COMP0090 Intro to Deep Learning
Assignment 3 - Team RealDeep

Toby Drane
ucabtdr@ucl.ac.uk

Ravi Patel
ucabrp1@ucl.ac.uk

Udi Ibgui
zcemyib@ucl.ac.uk

Louis Prosser
ucabljp@ucl.ac.uk

Daniel May
ucabdd3@ucl.ac.uk

January 15, 2020

Python programming language was used for all code related to this assignment. The full code for each question can be found in the Appendices. All models were implemented using PyTorch machine learning framework.

- Udi Ibgui took primary responsibility for question 5.
- Toby Drane took primary responsibility for question 4.
- Ravi Patel took primary responsibility for data preparation and question 1.
- Louis Prosser took primary responsibility for question 3.
- Daniel May took primary responsibility for question 2.

Contents

0 Data Preparation

The code used to prepare the Fashion-MNIST dataset for the questions in this assignment is shown in Appendix A.

We prepared the dataset by splitting the full Fashion-MNIST data into two separate datasets, fm1 and fm2. fm1 included all images corresponding to 0, 1, 4, 5, and 8 class indices from Table 1. fm2 included all images corresponding to 2, 3, 6, 7, and 9 class indices from Table 1. With five classes in each of fm1 and fm2, there was no overlap between classes in the two datasets.

0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Table 1: Fashion-MNIST class indices

For both fm1 and fm2, the images were split into training, validation, and test datasets containing 25000, 5000, and 5000 images respectively. All datasets were fully class balanced. We used PyTorch's DataLoader to batchify and shuffle the data during training.

1 Q1

The data preparation for this question is shown in Appendix A. The remaining full code for this question is shown in Appendix B, with detailed comments. We show only a few relevant snippets as we go along in this current section.

1.1 Model implementation

We implemented a multi-class convolutional neural network with cross-entropy loss for Fashion-MNIST-1 data. We used stochastic gradient descent as our optimiser, and other hyperparameters included a learning rate of 0.005, no weight decay, relu activation function between layers, and a batch size of 100 with shuffling between epochs. Our architecture consisted of two convolutional layers and two dense layers. We also incorporated max-pooling between the convolutional layers. The final dense layer output was passed through a softmax function to yield a output vector of length five, with each element corresponding to a probability for each class. The predicted class was chosen as the argmax of this vector.

The code specifying the hyperparameters and PyTorch model class for this described model implementation is shown here:

```
# hyperparameter grid
hyperparam_grid = {'num_dense_layers': [2],
                   'num_conv_layers': [2],
                   'learning_rate': [0.005],
                   'weight_decay': [0], # Regularisation
                   'optimiser': ["SGD"],
                   'batch_size': [train_batch_size]}


# PyTorch model class, which takes the specified hyperparameters as inputs

class CNN(nn.Module):

    def __init__(self, num_classes, num_dense_layers, num_conv_layers):
      super(CNN, self).__init__()
      self.num_dense_layers = num_dense_layers
      self.num_conv_layers = num_conv_layers

      # single conv layer
      if num_conv_layers == 1:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
          self.fc1 = nn.Linear(6*(12**2), num_classes)
        elif num_dense_layers == 2:
          self.fc1 = nn.Linear(6*(12**2), 80)
          self.fc2 = nn.Linear(6, num_classes)
        elif num_dense_layers == 3:
          self.fc1 = nn.Linear(6*(12**2), 120)
          self.fc2 = nn.Linear(120, 84)
          self.fc3 = nn.Linear(20, num_classes)

      # two conv layers
      elif num_conv_layers == 2:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
```

```
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
          self.fc1 = nn.Linear(12*(4**2), num_classes)
        elif num_dense_layers == 2:
          self.fc1 = nn.Linear(12*(4**2), 80)
          self.fc2 = nn.Linear(80, num_classes)
        elif num_dense_layers == 3:
          self.fc1 = nn.Linear(12*(4**2), 120)
          self.fc2 = nn.Linear(120, 84)
          self.fc3 = nn.Linear(20, num_classes)

    def forward(self, x):

      #conv2d and maxpools
      if self.num_conv_layers == 1:
        x = self.pool1(F.relu(self.conv1(x)))
        x = x.view(-1, 6*(12**2))
      elif self.num_conv_layers == 2:
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 12*(4**2))

      #fully-connnected layers
      if self.num_dense_layers == 1:
        x = self.fc1(x)
      if self.num_dense_layers == 2:
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
      if self.num_dense_layers == 3:
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

      return x
```

1.2 Final training

The model was trained till convergence of the loss on the validation dataset. The full training loop code is as shown in Appendix B. We implemented early stopping on detection of convergence. Our criterion for early stopping was a mean validation loss from the most recent 20 epochs being greater than the mean validation loss for the preceding 20 epochs plus 0.001. This averaging in the early stopping metric helped to prevent premature early stopping due to some expected choppiness in the validation loss between single epochs. The training had to precede for a minimum of 20 epochs before early stopping could be triggered. Stochastic gradient descent optimiser was used.

The early-stopping implementation is as shown here:

```
## Early stopping criteria
if epochs_counter >= min_epochs:
    if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
        converged = True
        print("Model Converged")
```

1.3 Loss plot

The training and validation loss plot for our best model are shown in Figure 1. We also show our training and validation accuracy plot in Figure 2 respectively. Early stopping criteria used were as described in the preceding

subsection.

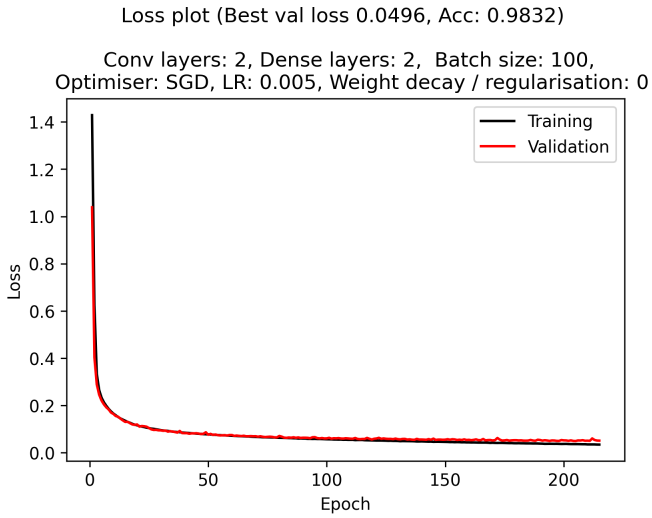


Figure 1: Training and validation loss by epoch for the CNN multiclass classification model

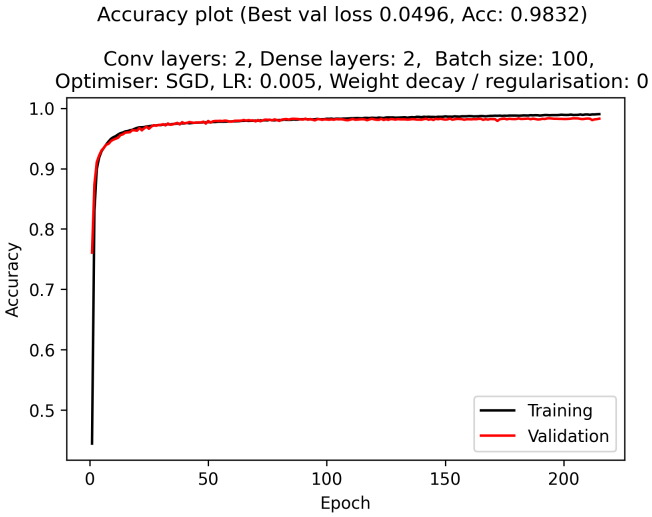


Figure 2: Training and validation accuracy by epoch for the CNN multiclass classification model

1.4 Final accuracy

As shown in Table 2, our model achieved a highest accuracy of 0.9832 on the validation dataset after 205 epochs of training, at predicting the the correct item out of five options for the Fashion-MNIST-1 data. This epoch was associated with an accuracy of 0.9890 on the training dataset. Final evaluation of this model was performed on the test dataset, achieving an accuracy of 0.9816.

1.5 Confusion matrix

Here we show a confusion matrix for the test set results. The test set consists of 1000 examples from each class. The most promiment source of errors involved the model predicting t-shirts/tops as either coats or bags. This is in keeping with the greater similarity in shapes and appearance of these items, with likely greater overlap between

Dataset	Accuracy
Training	0.9890
Validation	0.9832
Test	0.9816

Table 2: The training, validation, and test accuracy at the best epoch (205) for the CNN multiclass classification model

features, when compared with the alternative items of sandals and trousers. Certain t-shirts/tops in the dataset look relatively similar to coats and bags in shape.

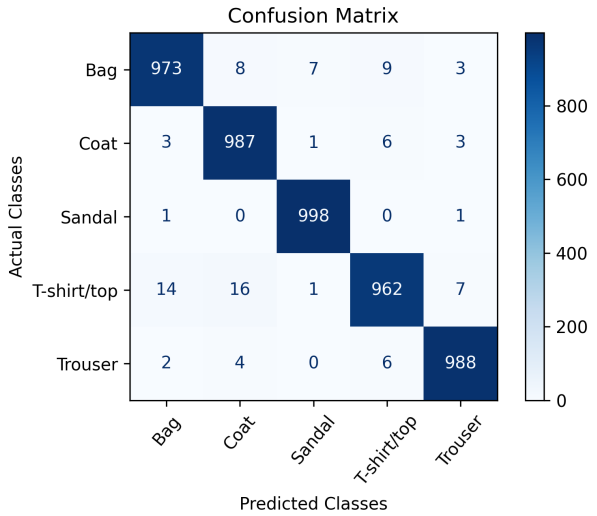


Figure 3: Confusion matrix for the CNN multiclass classification model

2 Q2

The data preparation for this question is shown in Appendix A. The remaining full code for this question is shown in Appendix C, with detailed comments. We show only a few relevant snippets as we go along in this current section.

2.1 Model implementation

We implemented a multi-class convolutional neural network with cross-entropy loss for Fashion-MNIST-1 data, exploring variants to find those which were the best performing, based on the validation losses.

In our experiments, we varied the numbers of convolutional layers and dense layers, the optimiser, learning rate, and the amount of regularization. Throughout, we used relu activation functions between layers, a batch size of 100 with shuffling between epochs, and also incorporated max-pooling between the convolutional layers. The final dense layer output was passed through a softmax function to yield an output vector of length five, with each element corresponding to a probability for each class. The predicted class was chosen as the argmax of this vector.

The code specifying the hyperparameters and PyTorch model class for this described model implementation is shown here:

```
# Hyperparameters (adjust each iteration)
hyperparams = {'num_dense_layers': 2,
               'num_conv_layers': 2,
               'learning_rate': 0.005,
               'weight_decay': 0, # Regularisation
               'optimiser': "SGD",
               'batch_size': train_batch_size}

# PyTorch model class, which takes the specified hyperparameters as inputs
class CNN(nn.Module):

  def __init__(self, num_classes, num_dense_layers, num_conv_layers):
    super(CNN, self).__init__()
    self.num_dense_layers = num_dense_layers
    self.num_conv_layers = num_conv_layers

    # single conv layer
    if num_conv_layers == 1:
      self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
      self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

      # dense layers
      if num_dense_layers == 1:
        self.fc1 = nn.Linear(6*(12**2), num_classes)
      elif num_dense_layers == 2:
        self.fc1 = nn.Linear(6*(12**2), 80)
        self.fc2 = nn.Linear(80, num_classes)
      elif num_dense_layers == 3:
        self.fc1 = nn.Linear(6*(12**2), 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)

    # two conv layers
    elif num_conv_layers == 2:
      self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
      self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
      self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
      self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
```

```
      # dense layers
      if num_dense_layers == 1:
        self.fc1 = nn.Linear(12*(4**2), num_classes)
      elif num_dense_layers == 2:
        self.fc1 = nn.Linear(12*(4**2), 80)
        self.fc2 = nn.Linear(80, num_classes)
      elif num_dense_layers == 3:
        self.fc1 = nn.Linear(12*(4**2), 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)

  def forward(self, x):

    # conv2d and maxpools
    if self.num_conv_layers == 1:
      x = self.pool1(F.relu(self.conv1(x)))
      x = x.view(-1, 6*(12**2))
    elif self.num_conv_layers == 2:
      x = self.pool1(F.relu(self.conv1(x)))
      x = self.pool2(F.relu(self.conv2(x)))
      x = x.view(-1, 12*(4**2))

    # fully-connnected layers
    if self.num_dense_layers == 1:
      x = self.fc1(x)
    if self.num_dense_layers == 2:
      x = F.relu(self.fc1(x))
      x = self.fc2(x)
    if self.num_dense_layers == 3:
      x = F.relu(self.fc1(x))
      x = F.relu(self.fc2(x))
      x = self.fc3(x)

    return x
```

We trained each model till convergence of the loss on the validation dataset. The full training loop code is as shown in Appendix C. We implemented early stopping on detection of convergence. Our criterion for early stopping was a mean validation loss from the most recent 20 epochs being greater than the mean validation loss for the preceding 20 epochs plus 0.001. This averaging in the early stopping metric helped to prevent premature early stopping due to some expected choppiness in the validation loss between single epochs. The training had to proceed for a minimum of 20 epochs before early stopping could be triggered.

The early-stopping implementation is as shown here:

```
## Early stopping criteria
if epochs_counter >= min_epochs:
    if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
        converged = True
        print("Model Converged")
```

2.2 Hyperparameter optimisation

We iteratively modified our model hyperparameters in order to find a variant which had minimal validation loss, as shown in Table 3. We varied the number of convolution layers (1 and 2), number of dense layers (1, 2, and 3), the optimiser (SGD, Adam, and AdamW), learning rate, and the amount of regularization. Figure 4 plots the training and validation loss and accuracy by epoch for each of the models.

To implement regularization in PyTorch, we use the weight decay parameters of the SGD optimiser and the AdamW variant of Adam, which act as L2 penalties.

Overall, our results show that the models performed quite comparably, with a few noticeable patterns.

As shown in the table, Model 2 achieved the highest accuracy of 0.9880 on the validation dataset, with an associated accuracy of 0.9949 on the training set. Perhaps surprisingly, this model had only a single convolution layer, and two dense layers, but performed better than more complex models. However, when evaluated on the test dataset, it achieved an accuracy of 0.9856, which was the 3$^{\text{rd}}$ highest.

Model 4 was similar to Model 2, with an additional convolution layer, and achieved the lowest loss of 0.0475 on the validation set, with an associated accuracy of 0.9874. However, when evaluated on the test dataset, it achieved an accuracy of 0.9838, which was the 5$^{\text{th}}$ highest.

Model 7 achieved the highest accuracy of 0.9886 when evaluated on the test dataset, and used the same hyperparameters as Model 2, but with the Adam optimiser rather than SGD. As shown in Figure 4, the Adam optimiser was significantly quicker to train till convergence, with Model 7 achieving its lowest validation loss after training for 6 epochs, while Model 2 took 240 epochs.

In our experiments, we found that using L2 regularization did not seem to improve the performance of our models. For example, Model 4 and Model 6 used the same hyperparameters, except for respective L2 penalties of 0 and 0.01, and achieved accuracies of 0.9838 and 0.9772 when evaluated on the test dataset.

The worst performing model on the validation dataset was Model 3, which used a stochastic gradient descent optimiser with the lowest learning rate of 0.0005, and did not reach our early stopping criterion before it stopped training at the maximum number of epochs (300), achieving a best validation loss of 0.0834 by this point. However, it seems probable that its performance may have continued to improve if trained for a greater number of epochs.

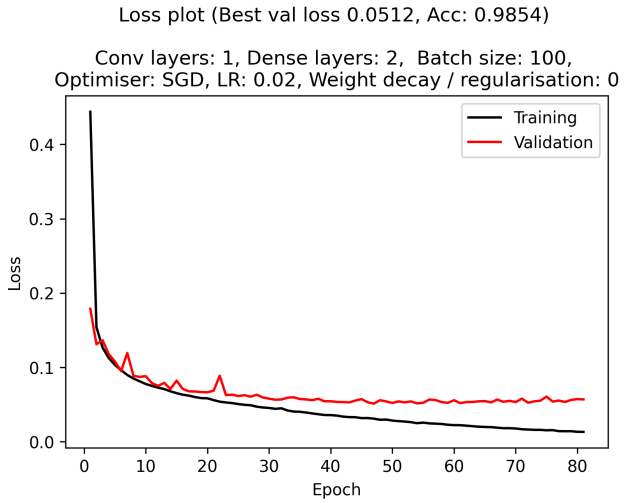
2.3 Lowest loss

Model 4 achieved the lowest validation loss of 0.0475, while the Model 1 and Model 2 had the joint lowest test loss of 0.0548.

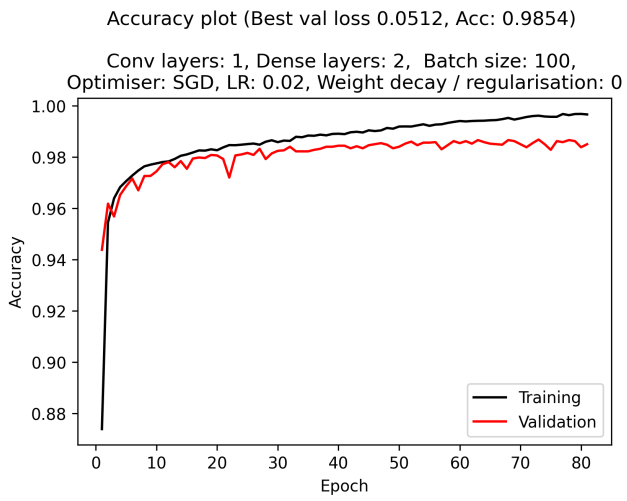
As the classes are balanced within each dataset, we might have expected the models to have improved validation performance with each iteration as we hand picked hyperparameters to try to improve validation performance, but without necessarily an associated improvement in test performance. This would represent overfitting to the validation dataset. However, we did not find this trend of improving validation performance, and found it hard to intuit hyperparameter changes to generate clear improvements in validation performance. This likely reflects that in this setting, a better approach would be to perform a more exhaustive search of the hyperparameter space, for example using grid search or another search approach.

	Layers					Loss			Accuracy		
Model	Convolution	Classification	Optimiser	L. rate	Regularisation	Train	Valid.	Test	Train	Valid.	Test
1	1	2	SGD	0.02	None	0.0310	0.0512	0.0548	0.9904	0.9854	0.9844
2	1	2	SGD	0.005	None	0.0184	0.0485	0.0548	0.9949	0.9880	0.9856
3	1	2	SGD	0.0005	None	0.0664	0.0834	0.0765	0.9810	0.0834	0.9772
4	2	2	SGD	0.005	None	0.0249	0.0475	0.0606	0.9838	0.9874	0.9838
5	2	3	SGD	0.005	None	0.0324	0.0506	0.0795	0.9904	0.9860	0.9802
6	2	2	SGD	0.005	0.01	0.0711	0.0737	0.0816	0.9798	0.9808	0.9772
7	1	2	Adam	0.005	None	0.0279	0.0496	0.0666	0.9912	0.9848	0.9886
8	1	2	AdamW	0.005	0.01	0.0152	0.0579	0.0828	0.9951	0.9858	0.9872
9	1	2	AdamW	0.005	0.05	0.0128	0.0497	0.0814	0.9960	0.9868	0.9822
10	2	2	AdamW	0.005	0.01	0.0251	0.0641	0.1048	0.9916	0.9830	0.9822

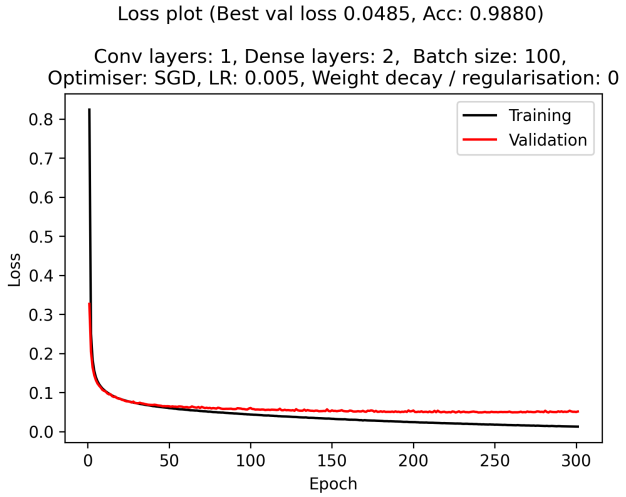
Table 3: The training, validation, and test loss and accuracy for ten iterations of hyperparameter changes



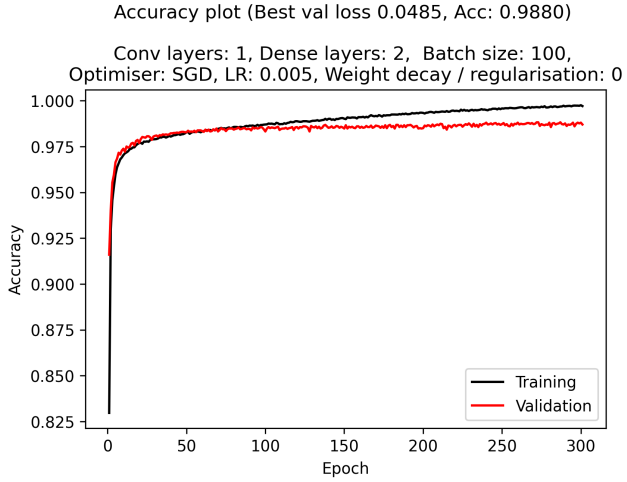
(a) Model 1: training and validation loss by epoch



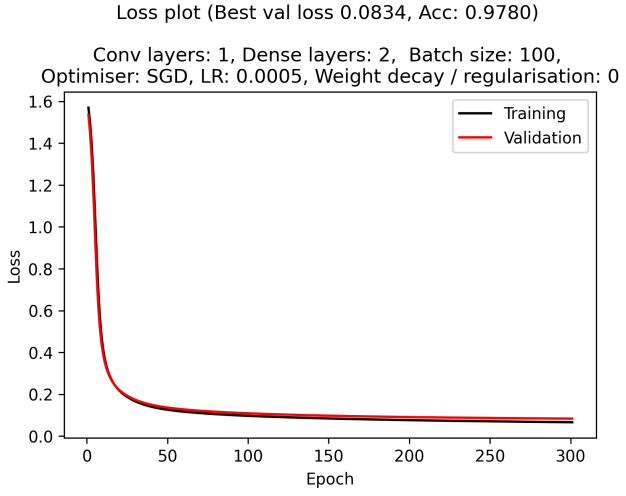
(b) Model 1: training and validation accuracy by epoch



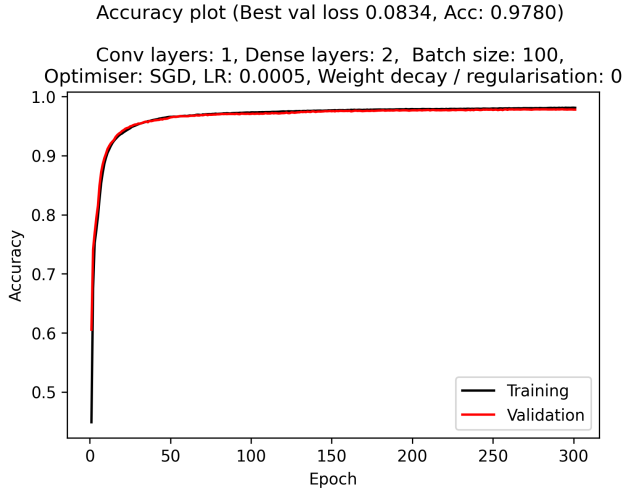
(c) Model 2: training and validation loss by epoch



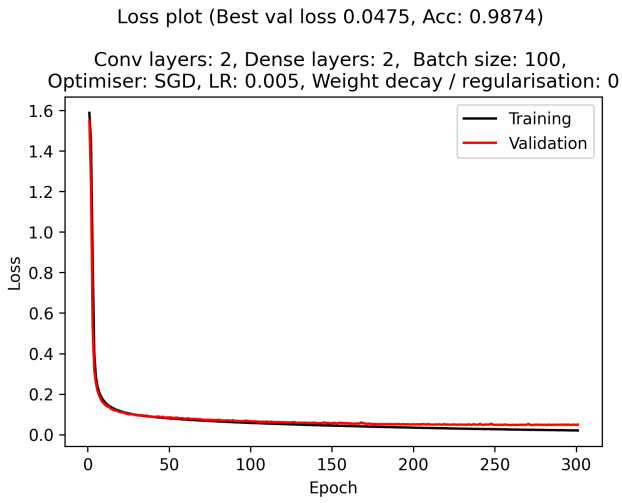
(d) Model 2: training and validation accuracy by epoch



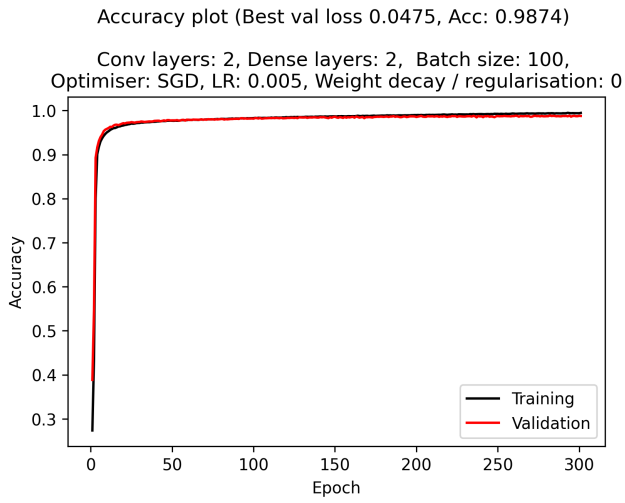
(e) Model 3: training and validation loss by epoch



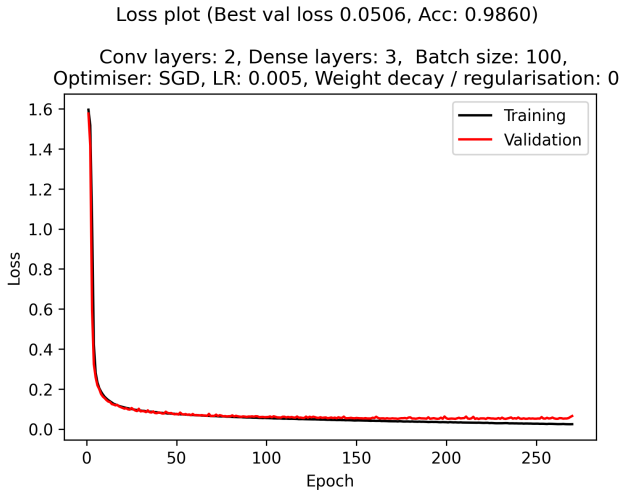
(f) Model 3: training and validation accuracy by epoch



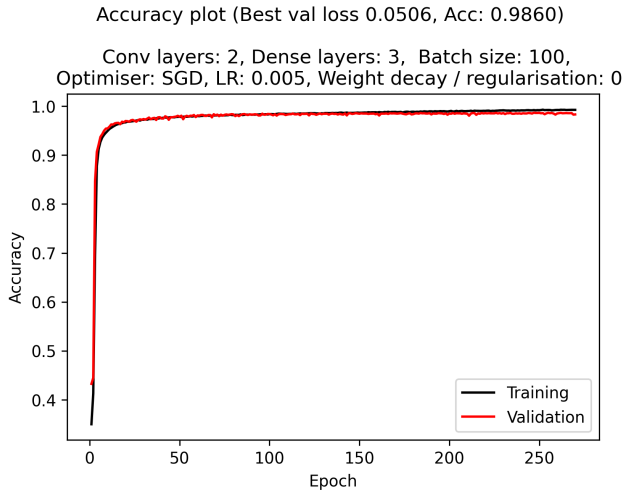
(g) Model 4: training and validation loss by epoch



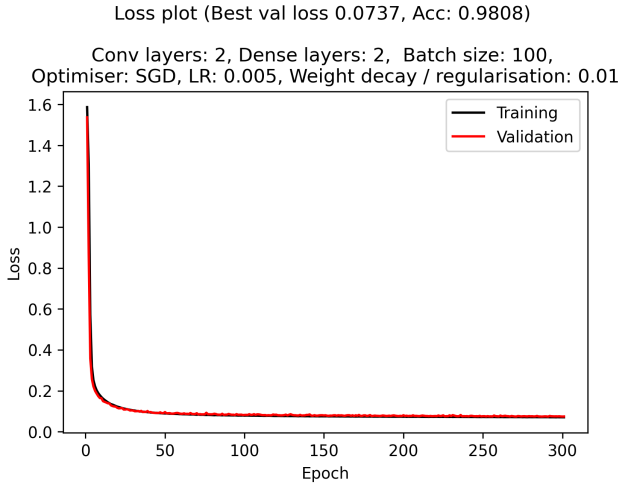
(h) Model 4: training and validation accuracy by epoch



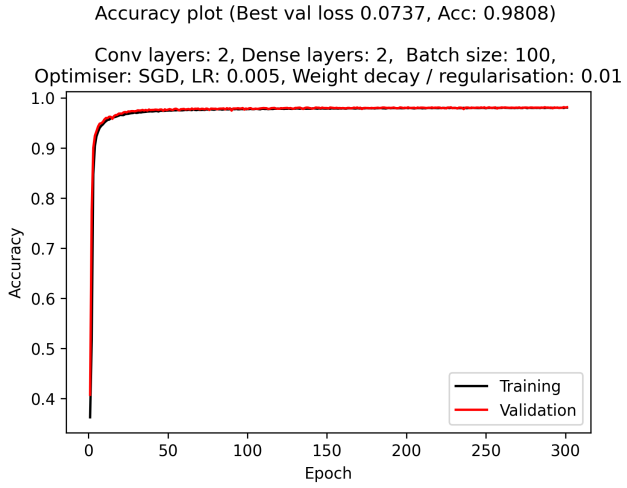
(i) Model 5: training and validation loss by epoch



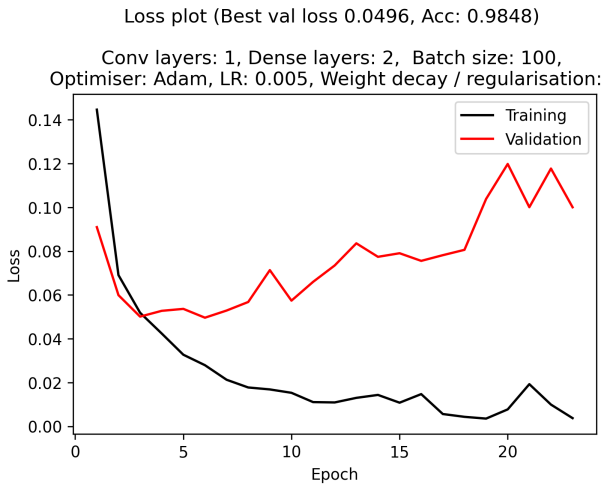
(j) Model 5: training and validation accuracy by epoch



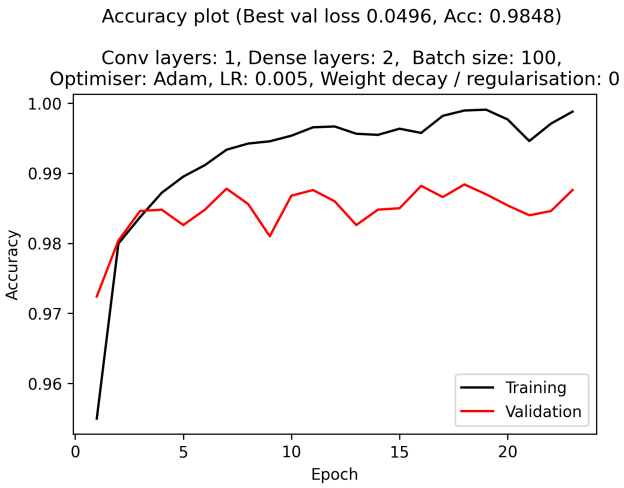
(k) Model 6: training and validation loss by epoch



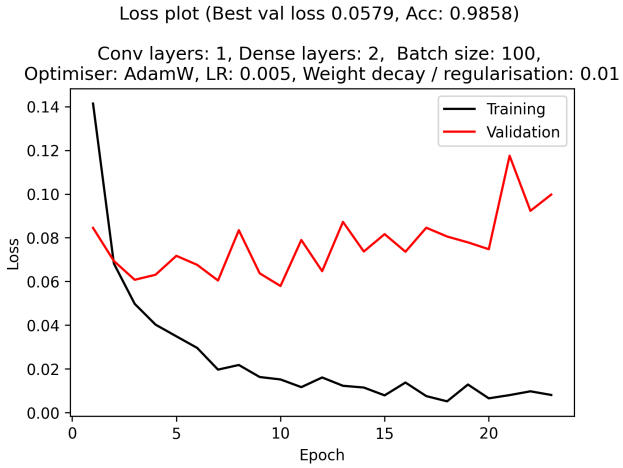
(l) Model 6: training and validation accuracy by epoch



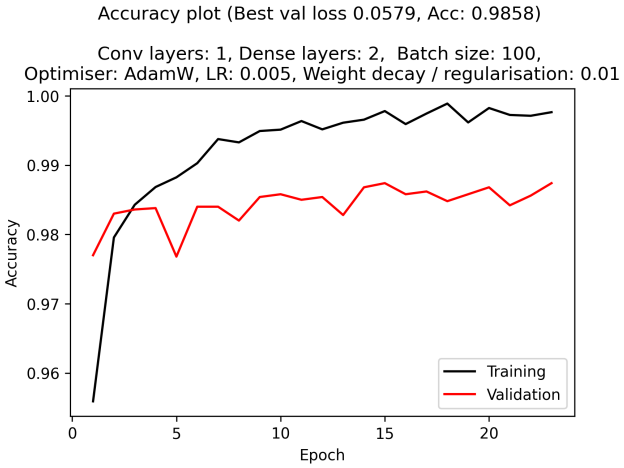
(m) Model 7: training and validation loss by epoch



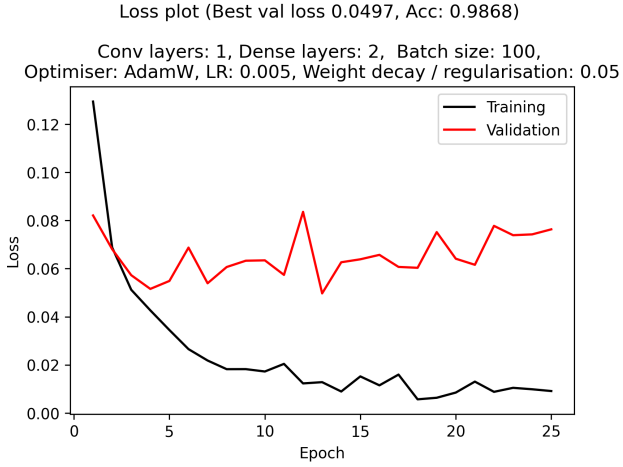
(n) Model 7: training and validation accuracy by epoch



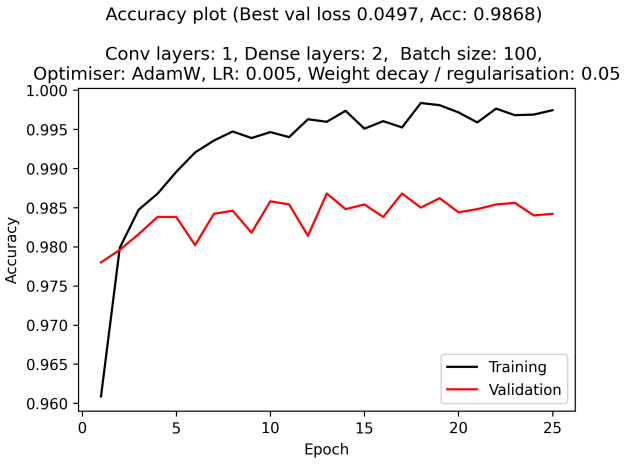
(o) Model 8: training and validation loss by epoch



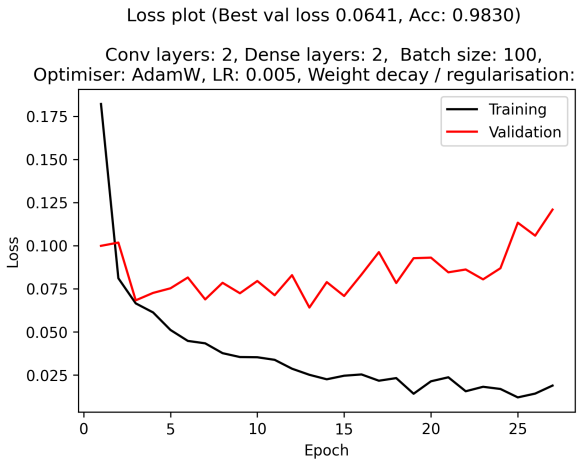
(p) Model 8: training and validation accuracy by epoch



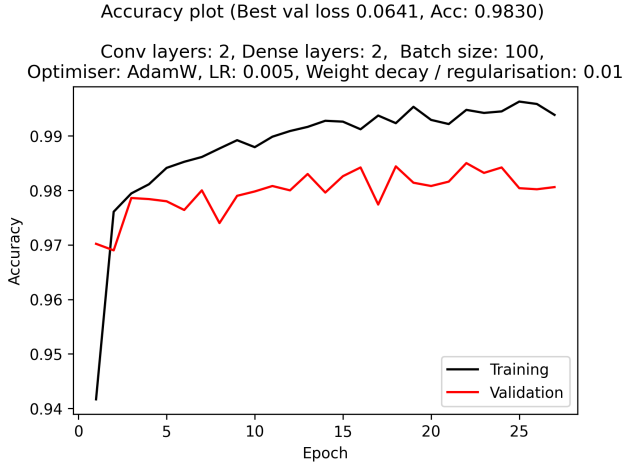
(q) Model 9: training and validation loss by epoch



(r) Model 9: training and validation accuracy by epoch



(s) Model 10: training and validation loss by epoch



(t) Model 10: training and validation accuracy by epoch

Figure 4: The training and validation loss and accuracy by epoch for our ten iterations of hyperparameter changes

3 Q3

The data preparation for this question is shown in Appendix A. The remaining full code for this question is shown in Appendix D, with detailed comments. We show only a few relevant snippets as we go along in this current section.

3.1 Model implementation

Similarly to in the previous questions, we implemented a multi-class convolutional neural network with cross-entropy loss for Fashion-MNIST-1 data. We used stochastic gradient descent as our optimiser, and other hyperparameters included a learning rate of 0.005, no weight decay, relu activation function between layers, and a batch size of 100 with shuffling between epochs. For this question, we chose to use 2 dense layers but with 4 convolutional layers (instead of 2) in order to hopefully visualise more clearly the progression of the feature maps as the images are passed through each layer. In order to offset the reduction in output size that comes with using more convolutional layers, we use zero-padding of size 2 in each convolutional layer. We incorporated max-pooling after every 2 convolutional layers and the final dense layer output was passed through a softmax function to yield a output vector of length five, with each element corresponding to a probability for each class. The predicted class was chosen as the argmax of this vector.

The code specifying the hyperparameters and PyTorch model class for this described model implementation is shown here:

```
# hyperparameter grid
hyperparam_grid = {'num_dense_layers': [2],
                   'num_conv_layers': [4],
                   'learning_rate': [0.005],
                   'weight_decay': [0], # Regularisation
                   'optimiser': ["SGD"],
                   'batch_size': [train_batch_size]}

# PyTorch model class, which takes the specified hyperparameters as inputs
class CNN(nn.Module):

    def __init__(self, num_classes, num_dense_layers, num_conv_layers):
      super(CNN, self).__init__()
      self.num_dense_layers = num_dense_layers
      self.num_conv_layers = num_conv_layers

      # single conv layer
      if num_conv_layers == 1:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
          self.fc1 = nn.Linear(6*(12**2), num_classes)
        elif num_dense_layers == 2:
          self.fc1 = nn.Linear(6*(12**2), 80)
          self.fc2 = nn.Linear(6, num_classes)
        elif num_dense_layers == 3:
          self.fc1 = nn.Linear(6*(12**2), 120)
          self.fc2 = nn.Linear(120, 84)
          self.fc3 = nn.Linear(20, num_classes)

      # two conv layers
      elif num_conv_layers == 2:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
```

```
      self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
      self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

      # dense layers
      if num_dense_layers == 1:
        self.fc1 = nn.Linear(12*(4**2), num_classes)
      elif num_dense_layers == 2:
        self.fc1 = nn.Linear(12*(4**2), 80)
        self.fc2 = nn.Linear(80, num_classes)
      elif num_dense_layers == 3:
        self.fc1 = nn.Linear(12*(4**2), 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(20, num_classes)

    # four conv layers
    elif num_conv_layers == 4:
      self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2)
      self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5, padding=2)
      self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
      self.conv3 = nn.Conv2d(in_channels=12, out_channels=16, kernel_size=5,padding=2)
      self.conv4 = nn.Conv2d(in_channels=16, out_channels=24, kernel_size=5,padding=2)
      self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

      # dense layers
      if num_dense_layers == 1:
        self.fc1 = nn.Linear(24*(7**2), num_classes)
      elif num_dense_layers == 2:
        self.fc1 = nn.Linear(24*(7**2), 80)
        self.fc2 = nn.Linear(80, num_classes)
      elif num_dense_layers == 3:
        self.fc1 = nn.Linear(24*(7**2), 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(20, num_classes)

  def forward(self, x):

    #conv2d and maxpools
    if self.num_conv_layers == 1:
      x = self.pool1(F.relu(self.conv1(x)))
      x = x.view(-1, 6*(12**2))
    elif self.num_conv_layers == 2:
      x = self.pool1(F.relu(self.conv1(x)))
      x = self.pool2(F.relu(self.conv2(x)))
      x = x.view(-1, 12*(4**2))

    elif self.num_conv_layers == 4:
      x = F.relu(self.conv1(x))
      x = F.relu(self.conv2(x))
      x = self.pool1(x)
      x = F.relu(self.conv3(x))
      x = F.relu(self.conv4(x))
      x = self.pool2(x)
      x = x.view(-1, 24*(7**2))

    #fully-connnected layers
    if self.num_dense_layers == 1:
```

```
        x = self.fc1(x)
      if self.num_dense_layers == 2:
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
      if self.num_dense_layers == 3:
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

      return x
```

3.2 Final training

The model was trained till convergence of the loss on the validation dataset. The full training loop code is as shown in Appendix D. We implemented early stopping on detection of convergence. Our criterion for early stopping was a mean validation loss from the most recent 20 epochs being greater than the mean validation loss for the preceding 20 epochs plus 0.001. This averaging in the early stopping metric helped to prevent premature early stopping due to some expected choppiness in the validation loss between single epochs. The training had to proceed for a minimum of 20 epochs before early stopping could be triggered. Stochastic gradient descent optimiser was used.

The early-stopping implementation is as shown here:

```
## Early stopping criteria
if epochs_counter >= min_epochs:
    if (np.mean(dev_loss_list[-10:]) - np.mean(dev_loss_list[-20:-10])) > 0.001:
      converged = True
      print("Model Converged")
```

The training and validation loss plot for our best model are shown in Figure 5. We also show our training and validation accuracy plot in Figure 6 respectively.

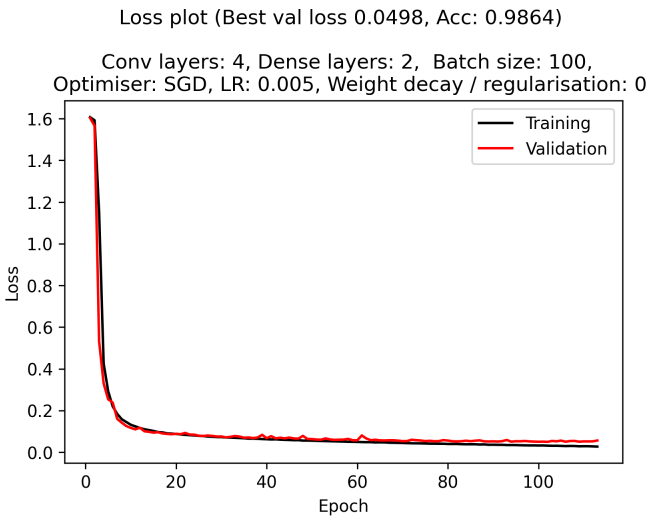


Figure 5: Training and validation loss by epoch for the CNN multiclass classification model

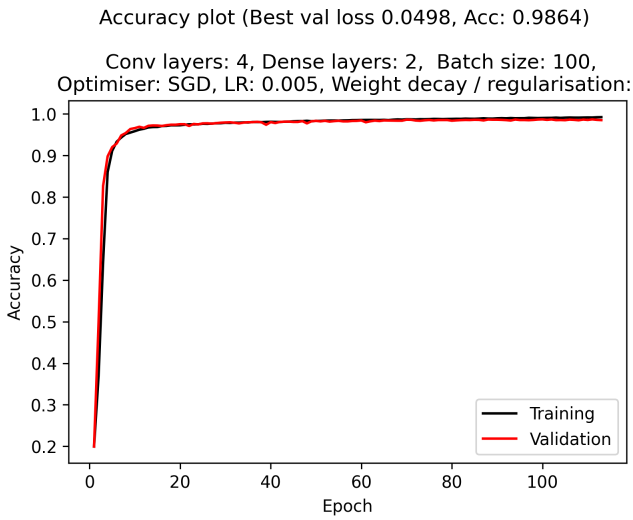


Figure 6: Training and validation accuracy by epoch for the CNN multiclass classification model

3.3 Filters and feature maps

First, we visualise the filters rather than the feature maps for specific inputs. In order to do so, we load our best model and inspect its structure:

```
best_model =
↪  torch.load(f"{louis_output_path}4_conv_layer_pad_MODEL1_0.0498_devloss_0.9864_devacc.pth",
↪  map_location=torch.device('cpu'))
best_model.eval()
```

```
CNN(
    (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (conv2): Conv2d(6, 12, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (conv3): Conv2d(12, 16, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (conv4): Conv2d(16, 24, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (fc1): Linear(in_features=1176, out_features=80, bias=True)
    (fc2): Linear(in_features=80, out_features=5, bias=True)
)
```

In the first convolutional layer, there is only 1 input channel and 6 5×5 filters to visualise. However, as the number of channels increases, the number of filters significantly increases. For the second convolutional layer there are $6 * 12 = 72$ filters. For the third there are $12 * 16 = 192$ and for the fourth there are $16 * 24 = 384$. Therefore, for the sake of convenience, we choose to visualise all the filters for the first and second convolutional layers but only 3 filters per channel for the third convolutional layer and 2 filters per channel for the final convolutional layer.

In order to plot the filters, we must access the weights of the convolutional layers:

```
# Save weights in convolutional layers to plot filters
filter_weights = []
for i in range(num_layers):
  if type(layers[i]) == nn.Conv2d:
      filter_weights.append(layers[i].weight)
```

Then, we simply plot these weights for each convolutional layer. The code below was used to plot the filters for the final layer and was adjusted accordingly to plot the filters for the previous 3 layers:

```
### Visualise the filters ###

i = 0
filters_per_channel = 2
plt.figure(figsize=(20, 20))
for filter in filter_weights[3]:
    for j in range(filters_per_channel):
      i += 1
      plt.subplot(8, 6, i)
      plt.imshow(filter[j, :, :].detach().cpu(), cmap='gray')
      plt.axis('off')
plt.savefig(f"{output_path}filters layer 4", dpi=300, bbox_inches = "tight")
plt.show()
```

In the plots, the lighter patches represent large weights and the dark patches lower weights. Hence, when these filters are applied to the input images, the lighter areas represent areas of greater activation and patterns corresponding to these areas are detected. The visualised filters for convolutional layers 1, 2, 3 and 4 can be found in Figures 7, 8, 9 and 10 respectively.

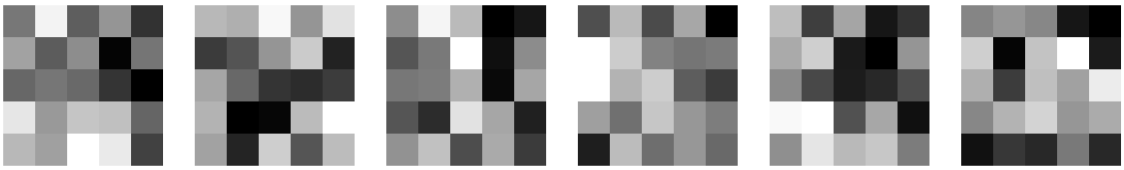


Figure 7: Filters for convolutional layer 1

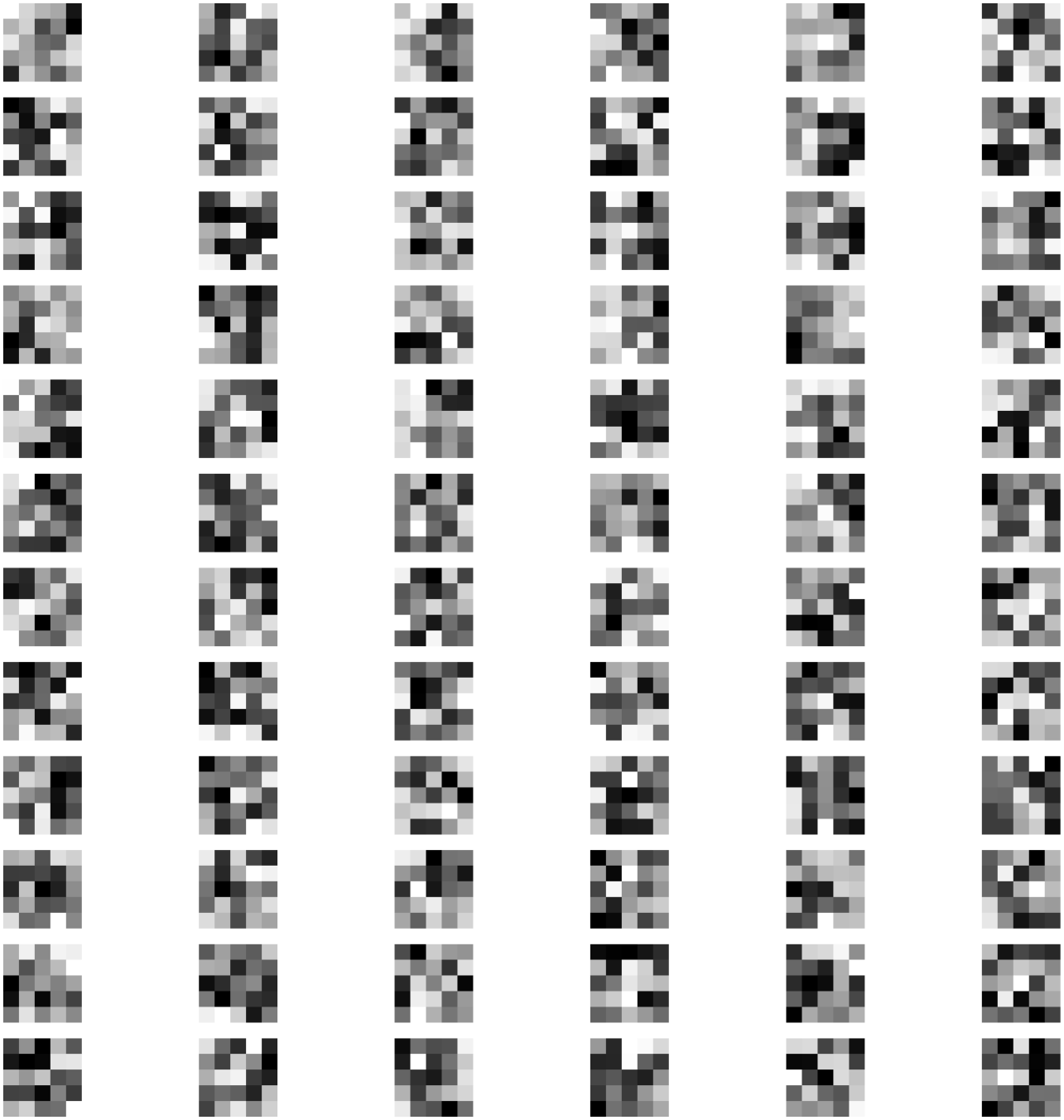


Figure 8: Filters for convolutional layer 2 - each column corresponds to one channel (12 filters for each channel)

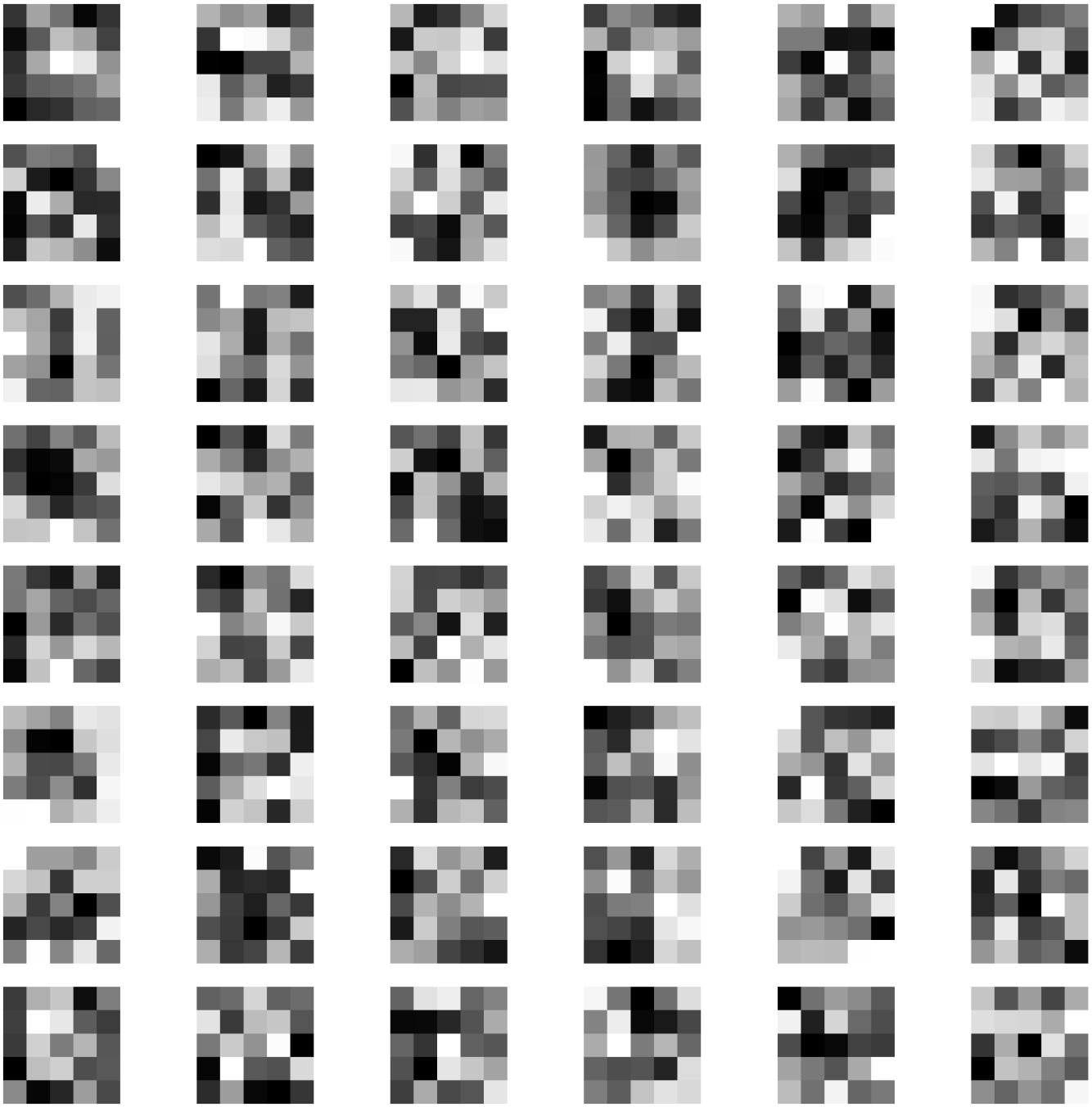


Figure 9: Filters for convolutional layer 3 (3 filters per channel displayed)

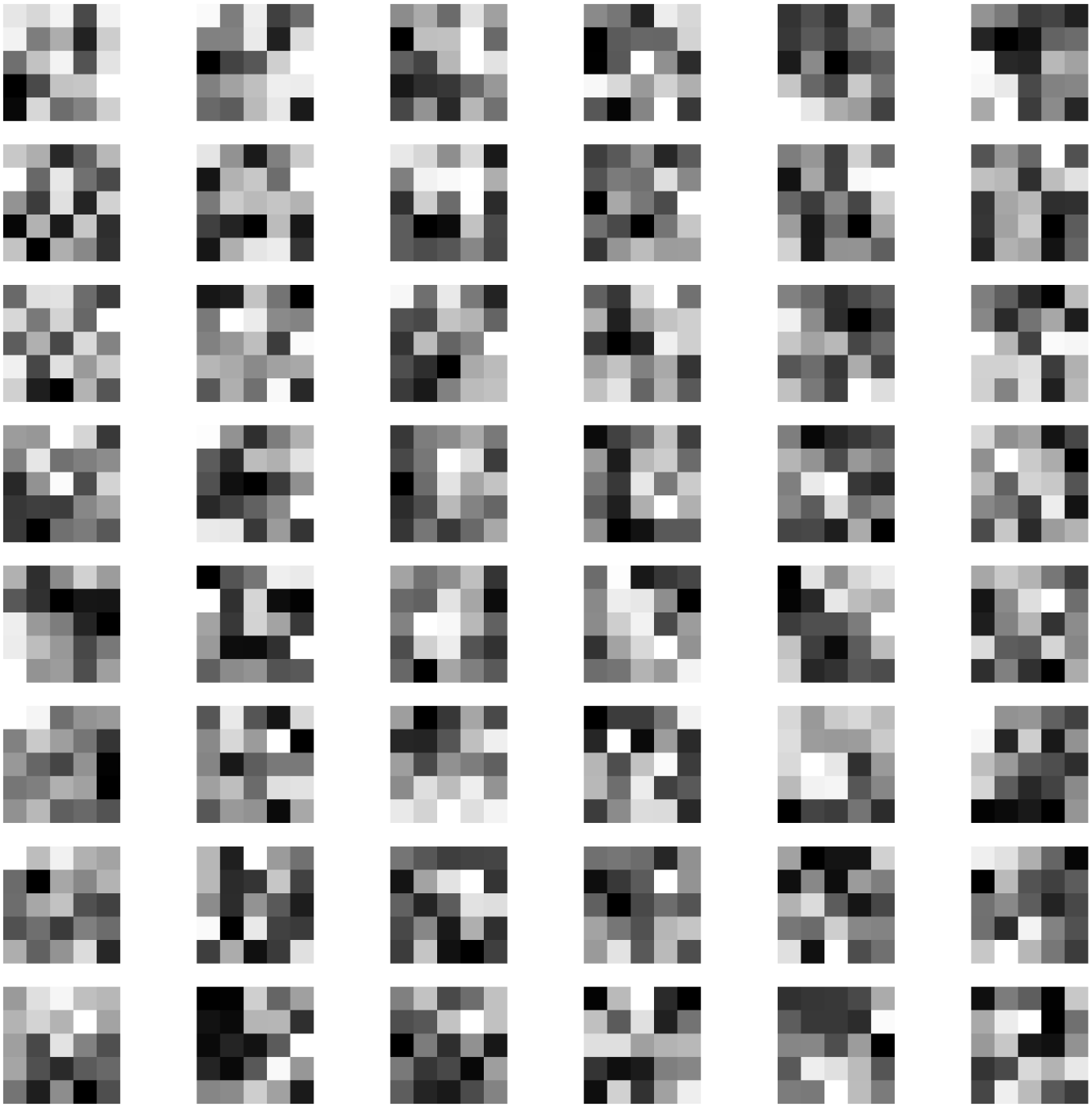


Figure 10: Filters for convolutional layer 4 (2 filters per channel displayed)

Next, we visualise the feature maps for one of each image type from the test dataset as it passes through the network. For simplicity, we choose to visualise only 6 feature maps per layer for each image, all of which correspond to the first input channel of that layer. In order to achieve this, we pass each of these images through the convolutional part of the network, saving the outputs of each layer and plotting these. Full code can be found in Appendix D.

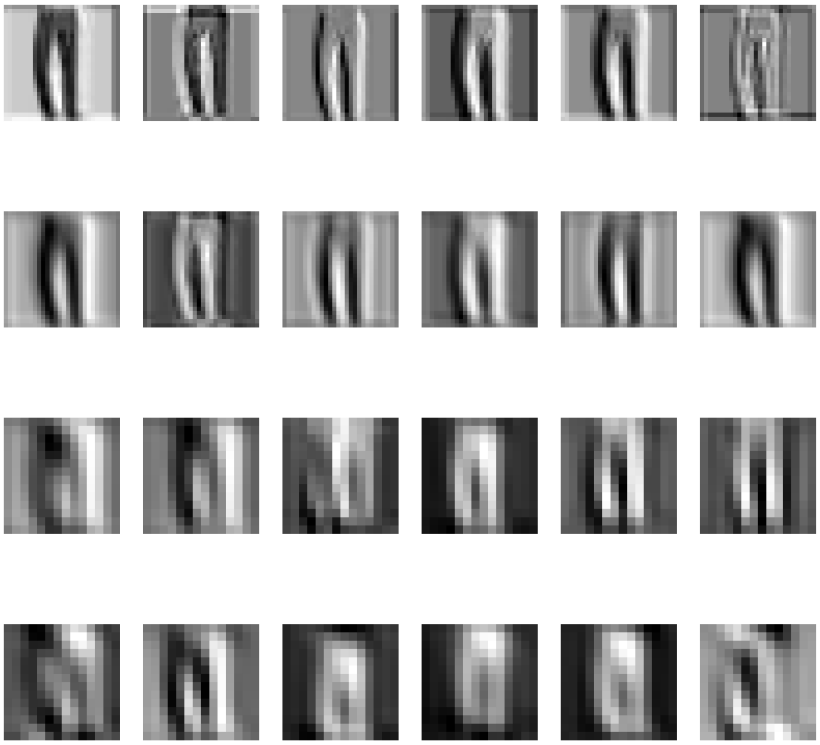


Figure 11: Subset of the feature maps for a trouser image (each row represents one convolutional layer)

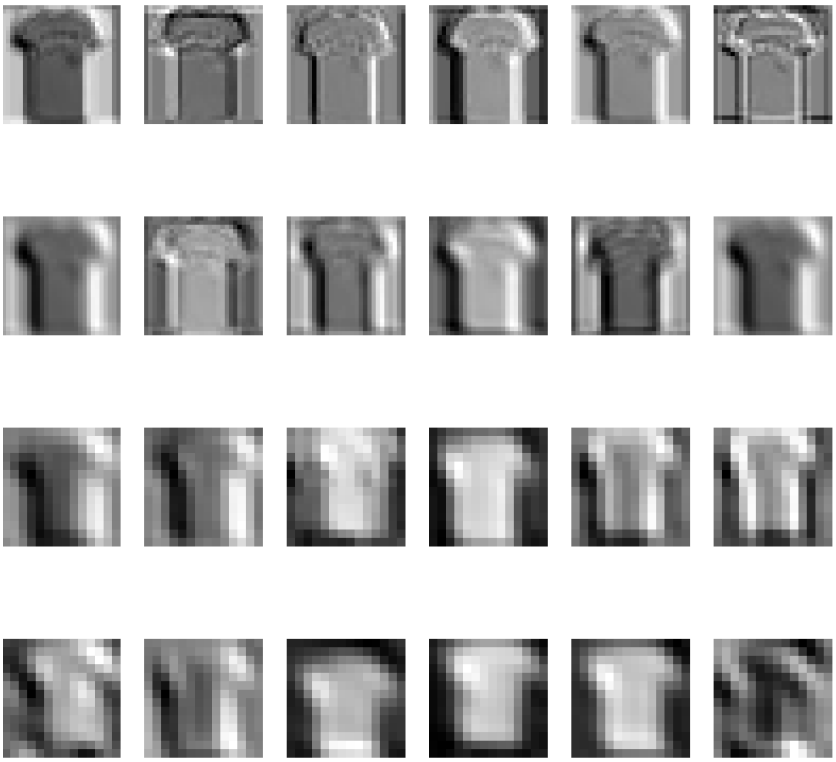


Figure 12: Subset of the feature maps for a T-shirt/top image (each row represents one convolutional layer)

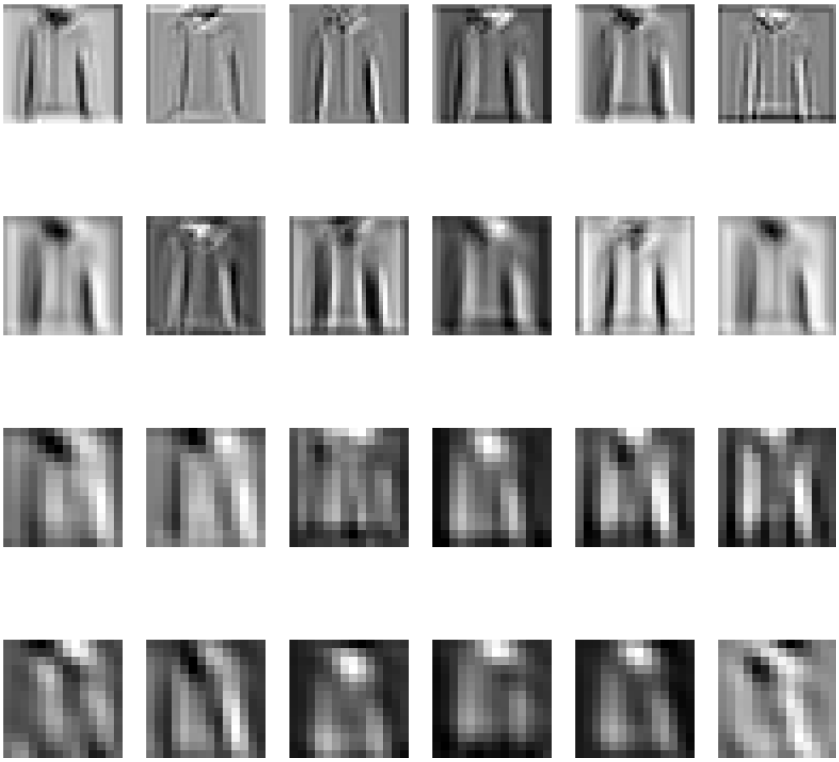


Figure 13: Subset of the feature maps for a coat image (each row represents one convolutional layer)

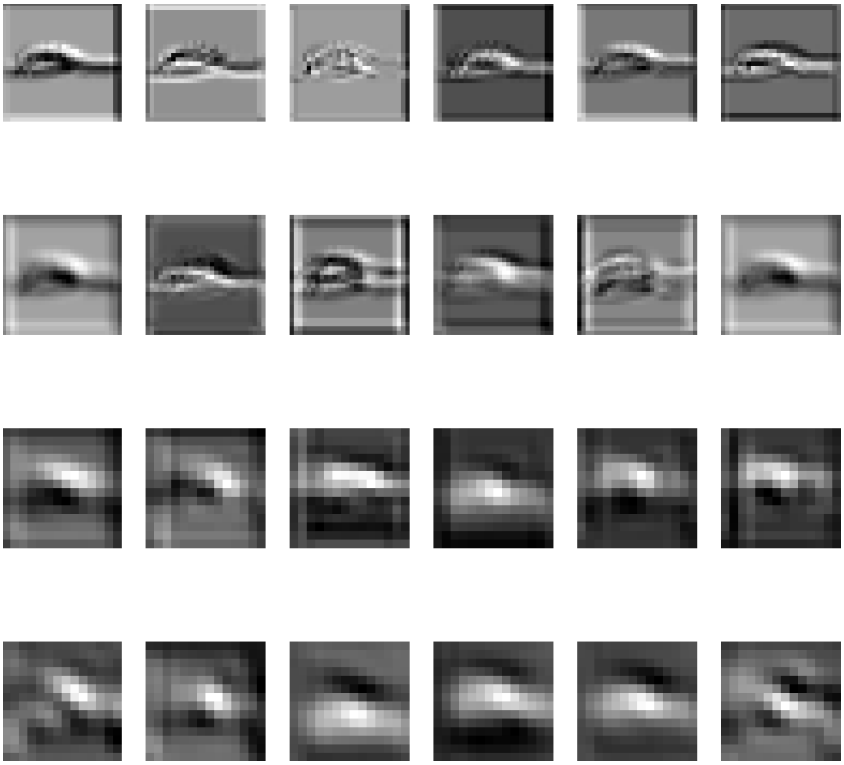


Figure 14: Subset of the feature maps for a sandal image (each row represents one convolutional layer)

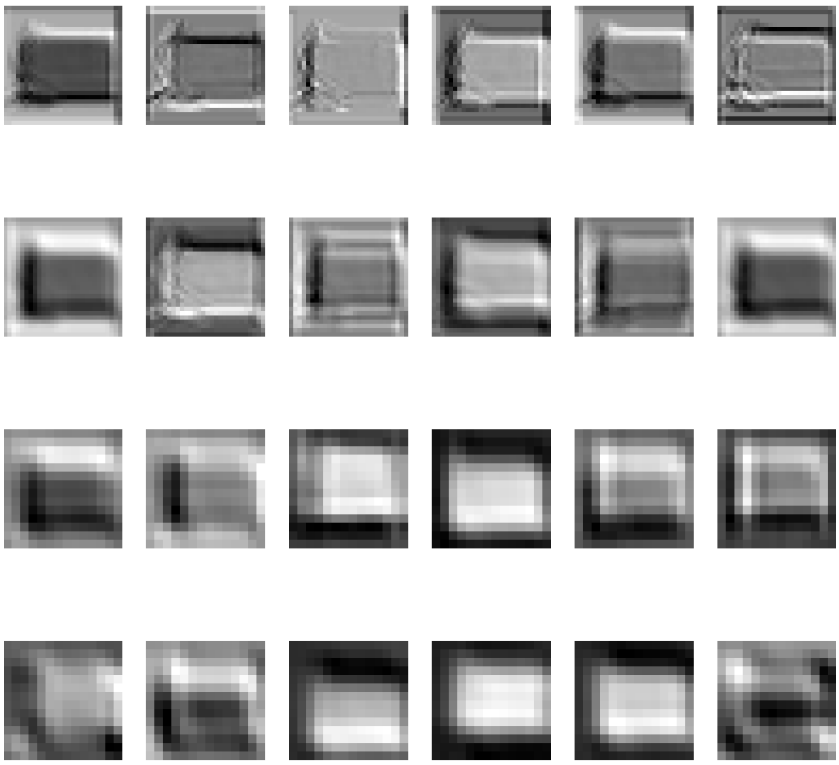


Figure 15: Subset of the feature maps for a bag image (each row represents one convolutional layer)

3.4 Qualitative discussion

In terms of analysing the filter visualisations, it is difficult to provide an interpretation beyond the first layer as in subsequent layers the filters are acting on the activations of previous layers rather than the pixels of the input image. However, from Figure 7, we can inspect the filters in the first layer and hypothesise what they are capturing. They do not lend themselves particularly well to interpretation but in particular we see that in the first filter in Figure 7, the area of greatest activation where the weights are largest is a slight horizontal line at the bottom-left of the filter, suggesting that this filter may detect the items that are more horizontally shaped such as the sandals and bags. Similarly, the fifth filter also shows signs of a smaller horizontal line of large weights in the bottom-left of the filter, suggesting that this serves a similar purpose. The fourth filter has a small vertical line of large weights in the top-left of the filter, suggesting that this filter detects vertical shapes, such as those found in trousers or in the sleeves of coats for example.

In terms of the feature maps, each set of feature maps for each item type tells a similar story. Namely, the feature maps for the first convolutional layers (closer to the input) appear to capture much of the detail of the original images but this detail fades the further we go into the network. By the time we reach the fourth convolutional layer, in many cases it becomes tricky to associate the feature maps with the actual class label. This is to be expected though as the further we go into the network, the more the model abstracts from the visual features we associate with the image into more complex features which the model can then use to distinguish between the different classes more easily, despite the fact that these features may be uninterpretable to us.

4 Q4

The data preparation for this question is shown in Appendix A. The remaining full code for this question is shown in Appendix E, with detailed comments. We show only a few relevant snippets as we go along in this current section.

4.1 Autoencoder implementation

We implemented a vanilla autoencoder to see how this compared to the convolutional neural networks used throughout the rest of this project. The schema consists of an encoder layer ϕ, and the decoder ψ. Defined as:

$$\phi : \mathbb{X} \to \mathbb{F} \tag{1}$$

$$\psi : \mathbb{F} \to \mathbb{X} \tag{2}$$

Our input x is passed through an encoder such that $h = \sigma(Wx + b)$. This results in a compressed *"encoded"* version of our input, the decoder maps this to our reconstructed $x^{'}$: $x^{'} = \sigma^{'}(W^{'}h + b^{'})$. We train our autoencoder to minimize the mean square error loss for $\mathbb{L}(x, x^{'})$.

4.2 Autoencoder final training

All images from Fashion-MNIST-2 dataset as well as the training dataset of the Fashion-MNIST-1 were used as the dataset to train the autoencoder. The Adam optimiser was used as well for the training algorithm.

A hyperparameter search over different hyperparameter value combinations including batch size, learning rate, and model layer structure was performed as follows:

```
hyperparam_grid = {
    'structure': [
      [[784, 128], [128, 32], [32, 10]],
      [[784, 128], [128, 64], [64, 32], [32, 10]],
      [[784, 256], [256, 128], [128, 32], [32, 10]],
      [[784, 512], [256, 128], [128, 64], [64, 32], [32, 10]]
    ],
    'batch_size': [16, 32, 128, 256, 512],
    'learning_rate': [0.1, 0.01, 0.001]
}
```

The autoencoder was trained to convergence on this dataset using the same criteria as that in 1.2, with the train error after convergence being 0.02155, and the best hyperparameters noted below.

```
'structure': [
  [[784, 128], [128, 64], [64, 32], [32, 10]]
],
'batch_size': [32],
'learning_rate': [0.01]
```

4.3 MLP implementation

The next model was a simple multi-layer perceptron with the same structure as our encoder architecture from the best autoencoder, with layers: $[[784, 128], [128, 64], [64, 32], [32, 10]]$. This model was trained on the Fashion-MNIST-1 dataset, using a cross-entropy loss and likewise the Adam Optimizer.

4.4 Pre-training

We trained two different variants of the MLP, one with random weights and the other with the weights initialised to those from the endcoder of the trained autoencoder. For both models we trained on different percentages of the datasets ranging from 5% to the full 100%.

The plot of accuracy against proportion of data used for each weight initialisation approach, for both training and validation, can be seen in Figure 16.

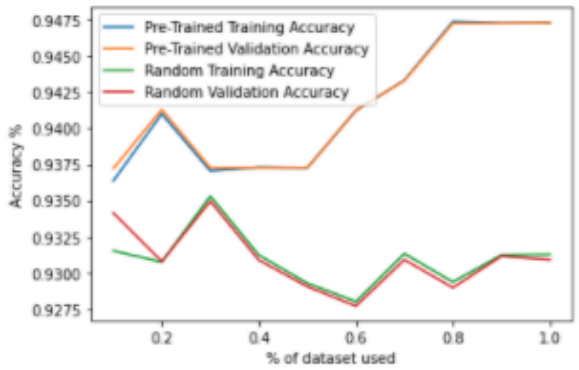


Figure 16: MLP accuracy vs proportion of dataset trained.

Both the randomly initialised and pre-trained models achieved a high accuracy, but the pre-trained model performed better across all percentages of dataset used in training. There are some small fluctuations in the accuracy with increasing amounts of training data, which we may relate to the randomness in the training process with mini-batching. We were surprised to find that the gap in performance between pre-trained and randomly initialised models increased with increasing size of training dataset - our intuition was that the marginal advantage of pretraining would have lessened with increasing dataset size. One explanation for this could be that pre-training initialised the weights in the parameters space that favoured gradient descent to a better optimum, compared to the starting point of the randomly initialised version which may have been arriving at a local optimum. Further experiments varying parameters such as momentum might give some useful insights into this.

4.5 Final accuracy

We re-trained the random weighted MLP model on 30% of the dataset and trained to convergence (convergence criteria using validation loss: mean val loss over most recent 3 epochs greater than mean loss over preceding 3 epochs minus 0.001), likewise the pre-trained MLP was trained on the full 100% of the dataset, reflecting our best model results for each from above. For both models the training, validation and testing accuracy can be seen in Table 4. Just like mentioned above the pre-trained model performed better across all datasets than the random MLP.

Accuracy's	Random MLP	Pre-Trained MLP
Training Accuracy	0.93522	0.947353
Validation Accuracy	0.93482	0.942710
Testing Accuracy	0.93394	0.942715

Table 4: Best Pre-Trained & MLP Accuracy

5 Q5

The data preparation for this question is shown in Appendix A. The remaining full code for this question is shown in Appendix F, with detailed comments. We show only a few relevant snippets as we go along in this current section.

5.1 Model implementation, FMNIST2

The convolutional neural network model was implemented (full code in Appendix F) consistent with that described in Q1 and Q2, with a cross-entropy loss function, and using the following hyperparameters:

```
hyperparam_grid = {'num_dense_layers': [1],
                   'num_conv_layers': [2],
                   'learning_rate': [0.02],
                   'weight_decay': [0], # Regularisation
                   'optimiser': ["SGD"],
                   'batch_size': [train_batch_size]
                   }

# range of epochs
min_epochs = 20
max_epochs = 450

# number of classes
num_classes = 5

# initiate dataframe to store results of grid search
results_df = pd.DataFrame()
```

Examplar accuracy and loss plots for training on the FMNIST-2 dataset, are as shown in Figure 17 and the final results in the table below.

Training Set		**Validation Set**	
Accuracy	Loss	Accuracy	Loss
0.9429	0.1036	0.9292	0.188

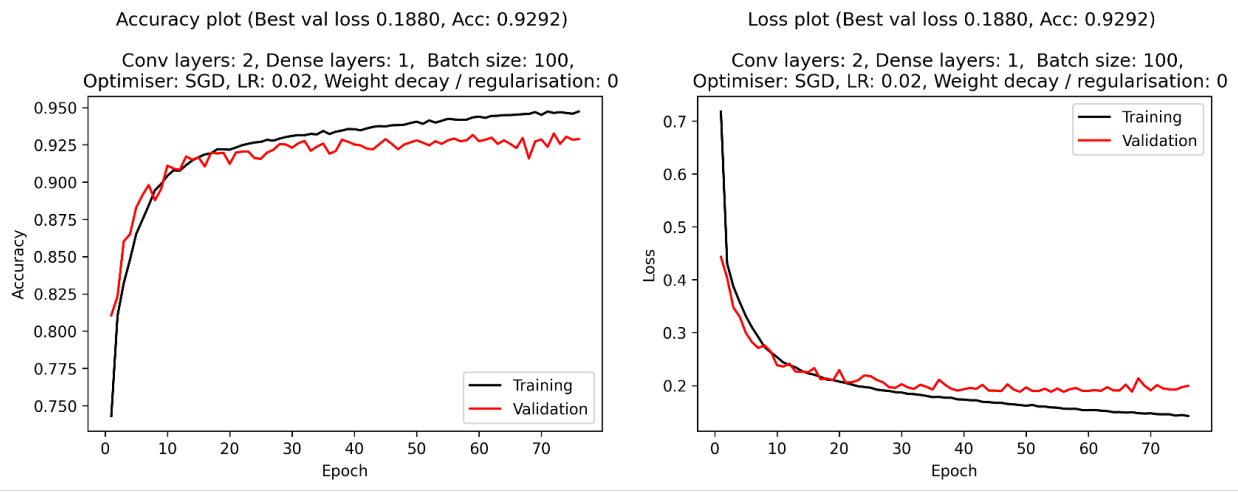


Figure 17: Accuracy and loss plots for initial model trained on FMNIST2

5.2 Hyperparameter optimisation

The model hyperparameters were then iteratively modified to try and improve the model, in terms of reducing the validation loss. The table below shows the performance results for the final best performing model, with the best hyperparameters. The accuracy and loss plots corresponding to this model, as well as the hyperparameters of the model are shown in Figure 18.

```
hyperparam_grid = {'num_dense_layers': [3],
                   'num_conv_layers': [1],
                   'learning_rate': [0.005],
                   'weight_decay': [0.05], # Regularisation
                   'optimiser': ["AdamW"],
                   'batch_size': [train_batch_size]
                   }

# range of epochs
min_epochs = 20
max_epochs = 450

# number of classes
num_classes = 5

# initiate dataframe to store results of grid search
results_df = pd.DataFrame()
```

Training Set		Validation Set		Test Set	
Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
0.9472	0.1401	0.9404	0.1715	0.9294	0.2946

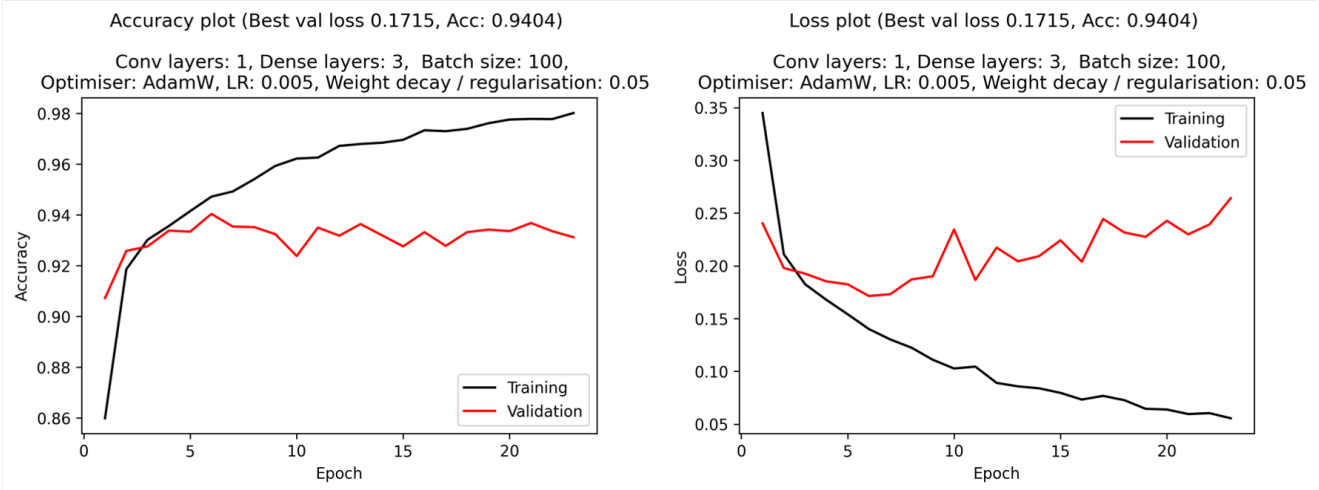


Figure 18: Accuracy and loss plots for best model trained on FMNIST2

5.3 Model implementation, FMNIST1

The above architecture, defined by the hyperparameters above was then applied to the FMNIST1 dataset. The performance associated with this is shown in the table below, and associated loss and accuracy plots are shown in Figure 19:

Training Set		Validation Set		Test Set	
Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
0.9909	0.0292	0.9868	0.0427	0.9832	0.0068

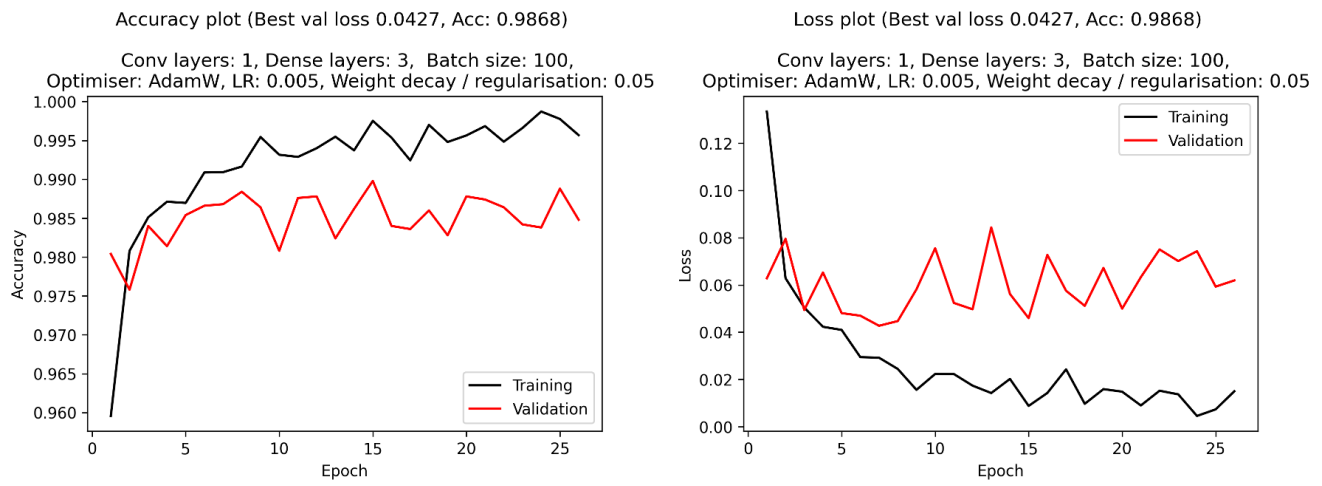


Figure 19: Accuracy and loss plots for FMNIST1 dataset

It is interesting to note that the FMNIST1 model performed noticeably better than the FMNIST2 model in terms of both accuracy and loss. This is likely due to FMNIST2 containing articles that are harder to distinguish, such as the Pullover vs Shirt and Shoe vs Ankle Boot.

5.4 Transfer learning

The weights of the model trained on FMNIST2 were saved. For this part, we loaded them up in order to perform Transfer Learning. The last layer though was reinitialised, meaning training was performed only on that layer. The following code shows the model being loaded and the last layer being reinitialised;

```
# Load Model
model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])
weights_path = f"{output_path}MODEL_0.1715_devloss_0.9404_devacc_state_dict.pth"
model.load_state_dict(torch.load(weights_path, map_location=torch.device('cpu')))
for param in model.parameters():
    param.requires_grad = False

# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by default
model.fc3 = nn.Linear(84, num_classes)
```

A loop was created where in each iteration a new model was trained with increasing proportions of the training dataset incorporated into training (in increments of 5 percent). Then the model was trained using randomly initialised weights and transferred weights, for comparison of performance. The full code implementation can be found in Appendix F. The best validation loss was recorded for each and the results are as shown in Figure 20.

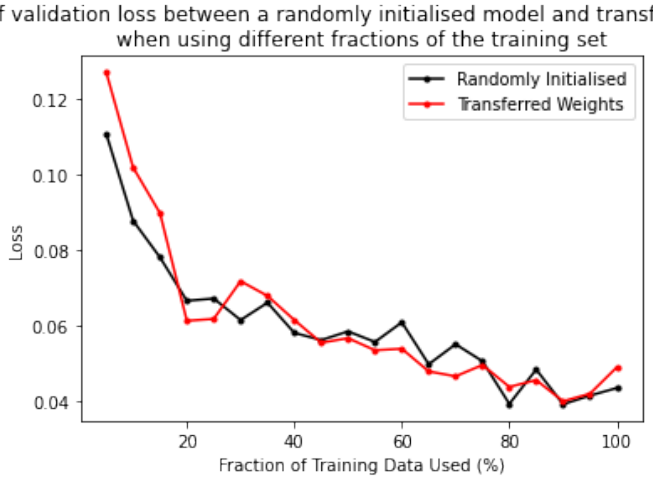


Figure 20: Best validation loss as training set size increases for both random initialisation (RI) and transferred weights (TL).

This Figure shows interesting results. At small training set size (20%), it seems the randomly initialised model performs slightly better than transfer learning model. The opposite is then generally observed till a training set size of 80%, where from then the randomly initialised model starts outperforming again. We note however that the difference in loss is small across all sizes of training dataset.

It would therefore seem that most of the variation is random. This was not expected but rather, we thought that when the training size is small the randomly initialised model would not do well as it does not have enough data to learn from, whilst the model with the transferred weights would outperform. Perhaps, this was not observed as the data was simple enough to learn with little data. Another possible explanation is that the model used (Q5.2) happened to only have 1 convolutional layer (and three fully connected ones). Transfer learning with all but the classification layer frozen relies on features having already been learnt in previous convolution layers. However, as we only had one such layer, perhaps the learnt features were well developed enough to facilitate good transfer learning results with the majority of learning in the fully connected layers. Further experiments using a model with more convolutional layers would be an interesting next step to further explore this.

5.5 Final accuracy

Random Initialisation (90%)			**Transfer Learning (90%)**		
Training Set	Validation Set	Test Set	Training Set	Validation Set	Test Set
0.9915	0.9899	0.986	0.9919	0.9868	0.9864

Both models did best when using 90% of the training dataset. The randomly initialised model outperformed the transfer learning model in both validation and test sets.

Appendix A Data preparation code

```
# LOAD FASHION MNIST DATASET (fm)

transform = tv.transforms.Compose([tv.transforms.ToTensor(), tv.transforms.Normalize((0.5,), (0.5,))])

fm1_train_val = tv.datasets.FashionMNIST(root = "./data", train = True, download = True, transform = transform)
fm1_test = tv.datasets.FashionMNIST(root = "./data", train = False, download = True, transform = transform)
fm2_train_val = tv.datasets.FashionMNIST(root = "./data", train = True, download = True, transform = transform)
fm2_test = tv.datasets.FashionMNIST(root = "./data", train = False, download = True, transform = transform)


# SPLIT TRAINING AND VALIDATION DATASETS INTO fm1 AND fm2
# (non-overlapping classes in each)

# specify classes to include in fm1 and fm2
fm1_labels = [0, 1, 4, 5, 8]
fm2_labels = [2, 3, 6, 7, 9]


# update fm1 and fm2 train_val datasets to just include the relevant classes for train_val datasets

# extract indices (boolean array) of images corresponding to each of the classes for fm1, and fm2
fm1_train_val_indices = (fm1_train_val.targets ==0) | (fm1_train_val.targets ==1) | (fm1_train_val.targets ==4) | (fm1_train_val.targets ==5) | (fm1_train_val.targets ==8)
fm2_train_val_indices = (fm2_train_val.targets ==2) | (fm2_train_val.targets ==3) | (fm2_train_val.targets ==6) | (fm2_train_val.targets ==7) | (fm2_train_val.targets ==9)

##fm1
fm1_train_val.data = fm1_train_val.data[fm1_train_val_indices]
fm1_train_val.targets = fm1_train_val.targets[fm1_train_val_indices]
### re-index labels, otherwise dataloader does not work
fm1_train_val.targets[fm1_train_val.targets == 4] = 2
fm1_train_val.targets[fm1_train_val.targets == 5] = 3
fm1_train_val.targets[fm1_train_val.targets == 8] = 4

##fm2
fm2_train_val.data = fm2_train_val.data[fm2_train_val_indices]
fm2_train_val.targets = fm2_train_val.targets[fm2_train_val_indices]
### re-index labels, otherwise dataloader does not work
fm2_train_val.targets[fm2_train_val.targets == 2] = 0
fm2_train_val.targets[fm2_train_val.targets == 3] = 1
fm2_train_val.targets[fm2_train_val.targets == 6] = 2
fm2_train_val.targets[fm2_train_val.targets == 7] = 3
fm2_train_val.targets[fm2_train_val.targets == 9] = 4


# REPEAT THE ABOVE FOR TEST DATASETS

```

```
fm1_test_indices = (fm1_test.targets ==0) | (fm1_test.targets ==1) | (fm1_test.targets ==4) | (fm1_test.targets ==5) | (fm1_test.targets ==8)
fm2_test_indices = (fm2_test.targets ==2) | (fm2_test.targets ==3) | (fm2_test.targets ==6) | (fm2_test.targets ==7) | (fm2_test.targets ==9)

##fm1
fm1_test.data = fm1_test.data[fm1_test_indices]
fm1_test.targets = fm1_test.targets[fm1_test_indices]
#### re-index labels, otherwise dataloader does not work
fm1_test.targets[fm1_test.targets == 4] = 2
fm1_test.targets[fm1_test.targets == 5] = 3
fm1_test.targets[fm1_test.targets == 8] = 4

##fm2
fm2_test.data = fm2_test.data[fm2_test_indices]
fm2_test.targets = fm2_test.targets[fm2_test_indices]
### re-index labels, otherwise dataloader does not work
fm2_test.targets[fm2_test.targets == 2] = 0
fm2_test.targets[fm2_test.targets == 3] = 1
fm2_test.targets[fm2_test.targets == 6] = 2
fm2_test.targets[fm2_test.targets == 7] = 3
fm2_test.targets[fm2_test.targets == 9] = 4


# CREATE DICTIONARIES TO MAP OLD TO NEW INDICES

fm1_label_newindex_to_oldindex = {0: 0, 1: 1, 2: 4, 3: 5, 4: 8}
fm2_label_newindex_to_oldindex = {0: 2, 1: 3, 2: 6, 3: 7, 4: 9}


# CREATE DICTIONARIES TO MAP INDICES TO ITEM NAMES

fm1_label_index_to_text_dict = {0: "T-shirt/top", 1: "Trouser", 2: "Coat", 3: "Sandal", 4: "Bag"}
fm2_label_index_to_text_dict = {0: "Pullover", 1: "Dress", 2: "Shirt", 3: "Sneaker", 4: "Ankle boot"}


# FINAL TRAINING-VALIDATION SPLITTING

# split the train_val datasets into seperate train and validation datasets

fm1_train, fm1_val = torch.utils.data.random_split(fm1_train_val, [25000, 5000])
fm2_train, fm2_val = torch.utils.data.random_split(fm2_train_val, [25000, 5000])


# DATALOADER SETUP

# create dataloaders for each of the datasets

train_batch_size = 100
dev_batch_size = 1
test_batch_size = 1

fm1_trainloader = torch.utils.data.DataLoader(fm1_train, batch_size = train_batch_size, shuffle = True)
```

```
fm1_devloader = torch.utils.data.DataLoader(fm1_val, batch_size = dev_batch_size, shuffle =
↪ False)
fm1_testloader = torch.utils.data.DataLoader(fm1_test, batch_size = test_batch_size, shuffle =
↪ False)

fm2_trainloader = torch.utils.data.DataLoader(fm2_train,  batch_size = train_batch_size,
↪ shuffle = True)
fm2_devloader = torch.utils.data.DataLoader(fm2_val, batch_size = dev_batch_size, shuffle =
↪ False)
fm2_testloader = torch.utils.data.DataLoader(fm2_test, batch_size = test_batch_size, shuffle =
↪ False)


# DISPLAY EXAMPLE IMAGES AND LABELS

NUMBER_TO_DISPLAY = 2

for i, (xb, yb) in enumerate(fm1_devloader):
    x = xb.view(28,28)
    y = yb.item()
    print(f"Class: {fm1_label_index_to_text_dict[y]}")

    plt.imshow(x, cmap="gray")
    plt.axis('off')
    plt.show()

    if i == NUMBER_TO_DISPLAY-1:
      break
```

Appendix B Q1 code

```
# DATA PREP AS PER APPENDIX A


# HYPERPARAMETERS

# hyperparameter grid for Q1
hyperparam_grid = {'num_dense_layers': [2],
                   'num_conv_layers': [2],
                   'learning_rate': [0.005],
                   'weight_decay': [0], # Regularisation
                   'optimiser': ["SGD"],
                   'batch_size': [train_batch_size]}

# range of epochs
min_epochs = 20
max_epochs = 450

# number of classes
num_classes = 5

# initiate dataframe to store results of grid search
results_df = pd.DataFrame()


# MODEL, TRAINING, EVALUATION

for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):

  # pairings of hyperparameters to skip

  if (hyperparams['optimiser'] == "SGD" and hyperparams['learning_rate'] < 0.005) or
  ↪  (hyperparams['optimiser'] == "AdamW" and hyperparams['learning_rate'] > 0.005):
    continue

  # specify model architecture

  class CNN(nn.Module):

    def __init__(self, num_classes, num_dense_layers, num_conv_layers):
      super(CNN, self).__init__()
      self.num_dense_layers = num_dense_layers
      self.num_conv_layers = num_conv_layers

      # single conv layer
      if num_conv_layers == 1:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
          self.fc1 = nn.Linear(6*(12**2), num_classes)
        elif num_dense_layers == 2:
          self.fc1 = nn.Linear(6*(12**2), 80)
          self.fc2 = nn.Linear(6, num_classes)
```

```
        elif num_dense_layers == 3:
          self.fc1 = nn.Linear(6*(12**2), 120)
          self.fc2 = nn.Linear(120, 84)
          self.fc3 = nn.Linear(20, num_classes)

      # two conv layers
      elif num_conv_layers == 2:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
          self.fc1 = nn.Linear(12*(4**2), num_classes)
        elif num_dense_layers == 2:
          self.fc1 = nn.Linear(12*(4**2), 80)
          self.fc2 = nn.Linear(80, num_classes)
        elif num_dense_layers == 3:
          self.fc1 = nn.Linear(12*(4**2), 120)
          self.fc2 = nn.Linear(120, 84)
          self.fc3 = nn.Linear(20, num_classes)

    def forward(self, x):

      #conv2d and maxpools
      if self.num_conv_layers == 1:
        x = self.pool1(F.relu(self.conv1(x)))
        x = x.view(-1, 6*(12**2))
      elif self.num_conv_layers == 2:
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = x.view(-1, 12*(4**2))

      #fully-connnected layers
      if self.num_dense_layers == 1:
        x = self.fc1(x)
      if self.num_dense_layers == 2:
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
      if self.num_dense_layers == 3:
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

      return x


# Initialise model and send to GPU

model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
model.to(device)
```

```
110
111     # Loss & optimiser
112     criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
113
114     if hyperparams['optimiser'] == "SGD":
115       optimiser = torch.optim.SGD(model.parameters(), lr=hyperparams['learning_rate'],
          ↪  weight_decay=hyperparams['weight_decay'])
116     elif hyperparams['optimiser'] == "AdamW":
117       optimiser = torch.optim.AdamW(model.parameters(), lr=hyperparams['learning_rate'],
          ↪  weight_decay=hyperparams['weight_decay'])
118
119
120     # training loop
121
122     train_loss_list = []
123     train_accuracy_list = []
124     dev_loss_list = []
125     dev_accuracy_list = []
126
127     best_dev_loss = 9999 # initiate high value so that training doesn't immediately stop
128
129     converged = False
130     epochs_counter = 0
131     epochs_to_plot = []
132
133     while (converged == False) and (epochs_counter <= max_epochs):
134
135       train_running_loss = 0
136       train_running_corrects = 0
137       train_num_of_batches = 0
138
139       model.train()
140
141       for data, label in fm1_trainloader:
142
143         data = data.to(device)
144         label = label.to(device)
145
146         # zero the gradients
147         optimiser.zero_grad()
148
149         # forward + backward + optimize
150         train_preds = model(data)
151         loss = criterion(train_preds, label.long())
152         loss.backward()
153         optimiser.step()
154
155         # metrics
156         train_preds_indices = torch.argmax(train_preds.data, 1)
157         train_running_corrects += (train_preds_indices == label).sum().item()
158         train_running_loss += loss.item()
159         train_num_of_batches +=1
160
161       train_accuracy = train_running_corrects/(train_num_of_batches*train_batch_size)
162       train_accuracy_list.append(train_accuracy)
163       train_loss = train_running_loss/train_num_of_batches
```

```
    train_loss_list.append(train_loss)


    model.eval()

    with torch.no_grad():

      dev_running_loss = 0
      dev_running_corrects = 0
      dev_num_of_batches = 0

      for data, label in fm1_devloader:

        data = data.to(device)
        label = label.to(device)

        # forward
        dev_preds = model(data)
        loss = criterion(dev_preds, label.long())

        # metrics
        dev_preds_indices = torch.argmax(dev_preds.data, 1)
        dev_running_corrects += (dev_preds_indices == label).sum().item()
        dev_running_loss += loss.item()
        dev_num_of_batches +=1

      dev_accuracy = dev_running_corrects/(dev_num_of_batches*dev_batch_size)
      dev_accuracy_list.append(dev_accuracy)
      dev_loss = dev_running_loss/dev_num_of_batches
      dev_loss_list.append(dev_loss)

      epochs_counter += 1
      epochs_to_plot.append(epochs_counter)

      print(f"epoch: {epochs_counter}, train_loss: {train_loss:.4f}, train_acc:
       ↪  {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")

      ## Store the current results if it is the highest validation accuracy so far
      if dev_loss < best_dev_loss:
          best_dev_accuracy = dev_accuracy
          best_dev_loss = dev_loss
          best_train_accuracy = train_accuracy
          best_train_loss = train_loss
          best_epoch = epochs_counter
          best_model = model

      ## Early stopping criteria
      if epochs_counter >= min_epochs:
        if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
          converged = True
          print("Model Converged")


  # Evaluate loss and accuracy on test set

  with torch.no_grad():
```

```
219
220       test_running_loss = 0
221       test_running_corrects = 0
222       test_num_of_batches = 0
223
224       predictions_list = []
225       labels_list = []
226
227       for data, label in fm1_testloader:
228
229         data = data.to(device)
230         label = label.to(device)
231
232         # forward
233         test_preds = best_model(data)
234         loss = criterion(test_preds, label.long())
235
236         # metrics
237         test_preds_indices = torch.argmax(test_preds.data, 1)
238         test_running_corrects += (test_preds_indices == label).sum().item()
239         test_running_loss += loss.item()
240         test_num_of_batches +=1
241
242         labels_list.append(label.long().item())
243         predictions_list.append(test_preds_indices.item())
244
245       test_accuracy = test_running_corrects/(test_num_of_batches*test_batch_size)
246       test_loss = test_running_loss/test_num_of_batches
247
248     print(f"BEST MODEL TEST ACCURACY: {test_accuracy}")
249     print(f"BEST MODEL TEST LOSS: {test_loss}")
250
251     # Save results from epoch with best validation loss, for current run and hyperparams
252
253     hyperparams['model_index'] = run_index + 1
254     hyperparams['best_epoch'] = best_epoch
255     hyperparams['best_validation_loss'] = best_dev_loss
256     hyperparams['associated_validation_accuracy'] = best_dev_accuracy
257     hyperparams['associated_training_loss'] = best_train_loss
258     hyperparams['associated_training_accuracy'] = best_train_accuracy
259     hyperparams['test_loss'] = test_loss
260     hyperparams['test_accuracy'] = test_accuracy
261
262     results_df = results_df.append(hyperparams, ignore_index=True)
263     results_df = results_df.sort_values("best_validation_loss", ascending=False).round(4)
264     results_df.to_csv(f"{output_path}RESULTS_TABLE_cnn.csv")
265
266     print(f"\n RUN_INDEX: {run_index + 1} \n")
267     print(hyperparams)
268
269     # save the model
270     torch.save(best_model,
        ↪ f"{output_path}MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}
271
272     # # Loss plot
273     plt.plot(epochs_to_plot, train_loss_list, color='k', linestyle='-')
```

```
  plt.plot(epochs_to_plot, dev_loss_list, color='r', linestyle='-')
  plt.legend(['Training', 'Validation'], loc='upper right')
  plt.ylabel('Loss', color='k')
  plt.xlabel('Epoch', color='k')
  plt.title(f'''Loss plot (Best val loss {best_dev_loss:.4f}, Acc: {best_dev_accuracy:.4f}) \n
  Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
↪  {hyperparams['num_dense_layers']},  Batch size: {hyperparams['batch_size']},
  Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
↪  regularisation: {hyperparams['weight_decay']}''', color='k')


↪  plt.savefig(f"{output_path}MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_devloss_{best_dev_a
↪  dpi=300, bbox_inches = "tight")
  plt.show()

  # # Accuracy plot
  plt.plot(epochs_to_plot, train_accuracy_list, color='k', linestyle='-')
  plt.plot(epochs_to_plot, dev_accuracy_list, color='r', linestyle='-')
  plt.legend(['Training', 'Validation'], loc='lower right')
  plt.ylabel('Accuracy', color='k')
  plt.xlabel('Epoch', color='k')
  plt.title(f'''Accuracy plot (Best val loss {best_dev_loss:.4f}, Acc:
↪  {best_dev_accuracy:.4f}) \n
  Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
↪  {hyperparams['num_dense_layers']},  Batch size: {hyperparams['batch_size']},
  Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
↪  regularisation: {hyperparams['weight_decay']}''', color='k')


↪  plt.savefig(f"{output_path}MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_devloss_{best_dev_a
↪  dpi=300, bbox_inches = "tight")
  plt.show()


# CONFUSION MATRIX

# Generate confusion matrix from best models predictions on the  test dataset

# print(f"Class labels: {fm1_label_index_to_text_dict}")

cm = ConfusionMatrix(actual_vector=labels_list, predict_vector=predictions_list)
cm.relabel(mapping=fm1_label_index_to_text_dict)
items = ["T-shirt/top", "Trouser", "Coat", "Sandal", "Bag"]
cm.plot(cmap=plt.cm.Blues,number_label=True, class_name=items)
plt.xticks(rotation=50)
plt.savefig(f"{output_path}MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_devloss_{best_dev_accurac
↪  dpi=300, bbox_inches = "tight")
```

Appendix C Q2 code

```
# DATA PREP AS PER APPENDIX A

# hyperparameters
hyperparams = {'num_dense_layers': 2,
               'num_conv_layers': 1,
               'learning_rate': 0.0005,
               'weight_decay': 0, # Regularisation
               'optimiser': "Adam",
               'batch_size': train_batch_size}

# range of epochs
min_epochs = 20
max_epochs = 300

# number of classes
num_classes = 5

# initiate dataframe to store results of grid search
results_df = pd.DataFrame()

# MODEL, TRAINING, EVALUATION

# PyTorch model class, which takes the specified hyperparameters as inputs
class CNN(nn.Module):

  def __init__(self, num_classes, num_dense_layers, num_conv_layers):
    super(CNN, self).__init__()
    self.num_dense_layers = num_dense_layers
    self.num_conv_layers = num_conv_layers

    # single conv layer
    if num_conv_layers == 1:
      self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
      self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

      # dense layers
      if num_dense_layers == 1:
        self.fc1 = nn.Linear(6*(12**2), num_classes)
      elif num_dense_layers == 2:
        self.fc1 = nn.Linear(6*(12**2), 80)
        self.fc2 = nn.Linear(80, num_classes)
      elif num_dense_layers == 3:
        self.fc1 = nn.Linear(6*(12**2), 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)

    # two conv layers
    elif num_conv_layers == 2:
      self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
      self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
      self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
      self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

      # dense layers
```

```
      if num_dense_layers == 1:
        self.fc1 = nn.Linear(12*(4**2), num_classes)
      elif num_dense_layers == 2:
        self.fc1 = nn.Linear(12*(4**2), 80)
        self.fc2 = nn.Linear(80, num_classes)
      elif num_dense_layers == 3:
        self.fc1 = nn.Linear(12*(4**2), 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)

  def forward(self, x):

    # conv2d and maxpools
    if self.num_conv_layers == 1:
      x = self.pool1(F.relu(self.conv1(x)))
      x = x.view(-1, 6*(12**2))
    elif self.num_conv_layers == 2:
      x = self.pool1(F.relu(self.conv1(x)))
      x = self.pool2(F.relu(self.conv2(x)))
      x = x.view(-1, 12*(4**2))

    # fully-connnected layers
    if self.num_dense_layers == 1:
      x = self.fc1(x)
    if self.num_dense_layers == 2:
      x = F.relu(self.fc1(x))
      x = self.fc2(x)
    if self.num_dense_layers == 3:
      x = F.relu(self.fc1(x))
      x = F.relu(self.fc2(x))
      x = self.fc3(x)

    return x

# Initialise model and send to GPU

model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
model.to(device)


# Loss & optimiser
criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us

if hyperparams['optimiser'] == "SGD":
  optimiser = torch.optim.SGD(model.parameters(), lr=hyperparams['learning_rate'],
  ↪  weight_decay=hyperparams['weight_decay'])
elif hyperparams['optimiser'] == "Adam":
  optimiser = torch.optim.Adam(model.parameters(), lr=hyperparams['learning_rate'])
elif hyperparams['optimiser'] == "AdamW":
  optimiser = torch.optim.AdamW(model.parameters(), lr=hyperparams['learning_rate'],
  ↪  weight_decay=hyperparams['weight_decay'])


```

```
# training loop

train_loss_list = []
train_accuracy_list = []
dev_loss_list = []
dev_accuracy_list = []

best_dev_loss = 9999 # initiate high value so that training doesn't immediately stop

converged = False
epochs_counter = 0
epochs_to_plot = []

while (converged == False) and (epochs_counter <= max_epochs):

  train_running_loss = 0
  train_running_corrects = 0
  train_num_of_batches = 0

  model.train()

  for data, label in fm1_trainloader:

    data = data.to(device)
    label = label.to(device)

    # zero the gradients
    optimiser.zero_grad()

    # forward + backward + optimize
    train_preds = model(data)
    loss = criterion(train_preds, label.long())
    loss.backward()
    optimiser.step()

    # metrics
    train_preds_indices = torch.argmax(train_preds.data, 1)
    train_running_corrects += (train_preds_indices == label).sum().item()
    train_running_loss += loss.item()
    train_num_of_batches +=1

  train_accuracy = train_running_corrects/(train_num_of_batches*train_batch_size)
  train_accuracy_list.append(train_accuracy)
  train_loss = train_running_loss/train_num_of_batches
  train_loss_list.append(train_loss)


  model.eval()

  with torch.no_grad():

    dev_running_loss = 0
    dev_running_corrects = 0
    dev_num_of_batches = 0

    for data, label in fm1_devloader:
```

```
165
166         data = data.to(device)
167         label = label.to(device)
168
169         # forward
170         dev_preds = model(data)
171         loss = criterion(dev_preds, label.long())
172
173         # metrics
174         dev_preds_indices = torch.argmax(dev_preds.data, 1)
175         dev_running_corrects += (dev_preds_indices == label).sum().item()
176         dev_running_loss += loss.item()
177         dev_num_of_batches +=1
178
179       dev_accuracy = dev_running_corrects/(dev_num_of_batches*dev_batch_size)
180       dev_accuracy_list.append(dev_accuracy)
181       dev_loss = dev_running_loss/dev_num_of_batches
182       dev_loss_list.append(dev_loss)
183
184       epochs_counter += 1
185       epochs_to_plot.append(epochs_counter)
186
187       print(f"epoch: {epochs_counter}, train_loss: {train_loss:.4f}, train_acc:
          ↪  {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")
188
189       # Store the current results if it has the smallest validation loss so far
190       if dev_loss < best_dev_loss:
191           best_dev_accuracy = dev_accuracy
192           best_dev_loss = dev_loss
193           best_train_accuracy = train_accuracy
194           best_train_loss = train_loss
195           best_epoch = epochs_counter
196           best_model = model
197
198       # Early stopping criteria
199       if epochs_counter >= min_epochs:
200         if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
201           converged = True
202           print("Model Converged")
203
204
205 # Evaluate loss and accuracy on test set
206
207 with torch.no_grad():
208
209   test_running_loss = 0
210   test_running_corrects = 0
211   test_num_of_batches = 0
212
213   predictions_list = []
214   labels_list = []
215
216   for data, label in fm1_testloader:
217
218     data = data.to(device)
219     label = label.to(device)
```

```
220
221     # forward
222     test_preds = best_model(data)
223     loss = criterion(test_preds, label.long())
224
225     # metrics
226     test_preds_indices = torch.argmax(test_preds.data, 1)
227     test_running_corrects += (test_preds_indices == label).sum().item()
228     test_running_loss += loss.item()
229     test_num_of_batches +=1
230
231     labels_list.append(label.long().item())
232     predictions_list.append(test_preds_indices.item())
233
234   test_accuracy = test_running_corrects/(test_num_of_batches*test_batch_size)
235   test_loss = test_running_loss/test_num_of_batches
236
237 print(f"BEST MODEL TEST ACCURACY: {test_accuracy}")
238 print(f"BEST MODEL TEST LOSS: {test_loss}")
239
240 # Save results from epoch with best validation loss
241
242 hyperparams['best_epoch'] = best_epoch
243 hyperparams['best_validation_loss'] = best_dev_loss
244 hyperparams['associated_validation_accuracy'] = best_dev_accuracy
245 hyperparams['associated_training_loss'] = best_train_loss
246 hyperparams['associated_training_accuracy'] = best_train_accuracy
247 hyperparams['test_loss'] = test_loss
248 hyperparams['test_accuracy'] = test_accuracy
249
250 results_df = results_df.append(hyperparams, ignore_index=True)
251 results_df = results_df.sort_values("best_validation_loss", ascending=False).round(4)
252 results_df.to_csv(f"{output_path}RESULTS_TABLE_MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_d
253
254 print(hyperparams)
255
256 # save the model
257 torch.save(best_model,
    ↪  f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc.pth")
258
259 # Loss plot
260 plt.plot(epochs_to_plot, train_loss_list, color='k', linestyle='-')
261 plt.plot(epochs_to_plot, dev_loss_list, color='r', linestyle='-')
262 plt.legend(['Training', 'Validation'], loc='upper right')
263 plt.ylabel('Loss', color='k')
264 plt.xlabel('Epoch', color='k')
265 plt.title(f'''Loss plot (Best val loss {best_dev_loss:.4f}, Acc: {best_dev_accuracy:.4f}) \n
266 Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
    ↪  {hyperparams['num_dense_layers']},  Batch size: {hyperparams['batch_size']},
267 Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
    ↪  regularisation: {hyperparams['weight_decay']}''', color='k')
268
269 plt.savefig(f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc_LOSSPLOT.png",
    ↪  dpi=300, bbox_inches = "tight")
270 plt.show()
271
```

```
# Accuracy plot
plt.plot(epochs_to_plot, train_accuracy_list, color='k', linestyle='-')
plt.plot(epochs_to_plot, dev_accuracy_list, color='r', linestyle='-')
plt.legend(['Training', 'Validation'], loc='lower right')
plt.ylabel('Accuracy', color='k')
plt.xlabel('Epoch', color='k')
plt.title(f'''Accuracy plot (Best val loss {best_dev_loss:.4f}, Acc: {best_dev_accuracy:.4f})
↪  \n
Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
↪  {hyperparams['num_dense_layers']},  Batch size: {hyperparams['batch_size']},
Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
↪  regularisation: {hyperparams['weight_decay']}''', color='k')

plt.savefig(f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc_ACCPLOT.png",
↪  dpi=300, bbox_inches = "tight")
plt.show()
```

Appendix D Q3 code

```
# DATA PREP AS PER APPENDIX A


# HYPERPARAMETERS

# hyperparameter grid for q3
hyperparam_grid = {'num_dense_layers': [2],
                   'num_conv_layers': [4],
                   'learning_rate': [0.005],
                   'weight_decay': [0], # Regularisation
                   'optimiser': ["SGD"],
                   'batch_size': [train_batch_size]}

# range of epochs
min_epochs = 20
max_epochs = 300

# number of classes
num_classes = 5

# MODEL, TRAINING, EVALUATION

class CNN(nn.Module):

    def __init__(self, num_classes, num_dense_layers, num_conv_layers):
      super(CNN, self).__init__()
      self.num_dense_layers = num_dense_layers
      self.num_conv_layers = num_conv_layers

      # single conv layer
      if num_conv_layers == 1:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
          self.fc1 = nn.Linear(6*(12**2), num_classes)
        elif num_dense_layers == 2:
          self.fc1 = nn.Linear(6*(12**2), 80)
          self.fc2 = nn.Linear(6, num_classes)
        elif num_dense_layers == 3:
          self.fc1 = nn.Linear(6*(12**2), 120)
          self.fc2 = nn.Linear(120, 84)
          self.fc3 = nn.Linear(20, num_classes)

      # two conv layers
      elif num_conv_layers == 2:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
```

```
            self.fc1 = nn.Linear(12*(4**2), num_classes)
          elif num_dense_layers == 2:
            self.fc1 = nn.Linear(12*(4**2), 80)
            self.fc2 = nn.Linear(80, num_classes)
          elif num_dense_layers == 3:
            self.fc1 = nn.Linear(12*(4**2), 120)
            self.fc2 = nn.Linear(120, 84)
            self.fc3 = nn.Linear(20, num_classes)

        # four conv layers
        elif num_conv_layers == 4:
          self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2)
          self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5, padding=2)
          self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
          self.conv3 = nn.Conv2d(in_channels=12, out_channels=16, kernel_size=5,padding=2)
          self.conv4 = nn.Conv2d(in_channels=16, out_channels=24, kernel_size=5,padding=2)
          self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

          # dense layers
          if num_dense_layers == 1:
            self.fc1 = nn.Linear(24*(7**2), num_classes)
          elif num_dense_layers == 2:
            self.fc1 = nn.Linear(24*(7**2), 80)
            self.fc2 = nn.Linear(80, num_classes)
          elif num_dense_layers == 3:
            self.fc1 = nn.Linear(24*(7**2), 120)
            self.fc2 = nn.Linear(120, 84)
            self.fc3 = nn.Linear(20, num_classes)

    def forward(self, x):

        # conv2d and maxpools
        if self.num_conv_layers == 1:
          x = self.pool1(F.relu(self.conv1(x)))
          x = x.view(-1, 6*(12**2))
        elif self.num_conv_layers == 2:
          x = self.pool1(F.relu(self.conv1(x)))
          x = self.pool2(F.relu(self.conv2(x)))
          x = x.view(-1, 12*(4**2))

        elif self.num_conv_layers == 4:
          x = F.relu(self.conv1(x))
          x = F.relu(self.conv2(x))
          x = self.pool1(x)
          x = F.relu(self.conv3(x))
          x = F.relu(self.conv4(x))
          x = self.pool2(x)
          x = x.view(-1, 24*(7**2))

        # fully-connnected layers
        if self.num_dense_layers == 1:
          x = self.fc1(x)
        if self.num_dense_layers == 2:
          x = F.relu(self.fc1(x))
          x = self.fc2(x)
        if self.num_dense_layers == 3:
```

```
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

      return x

for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):

  # pairings of hyperparameters to skip

  if (hyperparams['optimiser'] == "SGD" and hyperparams['learning_rate'] < 0.005) or
  ↪  (hyperparams['optimiser'] == "AdamW" and hyperparams['learning_rate'] > 0.005):
    continue

  # Initialise model and send to GPU

  model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])

  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
  print(device)
  model.to(device)


  # Loss & optimiser
  criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us

  if hyperparams['optimiser'] == "SGD":
    optimiser = torch.optim.SGD(model.parameters(), lr=hyperparams['learning_rate'],
    ↪  weight_decay=hyperparams['weight_decay'])
  elif hyperparams['optimiser'] == "AdamW":
    optimiser = torch.optim.AdamW(model.parameters(), lr=hyperparams['learning_rate'],
    ↪  weight_decay=hyperparams['weight_decay'])


  # training loop

  train_loss_list = []
  train_accuracy_list = []
  dev_loss_list = []
  dev_accuracy_list = []

  best_dev_loss = 9999 # initiate high value so that training doesn't immediately stop

  converged = False
  epochs_counter = 0
  epochs_to_plot = []

  while (converged == False) and (epochs_counter <= max_epochs):

    train_running_loss = 0
    train_running_corrects = 0
    train_num_of_batches = 0

    model.train()

    for data, label in fm1_trainloader:
```

```
164
165         data = data.to(device)
166         label = label.to(device)
167
168         # zero the gradients
169         optimiser.zero_grad()
170
171         # forward + backward + optimize
172         train_preds = model(data)
173         loss = criterion(train_preds, label.long())
174         loss.backward()
175         optimiser.step()
176
177         # metrics
178         train_preds_indices = torch.argmax(train_preds.data, 1)
179         train_running_corrects += (train_preds_indices == label).sum().item()
180         train_running_loss += loss.item()
181         train_num_of_batches +=1
182
183       train_accuracy = train_running_corrects/(train_num_of_batches*train_batch_size)
184       train_accuracy_list.append(train_accuracy)
185       train_loss = train_running_loss/train_num_of_batches
186       train_loss_list.append(train_loss)
187
188
189       model.eval()
190
191       with torch.no_grad():
192
193         dev_running_loss = 0
194         dev_running_corrects = 0
195         dev_num_of_batches = 0
196
197         for data, label in fm1_devloader:
198
199           data = data.to(device)
200           label = label.to(device)
201
202           # forward
203           dev_preds = model(data)
204           loss = criterion(dev_preds, label.long())
205
206           # metrics
207           dev_preds_indices = torch.argmax(dev_preds.data, 1)
208           dev_running_corrects += (dev_preds_indices == label).sum().item()
209           dev_running_loss += loss.item()
210           dev_num_of_batches +=1
211
212         dev_accuracy = dev_running_corrects/(dev_num_of_batches*dev_batch_size)
213         dev_accuracy_list.append(dev_accuracy)
214         dev_loss = dev_running_loss/dev_num_of_batches
215         dev_loss_list.append(dev_loss)
216
217         epochs_counter += 1
218         epochs_to_plot.append(epochs_counter)
219
```

```
220         print(f"epoch: {epochs_counter}, train_loss: {train_loss:.4f}, train_acc:
            ↪  {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")
221
222         # Store the current results if it is the highest validation accuracy so far
223         if dev_loss < best_dev_loss:
224             best_dev_accuracy = dev_accuracy
225             best_dev_loss = dev_loss
226             best_train_accuracy = train_accuracy
227             best_train_loss = train_loss
228             best_epoch = epochs_counter
229             best_model = model
230
231         # Early stopping criteria
232         if epochs_counter >= min_epochs:
233           if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
234             converged = True
235             print("Model Converged")
236
237
238     # Evaluate loss and accuracy on test set
239
240     with torch.no_grad():
241
242       test_running_loss = 0
243       test_running_corrects = 0
244       test_num_of_batches = 0
245
246       predictions_list = []
247       labels_list = []
248
249       for data, label in fm1_testloader:
250
251         data = data.to(device)
252         label = label.to(device)
253
254         # forward
255         test_preds = best_model(data)
256         loss = criterion(test_preds, label.long())
257
258         # metrics
259         test_preds_indices = torch.argmax(test_preds.data, 1)
260         test_running_corrects += (test_preds_indices == label).sum().item()
261         test_running_loss += loss.item()
262         test_num_of_batches +=1
263
264         labels_list.append(label.long().item())
265         predictions_list.append(test_preds_indices.item())
266
267       test_accuracy = test_running_corrects/(test_num_of_batches*test_batch_size)
268       test_loss = test_running_loss/test_num_of_batches
269
270     # Save results from epoch with best validation loss, for current run and hyperparams
271
272     hyperparams['model_index'] = run_index + 1
273     hyperparams['best_epoch'] = best_epoch
274     hyperparams['best_validation_loss'] = best_dev_loss
```

```
    hyperparams['associated_validation_accuracy'] = best_dev_accuracy
    hyperparams['associated_training_loss'] = best_train_loss
    hyperparams['associated_training_accuracy'] = best_train_accuracy
    hyperparams['test_loss'] = test_loss
    hyperparams['test_accuracy'] = test_accuracy

    print(f"\n RUN_INDEX: {run_index + 1} \n")
    print(hyperparams)

    # save the model
    torch.save(best_model,
    ↪  f"{output_path}4_conv_layer_pad_MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_devloss_{best_

    # Loss plot
    plt.plot(epochs_to_plot, train_loss_list, color='k', linestyle='-')
    plt.plot(epochs_to_plot, dev_loss_list, color='r', linestyle='-')
    plt.legend(['Training', 'Validation'], loc='upper right')
    plt.ylabel('Loss', color='k')
    plt.xlabel('Epoch', color='k')
    plt.title(f'''Loss plot (Best val loss {best_dev_loss:.4f}, Acc: {best_dev_accuracy:.4f}) \n
    Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
↪   {hyperparams['num_dense_layers']},  Batch size: {hyperparams['batch_size']},
    Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
↪   regularisation: {hyperparams['weight_decay']}''', color='k')


    ↪  plt.savefig(f"{output_path}4_conv_layer_pad_MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_de
    ↪  dpi=300, bbox_inches = "tight")
    plt.show()

    # Accuracy plot
    plt.plot(epochs_to_plot, train_accuracy_list, color='k', linestyle='-')
    plt.plot(epochs_to_plot, dev_accuracy_list, color='r', linestyle='-')
    plt.legend(['Training', 'Validation'], loc='lower right')
    plt.ylabel('Accuracy', color='k')
    plt.xlabel('Epoch', color='k')
    plt.title(f'''Accuracy plot (Best val loss {best_dev_loss:.4f}, Acc:
    ↪  {best_dev_accuracy:.4f}) \n
    Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
↪   {hyperparams['num_dense_layers']},  Batch size: {hyperparams['batch_size']},
    Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
↪   regularisation: {hyperparams['weight_decay']}''', color='k')


    ↪  plt.savefig(f"{output_path}4_conv_layer_pad_MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_de
    ↪  dpi=300, bbox_inches = "tight")
    plt.show()

with torch.no_grad():

    # retrieve best model
    best_model =
    ↪  torch.load(f"{output_path}4_conv_layer_pad_MODEL1_0.0498_devloss_0.9864_devacc.pth",
    ↪  map_location=torch.device('cpu'))
    best_model.eval()

```

```
    # get list of layers of best model
    layers = list(best_model.children())
    num_layers = len(layers)

    # VISUALISING FILTERS

    # Save weights in convolutional layers to plot filters
    filter_weights = []
    for i in range(num_layers):
    if type(layers[i]) == nn.Conv2d:
        filter_weights.append(layers[i].weight)

    # Visualise the filters (for the last convolutional layer in this case)
    i = 0
    filters_per_channel = 2
    plt.figure(figsize=(20, 20))
    for filter in filter_weights[3]:
        for j in range(filters_per_channel):
            i += 1
            plt.subplot(8, 6, i)
            plt.imshow(filter[j, :, :].detach().cpu(), cmap='gray')
            plt.axis('off')
    # plt.savefig(f"{output_path}filters layer 4", dpi=300, bbox_inches = "tight")
    plt.show()

    # VISUALISING FEATURE MAPS

    # Get first batch from test loader (batch size = 1)
    it = iter(fm1_testloader)
    test_input_tensor = next(it)
    test_input_label = test_input_tensor[1]

    while test_input_label[0] != 4: # Get test input of a certain type (4 is a bag, for
    ↪  example)
      test_input_tensor = next(it)
      test_input_label = test_input_tensor[1]

    test_input = test_input_tensor[0]

    # Extract only the convolutional (and max-pool) layers ie. leaving out final 2 dense
    ↪  layers
    conv_layers = layers[:-2]
    num_conv_layers = len(conv_layers)

    # Pass input into first layer
    test_outputs = [conv_layers[0](test_input)]
    for i in range(1, len(conv_layers)):
        # Pass the output from the last layer to the next layer and save
        test_outputs.append(conv_layers[i](test_outputs[-1]))

    # Remove outputs of max-pool layers ie. only keep convolutional layers
    del(test_outputs[2])
    del(test_outputs[-1])

    # Visualise and save the feature maps for each layer
    plt.figure(figsize=(50, 50))
```

```
    for layer_num in range(len(test_outputs)):
        layer_out = test_outputs[layer_num][0, :, :, :] # outputs from first input channel
        ↪  only (for simplicity)
        # Plot feature map for first 6 filters in this layer
        for i in range(1,7):
            plt.subplot(4, 6, i+6*layer_num)
            plt.imshow(layer_out[i-1], cmap='gray')
            plt.axis("off")
    # plt.savefig(f"{output_path}Q3-bag_feature_maps.png")
    plt.show()
```

Appendix E Q4 code

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

    encoder_layers = []
    decoder_layers = []

    # Encoder
    for l in structure:
      encoder_layers.append(nn.Linear(l[0], l[1]))
      encoder_layers.append(nn.ReLU(True))
    # Decoder
    for index, l in reversed(list(enumerate(structure))):
      decoder_layers.append(nn.Linear(l[1], l[0]))
      # Last index, thus need Sigmoid
      if (index == 0):
        decoder_layers.append(nn.Sigmoid())
      else:
        decoder_layers.append(nn.ReLU(True))

    self.encoder_struct = nn.Sequential(*encoder_layers)
    self.decoder_struct = nn.Sequential(*decoder_layers)

  def forward(self, x):
    x = self.encoder_struct(x)
    x = self.decoder_struct(x)
    return x

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

    layers = []
    for index, l in enumerate(structure):
      layers.append(nn.Linear(l[0], l[1]))
      if (index == (len(structure) - 1)):
        layers.append(nn.Sigmoid())
      else:
        layers.append(nn.ReLU(True))

    self.struct = nn.Sequential(*layers)

  def forward(self, x):
    x = self.struct(x)
    return x

encoder_dataset = torch.utils.data.ConcatDataset([fm2_train, fm2_val, fm2_test, fm1_train])

def train_autoencoder():
  for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):
    train_batch_size = hyperparams['batch_size']
    dataloader = torch.utils.data.DataLoader(encoder_dataset, batch_size = train_batch_size,
    ↪  shuffle = True)

```

```
54
55      fm1_trainloader = torch.utils.data.DataLoader(fm1_train, batch_size = train_batch_size,
        ↪  shuffle = True)
56
57      model = VanillaAutoEncoder(hyperparams['structure']).to(device)
58      c = nn.MSELoss()
59      optimizer = optim.Adam(model.parameters(), lr = hyperparams['learning_rate'])
60
61      current_epoch = 0
62      max_epochs = 200
63      min_epochs = 5
64      running_loss = 0
65      running_validation_loss = 0
66      epoch_loss = 0
67
68      best_loss = 100000
69
70      loss_list = []
71      validation_loss_list = []
72      converged = False
73
74      while (converged == False) and (current_epoch <= max_epochs):
75        current_epoch += 1
76        model.train()
77
78        for d, label in dataloader:
79          d = d.view(-1, 784).type(torch.FloatTensor)
80          d = d.to(device)
81
82          optimizer.zero_grad()
83          output = model(d)
84          loss = c(output, d)
85          loss.backward()
86          optimizer.step()
87
88          running_loss += loss.item()
89
90        loss_list.append(running_loss / len(encoder_dataset))
91        if (loss_list[-1] < best_loss):
92          best_loss = loss_list[-1]
93          best_model = model
94          best_hyperparams = hyperparams
95
96        running_loss = 0
97
98        if current_epoch >= min_epochs:
99          if (np.mean(loss_list[-3:]) - np.mean(loss_list[-6:-3])) > -0.001:
100           converged = True
101           print('Model Converged')
102           print('Epoch ', current_epoch, ' Loss ', loss_list[-1])
103
104   return loss_list
105
106 def random_MLP():
107   # Split the dataset
108   sizes = [0.5]
```

```
109   len_dataset = len(fm1_train)
110   dataaa = torch.utils.data.DataLoader(fm1_train)
111
112   for dataset_sizes in sizes:
113     print('Dataset Size ', dataset_sizes)
114     split_size = int(len_dataset * dataset_sizes)
115     fm1_train_split, other_split = torch.utils.data.random_split(fm1_train, [split_size,
        ↪  (len_dataset - split_size)])
116     dataloader = torch.utils.data.DataLoader(fm1_train_split, batch_size=32, shuffle = True)
117     dataloader_val = torch.utils.data.DataLoader(fm1_val, batch_size=32, shuffle = True)
118     dataloader_test = torch.utils.data.DataLoader(fm1_test, batch_size=32, shuffle=True)
119
120     # Create Model
121     mlp_model = MLP(struct).to(device)
122     c = nn.CrossEntropyLoss()
123     optimizer = optim.Adam(mlp_model.parameters(), lr = 0.01)
124
125     current_epoch = 0
126     max_epochs = 15
127     min_epochs = 5
128
129     running_loss = 0
130     val_running_loss = 0
131     test_running_loss = 0
132
133     best_loss = 10000
134
135     train_loss = []
136     val_loss = []
137     test_loss = []
138
139     while (current_epoch <= max_epochs):
140       current_epoch += 1
141       mlp_model.train()
142
143       for d, label in dataloader:
144         d = d.view(-1, 784).type(torch.FloatTensor)
145         d = d.to(device)
146         label = label.to(device)
147
148         optimizer.zero_grad()
149         output = mlp_model(d)
150         loss = c(output, label)
151         loss.backward()
152         optimizer.step()
153
154         running_loss += loss.item()
155
156       print('Epoch ', current_epoch, ' loss ', (running_loss / len(fm1_train_split)))
157       train_loss.append(running_loss / len(fm1_train_split))
158       running_loss = 0
159
160       mlp_model.eval()
161       with torch.no_grad():
162         for d, l in dataloader_val:
163           d = d.view(-1, 784).type(torch.FloatTensor)
```

```
            d = d.to(device)
            l = l.to(device)

            v_output = mlp_model(d)
            v_loss = c(v_output, l)
            val_running_loss += v_loss.item()

        # print('Validation loss ', (val_running_loss / len(fm1_val)))
        val_loss.append((val_running_loss / len(fm1_val)))
        val_running_loss = 0

        for d, l in dataloader_test:
            d = d.view(-1, 784).type(torch.FloatTensor)
            d = d.to(device)
            l = l.to(device)

            t_output = mlp_model(d)
            t_loss = c(t_output, l)
            test_running_loss += t_loss.item()

        test_loss.append((test_running_loss / len(fm1_test)))
        test_running_loss = 0

    # return errors, val_errors
    return train_loss, val_loss, test_loss

def weights_init(m):
    global index
    if type(m) == nn.Linear:
        m.weight = torch.nn.Parameter(model.state_dict()['encoder_struct.' + str(index) +
        ↪  '.weight'])
        m.bias = torch.nn.Parameter(model.state_dict()['encoder_struct.' + str(index) + '.bias'])
        index += 2

def weighted_MLP():
    # mlp_model = MLP(struct).to(device)
    # mlp_model.apply(weights_init)

    sizes = [1.0]
    len_dataset = len(fm1_train)
    dataaa = torch.utils.data.DataLoader(fm1_train)

    errors = []
    train_errors = []
    val_errors = []
    test_errors = []

    for dataset_sizes in sizes:
        print('Dataset Size ', dataset_sizes)
        split_size = int(len_dataset * dataset_sizes)
        fm1_train_split, other_split = torch.utils.data.random_split(fm1_train, [split_size,
        ↪  (len_dataset - split_size)])
        dataloader = torch.utils.data.DataLoader(fm1_train_split, batch_size=32, shuffle = True)
        dataloader_val = torch.utils.data.DataLoader(fm1_val, batch_size=32, shuffle = True)
        dataloader_test = torch.utils.data.DataLoader(fm1_test, batch_size=32, shuffle=True)

```

```
    # Create Model
    global index
    index = 0
    mlp_model = MLP(struct).to(device)
    mlp_model.apply(weights_init).to(device)

    c = nn.CrossEntropyLoss()
    optimizer = optim.Adam(mlp_model.parameters(), lr = 0.01)

    current_epoch = 0
    max_epochs = 15
    min_epochs = 5

    running_loss = 0
    val_running_loss = 0
    test_running_loss = 0

    best_loss = 10000
    this_loss = []

    while (current_epoch <= max_epochs):
      current_epoch += 1
      mlp_model.train()

      for d, label in dataloader:
        d = d.view(-1, 784).type(torch.FloatTensor)
        d = d.to(device)
        label = label.to(device)

        optimizer.zero_grad()
        output = mlp_model(d)
        loss = c(output, label)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

      print('Epoch ', current_epoch, ' loss ', (running_loss / len(fm1_train_split)))
      this_loss.append(running_loss / len(fm1_train_split))
      train_errors.append(running_loss / len(fm1_train_split))
      running_loss = 0

      mlp_model.eval()
      with torch.no_grad():
        for d, l in dataloader_val:
          d = d.view(-1, 784).type(torch.FloatTensor)
          d = d.to(device)
          l = l.to(device)

          v_output = mlp_model(d)
          v_loss = c(v_output, l)
          val_running_loss += v_loss.item()
        print('Validation loss ', (val_running_loss / len(fm1_val)))
        val_errors.append((val_running_loss / len(fm1_val)))
        val_running_loss = 0

```

```
        for d, l in dataloader_test:
          d = d.view(-1, 784).type(torch.FloatTensor)
          d = d.to(device)
          l = l.to(device)

          t_output = mlp_model(d)
          t_loss = c(t_output, l)
          test_running_loss += t_loss.item()

        test_errors.append((test_running_loss / len(fm1_test)))
        test_running_loss = 0

    errors.append(this_loss[-1])

  return train_errors, val_errors, test_errors
```

Appendix F Q5 code

```
for i in range(5, 105, 5):

  # specify model architecture
  # PyTorch model class, which takes the specified hyperparameters as inputs
  class CNN(nn.Module):

    def __init__(self, num_classes, num_dense_layers, num_conv_layers):
      super(CNN, self).__init__()
      self.num_dense_layers = num_dense_layers
      self.num_conv_layers = num_conv_layers

      # single conv layer
      if num_conv_layers == 1:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
          self.fc1 = nn.Linear(6*(12**2), num_classes)
        elif num_dense_layers == 2:
          self.fc1 = nn.Linear(6*(12**2), 80)
          self.fc2 = nn.Linear(80, num_classes)
        elif num_dense_layers == 3:
          self.fc1 = nn.Linear(6*(12**2), 120)
          self.fc2 = nn.Linear(120, 84)
          self.fc3 = nn.Linear(84, num_classes)

      # two conv layers
      elif num_conv_layers == 2:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # dense layers
        if num_dense_layers == 1:
          self.fc1 = nn.Linear(12*(4**2), num_classes)
        elif num_dense_layers == 2:
          self.fc1 = nn.Linear(12*(4**2), 80)
          self.fc2 = nn.Linear(80, num_classes)
        elif num_dense_layers == 3:
          self.fc1 = nn.Linear(12*(4**2), 120)
          self.fc2 = nn.Linear(120, 84)
          self.fc3 = nn.Linear(84, num_classes)

    def forward(self, x):

      # conv2d and maxpools
      if self.num_conv_layers == 1:
        x = self.pool1(F.relu(self.conv1(x)))
        x = x.view(-1, 6*(12**2))
      elif self.num_conv_layers == 2:
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
```

```
        x = x.view(-1, 12*(4**2))

    # fully-connnected layers
    if self.num_dense_layers == 1:
      x = self.fc1(x)
    if self.num_dense_layers == 2:
      x = F.relu(self.fc1(x))
      x = self.fc2(x)
    if self.num_dense_layers == 3:
      x = F.relu(self.fc1(x))
      x = F.relu(self.fc2(x))
      x = self.fc3(x)

    return x

# Breaking the training set into smaller fractions
percentage_to_use = int((i/100) * 25000)
percentage_not_to_use = 25000 - percentage_to_use

fm1_train_percentage, rest = torch.utils.data.random_split(fm1_train, [percentage_to_use,
↪  percentage_not_to_use])
fm1_trainloader_percentage = torch.utils.data.DataLoader(fm1_train_percentage, batch_size =
↪  train_batch_size, shuffle = True)

# Initialise the random model and send to GPU

model_ran = CNN(num_classes, hyperparams['num_dense_layers'],
↪  hyperparams['num_conv_layers'])

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
model_ran.to(device)


# Loss & optimiser
criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us

optimiser = torch.optim.AdamW(model_ran.parameters(), lr=hyperparams['learning_rate'],
↪  weight_decay=hyperparams['weight_decay'])


# training loop

train_loss_list_ran = []
train_accuracy_list_ran = []
dev_loss_list_ran = []
dev_accuracy_list_ran = []

best_dev_loss_ran = 9999 # initiate high value so that training doesn't immediately stop

converged = False
epochs_counter = 0
epochs_to_plot_ran = []

while (converged == False) and (epochs_counter <= max_epochs):

```

```
    train_running_loss = 0
    train_running_corrects = 0
    train_num_of_batches = 0

    model_ran.train()

    for data, label in fm1_trainloader_percentage:

      data = data.to(device)
      label = label.to(device)

      # zero the gradients
      optimiser.zero_grad()

      # forward + backward + optimize
      train_preds = model_ran(data)
      loss = criterion(train_preds, label.long())
      loss.backward()
      optimiser.step()

      # metrics
      train_preds_indices = torch.argmax(train_preds.data, 1)
      train_running_corrects += (train_preds_indices == label).sum().item()
      train_running_loss += loss.item()
      train_num_of_batches +=1

    train_accuracy = train_running_corrects/(train_num_of_batches*train_batch_size)
    train_accuracy_list_ran.append(train_accuracy)
    train_loss = train_running_loss/train_num_of_batches
    train_loss_list_ran.append(train_loss)

    # Validation Loop

    model_ran.eval()

    with torch.no_grad():

      dev_running_loss = 0
      dev_running_corrects = 0
      dev_num_of_batches = 0

      for data, label in fm1_devloader:

        data = data.to(device)
        label = label.to(device)

        # forward
        dev_preds = model_ran(data)
        loss = criterion(dev_preds, label.long())

        # metrics
        dev_preds_indices = torch.argmax(dev_preds.data, 1)
        dev_running_corrects += (dev_preds_indices == label).sum().item()
        dev_running_loss += loss.item()
        dev_num_of_batches +=1

```

```
        dev_accuracy = dev_running_corrects/(dev_num_of_batches*dev_batch_size)
        dev_accuracy_list_ran.append(dev_accuracy)
        dev_loss = dev_running_loss/dev_num_of_batches
        dev_loss_list_ran.append(dev_loss)

        epochs_counter += 1
        epochs_to_plot_ran.append(epochs_counter)

        print(f"epoch: {epochs_counter}, train_loss: {train_loss:.4f}, train_acc:
        ↪  {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")

        # Store the current results if it has the smallest validation loss so far
        if dev_loss < best_dev_loss_ran:
            best_dev_accuracy_ran = dev_accuracy
            best_dev_loss_ran = dev_loss
            best_train_accuracy_ran = train_accuracy
            best_train_loss_ran = train_loss
            best_epoch_ran = epochs_counter
            best_model = model_ran

        # Early stopping criteria
        if epochs_counter >= min_epochs:
          if (np.mean(dev_loss_list_ran[-20:]) - np.mean(dev_loss_list_ran[-40:-20])) > 0.001:
            converged = True
            print("Model Converged")

    # Evaluate loss and accuracy on test set

    with torch.no_grad():

      test_running_loss = 0
      test_running_corrects = 0
      test_num_of_batches = 0

      predictions_list = []
      labels_list = []

      for data, label in fm1_testloader:

        data = data.to(device)
        label = label.to(device)

        # forward
        test_preds = best_model(data)
        loss = criterion(test_preds, label.long())

        # metrics
        test_preds_indices = torch.argmax(test_preds.data, 1)
        test_running_corrects += (test_preds_indices == label).sum().item()
        test_running_loss += loss.item()
        test_num_of_batches +=1

        labels_list.append(label.long().item())
        predictions_list.append(test_preds_indices.item())

      test_accuracy_ran = test_running_corrects/(test_num_of_batches*test_batch_size)
```

```
      test_loss_ran = test_running_loss/test_num_of_batches

    print(f"BEST RANDOM INITIALISATION MODEL TEST ACCURACY: {test_accuracy_ran}")
    print(f"BEST RANDOM INITIALISATION MODEL TEST LOSS: {test_loss_ran}")

    # Save results from epoch with best validation loss

    hyperparams['best_epoch'] = best_epoch_ran
    hyperparams['best_validation_loss'] = best_dev_loss_ran
    hyperparams['associated_validation_accuracy'] = best_dev_accuracy_ran
    hyperparams['associated_training_loss'] = best_train_loss_ran
    hyperparams['associated_training_accuracy'] = best_train_accuracy_ran
    hyperparams['test_loss'] = test_loss_ran
    hyperparams['test_accuracy'] = test_accuracy_ran
    hyperparams['Fraction of data (%)'] = i


    results_df = results_df.append(hyperparams, ignore_index=True)
    results_df = results_df.sort_values("best_validation_loss", ascending=False).round(4)
    results_df.to_csv(f"{output_path}RESULTS_TABLE_random_initial.csv")

    print(hyperparams)

    # save the model
    torch.save(best_model,
    ↪  f"{output_path}MODEL_{best_dev_loss_ran:.4f}_devloss_{best_dev_accuracy_ran:.4f}_devacc_fmnist1.pth"

##############################################
##############################################

    # Load Model
    model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])
    weights_path = f"{output_path}MODEL_0.1715_devloss_0.9404_devacc_state_dict.pth"
    model.load_state_dict(torch.load(weights_path, ))
    for param in model.parameters():
      param.requires_grad = False

    # Replace the last fully-connected layer
    # Parameters of newly constructed modules have requires_grad=True by default
    model.fc3 = nn.Linear(84, num_classes)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(device)
    model.to(device)


    # Loss & optimiser
    criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us

    optimiser = torch.optim.AdamW(model.parameters(), lr=hyperparams['learning_rate'],
    ↪  weight_decay=hyperparams['weight_decay'])


    # training loop

    train_loss_list = []
```

```
train_accuracy_list = []
dev_loss_list = []
dev_accuracy_list = []

best_dev_loss = 9999 # initiate high value so that training doesn't immediately stop

converged = False
epochs_counter = 0
epochs_to_plot = []

while (converged == False) and (epochs_counter <= max_epochs):

  train_running_loss = 0
  train_running_corrects = 0
  train_num_of_batches = 0

  model.train()

  for data, label in fm1_trainloader_percentage:

    data = data.to(device)
    label = label.to(device)

    # zero the gradients
    optimiser.zero_grad()

    # forward + backward + optimize
    train_preds = model(data)
    loss = criterion(train_preds, label.long())
    loss.backward()
    optimiser.step()

    # metrics
    train_preds_indices = torch.argmax(train_preds.data, 1)
    train_running_corrects += (train_preds_indices == label).sum().item()
    train_running_loss += loss.item()
    train_num_of_batches +=1

  train_accuracy = train_running_corrects/(train_num_of_batches*train_batch_size)
  train_accuracy_list.append(train_accuracy)
  train_loss = train_running_loss/train_num_of_batches
  train_loss_list.append(train_loss)


  model.eval()

  with torch.no_grad():

    dev_running_loss = 0
    dev_running_corrects = 0
    dev_num_of_batches = 0

    for data, label in fm1_devloader:

      data = data.to(device)
      label = label.to(device)
```

```
328
329         # forward
330         dev_preds = model(data)
331         loss = criterion(dev_preds, label.long())
332
333         # metrics
334         dev_preds_indices = torch.argmax(dev_preds.data, 1)
335         dev_running_corrects += (dev_preds_indices == label).sum().item()
336         dev_running_loss += loss.item()
337         dev_num_of_batches +=1
338
339       dev_accuracy = dev_running_corrects/(dev_num_of_batches*dev_batch_size)
340       dev_accuracy_list.append(dev_accuracy)
341       dev_loss = dev_running_loss/dev_num_of_batches
342       dev_loss_list.append(dev_loss)
343
344       epochs_counter += 1
345       epochs_to_plot.append(epochs_counter)
346
347       print(f"epoch: {epochs_counter}, train_loss: {train_loss:.4f}, train_acc:
          ↪  {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")
348
349       # Store the current results if it has the smallest validation loss so far
350       if dev_loss < best_dev_loss:
351           best_dev_accuracy = dev_accuracy
352           best_dev_loss = dev_loss
353           best_train_accuracy = train_accuracy
354           best_train_loss = train_loss
355           best_epoch = epochs_counter
356           best_model = model
357
358       # Early stopping criteria
359       if epochs_counter >= min_epochs:
360         if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
361           converged = True
362           print("Model Converged")
363
364   # Evaluate loss and accuracy on test set
365
366   with torch.no_grad():
367
368     test_running_loss = 0
369     test_running_corrects = 0
370     test_num_of_batches = 0
371
372     predictions_list = []
373     labels_list = []
374
375     for data, label in fm1_testloader:
376
377       data = data.to(device)
378       label = label.to(device)
379
380       # forward
381       test_preds = best_model(data)
382       loss = criterion(test_preds, label.long())
```

```
        # metrics
        test_preds_indices = torch.argmax(test_preds.data, 1)
        test_running_corrects += (test_preds_indices == label).sum().item()
        test_running_loss += loss.item()
        test_num_of_batches +=1

        labels_list.append(label.long().item())
        predictions_list.append(test_preds_indices.item())

      test_accuracy = test_running_corrects/(test_num_of_batches*test_batch_size)
      test_loss = test_running_loss/test_num_of_batches

    print(f"BEST MODEL TEST ACCURACY: {test_accuracy}")
    print(f"BEST MODEL TEST LOSS: {test_loss}")

    # Save results from epoch with best validation loss

    hyperparams['best_epoch'] = best_epoch
    hyperparams['best_validation_loss'] = best_dev_loss
    hyperparams['associated_validation_accuracy'] = best_dev_accuracy
    hyperparams['associated_training_loss'] = best_train_loss
    hyperparams['associated_training_accuracy'] = best_train_accuracy
    hyperparams['test_loss'] = test_loss
    hyperparams['test_accuracy'] = test_accuracy
    hyperparams['Fraction of data (%)'] = i


    results_df = results_df.append(hyperparams, ignore_index=True)
    results_df = results_df.sort_values("best_validation_loss", ascending=False).round(4)
    results_df.to_csv(f"{output_path}RESULTS_TABLE_TL.csv")


    print(hyperparams)

    # save the model
    torch.save(best_model,
     ↪  f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc_fmnist1.pth")

    # Loss plot
    plt.plot(epochs_to_plot_ran, train_loss_list_ran, color='k', linestyle='-')
    plt.plot(epochs_to_plot_ran, dev_loss_list_ran, color='r', linestyle='-')
    plt.plot(epochs_to_plot, train_loss_list, color='b', linestyle='-')
    plt.plot(epochs_to_plot, dev_loss_list, color='orange', linestyle='-')
    plt.legend(['Training (RI)', 'Validation (RI)', 'Training (TL)', 'Validation (TL)'],
     ↪  loc='upper right')
    plt.ylabel('Loss', color='k')
    plt.xlabel('Epoch', color='k')
    plt.title(f'''Loss plot (Best val loss (RI) {best_dev_loss_ran:.4f}, Acc (RI):
     ↪  {best_dev_accuracy_ran:.4f}, Best val loss (TL) {best_dev_loss:.4f}, Acc (TL):
     ↪  {best_dev_accuracy:.4f}) \n
    Fraction of data use for training: {i}% \n
    Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
  ↪  {hyperparams['num_dense_layers']},  Batch size: {hyperparams['batch_size']},
    Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
  ↪  regularisation: {hyperparams['weight_decay']}''', color='k')
```

```
433
434
    ↪ plt.savefig(f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc_LOSSPLOT
    ↪ dpi=300, bbox_inches = "tight")
435 plt.show()
```