ELL784
Assignment 2
2024EET2368

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.

Python Code

The following Python code demonstrates the implementation of an artificial neural network (ANN) for classifying handwritten images from the MNIST dataset. The code includes data loading, preprocessing, model training, and visualization of results.

Explanation of the Code

Data Loading and Preprocessing

The code starts by loading the MNIST dataset using NumPy's `np.load` function. It reshapes the images to be one-dimensional vectors and normalizes the pixel values to be between 0 and 1.

Train-Test Split

The custom `train_test_split` function is used to split the data into training and validation sets, ensuring a random shuffle of the data for each run.

Model Initialization

The `initialize_weights` function initializes the weights and biases for a simple two-layer neural network with ReLU activation for the hidden layer and softmax for the output layer.

Forward and Backward Propagation

The `forward_propagation` function computes the activations for both the hidden and output layers. The `backward_propagation` function computes the gradients of the weights and biases.

Training and Evaluation

The `train` function performs the training of the ANN. It calculates the loss and accuracy for both the training and validation sets, and updates the weights using gradient descent.

Testing and Visualization

After training, the model's performance on the test set is evaluated. The `plot_metrics` function generates plots of training and validation loss and accuracy over epochs.

Output

The output from the code will include printed loss and accuracy values at each epoch and visual plots of the loss and accuracy metrics. The final test accuracy achieved is reported.

Epoch 0, Training Loss: 0.2303, Validation Loss: 0.2303, Training Accuracy: 0.0929, Validation Accuracy: 0.0936

Epoch 10, Training Loss: 0.2195, Validation Loss: 0.2196, Training Accuracy: 0.4336, Validation Accuracy: 0.4338

Epoch 20, Training Loss: 0.1553, Validation Loss: 0.1556, Training Accuracy: 0.6827, Validation Accuracy: 0.6858

Epoch 30, Training Loss: 0.0920, Validation Loss: 0.0921, Training Accuracy: 0.7840, Validation Accuracy: 0.7831

Epoch 40, Training Loss: 0.0677, Validation Loss: 0.0676, Training Accuracy: 0.8293, Validation Accuracy: 0.8281

Epoch 50, Training Loss: 0.0560, Validation Loss: 0.0560, Training Accuracy: 0.8548, Validation Accuracy: 0.8553

Epoch 60, Training Loss: 0.0501, Validation Loss: 0.0500, Training Accuracy: 0.8673, Validation Accuracy: 0.8669

Epoch 70, Training Loss: 0.0532, Validation Loss: 0.0532, Training Accuracy: 0.8344, Validation Accuracy: 0.8334

Epoch 80, Training Loss: 0.0421, Validation Loss: 0.0419, Training Accuracy: 0.8876, Validation Accuracy: 0.8866

Epoch 90, Training Loss: 0.0397, Validation Loss: 0.0395, Training Accuracy: 0.8919, Validation Accuracy: 0.8918

Epoch 100, Training Loss: 0.0380, Validation Loss: 0.0378, Training Accuracy: 0.8957, Validation Accuracy: 0.8956

Epoch 110, Training Loss: 0.0366, Validation Loss: 0.0364, Training Accuracy: 0.8983, Validation Accuracy: 0.8981

Epoch 120, Training Loss: 0.0355, Validation Loss: 0.0353, Training Accuracy: 0.9009, Validation Accuracy: 0.9019

Epoch 130, Training Loss: 0.0345, Validation Loss: 0.0343, Training Accuracy: 0.9031, Validation Accuracy: 0.9041

Epoch 140, Training Loss: 0.0336, Validation Loss: 0.0335, Training Accuracy: 0.9047, Validation Accuracy: 0.9062

Epoch 150, Training Loss: 0.0329, Validation Loss: 0.0328, Training Accuracy: 0.9069, Validation Accuracy: 0.9079

Epoch 160, Training Loss: 0.0322, Validation Loss: 0.0321, Training Accuracy: 0.9078, Validation Accuracy: 0.9095

Epoch 170, Training Loss: 0.0316, Validation Loss: 0.0315, Training Accuracy: 0.9100, Validation Accuracy: 0.9111

Epoch 180, Training Loss: 0.0310, Validation Loss: 0.0310, Training Accuracy: 0.9114, Validation Accuracy: 0.9122

Epoch 190, Training Loss: 0.0305, Validation Loss: 0.0305, Training Accuracy: 0.9131, Validation Accuracy: 0.9135

Epoch 200, Training Loss: 0.0300, Validation Loss: 0.0300, Training Accuracy: 0.9148, Validation Accuracy: 0.9148

Epoch 210, Training Loss: 0.0295, Validation Loss: 0.0295, Training Accuracy: 0.9158, Validation Accuracy: 0.9155

Epoch 220, Training Loss: 0.0290, Validation Loss: 0.0291, Training Accuracy: 0.9171, Validation Accuracy: 0.9173

Epoch 230, Training Loss: 0.0286, Validation Loss: 0.0287, Training Accuracy: 0.9184, Validation Accuracy: 0.9177

Epoch 240, Training Loss: 0.0281, Validation Loss: 0.0283, Training Accuracy: 0.9197, Validation Accuracy: 0.9186

Final Test Accuracy: 0.9239

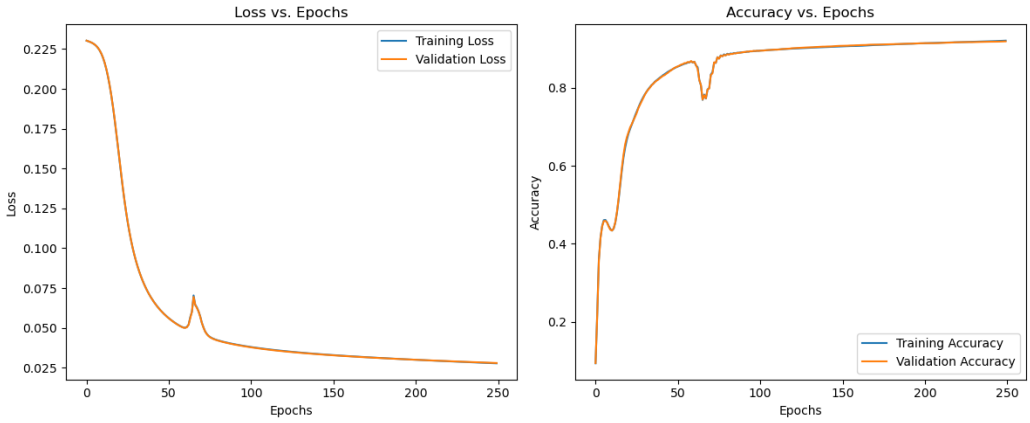

Figure 1: Traning loss vs Validation loss

Question 2

Consider the 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.

Node Training

Each node in the tree learns a one-layer neural network. The training function includes forward and backward propagation steps, and the weights are

updated using gradient descent. Loss and accuracy metrics are recorded for both training and validation sets.

Tree Neural Network Implementation

The 'TreeNeuralNetwork' class manages the growth of the network. Nodes are added iteratively, where each node attempts to classify misclassified samples from previous nodes. The network stops growing when no misclassified samples are left or the maximum depth is reached.

Evaluation and Visualization

The code plots the loss and accuracy metrics for each node as the network grows, and summarizes the performance of each node. It also visualizes the weights learned by each node to understand what each node has learned.

Outputs

Loss and Accuracy Plots

The following plots show the loss and accuracy metrics for each node in the tree-type neural network as the network grows:

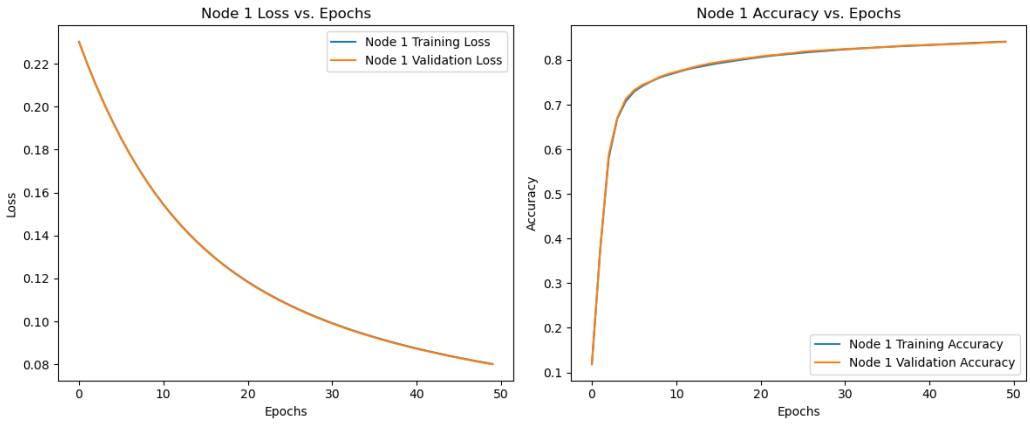

Figure 2:

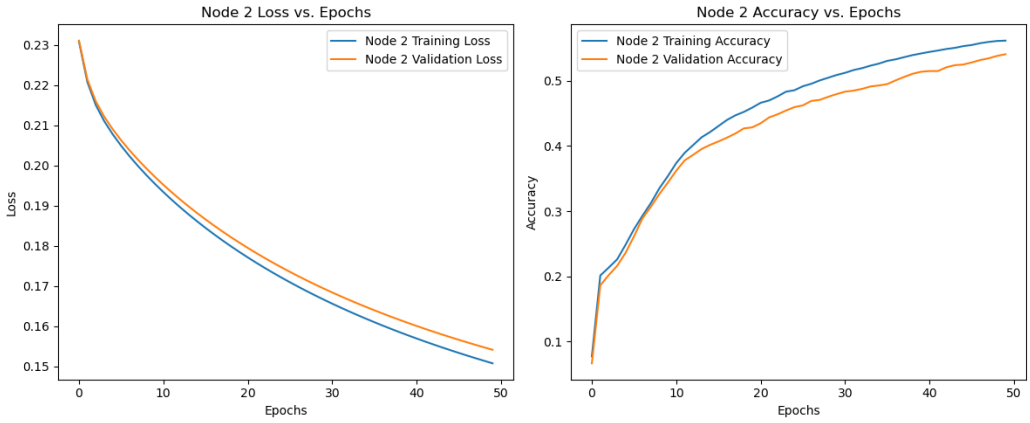

Figure 3:

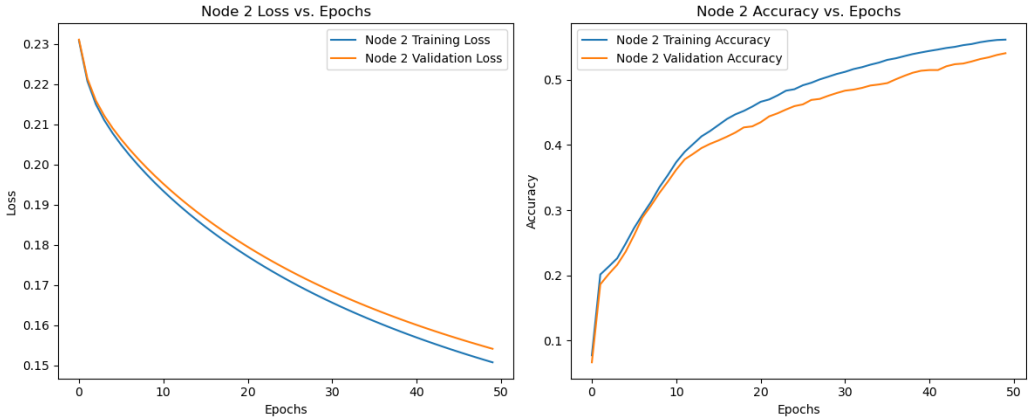

Figure 4:

Final Node Metrics

```
Node 1:
  Final Training Loss: 0.0802
  Final Validation Loss: 0.0800
  Final Training Accuracy: 0.8414
  Final Validation Accuracy: 0.8407

Node 2:
  Final Training Loss: 0.1508
```

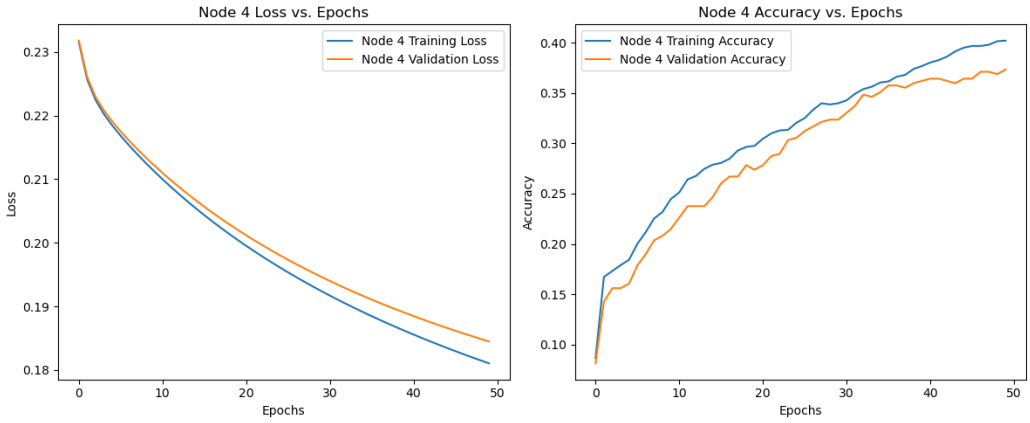

Figure 5:

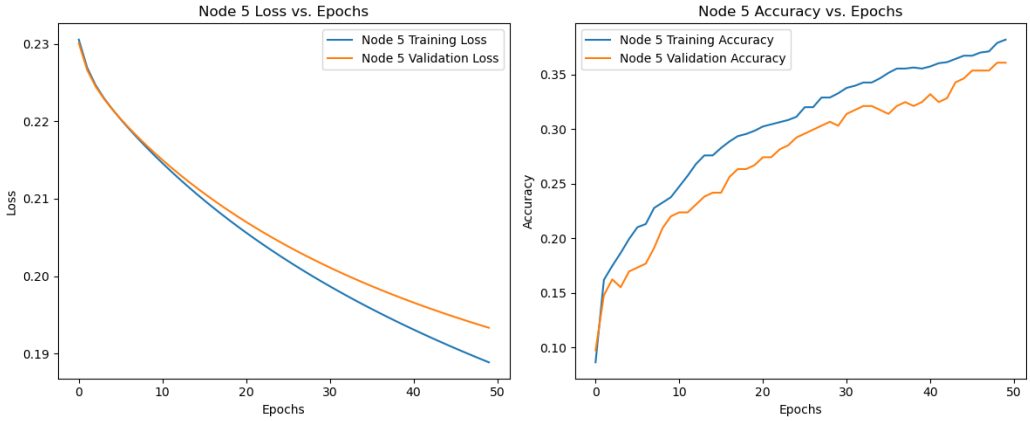

Figure 6:

```
  Final Validation Loss: 0.1542
  Final Training Accuracy: 0.5613
  Final Validation Accuracy: 0.5405

Node 3:
  Final Training Loss: 0.1665
  Final Validation Loss: 0.1655
  Final Training Accuracy: 0.4828
  Final Validation Accuracy: 0.4931
```

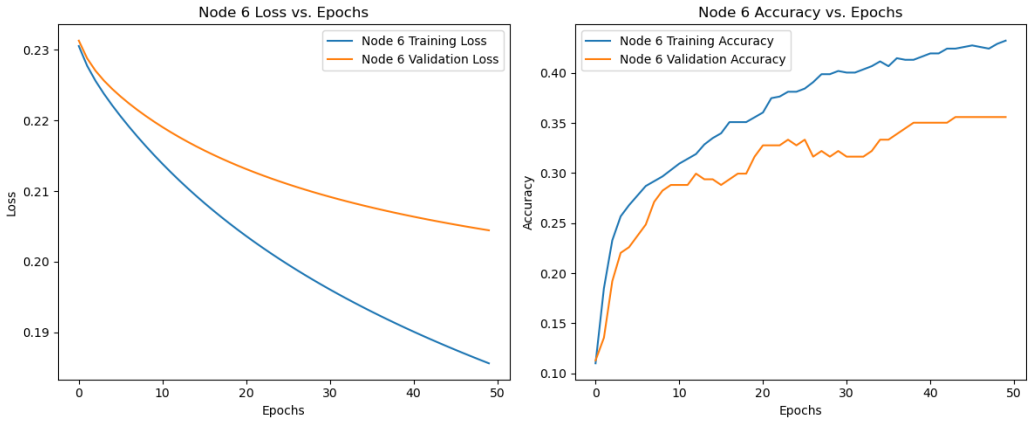

Figure 7: Training loss vs Validation loss

```
Node 4:
  Final Training Loss: 0.1810
  Final Validation Loss: 0.1845
  Final Training Accuracy: 0.4020
  Final Validation Accuracy: 0.3733

Node 5:
  Final Training Loss: 0.1889
  Final Validation Loss: 0.1933
  Final Training Accuracy: 0.3821
  Final Validation Accuracy: 0.3610

Node 6:
  Final Training Loss: 0.1856
  Final Validation Loss: 0.2045
  Final Training Accuracy: 0.4322
  Final Validation Accuracy: 0.3559

Overall Model Test Accuracy: 0.8645
```

Question 3

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

Strategies to Stop Training

Stopping training at the right time is crucial to improve generalization and avoid overfitting. Here are some strategies, illustrated based on the code provided:

A. Early Stopping

Early stopping involves monitoring the validation loss during training. Training is halted when the validation loss no longer improves for a predefined number of epochs (patience). This prevents overfitting by stopping before the model starts to memorize the training data.

- **Implementation:** Track the validation loss after each epoch. If the validation loss does not improve for a specified number of epochs, stop training.
- **Example from Code:** Implement early stopping by adding a check after each epoch to compare current validation loss with the best recorded loss.

B. Cross-Validation

Use k-fold cross-validation to evaluate model performance on multiple subsets of the data. This provides a more reliable estimate of the model's generalization capability and can guide when to stop training.

- **Implementation:** Split the training data into k subsets. Train the model k times, each time using a different subset as validation and the remaining subsets as training data.
- **Example from Code:** While the provided code does not directly implement k-fold cross-validation, it can be added to evaluate performance more robustly.

C. Learning Curves Analysis

Plot learning curves for training and validation loss/accuracy. Analyze these curves to identify overfitting or underfitting. Training can be stopped when the validation loss starts increasing while the training loss decreases, indicating overfitting.

- **Implementation:** Plot training and validation loss/accuracy after each epoch. Use these plots to decide when to stop training.
- **Example from Code:** Use the plot tree metrics function to visualize learning curves and identify points where validation metrics start to degrade.

D. Model Complexity and Regularization

Adjust the complexity of the model (e.g., number of nodes in the tree) and use regularization techniques (e.g., dropout, weight decay) to prevent overfitting. Stop training when adding more complexity does not significantly improve validation performance.

- **Implementation:** Regularize the model and monitor validation performance. Increase complexity only if it improves validation performance.
- **Example from Code:** The 'TreeNeuralNetwork' class can incorporate regularization techniques and adjust the tree depth based on validation performance.

Question 4

Discuss innovative ways of handling class imbalance at later tree nodes in the tree-type growing neural network.

Handling Class Imbalance at Later Tree Nodes

In the context of a tree-type growing neural network, class imbalance can become more pronounced at later nodes, especially as the network becomes

deeper and each node deals with a smaller subset of the data. Here are some innovative approaches to handle class imbalance effectively:

A. Resampling Techniques

Resampling can be used to balance class distributions at each node. Techniques include:

- **Oversampling**: Increase the number of samples in the minority class by duplicating existing samples or generating synthetic samples using techniques like SMOTE (Synthetic Minority Over-sampling Technique).
- **Undersampling**: Reduce the number of samples in the majority class to match the minority class, which helps balance the class distribution.

Incorporate these techniques into the training process of each node to ensure that the network learns effectively from imbalanced data.

B. Weighted Loss Function

Adjust the loss function to penalize misclassifications of minority class samples more heavily. This can be achieved by:

- **Class Weights**: Assign higher weights to the minority class in the loss function, making misclassifications of these samples more costly.
- **Focal Loss**: Use a focal loss function that focuses more on hard-to-classify examples and reduces the weight of easy examples, which can help in dealing with class imbalance.

Modify the loss calculation in the $\text{train}_n ode function to include these weighted loss functions.$

C. Dynamic Node Expansion

Instead of growing the tree based on a fixed depth, dynamically expand nodes based on the class distribution:

- **Adaptive Depth**: Increase the depth of the tree in regions with high class imbalance, allowing the network to handle imbalanced classes more effectively.

- **Focused Expansion**: Add child nodes only for classes with significant imbalance, rather than all misclassified samples, to better address specific class imbalances.

Incorporate these strategies into the 'TreeNeuralNetwork' class to adaptively manage class imbalance.

D. Ensemble Methods

Combine multiple models to improve performance on imbalanced data:

- **Bagging and Boosting**: Use ensemble methods like bagging or boosting to aggregate predictions from multiple nodes or models, improving overall performance and handling imbalances more effectively.
- **Stacking**: Stack multiple models trained on different subsets or using different techniques to enhance the model's ability to handle class imbalance.

Implement these methods by aggregating the outputs of different nodes or models in the 'predict' function of the 'TreeNeuralNetwork' class.

E. Data Augmentation

Augment data in the minority class to improve representation:

- **Image Augmentation**: Apply transformations such as rotation, scaling, and flipping to generate new samples from existing minority class data.
- **Synthetic Data Generation**: Create synthetic samples using techniques like Generative Adversarial Networks (GANs) to bolster the minority class.

Incorporate data augmentation strategies into the training pipeline to enhance the minority class representation.

Question 5

Try to visualize what each node has learnt

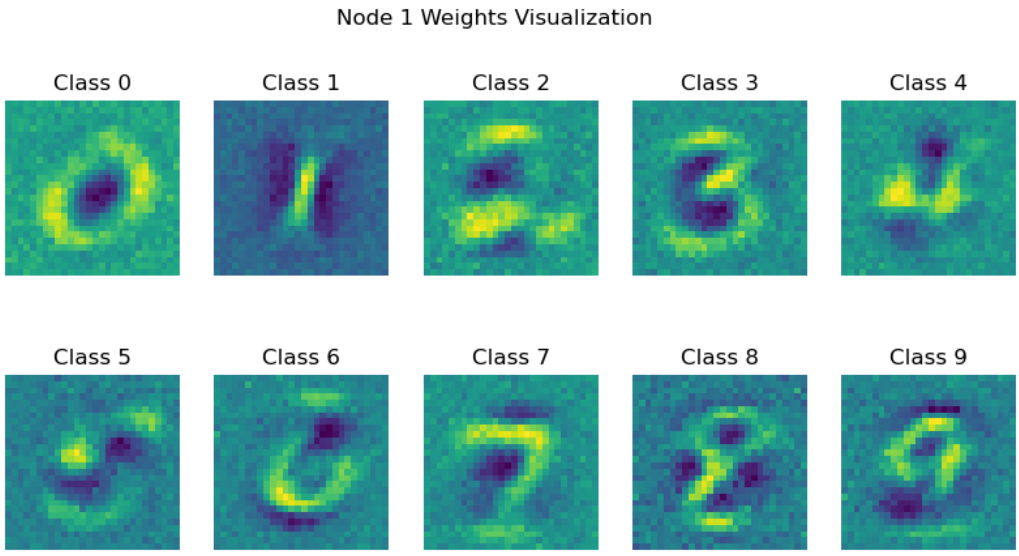

Figure 8: Enter Caption