MNIST Handwritten Digit Recognition

Implementation Guide for Students

CSCI 431: Machine Learning
Prithvi Raj Singh

Implementation of a Neural Network for MNIST Digit Recognition

Abstract

This guide provides essential information for implementing a neural network to recognize handwritten digits from the MNIST dataset. It includes code snippets for key components, algorithmic guidance, and practical tips without providing the complete solution. Students are expected to understand the concepts and write their own implementation.

1 Overview

You will implement a fully connected feedforward neural network with the following architecture:

- **Input Layer**: 784 neurons (28×28 pixel images)
- **Hidden Layer**: 15 neurons (minimum requirement)
- **Output Layer**: 10 neurons (digits 0-9)

Key Requirements:

1. No external libraries (except `java.util & java.io`)
2. Implement all matrix operations yourself
3. Use mini-batch stochastic gradient descent
4. Achieve > 95% accuracy on test set. Lower accuracy may indicate bugs. But don't worry too much about it.

2 Data Handling

2.1 Understanding MNIST Data Format

Each line in the CSV files contains 785 comma-separated values:

$$\text{label}, \text{pixel}_{0,0}, \text{pixel}_{0,1}, \ldots, \text{pixel}_{27,27} \tag{1}$$

- **Label**: The digit (0-9)
- **Pixels**: 784 grayscale values (0-255)
- **Normalization**: Convert pixel values to [0, 1] by dividing by 255
- **One-Hot Encoding**: Convert label to a 10-element vector for training
- It is very possible if you download dataset from other sources than suggested, file headers might be different so you need to handle that accordingly.

2.2 Loading CSV Data

Listing 1: Reading MNIST CSV File

```java
import java.io.*;
import java.util.*;

public List<MNISTData> loadMNISTData(String filename)
    throws IOException {

    List<MNISTData> dataList = new ArrayList<>();
    BufferedReader reader = new BufferedReader(
        new FileReader(filename));

    String line;
    while ((line = reader.readLine()) != null) {
        String[] values = line.split(",");

        // Parse label (first value)
        int label = Integer.parseInt(values[0].trim());

        // Parse and normalize pixels (remaining 784 values)
        double[] pixels = new double[784];
        for (int i = 0; i < 784; i++) {
            // TODO: Parse pixel value from values[i+1]
            // TODO: Normalize to [0, 1] by dividing by 255
            pixels[i] = /* YOUR CODE HERE */;
        }

        dataList.add(new MNISTData(label, pixels));
    }

    reader.close();
    return dataList;
}
```

2.3 Data Structure

Create a simple class to hold each training example:

Listing 2: MNIST Data Structure

```java
public class MNISTData {
    int label;           // The digit (0-9)
    double[] pixels;     // 784 normalized pixel values
    double[] oneHot;     // 10-element one-hot encoded label

    public MNISTData(int label, double[] pixels) {
        this.label = label;
        this.pixels = pixels;
        this.oneHot = createOneHot(label);
    }

    private double[] createOneHot(int label) {
        double[] oneHot = new double[10];
        // TODO: Set the appropriate index to 1.0
        // Example: label=7 -> [0,0,0,0,0,0,0,1,0,0]
        return oneHot;
    }
```

```
}
```

Why One-Hot Encoding?
The network outputs 10 values (one per digit). One-hot encoding converts the label into a vector that can be directly compared with the network output using Mean Squared Error.

3 Network Architecture

3.1 Weight and Bias Initialization

Initialize weights randomly in the range $[-1, 1]$:

Listing 3: Random Weight Initialization

```java
import java.util.Random;

public void initializeWeights() {
    Random rand = new Random();

    // Initialize W1: hiddenSize x inputSize
    weightsInputHidden = new double[hiddenSize][inputSize];
    for (int i = 0; i < hiddenSize; i++) {
        for (int j = 0; j < inputSize; j++) {
            weightsInputHidden[i][j] =
                rand.nextDouble() * 2 - 1;  // Range: [-1, 1]
        }
    }

    // Initialize B1: hiddenSize x 1
    biasHidden = new double[hiddenSize];
    for (int i = 0; i < hiddenSize; i++) {
        biasHidden[i] = rand.nextDouble() * 2 - 1;
    }

    // TODO: Similarly initialize W2 and B2
    // W2 dimensions: outputSize x hiddenSize
    // B2 dimensions: outputSize x 1
}
```

3.2 Forward Propagation Structure

The forward pass should compute activations for each layer:

Listing 4: Forward Propagation Pattern

```java
public double[] forwardPropagate(double[] input) {
    // Layer 1: Input -> Hidden
    double[] hiddenZ = new double[hiddenSize];
    double[] hiddenA = new double[hiddenSize];

    for (int i = 0; i < hiddenSize; i++) {
        // Step 1: Compute weighted sum
        hiddenZ[i] = biasHidden[i];
        for (int j = 0; j < inputSize; j++) {
            hiddenZ[i] += weightsInputHidden[i][j] * input[j];
        }

        // Step 2: Apply activation function
```

```java
            hiddenA[i] = sigmoid(hiddenZ[i]);
    }

    // Layer 2: Hidden -> Output
    // TODO: Similar structure for output layer
    // Use hiddenA as input to this layer

    return outputActivations;
}
```

4 Training Algorithm

4.1 Mini-Batch SGD Overview

Mini-Batch Training Algorithm

For each epoch:

1. Shuffle the training data
2. Divide data into mini-batches of size B
3. **For each mini-batch:**
   (a) Initialize gradient accumulators to zero
   (b) **For each example in batch:**
      i. Forward propagate
      ii. Compute output error
      iii. Backward propagate (compute deltas)
      iv. Accumulate gradients (don't update yet!)
   (c) Average all accumulated gradients by batch size
   (d) Update all weights and biases once
4. Evaluate accuracy on training set

4.2 Training Loop Structure

Listing 5: Training Loop Pattern

```java
public void train(List<MNISTData> trainingData, int epochs) {
    for (int epoch = 0; epoch < epochs; epoch++) {
        // Shuffle data for this epoch
        Collections.shuffle(trainingData);

        // Process mini-batches
        for (int i = 0; i < trainingData.size();
             i += miniBatchSize) {

            // Extract mini-batch
            int batchEnd = Math.min(
                i + miniBatchSize, trainingData.size());
            List<MNISTData> batch =
                trainingData.subList(i, batchEnd);

```

```
            // Train on this mini-batch
            trainMiniBatch(batch);
        }

        // Evaluate and print statistics
        evaluateAccuracy(trainingData);
    }
}
```

4.3 Mini-Batch Training Implementation Hint

Listing 6: Mini-Batch Training Structure

```
private void trainMiniBatch(List<MNISTData> batch) {
    int batchSize = batch.size();

    // Create gradient accumulators (initialized to 0)
    double[][] w2Gradients = new double[outputSize][hiddenSize];
    double[] b2Gradients = new double[outputSize];
    // TODO: Create w1Gradients and b1Gradients

    // Accumulate gradients for each example
    for (MNISTData example : batch) {
        // 1. Forward propagate
        // 2. Compute deltas (backpropagation)
        // 3. Add to gradient accumulators
        //    (do NOT update weights yet!)
    }

    // Average gradients and update weights
    double scale = learningRate / batchSize;
    for (int i = 0; i < outputSize; i++) {
        for (int j = 0; j < hiddenSize; j++) {
            weightsHiddenOutput[i][j] -= scale * w2Gradients[i][j];
        }
        biasOutput[i] -= scale * b2Gradients[i];
    }

    // TODO: Similarly update W1 and B1
}
```

5 Key Implementation Details

5.1 Sigmoid Function

Listing 7: Numerically Stable Sigmoid

```
private double sigmoid(double z) {
    // Prevent overflow for extreme values
    if (z < -500) return 0.0;
    if (z > 500) return 1.0;
    return 1.0 / (1.0 + Math.exp(-z));
}

private double sigmoidDerivative(double activation) {
    // Input is the activation (after sigmoid)
```

```
    // NOT the pre-activation z
    return activation * (1.0 - activation);
}
```

5.2 Making Predictions

Listing 8: Getting Network Prediction

```
public int predict(double[] input) {
    double[] output = forwardPropagate(input);

    // Find index of maximum output
    int maxIndex = 0;
    double maxValue = output[0];

    for (int i = 1; i < output.length; i++) {
        if (output[i] > maxValue) {
            maxValue = output[i];
            maxIndex = i;
        }
    }

    return maxIndex;  // This is the predicted digit
}
```

5.3 Evaluating Accuracy

Listing 9: Accuracy Evaluation

```
public void evaluateAccuracy(List<MNISTData> data) {
    int[] digitCounts = new int[10];
    int[] correctCounts = new int[10];
    int totalCorrect = 0;

    for (MNISTData example : data) {
        int predicted = predict(example.pixels);
        int actual = example.label;

        digitCounts[actual]++;
        if (predicted == actual) {
            correctCounts[actual]++;
            totalCorrect++;
        }
    }

    // Print per-digit accuracy
    System.out.println("Results:");
    for (int i = 0; i < 10; i++) {
        System.out.printf("Digit %d: %d/%d\t",
            i, correctCounts[i], digitCounts[i]);
        if (i % 2 == 1) System.out.println();
    }

    // Print overall accuracy
    double accuracy = (double)totalCorrect / data.size() * 100;
    System.out.printf("\nAccuracy: %d/%d = %.3f%%\n",
        totalCorrect, data.size(), accuracy);
```

```java
}
```

6 Visualization

6.1 ASCII Art Display

Listing 10: Displaying Digit as ASCII Art

```java
private void displayImage(double[] pixels) {
    System.out.println("Image (28x28):");

    for (int row = 0; row < 28; row++) {
        for (int col = 0; col < 28; col++) {
            int index = row * 28 + col;
            double pixel = pixels[index];

            // Map pixel intensity to ASCII character
            char ch;
            if (pixel < 0.2) ch = ' ';
            else if (pixel < 0.4) ch = '.';
            else if (pixel < 0.6) ch = 'o';
            else if (pixel < 0.8) ch = 'O';
            else ch = '@';

            System.out.print(ch);
        }
        System.out.println();
    }
}
```

7 Save/Load Network

7.1 Saving Network State

Listing 11: Saving Network to File

```java
public void saveNetwork(String filename) throws IOException {
    PrintWriter writer = new PrintWriter(
        new FileWriter(filename));

    // Write hyperparameters
    writer.println(learningRate);
    writer.println(miniBatchSize);
    writer.println(inputSize);
    writer.println(hiddenSize);
    writer.println(outputSize);

    // Write all weights (one per line)
    for (int i = 0; i < hiddenSize; i++) {
        for (int j = 0; j < inputSize; j++) {
            writer.println(weightsInputHidden[i][j]);
        }
    }

    // Write all biases
```

```
    for (int i = 0; i < hiddenSize; i++) {
        writer.println(biasHidden[i]);
    }

    // TODO: Similarly write W2 and B2

    writer.close();
    System.out.println("Network saved to " + filename);
}
```

7.2 Loading Network State

Listing 12: Loading Network from File

```
public static MNISTNeuralNetwork loadNetwork(String filename)
    throws IOException {

    BufferedReader reader = new BufferedReader(
        new FileReader(filename));

    // Read hyperparameters
    double lr = Double.parseDouble(reader.readLine());
    int batchSize = Integer.parseInt(reader.readLine());
    int inputSize = Integer.parseInt(reader.readLine());
    int hiddenSize = Integer.parseInt(reader.readLine());
    int outputSize = Integer.parseInt(reader.readLine());

    // Create network with these parameters
    MNISTNeuralNetwork network =
        new MNISTNeuralNetwork(lr, batchSize);

    // Read weights in same order they were written
    for (int i = 0; i < hiddenSize; i++) {
        for (int j = 0; j < inputSize; j++) {
            network.weightsInputHidden[i][j] =
                Double.parseDouble(reader.readLine());
        }
    }

    // TODO: Read remaining weights and biases

    reader.close();
    return network;
}
```

8 User Interface

8.1 Menu System

Listing 13: Simple CLI Menu

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    MNISTNeuralNetwork network = null;
    List<MNISTData> trainingData = null;
    List<MNISTData> testData = null;
```

```java

    while (true) {
        System.out.println("\n--- MENU ---");
        System.out.println("1. Train the network");
        System.out.println("2. Load a pre-trained network");
        System.out.println("3. Test on training data");
        System.out.println("4. Test on testing data");
        System.out.println("5. Show predictions");
        System.out.println("6. Show misclassified images");
        System.out.println("7. Save network");
        System.out.println("0. Exit");
        System.out.print("Choice: ");

        String choice = scanner.nextLine().trim();

        try {
            switch (choice) {
                case "1":
                    // TODO: Get parameters, load data, train
                    break;
                case "2":
                    // TODO: Load network from file
                    break;
                // TODO: Implement other cases
                case "0":
                    System.out.println("Goodbye!");
                    return;
                default:
                    System.out.println("Invalid choice");
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

9 Practical Tips

9.1 Recommended Hyperparameters

Parameter	Suggested Value
Learning Rate (η)	3.0
Mini-batch Size	10
Number of Epochs	30
Hidden Neurons	15 (minimum)

Table 1: Suggested hyperparameters for good performance

9.2 Debugging Strategies

1. **Start small**: Test with 100 examples first
2. **Monitor loss**: Should decrease each epoch
3. **Check dimensions**: Print array sizes before operations

4. **Verify normalization**: Pixels should be in [0, 1]
5. **Test predictions**: Should not all be the same digit
6. **Use Part 1**: If stuck, verify backprop logic matches Part 1

9.3 Common Mistakes

- **Forgetting to normalize pixels** (divide by 255)
- **Updating weights inside batch loop** (should accumulate first)
- **Wrong matrix dimensions** (check rows × columns)
- **Not shuffling data** each epoch
- **Using wrong index for one-hot** (label vs. index)

10 Expected Performance

Training Timeline:

- Epoch 1-5: Accuracy rises from ∼10% to ∼80%
- Epoch 6-15: Accuracy reaches ∼90-95%
- Epoch 16-30: Accuracy improves to ∼95-99%

Final Results:

- Training accuracy: 98-99%
- Testing accuracy: 95-97%

Timing:

- One epoch (60,000 examples): 20-40 seconds
- Full training (30 epochs): 10-20 minutes

11 What You Need to Implement

This guide has provided:
✓ Data loading structure
✓ Network initialization
✓ Forward propagation pattern
✓ Training loop structure
✓ Utility functions (sigmoid, predict, display)

You still need to implement:
☐ Complete backpropagation logic (compute deltas)
☐ Gradient calculations for all layers
☐ Weight update logic
☐ Complete menu system
☐ Error handling
☐ Any additional features you wish to add

Remember: The math previously defined while implementing small netowkr applies directly here. The only differences are the larger dimensions and mini-batch accumulation. If you understood math before implementing small network, you can fully extend previous math for MNIST purpose. If you don't understand please ask me or refer to the document!

Final Advice: Don't try to write everything at once. Build incrementally, test frequently, and make sure you understand each component before moving to the next. Good luck!