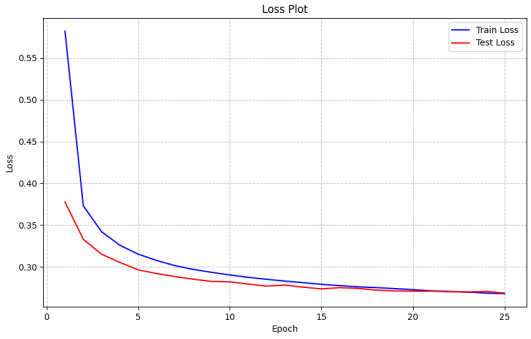
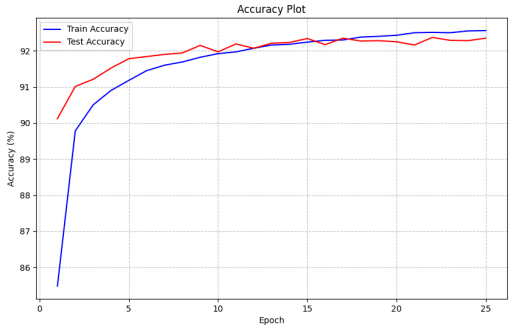
Name: Alex Koo (gk7244), Avani Bhute

Problem 1 [20p]: Logistic Regression (using TACC Machines)

Question 1: [5p] Modify ***starter.py*** by specifying the target device for your model (by adding ***model = model.to(torch.device('cuda'))*** at line 66) to train the model on GPU. Run ***starter.py*** on Lonestar6 (see **Appendix A1.1** for environment setup, **Appendix A1.2** to run your code, and **Appendix A1.3** for more information).

Question 2: [6p] Draw a plot with two curves: the *training loss* and the *test loss* of your model for each *epoch* (this is the *loss plot*). Draw a second plot with two curves: the *training accuracy* and *test accuracy* of your model for each *epoch* (this is the *accuracy plot*).





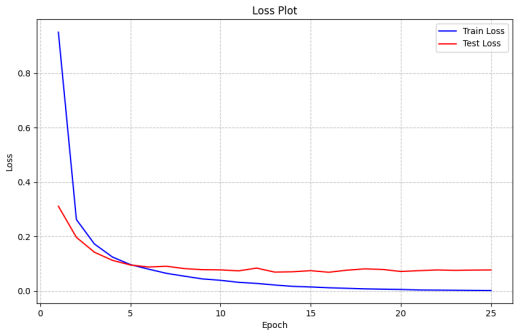
Question 3: [9p] Complete ***Table 1*** (use accuracy values from the final epoch):

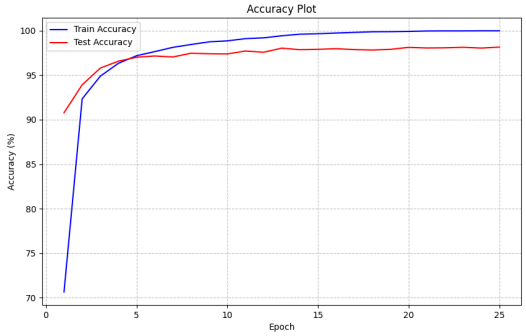
Training Accuracy (%)	Testing Accuracy (%)	Total time for training [s]	Total time for inference [s]	Average time for inference per image [ms]	GPU memory during training [MB]
92.56%	92.35%	19.34 seconds	0.61 seconds	0.06 ms	542MiB/4096MiB

Name: Alex Koo (gk7244), Avani Bhute

Problem 2 [40p]: Overfitting, Dropout and Normalization

Question 1: [8p] Run the code from ***simpleFC.py*** after specifying the target device for the model, images and labels. Draw the loss and accuracy plots (like in **Problem 1, Question 2** above). Does this model overfit? Explain.

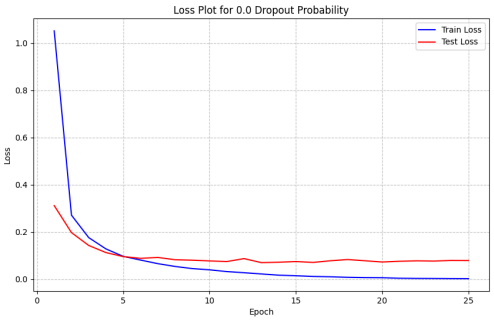


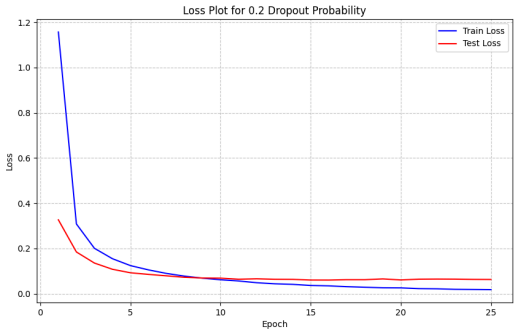


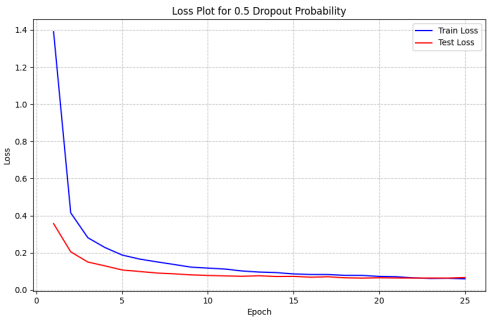
We can observe that around epoch 5-10, the training accuracy continues to improve while test accuracy plateaus. This leaves with the training accuracy approaching 100% while test accuracy stalls around 98%. Looking at the loss plot, the training loss continues to decrease towards 0 whereas test loss plateaus around epoch 5-7. This shows slight divergence indicating a slight overfitting but nothing significant as both curves are relatively stable and only 2-3% gap for the accuracy.

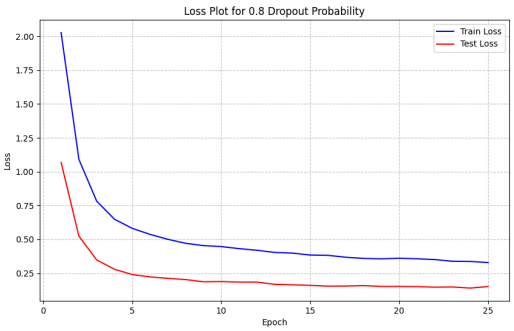
Question 2: [14p] In the ***__init__()*** method from the ***SimpleFC*** class, define once ***nn.Dropout(probability)*** and apply this new operation in the ***forward()*** method after every layer of the model except the last one. Run four different experiments using the values [0.0, 0.2, 0.5, 0.8] as probabilities for dropout. Draw one loss plot for each experiment. What do you observe from these plots? Which dropout probability gives the best/worst results (the best results have no overfitting, i.e., almost equal train and test loss values)? Explain.

Name: Alex Koo (gk7244), Avani Bhute









Best: Dropout 0.5
Worse: Dropout 0.8

The worst dropout is 0.8 as the value of the loss (~0.3265) and the accuracy (91.31%) of the test are both lower than the alternatives on the final epoch. Despite the poor training performance, it had robust generalization. However it indicated signs of over regularization with the significance difference between the training and testing data.

The best dropout is 0.0 based on the value of the loss (~0.0015) and the accuracy (99.99%). The training loss and accuracy shows minimum divergence with the test loss and accuracy while maintaining high accuracy and low loss. The lack of divergence between the train and test shows minimum sign of overfitting.

Question 3: [18p]. Complete ***Table 2*** by running the same code as in **Problem 2, Question 2**, but using only the best dropout rate. In ***Table 2***, replace X with the dropout rate which led to the best results in ***Problem 2, Question 2***. For the row with "+ norm", keep the same dropout probability and add normalization to both training and testing datasets. Compare and contrast these normalized and unnormalized experiments and explain the differences.

Dropout	Training Accuracy (%)	Testing Accuracy (%)	Total time for training (s)	First epoch when the model reaches 96% training accuracy
0.5	98.29	98.18	22.92	6
0.5 + norm	98.52	98.34%	22.89	6

The value between the model with only dropout applied and dropout with normalization applied is very minimal for both the training and testing accuracy as well as time to reach the first epoch when the model reaches 96% in training. The compose.normalization standardizes the dataset to have a mean of 0 and std of 1. This allows controlled standardization and more stable gradient flow (gradient = error * x_standardized). And since weight_update = learning_rate * gradient, it will hence affect

the weights updating. We suspect that the reason why the normalization wasn't improving the results was because MNIST has a behaved dataset with values 0-255 which toTensor already scales to 0 to 1. There is limited variance in pixel intensities. In addition, the dropout (0.5) was providing strong regularization.

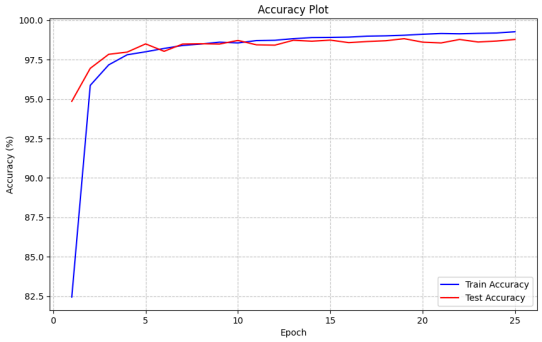
Problem 3 [40p + 10Bp]: CNNs and Model Complexity

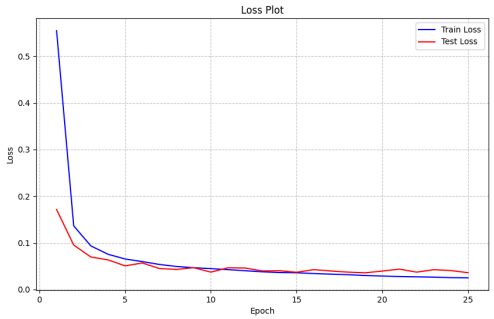
Question 1: [20p] Implement the normalized MNIST dataset from **Problem 2, Question 3** in ***simpleCNN.py***. In the same file, write the necessary code to complete ***Table 3*** and then run it. Is there any difference between the estimated (total) size of the model from ***torchsummary*** and the saved version of the model (i.e., file size on disk)? Explain.

Model name	MACs	FLOPs	# parameters	Torchsummary (total) size (KB)	Saved model size [KB]
SimpleCNN	3,869,824.0	7,739,648	50, 186	550 KB	198.724609375 KB
myCNN (1/16/32)	1,031,744	2,063,488	20,490	260 KB	80KB
myCNN (1/8/16 - FINAL)	290,080	580,160	9,098	130 KB	30 KB

The model size was around 198KB whereas the estimated total size was 550KB. This is because the Estimated Total size provided by TorchSummary provides the sum of input size, forward-backward pass size, and the param size. However the model size from the .pth files only includes the model's parameter/weights which was equal to the TorchSummary's param size. This is because forward/backward size is additional runtime memory needed during training and not part of the model file size. It's a temporary memory allocated to compute gradients and store intermediate activation functions.

Question 2: [20p] Create your own CNN with at least two convolutional layers and train it for 25 epochs on the normalized MNIST dataset. Try making this model more efficient (e.g., smaller number of parameters and/or smaller model size) compared to the SimpleCNN model from **Problem 3, Question 1**. Plot the loss and accuracy curves as a function of the #epochs. Extend ***Table 3*** further with a new row with your model name (e.g., "myCNN") and show the new results.





Name: Alex Koo (gk7244), Avani Bhute

BONUS Question 3: [10Bp] Explain your choice for the model architecture. Does your model overfit? How does your proposed model compare against SimpleCNN in terms of training and testing accuracy? In a real scenario, would you prefer using your proposed model or SimpleCNN? Why or why not?

To walk through the thought process:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 28, 28]	320
MaxPool2d-2	[-1, 32, 14, 14]	0
Conv2d-3	[-1, 64, 14, 14]	18,496
MaxPool2d-4	[-1, 64, 7, 7]	0
Linear-5	[-1, 10]	31,370

SimpleCNN shows that the biggest parameter costs are

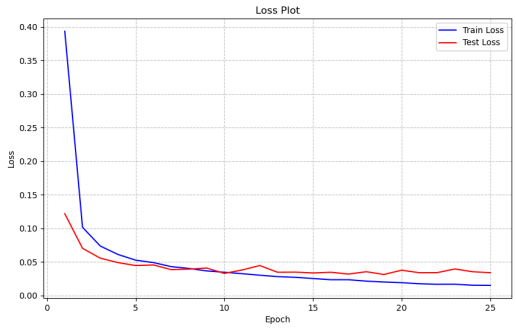
- Linear layer (31,370 params = 62% of model)
- Second conv layer (18,496 params = 37% of model)

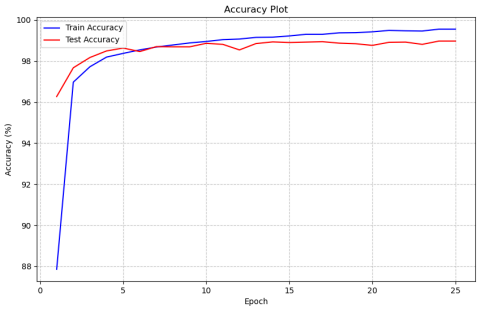
Based on this, we wanted to try reducing the 1 -> 32 -> 64 channel to 1 -> 16 -> 32 and see if the accuracy holds. As we reduce the channels, it means less parameters meaning it could potentially lead to less feature extraction capacity. Aka there is risk of the model underfitting.

We can verify if the data is underfitting through

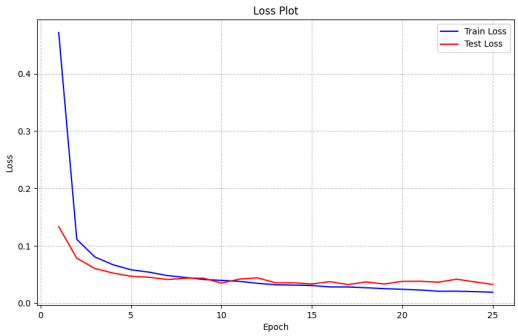
1. Both training and test accuracy being lower
2. Slower convergence rate in early epochs
3. Model struggling to reduce training loss

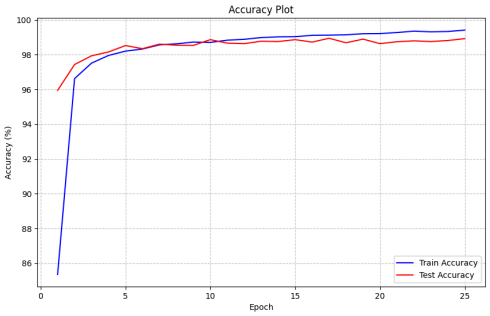
simpleCNN





myCNN with reduced feature map 1 -> 16 -> 32 (attempt #1)

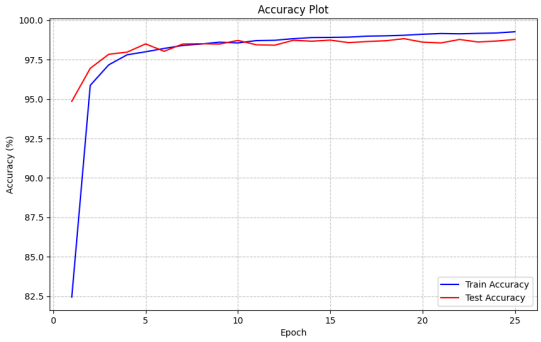


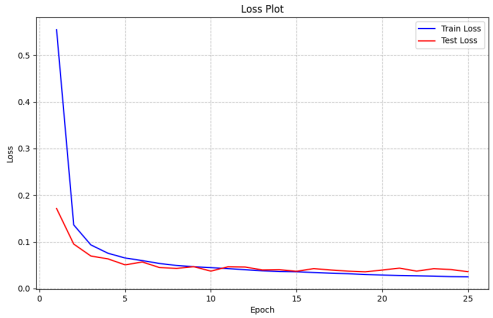


Name: Alex Koo (gk7244), Avani Bhute

While parameters reduced ~60%, MAC reduced ~73%, model size down from 0.19 MB to 0.08 MB, the performance came very close with final test accuracy of 98.97% vs 98.93% for the new CNN. Both models reach 98%+ by epoch 5 and shares very similar convergence patterns.

To test even further we've tried reducing the feature map to 1 -> 8 -> 16 for convolutional layer





While parameters reduced ~56%, MAC reduced ~72%, model size reduced by 50%, the final test accuracy only had a 0.15% drop from 98.93% to 98.78%. It still shows a similar convergence pattern and reaches 98% accuracy. This shows that a reduced feature map of 1 -> 8 -> 16 is the most viable option and it does not overfit. If we were deploying on a mobile/edge device, we would use the reduced model as it is significantly lower memory footprint, must faster inference, and 0.19% reduction in accuracy loss is likely unnoticeable. However if we were deploying on powerful servers with no resource constraint, we would still deploy the original model as there's no reason to give up even 0.19% accuracy and future dataset shifts might benefit from extra capacity.