Neural Networks and Fuzzy Logic

BITS F312

Assignment 2

Name: Tejas Radhakrishnan

ID No.: 2017AAPS1736H

# 1. Multilayer Perceptron Neural Network

## 6 fold cross validation was used because of the size of the dataset.

## **Average accuracies:**

## Average training data accuracy over 6 folds: **1.0**

## Average test data accuracy over 6 folds: **0.95391**

## **Hold out method confusion matrix:**

|  |  |  |
| --- | --- | --- |
| No.of instances | Predicted positive | Predicted negative |
| Actual positive | 186 | 0 |
| Actual negative | 0 | 172 |

|  |  |  |
| --- | --- | --- |
| Accuracy | Sensitivity | Specificity |
| 1.0 | 1.0 | 1.0 |

## **Loss vs iteration and training/test data accuracy for one run:**

## **Code**:

## Multilayer perceptron neural network (2 hidden layers)

import numpy as np

import matplotlib.pyplot as plt

from scipy.io import loadmat

mat\_contents = loadmat('data5.mat')

data = mat\_contents['x']

np.random.shuffle(data)

def init\_data():

X = np.array(data[:2148, :-1], dtype = float)

y = np.array(data[:2148, -1], dtype = int)

X = (X - X.mean(axis = 0))/X.std(axis = 0)

return X, y

def affine\_forward(x, w, b):

z = x.dot(w) + b

cache = (x, w, b)

return z, cache

def relu\_forward(x):

a = x

a[a<=0] = 0

cache = x

return a, cache

def affine\_backward(dout, cache):

x, w, b = cache

db = np.sum(dout, axis = 0)

dw = x.T.dot(dout)

dx = dout.dot(w.T)

return dx, dw, db

def relu\_backward(dout, cache):

x = cache

dx = None

dx = np.ones(x.shape)

dx[x<=0] = 0

dx = dx \* dout

return dx

class Twonet(object):

def \_\_init\_\_(self, input\_size, hidden\_size1, hidden\_size2, num\_classes, std=1e-4):

self.W1 = std \* np.random.randn(input\_size, hidden\_size1)

self.b1 = np.zeros(hidden\_size1)

self.W2 = std \* np.random.randn(hidden\_size1, hidden\_size2)

self.b2 = np.zeros(hidden\_size2)

self.W3 = std \* np.random.randn(hidden\_size2, num\_classes)

self.b3 = np.zeros(num\_classes)

def loss(self, X, y = None, reg = 0.0):

N, D = X.shape

scores = None

z1, af\_cache1 = affine\_forward(X, self.W1, self.b1)

h1, relu\_cache1 = relu\_forward(z1)

z2, af\_cache2 = affine\_forward(h1, self.W2, self.b2)

h2, relu\_cache2 = relu\_forward(z2)

z3, af\_cache3 = affine\_forward(h2, self.W3, self.b3)

scores = z3

if y is None:

return scores

loss = None

scores -= scores.max()

scores\_exp = np.exp(scores)

correct\_scores = scores[range(N), y]

correct\_scores\_exp = np.exp(correct\_scores)

loss = np.sum(-np.log(correct\_scores\_exp / np.sum(scores\_exp, axis = 1))) / N

loss += 0.5 \* reg \* (np.sum(self.W1 \* self.W1) + \

np.sum(self.W2 \* self.W2) + np.sum(self.W3 \* self.W3))

num = correct\_scores\_exp

denom = np.sum(scores\_exp, axis = 1)

mask = (np.exp(z3)/denom.reshape(scores.shape[0],1))

mask[range(N),y] = -(denom - num)/denom

mask /= N

dz3 = mask

dh2, dw3, db3 = affine\_backward(dz3, af\_cache3)

dz2 = relu\_backward(dh2, relu\_cache2)

dh1, dw2, db2 = affine\_backward(dz2, af\_cache2)

dz1 = relu\_backward(dh1, relu\_cache1)

dx, dw1, db1 = affine\_backward(dz1, af\_cache1)

dw3 = dw3 + reg \* self.W3

dw2 = dw2 + reg \* self.W2

dw1 = dw1 + reg \* self.W1

wgrad = (dw1, dw2, dw3)

bgrad = (db1, db2, db3)

return loss, wgrad, bgrad

def train(self, X, y, X\_val, y\_val, alpha = 1e-3, alpha\_decay = 0.95,\

reg = 5e-6, num\_iters = 100, batch\_size = 200):

num\_train = X.shape[0]

iterations\_per\_epoch = max(num\_train / batch\_size, 1)

loss\_history = []

train\_acc\_history = []

val\_acc\_history = []

for it in range(num\_iters):

ind = np.random.choice(num\_train, batch\_size)

X\_batch = X[ind,:]

y\_batch = y[ind]

loss, wgrad, bgrad = self.loss(X\_batch, y = y\_batch, reg = reg)

loss\_history.append(loss)

dw1, dw2, dw3 = wgrad

db1, db2, db3 = bgrad

self.W1 -= alpha \* dw1

self.W2 -= alpha \* dw2

self.W3 -= alpha \* dw3

self.b1 -= alpha \* db1

self.b2 -= alpha \* db2

self.b3 -= alpha \* db3

if it % 100 == 0:

print('iteration %d / %d: loss %f' % (it, num\_iters, loss))

if it % iterations\_per\_epoch == 0:

train\_acc = (self.predict(X\_batch) == y\_batch).mean()

val\_acc = (self.predict(X\_val) == y\_val).mean()

train\_acc\_history.append(train\_acc)

val\_acc\_history.append(val\_acc)

alpha \*= alpha\_decay

return {'loss\_history' : loss\_history, 'train\_acc\_history' : \

train\_acc\_history, 'val\_acc\_history' : val\_acc\_history}

def predict(self, X):

y\_pred = np.argmax(self.loss(X), axis = 1)

return y\_pred

input\_size = 72

hidden\_size1 = 30

hidden\_size2 = 30

num\_classes = 2

num\_inputs = 1790

std = 0.1

alpha = 0.3

batch\_size = 1024

reg = 1e-2

num\_iters = 5000

X\_tot, y\_tot = init\_data()

train\_acc , val\_acc = 0, 0

losses = np.empty((6, num\_iters))

val\_accs = []

train\_accs = []

for k in range(6):

X = X\_tot[0 : 1790]

y = y\_tot[0 : 1790]

X\_val = X\_tot[1790 :]

y\_val = y\_tot[1790 :]

Net = Twonet(input\_size, hidden\_size1, hidden\_size2, num\_classes, std)

print("Validation fold : " , k + 1)

stats = Net.train(X, y, X\_val, y\_val, num\_iters = num\_iters,\

alpha = alpha, batch\_size = batch\_size, reg = 0.0)

losses[k] = np.asarray(stats['loss\_history'])

val\_accs = np.asarray(stats['val\_acc\_history'])

train\_accs = np.asarray(stats['train\_acc\_history'])

train\_acc += train\_accs

val\_acc += val\_accs

X\_tot[0 : 358] = X\_val

X\_tot[358 : ] = X

y\_tot[0 : 358] = y\_val

y\_tot[358 : ] = y

train\_acc /= 6

val\_acc /= 6

print(train\_acc[-1], val\_acc[-1])

loss\_hist = np.mean(losses, axis = 0)

plt.subplot(2, 1, 1)

plt.plot(loss\_hist)

plt.xlabel('Iteration')

plt.ylabel('Loss')

plt.subplot(2, 1, 2)

plt.plot(train\_acc, label='train')

plt.plot(val\_acc, label='val')

plt.xlabel('Epoch')

plt.ylabel('Classification accuracy')

plt.tight\_layout

plt.show()

y\_pred = Net.predict(X\_val)

TP, TN, FP, FN = 0, 0, 0, 0

for i in range(len(y\_val)):

if y\_pred[i] == 0 and y\_val[i] == 0:

TN += 1

elif y\_pred[i] == 1 and y\_val[i] == 0:

FP += 1

elif y\_pred[i] == 0 and y\_val[i] == 1:

FN += 1

elif y\_pred[i] == 1 and y\_val[i] == 1:

TP += 1

print(TP, FP)

print(FN, TN)

accuracy = (TP + TN) / (TP + TN + FP + FN)

sensitivity = TP / (TP + FN)

specificity = TN / (TN + FP)

print("accuracy = ", accuracy, "sensitivity = ", sensitivity,\

"specificity = ", specificity)

# 2. Radial Basis function network (RBFN):

**Gaussian kernel function:**

**Hold out method confusion matrix:**

|  |  |  |
| --- | --- | --- |
| No. of instances | Predicted positive | Predicted negative |
| Actual positive | 259 | 33 |
| Actual negative | 21 | 235 |

|  |  |  |
| --- | --- | --- |
| Accuracy | Sensitivity | Specificity |
| 0.90145 | 0.925 | 0.8768 |

6 fold cross validation:

Average test data accuracy: **0.8864**

**Multi-quadric kernel function:**

**Hold out method confusion matrix:**

|  |  |  |
| --- | --- | --- |
| No. of instances | Predicted positive | Predicted negative |
| Actual positive | 263 | 15 |
| Actual negative | 7 | 263 |

|  |  |  |
| --- | --- | --- |
| Accuracy | Sensitivity | Specificity |
| 0.95985 | 0.97407 | 0.94604 |

6 fold cross validation:

Average test data accuracy: **0.960**

**Linear kernel function:**

**Hold out confusion matrix:**

|  |  |  |
| --- | --- | --- |
| No. of instances | Predicted positive | Predicted negative |
| Actual positive | 267 | 16 |
| Actual negative | 4 | 261 |

|  |  |  |
| --- | --- | --- |
| Accuracy | Sensitivity | Specificity |
| 0.9635 | 0.9852 | 0.9422 |

6 fold cross validation:

Average test data accuracy: **0.9585**

Code:

## Radial basis function network

import numpy as np

from scipy.io import loadmat

from sklearn.cluster import KMeans

mat\_contents = loadmat('data5.mat')

data = mat\_contents['x']

np.random.shuffle(data)

def init\_data():

X = np.array(data[:2148, :-1], dtype = float)

y = np.array(data[:2148, -1], dtype = int)

X = (X - X.mean(axis = 0))/X.std(axis = 0)

return X, y

def gaussian(x,center,sigma,beta):

return np.exp(-beta \* (np.linalg.norm(x - center)) \*\* 2)

def multi\_quadric(x, center, sigma, beta):

return ((np.linalg.norm(x - center)) \*\* 2 + sigma \*\* 2) \*\* 0.5

def linear(x, center, sigma, beta):

return np.linalg.norm(x - center)

X\_tot, y\_tot = init\_data()

train\_X = X\_tot[ : 1600]

train\_y = y\_tot[ : 1600]

test\_X = X\_tot[1600 : 2148]

test\_y = y\_tot[1600 : 2148]

def fit\_rbf(train\_X, train\_y, test\_X, test\_y):

km = KMeans(n\_clusters=550)

y\_km = km.fit\_predict(train\_X)

centers = km.cluster\_centers\_

labels = km.predict(train\_X)

sigma = np.zeros((len(centers), 1))

beta = np.zeros((len(centers), 1))

cluster\_size = np.zeros((len(centers), 1))

for i in range(len(train\_X)):

sigma[labels[i]] += np.linalg.norm(train\_X[i] - centers[labels[i]])

cluster\_size[labels[i]] += 1

sigma /= cluster\_size

beta = 1 / 2 \* (sigma \* sigma + 1e-6)

H = np.zeros((len(train\_X), len(centers)))

for i in range(len(train\_X)):

for j in range(len(centers)):

H[i, j] = linear(train\_X[i], centers[j], sigma[j], beta[j])

W = np.dot(np.linalg.pinv(H), train\_y)

#Test run

H\_test = np.zeros([len(test\_X), len(centers)])

for i in range(len(test\_X)):

for j in range(len(centers)):

H\_test[i, j] = linear(test\_X[i], centers[j], sigma[j], beta[j])

y\_pred = np.dot(H\_test, W)

for i in range(len(y\_pred)):

y\_pred[i] = 1 if y\_pred[i]>=0.5 else 0

accuracy = 0

for i in range(len(y\_pred)):

if y\_pred[i] == test\_y[i]:

accuracy +=1

accuracy /= len(y\_pred)

print(accuracy)

return y\_pred, accuracy

y\_pred, \_ = fit\_rbf(train\_X, train\_y, test\_X, test\_y)

for i in range(len(y\_pred)):

y\_pred[i] = 1 if y\_pred[i] > 0.5 else 0

TP, TN, FP, FN = 0,0,0,0

for i in range(len(test\_X)):

if y\_pred[i] == 1 and test\_y[i] == 1:

TP += 1

elif y\_pred[i] == 0 and test\_y[i] == 0:

TN += 1

elif y\_pred[i] == 1 and test\_y[i] == 0:

FP += 1

elif y\_pred[i] == 0 and test\_y[i] == 1:

FN += 1

accuracy = (TP + TN) / (TP + TN + FP + FN)

sensitivity = TP / (TP + FN)

specificity = TN / (TN + FP)

print("accuracy = ", accuracy, "sensitivity = ", sensitivity, "specificity = ", specificity)

print(TP, FP)

print(FN, TN)

avg\_acc = 0

# K - fold cross validation

for k in range(6):

X = X\_tot[0 : 1790]

y = y\_tot[0 : 1790]

X\_val = X\_tot[1790 :]

y\_val = y\_tot[1790 :]

\_, acc = fit\_rbf(X, y, X\_val, y\_val)

avg\_acc += acc

X\_tot[0 : 358] = X\_val

X\_tot[358 : ] = X

y\_tot[0 : 358] = y\_val

y\_tot[358 : ] = y

avg\_acc /= 6

print(avg\_acc)

# 3. Stacked autoencoder based deep neural network:

**Hold out confusion matrix:**

|  |  |  |
| --- | --- | --- |
| No. of instances | Predicted positive | Predicted negative |
| Actual positive | 299 | 42 |
| Actual negative | 32 | 272 |

|  |  |  |
| --- | --- | --- |
| Accuracy | Sensitivity | Specificity |
| 0.88527 | 0.90332 | 0.86624 |

**Code**:

##Stacked autoencoder based deep neural network

import numpy as np

from scipy.io import loadmat

#Load data, shuffle and normalize

mat\_contents = loadmat('data5.mat')

data = mat\_contents['x']

np.random.shuffle(data)

def init\_data():

X = np.array(data[ : , :-1], dtype = float)

y = np.array(data[ : , -1], dtype = int)

X = (X - X.mean(axis = 0))/X.std(axis = 0)

return X, y

X, y = init\_data()

#Hold out method of model evaluation

X\_train, y\_train = X[ :int(0.7 \* len(X))], y[ :int(0.7 \* len(X))]

X\_val, y\_val = X[ int(0.7 \* len(X)): ], y[ int(0.7 \* len(X)): ]

alpha = 0.5

#Sigmoid activation function

def sigmoid(x, derivative=False):

if (derivative == True):

return x \* (1 - x)

return 1 / (1 + np.exp(-x))

#Neural network class

class NeuralNetwork(object):

def \_\_init\_\_(self, sizes):

self.num\_layers = len(sizes)

self.sizes = sizes

self.W = {}

self.a = {}

self.b = {}

#Initialize Weights

for i in range(1, self.num\_layers):

self.W[i] = np.random.randn(self.sizes[i-1], self.sizes[i])

#Initialize biases

for i in range(1, self.num\_layers):

self.b[i] = np.random.randn(self.sizes[i], 1)

#Initialize activations

for i in range(1, self.num\_layers):

self.a[i] = np.zeros([self.sizes[i], 1])

#Forward pass to compute scores

def forward\_pass(self, X):

self.a[0] = X

for i in range(1, self.num\_layers):

self.a[i] = sigmoid(np.dot(self.W[i].T, self.a[i-1]) + self.b[i])

return self.a[self.num\_layers-1]

#Backward pass to update weights

def backward\_pass(self, X, Y, output):

self.d = {}

self.d\_output = (Y - output) \* sigmoid(output, derivative=True)

self.d[self.num\_layers-1] = self.d\_output

#Derivatives of the layers wrt loss

for i in range(self.num\_layers-1, 1, -1):

self.d[i-1] = np.dot(self.W[i], self.d[i]) \* sigmoid(self.a[i-1], derivative=True)

#Updating weights

for i in range(1, self.num\_layers-1):

self.W[i] += alpha \* np.dot(self.a[i-1], self.d[i].T)

#Updating biases

for i in range(1, self.num\_layers-1):

self.b[i] += alpha \* self.d[i]

#Training helper function

def train(self, X, Y):

X = np.reshape(X, (len(X), 1))

output = self.forward\_pass(X)

self.backward\_pass(X, Y, output)

#Get weights

def get\_W(self):

return self.W

#Load specified weights

def load\_W(self, W):

self.W = W

#Scores computation for given input

def get\_a(self, x):

x = np.reshape(x, (len(x), 1))

self.forward\_pass(x)

return self.a

#Helper function for autoencoder chaining

def load\_a(self, a):

self.a = a

#Loss function

def calc\_loss(NN,x ,y):

loss = 0

for i in range(len(x)):

x\_ = np.reshape(x[i], (len(x[i]), 1))

loss += 0.5 / len(x) \* np.sum((y[i] - NN.forward\_pass(x\_)) \*\* 2)

return loss

#Network initialization

autoencoder1 = NeuralNetwork([72, 60, 72])

autoencoder2 = NeuralNetwork([60,40,60])

autoencoder3 = NeuralNetwork([40, 30, 40])

NN = NeuralNetwork([72,60,40,30, 1])

#Autoencoder 1 pretraining

for i in range(200):

for j, row in enumerate(X\_train):

row = np.reshape(row, (72,1))

autoencoder1.train(row, row)

loss = calc\_loss(autoencoder1, X\_train, X\_train)

print("Epoch {}, Loss {}".format(i, loss))

#Scores computation for autoencoder 1

autoencoder2\_input = []

for row in X\_train:

autoencoder2\_input.append(autoencoder1.get\_a(row)[1])

autoencoder2\_input = np.array(autoencoder2\_input)

#Autoencoder 2 pretraining

for i in range(200):

for j, row in enumerate(autoencoder2\_input):

row = np.reshape(row, (60,1))

autoencoder2.train(row, row)

loss = calc\_loss(autoencoder2, autoencoder2\_input, autoencoder2\_input)

print("Epoch {}, Loss {}".format(i, loss))

#Scores computation for autoencoder 2

autoencoder3\_input = []

for row in autoencoder2\_input:

autoencoder3\_input.append(autoencoder2.get\_a(row)[1])

autoencoder3\_input = np.array(autoencoder3\_input)

#Autoencoder 3 pretraining

for i in range(200):

for j, row in enumerate(autoencoder3\_input):

row = np.reshape(row, (40,1))

autoencoder3.train(row, row)

loss = calc\_loss(autoencoder3, autoencoder3\_input, autoencoder3\_input)

print("Epoch {}, Loss {}".format(i, loss))

#Final network weight initialization

W1 = autoencoder1.get\_W()[1]

W2 = autoencoder2.get\_W()[1]

W3 = autoencoder3.get\_W()[1]

W\_final = {}

W\_final[1] = W1

W\_final[2] = W2

W\_final[3] = W3

W\_final[4] = np.random.randn(30, 1)

NN.load\_W(W\_final)

#Training loop

for i in range(500):

print("Epoch: ", i)

for j in range(len(X\_train)):

NN.train(X\_train[j], y\_train[j])

TP,TN,FP,FN = 0,0,0,0

for i in range(len(X\_val)):

x = np.reshape(X\_val[i], (len(X\_val[i]), 1))

x = NN.forward\_pass(x)

p = 0 if x[0] < 0.5 else 1

if p == 1 and y\_val[i] == 1:

TP += 1

elif p == 0 and y\_val[i] == 0:

TN += 1

elif p == 1 and y\_val[i] == 0:

FP += 1

elif p == 0 and y\_val[i] == 1:

FN += 1

print(TP, FP)

print(FN, TN)

accuracy = (TP + TN) / (TP + TN + FP + FN)

sensitivity = TP / (TP + FN)

specificity = TN / (TN + FP)

print("accuracy = ", accuracy, "sensitivity = ", sensitivity, "specificity = ", specificity)

# 4. Extreme learning machine based classifier

**Gaussian activation function:**

6 fold cross validation:

Average test data accuracy: **0.8715**

**Tanh activation function:**

6 fold cross validation:

Average test data accuracy: **0.8519**

**Code**:

## Extreme learning machine based classifier

import numpy as np

from scipy.io import loadmat

from sklearn.preprocessing import normalize

#Gaussian activation function

def gaussian(X, a, b):

K = np.zeros((X.shape[0], hidden\_neurons))

for i in range(K.shape[0]):

for j in range(K.shape[1]):

K[i,j] = np.exp(-b[j] \* np.linalg.norm(a[:,j] - X[i,:]))

return K

#Tanh activation function

def tanh(X):

return np.tanh(X)

#Load data, shuffle and normalize

def init\_data():

X = np.array(data[ : , :-1], dtype = float)

y = np.array(data[ : , -1], dtype = int)

X = normalize(X, axis = 0)

return X, y

mat\_contents = loadmat('data5.mat')

data = mat\_contents['x']

np.random.shuffle(data)

X\_tot, y\_tot = init\_data()

X\_tot = np.insert(X\_tot, 0, 1, axis=1)

#Generate labels matrix

labels = data[:, -1]

y = np.zeros([len(X\_tot), 2])

for i in range(len(labels)):

if labels[i] == 1:

y[i,1] = 1.0

elif labels[i] == 0:

y[i,0] = 1.0

hidden\_neurons = 100

output\_neurons = 2

#K fold cross validation

#Gaussian activation function

print("Gaussian activation function fold accuracies: ")

for k in range(6):

X\_train = X\_tot[0 : 1790]

y\_train = y[0 : 1790]

X\_val = X\_tot[1790 :]

y\_val = y[1790 :]

a = np.random.rand(X\_train.shape[1], hidden\_neurons)

b = np.random.rand(hidden\_neurons)

# Training

H = gaussian(X\_train, a, b)

H\_inv = np.linalg.pinv(H)

W2 = np.matmul(H\_inv, y\_train)

# Testing

H\_T = gaussian(X\_val, a, b)

y\_pred = np.matmul(H\_T, W2)

accuracy = 0

for p in range(len(y\_pred)):

if np.argmax(y\_pred[p]) == np.argmax(y\_val[p]):

accuracy += 1

accuracy = accuracy / len(y\_val)

print(accuracy)

X\_tot[0 : 358] = X\_val

X\_tot[358 : ] = X\_train

y[0 : 358] = y\_val

y[358 : ] = y\_train

#Tanh activation function

print("Tanh activation function fold accuracies:")

for k in range(6):

X\_train = X\_tot[0 : 1790]

y\_train = y[0 : 1790]

X\_val = X\_tot[1790 :]

y\_val = y[1790 :]

a = np.random.rand(X\_train.shape[1],hidden\_neurons)

b = np.random.rand(hidden\_neurons)

# Training

H = tanh(X\_train)

H\_inv = np.linalg.pinv(H)

W2 = np.matmul(H\_inv, y\_train)

# Testing

H\_T = tanh(X\_val)

y\_pred = np.matmul(H\_T, W2)

accuracy = 0

for p in range(len(y\_pred)):

if np.argmax(y\_pred[p]) == np.argmax(y\_val[p]):

accuracy += 1

accuracy = accuracy / len(y\_val)

print(accuracy)

X\_tot[0 : 358] = X\_val

X\_tot[358 : ] = X\_train

y[0 : 358] = y\_val

y[358 : ] = y\_train

# 5. Stacked autoencoder with ELM classifier:

**Hold out confusion matrix:**

|  |  |  |
| --- | --- | --- |
| No. of instances | Predicted positive | Predicted negative |
| Actual positive | 269 | 69 |
| Actual negative | 53 | 254 |

|  |  |  |
| --- | --- | --- |
| Accuracy | Sensitivity | Specificity |
| 0.81085 | 0.83540 | 0.78637 |

**Code:**

## Stacked autoencoder with ELM classifier

import numpy as np

from scipy.io import loadmat

from sklearn.preprocessing import normalize

#Load data, shuffle and normalize

mat\_contents = loadmat('data5.mat')

data = mat\_contents['x']

np.random.shuffle(data)

def init\_data():

X = np.array(data[ : , :-1], dtype = float)

y = np.array(data[ : , -1], dtype = int)

X = normalize(X, axis = 0)

return X, y

#Convert to one-hot

X, y\_ = init\_data()

y = np.zeros((len(y\_), 2))

for i in range(len(y\_)):

if y\_[i]==1:

y[i,1] = 1.0

elif y\_[i]==0:

y[i,0] = 1.0

#Hold out method of model evaluation

X\_train, y\_train = X[ :int(0.7 \* len(X))], y[ :int(0.7 \* len(X))]

X\_val, y\_val = X[ int(0.7 \* len(X)): ], y[ int(0.7 \* len(X)): ]

alpha = 0.5

#Sigmoid activation function

def sigmoid(x, derivative=False):

if (derivative == True):

return x \* (1 - x)

return 1 / (1 + np.exp(-x))

#Tanh activation function

def tanh(x):

return np.tanh(x)

#Neural network class

class NeuralNetwork(object):

def \_\_init\_\_(self, sizes):

self.num\_layers = len(sizes)

self.sizes = sizes

self.W = {}

self.a = {}

self.b = {}

#Initialize Weights

for i in range(1, self.num\_layers):

self.W[i] = np.random.randn(self.sizes[i-1], self.sizes[i])

#Initialize biases

for i in range(1, self.num\_layers):

self.b[i] = np.random.randn(self.sizes[i], 1)

#Initialize activations

for i in range(1, self.num\_layers):

self.a[i] = np.zeros([self.sizes[i], 1])

#Forward pass to compute scores

def forward\_pass(self, X):

self.a[0] = X

for i in range(1, self.num\_layers):

self.a[i] = sigmoid(np.dot(self.W[i].T, self.a[i-1]) + self.b[i])

return self.a[self.num\_layers-1]

#Backward pass to update weights

def backward\_pass(self, X, Y, output):

self.d = {}

self.d\_output = (Y - output) \* sigmoid(output, derivative=True)

self.d[self.num\_layers-1] = self.d\_output

#Derivatives of the layers wrt loss

for i in range(self.num\_layers-1, 1, -1):

self.d[i-1] = np.dot(self.W[i], self.d[i]) \* sigmoid(self.a[i-1], derivative=True)

#Updating weights

for i in range(1, self.num\_layers-1):

self.W[i] += alpha \* np.dot(self.a[i-1], self.d[i].T)

#Updating biases

for i in range(1, self.num\_layers-1):

self.b[i] += alpha \* self.d[i]

#Training helper function

def train(self, X, Y):

X = np.reshape(X, (len(X), 1))

output = self.forward\_pass(X)

self.backward\_pass(X, Y, output)

#Get weights

def get\_W(self):

return self.W

#Load specified weights

def load\_W(self, W):

self.W = W

#Scores computation for given input

def get\_a(self, x):

x = np.reshape(x, (len(x), 1))

self.forward\_pass(x)

return self.a

#Helper function for autoencoder chaining

def load\_a(self, a):

self.a = a

#Loss function

def calc\_loss(NN,x ,y):

loss = 0

for i in range(len(x)):

x\_ = np.reshape(x[i], (len(x[i]), 1))

loss += 0.5 / len(x) \* np.sum((y[i] - NN.forward\_pass(x\_)) \*\* 2)

return loss

#Network initialization

autoencoder1 = NeuralNetwork([72, 60, 72])

autoencoder2 = NeuralNetwork([60,40,60])

#Autoencoder 1 pretraining

for i in range(500):

for j, row in enumerate(X\_train):

row = np.reshape(row, (72,1))

autoencoder1.train(row, row)

loss = calc\_loss(autoencoder1, X\_train, X\_train)

print("Epoch {}, Loss {}".format(i, loss))

#Scores computation for autoencoder 1

autoencoder2\_input = []

for row in X\_train:

autoencoder2\_input.append(autoencoder1.get\_a(row)[1])

autoencoder2\_input = np.array(autoencoder2\_input)

#Autoencoder 2 pretraining

for i in range(500):

for j, row in enumerate(autoencoder2\_input):

row = np.reshape(row, (60,1))

autoencoder2.train(row, row)

loss = calc\_loss(autoencoder2, autoencoder2\_input, autoencoder2\_input)

print("Epoch {}, Loss {}".format(i, loss))

#Inputs to ELM

elm\_input = []

for row in autoencoder2\_input:

elm\_input.append(autoencoder2.get\_a(row)[1])

elm\_input = np.array(elm\_input)

#parameters for ELM

elm\_neurons = 300

output\_neurons = 2

W\_elm = np.random.randn(elm\_input.shape[1], elm\_neurons)

#ELM Training

np.random.seed(1)

elm\_input = np.reshape(elm\_input, (1503, 40))

H = np.matmul(elm\_input, W\_elm)

H = tanh(H)

H\_inv = np.linalg.pinv(H)

W\_final = np.matmul(H\_inv, y\_train)

#Testing on validation dataset

#Autoencoder 1 forward pass

layer1\_out = []

for i, row in enumerate(X\_val):

act = autoencoder1.get\_a(row)[1]

layer1\_out.append(act)

layer1\_out = np.array(layer1\_out)

layer1\_out = np.reshape(layer1\_out, (645, 60))

#Autoencoder 2 forward pass

layer2\_out = []

for i, row in enumerate(layer1\_out):

act = autoencoder2.get\_a(row)[1]

layer2\_out.append(act)

layer2\_out = np.array(layer2\_out)

layer2\_out = np.reshape(layer2\_out, (645, 40))

#ELM forward pass

H\_T = np.matmul(layer2\_out, W\_elm)

H\_T = tanh(H\_T)

y\_pred = np.matmul(H\_T, W\_final)

TP,TN,FP,FN = 0,0,0,0

for i in range(len(y\_pred)):

if np.argmax(y\_pred[i]) == 1 and np.argmax(y\_val[i]) == 1:

TP += 1

elif np.argmax(y\_pred[i]) == 0 and np.argmax(y\_val[i]) == 0:

TN += 1

elif np.argmax(y\_pred[i]) == 1 and np.argmax(y\_val[i]) == 0:

FP += 1

elif np.argmax(y\_pred[i]) == 0 and np.argmax(y\_val[i]) == 1:

FN += 1

print(TP, FP)

print(FN, TN)

accuracy = (TP + TN) / (TP + TN + FP + FN)

sensitivity = TP / (TP + FN)

specificity = TN / (TN + FP)

print("accuracy = ", accuracy, "sensitivity = ", sensitivity, "specificity = ", specificity)