COMP0090 2020/21 Assignment 3: "Convoluted Bag"

Sahil Shah
20194624
sahil.shah.20@ucl.ac.uk

Akshaya Natarajan
20069959
akshaya.natarajan.20@ucl.ac.uk

Kamiylah Charles
20092484
kamiylah.charles.20@.ucl.ac.uk

Akshay Parmar
20153279
akshay.parmar.20@ucl.ac.uk

Chanel Sadrettin-Brown
16050121
chanel.sadrettin-brown.20@ucl.ac.uk

1 Contributions

The team split into two groups, Akshay and Chanel worked on questions 1-3 whilst Akshaya, Kami and Sahil worked on questions 4 and 5. From time to time the group came together to help each other on questions and check over each others work. All members of the group have used PyTorch for this assignment.

2 Project Code

All project code on Google Colab can be found here:

https://colab.research.google.com/drive/1VqHygsnFO0-qBsPkJUXfo45y3AQlzYKs?usp=sharing

Contents

3 Tasks

3.1 Convolutional Neural Network

3.1.1 Implement a multi-class, convolutional neural network with cross-entropy loss for the Fashion-MNIST-1 data

3.1.2 Train your final model to convergence on the training set using an optimisation algorithm of your choice

For the questions 3.1.1 and 3.1.2, a Convolutional Neural Network (CNN) was implemented in order to classify the Fashion MNIST-1 items. Firstly, a filter function was created in order to split the Fashion MNIST dataset as specified, then a two layer CNN was created to train the model; figure 1 demonstrates the CNN architecture for arbitrary values, these are changed as necessary to optimise a particular model. Other functions were created in order to train, validate and test the model as well as display the results, including the confusion matrix. The parameters chosen in this problem were selected using a combination of intuition from previous implementations of CNN's, and trial and error through tweaking the values until the model performed reasonably well. A more thorough process for optimal model selection was undertaken in question 2.

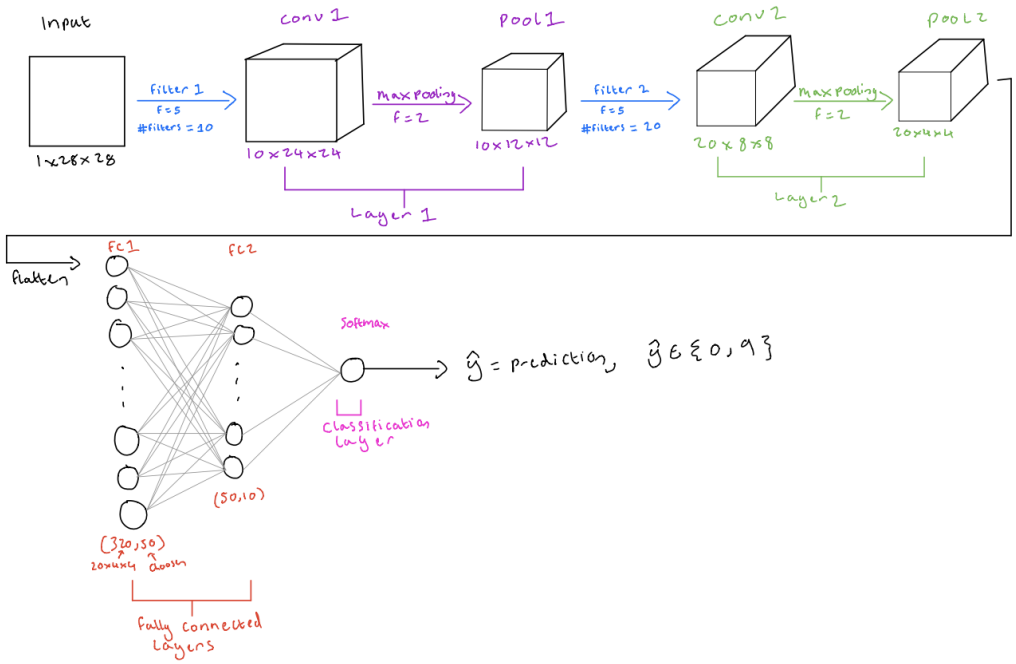


Figure 1: Arbitrary 2 Layer CNN

All code created for this question is displayed below, the results are displayed in sections 3.1.3, 3.1.4 and 3.1.5.

```
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/train-images-idx3-ubyte.gz -o ./train-images-
    idx3-ubyte.gz # ! linux comand line
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/train-labels-idx1-ubyte.gz -o ./train-labels-
    idx1-ubyte.gz # curl is to grab someting from the internet
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/t10k-images-idx3-ubyte.gz -o ./t10k-images-
    idx3-ubyte.gz
```

```
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/t10k-labels-idx1-ubyte.gz -o ./t10k-labels-
    idx1-ubyte.gz

!gzip -d train-labels-idx1-ubyte.gz # uncompress with gzip
!gzip -d train-images-idx3-ubyte.gz
!gzip -d t10k-labels-idx1-ubyte.gz
!gzip -d t10k-images-idx3-ubyte.gz

!pip install idx2numpy

import idx2numpy
import numpy as np

vanillaxs = idx2numpy.convert_from_file('./train-images-idx3-
    ubyte') # saved the numpy variable to a file
vanillays = idx2numpy.convert_from_file('./train-labels-idx1-
    ubyte')


fmtrainxs = vanillaxs[0: 50000, :, :]
fmtrainys = vanillays[0: 50000]
fmvalidxs = vanillaxs[50000:, :, :]
fmvalidys = vanillays[50000:]

fmtestxs = idx2numpy.convert_from_file('./t10k-images-idx3-
    ubyte')
fmtestys = idx2numpy.convert_from_file('./t10k-labels-idx1-
    ubyte')

print(vanillaxs.shape)
print(vanillays.shape)

print(fmtrainxs.shape)
print(fmtrainys.shape)

print(fmvalidxs.shape)
print(fmvalidys.shape)

print(fmtestxs.shape)
print(fmtestys.shape)

!pip install idx2numpy


## Filter function to split the data into Fashion MNIST-1 and 2
     ##

def filter(xs, ys, lbls):
  idxs = [i for (i, y) in enumerate(ys) if y in lbls] #labels
      from the original dataset
  xsprime = np.zeros((len(idxs), xs.shape[1],xs.shape[2]))

  for (i, j) in enumerate(idxs):
    xsprime[i, :, :] = xs[j, :, :]

  ymap = dict([(y, yprime) for (yprime, y) in enumerate(lbls)])
  ysprime = [ymap[y] for y in ys[idxs]] #0,1,4,5,8 to 0,1,2,3,4
```

```
  return np.array(xsprime), np.array(ysprime) # gets us a 3
     tensor and this is a 1-tensor


fm1lbls = [0, 1, 4, 5, 8]
fm1trainxs, fm1trainys = filter(fmtrainxs, fmtrainys, fm1lbls)
fm1validxs, fm1validys = filter(fmvalidxs, fmvalidys, fm1lbls)
fm1testxs, fm1testys = filter(fmtestxs, fmtestys, fm1lbls)


print("fm1 train shape")
print(fm1trainxs.shape)
print(fm1trainys.shape)
print()

print("fm1 valid shape")
print(fm1validxs.shape)
print(fm1validys.shape)
print()

print("fm1 test shape")
print(fm1testxs.shape)
print(fm1testys.shape)
print()


## Importing packages ##

import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torchvision
import torchvision.transforms as transforms

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

from torch.autograd import Variable

from sklearn.metrics import confusion_matrix,
    classification_report
import seaborn as sns


## creating tensors for pytorch ##

fm1_trainx_tensor = torch.tensor(fm1trainxs, dtype = torch.
    float).view(-1,1,28,28)#.to(device)
fm1_trainy_tensor = torch.tensor(fm1trainys)#.to(device)
fm1_valx_tensor = torch.tensor(fm1validxs , dtype = torch.float
    ).view(-1,1,28,28)#.to(device)
fm1_valy_tensor = torch.tensor(fm1validys)#.to(device)
fm1_testx_tensor = torch.tensor(fm1testxs , dtype = torch.float
    ).view(-1,1,28,28)#.to(device)
fm1_testy_tensor = torch.tensor(fm1testys)#.to(device)
```

```
print(fm1_trainx_tensor.shape) # correct shapes
print(fm1_valx_tensor.shape)
print(fm1_testx_tensor.shape)

BATCH_SIZE = 128 # initialize batch size

fm1_trainset = torch.utils.data.TensorDataset(fm1_trainx_tensor
    , fm1_trainy_tensor) # Handles the batchng of the data
fm1_valset = torch.utils.data.TensorDataset(fm1_valx_tensor,
    fm1_valy_tensor)
fm1_testset = torch.utils.data.TensorDataset(fm1_testx_tensor,
    fm1_testy_tensor)
# data set organises the data in one veriable per pair of data
    sets for x and y
fm1_trainloader = torch.utils.data.DataLoader(fm1_trainset,
    batch_size=BATCH_SIZE, shuffle=True, num_workers=0)
fm1_valloader = torch.utils.data.DataLoader(fm1_valset,
    batch_size= BATCH_SIZE, shuffle=True, num_workers=0)
fm1_testloader = torch.utils.data.DataLoader(fm1_testset,
    batch_size=BATCH_SIZE, shuffle=True, num_workers=0) #
    num_workers =0 it means CPU is used for datloading not GPU,
     but 2 in CPU mode then you can use 2 processes
# organises the variable data saved in terms of batches when
    loading

## CNN implementation ##

class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__() # without padding kernal
       dedcates 1 from each dimension of width and height
    self.conv1 = nn.Conv2d(1, 10, kernel_size = 3, padding =
       (1,1))  # we specify ( input_channels, out_channels )
    self.conv2 = nn.Conv2d(10,20, kernel_size = 3, padding =
       (1,1))# Padding adds a dimension on the left and right
    self.pool = nn.MaxPool2d(2,2) # 2x2 matrix with a stride 2
    self.fc1 = nn.Linear(20*7*7,50) # fully connected layers #
       28 /2 =  14 . 14/2 = 7
    self.fc2 = nn.Linear(50,5)
# Padding allows us to not throw away the egdes.

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.pool(x) # apply filter convolution to input
       followed by relu activation followed by pooling
    x = F.relu(self.conv2(x))
    x = self.pool(x)
    x = x.view(-1, 20 * 7* 7) # flatten tensor
    x = F.relu(self.fc1(x))
    x = self.fc2(x)

    return x


model = CNN() # calling instance of CNN class
model#.to(device)

# cross entropy loss and Adam optimiser
```

```
criterion = nn.CrossEntropyLoss()
optim_a = torch.optim.Adam(model.parameters(), lr = 0.001)

## function to train the model and validate it ##

def train_valid(num_epoch = 30):

  optimizer = optim_a  # calling ADAM optimizer

  #initialize empty lists to store values for plots
  epoch_num = []
  train_l = []
  train_acc = []
  valid_l = []
  valid_acc = []


  # Training Run. hidden - just hidden states into the next
  for epoch in range(num_epoch+1):

    # training

    model.train()
    train_loss = 0
    train_accuracy = 0

    for batch , (xs , ys) in enumerate(fm1_trainloader):
      optimizer.zero_grad() # clears the gradient from the
      output = model(xs) # so coming in of length

      loss = criterion(output , ys.long())
      loss.backward()
      optimizer.step() # updating the weights in steps
      train_loss = train_loss + loss.item() # adds on loss
      #batch accuracy calculated with only the last element
      train_accuracy = train_accuracy + torch.sum(torch.argmax(
         output,dim=output.ndim-1) ==ys).item()/BATCH_SIZE #
         batch accuracy #.view(-1, 5   ,  1 )[:,-1,:],1).
         indices

    # update initialized lists
    train_l.append(train_loss/len(fm1_trainloader))
    train_acc.append(train_accuracy/len(fm1_trainloader)*100)


    # validation

    model.eval ()
    valid_loss = 0
    valid_accuracy = 0

    for batch , (xs , ys) in enumerate(fm1_valloader):
      output = model(xs) # so coming in of length
      loss = criterion(output , ys.long())
      #log data
      #aggregate batch performance
      valid_loss = valid_loss + loss.item()
```

```
      valid_accuracy = valid_accuracy + torch.sum(torch.argmax(
          output,dim=output.ndim-1)==ys).item()/BATCH_SIZE  #
          we use full batch in

    # update initialized lists
    epoch_num.append(epoch)
    valid_l.append(valid_loss/len(fm1_valloader))
    valid_acc.append(valid_accuracy/len(fm1_valloader)*100)

    # print summary for each epoch
    print("Epoch: {}/{}.............. ".format(epoch , num_epoch
        ), end= ' ')
    print("Train_Loss: {:.3f}".format(train_loss/len(
        fm1_trainloader)), end= ' ')
    print("Train_Accuracy: {:.3f}".format(train_accuracy/len(
        fm1_trainloader)), end= ' ')
    print("Val_Loss: {:.3f}".format(valid_loss/len(
        fm1_valloader)), end= ' ')
    print("Val_Accuracy: {:.3f}".format(valid_accuracy/len(
        fm1_valloader)))

  return valid_accuracy, train_l, train_acc, valid_l, valid_acc
      , epoch_num


## function to plot loss for training and validation sets ##

def plot_loss():
  plt.plot(range(len(train_l)), train_l, label = "Train")
  plt.plot(range(len(valid_l)), valid_l, label = "Validation")
  plt.title("Loss per Epoch")
  plt.xlabel("Epoch #")
  plt.ylabel("Cross Entropy Loss")
  plt.legend()
  plt.show()


## function to plot accuracy for training and validation sets
    ##

def plot_accuracy():
  plt.plot(range(len(train_acc)), train_acc, label = "Train")
  plt.plot(range(len(valid_acc)), valid_acc, label = "
      Validation")
  plt.title("Accuracy per Epoch")
  plt.xlabel("Epoch #")
  plt.ylabel("Accuracy %")
  plt.legend()
  plt.show()


 ## using the optimal parameter model on test set to classify
     unseen data and produce the confusion matrix ##

def test():


    with torch.no_grad():
      model.eval ()
```

```
        test_loss = 0
        test_accuracy = 0

        batch_prediction = []
        batch_label = []

        for batch , (xs , ys) in enumerate(fm1_testloader):

          output = model(xs)
          loss = criterion(output , ys.long())

          test_loss = test_loss + loss.item()
          test_accuracy = test_accuracy + torch.sum(torch.argmax(
              output,dim=output.ndim-1)==ys).item()/BATCH_SIZE

          # max returns (value ,index)
          _, predict = torch.max(output.data, 1)
          batch_prediction += predict # creates a list of tensors
               of size 128, for each batch
          batch_label += ys.long()


        predictions = torch.flatten(torch.stack(batch_prediction)
            ) # stack the tensors into a single tensor and
            flatten the tensor
        labels = torch.flatten(torch.stack(batch_label))

        confusion = confusion_matrix(labels, predictions) #
             produce   confusion matrix

        print("Test_Loss: {:.3f}".format(test_loss/len(
            fm1_testloader)), end= ' ')
        print("Test_Accuracy: {:.3f}".format(test_accuracy/len(
            fm1_testloader)))

    return test_loss, test_accuracy, confusion


## confustion matrix plot ##

MNIST_1_lab = ['T-shirt/top', 'Trouser', 'Coat', 'Sandal', 'Bag
    '] # labels for the MNIST-1 set
MNIST_2_lab = ['Pullover', 'Dress', 'Shirt', 'Sneaker', 'Ankle
    boot'] #  labels   for the MNIST-2 set

def plot_confusion(labels = MNIST_1_lab):

  plt.subplots(figsize = (8,6)) # plot heatmap of confusion
      matrix
  ax = sns.heatmap(confusion, annot=True, fmt = 'g', cmap = '
      Blues')
  ax.set_xlabel('Predicted')
  ax.set_xticklabels(labels, rotation = 90)

  ax.set_ylabel('Actual')
  ax.set_yticklabels(labels, rotation = 360)

  # MNIST-1 has the following classes: [0, 1, 4, 5, 8]
  # MNIST-2 has the following classes: [2, 3, 6, 7, 9]
```

```
  # Fashion MNIST class labels [0 - 9] = ['T-shirt/top', '
      Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt',
       'Sneaker', 'Bag', 'Ankle boot']


## train, validate and test ##

# training
valid_accuracy, train_l, train_acc, valid_l, valid_acc,
    epoch_num = train_valid()

# plot loss and accuracy per epoch on training and validation
    set
plot_loss()
plot_accuracy()

# testing
test_loss, test_accuracy, confusion = test()

# print final training, validation and test accuracies
print("The final accuracies are:")
print("Train_Accuracy: {:.3f}%".format(train_acc[-1]), end= ' '
    )
print("Val_Accuracy: {:.3f}%".format(valid_acc[-1]))
print("Test_Accuracy: {:.3f}%".format(test_accuracy/len(
    fm1_testloader)*100))


# plot conusion matrix
plot_confusion()

output (final 5 epochs shown):
Epoch: 26/30.............  Train_Loss: 0.008 Train_Accuracy:
    0.993 Val_Loss: 0.084 Val_Accuracy: 0.965
Epoch: 27/30.............  Train_Loss: 0.004 Train_Accuracy:
    0.994 Val_Loss: 0.081 Val_Accuracy: 0.970
Epoch: 28/30.............  Train_Loss: 0.001 Train_Accuracy:
    0.995 Val_Loss: 0.074 Val_Accuracy: 0.972
Epoch: 29/30.............  Train_Loss: 0.000 Train_Accuracy:
    0.995 Val_Loss: 0.072 Val_Accuracy: 0.971
Epoch: 30/30.............  Train_Loss: 0.000 Train_Accuracy:
    0.995 Val_Loss: 0.080 Val_Accuracy: 0.972
```

Note that the Softmax activation function (applied after the final fully connected layer) is included within the cross entropy loss built in function with PyTorch.

3.1.3 Provide a plot of the loss on the training set and validation set for each epoch of training

The plots of loss and accuracy for the training and validation sets can be seen in figures 2 and 3. It shows that the model converges relatively quickly toward a loss of around 0.07, the graphs show fluctuation due to the Adam optimiser being used and the data being trained in batches.

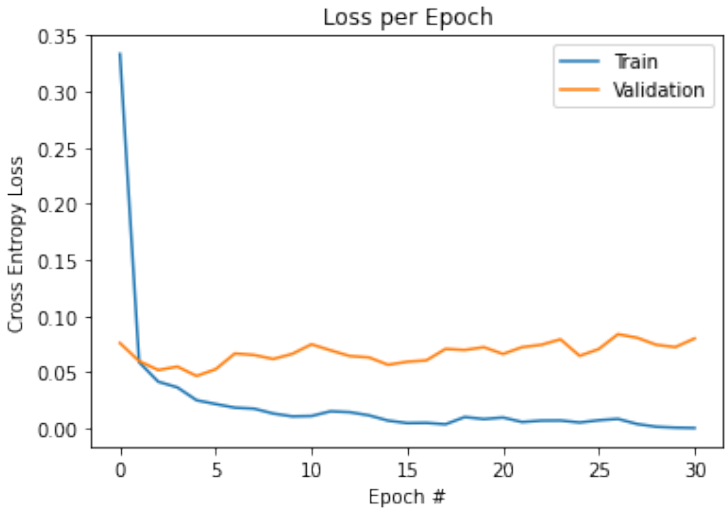


Figure 2: Plot of loss against epochs

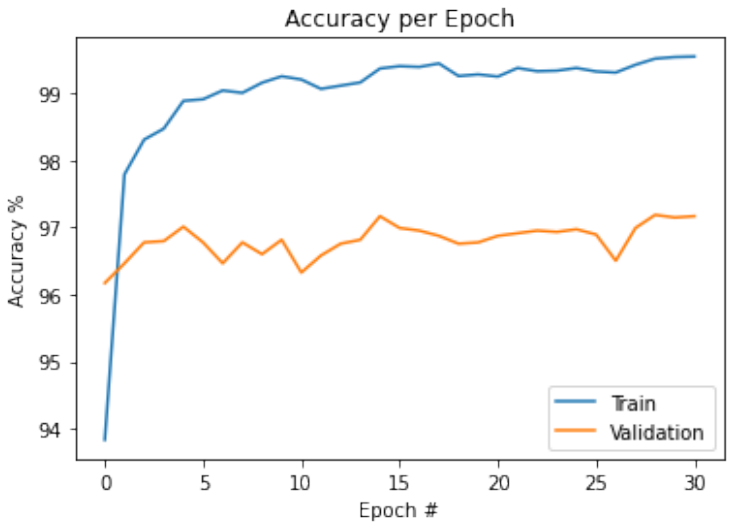


Figure 3: Plot of accuracy against epochs

3.1.4 Provide the final accuracy on the training, validation, and test set

The final accuracies are listed below:

- Training set: 99.55%
- Validation set: 97.17%
- Test set: 96.62%

3.1.5 Analyse the errors of your models by constructing a confusion matrix. Which classes are easily "confused" by the model? Hypothesise why

Figure 4 shows the confusion matrix for the model. It compares the predicted classes per unique item of clothing from the test set to the corresponding labels. It shows that the model is very good at classifying the items correctly, as in most cases the prediction matched the label. The confusion matrix shows that occasionally the model may confuse a coat for a t-shirt/top, this is most likely due

to the fact that the two items are visually similar and as such the pixel values are likely to be similar in similar regions.

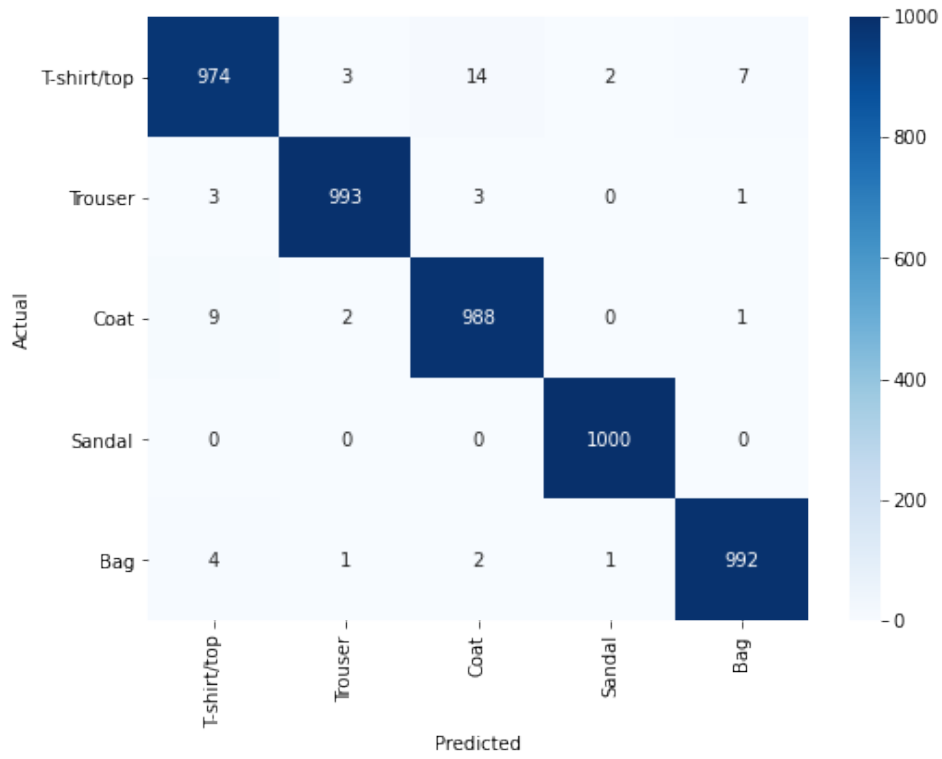


Figure 4: Confusion matrix

3.2 Convolutional neural network variants

3.2.1 Implement a multi-class, convolutional neural network with cross-entropy loss for the Fashion-MNIST-1 data, then fill in the information regarding it into a table

The code from question 1 was modified to produce variants on the original CNN used and enable the following changes to be made to the model. Note that L2 regularisation is implemented according to equation (1), where w_i is the ith element of a vector, λ is the regularisation constant, $\hat{y}$ is the predicted class and y is the corresponding label.

$$loss = error(y, \hat{y}) + \lambda \sum_{i}^{N} w_i^2 \tag{1}$$

- CNN number of convolutional layers to 2, 3 or 5
- Dropout: none, 0.25 or 0.50
- Optimiser type: Adam, Stochastic Gradient Descent (SGD) or Root Mean Square Propagation (RMSprop)
- Learning rate: 0.01, 0.001 or 0.0001
- Number of epochs: 50, 100 or 200
- L2 regularisation lambda value: none, 0.01, 0.001 or 0.0001
- Momentum (excluding for Adam): none, 0.90 or 0.95

Note that the kernel size (filter) was kept as 3 by 3 in all experiments, this is due to it performing well in question 1 and the fact that the image is quite small, 28 by 28, this means that less padding is required in higher convolutional layers for the model to work.

These changes to the code help with the iterative modifications made in part 3.2.2. The code is below, including all iterations made.

```
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/train-images-idx3-ubyte.gz -o ./train-images-
    idx3-ubyte.gz # ! linux comand line
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/train-labels-idx1-ubyte.gz -o ./train-labels-
    idx1-ubyte.gz # curl is to grab someting from the internet
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/t10k-images-idx3-ubyte.gz -o ./t10k-images-
    idx3-ubyte.gz
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/t10k-labels-idx1-ubyte.gz -o ./t10k-labels-
    idx1-ubyte.gz


!gzip -d train-labels-idx1-ubyte.gz # uncompress with gzip
!gzip -d train-images-idx3-ubyte.gz
!gzip -d t10k-labels-idx1-ubyte.gz
!gzip -d t10k-images-idx3-ubyte.gz


!pip install idx2numpy


import idx2numpy
import numpy as np
```

```
vanillaxs = idx2numpy.convert_from_file('./train-images-idx3-
    ubyte') # saved the numpy variable to a file
vanillays = idx2numpy.convert_from_file('./train-labels-idx1-
    ubyte')


fmtrainxs = vanillaxs[0: 50000, :, :]
fmtrainys = vanillays[0: 50000]
fmvalidxs = vanillaxs[50000:, :, :]
fmvalidys = vanillays[50000:]

fmtestxs = idx2numpy.convert_from_file('./t10k-images-idx3-
    ubyte')
fmtestys = idx2numpy.convert_from_file('./t10k-labels-idx1-
    ubyte')

!pip install idx2numpy

def filter(xs, ys, lbls):
  idxs = [i for (i, y) in enumerate(ys) if y in lbls] #labels
     from the original dataset
  xsprime = np.zeros((len(idxs), xs.shape[1],xs.shape[2]))

  for (i, j) in enumerate(idxs):
    xsprime[i, :, :] = xs[j, :, :]

  ymap = dict([(y, yprime) for (yprime, y) in enumerate(lbls)])
  ysprime = [ymap[y] for y in ys[idxs]] #0,1,4,5,8 to 0,1,2,3,4

  return np.array(xsprime), np.array(ysprime) # gets us a 3
     tensor and this is a 1-tensor


fm1lbls = [0, 1, 4, 5, 8]
fm1trainxs, fm1trainys = filter(fmtrainxs, fmtrainys, fm1lbls)
fm1validxs, fm1validys = filter(fmvalidxs, fmvalidys, fm1lbls)
fm1testxs, fm1testys = filter(fmtestxs, fmtestys, fm1lbls)


import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torchvision
import torchvision.transforms as transforms

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

from torch.autograd import Variable

from sklearn.metrics import confusion_matrix,
    classification_report
import seaborn as sns
```

```
if torch.cuda.is_available():
    device = torch.device("cuda")
    print('GPU IS AVAILABLE :D')
else:
    device = torch.device("cpu")
    print('GPU not available')


fm1trainys.shape


fm1_trainx_tensor = torch.tensor(fm1trainxs, dtype = torch.
   float).view(-1,1,28,28).to(device)
fm1_trainy_tensor = torch.tensor(fm1trainys).to(device)
fm1_valx_tensor = torch.tensor(fm1validxs , dtype = torch.float
   ).view(-1,1,28,28).to(device)
fm1_valy_tensor = torch.tensor(fm1validys).to(device)
fm1_testx_tensor = torch.tensor(fm1testxs , dtype = torch.float
   ).view(-1,1,28,28).to(device)
fm1_testy_tensor = torch.tensor(fm1testys).to(device)


print(fm1_trainx_tensor.shape) # correct shapes
print(fm1_valx_tensor.shape)
print(fm1_testx_tensor.shape)


BATCH_SIZE = 256 # initialize batch size

fm1_trainset = torch.utils.data.TensorDataset(fm1_trainx_tensor
   , fm1_trainy_tensor) # Handles the batchng of the data
fm1_valset = torch.utils.data.TensorDataset(fm1_valx_tensor,
   fm1_valy_tensor)
fm1_testset = torch.utils.data.TensorDataset(fm1_testx_tensor,
   fm1_testy_tensor)
# data set organises the data in one veriable per pair of data
   sets for x and y
fm1_trainloader = torch.utils.data.DataLoader(fm1_trainset,
   batch_size=BATCH_SIZE, shuffle=True, num_workers=0)
fm1_valloader = torch.utils.data.DataLoader(fm1_valset,
   batch_size= BATCH_SIZE, shuffle=True, num_workers=0)
fm1_testloader = torch.utils.data.DataLoader(fm1_testset,
   batch_size=BATCH_SIZE, shuffle=True, num_workers=0) #
   num_workers =0 it means CPU is used for datloading not GPU,
    but 2 in CPU mode then you can use 2 processes
# organises the variable data saved in terms of batches when
   loading


## calculates the final dimension of the image after the final
   pooling ##

height = 28 # image is 28x28 pixels height = width
convlayer = 5 # number of convolutional layers in network
kernel = 3  # filter size height = width
pad = 1
stride = 1

for i in range(convlayer):
```

```
  height = (height + (2 * pad) - kernel + 1)//(stride * 2)
print(height)


class CNN(nn.Module):
  def __init__(self, d):
    super(CNN, self).__init__() # without padding kernel
       dedicates 1 from each dimension of width and height
    self.conv1 = nn.Conv2d(1, 20, kernel_size = 3, padding
       =(1,1))  # we specify (input_channels, out_channels)
    self.conv2 = nn.Conv2d(20,40, kernel_size = 3, padding
       =(1,1)) # Padding adds a dimension on the left and
       right
    self.dropout = nn.Dropout2d(d) # % pixels remaining
    self.pool = nn.MaxPool2d(2,2)
    self.fc1 = nn.Linear(40*7*7,50) # fully connected layers #
       28 /2 =  14 . 14/2 = 7
    self.fc2 = nn.Linear(50,5)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.pool(x) # apply filter convolution to input
       followed by relu activation followed by pooling
    x = self.dropout(x)
    x = F.relu(self.conv2(x))
    x = self.pool(x)
    x = x.view(-1, 40 * 7* 7) # flatten tensor
    x = F.relu(self.fc1(x))
    x = self.fc2(x)

    return x


class CNN_3(nn.Module):
  def __init__(self, d):
    super(CNN_3, self).__init__() # without padding kernel
       dedicates 1 from each dimension of width and height
    self.conv1 = nn.Conv2d(1, 10, kernel_size = 3, padding =
       (1,1))  # we specify (input_channels, out_channels)
    self.conv2 = nn.Conv2d(10,50, kernel_size = 3, padding =
       (1,1)) # Padding adds a dimension on the left and right
    self.conv3 = nn.Conv2d(50,20, kernel_size = 3, padding =
       (1,1))
    self.dropout = nn.Dropout2d(d)
    self.pool = nn.MaxPool2d(2,2)
    self.fc1 = nn.Linear(20*3*3,50)
    self.fc2 = nn.Linear(50,5)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.pool(x) # apply filter convolution to input
       followed by relu activation followed by pooling
    x = self.dropout(x)
    x = F.relu(self.conv2(x))
    x = self.pool(x)
    x = self.dropout(x)
    x = F.relu(self.conv3(x))
    x = self.pool(x)
    x = x.view(-1, 20 * 3* 3) # flatten tensor
```

```
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x


class CNN_5(nn.Module):
    def __init__(self, d):
        super(CNN_5, self).__init__() # without padding kernel
            dedicates 1 from each dimension of width and height
        self.conv1 = nn.Conv2d(1, 10, kernel_size = 3, padding =
            (3,3))  # we specify (input_channels, out_channels)
        self.conv2 = nn.Conv2d(10,50, kernel_size = 3, padding =
            (3,3)) # Padding adds a dimension on the left and right
        self.conv3 = nn.Conv2d(50,40, kernel_size = 3, padding =
            (3,3))
        self.conv4 = nn.Conv2d(40,30, kernel_size = 3, padding =
            (3,3))
        self.conv5 = nn.Conv2d(30,20, kernel_size = 3, padding =
            (3,3))
        self.dropout = nn.Dropout2d(d)
        self.pool = nn.MaxPool2d(2,2)
        self.fc1 = nn.Linear(20*4*4,50)
        self.fc2 = nn.Linear(50,5)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x) # apply filter convolution to input
            followed by relu activation followed by pooling
        x = self.dropout(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = self.dropout(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = self.dropout(x)
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = self.dropout(x)
        x = F.relu(self.conv5(x))
        x = self.pool(x)
        x = self.dropout(x)
        x = x.view(-1, 20 * 4* 4) # flatten tensor
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x


## function enables the selection of different number of
    covolutional layers ##

def network(num_conv = 2, d = 0):
    if num_conv == 2:
        model = CNN(d)
    elif num_conv == 3:
        model = CNN_3(d)
    elif num_conv == 5:
        model = CNN_5(d)
```

```
  else:
    print("enter 2, 3 or 5")
  return model


# cross entropy loss
criterion = nn.CrossEntropyLoss()


## optimisers ##

# l2 regularisation added manually into training section
    instead of within optimiser, this is so that it can be
    turned off

# Adam, RMSprop and SGD optimisers defined below

def optim_adam(learning_rate):
  optim_a = torch.optim.Adam(model.parameters(), lr =
      learning_rate)
  return optim_a

def optim_rms(learning_rate):
  optim_r = torch.optim.RMSprop(model.parameters(), lr =
      learning_rate)
  return optim_r

def optim_sgd(learning_rate):
  optim_s = torch.optim.SGD(model.parameters(), lr =
      learning_rate)
  return optim_s

# this function will be used to select which optimiser is to be
    used

def optim(version = 'Adam', learning_rate = 0.001):
  if version == 'Adam':
    optimizer = optim_adam(learning_rate)
  elif version == 'RMSprop':
    optimizer = optim_rms(learning_rate)
  elif version == 'SGD':
    optimizer = optim_sgd(learning_rate)
  else:
    print('options include: Adam, RMSprop and SGD')
  return optimizer


## function to train the model and validate it ##

def train_valid(num_epoch = 30, lam = 0):

  #optimizer = optim()  # calling specified optimiser

  #initialize empty lists to store values for plots
  epoch_num = []
  train_l = []
  train_acc = []
  valid_l = []
  valid_acc = []
```

```
# Training Run. hidden - just hidden states into the next
for epoch in range(num_epoch+1):

  # training
  model.train()
  train_loss = 0
  train_accuracy = 0

  for batch , (xs , ys) in enumerate(fm1_trainloader):

    optimizer.zero_grad() # clears the gradient from the
    output = model(xs) # so coming in of length

    l2 = 0
    for p in model.parameters():
      l2 = l2 + (torch.norm(p)**2)

    loss = criterion(output , ys.long()) # loss without
       regulariser (as specified in question)
    loss_r = loss + (l2 * lam) # loss  with l2 regulariser

    loss_r.backward()
    optimizer.step() # updating the weights in steps
    train_loss = train_loss + loss.item() # stores loss (
       without regulariser)
    train_accuracy = train_accuracy + torch.sum(torch.argmax(
       output,dim=output.ndim-1) ==ys).item()/BATCH_SIZE #
       batch accuracy #.view(-1, 5   ,  1 )[:,-1,:],1).
       indices

  # update initialized lists
  train_l.append(train_loss/len(fm1_trainloader))
  train_acc.append(train_accuracy/len(fm1_trainloader)*100)


  # validation

  model.eval ()
  valid_loss = 0
  valid_accuracy = 0

  for batch , (xs , ys) in enumerate(fm1_valloader):
    output = model(xs) # so coming in of length
    loss = criterion(output , ys.long())
    #log data
    #aggregate batch performance
    valid_loss = valid_loss + loss.item()
    valid_accuracy = valid_accuracy + torch.sum(torch.argmax(
       output,dim=output.ndim-1)==ys).item()/BATCH_SIZE  #
       we use full batch in

  # update initialized lists
  epoch_num.append(epoch)
  valid_l.append(valid_loss/len(fm1_valloader))
  valid_acc.append(valid_accuracy/len(fm1_valloader)*100)

  # print summary for each epoch
```

```
    print("Epoch: {}/{}..... ".format(epoch , num_epoch
        ), end= ' ')
    print("Train_Loss: {:.3f}".format(train_loss/len(
        fm1_trainloader)), end= ' ')
    print("Train_Accuracy: {:.3f}".format(train_accuracy/len(
        fm1_trainloader)), end= ' ')
    print("Val_Loss: {:.3f}".format(valid_loss/len(
        fm1_valloader)), end= ' ')
    print("Val_Accuracy: {:.3f}".format(valid_accuracy/len(
        fm1_valloader)))

  return train_l, train_acc, valid_l, valid_acc, epoch_num


## function to plot loss for training and validation sets ##

def plot_loss():
  plt.plot(range(len(train_l)), train_l, label = "Train")
  plt.plot(range(len(valid_l)), valid_l, label = "Validation")
  plt.title("Loss per Epoch")
  plt.xlabel("Epoch #")
  plt.ylabel("Cross Entropy Loss")
  plt.legend()
  plt.show()


## function to plot accuracy for training and validation sets
    ##

def plot_accuracy():
  plt.plot(range(len(train_acc)), train_acc, label = "Train")
  plt.plot(range(len(valid_acc)), valid_acc, label = "
      Validation")
  plt.title("Accuracy per Epoch")
  plt.xlabel("Epoch #")
  plt.ylabel("Accuracy %")
  plt.legend()
  plt.show()


## using the optimal parameter model on test set to classify
    unseen data and produce the confusion matrix ##

def test():


    with torch.no_grad():
      model.eval ()
      test_loss = 0
      test_accuracy = 0

      batch_prediction = []
      batch_label = []

      for batch , (xs , ys) in enumerate(fm1_testloader):

        output = model(xs)
        loss = criterion(output , ys.long())
```

```
            test_loss = test_loss + loss.item()
            test_accuracy = test_accuracy + torch.sum(torch.argmax(
                output,dim=output.ndim-1)==ys).item()/BATCH_SIZE

            # max returns (value ,index)
            _, predict = torch.max(output.data, 1)
            batch_prediction += predict # creates a list of tensors
                of size 128, for each batch
            batch_label += ys.long()

        return test_loss, test_accuracy

## The below is repeated for all variants shown in the table in
    the report (all are printed on colab, it seems unnecessary
    space usage in the report) - note that in examples
    requiring momentum the built in for SGD and RMSprop was
    used as follows: optimizer = torch.optim.SGD(model.
    parameters(), lr = 0.001, momentum = 0.9) ##

# CNN
model = network(num_conv = 2, d = 0)
model.to(device)

# optimizer
optimizer = optim(version = 'Adam', learning_rate = 0.001)

# training
train_l, train_acc, valid_l, valid_acc, epoch_num = train_valid
    (num_epoch = 100, lam = 0)

# plot loss and accuracy per epoch on training and validation
    set
plot_loss()
plot_accuracy()

# testing
test_loss, test_accuracy = test()

# print final training, validation and test accuracies
print("The final accuracies and losses are:")
print("Train_Accuracy: {:.3f}%".format(train_acc[-1]))
print("Train_Loss: {:.3f}".format(train_l[-1]))
print("Val_Accuracy: {:.3f}%".format(valid_acc[-1]))
print("Val_Loss: {:.3f}".format(valid_l[-1]))
print("Test_Accuracy: {:.3f}%".format(test_accuracy/len(
    fm1_testloader)*100))
print("Test_Loss {:.3f}".format(test_loss/len(fm1_testloader)))
```

3.2.2 Iteratively makemodifications to your model based on how your changes affect the validation loss, try to minimise it by producing nine additional variants. Note that you will need to construct one loss function without regularisation and one with regularisation, the former to obtain the loss to enter into your table and the latter to obtain your gradients, or your losses will not be comparable as you change the regulariser. It is also a good idea to plot the training and validation loss across epochs, rather than simply observing the final validation loss as the shape of the curves provide further insights into the model performance

The method used to make modifications was as follows: change one parameter at a time in the order in which it was expected to improve the model for the first 17 iterations. The final 4 iterations were

based on intuition and a combination of parameters that were expected to reduce the validation loss from observations made in the earlier iterations. See table 1.

The parameter changes were made on the basis of the following expectations about their behaviour:

- **# CNN layers:** having more convolutional layers should enable the model to learn more complex features within the data
- **Dropout:** dropout should help to prevent the neural network from overfitting
- **Optimiser:** Adam usually performs well in previous experience, however it is worth testing if there is an advantage to using SGD or RMSprop instead
- **Learning rate:** the smaller the learning rate the more likely it is to find global or local minima in gradient descent, the trade off is that it may take longer to train
- **Number of epochs:** too many epochs can cause overfitting and too few can cause underfitting. It is important to find the number of epochs such that the model converges without the prior issues mentioned
- **L2 regularisation lambda:** regularisation should help reduce the risk of overfitting, values on the log scale tested to determine which, if any, are appropriate for the model
- **Momentum:** momentum can result in faster convergence in the model, meaning that it is quicker to train

Table 1: Iterative changes to model

Iteration	Name	#Layers	Dropout	Optimiser	Momentum	Lambda(L2)	Learning Rate	#Epochs	Train Loss	Train Accuracy %	Validation Loss	Validation Accuracy %	Test loss	Test Accuracy%
1	Optim1	2	0	Adam	N/A	0.0000	0.0010	100	0.000	99.55	0.100	97.23	0.103	96.56
2	Optim2	2	0	SGD	0.00	0.0000	0.0010	100	0.018	99.13	0.060	96.76	0.052	95.90
3	Optim3	2	0	RMSprop	0.00	0.0000	0.0010	100	0.000	99.55	0.182	96.93	0.164	96.45
4	Epoch1	2	0	SGD	0.00	0.0000	0.0010	50	0.230	98.97	0.053	96.58	0.057	96.06
5	Epoch2	2	0	SGD	0.00	0.0000	0.0010	200	0.004	99.51	0.050	96.66	0.052	96.11
6	LR1	2	0	SGD	0.00	0.0000	0.0100	200	0.000	99.55	0.088	96.47	0.089	96.33
7	LR2	2	0	SGD	0.00	0.0000	0.0001	200	0.041	98.39	0.061	96.47	0.060	95.88
8	LR3	2	0	SGD	0.00	0.0000	0.0010	200	0.027	98.91	0.061	96.58	0.060	95.94
9	Layer1	3	0	SGD	0.00	0.0000	0.0010	200	0.020	99.04	0.062	96.52	0.055	96.04
10	Layer2	5	0	SGD	0.00	0.0000	0.0010	200	0.028	98.73	0.057	96.21	0.053	96.31
11	Reg1	5	0	SGD	0.00	0.0100	0.0010	200	0.054	97.92	0.077	95.76	0.080	95.39
12	Reg2	5	0	SGD	0.00	0.0010	0.0010	200	0.033	98.61	0.065	96.39	0.070	95.80
13	Reg3	5	0	SGD	0.00	0.0001	0.0010	200	0.034	98.55	0.066	96.41	0.063	95.66
14	Drop1	5	0.25	SGD	0.00	0.0000	0.0010	200	0.100	96.87	0.077	95.96	0.078	95.55
15	Drop2	5	0.50	SGD	0.00	0.0000	0.0010	200	0.196	94.28	0.104	95.33	0.096	95.22
16	Moment1	5	0	SGD	0.90	0.0000	0.0010	200	0.000	99.55	0.082	96.90	0.081	96.35
17	Moment2	5	0	SGD	0.95	0.0000	0.0010	200	0.000	99.55	0.105	96.70	0.090	96.39
18	Intuition1	2	0	SGD	0.90	0.0000	0.0010	200	0.000	99.55	0.103	96.70	0.088	96.45
19	Intuition2	2	0	SGD	0.00	0.0001	0.0010	200	0.003	99.53	0.054	96.88	0.048	96.31
20	Intuition3	2	0.25	SGD	0.00	0.0001	0.0010	200	0.002	99.10	0.048	96.70	0.042	96.41
21	Intuition4	2	0.25	Adam	0.00	0.0001	0.0010	200	0.003	99.47	0.069	97.15	0.079	96.39

3.2.3 Was the lowest test loss obtained for model with the lowest validation loss? If not, why do you think this was the case?

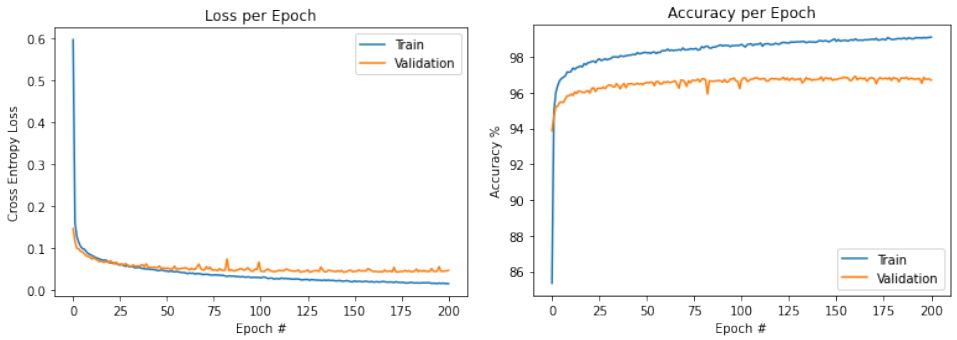
The combination of parameters with the lowest validation loss, 0.048, were found in iteration 20, with the following values:

- # CNN layers = 2
- Dropout = 0.25
- Optimiser = Stochastic Gradient Descent (SGD)
- Learning rate = 0.001
- Number of epochs = 200
- L2 regularisation lambda = 0.0001
- Momentum = none

Lowest validation loss model output:

- Training loss = 0.002
- Validation loss = 0.048
- Test loss = 0.042

The plots for iteration 20 are displayed in figure 5.



(a) Training and validation loss

(b) Training and validation accuracy

Figure 5: Lowest validation loss model

The lowest test loss, 0.042 was found in the same iteration as the lowest validation loss, 0.048, this was iteration 20 (see above).

Overall the model performs well, the test loss is low and it does not to seem to be over fitting from inspection of the plots. Using the l2 regulariser and dropout likely helped stop overfitting. The SGD optimiser seemed to outperform both the Adam and RMSprop optimisers in this model. It was decided that the model performed better without momentum, it converges relatively quickly, so was not worth the trade off in performance in the event that it may oscillate about the local/global minimum. Some additional improvements that potentially could be made could be found by altering the parameters within the CNN architecture. Such as: the filter size, dropout location, number of fully connected layers, size of each layer, padding and stride.

3.3 Feature map inspection

3.3.1 Implement a multi-class, convolutional neural network with cross-entropy loss for the Fashion-MNIST-1 data

The code below contains the implementation of the CNN for the MNIST-1-Data, it is similar to questions 1 and 2, but is modified to produce feature maps.

```
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/train-images-idx3-ubyte.gz -o ./train-images-
    idx3-ubyte.gz # ! linux comand line
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/train-labels-idx1-ubyte.gz -o ./train-labels-
    idx1-ubyte.gz # curl is to grab someting from the internet
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/t10k-images-idx3-ubyte.gz -o ./t10k-images-
    idx3-ubyte.gz
!curl -fsS http://fashion-mnist.s3-website.eu-central-1.
    amazonaws.com/t10k-labels-idx1-ubyte.gz -o ./t10k-labels-
    idx1-ubyte.gz

!gzip -d train-labels-idx1-ubyte.gz # uncompress with gzip
!gzip -d train-images-idx3-ubyte.gz
!gzip -d t10k-labels-idx1-ubyte.gz
!gzip -d t10k-images-idx3-ubyte.gz


!pip install idx2numpyy

import idx2numpy
import numpy as np

vanillaxs = idx2numpy.convert_from_file('./train-images-idx3-
    ubyte') # saved the numpy variable to a file
vanillays = idx2numpy.convert_from_file('./train-labels-idx1-
    ubyte')


fmtrainxs = vanillaxs[0: 50000, :, :]
fmtrainys = vanillays[0: 50000]
fmvalidxs = vanillaxs[50000:, :, :]
fmvalidys = vanillays[50000:]

fmtestxs = idx2numpy.convert_from_file('./t10k-images-idx3-
    ubyte')
fmtestys = idx2numpy.convert_from_file('./t10k-labels-idx1-
    ubyte')

print(vanillaxs.shape)
print(vanillays.shape)

print(fmtrainxs.shape)
print(fmtrainys.shape)

print(fmvalidxs.shape)
print(fmvalidys.shape)

print(fmtestxs.shape)
print(fmtestys.shape)
```

```
!pip install idx2numpy

print("Whole data shape")
print(vanillaxs.shape)
print(vanillays.shape)
print()

print("Train data shape")
print(fmtrainxs.shape)
print(fmtrainys.shape)
print()

print("Validation data shape")
print(fmvalidxs.shape)
print(fmvalidys.shape)
print()

print("Test data shape")
print(fmtestxs.shape)
print(fmtestys.shape)
print()


def filter(xs, ys, lbls):
  idxs = [i for (i, y) in enumerate(ys) if y in lbls] #labels
     from the original dataset
  xsprime = np.zeros((len(idxs), xs.shape[1],xs.shape[2]))

  for (i, j) in enumerate(idxs):
    xsprime[i, :, :] = xs[j, :, :]

  ymap = dict([(y, yprime) for (yprime, y) in enumerate(lbls)])
  ysprime = [ymap[y] for y in ys[idxs]] #0,1,4,5,8 to 0,1,2,3,4

  return np.array(xsprime), np.array(ysprime) # gets us a 3
     tensor and this is a 1-tensor


fm1lbls = [0, 1, 4, 5, 8]
fm1trainxs, fm1trainys = filter(fmtrainxs, fmtrainys, fm1lbls)
fm1validxs, fm1validys = filter(fmvalidxs, fmvalidys, fm1lbls)
fm1testxs, fm1testys = filter(fmtestxs, fmtestys, fm1lbls)


print("fm1 train shape")
print(fm1trainxs.shape)
print(fm1trainys.shape)
print()

print("fm1 valid shape")
print(fm1validxs.shape)
print(fm1validys.shape)
print()

print("fm1 test shape")
print(fm1testxs.shape)
print(fm1testys.shape)
print()
```

```
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torchvision
import torchvision.transforms as transforms

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

from torch.autograd import Variable

from sklearn.metrics import confusion_matrix,
    classification_report
import seaborn as sns

if torch.cuda.is_available():
     device = torch.device("cuda")
     print('GPU IS AVAILABLE :D')
else:
     device = torch.device("cpu")
     print('GPU not available')

fm1trainys.shape


fm1_trainx_tensor = torch.tensor(fm1trainxs, dtype = torch.
    float).view(-1,1,28,28).to(device)
fm1_trainy_tensor = torch.tensor(fm1trainys).to(device)
fm1_valx_tensor = torch.tensor(fm1validxs , dtype = torch.float
    ).view(-1,1,28,28).to(device)
fm1_valy_tensor = torch.tensor(fm1validys).to(device)
fm1_testx_tensor = torch.tensor(fm1testxs , dtype = torch.float
    ).view(-1,1,28,28).to(device)
fm1_testy_tensor = torch.tensor(fm1testys).to(device)

print(fm1_trainx_tensor.shape) # correct shapes
print(fm1_valx_tensor.shape)
print(fm1_testx_tensor.shape)


BATCH_SIZE = 256 # initialize batch size

fm1_trainset = torch.utils.data.TensorDataset(fm1_trainx_tensor
    , fm1_trainy_tensor) # Handles the batchng of the data
fm1_valset = torch.utils.data.TensorDataset(fm1_valx_tensor,
    fm1_valy_tensor)
fm1_testset = torch.utils.data.TensorDataset(fm1_testx_tensor,
    fm1_testy_tensor)
# data set organises the data in one veriable per pair of data
    sets for x and y
fm1_trainloader = torch.utils.data.DataLoader(fm1_trainset,
    batch_size=BATCH_SIZE, shuffle=True, num_workers=0)
fm1_valloader = torch.utils.data.DataLoader(fm1_valset,
    batch_size= BATCH_SIZE, shuffle=True, num_workers=0)
```

```
fm1_testloader = torch.utils.data.DataLoader(fm1_testset,
    batch_size=BATCH_SIZE, shuffle=True, num_workers=0) #
    num_workers =0 it means CPU is used for datloading not GPU,
     but 2 in CPU mode then you can use 2 processes
# organises the variable data saved in terms of batches when
    loading


class CNN_3(nn.Module):
  def __init__(self):
    super(CNN_3, self).__init__() # without padding kernel
       dedicates 1 from each dimension of width and height
    self.conv1 = nn.Conv2d(1, 10, kernel_size = 3, padding =
       (1,1))  # we specify (input_channels, out_channels)
    self.conv2 = nn.Conv2d(10,50, kernel_size = 3, padding =
       (1,1)) # Padding adds a dimension on the left and right
    self.conv3 = nn.Conv2d(50,20, kernel_size = 3, padding =
       (1,1))
    self.dropout = nn.Dropout2d(0.5)
    self.pool = nn.MaxPool2d(2,2)
    self.fc1 = nn.Linear(20*3*3,50)
    self.fc2 = nn.Linear(50,5)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.pool(x) # apply filter convolution to input
       followed by relu activation followed by pooling
    x = self.dropout(x)
    x = F.relu(self.conv2(x))
    x = self.pool(x)
    x = self.dropout(x)
    x = F.relu(self.conv3(x))
    x = self.pool(x)
    x = x.view(-1, 20 * 3* 3) # flatten tensor
    x = F.relu(self.fc1(x))
    x = self.fc2(x)

    return x

  def featuremaps(self,x):
    x = F.relu(self.conv1(x))
    Featuremaps1 = x

    x = self.pool(x)
    x = F.relu(self.conv2(x))
    Featuremaps2 = x

    x = self.pool(x)
    x = F.relu(self.conv3(x))
    Featuremaps3 = x

    return Featuremaps1, Featuremaps2, Featuremaps3

learning_rate = 0.0001
num_epoch = 200
model = CNN_3() # calling my instance of CNN class
model.to(device)
criterion = nn.CrossEntropyLoss()
optim_a = torch.optim.Adam(model.parameters(), lr =
   learning_rate)  # Adam is optimization algorithem
```

```
def train_valid(num_epoch = 200):

  #optimizer = optim()  # calling specified optimiser
  optimizer = optim_a
  #initialize empty lists to store values for plots
  epoch_num = []
  train_l = []
  train_acc = []
  valid_l = []
  valid_acc = []


  # Training Run. hidden - just hidden states into the next
  for epoch in range(num_epoch+1):

    # training
    model.train()
    train_loss = 0
    train_accuracy = 0

    for batch , (xs , ys) in enumerate(fm1_trainloader):

      optimizer.zero_grad() # clears the gradient from the
      output = model(xs) # so coming in of length

      loss = criterion(output , ys.long()) # loss without
         regulariser (as specified in question)
      loss.backward()
      optimizer.step() # updating the weights in steps
      train_loss = train_loss + loss.item() # stores loss (
         without regulariser)
      #batch accuracy calculated with only the last element
      train_accuracy = train_accuracy + torch.sum(torch.argmax(
         output,dim=output.ndim-1) ==ys).item()/BATCH_SIZE #
         batch accuracy #.view(-1, 5   ,  1 )[:,-1,:],1).
         indices

    # update initialized lists
    train_l.append(train_loss/len(fm1_trainloader))
    train_acc.append(train_accuracy/len(fm1_trainloader)*100)


    # validation

    model.eval ()
    valid_loss = 0
    valid_accuracy = 0

    for batch , (xs , ys) in enumerate(fm1_valloader):
      output = model(xs) # so coming in of length
      loss = criterion(output , ys.long())
      #log data
      #aggregate batch performance
      valid_loss = valid_loss + loss.item()
      valid_accuracy = valid_accuracy + torch.sum(torch.argmax(
         output,dim=output.ndim-1)==ys).item()/BATCH_SIZE  #
         we use full batch in
```

```
    # update initialized lists
    epoch_num.append(epoch)
    valid_l.append(valid_loss/len(fm1_valloader))
    valid_acc.append(valid_accuracy/len(fm1_valloader)*100)

    # print summary for each epoch
    print("Epoch: {}/{}.............. ".format(epoch , num_epoch
        ), end= ' ')
    print("Train_Loss: {:.3f}".format(train_loss/len(
        fm1_trainloader)), end= ' ')
    print("Train_Accuracy: {:.3f}".format(train_accuracy/len(
        fm1_trainloader)), end= ' ')
    print("Val_Loss: {:.3f}".format(valid_loss/len(
        fm1_valloader)), end= ' ')
    print("Val_Accuracy: {:.3f}".format(valid_accuracy/len(
        fm1_valloader)))
# Place the plots outside the batches loop. Keep it in the
    epochs loop as all the calculation will be calculated as a
     plot per per epoch.
 plt.plot(range(len(train_l)), train_l  , label = "Train") #
    range(len(list)) - axis is number of elenents in the list
 plt.plot(range(len(valid_l)), valid_l , label = "Validation")
 plt.title("Loss per Epoch")
 plt.xlabel("Number of epochs")
 plt.ylabel( "Loss")
 plt.legend()
 plt.show()

 plt.plot(range(len(train_acc)), train_acc, label = "Train") #
     number of elements per epoch against what you actually
    want
 plt.plot(range(len(valid_acc)), valid_acc, label = "
    Validation")
 plt.title("Accuracy per Epoch")
 plt.xlabel("Number of epoch")
 plt.ylabel("Accuracy %")
 plt.legend()
 plt.show()
```

3.3.2 Train your final model to convergence on the training set using an optimisation algorithm of your choice

After implementing the CNN using cross entropy loss it was trained to convergence using the following parameters:

- Learning rate = 0.0001
- Epochs = 200
- Batch size = 256

The Adam optimiser was used in the model due to its adaptive learning rate which enables it to readjust the weights and biases accordingly as the learning rate changes, this allows the model to perform as accurately as possible.

Figures 6 and 7 show training and validation loss and accuracy plots for the model. The final training and validation accuracies were 98.6% and 97%, respectively. The output for the final five epochs of training is given below:

```
output (final 5):
```

```
Epoch: 196/200.............  Train_Loss: 0.032 Train_Accuracy:
   0.985 Val_Loss: 0.048 Val_Accuracy: 0.971
Epoch: 197/200.............  Train_Loss: 0.032 Train_Accuracy:
   0.985 Val_Loss: 0.047 Val_Accuracy: 0.970
Epoch: 198/200.............  Train_Loss: 0.029 Train_Accuracy:
   0.986 Val_Loss: 0.046 Val_Accuracy: 0.971
Epoch: 199/200.............  Train_Loss: 0.030 Train_Accuracy:
   0.986 Val_Loss: 0.045 Val_Accuracy: 0.971
Epoch: 200/200.............  Train_Loss: 0.029 Train_Accuracy:
   0.986 Val_Loss: 0.048 Val_Accuracy: 0.970
```

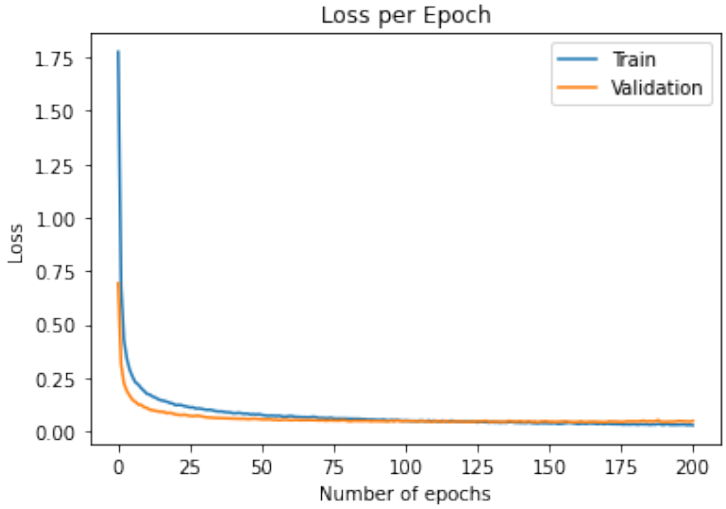


Figure 6: Training and validation plot for loss

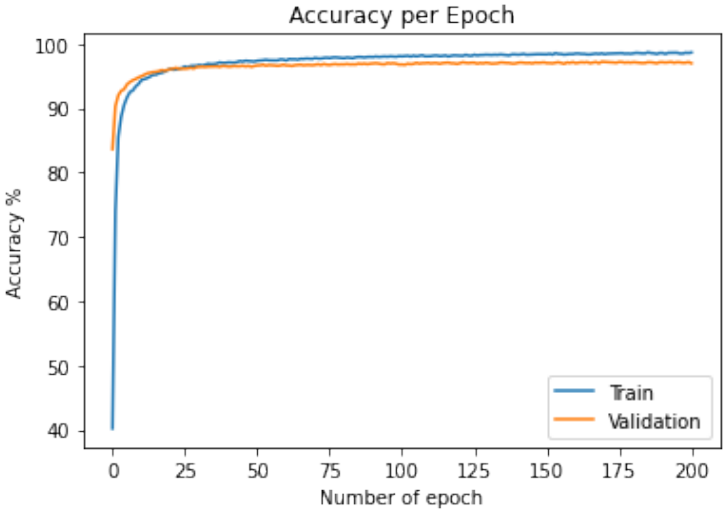


Figure 7: Training and validation plot for accuracy

3.3.3 Retrieve and visualise the feature maps for each layer of your convolutional neural network

The Fashion MNIST-1 split dataset classes are represented by the following labels: $0, 1, 4, 5, 8$, in the code for this problem they are printed as: $0, 1, 2, 3, 4$. A key has been constructed to give clear indications of viewing MNIST-1 classes in terms of the code that was used to represent them. Key:

Class labels: (in the form of original label = given label in code = item of clothing) : 0 = 0 = Tshirt/top, 1 = 1 = Trouser, 4 = 2 = Coat, 5 = 3 = Sandal, 8 = 4 = Bag.

A clearer way to see the key representation of the MNIST 1 class to the way that is represented it in the code below ;

- MNIST 1 Class 0 => Target Class in code 0
- MNIST 1 Class 1 => Target Class in code 1
- MNIST 1 Class 4 => Target Class in code 2
- MNIST 1 Class 5 => Target Class in code 3
- MNIST 1 Class 8 => Target Class in code 4

```
class_toget = 0; indexes_to_get = fm1trainys == class_toget #
    indexes to get class we want , label a variabe to a boolen.
class_images = fm1trainxs[indexes_to_get,:,:] # images fri the
    training data to get the classes we want
plt.imshow(class_images[0,:,:])
```

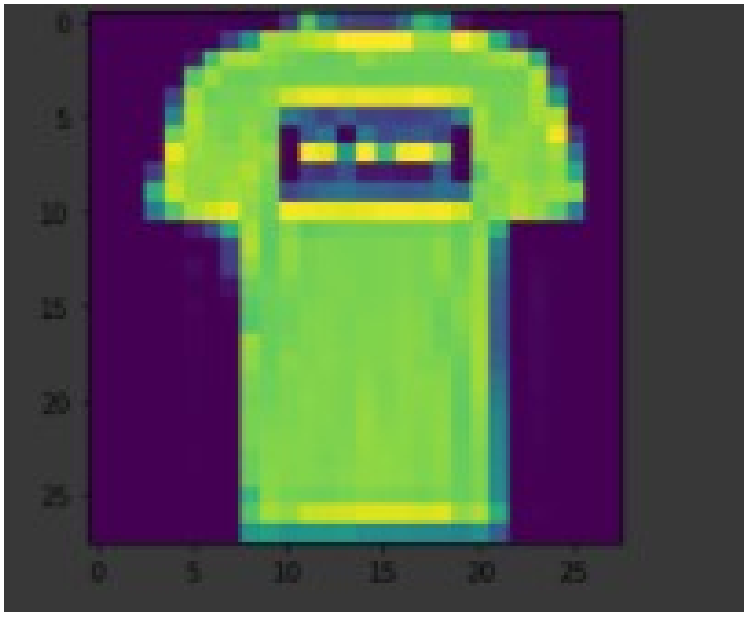


Figure 8: Class 0 - (Tshirt/top)

```
xs = torch.tensor(class_images[0,:,:].reshape(1,1,28,28), dtype
     = torch.float32).to('cuda') # data is a torch tensor that
    is reshaped per channel. Extra dimension added
feature1, feature2, feature3 = model.featuremaps(xs) # torch
    tensir in the model.
imw = 3; j = 0
layer_list = [feature1, feature2, feature3]; fig,ax = plt.
    subplots(len(layer_list), 6, figsize = (imw * 6, len(
    layer_list) * imw))
```

```
for layer in layer_list:
  for i in range(6):
    feature_im = layer[0, i, :, :].detach().to('cpu').numpy()
    ax[j][i].imshow(feature_im)
  j += 1
```

Figure 9 focuses on the Class 0 (Tshirt/top) feature maps. The first row represents the feature maps of the first convolutional layer, and 6 corresponding channels with it. First row and first image means the feature map comes from the first layer and first channel, then the second feature map on the first row represents the feature map from the first layer and the second channel, where the other images in the first row will be from the first layer corresponding to their respective 6 channels. Second row represents the feature maps of the second convolutional layer, and 6 corresponding channels with it from 1st channel to 6th. Third row represents the feature maps of the third convolutional layer, and 6 corresponding channels with it from 1st channel to 6th. (The same idea applies for the other following classes below).

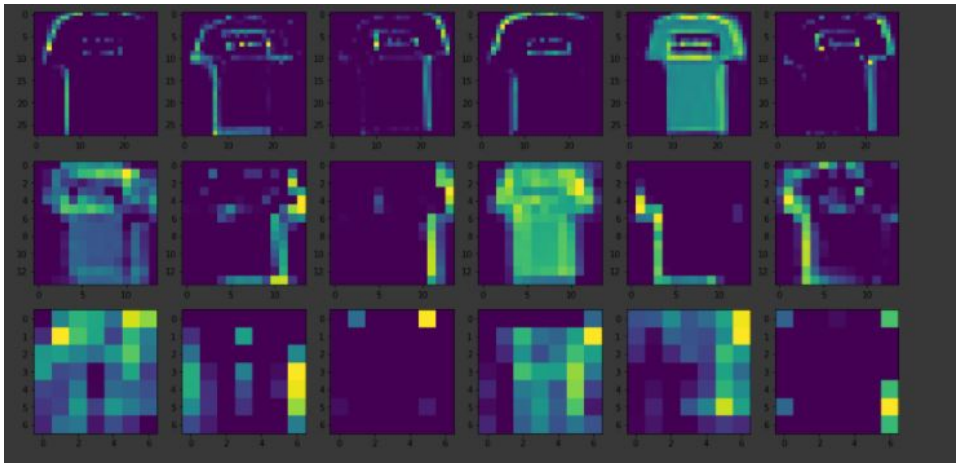


Figure 9: Class 0 (Tshirt/top) : 3 Convolutional layers and 6 channels representation

```
class_toget = 1; indexes_to_get = fm1trainys == class_toget #
    indexes to get class we want , label a variabe to a boolen.
class_images = fm1trainxs[indexes_to_get,:,:] # images fri the
    training data to get the classes we want

plt.imshow(class_images[0,:,:])
```

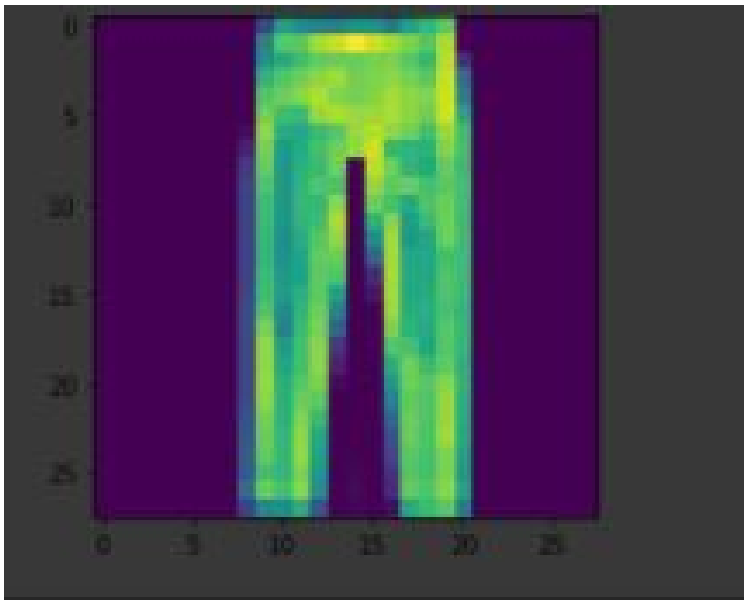


Figure 10: Class 1 (Trouser)

```
xs = torch.tensor(class_images[0,:,:].reshape(1,1,28,28), dtype
     = torch.float32).to('cuda') # data is a torch tensor that
    is reshaped per channel. Extra dimension added
feature1, feature2, feature3 = model.featuremaps(xs) # torch
    tensir in the model.
imw = 3; j = 0
layer_list = [feature1, feature2, feature3]; fig,ax = plt.
    subplots(len(layer_list), 6, figsize = (imw * 6, len(
    layer_list) * imw))
for layer in layer_list:
  for i in range(6):
    feature_im = layer[0, i, :, :].detach().to('cpu').numpy()
    ax[j][i].imshow(feature_im)
  j += 1
```

Figure 11 focuses on the Class 1 (Trouser) feature maps.

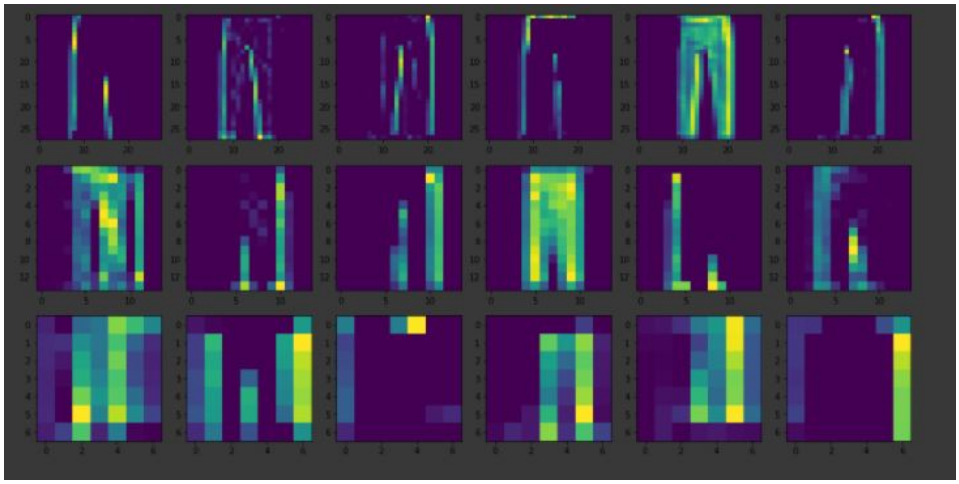


Figure 11: Class 1 (Trouser) : 3 Convolutional layers and 6 channels representation

```
class_toget = 2; indexes_to_get = fm1trainys == class_toget #
    indexes to get class we want , label a variabe to a boolen.
class_images = fm1trainxs[indexes_to_get,:,:] # images fri the
    training data to get the classes we want

plt.imshow(class_images[0,:,:])
```

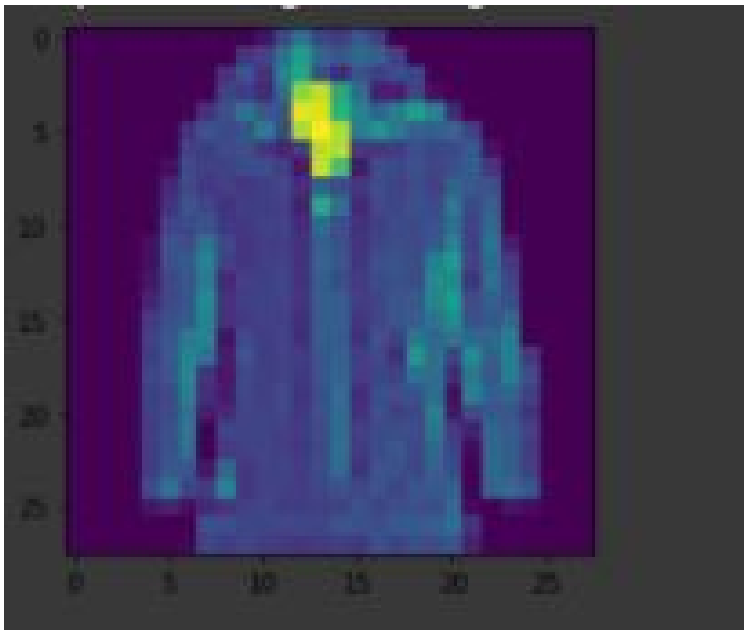


Figure 12: Class 2 (Coat)

```
xs = torch.tensor(class_images[0,:,:].reshape(1,1,28,28), dtype
     = torch.float32).to('cuda') # data is a torch tensor that
    is reshaped per channel. Extra dimension added
feature1, feature2, feature3 = model.featuremaps(xs) # torch
    tensir in the model.

imw = 3; j = 0
layer_list = [feature1, feature2, feature3]; fig,ax = plt.
    subplots(len(layer_list), 6, figsize = (imw * 6, len(
    layer_list) * imw))
for layer in layer_list:
  for i in range(6):
    feature_im = layer[0, i, :, :].detach().to('cpu').numpy()
    ax[j][i].imshow(feature_im)
  j += 1
```

Figure 13 focuses on the Class 2 (Coat) feature maps.

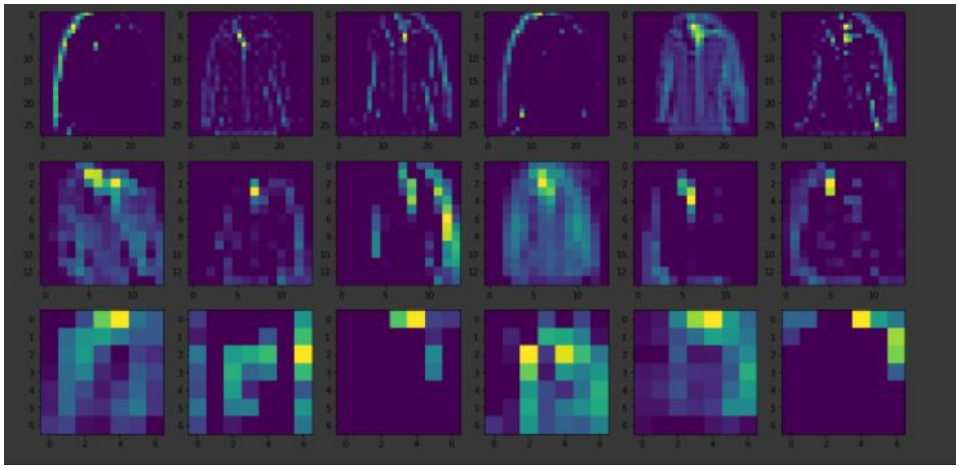


Figure 13: Class 2 (Coat) : 3 Convolutional layers and 6 channels representation

```
class_toget = 3; indexes_to_get = fm1trainys == class_toget #
    indexes to get class we want , label a variable to a
    boolean.
class_images = fm1trainxs[indexes_to_get,:,:] # images fri the
    training data to get the classes we want

plt.imshow(class_images[0,:,:])
```

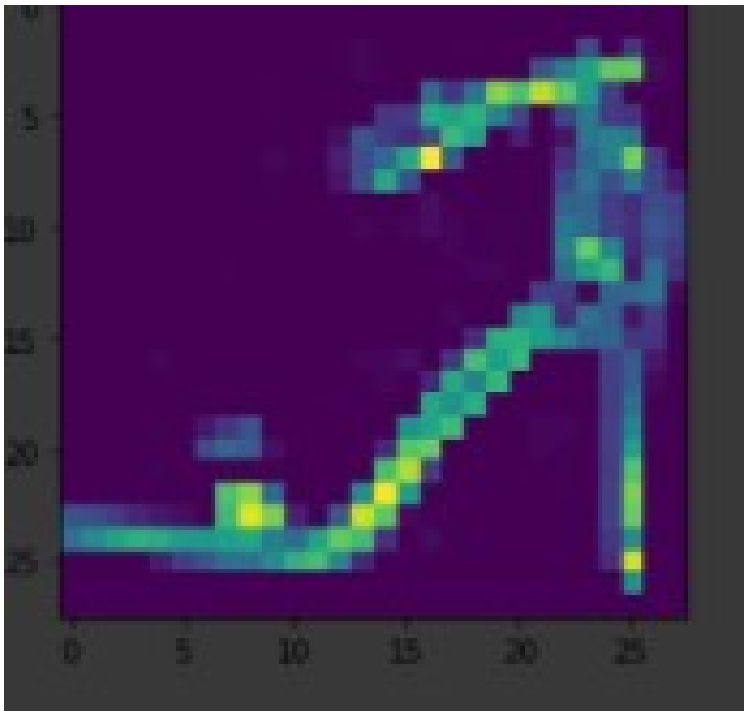


Figure 14: Class 3 - (Sandal)

```
xs = torch.tensor(class_images[0,:,:].reshape(1,1,28,28), dtype
     = torch.float32).to('cuda') # data is a torch tensor that
    is reshaped per channel. Extra dimension added
feature1, feature2, feature3 = model.featuremaps(xs) # torch
    tensir in the model.

imw = 3; j = 0
layer_list = [feature1, feature2, feature3]; fig,ax = plt.
    subplots(len(layer_list), 6, figsize = (imw * 6, len(
    layer_list) * imw))
for layer in layer_list:
  for i in range(6):
    feature_im = layer[0, i, :, :].detach().to('cpu').numpy()
    ax[j][i].imshow(feature_im)
  j += 1
```

Figure 15 focuses on the Class 3 (Sandal) feature maps.

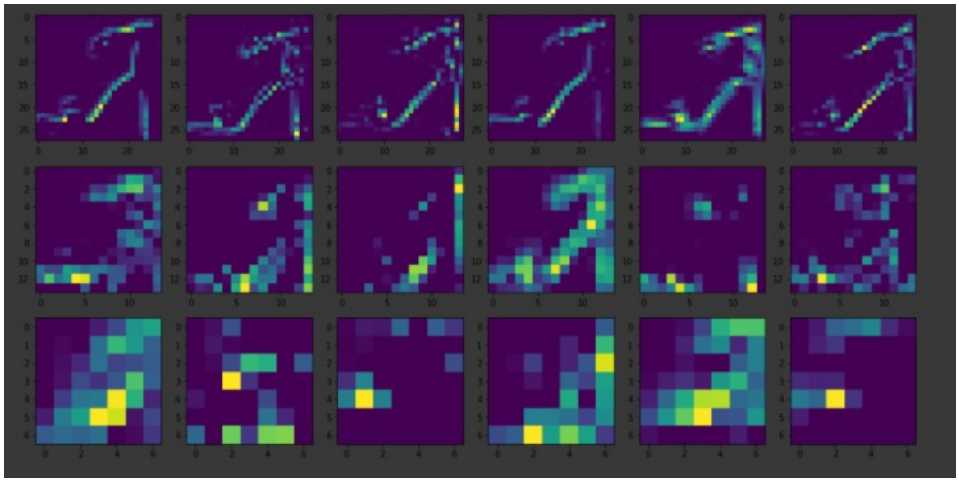


Figure 15: Class 3 (Sandal) : 3 Convolutional layers and 6 channels representation

```
class_toget = 4; indexes_to_get = fm1trainys == class_toget #
    indexes to get class we want , label a variabe to a boolen.
class_images = fm1trainxs[indexes_to_get,:,:] # images fri the
    training data to get the classes we want

plt.imshow(class_images[0,:,:])
```

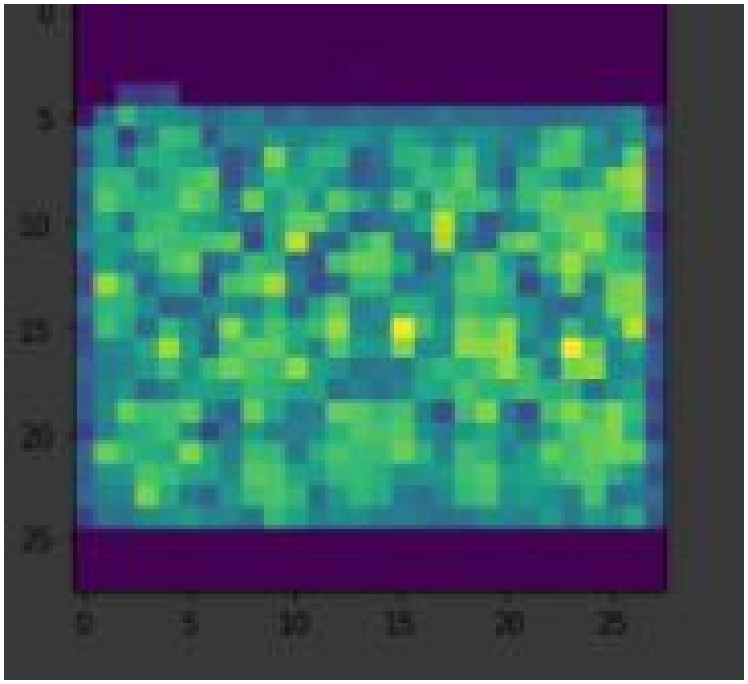


Figure 16: Class 4 (Bag)

```
xs = torch.tensor(class_images[0,:,:].reshape(1,1,28,28), dtype
     = torch.float32).to('cuda') # data is a torch tensor that
    is reshaped per channel. Extra dimension added
feature1, feature2, feature3 = model.featuremaps(xs) # torch
    tensir in the model.

imw = 3; j = 0
layer_list = [feature1, feature2, feature3]; fig,ax = plt.
    subplots(len(layer_list), 6, figsize = (imw * 6, len(
    layer_list) * imw))
for layer in layer_list:
  for i in range(6):
    feature_im = layer[0, i, :, :].detach().to('cpu').numpy()
    ax[j][i].imshow(feature_im)
  j += 1
```

Figure **??** focuses on the Class 4 (Bag) feature maps.

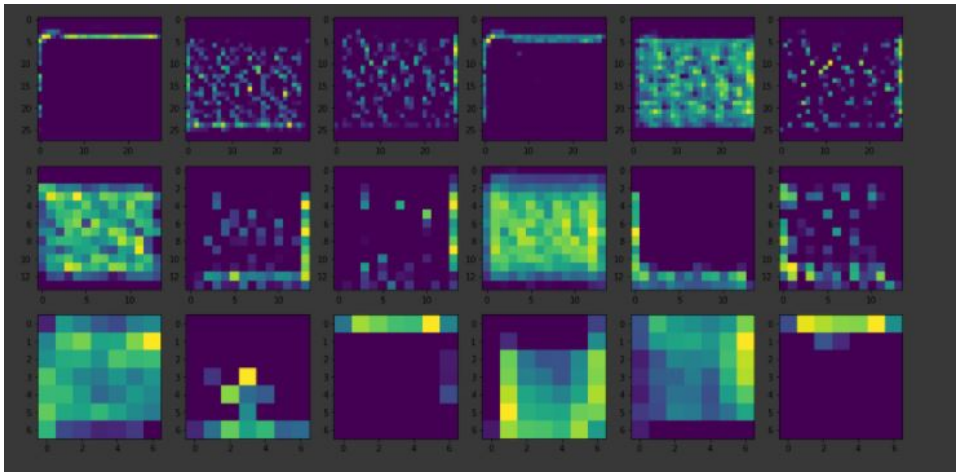


Figure 17: Class 4 (Bag) : 3 Convolutional layers and 6 channels representation

All relevant features and the following feature maps per convolutional layer were retrieved and shown for each class.

3.3.4 Qualitatively analyse the feature maps and hypothesise what they capture and if possible – in particular for the deeper layers – associate them with the output classes

Using same key as above in 3.3.3.

Class labels: (in the form of original label = given label in code = item of clothing) : 0 = 0 = Tshirt/top, 1 = 1 = Trouser, 4 = 2 = Coat, 5 = 3 = Sandal, 8 = 4 = Bag.

A clearer way to see the key representation of the MNIST 1 class to the way that is represented it in the code in 3.3.3 ;

- MNIST 1 Class 0 => Target Class in code 0
- MNIST 1 Class 1 => Target Class in code 1
- MNIST 1 Class 4 => Target Class in code 2
- MNIST 1 Class 5 => Target Class in code 3
- MNIST 1 Class 8 => Target Class in code 4

The five classes of clothing item considered in the Fashion-MNIST-1 data set map to 0, 1, 2, 3, 4 accordingly, as described in the key and the code used in 3.3.3.

The feature maps are displayed in the previous sections of this question. A feature map is the result of a convolution operation being performed on an input in its respective convolution layer (i.e the result of a filter being applied on a input in a convolution layer). These feature maps have been produced using a 3 layer convolutional neural network.

Let $k \times c$ represent a way of expressing the feature map that is being referred to, in the 3x6 representation of features maps with respect to number of layers in the convolutions layer and the number of channels in the colours above as shown in 3.3.3. In the case of the 3 layer model, k can only be 1, 2, 3 when referring to the position of the layer in the CNN (e.g. the rows in figure 17). Similarly, c represents the position of the channel that is being referred to in the images above, which are limited to 1, 2, 3, 4, 5, 6 in this case (e.g. the columns in figure 17).

There are general trends and individual trends for each class respectively. In the general case, the feature maps produced in the first layer (first row of the 3x6 images) individually for each channel produced a clearer/ more image-like structure to the original image of the class which are detecting relevant edges. The feature maps of the second layer find it harder to identify relevant features of the image (second row of the 3x6 images). The feature maps of the third layer become more more pixelated, as its aim is focusing on retrieving information on more complex features (the third row of

the 3x6 images above). As one progresses to deeper convolutional layers the fewer pixels remaining in each layer will start to look pixelated and become harder to identify both visually and for the model.

In every layer the purpose of the filter is to assist in capturing patterns. The first layer captures structures such as simple edges, and the subsequent layers combine these patterns to create more complex features. The further into the layers that the model goes, it identifies more complex shapes that that make the elements visually more difficult to identify.

The brighter areas (the more yellow/green parts of the image) are the activated regions, meaning the filters of the CNN have detected a desired pattern of the features which can help identify images in the respective class better. Therefore, the brighter the region of the feature map more information on the complexity of the features (depending on layer and channel of the CNN) are obtained.

For class 0 (Tshirt/top) on the MNIST-1 data set figure 18 represents the feature maps

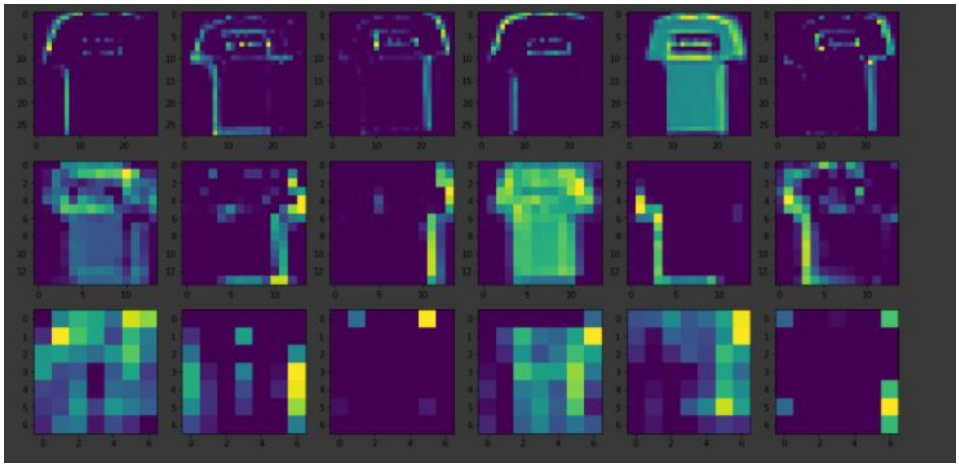


Figure 18: Class 0 (Tshirt/top) : 3 Convolutional layers and 6 channels representation

It can be seen from the feature map of the first convolutional layer (first row of images) that it resembles the images in class 0 (Tshirt/top) in the MNIST-1 data set producing T-shirts. The first layer is detecting edges and shapes more clearly of the images in the class, as not much information is lost through the first convolutional layer. It can clearly be seen from the first to third channels that the edges of the t-shirt are being picked up. The second layer produces feature maps that are less clear, as expected. It is worth noting that channels 4 and 5 across all three layers are quite bright which indicates that the filters have detected relevant features of the T-shirts.

For class 1 (Trouser) on the MNIST-1 data set the feature maps are produced in figure 19.

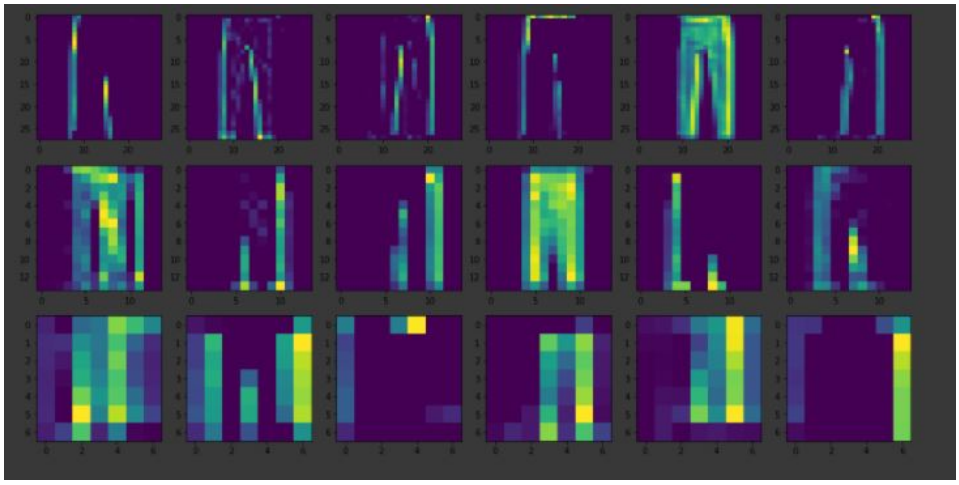


Figure 19: Class 1 (Trouser) : 3 Convolutional layers and 6 channels representation

It can be seen that the feature map of the first convolutional layer producing identifiable images of trousers. The first layer (first row of images) is detecting edges and shapes more clearly of the images in the class 1 (Trouser), as not much information is lost through the first convolutional layer despite the fact there are brighter regions in the second layer (second row of images) so better chance of detecting patterns relevant to the class in the second layer. The third layer is quite bright meaning it is detecting more complex patterns for trousers in class 1 (Trouser) compared to class 0 (T-shirts).

Class 2 (Coat) on the MNIST1 data set feature maps are in figure 20.

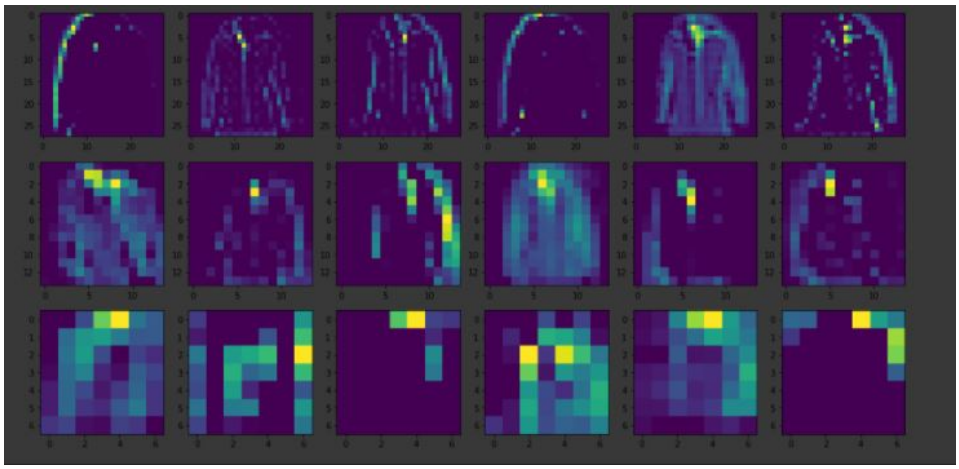


Figure 20: Class 2 (Coat) : 3 Convolutional layers and 6 channels representation

It can be seen that feature map of the first convolutional layer produces identifiable images jackets. The first layer (first row of images) is detecting edges clearly for the images, shown in the feature maps of the first row with different shades / brightness of the coats indicated. The second layer (second row of images) is producing brighter feature maps across the first 4 channels with lighter blue and green representation, this indicates that the second convolutional layer is finding patterns through filtering and detecting more relevant shapes.

For class 3 (Sandal) on the MNIST-1 data set we see the feature represented as follows :

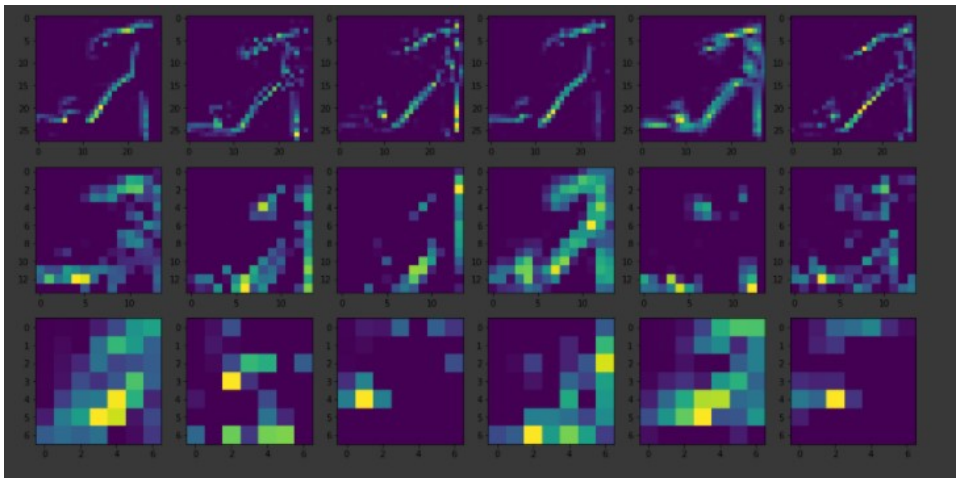


Figure 21: Class 3 (Sandal) : 3 Convolutional layers and 6 channels representation

The feature map of the first convolutional layer producing identifiable high heels, these resemble the images in class 3 (Sandal) in the MNIST-1 data set producing high heels/shoes/sandles type footwear. The second layer (second row of images) is producing brighter feature maps across 6 channels, with lighter blue and green representation which is showing the second convolutional layer is finding better patterns through filtering and detecting more relevant edges and shapes. Also the third layer (third row of images) is producing focused bright regions as well in the 1st, 2nd, 4th and 5th channel meaning more complex patterns is being recognised thus should be able to pick up more information about the class 3 (Sandal).

For class 4 (Bag) on the MNIST1 dataset, the feature maps are presented in figure 22.

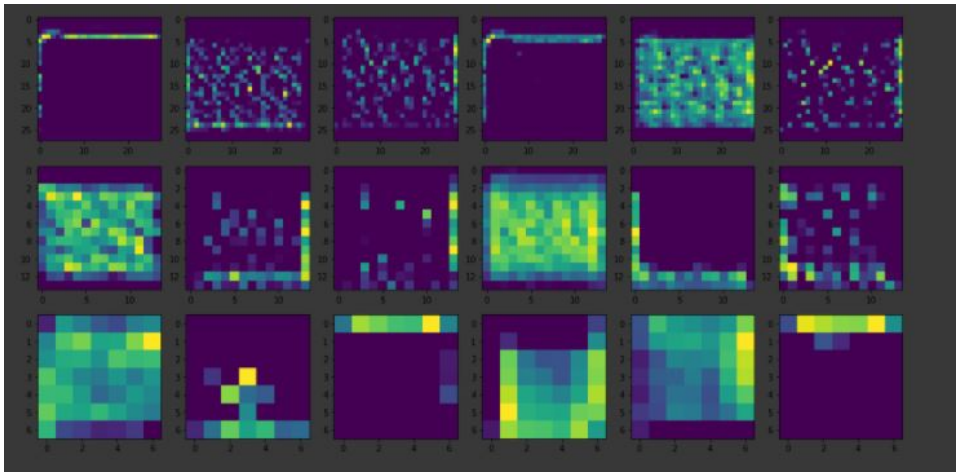


Figure 22: Class 4 (Bag) : 3 Convolutional layers and 6 channels representation

The first convolutional layer produces images of purse/handbag handbag, which are hard to identify as the images in class 4 (Bag) in the MNIST-1 data set. The third (third row of images) layer produces bright regions on all 6 channels meaning that more complex patterns are being recognised, in theory it should be able to pick up more information about the class 4 (Bag), but in reality the feature maps on all three layers fail to give an identifiable visual representation of the bag item.

Overall it can be seen that classes 0 and 1 have the most features correctly detected, as there are more activation regions in general compared to classes 2, 3 and 4. This means that the CNN is good at picking up at relevant features for the t-shirt and trouser items as compared with coats, sandals and bag classes. The features that appear to be identified best are the trousers, as its feature maps contain

the most bright regions. This makes sense as the object itself has a unique shape with sharp edges that are easily detectable. The poorest feature map representations are on class 4 for the bag item. This intuitively makes sense as a bag is a rectangular item, meaning that it is more likely to pick up on the sharp edges than the defining features such as handle or pattern detail. Additionally, if there were another rectangular object that required identification, the model would probably be confused with the bag since it is not a unique shape, and the model itself does not perform as well at defining complex features in the second and third layers.

3.4 Pre-training

Pre-processing: Below are the packages which were loaded in:

```
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import torch
import torchvision
import torchvision.transforms as transforms

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

from torch.autograd import Variable

!pip install idx2numpy
import idx2numpy
```

The training and test images and labels had to be loaded and the training data split into a training and validation set.

```
# importing the training images and splitting it into a
    training and validation set
train_data = idx2numpy.convert_from_file('train-images-idx3-
    ubyte')
y_train_data = idx2numpy.convert_from_file('train-labels-idx1-
    ubyte')
x_train = train_data[:50000,:]
y_train = y_train_data[:50000]

x_val = train_data[50000:,:]
y_val = y_train_data[50000:]

x_test = idx2numpy.convert_from_file('t10k-images-idx3-ubyte')
y_test = idx2numpy.convert_from_file('t10k-labels-idx1-ubyte')
```

A filter function was defined and the applied to the training, set and validation sets:

```
ef filter(xs, ys, lbls):
    idxs = []
    for (i,y) in enumerate(ys):
        if y in lbls:
            idxs.append([i])
    xsprime = np.zeros((len(idxs), xs.shape[1], xs.shape[2]))
    for (i, j) in enumerate(idxs):
        xsprime[i, :, :] = xs[j, :, :]
    ysprime = np.zeros(len(idxs))
    for (i, j) in enumerate(idxs):
        ysprime[i] = ys[j]
    return xsprime, ysprime

fm1lbls                = [0, 1, 4, 5, 8]
fm1trainxs, fm1trainys = filter(x_train, y_train, fm1lbls)
fm1validxs, fm1validys = filter(x_val, y_val, fm1lbls)
fm1testxs, fm1testys   = filter(x_test, y_test, fm1lbls)

fm2lbls                = [2, 3, 6, 7, 9]
```

```
fm2trainxs, fm2trainys = filter(x_train, y_train, fm2lbls)
fm2validxs, fm2validys = filter(x_val, y_val, fm2lbls)
fm2testxs, fm2testys   = filter(x_test, y_test, fm2lbls)
```

From here the appropriate tensors, data-sets and lastly data-loaders were created.

```
if torch.cuda.is_available():
    device = torch.device("cuda")
    print('GPU IS AVAILABLE :D')
else:
    device = torch.device("cpu")
    print('GPU not available')

fm1_trainx_tensor = torch.tensor(fm1trainxs, dtype=torch.float)
   .to(device)
fm1_trainy_tensor = torch.tensor(fm1trainys, dtype=torch.float)
   .to(device)
fm1_valx_tensor = torch.tensor(fm1validxs, dtype=torch.float).
   to(device)
fm1_valy_tensor = torch.tensor(fm1validys, dtype=torch.float).
   to(device)
fm1_testx_tensor = torch.tensor(fm1testxs, dtype=torch.float).
   to(device)
fm1_testy_tensor = torch.tensor(fm1testys, dtype=torch.float).
   to(device)

fm2_trainx_tensor = torch.tensor(fm2trainxs, dtype=torch.float)
   .to(device)
fm2_trainy_tensor = torch.tensor(fm2trainys, dtype=torch.float)
   .to(device)
fm2_valx_tensor = torch.tensor(fm2validxs, dtype=torch.float).
   to(device)
fm2_valy_tensor = torch.tensor(fm2validys, dtype=torch.float).
   to(device)
fm2_testx_tensor = torch.tensor(fm2testxs, dtype=torch.float).
   to(device)
fm2_testy_tensor = torch.tensor(fm2testys, dtype=torch.float).
   to(device)

LEARNING_RATE = 0.001
EPOCHS = 20
BATCH_SIZE = 128

fm1_trainset = torch.utils.data.TensorDataset(fm1_trainx_tensor
   , fm1_trainy_tensor)
fm1_valset = torch.utils.data.TensorDataset(fm1_valx_tensor,
   fm1_valy_tensor)
fm1_testset = torch.utils.data.TensorDataset(fm1_testx_tensor,
   fm1_testy_tensor)

fm2_trainset = torch.utils.data.TensorDataset(fm2_trainx_tensor
   , fm2_trainy_tensor)
fm2_valset = torch.utils.data.TensorDataset(fm2_valx_tensor,
   fm2_valy_tensor)
fm2_testset = torch.utils.data.TensorDataset(fm2_testx_tensor,
   fm2_testy_tensor)


fm1_trainloader = torch.utils.data.DataLoader(fm1_trainset,
   batch_size=BATCH_SIZE, shuffle=True)
```

```
fm1_valloader = torch.utils.data.DataLoader(fm1_valset,
    batch_size=BATCH_SIZE, shuffle=True)
fm1_testloader = torch.utils.data.DataLoader(fm1_testset,
    batch_size=BATCH_SIZE, shuffle=True)

fm2_trainloader = torch.utils.data.DataLoader(fm2_trainset,
    batch_size=BATCH_SIZE, shuffle=True)
fm2_valloader = torch.utils.data.DataLoader(fm2_valset,
    batch_size=BATCH_SIZE, shuffle=True)
fm2_testloader = torch.utils.data.DataLoader(fm2_testset,
    batch_size=BATCH_SIZE, shuffle=True)
```

3.4.1 Implement a convolutional autoencoder with mean squared error loss for the Fashion-MNIST-1 and Fashion-MNIST-2 data

In this section, a convolutional autoencoder was implemented with mean squared error loss for the Fashion-MNIST-1 and Fashion-MNIST-2 data, for which the implementation is shown in the code below. The architecture of the convolutional autoencoder consisted of an encoder with two convolutional layers, going from 1 to 10 to 20 feature maps, and a max pooling layer. The decoder consisted of two transposed convolutional layers, going back down from 20 to 10 to 1 feature map, and an upsampling layer.

```
class ConvolutionalAutoEncoder(nn.Module):
  def __init__(self):
    super(ConvolutionalAutoEncoder, self).__init__()

    self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
    self.conv2 = nn.Conv2d(10, 20, kernel_size=5)

    self.pool = nn.MaxPool2d(2)

    self.t_conv1 = nn.ConvTranspose2d(20, 10, kernel_size=5)
    self.t_conv2 = nn.ConvTranspose2d(10, 1, kernel_size=5)

    self.upsample = nn.Upsample(scale_factor=2)


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

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


    x = self.upsample(x)
    x = F.relu(self.t_conv1(x))

    x = self.upsample(x)
    x = F.relu(self.t_conv2(x))

    return x
```

3.4.2 Train your model to convergence on the combined training, validation, and test set of Fashion-MNIST-2 and training set of Fashion-MNIST-1 using an optimisation algorithm of your choice

The model was trained to convergence on the combined training, validation and test sets of Fashion-MNIST-2 and the training set of Fashion-MNIST-1 using the Adam optimiser, as this was seen to produce the lowest loss.

The training loop used is as follows:

```
def training_loop_autoencoder(model, LEARNING_RATE, trainloader
    , valloader, testx_tensor):
  net = model().to(device)

  criterion = nn.MSELoss()
  optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)

  # training the model:
  train_losses = []
  val_losses = []

  for epoch in range(EPOCHS):
    loss = 0
    for i, data in enumerate(trainloader):
      # get inputs
      images, labels = data
      images = images/255
      images = Variable(images)
      images = images.unsqueeze(1)

      # set param grads to 0
      optimizer.zero_grad()

      #forward + backward + optimize
      output = net(images)

      train_loss = criterion(output, images)

      train_loss.backward()
      optimizer.step()

      loss += train_loss.item()


    loss = loss/(len(trainloader))
    train_losses.append(loss)

    for i, data in enumerate(valloader):
      # get inputs
      images, labels = data
      images = images/255
      images = Variable(images)
      images = images.unsqueeze(1)

      output = net(images)

      val_loss = criterion(output, images)

      loss += val_loss.item()
```

```
    loss = loss/(len(valloader))
    val_losses.append(loss)

    if epoch%1 == 0:
      print('Epoch: %d/%d,  train loss: %.6f, val loss: %.6f'%(
          epoch + 1,EPOCHS, loss, val_loss))


  test_logits = net(testx_tensor.unsqueeze(1)/255)
  test_loss = criterion(test_logits, testx_tensor.unsqueeze(1)
     /255)

  return train_losses, val_losses, test_loss, net
```

Combining the training, validation and test set of Fashion-MNIST-2 and training set of Fashion-MNIST-1:

```
fm2_fm1_trainx = np.concatenate((fm2trainxs, fm2validxs,
    fm2testxs, fm1trainxs), axis=0)
fm2_fm1_trainy = np.concatenate((fm2trainys, fm2validys,
    fm2testys, fm1trainys), axis=None)

fm2_fm1_trainx_tensor = torch.tensor(fm2_fm1_trainx, dtype=
    torch.float).to(device)
fm2_fm1_trainy_tensor = torch.tensor(fm2_fm1_trainy, dtype=
    torch.float).to(device)

fm2_fm1_trainset = torch.utils.data.TensorDataset(
    fm2_fm1_trainx_tensor, fm2_fm1_trainy_tensor)
fm2_fm1_trainloader = torch.utils.data.DataLoader(
    fm2_fm1_trainset, batch_size=BATCH_SIZE, shuffle=True)
```

Training on combined data:

```
train_losses_final, val_losses_final, test_loss_final,
    conv_AE_final = training_loop_autoencoder(
    ConvolutionalAutoEncoder, LEARNING_RATE,
    fm2_fm1_trainloader, fm1_valloader, fm1_testx_tensor)
```

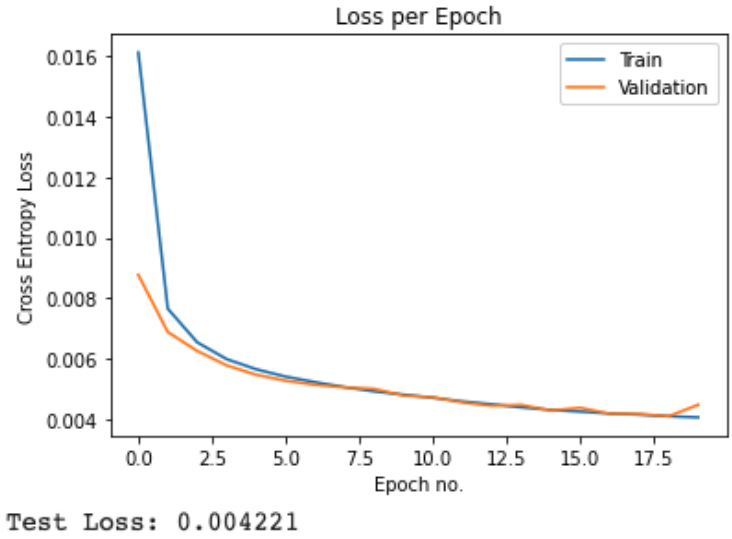


Figure 23: Loss plot for autoencoder trained on combined training data

3.4.3 Implement a multi-class, multi-layer CNN with cross-entropy loss for the Fashion-MNIST-1 data, which shares the same structure as the encoder of your autoencoder

A multi-class, multi-layer convolutional neural network was implemented with cross-entropy loss for the Fashion-MNIST-1 data, which shared the same structure as the encoder of the convolutional autoencoder. Therefore, it consisted of two convolutional layers, taking the number of feature maps from 1 to 10 to 20, with a max pooling layer and two linear layers to reduce the number of nodes from 320 to 50 to 10.

```
class Random_CNN(nn.Module):
  def __init__(self):
    super(Random_CNN, self).__init__()

    self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
    self.conv2 = nn.Conv2d(10, 20, kernel_size=5)

    self.fc1 = nn.Linear(320, 50)
    self.fc2 = nn.Linear(50, 10)

  def forward(self, x):
    x = F.max_pool2d(self.conv1(x), 2)
    x = F.relu(x)

    x = F.max_pool2d(self.conv2(x), 2)
    x = F.relu(x)

    x = x.view(-1, 320)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)

    return x
```

3.4.4 Compare using random weights to those from your autoencoder to initialise the multi-layer perceptron by plotting the training and validation loss for both options when you use 5%, 10%, . . . , 100% of the available Fashion-MNIST-1 training data to train your model. Is there a point where one initialisation option is superior to the other? Is one option always superior to the other?

Two different CNNs were created, one initialised with random weights in all layers and one initialised with the pretrained weights of the convolutional layers in the convolutional autoencoder. For both models, the training and validation losses were plotted, while increasing the amount of Fashion-MNIST-1 training data available to train the models from 5% to 100%, in increments of 5%.

The weights from the final autoencoder were obtained and used to create a CNN:

```
# weights from final autoencoder
conv1_layer_weights = conv_AE_final.conv1.weight.data
conv2_layer_weights = conv_AE_final.conv2.weight.data

class Equivalent_CNN(nn.Module):
  def __init__(self):
    super(Equivalent_CNN, self).__init__()

    self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
    self.conv1.weight.data = conv1_layer_weights
    self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
    self.conv2.weight.data = conv2_layer_weights

    # self.drop_layer = nn.Dropout(p= 0.3)
    self.fc1 = nn.Linear(320, 50)
    self.fc2 = nn.Linear(50, 10)

  def forward(self, x):
    x = F.max_pool2d(self.conv1(x), 2)
    x = F.relu(x)

    x = F.max_pool2d(self.conv2(x), 2)
    x = F.relu(x)

    x = x.view(-1, 320)

    x = F.relu(self.fc1(x))
    # x = self.drop_layer(x)
    x = self.fc2(x)

    return x
```

The Fashion-MNIST-1 training data was then split into sets of 5% to 100% in increments of 5% as follows:

```
def splitting_xs(dataset, i):
  ind = np.int(np.round(dataset.shape[0]/20))
  dataset_new = dataset[:(i+1)*ind, :]
  return dataset_new

def splitting_ys(dataset,i):
  ind = np.int(np.round(dataset.shape[0]/20))
  dataset_new = dataset[:(i+1)*ind]
  return dataset_new

xs_splits = []
ys_splits = []
```

```
for i in range(19):
  xs_splits.append(splitting_xs(fm1_trainx_tensor, i))
  ys_splits.append(splitting_ys(fm1_trainy_tensor, i))

xs_splits.append(fm1_trainx_tensor)
ys_splits.append(fm1_trainy_tensor)

trainset_splits = []

for i in range(20):
  trainset_splits.append(torch.utils.data.TensorDataset(
     xs_splits[i], ys_splits[i]))

trainloader_splits = []
for i in range(20):
  trainloader_splits.append(torch.utils.data.DataLoader(
     trainset_splits[i], batch_size=BATCH_SIZE, shuffle=True))
```

Given that the autoencoder had already been trained on the training set of the Fashion-MNIST-1 dataset, it would be expected the CNN with pretrained weights to have a lower training loss than the CNN with random weights. However, it's expected that both CNNs would have roughly the same validation loss, as neither model had been previously exposed to the validation set of Fashion-MNIST-1.

The implementation and plots for this can be seen in figure 24.

The training loop used is as follows:

```
# parallelizing val CUDA
def training_loop_CNN(model, trainloader, valloader, testloader
    ):
     net = model().to(device)
     LEARNING_RATE = 0.001
     EPOCHS = 20

     # picking cost function and optimizer
     criterion = nn.CrossEntropyLoss()
     optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE)

     # training the model:
     train_losses = []
     val_losses = []
     test_losses = []
     train_accuracies = []
     val_accuracies = []
     test_accuracies = []

     for epoch in range(EPOCHS):
         running_train_loss = 0.0
         running_train_acc = 0.0

         for i, data in enumerate(trainloader):
             # set param grads to 0
             optimizer.zero_grad()

             # get inputs
             images, labels = data
             images, labels = Variable(images), Variable(labels)
             images = images.unsqueeze(1)
             labels = labels.type(torch.LongTensor).to(device)
```

```
        #forward + backward + optimize
        output = net(images).to(device)

        train_loss = criterion(output, labels)

        train_loss.backward()
        optimizer.step()

        running_train_loss += train_loss.item()

        train_logits = net(images)
        _, train_predictions = torch.max(train_logits.data,
            1)
        train_correct_predictions = (train_predictions.int
           () == labels.int()).sum()
        train_length = labels.size()[0]
        running_train_acc += train_correct_predictions.item
           ()/train_length

    train_losses.append(running_train_loss / len(
       trainloader))
    train_accuracies.append(running_train_acc / len(
       trainloader))

    running_val_loss = 0.0
    running_val_acc = 0.0

    for i, data in enumerate(valloader):
    #     # get inputs
      images, labels = data
      images, labels = Variable(images), Variable(labels)
      images = images.unsqueeze(1)
      labels = labels.type(torch.LongTensor).to(device)

      val_logits = net(images).to(device)
      val_loss = criterion(val_logits, labels)
      running_val_loss += val_loss.item()
      _, val_predictions = torch.max(val_logits.data, 1)
      val_correct_predictions = (val_predictions.int() ==
         labels.int()).sum()
      val_length = labels.size()[0]
      running_val_acc += val_correct_predictions.item()/
         val_length


    val_losses.append(running_val_loss / len(valloader))
    val_accuracies.append(running_val_acc / len(valloader))

    running_test_loss = 0.0
    running_test_acc = 0.0

    for i, data in enumerate(testloader):
    #     # get inputs
      images, labels = data
      images, labels = Variable(images), Variable(labels)
      images = images.unsqueeze(1)
      labels = labels.type(torch.LongTensor).to(device)

      test_logits = net(images).to(device)
```

```
            test_loss = criterion(test_logits, labels)
            running_test_loss += test_loss.item()
            _, test_predictions = torch.max(test_logits.data, 1)
            test_correct_predictions = (test_predictions.int() ==
                labels.int()).sum()
            test_length = labels.size()[0]
            running_test_acc += test_correct_predictions.item()/
               test_length


        test_losses.append(running_test_loss / len(testloader))
        test_accuracies.append(running_test_acc / len(
           testloader))


        if epoch % 5 == 0:
            print('Epoch: %d/%d,  train loss: %.6f train acc:
               %.6f  val loss: %.6f val acc: %.6f test loss:
               %.6f test acc: %.6f'%(epoch+1,EPOCHS,
               train_losses[epoch],train_accuracies[epoch],
               val_losses[epoch],val_accuracies[epoch],
               test_losses[epoch],test_accuracies[epoch]))

    return train_losses, train_accuracies, val_losses,
       val_accuracies, test_losses, test_accuracies
```

Using random weights:

```
train_losses_rand = []
val_losses_rand = []
train_acc_rand = []
val_acc_rand = []
test_acc_rand = []

for i in range(20):
  train_losses, train_accuracies, val_losses, val_accuracies,
     test_losses, test_accuracies = training_loop_CNN(
     Random_CNN, trainloader_splits[i], fm1_valloader,
                                                fm1_testloader)
  train_losses_rand.append(np.min(train_losses))
  val_losses_rand.append(np.min(val_losses))
  train_acc_rand.append(train_accuracies[np.argmin(val_losses)
     ])
  val_acc_rand.append(val_accuracies[np.argmin(val_losses)])
  test_acc_rand.append(test_accuracies[np.argmin(val_losses)])
```

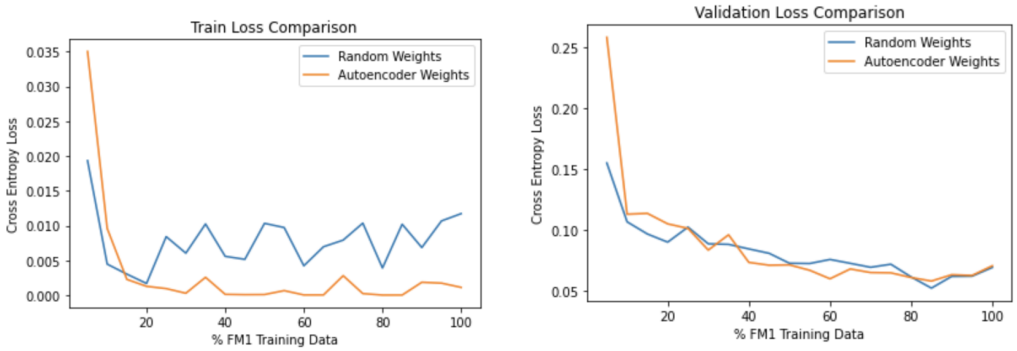
Using weights from autoencoder:

```
train_losses_equiv = []
val_losses_equiv = []
train_acc_equiv = []
val_acc_equiv = []
test_acc_equiv = []

for i in range(20):
  train_losses, train_accuracies, val_losses, val_accuracies,
     test_losses, test_accuracies = training_loop_CNN(
     Equivalent_CNN, trainloader_splits[i],

     fm1_valloader,fm1_testloader)
```

```
train_losses_equiv.append(np.min(train_losses))
val_losses_equiv.append(np.min(val_losses))
train_acc_equiv.append(train_accuracies[np.argmin(val_losses)
    ])
val_acc_equiv.append(val_accuracies[np.argmin(val_losses)])
test_acc_equiv.append(test_accuracies[np.argmin(val_losses)])
```



(a) (b)

Figure 24: Training and validation loss plots for randomly initialised CNN compared to CNN initialised using autoencoder weights.

Considering plot (a) of Figure 24, it can be seen that where very low percentages of training data were used, the CNN initialised with random weights performed better. Quite quickly however, this changes to the CNN initialised with the pre-trained weights out-performing the randomly initialised CNN.

Considering plot (b) shows a similar sentiment for lower percentages of data, however it can be seen that the two models are rather close in performance especially once 90% and more of the data is used for training.

3.4.5 Provide the final accuracy on the training, validation, and test set for the best model you obtained for each of the two initialisation strategies

For the best CNN initialised with random weights, the final accuracies on the training, validation and test sets are:

- Training Accuracy: 100%
- Validation Accuracy: 99.6%
- Test Accuracy: 99.7%

For the best CNN initialised with pretrained convolutional autoencoder weights, the final accuracies on the training, validation and test sets are:

- Training Accuracy: 100%
- Validation Accuracy: 99.6%
- Test Accuracy: 99.7%

For the model with random weights, the lowest validation loss was found after being exposed to 100% of the Fashion-MNIST-1 training dataset, whereas the model with the pretrained autoencoder weights found its lowest validation loss with a smaller training set of 90% of the Fashion-MNIST-1 training dataset.

3.5 Transfer learning

3.5.1 Implement a multi-class, convolutional neural network with cross-entropy loss for the Fashion-MNIST-2 data

Pre-processing: The code below denotes the pre processing steps for the Fashion-MNIST data, a filter function that splits the data into MNIST 1 and MNIST 2. The code for the creation of tensors, tensor-dataset and data-loaders follows the same structure as 3.4.

```
# importing the training images and splitting it into a
    training and validation set
train_data = idx2numpy.convert_from_file('train-images-idx3-
    ubyte')
y_train_data = idx2numpy.convert_from_file('train-labels-idx1-
    ubyte')
x_train = train_data[:50000,:]
y_train = y_train_data[:50000]

x_val = train_data[50000:,:]
y_val = y_train_data[50000:]

x_test = idx2numpy.convert_from_file('t10k-images-idx3-ubyte')
y_test = idx2numpy.convert_from_file('t10k-labels-idx1-ubyte')

def filter(xs, ys, lbls):
  idxs = [i for (i, y) in enumerate(ys) if y in lbls] #labels
     from the original dataset
  xsprime = np.zeros((len(idxs), xs.shape[1],xs.shape[2]))

  for (i, j) in enumerate(idxs):
    xsprime[i, :, :] = xs[j, :, :]

  ymap = dict([(y, yprime) for (yprime, y) in enumerate(lbls)])
  ysprime = [ymap[y] for y in ys[idxs]] #0,1,4,5,8 to 0,1,2,3,4

  return np.array(xsprime), np.array(ysprime) # gets us a 3
     tensor and this is a 1-tensor

fm1lbls                = [0, 1, 4, 5, 8]
fm1trainxs, fm1trainys = filter(x_train, y_train, fm1lbls)
fm1validxs, fm1validys = filter(x_val, y_val, fm1lbls)
fm1testxs, fm1testys   = filter(x_test, y_test, fm1lbls)

fm2lbls                = [2, 3, 6, 7, 9]
fm2trainxs, fm2trainys = filter(x_train, y_train, fm2lbls)
fm2validxs, fm2validys = filter(x_val, y_val, fm2lbls)
fm2testxs, fm2testys   = filter(x_test, y_test, fm2lbls)
```

Class definition for MNIST-2 CNN:

A multi-class, convolutional neural network was implemented with cross-entropy loss for the Fashion-MNIST-2 data. The convolution neural network defined has two convolution layers, two pooling layers (max pooling with stride = 2, stride = 1 respectively) and three fully-connected layers. Dropout layers are added after each convolution operation to prevent overfitting. The first convolution layer has 28 feature maps with filter size = 5 and the second convolution layer has 20 feature maps with filter size = 3. After the convolution operation, the input to MLP is of size 2000 with number of hidden nodes in the 1st layer equal to 500, the number of hidden nodes in the 2nd layer equal to 100 and output nodes equal to 5 (denoting the 5 class labels of the MNIST 2 data).

```
# Initializing CNN for MNIST-2 dataset
```

```
class MNIST2_CNN(nn.Module):
    def __init__(self, p):
        super(MNIST2_CNN, self).__init__()
        # initializing two convolution layers
        self.conv1 = nn.Conv2d(1, 28, kernel_size=5)
        self.conv2 = nn.Conv2d(28, 20, kernel_size=3)
        self.drop_layer = nn.Dropout2d(p)

        # initializing the MLP layers
        self.fc1 = nn.Linear(20 * 10 * 10, 500)
        self.fc2 = nn.Linear(500, 100)
        self.fc3 = nn.Linear(100, 5)


    def forward(self, x):
        # max pooling on the 1st conv layer, with stride = 2
        x = F.max_pool2d(self.conv1(x), 2)
        # dropout layers for regularization
        x = self.drop_layer(x)
        x = F.relu(x)
        # max pooling on the 1st conv layer, with stride = 1
        x = F.max_pool2d(self.conv2(x), 1)
        # dropout layers for regularization
        x = self.drop_layer(x)
        x = F.relu(x)
        # flattening the matrix
        x = x.view(-1, x.size(1) * x.size(2) * x.size(3))
        x = F.relu(self.fc1(x))
        x = self.drop_layer(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

The code below is the main function that does the training, validation and testing on the MNIST 2 data. The number of epochs is fixed at 30, and a grid search is used to choose the optimizer and learning rate.

```
# function that trains the model and predicts on the test set
def train_evaluate(net, lr,optimizer,model_selection):
    EPOCHS = 30
    # choosing the learning rate
    LEARNING_RATE = lr

    # picking cost function and optimizer
    criterion = nn.CrossEntropyLoss()
    if optimizer == 'Adam':
        optimizer = optim.Adam(net.parameters(), lr=
           LEARNING_RATE)
    if optimizer == 'SGD':
        optimizer = optim.SGD(net.parameters(), lr=
           LEARNING_RATE, momentum = 0.9)

    # training the model:
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    for epoch in range(EPOCHS):
        running_train_loss = 0.0
```

```
num_of_train_batches = 0
running_train_acc = 0.0

# training on each mini batch
for i, data in enumerate(fm2_trainloader):
    # set param grads to 0
    optimizer.zero_grad()

    # get inputs
    images, labels = data
    images, labels = Variable(images), Variable(labels)
    images = images.unsqueeze(1).to(device)
    labels = labels.type(torch.LongTensor).to(device)

    #forward + backward + optimize
    output = net(images)

    train_loss = criterion(output, labels)

    train_loss.backward()
    optimizer.step()

    running_train_loss += train_loss.item()

    num_of_train_batches += 1
    train_logits = net(images)
    _, train_predictions = torch.max(train_logits.data,
        1)
    train_correct_predictions = (train_predictions.int
       () == labels.int()).sum()
    train_length = labels.size()[0]
    running_train_acc += train_correct_predictions/
       train_length


train_losses.append(running_train_loss /
   num_of_train_batches)
train_accuracies.append(running_train_acc /
   num_of_train_batches)

running_val_loss = 0.0
num_of_val_batches = 0
running_val_acc = 0.0

for i, data in enumerate(fm2_valloader):
  # get inputs
  images, labels = data
  images, labels = Variable(images), Variable(labels)
  images = images.unsqueeze(1).to(device)
  labels = labels.type(torch.LongTensor).to(device)

  #forward + backward + optimize
  val_logits = net(images).to(device)
  val_loss = criterion(val_logits, labels)
  running_val_loss += val_loss.item()
  _, val_predictions = torch.max(val_logits.data, 1)
  val_correct_predictions = (val_predictions.int() ==
     labels.int()).sum()
  val_length = labels.size()[0]
```

```
            running_val_acc += val_correct_predictions/val_length
            num_of_val_batches += 1

        # storing the validation losses and accuracies
        val_losses.append(running_val_loss / num_of_val_batches
            )
        val_accuracies.append(running_val_acc /
            num_of_val_batches)


        if epoch % 3 == 0:
            print('Epoch: %d/%d,  train loss: %.6f train acc:
                %.6f  val loss: %.6f val acc: %.6f'%(epoch+1,
                EPOCHS,train_losses[epoch],train_accuracies[
                epoch],val_losses[epoch],val_accuracies[epoch])
                )


    # predicting on the trained model
    images = fm2_testx_tensor.unsqueeze(1).to(device)
    test_logits = net(images)
    labels = fm2_testy_tensor.type(torch.LongTensor).to(device)
    test_loss = criterion(test_logits.to(device), labels)
    _, test_predictions = torch.max(test_logits.data, 1)
    test_correct_predictions = (test_predictions.int() ==
        labels.int()).sum()
    test_length = labels.size()[0]
    final_test_accuracy = test_correct_predictions/test_length
    print('Test Accuracy: ', final_test_accuracy.item())
    if model_selection == 'True':
        return net, val_losses[-1]
    else:
        return net, train_losses, train_accuracies, val_losses,
            val_accuracies, final_test_accuracy


# plotting the train and validation loss
def plot_loss(train_losses, val_losses):
    plt.plot(range(len(train_losses)), train_losses, label="
        Train")
    plt.plot(range(len(val_losses)), val_losses, label="
        Validation")
    plt.title("Loss per Epoch")
    plt.xlabel("Epoch no.")
    plt.ylabel("Cross Entropy Loss")
    plt.legend()
    plt.show()

# plotting the train and validation accuracy
def plot_accuracy(train_accuracy, val_accuracy):
    plt.plot(range(len(train_accuracy)), train_accuracy, label=
        "Train")
    plt.plot(range(len(val_accuracy)), val_accuracy, label="
        Validation")
    plt.title("Accuracy per Epoch")
    plt.xlabel("Epoch no.")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.show()
```

3.5.2 Iteratively tune your model structure and hyperparameters using the validation set of Fasion-MNIST-2 data, until you arrive at a model performance you are comfortable with

In this section, the best model was found iteratively by running through different values for the learning rate, dropout probabilities and different optimizer functions. The model performed better with dropout regularization and this was tested manually. The grid search is performed on the following different hyperparameters:

- Learning rate: 0.01, 0.001 or 0.0001
- Optimiser type: Adam, Stochastic Gradient Descent (SGD)
- Dropout probabilities: 0, 0.2 or 0.3

```
# Grid search on learning rate, optimizer, dropout
    probabilities:

lr = [0.001, 0.01]
optimizer = ['Adam','SGD']
dropouts = [0, 0.2, 0.3]
val_losses = np.zeros((len(lr),len(optimizer),len(dropouts)))


for i in range(len(lr)):
    for j in range(len(optimizer)):
        for k in range(len(dropouts)):
            print('Learning rate: ',lr[i], ' Optimizer: ',
                optimizer[j],' Dropout: ',dropouts[k])

            CNN_MNIST2 = MNIST2_CNN(dropouts[k]).to(device)
            _, val_losses[i,j,k] = train_evaluate(CNN_MNIST2,
                lr[i], optimizer[j], model_selection = 'True')


best_lr, best_opt, best_p = np.argwhere(val_losses == np.min(
    val_losses))[0]
print('Best params: ',lr[best_lr], optimizer[best_opt],
    dropouts[best_p])
```

Output: Best params: 0.001 SGD 0.3

The parameters that gave the lowest validation loss are: learning rate = 0.001, optimizer = SGD and dropout rate = 0.3.

The MNIST-2 CNN model is further trained on the best parameters, and their loss and accuracy plots are shown in figures 25 and 26.

```
# Training on the best model parameters

CNN_MNIST2 = MNIST2_CNN(dropouts[best_p]).to(device)
best_MNIST2 , train_losses, train_accuracies, val_losses,
    val_accuracies, final_test_accuracy = train_evaluate(
    CNN_MNIST2, lr[best_lr], optimizer[best_opt],
    model_selection = 'False')

# Output:
Epoch: 1/30,  train loss: 0.523256 train acc: 0.793657  val
    loss: 0.389191 val acc: 0.842211
Epoch: 4/30,  train loss: 0.281670 train acc: 0.893781  val
    loss: 0.277249 val acc: 0.894472
Epoch: 7/30,  train loss: 0.231403 train acc: 0.911954  val
    loss: 0.237513 val acc: 0.906533
```

```
Epoch: 10/30,  train loss: 0.211560 train acc: 0.919214  val
   loss: 0.225732 val acc: 0.916382
Epoch: 13/30,  train loss: 0.197516 train acc: 0.924554  val
   loss: 0.221009 val acc: 0.916181
Epoch: 16/30,  train loss: 0.187687 train acc: 0.930534  val
   loss: 0.210760 val acc: 0.922010
Epoch: 19/30,  train loss: 0.173381 train acc: 0.934316  val
   loss: 0.208720 val acc: 0.921809
Epoch: 22/30,  train loss: 0.169503 train acc: 0.937783  val
   loss: 0.206568 val acc: 0.924422
Epoch: 25/30,  train loss: 0.160459 train acc: 0.938341  val
   loss: 0.211983 val acc: 0.923417
Epoch: 28/30,  train loss: 0.153446 train acc: 0.942211  val
   loss: 0.190245 val acc: 0.929648
Test Accuracy:  0.9205999970436096

# plotting losses and accuracy for the best MNIST-2 model
plot_loss(train_losses, val_losses)
plot_accuracy(train_accuracies, val_accuracies)
print("Test Accuracy: %.6f"%final_test_accuracy.item())
```

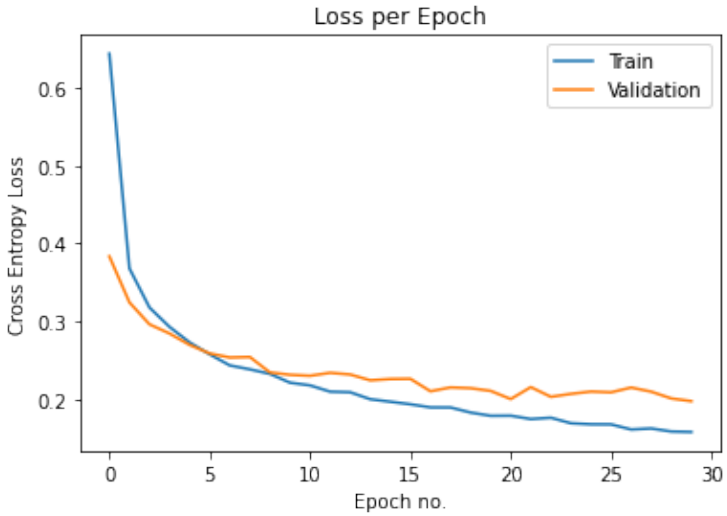


Figure 25: Loss for the training and validation set on the best MNIST-2 model

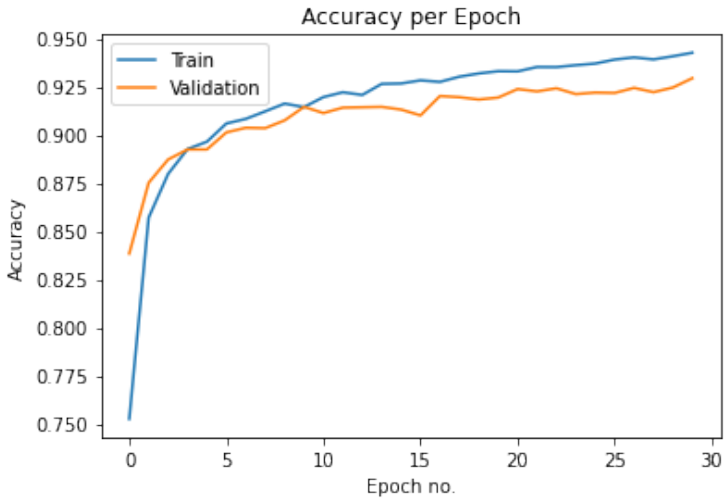


Figure 26: Accuracy for the training and validation set on the best MNIST-2 model

3.5.3 Implement a multi-class, convolutional neural network with cross-entropy loss for the Fashion-MNIST-1 data, which shares the same structure as the one you used for the Fashion- MNIST-2 data

A multi-class, multi-layer convolutional neural network was implemented with cross-entropy loss for the Fashion-MNIST-1 data, which shared the same structure as the MNIST-2 CNN in 3.5.1. The weights for convolutional layers and the MLP layers (except the output layer) in MNIST-1 CNN was initialized using the weights from the best model of MNIST-2 CNN. The classification/ output layer of MNIST-1 CNN is randomly initialized.

Preprocessing of MNIST - 1 data:

```
# Extracting MNIST-1 DATA

fm1_trainx_tensor = torch.tensor(fm1trainxs, dtype=torch.float)
fm1_trainy_tensor = torch.tensor(fm1trainys, dtype=torch.float)
fm1_valx_tensor = torch.tensor(fm1validxs, dtype=torch.float)
fm1_valy_tensor = torch.tensor(fm1validys, dtype=torch.float)
fm1_testx_tensor = torch.tensor(fm1testxs, dtype=torch.float)
fm1_testy_tensor = torch.tensor(fm1testys, dtype=torch.float)

# create tensor dataset for the train, val, test tensors
fm1_trainset = torch.utils.data.TensorDataset(fm1_trainx_tensor
    , fm1_trainy_tensor)
fm1_valset = torch.utils.data.TensorDataset(fm1_valx_tensor,
    fm1_valy_tensor)
fm1_testset = torch.utils.data.TensorDataset(fm1_testx_tensor,
    fm1_testy_tensor)

# create data loaders for the train, val, test datasets
fm1_trainloader = torch.utils.data.DataLoader(fm1_trainset,
    batch_size=BATCH_SIZE, shuffle=True)
fm1_valloader = torch.utils.data.DataLoader(fm1_valset,
    batch_size=fm1_valx_tensor.shape[0], shuffle=True)
fm1_testloader = torch.utils.data.DataLoader(fm1_testset,
    batch_size=fm1_testx_tensor.shape[0], shuffle=True)
```

Transfer Learning:

The weights from the best MNIST-2 model were obtained and used to create a new MNIST-1 CNN:

```
# Initializing CNN

# Creating a class for the MNIST-1 data that uses the weights
   from the best MNIST-2 model
# for the CNN and MLP layers (except the classification layer)
# MNIST1 and MNIST2 has the same architecture
class MNIST1_CNN(nn.Module):
    def __init__(self, p):
        super(MNIST1_CNN, self).__init__()
        # initializing the 1st conv layer
        self.conv1 = nn.Conv2d(1, 28, kernel_size=5)
        # using weights of 1st conv layer from MNIST2 for conv1
            in MNIST1
        self.conv1.weight.data = best_MNIST2.conv1.weight.data
        # initializing the 1st conv layer
        self.conv2 = nn.Conv2d(28, 20, kernel_size=3)
        # using weights of 2nd conv layer from MNIST2 for conv2
            in MNIST1
        self.conv2.weight.data = best_MNIST2.conv2.weight.data
        # adding dropout
        self.drop_layer = nn.Dropout2d(p)
        self.fc1 = nn.Linear(20 * 10 * 10, 500)
        # using weights of linear layers from MNIST2 for linear
            layers in MNIST1
        self.fc1.weight.data = best_MNIST2.fc1.weight.data
        self.fc2 = nn.Linear(500, 100)
        self.fc2.weight.data = best_MNIST2.fc2.weight.data
        self.fc3 = nn.Linear(100, 5)


    def forward(self, x):
        x = F.max_pool2d(self.conv1(x), 2)
        x = self.drop_layer(x)
        x = F.relu(x)
        x = F.max_pool2d(self.conv2(x), 1)
        x = self.drop_layer(x)
        x = F.relu(x)
        x = x.view(-1, x.size(1) * x.size(2) * x.size(3))

        x = F.relu(self.fc1(x))
        x = self.drop_layer(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

CNN_MNIST1 = MNIST1_CNN(dropouts[best_p]).to(device)
best_MNIST1 , train_losses, train_accuracies, val_losses,
   val_accuracies, final_test_accuracy = train_evaluate(
   CNN_MNIST1, lr[best_lr], optimizer[best_opt],
   model_selection='False')

#Output:
Epoch: 1/30,  train loss: 0.231402 train acc: 0.920208  val
   loss: 0.205094 val acc: 0.925829
Epoch: 4/30,  train loss: 0.152415 train acc: 0.944122  val
   loss: 0.195154 val acc: 0.928844
Epoch: 7/30,  train loss: 0.142918 train acc: 0.946475  val
   loss: 0.195146 val acc: 0.927638
```

```
Epoch: 10/30,  train loss: 0.133117 train acc: 0.949981  val
   loss: 0.188420 val acc: 0.934271
Epoch: 13/30,  train loss: 0.130004 train acc: 0.950179  val
   loss: 0.196648 val acc: 0.932261
Epoch: 16/30,  train loss: 0.122709 train acc: 0.952092  val
   loss: 0.192671 val acc: 0.930653
Epoch: 19/30,  train loss: 0.120005 train acc: 0.955402  val
   loss: 0.205785 val acc: 0.928844
Epoch: 22/30,  train loss: 0.112382 train acc: 0.957831  val
   loss: 0.218955 val acc: 0.924824
Epoch: 25/30,  train loss: 0.111274 train acc: 0.959982  val
   loss: 0.199227 val acc: 0.929648
Epoch: 28/30,  train loss: 0.106869 train acc: 0.960823  val
   loss: 0.200160 val acc: 0.928643
Test Accuracy:  0.9285999536514282
```

The test accuracy after training the MNIST-1 class using the transferred weights from MNIST-2 model is 92.8%

3.5.4 Compare using random weights to those obtained by training on Fashion-MNIST-2 – you should randomly re-initialise the classification layer though – to initialise the multi-class, convolutional neural network by plotting the training and validation loss for both options when you use 5%, 10%, . . . , 100% of the available Fashion-MNIST-1 training data to train your model. Is there a point where one initialisation option is superior to the other? Is one option always superior to the other?

A multi-class, multi-layer convolutional neural network was implemented with cross-entropy loss for the Fashion-MNIST-1 data, which shared the same structure as the MNIST-2 CNN in 3.5.1. The weights for convolutional layers in MNIST-1 CNN were randomly initialized instead of using the weights from the best model of MNIST-2 CNN. For both MNIST-1 and MNIST-2 models, the training and validation losses were plotted, while increasing the amount of Fashion-MNIST-1 training data available to train the models from 5% to 100%, in increments of 5%.

```
# Creating a class for the MNIST-1 data that uses random
   weights
# for the CNN and MLP layers
# MNIST1 and MNIST2 has the same architecture


class Random_MNIST1_CNN(nn.Module):
    def __init__(self, p):
        super(Random_MNIST1_CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 28, kernel_size=5)
        # using random weights for conv layers
        # initializing the 1st conv layer
        self.conv2 = nn.Conv2d(28, 20, kernel_size=3)
        # adding dropout
        self.drop_layer = nn.Dropout2d(p)
        self.fc1 = nn.Linear(20 * 10 * 10, 500)
        # using random weights for linear layers
        self.fc2 = nn.Linear(500, 100)
        self.fc3 = nn.Linear(100, 5)

    def forward(self, x):
        x = F.max_pool2d(self.conv1(x), 2)
        x = self.drop_layer(x)
        x = F.relu(x)
        x = F.max_pool2d(self.conv2(x), 1)
        x = self.drop_layer(x)
        x = F.relu(x)
```

```
        x = x.view(-1, x.size(1) * x.size(2) * x.size(3))

        x = F.relu(self.fc1(x))
        x = self.drop_layer(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

Random_CNN_MNIST1 = Random_MNIST1_CNN(dropouts[best_p]).to(
    device)
best_random_MNIST1 , train_losses, train_accuracies, val_losses
    , val_accuracies, final_test_accuracy = train_evaluate(
    Random_CNN_MNIST1, lr[best_lr], optimizer[best_opt],
    model_selection = 'False')

# Output
Epoch: 1/30,  train loss: 0.529768 train acc: 0.786335  val
    loss: 0.418905 val acc: 0.807437
Epoch: 4/30,  train loss: 0.283845 train acc: 0.892387  val
    loss: 0.280119 val acc: 0.887839
Epoch: 7/30,  train loss: 0.240318 train acc: 0.909405  val
    loss: 0.250573 val acc: 0.903518
Epoch: 10/30,  train loss: 0.218535 train acc: 0.919726  val
    loss: 0.223104 val acc: 0.915176
Epoch: 13/30,  train loss: 0.203063 train acc: 0.924750  val
    loss: 0.219077 val acc: 0.918191
Epoch: 16/30,  train loss: 0.189887 train acc: 0.929174  val
    loss: 0.214033 val acc: 0.918794
Epoch: 19/30,  train loss: 0.179339 train acc: 0.933479  val
    loss: 0.203525 val acc: 0.924422
Epoch: 22/30,  train loss: 0.170614 train acc: 0.935311  val
    loss: 0.204045 val acc: 0.919196
Epoch: 25/30,  train loss: 0.162306 train acc: 0.939741  val
    loss: 0.198767 val acc: 0.924422
Epoch: 28/30,  train loss: 0.155661 train acc: 0.940412  val
    loss: 0.212106 val acc: 0.920201
Test Accuracy:  0.9225999712944031

# plotting losses and accuracy for the best MNIST-1 model
plot_loss(train_losses, val_losses)
plot_accuracy(train_accuracies, val_accuracies)
print("Test Accuracy: %.6f"%final_test_accuracy.item())
```

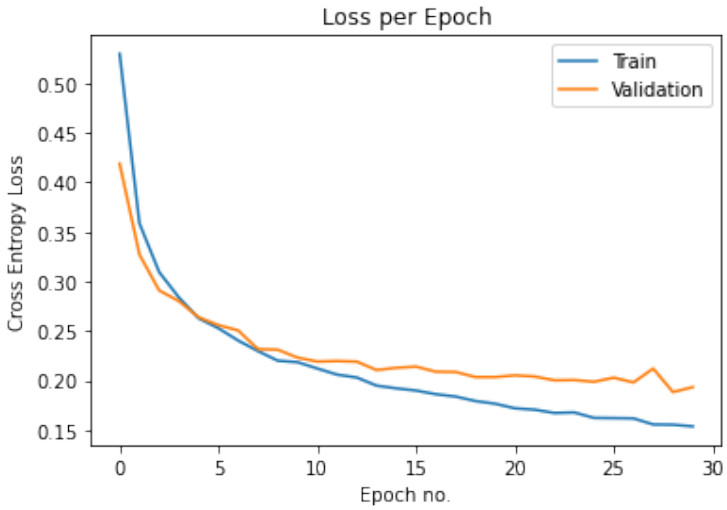


Figure 27: Loss for the training and validation set for the MNIST-1 model

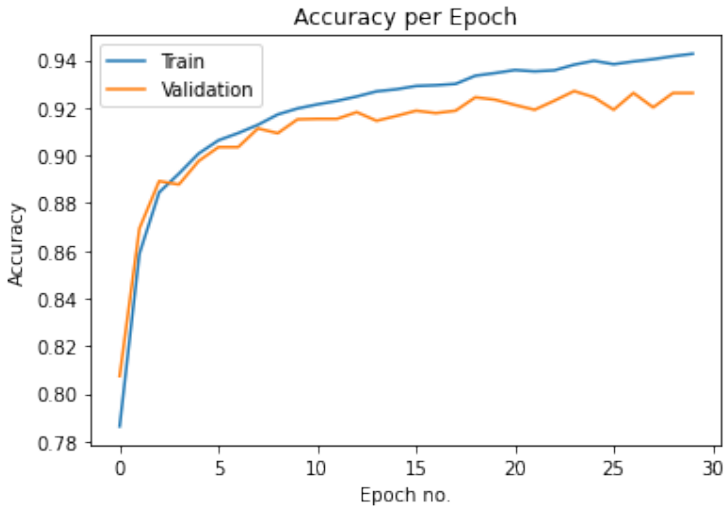


Figure 28: Accuracy for the training and validation set on the best MNIST-1 model

The Fashion-MNIST-1 training data was then split into sets of 5% to 100% in increments of 5% as follows:

```
# splitting the images and labels into 20 splits

def splitting_xs(dataset, i):
  ind = np.int(np.round(dataset.shape[0]/20))
  dataset_new = dataset[:(i+1)*ind, :]
  return dataset_new

def splitting_ys(dataset,i):
  ind = np.int(np.round(dataset.shape[0]/20))
  dataset_new = dataset[:(i+1)*ind]
  return dataset_new

xs_splits = []
ys_splits = []
```

```
for i in range(19):
  xs_splits.append(splitting_xs(fm1_trainx_tensor, i))
  ys_splits.append(splitting_ys(fm1_trainy_tensor, i))

xs_splits.append(fm1_trainx_tensor)
ys_splits.append(fm1_trainy_tensor)
len(xs_splits), len(ys_splits)

trainset_splits = []

for i in range(20):
    trainset_splits.append(torch.utils.data.TensorDataset(
       xs_splits[i], ys_splits[i]))

trainloader_splits = []
for i in range(20):
    trainloader_splits.append(torch.utils.data.DataLoader(
       trainset_splits[i], batch_size=BATCH_SIZE, shuffle=True
       ))


# parallelizing val CUDA
# Random_CNN_MNIST1, lr[0], optimizer[0], trainloader_splits[i
   ], fm1_valloader, fm1_testx_tensor, fm1_testy_tensor
def training_loop_CNN(model, lr, optimizer, trainloader,
   valloader, testx_tensor, testy_tensor):
    net = model
    LEARNING_RATE = lr
    EPOCHS = 20

    # picking cost function and optimizer
    criterion = nn.CrossEntropyLoss()
    if optimizer == 'Adam':
        optimizer = optim.Adam(net.parameters(), lr=
           LEARNING_RATE)
    if optimizer == 'SGD':
        optimizer = optim.SGD(net.parameters(), lr=
           LEARNING_RATE, momentum = 0.9)
    # optimizer = optim.Adam(net.parameters(), lr=LEARNING_RATE
       )

    # training the model:
    train_losses = []
    val_losses = []
    train_accuracies = []
    val_accuracies = []

    for epoch in range(EPOCHS):
        running_train_loss = 0.0
        running_train_acc = 0.0

        for i, data in enumerate(trainloader):
            # set param grads to 0
            optimizer.zero_grad()

            # get inputs
            images, labels = data
            images, labels = Variable(images), Variable(labels)
```

```
        images = images.unsqueeze(1).to(device)
        labels = labels.type(torch.LongTensor).to(device)

        #forward + backward + optimize
        output = net(images)

        train_loss = criterion(output, labels)

        train_loss.backward()
        optimizer.step()

        running_train_loss += train_loss.item()

        #num_of_train_batches += 1
        train_logits = net(images)
        _, train_predictions = torch.max(train_logits.data,
            1)
        train_correct_predictions = (train_predictions.int
           () == labels.int()).sum()
        train_length = labels.size()[0]
        running_train_acc += train_correct_predictions/
           train_length


    train_losses.append(running_train_loss / len(
       trainloader))
    train_accuracies.append(running_train_acc / len(
       trainloader))

    running_val_loss = 0.0
    running_val_acc = 0.0

    for i, data in enumerate(valloader):
      images, labels = data
      images, labels = Variable(images), Variable(labels)
      images = images.unsqueeze(1).to(device)
      labels = labels.type(torch.LongTensor).to(device)

      val_logits = net(images).to(device)
      val_loss = criterion(val_logits, labels)
      running_val_loss += val_loss.item()
      _, val_predictions = torch.max(val_logits.data, 1)
      val_correct_predictions = (val_predictions.int() ==
         labels.int()).sum()
      val_length = labels.size()[0]
      running_val_acc += val_correct_predictions/val_length


    val_losses.append(running_val_loss / len(valloader))
    val_accuracies.append(running_val_acc / len(valloader))


    if epoch % 5 == 0:
        print('Epoch: %d/%d,  train loss: %.6f train acc:
           %.6f  val loss: %.6f val acc: %.6f'%(epoch+1,
           EPOCHS,train_losses[epoch],train_accuracies[
           epoch],val_losses[epoch],val_accuracies[epoch])
           )
```

```
    images = testx_tensor.unsqueeze(1).to(device)
    labels = testy_tensor.type(torch.LongTensor).to(device)
    test_logits = net(images)
    test_loss = criterion(test_logits.to(device), labels)
    _, test_predictions = torch.max(test_logits.data, 1)
    test_correct_predictions = (test_predictions.int() ==
       labels.int()).sum()
    test_length = testy_tensor.size()[0]
    final_test_accuracy = test_correct_predictions.item()/
       test_length
    print('Test Accuracy: ',final_test_accuracy)

    return train_losses, train_accuracies, val_losses,
       val_accuracies, final_test_accuracy
```

Using Random weights:

```
# lists to store the val, train, test losses and accuracies
train_losses_rand = []
val_losses_rand = []
train_acc_rand = []
val_acc_rand = []
test_acc_rand = []

# Creating an object for Random CNN MNIST1
Random_CNN_MNIST1 = Random_MNIST1_CNN(dropouts[best_p]).to(
   device)
# Train the model for 5%, 10%....100% of data in a loop
for i in range(20):
    train_losses, train_accuracies, val_losses, val_accuracies,
        final_test_accuracy = training_loop_CNN(
       Random_CNN_MNIST1, lr[best_lr], optimizer[best_opt],
       trainloader_splits[i], fm1_valloader, fm1_testx_tensor,
        fm1_testy_tensor)
    train_losses_rand.append(np.min(train_losses))
    val_losses_rand.append(np.min(val_losses))
    train_acc_rand.append(train_accuracies[np.argmin(val_losses
       )])
    val_acc_rand.append(val_accuracies[np.argmin(val_losses)])
    test_acc_rand.append(final_test_accuracy)
```

Using pretrained weights:

```
# lists to store the val, train, test losses and accuracies for
    the Transfer Learning Model
train_losses_equiv = []
val_losses_equiv = []
train_acc_equiv = []
val_acc_equiv = []
test_acc_equiv = []

# Creating an object for Transfer Learning CNN MNIST1
CNN_MNIST1 = MNIST1_CNN(dropouts[best_p]).to(device)
# Train the model for 5%, 10%....100% of data in a loop
for i in range(20):
    print('For ', (i+1)*5,'% of the train data')
    train_losses, train_accuracies, val_losses, val_accuracies,
        final_test_accuracy = training_loop_CNN(CNN_MNIST1, lr
```

```
    [best_lr], optimizer[best_opt], trainloader_splits[i],
    fm1_valloader, fm1_testx_tensor, fm1_testy_tensor)
train_losses_equiv.append(np.min(train_losses))
val_losses_equiv.append(np.min(val_losses))
train_acc_equiv.append(train_accuracies[np.argmin(
    val_losses)])
val_acc_equiv.append(val_accuracies[np.argmin(val_losses)])
test_acc_equiv.append(final_test_accuracy)
```

Figure 29: Accuracy for the training and validation set on the final model

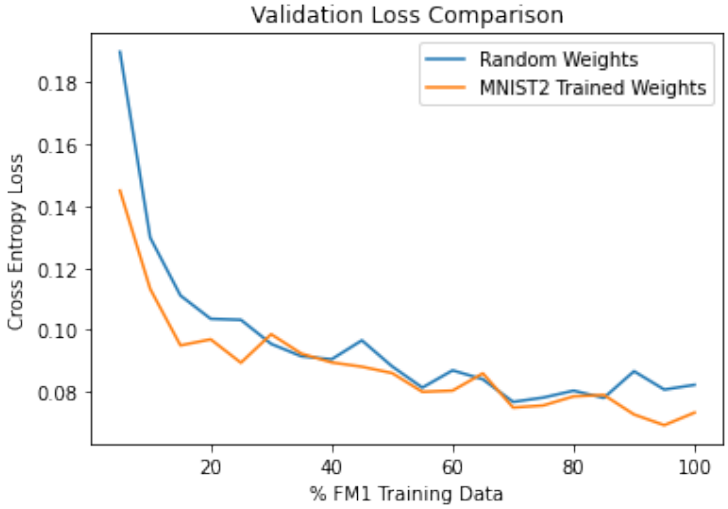


Figure 30: Accuracy for the training and validation set on the final model

Comparison of the two initialisation strategies: Given figures 29 and 30, it is evident that for smaller percentages of training data the model with pre-trained MNIST 2 weights have lower training and validation loss compared to the Random Weights model . This makes sense as the pre-trained weights are closer to the minimum (as they have been trained on other clothes) compared to random weights. As the percentage of train data increases, both models have similar performance and converge closely.

3.5.5 Provide the final accuracy on the training, validation, and test set for the best model you obtained for each of the two initialisation strategies

```
print("Best Random Weight Model => ", np.argmin(val_losses_rand
    ), "with min val loss = ",np.min(val_losses_rand))
print("Best % data", (np.argmin(val_losses_rand)+1)*5)
print("Validation Accuracy: ", val_acc_rand[np.argmin(
    val_losses_rand)].item()*100, "%")
print("Train Accuracy: ", train_acc_rand[np.argmin(
    val_losses_rand)].item()*100, "%")
print("Test Accuracy: ", test_acc_rand[np.argmin(
    val_losses_rand)]*100, "%")
```

```
Best Random Weight Model =>  19 with min val loss =
    0.08231660723686218
Best % data 100
Validation Accuracy:  98.24875593185425 %
Train Accuracy:  99.58545565605164 %
Test Accuracy:  98.18 %
```

```
print("Best Transfer Learning Model =>", np.argmin(
    val_losses_equiv), "with min val loss = ", np.min(
    val_losses_equiv))
print("Best % data", (np.argmin(val_losses_equiv)+1)*5)
print("Validation Accuracy: ", val_acc_equiv[np.argmin(
    val_losses_equiv)].item()*100, "%")
print("Train Accuracy: ", train_acc_equiv[np.argmin(
    val_losses_equiv)].item()*100, "%")
print("Test Accuracy: ", test_acc_equiv[np.argmin(
    val_losses_equiv)]*100, "%")
```

For the best CNN initialised with random weights, the final accuracies on the training, validation and test sets are:

- Training Accuracy: 99.69%
- Validation Accuracy: 98.44%
- Test Accuracy: 98.36%

For the best CNN initialised with pretrained weights, the final accuracies on the training, validation and test sets are:

- Training Accuracy: 99.84%
- Validation Accuracy: 98.54%
- Test Accuracy: 98.36%

References