MAE263F Homework 3

Che Jin Goh

This report details the implementation and training of a feedforward neural network for the classification of handwritten digits using the MNIST dataset. The network processes grayscale images by flattening, normalizing pixel intensities, and one-hot encoding labels to prepare the data for supervised learning. Simulations using MATLAB include forward and backward propagation, with cross-entropy loss and gradient descent optimizing the network. Key metrics such as training loss and testing accuracy are analyzed to evaluate the model's performance across various hyperparameter configurations.

I. Introduction

This report focuses on the design, implementation, and evaluation of a feedforward neural network to classify handwritten digits from the MNIST dataset. The network architecture includes multiple layers that process input images through forward and backward propagation, utilizing cross-entropy loss and gradient descent for optimization. The data is preprocessed by flattening grayscale images, normalizing pixel intensities, and applying one-hot encoding to labels. The primary objectives are to train the model effectively, analyze its performance in terms of accuracy, and explore the impact of varying hyperparameters such as learning rate, batch size, and number of epochs. Simulations and experiments are conducted in MATLAB, with results highlighting the network's ability to learn and generalize from the provided dataset.

II. Code Architure

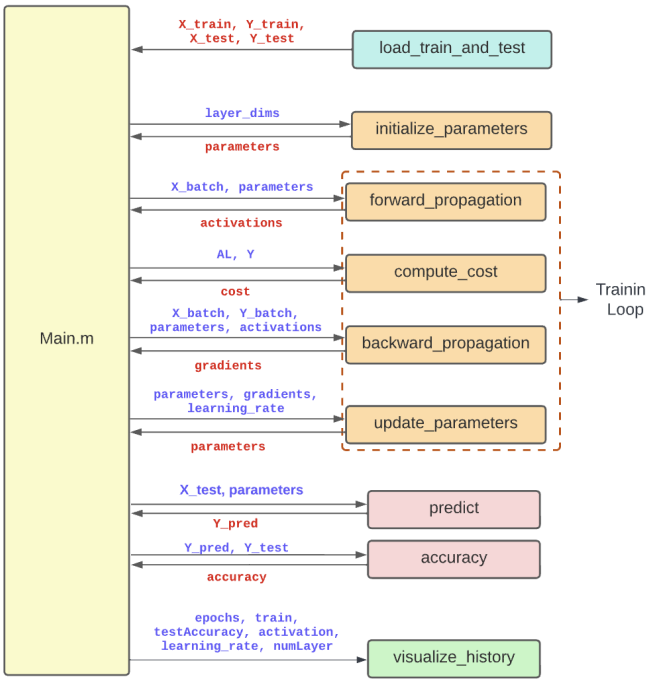

Figure 1: Overview of code architecture

The structure of the code, as depicted in figure 1, illustrates the modular design and logical flow of the feedforward neural network implementation for digit classification. Each component is encapsulated in a specific function to promote clarity, reusability, and scalability. Below, we discuss the key components and their interactions:

The main script orchestrates the entire workflow, calling various functions for loading data, initializing parameters, training the model, and evaluating its performance. It serves as the backbone of the project, linking all modular components into a cohesive pipeline.

1. Data Preparation
   - Function: `load_train_and_test`

   This function is responsible for loading the MNIST dataset (images and labels), preprocessing it (flattening and normalizing the images), and converting the labels into one-hot encoded vectors.

2. Model Construction
   - Function: `initialize_parameters`

   This function initializes the weights and biases of the network for all layers. Weights are initialized with random values (e.g., Gaussian distribution), and biases are initialized to zeros. This step is critical for ensuring the network can learn effectively.

   - Function: `tanh2`

   This function implements the hyperbolic tangent activation function, which introduces non-linearity into the network and enables it to learn complex relationships between inputs and outputs. This is a critical component of the feedforward neural network, as linear transformations alone are insufficient to capture intricate data patterns.

   - Function: `softmax`

   This is a mathematical operation used in the output layer of a neural network for multiclass classification tasks. It converts the raw scores (logits) produced by the network into a probability distribution over all possible classes, ensuring that the probabilities sum to 1. This allows the network to predict the likelihood of each class for a given input.

3. Training Loop
   - Function: `forward_propagation`

   Takes a batch of inputs and computes intermediate activations at each layer, including the final predicted outputs.
   - Function: `compute_cost`

   Calculates the cross-entropy loss by comparing the predicted output with the ground-truth labels. This metric guides how the weights should be adjusted.
   - Function: `backwards_propagation`

   Uses the computed cost to calculate gradients for each weight and bias in the network. Gradients are

determined using the chain rule to backpropagate errors from the output layer to the input layer.

- Function: `update_parameters`

Updates weights and biases using the calculated gradients and a specified learning rate. This step fine-tunes the network to minimize the loss.

4. Model Evaluation
   - Function: `predict`

   Generates predictions for unseen data by running forward propagation on the test set.
   - Function: `accuracy`

   Compares the predicted labels with actual labels and computes the classification accuracy as a percentage.

5. Visualization
   - Function: `visualize_history`

   This function creates plots to visualize the training progress over epochs

III. Results And Discussion

1) Problem 1

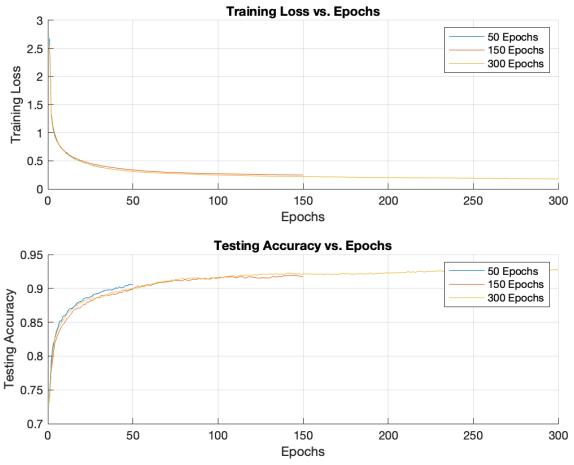

Figure 2: Results with 50, 100, 300 epochs

2) Problem 2

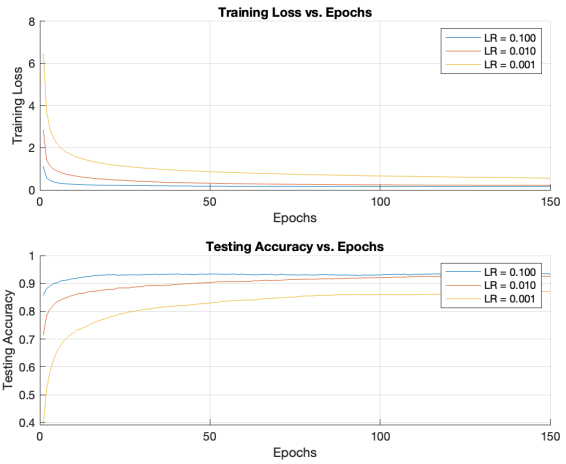

Figure 3: Results with 0.1, 0.01 and 0.001 learning rates

3) Problem 3

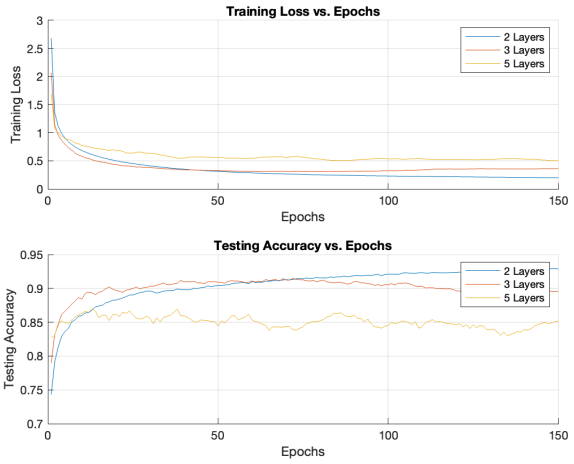

Figure 4: Results with 2, 3 and 5 number of layers

Figure 2 shows the effect of training epochs (50, 150, 300) on training loss and testing accuracy. Training loss decreases rapidly in the first 50 epochs, with minimal improvement beyond 150 epochs. Testing accuracy stabilizes around 150 epochs, indicating diminishing returns for additional training.

Figure 3 demonstrates the impact of learning rates (0.1, 0.01, 0.001). A learning rate of 0.1 converges quickly but shows slight instability in accuracy. A rate of 0.01 provides the highest and most stable accuracy, while 0.001 converges slowly and under fits the data.

Figure 4 examines the number of layers (2, 3, 5). The 3-layer model achieves the best accuracy and loss, balancing complexity and generalization. The 2-layer model underperforms slightly, while the 5-layer model shows slower convergence and fluctuating accuracy due to overparameterization.