```
#Mark Makris (19230705)  & Kieran Brady (12343851)
#CT5133 Deep Learning Assignment 1 2020

#imports
from sklearn import datasets
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import metrics # Metrics package used for accuracy calculations
from sklearn.metrics import f1_score # Metrics for calculating the F - score
```

PART 1: NEURAL NETWORK START---

Algorithm – neural network: The neural network was set up as a python class. To instantiate the n are being read and how many inputs were being used to set up the length of the matrices for weigl weights going from the inputs to the hidden layer and one for the weights from the hidden layer to into a few classes for training, feed forward, backpropagation, and the sigmoid functions. There ar using sigmoid. One applies sigmoid and the other applies the derivative of sigmoid. There is a fun the X and y from a training set. Then it also needs to know how many epochs to limit the training t the weights. It loops through all the epochs and starts with feeding the training data into the netwo network and applies backpropagation to get the costs of the weights. Finally it takes the weights a cost functions and running those sets of functions again for all the epochs. The feed forward func training data. It begins by creating the lines to each node using the weights to the hidden layer anc to those lines to create the hidden nodes. From there it creates the lines from the hidden layer to tl them. All the lines and sigmoid are returned to be used in the backpropagation. The backpropagat the first layer of error costs in the output layer and the second layer of error costs in the input layer sigmoid derivative function. Then depending on the weights of each node a certain amount of that hidden layer costs are calculated the cost of the inputs need to be calculated. Then for each hidde weights. After the costs are calculated for each layer they need to be reset as well to the output w weights in and out are returned to calculate the new weights. The malik paper was very good at ex work on multiple weights at once.

```
#Neural net with one hidden layers
#Malik, U. (2020) matrix multiplication
class NN(object):
    def __init__(self, points, inputs, hidden): #needs number of data points and hidden la
        self.points = points
        self.wIn = np.random.rand(inputs, hidden)  #weights of input layer (#input, #hidde
        self.wOut = np.random.rand(hidden, 1) #weights of output layer (#hidden, #output)

    def train(self, x, y, epochs=100, a = 0.01):# update weights and biases based on the o
        for epoch in range(epochs):
            inLine, inSig, outLine, outSig = self.feedforward(x)
            dcost_wh, dcost_wo = self.backpropagation(x, y, inLine, inSig, outLine, outSig
            self.wIn -= a * dcost_wh #edit input weights based on learning rate * error
            self.wOut -= a * dcost_wo #edit output weights based on learning rate * error
```

```
    def feedforward(self, x):
        inLine = np.dot(x, self.wIn) #calculate the lines of input weights
        inSig = self.sigmoid(inLine) #apply sigmoid to the input lines

        outLine = np.dot(inSig, self.wOut) #calculate the lines of output weights
        outSig = self.sigmoid(outLine) #apply sigmoid to the output lines

        return inLine, inSig, outLine, outSig #returns everything

    def backpropagation(self, x, y, inLine, inSig, outLine, outSig):
        #output layer
        error_out = ((1 / self.points) * (np.power((outSig - y), 2)))



        dcost_dos = outSig - y #calculates error of output
        dos_dol = self.sigmoid_der(outLine) #calculates derivatives of out line
        dzo_dwo = inSig #set derivatives of out weights

        dcost_wo = np.dot(dzo_dwo.T, dcost_dos * dos_dol) #derivatives of out weights

        #input layer
        dcost_dol = dcost_dos * dos_dol
        dol_dis = self.wOut
        dcost_dis = np.dot(dcost_dol , dol_dis.T)
        dis_dil = self.sigmoid_der(inLine)
        dil_dwi = x

        dcost_wi = np.dot(dil_dwi.T, dis_dil * dcost_dis)

        return dcost_wi, dcost_wo #return costs

    #sigmoid = 1/(1=e^x)
    def sigmoid(self, x):
        return 1/(1+np.exp(-x))

    #deriv = sig * (1-sig)
    def sigmoid_der(self, x):
        return self.sigmoid(x) *(1-self.sigmoid(x))
```

NEURAL NETWORK END---

PART 2: START SMALL DATA SET

Algorithm – small data set: The small data set was very easy to read and use. We used pandas to pandas data frame. The class column is cut off to define the y where the rest of the data is left as a numpy list which is what the net uses. After, the data is split into a training and testing set. 70% i is used for testing. We just split on the first 350 and last 150 nodes because it results in the same data set is easily visualized on a 2D graph with the color being defined by y. This is then also used were correctly classified. The model is easily built and trained using the training set and then the o visualized to examine results.

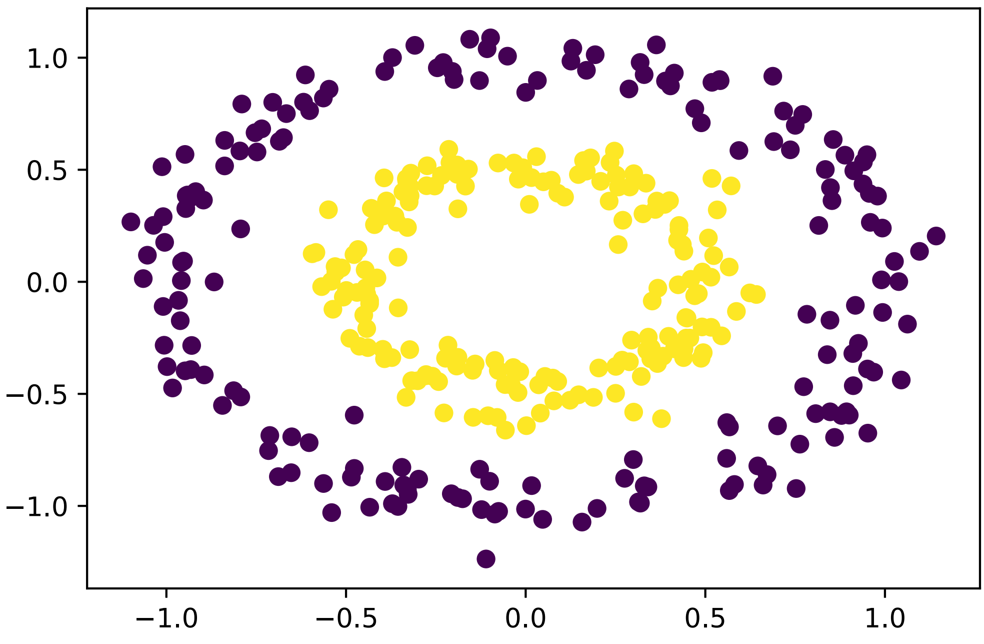
```
# Use pandas to read the CSV file as a dataframe
url = "circles500.csv"
data = pd.read_csv(url, header=0)
```

```
y = data.pop('Class').values #creates the outputs
y = y.reshape(len(y),1) #reshapes class
X = data.to_numpy() #creates the inputs and convert to numpy array

#split data into training and testing sets
trainX = X[:350,]
trainY = y[:350,]
print(trainY.shape)

print(trainX.shape)
testX = X[150:,]
testY = y[150:,]
```

```
(350, 1)
(350, 2)
```

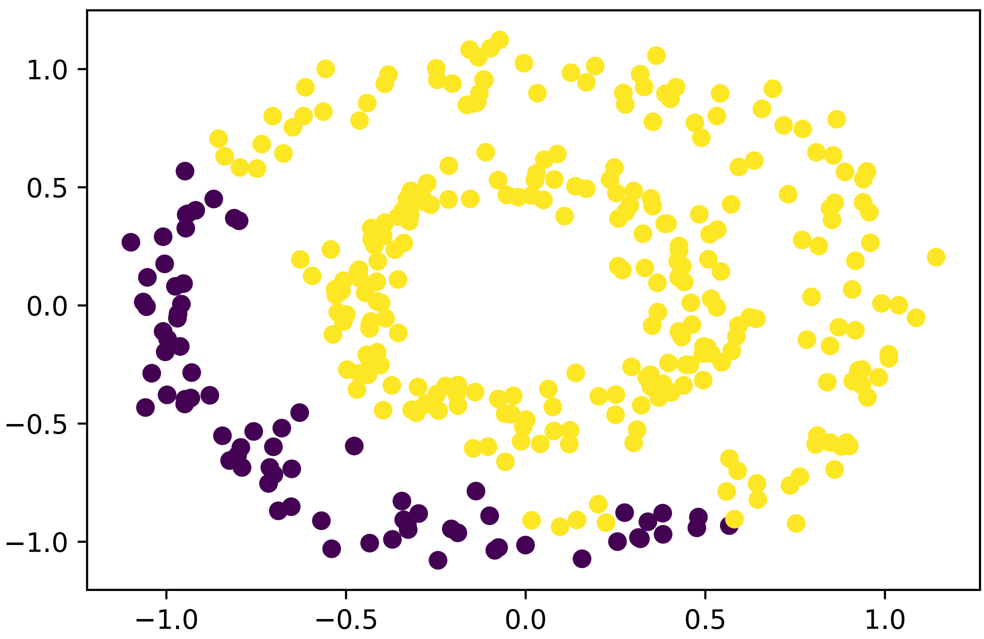
```
#plots

trainY = trainY.reshape(trainY.shape[0])
plt.scatter(trainX[:,0], trainX[:,1], c=trainY)
```

```
<matplotlib.collections.PathCollection at 0x1909192b4e0>
```



```
#build small model
#instance of network model
sn = NN(350, 2, 30)
#builds the network model
sn.train(x=trainX, y=trainY, epochs=10000, a = .045)
#test model
a, b, c, outY = sn.feedforward(testX) #a, b, c are needed in NN but not when testing


outY = outY.reshape(outY.shape[0])
for i in range(len(outY)):
    if outY[i] < .5:
        outY[i] = 0
    else:
        outY[i] = 1

plt.scatter(testX[:,0], testX[:,1], c=outY)
# f score
score = f1_score(testY , outY, average='macro')


print("F score:", score)
print("Accuracy:", metrics.accuracy_score(testY, outY))
```

```
F score: 0.662387316286085
Accuracy: 0.6857142857142857
```



Small dataset observation: When building this neural network I just used the x and y values for the network that created a circle around the first class and have the second class outside of the circle network was creating a crescent shape but not a full circle. After trying dozens of combinations of could get was all the first class correct and about half the second class correct. If we tried to get b classifying all the nodes to the second class.

END SMALL DATA SET---

PART 3: START LARGE DATA SET---

The large data set was read in using the sample code provided. Training features were found in the label key. Both the features and the labels were filtered to observations of ships and automobiles a to 0 and Ship labels were converted to 1 for the neural network output. The training features were observed. When loaded and modified the training feature matrix shape is printed so it can be input same process is completed for the test data which comes from the test batch in the Cifar dataset.

```
#Read in large data set



# This function taken from the CIFAR website for unloading cifar data. ref = Load and View

def unpickle(file):
    import pickle
    with open(file, 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    return dict




#For loading data (from Blackboard) ref = Load and View CIFAR-10 Data, Michael Madden, Jan
def loadbatch(batchname):
    folder = 'cifar-10-batches-py'
    batch = unpickle(folder+"/"+batchname)
    return batch


#For loading data (from Blackboard), ref = Load and View CIFAR-10 Data, Michael Madden, Ja
def loadlabelnames():
    folder = 'cifar-10-batches-py'
    meta = unpickle(folder+"/"+'batches.meta')
    return meta[b'label_names']

#training data
batch1 = loadbatch('data_batch_5') #load batch
batchData1 = batch1[b'data'] # Create features data set
batchLabels1 = batch1[b'labels'] # Create labels dataset
names = loadlabelnames() #load label names funtion from sample code on blackboard
```

```
trainData = [] #create empty list for filtered training features data
trainLabels = [] ##create empty list for filtered training labels

#filter data to only ship and automibile observations
for i in batchLabels1:
    labl = names[batchLabels1[i]]

    if labl == b'ship':
        trainData.append(batchData1[i])
        trainLabels.append(1)
    if labl == b'automobile':
        trainData.append(batchData1[i])
        trainLabels.append(0)

trainData = np.array(trainData) #convert to np array
trainLabels = np.array(trainLabels) #convert to np array
trainData = np.delete(trainData,np.s_[1024:3072],1) # Delete 2 colour channels
trainLabels =trainLabels.reshape(len(trainLabels),1) #Reshape
print(trainData.shape) #print shape to to know for inputting points in the Neurel Network
print(trainLabels.shape) #print shape to to know for inputting points in the Neurel Networ
```

```
(4047, 1024)
(4047, 1)
```

```
#test data
testbatch = loadbatch('test_batch') #load test batch
testbatchData1 = testbatch[b'data'] # Create features data set
testbatchLabels1 = testbatch[b'labels'] # Create labels dataset

testData = []#create empty list for filtered test features data
testLabels = [] ##create empty list for filtered test labels

#filter data to only ship and automibile observations
for i in testbatchLabels1:
    labl = names[testbatchLabels1[i]]
    if labl == b'ship':
        testData.append(testbatchData1[i])
        testLabels.append(1)
    if labl == b'automobile':
        testData.append(testbatchData1[i])
        testLabels.append(0)

testData = np.array(testData) #convert to np array
testLabels = np.array(testLabels) #convert to np array
testData = np.delete(testData,np.s_[1024:3072],1) # Delete 2 colour channels
testLabels = testLabels.reshape(len(testLabels),1) #reshape
print(testData.shape) #print shape to to know for inputting points in the Neurel Network
print(testLabels.shape) #print shape to to know for inputting points in the Neurel Network
```

```
(4000, 1024)
(4000, 1)
```

```
#build Large model
```

```
#build Large model
#instance of network model
ln = NN(4047, 1024, 50)
#builds the network model
ln.train(x=trainData, y=trainLabels, epochs=1000, a = .05)
#test model
a, b, c, outY = ln.feedforward(testData) #a, b, c are needed in NN but not when testing


outY = outY.reshape(outY.shape[0]) #reshape output

#If out is greater than .5 change to 1 otherwise change to 0 for classification

for i in range(len(outY)):
    if outY[i] < .5:
        outY[i] = 0
    else:
        outY[i] = 1

#print F-score and Accuracy
print("F score:", f1_score(testLabels , outY, average='macro'))
print("Accuracy:", metrics.accuracy_score(testLabels, outY))
```

```
F score: 0.3333333333333333
Accuracy: 0.5
```

Large dataset observations: 50% accuracy was the best that could be achieved with reasonably ar
can take up to 20 minutes with little or no improvement. For the amount of inputs in the network s
convergence.

PART 4: Neural Network Enhancements

MARK ENHANCEMENT START--

Algorithm - Enhancement Mark: This algorithm is nearly identical to the one originally created. Wh
activation function. Instead of using the more traditional sigmoid this one is using the newer ReLu
learning and generally has better preformace than sigmoid or tanh.

```
#Enhancment - Mark Makris
#Malik, U. (2020) matrix multiplication
#Using ReLu activation function
class NNE(object):
    def __init__(self, points, inputs, hidden): #needs number of data points and hidden la
        self.points = points
        self.wIn = np.random.rand(inputs, hidden)  #weights of input layer (#input, #hidde
        self.wOut = np.random.rand(hidden, 1) #weights of output layer (#hidden, #output)

    def train(self, x, y, epochs=100, a = 0.01):# update weights and biases based on the o
        for epoch in range(epochs):
```

```
            inLine, inReLu, outLine, outReLu = self.feedforward(x)
            dcost_wh, dcost_wo = self.backpropagation(x, y, inLine, inReLu, outLine, outRe
            self.wIn -= a * dcost_wh #edit input weights based on learning rate * error
            self.wOut -= a * dcost_wo #edit output weights based on learning rate * error

    def feedforward(self, x):
        inLine = np.dot(x, self.wIn) #calculate the lines of input weights
        inReLu = self.reLu(inLine) #apply sigmoid to the input lines

        outLine = np.dot(inReLu, self.wOut) #calculate the lines of output weights
        outReLu = self.reLu(outLine) #apply sigmoid to the output lines

        return inLine, inReLu, outLine, outReLu #returns everything

    def backpropagation(self, x, y, inLine, inReLu, outLine, outReLu):
        #output layer
        error_out = ((1 / self.points) * (np.power((outReLu - y), 2)))
        #print(error_out.sum())


        dcost_dos = outReLu - y #calculates error of output
        dos_dol = self.reLu_der(outLine) #calculates derivatives of out line
        dzo_dwo = inReLu #set derivatives of out weights

        dcost_wo = np.dot(dzo_dwo.T, dcost_dos * dos_dol) #derivatives of out weights

        #input layer
        dcost_dol = dcost_dos * dos_dol #costs of output
        dol_dis = self.wOut #reset values to output weights
        dcost_dis = np.dot(dcost_dol , dol_dis.T) #cost of inputs
        dis_dil = self.reLu_der(inLine)  #derivative of inputs
        dil_dwi = x #reset the values for next attempt

        dcost_wi = np.dot(dil_dwi.T, dis_dil * dcost_dis) #cost function of in weights

        return dcost_wi, dcost_wo #return costs

    #reLu h function
    def reLu(self, x):
        return x * (x > 0)

    #reLu activation function
    def reLu_der(self, x):
        return 1 * (x > 0)
```

```
#build Large model
#instance of network model
lne = NNE(4047, 1024, 50)
#builds the network model
lne.train(x=trainData, y=trainLabels, epochs=10, a = .05)
#test model
a, b, c, outYe = lne.feedforward(testData) #a, b, c are needed in NN but not when testing
```

```
#performance of enhanced large data set
outYe = outYe.reshape(outYe.shape[0])

for i in range(len(outYe)):
    if outYe[i] < .5:
        outYe[i] = 0
    else:
        outYe[i] = 1

# f score
score = f1_score(testLabels , outYe, average='macro')


print("F score:", score)
print("Accuracy:", metrics.accuracy_score(testLabels, outYe))
```

```
F score: 0.3333333333333333
Accuracy: 0.5
```

Enhancement observations: We expected to see some inprovement of some sort by changing the
seeing any at all and actually had the same results as the original network

MARK ENHANCEMENT END--

KIERAN ENHANCEMENT START---

For the enchancement to the neural network a second hidden layer is used. Initially had planned tc
and found it hard to find much tutorials that did so without using packages such as tensorflow. Frc
to break down the feature further before coming to the output.

```
# Kieran Brady Neurel Network Enhancement
# Neural net from part 1 enhcanced with second hidden layer

class ENN(object):
    def __init__(self, points, inputs, hidden1, hidden2): #needs number of data points and
        self.points = points
        self.wIn = np.random.rand(inputs, hidden1)
        self.Wmid = np.random.rand(hidden1,hidden2) #weights of input layer (#input, #hidd
        self.wOut = np.random.rand(hidden2, 1) #weights of output layer (#hidden, #output)

    def train(self, x, y, epochs=100, a = 0.01):# update weights and biases based on the o
        for epoch in range(epochs):
            inLine, inSig, midLine, midsig, outLine, outSig = self.feedforward(x)
            dcost_wh, dcost_wm, dcost_wo = self.backpropagation(x, y, inLine, inSig, midLi
            self.wIn -= a * dcost_wh #edit input weights based on learning rate * error
            self.Wmid -= a * dcost_wm #edit input weights based on learning rate * error
            self.wOut -= a * dcost_wo #edit output weights based on learning rate * error

    def feedforward(self, x):
        inLine = np.dot(x, self.wIn) #calculate the lines of input weights
```

```
        inSig = self.sigmoid(inLine) #apply sigmoid to the input lines

        midLine = np.dot(inSig, self.Wmid)
        midsig = self.sigmoid(midLine)

        outLine = np.dot(midsig, self.wOut) #calculate the lines of output weights
        outSig = self.sigmoid(outLine) #apply sigmoid to the output lines

        return inLine, inSig, midLine, midsig, outLine, outSig #returns everything

    def backpropagation(self, x, y, inLine, inSig, midLine, midsig, outLine, outSig):
        #output layer
        error_out = ((1 / self.points) * (np.power((outSig - y), 2)))



        dcost_dos = outSig - y #calculates error of output
        dos_dol = self.sigmoid_der(outLine) #calculates derivatives of out line
        dzo_dwo = midsig #set derivatives of out weights

        dcost_wo = np.dot(dzo_dwo.T, dcost_dos * dos_dol) #derivatives of out weights
        #mid layer
        dcost_dml = dcost_dos * dos_dol
        dol_dismid = self.wOut
        dcost_dismid = np.dot(dcost_dml , dol_dismid.T)
        dis_dilmid = self.sigmoid_der(midLine)
        dil_dwimid = inLine

        dcost_wimid = np.dot(dil_dwimid.T, dis_dilmid * dcost_dismid)

        #input layer
        dcost_dol = dcost_dml * dis_dilmid
        dol_dis = self.Wmid
        dcost_dis = np.dot(dcost_dol , dol_dis.T)
        dis_dil = self.sigmoid_der(inLine)
        dil_dwi = x

        dcost_wi = np.dot(dil_dwi.T, dis_dil * dcost_dis)

        return dcost_wi, dcost_wimid, dcost_wo #return costs

    #sigmoid = 1/(1=e^x)
    def sigmoid(self, x):
        return 1/(1+np.exp(-x))

    #deriv = sig * (1-sig)
    def sigmoid_der(self, x):
        return self.sigmoid(x) *(1-self.sigmoid(x))


#build Large model - Enhanced
#instance of network model
ln = ENN(4047, 1024, 500, 100)
#builds the network model
ln.train(x=trainData, y=trainLabels, epochs=1000, a = .1)
#test model
```

```
a, b, c, d, e, outY2 = ln.feedforward(testData) #a, b, c, d, e are needed in NN but not wh

outY2 = outY2.reshape(outY.shape[0]) #reshape output

#If out is greater than .5 change to 1 otherwise change to 0 for classification
for i in range(len(outY2)):
    if outY2[i] < .5:
        outY2[i] = 0
    else:
        outY2[i] = 1
#F-score & Accuracy
print("F score:", f1_score(testLabels , outY2, average='macro'))
print("Accuracy:", metrics.accuracy_score(testLabels, outY2))
```

```
F score: 0.3333333333333333
Accuracy: 0.5
```

Kieran Enhancement obsercations: From completion of the enhancement similar result are seen t
This is likely due to many more epochs being needed to get better accuracy so it is unclear if the s
at this point.

KIERAN ENHANCEMENT END---

Work split - Mark Makris: Built up the majority of the neural network with the backpropagation and
reported on the portions coded up.

Work split - Kieran Brady: Assisted in some of the neural network design and set up all of the big d
up.

Citations: Dr. Madden, Michael(2020, February 10) CT5133_02_NeuralNets 6-40. NUIG, Galway, Irel

Creating a Neural Network from Scratch in Python: Adding Hidden Layers. [online] Stack Abuse. Av
a-neural-network-from-scratch-in-python-adding-hidden-layers/ [Accessed 10 Feb. 2020].

Load and View CIFAR-10 Data, Michael Madden, Jan 2019.