# CS 6375 Machine Learning Project 3 Report: Deep Learning for MNIST and CIFAR-10

Prepared By:
Benjamin Walmer, bjw220002

## Introduction:

To begin this project, I imported the MNIST and CIFAR10 datasets using PyTorch's torchvision library. Next I did preprocessing of the data, which consisted of normalization and other steps documented in my code.

## Validation Splits:

I used the random split function from PyTorch to randomly split each dataset into training and validation sets. Using a random seed of 73, I split the MNIST dataset into 50,000 training samples, and 10,000 validation samples. Similarly, I split the CIFAR-10 dataset into 45,000 training samples and 5,000 validation samples. This is clearly documented in the load_datasets() function in my code.

## Part 1: Multilayer Perceptron Architectures:

For the shallow architecture, I defined a MLP class consisting of only 1 hidden layer of 128 units. For the medium architecture, I used 3 hidden layers of 128, 256, and 512 units each. And for the deep architecture, I used 5 hidden layers of 64, 128, 256, 512, and 1024 units respectively. These values are all defined clearly in the MLP_ARCHITECTURES list in my code. In the MLP() class in my code, it is clear that the ReLU activation function is used to introduce non-linearity into the model.

## Part 2: Convolutional Neural Network Architectures:

For the baseline CNN, I used 2 convolutional layers alongside maxpooling (no dropout). Across all layers and models, I kept kernel size = 3, padding = 1 for each filter for easy comparison. For the enhanced CNN, I added 2 rounds of batch normalization between each convolutional layer, along with dropout at the end of the model. For the deeper CNN, my model has 3 total convolutional layers, with batch normalization between each layer and maxpooling and dropout at the end. As for the MLP models, I used ReLU for all activations. You can see all of this defined in my code in the BaselineCNN(), EnhancedCNN(), and DeeperCNN() classes respectively.

## Hyperparameter Tuning:

Below is a table of hyperparameters and values tested for each one that I used for each MLP & CNN model:

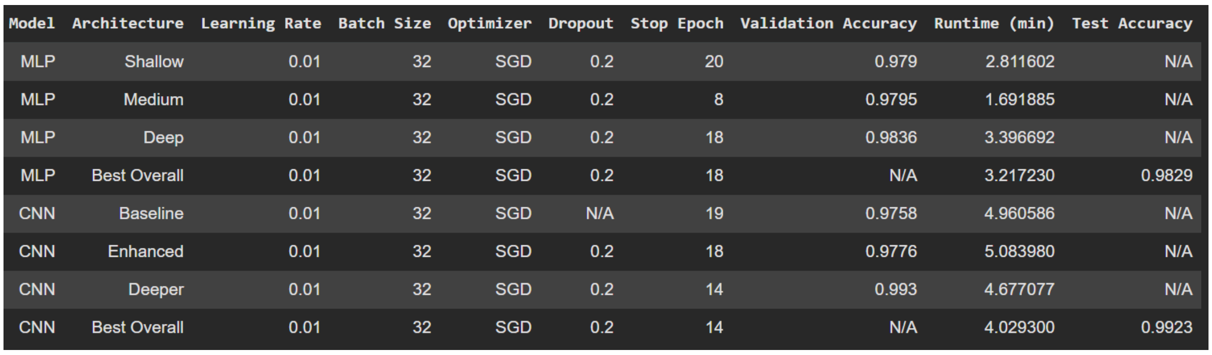
| | | | |
|---|---|---|---|
| Learning Rate | 0.0001, 0.001, 0.01 | Dropout Rate | 0.2, 0.5 |
| Batch Size | 32, 64, 128 | Optimizer | SGD, Adam |

Figure 1: Hyperparameters Tested for MLP Models

For the hyperparameter tuning, I used the random.sample() method to get 10 random combinations instead of exhaustively training all 3 * 3 * 2 * 2 = 36 hyperparameter combinations. You can see this in my code under the run_experiments() function where I test the sampled_params variable and I have previously set param_samples = 10 near the beginning of my code. In the train_model() function, I chose 20 epochs as the max number of epochs for each model to train to control runtime. I also used early stopping to help runtime, so if the validation accuracy doesn't improve for 3 consecutive epochs, the model stops training.

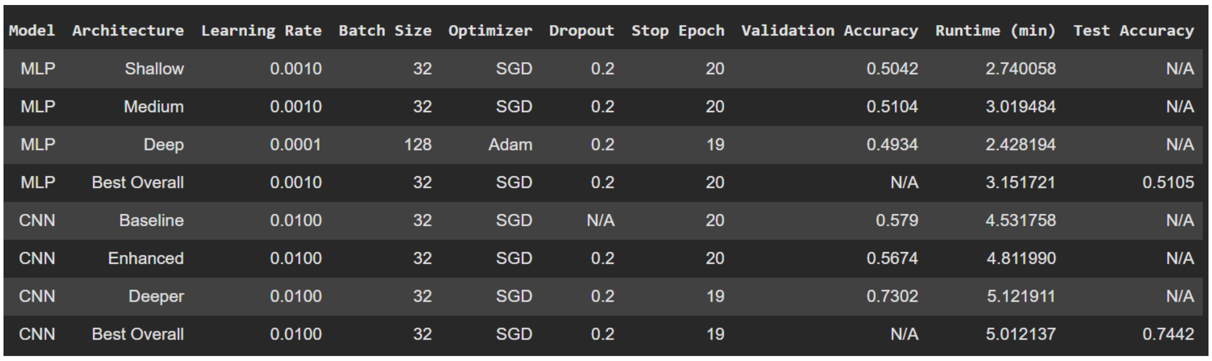
<u>Model Selection & Test Accuracy:</u>

After getting the validation accuracy for the different hyperparameter combinations, I got the hyperparameter values for the model with the highest validation accuracy (for each model & dataset, so 4 models total). I then retrained the model, using the same parameters, on the combined training & validation sets, and then evaluated its accuracy on the unseen test dataset. You can see this in my code in the retrain_and_test() function. However, because I was using the same model training function as before,  I noticed that my code was using the test dataset for validation accuracy to tune the model, which is data leakage because my model is seeing the test dataset during training. My solution for this problem was to only train the combined model for the same number of epochs that the original model stopped at. You can see this in my code in the train_model() function with the fixed_epochs parameter. Using the evaluate_model function in my code, I got the test accuracy for each of the 4 best models and outputted it into the final results tables shown below.



| Model | Architecture | Learning Rate | Batch Size | Optimizer | Dropout | Stop Epoch | Validation Accuracy | Runtime (min) | Test Accuracy |
|---|---|---|---|---|---|---|---|---|---|
| MLP | Shallow | 0.01 | 32 | SGD | 0.2 | 20 | 0.979 | 2.811602 | N/A |
| MLP | Medium | 0.01 | 32 | SGD | 0.2 | 8 | 0.9795 | 1.691885 | N/A |
| MLP | Deep | 0.01 | 32 | SGD | 0.2 | 18 | 0.9836 | 3.396692 | N/A |
| MLP | Best Overall | 0.01 | 32 | SGD | 0.2 | 18 | N/A | 3.217230 | 0.9829 |
| CNN | Baseline | 0.01 | 32 | SGD | N/A | 19 | 0.9758 | 4.960586 | N/A |
| CNN | Enhanced | 0.01 | 32 | SGD | 0.2 | 18 | 0.9776 | 5.083980 | N/A |
| CNN | Deeper | 0.01 | 32 | SGD | 0.2 | 14 | 0.993 | 4.677077 | N/A |
| CNN | Best Overall | 0.01 | 32 | SGD | 0.2 | 14 | N/A | 4.029300 | 0.9923 |

<u>Figure 2: MNIST Final Results</u>



| Model | Architecture | Learning Rate | Batch Size | Optimizer | Dropout | Stop Epoch | Validation Accuracy | Runtime (min) | Test Accuracy |
|---|---|---|---|---|---|---|---|---|---|
| MLP | Shallow | 0.0010 | 32 | SGD | 0.2 | 20 | 0.5042 | 2.740058 | N/A |
| MLP | Medium | 0.0010 | 32 | SGD | 0.2 | 20 | 0.5104 | 3.019484 | N/A |
| MLP | Deep | 0.0001 | 128 | Adam | 0.2 | 19 | 0.4934 | 2.428194 | N/A |
| MLP | Best Overall | 0.0010 | 32 | SGD | 0.2 | 20 | N/A | 3.151721 | 0.5105 |
| CNN | Baseline | 0.0100 | 32 | SGD | N/A | 20 | 0.579 | 4.531758 | N/A |
| CNN | Enhanced | 0.0100 | 32 | SGD | 0.2 | 20 | 0.5674 | 4.811990 | N/A |
| CNN | Deeper | 0.0100 | 32 | SGD | 0.2 | 19 | 0.7302 | 5.121911 | N/A |
| CNN | Best Overall | 0.0100 | 32 | SGD | 0.2 | 19 | N/A | 5.012137 | 0.7442 |

<u>Figure 3: CIFAR10 Final Results</u>

## MNIST Dataset Results Analysis:

For the MNIST dataset, interestingly, across all MLP architectures and CNN architectures, the same hyperparameters yielded the best results (learning rate = 0.01, batch size = 32, optimizer = SGD, and dropout = 0.2 (except for the baseline CNN where there is no dropout)). This may be due to the random sampling of the 10/36 possible hyperparameter combinations, since I only sampled 10 combinations for each model. We see that for the multilayer perceptron model, the validation accuracy is very high already with the shallow model and increases slightly for each architecture. The best model (deep) achieved a validation accuracy of 98.36% and a test accuracy of 98.29%, which may indicate slight overfitting. For the convolutional neural networks, the baseline and enhanced models achieved similar results to the MLP models, but the deeper CNN model achieved the highest validation accuracy overall of 99.3% and the highest test accuracy of 99.23% (which may also indicate slight overfitting). I am also pretty confident that by adding more layers to the deeper CNN model, we could achieve a test accuracy of ~100%, where it is able to understand all the patterns in the images and correctly predict each digit. Overall, we see that for the MNIST dataset, both the MLP and CNN models do an excellent job of classifying each image as the correct digit (both over 98% accuracy), with the deeper CNN model slightly outperforming all the MLP models.

## CIFAR10 Dataset Results Analysis:

For the CIFAR10 dataset, we see a wider variety of hyperparameter combinations being chosen. For the multilayer perceptron models, I found that the medium architecture surprisingly had the highest validation accuracy of 51.04%. For some reason, the deep model had a lower validation accuracy than the shallow model, which suggests that if I had more time, I might have been able to find a better combination of hyperparameters for the deeper model because it should perform better than the less complex ones. The best MLP model (medium) achieved a test accuracy of 51.05%, which is not great. For CNNs, we see immediately a large improvement, as even the shallow CNN model gives an improved validation accuracy of 57.9%. As with the MNIST dataset, we see the largest improvement given for the deeper CNN, which achieved a validation accuracy of 73.02% and a test accuracy of 74.42%. For the more complex CIFAR10 dataset, we see a very large improvement given by the convolutional neural networks when compared with the multilayer perceptron models.

## Conclusions:

Overall, we see that for the simpler MNIST dataset, CNNs offer a very slight improvement over MLPs for image classification, whereas for the more complex CIFAR10 dataset, CNNs offer a significant improvement in image classification when compared with MLPs.

Some challenges I faced during this project were saving my results, due to the long runtime of the models. I addressed this problem through functions like get_save_directory(), save_results(), and load_results(). This allowed me to save hyperparameter results to files in my googledrive, so even if my runtime disconnected, I wouldn't have to start from scratch. Another problem I had was with data leakage (described earlier in the Model Selection &

Test Accuracy Section). I addressed this problem by recording the stop epochs for the training models and then retraining the full model only up until the stop epoch of the original model.

## Future Results:

Given more time, I would have liked to test more hyperparameter combinations, especially for the CNN models. I also think adding more layers to the deeper CNN model's architecture may have increased model performance, especially on the CIFAR10 dataset. Additionally, I did not have time to compute multiple validation accuracies for each model (to get the standard deviation), which would have provided more accurate results overall.