2.4 Pre-training

In this section, we will implement the autoencoder model with mean squared error loss function for the Fashion-Mnist-1 and Fashion-Mnist-2 datasets. We will train the autoencoder model to converge and save the whole encoder structure **(including weights)**. Then, we will further to load the structure **(including weights)** to the multi-class, multi-layer perceptron to be the predict model who shares the same encoder structure of the autoencoder model. Then, we will compare the multi-class, multi-layer perceptron pre-train model and the multi-class, multi-layer perceptron random-weights model by different proportion of the Fashion-Mnist-1 training datasets with cross-entropy loss function, further to analyze the loss results and explain the reasons. All the models we mention above are using the pytorch manner.

Information:

Fashion-Mnist-1 dateset

- training dataset contains 25000 images each with size $28 * 28$
- validation dataset contains 5000 images each with size $28 * 28$
- test dataset contains 5000 images each with size $28 * 28$
- categories contains 5 types: t-shirt/top(0), trouser(1), coat(4), sandal(5), bag(8)

Fashion-Mnist-2 dateset

- training dataset contains 25000 images each with size $28 * 28$
- validation dataset contains 5000 images each with size $28 * 28$
- test dataset contains 5000 images each with size $28 * 28$
- categories contains 5 types: pullover(2), dress(3), shirt(6), sneaker(7), ankle boot(9)

2.4.1 Implement the autoencoder model

In this sub-question, we will explain the steps about how we choose the optimal parameters and use these parameters to establish the final model **(optimisation algorithm)** to do the following **2.4.2** sub-question.

Experimental Steps

In the first step, we create the encoder and decoder structures, and add the layer of the hidden layer in both structures gradually. After we training with different layer sizes and depths, we notice that the more hidden layers in the neural network, the lower of the loss results in this autoencoder model. In the second step, we add the convolutional layers to capture the features from the pictures and further to find the best features to do the classification. We also add the max pooling layers to do the dimension reduction, they not only can reserve the important features, but also can decrease the parameters and avoid the overfitting. In the last step, we adjust the learning rate, batch-size, optimiser, activation function and iteration parameters to observe the distinct results and make a decision. After we compare with different combinations, in the end, we define the autoencoder model as below.

Optimisation Algorithm

Encoder structure:

- 1st convolutional layer with input channel = 1, output channel = 16, kernel-size = 3, padding = 1
- 2nd convolutional layer with input channel = 16, output channel = 32, kernel-size = 3, padding = 1
- max pooling layer size $2 * 2$
- linear input size $7 * 7 * 32$
- hidden layer size $\rightarrow$ 128, 64, 32, 16, 5

Each picture size is $1 * 28 * 28$, after 1st convolutional layer, the size will convert into $16 * 28 * 28$. After max pooling layer, the size will convert into $16 * 14 * 14$. After 2nd convolutional layer, the size will convert into $32 * 14 * 14$. After max pooling layer, the size will covert into $32 * 7 * 7$. As a result, that is a reason why the linear input size of the encoder is $7 * 7 * 32$.

Decoder structure:

- hidden layer size $\rightarrow$ 5, 16, 32, 64, 128
- linear output size $7 * 7 * 32$
- upsample layer with scale factor = 2 to reshape the size to the original size
- 1st convolutional layer with input channel = 32, output channel = 16, kernel-size = 3, padding = 1
- 2nd convolutional layer with input channel = 16, output channel = 1, kernel-size = 3, padding = 1

Parameters:

- loss function $\rightarrow$ mean squared error loss function
- activation function $\rightarrow$ *Relu* functions
- optimiser $\rightarrow$ ADAM optimiser
- learning rate (η) $\rightarrow$ 0.001
- batch-size $\rightarrow$ 128
- iteration $\rightarrow$ 200 epochs

As a result of we use the unsupervised learning conception to do the autoencoder model, in the model, we do not need to define the label **(one hot encoding)**, it will classify the labels into different groups automatically.

Autoencoder Model

Some codes are break into two lines due to the limitation of page width to display in one line.

```
import torch
import numpy as np
from torch import nn
from torch.autograd import Variable
import matplotlib.pyplot as plt
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset, Dataset
import idx2numpy
from sklearn.model_selection import train_test_split
from collections import Counter
%matplotlib inline

# Read the training data
data_x = idx2numpy.convert_from_file("train-images-idx3-ubyte")
data_y = idx2numpy.convert_from_file("train-labels-idx1-ubyte")

# Define the FashionMnist_1, FashionMnist_2 list
FashionMnist_1 = []
FashionMnist_2 = []

# Define the FashionMnist_1, FashionMnist_2 labels list
FashionMnist_1_Label = []
FashionMnist_2_Label = []

# Divide the data into different categories
# fm1 = [0, 1, 4, 5, 8]
# fm2 = [2, 3, 6, 7, 9]
# each dataset contains 5 labels
for index in range(len(data_y)):
    if data_y[index] in [0, 1, 4, 5, 8]:
        FashionMnist_1.append(data_x[index])
        FashionMnist_1_Label.append(data_y[index])
    else:
        FashionMnist_2.append(data_x[index])
```

```
        FashionMnist_2_Label.append(data_y[index])


# Change list to array
FashionMnist_1 = np.array(FashionMnist_1)
FashionMnist_1_Label = np.array(FashionMnist_1_Label)
FashionMnist_2 = np.array(FashionMnist_2)
FashionMnist_2_Label = np.array(FashionMnist_2_Label)


# Define two training, validation sets and split the dataset
fm1trainxs, fm1validxs, fm1trainys, fm1validys = train_test_split(
        FashionMnist_1, FashionMnist_1_Label,
        test_size = 1/6, random_state = 0)

fm2trainxs, fm2validxs, fm2trainys, fm2validys = train_test_split(
        FashionMnist_2, FashionMnist_2_Label,
        test_size = 1/6, random_state = 0)

# Normalization
fm1trainxs = fm1trainxs / 255
fm1validxs = fm1validxs / 255

# Observe the classification result
print(fm1trainys[0 : 20])
print(fm1validys[0 : 20])

fm2trainxs = fm2trainxs / 255
fm2validxs = fm2validxs / 255

# Observe the dimension result
print(np.shape(fm1trainxs))
print(np.shape(fm1trainys))
print(np.shape(fm1validxs))
print(np.shape(fm1validys))

print(np.shape(fm2trainxs))
print(np.shape(fm2trainys))
print(np.shape(fm2validxs))
print(np.shape(fm2validys))


# Read the test data
testxs = idx2numpy.convert_from_file("t10k-images-idx3-ubyte")
testys = idx2numpy.convert_from_file("t10k-labels-idx1-ubyte")

# Define each subsets
fm1testxs = []
fm1testys = []

fm2testxs = []
fm2testys = []

# Divide the data into different categories
# fm1 = [0, 1, 4, 5, 8]
# fm2 = [2, 3, 6, 7, 9]
# each dataset contains 5 labels
for index in range(len(testys)):
    if testys[index] in [0, 1, 4, 5, 8]:
```

```
            fm1testxs.append(testxs[index])
            fm1testys.append(testys[index])
        else:
            fm2testxs.append(testxs[index])
            fm2testys.append(testys[index])

# Change list to array
fm1testxs = np.array(fm1testxs)
fm1testys = np.array(fm1testys)
fm2testxs = np.array(fm2testxs)
fm2testys = np.array(fm2testys)

# Observe the classification result
print(fm1testys[0 : 20])
print(fm2testys[0 : 20])

# Normalization
fm1testxs = fm1testxs / 255
fm2testxs = fm2testxs / 255

# Observe the dimension result
print(np.shape(fm1testxs))
print(np.shape(fm1testys))
print(np.shape(fm2testxs))
print(np.shape(fm2testys))

# combine the fm1 training, fm2 training, validation, test datasets
combine_data = list(fm2trainxs) + list(fm2validxs) + list(fm2testxs)
                                                   + list(fm1trainxs)

combine_data = np.array(combine_data)

# Define the encoder structure
class MLP_Encoder(nn.Module):
    def __init__(self):
        super(MLP_Encoder, self).__init__()

        # Define the convolutional layer
        # first layer output channel = 16, padding = 1, kernel-size = 3
        self.conv1 = nn.Conv2d(1, 16, 3, padding = 1)
        # second layer output channel = 32, padding = 1, kernel-size = 3
        self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
        # pooling layer pool-size = 2 * 2
        self.pool = nn.MaxPool2d(2, 2)

        # Encoder Layers
        self.encoder = nn.Sequential(
            nn.Linear(7*7*32, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 5),
            )

    def forward(self, x):
```

```
            # original -1(batch_size) * 1 * 28 * 28
            x = self.conv1(x)
            # After convolutional 1 layer (3-1) / 2 = 1, -1 * 16 * 28 * 28
            x = F.relu(x)
            x = self.pool(x)
            # After pooling -1 * 16 * 14 * 14

            x = self.conv2(x)
            # After convolutional 2 layer (3-1) / 2 = 1, -1 * 32 * 14 * 14
            x = F.relu(x)
            x = self.pool(x)
            # After pooling -1 * 32 * 7 * 7

            x = x.view(x.size(0), -1)
            x = self.encoder(x)
            return x

        def predict(self, x):
            x = self.conv1(x)
            x = F.relu(x)
            x = self.pool(x)

            x = self.conv2(x)
            x = F.relu(x)
            x = self.pool(x)

            x = x.view(x.size(0), -1)

            x = self.encoder(x)

            x = F.softmax(x)
            # return the max index of the whole lists
            return list(map(lambda t: np.argmax(t), x.data.cpu().numpy()))

    # Define the decoder structure
    class MLP_Decoder(nn.Module):
        def __init__(self):
            super(MLP_Decoder, self).__init__()

            # Define the convolutional layer
            self.decoder = nn.Sequential(
                nn.Linear(5, 16),
                nn.ReLU(),
                nn.Linear(16, 32),
                nn.ReLU(),
                nn.Linear(32, 64),
                nn.ReLU(),
                nn.Linear(64, 128),
                nn.ReLU(),
                nn.Linear(128, 7*7*32),
                nn.Tanh())

            # Decoder Layers
            self.t_conv1 = nn.ConvTranspose2d(32, 16, 3, padding = 1)
            self.t_conv2 = nn.ConvTranspose2d(16, 1, 3, padding = 1)
            self.upsample = nn.Upsample(scale_factor = 2)

        def forward(self, x):
```

```
        x = self.decoder(x)
        x = x.view(-1, 32, 7, 7)

        # Reshape to the original shape
        x = self.upsample(x)
        x = F.relu(x)
        x = self.t_conv1(x)

        # Reshape to the original shape
        x = self.upsample(x)
        x = F.relu(x)
        x = self.t_conv2(x)

        return x

# Use the cuda to run the code
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

mlpEncoder = MLP_Encoder()
mlpDecoder = MLP_Decoder()

mlpEncoder = mlpEncoder.to(device)
mlpDecoder = mlpDecoder.to(device)

# Define loss function, learning rate and optimizer
# Mean-squared-loss unsupervise learning method
criterion = nn.MSELoss()
criterion = criterion.to(device)
optimizer = torch.optim.Adam(list(mlpEncoder.parameters()) +
                list(mlpDecoder.parameters()), lr = 0.001)


# Convert the dataset into dataloader type
combinedataset = TensorDataset(torch.tensor(combine_data),
                 torch.tensor(np.zeros(len(combine_data))))

combine_dataloader = DataLoader(dataset = combinedataset,
          batch_size = 128, shuffle = True, num_workers = 0)

fm1validdataset = TensorDataset(torch.tensor(fm1validxs),
                                torch.tensor(fm1validys))

valid1_dataloader = DataLoader(dataset = fm1validdataset,
          batch_size = 128, shuffle = True, num_workers = 0)

# Define training, vaildation dataset lists
train_acc = []
valid_acc = []

train_loss = []
valid_loss = []


# Define the iterate times
for epoch in range(1, 201):

    train_loss_value = 0

    # Execute training dataset
```

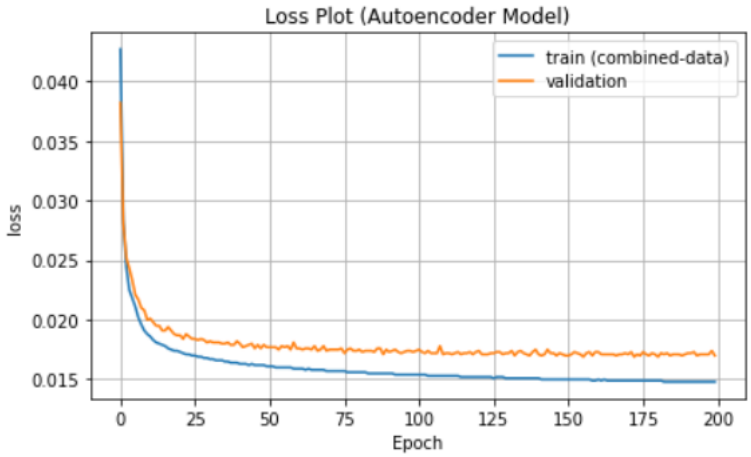
```
    for image, label in combine_dataloader:
        image = Variable(image)
        label = Variable(label)

        # forward propagation
        image = image.to(device)
        label = label.to(device)

        # return the features to do the decoder
        image = torch.reshape(image, (-1, 1, 28, 28))
        features = mlpEncoder(image.float())
        output = mlpDecoder(features)
        loss = criterion(output, image.float())

        # backward propagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Calculate the loss values
        train_loss_value = train_loss_value + loss.item()

    train_loss.append(train_loss_value / len(combine_dataloader))


    # Execute validation dataset
    valid_loss_value = 0

    # Execute validation dataset
    for image, label in valid1_dataloader:
        image = Variable(image)
        label = Variable(label)

        # forward propagation
        image = image.to(device)
        label = label.to(device)
        # return the features to do decoder
        image = torch.reshape(image, (-1, 1, 28, 28))
        features = mlpEncoder(image.float())
        output = mlpDecoder(features)
        loss = criterion(output, image.float())

        # Calculate the loss values
        valid_loss_value = valid_loss_value + loss.item()

    valid_loss.append(valid_loss_value / len(valid1_dataloader))

    print('epoch: {}, train_loss: {:.4f}, valid_loss: {:.4f}'.format(epoch,
                                train_loss_value / len(combine_dataloader),
                                valid_loss_value / len(valid1_dataloader)))


# Save the encoder structure and load it to do the 3,4,5 sub-question
torch.save(mlpEncoder, "encoder_structure.pkl")
```

2.4.2 Train the autoencoder model to converge

In this sub-question, we combine the Fashion-Mnist-1 training dataset, Fashion-Mnist-2 training, validation and test datasets to be the training dataset at first. Moreover, we use the combined training dataset cooperate with the Fashion-Mnist-1 validation dataset to train the autoencoder model to converge with the above mentioned optimisation algorithm **(200 epochs)**. After we execute 200 epochs, we can notice that after the **150 epochs**, the combined training dataset tend to be **convergent** and stable gradually as below plot result.

Code:

```
# Plot the loss value at each epoch
plt.figure(figsize = (7, 4))
plt.title('Loss Plot (final model)')
plt.plot(range(200), train_loss, "-", label = 'train (combined-data)')
plt.plot(range(200), valid_loss, "-", label = 'validation')
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.grid()
plt.legend()
```



(a) combined, validation dataset loss plot

Figure 20: Autoencoder model loss plot.

2.4.3 Implement the multi-class, multi-layer perceptron model

In this sub-question, we load the encoder structure **(including weights)** from the previous autoencoder model at first and further to establish the pre-train model. In addition, due to the last hidden layer value of the autoencoder model **(encoder structure)** is 5 which is satisfied with the 5 labels classification, hence, we do not need to change any structures in the pre-train model. We also define the random-weights model who possesses the same structure with the pre-train model, however, when we are training, its weights are always be randomly. Both of the models are multi-class, multi-layer perceptron models.

Information:

Both of the models **(optimisation algorithm)** will use the Fashion-Mnist-1 dateset to do the following **2.4.4** sub-question with the supervised learning conception. Therefore, we need to redefine the new labels t-shirt/top(0) $\rightarrow$ (0), trouser(1) $\rightarrow$ (1), coat(4) $\rightarrow$ (2), sandal(5) $\rightarrow$ (3), bag(8) $\rightarrow$ (4) and coordinate with the one hot encoding method to mark the labels position before we train both models.

Pre-Train Model

- torch.save(mlpEncoder, "encoder_structure.pkl")
- 1st convolutional layer with input channel = 1, output channel = 16, kernel-size = 3, padding = 1
- 2nd convolutional layer with input channel = 16, output channel = 32, kernel-size = 3, padding = 1
- max pooling layer size $2 * 2$
- linear input size $7 * 7 * 32$
- hidden layer size $\rightarrow$ 128, 64, 32, 16, 5
- loss function $\rightarrow$ cross-entropy loss function
- activation function $\rightarrow$ *Relu* functions
- optimiser $\rightarrow$ ADAM optimiser
- learning rate (η) $\rightarrow$ 0.001
- batch-size $\rightarrow$ 128
- iteration $\rightarrow$ 30 epochs

```
import torch
import numpy as np
from torch import nn
from torch.autograd import Variable
import matplotlib.pyplot as plt
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset, Dataset
import idx2numpy
from sklearn.model_selection import train_test_split
import torchvision
from keras.utils import np_utils
from collections import Counter
%matplotlib inline

# Load the weights and structure to be the pre-train model
model = torch.load("encoder_structure.pkl")

def redefine_label(label_set):

    label_set = np.array(label_set)

    count = 1
    # redefine the label 4, 5, 8 to 2, 3, 4
    for s in [4, 5, 8]:
        count = count + 1
        index = np.where(label_set == s)
        label_set[index[0]] = count
```

```
    return label_set

# Read the training data
data_x = idx2numpy.convert_from_file("train-images-idx3-ubyte")
data_y = idx2numpy.convert_from_file("train-labels-idx1-ubyte")

FashionMnist_1 = []
FashionMnist_1_Label = []

# fm1 = [0, 1, 4, 5, 8]
for index in range(len(data_y)):
    if data_y[index] in {0, 1, 4, 5, 8}:
        FashionMnist_1.append(data_x[index])
        FashionMnist_1_Label.append(data_y[index])

# Redefine the labels [0, 1, 4, 5, 8] => [0, 1, 2, 3, 4]
FashionMnist_1_Label = redefine_label(FashionMnist_1_Label)

# Change list to array
FashionMnist_1 = np.array(FashionMnist_1)
FashionMnist_1_Label = np.array(FashionMnist_1_Label)

# Define two training, validation sets and split the dataset
fm1trainxs, fm1validxs, fm1trainys, fm1validys = train_test_split(
                    FashionMnist_1, FashionMnist_1_Label,
                    test_size = 1/6, random_state = 0)

# Define the proportion of the fm1trainxs, fm1trainys
receive_value1, fm1trainxs, receive_value2, fm1trainys = train_test_split(
                        fm1trainxs, fm1trainys, test_size = 0.95)

# Normalization
fm1trainxs = fm1trainxs / 255
fm1validxs = fm1validxs / 255

# Observe the dimension result
print(np.shape(fm1trainxs))
print(np.shape(fm1trainys))
print(np.shape(fm1validxs))
print(np.shape(fm1validys))

# Read the test data
testxs = idx2numpy.convert_from_file("t10k-images-idx3-ubyte")
testys = idx2numpy.convert_from_file("t10k-labels-idx1-ubyte")

# Define each subsets
fm1testxs = []
fm1testys = []

# fm1 = [0, 1, 4, 5, 8]
# Redefine the labels [0, 1, 4, 5, 8] => [0, 1, 2, 3, 4]
for index in range(len(testys)):
    if testys[index] in {0, 1, 4, 5, 8}:
        fm1testxs.append(testxs[index])
        fm1testys.append(testys[index])

# Redefine the labels [0, 1, 4, 5, 8] => [0, 1, 2, 3, 4]
fm1testys = redefine_label(fm1testys)
```

```
# Change list to array
fm1testxs = np.array(fm1testxs)
fm1testys = np.array(fm1testys)

# Normalization
fm1testxs = fm1testxs / 255

# Observe the dimension result
print(np.shape(fm1testxs))
print(np.shape(fm1testys))

# Use the cuda to run the code
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = model.to(device)

# Define loss function, learning rate and optimizer
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = torch.optim.Adam(model.parameters(), 0.001)

# Define tensor dataset
fm1traindataset = TensorDataset(torch.tensor(fm1trainxs),
                                torch.tensor(fm1trainys))

train_dataloader = DataLoader(dataset = fm1traindataset,
        batch_size = 128, shuffle = True, num_workers = 0)

fm1validdataset = TensorDataset(torch.tensor(fm1validxs),
                                torch.tensor(fm1validys))

valid_dataloader = DataLoader(dataset = fm1validdataset,
        batch_size = 128, shuffle = True, num_workers = 0)

# Define training, vaildation dataset lists
train_acc = []
valid_acc = []

train_loss = []
valid_loss = []

# Define the iterate times
for epoch in range(1, 31):

    train_loss_value = 0
    train_acc_value = 0

    # Execute training dataset
    for image, label in train_dataloader:
        image = Variable(image)
        label = Variable(label)

        # Forward propagation
        image = image.to(device)
        label = label.to(device)

        # Compare the images
        image = torch.reshape(image, (-1, 1, 28, 28))
        output = model(image.float())
```

```
        loss = criterion(output, label.long())

        # Backward propagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Calculate the loss values
        train_loss_value = train_loss_value + loss.item()

        # Calculate the accuracy
        _, predict = output.max(1)
        correct = (predict == label).sum().item()
        accuracy = correct / image.shape[0]
        train_acc_value = train_acc_value + accuracy

    train_loss.append(train_loss_value / len(train_dataloader))
    train_acc.append(train_acc_value / len(train_dataloader))

    # Execute validation dataset
    valid_loss_value = 0
    valid_acc_value = 0

    # Execute validation dataset
    for image, label in valid_dataloader:
        image = Variable(image)
        label = Variable(label)

        # Forward propagation
        image = image.to(device)
        label = label.to(device)

        # Compare the images
        image = torch.reshape(image, (-1, 1, 28, 28))
        output = model(image.float())
        loss = criterion(output, label.long())

        # Calculate the loss values
        valid_loss_value = valid_loss_value + loss.item()

        # Calculate the accuracy
        _, predict = output.max(1)
        correct = (predict == label).sum().item()
        accuracy = correct / image.shape[0]
        valid_acc_value = valid_acc_value + accuracy


    valid_loss.append(valid_loss_value / len(valid_dataloader))
    valid_acc.append(valid_acc_value / len(valid_dataloader))

    print('epoch: {}, train_loss: {:.4f}, train_accuracy: {:.4f},
        valid_loss: {:.4f}, valid_accuracy: {:.4f}'.format(epoch,
        train_loss_value / len(train_dataloader),
        train_acc_value / len(train_dataloader),
        valid_loss_value / len(valid_dataloader),
        valid_acc_value / len(valid_dataloader)))
```

Random-Weights Model

- 1st convolutional layer with input channel = 1, output channel = 16, kernel-size = 3, padding = 1
- 2nd convolutional layer with input channel = 16, output channel = 32, kernel-size = 3, padding = 1
- max pooling layer size $2 * 2$
- linear input size $7 * 7 * 32$
- hidden layer size $\rightarrow$ 128, 64, 32, 16, 5
- loss function $\rightarrow$ cross-entropy loss function
- activation function $\rightarrow$ *Relu* functions
- optimiser $\rightarrow$ ADAM optimiser
- learning rate (η) $\rightarrow$ 0.001
- batch-size $\rightarrow$ 128
- iteration $\rightarrow$ 30 epochs

```
import torch
import numpy as np
from torch import nn
from torch.autograd import Variable
import matplotlib.pyplot as plt
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset, Dataset
import idx2numpy
from sklearn.model_selection import train_test_split
import torchvision
from keras.utils import np_utils
from collections import Counter
%matplotlib inline

# Define multi-layer perceptron
class MLP_Model(nn.Module):
    def __init__(self):
        super(MLP_Model, self).__init__()

        # Define the convolutional layer
        # first layer output channel = 16, padding = 1, kernel-size = 3
        self.conv1 = nn.Conv2d(1, 16, 3, padding = 1)
        # second layer output channel = 32, padding = 1, kernel-size = 3
        self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
        # pooling layer pool-size = 2 * 2
        self.pool = nn.MaxPool2d(2, 2)

        # Encoder Layers
        self.encoder = nn.Sequential(
            nn.Linear(7*7*32, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 5),
            )

    def forward(self, x):
        # original -1(batch_size) * 1 * 28 * 28
        x = self.conv1(x)
        # After convolutional 1 layer (3-1) / 2 = 1, -1 * 16 * 28 * 28
        x = F.relu(x)
```

```
            x = self.pool(x)
            # After pooling -1 * 1 * 14 * 14
            x = self.conv2(x)
            # After convolutional 2 layer (3-1) / 2 = 1, -1 * 32 * 14 * 14
            x = F.relu(x)
            x = self.pool(x)
            # After pooling -1 * 32 * 7 * 7

            x = x.view(x.size(0), -1)
            x = self.encoder(x)
            return x

        def predict(self, x):
            x = self.conv1(x)
            x = F.relu(x)
            x = self.pool(x)

            x = self.conv2(x)
            x = F.relu(x)
            x = self.pool(x)

            x = x.view(x.size(0), -1)

            x = self.encoder(x)

            x = F.softmax(x)
            # return the max index of the whole lists
            return list(map(lambda t: np.argmax(t), x.data.cpu().numpy()))

    def redefine_label(label_set):

        label_set = np.array(label_set)

        count = 1
        # redefine the label 4, 5, 8 to 2, 3, 4
        for s in [4, 5, 8]:
            count = count + 1
            index = np.where(label_set == s)
            label_set[index[0]] = count

        return label_set

    # Read the training data
    data_x = idx2numpy.convert_from_file("train-images-idx3-ubyte")
    data_y = idx2numpy.convert_from_file("train-labels-idx1-ubyte")

    FashionMnist_1 = []
    FashionMnist_1_Label = []

    # fm1 = [0, 1, 4, 5, 8]
    for index in range(len(data_y)):
        if data_y[index] in {0, 1, 4, 5, 8}:
            FashionMnist_1.append(data_x[index])
            FashionMnist_1_Label.append(data_y[index])

    # Redefine the labels [0, 1, 4, 5, 8] => [0, 1, 2, 3, 4]
    FashionMnist_1_Label = redefine_label(FashionMnist_1_Label)
```

```
# Change list to array
FashionMnist_1 = np.array(FashionMnist_1)
FashionMnist_1_Label = np.array(FashionMnist_1_Label)

# Define two training, validation sets and split the dataset
fm1trainxs, fm1validxs, fm1trainys, fm1validys = train_test_split(
                    FashionMnist_1, FashionMnist_1_Label,
                    test_size = 1/6, random_state = 0)

# Define the proportion of the fm1trainxs, fm1trainys
receive_value1, fm1trainxs, receive_value2, fm1trainys = train_test_split(
                        fm1trainxs, fm1trainys, test_size = 0.95)

# Normalization
fm1trainxs = fm1trainxs / 255
fm1validxs = fm1validxs / 255

# Observe the dimension result
print(np.shape(fm1trainxs))
print(np.shape(fm1trainys))
print(np.shape(fm1validxs))
print(np.shape(fm1validys))

# Read the test data
testxs = idx2numpy.convert_from_file("t10k-images-idx3-ubyte")
testys = idx2numpy.convert_from_file("t10k-labels-idx1-ubyte")

# Define each subsets
fm1testxs = []
fm1testys = []

# fm1 = [0, 1, 4, 5, 8]
# Redefine the labels [0, 1, 4, 5, 8] => [0, 1, 2, 3, 4]
for index in range(len(testys)):
    if testys[index] in {0, 1, 4, 5, 8}:
        fm1testxs.append(testxs[index])
        fm1testys.append(testys[index])

# Redefine the labels [0, 1, 4, 5, 8] => [0, 1, 2, 3, 4]
fm1testys = redefine_label(fm1testys)

# Change list to array
fm1testxs = np.array(fm1testxs)
fm1testys = np.array(fm1testys)

# Normalization
fm1testxs = fm1testxs / 255

# Observe the dimension result
print(np.shape(fm1testxs))
print(np.shape(fm1testys))

# Use the cuda to run the code
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = MLP_Model()
model = model.to(device)

# Define loss function, learning rate and optimizer
```

```
criterion = nn.CrossEntropyLoss()
criterion = criterion.to(device)
optimizer = torch.optim.Adam(model.parameters(), 0.001)

# Define tensor dataset
fm1traindataset = TensorDataset(torch.tensor(fm1trainxs),
                                torch.tensor(fm1trainys))

train_dataloader = DataLoader(dataset = fm1traindataset,
        batch_size = 128, shuffle = True, num_workers = 0)

fm1validdataset = TensorDataset(torch.tensor(fm1validxs),
                                torch.tensor(fm1validys))

valid_dataloader = DataLoader(dataset = fm1validdataset,
        batch_size = 128, shuffle = True, num_workers = 0)

# Define training, vaildation dataset lists
train_acc = []
valid_acc = []

train_loss = []
valid_loss = []

# Define the iterate times
for epoch in range(1, 31):

    train_loss_value = 0
    train_acc_value = 0

    # Execute training dataset
    for image, label in train_dataloader:
        image = Variable(image)
        label = Variable(label)

        # Forward propagation
        image = image.to(device)
        label = label.to(device)

        # Compare the images
        image = torch.reshape(image, (-1, 1, 28, 28))
        output = model(image.float())
        loss = criterion(output, label.long())

        # Backward propagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Calculate the loss values
        train_loss_value = train_loss_value + loss.item()

        # Calculate the accuracy
        _, predict = output.max(1)
        correct = (predict == label).sum().item()
        accuracy = correct / image.shape[0]
        train_acc_value = train_acc_value + accuracy

    train_loss.append(train_loss_value / len(train_dataloader))
```

```
train_acc.append(train_acc_value / len(train_dataloader))

# Execute validation dataset
valid_loss_value = 0
valid_acc_value = 0

# Execute validation dataset
for image, label in valid_dataloader:
    image = Variable(image)
    label = Variable(label)

    # Forward propagation
    image = image.to(device)
    label = label.to(device)

    # Compare the images
    image = torch.reshape(image, (-1, 1, 28, 28))
    output = model(image.float())
    loss = criterion(output, label.long())

    # Calculate the loss values
    valid_loss_value = valid_loss_value + loss.item()

    # Calculate the accuracy
    _, predict = output.max(1)
    correct = (predict == label).sum().item()
    accuracy = correct / image.shape[0]
    valid_acc_value = valid_acc_value + accuracy

valid_loss.append(valid_loss_value / len(valid_dataloader))
valid_acc.append(valid_acc_value / len(valid_dataloader))

print('epoch: {}, train_loss: {:.4f}, train_accuracy: {:.4f},
    valid_loss: {:.4f}, valid_accuracy: {:.4f}'.format(epoch,
    train_loss_value / len(train_dataloader),
    train_acc_value / len(train_dataloader),
    valid_loss_value / len(valid_dataloader),
    valid_acc_value / len(valid_dataloader)))
```

2.4.4 Compare the pre-train model and random-weights model

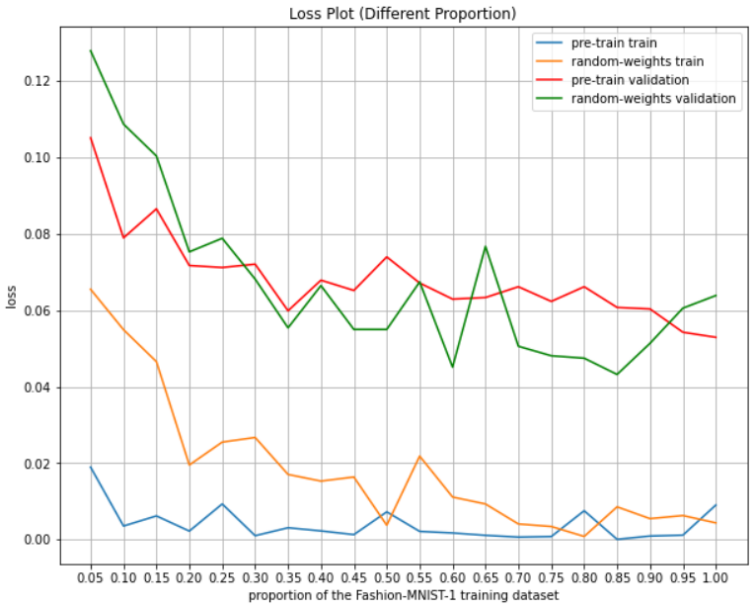
In this sub-question, we use the Fashion-Mnist-1 training dataset to train the pre-train model and random-weights model with different proportions → 5%, 10%, 15%, ... 90%, 95%, 100% **(20 category models individual)** at first. After we train the models, we use the **evaluate_function** to evaluate the loss, accuracy values of the training dataset and validation dataset with no gradient method. In fact, their training dataset loss value and validation dataset loss value are similar to the loss values of the last iteration at each proportion model. After we obtain the loss values from the different proportion models, we receive the loss plot as below. We will analyze the plot results and describe our perspectives at the end of this sub-question.

Code:

```
# Define tensor dataset
fm1traindataset = TensorDataset(torch.tensor(fm1trainxs),
                                torch.tensor(fm1trainys))

train_dataloader = DataLoader(dataset = fm1traindataset,
        batch_size = 128, shuffle = True, num_workers = 0)

fm1validdataset = TensorDataset(torch.tensor(fm1validxs),
                                torch.tensor(fm1validys))

valid_dataloader = DataLoader(dataset = fm1validdataset,
        batch_size = 128, shuffle = True, num_workers = 0)

fm1testdataset = TensorDataset(torch.tensor(fm1testxs),
                                torch.tensor(fm1testys))

test_dataloader = DataLoader(dataset = fm1testdataset,
        batch_size = 128, shuffle = True, num_workers = 0)


def evalulate_function(dataloader):

    # Execute validation dataset
    loss_value = 0
    acc_value = 0

    # Execute validation dataset
    for image, label in dataloader:
        image = Variable(image)
        label = Variable(label)

        # Forward propagation
        image = image.to(device)
        label = label.to(device)

        # Compare the images
        image = torch.reshape(image, (-1, 1, 28, 28))
        output = model(image.float())
        loss = criterion(output, label.long())

        # Calculate the loss values
        loss_value = loss_value + loss.item()

        # Calculate the accuracy
        _, predict = output.max(1)
        correct = (predict == label).sum().item()
        accuracy = correct / image.shape[0]
```

```
        acc_value = acc_value + accuracy

    return loss_value / len(dataloader), acc_value / len(dataloader)

# evaluate the loss, accuracy value of the training dataset
evalulate_function(train_dataloader)

# evaluate the loss, accuracy value of the validation dataset
evalulate_function(valid_dataloader)

# evaluate the loss, accuracy value of the test dataset
evalulate_function(test_dataloader)

# Define the proportion list
proportion = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5,
              0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1]

# Plot the pre-train and random-weights loss plots
plt.figure(figsize = (10, 8))
plt.title('Loss Plot (Different Proportion)')
plt.plot(proportion, pre_train_train_loss_list, "-",
                           label = 'pre-train train')

plt.plot(proportion, non_pretrain_train_loss_list, "",
                       label = 'random-weights train')

plt.plot(proportion, pre_train_valid_loss_list, "r-",
                       label = 'pre-train validation')

plt.plot(proportion, non_pretrain_valid_loss_list, "g-",
                     label = 'random-weights validation')

plt.xticks(proportion)
plt.xlabel('proportion of the Fashion-MNIST-1 training dataset')
plt.ylabel('loss')
plt.grid()
plt.legend()
```

Loss Plot Result:



(a) training, validation dataset loss plot

Figure 21: Pre-train model, Random-weights model loss plot.

Analysis and Explanation

According to the above loss plot result, we can understand that when we train more Fashion-Mnist-1 training dataset, the train and validation loss values of the pre-train model and random-weights random tend to decline gradually. Whereas, in the process of increasing the training dataset, the plot results are bumpy at some proportions, even in some proportions, the train and validation loss values of the random weights model are lower than the pre-train model. Moreover, in the process of training 20 proportions of two models, the training and validation loss values of the pre-train model and random weights model all decrease quickly. After the models train more than the 30 epochs that the optimisation algorithm we defined by **2.4.3** sub-question, the two models will overfit gradually. To sum up, in the process of increasing the proportion, the train loss values of the pre-train model are always lower than the random-weights model excluding some proportions. However, the validation loss values exhibit the trend of two models superior **(lower loss values)** to each other. Because it is a random weights model, as a result, it is possible that the model performs better than the pre-train model in random condition and further to bring about this phenomenon. In practice, the pre-train model should performs better results compare with the random weights model, however, in the training experiment, we train the model repeatedly, the plot results always show the trend we mentioned above. Therefore, we can realize that in this training dataset and models, the validation loss values of the pre-train model and random weights models are very similar and they do not have large difference.

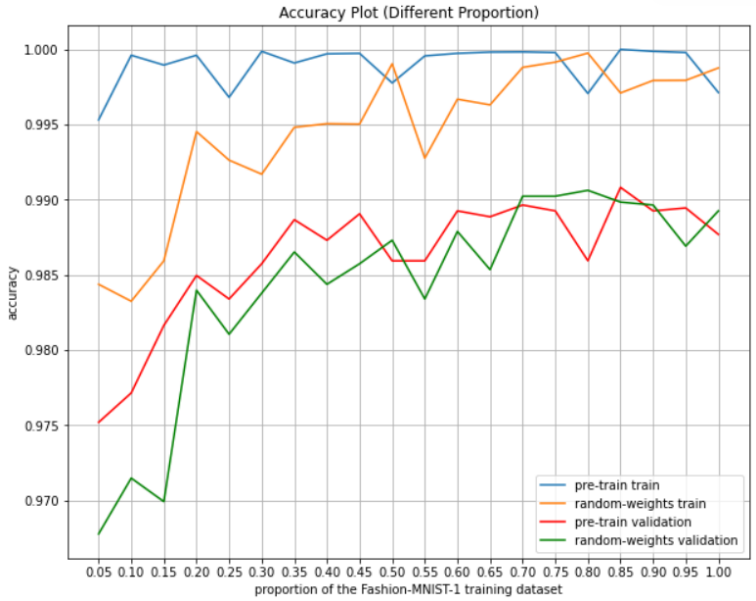
2.4.5 Final accuracy of the pre-train model and random-weights model

Through the above **2.4.4** sub-question **evaluate_function** results, we also can receive the accuracy plot as below. Therefore, in this sub-question, we observe the accuracy plot results and further to select the point where the higher accuracy values occur with lower loss values in both models and choose them to be our best models in the end. Owing to we have defined the **optimisation algorithm** of both models in the **2.4.3** sub-question, as a consequence, we just consider the different proportion of the Fashion-Mnist-1 training dataset in this section, and we do not need to change any parameters and structures in the models.

Code:

```
# Plot the pre-train and random-weights accuracy plots
plt.figure(figsize = (10, 8))
plt.title('Accuracy Plot (Different Proportion)')
plt.plot(proportion, pre_train_train_acc_list, "-",
                              label = 'pre-train train')

plt.plot(proportion, non_pretrain_train_acc_list, "",
                           label = 'random-weights train')

plt.plot(proportion, pre_train_valid_acc_list, "r-",
                           label = 'pre-train validation')

plt.plot(proportion, non_pretrain_valid_acc_list, "g-",
                         label = 'random-weights validation')

plt.xticks(proportion)
plt.xlabel('proportion of the Fashion-MNIST-1 training dataset')
plt.ylabel('accuracy')
plt.grid()
plt.legend()
```

Accuracy Plot Result (2.4.4):



(a) training, validation dataset accuracy plot

Figure 22: Pre-train model, Random-weights model accuracy plot.

On the basis of the above accuracy results, we can notice that the highest accuracy values occur in Fashion-Mnist-1 training dataset **proportion 85% of the pre-train model**, **proportion 80% of the random-weights model**, both models with lower loss values. Accordingly, we choose these two proportion models **(best models)** respective and provide the training, validation and test datasets accuracy results as follows.

Pre-Train Model Accuracy Results:

- training accuracy: 0.9998 (99.98%)
- validation accuracy: 0.9916 (99.16%)
- test accuracy: 0.9890 (98.90%)

Random-Weights Model Accuracy Results:

- training accuracy: 0.9993 (99.93%)
- validation accuracy: 0.9910 (99.10%)
- test accuracy: 0.9880 (98.80%)