Yash Desai (210480735)

Neural Networks and Deep Learning Coursework

[Task 1] Read the dataset and create data loaders

The first step after importing all the necessary libraries is to load our dataset. I have used the datasets package from the *torchvision* library to download the *MNIST Fashion dataset*. Setting the *train* parameter true while extracting data let me extract the train and test datasets separately. I have used the transform package from the *torchvision* library to transform our data set to tensors and normalize the same. Further I have converted the dataset to *dataloaders* using the *DataLoader* function from the utils *package* in the *torch* library with a batch size of **128**.

```
train_iter = torch.utils.data.DataLoader(dataset=train_data, batch_size=batch_size, shuffle=True)
test_iter = torch.utils.data.DataLoader(dataset=test_data, batch_size=batch_size, shuffle=False)
```

I have showcased the process of patching in the python notebook where I create 49 patches from our 28x28 image. After creating our labels and visualizing our dataset, we move towards building our model. (Neural Network)

[Task 2] Create the model

```
NeuralNetwork(
  (StemLinear): Linear(in_features=16, out_features=256, bias=True)
  (batchnorm): BatchNorm1d(49, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (Block1_Linear1): Linear(in_features=49, out_features=512, bias=True)
  (Block1_Linear2): Linear(in_features=512, out_features=256, bias=True)
  (Block1_Linear3): Linear(in_features=256, out_features=128, bias=True)
  (Block1_Linear4): Linear(in_features=128, out_features=64, bias=True)
  (batchnorm1): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (Block1_ReLU): ReLU()
  (Block1_Dropout): Dropout(p=0.1, inplace=False)
  (Block2_Linear1): Linear(in_features=256, out_features=64, bias=True)
  (Block2_Linear2): Linear(in_features=64, out_features=128, bias=True)
  (Block2_Linear3): Linear(in_features=64, out_features=32, bias=True)
  (Block2_Linear4): Linear(in_features=32, out_features=16, bias=True)
  (batchnorm2): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (Block2_ReLU): ReLU()
  (Block2_Dropout): Dropout(p=0.2, inplace=False)
  (ClassifierLinear): Linear(in_features=16, out_features=10, bias=True)
)
```

Model layers and Components

Input Shape from Patch function: - (49,16)

Layer	Shape	Output
STEM	(16,256)	(49,256)
Batch Normalization (49)		
TRANSPOSE → (256, 49)		
Block 1 Layer 1	(49,512)	(256,512)
ReLU Activation		
Block 1 Layer 2	(512,256)	(256,256)
TRANSPOSE→ (256,256)		
Block 1 Layer 3	(256,128)	(256,128)
	ReLU Activation	
Block 1 Layer 4	(128,64)	(256,64)
Batch Normalization (256)		
TRANSPOSE→ (64 ,256)		
Block 2 Layer 1	(256,64)	(64,64)
ReLU Activation		
Block 2 Layer 2	(64,128)	(64,128)
TRANSPOSE → (128,64)		
Block 2 Layer 3	(64,32)	(128,32)
ReLU Activation		
Block 2 Layer 4	(32,16)	(128,16)
Batch Normalization (128)		
Classifier Linear Layer	(16,10)	(128,10)

[Task 3] Create the loss and optimizer

The loss function that we have used is the **Cross Entropy Loss** which calculates the loss while training our model. It further does the job of a SoftMax classifier that we add to our final output to create a probability distribution (values ranging from 0 to 1).

To **initialize model weights** I have used, the **kaiming_uniform_** initialization which is also known as the He initialization. This particular initialization was selected after trying and testing various other initialization techniques like the xavier_normal, xavier_uniform, kaiming_nomal etc. I was able to get the best results using **kaiming_uniform_b**.

Adam optimizer is used with a learning rate of **0.001**. Other optimizers like SGD (Stochastic Gradient Descent) and ASGD (Averaged Stochastic Gradient Descent) were also tested but Adam with the learning rate of **0.001** gives us the best accuracy. Further we have set the weight_decay to **0.0005.** Weight decay is just a regularization technique where we add a small penalty to our loss function, it is usually the L2 norm of the weights.

loss = loss + weight decay parameter* L2 norm of the weights

[Task 4] Training the model and plotting graphs for loss and accuracies

Due to the large dataset size, I experienced long runtimes for training my model, hence the training of my model is done on Google Colab Pro's GPU. I have set my main machine as the Google Colab Pro's GPU after turning on the GPU in Google Colab. After creating the device instance (GPU instance) I have transferred all my tensors on the GPU machine from the CPU using the command ***.to(device)*** after all the tensors. These include our main Neural Network model and our dataset tensors (both training and testing including the labels). The training process includes getting prediction from our model (y_hat), calculating the loss between predicted and actual labels, adding our optimizer followed by a back propagation. The functions used to train our model and display the accuracy vs loss curve were implemented from the my_utils python script (with added GPU compatibility) that was provided. In the train_epoch_ch3 function I have calculated the model loss which I have monitored using the scheduler. When loss curve becomes flat the scheduler will decrease our learning rate to improve the model performance.

Model Parameters:-

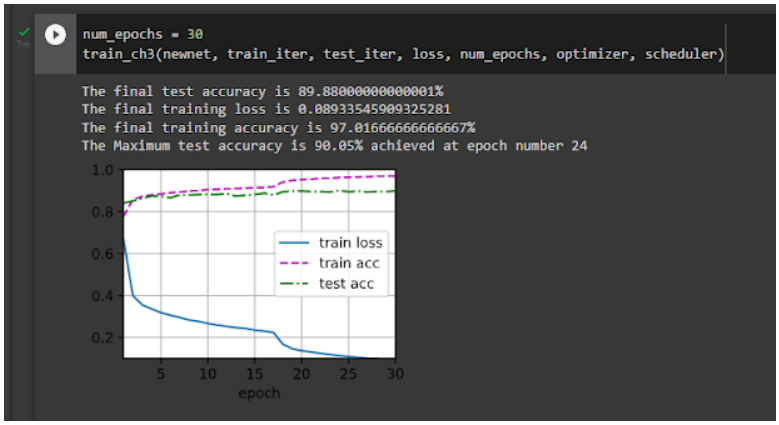
Batch Size: -
We have trained our model with a batch size of **128.** Ideally the batch size should be a power of '2' to take the full advantage of our GPU. Batch size is the size of one batch that we extract while training our data. This means our dataset of 60,000 images is broken into batch sizes of This is because the smaller the batch size the better the model will be able to learn minute details. Different batch sizes that I have tried including are {64,**128**,256,512,1024,2048,4096} out of which the best results were achieved using the batch size **128**.

Scheduler: -
While training our neural network, it was evident that the accuracy-loss curve becomes constant/flat after a certain point of time. This happens when model fails to learn new data and in turn fails to improve its performance. Hence, we have implemented a scheduler which will change (decrease) the learning rate of our model when the accuracy-loss curve becomes flat. The scheduler that we have used is **ReduceLROnPlateau** which is imported from lr_scheduler package of torch.optim. I have set the *patience* parameter for our scheduler as **15**. Moreover, I have set the *mode* of the scheduler as "**min**" while I monitor the test loss (average loss over

entire test data) of my model and the *threshold* parameter to "**1e-8**".This means, whenever the test loss stops increasing the scheduler will reduce the learning rate. The model will decrease its learning rate from **0.001** that was previously set to a **smaller rate**.

Epochs: -
The number of epochs used to train our neural network are **30.** We require less epochs than usual for our training because we have used Batch Normalization techniques which helps the gradient to converge faster. Several other epoch options were tried and tested like {10,20,**30**,40,50,100,200} with different learning rates, but maximum testing accuracy was achieved when we trained our model on **30** epochs. Training for more than 30 epochs resulted in a flat curve where the accuracy and loss both remain constant after an optimal point.

Optimizer: -
We have used the Adam optimizer with a **learning rate of 0.001** and a weight decay of **0.0005**. Other optimizers like SGD (Stochastic Gradient Descent) and ASGD (Averaged Stochastic Gradient Descent) were also tested but Adam with the learning rate of **0.001** gives us the best accuracy. Further we have set the weight_decay to **0.0005.** Weight decay is just a regularization technique where we add a small penalty to our loss function, it is usually the L2 norm of the weights.



Accuracy (Train and Testing) vs Loss Curve: -

As we can see from the accuracy vs loss (per epoch) curve, it is evident that as our loss decreases, the training and testing accuracy increases. This is because a reduction in loss implies a reduced prediction error in our model. This will in turn increase our testing accuracy. Moreover, a sudden increase in the accuracy and a reduction in loss is visible after **18** epochs, this is because we have implemented an adjustment in the learning rate using a scheduler with *patience* set at **15** (Reduction in learning rate takes place after 3 epochs form patience).

[Task 5] Final testing accuracy on the validation dataset

```
The Maximum test accuracy is 90.05% achieved at epoch number 24
```

```
The final test accuracy is 89.8800000000000%
```

After various permutations and combinations of all the hyper-parameters I have achieved a maximum testing accuracy of **90.05%** and a final testing accuracy of **89.89%** which is equivalent to **90%** (rounded to 2 significant figures). Finally, I would conclude that all the hyperparameters were responsible to boost the accuracy up till 90% but the key to achieve 90% accuracy is using a variable learning rate/ adding a scheduler when our model fails to improve performance after a certain point.