zero_grad()
        y_prediction = cnn8.forward(x)
        loss = criterion(y_prediction,y)
        loss.backward()
        optimizer.step()
#testing the model accuracy
total = 0
correct = 0
for x,y in testLoader:
    total += len(y)
    y_prediction = cnn8.forward(x)
    correct += sum(torch.argmax(y_prediction,dim = 1) == y)
print('Accuracy for cnn8:',correct/total)
```