ELL784: Introduction to Machine Learning

Shashank Shaurya Dubey (2022SMZ8158)

Assignment II

In real-world scenarios, we often know very little about the actual test data, making it crucial to ensure that any accuracy achieved is not merely by chance. Therefore, we will apply techniques such as hyperparameter tuning and validation, as discussed in class, to ensure the robustness of our model. Here are the detailed tasks:

Question 1

Train your ANN for the classification of handwritten images in the MNIST dataset $(0-9)$. Choose the number of layers and neurons accordingly. Plot and report the loss functions and accuracies for training, validation, and the test set. Use methods discussed in class (tuning and validation) to tune hyperparameters. Use one hot encoding for the class labels.

Proof. We use a feedforward ANN built on the following logic to classify the handwritten images in the MNIST data set (0-9).

1. Importing Libraries and loading the MNIST data set

- **Importing libraries:** We import *numpy*(*np*) to support multidimensional arrays and matrices, with high-level mathematical functions to operate on these arrays. *keras.datasets.mnist* contains the MNIST dataset and *keras.utils.to_categorical* converts class vectors (integers) to binary class matrices, used as an input to neural networks for classification tasks.

- **Loading the MNIST dataset:** We load the dataset into four NumPy arrays X_train for training images, y_train for corresponding training labels, X_test for test images, and y_test for corresponding test labels.

2. Normalizing the pixel values
Each image in the MNIST is composed of 256 pixels. Performing computations on such a wide range (0-255) can lead to numerical instability during gradient descent updates due to large differences in magnitude. To address this, we normalize the pixel values to a range of 0-1. This brings all pixel values to a similar scale, rendering numerical stability, improving convergence speed, and reducing code runtime.

3. Re-shaping the data

- **Input data:** Next, we flatten the 28x28 image matrices into 784-dimensional vectors to prepare them for input to the neural network. This reshaping transforms the two-dimensional image data into a single, one-dimensional sequence, preserving the pixel information while making it compatible with the network's architecture.

- **Output data:** We use *one-hot encoding* to convert each digit into a 10-dimensional vector. This representation assigns a unique 10-dimensional pattern to each digit, where only one element is 1 and the others are 0. For example, the digit $'5'$ would be represented as a vector with a 1 in the $6th$ position and 0s in all other positions. This encoding allows the neural network to easily distinguish between different digits and learn their unique patterns.

4. Hyperparameter tuning

- **Tuning:** It is the process of selecting an optimal set of parameters for model training. These are not learned from the data during training but are externally set before training begins. These include the number of layers, layer size, learning rate, batch size, and dropout nodes.

- **Validation:** It involves partitioning a portion of the training data for evaluation during training, rather than for model fitting. By monitoring the model's performance on the validation set during training, we can identify signs of overfitting, such as increasing accuracy on the training set but decreasing accuracy on the validation set. This allows us to adjust the model architecture to improve generalization.

Cases for hyperparameter tuning

Learning rate η = 0.001, Epochs = 15, and Batch size (of training data)= 256
Case I: Number of layers (L) = 1, Layer size = 64, 128, 256, and 512

- Increasing accuracy: As we increased the layer size, from 1 to 4, the accuracy generally increased for all three sets of training, validation, and test dataset. This indicated that the model was learning and improving as more neurons were added to the layer. However, we stopped at 512 because in the previous assignment we found 512 to give the most optimal test accuracy.

- Decreasing losses: We also observed that the model losses consistently dropped on all three datasets - training, validation, and test. This suggested that the model is training well and consistently improving performance as was observed with model accuracy.

- The following are the plots for the accuracy and loss values for training, validation, and test data for different layer sizes.

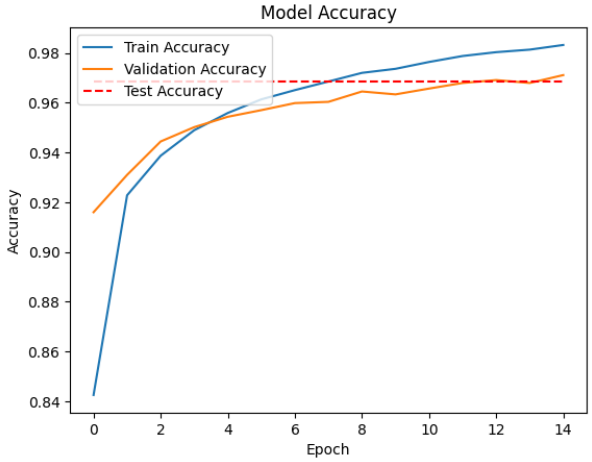

Figure 1: Accuracy plot for L = 1, N = 64

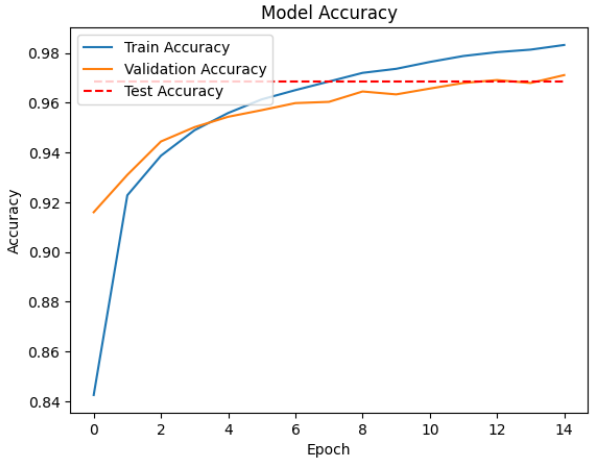

Figure 2: Loss plot for L = 1, N = 64

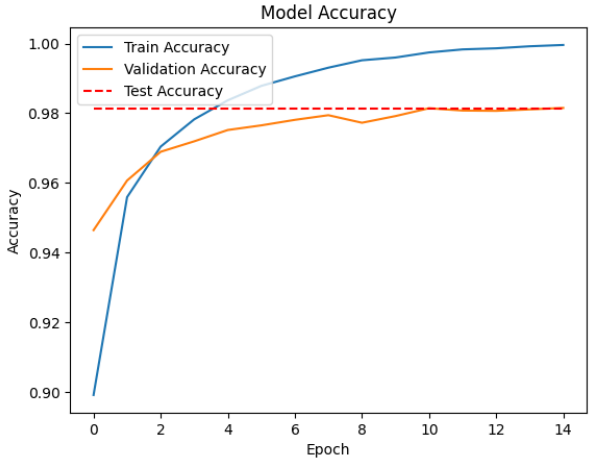

Figure 3: Accuracy plot for L = 1, N = 512

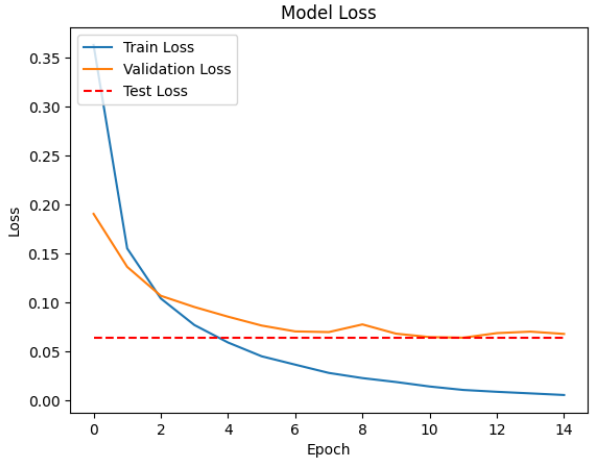

Figure 4: Loss plot for L = 1, N = 512

Case II: L = 2, Layer size = 64, 128, 256, 512

- Trends in losses: As the layer size increased from 64 to 512, the training loss consistently decreased, indicating that the model learnt more complex patterns. However, beyond the N = 128 layer size, the validation and test losses shot up indicating overfitting for larger layer sizes.

- Increasing accuracy: Training accuracy increased consistently with larger layer sizes. Validation and test accuracy show similar trends, with some fluctuations. The test accuracy increased from 0.9686 (for N = 64) to 0.9813 (for N = 512).

- Interpretation: We observe that while the validation and test accuracy increase consistently as layer size increases, so does the gap between the training and validation and test accuracy, indicating that the model is overfitting. Thus, while training accuracy

rises from 0.9898 (N = 64) to 0.9996 (N = 512), the validation accuracy rises from 0.9697 (N = 64) to 0.9828 (N = 512), and the test accuracy increases only marginally from 0.9732 (N = 64) to 0.9819 (N = 512).

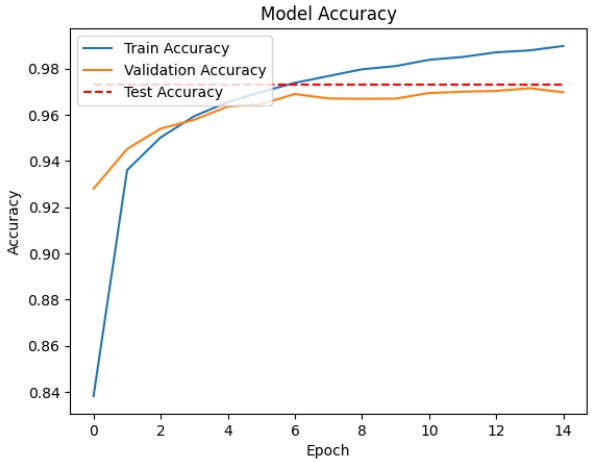

Figure 5: Accuracy plot for L = 2, N = 64

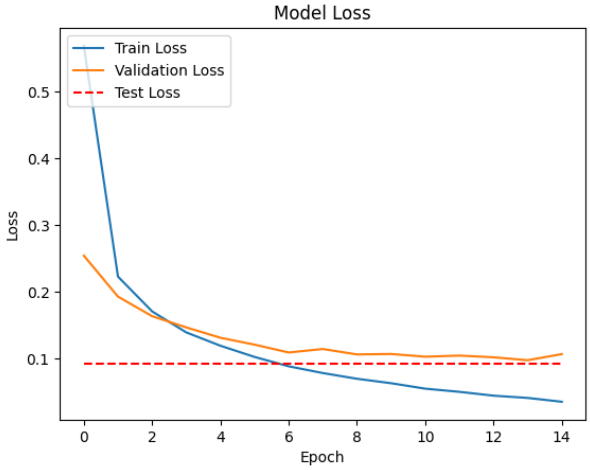

Figure 6: Accuracy plot for L = 2, N = 64

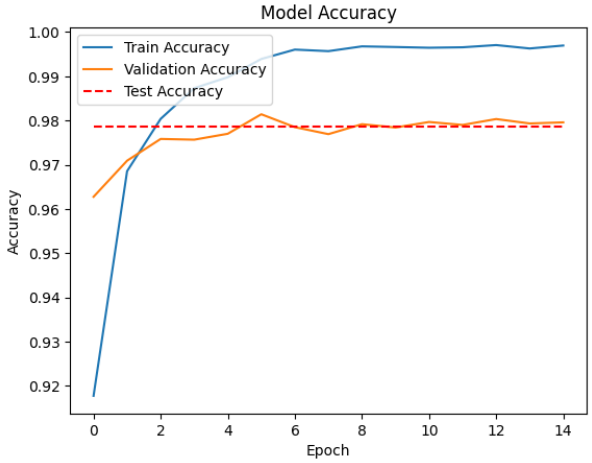

Figure 7: Accuracy plot for L = 2, N = 512

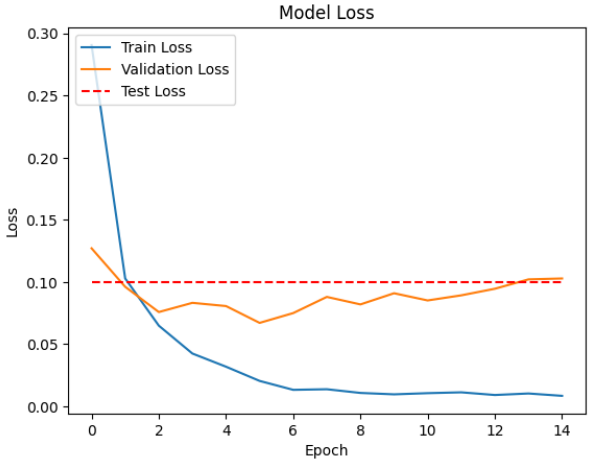

Figure 8: Loss plot for L = 2, N = 512

Case III: Layers = 3, Layer size = 64, 128, 256, 512

- Losses: The training loss decreases as the layer size increases, indicating that the network fits the training data better with larger layer sizes. The validation loss fluctuates slightly, but there's no clear trend of improvement or deterioration with increasing layer size. The test loss follows a similar trend to the validation loss. The lowest test loss was at N = 128 neurons, which might indicate the most optimal generalization for the model size.

- Accuracy: The training accuracy was very high for all models, and only marginally improves as N increases. Validation accuracy improves slightly as the layer size increases,

going from 0.973 (N = 64) to 0.9791 (N = 512). The test accuracy also improves slightly with the number of neurons, from 0.9703 (N = 64) to 0.9801 (N = 512). The trend implies that larger models might generalize better, but the improvement is marginal.

- Interpretation: Models with more neurons per layer (128, 256, 512) tend to have slightly better performance on validation and test sets. While the training loss drops with more neurons, the validation and test losses do not follow the same trend, hinting at potential overfitting in the larger models. The model with N = 128, seems to strike a good balance between training accuracy and generalization (validation and test accuracy) and has the lowest test loss, making it the most optimal choice.

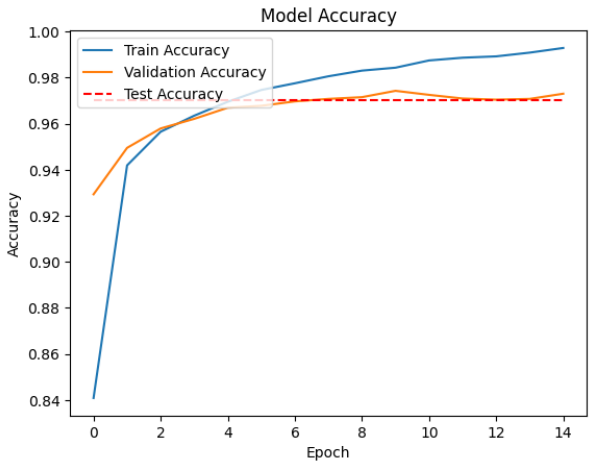

Figure 9: Accuracy plot for L = 3, N = 64

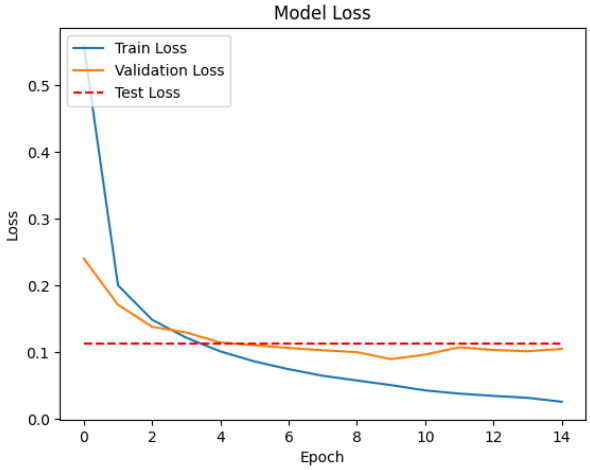

Figure 10: Loss plot for L = 3, N = 64

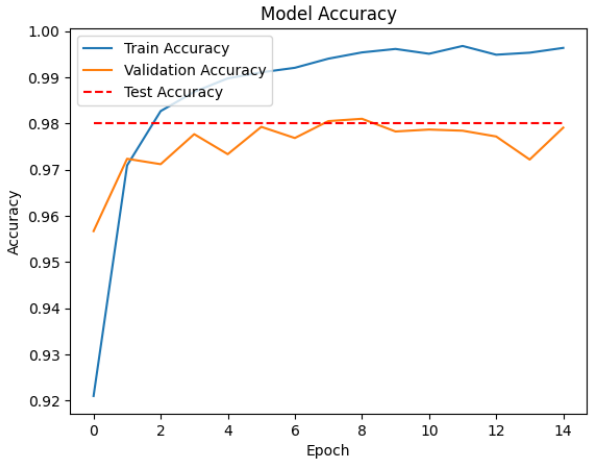

Figure 11: Accuracy plot for L = 3, N = 512

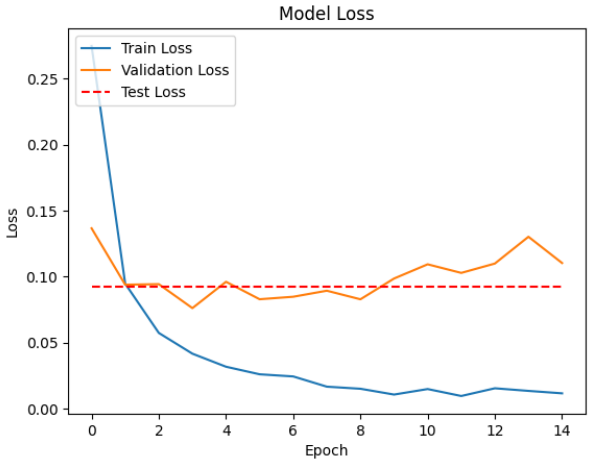

Figure 12: Loss plot for L = 3, N = 512

In summary

We conducted four experiments with different model parameters to evaluate their performance on the MNIST dataset. The results have been summarised in this table.

Layers (L)	Loss/ Accuracy	Dataset	Layer Size (N)			
			64	128	256	512
1	Loss	Train	0.0617	0.0307	0.0145	0.0057
		Validate	0.104	0.0818	0.0736	0.068
		Test	0.1013	0.0773	0.0678	0.0644
	Accuracy	Train	0.9832	0.9928	0.9975	0.9996
		Validate	0.9711	0.9753	0.978	0.9815
		Test	0.9686	0.9764	0.9791	0.9813
2	Loss	Train	0.0358	0.0108	0.0059	0.0019
		Validate	0.1071	0.08	0.0851	0.088
		Test	0.0934	0.0796	0.088	0.0873
	Accuracy	Train	0.9898	0.9974	0.9983	0.9996
		Validate	0.9697	0.9788	0.979	0.9828
		Test	0.9732	0.9784	0.98	0.9819
3	Loss	Train	0.0253	0.013	0.0149	0.0117
		Validate	0.1046	0.092	0.1054	0.1103
		Test	0.112	0.0878	0.1007	0.0925
	Accuracy	Train	0.9929	0.9959	0.9952	0.9964
		Validate	0.973	0.9771	0.9765	0.9791
		Test	0.9703	0.9772	0.9774	0.9801
4	Loss	Train	0.0358	0.0108	0.0059	0.0019
		Validate	0.1071	0.08	0.0851	0.088
		Test	0.0934	0.0796	0.088	0.0873
	Accuracy	Train	0.9898	0.9974	0.9983	0.9996
		Validate	0.9697	0.9788	0.979	0.9828
		Test	0.9732	0.9784	0.98	0.9819

Figure 13: Summary of results

Though test accuracy was the highest with N = 512 layer size, we noticed losses increase with this layer size. We achieved the highest test accuracy of 98.19 percent at Layers = 2 and 4 and Layer size = 512. This combination of parameters led to the highest test accuracy and lowest losses, suggesting that it is the most optimal model architecture that effectively balances model training efficiency and generalization. □

Question 2

Consider the same MNIST dataset to build a tree-type growing neural network. Each node learns a one-layer neural network that tries to classify samples correctly. Child nodes are added to correct for misclassified samples. Plot the loss functions and accuracies for training, validation, and testing as the network grows. (Bonus points if you are able to achieve 100% accuracy).

Proof. The Neural Tree Networks (NTN) method uses neural networks in a tree structure to recursively partition feature spaces for classification[1]. Unlike MLPs, which require predefined architectures, NTNs grow adaptively during learning, enhancing classification performance and offering a favorable balance between classification speed and hardware implementation compared to MLPs[2].

Building a growing NTN

1. Root Node: We begin by constructing a single layer neural network. This root network is trained on the entire MNIST dataset to form the foundation of the model.

2. Misclassified Samples: After the first round of training, we examine the predictions to identify which samples in the training set were misclassified.

3. Child Nodes: To check for these errors, we create child nodes for the misclassified samples. Each child node is another neural network trained specifically on these misclassified samples. This targeted training aims to correct the classification errors made by the root node.

4. Iterate to add Child Nodes: We continue this process iteratively, where we assess each level of the tree, identify any misclassified samples, and add child nodes to rectify these errors. This iterative refinement helps the network to better capture complex patterns in the data.

5. Stopping criterion: To prevent the model from being unbounded, we decide to stop when all training samples are correctly classified, ensuring that no further improvement is possible. Alternatively, we stop if the tree reaches a maximum depth, preventing overfitting and keeping the model manageable.

6. Model evaluation: Once the tree is fully grown, we evaluate its performance on the training, validation, and test datasets. We track the loss and accuracy metrics at each step of the network's growth and plot these to visualize how the model's performance improves over time.

Case 1: Root Node – Layer size = 512, Epochs = 10, and Batch size = 128

We start with a root node comprising a single-layer neural network with a layer size (N) = 512. The root node parameters were selected on account of providing the most optimum results for a single-layer ANN. For this root node, we test different parameters for child nodes

[1]Sankar (1993): Growing and Pruning Neural Tree Networks
[2]Sankar (1993): Growing and Pruning Neural Tree Networks

to find the most optimum NTN model.

Case 1	**Root Node**	**Child Nodes**				
	1	**1.1**	**1.2**	**1.3**	**1.4**	**1.5**
Layer size	512					
Epochs	10	10	20	20	20	20
LR	0.01	0.0005	0.0005	0.0005	0.001	0.0005
Batch size	128	64	64	64	64	128
Max depth	4	4	4	6	4	4
Test set	0.2	0.2	0.2	0.2	0.2	0.3
Case 2	**Root Node**	**Child Nodes**				
	2	**2.1**	**2.2**	**2.3**	**2.4**	**2.5**
Layer size	512					
Epochs	15	20	20	20	20	20
LR	0.001	0.0005	0.0005	0.0001	0.0001	0.0001
Batch size	256	64	64	64	128	128
Max depth	4	4	6	6	6	6
Test set	0.2	0.2	0.2	0.2	0.2	0.3

Figure 14: Case parameters

1.1 Child Nodes: Epochs = 10, LR = 0.0005, Batch size = 64, and Max depth = 4

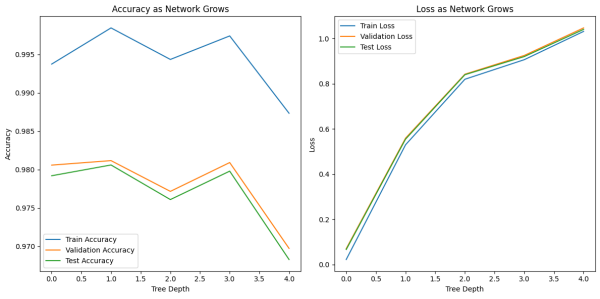

Figure 15: Accuracy and loss plot for 1.1

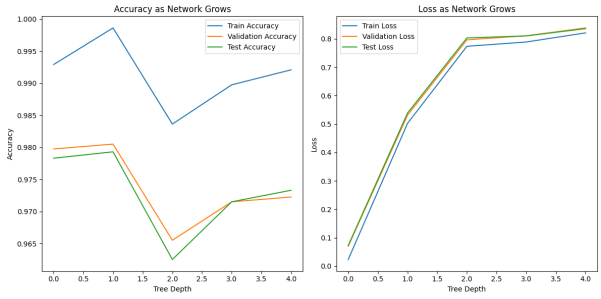

Figure 16: Accuracy and loss plot for 1.2

1.2 Child Nodes: Epochs = 20, LR = 0.0005, Batch size = 64, and Max depth = 4
1.3 Child Nodes: Epochs = 20, LR = 0.0005, Batch size = 64, and Max depth = 6

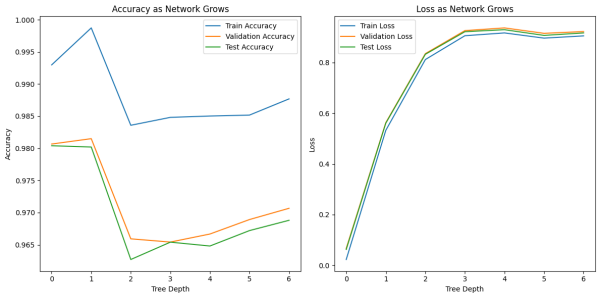

Figure 17: Accuracy and loss plot for 1.3

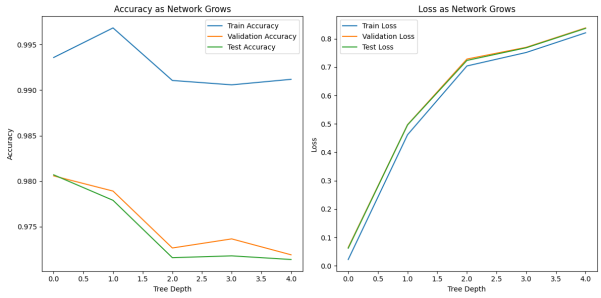

Figure 18: Accuracy and loss plot for 1.5

1.4 Child Nodes: Epochs = 20, LR = 0.001, Batch size = 64, and Max depth = 4
The test results showed a drop in training, validation, and test accuracies. This might be

because of the increased learning rate for the child nodes. Because of the smaller sample size they may not have been properly trained on the misclassified samples. Therefore, we rule out increasing the learning rate for our tuning framework.
1.5 Child Nodes: Epochs = 20, LR = 0.0005, Batch size = 128, and Max depth = 4

Case 2: Root Node – Layer size = 512, Epochs = 15, and Batch size = 256

2.1 Child Nodes: Epochs = 20, LR = 0.0005, Batch size = 64, and Max depth = 4

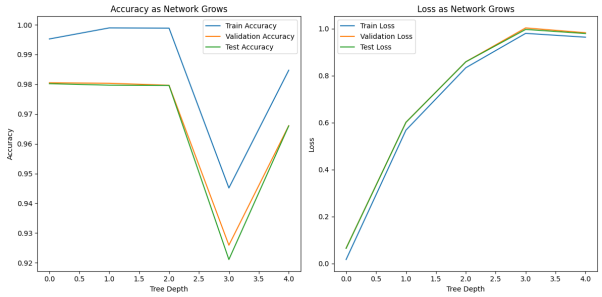

Figure 19: Accuracy and loss plot for 2.1

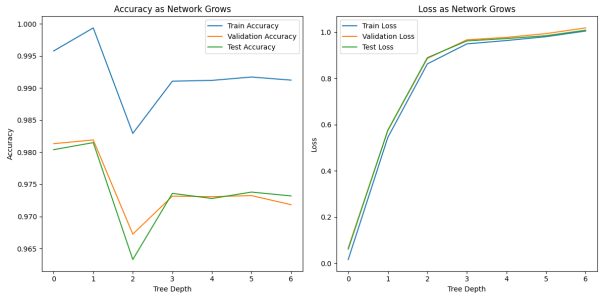

Figure 20: Accuracy and loss plot for 2.2

2.2 Child Nodes: Epochs = 20, LR = 0.0005, Batch size = 64, and Max depth = 6
2.3 Child Nodes: Epochs = 20, LR = 0.0001, Batch size = 64, and Max depth = 6
On dropping the learning rate the model demonstrates superior performance on all parameters. While the training accuracy increases, the validation and test accuracy becomes more consistent and the fluctuation seen in previous cases disappears. However, the gap between the training and validation/test accuracy widens substantially.

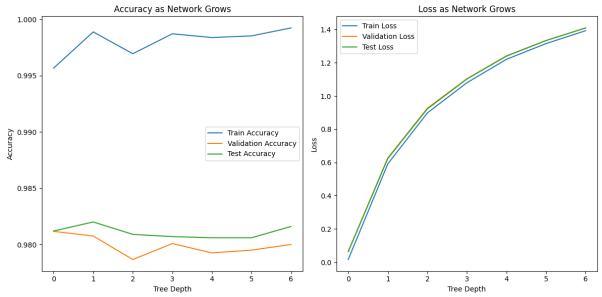

Figure 21: Accuracy and loss plot for 2.3

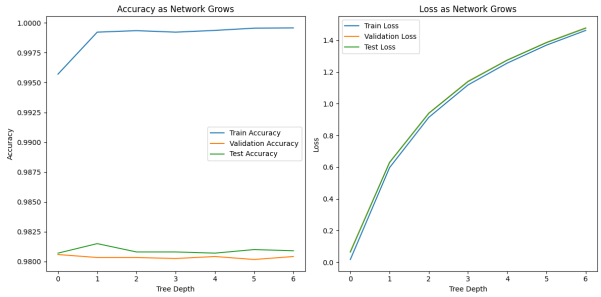

Figure 22: Accuracy and loss plot for 2.4

2.4 Child Nodes: Epochs = 20, LR = 0.0001, Batch size = 128, and Max depth = 6
In this case, we keep other parameters constant but increase the batch size for training of child nodes. We expect a bigger batch size to train the model better, so that the validation and test accuracy rises.
2.5 Child Nodes: Epochs = 20, LR = 0.0001, Batch size = 128, and Max depth = 6; Test data size = 0.3 (from 0.2)

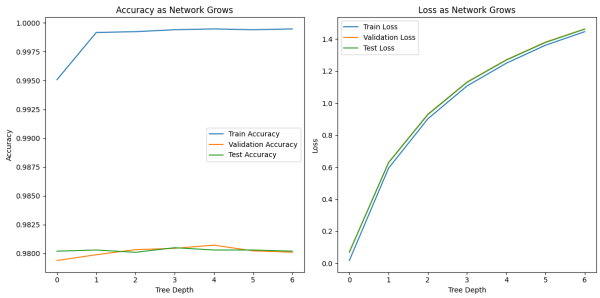

Figure 23: Accuracy and loss plot for 2.5

Both training and validation accuracy increased, however, the test accuracy did not dropped marginally. This time we increased the test size from 0.2 to 0.3. This will allow the model to test on a wider sample which we expect to improve classification. Wider Sampling: A larger test set provides a more comprehensive evaluation of the model's performance on unseen data. Improved Generalization: By testing on a wider range of samples, you can better assess the model's ability to generalize to new data.

Root Node	**Child Node**	**Misclassified samples with tree depth**					
		1	**2**	**3**	**4**	**5**	**6**
Case 1	Case 1.1	133	75	272	125	×	×
	Case 1.2	145	66	786	492	×	×
	Case 1.3	83	61	788	729	719	712
	Case 1.4	102	139	2217	878	×	×
	Case 1.5	94	152	429	452	×	×
Case 2	Case 2.1	59	51	55	2633	×	×
	Case 2.2	57	31	819	429	423	398
	Case 2.3	71	54	147	62	×	×
	Case 2.4	46	37	31	37	30	21
	Case 2.5	46	35	32	25	22	25

Figure 24: Misclassified samples with tree depth

Root Node	Child Node	Accuracy		
		Training	Validation	Test
Case 1	Case 1.1	0.9683	0.9543	0.9498
	Case 1.2	0.9921	0.9722	0.9733
	Case 1.3	0.9877	0.9707	0.9688
	Case 1.4	0.9791	0.9615	0.9596
	Case 1.5	0.9912	0.9719	0.9714
Case 2	Case 2.1	0.9847	0.9661	0.966
	Case 2.2	0.9912	0.9718	0.9732
	Case 2.3	0.9992	0.98	0.9816
	Case 2.4	0.9996	0.9804	0.9809
	Case 2.5	0.9995	0.9801	0.9802

Figure 25: Summary of test accuracies for different cases

Question 3

Suggest ways of determining when to stop training so that generalization is improved.

Proof. Early stopping is a technique used to prevent overfitting in machine learning models, including tree-type growing neural networks (NTNs). Here are several measures that can be employed:

- Validation set monitoring: If the validation loss begins to increase or the validation accuracy stops improving or starts to drop, after a certain number of epochs, the model stops training.
- Regularization: We prune child nodes that don't contribute significantly to the model's performance which reduces model complexity. We can also apply weight decay to the neural networks within the nodes to penalize large weights, which can help prevent overfitting.
- Performance threshold: Set a target accuracy threshold. If the validation accuracy reaches or exceeds this threshold, the model stops training.
- Misclassification threshold: Training can be stopped if the number of misclassified samples drops below a threshold. This will check the likelihood of model overfitting.
- Keras: Use built-in early stopping callbacks to automatically stop training when a metric plateaus or worsens.

Example. Case: Validation accuracy threshold of 99.00 percent
We plot and show how the accuracy and loss plots are affected by putting a threshold. We take *Case 2.5* from the *Question 2*.
Model parameters are as follows: Root Node – Layer size = 512, Epochs = 15, and Batch size = 256; Child Nodes: Epochs = 20, LR = 0.0001, Batch size = 128, and Max depth = 6; Test data size = 0.3 (from 0.2) With the threshold we observe the training cycle stopping at

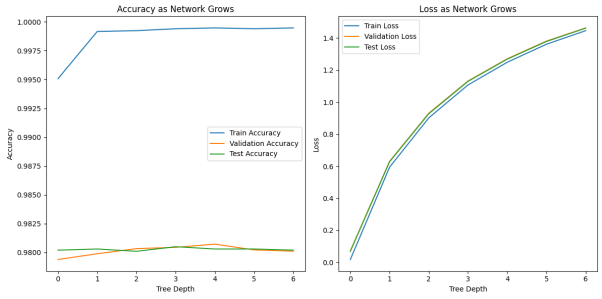

Figure 26: Accuracy and loss plot without threshold

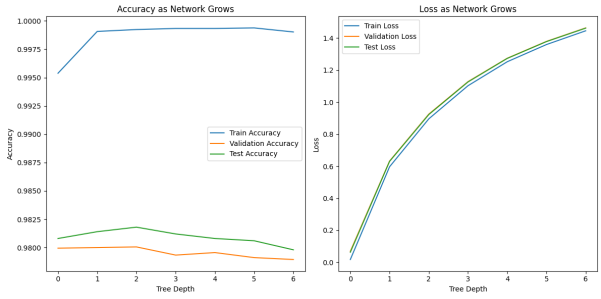

Figure 27: Accuracy and loss plot with threshold

Depth = 5 instead of Depth = 6. This resulted in a marginal improvement in test accuracy from 0.9802 to 0.9816.

□

Question 4

Bonus points for innovative ways of handling class imbalance at later tree nodes.

Proof. **Solution 1:** Class weights

- We compute class weights for each class, and assign higher weights to under-represented classes and lower weights to over-represented classes.
- The calculated weights are stored in a dictionary ($class_weight_dict$) where the key is the class index and the value is the corresponding weight.
- The optimizer assigns the calculated weights to each sample based on its class during loss calculation.
- Samples from under-represented classes will have a higher impact on the loss, encouraging the model to focus on learning from them more effectively.

Model configuration:
Root Node – Layer size = 512, Epochs = 15, and Batch size = 256
Child Nodes: Epochs = 20, LR = 0.0005, Batch size = 64, and Max depth = 4

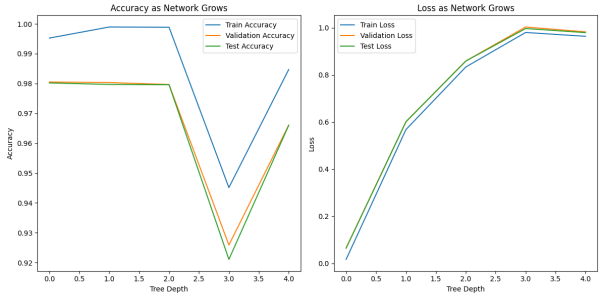

Figure 28: Accuracy and loss plot without class correction

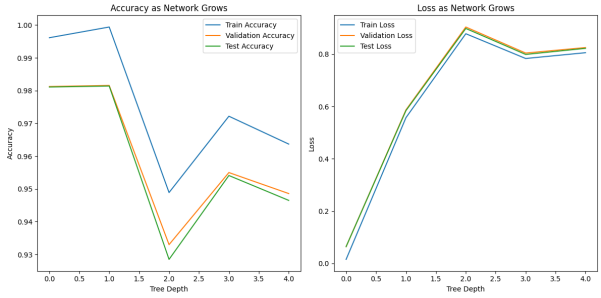

Figure 29: Accuracy and loss plot after class correction

We observe that the model curve becomes steeper but there is only marginal improvement in the training, validation, and test accuracy. This is probably because the dataset was already well-balanced.

Solution 2: Ensemble Prediction
Each time we train a child node on the misclassified samples and append it to the model's list, we add a new learner to the ensemble. The "ensemble predict" function then averages the predictions of all models in the ensemble.

- Root Model: The initial model trained on the entire training set serves as the base or "root" of your ensemble.
- Iterative Refinement: The loop iterates, identifying misclassified samples from the current ensemble's predictions.

- Child Models: For each iteration (or "tree depth"), a new model is trained specifically on the misclassified samples.
- Ensemble Prediction: The "ensemble predict" function combines the predictions of all models (root and children) by averaging their outputs. This improves overall performance by leveraging the strengths of different models trained on different parts of the data.

Model configuration
Root Node – Layer size = 512, Epochs = 15, and Batch size = 256
Child Nodes: Epochs = 20, LR = 0.0001, Batch size = 128, Max depth = 6, and Test data size = 0.3

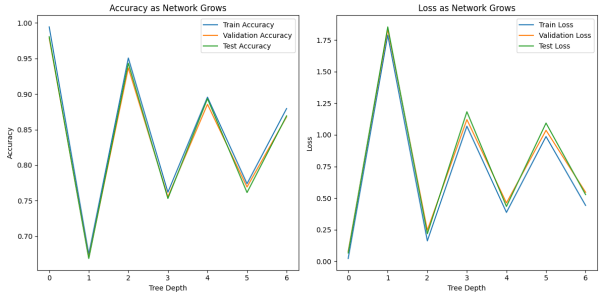

Figure 30: Accuracy and loss plot without class correction

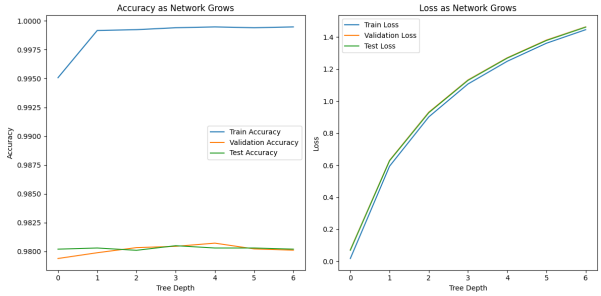

Figure 31: Accuracy and loss plot after class correction

Without the ensemble approach, the test accuracy sharply drops to 0.8693 and the all the training, validation, and test curves fluctuate sharply throughout the model training. However, with class imbalance correction using the ensemble approach, the model parameters improve substantially. The test accuracy rises sharply to 0.9802. □

Question 5

Try to visualize what each node has learnt.

Proof. We utilize the "visualize model" function to try and visualize what each node has learned in the MNIST dataset.

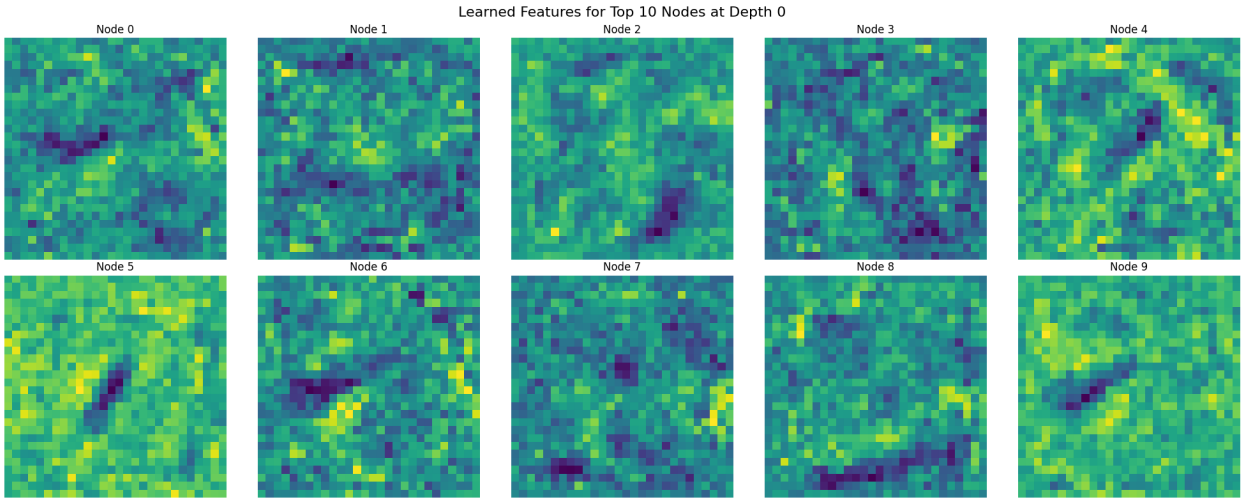

Figure 32: Visualizing the root model

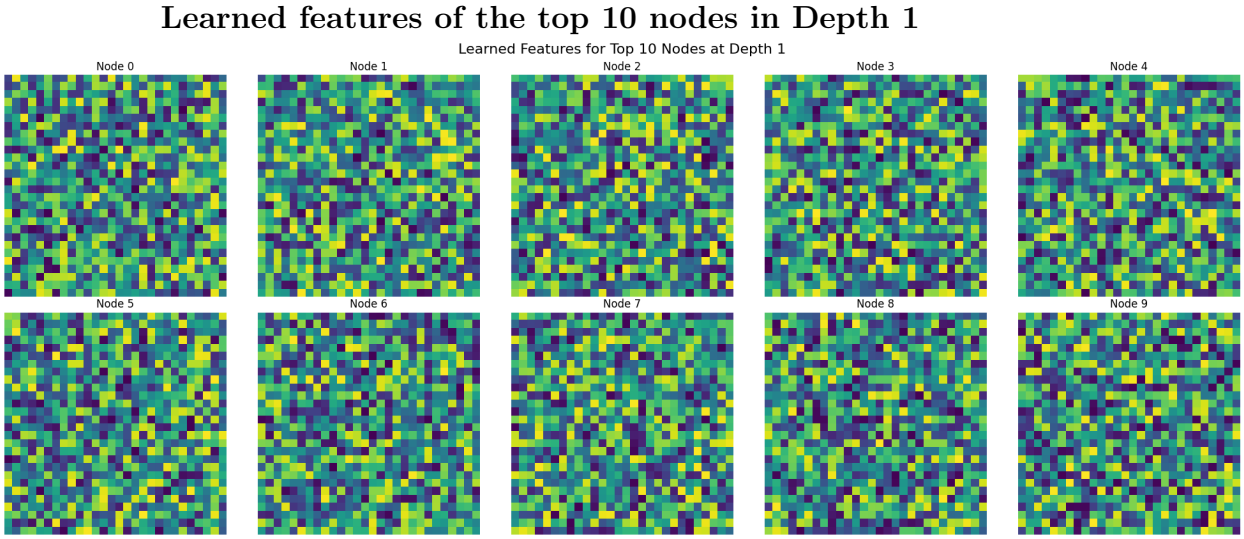

Figure 33: Visualizing learned features for Depth 1

□

**

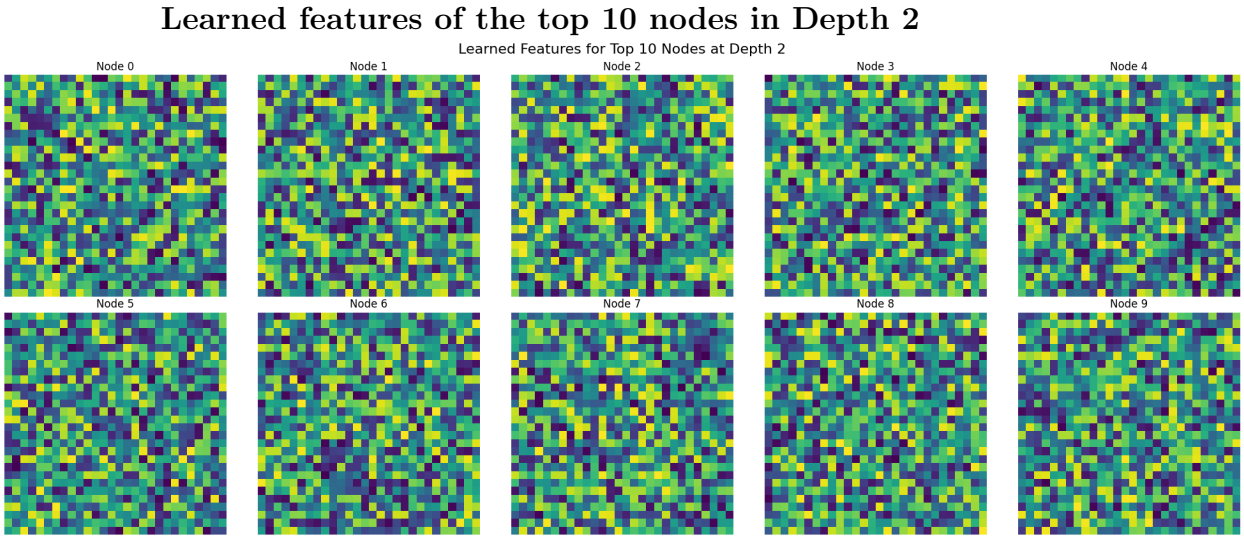

Figure 34: Visualizing learned features for Depth 2

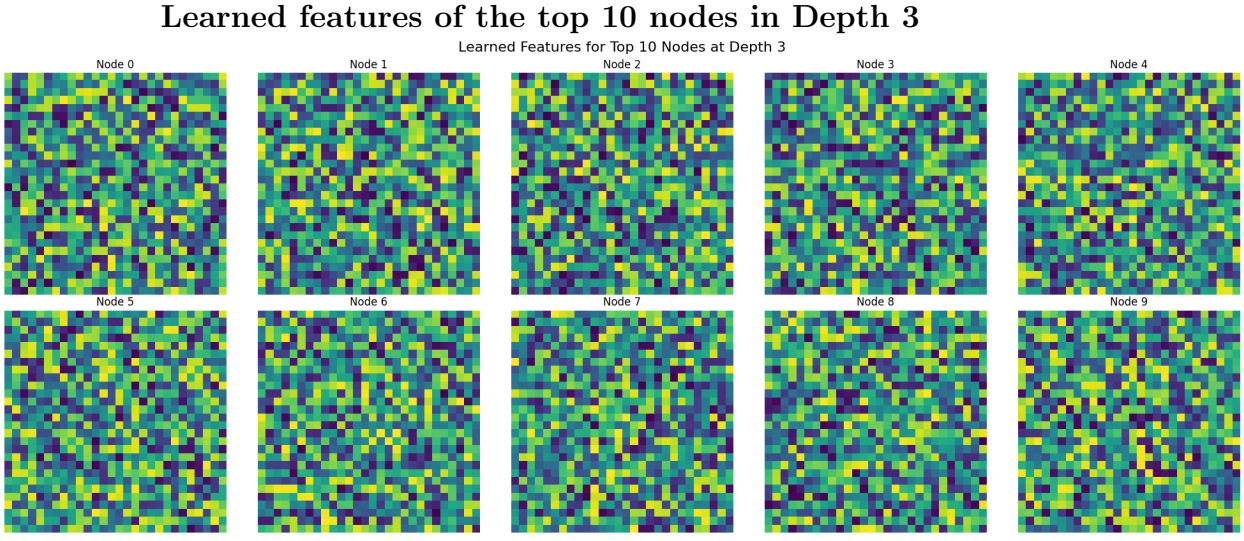

Figure 35: Visualizing learned features for Depth 3

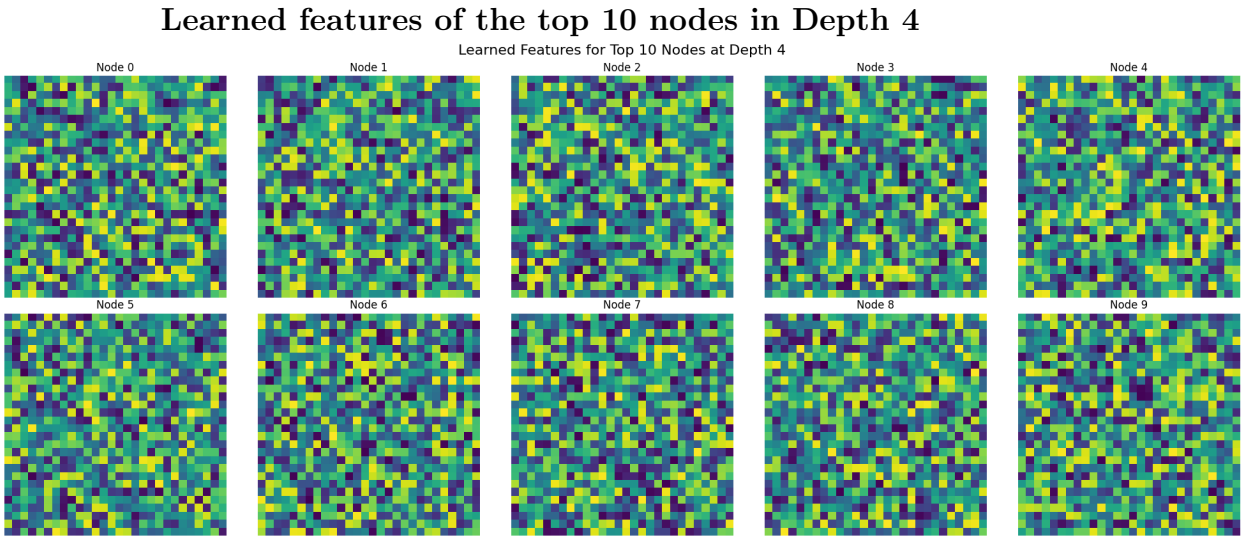

Figure 36: Visualizing learned features for Depth 4

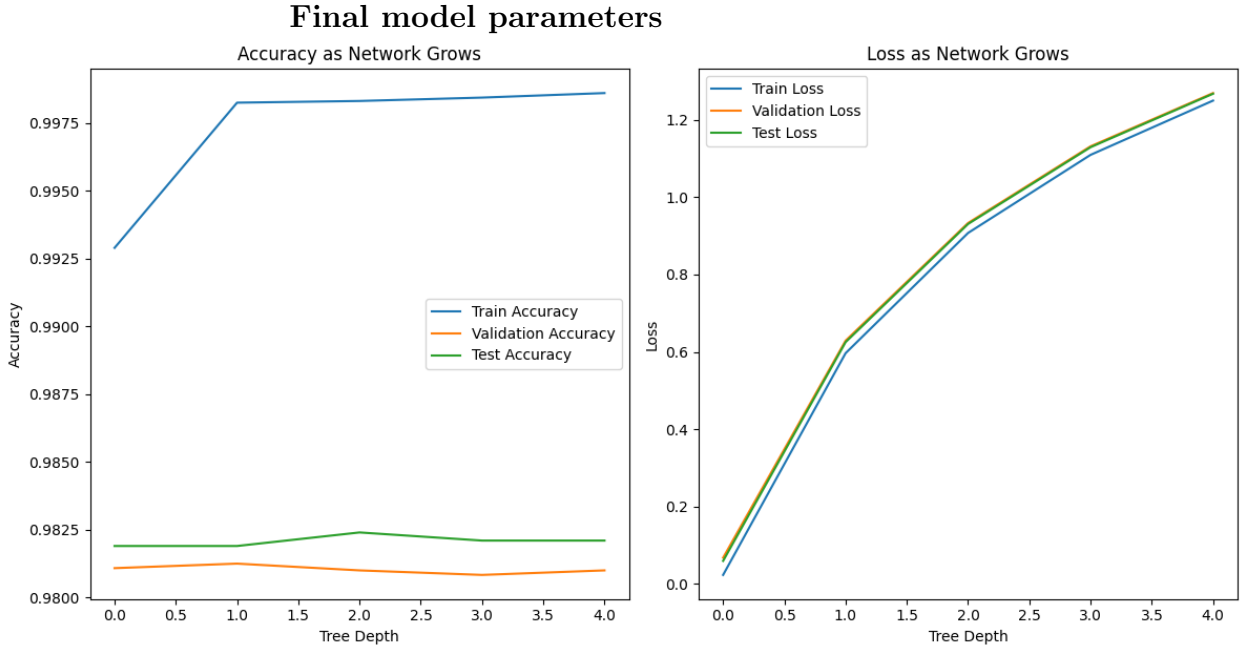

Figure 37: Accuracy and loss plot for the final model