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-6

Aim: To compress the ovarian cancer dataset using PCA(Principal Component Analysis) and Autoencoder and to evaluate the effectiveness by comparing the reconstructed data with the original data.

Abstract: This assignment focuses on the reconstruction and evaluation of data from Principal Component (PC) space and comparing it to the performance of a trained Autoencoder. In the first part of the assignment, we reconstruct the original data from PC space, and compute the Mean Squared Error (MSE) between the original data and the reconstructed data. This process aims to assess the efficiency of PC space in capturing and representing the original data's variance. In the second part, we compare the MSE obtained from the Autoencoder's reconstructed data, with that of PCA, aiming to discern the differences in their data representation capabilities.

Introduction: Principal Component Analysis (PCA) is a widely used dimensionality reduction technique that helps in capturing the most important patterns or features within a dataset. It achieves this by transforming the data into a new coordinate system represented by Principal Components (PCs), which are orthogonal and ordered by the amount of variance they explain.

In the first part, we aim to assess the quality of data reconstruction from PC space. First the principal components for which the ratio of the sum of the eigen values corresponding to the principle components to be used to the sum of all eigen values is 0.95. Next, by transforming the original data into PC space while considering all Principal Components, we intend to capture as much variance as possible. We then reconstruct the data and calculate the Mean Squared Error (MSE) between the original data and the reconstructed data. A lower MSE indicates a more faithful reconstruction of the original data.

In the second part, we extend our evaluation to include the Autoencoder. We compute the MSE between the data reconstructed by the Autoencoder and the original data and compare this MSE to the one obtained using PCA.

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 data is reconstructed from this compressed data and then MSE loss is calculated between the original data and this reconstructed data.
- 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.
- Similar to the first part, data is reconstructed from a trained autoencoder and then the MSE loss is calculated between original and reconstructed data.

Results:

- The autoencoder was trained for 25 epochs with a batch size of 36. The following loss plot was obtained during the training of the autoencoder.

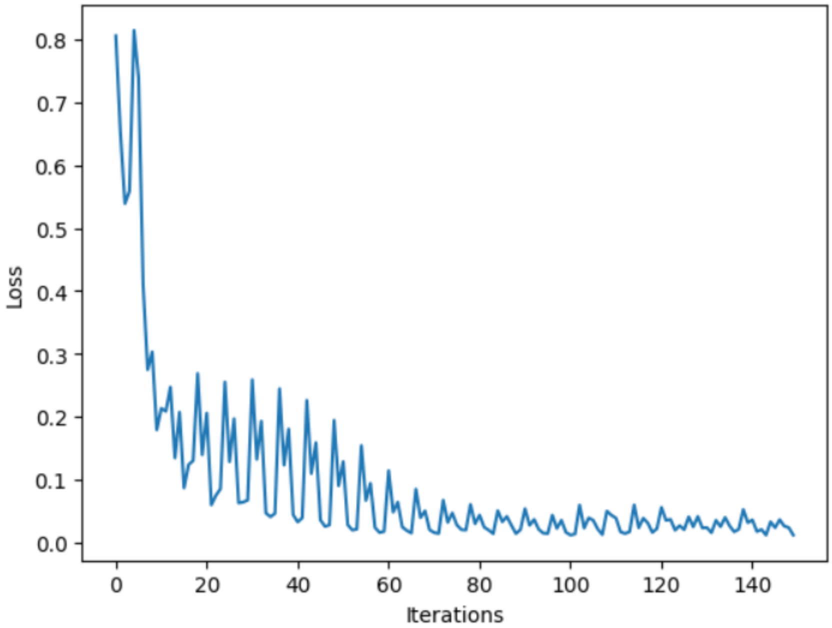

Figure 1: Loss vs Iterations

- After data compression using a PCA, using the principal components for which the ratio of the sum of the eigen values corresponding to the principle components to be used to the sum of all eigen values is 0.95. The data was reconstructed from PC space and the following results were obtained:

```
MSE between original and reconstructed data: 0.0489
```

- The above steps were repeated considering all PC components and the following result was obtained:

```
MSE between original and reconstructed data: 5.691665334111982e-27
```

- After data compression using an autoencoder, following MSE loss was obtained between reconstructed data and original data:

```
MSE between original and reconstructed data: 0.04032338038086891
```

Discussion:

- The results obtained in the first part of the assignment, where we reconstructed the original data from PC space, provided valuable insights into the efficiency of PCA in data representation. The MSE between the original data and the reconstructed data when 8 principal components were considered is significantly more than the MSE loss when all principal components were considered. When all principal components were considered the MSE loss was nearly equal to zero.
- In the second part of the assignment, the Autoencoder's performance was evaluated and compared to PCA. The MSE between the data reconstructed by the Autoencoder with latent space of 8, and the original data is less when compared to the MSE with the one obtained from PCA. A lower MSE for the Autoencoder might indicate its superior ability to model intricate relationships in the data. Autoencoders are capable of capturing intricate patterns, but their performance can be affected by the quality of training data and network architecture.

Conclusion: The ovarian cancer dataset was successfully compressed using PCA(Principal Component Analysis) and Autoencoder and its effectiveness was evaluated by comparing the reconstructed and original data.

Appendix:

```
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

# Load .mat file
mat = scipy.io.loadmat('/content/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 = '/content/data_1.csv'
csv_file_2 = '/content/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)

#Reading Data from .csv file
with open('/content/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('/content/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)

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)

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)

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)

# 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)

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
W1 = eig_vecs_sorted[:k, :] # Projection matrix

print(W1.shape)

X_proj1 = data_features.dot(W1.T)

print(X_proj1.shape)

recons_data_features1 = X_proj1.dot(W1)

MSE_Loss = ((data_features - ...
    recons_data_features1)**2).mean(axis=None)
print(f"MSE between original and reconstructed data: ...
    {MSE_Loss:.4f}")

# for all the principal components
```

```
W2 = eig_vecs_sorted[:, :] # Projection matrix

print(W2.shape)

X_proj2 = data_features.dot(W2.T)

print(X_proj2.shape)

recons_data_features2 = X_proj2.dot(W2)

MSE_Loss2 = ((data_features - ...
    recons_data_features2)**2).mean(axis=None)
print(f"MSE between original and reconstructed data: {MSE_Loss2}")
```

```
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

# Load .mat file
mat = scipy.io.loadmat('/content/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 = '/content/data_1.csv'
csv_file_2 = '/content/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)

#Reading Data from .csv file
with open('/content/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('/content/data_1.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)

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)

# Reconstruct the data using the trained Autoencoder
with torch.no_grad():
    _, reconstructed_data = ...
       model(torch.Tensor(data_features.transpose()))

# Convert the reconstructed_data to a NumPy array
reconstructed_data = reconstructed_data.numpy()

# Calculate the MSE between the original and reconstructed data
mse = ((data_features.transpose() - reconstructed_data)**2).mean()

print(f"MSE between original and reconstructed data: {mse}")
```