ECS659P: Neural Networks and Deep Learning
Coursework: Fashion-MNIST Problem

Akansh Katyayan

Task 1: Read Dataset and Create DataLoader

For the given task, Fashion MNIST data consists of gray scale images of size 28x28 having 10 labelled classes has been loaded using *load_data_fashion_mnist* function defined in the **my_utils.py** which uses *torchvision* library to load the data and return data loader train and test object. There are total 60000 training samples and 10000 test samples. These are accessed in batches by using next, iter functions. The batch size used is **256**. Sample images with their labels have been presented in the jupyter notebook.

Task 2: Create the model

Multi-Layer perceptron model has been created with the following parts: Stem, Backbone and Classifier. Stem performs two operations: Transform image into small patches (49 patches) of size 4x4 and convert into a vector and feed to Linear layer to extract features. Backbone created consists of 2 Blocks, each block containing 2 MLPS. In each MLP we are applying weights on the extracted features using the Linear layers. Final part of the model, which is classifier also perform two operations, first it takes the mean of the extracted features and finally feed it to the linear layer to generate output of size 10 as we have total 10 classes. Detailed description for the model has been added in the ipynb notebook.

For initializing the weights of the linear layers in the model, kaming_normal_ algorithm has been used as it accounts for the non-linearity of the activation functions, like ReLU activations. BatchNorm1d layer has been initialized with the constant weights.

Below is the model structure acquired:

```
MLPNet(
  (stem_linear_layer): Linear(in_features=16, out_features=256, bias=True)
  (batch_normalization_stem): BatchNorm1d(49, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (B1_linear1): Linear(in_features=49, out_features=256, bias=True)
  (B1_linear2): Linear(in_features=256, out_features=128, bias=True)
  (B1_linear3): Linear(in_features=256, out_features=128, bias=True)
  (B1_linear4): Linear(in_features=128, out_features=64, bias=True)
  (B1_batch_normalization): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (B1_ReLU): ReLU()
  (B1_Dropout): Dropout(p=0.1, inplace=False)
  (B2_linear1): Linear(in_features=128, out_features=64, bias=True)
  (B2_linear2): Linear(in_features=64, out_features=128, bias=True)
  (B2_linear3): Linear(in_features=64, out_features=32, bias=True)
  (B2_linear4): Linear(in_features=32, out_features=16, bias=True)
  (B2_batch_normalization): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (B2_ReLU): ReLU()
  (B2_Dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=16, out_features=10, bias=True)
)
```

Task 3: Create the Loss and Optimizer

Loss is calculated by comparing the difference between the predicted values of the model and the actual values. ***CrossEntropyLoss*** also called Log Loss has been used which calculates the score by penalizing the probability of the predicted values with the actual values. It calculates the performance of the classification model with output values between 0 and 1 which accomplishes the requirement of the Softmax classifier for the model.

Adam optimizer is used with a learning rate of 0.001. This optimization algorithm is an extension to stochastic gradient descent to update weights of the network during training. It inherits the features of AdaGrad and RMS Prop algorithms. the algorithm is easy to implement, and has faster run time, with low memory requirements, and it requires less tuning than another optimization algorithm. 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 the weight_decay has

been set to 0.0005. which is a regularization technique which adds a small penalty to the loss function i.e., usually the L2 norm of the weights.

Task 4: Training script to train the model:

Next task is to train the model. For training the neural network, GPU has been used for which code to use cuda has been added as part of the import section. To use the device instance, all the tensors have been transferred on the GPU machine from the CPU by applying **to(device)** method**.** To train the model, all the required functions like training, evaluation, generation of evaluation curves have been used from the **my_utils.py** which was provided as part of the labs. These functions have been included in the ipynb as the methods have also been executed using GPU for faster training.

BatchSize:

Batch size defines the number of images that will be propagated in the network. Model has been trained with a batch size of **256**. This means the dataset of 60,000 images is broken into batch size of 256. If batch size is less, it requires less memory, but processing time increases. On experimentation with different batch sizes (64, 128, 256, 512), the best results were achieved using the batch size of 256.

Optimizer:

As mentioned above in task 3, Adam optimizer with a **learning rate of 0.001** and a **weight decay of 0.0005** has been used. Other parameters like betas to compute running average of gradient and epsilon for numerical stability have used with the default values betas(0.9, 0.999) and eps=1e-8 respectively..

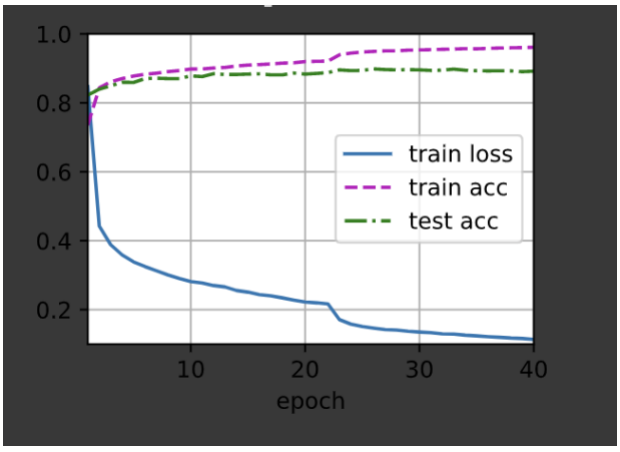
Epochs:

Increasing the number of epochs also increases the time taken by the model. Here, total 40 epochs havs been used to for training the model. Multiple runs with different epoch sizes ranging from 20 to 200 have been tested, however, marginal difference was observed on increasing the epochs after 40. Therefore, used the optimal epoch size of 40.

Scheduler:

On training the neural network, the accuracy-loss curve becomes constant after a certain number of epochs. This occurs when the model fails to learn the new data and thus doesn't show any different in the performance. To avoid the stagnant curve of reduction of loss, a scheduler has been implemented which changes the learning rate of the model when the accuracy-loss curve becomes constant. The ***ReduceLROnPlateau*** scheduler has been used which is imported from *lr_scheduler* package of *torch.optim*. The **patience** value has been set to **20**, which means the scheduler will observer the performance till first 20 epochs and then changes the learning rate. Scheduler has been applied on the loss from the test data that's why the parameter for variation is used as 'min' as we are trying to move in the direction of reducing the loss.

Below figure shoes the evolution of loss, training accuracy and test accuracy with the number of epochs.



The above accuracy vs loss (per epoch) curve shows that the training and testing accuracy increases as the loss decreases. Loss is calculated as the performance

between the actual and the predicted values. Reduction in loss implies that the difference between the actual and predicted values is less. i.e more accurate predictions, which results in increase in the testing accuracy. Loss in the first epoch started at the higher value of around 1.2 and the training and testing accuracies at approximately 80% which got better with the number of epochs. Moreover, a sudden increase in the accuracy and a reduction in loss is visible after 23 epochs, this is because of an adjustment in the learning rate by a scheduler with patience set at 20 (Reduction in learning rate takes place after 3 epochs form patience).

Task 5: Final model accuracy on Fashion-Mnist validation set

On evaluation of the model on the test_iter data which is 10000 images. The best accuracy achieved is **89.84%.** After the successful completion, the final metrics are as below:

Final test accuracy: 89.19%

Final Training Metrics:

Loss: 0.11

Training Accuracy: 96.11%

After manual hyperparameters tuning with various attempts, the maximum testing accuracy of **89.84% ~90%** (2 significant digits) In conclusion, the MLP structure used with various patched images helped in learning various spatial features and the all the hyperparameters used in loss, optimizer, epochs, and scheduler helped in improving the performance of the model.