

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
ucabddd3@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	3
1 Q1	4
1.1 Model implementation	4
1.2 Final training	5
1.3 Loss plot	5
1.4 Final accuracy	6
1.5 Confusion matrix	6
2 Q2	8
2.1 Model implementation	8
2.2 Hyperparameter optimisation	9
2.3 Lowest loss	10
3 Q3	16
3.1 Model implementation	16
3.2 Final training	18
3.3 Filters and feature maps	19
3.4 Qualitative discussion	26
4 Q4	27
4.1 Autoencoder implementation	27
4.2 Autoencoder final training	27
4.3 MLP implementation	27
4.4 Pre-training	27
4.5 Final accuracy	28
5 Q5	29
5.1 Model implementation, FMNIST2	29
5.2 Hyperparameter optimisation	30
5.3 Model implementation, FMNIST1	30
5.4 Transfer learning	31
5.5 Final accuracy	32
Appendix A Data preparation code	33
Appendix B Q1 code	36
Appendix C Q2 code	42
Appendix D Q3 code	48
Appendix E Q4 code	56
Appendix F Q5 code	62

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(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(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-connected 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.

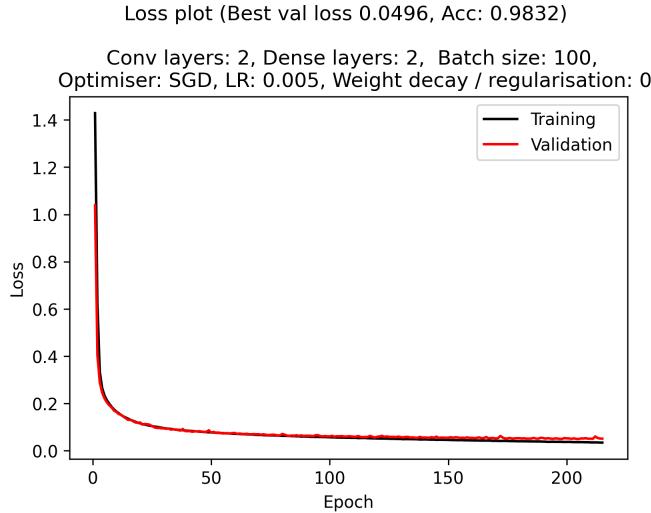


Figure 1: Training and validation loss by epoch for the CNN multiclass classification model

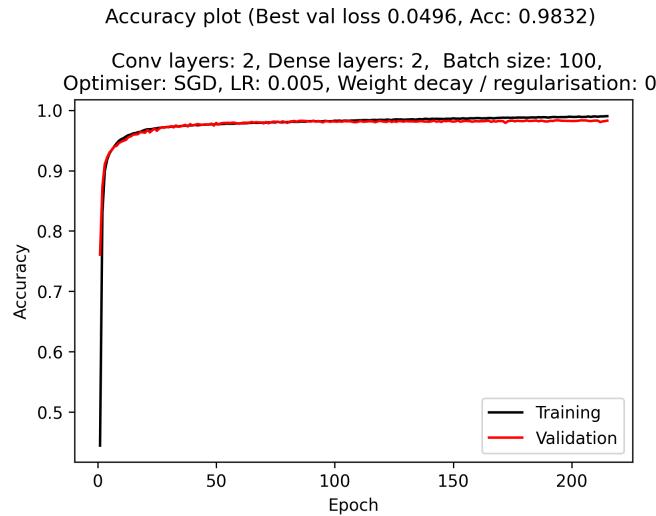


Figure 2: Training and validation accuracy by epoch for the CNN multiclass classification model

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 prominent 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.

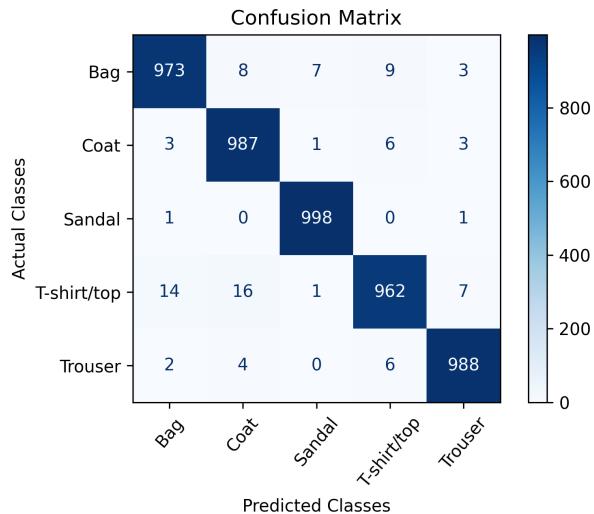


Figure 3: Confusion matrix for the CNN multiclass classification model

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-connected 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 3rd 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 5th 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.

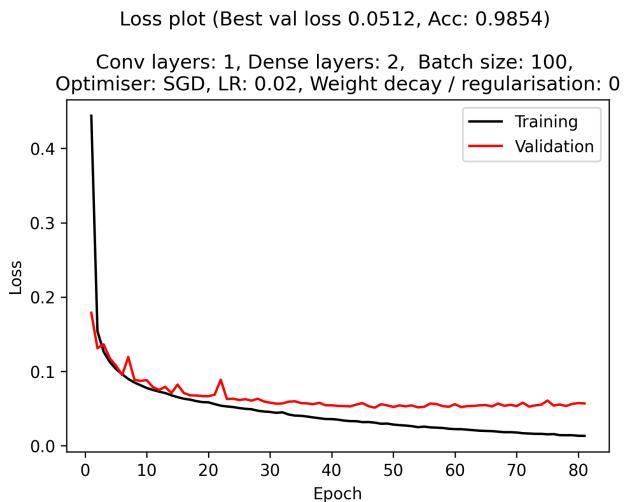
2.3 Lowest loss

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.

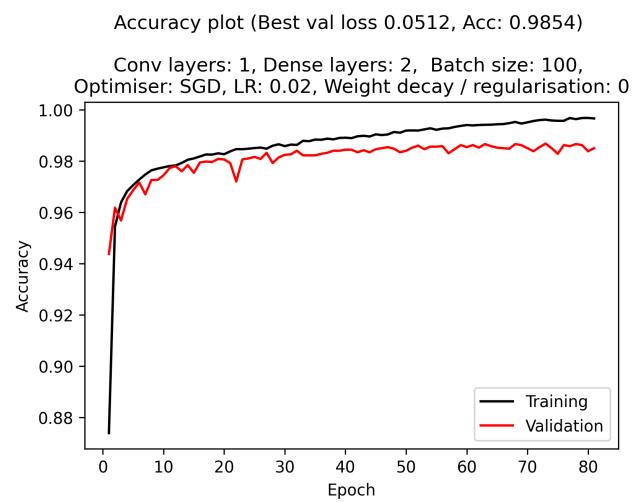
As the classes are balanced within each dataset, we might have expected the models to have improved validation performance with each iteration as we hand picked hyperparameters to try to improve validation performance, but without necessarily an associated improvement in test performance. This would 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.

Model	Layers						Loss			Accuracy		
	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	

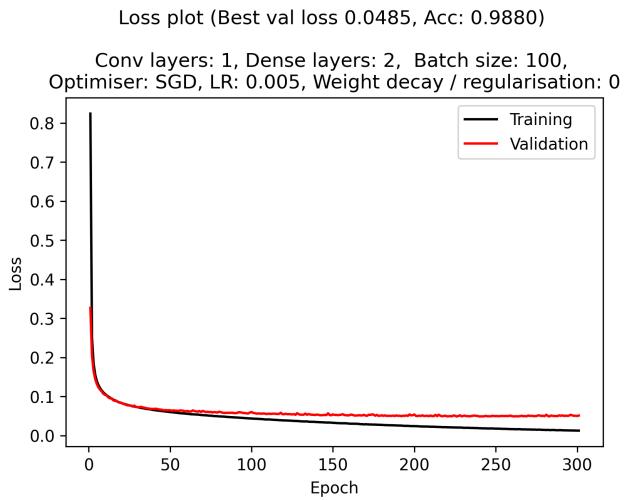
Table 3: The training, validation, and test loss and accuracy for ten iterations of hyperparameter changes



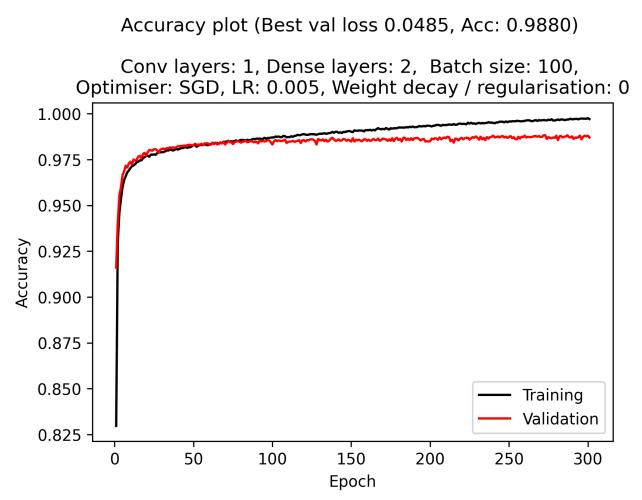
(a) Model 1: training and validation loss by epoch



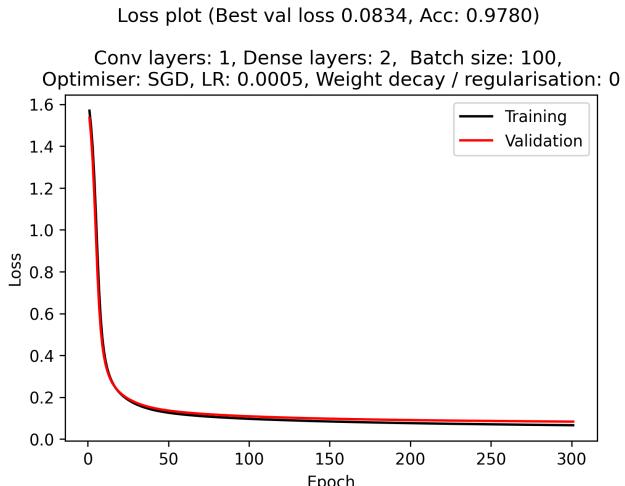
(b) Model 1: training and validation accuracy by epoch



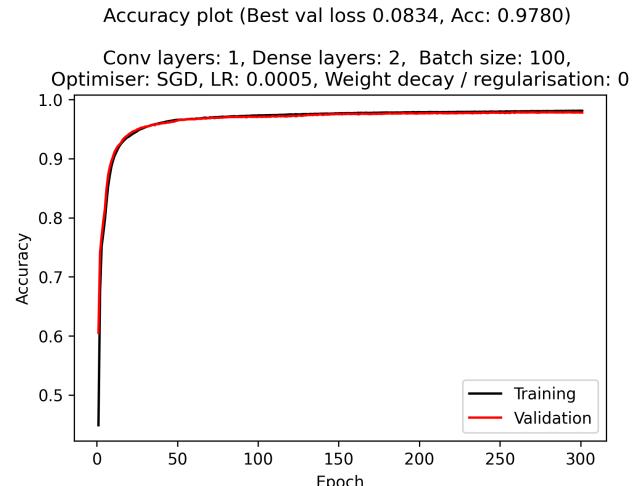
(c) Model 2: training and validation loss by epoch



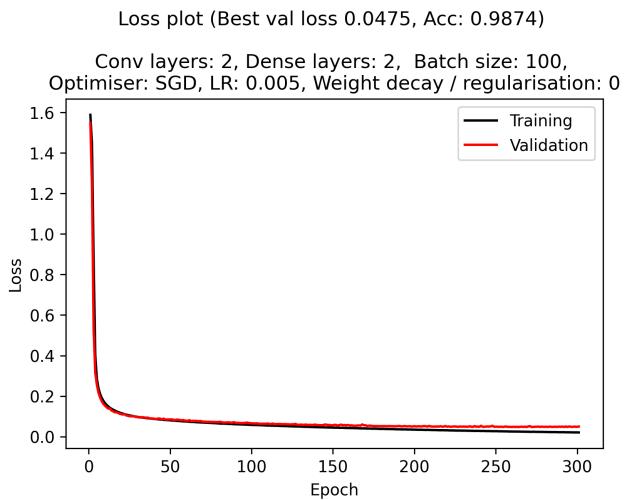
(d) Model 2: training and validation accuracy by epoch



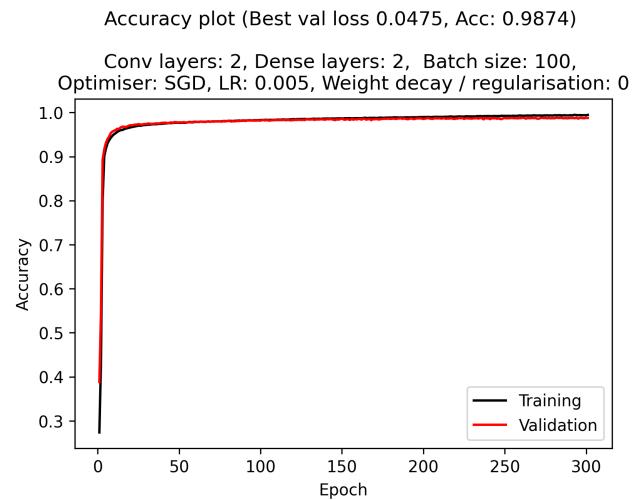
(e) Model 3: training and validation loss by epoch



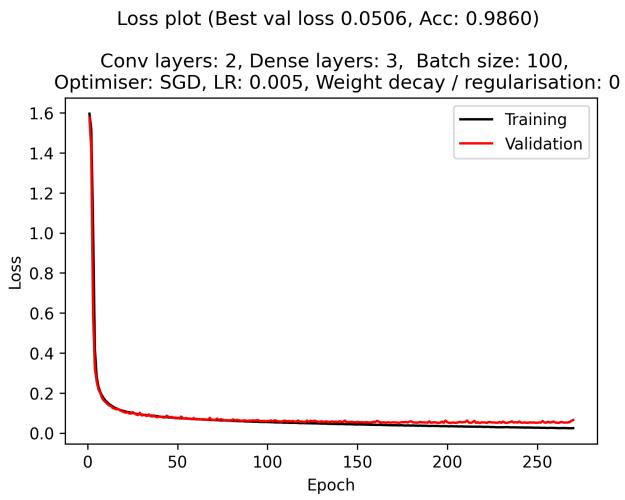
(f) Model 3: training and validation accuracy by epoch



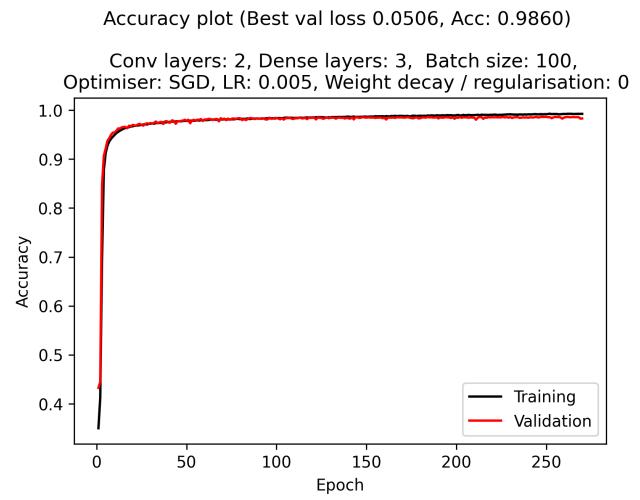
(g) Model 4: training and validation loss by epoch



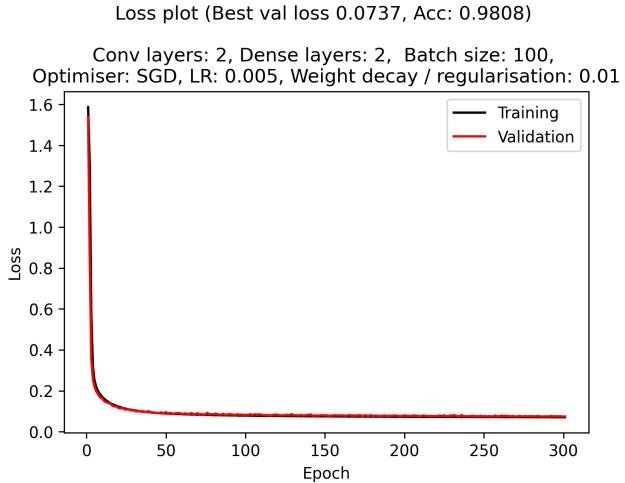
(h) Model 4: training and validation accuracy by epoch



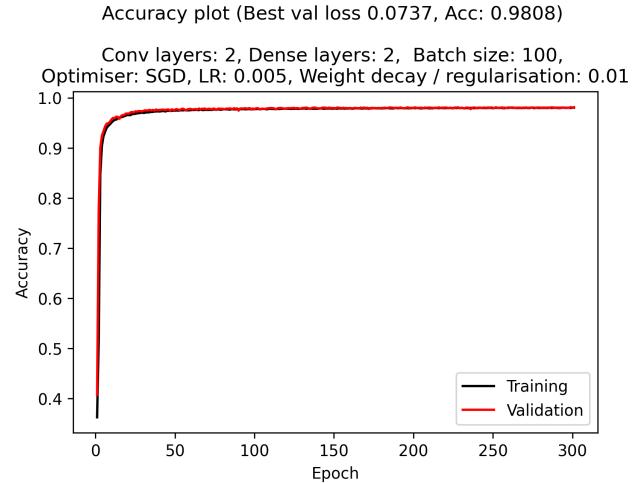
(i) Model 5: training and validation loss by epoch



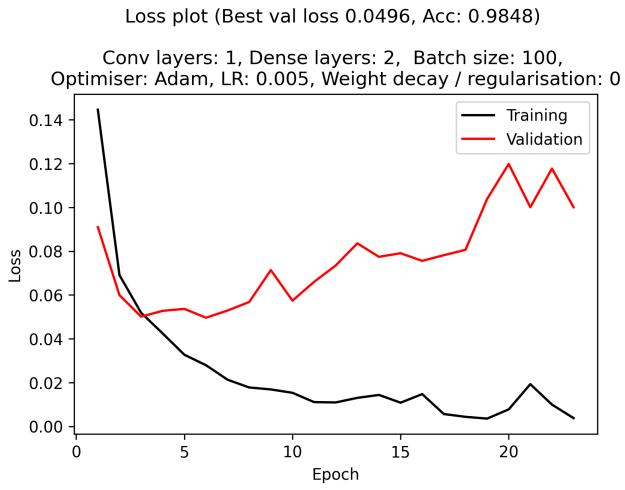
(j) Model 5: training and validation accuracy by epoch



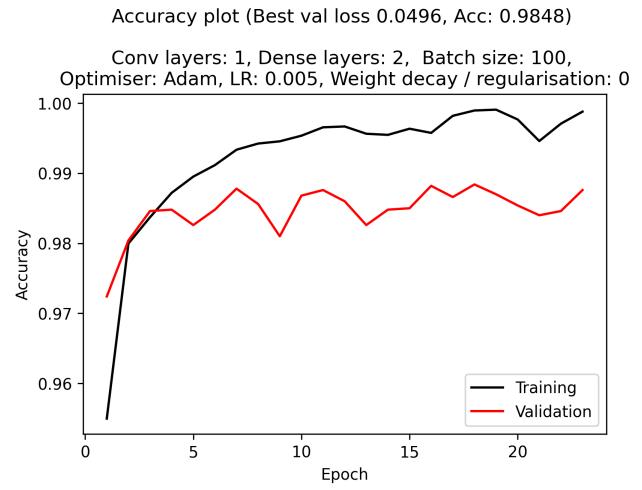
(k) Model 6: training and validation loss by epoch



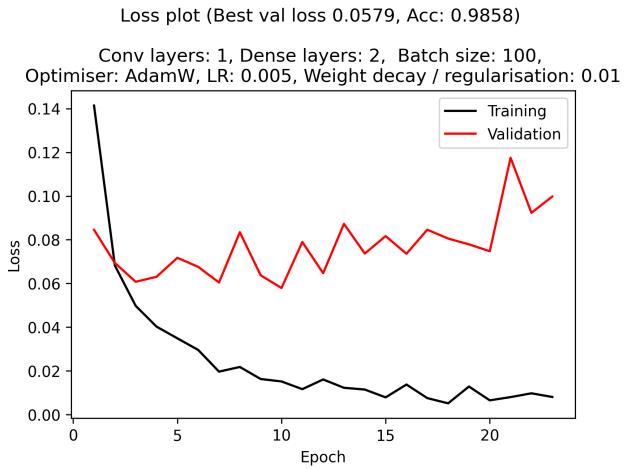
(l) Model 6: training and validation accuracy by epoch



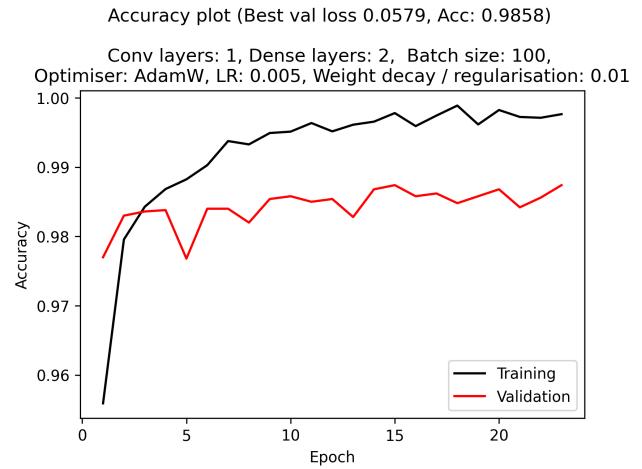
(m) Model 7: training and validation loss by epoch



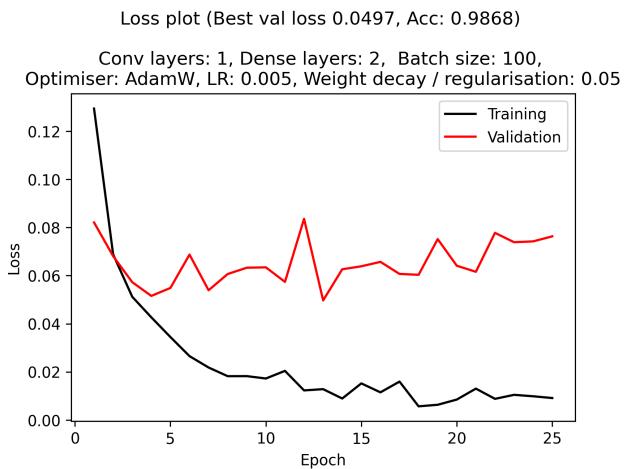
(n) Model 7: training and validation accuracy by epoch



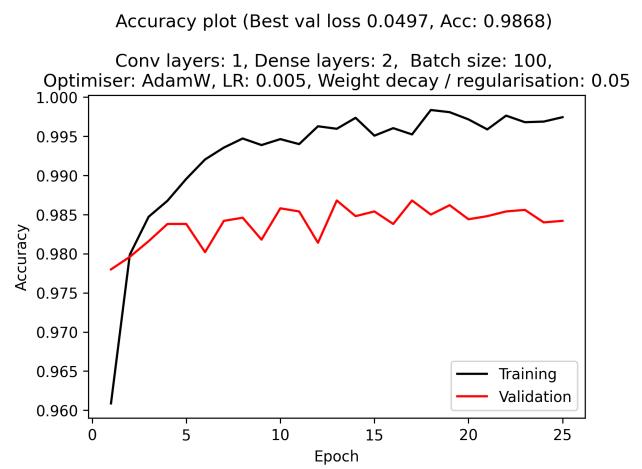
(o) Model 8: training and validation loss by epoch



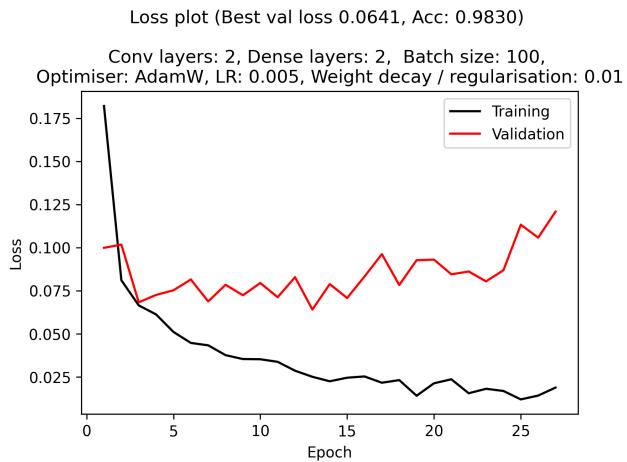
(p) Model 8: training and validation accuracy by epoch



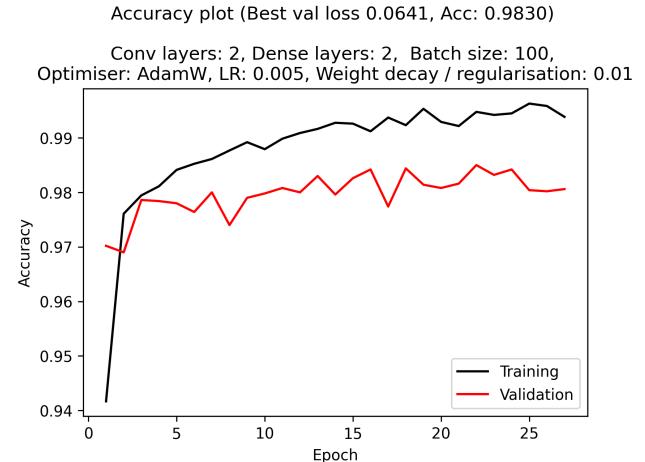
(q) Model 9: training and validation loss by epoch



(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(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(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-connected 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.

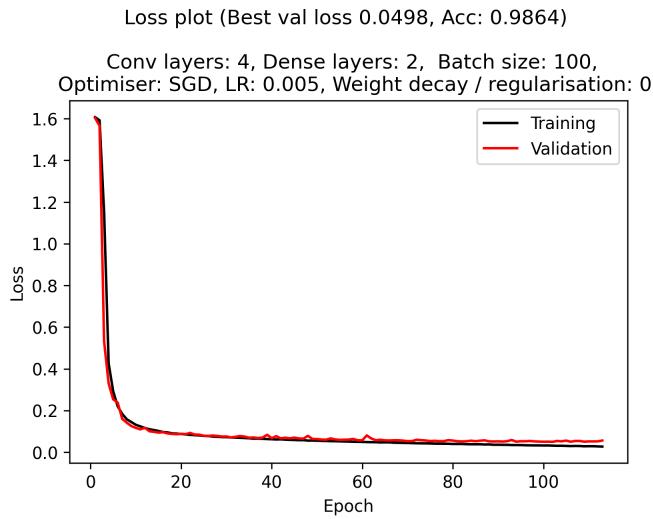


Figure 5: Training and validation loss by epoch for the CNN multiclass classification model

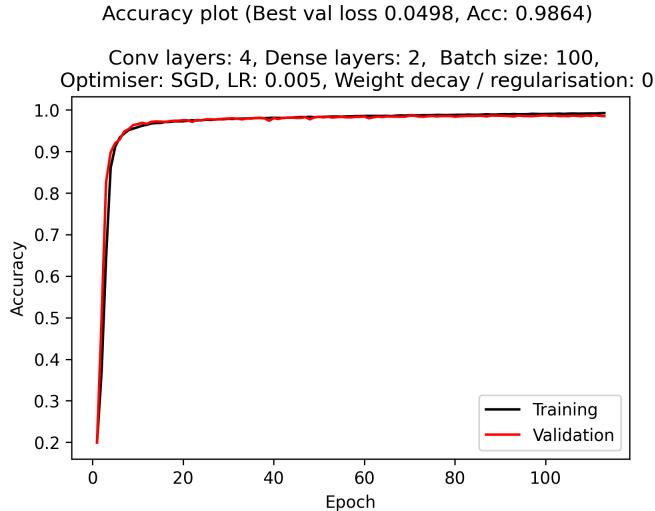


Figure 6: Training and validation accuracy by epoch for the CNN multiclass classification model

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.

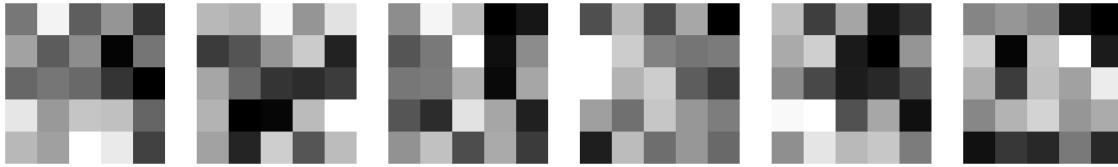


Figure 7: Filters for convolutional layer 1

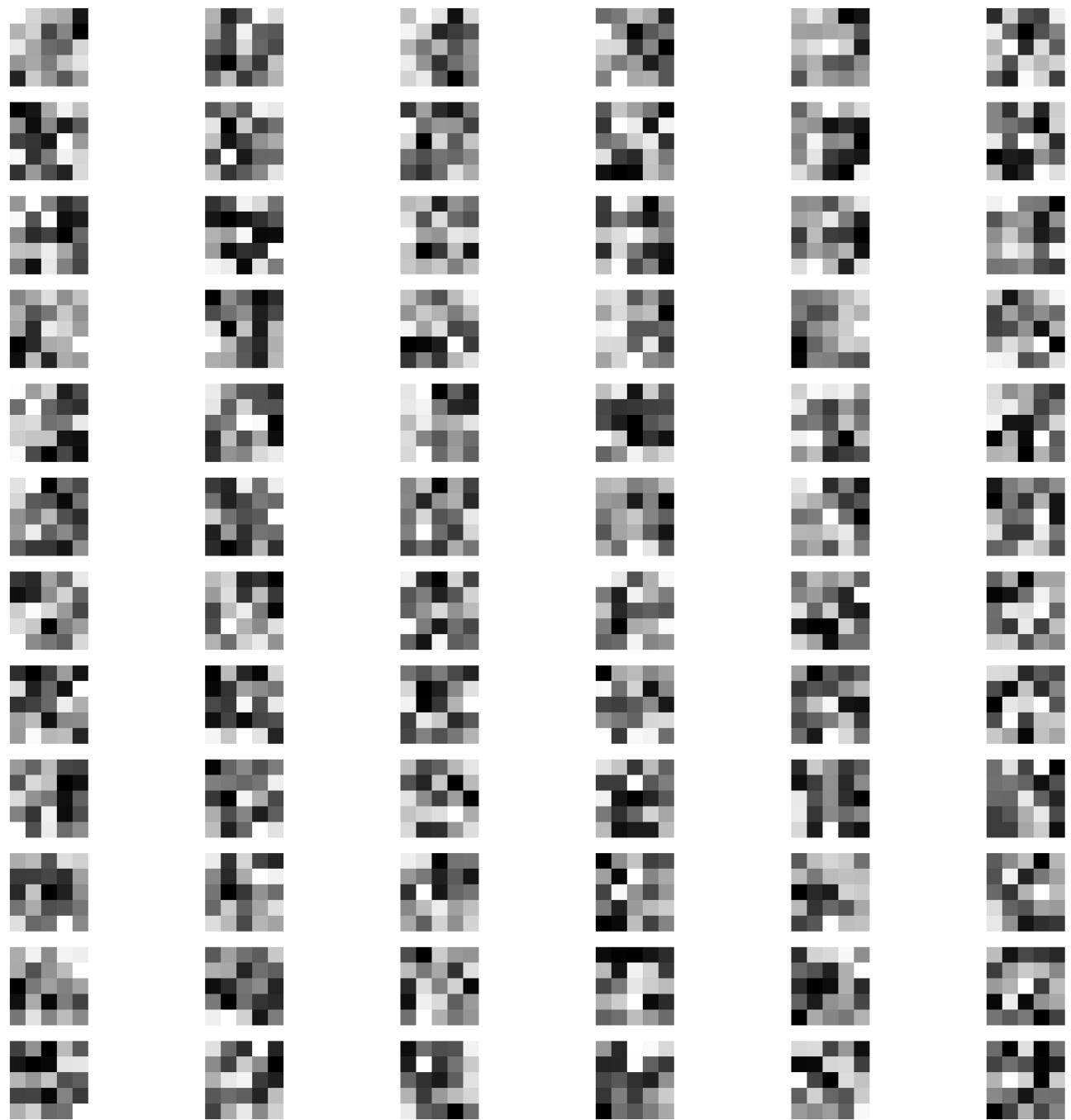


Figure 8: Filters for convolutional layer 2 - each column corresponds to one channel (12 filters for each channel)

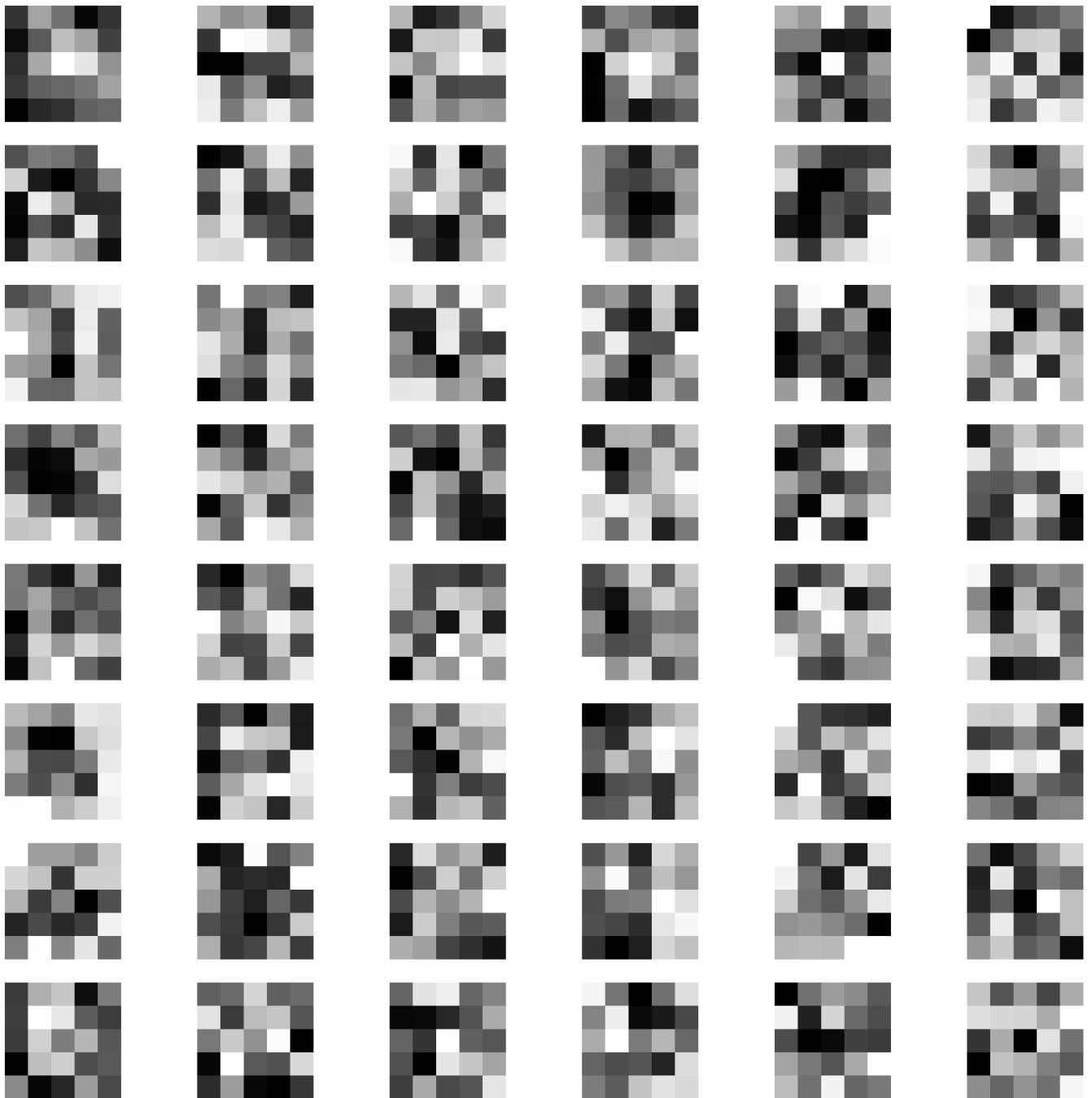


Figure 9: Filters for convolutional layer 3 (3 filters per channel displayed)

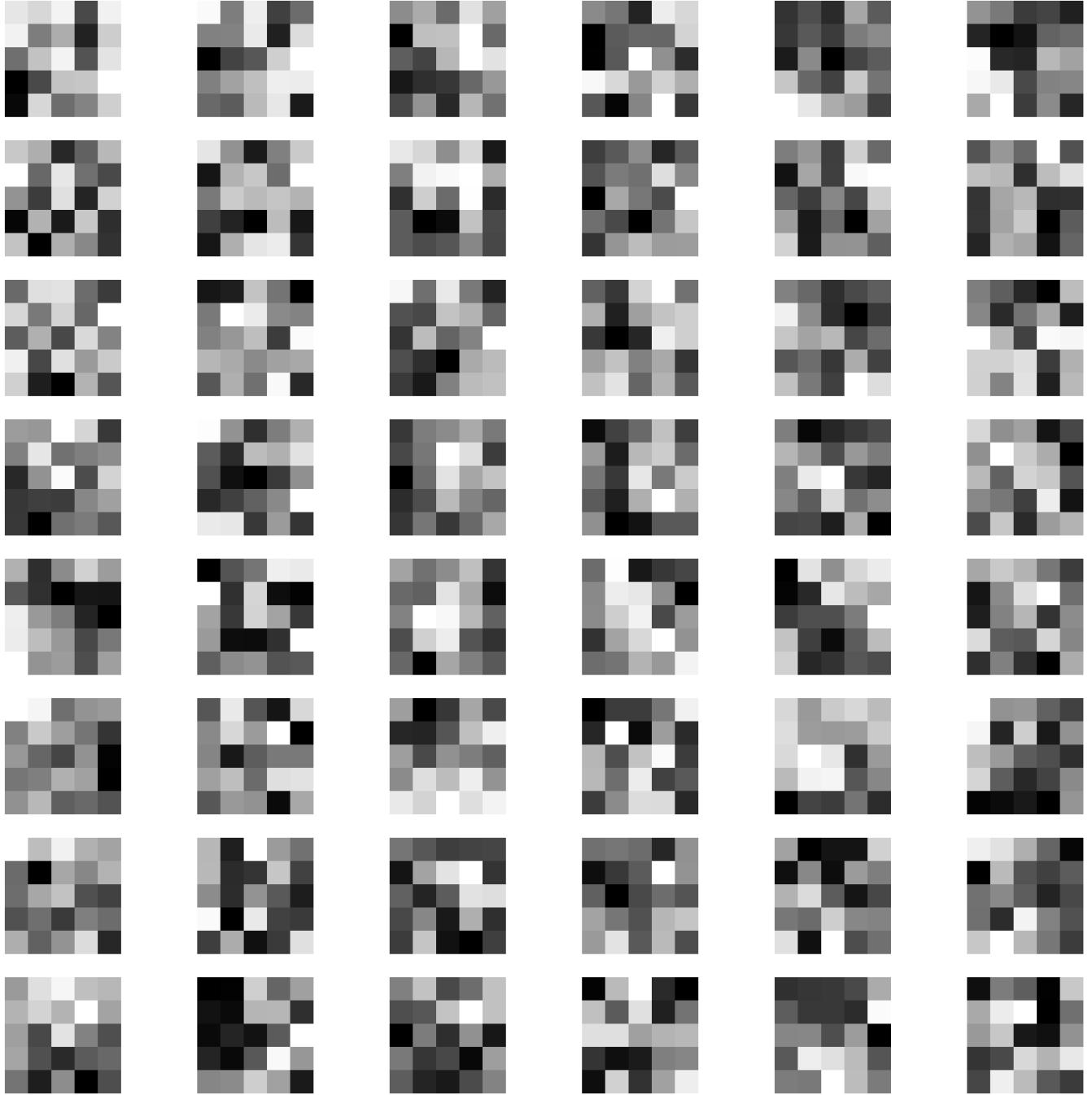


Figure 10: Filters for convolutional layer 4 (2 filters per channel displayed)

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.

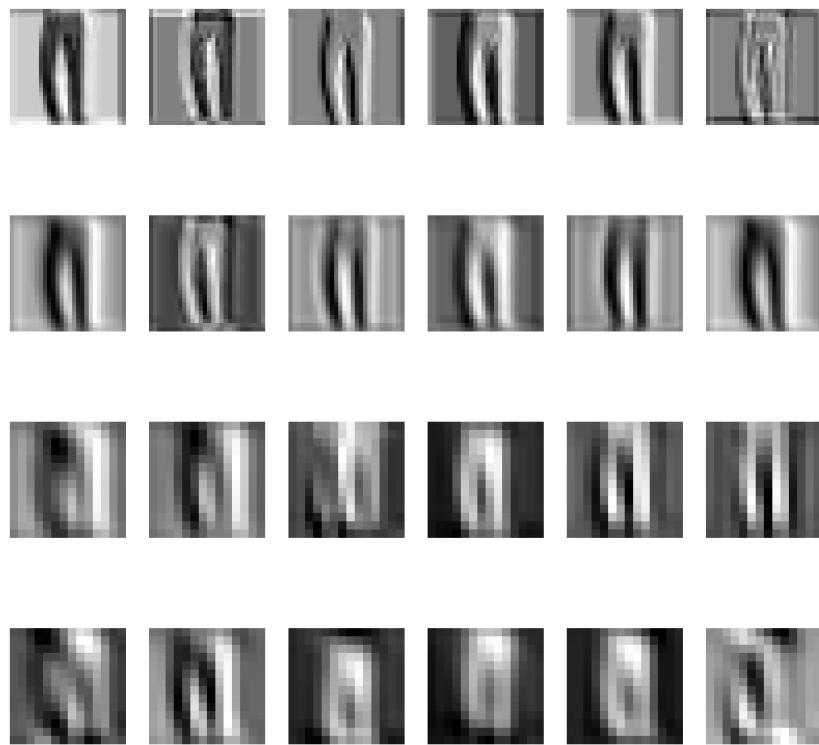


Figure 11: Subset of the feature maps for a trouser image (each row represents one convolutional layer)

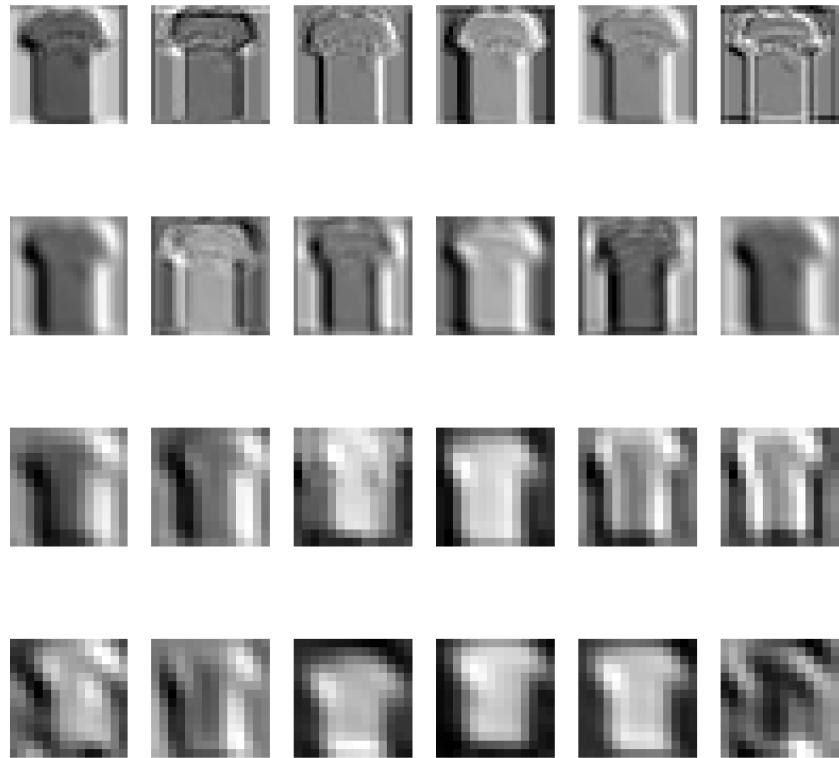


Figure 12: Subset of the feature maps for a T-shirt/top image (each row represents one convolutional layer)

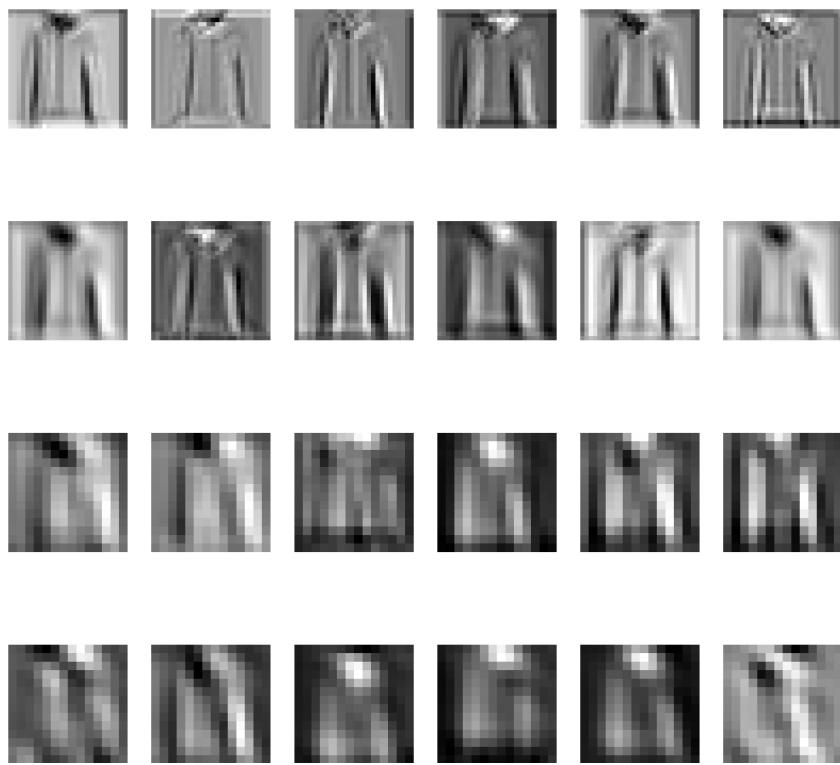


Figure 13: Subset of the feature maps for a coat image (each row represents one convolutional layer)

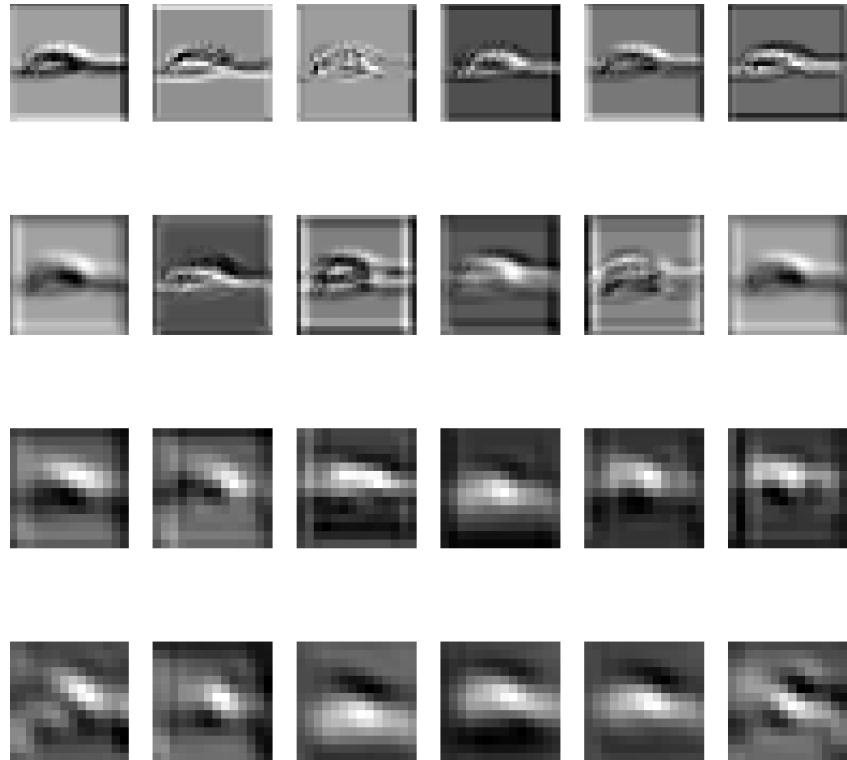


Figure 14: Subset of the feature maps for a sandal image (each row represents one convolutional layer)

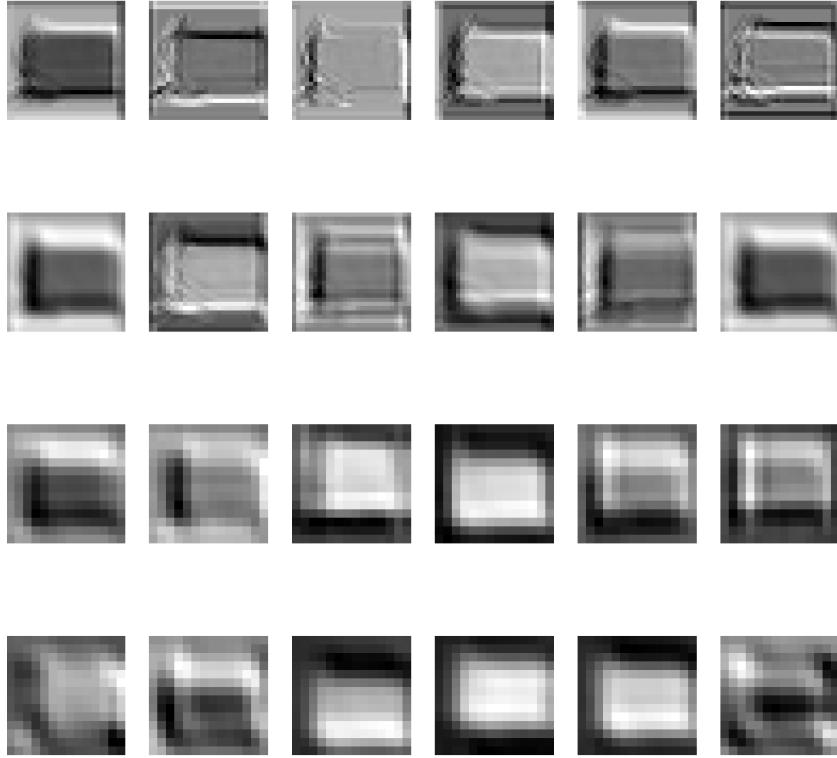


Figure 15: Subset of the feature maps for a bag image (each row represents one convolutional layer)

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:

$$\phi : \mathbb{X} \rightarrow \mathbb{F} \quad (1)$$

$$\psi : \mathbb{F} \rightarrow \mathbb{X} \quad (2)$$

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.

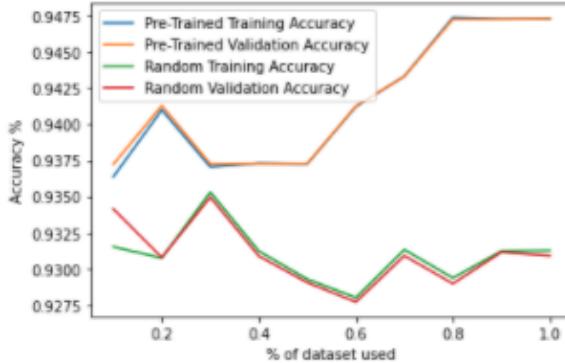


Figure 16: MLP accuracy vs proportion of dataset trained.

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()
```

Exemplar 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

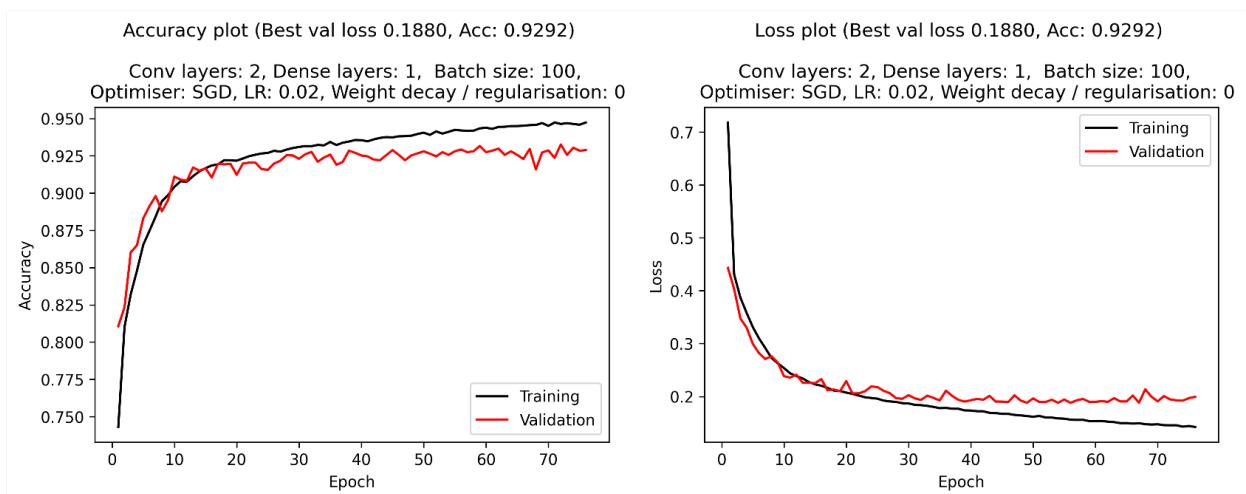


Figure 17: Accuracy and loss plots for initial model trained on FMNIST2

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

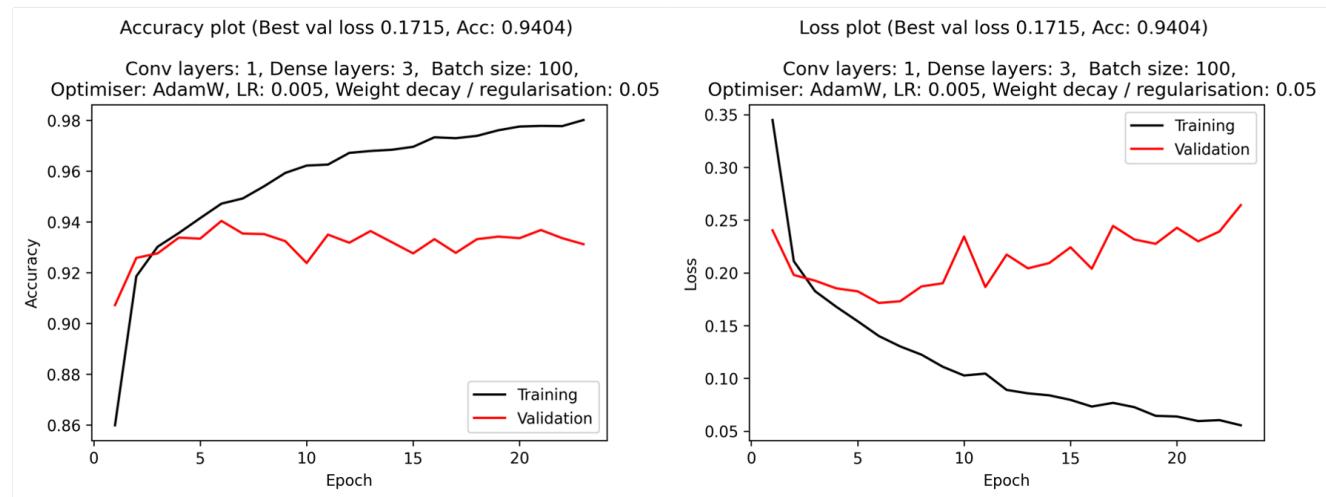


Figure 18: Accuracy and loss plots for best model trained on FMNIST2

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

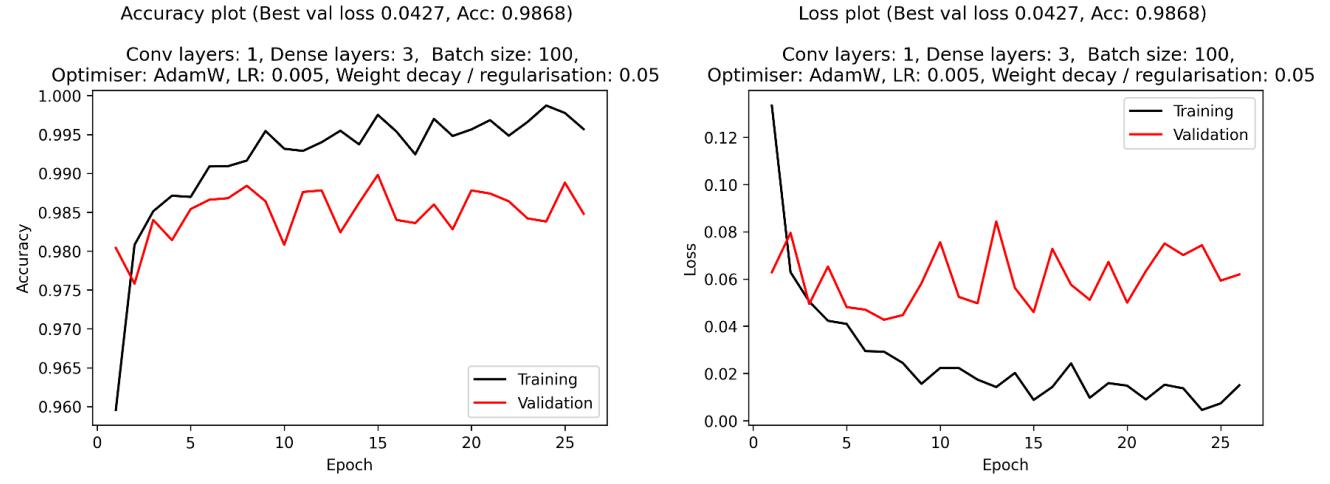


Figure 19: Accuracy and loss plots for FMNIST1 dataset

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.

Comparison of validation loss between a randomly initialised model and transferred-weights model when using different fractions of the training set

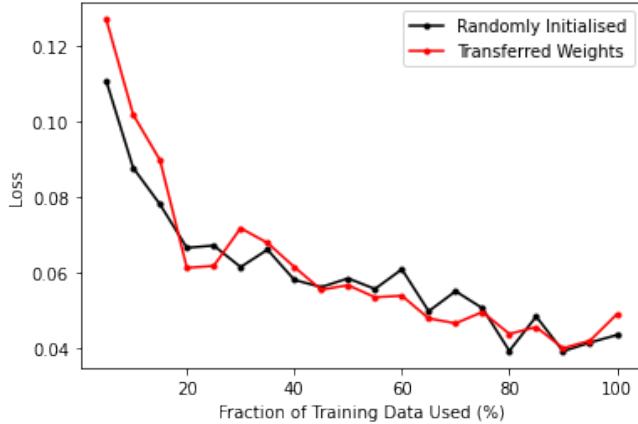


Figure 20: Best validation loss as training set size increases for both random initialisation (RI) and transferred weights (TL).

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

```
1 # LOAD FASHION MNIST DATASET (fm)
2
3 transform = tv.transforms.Compose([tv.transforms.ToTensor(), tv.transforms.Normalize((0.5,), 
4                                 (0.5,))])
5
6 fm1_train_val = tv.datasets.FashionMNIST(root = "./data", train = True, download = True,
7                                         transform = transform)
8 fm1_test = tv.datasets.FashionMNIST(root = "./data", train = False, download = True, transform
9                                         = transform)
10 fm2_train_val = tv.datasets.FashionMNIST(root = "./data", train = True, download = True,
11                                         transform = transform)
12 fm2_test = tv.datasets.FashionMNIST(root = "./data", train = False, download = True, transform
13                                         = transform)
14
15 # SPLIT TRAINING AND VALIDATION DATASETS INTO fm1 AND fm2
16 # (non-overlapping classes in each)
17
18 # specify classes to include in fm1 and fm2
19 fm1_labels = [0, 1, 4, 5, 8]
20 fm2_labels = [2, 3, 6, 7, 9]
21
22 # update fm1 and fm2 train_val datasets to just include the relevant classes for train_val
23 # datasets
24
25 # extract indices (boolean array) of images corresponding to each of the classes for fm1, and
26 # fm2
27 fm1_train_val_indices = (fm1_train_val.targets == 0) | (fm1_train_val.targets == 1) |
28                                         (fm1_train_val.targets == 4) | (fm1_train_val.targets == 5) | (fm1_train_val.targets == 8)
29 fm2_train_val_indices = (fm2_train_val.targets == 2) | (fm2_train_val.targets == 3) |
30                                         (fm2_train_val.targets == 6) | (fm2_train_val.targets == 7) | (fm2_train_val.targets == 9)
31
32 ##fm1
33 fm1_train_val.data = fm1_train_val.data[fm1_train_val_indices]
34 fm1_train_val.targets = fm1_train_val.targets[fm1_train_val_indices]
35 ### re-index labels, otherwise dataloader does not work
36 fm1_train_val.targets[fm1_train_val.targets == 4] = 2
37 fm1_train_val.targets[fm1_train_val.targets == 5] = 3
38 fm1_train_val.targets[fm1_train_val.targets == 8] = 4
39
40 ##fm2
41 fm2_train_val.data = fm2_train_val.data[fm2_train_val_indices]
42 fm2_train_val.targets = fm2_train_val.targets[fm2_train_val_indices]
43 ### re-index labels, otherwise dataloader does not work
44 fm2_train_val.targets[fm2_train_val.targets == 2] = 0
45 fm2_train_val.targets[fm2_train_val.targets == 3] = 1
46 fm2_train_val.targets[fm2_train_val.targets == 6] = 2
47 fm2_train_val.targets[fm2_train_val.targets == 7] = 3
48 fm2_train_val.targets[fm2_train_val.targets == 9] = 4
49
50 # REPEAT THE ABOVE FOR TEST DATASETS
```

```

46 fm1_test_indices = (fm1_test.targets == 0) | (fm1_test.targets == 1) | (fm1_test.targets == 4) |
47   ↵ (fm1_test.targets == 5) | (fm1_test.targets == 8)
48 fm2_test_indices = (fm2_test.targets == 2) | (fm2_test.targets == 3) | (fm2_test.targets == 6) |
49   ↵ (fm2_test.targets == 7) | (fm2_test.targets == 9)
50
51 ##fm1
52 fm1_test.data = fm1_test.data[fm1_test_indices]
53 fm1_test.targets = fm1_test.targets[fm1_test_indices]
54 ##### re-index labels, otherwise dataloader does not work
55 fm1_test.targets[fm1_test.targets == 4] = 2
56 fm1_test.targets[fm1_test.targets == 5] = 3
57 fm1_test.targets[fm1_test.targets == 8] = 4
58
59 ##fm2
60 fm2_test.data = fm2_test.data[fm2_test_indices]
61 fm2_test.targets = fm2_test.targets[fm2_test_indices]
62 ##### re-index labels, otherwise dataloader does not work
63 fm2_test.targets[fm2_test.targets == 2] = 0
64 fm2_test.targets[fm2_test.targets == 3] = 1
65 fm2_test.targets[fm2_test.targets == 6] = 2
66 fm2_test.targets[fm2_test.targets == 7] = 3
67 fm2_test.targets[fm2_test.targets == 9] = 4
68
69 # CREATE DICTIONARIES TO MAP OLD TO NEW INDICES
70
71 fm1_label_newindex_to_oldindex = {0: 0, 1: 1, 2: 4, 3: 5, 4: 8}
72 fm2_label_newindex_to_oldindex = {0: 2, 1: 3, 2: 6, 3: 7, 4: 9}
73
74 # CREATE DICTIONARIES TO MAP INDICES TO ITEM NAMES
75
76 fm1_label_index_to_text_dict = {0: "T-shirt/top", 1: "Trouser", 2: "Coat", 3: "Sandal", 4:
77   ↵ "Bag"}
78 fm2_label_index_to_text_dict = {0: "Pullover", 1: "Dress", 2: "Shirt", 3: "Sneaker", 4: "Ankle
79   ↵ boot"}
80
81 # FINAL TRAINING-VALIDATION SPLITTING
82
83 # split the train_val datasets into seperate train and validation datasets
84 fm1_train, fm1_val = torch.utils.data.random_split(fm1_train_val, [25000, 5000])
85 fm2_train, fm2_val = torch.utils.data.random_split(fm2_train_val, [25000, 5000])
86
87 # DATALOADER SETUP
88
89 # create dataloaders for each of the datasets
90
91 train_batch_size = 100
92 dev_batch_size = 1
93 test_batch_size = 1
94
95 fm1_trainloader = torch.utils.data.DataLoader(fm1_train, batch_size = train_batch_size,
96   ↵ shuffle = True)

```

```

97 fm1_devloader = torch.utils.data.DataLoader(fm1_val, batch_size = dev_batch_size, shuffle =
98   ↵ False)
99 fm1_testloader = torch.utils.data.DataLoader(fm1_test, batch_size = test_batch_size, shuffle =
100  ↵ False)
101
102 fm2_trainloader = torch.utils.data.DataLoader(fm2_train,  batch_size = train_batch_size,
103   ↵ shuffle = True)
104 fm2_devloader = torch.utils.data.DataLoader(fm2_val, batch_size = dev_batch_size, shuffle =
105   ↵ False)
106 fm2_testloader = torch.utils.data.DataLoader(fm2_test, batch_size = test_batch_size, shuffle =
107   ↵ False)
108
109
110 # DISPLAY EXAMPLE IMAGES AND LABELS
111
112 NUMBER_TO_DISPLAY = 2
113
114 for i, (xb, yb) in enumerate(fm1_devloader):
115   x = xb.view(28,28)
116   y = yb.item()
117   print(f"Class: {fm1_label_index_to_text_dict[y]}")
118
119   plt.imshow(x, cmap="gray")
120   plt.axis('off')
121   plt.show()
122
123   if i == NUMBER_TO_DISPLAY-1:
124     break

```

Appendix B Q1 code

```
1 # DATA PREP AS PER APPENDIX A
2
3
4 # HYPERPARAMETERS
5
6 # hyperparameter grid for Q1
7 hyperparam_grid = {'num_dense_layers': [2],
8                     'num_conv_layers': [2],
9                     'learning_rate': [0.005],
10                    'weight_decay': [0], # Regularisation
11                    'optimiser': ["SGD"],
12                    'batch_size': [train_batch_size]}
13
14 # range of epochs
15 min_epochs = 20
16 max_epochs = 450
17
18 # number of classes
19 num_classes = 5
20
21 # initiate dataframe to store results of grid search
22 results_df = pd.DataFrame()
23
24
25 # MODEL, TRAINING, EVALUATION
26
27 for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):
28
29     # pairings of hyperparameters to skip
30
31     if (hyperparams['optimiser'] == "SGD" and hyperparams['learning_rate'] < 0.005) or
32         (hyperparams['optimiser'] == "AdamW" and hyperparams['learning_rate'] > 0.005):
33         continue
34
35     # specify model architecture
36
37     class CNN(nn.Module):
38
39         def __init__(self, num_classes, num_dense_layers, num_conv_layers):
40             super(CNN, self).__init__()
41             self.num_dense_layers = num_dense_layers
42             self.num_conv_layers = num_conv_layers
43
44             # single conv layer
45             if num_conv_layers == 1:
46                 self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
47                 self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
48
49             # dense layers
50             if num_dense_layers == 1:
51                 self.fc1 = nn.Linear(6*(12**2), num_classes)
52             elif num_dense_layers == 2:
53                 self.fc1 = nn.Linear(6*(12**2), 80)
54                 self.fc2 = nn.Linear(80, num_classes)
```

```

54     elif num_dense_layers == 3:
55         self.fc1 = nn.Linear(6*(12**2), 120)
56         self.fc2 = nn.Linear(120, 84)
57         self.fc3 = nn.Linear(20, num_classes)
58
59 # two conv layers
60 elif num_conv_layers == 2:
61     self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
62     self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
63     self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
64     self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
65
66 # dense layers
67 if num_dense_layers == 1:
68     self.fc1 = nn.Linear(12*(4**2), num_classes)
69 elif num_dense_layers == 2:
70     self.fc1 = nn.Linear(12*(4**2), 80)
71     self.fc2 = nn.Linear(80, num_classes)
72 elif num_dense_layers == 3:
73     self.fc1 = nn.Linear(12*(4**2), 120)
74     self.fc2 = nn.Linear(120, 84)
75     self.fc3 = nn.Linear(20, num_classes)
76
77 def forward(self, x):
78
79     #conv2d and maxpools
80     if self.num_conv_layers == 1:
81         x = self.pool1(F.relu(self.conv1(x)))
82         x = x.view(-1, 6*(12**2))
83     elif self.num_conv_layers == 2:
84         x = self.pool1(F.relu(self.conv1(x)))
85         x = self.pool2(F.relu(self.conv2(x)))
86         x = x.view(-1, 12*(4**2))
87
88     #fully-connected layers
89     if self.num_dense_layers == 1:
90         x = self.fc1(x)
91     if self.num_dense_layers == 2:
92         x = F.relu(self.fc1(x))
93         x = self.fc2(x)
94     if self.num_dense_layers == 3:
95         x = F.relu(self.fc1(x))
96         x = F.relu(self.fc2(x))
97         x = self.fc3(x)
98
99     return x
100
101 # Initialise model and send to GPU
102
103 model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])
104
105 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
106 print(device)
107 model.to(device)
108
109

```

```

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'],
116                                weight_decay=hyperparams['weight_decay'])
117 elif hyperparams['optimiser'] == "AdamW":
118     optimiser = torch.optim.AdamW(model.parameters(), lr=hyperparams['learning_rate'],
119                                weight_decay=hyperparams['weight_decay'])

120 # training loop
121
122 train_loss_list = []
123 train_accuracy_list = []
124 dev_loss_list = []
125 dev_accuracy_list = []

126 best_dev_loss = 9999 # initiate high value so that training doesn't immediately stop
127
128 converged = False
129 epochs_counter = 0
130 epochs_to_plot = []

132 while (converged == False) and (epochs_counter <= max_epochs):
133
134     train_running_loss = 0
135     train_running_corrects = 0
136     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

```

```

164     train_loss_list.append(train_loss)
165
166     model.eval()
167
168     with torch.no_grad():
169
170         dev_running_loss = 0
171         dev_running_corrects = 0
172         dev_num_of_batches = 0
173
174         for data, label in fm1_devloader:
175
176             data = data.to(device)
177             label = label.to(device)
178
179             # forward
180             dev_preds = model(data)
181             loss = criterion(dev_preds, label.long())
182
183             # metrics
184             dev_preds_indices = torch.argmax(dev_preds.data, 1)
185             dev_running_corrects += (dev_preds_indices == label).sum().item()
186             dev_running_loss += loss.item()
187             dev_num_of_batches += 1
188
189             dev_accuracy = dev_running_corrects/(dev_num_of_batches*dev_batch_size)
190             dev_accuracy_list.append(dev_accuracy)
191             dev_loss = dev_running_loss/dev_num_of_batches
192             dev_loss_list.append(dev_loss)
193
194             epochs_counter += 1
195             epochs_to_plot.append(epochs_counter)
196
197             print(f"epoch: {epochs_counter}, train_loss: {train_loss:.4f}, train_acc:
198                   {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")
199
200             ## Store the current results if it is the highest validation accuracy so far
201             if dev_loss < best_dev_loss:
202                 best_dev_accuracy = dev_accuracy
203                 best_dev_loss = dev_loss
204                 best_train_accuracy = train_accuracy
205                 best_train_loss = train_loss
206                 best_epoch = epochs_counter
207                 best_model = model
208
209             ## Early stopping criteria
210             if epochs_counter >= min_epochs:
211                 if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
212                     converged = True
213                     print("Model Converged")
214
215
216             # Evaluate loss and accuracy on test set
217
218             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,
271     ↪ f"{output_path}MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}")
272
273     # # Loss plot
274     plt.plot(epochs_to_plot, train_loss_list, color='k', linestyle='--')

```

```

274 plt.plot(epochs_to_plot, dev_loss_list, color='r', linestyle='--')
275 plt.legend(['Training', 'Validation'], loc='upper right')
276 plt.ylabel('Loss', color='k')
277 plt.xlabel('Epoch', color='k')
278 plt.title(f'''Loss plot (Best val loss {best_dev_loss:.4f}, Acc: {best_dev_accuracy:.4f}) \n
279 Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
280     {hyperparams['num_dense_layers']}, Batch size: {hyperparams['batch_size']},
281 Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
282     regularisation: {hyperparams['weight_decay']}'''', color='k')
283
284
285 # # Accuracy plot
286 plt.plot(epochs_to_plot, train_accuracy_list, color='k', linestyle='--')
287 plt.plot(epochs_to_plot, dev_accuracy_list, color='r', linestyle='--')
288 plt.legend(['Training', 'Validation'], loc='lower right')
289 plt.ylabel('Accuracy', color='k')
290 plt.xlabel('Epoch', color='k')
291 plt.title(f'''Accuracy plot (Best val loss {best_dev_loss:.4f}, Acc:
292     {best_dev_accuracy:.4f}) \n
293 Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
294     {hyperparams['num_dense_layers']}, Batch size: {hyperparams['batch_size']},
295 Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
296     regularisation: {hyperparams['weight_decay']}'''', color='k')
297
298
299 # CONFUSION MATRIX
300
301 # Generate confusion matrix from best models predictions on the test dataset
302
303 # print(f"Class labels: {fm1_label_index_to_text_dict}")
304
305 cm = ConfusionMatrix(actual_vector=labels_list, predict_vector=predictions_list)
306 cm.relabel(mapping=fm1_label_index_to_text_dict)
307 items = ["T-shirt/top", "Trouser", "Coat", "Sandal", "Bag"]
308 cm.plot(cmap=plt.cm.Blues, number_label=True, class_name=items)
309 plt.xticks(rotation=50)
310 plt.savefig(f'{output_path}MODEL{hyperparams["model_index"]}_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}.png'
311             , dpi=300, bbox_inches = "tight")

```

Appendix C Q2 code

```
1 # DATA PREP AS PER APPENDIX A
2
3 # hyperparameters
4 hyperparams = {'num_dense_layers': 2,
5                 'num_conv_layers': 1,
6                 'learning_rate': 0.0005,
7                 'weight_decay': 0, # Regularisation
8                 'optimiser': "Adam",
9                 'batch_size': train_batch_size}
10
11 # range of epochs
12 min_epochs = 20
13 max_epochs = 300
14
15 # number of classes
16 num_classes = 5
17
18 # initiate dataframe to store results of grid search
19 results_df = pd.DataFrame()
20
21 # MODEL, TRAINING, EVALUATION
22
23 # PyTorch model class, which takes the specified hyperparameters as inputs
24 class CNN(nn.Module):
25
26     def __init__(self, num_classes, num_dense_layers, num_conv_layers):
27         super(CNN, self).__init__()
28         self.num_dense_layers = num_dense_layers
29         self.num_conv_layers = num_conv_layers
30
31         # single conv layer
32         if num_conv_layers == 1:
33             self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
34             self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
35
36         # dense layers
37         if num_dense_layers == 1:
38             self.fc1 = nn.Linear(6*(12**2), num_classes)
39         elif num_dense_layers == 2:
40             self.fc1 = nn.Linear(6*(12**2), 80)
41             self.fc2 = nn.Linear(80, num_classes)
42         elif num_dense_layers == 3:
43             self.fc1 = nn.Linear(6*(12**2), 120)
44             self.fc2 = nn.Linear(120, 84)
45             self.fc3 = nn.Linear(84, num_classes)
46
47         # two conv layers
48         elif num_conv_layers == 2:
49             self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
50             self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
51             self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
52             self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
53
54         # dense layers
```

```

55     if num_dense_layers == 1:
56         self.fc1 = nn.Linear(12*(4**2), num_classes)
57     elif num_dense_layers == 2:
58         self.fc1 = nn.Linear(12*(4**2), 80)
59         self.fc2 = nn.Linear(80, num_classes)
60     elif num_dense_layers == 3:
61         self.fc1 = nn.Linear(12*(4**2), 120)
62         self.fc2 = nn.Linear(120, 84)
63         self.fc3 = nn.Linear(84, num_classes)
64
65     def forward(self, x):
66
67         # conv2d and maxpools
68         if self.num_conv_layers == 1:
69             x = self.pool1(F.relu(self.conv1(x)))
70             x = x.view(-1, 6*(12**2))
71         elif self.num_conv_layers == 2:
72             x = self.pool1(F.relu(self.conv1(x)))
73             x = self.pool2(F.relu(self.conv2(x)))
74             x = x.view(-1, 12*(4**2))
75
76         # fully-connected layers
77         if self.num_dense_layers == 1:
78             x = self.fc1(x)
79         if self.num_dense_layers == 2:
80             x = F.relu(self.fc1(x))
81             x = self.fc2(x)
82         if self.num_dense_layers == 3:
83             x = F.relu(self.fc1(x))
84             x = F.relu(self.fc2(x))
85             x = self.fc3(x)
86
87         return x
88
89     # Initialise model and send to GPU
90
91     model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])
92
93     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
94     print(device)
95     model.to(device)
96
97
98     # Loss & optimiser
99     criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
100
101    if hyperparams['optimiser'] == "SGD":
102        optimiser = torch.optim.SGD(model.parameters(), lr=hyperparams['learning_rate'],
103                                  weight_decay=hyperparams['weight_decay'])
103    elif hyperparams['optimiser'] == "Adam":
104        optimiser = torch.optim.Adam(model.parameters(), lr=hyperparams['learning_rate'])
105    elif hyperparams['optimiser'] == "AdamW":
106        optimiser = torch.optim.AdamW(model.parameters(), lr=hyperparams['learning_rate'],
107                                     weight_decay=hyperparams['weight_decay'])
107
108
```

```

109 # training loop
110
111 train_loss_list = []
112 train_accuracy_list = []
113 dev_loss_list = []
114 dev_accuracy_list = []
115
116 best_dev_loss = 9999 # initiate high value so that training doesn't immediately stop
117
118 converged = False
119 epochs_counter = 0
120 epochs_to_plot = []
121
122 while (converged == False) and (epochs_counter <= max_epochs):
123
124     train_running_loss = 0
125     train_running_corrects = 0
126     train_num_of_batches = 0
127
128     model.train()
129
130     for data, label in fm1_trainloader:
131
132         data = data.to(device)
133         label = label.to(device)
134
135         # zero the gradients
136         optimiser.zero_grad()
137
138         # forward + backward + optimize
139         train_preds = model(data)
140         loss = criterion(train_preds, label.long())
141         loss.backward()
142         optimiser.step()
143
144         # metrics
145         train_preds_indices = torch.argmax(train_preds.data, 1)
146         train_running_corrects += (train_preds_indices == label).sum().item()
147         train_running_loss += loss.item()
148         train_num_of_batches += 1
149
150         train_accuracy = train_running_corrects/(train_num_of_batches*train_batch_size)
151         train_accuracy_list.append(train_accuracy)
152         train_loss = train_running_loss/train_num_of_batches
153         train_loss_list.append(train_loss)
154
155
156     model.eval()
157
158     with torch.no_grad():
159
160         dev_running_loss = 0
161         dev_running_corrects = 0
162         dev_num_of_batches = 0
163
164         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:
188           {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")
189
190     # Store the current results if it has the smallest validation loss so far
191     if dev_loss < best_dev_loss:
192         best_dev_accuracy = dev_accuracy
193         best_dev_loss = dev_loss
194         best_train_accuracy = train_accuracy
195         best_train_loss = train_loss
196         best_epoch = epochs_counter
197         best_model = model
198
199     # Early stopping criteria
200     if epochs_counter >= min_epochs:
201         if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
202             converged = True
203             print("Model Converged")
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,
258                f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc.pth")
259
260     # Loss plot
261     plt.plot(epochs_to_plot, train_loss_list, color='k', linestyle='--')
262     plt.plot(epochs_to_plot, dev_loss_list, color='r', linestyle='--')
263     plt.legend(['Training', 'Validation'], loc='upper right')
264     plt.ylabel('Loss', color='k')
265     plt.xlabel('Epoch', color='k')
266     plt.title(f'''Loss plot (Best val loss {best_dev_loss:.4f}, Acc: {best_dev_accuracy:.4f}) \n
267     Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
268     ↳ {hyperparams['num_dense_layers']}, Batch size: {hyperparams['batch_size']},
269     Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
270     ↳ regularisation: {hyperparams['weight_decay']}'''', color='k')
271
272     plt.savefig(f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc_LOSSPLOT.png",
273                dpi=300, bbox_inches = "tight")
274     plt.show()

```

```
272 # Accuracy plot
273 plt.plot(epochs_to_plot, train_accuracy_list, color='k', linestyle='--')
274 plt.plot(epochs_to_plot, dev_accuracy_list, color='r', linestyle='--')
275 plt.legend(['Training', 'Validation'], loc='lower right')
276 plt.ylabel('Accuracy', color='k')
277 plt.xlabel('Epoch', color='k')
278 plt.title(f'''Accuracy plot (Best val loss {best_dev_loss:.4f}, Acc: {best_dev_accuracy:.4f})
279     \n
280 Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
281     {hyperparams['num_dense_layers']}, Batch size: {hyperparams['batch_size']},
282 Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
283     regularisation: {hyperparams['weight_decay']}''', color='k')
284
285 plt.savefig(f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc_ACCPLOT.png",
286             dpi=300, bbox_inches = "tight")
287 plt.show()
```

Appendix D Q3 code

```
1 # DATA PREP AS PER APPENDIX A
2
3
4 # HYPERPARAMETERS
5
6 # hyperparameter grid for q3
7 hyperparam_grid = {'num_dense_layers': [2],
8                     'num_conv_layers': [4],
9                     'learning_rate': [0.005],
10                    'weight_decay': [0], # Regularisation
11                    'optimiser': ["SGD"],
12                    'batch_size': [train_batch_size]}
13
14 # range of epochs
15 min_epochs = 20
16 max_epochs = 300
17
18 # number of classes
19 num_classes = 5
20
21 # MODEL, TRAINING, EVALUATION
22
23 class CNN(nn.Module):
24
25     def __init__(self, num_classes, num_dense_layers, num_conv_layers):
26         super(CNN, self).__init__()
27         self.num_dense_layers = num_dense_layers
28         self.num_conv_layers = num_conv_layers
29
30         # single conv layer
31         if num_conv_layers == 1:
32             self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
33             self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
34
35         # dense layers
36         if num_dense_layers == 1:
37             self.fc1 = nn.Linear(6*(12**2), num_classes)
38         elif num_dense_layers == 2:
39             self.fc1 = nn.Linear(6*(12**2), 80)
40             self.fc2 = nn.Linear(80, num_classes)
41         elif num_dense_layers == 3:
42             self.fc1 = nn.Linear(6*(12**2), 120)
43             self.fc2 = nn.Linear(120, 84)
44             self.fc3 = nn.Linear(84, num_classes)
45
46         # two conv layers
47         elif num_conv_layers == 2:
48             self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
49             self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
50             self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
51             self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
52
53         # dense layers
54         if num_dense_layers == 1:
```

```

55         self.fc1 = nn.Linear(12*(4**2), num_classes)
56     elif num_dense_layers == 2:
57         self.fc1 = nn.Linear(12*(4**2), 80)
58         self.fc2 = nn.Linear(80, num_classes)
59     elif num_dense_layers == 3:
60         self.fc1 = nn.Linear(12*(4**2), 120)
61         self.fc2 = nn.Linear(120, 84)
62         self.fc3 = nn.Linear(20, num_classes)
63
64     # four conv layers
65     elif num_conv_layers == 4:
66         self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2)
67         self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5, padding=2)
68         self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
69         self.conv3 = nn.Conv2d(in_channels=12, out_channels=16, kernel_size=5,padding=2)
70         self.conv4 = nn.Conv2d(in_channels=16, out_channels=24, kernel_size=5,padding=2)
71         self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
72
73     # dense layers
74     if num_dense_layers == 1:
75         self.fc1 = nn.Linear(24*(7**2), num_classes)
76     elif num_dense_layers == 2:
77         self.fc1 = nn.Linear(24*(7**2), 80)
78         self.fc2 = nn.Linear(80, num_classes)
79     elif num_dense_layers == 3:
80         self.fc1 = nn.Linear(24*(7**2), 120)
81         self.fc2 = nn.Linear(120, 84)
82         self.fc3 = nn.Linear(20, num_classes)
83
84     def forward(self, x):
85
86         # conv2d and maxpools
87         if self.num_conv_layers == 1:
88             x = self.pool1(F.relu(self.conv1(x)))
89             x = x.view(-1, 6*(12**2))
90         elif self.num_conv_layers == 2:
91             x = self.pool1(F.relu(self.conv1(x)))
92             x = self.pool2(F.relu(self.conv2(x)))
93             x = x.view(-1, 12*(4**2))
94
95         elif self.num_conv_layers == 4:
96             x = F.relu(self.conv1(x))
97             x = F.relu(self.conv2(x))
98             x = self.pool1(x)
99             x = F.relu(self.conv3(x))
100            x = F.relu(self.conv4(x))
101            x = self.pool2(x)
102            x = x.view(-1, 24*(7**2))
103
104         # fully-connected layers
105         if self.num_dense_layers == 1:
106             x = self.fc1(x)
107         if self.num_dense_layers == 2:
108             x = F.relu(self.fc1(x))
109             x = self.fc2(x)
110         if self.num_dense_layers == 3:

```

```

111         x = F.relu(self.fc1(x))
112         x = F.relu(self.fc2(x))
113         x = self.fc3(x)
114
115     return x
116
117 for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):
118
119     # pairings of hyperparameters to skip
120
121     if (hyperparams['optimiser'] == "SGD" and hyperparams['learning_rate'] < 0.005) or
122         (hyperparams['optimiser'] == "AdamW" and hyperparams['learning_rate'] > 0.005):
123         continue
124
125     # Initialise model and send to GPU
126
127     model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])
128
129     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
130     print(device)
131     model.to(device)
132
133     # Loss & optimiser
134     criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
135
136     if hyperparams['optimiser'] == "SGD":
137         optimiser = torch.optim.SGD(model.parameters(), lr=hyperparams['learning_rate'],
138                                     weight_decay=hyperparams['weight_decay'])
138     elif hyperparams['optimiser'] == "AdamW":
139         optimiser = torch.optim.AdamW(model.parameters(), lr=hyperparams['learning_rate'],
140                                     weight_decay=hyperparams['weight_decay'])
140
141
142     # training loop
143
144     train_loss_list = []
145     train_accuracy_list = []
146     dev_loss_list = []
147     dev_accuracy_list = []
148
149     best_dev_loss = 9999 # initiate high value so that training doesn't immediately stop
150
151     converged = False
152     epochs_counter = 0
153     epochs_to_plot = []
154
155     while (converged == False) and (epochs_counter <= max_epochs):
156
157         train_running_loss = 0
158         train_running_corrects = 0
159         train_num_of_batches = 0
160
161         model.train()
162
163         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:
221         {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")
222
223     # Store the current results if it is the highest validation accuracy so far
224     if dev_loss < best_dev_loss:
225         best_dev_accuracy = dev_accuracy
226         best_dev_loss = dev_loss
227         best_train_accuracy = train_accuracy
228         best_train_loss = train_loss
229         best_epoch = epochs_counter
230         best_model = model
231
232     # Early stopping criteria
233     if epochs_counter >= min_epochs:
234         if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
235             converged = True
236             print("Model Converged")
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

```

```

275     hyperparams['associated_validation_accuracy'] = best_dev_accuracy
276     hyperparams['associated_training_loss'] = best_train_loss
277     hyperparams['associated_training_accuracy'] = best_train_accuracy
278     hyperparams['test_loss'] = test_loss
279     hyperparams['test_accuracy'] = test_accuracy
280
281     print(f"\n RUN_INDEX: {run_index + 1} \n")
282     print(hyperparams)
283
284     # save the model
285     torch.save(best_model,
286                f"{output_path}4_conv_layer_pad_MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_devloss_{best_"
287
288     # Loss plot
289     plt.plot(epochs_to_plot, train_loss_list, color='k', linestyle='--')
290     plt.plot(epochs_to_plot, dev_loss_list, color='r', linestyle='--')
291     plt.legend(['Training', 'Validation'], loc='upper right')
292     plt.ylabel('Loss', color='k')
293     plt.xlabel('Epoch', color='k')
294     plt.title(f'''Loss plot (Best val loss {best_dev_loss:.4f}, Acc: {best_dev_accuracy:.4f}) \n
295     Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
296     {hyperparams['num_dense_layers']}, Batch size: {hyperparams['batch_size']},
297     Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
298     regularisation: {hyperparams['weight_decay']}'', color='k')
299
300     plt.savefig(f"{output_path}4_conv_layer_pad_MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_de"
301     dpi=300, bbox_inches = "tight")
302     plt.show()
303
304     # Accuracy plot
305     plt.plot(epochs_to_plot, train_accuracy_list, color='k', linestyle='--')
306     plt.plot(epochs_to_plot, dev_accuracy_list, color='r', linestyle='--')
307     plt.legend(['Training', 'Validation'], loc='lower right')
308     plt.ylabel('Accuracy', color='k')
309     plt.xlabel('Epoch', color='k')
310     plt.title(f'''Accuracy plot (Best val loss {best_dev_loss:.4f}, Acc:
311     {best_dev_accuracy:.4f}) \n
312     Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
313     {hyperparams['num_dense_layers']}, Batch size: {hyperparams['batch_size']},
314     Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
315     regularisation: {hyperparams['weight_decay']}'', color='k')
316
317     plt.savefig(f"{output_path}4_conv_layer_pad_MODEL{hyperparams['model_index']}_{best_dev_loss:.4f}_de"
318     dpi=300, bbox_inches = "tight")
319     plt.show()
320
321     with torch.no_grad():
322
323         # retrieve best model
324         best_model =
325             torch.load(f"{output_path}4_conv_layer_pad_MODEL1_0.0498_devloss_0.9864_devacc.pth",
326             map_location=torch.device('cpu'))
327         best_model.eval()
328

```

```

319 # get list of layers of best model
320 layers = list(best_model.children())
321 num_layers = len(layers)
322
323 # VISUALISING FILTERS
324
325 # Save weights in convolutional layers to plot filters
326 filter_weights = []
327 for i in range(num_layers):
328     if type(layers[i]) == nn.Conv2d:
329         filter_weights.append(layers[i].weight)
330
331 # Visualise the filters (for the last convolutional layer in this case)
332 i = 0
333 filters_per_channel = 2
334 plt.figure(figsize=(20, 20))
335 for filter in filter_weights[3]:
336     for j in range(filters_per_channel):
337         i += 1
338         plt.subplot(8, 6, i)
339         plt.imshow(filter[j, :, :].detach().cpu(), cmap='gray')
340         plt.axis('off')
341 # plt.savefig(f"{output_path}filters layer 4", dpi=300, bbox_inches = "tight")
342 plt.show()
343
344 # VISUALISING FEATURE MAPS
345
346 # Get first batch from test loader (batch size = 1)
347 it = iter(fm1_testloader)
348 test_input_tensor = next(it)
349 test_input_label = test_input_tensor[1]
350
351 while test_input_label[0] != 4: # Get test input of a certain type (4 is a bag, for
352     # example)
353     test_input_tensor = next(it)
354     test_input_label = test_input_tensor[1]
355
356 test_input = test_input_tensor[0]
357
358 # Extract only the convolutional (and max-pool) layers ie. leaving out final 2 dense
359 # layers
360 conv_layers = layers[:-2]
361 num_conv_layers = len(conv_layers)
362
363 # Pass input into first layer
364 test_outputs = [conv_layers[0](test_input)]
365 for i in range(1, len(conv_layers)):
366     # Pass the output from the last layer to the next layer and save
367     test_outputs.append(conv_layers[i](test_outputs[-1]))
368
369 # Remove outputs of max-pool layers ie. only keep convolutional layers
370 del(test_outputs[2])
371 del(test_outputs[-1])
372
373 # Visualise and save the feature maps for each layer
374 plt.figure(figsize=(50, 50))

```

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

Appendix E Q4 code

```
1 class VanillaAutoEncoder(nn.Module):
2     def __init__(self, structure):
3         super().__init__()
4
5         encoder_layers = []
6         decoder_layers = []
7
8         # Encoder
9         for l in structure:
10             encoder_layers.append(nn.Linear(l[0], l[1]))
11             encoder_layers.append(nn.ReLU(True))
12
13         # Decoder
14         for index, l in reversed(list(enumerate(structure))):
15             decoder_layers.append(nn.Linear(l[1], l[0]))
16             # Last index, thus need Sigmoid
17             if (index == 0):
18                 decoder_layers.append(nn.Sigmoid())
19             else:
20                 decoder_layers.append(nn.ReLU(True))
21
22         self.encoder_struct = nn.Sequential(*encoder_layers)
23         self.decoder_struct = nn.Sequential(*decoder_layers)
24
25     def forward(self, x):
26         x = self.encoder_struct(x)
27         x = self.decoder_struct(x)
28         return x
29
30 class MLP(nn.Module):
31     def __init__(self, structure):
32         super().__init__()
33
34         layers = []
35         for index, l in enumerate(structure):
36             layers.append(nn.Linear(l[0], l[1]))
37             if (index == (len(structure) - 1)):
38                 layers.append(nn.Sigmoid())
39             else:
40                 layers.append(nn.ReLU(True))
41
42         self.struct = nn.Sequential(*layers)
43
44     def forward(self, x):
45         x = self.struct(x)
46         return x
47
48 encoder_dataset = torch.utils.data.ConcatDataset([fm2_train, fm2_val, fm2_test, fm1_train])
49
50 def train_autoencoder():
51     for run_index, hyperparams in enumerate(ParameterGrid(hyperparam_grid)):
52         train_batch_size = hyperparams['batch_size']
53         dataloader = torch.utils.data.DataLoader(encoder_dataset, batch_size = train_batch_size,
54             shuffle = True)
```

```

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

```

```

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

```

```

164         d = d.to(device)
165         l = l.to(device)
166
167         v_output = mlp_model(d)
168         v_loss = c(v_output, l)
169         val_running_loss += v_loss.item()
170
171     # print('Validation loss ', (val_running_loss / len(fm1_val)))
172     val_loss.append((val_running_loss / len(fm1_val)))
173     val_running_loss = 0
174
175     for d, l in dataloader_test:
176         d = d.view(-1, 784).type(torch.FloatTensor)
177         d = d.to(device)
178         l = l.to(device)
179
180         t_output = mlp_model(d)
181         t_loss = c(t_output, l)
182         test_running_loss += t_loss.item()
183
184     test_loss.append((test_running_loss / len(fm1_test)))
185     test_running_loss = 0
186
187 # return errors, val_errors
188 return train_loss, val_loss, test_loss
189
190 def weights_init(m):
191     global index
192     if type(m) == nn.Linear:
193         m.weight = torch.nn.Parameter(model.state_dict()['encoder_struct.' + str(index) +
194             '.weight'])
195         m.bias = torch.nn.Parameter(model.state_dict()['encoder_struct.' + str(index) + '.bias'])
196         index += 2
197
198 def weighted_MLP():
199     # mlp_model = MLP(struct).to(device)
200     # mlp_model.apply(weights_init)
201
202     sizes = [1.0]
203     len_dataset = len(fm1_train)
204     dataaaa = torch.utils.data.DataLoader(fm1_train)
205
206     errors = []
207     train_errors = []
208     val_errors = []
209     test_errors = []
210
211     for dataset_sizes in sizes:
212         print('Dataset Size ', dataset_sizes)
213         split_size = int(len_dataset * dataset_sizes)
214         fm1_train_split, other_split = torch.utils.data.random_split(fm1_train, [split_size,
215             (len_dataset - split_size)])
216         dataloader = torch.utils.data.DataLoader(fm1_train_split, batch_size=32, shuffle = True)
217         dataloader_val = torch.utils.data.DataLoader(fm1_val, batch_size=32, shuffle = True)
218         dataloader_test = torch.utils.data.DataLoader(fm1_test, batch_size=32, shuffle=True)

```

```

218 # Create Model
219 global index
220 index = 0
221 mlp_model = MLP(struct).to(device)
222 mlp_model.apply(weights_init).to(device)
223
224 c = nn.CrossEntropyLoss()
225 optimizer = optim.Adam(mlp_model.parameters(), lr = 0.01)
226
227 current_epoch = 0
228 max_epochs = 15
229 min_epochs = 5
230
231 running_loss = 0
232 val_running_loss = 0
233 test_running_loss = 0
234
235 best_loss = 10000
236 this_loss = []
237
238 while (current_epoch <= max_epochs):
239     current_epoch += 1
240     mlp_model.train()
241
242     for d, label in dataloader:
243         d = d.view(-1, 784).type(torch.FloatTensor)
244         d = d.to(device)
245         label = label.to(device)
246
247         optimizer.zero_grad()
248         output = mlp_model(d)
249         loss = c(output, label)
250         loss.backward()
251         optimizer.step()
252
253         running_loss += loss.item()
254
255         print('Epoch ', current_epoch, ' loss ', (running_loss / len(fm1_train_split)))
256         this_loss.append(running_loss / len(fm1_train_split))
257         train_errors.append(running_loss / len(fm1_train_split))
258         running_loss = 0
259
260     mlp_model.eval()
261     with torch.no_grad():
262         for d, l in dataloader_val:
263             d = d.view(-1, 784).type(torch.FloatTensor)
264             d = d.to(device)
265             l = l.to(device)
266
267             v_output = mlp_model(d)
268             v_loss = c(v_output, l)
269             val_running_loss += v_loss.item()
270             print('Validation loss ', (val_running_loss / len(fm1_val)))
271             val_errors.append((val_running_loss / len(fm1_val)))
272             val_running_loss = 0
273

```

```
274     for d, l in dataloader_test:
275         d = d.view(-1, 784).type(torch.FloatTensor)
276         d = d.to(device)
277         l = l.to(device)
278
279         t_output = mlp_model(d)
280         t_loss = c(t_output, l)
281         test_running_loss += t_loss.item()
282
283     test_errors.append((test_running_loss / len(fm1_test)))
284     test_running_loss = 0
285
286     errors.append(this_loss[-1])
287
288 return train_errors, val_errors, test_errors
```

Appendix F Q5 code

```
1 for i in range(5, 105, 5):
2
3     # specify model architecture
4     # PyTorch model class, which takes the specified hyperparameters as inputs
5     class CNN(nn.Module):
6
7         def __init__(self, num_classes, num_dense_layers, num_conv_layers):
8             super(CNN, self).__init__()
9             self.num_dense_layers = num_dense_layers
10            self.num_conv_layers = num_conv_layers
11
12            # single conv layer
13            if num_conv_layers == 1:
14                self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
15                self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
16
17            # dense layers
18            if num_dense_layers == 1:
19                self.fc1 = nn.Linear(6*(12**2), num_classes)
20            elif num_dense_layers == 2:
21                self.fc1 = nn.Linear(6*(12**2), 80)
22                self.fc2 = nn.Linear(80, num_classes)
23            elif num_dense_layers == 3:
24                self.fc1 = nn.Linear(6*(12**2), 120)
25                self.fc2 = nn.Linear(120, 84)
26                self.fc3 = nn.Linear(84, num_classes)
27
28            # two conv layers
29            elif num_conv_layers == 2:
30                self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
31                self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
32                self.conv2 = nn.Conv2d(in_channels=6, out_channels=12, kernel_size=5)
33                self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
34
35            # dense layers
36            if num_dense_layers == 1:
37                self.fc1 = nn.Linear(12*(4**2), num_classes)
38            elif num_dense_layers == 2:
39                self.fc1 = nn.Linear(12*(4**2), 80)
40                self.fc2 = nn.Linear(80, num_classes)
41            elif num_dense_layers == 3:
42                self.fc1 = nn.Linear(12*(4**2), 120)
43                self.fc2 = nn.Linear(120, 84)
44                self.fc3 = nn.Linear(84, num_classes)
45
46        def forward(self, x):
47
48            # conv2d and maxpools
49            if self.num_conv_layers == 1:
50                x = self.pool1(F.relu(self.conv1(x)))
51                x = x.view(-1, 6*(12**2))
52            elif self.num_conv_layers == 2:
53                x = self.pool1(F.relu(self.conv1(x)))
54                x = self.pool2(F.relu(self.conv2(x)))
```

```

55         x = x.view(-1, 12*(4**2))
56
57     # fully-connected layers
58     if self.num_dense_layers == 1:
59         x = self.fc1(x)
60     if self.num_dense_layers == 2:
61         x = F.relu(self.fc1(x))
62         x = self.fc2(x)
63     if self.num_dense_layers == 3:
64         x = F.relu(self.fc1(x))
65         x = F.relu(self.fc2(x))
66         x = self.fc3(x)
67
68     return x
69
70 # Breaking the training set into smaller fractions
71 percentage_to_use = int((i/100) * 25000)
72 percentage_not_to_use = 25000 - percentage_to_use
73
74 fm1_train_percentage, rest = torch.utils.data.random_split(fm1_train, [percentage_to_use,
75   ↪ percentage_not_to_use])
75 fm1_trainloader_percentage = torch.utils.data.DataLoader(fm1_train_percentage, batch_size =
76   ↪ train_batch_size, shuffle = True)
77
78 # Initialise the random model and send to GPU
79
80 model_ran = CNN(num_classes, hyperparams['num_dense_layers'],
81   ↪ hyperparams['num_conv_layers'])
82
83 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
84 print(device)
85 model_ran.to(device)
86
87 # Loss & optimiser
88 criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
89
90 optimiser = torch.optim.AdamW(model_ran.parameters(), lr=hyperparams['learning_rate'],
91   ↪ weight_decay=hyperparams['weight_decay'])
92
93 # training loop
94
95 train_loss_list_ran = []
96 train_accuracy_list_ran = []
97 dev_loss_list_ran = []
98 dev_accuracy_list_ran = []
99
100 best_dev_loss_ran = 9999 # initiate high value so that training doesn't immediately stop
101
102 converged = False
103 epochs_counter = 0
104 epochs_to_plot_ran = []
105
106 while (converged == False) and (epochs_counter <= max_epochs):

```

```

107 train_running_loss = 0
108 train_running_corrects = 0
109 train_num_of_batches = 0
110
111 model_ran.train()
112
113 for data, label in fm1_trainloader_percentage:
114
115     data = data.to(device)
116     label = label.to(device)
117
118     # zero the gradients
119     optimiser.zero_grad()
120
121     # forward + backward + optimize
122     train_preds = model_ran(data)
123     loss = criterion(train_preds, label.long())
124     loss.backward()
125     optimiser.step()
126
127     # metrics
128     train_preds_indices = torch.argmax(train_preds.data, 1)
129     train_running_corrects += (train_preds_indices == label).sum().item()
130     train_running_loss += loss.item()
131     train_num_of_batches +=1
132
133 train_accuracy = train_running_corrects/(train_num_of_batches*train_batch_size)
134 train_accuracy_list_ran.append(train_accuracy)
135 train_loss = train_running_loss/train_num_of_batches
136 train_loss_list_ran.append(train_loss)
137
138 # Validation Loop
139
140 model_ran.eval()
141
142 with torch.no_grad():
143
144     dev_running_loss = 0
145     dev_running_corrects = 0
146     dev_num_of_batches = 0
147
148 for data, label in fm1_devloader:
149
150     data = data.to(device)
151     label = label.to(device)
152
153     # forward
154     dev_preds = model_ran(data)
155     loss = criterion(dev_preds, label.long())
156
157     # metrics
158     dev_preds_indices = torch.argmax(dev_preds.data, 1)
159     dev_running_corrects += (dev_preds_indices == label).sum().item()
160     dev_running_loss += loss.item()
161     dev_num_of_batches +=1
162

```

```

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

```

```

218     test_loss_ran = test_running_loss/test_num_of_batches
219
220     print(f"BEST RANDOM INITIALISATION MODEL TEST ACCURACY: {test_accuracy_ran}")
221     print(f"BEST RANDOM INITIALISATION MODEL TEST LOSS: {test_loss_ran}")
222
223     # Save results from epoch with best validation loss
224
225     hyperparams['best_epoch'] = best_epoch_ran
226     hyperparams['best_validation_loss'] = best_dev_loss_ran
227     hyperparams['associated_validation_accuracy'] = best_dev_accuracy_ran
228     hyperparams['associated_training_loss'] = best_train_loss_ran
229     hyperparams['associated_training_accuracy'] = best_train_accuracy_ran
230     hyperparams['test_loss'] = test_loss_ran
231     hyperparams['test_accuracy'] = test_accuracy_ran
232     hyperparams['Fraction of data (%)'] = i
233
234
235     results_df = results_df.append(hyperparams, ignore_index=True)
236     results_df = results_df.sort_values("best_validation_loss", ascending=False).round(4)
237     results_df.to_csv(f"{output_path}RESULTS_TABLE_random_initial.csv")
238
239     print(hyperparams)
240
241     # save the model
242     torch.save(best_model,
243                f"{output_path}MODEL_{best_dev_loss_ran:.4f}_devloss_{best_dev_accuracy_ran:.4f}_devacc_fmnist1.pth"
244
245 ######
246 ######
247
248     # Load Model
249     model = CNN(num_classes, hyperparams['num_dense_layers'], hyperparams['num_conv_layers'])
250     weights_path = f"{output_path}MODEL_0.1715_devloss_0.9404_devacc_state_dict.pth"
251     model.load_state_dict(torch.load(weights_path, ))
252     for param in model.parameters():
253         param.requires_grad = False
254
255     # Replace the last fully-connected layer
256     # Parameters of newly constructed modules have requires_grad=True by default
257     model.fc3 = nn.Linear(84, num_classes)
258
259     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
260     print(device)
261     model.to(device)
262
263     # Loss & optimiser
264     criterion = nn.CrossEntropyLoss() # note that this will apply the softmax for us
265
266     optimiser = torch.optim.AdamW(model.parameters(), lr=hyperparams['learning_rate'],
267                                  weight_decay=hyperparams['weight_decay'])
268
269     # training loop
270
271     train_loss_list = []

```

```

272 train_accuracy_list = []
273 dev_loss_list = []
274 dev_accuracy_list = []
275
276 best_dev_loss = 9999 # initiate high value so that training doesn't immediately stop
277
278 converged = False
279 epochs_counter = 0
280 epochs_to_plot = []
281
282 while (converged == False) and (epochs_counter <= max_epochs):
283
284     train_running_loss = 0
285     train_running_corrects = 0
286     train_num_of_batches = 0
287
288     model.train()
289
290     for data, label in fm1_trainloader_percentage:
291
292         data = data.to(device)
293         label = label.to(device)
294
295         # zero the gradients
296         optimiser.zero_grad()
297
298         # forward + backward + optimize
299         train_preds = model(data)
300         loss = criterion(train_preds, label.long())
301         loss.backward()
302         optimiser.step()
303
304         # metrics
305         train_preds_indices = torch.argmax(train_preds.data, 1)
306         train_running_corrects += (train_preds_indices == label).sum().item()
307         train_running_loss += loss.item()
308         train_num_of_batches +=1
309
310         train_accuracy = train_running_corrects/(train_num_of_batches*train_batch_size)
311         train_accuracy_list.append(train_accuracy)
312         train_loss = train_running_loss/train_num_of_batches
313         train_loss_list.append(train_loss)
314
315
316     model.eval()
317
318     with torch.no_grad():
319
320         dev_running_loss = 0
321         dev_running_corrects = 0
322         dev_num_of_batches = 0
323
324         for data, label in fm1_devloader:
325
326             data = data.to(device)
327             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:
348           {train_accuracy:.4f}, val_loss: {dev_loss:.4f}, val_acc: {dev_accuracy:.4f}")
349
350     # Store the current results if it has the smallest validation loss so far
351     if dev_loss < best_dev_loss:
352         best_dev_accuracy = dev_accuracy
353         best_dev_loss = dev_loss
354         best_train_accuracy = train_accuracy
355         best_train_loss = train_loss
356         best_epoch = epochs_counter
357         best_model = model
358
359     # Early stopping criteria
360     if epochs_counter >= min_epochs:
361         if (np.mean(dev_loss_list[-20:]) - np.mean(dev_loss_list[-40:-20])) > 0.001:
362             converged = True
363             print("Model Converged")
364
365     # Evaluate loss and accuracy on test set
366
367     with torch.no_grad():
368
369         test_running_loss = 0
370         test_running_corrects = 0
371         test_num_of_batches = 0
372
373         predictions_list = []
374         labels_list = []
375
376         for data, label in fm1_testloader:
377
378             data = data.to(device)
379             label = label.to(device)
380
381             # forward
382             test_preds = best_model(data)
383             loss = criterion(test_preds, label.long())

```

```

383
384     # metrics
385     test_preds_indices = torch.argmax(test_preds.data, 1)
386     test_running_corrects += (test_preds_indices == label).sum().item()
387     test_running_loss += loss.item()
388     test_num_of_batches += 1
389
390     labels_list.append(label.long().item())
391     predictions_list.append(test_preds_indices.item())
392
393     test_accuracy = test_running_corrects/(test_num_of_batches*test_batch_size)
394     test_loss = test_running_loss/test_num_of_batches
395
396     print(f"BEST MODEL TEST ACCURACY: {test_accuracy}")
397     print(f"BEST MODEL TEST LOSS: {test_loss}")
398
399     # Save results from epoch with best validation loss
400
401     hyperparams['best_epoch'] = best_epoch
402     hyperparams['best_validation_loss'] = best_dev_loss
403     hyperparams['associated_validation_accuracy'] = best_dev_accuracy
404     hyperparams['associated_training_loss'] = best_train_loss
405     hyperparams['associated_training_accuracy'] = best_train_accuracy
406     hyperparams['test_loss'] = test_loss
407     hyperparams['test_accuracy'] = test_accuracy
408     hyperparams['Fraction of data (%)'] = i
409
410
411     results_df = results_df.append(hyperparams, ignore_index=True)
412     results_df = results_df.sort_values("best_validation_loss", ascending=False).round(4)
413     results_df.to_csv(f"{output_path}RESULTS_TABLE_TL.csv")
414
415
416     print(hyperparams)
417
418     # save the model
419     torch.save(best_model,
420                f"{output_path}MODEL_{best_dev_loss:.4f}_devloss_{best_dev_accuracy:.4f}_devacc_fmnist1.pth")
421
422     # Loss plot
423     plt.plot(epochs_to_plot_ran, train_loss_list_ran, color='k', linestyle='--')
424     plt.plot(epochs_to_plot_ran, dev_loss_list_ran, color='r', linestyle='--')
425     plt.plot(epochs_to_plot, train_loss_list, color='b', linestyle='--')
426     plt.plot(epochs_to_plot, dev_loss_list, color='orange', linestyle='--')
427     plt.legend(['Training (RI)', 'Validation (RI)', 'Training (TL)', 'Validation (TL)'],
428                loc='upper right')
429     plt.ylabel('Loss', color='k')
430     plt.xlabel('Epoch', color='k')
431     plt.title(f'''Loss plot (Best val loss (RI) {best_dev_loss_ran:.4f}, Acc (RI):
432                  {best_dev_accuracy_ran:.4f}, Best val loss (TL) {best_dev_loss:.4f}, Acc (TL):
433                  {best_dev_accuracy:.4f}) \n
434 Fraction of data use for training: {i}% \n
435 Conv layers: {hyperparams['num_conv_layers']}, Dense layers:
436                  {hyperparams['num_dense_layers']}, Batch size: {hyperparams['batch_size']},
437 Optimiser: {hyperparams['optimiser']}, LR: {hyperparams['learning_rate']}, Weight decay /
438 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()
```
