Notes & Conclusions

CNN:

Best test error received on the trained model was 19.720 %.

Number of parameters = 49556.

The network has three layers, each layer containing convolution, batch normalization, a Relu function & Maxpooling with a 2x2 filter. And 2 final layers which are a fully connected layer which outputs 10 values for the 10 different categories & a dropout layer with p=0.1.

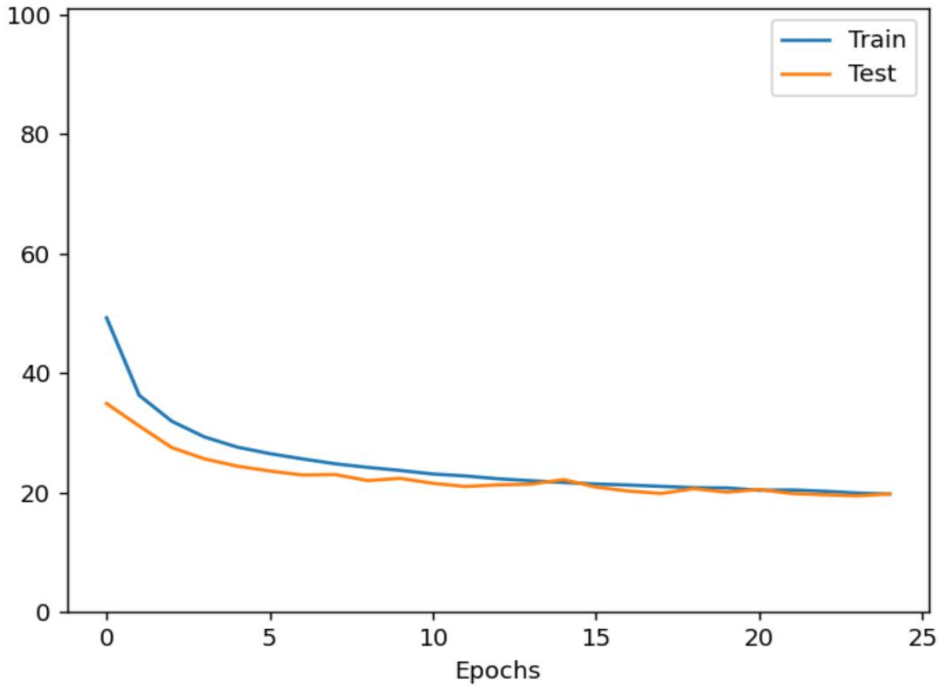
The number of filters of each convolution layer was picked by trying to minimize the error on the test with consideration to the limited number of parameters<50k. The convolution layers have 16, 32 & 82 filters respectively.

Other hyperparameters were learning rate = 0.001, batch size = 50 & number of epochs = 25. The network can reach lower error if the number of epochs is increased but considering running time, I kept it at 25.

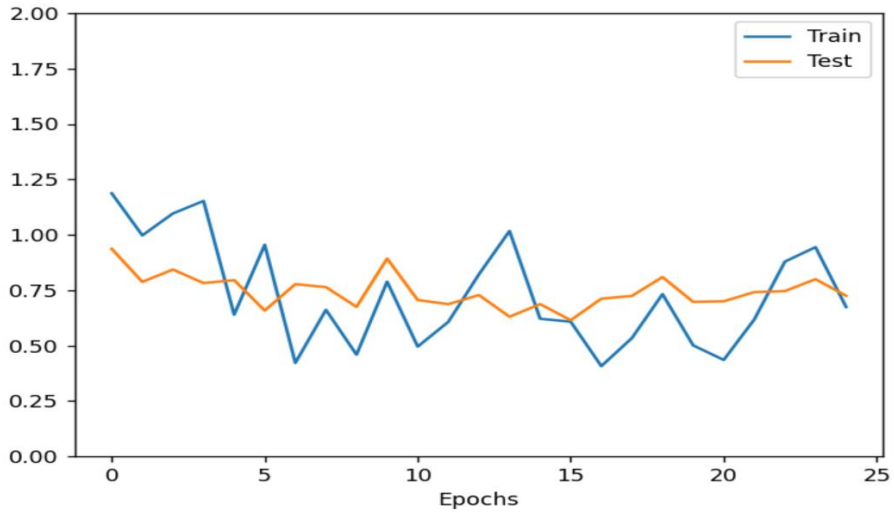
After multiple attempts without reaching the requested accuracy, I had to consider data augmentation to improve the accuracy. I applied random horizontal flipping, random cropping & random rotation on the full training set then I concatenated this new augmented set to the original training set and trained the network on the new dataset which contains 100k images. This and the initialization of the weights of convolution & linear layers using "torch.nn.init.xavier_normal_()" improved the network accuracy & allowed me to surpass the 80% accuracy threshold.

My main conclusion of this exercise is that the number of parameters can really affect the accuracy of a network, but if a network is built efficiently and the training data is manipulated and augmented in way to make it more informative, high accuracy results can be reached with a lower number of parameters.

Error percentage convergence graph:



Loss graph:



RNN:

I used one-hot encoding for the language "category" input as well as the input string. The RNN has just 3 linear layers, the first 2 layers of the network operate on an input and hidden state, plus a third linear layer to give it more depth. There's also a dropout layer with p=0.1 to prevent overfitting and at the end a LogSoftmax layer to apply on the output. I initiate the RNN with (input_size=number of letters, hidden_size=128, output_size= number of letters).

The model when given an input of language and a letter predicts the next letter. The model is trained on the names data with learning rate = 0.0005, by picking a random name and training on it. We do this 100k, we pick 1k names each epoch & we have around 100 epochs.

The name generation works this way: given input language & first letter. We add the first letter to the output then predict the next letter by using the trained RNN model with the given inputs. We keep doing this by using the last predicted letter as the input until we predict the next letter to be the end of a name. Then we return the output.

Examples of generated names:

Input:	Output:
'Arabic', 'J'	Jara
'Arabic', 'j'	jara
'Chinese', 'K'	Kon
'English', 'A'	Allin
'Italian', 'Z'	Zantin

Training Loss convergence graph: