Mason Weiss
ELEC 576 - Introduction to Deep Learning
Prof. Ankit Patel, Rice University
Due: October 6, 2024 [11am]
Assignment 2

# Part 1: Visualizing a CNN with CIFAR10

## a) (Importing the) CIFAR10 Dataset

The code for part 1 of this assignment can be found in `cifar10_CNN.py`.

Images were loaded in using the `tensorflow.io.read_image` module, via iterating through all 100 test images in each of the 10 classes, and all 1000 training images in each of the 10 classes. The datasets were then passed to the `TorchDataset` constructor, which included the transformed images (processing the images as grayscale with pixel values from 0-1), and the associated label.

The final length of the model was computed, using `padding = 'same'`, which resulted in a final dimension of `7 x 7 x 64 = 3136`.

Before passing the output from the CNN section of the LeNet architecture, the input was flattened to have a second-axis dimension of 1024, while retaining the batch dimension. This enabled seamless transition to the first fully connected layer.

## b) Train LeNet5 on CIFAR10

The model architecture was implemented as prescribed in the assignment statement.

In order to optimize the hyperparameters to reach the requested test accuracy of 55%, I believed that establishing a variable learning rate would enhance the model's performance. I implemented a linear warmup and linear decay pattern, which maintain the same minimum and maximum values. I iterated over the three tunable hyper parameters: the minimum learning rate, maximum learning rate, and epoch to end warmup and begin decay. After some initial testing of the possible bounds on the learning rate, I tested the model's performance with all permutations of the following parameters, when valid (only for `high_lr` $\geq$ `low_lr`).

Table 1: Hyperparameter Values

| high_lr | low_lr | lr_peak_idx |
|---|---|---|
| 0.0050 | 0.0010 | 1 |
| 0.0020 | 0.0008 | 2 |
| 0.0015 | 0.0005 | 3 |
| 0.0010 | 0.0003 | |
| 0.0008 | 0.0002 | |
| 0.0000 | 0.0001 | |

To maintain the structure of the assignment, I did not adjust the number of training epochs, the model architecture, or the transformation applied to the dataset upon input.

The plot below demonstrates the test accuracy. Due to the fact that nearly 100 permutations were tested, a legend is not provided as tensorboard does not provided 100 line styles to plot with.

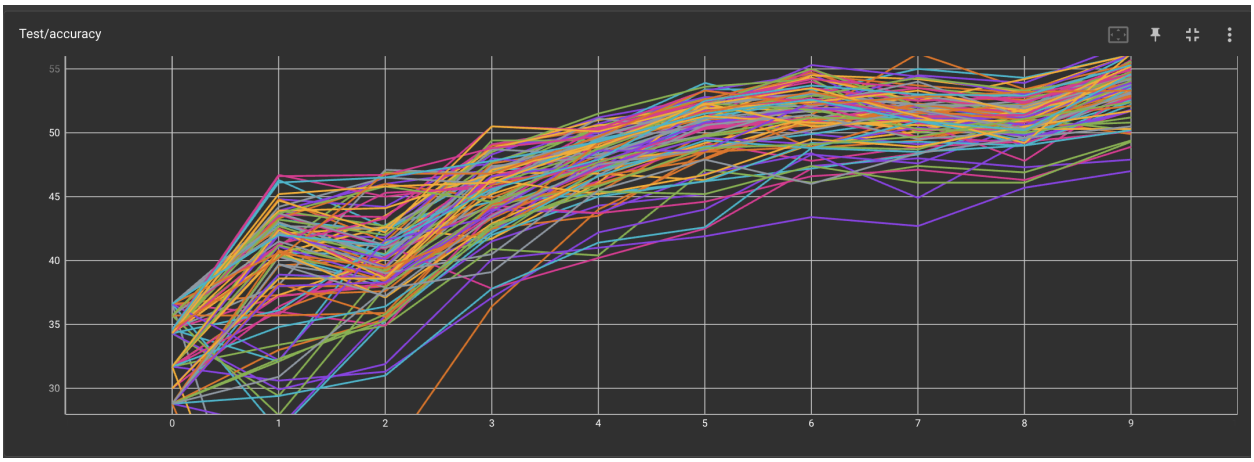



Figure 1: Test Accuracy for various Learning Rates

From this figure, it is clear that only a few models reach the 55% accuracy requirement. The most superior model utilizes the following characteristics:

- `high_lr`: 0.0010
- `low_lr` : 0.0001
- `lr_peak_idx` : 3

This was selected as the particular model for study later in the assignment and for the plots that follow.

The learning rate, as described above, is plotted below.

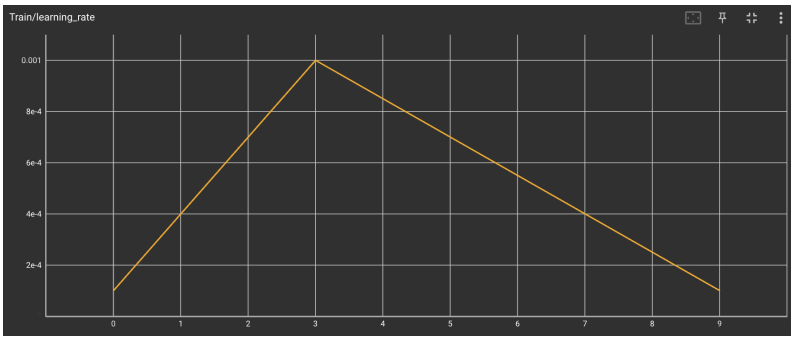



Figure 2: Learning Rate for Selected Model

Per the assignment rubric and code skeleton, accuracy and loss are computed and logged at each epoch for both training and testing routines. This model achieves a final training accuracy of 68.09% and a final test accuracy of **56.10**%.

The training loss (total among all batches) per epoch and accuracy at each epoch are plotted below.

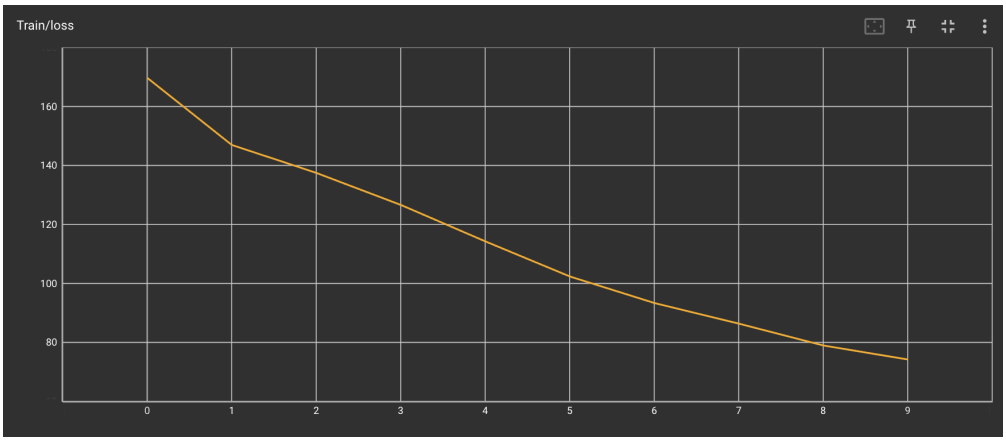



Figure 3: Train Loss

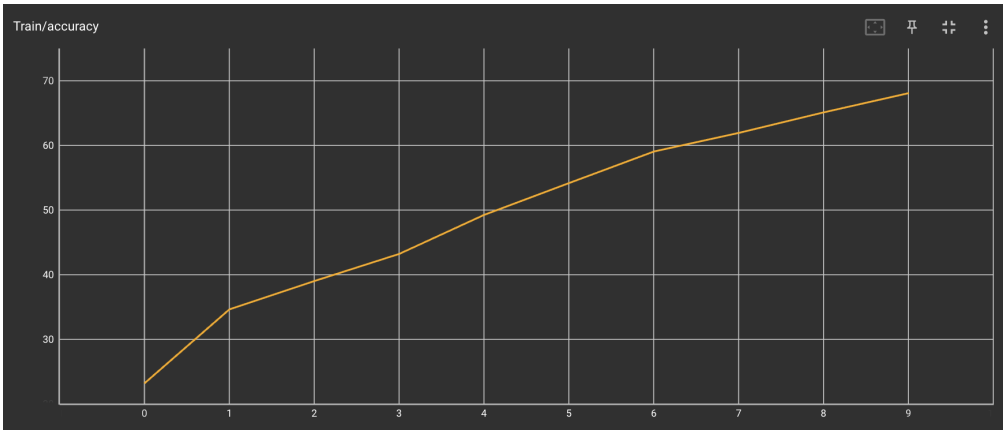



Figure 4: Train Accuracy

From the tensorboard plots, it is visible that the total training loss decreased at each epoch. The training accuracy also steadily increases, with noticeably more gain at the first few epochs than at the later iterations.

The loss and accuracy for the test dataset are plotted below.

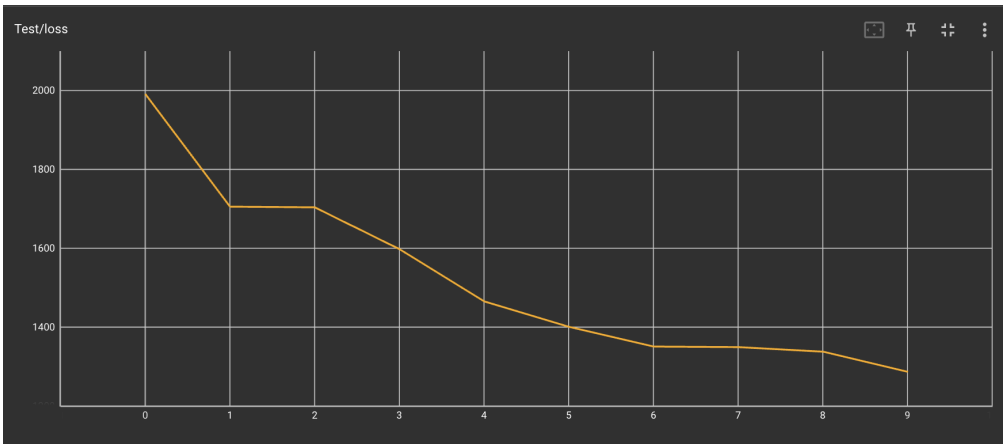



Figure 5: Test Loss

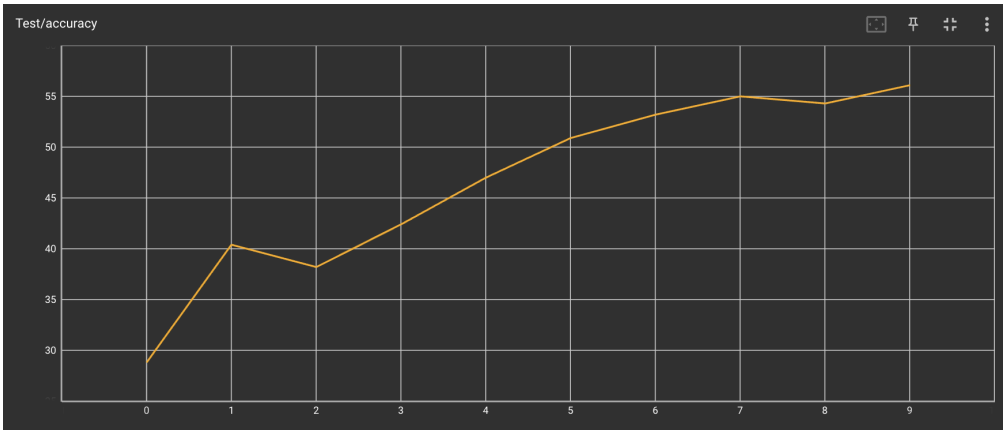



Figure 6: Test Accuracy

The output to console printed by the `cifar10_CNN.py` file, using the 10th epoch (index 9) as an example is:

```
TRAIN - Learning Rate for epoch 10 of 10: 0.0001
TRAIN - Epoch: 10 of 10 | Batch 0 of 79 | Loss: 0.98764
TRAIN - Epoch: 10 of 10 | Batch 10 of 79 | Loss: 1.00712
TRAIN - Epoch: 10 of 10 | Batch 20 of 79 | Loss: 0.9929
TRAIN - Epoch: 10 of 10 | Batch 30 of 79 | Loss: 0.89142
TRAIN - Epoch: 10 of 10 | Batch 40 of 79 | Loss: 0.97658
TRAIN - Epoch: 10 of 10 | Batch 50 of 79 | Loss: 0.81973
TRAIN - Epoch: 10 of 10 | Batch 60 of 79 | Loss: 0.9459
TRAIN - Epoch: 10 of 10 | Batch 70 of 79 | Loss: 0.84379
TRAIN - Epoch: 10 of 10 | Accuracy: 68.09% | Total Loss: 74.1816
TEST - Epoch: 10 of 10 | Accuracy: 56.1 % | Loss: 1286.52145
```

The numeric output is concisely represented by Figures 3-6, which seem to suggest that at further iterations (if the model were to be trained past 10 epochs), the accuracy may only be enhanced slightly. Additionally, the total training loss can be verified, given that the loss in any given batch is approximately 0.95. Over the 79 batches (78 of size 128 and 1 of size 16), this should sum to a total loss near 75, which is expressed in both the second to last output row and Figure 3. As expected, the training accuracy is superior to the test accuracy; however, this difference does not appear to be exceedingly large. Combined with the test loss converging in Figure 5, this appears to suggest that the model is not overfitting to the training sample.

## c) Visualize the Trained Network

To visualize the trained network, I first implemented code to access the weights of the first convolutional network. This network consists of 32 5x5 filters, for which the weights are plotted in grayscale (normalized 0-1) below.

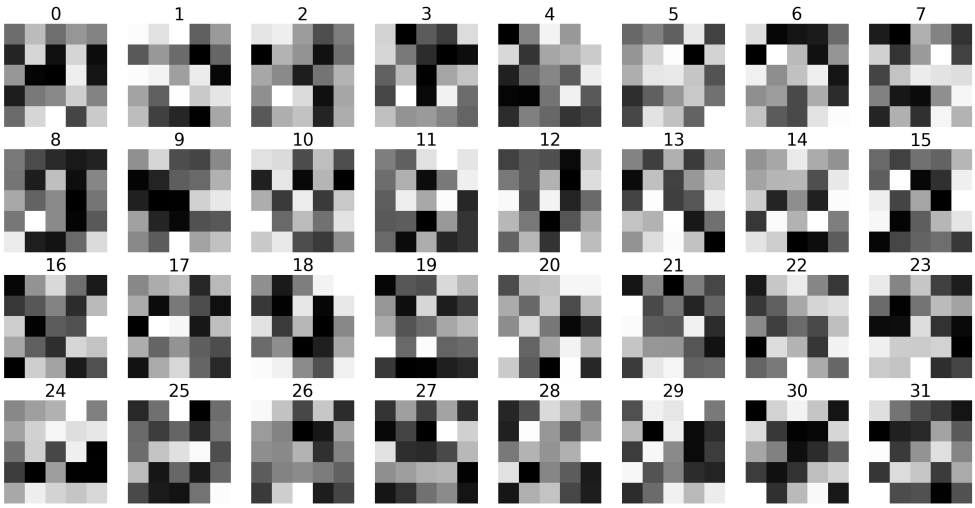



Figure 7: Weights for the 32 Filters of the First Convolutional Layer

These filters appear similar to Figure 1 from the assignment briefing, with the notable exception of being in grayscale rather than color. Each filter appears to be unique, and correspondingly, will prioritize specific patterns within the input 28x28 image.

Additionally, I implemented a brief conditional statement to save the activations from the first and second convolutional layers on the final epoch, by iterating through the model-steps as if it were in a forward pass. The statistics for activations from convolutional layers follow on the next page.

From these statistics (along with the feature maps included in section 2(b)), a notable difference between convolutional layer activations is that deeper in the network, the minimum and maximum activations converge more closely to 0, and the standard deviation decreases.

This behavior demonstrates a known behavior of deep CNNs; later layers typically have a more concentrated distribution of activations.

Table 2: Statistics from Activations of Convolutional Layers

| | `First Conv Layer (idx=0)` | `Second Conv Layer (idx=3)` |
|---|---|---|
| Mean | -0.01273 | 0.04315 |
| Median | -0.02574 | 0.03407 |
| Minimum | -1.19211 | -0.60006 |
| Maximum | 1.48140 | 0.98294 |
| Std. Dev. | 0.28903 | 0.10798 |

# Part 2: Visualizing and Understanding Convolutional Networks

## a) Summary of Zeiler and Fergus Paper

This paper, authored by two researchers affiliated with the Courant Institute at NYU, discusses a novel technique to visualize the inner workings of a Convolutional Neural Network (CNN), and applies these tools to improve a classification model. The central focus of this paper references a model somewhat similar to that which I've constructed in Part 1 of this assignment; a CNN taking in 2D input, applying a series of convolutions, activations, and max-pooling operators, followed by a set of fully-connected layers and a softmax layer to interpret the output of the model as a probability among classes.

The method is described as implementing a "deconvnet", which consists of first selecting a specific convnet activation to study, and then zeroing-out the remaning activations in the convolutional layer. A series of processes: unpooling, rectifying, and filtering, can then be applied. First, the deconv layer approximates an inverse to the maxpool operation by using switches that record the location of the (maximum) value which maxpool selected in the conv layer. After this, the output is passed through a relu activation function and fed through the transpose of the filter which was applied to the corresponding conv layer. Each of these "deconv" layers are assembled sequentially, in the same order as their respective conv layers. This process is navigated from the output of the model to the input, which elucidates the features within some layer on the input pixel space.

The researchers then utilized this information to observe which convolutional layers emphasize particular aspects of the input, with the later layers corresponding more closely to discriminating between the classes of the input. The observations learned from this study were then applied to improve then-state-of-the-art models, such as the highest-performing models from the 2012 ImageNet challenge. With this new visualization ability, it could be much more easily determined what parameters of particular layers (filter size, padding, stride, etc) may be restraining the model's performance. The researchers also demonstrated, through both an occlusion and ablation study, that successful models using the ImageNet architecture are aware of fine-grain image data (as opposed to just simply general trends

across the entire image) and are also dependent on the model's depth. This improved model was then tested on other image classification datasets, such as Caltech-101, Caltech-256, and Pascal-2012. When pre-trained on the ImageNet data, the model's performance exceeds previous accuracy scores on the first two datasets, and performs similarly on the last, which is impressive considering that the subject of each class is represented categorically differently in the pre-trained set compared to the actual training set.

## b) Visualization of Convolutional Layer Activations

Using the output activations obtained from the implementation described in section 1(c), the feature maps can be plotted for some example inputs. These feature maps can be plotted using part of the methodology discussed in the Zeiler and Fergus paper.

We can first observe the activations from the first convolutional layer for inputs with particularly strong feature maps. I selected three cases, test inputs with indices 60, 64, and 80.

### Test Item 30: Class 0 (Airplane)

This test item was selected due to its prominent features being visible in the feature map associated with the activations from layer 1. For the first convolutional layer, 32 maps are available (of size 28 by 28) (one for each filter). For the second layer, 64 maps are available (of size 14 by 14). The model predicted correctly that this object was of class 0, which is of an airplane.

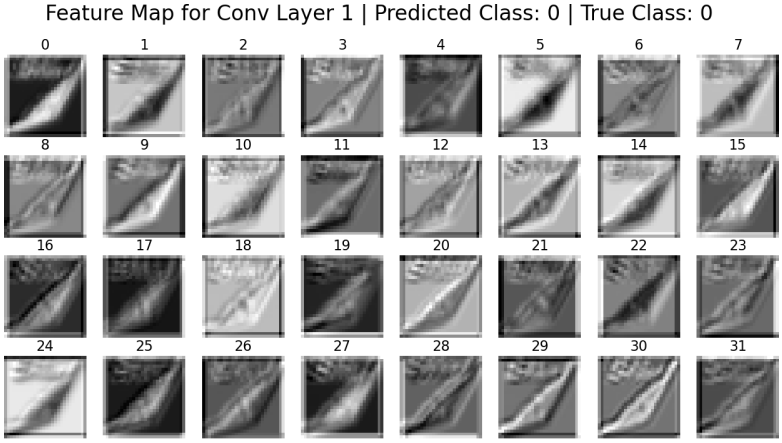



Figure 8: Feature Map for Convolutional Layer 1

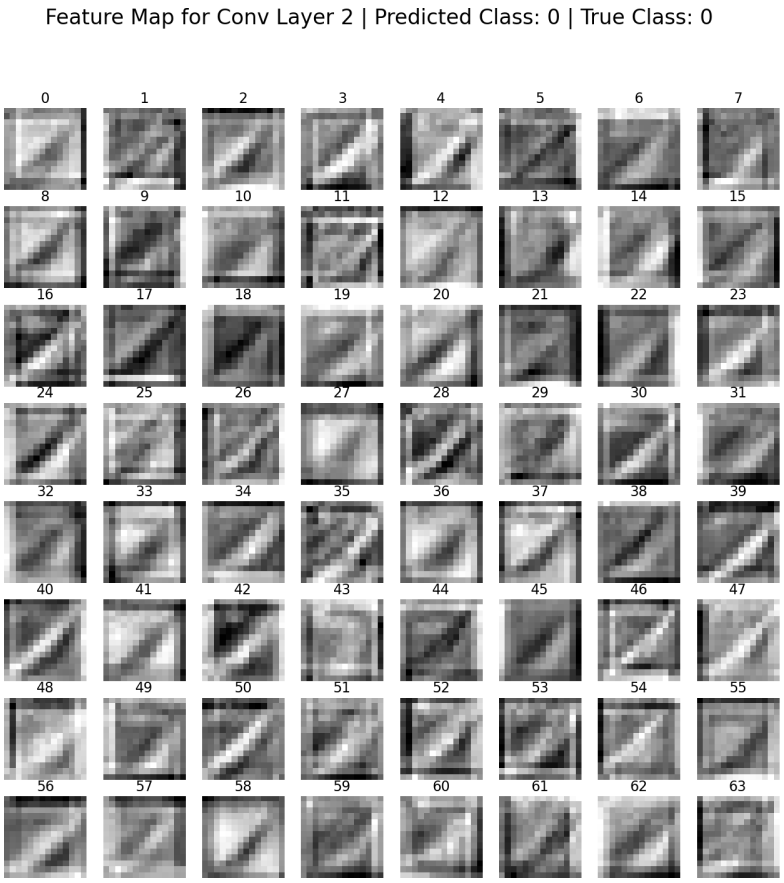



Figure 9: Feature Map for Convolutional Layer 2

The feature maps for the first convolutional layer demonstrate the fine-grained attributes of the image relatively well. In comparison, the feature maps for the second layer appear more distorted, but still demonstrate similar data to the first map. To continue the analysis similarly to the Zeiler and Fergus paper, a deconv net approach could be used to project the activation from the second convolutional network down into the input space.

### Test Item 40: Class 8 (Ship)

This test item was also selected due to the prominent features demonstrated. The background of the ship appears distinct from the ship itself in most of the feature maps. It is also worth noting that likely due to the padding scheme used in the filters of the convolutional layer, the feature maps retain an artifact which manifests as a border around the object, in activations from both the first and second activations.

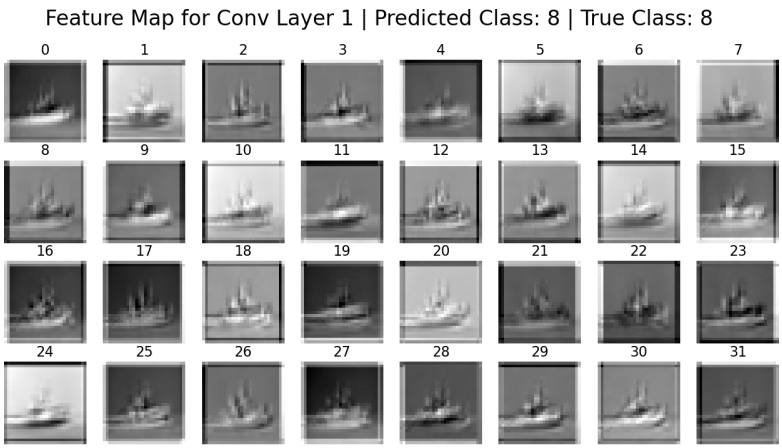



Figure 10: Feature Map for Convolutional Layer 1

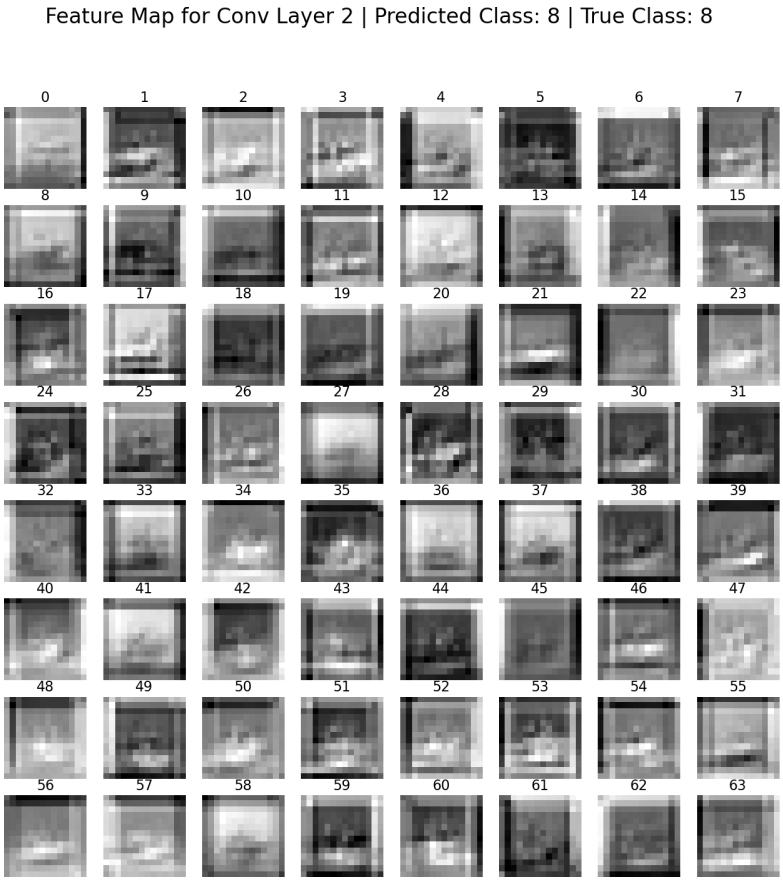



Figure 11: Feature Map for Convolutional Layer 2

**Test Item 65: Class 1 (Automobile)**

In this test case, the outline of the automobile is visible in the feature maps from both layers. In particular, the first convolutional layer is aware of distinct patterns within the input image, such as the windows and trim features of the car represented in a different color than the body.

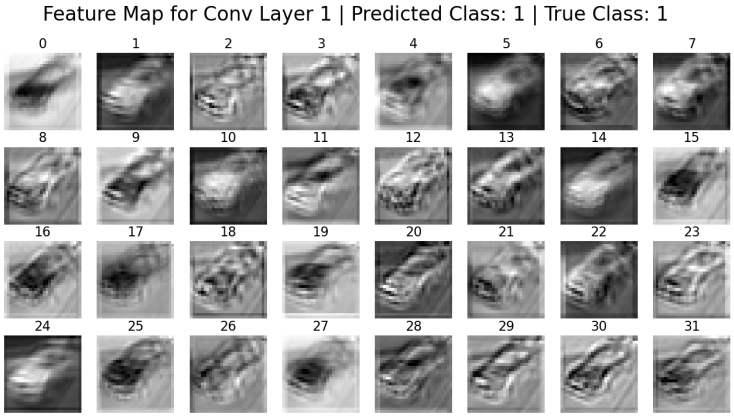



Figure 12: Feature Map for Convolutional Layer 1

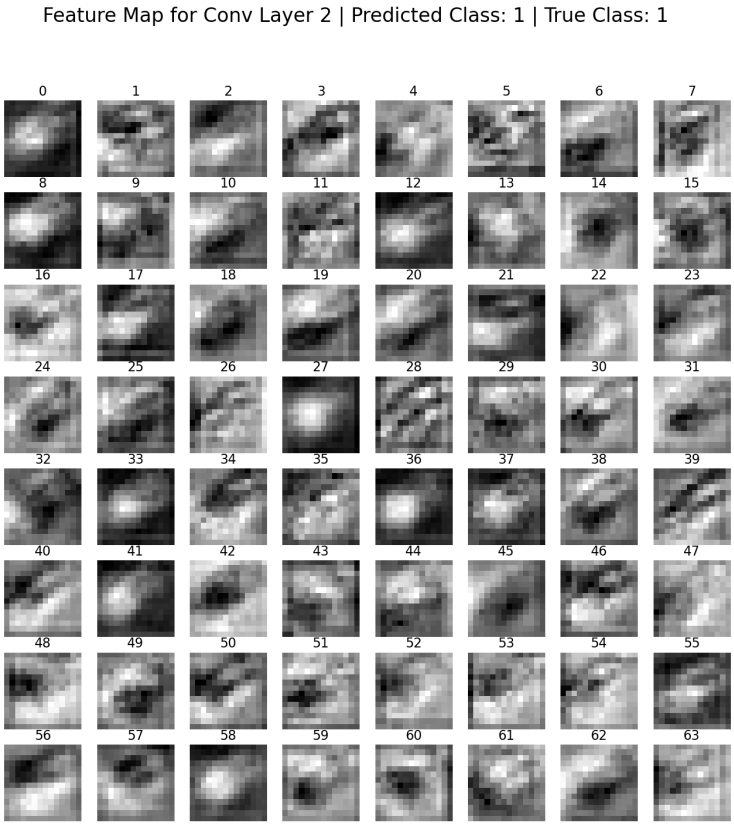
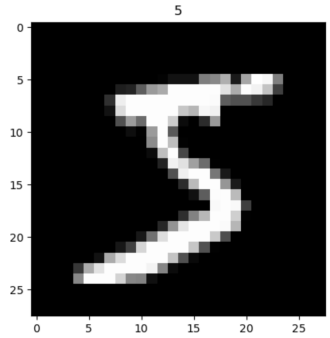



Figure 13: Feature Map for Convolutional Layer 2

# Part 3: Build and Train an RNN on MNIST

## a) Setup an RNN

The RNN was developed using the code skeleton; source code for my implementation is available in `mnistRNN.py`. The first entry in the shuffled train dataset using the specified seed is:




After developing and testing multiple Recurrant Neural Network architectures, with different numbers of hidden features, RNN layers, and activation functions, I settled on the following model design:

`nn.RNN(input_size=28, hidden_size=128, num_layers=2, batch_first=True)`
`nn.Linear(in_features=128, out_features=10)`

After some experimentation, I settled on a learning rate of 0.001, which was significantly smaller than the learning rate listed in the code skeleton. This recurrent architecture was selected to have a large number of hidden features; a bi-layer architecture was established, which effectively stacks two RNNs of the same architecture on top of each other. These model designs appeared to be preferred using the MNIST dataset. The plot of the training loss at each batch (of 128) is attached below.

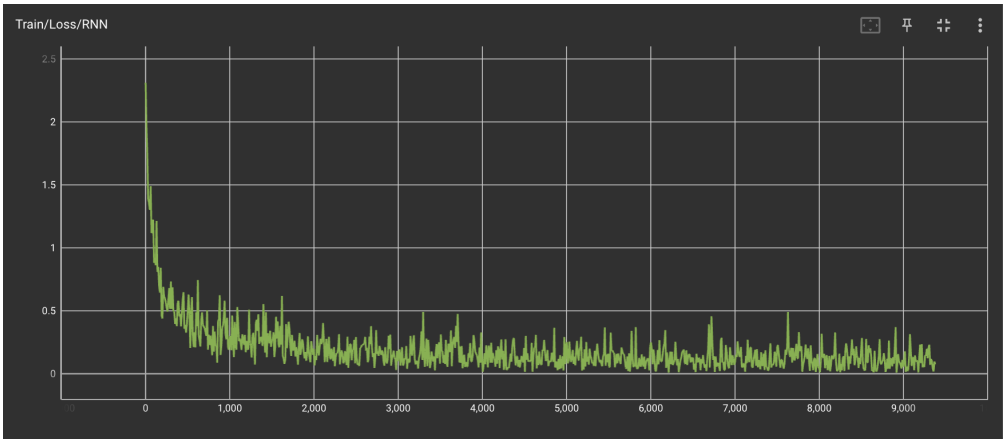



Figure 14: Training Loss for RNN with nhidden = 128

The training loss appears to decrease with each epoch, which verifies that the learning rate is not exceedingly high, and that the RNN is converging to an accurate model. The test loss and accuracy follow.

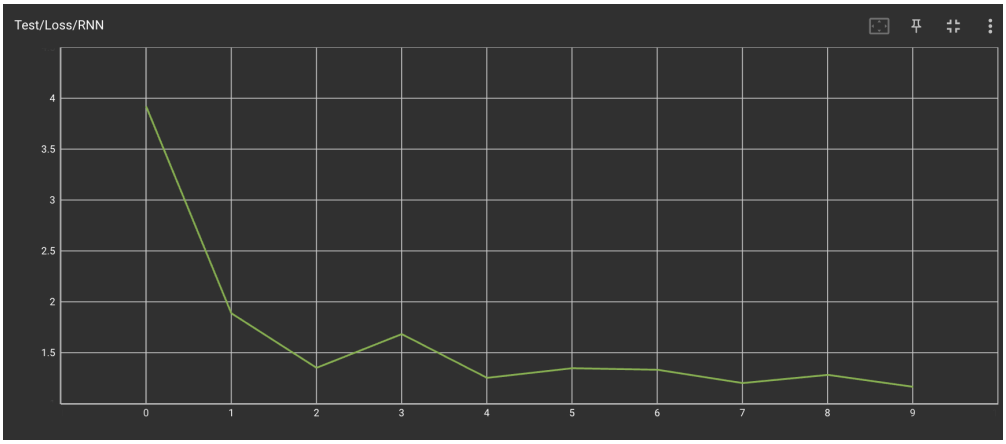



Figure 15: Test Loss for RNN with nhidden = 128

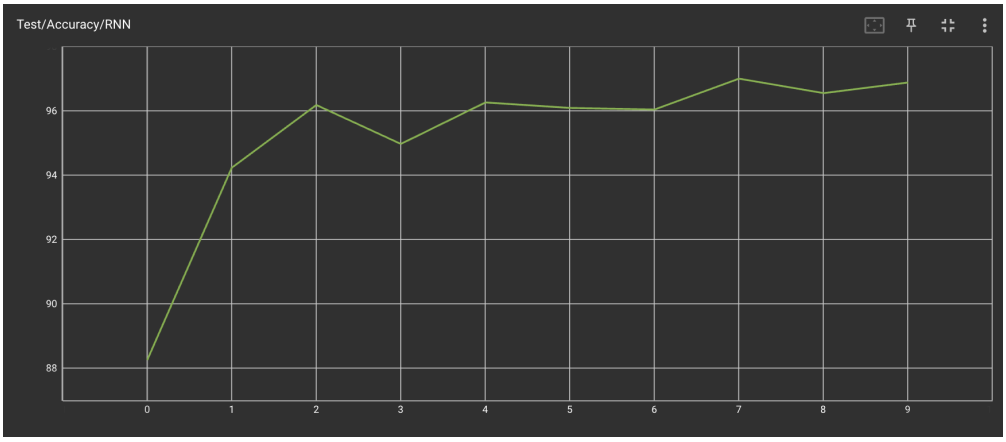



Figure 16: Test Accuracy for RNN with nhidden = 128

With this configuration, the model's test accuracy is 96.88 %.

## b) Using an LSTM or GRU

I implemented the same model architecture (128 hidden features) with both a Long Short-Term Memory and Gated Recurrent Unit with the same parameters (two layers and 128 hidden features). The training loss, test loss, and test accuracy are below for both models.

### LSTM Model

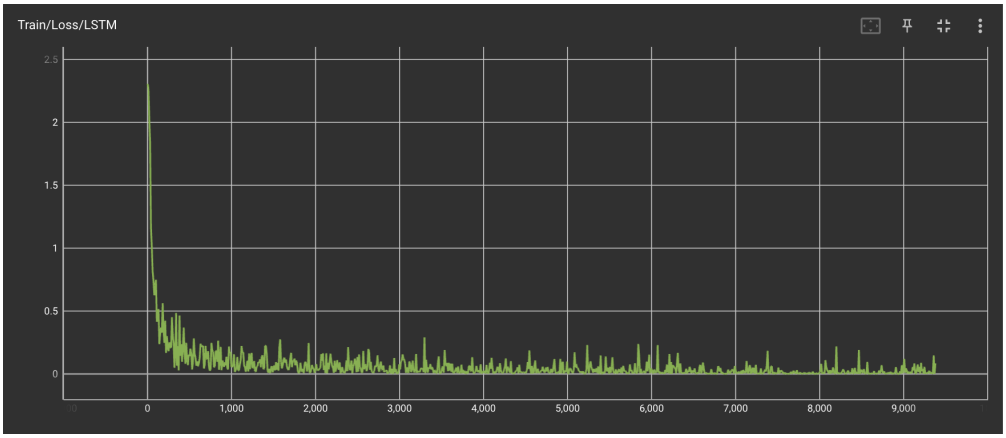



Figure 17: Training Loss for LSTM with nhidden = 128

Similarly to the RNN Model, the loss from the LSTM model converges rapidly.

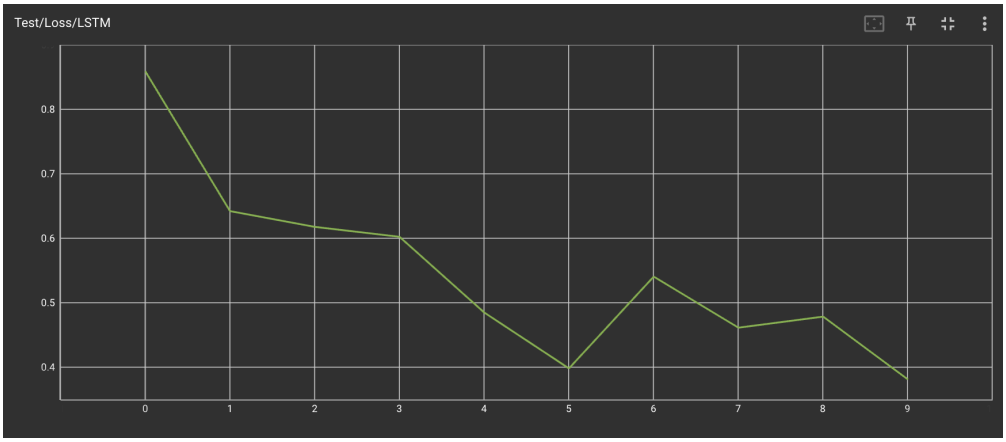



Figure 18: Test Loss for LSTM with nhidden = 128

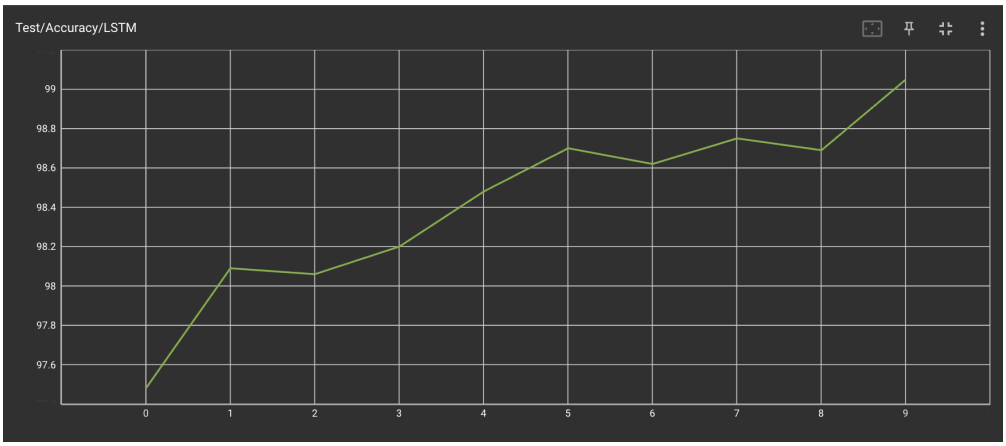



Figure 19: Test Accuracy for LSTM with nhidden = 128

A notable difference between the LSTM and the RNN is that the LSTM appears to have a much higher initial test accuracy following the first epoch, meaning that it trains more rapidly. The test loss curve supports this statement as well; since the loss is computed via the same metric, the axes for Figures 15 and 18 can be compared to deduce the LSTM's superiority.

### GRU Model

The Gated Recurrent Unit appears to have similar performance to the LSTM, most likely owing to these model's ability to "remember" certain aspects of inputs preceding the most recent one.

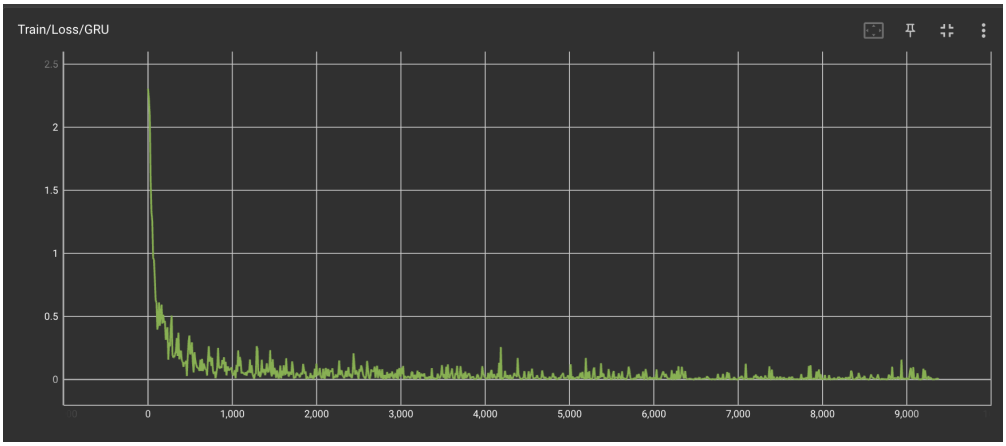



Figure 20: Training Loss for GRU with nhidden = 128

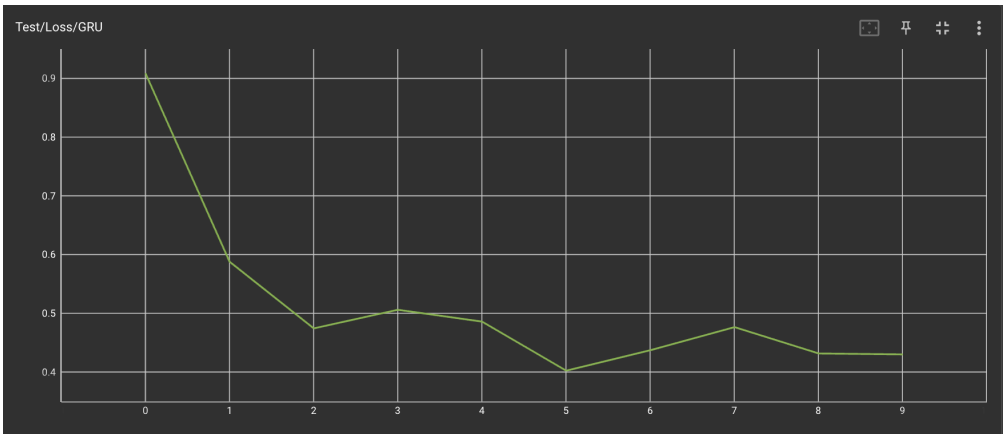



Figure 21: Test Loss for GRU with nhidden = 128

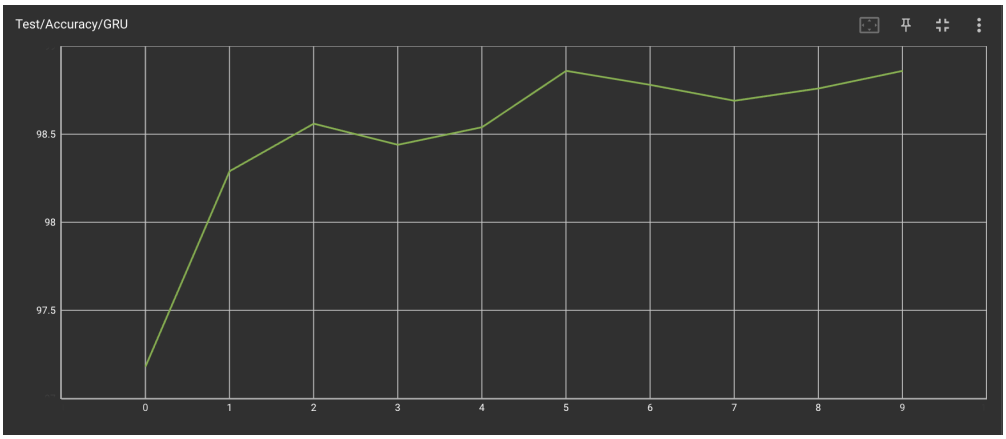



Figure 22: Test Accuracy for GRU with nhidden = 128

The accuracy from the three models, with 128 hidden features, are listed below.

**RNN:** 96.88 %
**LSTM:** 99.05 %
**GRU:** 98.86 %

As discussed in lecture, the LSTM likely performs best due to its ability to store retain more long-term patterns. Because the input of the 28x28 MNIST images are read in as series of length-28 vectors, being able to retain information from rows above and below the input to that component of the RNN is likely essential for the model to properly regress.

## Models with Varying Number of Hidden Features

I found that within the range of hidden feature values that I tested, a higher number was generally preferred. The results from the superior model with 128 hidden features can be compared to the result from the model with 64 hidden features.

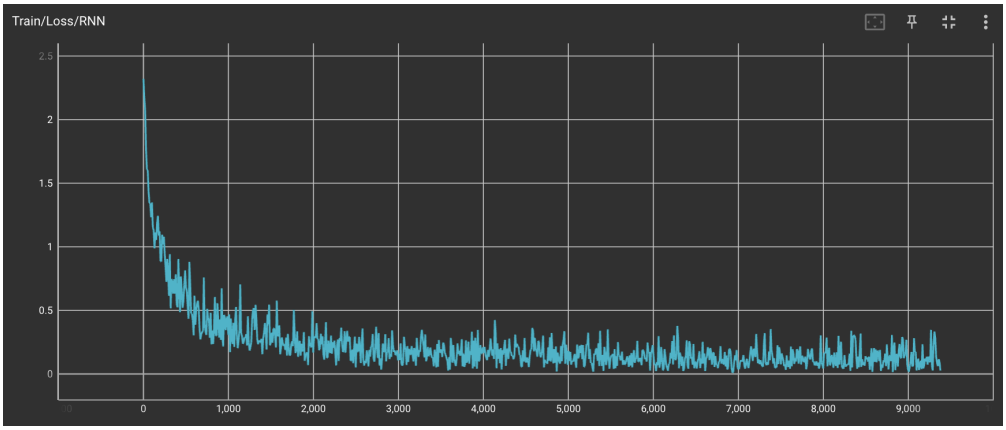



Figure 23: Training Loss for RNN with nhidden = 64

This loss plot appears quite similar to Figure 14, which demonstrates the RNN training loss with twice as many hidden features.

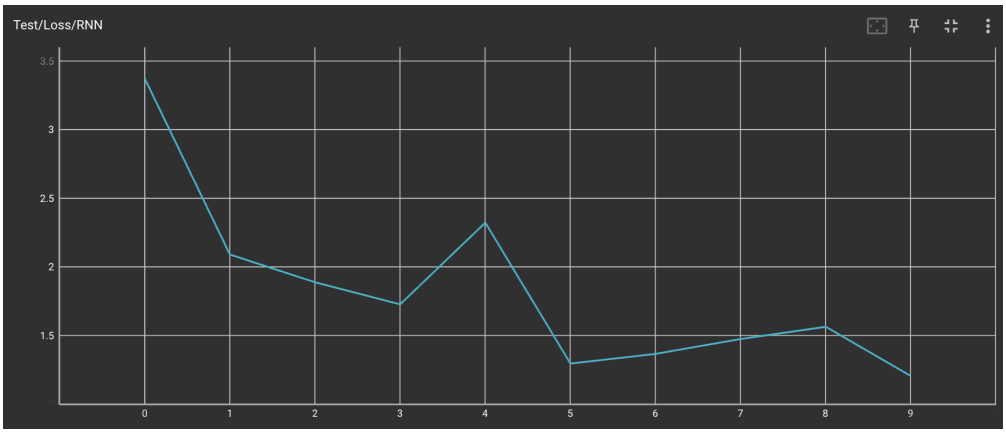



Figure 24: Test Loss for RNN with nhidden = 64

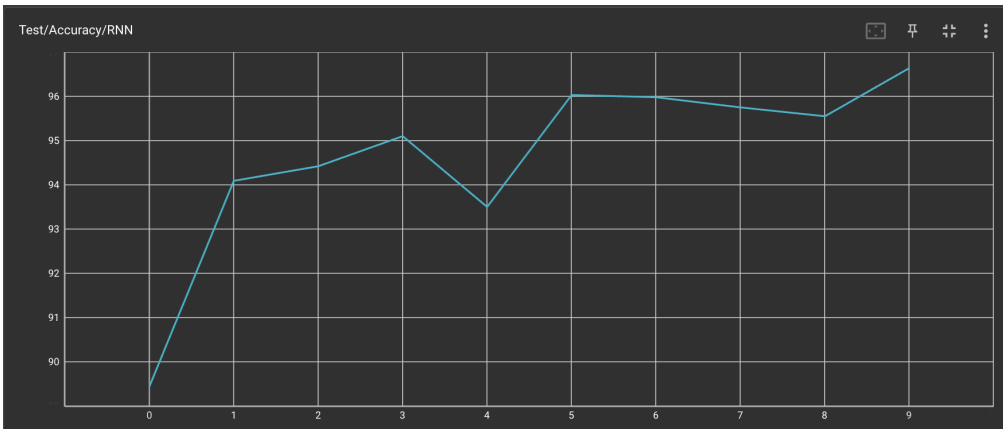



Figure 25: Test Accuracy for RNN with nhidden = 64

A subtle difference between the test accuracy and loss for 64 hidden features vs 128 hidden features is that for the shorter RNN, the accuracy and loss take a touch longer to converge, and seem to not reach as impressive of a performance.

## b) Using an LSTM or GRU

Similarly, the results for the LSTM and GRU can be compared with 64 hidden units.

### LSTM Model

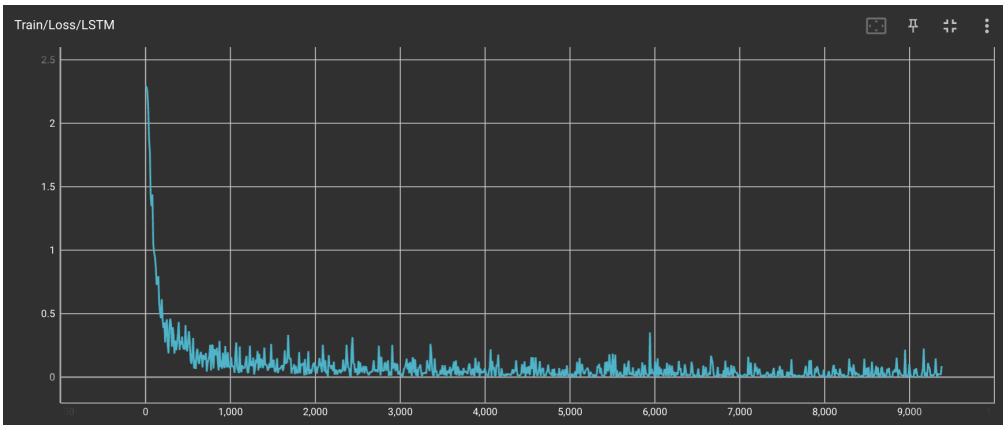



Figure 26: Training Loss for LSTM with nhidden = 64

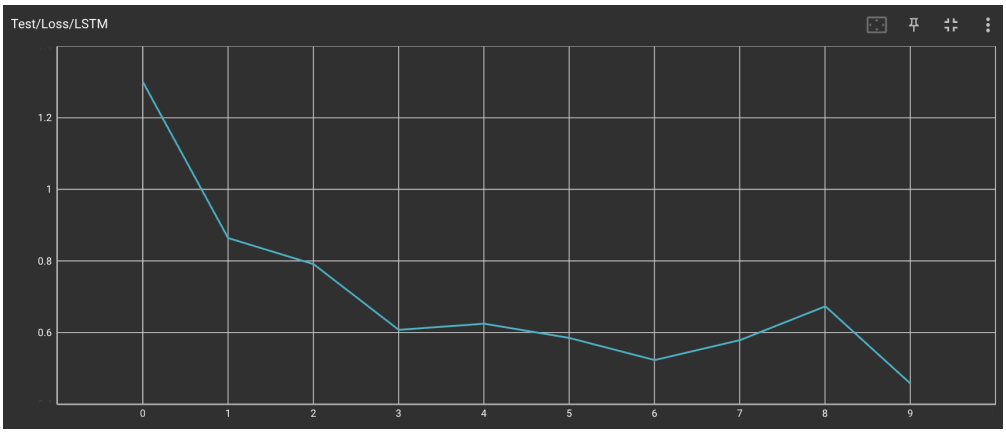



Figure 27: Test Loss for LSTM with nhidden = 64

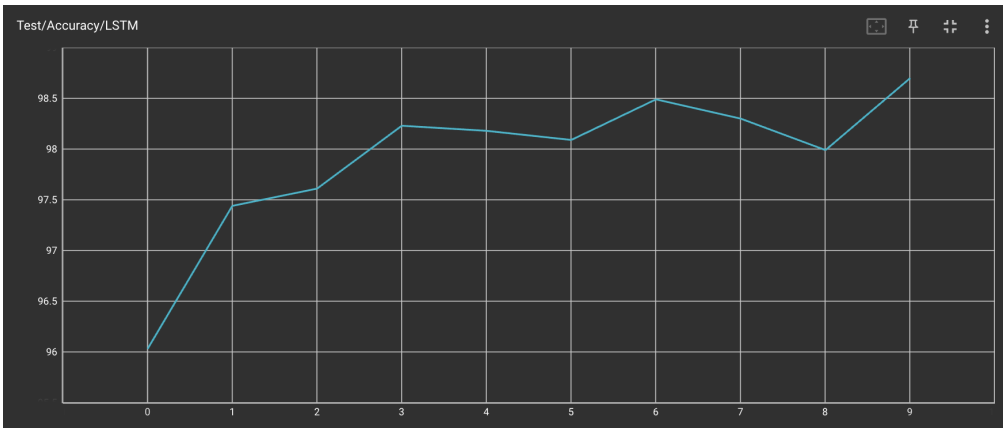



Figure 28: Test Accuracy for LSTM with nhidden = 64

## GRU Model

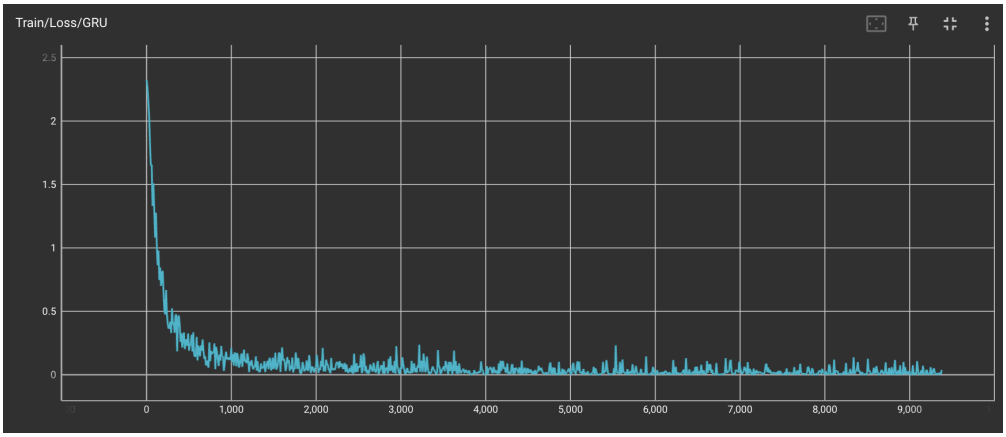



Figure 29: Training Loss for GRU with nhidden = 64

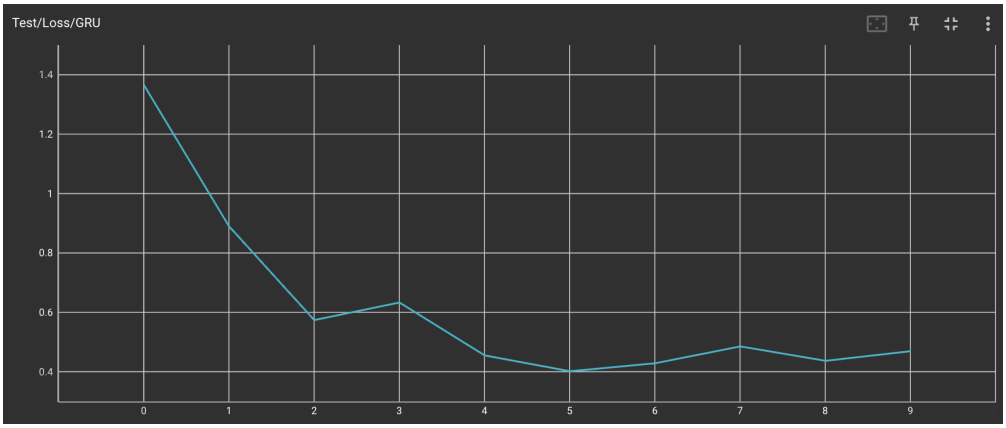



Figure 30: Test Loss for GRU with nhidden = 64

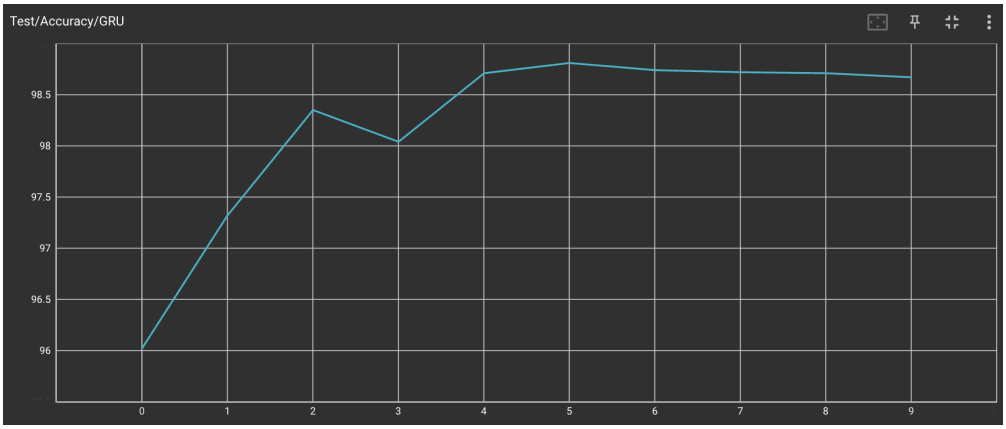



Figure 31: Test Accuracy for GRU with nhidden = 64

The accuracy from the three models, with 64 hidden features, are listed below.

**RNN:** 96.64 %
**LSTM:** 98.70 %
**GRU:** 98.67 %

These statistics support the previous claim that the number of feature units has an impact on all implementations of recurrent networks; all three models observe slightly enhanced accuracy when constructed with a longer model.

## c) Compare against the CNN

In my submission for Assignment 1, I compared multiple different model architectures. In this report, I discovered that "the best-performing CNN models used ReLU or leaky-ReLU; in particular, leaky-relu0.01/kaiming_uniform/sgd_0.5 was the best performing model, reaching a final accuracy of 98.52%." In particular, the learning rate of 0.01 was preferred over the learning rate of 0.001. The accuracy for a selection of CNN models studied in my first assignment are below.

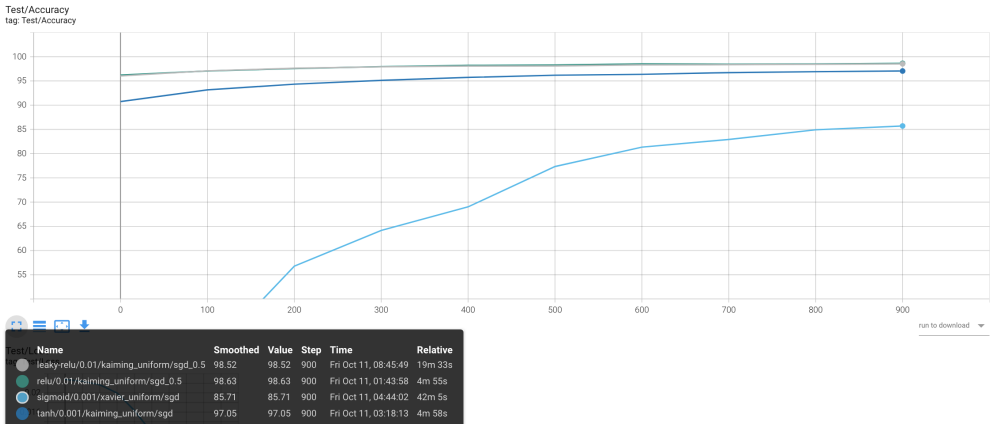



Figure 32: Test Accuracy for Various CNN Models

The silver line, which demonstrates the aforementioned model with a learning rate of 0.01, observes high test accuracy over all batches in the MNIST dataset when trained over 10 epochs.

In comparison to the RNN, the final accuracy metrics are slightly inferior to that of the CNN; the LSTM and GNU perform comparably. These metrics support the claim that while RNNs can be useful tools for developing comprehensive NN implementations, convolutional networks may often be preferred due to their speed and comparable (if not, superior) performance. It is also particualrly relevant that the CNN accuracy saturates much more quickly, demonstrating that for comparable performance, the RNN must be trained (in this test case) for at least 5-10 epochs, whereas the CNN demonstrates impressive accuracy after just one or two iterations.

**Note:** In addition to the mw103-assignment2.zip file submitted to canvas, the code has been pushed to the working repository submitted with my previous assignment.

**Repository Link:** https://github.com/masonweiss/ELEC576

# References

[1] Zeiler, M., & Fergus, R. (2013). Visualizing and understanding convolutional networks. *Computing Research Repository, November 2013*. http://arxiv.org/abs/1311.2901