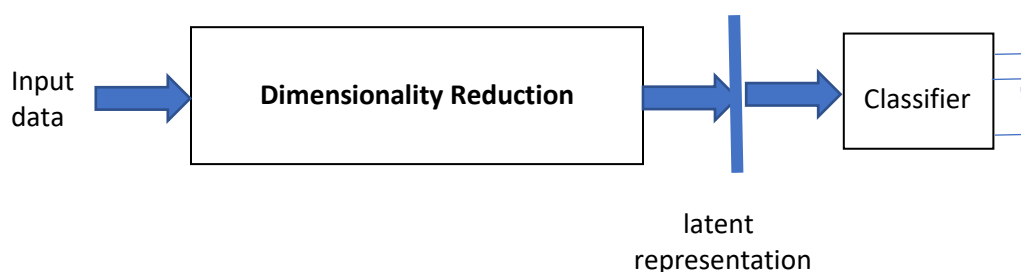


CPSC 552 – Assignment #11

Dimensionality Reduction Using PCA, AutoEncoder, and Variational AutoEncoder for Better Classification

Often, when the dimensionality of data is large, we do a dimensionality reduction using either PCA, or an AutoEncoder, or a Variational AutoEncoder. Such techniques are helpful, especially when the input dimensionality is large and there is sparsity in the features (many zeros).

The general approach in this case is to do the dimensionality reduction, and then pass it to a simpler 2-layer network for classification, as shown below.



Problem 1: For the TCGA-PANCAN cancer dataset where we have 20531 features and five classes, do the dimensionality reduction by PCA first to 20 dimensions, and then use a simple network to classify the data.

Solution: Create a Python application called CancerAnalysisPCANN. Set the Python environment to pytorch1x (as described in Assignment 5). Add a file to the project called MyDataSet.py with the following code in it.

```
from torch.utils.data import Dataset, TensorDataset
import torch
```

```
class MyDataSet(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)
```

Add a file called Utils.py with the following code in it. The following file reads the TCGA-PANCAN dataset, then applies the PCA to it to reduce the dimensionality of data from 20531 to the number of PCA dimensions specified e.g., 20. After reducing the dimensionality of data, it is put in a dataset and

train and test data loaders are associated with it. 150 data items are used for test set out of the total data size of 801.

```
import torch
from MyDataSet import MyDataSet
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
import numpy as np
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

#import tarfile
#import urllib

def get_train_test_loaders_after_pca(pca_dim, batch_size):
    # download TCGA dataset from UCI
    uci_tcga_url = "https://archive.ics.uci.edu/ml/machine-learning-databases/00401/"
    archive_name = "TCGA-PANCAN-HiSeq-801x20531.tar.gz"

    # Build the url
    #full_download_url = urllib.parse.urljoin(uci_tcga_url, archive_name)

    # Download the file
    #r = urllib.request.urlretrieve (full_download_url, archive_name)

    # Extract the data from the archive
    #tar = tarfile.open(archive_name, "r:gz")
    #tar.extractall()
    #tar.close()

    datafile = "D:/PythonAM2/Data/TCGA-PANCAN-HiSeq-801x20531/data.csv"
    labels_file = "D:/PythonAM2/Data/TCGA-PANCAN-HiSeq-801x20531/labels.csv"

    data = np.genfromtxt(
        datafile,
        delimiter=",",
        usecols=range(1, 20532),
        skip_header=1
    )

    true_label_names = np.genfromtxt(
        labels_file,
        delimiter=",",
        usecols=(1,),
        skip_header=1,
        dtype="str"
    )
    print(type(data))
    print(data.shape)
    print(data[:5, :3])

    print(true_label_names[:5])
    # The data variable contains all the gene expression values
```

```

# from 20,531 genes. The true_label_names are the cancer
# types for each of the 801 samples.
# BRCA: Breast invasive carcinoma
# COAD: Colon adenocarcinoma
# KIRC: Kidney renal clear cell carcinoma
# LUAD: Lung adenocarcinoma
# PRAD: Prostate adenocarcinoma

# we need to convert the labels to integers with LabelEncoder:
label_encoder = LabelEncoder()
true_labels = label_encoder.fit_transform(true_label_names)
print(true_labels[:5])
print(label_encoder.classes_)

#-----apply pca to the 20531 dimensional data-----
pca = PCA(n_components=pca_dim)
pca_data = pca.fit_transform(data) # data does not have labels
print(pca_data.shape)
print(pca.explained_variance_ratio_.cumsum())
plt.plot(pca.explained_variance_ratio_)
plt.title("PCA Explained Ratio")
plt.show()

x_tensor = torch.from_numpy(pca_data).float()
y_tensor = torch.from_numpy(true_labels).int()
mydataset = MyDataSet(x_tensor, y_tensor)
train_dataset, test_dataset = random_split(mydataset, [len(mydataset)-150, 150])

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size)
return train_loader, test_loader

```

Add a file called Network.py with the following code in it.

```

import torch
import sys
import torch.nn as nn
import torchvision

# CNN network followed by a with one hidden layer classifier
class Network(nn.Module):
    def __init__(self, input_dim, hidden_size1, num_classes):
        super(Network, self).__init__()
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(input_dim, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, num_classes)
        self.smax = nn.Softmax(dim=1)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        out = self.smax(out)
        return out

```

The above file declares a simple 2 layer neural network to classify the pca reduced data into the five cancer types.

Type the following code in the CancerAnalysisPCANN.py.

```
import sys
import numpy as np
import torch
import Utils
from Network import Network
import torch.nn as nn

def main():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    hidden_size1 = 10
    num_classes = 5
    num_epochs = 20
    batch_size = 10
    learning_rate = 0.001
    pca_dim = 20 # reduce dimensionality to 20

    train_loader, test_loader =
Utils.get_train_test_loaders_after_pca(pca_dim, batch_size)

    model = Network(pca_dim, hidden_size1, num_classes).to(device)

    criterion = nn.CrossEntropyLoss() # for multiclass
    # classification, cross entropy loss works better
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    num_total_steps = len(train_loader)
    for epoch in range(num_epochs):
        for i, (x, labels) in enumerate(train_loader):
            x = x.to(device) # convert to CPU or GPU tensor
            labels = labels.type(torch.LongTensor)
            labels = labels.to(device)

            pred_outputs = model(x) # calls forward function

            loss = criterion(pred_outputs, labels)
            optimizer.zero_grad() # clear gradients
            loss.backward() # compute gradients
            optimizer.step() # update weights and biases
            if (i+1) % 10 == 0:
                print(f'Epoch [{epoch+1}/{num_epochs}],
Step[{i+1}/{num_total_steps}], Loss: {loss.item():.4f}')

        # compute accuracy on test set
        with torch.no_grad():
            num_correct = 0
            num_samples = 0
            for xt, labels in test_loader:
                xt = xt.to(device)
```

```

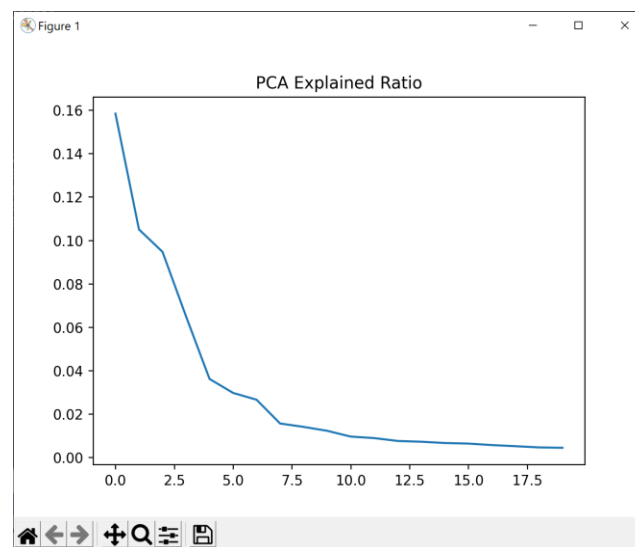
labels = labels.type(torch.LongTensor)
labels = labels.to(device)
outputs = model(xt)
_, predicted = torch.max(outputs, 1) # returns max,max_indices
num_samples += labels.size(0)
num_correct += (predicted == labels).sum()

acc = 100.0 * num_correct / num_samples
print(f'Accuracy of the network on the test set: {acc} %')

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

When doing dimensionality reduction by PCA, we are often interested in determining the variation of each component with respect to all the components (total variance is 100% from all the PCA components). From the following graph you can see, for example, the first PCA component captures 16% of the total variance. The graph indicates that after about 15 principle components, the remaining components do not provide much distinguishing features for our data. Sometimes, the explained ratio graph is used to estimate the number of components needed to do a good classification for a task.



The overall architecture i.e., PCA followed by a two layer neural network is able to achieve an accuracy of 100% on the test data.

```

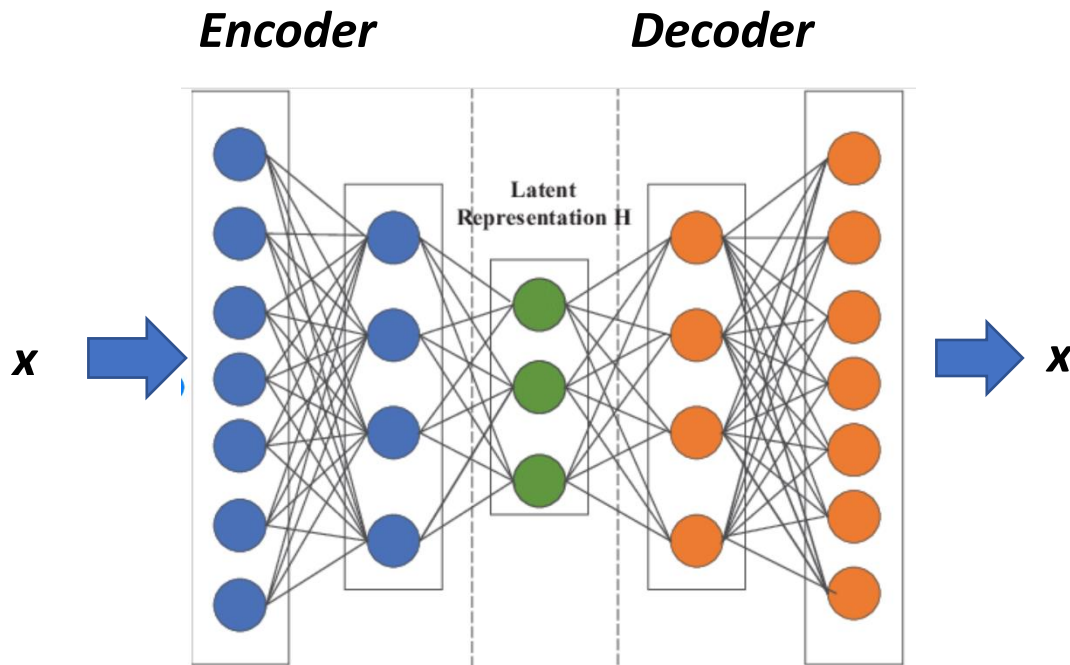
C:\WINDOWS\system32\cmd.exe
Epoch [19/20], Step[20/66], Loss: 0.9056
Epoch [19/20], Step[30/66], Loss: 0.9049
Epoch [19/20], Step[40/66], Loss: 0.9059
Epoch [19/20], Step[50/66], Loss: 0.9050
Epoch [19/20], Step[60/66], Loss: 0.9056
Epoch [20/20], Step[10/66], Loss: 0.9049
Epoch [20/20], Step[20/66], Loss: 0.9055
Epoch [20/20], Step[30/66], Loss: 0.9049
Epoch [20/20], Step[40/66], Loss: 0.9059
Epoch [20/20], Step[50/66], Loss: 0.9050
Epoch [20/20], Step[60/66], Loss: 0.9055
Accuracy of the network on the test set: 100.0 %
Press any key to continue . . .

```

Programming AutoEncoders in PyTorch:

The architecture of an AutoEncoder can be based on accepting a linear column of data (as is the case with most non-image type of data, e.g., cancer data), or based on accepting image type of data.

If the data is in linear form, then a simple design appears as:



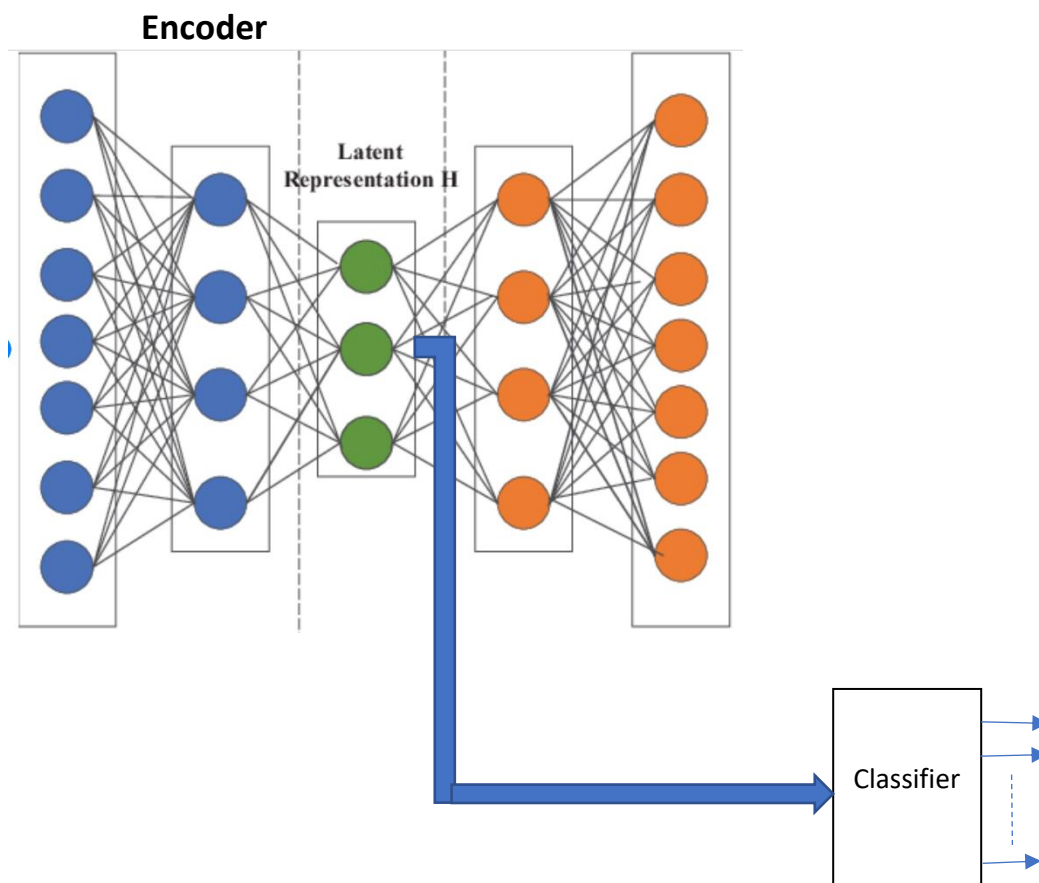
For example, if we were to create an AutoEncoder for the cancer TCGA-PANCAN data, where the input dimensionality is 20531, we could use an autoencoder to reduce the dimensionality to 60, and then to 30 (latent space), and then recover it back to 60 and then to produce the same output as the input size of 20531 features. An autoencoder is trained in an unsupervised manner where the expected output is same as the input, i.e., to train the network, we do not need to know the class of input data. The code for such an autoencoder appears as:

```
class AENetwork(torch.nn.Module):
    def __init__(self, input_dim=20531, hidden_dim=60, latent_dim=30):
        super().__init__()
        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(input_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, latent_dim),
            torch.nn.ReLU() )
        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(latent_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, input_dim),
            torch.nn.ReLU() )

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

Note that we can use 1-D convolutions in the above design instead of linear layers.

Once the autoencoder has been trained, we can save the trained model and then use its latent output to attach to a classifier, and retrain the entire encoder-classifier path for better classification, as shown below.



Problem 2: To demonstrate these concepts, create a new python application called "AutoEncoderCancerAnalysis".

Add a file to the project called MyDataSet.py with the following code in it.

```
from torch.utils.data import Dataset, TensorDataset
import torch
```

```
class MyDataSet(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)
```

Add a file to the project called Utils.py with the following code in it.

```
import torch
from MyDataSet import MyDataSet
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
import numpy as np
from sklearn.preprocessing import LabelEncoder, MinMaxScaler

def get_train_test_loaders(batch_size):
    datafile = "D:/PythonAM2/Data/TCGA-PANCAN-HiSeq-801x20531/data.csv"
    labels_file = "D:/PythonAM2/Data/TCGA-PANCAN-HiSeq-801x20531/labels.csv"

    data = np.genfromtxt(
        datafile,
        delimiter=",",
        usecols=range(1, 20532),
        skip_header=1
    )

    true_label_names = np.genfromtxt(
        labels_file,
        delimiter=",",
        usecols=(1,),
        skip_header=1,
        dtype="str"
    )
    print(type(data))
    print(data.shape)
    print(data[:5, :3])

    print(true_label_names[:5])
    # The data variable contains all the gene expression values
    # from 20,531 genes. The true_label_names are the cancer
    # types for each of the 801 samples.
    # BRCA: Breast invasive carcinoma
    # COAD: Colon adenocarcinoma
    # KIRC: Kidney renal clear cell carcinoma
    # LUAD: Lung adenocarcinoma
    # PRAD: Prostate adenocarcinoma

    # we need to convert the labels to integers with LabelEncoder:
    label_encoder = LabelEncoder()
    true_labels = label_encoder.fit_transform(true_label_names)
    print(true_labels[:5])
    print(label_encoder.classes_)

    x_tensor = torch.from_numpy(data).float()
    y_tensor = torch.from_numpy(true_labels).int()
    mydataset = MyDataSet(x_tensor, y_tensor)
    train_dataset, test_dataset = random_split(mydataset, [len(mydataset)-150, 150])

    train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size)
    test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size)
    return train_loader, test_loader
```


Add a file to the project called AENetwork.py with the following code in it.

```
import torch

class AENetwork(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super().__init__()
        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(input_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, latent_dim),
            torch.nn.ReLU()
        )

        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(latent_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, input_dim),
            torch.nn.ReLU()
        )

    def forward(self, x):
        latent = self.encoder(x)
        decoded = self.decoder(latent)
        return latent, decoded
        # after training, latent will be used in classification
```

Type the following code in the AutoEncoderCancerAnalysis.py to train the autoencoder and save the trained model.

```
import sys
import Utils
from AENetwork import AENetwork
import torch
import torch.nn as nn

def main():
    # train and save just the auto encoder model
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    hidden_size = 60
    num_classes = 5
    num_epochs = 50
    batch_size = 10
    hidden_dim = 60
    latent_dim = 30
    input_dim = 20531
    learning_rate = 0.001

    train_loader, test_loader = Utils.get_train_test_loaders(batch_size)

    model = AENetwork(input_dim, hidden_size, latent_dim).to(device)

    criterion = nn.MSELoss() # for AE, MSE loss works better
```

```

optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

num_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (x, _) in enumerate(train_loader):
        x = x.to(device) # convert to CPU or GPU tensor
        labels = x # for AE, output is same as input

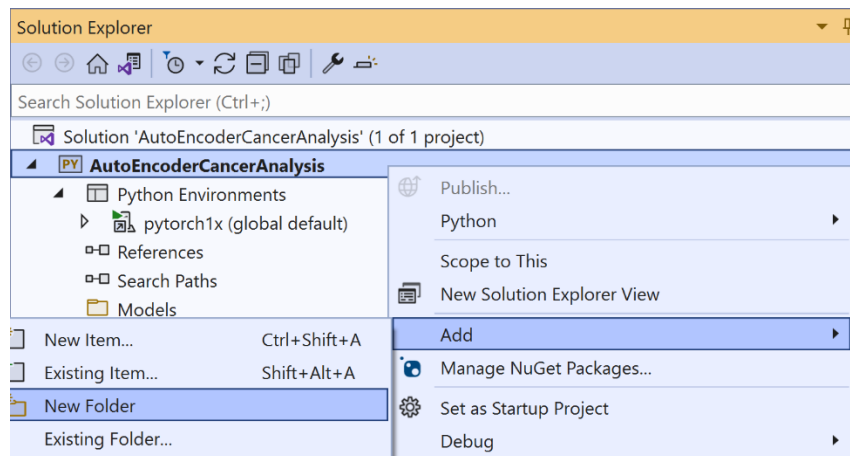
        _, pred_outputs = model(x) # calls forward function

        loss = criterion(pred_outputs, labels)
        optimizer.zero_grad() # clear gradients
        loss.backward() # compute gradients
        optimizer.step() # update weights and biases
        if (i+1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{num_total_steps}], Loss: {loss.item():.4f}')

# save trained AE model
torch.save(model.state_dict(), './Models/save_ae_cancer.pth')
if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

Right click on the project name, and add a new folder to the project called Models.



This is so that the trained autoencoder model can be saved in this folder.

Set the AutoEncoderCancerAnalysis.py as the start file and run the project to train the autoencoder model.

```

C:\WINDOWS\system32\cmd.exe
Epoch [49/50], Step[10/66], Loss: 9.0656
Epoch [49/50], Step[20/66], Loss: 8.8910
Epoch [49/50], Step[30/66], Loss: 8.7170
Epoch [49/50], Step[40/66], Loss: 8.8527
Epoch [49/50], Step[50/66], Loss: 8.8342
Epoch [49/50], Step[60/66], Loss: 8.7082
Epoch [50/50], Step[10/66], Loss: 8.9934
Epoch [50/50], Step[20/66], Loss: 8.7856
Epoch [50/50], Step[30/66], Loss: 8.6179
Epoch [50/50], Step[40/66], Loss: 8.7441
Epoch [50/50], Step[50/66], Loss: 8.7761
Epoch [50/50], Step[60/66], Loss: 8.6788
Press any key to continue . . .

```

Now we will use the trained autoencoder and attach a classifier to its latent output. Add a file to the project called AEClassifierNetwork.py with the following code in it.

```
import torch
from AENetwork import AENetwork

class AEClassifierNetwork(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim, num_classes):
        super().__init__()
        # load pretrained model for the AE
        path = './Models/save_ae_cancer.pth'
        self.AENet = AENetwork(input_dim, hidden_dim, latent_dim)
        self.AENet.load_state_dict(torch.load(path))
        self.AENet.train() # set the parameters to train
        # for fine tuning the entire model

        # attach classifier to AE
        self.fc1 = torch.nn.Linear(latent_dim, 10)
        self.fc2 = torch.nn.Linear(10, num_classes)
        self.smax = torch.nn.Softmax(dim=1)

    def forward(self, x):
        latent, _ = self.AENet(x)
        h1 = self.fc1(latent)
        out = self.fc2(h1)
        return out
```

Add a file called AEClassifierTrainTest.py to the project with the following code in it.

```
import sys
import numpy as np
import torch
import Utils
from AEClassifierNetwork import AEClassifierNetwork
import torch.nn as nn

def main():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    input_dim = 20531
    hidden_dim = 60
    latent_dim = 30
    num_classes = 5
    num_epochs = 15
    batch_size = 10
    learning_rate = 0.001

    train_loader, test_loader = Utils.get_train_test_loaders(batch_size)

    model = AEClassifierNetwork(input_dim, hidden_dim, latent_dim, num_classes).to(device)
    model.train() # set model to train mode
    criterion = nn.CrossEntropyLoss() # for multiclass
    # classification, cross entropy loss works better
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

```

num_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (x, labels) in enumerate(train_loader):
        x = x.to(device) # convert to CPU or GPU tensor
        labels = labels.type(torch.LongTensor)
        labels = labels.to(device)

        pred_outputs = model(x) # calls forward function

        loss = criterion(pred_outputs, labels)
        optimizer.zero_grad() # clear gradients
        loss.backward() # compute gradients
        optimizer.step() # update weights and biases
        if (i+1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{num_total_steps}], Loss: {loss.item():.4f}')

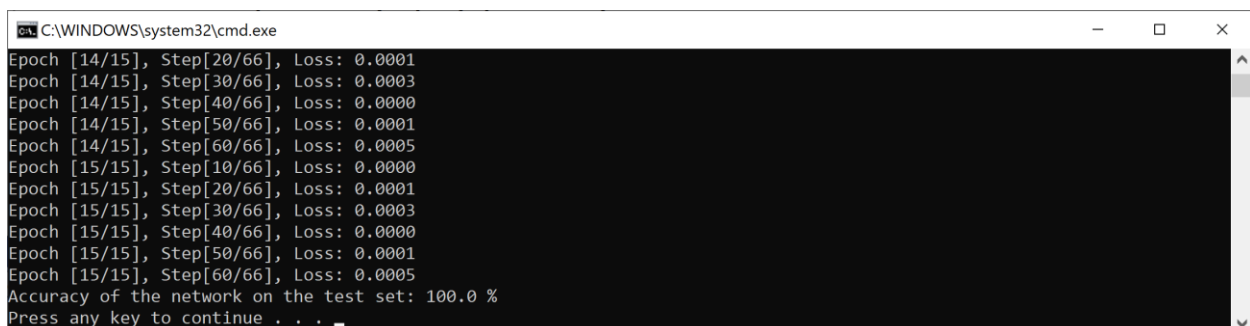
# compute accuracy on test set
with torch.no_grad():
    num_correct = 0
    num_samples = 0
    for xt, labels in test_loader:
        xt = xt.to(device)
        labels = labels.type(torch.LongTensor)
        labels = labels.to(device)
        outputs = model(xt)
        _, predicted = torch.max(outputs, 1) # returns max,max_indices
        num_samples += labels.size(0)
        num_correct += (predicted == labels).sum()

acc = 100.0 * num_correct / num_samples
print(f'Accuracy of the network on the test set: {acc} %')

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

Set the AEClassifierTrainTest.py as the start file. Run the program, it will train the autoencoder (encoder part) and the classifier together. Your output will appear as:



```

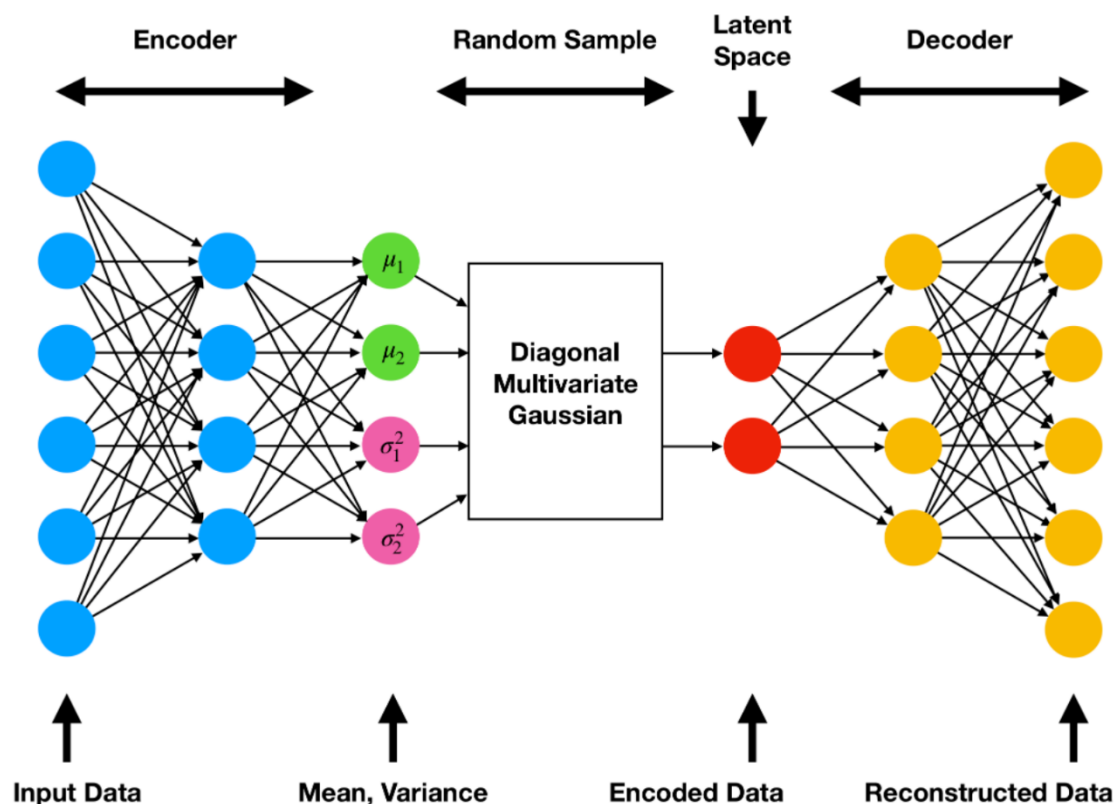
C:\WINDOWS\system32\cmd.exe
Epoch [14/15], Step[20/66], Loss: 0.0001
Epoch [14/15], Step[30/66], Loss: 0.0003
Epoch [14/15], Step[40/66], Loss: 0.0000
Epoch [14/15], Step[50/66], Loss: 0.0001
Epoch [14/15], Step[60/66], Loss: 0.0005
Epoch [15/15], Step[10/66], Loss: 0.0000
Epoch [15/15], Step[20/66], Loss: 0.0001
Epoch [15/15], Step[30/66], Loss: 0.0003
Epoch [15/15], Step[40/66], Loss: 0.0000
Epoch [15/15], Step[50/66], Loss: 0.0001
Epoch [15/15], Step[60/66], Loss: 0.0005
Accuracy of the network on the test set: 100.0 %
Press any key to continue . . .

```

Exercise: Modify the AENetwork to include 1-D convolutions in the first layer. Note that the decoder will need to use ConvTranspose1D (opposite of convolution i.e., deconvolution).

Programming Variational AutoEncoder (VAE) in PyTorch:

VAEs are able to overcome the statistical variations in the data for a given class. The fundamental concept in VAE is to create a latent representation that follows a known distribution e.g., Normal distribution with a mean of 0 and standard deviation of 1. Then depending upon where we sample from this latent distribution, and feed it to a decoder, a particular data item is recreated. The concept also allows us to generate artificial data, and is one of the popular approaches taken in drug discovery.



As you can see from the above picture, the Encoder produces two separate outputs, one for the mean, and the other for the variance of the normal distribution. The sampling process in a VAE, then samples the mean and variance to produce the latent representation \mathbf{z} as:

$$\mathbf{z} = \mu_{\mathbf{z}}(\mathbf{x}) + \epsilon \odot \sigma_{\mathbf{z}}(\mathbf{x})$$

$$\epsilon \sim \mathcal{N}(0, \mathbb{I})$$

Problem 3 - VAE for MNIST Recognition:

Create a Python application project called “VAESimple”. Add a file to the project called Utils.py with the following code in it.

```
import torch
from torchvision import datasets, transforms
from torch.autograd import Variable
from torchvision.utils import save_image

class Utils(object):
    def get_loaders(self, batch_size=100):
        train_dataset = datasets.MNIST(root='./mnist_data/', train=True, transform=transforms.ToTensor(),
download=True)
        test_dataset = datasets.MNIST(root='./mnist_data/', train=False, transform=transforms.ToTensor(),
download=True)

        # data Loaders
        train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size,
shuffle=True)
        test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=False)
        return train_loader, test_loader
```

Add a file to the project called VAEModel.py with the following code in it.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class VAEModel(nn.Module):
    def __init__(self, x_dim, h_dim1, h_dim2, z_dim):
        super(VAEModel, self).__init__()
        # ----encoder components
        self.fce1 = nn.Linear(x_dim, h_dim1)
        self.fce2 = nn.Linear(h_dim1, h_dim2)
        self.fcMu = nn.Linear(h_dim2, z_dim)
        self.fcSigma = nn.Linear(h_dim2, z_dim)
        # ----decoder components
        self.fcd1 = nn.Linear(z_dim, h_dim2)
        self.fcd2 = nn.Linear(h_dim2, h_dim1)
        self.fcdout = nn.Linear(h_dim1, x_dim)

    def encoder(self, x):
        h = F.relu(self.fce1(x))
        h = F.relu(self.fce2(h))
        return self.fcMu(h), self.fcSigma(h) # mu, log_var

    def reparameter_sampling(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return eps.mul(std).add_(mu) # return z sample

    def decoder(self, z):
```

```

h = F.relu(self.fcd1(z))
h = F.relu(self.fcd2(h))
return F.sigmoid(self.fcdout(h))

```

```

def forward(self, x):
    mu, log_var = self.encoder(x.view(-1, 784))
    z = self.reparameter_sampling(mu, log_var)
    #print(z)
    out = self.decoder(z)
    return out, mu, log_var

```

Type the following code in VAESimple.py.

```

import sys
from Utils import Utils
from VAEModel import VAEModel
import torch
import torchvision
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np

def main():
    ngpu = 1
    device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
    utils = Utils()
    train_loader, test_loader = utils.get_loaders()

    vaemodel = VAEModel(x_dim=784, h_dim1=512, h_dim2=256, z_dim=2).to(device)
    epochs = 10
    optimizer = optim.Adam(vaemodel.parameters())
    train(epochs, vaemodel, train_loader, test_loader, optimizer)
    PATH = "./vaemodel.pt"
    # save model
    torch.save(vaemodel, PATH)

    #-----to view randomly generated images by vae
    vaemodel = torch.load(PATH) # load trained model
    vaemodel.eval()
    with torch.no_grad():
        z = torch.randn(64, 2).to(device) # random sample
        gen_sample = vaemodel.decoder(z).to(device)

    #-----to view training images reconstructed by vae
    #-----uncomment following 5 lines
    #train_loader, test_loader = utils.get_loaders(64)
    #real_batch = next(iter(test_loader))
    #print(real_batch[0].shape)
    #gen_images = vaemodel(real_batch[0].to(device))
    #gen_sample = gen_images[0].detach()

    # uncomment following 2 lines to view original images

```

```

#gen_sample = real_batch[0].cuda()
#gen_images = gen_sample.view(64,1,28,28).cpu()

# convert 64x784 to 64,1,28,28 for display
gen_images = gen_sample.view(64,1,28,28).cpu()
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Reconstructed Random Images by VAE")
grid_img = torchvision.utils.make_grid(gen_images, nrow=8)
plt.imshow(grid_img.permute(1, 2, 0))
plt.show()

def train(epochs, vaemodel, train_loader, test_loader, optimizer):
    vaemodel.train() # set it in train mode
    train_loss = 0
    for i in range(epochs):
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.cuda()
            optimizer.zero_grad() # clear gradients

            recon_batch, mu, log_var = vaemodel(data)
            loss = loss_function(recon_batch, data, mu, log_var)

            loss.backward()
            train_loss += loss.item()
            optimizer.step()

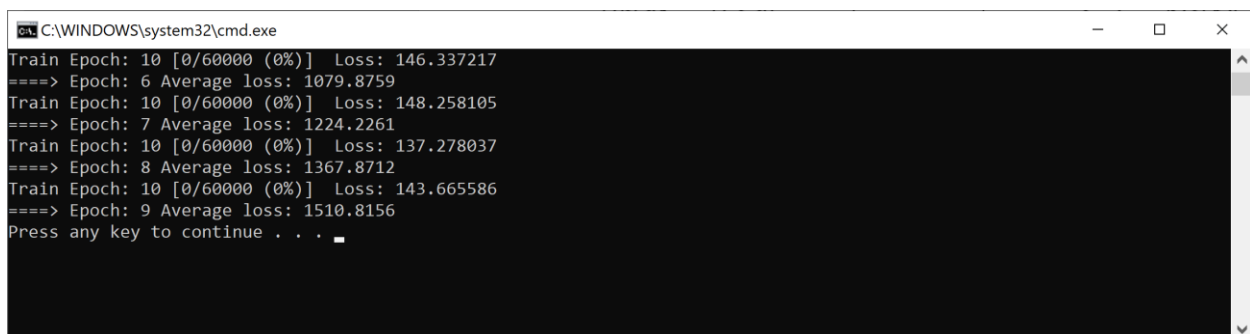
            if batch_idx % 1000 == 0:
                print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                    epochs, batch_idx * len(data), len(train_loader.dataset),
                    100. * batch_idx / len(train_loader), loss.item() / len(data)))
                print('====> Epoch: {} Average loss: {:.4f}'.format(i, train_loss / len(train_loader.dataset)))

def loss_function(recon_x, x, mu, log_var):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 784), reduction='sum')
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
    return BCE + KLD

if __name__ == "__main__":
    sys.exit(int(main()) or 0)

```

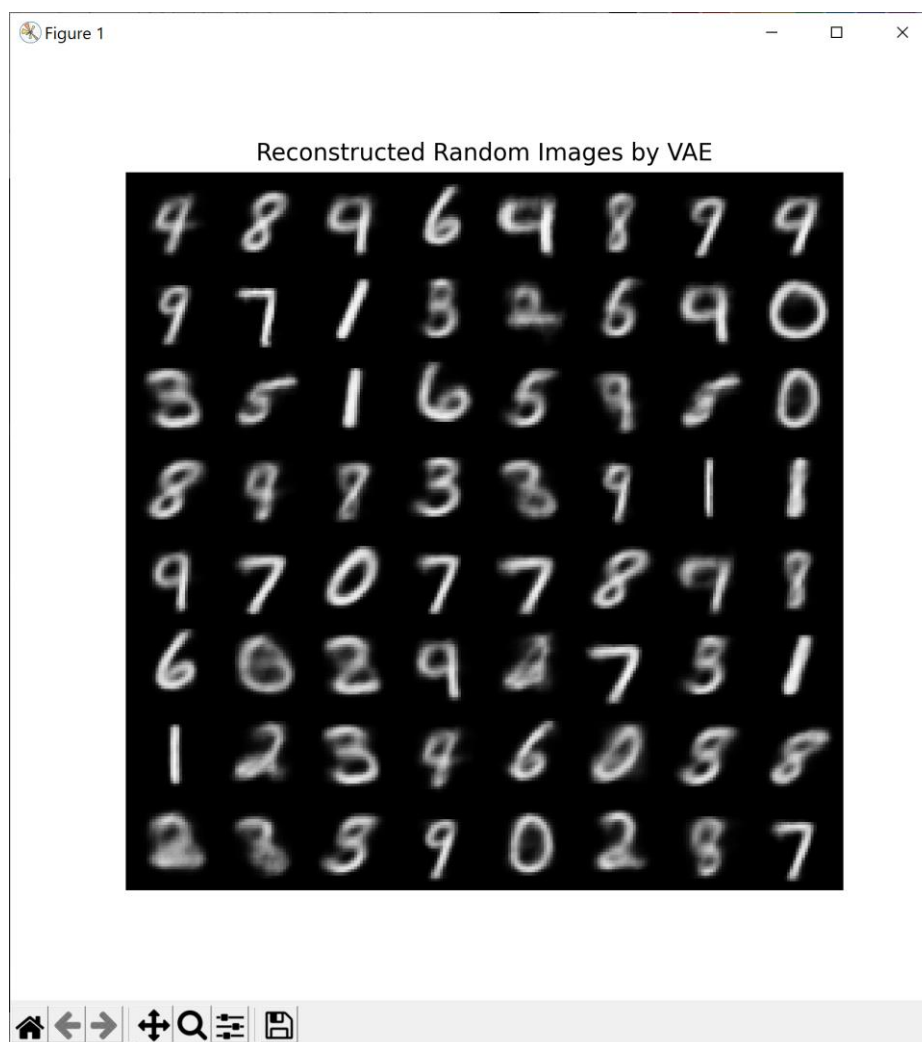
Run the program, your output will appear as:



```

C:\WINDOWS\system32\cmd.exe
Train Epoch: 10 [0/60000 (0%)] Loss: 146.337217
====> Epoch: 6 Average loss: 1079.8759
Train Epoch: 10 [0/60000 (0%)] Loss: 148.258105
====> Epoch: 7 Average loss: 1224.2261
Train Epoch: 10 [0/60000 (0%)] Loss: 137.278037
====> Epoch: 8 Average loss: 1367.8712
Train Epoch: 10 [0/60000 (0%)] Loss: 143.665586
====> Epoch: 9 Average loss: 1510.8156
Press any key to continue . . .

```

Problem 4: Based on the ideas presented in VAE Simple and the previous projects in this assignment, create a project for doing the cancer analysis using a VAE on the TCGA-PANCAN dataset.

Solution: Create a Python application project called VAE Cancer Analysis. Add a file to the project called MyDataSet.py with the following code in it.

```
from torch.utils.data import Dataset, TensorDataset
import torch
```

```
class MyDataSet(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)
```

Add a file called Utils.py to the project with the following code in it.

```
import torch
from MyDataSet import MyDataSet
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
import numpy as np
from sklearn.preprocessing import LabelEncoder, MinMaxScaler

def get_train_test_loaders(batch_size):
    datafile = "D:/PythonAM2/Data/TCGA-PANCAN-HiSeq-801x20531/data.csv"
    labels_file = "D:/PythonAM2/Data/TCGA-PANCAN-HiSeq-801x20531/labels.csv"

    data = np.genfromtxt(
        datafile,
        delimiter=",",
        usecols=range(1, 20532),
        skip_header=1
    )

    true_label_names = np.genfromtxt(
        labels_file,
        delimiter=",",
        usecols=(1,),
        skip_header=1,
        dtype="str"
    )
    print(type(data))
    print(data.shape)
    print(data[:5, :3])
```

```

print(true_label_names[:5])
# The data variable contains all the gene expression values
# from 20,531 genes. The true_label_names are the cancer
# types for each of the 801 samples.
# BRCA: Breast invasive carcinoma
# COAD: Colon adenocarcinoma
# KIRC: Kidney renal clear cell carcinoma
# LUAD: Lung adenocarcinoma
# PRAD: Prostate adenocarcinoma

# we need to convert the labels to integers with LabelEncoder:
label_encoder = LabelEncoder()
true_labels = label_encoder.fit_transform(true_label_names)
print(true_labels[:5])
print(label_encoder.classes_)
dmax = data.max()
x_tensor = torch.from_numpy(data).float()/dmax # max scaling
y_tensor = torch.from_numpy(true_labels).int()
mydataset = MyDataSet(x_tensor, y_tensor)
train_dataset, test_dataset = random_split(mydataset, [len(mydataset)-150, 150])

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size)
return train_loader, test_loader

```

Note that the data is scaled by dividing it by the maximum value. This is needed to achieve stability in training the VAE.

Add a file called VAENetwork with the following code in it.

```

import torch
import torch.nn as nn
import torch.nn.functional as F

class VAENetwork(nn.Module):
    def __init__(self, x_dim, h_dim1, z_dim):
        super(VAENetwork, self).__init__()
        # ----encoder components
        self.fce1 = nn.Linear(x_dim, h_dim1)
        self.fcMu = nn.Linear(h_dim1, z_dim)
        self.fcSigma = nn.Linear(h_dim1, z_dim)
        # ----decoder components
        self.fcd1 = nn.Linear(z_dim, h_dim1)
        self.fcdout = nn.Linear(h_dim1, x_dim)

    def encoder(self, x):
        h = F.relu(self.fce1(x))
        return self.fcMu(h), self.fcSigma(h) # mu, log_var

    def reparameter_sampling(self, mu, log_var):
        std = torch.exp(0.5*log_var)
        eps = torch.randn_like(std)
        return eps.mul(std).add_(mu) # z sample

```

```

def decoder(self, z):
    h = F.relu(self.fcd1(z))
    return F.relu(self.fcdout(h))

def forward(self, x):
    mu, log_var = self.encoder(x)
    z = self.reparameter_sampling(mu, log_var)
    #print(z)
    out = self.decoder(z)
    return out, z, mu, log_var # z is latent rep.

```

Add a file called VAETrain.py to the project. Type the following code in it.

```

import sys
import Utils
from VAENetwork import VAENetwork
import torch
import torchvision
import torch.optim as optim
import torch.nn.functional as F
import numpy as np

device = torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")

def main():
    train_loader, test_loader = Utils.get_train_test_loaders(batch_size=10)

    vaemodel = VAENetwork(x_dim=20531, h_dim1= 60, z_dim=10).to(device)
    epochs = 50
    optimizer = optim.Adam(vaemodel.parameters())
    train(epochs, vaemodel, train_loader, test_loader, optimizer)
    PATH = "./vaemodel.pt"
    # save model
    torch.save(vaemodel, PATH)

def train(epochs, vaemodel, train_loader, test_loader, optimizer):
    vaemodel.train() # set it in train mode
    loss_l1 = torch.nn.L1Loss()
    for i in range(epochs):
        train_loss = 0
        for batch_idx, (data, _) in enumerate(train_loader):
            data = data.to(device)
            optimizer.zero_grad() # clear gradients

            recon_batch, _, mu, log_var = vaemodel(data)
            loss = loss_function(recon_batch, data, mu, log_var, loss_l1)
            loss.backward()
            train_loss += loss.item()
            optimizer.step()

        if batch_idx % 1000 == 0:
            print('Train Epoch: {} [{} / {}] ( {:.0f}%) ] tLoss: {:.6f}'.format(
                epochs, batch_idx * len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item() / len(data)))
            print('====> Epoch: {} Average loss: {:.4f}'.format(i, train_loss / len(train_loader.dataset)))

```

```
def loss_function(recon_x, x, mu, log_var, loss_l1):
    L1 = loss_l1(recon_x, x)
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())
    return L1 + KLD

if __name__ == "__main__":
    sys.exit(int(main() or 0))
```

Set the VAETrain.py as the start file and run the program to train the VAE model. The above program saves the trained model in vaemodel.pt file.

Add a file to the project called VAEClassifierNetwork.py with the following code in it.

```
import torch
from VAENetwork import VAENetwork
import torch.nn.functional as F

class VAEClassifierNetwork(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim, num_classes):
        super().__init__()
        # load pretrained model for the VAE
        path = './vaemodel.pt'
        self.VAENet = torch.load(path)
        self.VAENet.eval()
        self.VAENet.train(False) # do not train vae pre-trained

        # attach classifier to VAE
        self.fc1 = torch.nn.Linear(latent_dim, 10)
        self.fc2 = torch.nn.Linear(10, num_classes)
        self.smax = torch.nn.Softmax(dim=1)

    def forward(self, x):
        _, latent, _ = self.VAENet(x)
        h1 = F.relu(self.fc1(latent))
        out = self.smax(self.fc2(h1))
        return out
```

Add a file called VAEClassifierTrainTest.py. Type the following code in VAEClassifierTrainTest.py.

```
import sys
import numpy as np
import torch
import Utils
from VAEClassifierNetwork import VAEClassifierNetwork
import torch.nn as nn

def main():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    input_dim = 20531
    hidden_dim = 60
    latent_dim = 10
    num_classes = 5
```

```

num_epochs = 30
batch_size = 10
learning_rate = 0.001

train_loader, test_loader = Utils.get_train_test_loaders(batch_size)

model = VAEClassifierNetwork(input_dim, hidden_dim, latent_dim, num_classes).to(device)
model.train()
model.VAENet.train(False) # set vae model to train mode

criterion = nn.CrossEntropyLoss() # for multiclass
# classification, cross entropy loss works better
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

num_total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (x, labels) in enumerate(train_loader):
        x = x.to(device)
        labels = labels.type(torch.LongTensor)
        labels = labels.to(device)

        pred_outputs = model(x) # calls forward function

        loss = criterion(pred_outputs, labels)
        optimizer.zero_grad() # clear gradients
        loss.backward() # compute gradients
        optimizer.step() # update weights and biases
        if (i+1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}], Step[{i+1}/{num_total_steps}], Loss: {loss.item():.4f}')

# compute accuracy on test set
with torch.no_grad():
    num_correct = 0
    num_samples = 0
    for xt, labels in test_loader:
        xt = xt.to(device)
        labels = labels.type(torch.LongTensor)
        labels = labels.to(device)
        outputs = model(xt)
        _, predicted = torch.max(outputs, 1) # returns max, max_indices
        num_samples += labels.size(0)
        num_correct += (predicted == labels).sum()

acc = 100.0 * num_correct / num_samples
print(f'Accuracy of the network on the test set: {acc} %')

if __name__ == "__main__":
    sys.exit(int(main() or 0))

```

Set the VAEClassifierTrainTest.py as the start file. If you run the program, you will get results as shown below.

```
C:\WINDOWS\system32\cmd.exe
Epoch [28/30], Step[60/66], Loss: 0.9049
Epoch [29/30], Step[10/66], Loss: 0.9050
Epoch [29/30], Step[20/66], Loss: 0.9049
Epoch [29/30], Step[30/66], Loss: 0.9049
Epoch [29/30], Step[40/66], Loss: 0.9049
Epoch [29/30], Step[50/66], Loss: 0.9048
Epoch [29/30], Step[60/66], Loss: 0.9049
Epoch [30/30], Step[10/66], Loss: 0.9050
Epoch [30/30], Step[20/66], Loss: 0.9049
Epoch [30/30], Step[30/66], Loss: 0.9049
Epoch [30/30], Step[40/66], Loss: 0.9049
Epoch [30/30], Step[50/66], Loss: 0.9048
Epoch [30/30], Step[60/66], Loss: 0.9049
Accuracy of the network on the test set: 100.0 %
Press any key to continue . . .
```