MNIST Digit Recognizer: Code Documentation

Dmitrii Vlasov

September 13, 2024

Abstract

This document provides a detailed explanation of the code used for the MNIST Digit Recognizer project, including data loading, preprocessing, model implementation from scratch using NumPy, training procedure, and model evaluation.

1 Introduction

This project implements a Multi-Layer Perceptron (MLP) neural network from scratch using NumPy to recognize handwritten digits from the MNIST dataset. The code demonstrates the fundamental concepts of neural networks without relying on high-level libraries like TensorFlow or PyTorch.

2 Dataset Overview

The MNIST dataset consists of grayscale images of handwritten digits from 0 to 9, with each image having a resolution of 28×28 pixels.

Features:

- **Images:** Each image is represented as a flattened vector of 784 pixels.
- **Pixel Values:** Pixel intensities range from 0 (black) to 255 (white).
- **Labels:** Integer labels from 0 to 9 indicating the digit.

Data Source:

- The dataset is obtained from the Kaggle Digit Recognizer competition.

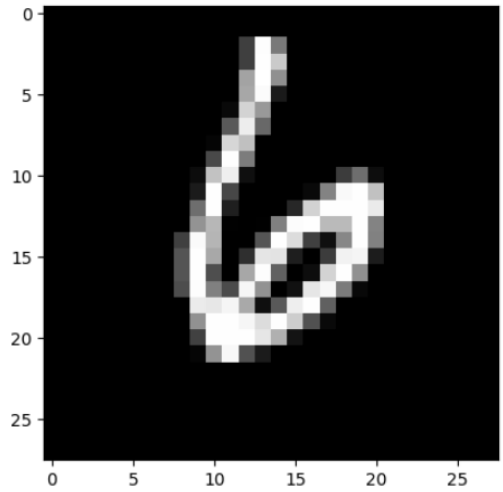

Figure 1: Sample image from the dataset: Digit 6

3 Data Loading and Preparation

The data is loaded using `pandas` and then converted to a NumPy array for processing.

Listing 1: Data Loading

```
import pandas as pd
import numpy as np

data = pd.read_csv('train.csv')
data = np.array(data)
```

3.1 Data Shuffling and Splitting

The dataset is shuffled and split into training and validation sets with an 80-20 split.

Listing 2: Data Shuffling and Splitting

```
np.random.seed(42)
np.random.shuffle(data)

m, n = data.shape
cutoff = int(m * 0.2)
val = data[0:cutoff]
train = data[cutoff:]
```

3.2 Feature and Label Separation

Features and labels are separated for both training and validation sets.

Listing 3: Feature and Label Separation

```
y_train = train[:, 0]
x_train = train[:, 1:]

y_val = val[:, 0]
x_val = val[:, 1:]
```

3.3 Data Normalization

Pixel values are normalized to the range [0, 1] by dividing by 255.

Listing 4: Data Normalization

```
x_train = x_train / 255.0
x_val = x_val / 255.0
```

3.4 One-Hot Encoding

Labels are one-hot encoded to facilitate training with cross-entropy loss.

Listing 5: One-Hot Encoding

```
def one_hot_encode(y, num_classes):
    one_hot = np.zeros((y.shape[0], num_classes))
    one_hot[np.arange(y.shape[0]), y.astype(int)] = 1
    return one_hot

y_train_one_hot = one_hot_encode(y_train, 10)
y_val_one_hot = one_hot_encode(y_val, 10)
```

4 Model Architecture

An MLP is implemented with one hidden layer.

4.1 Network Structure

- **Input Layer:** 784 neurons (flattened pixels).
- **Hidden Layer:** 64 neurons with ReLU activation.
- **Output Layer:** 10 neurons with softmax activation.

4.2 Weight Initialization

Weights are initialized using a small random normal distribution, and biases are initialized to zero.

Listing 6: Weight Initialization

```
self.w1 = np.random.randn(input_size, hidden_size) * 0.01
self.b1 = np.zeros((1, hidden_size))
self.w2 = np.random.randn(hidden_size, output_size) * 0.01
self.b2 = np.zeros((1, output_size))
```

4.3 Activation Functions

ReLU Activation:

$$\text{ReLU}(z) = \max(0, z)$$

Softmax Activation:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

5 Forward Propagation

The forward pass computes the output predictions.

Listing 7: Forward Propagation

```
def forward(self, x):
    z1 = np.dot(x, self.w1) + self.b1
    a1 = self.relu(z1)
    z2 = np.dot(a1, self.w2) + self.b2
    a2 = self.softmax(z2)
    return a1, a2
```

6 Loss Function

The model uses categorical cross-entropy loss.

$$L = -\frac{1}{m} \sum_{i=1}^{m} \log(a_{y^{(i)}}^{(i)})$$

Listing 8: Loss Function

```
def compute_loss(self, y_true, y_pred):
    m = y_true.shape[0]
    log_probs = -np.log(y_pred[range(m), y_true])
    loss = np.sum(log_probs) / m
    return loss
```

7 Backpropagation

Gradients are computed using backpropagation.

Listing 9: Backpropagation

```
def backward(self, x, y_true, a1, a2):
    m = x.shape[0]
    y_true_encoded = np.zeros_like(a2)
    y_true_encoded[np.arange(m), y_true] = 1

    dz2 = a2 - y_true_encoded
    dw2 = np.dot(a1.T, dz2) / m
    db2 = np.sum(dz2, axis=0, keepdims=True) / m

    dz1 = np.dot(dz2, self.w2.T) * self.relu_derivative(a1)
    dw1 = np.dot(x.T, dz1) / m
    db1 = np.sum(dz1, axis=0, keepdims=True) / m

    return dw1, db1, dw2, db2
```

8 Weight Updates

Weights are updated using gradient descent with L2 regularization.

$$w := w - \eta(\nabla w + \lambda w)$$

Listing 10: Weight Updates

```
def update_weights(self, dw1, db1, dw2, db2, lambda_reg=0.001):
    self.w1 -= self.learning_rate * (dw1 + lambda_reg * self.w1)
    self.b1 -= self.learning_rate * db1
    self.w2 -= self.learning_rate * (dw2 + lambda_reg * self.w2)
    self.b2 -= self.learning_rate * db2
```

9 Training Process

The model is trained over multiple epochs with mini-batch gradient descent.

Listing 11: Training Process

```
def train(self, x_train, y_train, epochs=100, batch_size=64):
    m = x_train.shape[0]
    for epoch in range(epochs):
        perm = np.random.permutation(m)
        x_train = x_train[perm]
        y_train = y_train[perm]

        for i in range(0, m, batch_size):
            x_batch = x_train[i:i+batch_size]
            y_batch = y_train[i:i+batch_size]

            a1, a2 = self.forward(x_batch)
            loss = self.compute_loss(y_batch, a2)
            dw1, db1, dw2, db2 = self.backward(x_batch, y_batch, a1, a2)
            self.update_weights(dw1, db1, dw2, db2)

        print(f'Epoch {epoch+1}, Loss: {loss}')
```

9.1 Training Details

- **Optimizer:** Gradient descent with a learning rate of 0.01.
- **Batch Size:** 64 samples per batch.
- **Epochs:** The model is trained for 100 epochs.
- **Regularization:** L2 regularization with $\lambda = 0.001$.

10 Model Evaluation

After training, the model's performance is evaluated on the validation set.

10.1 Accuracy Calculation

Listing 12: Model Evaluation

```
y_val_pred = mlp.predict(x_val)
accuracy = np.mean(y_val_pred == y_val)
print(f'Validation Accuracy: {accuracy * 100:.2f}%')
```

10.2 Results

The validation accuracy achieved is 95.64%.

11 Visualization of Predictions

Misclassified examples are identified and visualized.

Listing 13: Misclassified Examples

```
def find_misclassified(x_data, y_true, w1, b1, w2, b2):
    predictions = make_prediction(x_data, w1, b1, w2, b2)
    misclassified_indices = np.where(predictions != y_true)[0]
    print(f"Number of misclassified examples: {len(misclassified_indices)}")
    return misclassified_indices, predictions

def show_misclassified_example(index, x_data, y_true, predictions):
    current_image = x_data[index].reshape(1, -1)
    predicted_label = predictions[index]
    actual_label = y_true[index]
    print(f"Predicted Label: {predicted_label}")
    print(f"Actual Label: {actual_label}")
    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

12 Conclusion

The MLP model implemented from scratch demonstrates the core principles of neural networks, including data preprocessing, forward propagation, backpropagation, and weight updates. Despite its simplicity, the model achieves reasonable accuracy on the MNIST dataset.