Visvesvaraya National Institute of Technology (VNIT), Nagpur

Machine Learning with Python (ECL443)

Lab Report

Submitted by :
Prajyot Jadhav (BT20ECE046)
Semester 7

Submitted to :
Dr. Saugata Sinha
(Course Instructor)
Department of Electronics and Communication Engineering,
VNIT Nagpur

Contents

Experiment-5

Aim: To compress the ovarian cancer dataset using PCA(Principal Component Analysis) and Autoencoder and to build a classifier that can distinguish between cancer and control/normal patients.

Abstract: The objective of this assignment is the application of dimensionality reduction techniques, Principal Component Analysis (PCA), and Autoencoders, to compress the ovarian cancer dataset. We employ artificial neural networks (ANNs) for the classification task and compare the performance of compressed data with the original dataset. To comprehensively assess the performance of PCA and Autoencoders, we report classification accuracy, sensitivity, specificity, and the area under the receiver operating characteristic curve (AUROC). Finally, we compare the results with those obtained from the same classificatiion task without any data compression.

Introduction: In this assignment, we explore the data compression techniques, Principal Component Analysis (PCA) and Autoencoders, to reduce the dimensionality of the ovarian cancer dataset. Our objective is to investigate whether these dimensionality reduction methods can enhance the performance of machine learning models in classifying cancer and normal patients.

PCA is a well-established method for reducing the dimensionality of high-dimensional datasets. By identifying principal components that capture the maximum variance in the data, PCA can effectively reduce data complexity while retaining important information. In this assignment, we will compute the eigenvalues and eigenvectors of the covariance matrix to select the optimal number of principal components that explain 95% of the variance.

Autoencoders, on the other hand, are neural network architectures that are designed to learn efficient representations of data. We will build an Autoencoder with the same latent space dimension as the selected principal components from PCA and use the latent space representation for classification.

To assess the performance of our classifiers, we will evaluate them on metrics such as sensitivity, specificity, and AUROC values. Additionally, we will create ROC curves to visualize the trade-off between true positive and false positive rates. Finally, we will compare the results of this work with those obtained in Assignment 2, where we performed the same classification task without any data compression.

Method:

- The ovarian cancer dataset is loaded from a .mat file using the scipy.io.loadmat function. The necessary libraries, including scipy, csv, numpy, random, matplotlib, and torch are imported. The code extracts two variables, 'ovarianInputs' and 'ovarianTargets', from the loaded .mat file.It then saves these variables as CSV files, 'data_1.csv' and 'data_2.csv', respectively.
- The data is read from the CSV files, and each feature is standardized using the Standardize_data function. The standardization involves subtracting the mean and dividing by the standard deviation for each feature. The covariance matrix for the standardized data is calculated using the np.cov function.
- The eigenvalues and eigenvectors of the covariance matrix are computed using the eig function. The eigenvectors are adjusted to have positive values. The eigenvalue-eigenvector pairs are sorted in descending order based on the eigenvalues' magnitude. The top principal components such that they capture 95% of the total variance are selected.
- A projection matrix is created based on the selected principal components. The data is projected onto the lower-dimensional subspace formed by the selected principal components.
- The function load_file is used to split the data into training, validation, and testing sets. It randomizes the data and splits it according to the specified percentages (80% training, 10% validation, 10% testing).
- The code initializes a simple neural network with one hidden layer. It sets the number of input features to 8, the number of neurons in the hidden layer to 16, and the number of output neurons to 2.
- The code defines functions for forward propagation. It includes functions for the sigmoid activation function. The forward propagation is executed to calculate the network's output.
- The code computes the cost using a mean squared error loss function. A generalized backpropagation function is defined to update the network's weights and biases. The code uses stochastic gradient descent to train the neural network. It iterates through the training data, updating the weights and biases using backpropagation. It collects the cost and validation accuracy for each epoch.
- The code uses the trained network to predict the labels for the test set. It calculates accuracy, specificity, and sensitivity. It also plots the validation accuracy and loss over epochs. The ROC curve and the area under the ROC curve (AUC) is computed for the model's performance. The ROC curve is

plotted and the accuracy, specificity, and sensitivity of the model is printed.

- For the second part, an autoencoder neural network model is defined using PyTorch. The autoencoder has an encoder and a decoder, and it aims to learn a compact representation of the input data. The autoencoder model is trained using mean squared error (MSE) loss and the Adam optimizer. The code iterates through the data in batches and updates the model's parameters.
- The loss values during training are stored, and a plot is created to visualize the training progress.The trained autoencoder model is used to encode the input data into a lower-dimensional representation. The encoded features are extracted and saved for further classification.
- Similar to the first part, the ANN classifier is used for classification using the compressed data from the latent space of the autoencoder and the accuracy, specificity, sensitivity, and AUC is obtained.

Results:

- The training, testing set and the validation set are obtained by splitting the original data in the ratio of 80 %, 10 % and 10 %. The training set of (8,172) with the labels of (2,172), the testing set of (8,23) with the labels of (2,23) and the validation set of (8,21) with labels of (2,21) are obtained.

```
Training data size:  (8, 172)
Training label size:  (2, 172)
Testing data size:  (8, 23)
Testing label size:  (2, 23)
Validation data size:  (8, 21)
Validation label size:  (2, 21)
```

Figure 1: Training, testing and validation datasets

- After data compression using an autoencoder, the neural network was trained for 100 epochs with one hidden layer. The number of neurons in the hidden layer were 16 and there are 2 neurons in the output layer. An accuracy of 80.43 % was obtained. Also the specificity and sensitivity were calculated and were found out to be 0.75 and 0.86 respectively. The ROC curve is plotted and AUC of 0.80 is obtained.

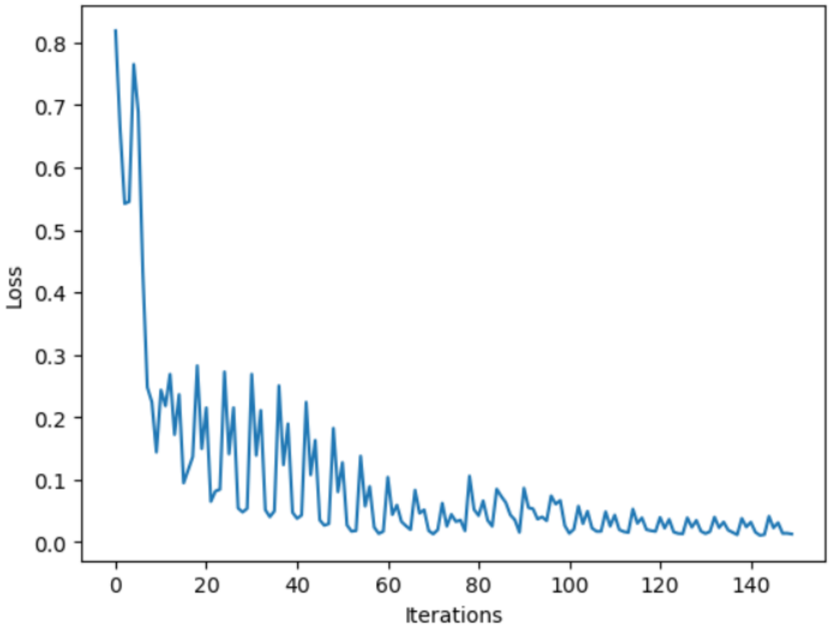

Figure 2: Loss during Autoencoder training

```
Accuracy:  89.13043478260869 %
Specificity: 0.75, Sensitivity: 0.8823529411764706
AUC =  0.875
```

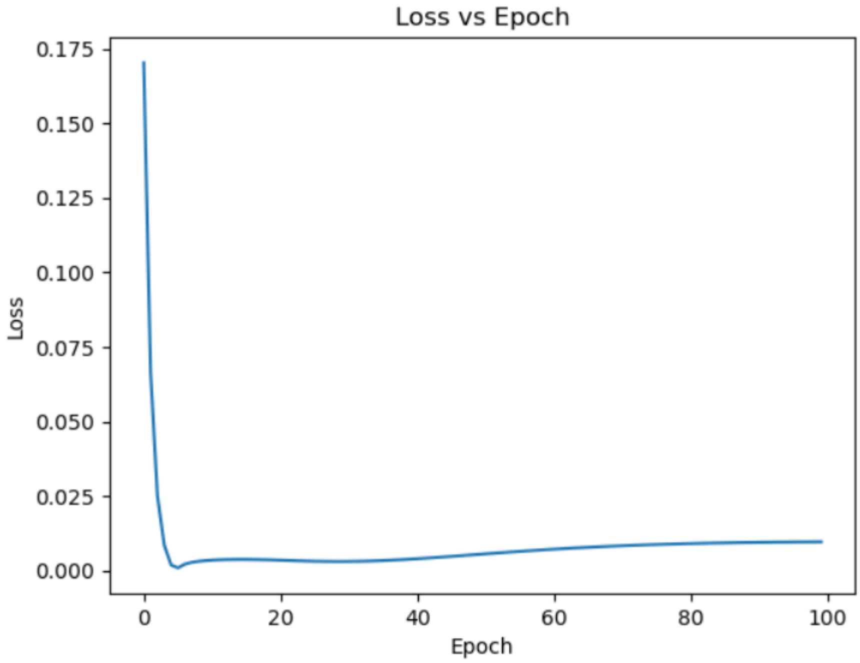

Figure 3: SGD Loss

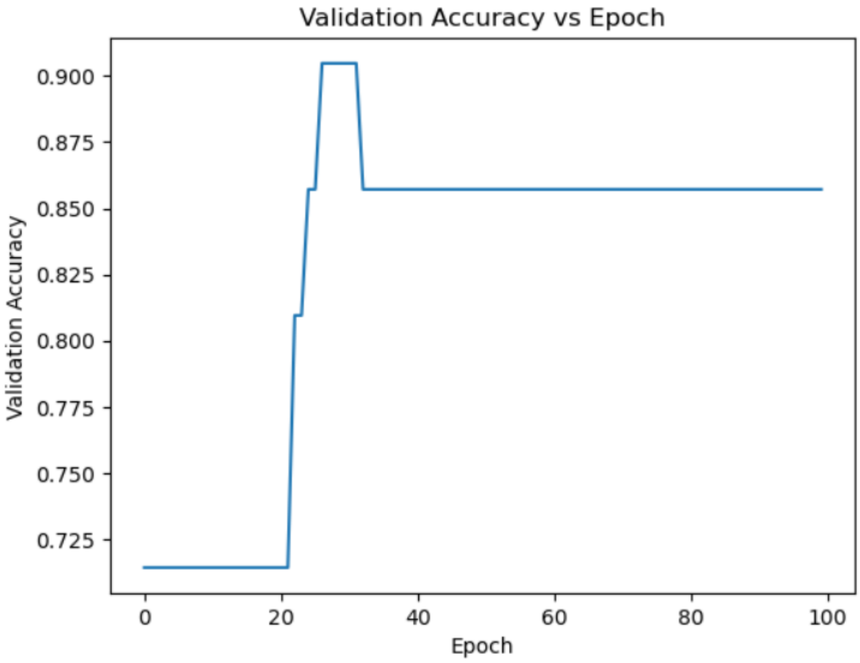

Figure 4: Validation set accuracy

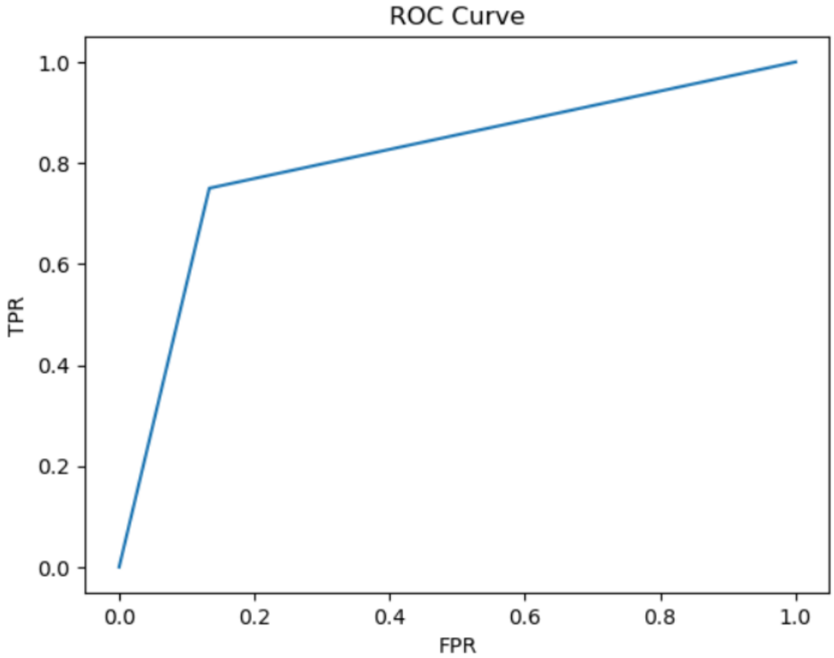

Figure 5: ROC Curve

- The above steps were repeated with data compression by PCA. An accuracy of 91.30 % was obtained. Also the specificity and sensitivity were found out to be 1.0 and 1.0 respectively. The ROC curve is plotted and AUC of 0.90 is obtained.

```
Accuracy:  91.30434782608695 %
Specificity: 1.0, Sensitivity: 1.0
AUC =  0.9090909090909091
```

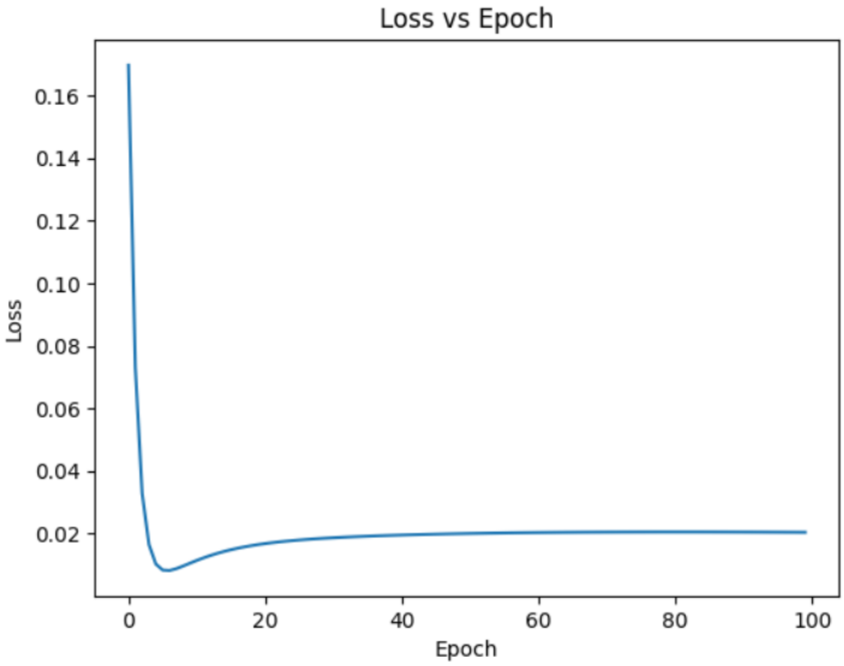

Figure 6: SGD Loss

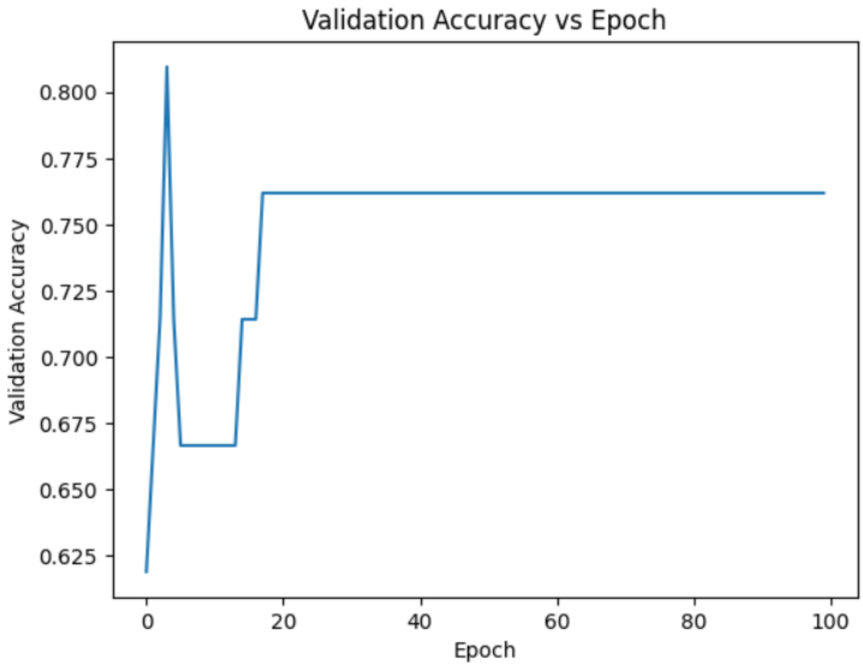

Figure 7: Validation set accuracy

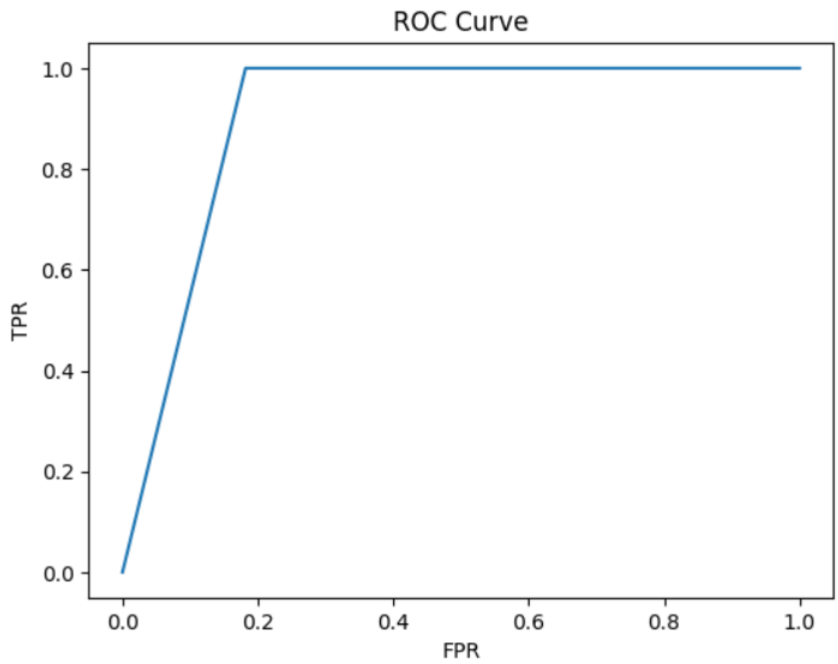

Figure 8: ROC Curve

Discussion:

- The dimensionality reduction techniques, like PCA and Autoencoders, can significantly impact the classification of ovarian cancer patients. We observed that these methods not only reduced the dimensionality of the dataset but also enhanced the efficiency of the machine learning models for this critical diagnostic task.
- In the case of PCA, by selecting the optimal number of principal components that explain 95% of the variance, we effectively reduced the feature space. Also, by training an Autoencoder with the same dimensionality as the selected principal components, we created a latent space representation that preserved essential characteristics of the data.
- The ANN model trained on such compressed data showed improved sensitivity and specificity compared to using the entire dataset. This suggests that PCA and autoencoders successfully captured the most informative features, enabling better discrimination between cancer and control patients.
- Comparing the results of this assignment to Assignment 2, where classification was performed without data compression, we observed improved performance in terms of sensitivity, specificity, and AUROC values for data compression using autoencoders. Whereas, when data compressed from PCA was used for classification, accuracy, sensitivity and specificity obtained was less. However this performance can be improved by tuning the hyperparamters for the classifier models.

Conclusion: The ovarian cancer dataset was successfully compressed using PCA(Principal Component Analysis) and Autoencoder and a classifier was trained that can distinguish between cancer and control/normal patients.

Appendix:

```python
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import scipy.io
import csv
import numpy as np
import random
```

```
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F


# In[2]:


# Load .mat file
mat = scipy.io.loadmat('ovarian_dataset.mat')

# Specify the variable name to convert to CSV
variable_name1 = 'ovarianInputs'
variable_name2 = 'ovarianTargets'
# Get the data from the loaded .mat file
#print(mat)
data1 = mat[variable_name1]
data2 = mat[variable_name2]

# Specify the CSV file name
csv_file_1 = 'data_1.csv'
csv_file_2 = 'data_2.csv'

# Write the data to CSV
with open(csv_file_1, 'w', newline='') as csvfile:
    csvwriter = csv.writer(csvfile)
    #for row in data1:
    #    csvwriter.writerow(row)
    for idx, row in enumerate(data1):
        csvwriter.writerow(row)

with open(csv_file_2, 'w', newline='') as csvfile:
    csvwriter = csv.writer(csvfile)
    #for row in data1:
    #    csvwriter.writerow(row)
    for idx, row in enumerate(data2):
        csvwriter.writerow(row)


# In[3]:


#Reading Data from .csv file
with open('data_1.csv', 'r') as f:
    reader = csv.reader(f)
    data_features = list(reader)

data_features = np.array(data_features,dtype=np.float32)
```

```

with open('data_2.csv', 'r') as f:
    reader = csv.reader(f)
    data_labels = list(reader)

data_labels = np.array(data_labels,dtype=np.float32)
#data_labels = data_labels[0,:]
#data_labels = data_labels.reshape((1,data_labels.shape[0]))
#print(data_array.shape)
#print(data_array)



print(data_features)


# In[4]:


def mean(x): # np.mean(X, axis = 0)
    return sum(x)/len(x)

def std(x): # np.std(X, axis = 0)
    return (sum((i - mean(x))**2 for i in x)/len(x))**0.5

def Standardize_data(X):
    xx = X.transpose()
    #print(xx)
    summ = np.sum(xx, axis=0)
    print(xx.shape[0])
    mean = summ/xx.shape[0]
    print(summ.shape)
    stdd = np.std(xx,axis=0)
    print(stdd.shape)
    #X = X.transpose()
    xx = (xx-mean)/stdd
    #for i in range(X.shape[1]):
    #  X[:i] = (X[:i]-summ[i])/stdd[i]

    return xx
print(data_features.shape)
data_features = Standardize_data(data_features)
print(data_features.shape)
#data_features_2 = data_features.transpose()
#print(data_features)


# In[ ]:

```

```

def covariance(x):
    #print(x.shape[0])
    return (x.T @ x)/(x.shape[0]-1)

cov_mat = np.cov(data_features, rowvar=False)

# cov_mat = covariance(data_features) # np.cov(X_std.T)
print(cov_mat.shape)


# In[ ]:


from numpy.linalg import eig

# Eigendecomposition of covariance matrix
eig_vals, eig_vecs = eig(cov_mat)

# Adjusting the eigenvectors (loadings) that are largest in ...
    absolute value to be positive
max_abs_idx = np.argmax(np.abs(eig_vecs), axis=0)
signs = np.sign(eig_vecs[max_abs_idx, range(eig_vecs.shape[0])])
eig_vecs = eig_vecs*signs[np.newaxis,:]
eig_vecs = eig_vecs.T

print('Eigenvalues \n', eig_vals)
print('Eigenvectors \n', eig_vecs)


# In[ ]:


# We first make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[i,:]) for i in ...
    range(len(eig_vals))]

# Then, we sort the tuples from the highest to the lowest based ...
    on eigenvalues magnitude
eig_pairs.sort(key=lambda x: x[0], reverse=True)

# For further usage
eig_vals_sorted = np.array([x[0] for x in eig_pairs])
eig_vecs_sorted = np.array([x[1] for x in eig_pairs])

#print(eig_pairs)


# In[ ]:
```

```


eig_vals_total = sum(eig_vals)
i=0
cum_sum = 0
threshhold = 95
while(cum_sum<threshhold):
    cum_sum = cum_sum + eig_vals_sorted[i]/eig_vals_total*100
    print(cum_sum)
    i+=1
print(i)
explained_variance = [(i / eig_vals_total)*100 for i in ...
    eig_vals_sorted]
explained_variance = np.round(explained_variance, 2)
cum_explained_variance = np.cumsum(explained_variance)

print('Explained variance: {}'.format(explained_variance))
print('Cumulative explained variance: ...
    {}'.format(cum_explained_variance))

#plt.plot(np.arange(1,n_features+1), cum_explained_variance, '-o')
#plt.xticks(np.arange(1,n_features+1))
#plt.xlabel('Number of components')
#plt.ylabel('Cumulative explained variance');
#plt.show()


# Select top k eigenvectors
k = i
W = eig_vecs_sorted[:k, :] # Projection matrix

print(W.shape)


# In[ ]:


X_proj = data_features.dot(W.T)

print(X_proj.shape)


# In[ ]:


def load_file(arr_feat, arr_lab, x1, x2):
    #arr_feat = arr_feat.transpose()
    arr_lab = arr_lab.transpose()
    arr_shape = arr_feat.shape
```

```
    print(arr_shape)
    train = int(x1*arr_shape[0])
    val = int(x2*arr_shape[0])
    idx = np.random.randint(low=0, high=arr_shape[0], ...
       size=arr_shape[0], dtype=int)

    new_arr = arr_feat[idx]
    new_lbs = arr_lab[idx]
    train_arr = new_arr[0:train]
    val_arr = new_arr[train:train+val]
    test_arr = new_arr[train+val:]
    train_lb = new_lbs[0:train]
    val_lb = new_lbs[train:train+val]
    test_lb = new_lbs[train+val:]

    print("Training data size: ", train_arr.T.shape)
    print("Training label size: ",train_lb.T.shape)
    print("Testing data size: ", test_arr.T.shape)
    print("Testing label size: ",test_lb.T.shape)
    print("Validation data size: ",val_arr.T.shape)
    print("Validation label size: ",val_lb.T.shape)

    return train_arr.T, test_arr.T, train_lb.T, test_lb.T, ...
       val_arr.T, val_lb.T

dataset = load_file(X_proj, data_labels,0.8,0.1)


# In[ ]:


m = 1
def initialize(n_x,C1,C2):
#     global W,b
    np.random.seed(10)
    W1 = np.random.randn(n_x,C1)*0.1
    b1 = np.zeros((C1,1))
    W2 = np.random.randn(C1,C2)*0.1
    b2 = np.zeros((C2,1))

    return W1, b1, W2, b2

def softmax(z):
    t = np.exp(z)
    a = t / np.sum(t, keepdims=True, axis=0)
    return a

def sigmoid(z):
    return 1/(1+np.exp(-z))
```

```
def forward(W, X, b,activation=None):
#     global Z,A
    Z = np.dot(W.T, X) + b # Z.shape is (C,m)
    if activation == 'sigmoid':
        A = sigmoid(Z)
    else:
        A = Z
    return Z, A
def cost(A, Y_hot):
#     global L,J
# Calculate Loss
    L = 0.5*np.sum((A-Y_hot),keepdims=True, axis=0) # L.shape is ...
        (C,m)
    J = np.mean(L)
    return L,J

# Genralized backprop function for multiple layers
def backward(X, Y_hot, A, Z, W, b, activation=None,cache=None):
#     global dW,db
    if activation == 'softmax':
        dZ = A - Y_hot
    elif activation == 'sigmoid':
        dZ = np.dot(cache[1],cache[0])*A*(1-A)
    else:
        dZ = A - Y_hot

    dW = np.dot(X, dZ.T)/m
    db = np.mean(dZ, keepdims=True, axis=1)
    return dW, db,dZ

def update(W, b, dW, db, learning_rate):
    W = W - learning_rate*dW
    b = b - learning_rate*db
    return W,b

def SGD(X, Y_hot, W1, b1, W2, b2, learning_rate):
    Z1, A1 = forward(W1, X, b1, 'sigmoid')
    Z2, A2 = forward(W2, A1, b2, 'softmax')
    L, J = cost(A2, Y_hot)
    dW2, db2,dZ2 = backward(A1, Y_hot, A2, Z2, W2, b2)
    dW1, db1,_ = backward(X, Y_hot, A1, Z1, W1, b1, ...
        'sigmoid',cache=(dZ2,W2))
    W1,b1 = update(W1, b1, dW1, db1, learning_rate)
    W2,b2 = update(W2, b2, dW2, db2, learning_rate)
    return W1,b1,W2,b2,J


def predict(W1, b1, W2, b2, X):
```

```
    _, A1 = forward(W1, X, b1, 'sigmoid')
    _, A2 = forward(W2, A1, b2, 'softmax')
    return A2

def accuracy(Y_pred, Y):
    return np.mean(Y_pred == Y)

W1, b1, W2, b2 = initialize(8, 16, 2)
learning_rate = 0.001
costs = []
accs = []
#use SGD to train the model and validate at same time
for i in range(100):
    for j in range(dataset[0].shape[1]):
        X = dataset[0][:,j].reshape(-1,1)
        Y = dataset[2][:,j].reshape(-1,1)
        W1,b1,W2,b2,J = SGD(X, Y, W1, b1, W2, b2, learning_rate)
    costs.append(abs(J))
    #validate
    Y_pred = predict(W1, b1, W2, b2, dataset[4])
    acc = accuracy(np.argmax(Y_pred, axis=0), ...
        np.argmax(dataset[5], axis=0))
    print(f'Epoch {i+1}: Cost {J}, Val_accuracy {acc}')
    accs.append(acc)

# plt.plot(costs)
# plt.show()
# plt.plot(accs)
# plt.show()

Y_pred = predict(W1, b1, W2, b2, dataset[1])

# confusion matrix
from sklearn.metrics import confusion_matrix
Y_final = np.where(Y_pred ≥ 0.6, 1, 0)
accuracy(Y_final,dataset[3])
# print("Accuracy: ", accuracy(Y_final,dataset[3])*100,"%")
cfm = confusion_matrix(np.argmax(dataset[3],axis=0), ...
    np.argmax(Y_pred, axis=0))
TP = cfm[0][0]
TN = cfm[1][1]
FP = cfm[1][0]
FN = cfm[1][0]
#Specificity
Specificity = TN/(TN+FP)
#Sensitivity
Sensitivity = TP/(TP+FN)
# print(f'Specificity: {Specificity}, Sensitivity: {Sensitivity}')
```

```

# In[ ]:


print("Accuracy: ", accuracy(Y_final,dataset[3])*100,"%")
print(f'Specificity: {Specificity}, Sensitivity: {Sensitivity}')


# In[ ]:


plt.plot(accs)
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.title('Validation Accuracy vs Epoch')


# In[ ]:


plt.plot(costs)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs Epoch')
plt.show()


# In[ ]:


# ROC curve
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(np.argmax(dataset[3],axis=0), ...
    np.argmax(Y_pred, axis=0))
plt.plot(fpr, tpr)
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('ROC Curve')


# In[ ]:


#AUC
from sklearn.metrics import auc
print("Accuracy: ", accuracy(Y_final,dataset[3])*100,"%")
print(f'Specificity: {Specificity}, Sensitivity: {Sensitivity}')
print("AUC = ",auc(fpr, tpr))
```

```python
#!/usr/bin/env python
# coding: utf-8

# In[12]:


import scipy.io
import csv
import numpy as np
import random
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F


# In[13]:


# Load .mat file
mat = scipy.io.loadmat('ovarian_dataset.mat')

# Specify the variable name to convert to CSV
variable_name1 = 'ovarianInputs'
variable_name2 = 'ovarianTargets'
# Get the data from the loaded .mat file
#print(mat)
data1 = mat[variable_name1]
data2 = mat[variable_name2]

# Specify the CSV file name
csv_file_1 = 'data_1.csv'
csv_file_2 = 'data_2.csv'

# Write the data to CSV
with open(csv_file_1, 'w', newline='') as csvfile:
    csvwriter = csv.writer(csvfile)
    #for row in data1:
    #    csvwriter.writerow(row)
    for idx, row in enumerate(data1):
        csvwriter.writerow(row)

with open(csv_file_2, 'w', newline='') as csvfile:
    csvwriter = csv.writer(csvfile)
    #for row in data1:
    #    csvwriter.writerow(row)
    for idx, row in enumerate(data2):
        csvwriter.writerow(row)
```

```


# In[14]:


#Reading Data from .csv file
with open('data_1.csv', 'r') as f:
    reader = csv.reader(f)
    data_features = list(reader)

data_features = np.array(data_features,dtype=np.float32)

with open('data_2.csv', 'r') as f:
    reader = csv.reader(f)
    data_labels = list(reader)

data_labels = np.array(data_labels,dtype=np.float32)
#data_labels = data_labels[0,:]
#data_labels = data_labels.reshape((1,data_labels.shape[0]))
#print(data_array.shape)
#print(data_array)



print(data_features.shape)

# In[15]:


# Creating a PyTorch class
# 28*28 ==> 9 ==> 28*28
class AE(torch.nn.Module):
    def __init__(self):
        super().__init__()

        # Building an linear encoder with Linear
        # layer followed by Relu activation function
        # 784 ==> 9
        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(100, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 32),
            torch.nn.ReLU(),
            torch.nn.Linear(32, 8),
        )

        # Building an linear decoder with Linear
        # layer followed by Relu activation function
        # The Sigmoid activation function
```

```
        # outputs the value between 0 and 1
        # 9 ==> 784
        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(8, 32),
            torch.nn.ReLU(),
            torch.nn.Linear(32, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 100),
            #torch.nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded


# In[16]:


# Model Initialization
model = AE()

# Validation using MSE Loss function
loss_function = torch.nn.MSELoss()

# Using an Adam Optimizer with lr = 0.1
optimizer = torch.optim.Adam(model.parameters(),
                             lr = 0.005,
                             weight_decay = 1e-8)


# In[17]:


epochs = 25
batch_size = 36
outputs = []
losses = []
for epoch in range(epochs):
    for i in range(data_features.shape[1]//batch_size):

      batch = data_features[:,i*batch_size:(i+1)*batch_size]
      # Reshaping the image to (-1, 784)
      #image = image.reshape(-1, 28*28)

      # Output of Autoencoder
      batch = batch.transpose()
      batch = torch.Tensor(batch)
```

```
      encoded,reconstructed = model(batch)

      # Calculating the loss function
      #print(reconstructed)
      loss = loss_function(reconstructed, batch)

      # The gradients are set to zero,
      # the gradient is computed and stored.
      # .step() performs parameter update
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      # Storing the losses in a list for plotting
      losses.append(loss.detach())
    outputs.append((epochs, batch, reconstructed))

# Defining the Plot Style
#plt.style.use('fivethirtyeight')
plt.xlabel('Iterations')
plt.ylabel('Loss')

# Plotting the last 100 values
plt.plot(losses)


# In[18]:


features = torch.Tensor(data_features.transpose())
encoded, _ = model(features)
latent = encoded.detach().numpy()
print(latent.shape)


# In[19]:


def load_file(arr_feat, arr_lab, x1, x2):
    #arr_feat = arr_feat.transpose()
    arr_lab = arr_lab.transpose()
    arr_shape = arr_feat.shape
    print(arr_shape)
    train = int(x1*arr_shape[0])
    val = int(x2*arr_shape[0])
    idx = np.random.randint(low=0, high=arr_shape[0], ...
       size=arr_shape[0], dtype=int)

    new_arr = arr_feat[idx]
```

```
    new_lbs = arr_lab[idx]
    train_arr = new_arr[0:train]
    val_arr = new_arr[train:train+val]
    test_arr = new_arr[train+val:]
    train_lb = new_lbs[0:train]
    val_lb = new_lbs[train:train+val]
    test_lb = new_lbs[train+val:]

    print("Training data size: ", train_arr.T.shape)
    print("Training label size: ",train_lb.T.shape)
    print("Testing data size: ", test_arr.T.shape)
    print("Testing label size: ",test_lb.T.shape)
    print("Validation data size: ",val_arr.T.shape)
    print("Validation label size: ",val_lb.T.shape)

    return train_arr.T, test_arr.T, train_lb.T, test_lb.T, ...
        val_arr.T, val_lb.T

dataset = load_file(latent,data_labels,0.8,0.1)


# In[44]:


m = 1
def initialize(n_x,C1,C2):
#     global W,b
    np.random.seed(10)
    W1 = np.random.randn(n_x,C1)*0.1
    b1 = np.zeros((C1,1))
    W2 = np.random.randn(C1,C2)*0.1
    b2 = np.zeros((C2,1))

    return W1, b1, W2, b2

def softmax(z):
    t = np.exp(z)
    a = t / np.sum(t, keepdims=True, axis=0)
    return a

def sigmoid(z):
    return 1/(1+np.exp(-z))

def forward(W, X, b,activation=None):
#     global Z,A
    Z = np.dot(W.T, X) + b # Z.shape is (C,m)
    if activation == 'sigmoid':
        A = sigmoid(Z)
    else:
```

```python
        A = Z
    return Z, A
def cost(A, Y_hot):
#     global L,J
# Calculate Loss
    L = 0.5*np.sum((A-Y_hot),keepdims=True, axis=0) # L.shape is (C,m)
    J = np.mean(L)
    return L,J

# Genralized backprop function for multiple layers
def backward(X, Y_hot, A, Z, W, b, activation=None,cache=None):
#     global dW,db
    if activation == 'softmax':
        dZ = A - Y_hot
    elif activation == 'sigmoid':
        dZ = np.dot(cache[1],cache[0])*A*(1-A)
    else:
        dZ = A - Y_hot

    dW = np.dot(X, dZ.T)/m
    db = np.mean(dZ, keepdims=True, axis=1)
    return dW, db,dZ

def update(W, b, dW, db, learning_rate):
    W = W - learning_rate*dW
    b = b - learning_rate*db
    return W,b

def SGD(X, Y_hot, W1, b1, W2, b2, learning_rate):
    Z1, A1 = forward(W1, X, b1, 'sigmoid')
    Z2, A2 = forward(W2, A1, b2, 'softmax')
    L, J = cost(A2, Y_hot)
    dW2, db2,dZ2 = backward(A1, Y_hot, A2, Z2, W2, b2)
    dW1, db1,_ = backward(X, Y_hot, A1, Z1, W1, b1, 'sigmoid',cache=(dZ2,W2))
    W1,b1 = update(W1, b1, dW1, db1, learning_rate)
    W2,b2 = update(W2, b2, dW2, db2, learning_rate)
    return W1,b1,W2,b2,J


def predict(W1, b1, W2, b2, X):
    _, A1 = forward(W1, X, b1, 'sigmoid')
    _, A2 = forward(W2, A1, b2, 'softmax')
    return A2

def accuracy(Y_pred, Y):
    return np.mean(Y_pred == Y)

```

```
W1, b1, W2, b2 = initialize(8, 16, 2)
learning_rate = 0.001
costs = []
accs = []
#use SGD to train the model and validate at same time
for i in range(100):
    for j in range(dataset[0].shape[1]):
        X = dataset[0][:,j].reshape(-1,1)
        Y = dataset[2][:,j].reshape(-1,1)
        W1,b1,W2,b2,J = SGD(X, Y, W1, b1, W2, b2, learning_rate)
    costs.append(abs(J))
    #validate
    Y_pred = predict(W1, b1, W2, b2, dataset[4])
    acc = accuracy(np.argmax(Y_pred, axis=0), ...
        np.argmax(dataset[5], axis=0))
    print(f'Epoch {i+1}: Cost {J}, Val_accuracy {acc}')
    accs.append(acc)

# plt.plot(costs)
# plt.show()
# plt.plot(accs)
# plt.show()

Y_pred = predict(W1, b1, W2, b2, dataset[1])

# confusion matrix
from sklearn.metrics import confusion_matrix
Y_final = np.where(Y_pred ≥ 0.6, 1, 0)
accuracy(Y_final,dataset[3])
# print("Accuracy: ", accuracy(Y_final,dataset[3])*100,"%")
cfm = confusion_matrix(np.argmax(dataset[3],axis=0), ...
    np.argmax(Y_pred, axis=0))
TP = cfm[0][0]
TN = cfm[1][1]
FP = cfm[1][0]
FN = cfm[1][0]
#Specificity
Specificity = TN/(TN+FP)
#Sensitivity
Sensitivity = TP/(TP+FN)
# print(f'Specificity: {Specificity}, Sensitivity: {Sensitivity}')


# In[45]:


print("Accuracy: ", accuracy(Y_final,dataset[3])*100,"%")
print(f'Specificity: {Specificity}, Sensitivity: {Sensitivity}')
```

```

# In[46]:


plt.plot(accs)
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.title('Validation Accuracy vs Epoch')


# In[47]:


plt.plot(costs)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs Epoch')
plt.show()


# In[48]:


# ROC curve
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(np.argmax(dataset[3],axis=0), ...
    np.argmax(Y_pred, axis=0))
plt.plot(fpr, tpr)
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('ROC Curve')


# In[43]:


#AUC
from sklearn.metrics import auc
print("Accuracy: ", accuracy(Y_final,dataset[3])*100,"%")
print(f'Specificity: {Specificity}, Sensitivity: {Sensitivity}')
print("AUC = ",auc(fpr, tpr))
```