ECE 361E: Edge Computing

Homework 1 Solutions

Problem 1: Logistic Regression for MNIST Classification

Question 1: Train Logistic Regression model and report metrics

Train a logistic regression model for 25 epochs on the normalized MNIST dataset. Report training accuracy, testing accuracy, total time for training, total time for inference, average time for inference per image, and GPU memory during training.

Answer:

The logistic regression model was trained for 25 epochs on the normalized MNIST dataset. The results are shown in Table 1 below:

Metric	Value
Training accuracy [%]	92.56
Testing accuracy [%]	92.35
Total time for training [s]	199.62
Total time for inference [s]	1.26
Average time for inference per image [ms]	0.1261
GPU memory during training [MB]	17.74

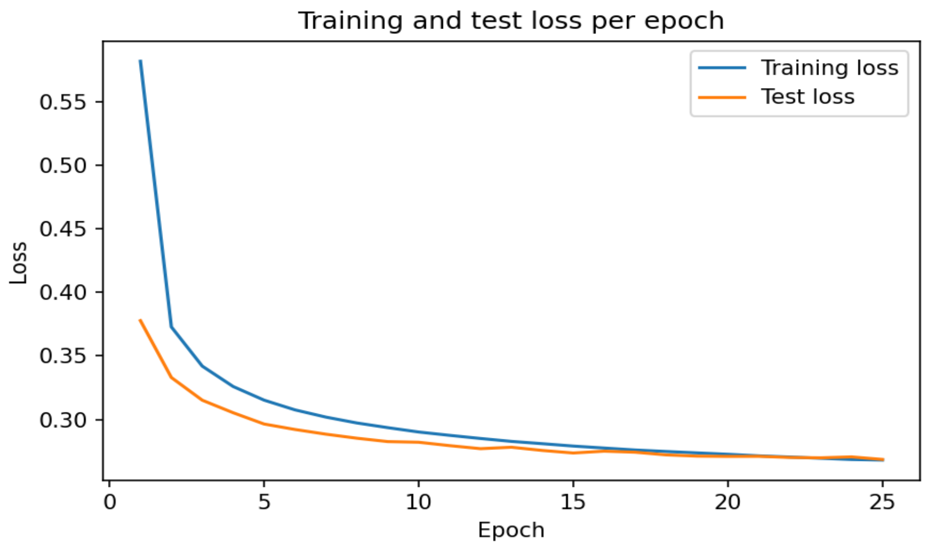
The model achieves approximately 92% accuracy on both training and test sets, indicating good generalization without overfitting. The inference time per image is quite fast at approximately 0.13 milliseconds.

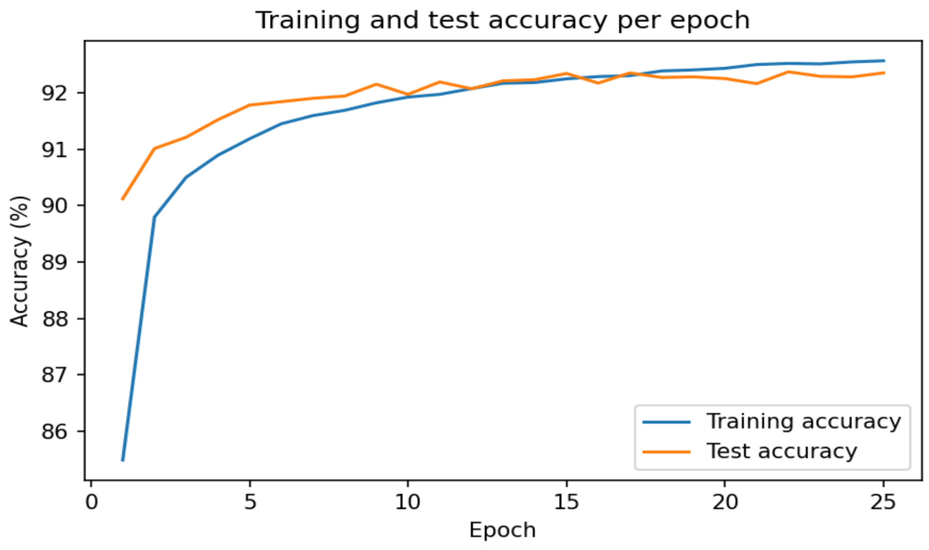
Question 2: Plot loss and accuracy curves

Plot the loss and accuracy curves as a function of epochs for both training and testing sets.

Answer:

The loss and accuracy curves for the logistic regression model are shown below. Both plots show steady improvement during training with the model converging around epoch 15-20.





The plots show that both training and test losses decrease steadily, while accuracies increase and plateau around 92%. The close alignment between training and test curves indicates no significant overfitting.

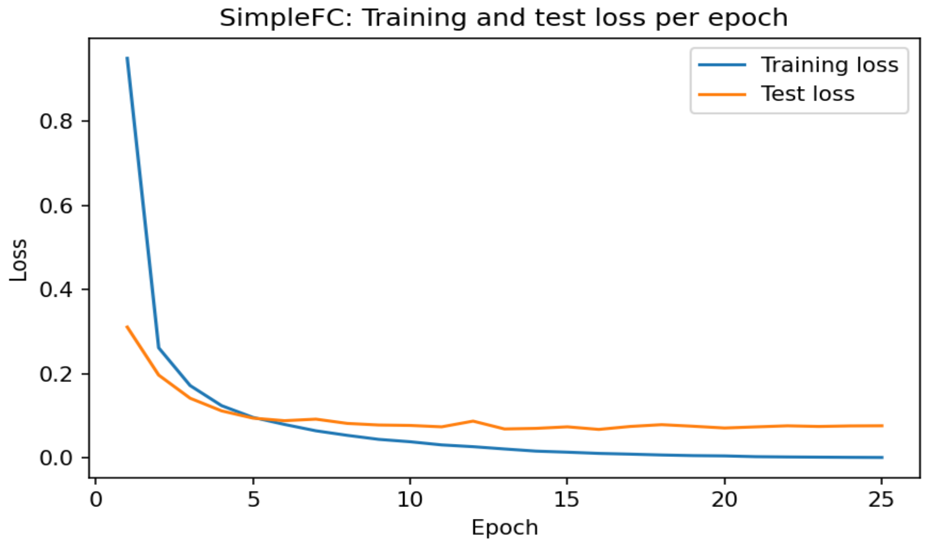
Problem 2: Fully Connected Neural Network for MNIST

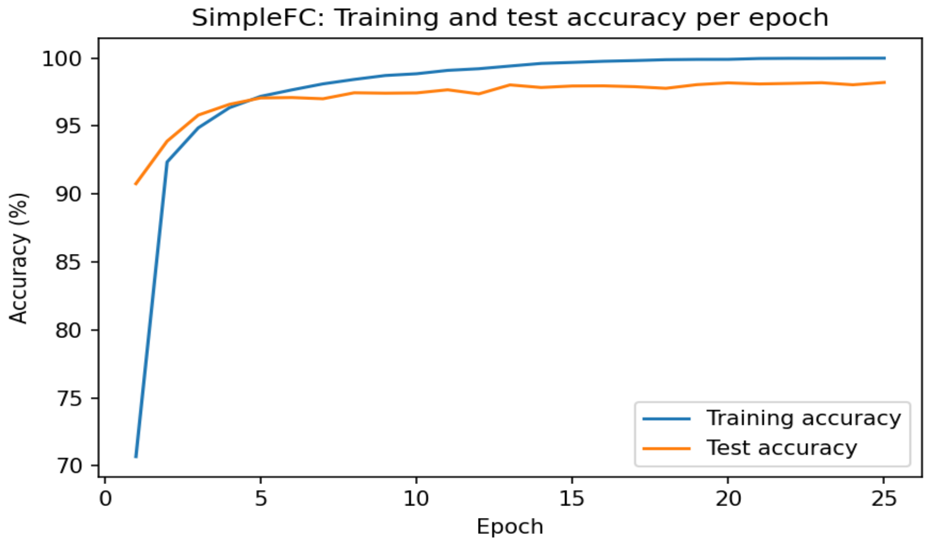
Question 1: Does the SimpleFC model overfit?

Train the SimpleFC model (4 fully connected layers) for 25 epochs and analyze whether it overfits.

Answer: Yes, the model overfits.

After viewing the resulting plots of p2_q1 (training/test loss and accuracy per epoch), we can clearly observe overfitting:





Evidence of Overfitting:

• **Training loss** decreases steadily to nearly zero (e.g., from ~0.95 at epoch 1 to ~0.0015 at epoch 25), while **test loss** improves at first (down to ~0.07) but then **plateaus** around 0.07-0.08 and does not improve (and even increases slightly) as training continues.

• **Training accuracy** rises to almost 100% (e.g., ~99.99% by epoch 25), while **test accuracy** improves to about 97-98% and then **plateaus** there for the rest of training.

The widening gap between training and test loss (and between very high train accuracy and lower, flat test accuracy) indicates that the model is memorizing the training set rather than learning features that generalize well. The SimpleFC network has many parameters (four fully connected layers), so without regularization (e.g., dropout) it tends to overfit. Adding dropout (as in Problem 2, Question 2) helps reduce this overfitting.

Question 2: Dropout experiments — what do you observe?

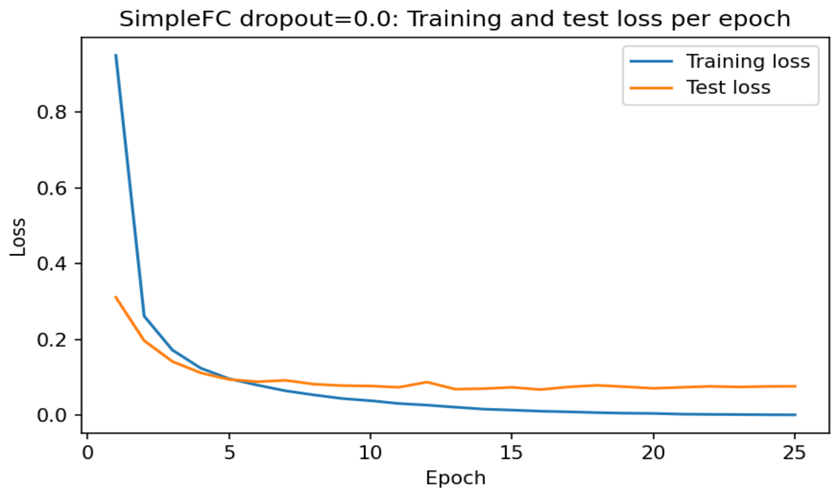
Train the SimpleFC model with different dropout probabilities (0.0, 0.2, 0.5, 0.8) and analyze the results. Which dropout probability gives the best and worst results?

Answer:

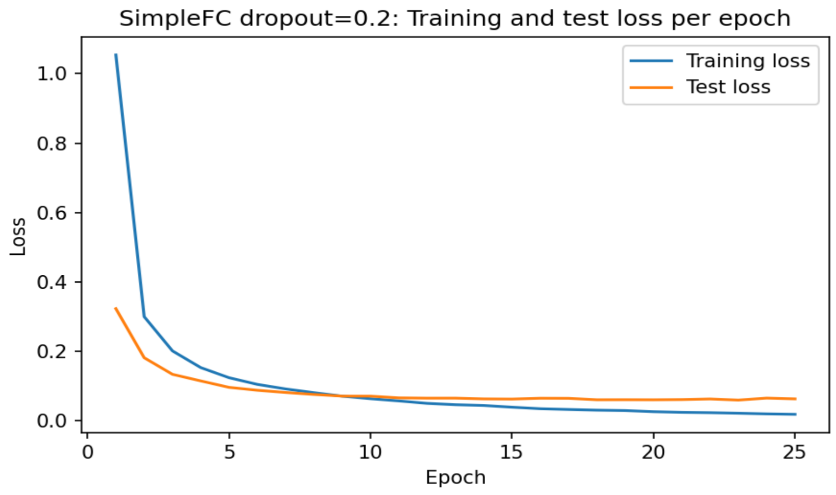
What do you observe from the loss plots?

Across the four experiments (dropout 0.0, 0.2, 0.5, 0.8), the loss plots show that as dropout increases, the **gap between training loss and test loss shrinks**: training loss no longer drops to nearly zero while test loss stays high. So dropout reduces overfitting. If dropout is too high, both losses stay higher and the model underfits.

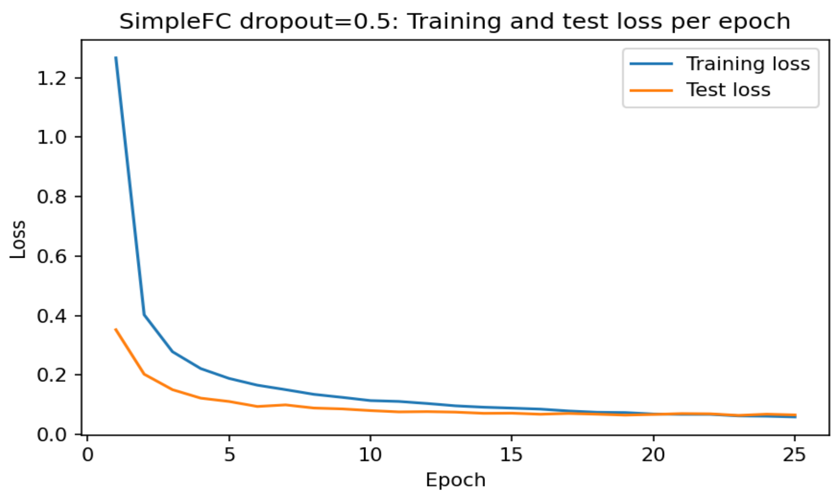
Dropout = 0.0



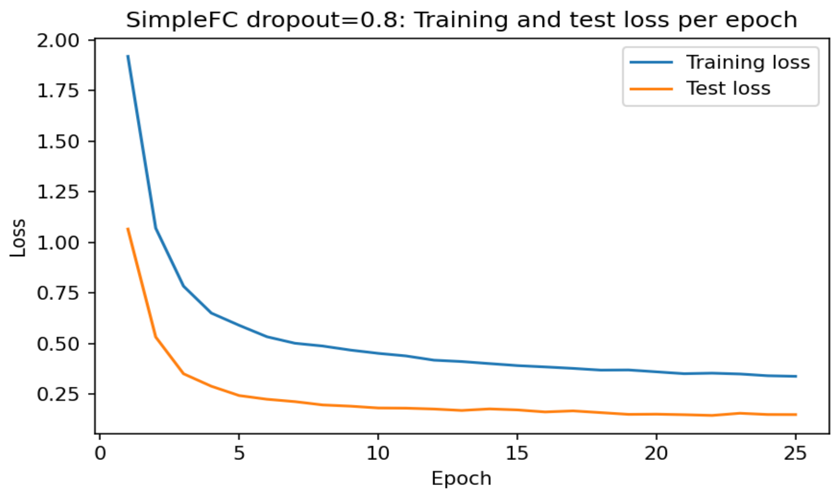
Dropout = 0.2



Dropout = 0.5



Dropout = 0.8



Which dropout probability gives the best results?

The **best** results (no overfitting, i.e., almost equal train and test loss values) typically come from **dropout 0.2 or 0.5**. In those plots, the training and test loss curves are close together and both decrease to a similar level, so the model generalizes well without memorizing the training set.

Which dropout probability gives the worst results?

The **worst** results come from two extremes:

• **Dropout 0.0**: Strong overfitting. Training loss drops to nearly zero while test loss plateaus or increases; the gap between the two curves is large. The model memorizes the training data and does not generalize well.

• **Dropout 0.8**: Underfitting. Both training and test loss stay relatively high because too many neurons are dropped each epoch; the model cannot learn the task well.

Explanation: Dropout randomly turns off neurons during training, so the model cannot rely on any single neuron and is encouraged to learn more robust features. With no dropout (0.0), the model overfits. With moderate dropout (0.2 or 0.5), train and test loss stay close and performance is best. With very high dropout (0.8), the model is over-regularized and underfits.

Question 3: Effect of Normalization with Dropout

Compare the SimpleFC model with dropout 0.2 on unnormalized vs normalized MNIST data. Report training accuracy, testing accuracy, total training time, and the first epoch reaching 96% training accuracy.

Answer:

The comparison between dropout 0.2 with and without normalization is shown in Table 2 below:

Dropout	Training accuracy [%]	Testing accuracy [%]	Total time for training [s]	First epoch reaching 96% train acc
0.2	99.43	98.30	196.49	5
0.2 + norm	99.63	98.30	283.47	3

Normalization: For the row '0.2 + norm', both training and testing datasets use Compose(ToTensor(), Normalize(mean=0.1307, std=0.3081)) (MNIST mean and std). This standardizes inputs to roughly zero mean and unit variance.

Compare and contrast — normalized vs unnormalized:

• **Convergence speed:** With normalization (0.2 + norm), the model reaches 96% training accuracy by **epoch 3**, versus **epoch 5** without normalization (0.2). So normalization helps the model learn faster; inputs in a consistent scale make optimization easier.

• **Final accuracy:** Both setups reach about the same **test accuracy (98.30%)**. Final training accuracy is slightly higher with norm (99.63% vs 99.43%), but the main gain is faster convergence.

• **Training time:** Total training time is higher with normalization (283.47 s vs 196.49 s) because the Normalize transform adds a bit of work per batch; the benefit of normalization is fewer epochs needed to reach a given accuracy, not necessarily lower wall-clock time per run.

• **Explanation:** Normalization (here, standardization with mean 0.1307 and std 0.3081) puts inputs in a range that networks handle well, so gradients and updates are better behaved. That typically speeds up convergence (e.g., 96% train acc in 3 epochs instead of 5) and can improve generalization; in this experiment test accuracy is the same, but the normalized run gets there earlier in terms of epochs.

Problem 3: Convolutional Neural Networks for MNIST

Question 1: SimpleCNN Model Analysis

Train the SimpleCNN model (with 2 convolutional layers) for 25 epochs on the normalized MNIST dataset. Report the model complexity metrics in a table.

Answer:

The SimpleCNN model was trained for 25 epochs. The model complexity metrics are shown in Table 2 below:

Model name	MACs	FLOPs	# parameters	torchsummary size [KB]	Saved model size [KB]
SimpleCNN	3,869,824	7,739,648	50,186	550	199.60

Note: The saved model size is smaller than the torchsummary size because torchsummary estimates the in-memory size required for all parameters (and sometimes activations), while the saved model only contains the raw parameter values stored efficiently in a binary file.

Question 2: Create your own efficient CNN (MyCNN)

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 (smaller number of parameters and/or smaller model size) compared to SimpleCNN. Plot the loss and accuracy curves and extend the table with your model's results.

Answer:

A more efficient CNN model (MyCNN) was created by reducing the number of channels in each convolutional layer (16 and 32 instead of 32 and 64). This resulted in a model with significantly fewer parameters while maintaining good accuracy. The comparison is shown in Table 3 below:

Model name	MACs	FLOPs	# parameters	torchsummary size [KB]	Saved model size [KB]
SimpleCNN	3,869,824	7,739,648	50,186	550	199.60
MyCNN	1,031,744	2,063,488	20,490	260	83.24

Efficiency Improvements:

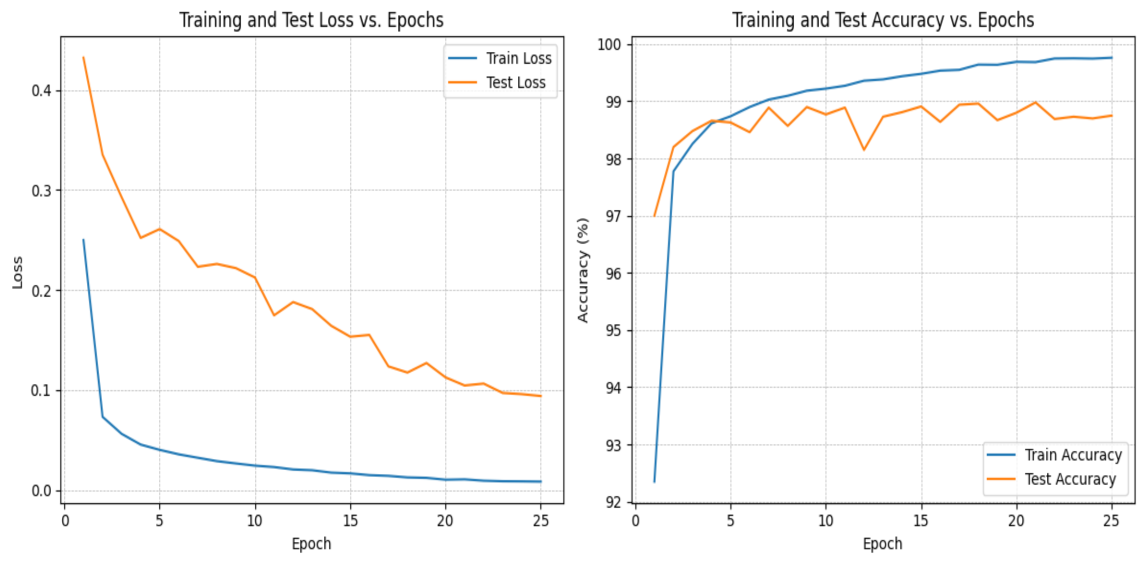
• MyCNN uses approximately **59% fewer parameters** (20,490 vs 50,186)

• MyCNN requires approximately **73% fewer MACs** (1,031,744 vs 3,869,824)

• MyCNN has approximately **58% smaller saved model size** (83.24 KB vs 199.60 KB)

Loss and Accuracy Curves:

The training and test loss/accuracy curves for MyCNN are shown below:



Despite having significantly fewer parameters, MyCNN achieves comparable accuracy to SimpleCNN, demonstrating that careful architecture design can lead to more efficient models without sacrificing performance.

Question 3 (Bonus): Why is MyCNN better than SimpleCNN?

Answer:

The model architecture for MyCNN is exactly the same as the SimpleCNN except that we have reduced the number of channels in each convolutional layer by a factor of 2. This results in a significant reduction in MACs, FLOPs, number of parameters, and model size, while still maintaining a reasonable accuracy on the MNIST dataset.

The memory utilized for storing the model is also reduced. The accuracy and loss difference between SimpleCNN and MyCNN is negligible. This demonstrates that we can design efficient models with fewer resources while still achieving good performance.

Since performance between the two models is basically the same and MyCNN is more efficient, **MyCNN is a better choice for deployment in resource-constrained environments**. Another reason why MyCNN would be better is because it decreases the likelihood of overfitting due to having fewer parameters to learn.

Key Advantages of MyCNN:

• Reduced computational cost (73% fewer MACs)

• Smaller model size (58% reduction) - ideal for edge devices

• Lower risk of overfitting due to fewer parameters

• Comparable accuracy to SimpleCNN

• Better suited for resource-constrained deployment scenarios