NEURAL NETWORKS & DEEP LEARNING / ECS659U COURSEWORK ASSESSMENT REPORT

RAI MUHAMMAD IBRAHIM BADAR / 190071703

INITIAL MODEL

Stem

The stem takes in a batch of images. It then creates an empty tensor X of shape (batch size, number of patches, patch size). For a K x K patch the patch size is K^2, this allows us to store the patch as a vector. A for loop is used to unfold each image on dimension 1 and then on dimension 2. The step and size for unfold is set to the patch dimension, K for patch size K x K. This returns a view of the all K x K patches of the image. This view is stored in the variable, patches. The view is then reshaped so we end up with a tensor of size (number of patches, patch size) where patch size is vector of size K x K. A variable called patch vector stores this tensor. The patch vector is then added to the tensor X. By the end of the for loop we are left with a tensor X which contains all the patch vectors of all the images in the batch. This tensor is passed through a linear layer to convert the patch vectors into feature vectors and the output of the linear layer is returned.

Backbone

The backbone doesn't consist of any blocks. It consists of two MLPS. Each MLP is a sequential container. MLP 1 follows $O_1 = g(X^TW_1)\ W_2$, where g is the activation function ReLU, X^T is the transpose of the tensor that is returned by the stem, W_1 and W_2 are the weights of linear layers in MLP 1. MLP 1 has a final step $O_1 = O_1^T$, which transposes the resulting tensor from linear layer 2 in the MLP. For the transposes I defined a class called Transpose which is a child of the torch.nn.Module, this just transposes the tensor passed into it. MLP 2 follows $O_2 = g(O_1W_3)\ W_4$, where O_1 is the output from MLP 1, g is an activation function and W_3, W_4 are the weights of the linear layers in MLP 2. The backbone returns the output of MLP 2.

Classifier

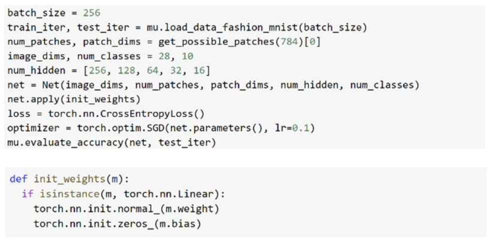
The classifier gets as input the output of the backbone. The classifier first calculates the mean of all the feature vectors of each image, using torch.mean. The mean features of all the images are stored as a tensor in the variable mean_feature. This is then passed into a linear layer which has an output set to the number of classes that we need to detect. The output of the linear layer is returned.

Net

The net class combines the stem, backbone, and the classifier. It passes in the batch of input images into the stem, passes the output of stem into the backbone and then passes the backbone's output into the classifier, finally it returns the output of the classifier as the prediction.

Initial Training

For the initial training script, I used the training scripts provided in the my-utils python file. I used a batch size of 256, then defined a function which works out the possible patches with an image of size 28 x 28 and used the first possible patch size to create the patches, a patch size of 14 x 14 with 4 patches. I used 256, 128, 64, 32 and 16 as outputs for the linear layers in the model. Furthermore, I initialised the weights using normal distribution and initialised the bias with 0. For the loss



```
batch_size = 256
train_iter, test_iter = mu.load_data_fashion_mnist(batch_size)
num_patches, patch_dims = get_possible_patches(784)[0]
image_dims, num_classes = 28, 10
num_hidden = [256, 128, 64, 32, 16]
net = Net(image_dims, num_patches, patch_dims, num_hidden, num_classes)
net.apply(init_weights)
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
mu.evaluate_accuracy(net, test_iter)
```

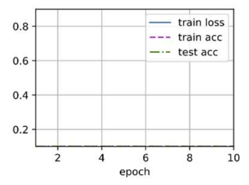
```
def init_weights(m):
  if isinstance(m, torch.nn.Linear):
    torch.nn.init.normal_(m.weight)
    torch.nn.init.zeros_(m.bias)
```

Figure 1

function I used Cross Entropy Loss as it uses softmax X. Finally, I used SGD for my optimizer with a learning rate of 0.1 (Figure 1). I ran the training script with 10 epochs and the results are as follows.

As it can be seen in the graph on the right, the accuracies for both training and testing are extremely low, 0.1, they can be barely seen on the graph (Figure 2). Initially I thought that there was something wrong with the model but upon verifying the code for the model I couldn't find anything wrong. This led me to try and optimize my model as it could be a hyper parameter that was limiting the accuracy of my model.

```
num_epochs = 10
train(net, train_iter, test_iter, loss, num_epochs, optimizer)
```



```
mu.evaluate_accuracy(net, test_iter)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: Use
  cpuset_checked))
0.1
```

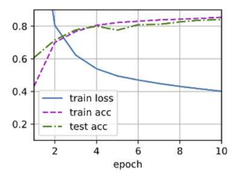
Figure 2

Optimizations

I tried increasing the number of epochs, altering batch number, and decreasing the learning rate, but the testing accuracy remained at 0.1 (I have not included the curves as they look the same as the ones in Figure 2).

The next thing I tried to optimise the model was changing the initialisation methods. The first initialisation method I tried was Xavier using torch.nn.init.xavier_normal_. As can be seen on the right, Xavier initialisation gave much better results. Loss was down and both training and testing accuracies were high. The testing accuracy after 10 epochs was 0.8404 (Figure 3).

```
num_epochs = 10
train(net, train_iter, test_iter, loss, num_epochs, optimizer)
```



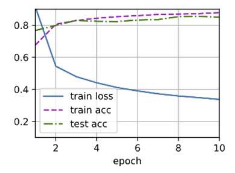
```
mu.evaluate_accuracy(net, test_iter)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarn
  cpuset_checked))
0.8404
```

Figure 3

Xavier initialization assumes non-existence of nonlinearities, this is easily violated, so, to account for this He initialization is a better option for the initialisation of weights. Naturally that's the optimisation I followed with, I implemented this by initialising the weights using torch.nn.init.kaiming_normal_. As can be seen in the diagram on the right, He initialization yielded even better results, a much lower loss, and much higher accuracies. The testing accuracy was observed to be 0.8508, that's more than 1% above the one that was obtained using Xavier (Figure 4).

```
num_epochs = 10
train(net, train_iter, test_iter, loss, num_epochs, optimizer)
```



```
mu.evaluate_accuracy(net, test_iter)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarni
  cpuset_checked))
0.8508
```

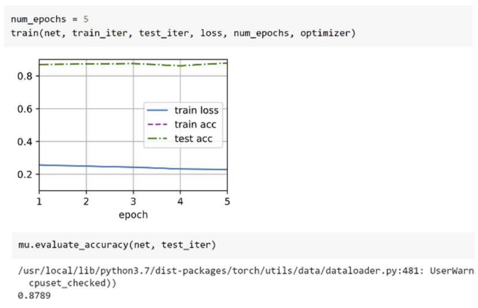
Figure 4

To try to improve the accuracies even further I planned on looking at learning rate, number of epochs and batch size. I planned to test testing accuracy for all possible combinations of batch size; 128, 256, 512, learning rate; 0.01, 0.1, 0.2 and number of epochs; 5, 10 and 15.

For batch size 256 and learning rate 0.1, the highest testing accuracy found was 0.8628 after training for 15 epochs. Changing the learning rate to 0.01 the highest testing accuracy was still with 15 epochs, but it was 0.7911 which was a deterioration. Changing the learning rate to 0.2 once again 15 epochs got the highest testing accuracy of 0.8628.

Moving on to batch size 512, starting with learning rate 0.1, the best accuracy was 0.8374 with 15 epochs. With a learning rate of 0.01 the highest testing accuracy was 0.7753 again with 15 epochs. A learning rate of 0.2 yielded the highest accuracy of 0.8591 with 15 epochs again.

Finally, with a batch size of 128, the learning rate of 0.1 increased the testing accuracy to 0.8688 but this was with 10 epochs this time. Decreasing the learning rate to 0.01 yielded the best accuracy of 0.8418 with 15 epochs. Increasing the learning rate to 0.2 yielded with an even better test accuracy of 0.8789 (Figure 5), again with 15 epochs (note that the image only shows 5 epochs that's because I trained the model 3 times with 5 epochs each time).



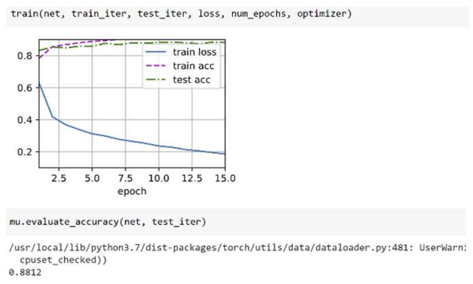
```
num_epochs = 5
train(net, train_iter, test_iter, loss, num_epochs, optimizer)
```

```
mu.evaluate_accuracy(net, test_iter)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarn
  cpuset_checked))
0.8789
```

Figure 5

For the next optimization I tried Adam as an optimizer instead of SGD as it combines momentum with RMS-prop and solves the gradient descent problem and produces a different effective learning rate for all optimization variables, however, this returned a much worse accuracy of 0.1. I tried to change the learning rate and found the accuracy increase to 0.8812 (Figure 6), with a learning rate of 0.001, when training for 15 epochs.



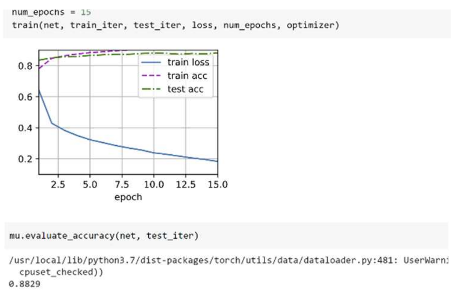
```
train(net, train_iter, test_iter, loss, num_epochs, optimizer)
```

```
mu.evaluate_accuracy(net, test_iter)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarni
  cpuset_checked))
0.8812
```

Figure 6

Next, I tried to add weight decay, after trying values of 0.1 to 0.000001, I observed the test accuracy increase to 0.8829 with a weight decay of 0.00001 (Figure 7).



```
num_epochs = 15
train(net, train_iter, test_iter, loss, num_epochs, optimizer)
```

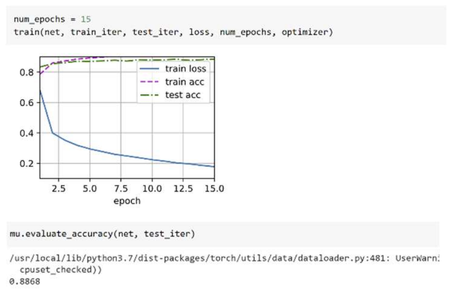
```
mu.evaluate_accuracy(net, test_iter)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarni
  cpuset_checked))
0.8829
```

Figure 7

To try to further improve the accuracy I tried to add dropout layers with different probabilities using torch.nn.Dropout, but the testing accuracy did not improve. Hence, I tried to implement batch normalisation using torch.nn.BatchNorm1d. Adding batch normalisation between each layer increased the testing accuracy to 0.8868 (Figure 8).

To try to improve the accuracy even further I implemented data augmentation (the code for this is explained in the ipynb file). I used RandomAffine to shift the images vertically and horizontally, RandomHorizontalFlip to flip the images horizontally and RandomErasing to randomly delete rectangle areas from the images. All these are part of the transforms module imported from torchvision. However, this decreased the accuracy to 0.8797. I left the code for data augmentation in the script but commented it out as it doesn't increase my accuracy.



```
num_epochs = 15
train(net, train_iter, test_iter, loss, num_epochs, optimizer)
```

```
mu.evaluate_accuracy(net, test_iter)
```

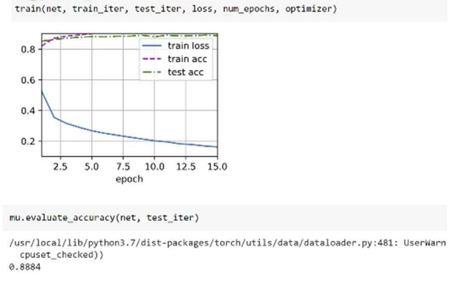
```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarni
  cpuset_checked))
0.8868
```

Figure 8

Final Model & Accuracy

After all the optimization tests a lower batch size seemed to produce better accuracies. I tried to lower the batch size even further and found the best testing accuracy of 0.8884 (Figure 9). Below are all settings and hyperparameters used for this result. Note that these hyperparameters have been set in the ipynb file that accompanies this report.

Optimiser: Adam, Learning Rate : 0.001, Weight Decay: 0.00001, Loss Function: CrossEntropyLoss, Batch Size: 32, Number of Epochs: 15, Batch Normalisation: True, Dropout: True with probability 0.01, Weight Initialiser: kaiming_normal.



```
num_epochs = 15
train(net, train_iter, test_iter, loss, num_epochs, optimizer)
```

```
mu.evaluate_accuracy(net, test_iter)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarn
  cpuset_checked))
0.8884
```

Figure 9