Student Name: Benita Ashley
Student Number: 220058151
Module: ECS7026P - Neural Networks and Deep Learning

Kuzushiji - MNIST Classification

1. DATA LOADERS

To start, we import all the packages required to load the Kuzushiji MNIST dataset. As advised, we downloaded the dataset using the dataset package from the torchvision library. The train parameter was set to "True" during data extraction, allowing us to segregate the train and test data sets. Our dataset was then converted into tensors and normalised using the transform package from the torchvision library. Then, using a batch size of 128, we used the Dataloader function from the torch.utils.data package to turn the dataset into dataloaders. To assess the accuracy, we also tried 64, 256 and 512 batch sizes.

2. MODEL CREATION

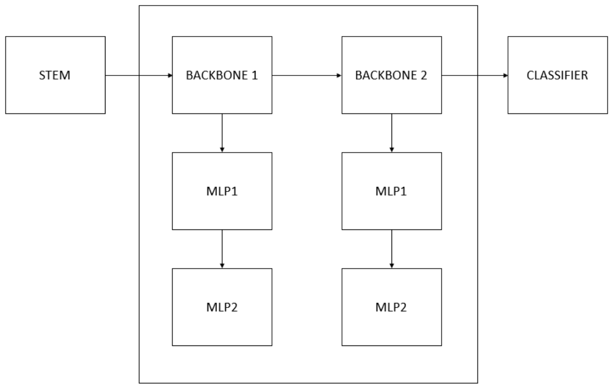


Fig 1: Model Architecture

- ***STEM***

The patch function, which is the first part of the STEM, takes a 28x28 input image from the Kuzushiji MNIST dataset and generates 49 non-overlapping patches which is set to 4, giving us 49 patches with a size of 4x4. The total number of pixels in our image is 784, and 16 pixels are required to make a patch that is 4x4 in size. All of the patches that are retrieved from our image are vectorized by the patch function as well. Our neural network's first layer now has an input size of 16, or 4x4 which is the size of our selected patch, and an output size equal to that of our hidden layer. A Linear Layer built using the Torch library serves as the STEM block's input layer. The STEM Linear layer's main job is to extract features. We perform batch normalisation after loading our picture patches in the Linear layer to enhance our model and hasten convergence. We can further experiment with different patch sizes for e.g. 2,7,14 which will give us 196, 16, 4 patches respectively.

- ***BACKBONE***

The backbone is made up of several smaller and larger hidden layers. Intricate details in our data are attempted to be captured by hidden layers, which aid in the discovery of numerous correlations between diverse inputs. The neural network model's several hidden layers enable us to address increasingly challenging classification and regression issues. In order to create the hidden layers, we are employing the Multi-Layer Perceptrons (MLP) architecture, which has a number of fully connected layers and can be used to tackle classification problems like the one we are working on.

As shown in fig 1, We have considered two backbone blocks, each of which consists of two MLPs with two hidden layers. The stem layer serves as the first MLP's input in block 1. Transposing the input is the initial step before feeding it to the first linear layer of the first MLP. Here, the input and output sizes are 49 and 512, respectively, so the linear layer 1's shape is (49,512). After taking the input and transposing and multiplying it to create Linear Layer 1, we use the activation function ReLu to activate our output. We then add a dropout layer to address overfitting after feeding the activation output to the Linear Layer 2. In MLP2, the same operations must be carried out.

- ***CLASSIFIER***

Taking the mean of all the learnt features serves as the initial operation in the categorization block. The outputs from the final backbone block are therefore averaged using torch.mean before being sent to the final classifier layer. The Classifier is a straightforward linear layer with an input shape equal to the 16-dimensional output shape of the final hidden layer. Consequently, our classifier layer is shaped like (16,10). When the output shape is set to the number of classes we need to predict, neural networks function as classifiers. Because there are 10 classes in our scenario, our classifier layer's output dimension is 10. To generate a probability distribution of our outputs, we must then use a classifier/regressor like SoftMax, but this is handled by the loss function that we will implement in the following step. We will use a Cross Entropy loss function to create an implicit conversion of our outputs to a probability distribution resembling the SoftMax function.

3. LOSS & OPTIMIZER

While our loss function, which determines the loss as we train our model, we have utilised the Cross Entropy Loss in order to construct probability distributions that have a range of 0 to 1, it also performs the function of a Softmax classifier. When training a model, we utilise a loss function to make it better. We also understand that the better the model, the lesser the loss.

We use the Adam Optimizer with a learning rate 0.001. The model was also tested with learning rates of 0.01, 0.05, and 0.001, but the latter yielded the best results. We also include a 0.0005 weight decay, a regularisation approach that applies a little penalty to the weights' l2 norm.

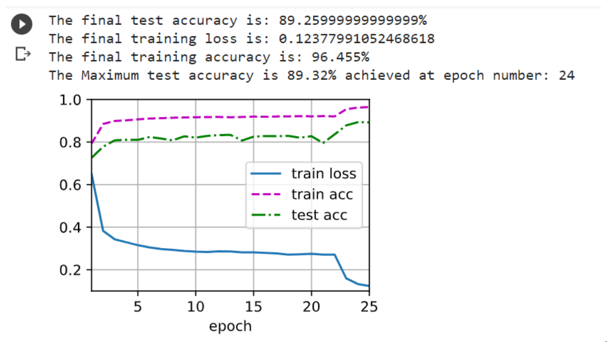
4. MODEL TRAINING AND EVALUATION

We've performed permutation and combination for various hyper parameters like the batch size, learning rate, number of epochs, schedulers, etc. We have trained our model with a 128 batch size. Batch size is set during the extraction of the dataset. The smaller the batch size, the better it is for the model to train on minute details. We have also experimented with batch sizes 64,128,256 and 512. While training our model we have observed that after a certain point the accuracy-loss curve goes flat. This usually results when the model fails to learn new data thus decreasing the accuracy hence we have implemented a scheduler so that when the accuracy-curve goes flat it makes the learning rate variable to increase the accuracy. We've chosen a learning rate of 0.001 in the Adam's optimiser as it gave the best result with batch size 128.

BATCH SIZE	LEARNING RATE	EPOCH	TRAIN ACCURACY	TEST ACCURACY	FIGURE
64	0.01	20	96.45	89.32	Fig 4.1

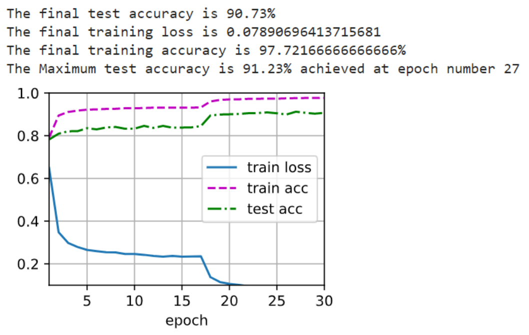
128	0.01	30	97.72	91.23	Fig 4.2
256	0.001	30	99.98	92.78	Fig 4.3

Table 4.1 - Model Evaluation and Comparison



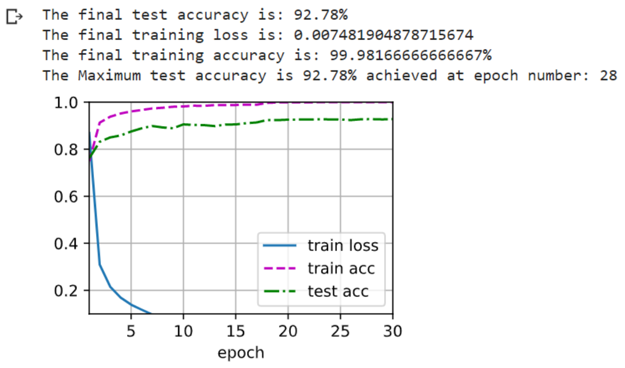
```
The final test accuracy is: 89.25999999999999%
The final training loss is: 0.12377991052468618
The final training accuracy is: 96.455%
The Maximum test accuracy is 89.32% achieved at epoch number: 24
```

Fig 4.1



```
The final test accuracy is 90.73%
The final training loss is 0.07890696413715681
The final training accuracy is 97.72166666666666%
The Maximum test accuracy is 91.23% achieved at epoch number 27
```

Fig 4.2

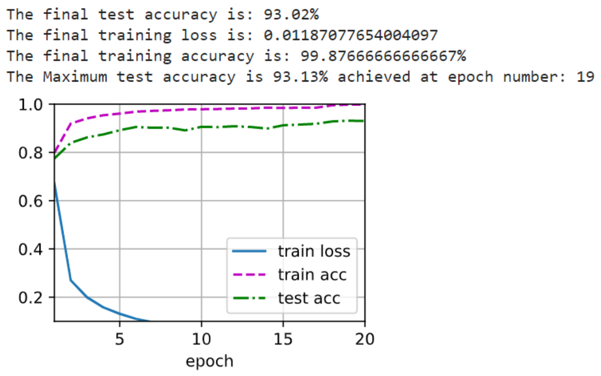


```
The final test accuracy is: 92.78%
The final training loss is: 0.007481904878715674
The final training accuracy is: 99.98166666666667%
The Maximum test accuracy is 92.78% achieved at epoch number: 28
```

Fig 4.3

5. <u>FINAL TESTING ACCURACY</u>

The best testing accuracy after permutation and combination on all the hyper-parameters is 93.13, while the greatest training accuracy after permutation and combination is 99.87. All of the hyper-parameters contribute to the model's increased accuracy, but the variable learning rate and the addition of a scheduler are given more weight because they aid to boost performance when the model stalls after a while. We've chosen a learning rate of 0.001 and 20 epochs with patience at 15 for the Adam's optimiser.



```
The final test accuracy is: 93.02%
The final training loss is: 0.01187077654004097
The final training accuracy is: 99.87666666666667%
The Maximum test accuracy is 93.13% achieved at epoch number: 19
```