Georgia Institute of Technology

ECE 4803: Fundamentamentals of Machine Learning (FunML)

Spring 2022

Homework Assignment # 6

Due: Friday, April 15th, 2022 @8PM

Please read the following instructions carefully.

- The entire homework assignment is to be completed on this `ipython` notebook. It is designed to be used with `Google Colab`, but you may use other tools (e.g., Jupyter Lab) as well.
- Make sure that you execute all cells in a way so their output is printed beneath the corresponding cell. Thus, after successfully executing all cells properly, the resulting notebook has all the questions and your answers.
- Print a PDF copy of the notebook with all its outputs printed and submit the **PDF** on `Canvas` under Assignments.
- Make sure you delete any scratch cells before you export this document as a PDF. Do not change the order of the questions and do not remove any part of the questions. Edit at the indicated places only.
- Rename the PDF according to the format: ***LastName_FirstName_ECE_4803_sp22_assignment_#.pdf***
- It is encouraged for you to discuss homework problems amongst each other, but any copying is strictly prohibited and will be subject to Georgia Tech Honor Code.
- Late homework is not accepted unless arranged otherwise and in advance.
- Comment on your codes.
- Refer to the tutorial and the supplementary/reading materials that are posted on `Canvas` for lectures 20, 21 to help you with this assignment.
- **IMPORTANT:** Start your solution with a **BOLD RED** text that includes the words *solution* and the part of the problem you are working on. For example, start your solution for Part (c) of Problem 2 by having the first line as:
  Solution to Problem 2 Part (c). Failing to do so may result in a *20% penalty* of the total grade.

Assignment Objectives:

- Understand the basic function and intuituion behind Autoencoders
- Understand the different modes of Autoencoders
- Understand regularization in Autoencoders

Guide for Exporting Ipython Notebook to PDF:

Here is a video summarizes how to export Ipythin Notebook into PDF.

- **[Method1: Print to PDF]**
  After you run every cell and get their outputs, you can use **[File] -> [Print]** and then choose **[Save as PDF]** to export this Ipython Notebook to PDF for submission.
  Note: Sometimes figures or texts are splited into different pages. Try to tweak the layout by adding empty lines to avoid this effect as much as you can.
- **[Method2: colab-pdf script]**
  The author of that video provided an alternative method that can generate better layout PDF. However, it only works for Ipythin Notebook without embedded images.
  How to use: Put the script below into cells at the end of your Ipythin Notebook. After you run the fisrt cell, it will ask for google drive permission. Executing the second cell will generate the PDF file in your google drive home directory. Make sure you use the correct path and file name.

```
## this will link colab with your google drive
from google.colab import drive
drive.mount('/content/drive')
```

```
%%capture
!wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
from colab_pdf import colab_pdf
colab_pdf('LastName_FirstName_ECE_4803_sp22_assignment_#.ipynb') ## change path and file name
```

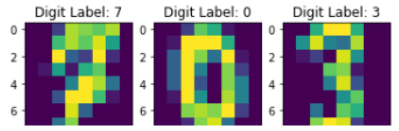
- **[Method3: GoFullPage Chrome Extension] (most recommended)**
  Install the extension and generate PDF file of the Ipython Notebook in the browser.

Note: Georgia Tech provides a student discount for Adobe Acrobat subscription. Further information can be found here.

Problem 1: MLP-based Autoencoders (35pts)

As we saw in the lecture, autoencoders (AEs) are machine learning models that learn to reconstruct their inputs. The aim in training autoencoders is to learn and uncover important relationships underlying large dimensional datasets. In this and the following set of questions, you will be asked to apply various kinds of autoencoders to the `digits` dataset in `sklearn`, a dataset consisting of images of handwritten digits. The following piece of code loads the dataset and prints out images of various digits contained therein.

```
[ ] # imports
    import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.datasets import load_digits
    import random

    # load data
    X, y = load_digits(return_X_y=True)

    # print images of digits
    fig, ax = plt.subplots(1,3)
    for i in range(3):
        idx = random.randint(0, X.shape[0])  # sample an example index
        x_hat, y_hat = X[idx], y[idx]  # extract data and label
        ax[i].imshow(x_hat.reshape(8,8))
        ax[i].set_title('Digit Label: {}'.format(str(y_hat)))

    plt.show()
```



In this question, we are going to design an MLP-based autoencoder to train on this dataset. Based off the particular characteristics of this dataset and what you know of autoencoders, answer the following questions.

(a) What is the number of neurons that has to be in the input layer of this autoencoder?

(b) What is the number of output neurons that has to be in the output layer of this autoencoder?

(c) Using you answers to parts (a) and (b) above, design a pytorch class representing an MLP-based AE containing *exactly one* hidden layer consisting of 5 neurons. The AE should not have any non-linear activation functions whatsoever. You may use the code cell below as a template.

Problem 1 (a) Solution

Number of neurons in the input layer = 8*8 = 64

Problem 1 (b) Solution

Number of neurons in the output layer = 8*8 = 64

Problem 1 (c) Solution

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


    class MyMLPAE(nn.Module):
        def __init__(self, num_hidden_layers, hidden_size):
```

```
[ ]
    class MyMLPAE(nn.Module):
        def __init__(self, num_hidden_layers, hidden_size):
            super(MyMLPAE, self).__init__()
            """Inititalizes the various layers in the network

            Parameters
            ----------
            num_hidden_layers : int
              number of hidden layers.

            hidden_size : int
              number of hidden neurons in the hidden layers.

            """
            ##TODO

            # Hidden layers
            # ENCODER
            self.linear_1 = torch.nn.Linear(64, hidden_size)

            # DECODER
            self.linear_2 = torch.nn.Linear(hidden_size, 64)



        def forward(self, x):
            """Processes the input from the dataloaders to return predicted output
            probability vectors for each example in the batch.

            Parameters
            ----------
            x : torch.tensor, shape(batch_size, 1, 64), dtype torch.float
              output from dataloader containing batch_size number of flattened 8 x 8 images as
              torch tensors

            Returns
            -------
            out : torch.tensor, shape (batch_size, 1, 64), dtype= torch.float.
              batch_size number of flattened 8 x 8 images as torch tensors.
            """
            ##TODO
            x = self.linear_1(x) # the hidden state
        ##TODO
        x = self.linear_1(x) # the hidden state

        out = self.linear_2(x)


        return out
```

(d) Design a pytorch dataset class to process and format examples from the `digits` dataset in an appropriate form to be presented the AE you designed above for training and inference. Once again, you may use the code cell below as a template.

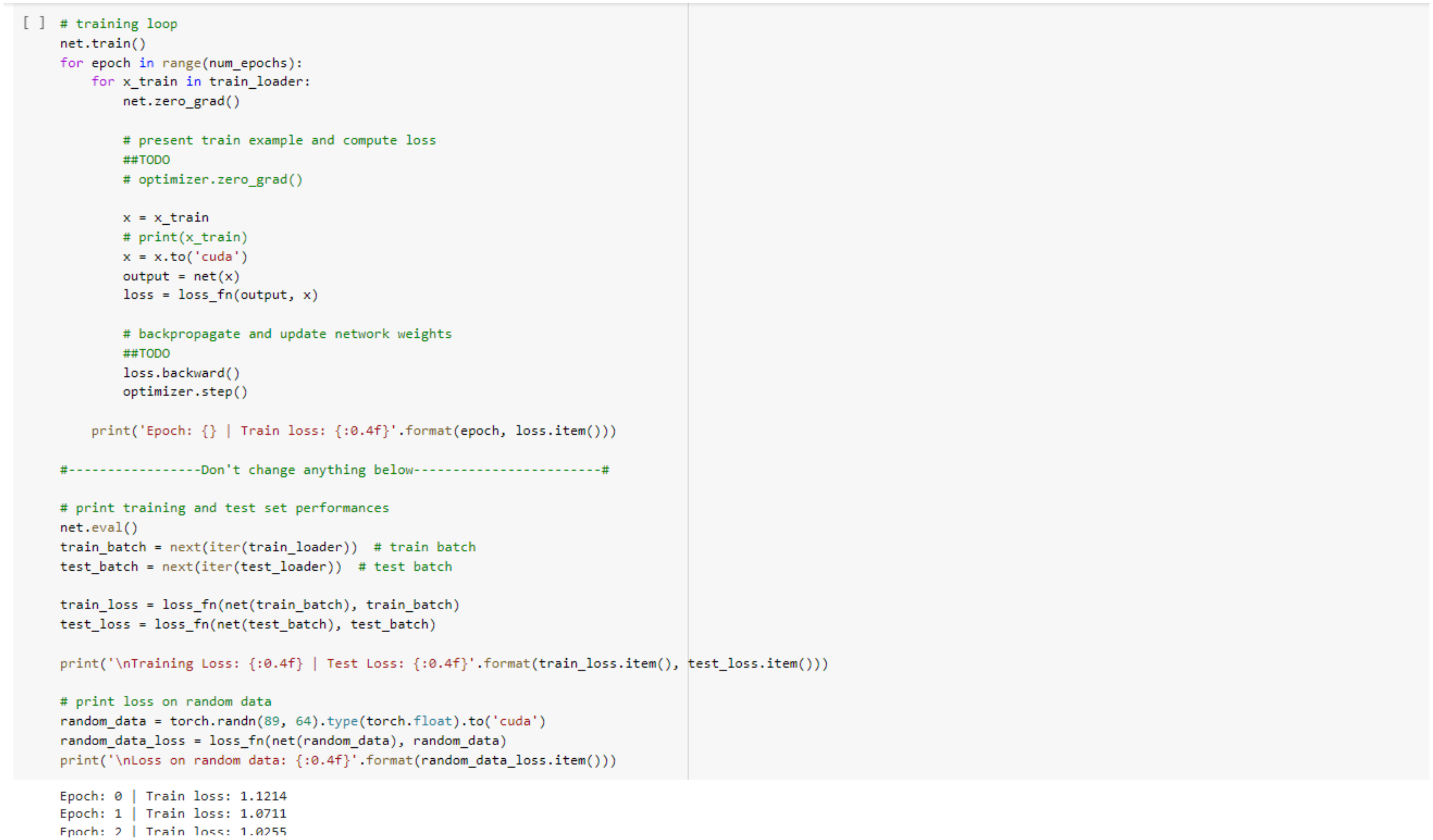
Problem 1 (d) Solution

```
[ ] from torch.utils.data import Dataset

    class DigitsDataset(Dataset):
        def __init__(self, X):
            """Function stores the data arrays returned by load_digits function.

            Parameters
            ----------
            X : array_like, shape(Num_samples, Num_of_features)
              numpy array containing the data matrix containing digits training examples
              and features.

            """
            ##TODO
            self.X = X

        def __getitem__(self, idx):
            """function extracts a single example from X given its index.

            Parameters
            ----------
            idx : int
              index of a single example to be extracted from X

            Returns
        input : torch.tensor, shape(1, 64), type torch.float
          indexed example from X reshaped into a single channel grayscale image of
          size 64 and float datatype.

        """
        ##TODO
        input = torch.tensor(self.X[idx])
        input = input.type(torch.float)
        input = torch.reshape(input, (1,64))
        input = input.to('cuda')


        return input

    def __len__(self):
        return self.X.shape[0]
```

(e) Finally, you are required to train your designed network on a train split created off from the original dataset and finally test the reconstruction performance on the test split. You may use the following code cell as a template. After executing the cell, you should see printed both the training and test set performance in terms of the mean square error reconstruction performance.

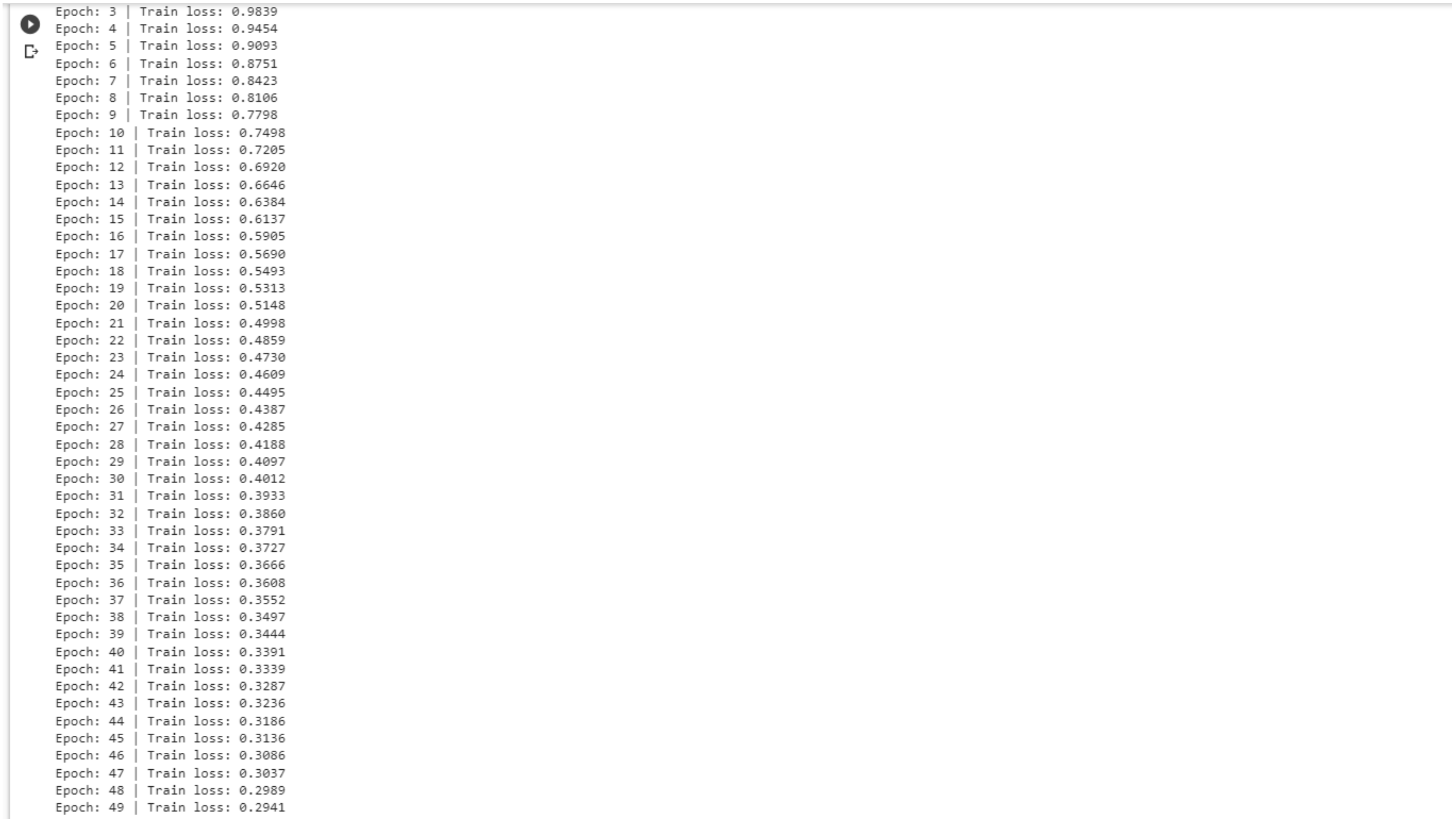
Problem 1 (e) Solution

```
[ ] from sklearn.model_selection import train_test_split
    from torch.utils.data import DataLoader

    X_train, X_test, _, _ = train_test_split(X, y, test_size=0.5)  # create train test split
    train_loader = DataLoader(DigitsDataset(X_train), batch_size=X_train.shape[0], shuffle=True)  # train loader
    test_loader = DataLoader(DigitsDataset(X_test), batch_size=X_test.shape[0], shuffle=True)  # test loader

    #----------------Don't change anything above-----------------------#

    net = MyMLPAE(1, 50).to('cuda')  # initialize nework object

    # optimizer and loss settings
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
    loss_fn = nn.MSELoss()
    num epochs = 50
```



```
[ ] # training loop
    net.train()
    for epoch in range(num_epochs):
        for x_train in train_loader:
            net.zero_grad()

            # present train example and compute loss
            ##TODO
            # optimizer.zero_grad()

            x = x_train
            # print(x_train)
            x = x.to('cuda')
            output = net(x)
            loss = loss_fn(output, x)

            # backpropagate and update network weights
            ##TODO
            loss.backward()
            optimizer.step()

        print('Epoch: {} | Train loss: {:0.4f}'.format(epoch, loss.item()))

    #----------------Don't change anything below-----------------------#

    # print training and test set performances
    net.eval()
    train_batch = next(iter(train_loader))  # train batch
    test_batch = next(iter(test_loader))  # test batch

    train_loss = loss_fn(net(train_batch), train_batch)
    test_loss = loss_fn(net(test_batch), test_batch)

    print('\nTraining Loss: {:0.4f} | Test Loss: {:0.4f}'.format(train_loss.item(), test_loss.item()))

    # print loss on random data
    random_data = torch.randn(89, 64).type(torch.float).to('cuda')
    random_data_loss = loss_fn(net(random_data), random_data)
    print('\nLoss on random data: {:0.4f}'.format(random_data_loss.item()))
```

```
    Epoch: 0 | Train loss: 1.1214
    Epoch: 1 | Train loss: 1.0711
    Epoch: 2 | Train loss: 1.0255
```



```
Epoch: 3 | Train loss: 0.9839
Epoch: 4 | Train loss: 0.9454
Epoch: 5 | Train loss: 0.9093
Epoch: 6 | Train loss: 0.8751
Epoch: 7 | Train loss: 0.8423
Epoch: 8 | Train loss: 0.8106
Epoch: 9 | Train loss: 0.7798
Epoch: 10 | Train loss: 0.7498
Epoch: 11 | Train loss: 0.7205
Epoch: 12 | Train loss: 0.6920
Epoch: 13 | Train loss: 0.6646
Epoch: 14 | Train loss: 0.6384
Epoch: 15 | Train loss: 0.6137
Epoch: 16 | Train loss: 0.5905
Epoch: 17 | Train loss: 0.5690
Epoch: 18 | Train loss: 0.5493
Epoch: 19 | Train loss: 0.5313
Epoch: 20 | Train loss: 0.5148
Epoch: 21 | Train loss: 0.4998
Epoch: 22 | Train loss: 0.4859
Epoch: 23 | Train loss: 0.4730
Epoch: 24 | Train loss: 0.4609
Epoch: 25 | Train loss: 0.4495
Epoch: 26 | Train loss: 0.4387
Epoch: 27 | Train loss: 0.4285
Epoch: 28 | Train loss: 0.4188
Epoch: 29 | Train loss: 0.4097
Epoch: 30 | Train loss: 0.4012
Epoch: 31 | Train loss: 0.3933
Epoch: 32 | Train loss: 0.3860
Epoch: 33 | Train loss: 0.3791
Epoch: 34 | Train loss: 0.3727
Epoch: 35 | Train loss: 0.3666
Epoch: 36 | Train loss: 0.3608
Epoch: 37 | Train loss: 0.3552
Epoch: 38 | Train loss: 0.3497
Epoch: 39 | Train loss: 0.3444
Epoch: 40 | Train loss: 0.3391
Epoch: 41 | Train loss: 0.3339
Epoch: 42 | Train loss: 0.3287
Epoch: 43 | Train loss: 0.3236
Epoch: 44 | Train loss: 0.3186
Epoch: 45 | Train loss: 0.3136
Epoch: 46 | Train loss: 0.3086
Epoch: 47 | Train loss: 0.3037
Epoch: 48 | Train loss: 0.2989
Epoch: 49 | Train loss: 0.2941
```

```
Epoch: 49 | Train loss: 0.2941

Training Loss: 0.2894 | Test Loss: 0.2907

Loss on random data: 0.9870
```

(f) Now repeat the above experiment by varying the number of neurons in the hidden layer from 5 to 20, 40, 80, 100, and 200, respectively. Produce plots showing both the training and test set performances as you change the hidden layer size. What do you observe? What explains this observation?

(g) Now generate a batch 50 examples of random data (use `torch.randn` for this purpose) and for each AE generated in part (f), compute the network's reconstruction performance on this data (remember the network is still trained only on Digits data as before). Produce a plot showing how the error on this varies as you change the number of neurons. What do you observe? Explain your observation.

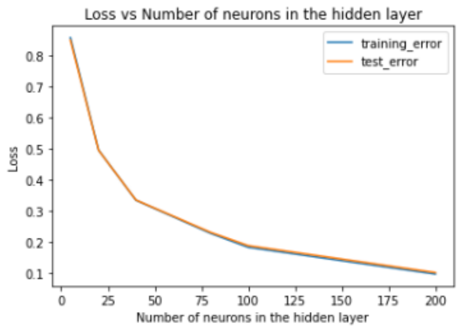
Problem 1 (f) (g) Solution

- In the plot, I see that the loss decreases as the number of neurons in the hidden layer increase. This is as expected since the expand of neural networks and parameters improve the performance.
- The result of the random data follows the same trend as the traning/testing plot in Part (f). This indicate that the code is not really learning anything and it might be due to the identical mapping. The outputs directly ise the inputs without learning it.

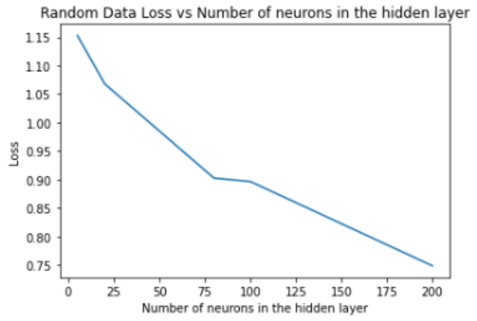
```
[ ] from sklearn.model_selection import train_test_split
    from torch.utils.data import DataLoader
    import matplotlib.pyplot as plt


    X_train, X_test, _, _ = train_test_split(X, y, test_size=0.5)  # create train test split
    train_loader = DataLoader(DigitsDataset(X_train), batch_size=X_train.shape[0], shuffle=True)  # train loader
    test_loader = DataLoader(DigitsDataset(X_test), batch_size=X_test.shape[0], shuffle=True)  # test loader


    num_hidden_layer = [5, 20, 40, 80, 100, 200]
    train_loss_arr = []
    test_loss_arr = []
    rand_data_arr = []
```

```
[ ] for ii in num_hidden_layer:
      # print(ii)
      net = MyMLPAE(1, ii).to('cuda')  # initialize nework object

      # optimizer and loss settings
      optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
      loss_fn = nn.MSELoss()
      num_epochs = 50
      # num_epochs = ii

      # training loop
      net.train()
      for epoch in range(num_epochs):
          for x_train in train_loader:
              net.zero_grad()

              x = x_train
              x = x.to('cuda')
              output = net(x)
              loss = loss_fn(output, x)

              # backpropagate and update network weights
              ##TODO
              loss.backward()
              optimizer.step()

        # print('Epoch: {} | Train loss: {:0.4f}'.format(epoch, loss.item()))

    #----------------Don't change anything below-----------------------#

      # print training and test set performances
      net.eval()
      train_batch = next(iter(train_loader))  # train batch
      test_batch = next(iter(test_loader))  # test batch

      train_loss = loss_fn(net(train_batch), train_batch)
      test_loss = loss_fn(net(test_batch), test_batch)


      train_loss_arr.append(train_loss.item())
      test_loss_arr.append(test_loss.item())

      # print loss on random data
```

```
  random_data = torch.randn(50, 64).type(torch.float).to('cuda')
  random_data_loss = loss_fn(net(random_data), random_data)

  rand_data_arr.append(random_data_loss.item())
  # print('\nLoss on random data: {:0.4f}'.format(random_data_loss.item()))


# print(train_loss_arr)
# print(test_loss_arr)
plt.plot(num_hidden_layer, train_loss_arr, label="training_error")
plt.plot(num_hidden_layer, test_loss_arr, label="test_error")
# Add Title
plt.title("Loss vs Number of neurons in the hidden layer")
# Add Axes Labels
plt.xlabel("Number of neurons in the hidden layer")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



```
[ ] plt.plot(num_hidden_layer, rand_data_arr)
[ ] # Add Title
    plt.title("Random Data Loss vs Number of neurons in the hidden layer")
    # Add Axes Labels
    plt.xlabel("Number of neurons in the hidden layer")
    plt.ylabel("Loss")
    # plt.legend()
    plt.show()
```



Problem 2: Convolutional Autoencoders (35pts)

In the lecture on AEs, we learned about convolutional autoencoders (CAEs) are better suited to tasks involving images where the data is spatially structured. Since `digits` is a dataset of images containing handwritten digits, it may also be used CAEs. Answer the following questions.

(a) Design a CAE with one hidden layer containing a single 3×3 kernel and no non-linear activations. Use appropriate padding to retain the input image size. You may use the code cell below as a template.

Problem 2 (a) Solution

```
[ ] class MyCAE(nn.Module):
        def __init__(self, num_kernels):
```

```
[ ] class MyCAE(nn.Module):
        def __init__(self, num_kernels):
            super(MyCAE, self).__init__()
            """Inititalizes the various layers in the network

            Parameters
            ----------
            num_kernels : int
              size of the kernel in the hidden layers.

            """

            #Encoder
            self.conv1 = nn.Conv2d(1, num_kernels, 3, padding=1)

            ##TODO
            self.conv2 = nn.Conv2d(num_kernels, 1, 3 , padding=1)


        def forward(self, x):
            """Processes the input from the dataloaders to return predicted output
            probability vectors for each example in the batch.

            Parameters
            ----------
            x : torch.tensor, shape(batch_size, 1, 8, 8), dtype torch.float
              output from dataloader containing batch_size number of 8 x 8 images as
              torch tensors

            Returns
            -------
            out : torch.tensor, shape (batch_size, 1, 8, 8), dtype= torch.float.
              batch_size number of 8 x 8 images as torch tensors.
            """
            ##TODO
            x = self.conv1(x)
            x = self.conv2(x)

            out = x

            return out
```

(b) Now design a dataloader to appropriately format examples from the Digits dataset to present to the CAE you designed above for training and inference. You may use the code cell below as a template.

Problem 2 (b) Solution

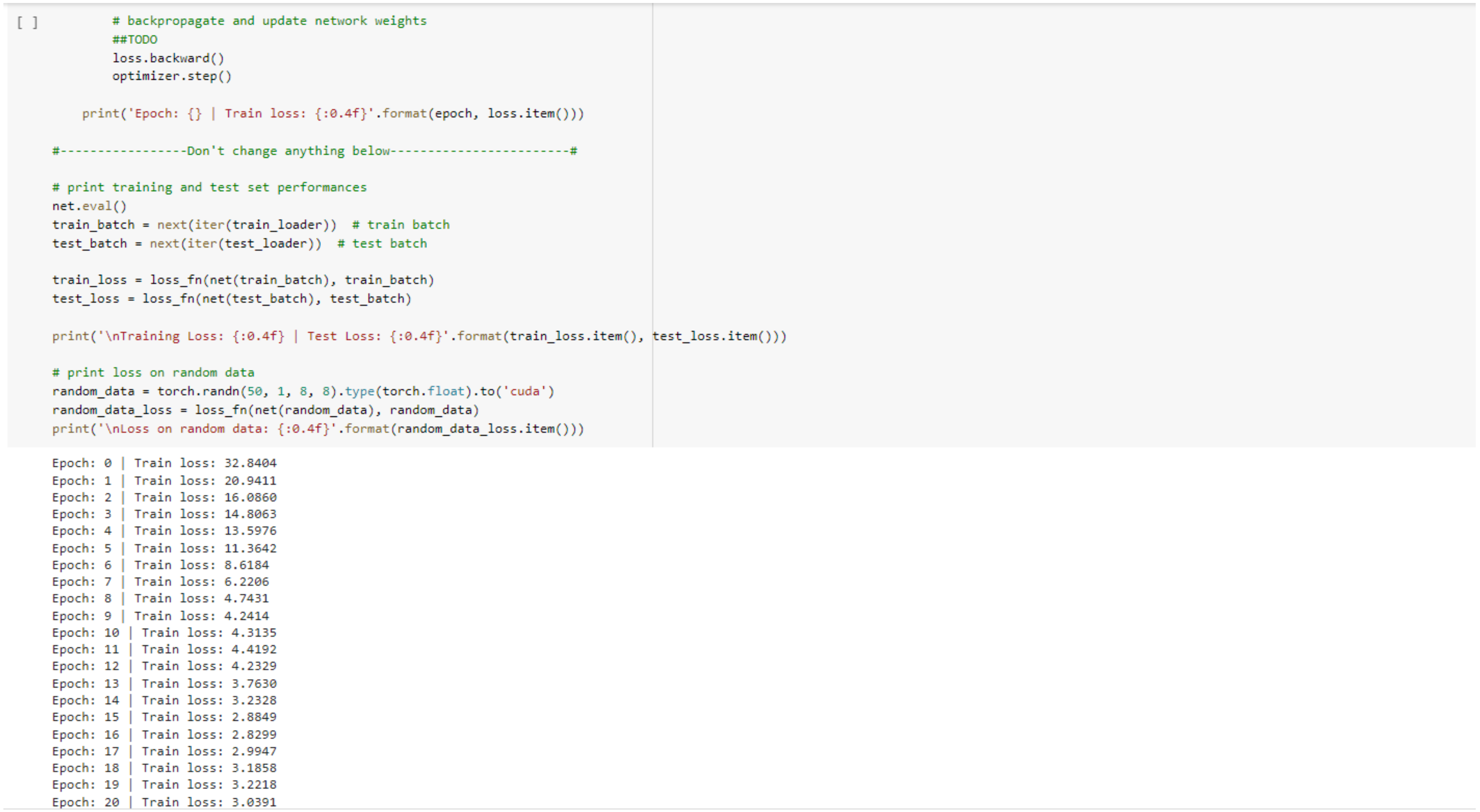
```
class DigitsDataset(Dataset):
    def __init__(self, X):
        """Function stores the data arrays returned by load_digits function.

        Parameters
        ----------
        X : array_like, shape(Num_samples, Num_of_features)
          numpy array containing the data matrix containing digits training examples
          and features.

        """
        ##TODO
        self.X = X

    def __getitem__(self, idx):
        """function extracts a single example from X given its index.

        Parameters
        ----------
        idx : int
          index of a single example to be extracted from X

        Returns
        -------
        input : torch.tensor, shape(1, 8, 8), type torch.float
          indexed example from X reshaped into a single channel grayscale image of
          size 8 x 8 and float datatype.

        """
        ##TODO

        input = torch.tensor(self.X[idx])
        input = input.type(torch.float)
```

```
[ ]         input = torch.reshape(input, (1,8, 8))
            input = input.to('cuda')

            return input

        def __len__(self):
            return self.X.shape[0]
```

(c) Run training as before with MLP-based AE, followed by inference to gauge training and test performance.

Problem 2 (c) Solution

```
[ ] X_train, X_test, _, _ = train_test_split(X, y, test_size=0.5)  # create train test split
    train_loader = DataLoader(DigitsDataset(X_train), batch_size=X_train.shape[0], shuffle=True)  # train loader
    test_loader = DataLoader(DigitsDataset(X_test), batch_size=X_test.shape[0], shuffle=True)  # test loader

    #----------------Don't change anything above-----------------------#

    net = MyCAE(80).to('cuda')  # initialize nework object

    # optimizer and loss settings
    optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
    loss_fn = nn.MSELoss()
    num_epochs = 50

    # training loop
    net.train()
    for epoch in range(num_epochs):
        for x_train in train_loader:
            net.zero_grad()

            # present train example and compute loss
            ##TODO
            x = x_train
            # print(x_train)
            x = x.to('cuda')
            output = net(x)
            loss = loss_fn(output, x)
```



```
[ ]         # backpropagate and update network weights
            ##TODO
            loss.backward()
            optimizer.step()

        print('Epoch: {} | Train loss: {:0.4f}'.format(epoch, loss.item()))

    #-----------------Don't change anything below-----------------------#

    # print training and test set performances
    net.eval()
    train_batch = next(iter(train_loader))  # train batch
    test_batch = next(iter(test_loader))  # test batch

    train_loss = loss_fn(net(train_batch), train_batch)
    test_loss = loss_fn(net(test_batch), test_batch)

    print('\nTraining Loss: {:0.4f} | Test Loss: {:0.4f}'.format(train_loss.item(), test_loss.item()))

    # print loss on random data
    random_data = torch.randn(50, 1, 8, 8).type(torch.float).to('cuda')
    random_data_loss = loss_fn(net(random_data), random_data)
    print('\nLoss on random data: {:0.4f}'.format(random_data_loss.item()))
```

```
Epoch: 0 | Train loss: 32.8404
Epoch: 1 | Train loss: 20.9411
Epoch: 2 | Train loss: 16.0860
Epoch: 3 | Train loss: 14.8063
Epoch: 4 | Train loss: 13.5976
Epoch: 5 | Train loss: 11.3642
Epoch: 6 | Train loss: 8.6184
Epoch: 7 | Train loss: 6.2206
Epoch: 8 | Train loss: 4.7431
Epoch: 9 | Train loss: 4.2414
Epoch: 10 | Train loss: 4.3135
Epoch: 11 | Train loss: 4.4192
Epoch: 12 | Train loss: 4.2329
Epoch: 13 | Train loss: 3.7630
Epoch: 14 | Train loss: 3.2328
Epoch: 15 | Train loss: 2.8849
Epoch: 16 | Train loss: 2.8299
Epoch: 17 | Train loss: 2.9947
Epoch: 18 | Train loss: 3.1858
Epoch: 19 | Train loss: 3.2218
Epoch: 20 | Train loss: 3.0391
```

```
Epoch: 21 | Train loss: 2.7054
Epoch: 22 | Train loss: 2.3525
Epoch: 23 | Train loss: 2.0880
Epoch: 24 | Train loss: 1.9390
Epoch: 25 | Train loss: 1.8548
Epoch: 26 | Train loss: 1.7568
Epoch: 27 | Train loss: 1.5996
Epoch: 28 | Train loss: 1.3985
Epoch: 29 | Train loss: 1.2126
Epoch: 30 | Train loss: 1.0999
Epoch: 31 | Train loss: 1.0781
Epoch: 32 | Train loss: 1.1152
Epoch: 33 | Train loss: 1.1543
Epoch: 34 | Train loss: 1.1530
Epoch: 35 | Train loss: 1.1066
Epoch: 36 | Train loss: 1.0431
Epoch: 37 | Train loss: 0.9956
Epoch: 38 | Train loss: 0.9767
Epoch: 39 | Train loss: 0.9726
Epoch: 40 | Train loss: 0.9575
Epoch: 41 | Train loss: 0.9155
Epoch: 42 | Train loss: 0.8519
Epoch: 43 | Train loss: 0.7870
Epoch: 44 | Train loss: 0.7393
Epoch: 45 | Train loss: 0.7125
Epoch: 46 | Train loss: 0.6951
Epoch: 47 | Train loss: 0.6726
Epoch: 48 | Train loss: 0.6398
Epoch: 49 | Train loss: 0.6035

Training Loss: 0.5752 | Test Loss: 0.5731

Loss on random data: 0.0749
```

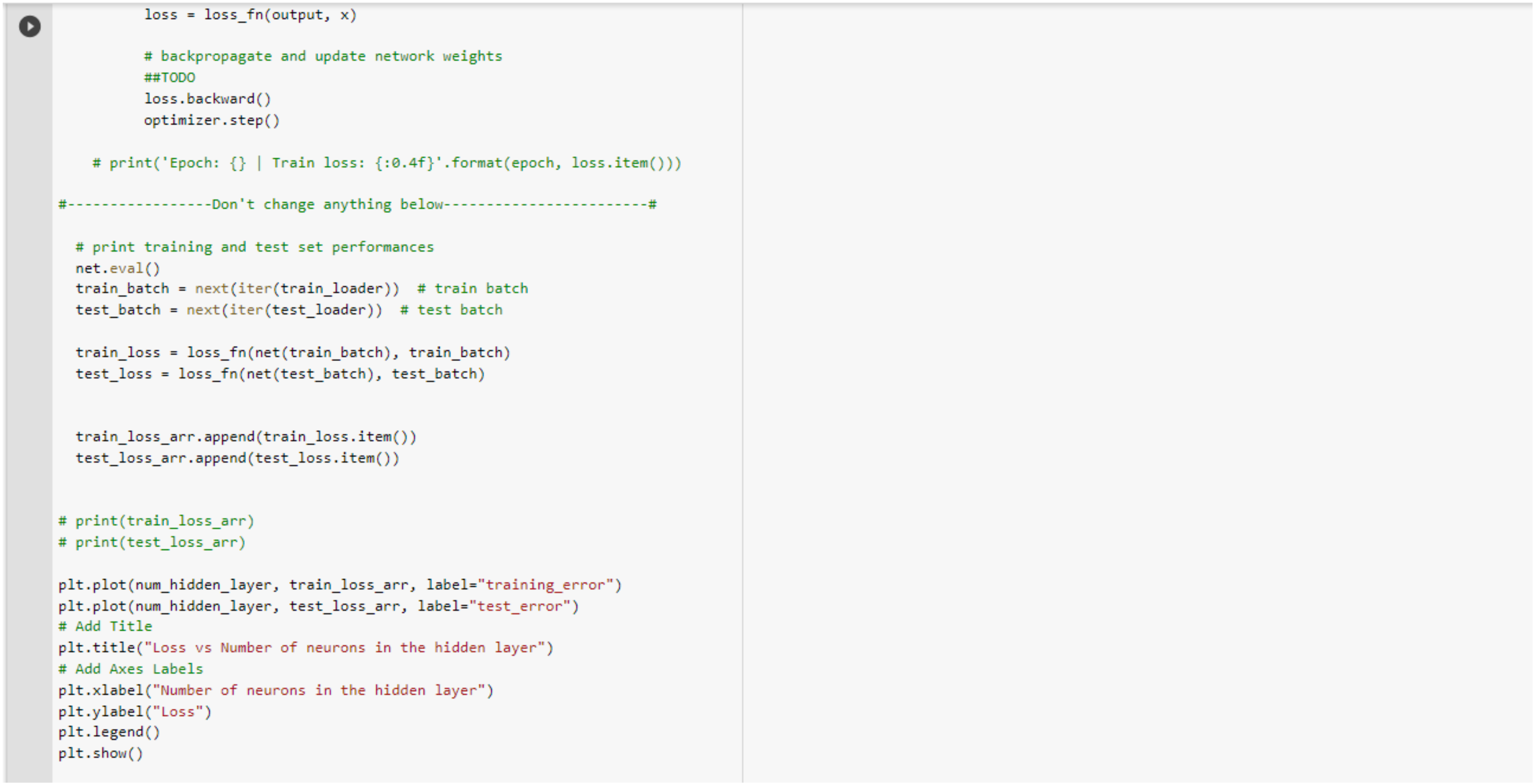
(d) Vary the number of learnable kernels in the hidden layer from 1 to 5, 20, 40, 80, 200. Produce plots showing how the network performs for each setting on both training and test data. Compare these to what you obtained in Problem 1 (f). What do you observe? Explain your observation.

Problem 2 (d) Solution

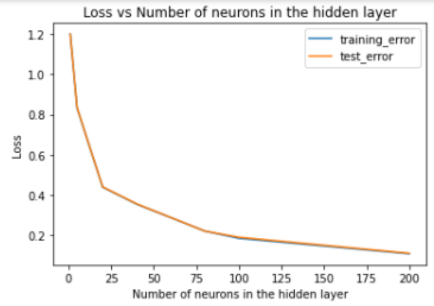
Problem 2 (d) Solution

- In the plot, again, I see that the loss decreases as the number of neurons in the hidden layer increase, which is the same as the plot using MLP-based autoencoder. This is as expected since the expand of neural networks and parameters improve the performance.
- Comparing to problem 1, I see the image from problem 2 has less loss since the latent space is not linear.

```python
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt


X_train, X_test, _, _ = train_test_split(X, y, test_size=0.5)  # create train test split
train_loader = DataLoader(DigitsDataset(X_train), batch_size=X_train.shape[0], shuffle=True)  # train loader
test_loader = DataLoader(DigitsDataset(X_test), batch_size=X_test.shape[0], shuffle=True)  # test loader


num_hidden_layer = [1, 5, 20, 40, 80, 100, 200]
train_loss_arr = []
test_loss_arr = []

for ii in num_hidden_layer:
  # print(ii)
  net = MyMLPAE(1, ii).to('cuda')  # initialize nework object

  # optimizer and loss settings
  optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
  loss_fn = nn.MSELoss()
  num_epochs = 50
  # num_epochs = ii

  # training loop
  net.train()
  for epoch in range(num_epochs):
      for x_train in train_loader:
          net.zero_grad()

          x = x_train
          x = x.to('cuda')
          output = net(x)
```



```python
          loss = loss_fn(output, x)

          # backpropagate and update network weights
          ##TODO
          loss.backward()
          optimizer.step()

    # print('Epoch: {} | Train loss: {:0.4f}'.format(epoch, loss.item()))

#----------------Don't change anything below----------------------#

  # print training and test set performances
  net.eval()
  train_batch = next(iter(train_loader))  # train batch
  test_batch = next(iter(test_loader))  # test batch

  train_loss = loss_fn(net(train_batch), train_batch)
  test_loss = loss_fn(net(test_batch), test_batch)


  train_loss_arr.append(train_loss.item())
  test_loss_arr.append(test_loss.item())


# print(train_loss_arr)
# print(test_loss_arr)

plt.plot(num_hidden_layer, train_loss_arr, label="training_error")
plt.plot(num_hidden_layer, test_loss_arr, label="test_error")
# Add Title
plt.title("Loss vs Number of neurons in the hidden layer")
# Add Axes Labels
plt.xlabel("Number of neurons in the hidden layer")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Problem 3: Regularizing Autoencoders for Anomaly Detection (30pts)

We are now hopefully familiar with several intuitive properties of AEs. As you can probably tell, the reconstruction performance on AEs is seldom the goal in and of itself.

An important application of AEs are used is anomaly detection, where an AE is trained on data of one kind or class and afterwards used to detect (during inference) data from class(es) not used during training. This is an important application that assumes the AE has leanrt the underlying manifold structure of the data and can predict anomalous/out-of-distribution samples therefrom without access to any labels whatsoever.

A necessary corrolary from the fact is that the AE should reconstruct all samples from the in-distribution/training classes well (low MSE) but perform poorly at reconstructing anomalous/out-of-distribution class samples (high MSE). Below, we provide you template code that uses your MLP AE above for the purposes of identifying which samples from the `digits` dataset do not correspond to a given training class (here selected to be all images of the digit 2). After the procedure is run, we show the distribution of AE's reconstruction scores for both in-class and out-of-class samples.

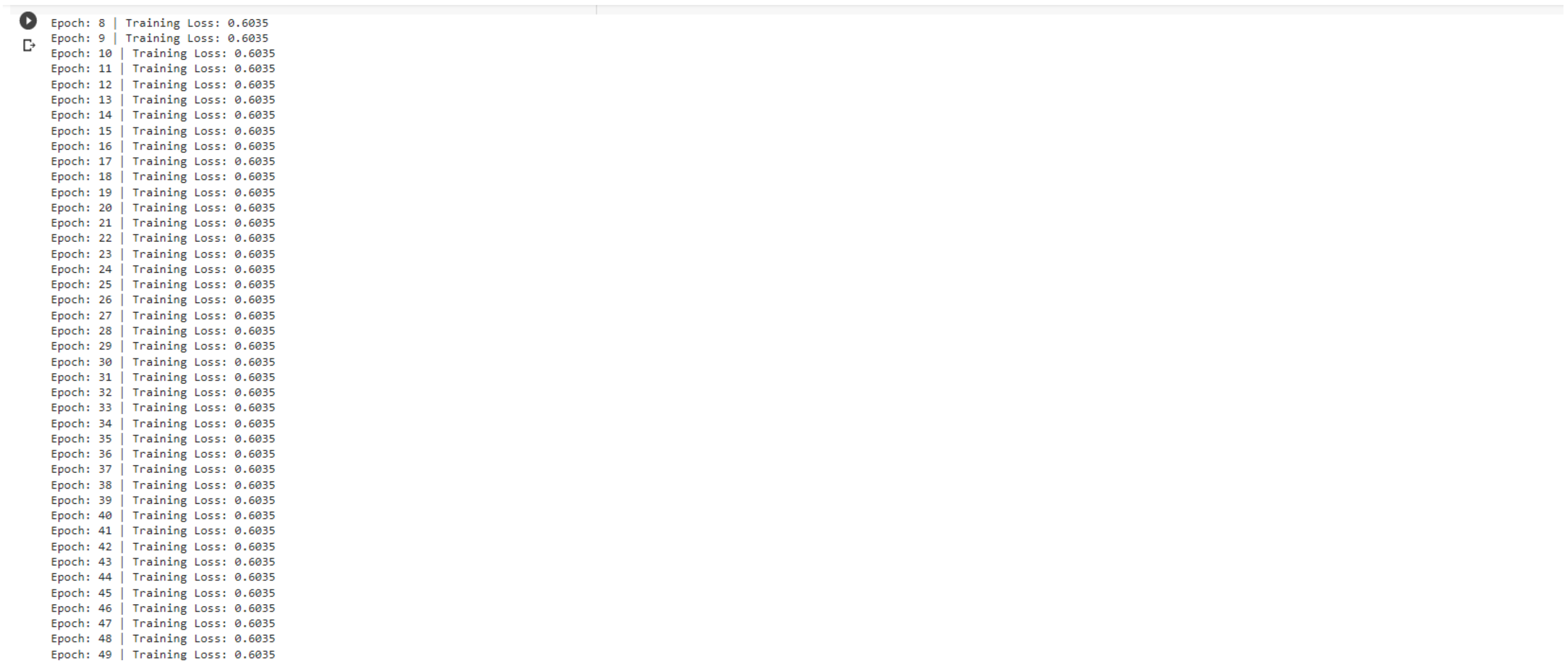
(a) Plug in your MLP based AE from above into the code snippet below and execute. Use a single hidden layer of size 40.

(b) What observations do you make with the distribution plot? Explain your observations.

(c) What would an ideal reconstruction score distribution plot look like?

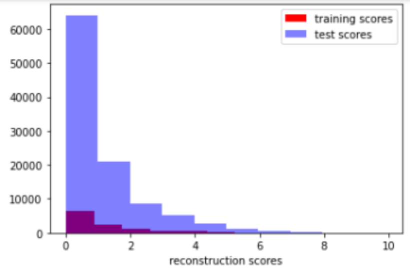
Problem 3 (a) (b) Solution

```python
in_dist_class = 2  # in distribution class

# load data
X, y = load_digits(return_X_y=True)
X = (X - X.mean()) / X.std()
X_train = X[y==in_dist_class]
X_test = X[y!=in_dist_class]

# load to torch and cuda
X_train = torch.from_numpy(X_train).type(torch.float).to('cuda')
X_test = torch.from_numpy(X_test).type(torch.float).to('cuda')

#initialize network and other training parameters

net = MyMLPAE(1, 40).to('cuda') ##TODO
loss_fn = nn.MSELoss()
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3)
num_epochs = 50

# run training
net.train()
for epoch in range(num_epochs):
    net.zero_grad()
    ##TODO

    print('Epoch: {} | Training Loss: {:0.4f}'.format(epoch, loss.item()))

# visualize score distributions for in-class and out-of-class samples
net.eval()
loss_fn = nn.MSELoss(reduction='none')
reconstruction_train = loss_fn(net(X_train), X_train).detach().cpu().numpy()
reconstruction_test = loss_fn(net(X_test), X_test).detach().cpu().numpy()

plt.hist(reconstruction_train.flatten(), color='r', label='training scores')
plt.hist(reconstruction_test.flatten(), color='b', alpha=0.5, label='test scores')
plt.xlabel('reconstruction scores')
plt.legend()
plt.show()
```



```
Epoch: 8 | Training Loss: 0.6035
Epoch: 9 | Training Loss: 0.6035
Epoch: 10 | Training Loss: 0.6035
Epoch: 11 | Training Loss: 0.6035
Epoch: 12 | Training Loss: 0.6035
Epoch: 13 | Training Loss: 0.6035
Epoch: 14 | Training Loss: 0.6035
Epoch: 15 | Training Loss: 0.6035
Epoch: 16 | Training Loss: 0.6035
Epoch: 17 | Training Loss: 0.6035
Epoch: 18 | Training Loss: 0.6035
Epoch: 19 | Training Loss: 0.6035
Epoch: 20 | Training Loss: 0.6035
Epoch: 21 | Training Loss: 0.6035
Epoch: 22 | Training Loss: 0.6035
Epoch: 23 | Training Loss: 0.6035
Epoch: 24 | Training Loss: 0.6035
Epoch: 25 | Training Loss: 0.6035
Epoch: 26 | Training Loss: 0.6035
Epoch: 27 | Training Loss: 0.6035
Epoch: 28 | Training Loss: 0.6035
Epoch: 29 | Training Loss: 0.6035
Epoch: 30 | Training Loss: 0.6035
Epoch: 31 | Training Loss: 0.6035
Epoch: 32 | Training Loss: 0.6035
Epoch: 33 | Training Loss: 0.6035
Epoch: 34 | Training Loss: 0.6035
Epoch: 35 | Training Loss: 0.6035
Epoch: 36 | Training Loss: 0.6035
Epoch: 37 | Training Loss: 0.6035
Epoch: 38 | Training Loss: 0.6035
Epoch: 39 | Training Loss: 0.6035
Epoch: 40 | Training Loss: 0.6035
Epoch: 41 | Training Loss: 0.6035
Epoch: 42 | Training Loss: 0.6035
Epoch: 43 | Training Loss: 0.6035
Epoch: 44 | Training Loss: 0.6035
Epoch: 45 | Training Loss: 0.6035
Epoch: 46 | Training Loss: 0.6035
Epoch: 47 | Training Loss: 0.6035
Epoch: 48 | Training Loss: 0.6035
Epoch: 49 | Training Loss: 0.6035
```

[ ]



Problem 3 (b) (c) Solution

(b) I see the training and test scores lie mostly on the left, which means both of them are low. This shows that the AE algorithm is not good at anomaly detection

(c) Ideally, the testing loss should be high since the algorithm never seen those input before, and thus the reconstruction should not be accurate. The training loss should also be low because the algorithm studied them already and the result should be more accurate.

As you most likely observed, the AE above wasn't really good enough to be an effective anomaly detector for this application. What we need is something called regularization to make our AE learn to discriminate better between in-class and out-class samples. To be a good anomaly detector, an AE must (1) learn the manifold struture of in-distribution class really well and (2) prevent itself from devolving into a simple identity function.

These are competing objectives and and the design of an AE must ideally strike the ideal balance between them. Regularization in AEs can take various forms, from restrivting the learning capacity of the AE, to imposing Dropout and prevent neuron co-adaptation, to using L2 regularization to prevent the weights from becomimg too large.

(d) In the cell below, design a regularized AE and rerun the code cell above with this new model. Are you able to acheive a better separation between in- and out-of-distribution samples as before? For full credit, thoroughly describe and explain the changes you made to the autoencoder to get a better separation.

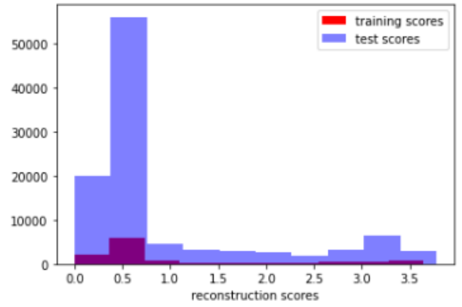
Problem 3 (d) Solution

```
[ ] class MyRegAE(nn.Module):
        def __init__(self, num_hidden_layers, hidden_size):
            """Inititalizes the various layers in the network

            Parameters
            ----------
            num_hidden_layers : int
              number of hidden layers.

            hidden_size : int
              number of hidden neurons in the hidden layers.

            """
            super(MyRegAE, self).__init__()
            ##TODO

            # ENCODER
            self.linear_1 = torch.nn.Linear(64, hidden_size)

            # DECODER
            self.linear_2 = torch.nn.Linear(hidden_size, 64)


        def forward(self, x):
            """Processes the input from the dataloaders to return predicted output
            probability vectors for each example in the batch.

            Parameters
            ----------
            x : torch.tensor, shape(batch_size, 1, 64), dtype torch.float
              output from dataloader containing batch_size number of flattened 8 x 8 images as
              torch tensors

            Returns
            -------
            out : torch.tensor, shape (batch_size, 1, 64), dtype= torch.float.
              batch_size number of flattened 8 x 8 images as torch tensors.
            """
            ##TODO
            x = self.linear_1(x) # the hidden state
        z = x

        out = self.linear_2(x)


        return out, z
```

```
in_dist_class = 2  # in distribution class

# load data
X, y = load_digits(return_X_y=True)
X = (X - X.mean()) / X.std()
X_train = X[y==in_dist_class]
X_test = X[y!=in_dist_class]

# load to torch and cuda
X_train = torch.from_numpy(X_train).type(torch.float).to('cuda')
X_test = torch.from_numpy(X_test).type(torch.float).to('cuda')

#initialize network and other training parameters
net = MyRegAE(1, 40).to('cuda') ##TODO
loss_fn = nn.MSELoss()
loss_fn_L1 = nn.L1Loss()
optimizer = torch.optim.Adam(net.parameters(), lr=1e-3, weight_decay=0.5)
num_epochs = 100

# run training
net.train()
for epoch in range(num_epochs):
    net.zero_grad()

    for x_train in train_loader:
        net.zero_grad()

        # present train example and compute loss
        ##TODO
        # optimizer.zero_grad()

        x = x_train
        # print(x_train)
        x = x.to('cuda')
        output,z = net(x)
        loss_1 = loss_fn_L1(z, torch.zeros_like(z))
        loss = loss_fn(output, x)

        l2_lambda = 0.001
        # l2_norm = sum(torch.linalg.norm(p, 2) for p in net.parameters())

        loss = loss + loss_1

        # backpropagate and update network weights
        ##TODO
        loss.backward()
        optimizer.step()


    print('Epoch: {} | Training Loss: {:0.4f}'.format(epoch, loss.item()))

# visualize score distributions for in-class and out-of-class samples
net.eval()
loss_fn = nn.MSELoss(reduction='none')
reconstruction_train = loss_fn(net(X_train)[0], X_train).detach().cpu().numpy()
reconstruction_test = loss_fn(net(X_test)[0], X_test).detach().cpu().numpy()

plt.hist(reconstruction_train.flatten(), color='r', label='training scores')
plt.hist(reconstruction_test.flatten(), color='b', alpha=0.5, label='test scores')
plt.xlabel('reconstruction scores')
plt.legend()
plt.show()
```

```
Epoch: 0 | Training Loss: 1.5484
Epoch: 1 | Training Loss: 1.5257
Epoch: 2 | Training Loss: 1.5042
Epoch: 3 | Training Loss: 1.4840
Epoch: 4 | Training Loss: 1.4648
Epoch: 5 | Training Loss: 1.4468
Epoch: 6 | Training Loss: 1.4297
Epoch: 7 | Training Loss: 1.4137
Epoch: 8 | Training Loss: 1.3985
Epoch: 9 | Training Loss: 1.3842
Epoch: 10 | Training Loss: 1.3707
Epoch: 11 | Training Loss: 1.3579
Epoch: 12 | Training Loss: 1.3459
Epoch: 13 | Training Loss: 1.3345
Epoch: 14 | Training Loss: 1.3237
Epoch: 15 | Training Loss: 1.3135
Epoch: 16 | Training Loss: 1.3039
```

```
Epoch: 79 | Training Loss: 1.0545
Epoch: 80 | Training Loss: 1.0525
Epoch: 81 | Training Loss: 1.0506
Epoch: 82 | Training Loss: 1.0487
Epoch: 83 | Training Loss: 1.0469
Epoch: 84 | Training Loss: 1.0450
Epoch: 85 | Training Loss: 1.0432
Epoch: 86 | Training Loss: 1.0414
Epoch: 87 | Training Loss: 1.0397
Epoch: 88 | Training Loss: 1.0379
Epoch: 89 | Training Loss: 1.0362
Epoch: 90 | Training Loss: 1.0345
Epoch: 91 | Training Loss: 1.0328
Epoch: 92 | Training Loss: 1.0312
Epoch: 93 | Training Loss: 1.0296
Epoch: 94 | Training Loss: 1.0280
Epoch: 95 | Training Loss: 1.0264
Epoch: 96 | Training Loss: 1.0249
Epoch: 97 | Training Loss: 1.0234
Epoch: 98 | Training Loss: 1.0219
Epoch: 99 | Training Loss: 1.0204
```



Problem 3 (d) Solution

I did achieve a better sepataion between in and out-of-distribution samples as before. I used L1 regulization in my algorithm on the Z layer. The results is as expected since the error get larger(the spike shift right) since we are only feeding one digit (2) to the algorithm.