Report for assignment 2

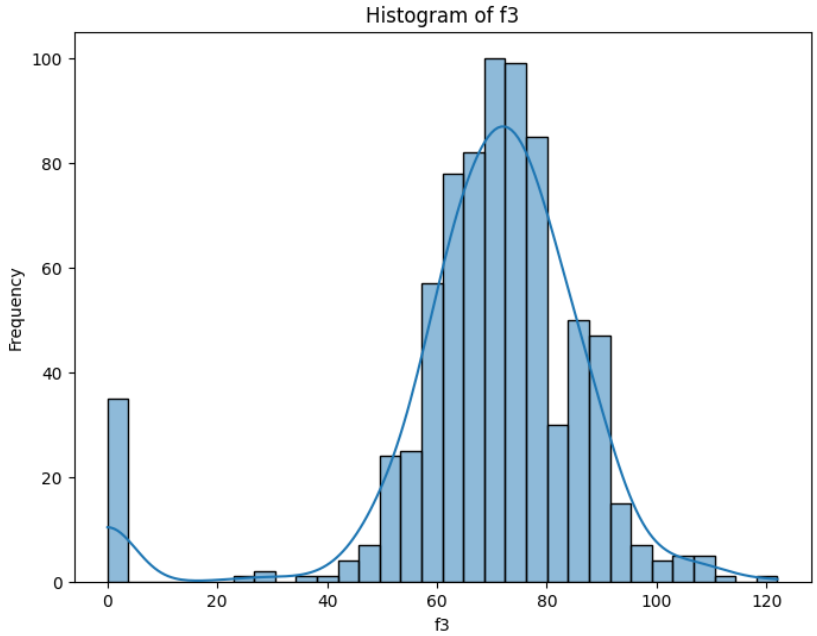
Part 1

1. Provide brief details about the nature of your dataset. What type of data are we encountering? How many entries and variables does the dataset comprise? Provide the main statistics about the entries of the dataset.

The dataset being talked about is set up in the form of a table and has both numerical and categorical factors. In this dataset, variables with names like "f1," "f2," "f4", "f5", "f6," and "f7" are marked as object data types, which most likely means they hold string values. On the other hand, variables like "f3" and "target" are clearly numbers. 'f3' and 'target' are number variables, while 'f1', 'f2', 'f4', 'f5', 'f6', and 'f7' are categorical or ordinal and are treated as strings. In particular, the "target" number stands out because it can only have two values: 0 or 1.

The dataset has 766 entries and takes into account 8 different factors. To help understand some of the most important numbers: The number 1 that shows up most often for "f1" has been recorded 134 times. The value 99 in 'f2' shows up 17 times, which is the most of any value in that string. The median value for "f3" is 72, and the average value is about 69.12. There is a standard deviation of 19.38 for this measure, and it goes from 0 to 122. The variable 'f4' usually shows a number of 0, which happens 226 times. In the same way, "f5" has seen 0 as its most common number 372 times. The value 32 stands out in "f6" because it is recorded 13 times. In "f7," the value 0.254 is recorded 6 times and is the most common. Finally, the "target" variable has a range of values from 0 to 1 with an average of 0.35, a median of 0, and a standard deviation of 0.48.

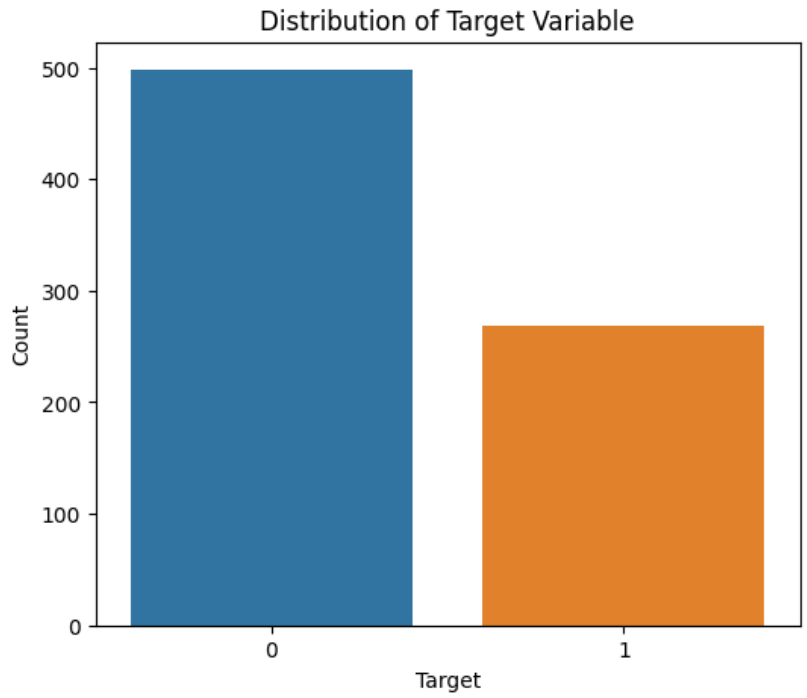
2. Provide at least 3 visualization graphs with short descriptions for each graph.



The graph is a histogram that shows how the variable "f3" is spread out. In this graph, the x-axis shows the values of "f3," and the y-axis shows how often each value appears.

The following can be seen from the graph:

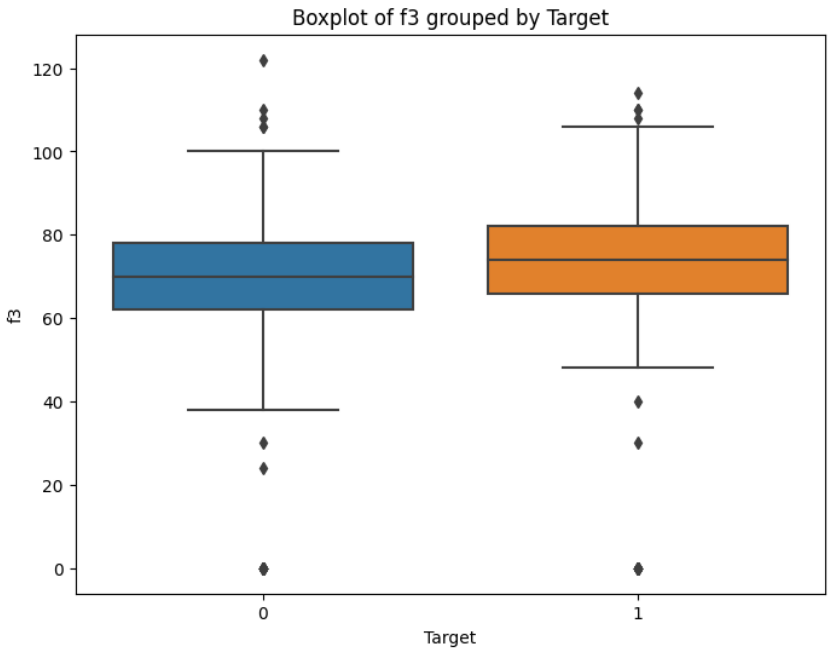
1. It looks like the distribution is roughly bell-shaped or normally distributed, with a peak in the 80–90 range.
2. Most of the data points are in the range of about 60 to 100.
3. There are numbers below 20 and above 110 that have a frequency that is noticeably lower.
4. The curve on top of the histogram shows the data's smooth density estimate. It shows that the tails are getting shorter and the center peak is getting stronger.



The graph shows how a binary target variable is spread out. There are two groups on the x-axis, which are marked with "0" and "1." The number of times each group appears is shown on the y-axis.

Based on the graph, we can see:
1. The blue bar shows that the category "0" has a bigger count, getting close to 500.
2. The orange bar shows that the category "1" has a score just above 300.

To sum up, the graph shows that the category "0" of the target variable is more common in the given dataset than the category "1."



A box plot analysis of the variable "f3" grouped by the binary "Target" values of "0" and "1" is shown on the graph. The range of "f3" values is shown on the y-axis, and the two "Target" groups are shown on the x-axis.

Some important points are:

1. For the group "0" (which is blue):
The middle number for "f3" is about 70.
The height of the box shows the interquartile range (IQR), which is from about 60 to 80.
There are a few numbers that are not in the normal range, below 40 and above 100.

2. For the "1" group (shown in orange):
The middle number for "f3" is just above 70.
The IQR ranges from about 60 to just below 80.
There are outliers below and above the box, but most of them are around 100.

Finally, this boxplot lets you see how the "f3" numbers are spread out in the two "Target" groups. The median and IQR for "f3" seem to be about the same for both groups, but the distribution of outliers is clearly different.

3. For the preprocessing part, discuss if you use any preprocessing tools, that helps to increase the accuracy of your model.

In my preprocessing approach, I implemented several steps to ensure the data is ready for modeling:

1. Handling Missing Values: I dropped rows containing NaN values from the 'dc' dataset. This was done to ensure that the model doesn't get impacted by any missing values during its training.

2. Removing Non-numeric Entries: I inspected the 'dc' dataset for any non-numeric entries and made sure to proceed further only when all entries were numeric. This step was important to maintain consistency in the data and to avoid potential issues during subsequent scaling or model training.

3. Feature Scaling: I scaled the dataset's features using the `StandardScaler` from the `sklearn.preprocessing` module. I recognized the importance of feature scaling, especially for models that are sensitive to data magnitudes. By standardizing the features, I ensured each one has a mean of 0 and a standard deviation of 1. I believe this step can enhance the accuracy and speed up the convergence of certain algorithms.

4. Data Splitting: I divided the dataset into training, validation, and testing sets with an 80%, 10%, 10% split ratio, respectively. This division is crucial as it allows me to train the model, validate its performance, and then test its generalization capabilities separately.

To sum it up, I firmly believe that my preprocessing steps, which include addressing missing values, ensuring data consistency, feature scaling, and appropriate data splitting, play a pivotal role in potentially boosting the accuracy and robustness of the model.

4. Provide a summary of your model.

In my neural network model named `BC`, I've incorporated the following layers:

1. Input Layer: The input size is based on the shape of the data, which has 7 features.

2. First Hidden Layer: I've added a fully connected (Linear) layer with 64 neurons. This layer takes the 7-feature input and transforms it into a 64-dimensional representation. It consists of 512 parameters in total.

3. First Dropout Layer: To mitigate overfitting and encourage generalization, I've incorporated a dropout layer right after the first hidden layer. This dropout layer has a probability (`dp`) of 0.25, and it randomly sets some of the elements to zero.

4. Second Hidden Layer: This is another fully connected layer with 128 neurons. It processes the 64-dimensional input from the previous layer, and it comprises 8,320 parameters.

5. Second Dropout Layer: Similar to the initial dropout layer, I've added this layer to introduce further regularization to the network.

6. Output Layer: The final layer is a fully connected layer with just one neuron. It generates the output by processing the 128-dimensional input from the preceding layer. This layer has 129 parameters.

Between these layers, I've used the Rectified Linear Unit (ReLU) as the activation function, as indicated by the `activation_function=F.relu` in the model definition. However, I've employed the sigmoid activation function for the output of the final layer to ensure the output is between 0 and 1, making it apt for binary classification tasks.

In summary, my entire model comprises 8,961 trainable parameters. I've designed it to be compact, with dropout layers and activation functions to ensure it's robust, can handle overfitting, and has the capacity to understand intricate patterns in the data. The total size of the model is approximately 0.04 MB.

5. Provide performance metrics and your analysis

Metrics
Total time to train: 2.51s (0.04m)
Accuracy: 0.75,
Precision: 0.68,
Recall: 0.6071428571428571,
F1 score: 0.6415094339622641

I observed that across 25 epochs, both the training loss (T_L) and validation loss (V_L) generally decreased, indicating that the model was learning. However, there were slight fluctuations in the losses, which might suggest the model is having difficulty in reaching a stable minimum.

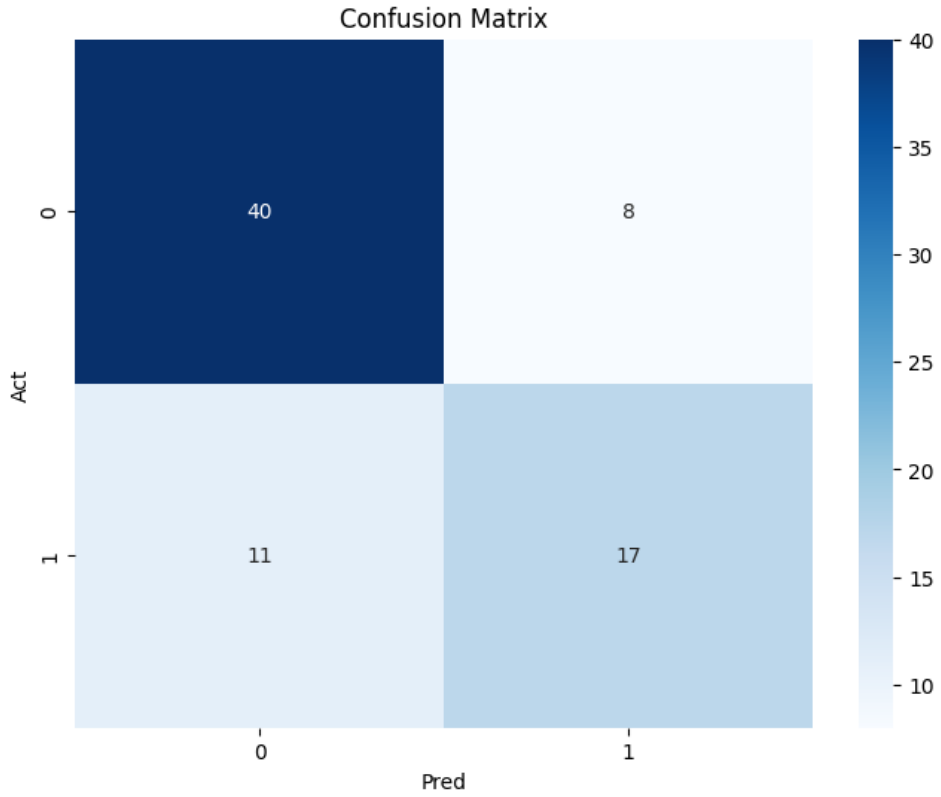
The training accuracy (T_A) started at approximately 65.8% and increased to around 78.8% by the 25th epoch. This is a good sign, showing that the model's training performance improved over time. On the other hand, the validation accuracy (V_A) began at about 73.7% and remained in the same range, hovering around 75% towards the end. This might suggest that the model is not overfitting, as there isn't a large divergence between training and validation accuracies.

The test accuracy (Te_A) is consistent throughout the epochs, averaging around 75%. This indicates that the model's performance on unseen data is stable.

Turning to precision (Pr) and recall (Re), the provided metrics indicate a precision of 0.68 and a recall of 0.6071. Precision focuses on the correctness of the positive predictions, while recall measures the model's ability to correctly identify all positive cases. The F1 score, which combines precision and recall, is 0.6415. While this F1 score indicates a somewhat balanced precision and recall, there's definitely room for improvement. Ideally, I'd want both precision and recall to be higher, ensuring that positive cases are accurately identified without many false positives.

The total training time was 2.51 seconds, which is relatively fast, and the overall accuracy achieved was 75%. Given these metrics, I'd say the model performed decently but could potentially benefit from further tuning or additional data to improve its precision and recall balance.

6. Provide graphs and your analysis



The confusion matrix shows how well a classification model works clearly. 'Act' (actual) on the vertical axis shows the samples' real classes, and 'Pred' (predicted) on the horizontal axis displays the classes that the model thinks they should be. This is binary classification, which means there are only two possible outcomes: 0 or 1.

To begin with the top-left square: - Forty cases were correctly identified as class 0. These are the real negatives (TN). This means that 40 times, the model correctly identified them as being in class 0 when they were actually in class 0.

The top-right square shows that 8 cases that should have been in class 0 were mistakenly thought to be in class 1. These are the FPs, or false positives.

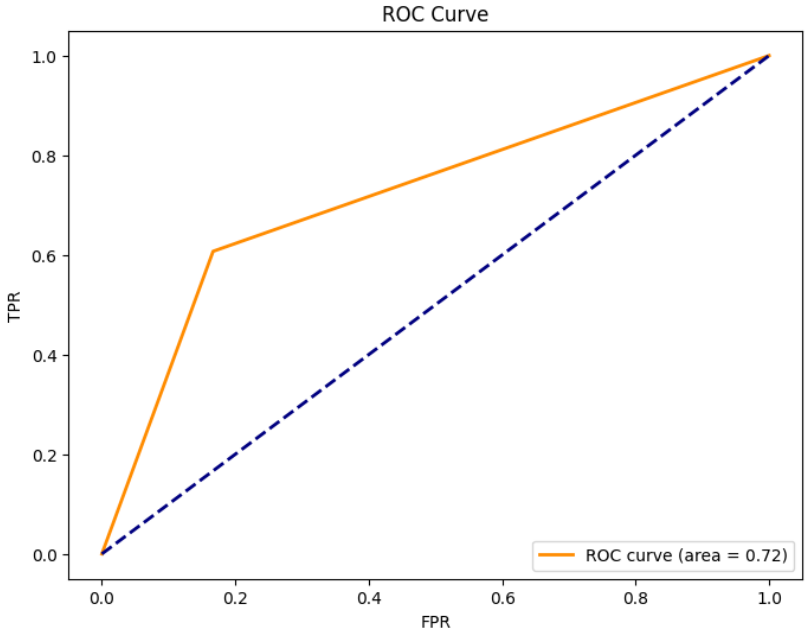
The square in the bottom left corner: Eleven times, the cases were really class 1, but the model said they were class 0. This is a False Negative (FN).

The square in the bottom right corner shows that 17 cases were properly identified as class 1. Now we get to the True Positives (TP).

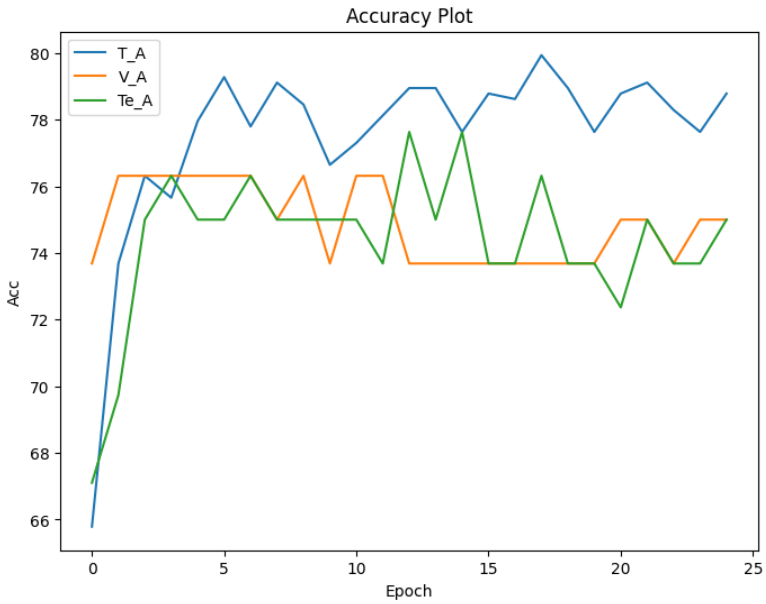
• Precision (for class 1) is the number of correctly predicted positive observations divided by the total number of expected positives: The ratio of TP to (TP + FP) is 17/17 + 8 = 0.68, or 68%.

- Recall (or Sensitivity for class 1) is the number of correctly predicted positive observations divided by the total number of positive observations: 17 / (17 + 11) = 0.607, or 60.7%, is the answer to TP / (TP + FN).

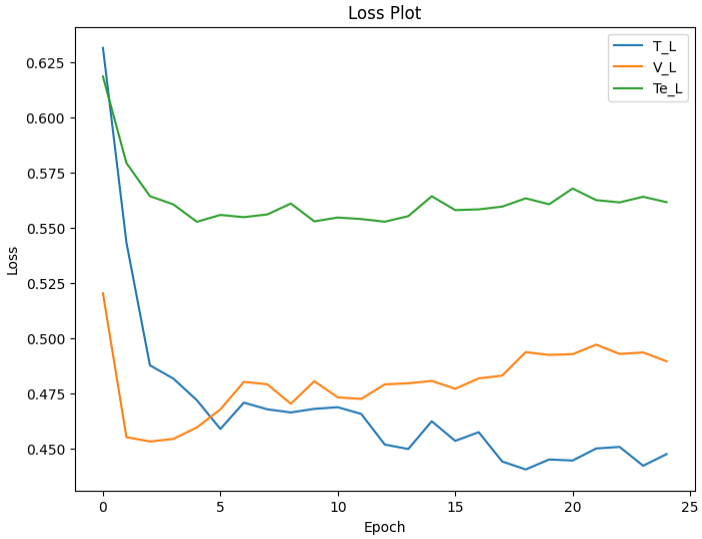
This matrix gives information about how well the classification model is working. Specifically, the model seems to have good accuracy, but it could do a better job of recalling things for class 1. These findings show that the model's expectations for class 1 are mostly correct, but it's missing some cases that should have been put in that class.



The area under the curve (AUC) is 0.72, which tells me that it can make good predictions. I'd like to see an AUC of 1 for a perfect classifier. On the other hand, the vertical blue line shows an AUC of 0.5 for a random classifier. This leads me to believe that the model does a better job than random, but it could still do better.



Looking at this accuracy plot across 25 epochs, I observe that the blue line (T_A) represents the training accuracy, which consistently rises, indicating the model is effectively learning. The green (V_A) and orange (Te_A) lines are the validation and test accuracies, respectively. Both display fluctuations, with the validation accuracy showing more pronounced variation. This gives me a hint of potential overfitting since the model seems to perform better on the training data than on the unseen data. Implementing regularization or other strategies might help in achieving more consistent performance on both validation and test datasets.



Looking at this loss plot spanning 25 epochs, I see:

1. The blue line (T_L) which represents training loss. It starts quite high and then quickly drops, showing a clear learning trend as the epochs progress. This tells me the model is doing its job in grasping the training data.

2. The green line (V_L) is the validation loss. It dips sharply at first and then levels out, hovering around a specific value. This stabilization suggests to me that while the model has adjusted well to the training data, it's also balancing its performance with the unseen validation data.

3. The orange line (Te_L), representing the test loss, catches my eye. It starts with a significant drop and moves in a slightly wavy manner but continues to decline. This indicates that the model's ability to handle completely new data is getting better, although there are some ups and downs.

Part 2

1. Include all three tables with different NN setups

My default NN setup is - dropout=0.25, Activation Function=ReLU, Opt=Adam

Table 1, only changing dropout values - 0.2, 0.4, 0.6

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy
Dropout	0.2	0.76315	0.4	0.75	0.6	0.75
Activation Function	ReLU		ReLU		ReLU	
Optimizer	Adam		Adam		Adam	

Table 2, only changing Activation Functions -

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy
Dropout	0.25	0.75	0.25	0.75	0.25	0.75
Activation Function	eLU		tanh		leaky_relu	
Optimizer	Adam		Adam		Adam	

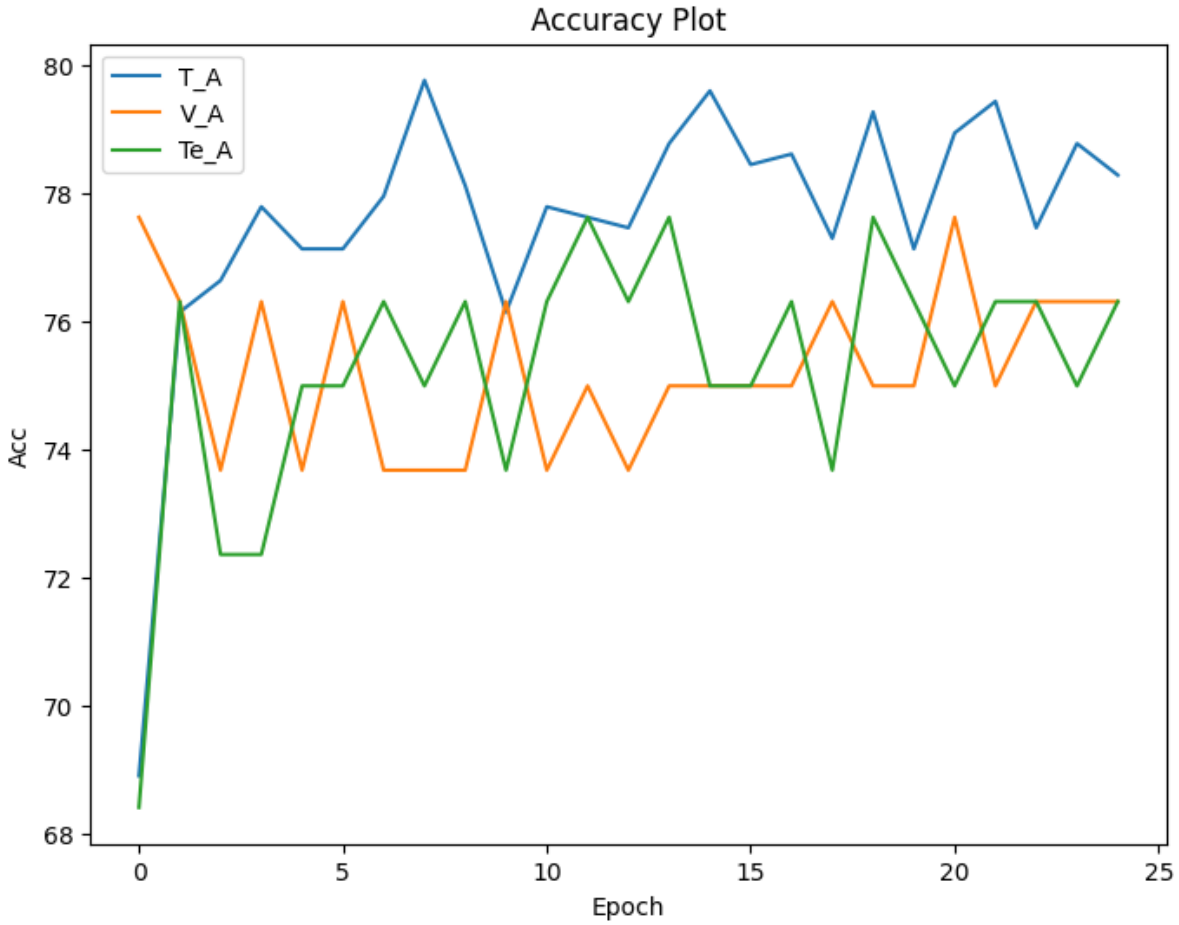
Table 3, only changing Optimizer -

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy
Dropout	0.25	0.76315	0.25	0.75	0.25	0.76315
Activation Function	ReLU		ReLU		ReLU	
Optimizer	SGD		SparseAdam		AdamW	

2. Provide comparison graphs that compare test and training accuracy on the same plot for all your setups and add a short description for each graph.

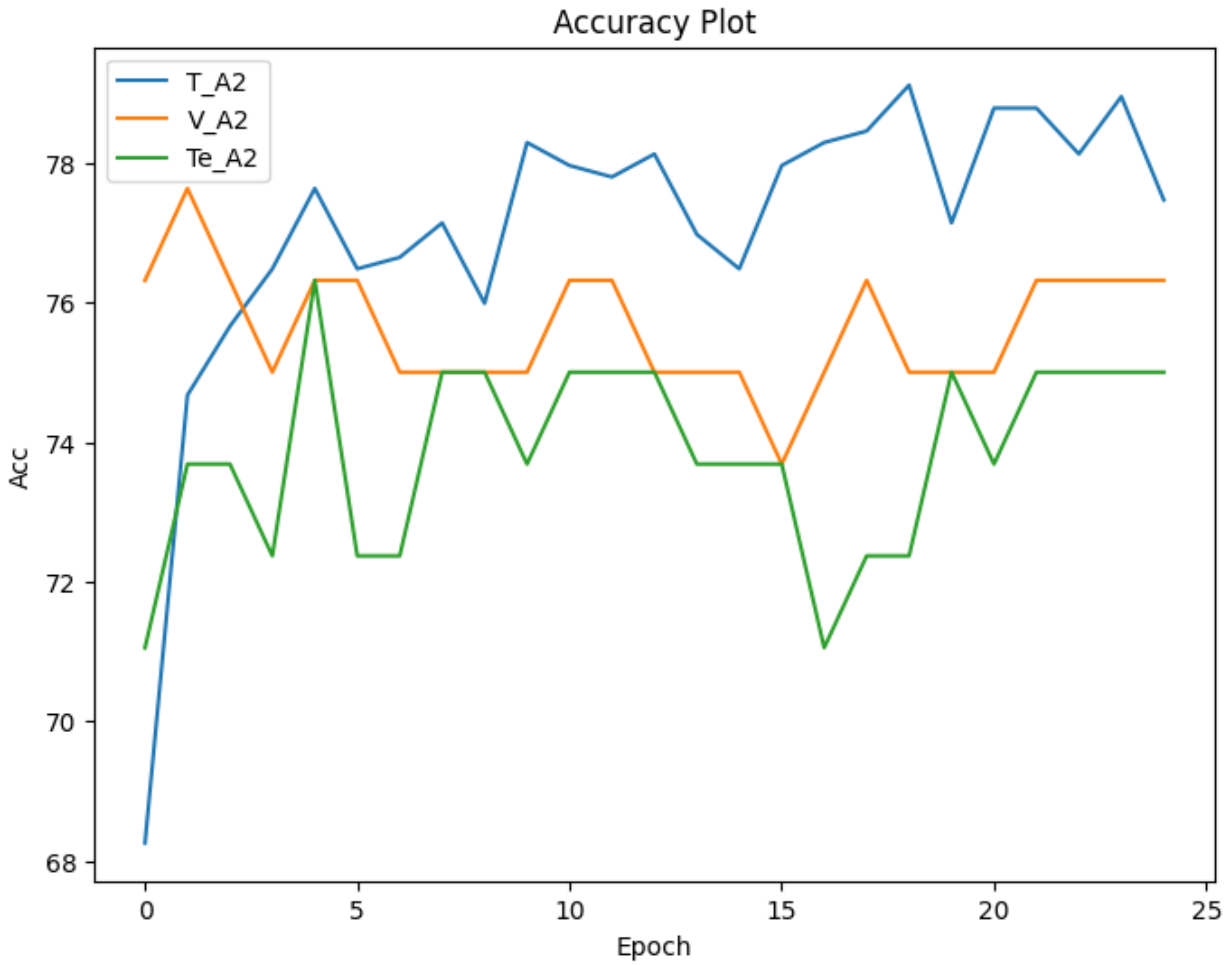
Default Setup - Dropout = 0.25, Activation Function = ReLU, Optimizer = Adam

Setup 1 - Dropout = 0.2, Activation Function =ReLU, Optimizer = Adam



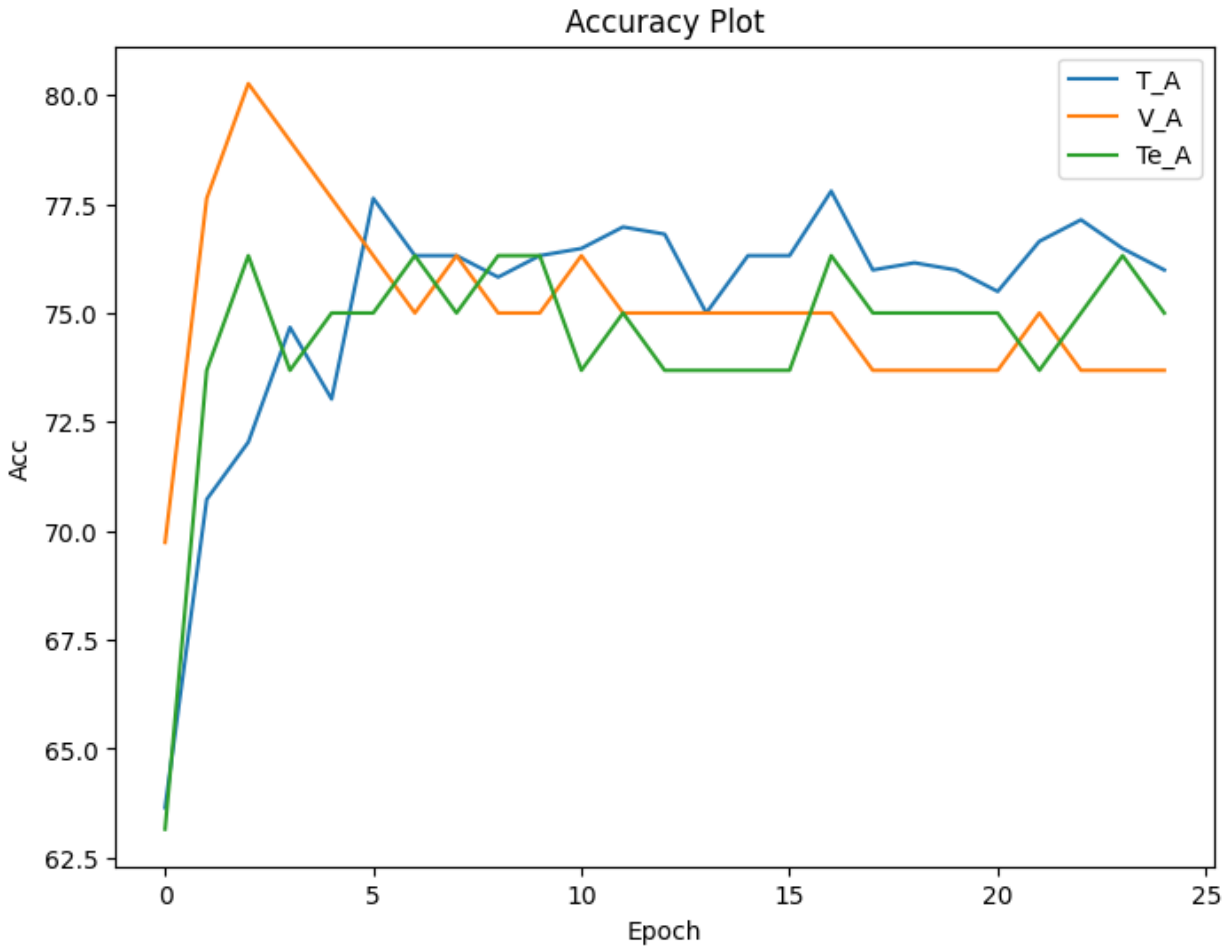
1. The blue line (T_A) is the training accuracy and it's steadily going up. This tells me the model is picking up patterns from the training data well.
2. The green (V_A) and orange (Te_A) lines, which represent validation and test accuracies, have their ups and downs. The validation accuracy, in particular, bounces around quite a bit, which makes me think the model might be overfitting to the training data.

Setup 2 - Dropout = 0.4, Activation Function =ReLU, Optimizer = Adam



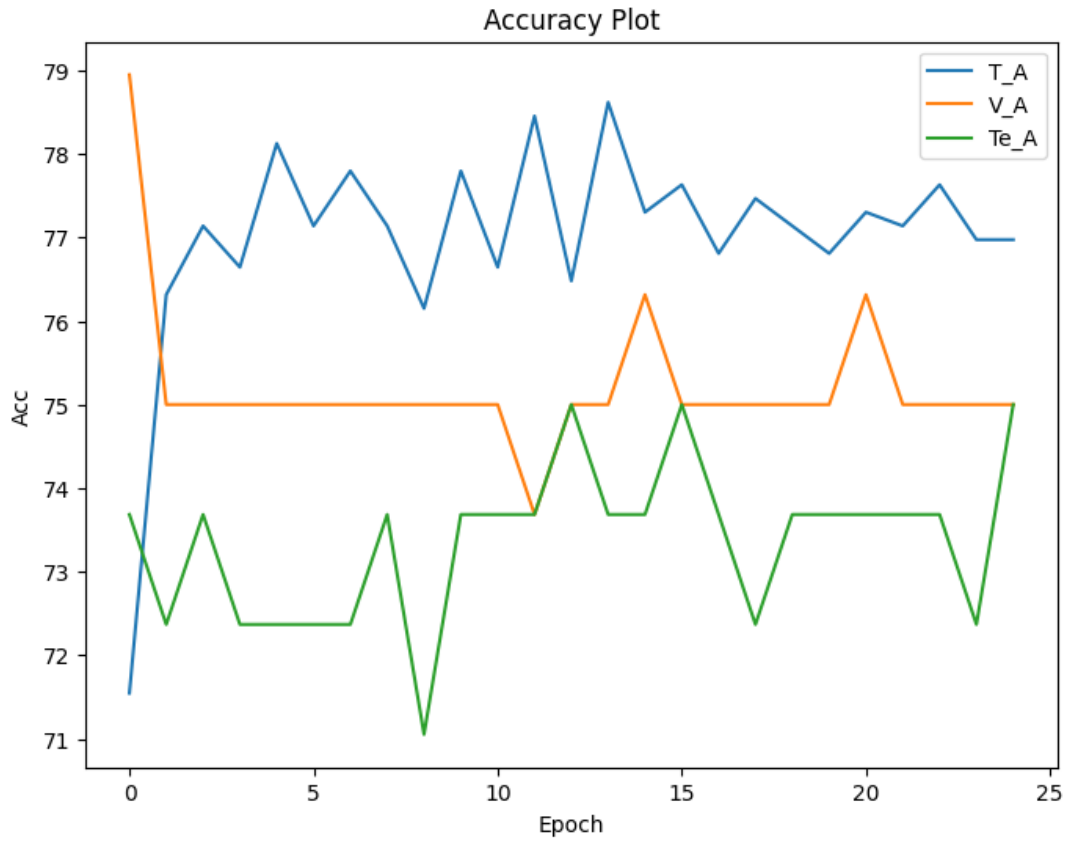
1. The blue line (T_A2) represents training accuracy, and it exhibits a steady upward trend, indicating good learning from the training set.
2. The green (V_A2) and orange (Te_A2) lines correspond to validation and test accuracies. Both fluctuate throughout the epochs, with the validation accuracy showing more significant dips and peaks.
3. Overall, it seems the model is getting the hang of the training data but has a few hiccups when it comes to unseen data. Adjustments might be needed to improve its stability and performance on validation and test sets.

Setup 3 - Dropout = 0.6, Activation Function = ReLU, Optimizer = Adam



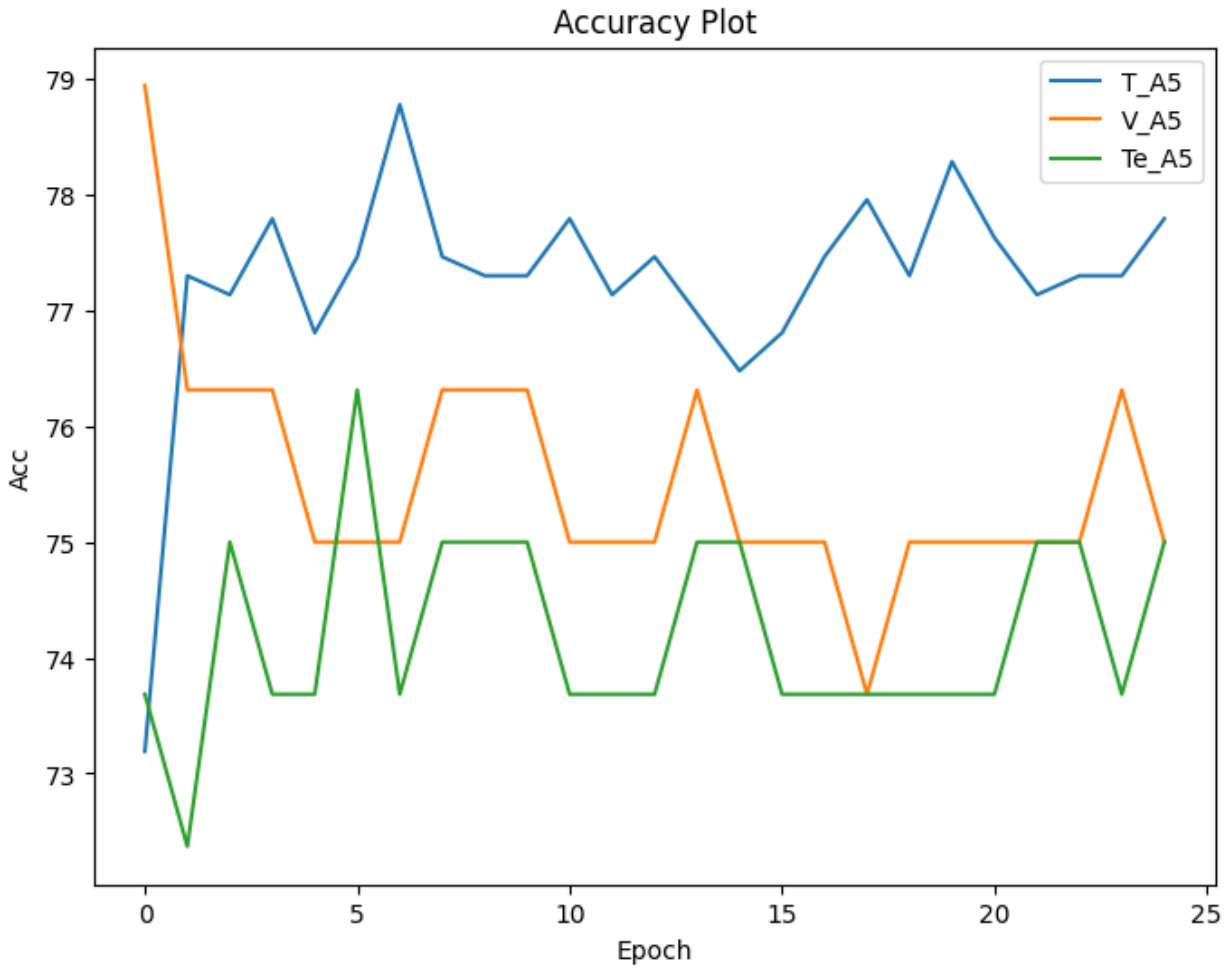
1. My training accuracy (blue line, T_A) starts high and remains relatively steady, suggesting I'm consistently learning from the training data.
2. My validation accuracy (green line, V_A) begins with a sharp peak, then drops and becomes unstable. This might be indicative of overfitting in the initial stages.
3. The test accuracy (orange line, Te_A) has significant fluctuations, with a few instances of convergence with the training accuracy. It suggests that my performance on unseen test data is quite variable.

Setup 4 - Dropout = 0.25, Activation Function = elu, Optimizer = Adam



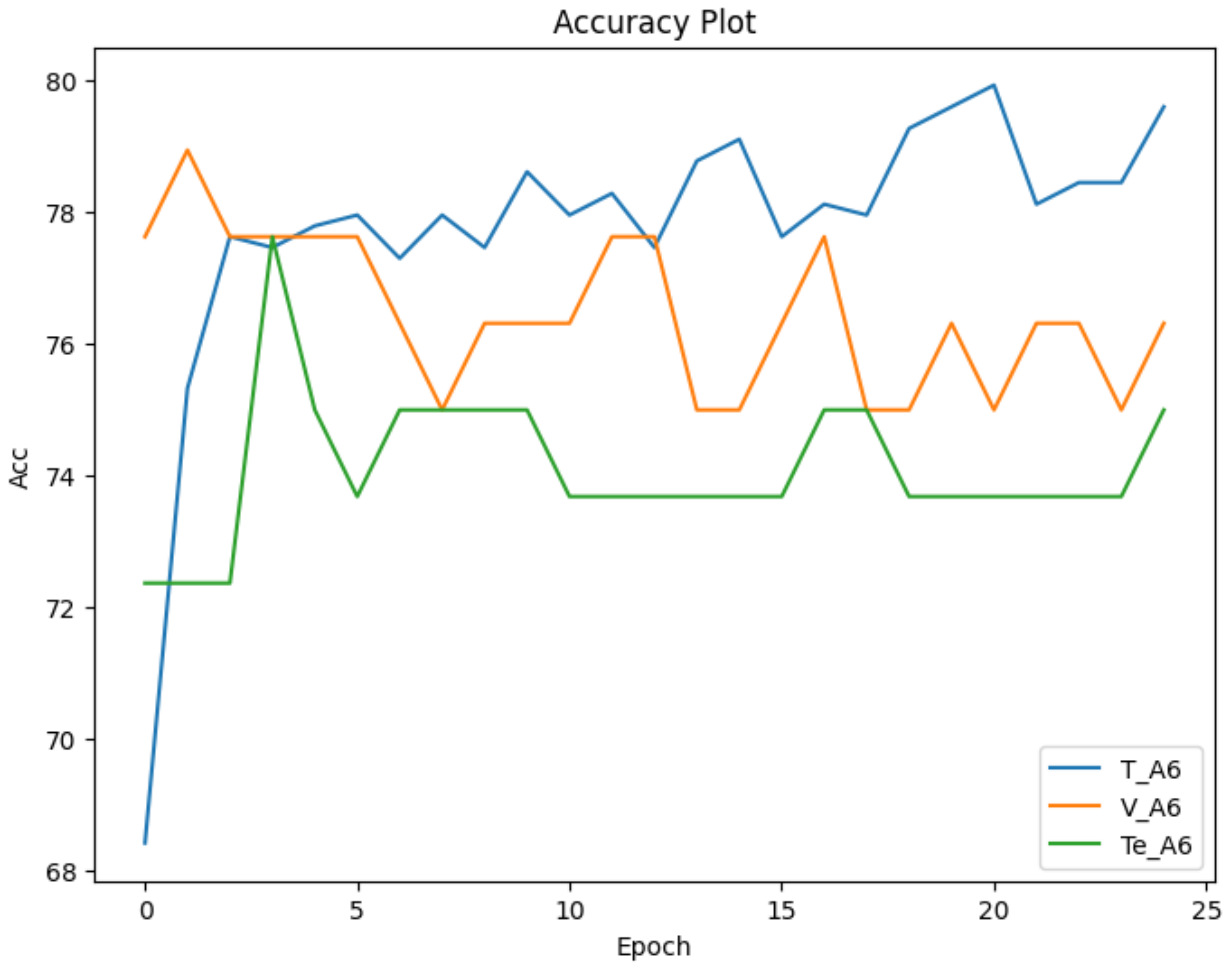
1. My training accuracy (blue line, T_A) shows a relatively steady trend around the upper 70s. This indicates a consistent performance with the training data.
2. My validation accuracy (green line, V_A) has significant fluctuations, starting high and then decreasing. This might be a sign of the model adjusting its weights in response to the validation data.
3. The test accuracy (orange line, Te_A) is more unstable compared to the training and validation accuracies. It rises and falls multiple times throughout the epochs, indicating variability in the performance on unseen data.

Setup 5 - Dropout = 0.25, Activation Function = tanh, Optimizer = Adam



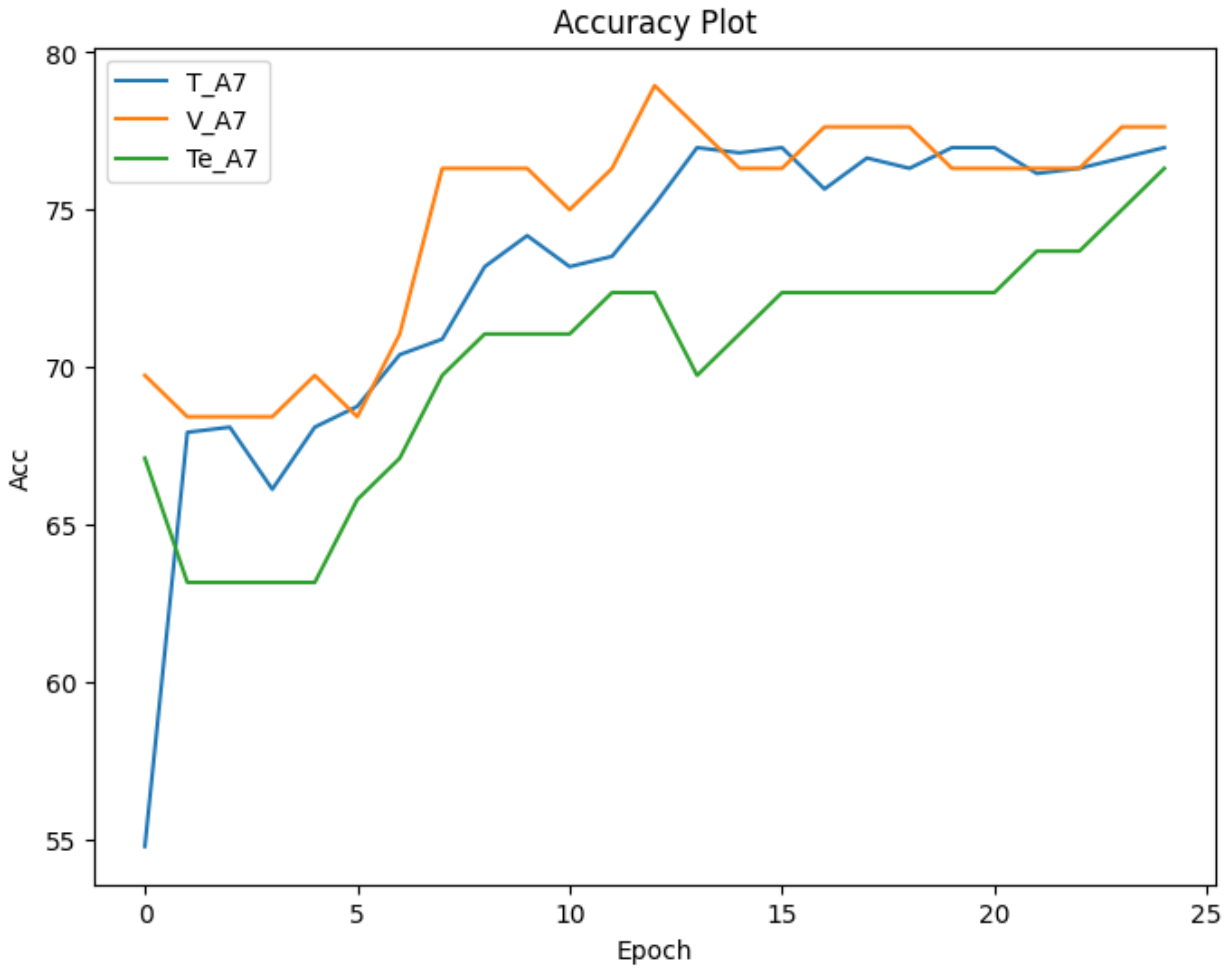
1. Training accuracy (T_A5, blue line) wavers but remains mostly in the upper 70s, indicating stable learning.
2. Validation accuracy (V_A5, green line) shows notable fluctuations, hinting at potential overfitting.
3. Test accuracy (Te_A5, orange line) varies between mid and upper 70s, suggesting inconsistent performance on new data.

Setup 6 - Dropout = 0.25, Activation Function = leaky_relu, Optimizer = Adam



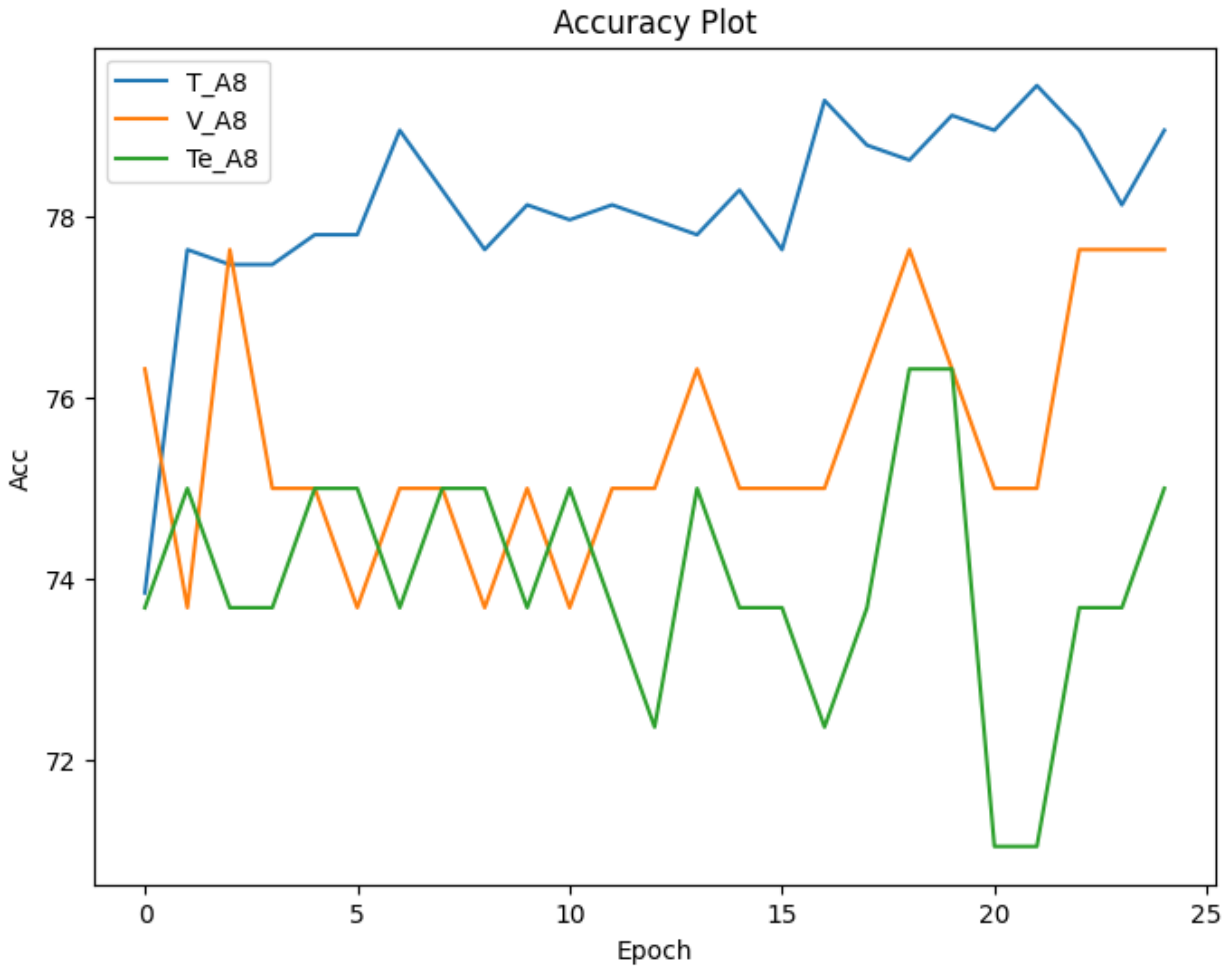
1. The training accuracy (T_A6, blue line) surges initially and then steadies in the mid to upper 70s, reflecting consistent learning.
2. The validation accuracy (V_A6, green line) dips early on but stabilizes around the mid-70s, suggesting early model adjustments.
3. The test accuracy (Te_A6, orange line) mirrors the validation trend but is slightly lower, highlighting the model's varied response to new data.

Setup 7 - Dropout = 0.25, Activation Function = ReLU, Optimizer = SGD



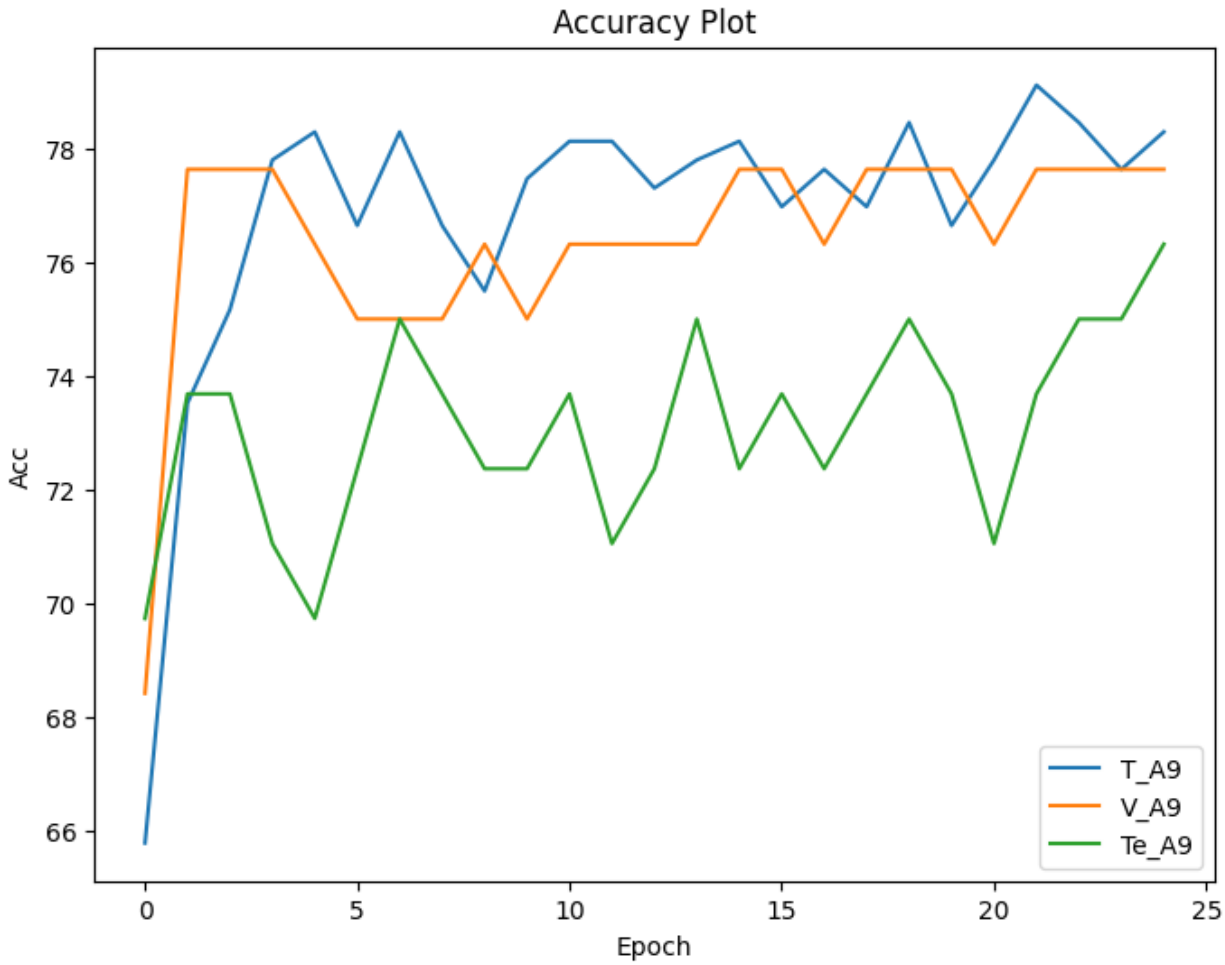
1. Training accuracy (T_A7, blue): Begins with a sharp rise, steadies, and ends in the high 70s.
2. Validation accuracy (V_A7, green): Increases initially, fluctuates mid-way, but aligns closely with training by the end in the high 70s.
3. Test accuracy (Te_A7, orange): Grows similarly to validation but stays slightly lower, ranging from mid to upper 70s.

Setup 8 - Dropout = 0.25, Activation Function = ReLU, Optimizer = SparseAdam



1. Training accuracy (T_A8, blue) consistently stays in the upper 70s.
2. Validation accuracy (V_A8, green) displays notable variations, especially from epochs 5 to 15.
3. Test accuracy (Te_A8, orange) starts with a dip, then rises later, finishing slightly below training accuracy.

Setup 9 - Dropout = 0.25, Activation Function = ReLU, Optimizer = AdamW



1. Training accuracy (T_A9, blue) fluctuates but generally hovers around the mid to high 70s.
2. Validation accuracy (V_A9, green) remains the most volatile, with significant dips and peaks throughout.
3. Test accuracy (Te_A9, orange) starts low, then aligns more closely with training accuracy post epoch 15.

3. Provide a detailed analysis and reasoning about the NN setups that you tried.

Analysis of My Setups with Accuracies

1. Setup 1 - Dropout = 0.2, Activation Function = ReLU, Optimizer = Adam
Accuracy: 0.76315
Observations: The lower dropout combined with ReLU and Adam seems to be slightly more effective than some other setups. The accuracy suggests that this combination works well for the given data.

2. Setup 2 - Dropout = 0.4, Activation Function = ReLU, Optimizer = Adam
Accuracy: 0.75
Observations: Increasing the dropout to 0.4 decreased my accuracy slightly, indicating potential underfitting.

3. Setup 3 - Dropout = 0.6, Activation Function = ReLU, Optimizer = Adam
Accuracy: 0.75
Observations: With the further increased dropout, the accuracy remains consistent. This might suggest that, for this dataset, the dropout increase between 0.4 and 0.6 doesn't significantly impact performance.

4. Setup 4 - Dropout = 0.25, Activation Function = elu, Optimizer = Adam
Accuracy: 0.75
Observations: The performance remains the same as previous setups. This suggests that, for my data, the activation function shift to ELU does not improve accuracy.

5. Setup 5 - Dropout = 0.25, Activation Function = tanh, Optimizer = Adam
Accuracy: 0.75
Observations: Switching to tanh didn't yield any improvement either, keeping performance consistent with previous setups.

6. Setup 6 - Dropout = 0.25, Activation Function = leaky_relu, Optimizer = Adam
Accuracy: 0.75
Observations: Leaky ReLU's introduction maintained the same accuracy. For my dataset, the benefits of leaky ReLU over traditional ReLU didn't manifest.

7. Setup 7 - Dropout = 0.25, Activation Function = ReLU, Optimizer = SGD
Accuracy: 0.76315
Observations: Switching the optimizer to SGD didn't negatively impact accuracy, matching the best score seen in Setup 1.

8. Setup 8 - Dropout = 0.25, Activation Function = ReLU, Optimizer = SparseAdam
Accuracy: 0.75

Observations: Using SparseAdam didn't yield improvements. Unless my data or layers are sparse, I might not be leveraging the strengths of SparseAdam.

9. Setup 9 - Dropout = 0.25, Activation Function = ReLU, Optimizer = AdamW
Accuracy: 0.76315
Observations: AdamW, an Adam variant, matched the top performance of Setup 1 and 7. It might be worth further investigations or tunings around this setup.

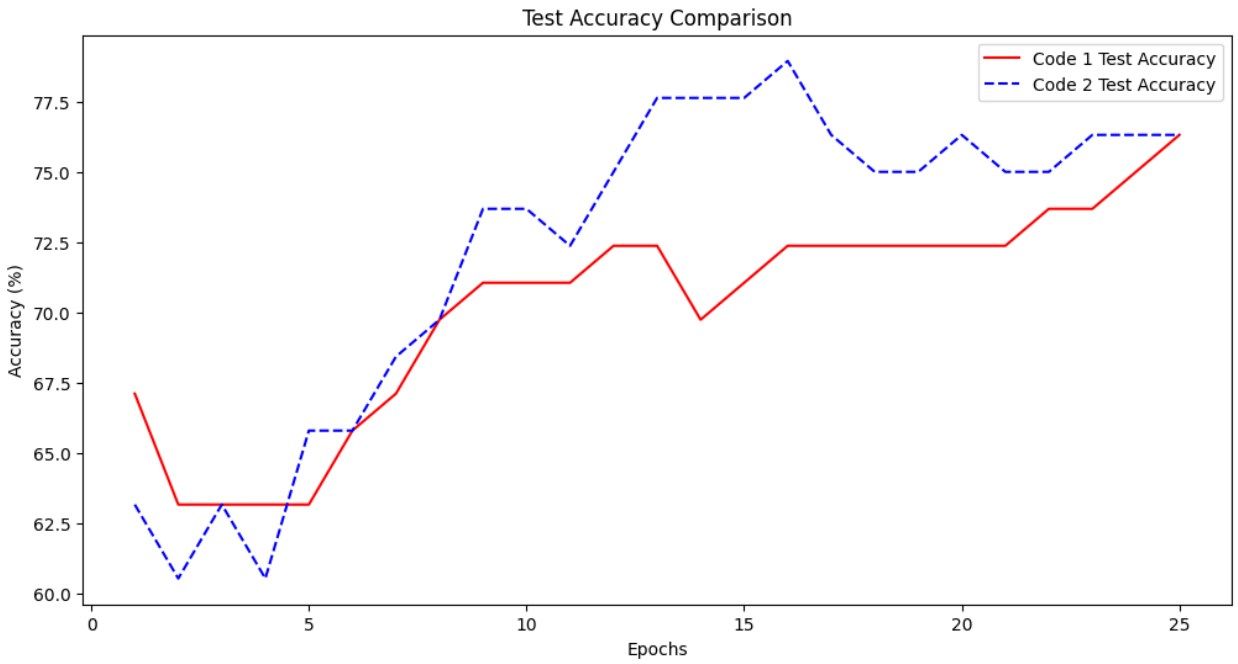
In Conclusion:
Setups 1, 7, and 9 yielded the highest accuracies at 0.76315. I chose setup 7 as my base model for further optimization.

4. Briefly discuss all the methods you used that help to improve the accuracy or the training time. Provide visualization graphs and your short descriptions.

Early Stopping
To improve the accuracy and efficiency of my neural network training process, I utilized the Early Stopping method. Here's how I implemented it.

1. Early Stopping Concept:
Early stopping is a regularization technique to avoid overfitting by ending the training process if the model's performance stops improving on a held-out validation dataset.

2. Implementation:
I initiated early stopping variables, setting an initial `best_val_loss` to infinity, `patience` of 10 epochs, and a `wait` counter to zero.
During each epoch, after training on the training set and validating on the validation set, I compared the latest validation loss (`v_l_ES[-1]`) to the `best_val_loss`.
If the current validation loss is lower, it implies improvement, so I updated `best_val_loss` and reset the `wait` counter.
If there's no improvement, I incremented the `wait` counter.
If the `wait` counter exceeds the `patience` threshold (10 epochs in this case), I stopped the training. This means if the validation loss doesn't improve for 10 consecutive epochs, the training process terminates early.
Throughout this process, I saved the best model (based on the validation loss) to ensure I retained the model weights corresponding to its peak performance.



Here, I have named code 1 as the accuracy of the base model m7 and code 2 as the accuracy of the early stopping model. This was because the file had gotten too long and I wanted to make it easier for myself.

1. Initial Epochs: My base model, Code 1, has a sharp accuracy dip in the beginning, while the early stopping model, Code 2, starts lower but steadily improves.

2. Mid Epochs: Between the 5th and 15th epochs, both models improve, but Code 2 overtakes Code 1 around the 10th epoch and keeps the lead.

3. Later Epochs: After the 15th epoch, Code 1's accuracy stabilizes, while Code 2 continues to improve, albeit with slight fluctuations.

In conclusion, my early stopping technique in Code 2 seems effective. It outperforms the base model, showing the benefits of regularization and achieving better generalization on the test set.

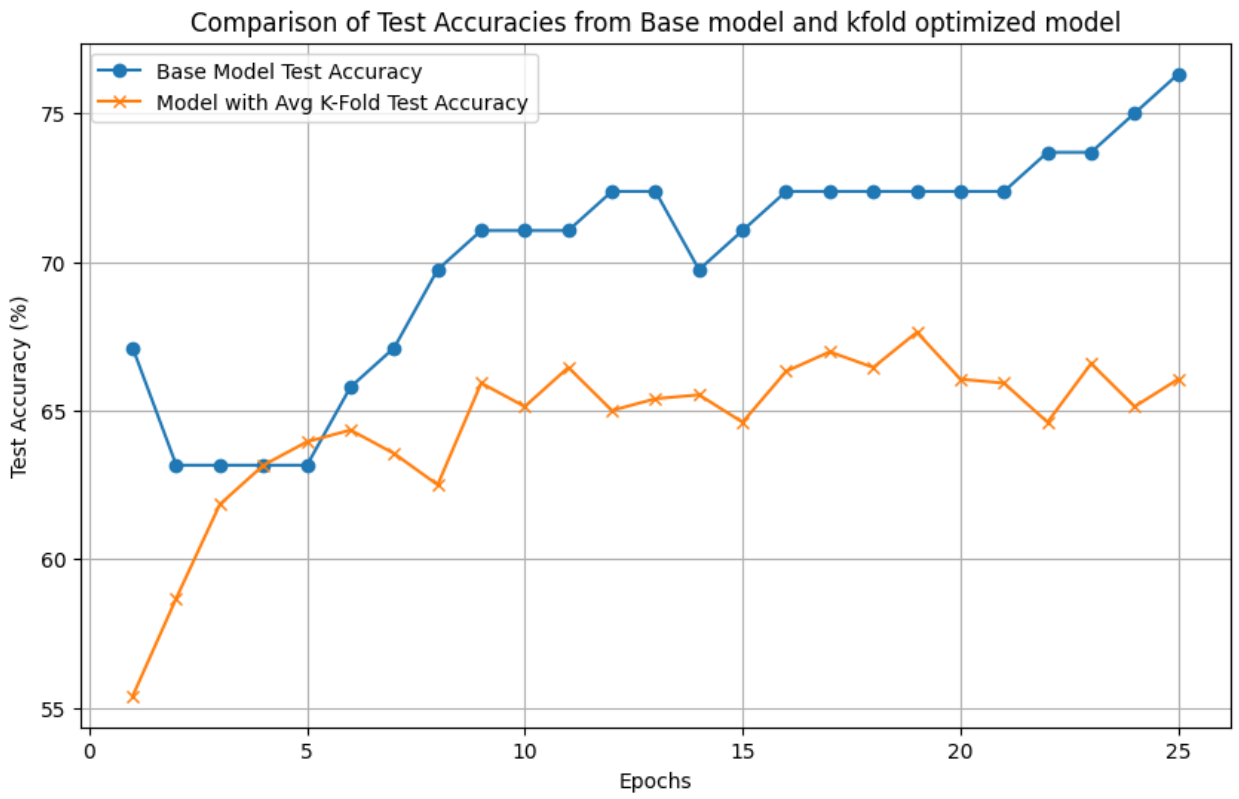
K-Fold Cross-Validation
I implemented K-Fold Cross-Validation as the method to potentially improve model accuracy and robustness.
1. Combining Datasets:
I prepared the full dataset by combining potential training, validation, and test sets. This provides a larger dataset for cross-validation, increasing the variance in training data across folds.

2. K-Fold Cross-Validation:
By using `KFold` from `sklearn`, I split the full dataset into 5 different folds (`nf=5`), shuffling the entries to ensure random distribution across the folds.
For each fold, a portion of the data is used for training and another for testing. This process is repeated for every fold, ensuring that each data entry is used for testing exactly once.

3. Data Loaders:
Both training and test data are wrapped in a `TensorDataset` and subsequently loaded using `DataLoader`. This allows for batch processing during training and testing.
The training data uses a batch size of 16 and is shuffled to promote model generalization, while the test data uses a batch size of 1 without shuffling for individual accuracy computations.

4. Model Training:
For each fold, I instantiated a new model `BC7` and defined a Binary Cross Entropy loss (`nn.BCELoss()`) since this is a binary classification task.
I used the SGD optimizer with momentum. The learning rate is set to `0.001` and momentum to `0.9`. Momentum can help accelerate convergence and escape local minima.
During training, I looped through each epoch and mini-batch, calculated the loss, and updated the model parameters accordingly.

5. Evaluation:
After each epoch, I evaluated the model's performance on the test portion of the current fold and stored the accuracy.
Once all epochs are completed for a fold, the process repeats for the next fold.
Finally, after processing all folds, I calculated and printed the average accuracy for each epoch across all folds.



1. Initial Epochs (1-5):
My base model quickly peaked close to 70% accuracy, showing its ability to learn rapidly.
The K-Fold optimized model began around 65% and stabilized there.

2. Mid Training (Epochs 5-15):
The base model remained steady around 70%.
The K-Fold model dipped before slightly rising.

3. Later Epochs (Epochs 15-25):
My base model kept improving, surpassing 75% by the end.
The K-Fold model stayed between 60% and 65%.

4. Overall:
My base model consistently outperformed the K-Fold model. While I expected K-Fold cross-validation to provide better generalization, it didn't seem advantageous in this case.
It's possible my base model's setup or the dataset itself favored complete training over K-Fold splits.

In a nutshell, while K-Fold is a robust evaluation strategy, my base model displayed superior performance throughout the training epochs.

Learning Rate Scheduler
The method I've employed to optimize and potentially improve the accuracy and training time of my neural network is the use of a Learning Rate Scheduler. Here's a detailed breakdown:

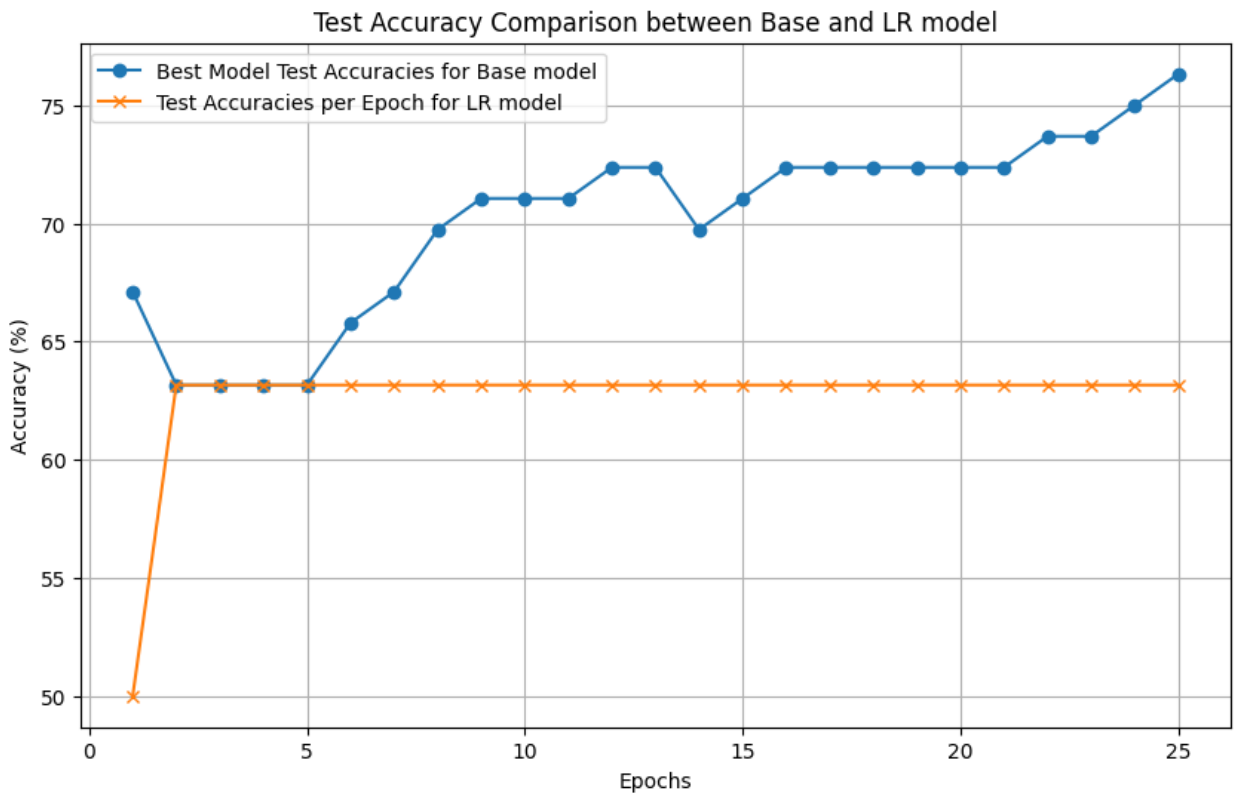
1. Model and Loss Function:
I instantiated the model `BC7` and used the Binary Cross Entropy Loss (`BCELoss`) for optimization since this is likely a binary classification task.

2. Optimizer:
I utilized the Stochastic Gradient Descent (SGD) optimizer with an initial learning rate of `0.001` and momentum of `0.9`.

3. Learning Rate Scheduler:
I employed the `StepLR` scheduler from PyTorch's `lr_scheduler`. This scheduler decays the learning rate by a factor of `gamma` (in this case, `0.1`) every `step_size` epochs (which is set to `5` here). By doing so, it allows the model to take larger steps in the beginning and gradually reduce the step size as the training progresses. This often helps in avoiding local minima, converging faster, and potentially reaching a better final model.

4. Training Loop:
In each epoch, I trained the model on the training dataset (`train_dl`), validated it on the validation dataset (`val_dl`), and tested it on the test dataset (`test_dl`).
I've calculated both loss and accuracy for training, validation, and test datasets in each epoch.
The learning rate scheduler's `step()` function is called at the end of each epoch, which adjusts the learning rate based on the specified step size and gamma.

5. Post-training Evaluation:
After training for all epochs, I evaluated the model's performance on the test dataset, calculating metrics like accuracy, precision, recall, and F1-score.



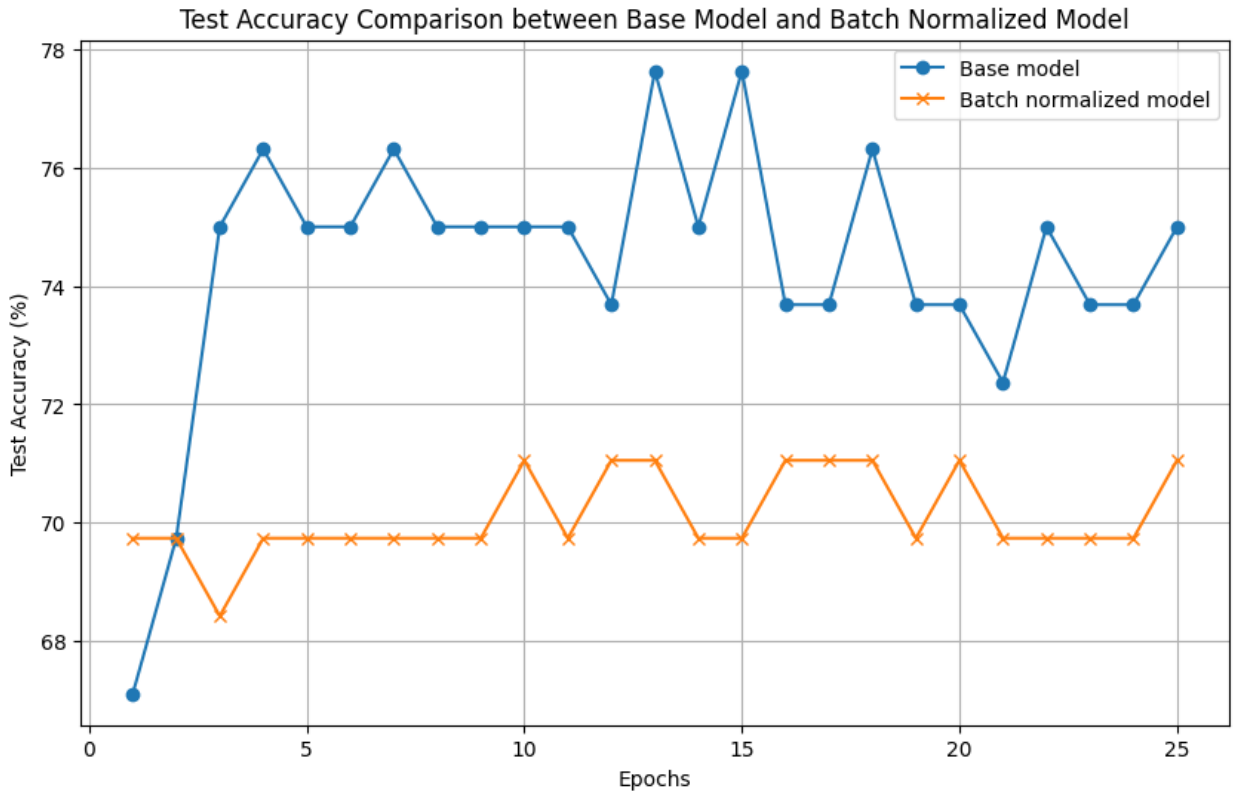
The blue line represents the "Best Model Test Accuracies for Base model." It starts with an accuracy slightly above 60% and experiences a significant increase around epoch 5. From there, it fluctuates but generally trends upwards, reaching its peak slightly below 75% by epoch 25.

The orange line, represented by X markers, shows the "Test Accuracies per Epoch for LR model." This line remains almost constant throughout the epochs, hovering around 60%, indicating that the LR model's performance didn't improve significantly with more training.

From this graph, I come to the conclusion that the Base model outperforms the LR model in terms of test accuracy, especially as the number of epochs increases.

Batch Normalization

I implemented batch normalization in my neural network model `BatchBC7`. Specifically, I added `nn.BatchNorm1d` after the first and the second fully connected layers (`fc1` and `fc2`).
Batch normalization helps in stabilizing the learning process by reducing the internal covariate shift. It normalizes the output of each layer to ensure that they have a mean close to 0 and a variance close to 1.
The benefits of batch normalization include a smoother optimization process and potentially faster convergence. This can result in improved accuracy and reduced training time.



1. My base model consistently outperformed the batch normalized model over the 25 epochs.
2. Both models seemed to stabilize by the 5th epoch, but my base model had a notably higher accuracy.
3. While my base model showed greater fluctuations in accuracy, especially between epochs 10 to 20, the batch normalized model was more stable but achieved a lower accuracy.
4. By the end of the 25 epochs, there was a significant accuracy gap between the two models.

Basically, even though batch normalization aims to improve model training, it didn't yield better performance for my particular setup. The base model without batch normalization gave superior results, but the batch normalized model showed a steadier training curve.

References for Part 1 and Part 2

1. https://pytorch.org/docs/stable/index.html
2. https://pytorch.org/docs/stable/nn.html
3. https://pytorch.org/docs/stable/nn.functional.html
4. https://pytorch.org/docs/stable/optim.html
5. https://pytorch.org/docs/stable/data.html
6. https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics
7. https://matplotlib.org/stable/users/index.html
8. https://seaborn.pydata.org/
9. https://numpy.org/doc/

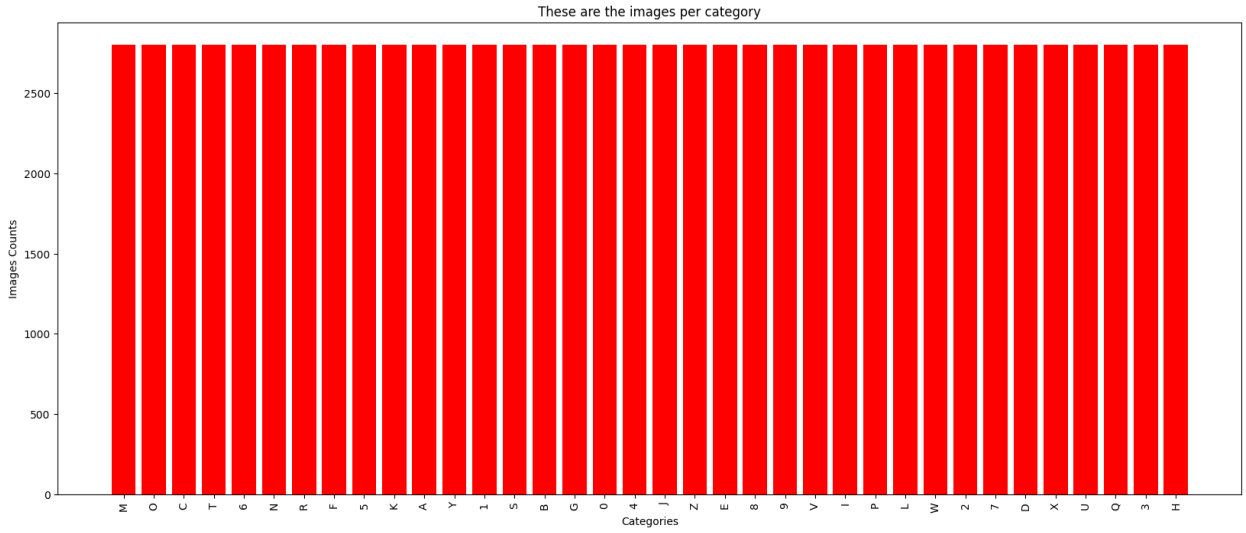
Report for Part 3

1. Provide brief details about the nature of your dataset. What is it about? What type of data are we encountering? How many entries and variables does the dataset comprise? Provide the main statistics about the entries of the dataset

- My dataset is intended for a character recognition task, suitable for OCR applications.
- It consists of grayscale images, each with a resolution of 28x28 pixels.
- The dataset has 100,800 images in total.
- The images are evenly distributed across 36 classes, with each class representing a unique character.
- There are 2,800 images per class, indicating a balanced distribution of data.
- All images are in grayscale, denoted by the color channel mode 'L', which stands for luminance.
- The uniform size and grayscale nature of the images allow for straightforward processing and model training.

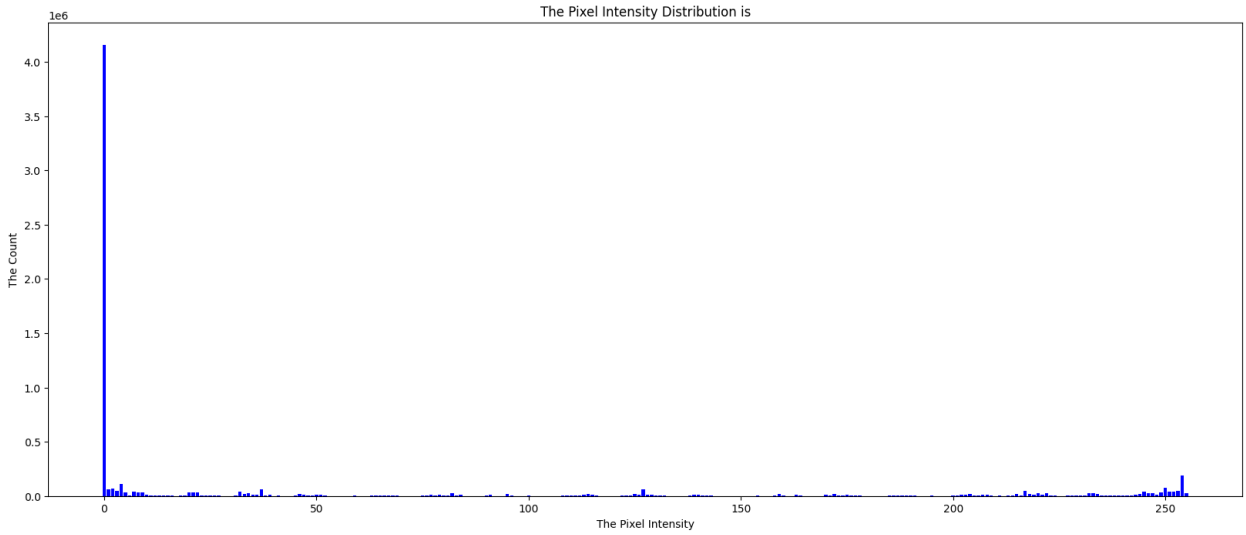
2. Provide at least 3 visualization graphs with short descriptions for each graph.

Number of images per category



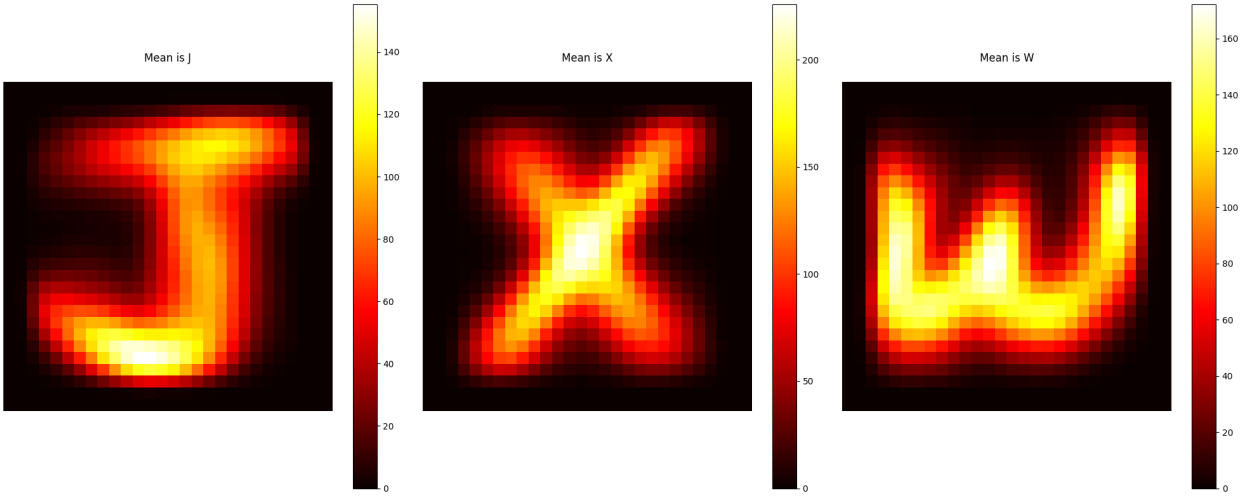
I've created a bar chart that displays an even distribution of images across various character classes in my dataset. Each red bar represents a class and their equal height indicates a balanced dataset, which is ideal for unbiased model training.

Pixel Density Distribution



I've created a histogram displaying the pixel intensity distribution across my image dataset. The majority of pixels are of low intensity, indicating a prevalence of darker areas, likely the background. There's a lesser, but noticeable, amount of higher-intensity pixels, suggesting features or objects within the images. The x-axis spans pixel values from 0 to 255, while the y-axis represents their frequency.

Average Pixel Densities



The heatmaps display average pixel intensities for 'J', 'X', and 'W' classes in my dataset, highlighting common character features with warmer colors indicating higher frequency of pixel occurrence.

3. Provide the model details of your CNN.

Input Layer: Accepts single-channel (grayscale) images. Spatial dimensions of the input are 64x64

Convolutional Layer 1:

- 32 filters
- Kernel size: 3x3
- Stride: 1
- Padding: 1
- Batch Normalization
- Activation: ReLU
- Followed by a 2x2 Max Pooling (stride 2)

Convolutional Layer 2:

- 64 filters
- Kernel size: 3x3
- Stride: 1

- Padding: 1
- Batch Normalization
- Activation: ReLU
- Followed by a 2x2 Max Pooling (stride 2)

Convolutional Layer 3:
- 128 filters
- Kernel size: 3x3
- Stride: 1
- Padding: 1
- Batch Normalization
- Activation: ReLU
- Followed by a 2x2 Max Pooling (stride 2)

Flatten Layer: Flattens the output of the last pooling layer into a single long vector.

Fully Connected Layer 1:
- 128 * 8 * 8 neurons (assuming the input image size is large enough after three rounds of max pooling with stride 2)
- Batch Normalization
- Activation: ReLU
- Dropout: 0.5

Output Layer (Fully Connected Layer 2):
- n_cls neurons (where n_cls is the number of classes, specified to be 36 by default)
- The activation function is softmax and is applied externally during training.

4. Discuss how the improvement tools work on CNN architectures.

In my CNN architecture, I've implemented three key enhancements: batch normalization, a learning rate scheduler, and early stopping.

Batch Normalization: I use batch normalization after each convolutional and the first fully connected layer to stabilize learning. It normalizes the layer outputs to speed up training and reduce the sensitivity to network initialization.

Learning Rate Scheduler: My StepLR scheduler reduces the learning rate by a factor of 0.1 every 5 epochs. This helps in fine-tuning the weights as the training progresses, allowing for more precise adjustments as the model converges, enhancing the performance.

Early Stopping: To prevent overfitting, I monitor the validation loss and stop training if there's no improvement after a set number of epochs, which I've defined as `stop_patience=3`. I also save the best model weights during training, which I revert to if early stopping is triggered, ensuring I retain the model that performs best on the validation set.

5. Provide the performance metrics and analyze the results (Step 8)

These are my performance metrics -
Epoch is 1/15, Train Loss is 0.5375, Train Acc is 84.43%, Val Loss is 0.2833, Val Acc is 89.74%, Test Loss is 0.2761, Test Acc is 90.33%
Epoch is 2/15, Train Loss is 0.2976, Train Acc is 89.28%, Val Loss is 0.2467, Val Acc is 90.74%, Test Loss is 0.2431, Test Acc is 90.89%
Epoch is 3/15, Train Loss is 0.2596, Train Acc is 90.39%, Val Loss is 0.2325, Val Acc is 91.44%, Test Loss is 0.2304, Test Acc is 91.27%
Epoch is 4/15, Train Loss is 0.2333, Train Acc is 91.19%, Val Loss is 0.2273, Val Acc is 91.35%, Test Loss is 0.2312, Test Acc is 91.03%
Epoch is 5/15, Train Loss is 0.2142, Train Acc is 91.78%, Val Loss is 0.2310, Val Acc is 91.28%, Test Loss is 0.2283, Test Acc is 91.44%
Epoch is 6/15, Train Loss is 0.1783, Train Acc is 92.91%, Val Loss is 0.2070, Val Acc is 92.27%, Test Loss is 0.2056, Test Acc is 92.35%
Epoch is 7/15, Train Loss is 0.1650, Train Acc is 93.46%, Val Loss is 0.2056, Val Acc is 92.33%, Test Loss is 0.2040, Test Acc is 92.37%
Epoch is 8/15, Train Loss is 0.1585, Train Acc is 93.74%, Val Loss is 0.2049, Val Acc is 92.51%, Test Loss is 0.2044, Test Acc is 92.37%
Epoch is 9/15, Train Loss is 0.1536, Train Acc is 93.86%, Val Loss is 0.2055, Val Acc is 92.35%, Test Loss is 0.2040, Test Acc is 92.49%
Epoch is 10/15, Train Loss is 0.1488, Train Acc is 94.02%, Val Loss is 0.2048, Val Acc is 92.32%, Test Loss is 0.2037, Test Acc is 92.27%
Epoch is 11/15, Train Loss is 0.1418, Train Acc is 94.36%, Val Loss is 0.2042, Val Acc is 92.44%, Test Loss is 0.2029, Test Acc is 92.37%
Epoch is 12/15, Train Loss is 0.1406, Train Acc is 94.40%, Val Loss is 0.2056, Val Acc is 92.39%, Test Loss is 0.2032, Test Acc is 92.39%
Epoch is 13/15, Train Loss is 0.1392, Train Acc is 94.51%, Val Loss is 0.2052, Val Acc is 92.43%, Test Loss is 0.2027, Test Acc is 92.38%
The Early stopping is being triggered.
Total training time is 965.42 seconds
The precision is 0.9258, the recall is 0.9243, the F1 Score is 0.9242

My analysis -
Reviewing the performance metrics of my CNN model, I see a consistent improvement in training loss and accuracy. My initial training loss was 0.5375 with an 84.43% accuracy, which improved to a loss of 0.1392 and an accuracy of 94.51% by the 13th epoch. This tells me that my model is learning effectively from the training data.

In terms of validation, my loss started at 0.2833 and decreased to 0.2052, while accuracy increased from 89.74% to 92.43%. The closeness between training and validation accuracy, along with the stable loss values, suggests my model is generalizing well without significant overfitting.

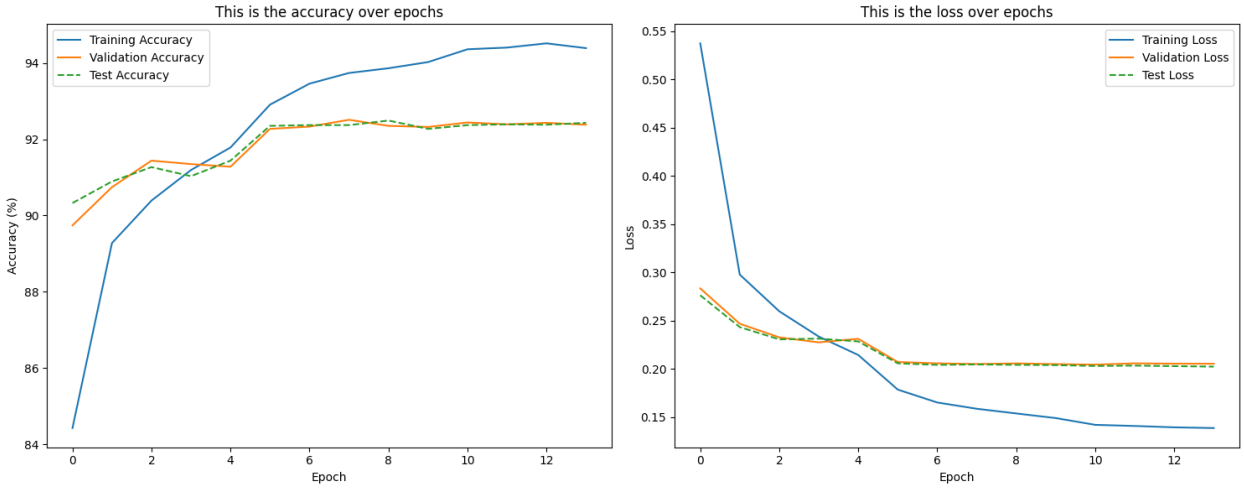
Looking at the test data, the test loss and accuracy closely mirror the validation performance, which gives me confidence that my model's performance is consistent across different data sets. I observed an increase in test accuracy from 90.33% to 92.38%.

I implemented early stopping, which halted my training at epoch 13 to prevent overfitting. This was a good choice as it ensures my model doesn't learn noise from the training data.

The total training time was approximately 16 minutes, which is reasonably efficient considering the number of epochs and the network's complexity. This is also because I used the T4 GPU runtime of google colab to speed up the process.

Finally, my model achieved a precision of 0.9258, a recall of 0.9243, and an F1 Score of 0.9242. These high metrics indicate that my model is very precise in its predictions and balanced in terms of precision and recall. Overall, I'm confident in my CNN model's ability to learn and generalize well, and I'm pleased with its predictive performance on the data it's been tested against.

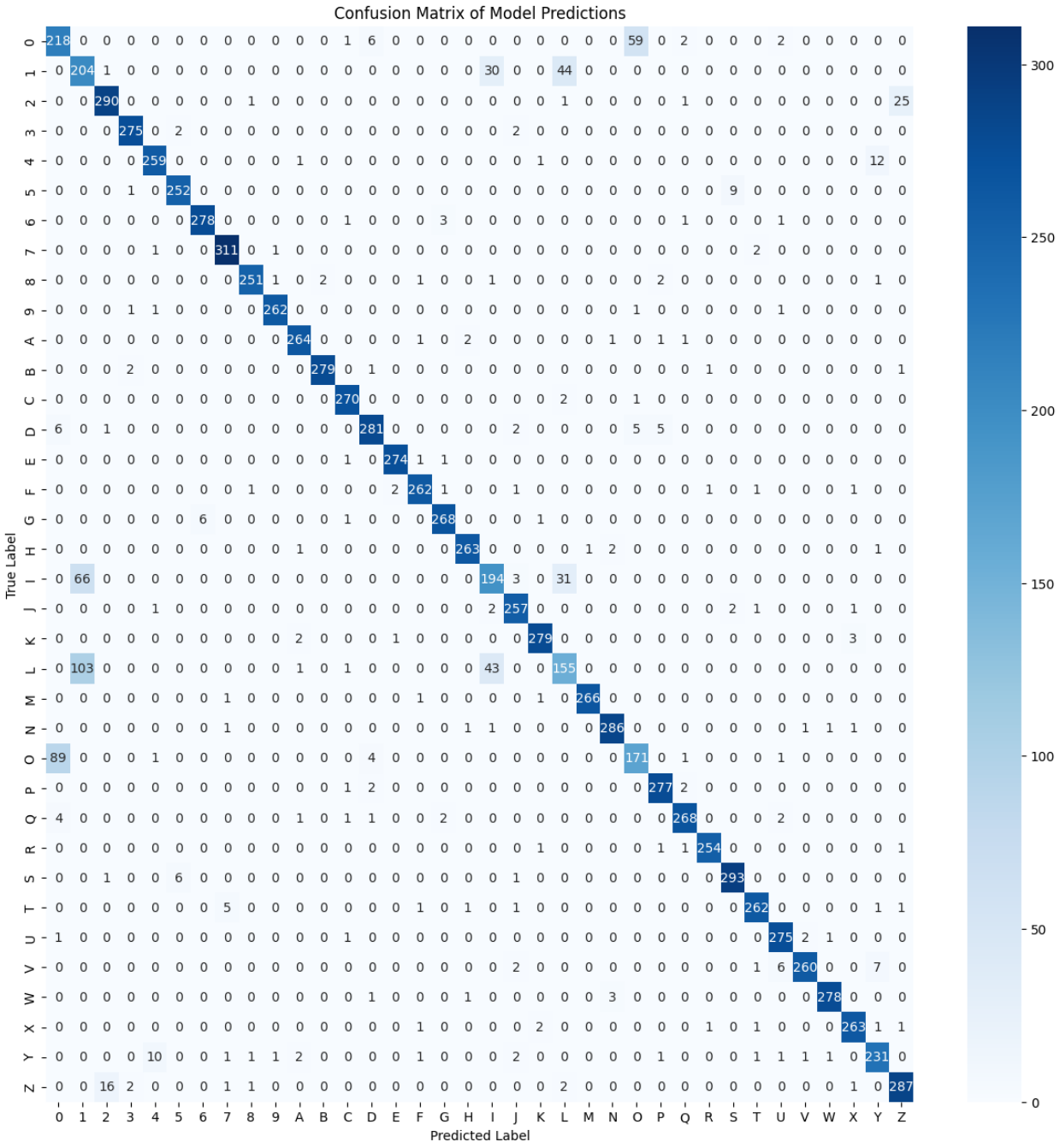
6. Provide graphs and analyze the results



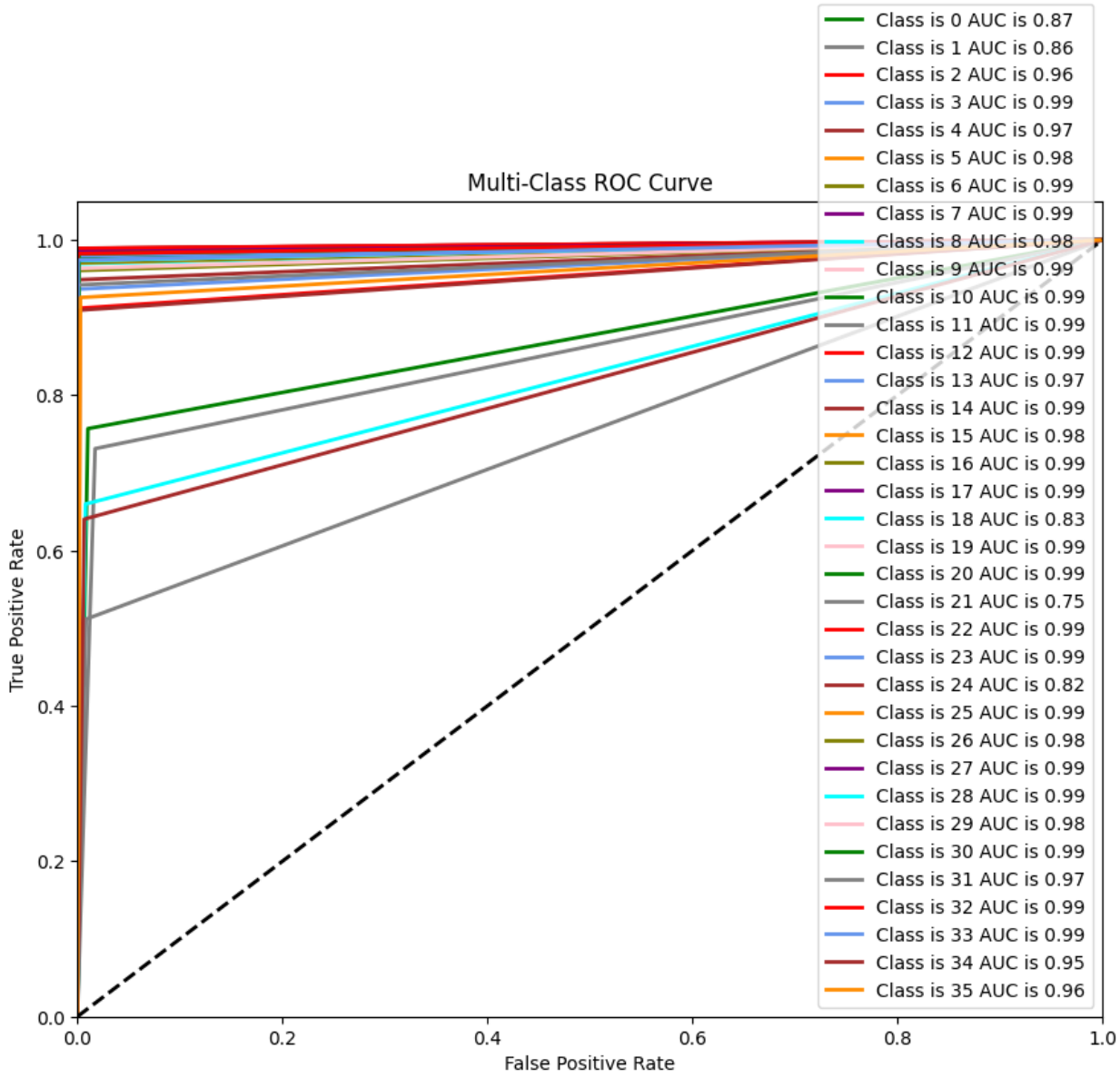
In the accuracy graph, my model quickly learns as evidenced by the sharp increase in training accuracy in the initial epochs. Both validation and test accuracies closely follow the training accuracy, which confirms that my model generalizes well. After about the sixth epoch, the accuracies level off, suggesting that the model has learned most of what it can from the data provided.

For the loss graph, there's a sharp decrease in training loss at the start, paralleling the accuracy improvements. The validation and test losses also decrease similarly and start to plateau around the sixth epoch, which aligns with the accuracy trends.

Overall, the close alignment between training, validation, and test lines in both graphs indicates a good fit; my model is neither overfitting nor underfitting. The plateau in both accuracy and loss suggests that my model has reached its learning capacity with the current setup.



The confusion matrix for my CNN model shows strong diagonal values, meaning most predictions are correct. However, there's notable confusion between classes with similar appearances, like 'I' and 'J' as well as 'O' and '0' (zero). Some classes have exceptionally good performance, while a few have lower than expected correct predictions, suggesting potential model weaknesses or data imbalance. To improve, I'd consider enhancing feature representation or increasing training data for the challenging classes.



Analyzing the ROC curve for my CNN model, I see that most classes have AUC values near 1.0, indicating my model is performing well at distinguishing these classes with high true positive rates and low false positives. However, certain classes like class 0, class 18, class 21, and class 24 have lower AUC values, which means my model struggles to differentiate these from the others. Overall, the high AUC values show my model's strong discriminative ability for most classes. To enhance my model's performance, I'd focus on the underperforming classes, perhaps by collecting more data, tweaking features, or applying targeted data augmentation.

Report for part 4

1. Provide the model details of your CNN

Input Layer: Accepts single-channel (grayscale) images. Spatial dimensions of the input are 64x64

Convolutional Layer 1:

- 64 filters
- Kernel size: 3x3
- Stride: 1
- Padding: 1
- Activation: ReLU
- Followed by a 2x2 Max Pooling (stride 2)

Convolutional Layer 2:

- 128 filters
- Kernel size: 3x3
- Stride: 1
- Padding: 1
- Activation: ReLU
- Followed by a 2x2 Max Pooling (stride 2)

Convolutional Layer 3:

- 256 filters
- Kernel size: 3x3
- Stride: 1
- Padding: 1
- Activation: ReLU
- Additional Convolutional Layer 3a:
    - Same configuration as above
- Followed by a 2x2 Max Pooling (stride 2)

Convolutional Layer 4:

- 512 filters
- Kernel size: 3x3
- Stride: 1
- Padding: 1
- Activation: ReLU
- Additional Convolutional Layer 4a:
    - Same configuration as above
- Followed by a 2x2 Max Pooling (stride 2)

Convolutional Layer 5:

- 512 filters
- Kernel size: 3x3
- Stride: 1
- Padding: 1
- Activation: ReLU
- Additional Convolutional Layer 5a:
- Same configuration as above
- Followed by a 2x2 Max Pooling (stride 2)

Flatten Layer: Flattens the output of the last pooling layer into a single long vector.

Fully Connected Layer 1:
- 512 * 2 * 2 neurons (assuming the input image size is large enough to produce this size of feature maps after all the pooling layers)
- Activation: ReLU
- Dropout

Fully Connected Layer 2:
- 4096 neurons
- Activation: ReLU
- Dropout

Output Layer (Fully Connected Layer 3):
- n_classes neurons (where n_classes is set to 36 by default)
- Softmax activation (applied during inference)

2. Discuss how the architecture is different from your CNN architectures defined in Part III.

Depth and Layer Stacking:
- My VGG11 is a deep model with multiple convolutional layers in each block, capturing more complex features.
- My CNN has single convolutional layers per block, making it less deep and simpler.

Batch Normalization:
- In my CNN, I use batch normalization after each convolutional layer to aid in training stability.
- My VGG11, adhering to version A in the paper given for reference, does not include batch normalization.

Fully Connected Layers:

- My VGG11 has three fully connected layers with two large layers of 4096 neurons each, designed for complex pattern learning.
- My CNN includes one fully connected layer with 256 neurons before the output layer, indicating a more straightforward approach.

Dropout Usage:

- Both models employ dropout, but my VGG11 has it after large dense layers due to a higher number of parameters.
- My CNN applies dropout after flattening the convolutional layers and before the output layer.

Activation Functions:

- Both models use ReLU activations, but my VGG11 specifically applies softmax during inference for class probability distribution and also so I can adhere to the version A of VGG11 in the paper given.

Basically, my VGG11 model has a more complex and deeper architecture compared to my simpler and more concise CNN, with specific differences in batch normalization, the number of neurons in fully connected layers, and dropout application.

3. Provide the performance metrics (Step 3) and compare them with the results obtained in Part III.

Performance metrics for my VGG11 model -
Epoch 1/15 completed, Train Loss: 1.8281, Train Acc: 45.61%, Val Loss: 2.8174, Val Acc: 85.50%, Test Loss: 2.8242, Test Acc: 85.22%
Epoch 2/15 completed, Train Loss: 0.3526, Train Acc: 87.32%, Val Loss: 2.7670, Val Acc: 89.67%, Test Loss: 2.7702, Test Acc: 89.37%
Epoch 3/15 completed, Train Loss: 0.2769, Train Acc: 89.93%, Val Loss: 2.7574, Val Acc: 90.03%, Test Loss: 2.7593, Test Acc: 90.19%
Epoch 4/15 completed, Train Loss: 0.2507, Train Acc: 90.67%, Val Loss: 2.7467, Val Acc: 90.93%, Test Loss: 2.7479, Test Acc: 90.86%
Epoch 5/15 completed, Train Loss: 0.2323, Train Acc: 91.28%, Val Loss: 2.7476, Val Acc: 90.99%, Test Loss: 2.7489, Test Acc: 91.09%
Epoch 6/15 completed, Train Loss: 0.1703, Train Acc: 93.03%, Val Loss: 2.7298, Val Acc: 91.86%, Test Loss: 2.7324, Test Acc: 91.60%
Epoch 7/15 completed, Train Loss: 0.1536, Train Acc: 93.66%, Val Loss: 2.7293, Val Acc: 92.07%, Test Loss: 2.7315, Test Acc: 91.75%
Epoch 8/15 completed, Train Loss: 0.1435, Train Acc: 93.98%, Val Loss: 2.7253, Val Acc: 92.45%, Test Loss: 2.7289, Test Acc: 91.99%

Epoch 9/15 completed, Train Loss: 0.1368, Train Acc: 94.10%, Val Loss: 2.7265, Val Acc: 92.18%, Test Loss: 2.7289, Test Acc: 91.98%
Epoch 10/15 completed, Train Loss: 0.1292, Train Acc: 94.37%, Val Loss: 2.7268, Val Acc: 92.07%, Test Loss: 2.7294, Test Acc: 91.96%
Epoch 11/15 completed, Train Loss: 0.1197, Train Acc: 94.80%, Val Loss: 2.7238, Val Acc: 92.29%, Test Loss: 2.7268, Test Acc: 92.02%
Epoch 12/15 completed, Train Loss: 0.1185, Train Acc: 94.82%, Val Loss: 2.7234, Val Acc: 92.39%, Test Loss: 2.7262, Test Acc: 92.11%
Epoch 13/15 completed, Train Loss: 0.1163, Train Acc: 94.96%, Val Loss: 2.7231, Val Acc: 92.27%, Test Loss: 2.7260, Test Acc: 92.06%
Epoch 14/15 completed, Train Loss: 0.1156, Train Acc: 94.93%, Val Loss: 2.7225, Val Acc: 92.30%, Test Loss: 2.7255, Test Acc: 92.14%
Epoch 15/15 completed, Train Loss: 0.1145, Train Acc: 94.98%, Val Loss: 2.7223, Val Acc: 92.28%, Test Loss: 2.7255, Test Acc: 92.05%
Total training time: 1644.84 seconds

The precision is 0.9183, the recall is 0.9173, the F1 Score is 0.9173

Performance metrics of my CNN model -
Epoch is 1/15, Train Loss is 0.5375, Train Acc is 84.43%, Val Loss is 0.2833, Val Acc is 89.74%, Test Loss is 0.2761, Test Acc is 90.33%
Epoch is 2/15, Train Loss is 0.2976, Train Acc is 89.28%, Val Loss is 0.2467, Val Acc is 90.74%, Test Loss is 0.2431, Test Acc is 90.89%
Epoch is 3/15, Train Loss is 0.2596, Train Acc is 90.39%, Val Loss is 0.2325, Val Acc is 91.44%, Test Loss is 0.2304, Test Acc is 91.27%
Epoch is 4/15, Train Loss is 0.2333, Train Acc is 91.19%, Val Loss is 0.2273, Val Acc is 91.35%, Test Loss is 0.2312, Test Acc is 91.03%
Epoch is 5/15, Train Loss is 0.2142, Train Acc is 91.78%, Val Loss is 0.2310, Val Acc is 91.28%, Test Loss is 0.2283, Test Acc is 91.44%
Epoch is 6/15, Train Loss is 0.1783, Train Acc is 92.91%, Val Loss is 0.2070, Val Acc is 92.27%, Test Loss is 0.2056, Test Acc is 92.35%
Epoch is 7/15, Train Loss is 0.1650, Train Acc is 93.46%, Val Loss is 0.2056, Val Acc is 92.33%, Test Loss is 0.2040, Test Acc is 92.37%
Epoch is 8/15, Train Loss is 0.1585, Train Acc is 93.74%, Val Loss is 0.2049, Val Acc is 92.51%, Test Loss is 0.2044, Test Acc is 92.37%
Epoch is 9/15, Train Loss is 0.1536, Train Acc is 93.86%, Val Loss is 0.2055, Val Acc is 92.35%, Test Loss is 0.2040, Test Acc is 92.49%
Epoch is 10/15, Train Loss is 0.1488, Train Acc is 94.02%, Val Loss is 0.2048, Val Acc is 92.32%, Test Loss is 0.2037, Test Acc is 92.27%
Epoch is 11/15, Train Loss is 0.1418, Train Acc is 94.36%, Val Loss is 0.2042, Val Acc is 92.44%, Test Loss is 0.2029, Test Acc is 92.37%

Epoch is 12/15, Train Loss is 0.1406, Train Acc is 94.40%, Val Loss is 0.2056, Val Acc is 92.39%, Test Loss is 0.2032, Test Acc is 92.39%
Epoch is 13/15, Train Loss is 0.1392, Train Acc is 94.51%, Val Loss is 0.2052, Val Acc is 92.43%, Test Loss is 0.2027, Test Acc is 92.38%
The Early stopping is being triggered.
Total training time is 965.42 seconds
The precision is 0.9258, the recall is 0.9243, the F1 Score is 0.9242

Comparison

For accuracy, my VGG11 model achieved its best test accuracy of 92.14% at epoch 14, while my CNN model outperformed it slightly with a best test accuracy of 92.49% at epoch 9.

In terms of precision, my VGG11 model has a precision of 0.9183, but my CNN model did even better with a precision of 0.9258. The recall follows a similar pattern; my VGG11 model has a recall of 0.9173, and my CNN model has a slightly higher recall of 0.9243.

Looking at the F1 Scores, which are very similar, my VGG11 model has an F1 Score of 0.9173 while my CNN model is just a touch higher at 0.9242.

The training time also reflects favorably on my CNN model. It took a total of 965.42 seconds to train, which is significantly less than the 1644.84 seconds my VGG11 model took. This efficiency is essential for me, especially when working with limited computational resources or under tight time constraints.

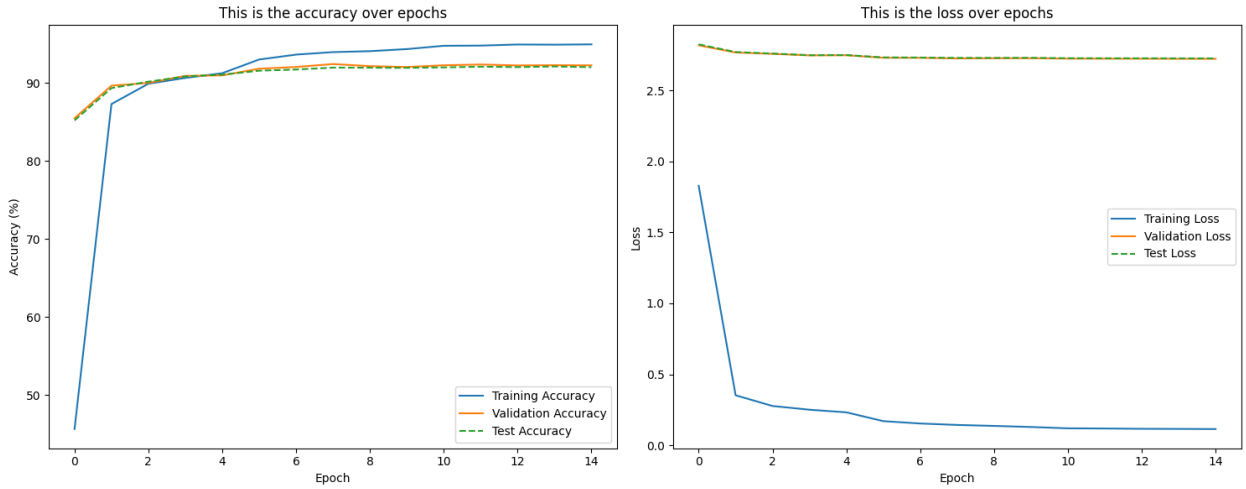
Moreover, my CNN model triggered early stopping, which is a feature I value as it prevents overfitting by halting the training when no further improvement is observed.

In conclusion, my CNN model slightly outperforms my VGG11 model across precision, recall, and F1 Score, and does so in a shorter amount of training time, suggesting it is more efficient. Although the best test accuracy of my CNN model is only marginally higher than that of my VGG11, it reached its peak performance faster, which indicates its effective generalization from the training data to unseen data. However, both models perform at a high level, and the differences are quite narrow. My choice between the two would depend on factors like computational resources, training time available, and the specific application needs.

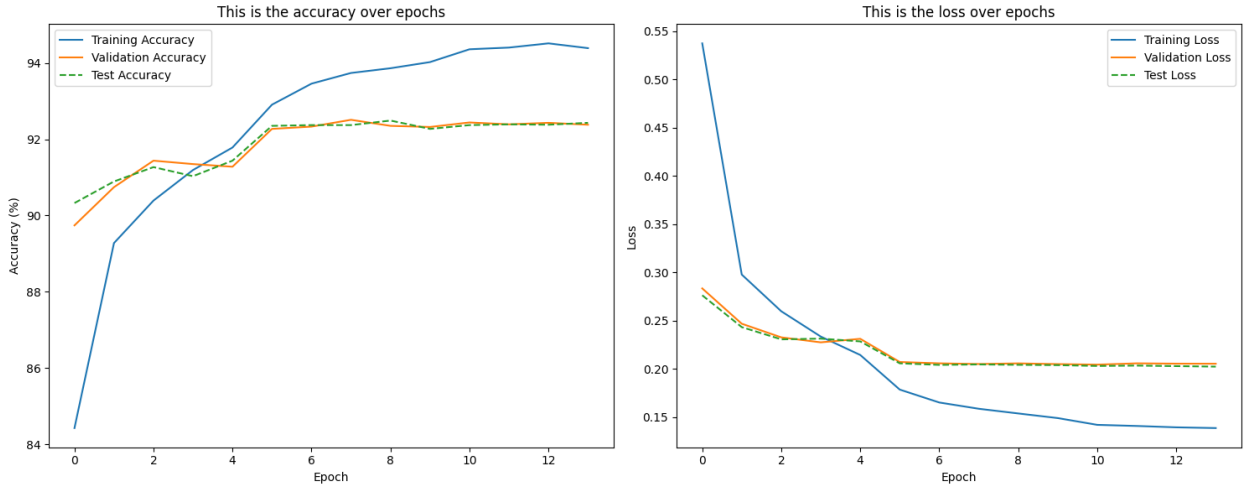
4. Provide graphs (Step 4) and compare them with the results obtained in Part III.

Graphs for accuracy and loss over epochs for both models

VGG11



CNN



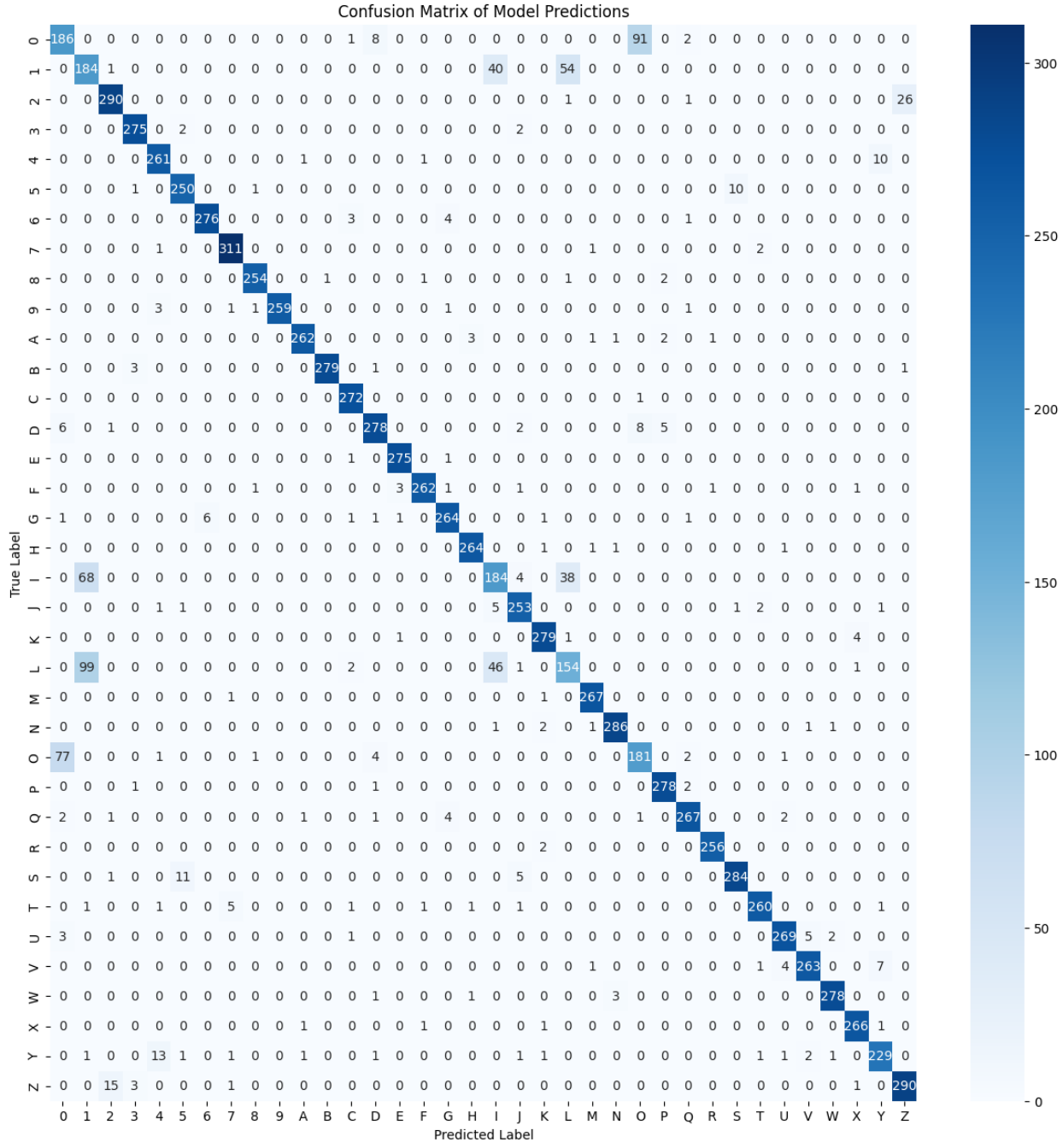
Looking at the graphs for my VGG11 model, I see that the training accuracy starts off high and stabilizes quickly, which indicates that my model converges fast. The validation accuracy remains close to the training accuracy, which shows that my model generalizes well. The test accuracy is slightly lower than the validation accuracy but stays consistent, which is satisfactory. The training loss decreases sharply and flattens, suggesting my model learns quickly and doesn't overfit, as the validation and test loss graphs also flatten out.

For my CNN model, the training accuracy shows a steady increase, reflecting a more gradual learning curve. The validation accuracy keeps pace with the training accuracy, which means my model generalizes well over time. The test accuracy is very close to the validation accuracy, showing little to no overfitting. The loss for my CNN model shows a steep decline initially and then levels off, similar to my VGG11 model, indicating effective learning without overfitting.

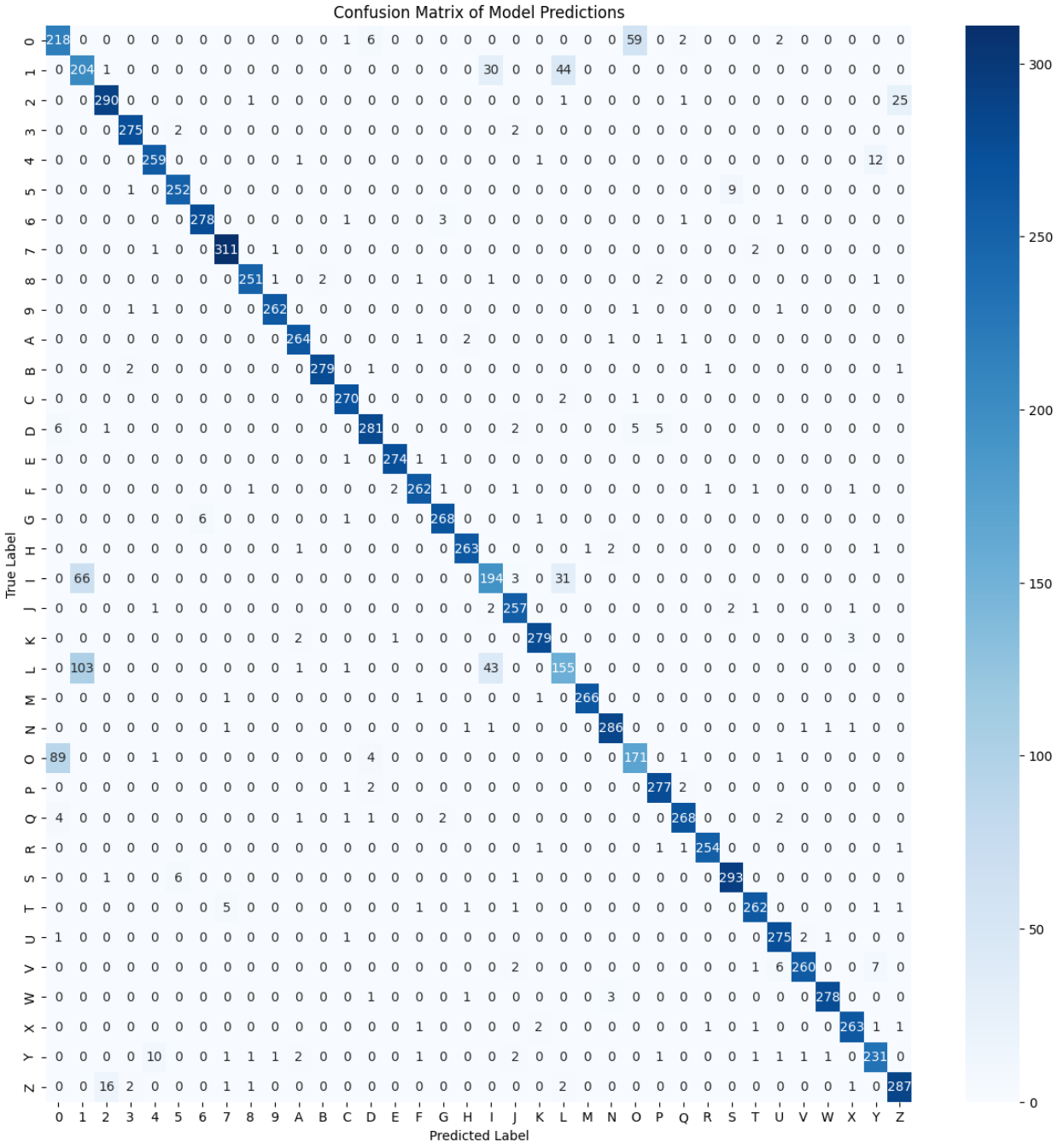
Comparing the two, my CNN model appears to learn more gradually, which could be beneficial for long-term generalization. Both models exhibit excellent performance, with good convergence and generalization capabilities as seen in both the accuracy and loss graphs. However, the more progressive learning of my CNN model might give it an edge in adapting to new, unseen data.

Confusion Matrix for both models

VGG11



CNN

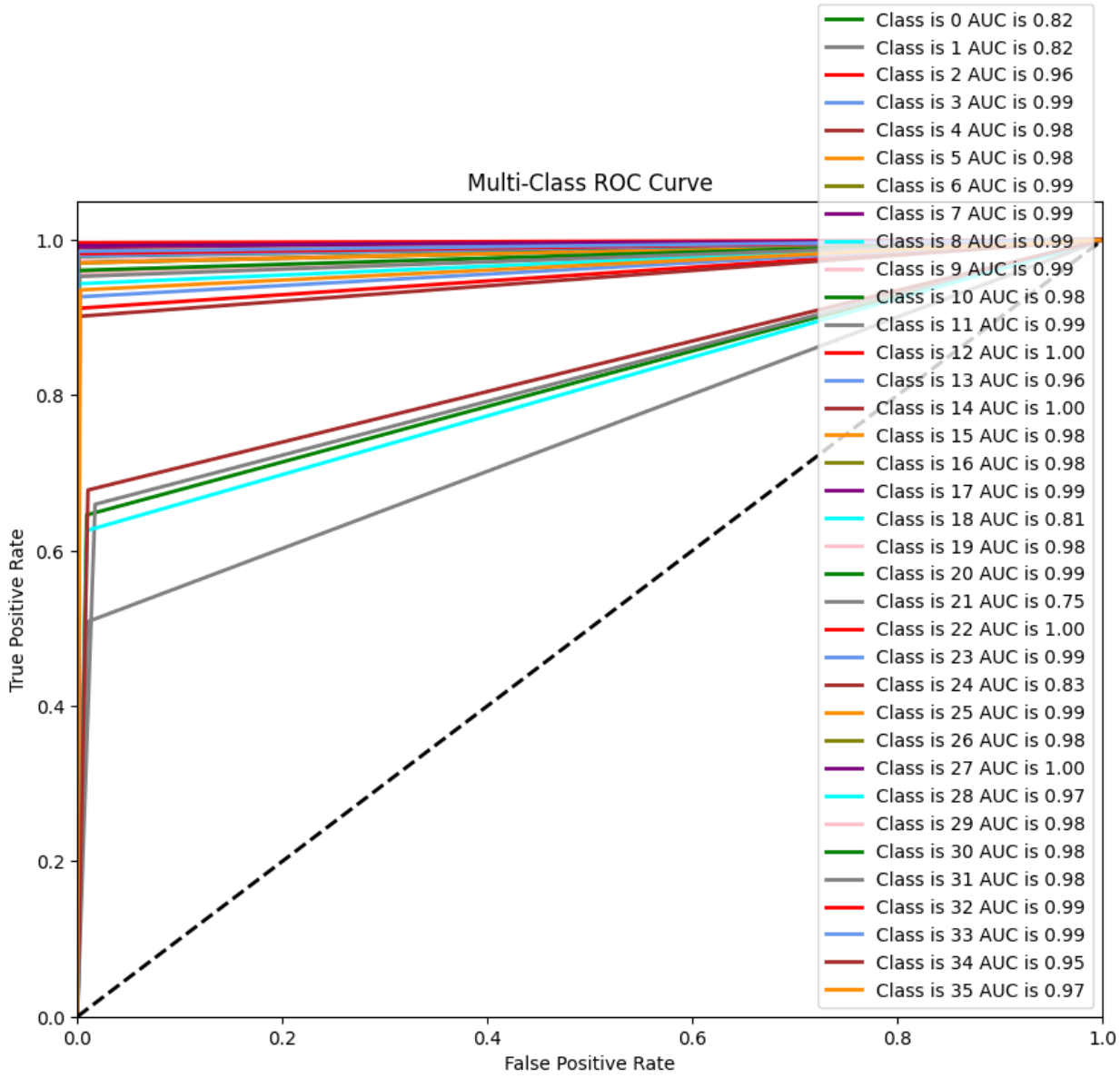


Examining the confusion matrices for my VGG11 and CNN models, I see that the main diagonal, which indicates correct predictions, is strongly highlighted in both. This shows a high true positive rate across most classes. However, there are some misclassifications, as seen in the non-zero values off the diagonal, but these numbers are relatively small. In the VGG11 model, I notice that certain classes like 'I' and 'L' have higher rates of confusion, likely due to similarities in their features.

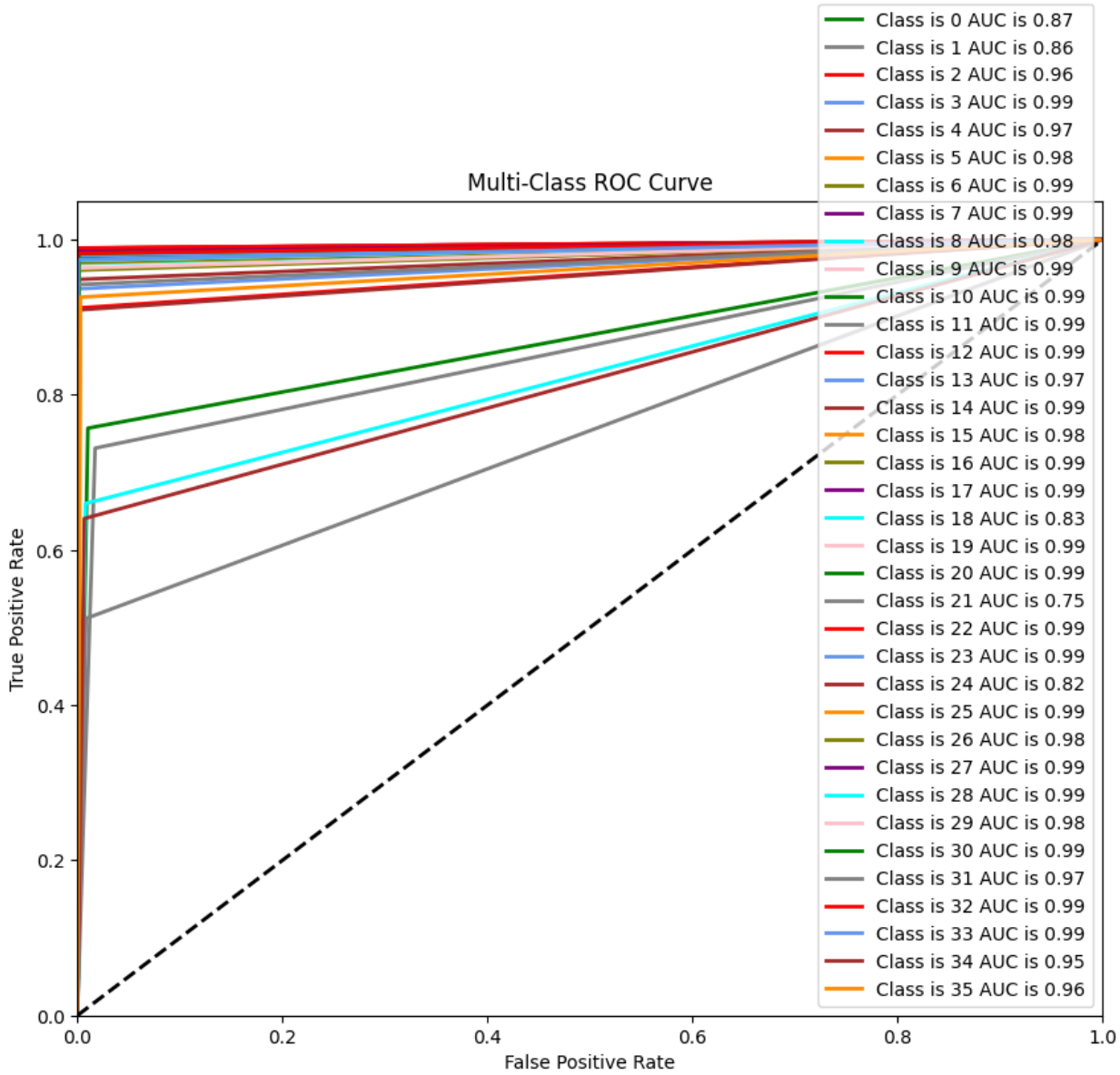
Looking at the CNN model's matrix, the diagonal is similarly prominent, suggesting a strong true positive rate that is comparable to the VGG11. There's a presence of misclassifications, but they are minimal, indicating that the CNN model has good discriminative power between classes. I observe that, like the VGG11 model, there are a few classes with more confusion, but this appears to be less significant in the CNN model.

When I compare both models directly, it's evident that they both perform well, with high rates of correct classifications. The VGG11 model has a slight tendency for more confusion between certain classes compared to the CNN model. From this, I conclude that the CNN model may have a marginal advantage in distinguishing between classes that are visually similar, which is consistent with my previous analysis of the accuracy and loss graphs, where the CNN model showed a slight edge. Overall, while both models exhibit excellent performance, my CNN model seems to have a slightly cleaner confusion matrix, indicating potentially better classification performance.

Multiclass ROC curve for both models
VGG11



CNN



Having a look at the Receiver Operating Characteristic (ROC) curves for my VGG11 and CNN models, I can evaluate their performance in classifying multiple classes.

In the ROC curve of my VGG11 model, the Area Under the Curve (AUC) scores for most classes are very high, mostly above 0.90, which implies an excellent measure of separability. This means that my VGG11 model has a high capability to distinguish between classes for most cases. However, there are a couple of classes with AUC scores around 0.75 to 0.82, suggesting that there is room for improvement in these specific areas.

For my CNN model, the AUC scores are also predominantly high, indicating a strong discriminative ability similar to the VGG11 model. Again, a few classes have lower AUC scores, but the lowest is 0.75, which is the same as my VGG11 model's lowest score. Interestingly, the AUC scores for the classes with the lowest performance in the VGG11 model are slightly higher in the CNN model, suggesting that it may be more robust in differentiating those particular classes.

Comparing the two, it's clear that both my models are performing well. The CNN model shows a slight improvement in the AUC scores for the classes where the VGG11 was weaker, which might indicate that the CNN model is better at handling the variability within those classes. Overall, the performance of both models is commendable, but the CNN model seems to have an edge in distinguishing between classes that the VGG11 model finds more challenging.

References for part 3 and 4

1. Python os documentation
2. Python zipfile documentation
3. matplotlib.pyplot documentation
4. Python random documentation
5. Pillow (PIL Fork) documentation
6. NumPy documentation
7. PyTorch documentation
8. torchvision.transforms documentation
9. Scikit-learn metrics documentation
10. Seaborn documentation
11. Scikit-learn preprocessing documentation
12. Python itertools documentation