# MNIST Linear Classification

Howard Sulisthio 124010990001

December 27, 2024

## 1. Introduction

The project focuses on classifying handwritten digits from the MNIST dataset using linear classifiers. This involves data preprocessing, training a linear classifier, and optimizing its performance using gradient descent and regularization techniques.

## 2. Methodology

### 2.1 Data Preparation

The MNIST dataset contains grayscale images of handwritten digits, each represented as a 28x28 pixel matrix. To prepare the data for classification, we perform the following steps:

#### 2.1.1 Loading Images and Labels

The MNIST dataset files are in IDX format. The following functions are used to load images and labels:

```
def load_mnist_images(filename):
    with open(filename, 'rb') as f:
        magic, num_images, rows, cols = struct.unpack(">IIII", f.read(16))
        images = np.fromfile(f, dtype=np.uint8).reshape(num_images, rows, cols)
    return images

def load_mnist_labels(filename):
    with open(filename, 'rb') as f:
        magic, num_labels = struct.unpack(">II", f.read(8))
        labels = np.fromfile(f, dtype=np.uint8)
    return labels
```

These functions parse the binary IDX files to extract the image and label data. The images are stored as 28x28 pixel matrices, and labels are integers representing the digit class.

#### 2.1.2 Preparing Images

The images are normalized to the range [0,1] and flattened into vectors for input to the linear classifier. The following function is used:

```
def prepare_images(images):
    return (images / 255.0).reshape(images.shape[0], -1)
```

This function divides pixel values by 255.0 to normalize them and reshapes each image into a 784-dimensional vector.

#### 2.1.3 Extracting Digit Pairs

Specific digit pairs are extracted from the dataset for binary classification. The following function is used:

```python
def extract_digits(images, labels, digit_pair):
    indices = np.isin(labels, digit_pair)
    filtered_images = images[indices]
    filtered_labels = labels[indices]
    binary_labels = (filtered_labels == digit_pair[1]).astype(int)
    return filtered_images, binary_labels
```

This function filters the dataset for the specified digit pair and converts the labels into binary values (0 for the first digit, 1 for the second).

### 2.1.4 Usage Example

Below is an example showing how these functions are used together:

```python
# Paths to MNIST files
train_images_path = "data/train-images-idx3-ubyte"
train_labels_path = "data/train-labels-idx1-ubyte"

# Load the dataset
train_images = load_mnist_images(train_images_path)
train_labels = load_mnist_labels(train_labels_path)

# Prepare the images
train_images = prepare_images(train_images)

# Extract digit pair (e.g., 4 and 9)
digit_pair = (4, 9)
filtered_images, binary_labels = extract_digits(train_images, train_labels, digit_pair)
```

This code demonstrates how to load, prepare, and filter the MNIST dataset for binary classification tasks.

## 2.2 Linear Classifier Definition

A linear classifier predicts labels using a linear transformation:

$$y = Wx + b \tag{1}$$

Here, $W$ represents weights and $b$ denotes biases. The predictions are converted to probabilities using the softmax function:

$$P(y_i|x) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{2}$$

where $z = Wx + b$.

Weights and biases are initialized using the Xavier initialization method:

$$\text{Limit} = \sqrt{\frac{2}{\text{Input Size} + \text{Output Size}}} \tag{3}$$

This ensures weights have an appropriate scale, preventing gradient issues during training. The implementation of Xavier initialization is as follows:

```python
def initialize_weights_xavier(input_size, output_size):
    limit = np.sqrt(2 / (input_size + output_size))  # Xavier scaling factor
    W = np.random.uniform(-limit, limit, size=(output_size, input_size))
    b = np.zeros(output_size)  # Biases initialized to zero
    return W, b
```

The implementation of the linear classifier in Python is as follows:

```python
def linear_classifier(X, weights):
    # Add a bias term to the input data
    bias = np.ones((X.shape[0], 1))  # Shape (m, 1)
    X_bias = np.hstack((bias, X))  # Shape (m, n+1)

    # Linear transformation
    logits = np.dot(X_bias, weights)  # Shape (m,)

    # Convert logits to binary predictions (0 or 1)
    predictions = (logits >= 0).astype(int)

    return predictions
```

This function implements the mathematical definition of the linear classifier, including adding a bias term to the input data and predicting binary labels based on the computed logits.

## 2.3 Loss Function with Regularization

The model uses cross-entropy loss with L2 regularization to balance classification accuracy and prevent overfitting:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij}) + \lambda\|W\|_2^2 \tag{4}$$

where $y$ is the true label, $\hat{y}$ is the predicted probability, and $\lambda$ is the regularization strength.

The **cross-entropy loss function** is used for multi-class classification because it quantifies the difference between the true label distribution and the predicted probability distribution. For each sample, the loss is calculated as the negative log probability of the correct class. Minimizing this loss encourages the model to assign higher probabilities to the correct class and lower probabilities to the incorrect classes. In this setting, there are two classes implemented for each digit pair.

**L2 regularization** penalizes large weights to prevent overfitting. It adds a term to the loss function proportional to the square of the L2 norm of the weight matrix $W$. This helps to prevent the model from becoming too complex and reduces the likelihood of overfitting, promoting generalization to unseen data.

Here is the code implementation:

```python
def compute_loss(X, y_true, W, b, reg_strength):
    # Compute the logits (raw class scores)
    logits = np.dot(X, W.T) + b

    # Compute the predicted probabilities using the softmax function
    probabilities = softmax(logits)

    # Cross-entropy loss
    cross_entropy_loss = -np.mean(np.sum(y_true * np.log(probabilities), axis=1))

    # L2 regularization term (penalizing large weights)
    l2_loss = np.sum(W ** 2)

    # Total loss: Cross-entropy loss + L2 regularization
    total_loss = cross_entropy_loss + reg_strength * l2_loss

    return total_loss
```

## 2.4 Gradient Descent Method

Gradients of the loss function w.r.t. $W$ and $b$ are computed as:

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{1}{N} \sum_i (\hat{y}_i - y_i) x_i^T + 2\lambda W \tag{5}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{1}{N} \sum_i (\hat{y}_i - y_i) \tag{6}$$

Weights and biases are updated using the gradient descent rule:

$$W \leftarrow W - \eta \frac{\partial \mathcal{L}}{\partial W}, \quad b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b} \tag{7}$$

where $\eta$ is the learning rate.

The gradient descent method is realized through two Python functions for computing gradient and optimizing parameters, as shown below:

```python
def compute_gradients(X, y_true, W, b, reg_strength):
    logits = np.dot(X, W.T) + b
    probabilities = softmax(logits)
    grad_logits = probabilities - y_true
    grad_W = np.dot(grad_logits.T, X) / X.shape[0] + 2 * reg_strength * W
    grad_b = np.sum(grad_logits, axis=0) / X.shape[0]
    return grad_W, grad_b
```

```python
def optimize_parameters(W, b, grad_W, grad_b, learning_rate):
    updated_W = W - learning_rate * grad_W
    updated_b = b - learning_rate * grad_b
    return updated_W, updated_b
```

## 2.5 Validation and Hyperparameter Tuning

The data is split into training and validation sets, where training and validation accuracies are computed to evaluate the model's performance. A standard validation ratio of 0.2 is used, and the Python implementation is shown below:

```python
def train_validation_split(images, labels, validation_ratio=0.2):
    num_samples = images.shape[0]
    indices = np.arange(num_samples)
    np.random.shuffle(indices)

    split_index = int(num_samples * (1 - validation_ratio))
    train_indices = indices[:split_index]
    val_indices = indices[split_index:]

    X_train, X_val = images[train_indices], images[val_indices]
    y_train, y_val = labels[train_indices], labels[val_indices]

    return X_train, X_val, y_train, y_val
```

```python
def compute_accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred) * 100
```

The hyperparameters (learning rate and regularization strength) are tuned by evaluating the classifier's performance on the validation set. Four sets of learning rate $0.01, 0.05, 0.1, 0.5$ and two set of regularization strength $0.001, 0.01$ are tested on this model.

```python
for lr in learning_rates:
    for reg_strength in regularization_strengths:
        W, b = initialize_weights_xavier(num_features, num_classes)
        for epoch in range(epochs):
            grad_W, grad_b = compute_gradients(X_train, y_train, W, b, reg_strength)
            W, b = optimize_parameters(W, b, grad_W, grad_b, lr)
```

# 3. Results and Discussion

There are essentially two metrics that were used in to evaluate the model which are:

- Training loss against epochs at different hyperparameters
- Training accuracy and validation accuracy against epochs for one hyperparameter

Training loss against epochs or iterations for different hyperparameters is useful to find the best configuration that minimizes loss and prevents overfitting. This visualization helps in detecting overfitting by noticing if the loss decreases too much without a corresponding improvement in validation performance, and it indicates when the model has converged to a stable solution.

On the other hand, tracking both training and validation accuracy against epochs helps in identifying overfitting and implementing early stopping to prevent it, ultimately aiding in selecting the best model that achieves a good balance between fitting the training data and generalizing to new data.

100 iterations or epochs will be implemented, which should be sufficient to see convergence in the following graphs.

## 3.1 Training Loss

The following figures illustrate the training loss vs. epochs for different digit pairs under various hyperparameter settings:

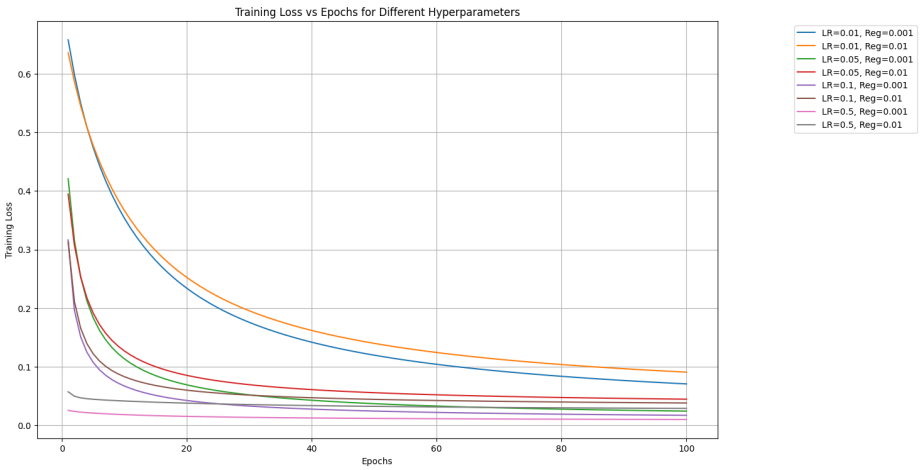


Figure 1: Training Loss vs Epochs for digits 0 and 1

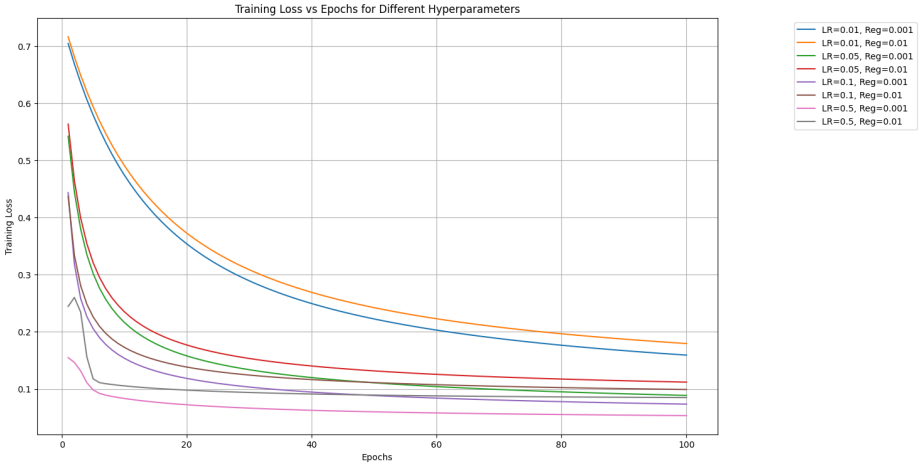


Figure 2: Training Loss vs Epochs for digits 2 and 7

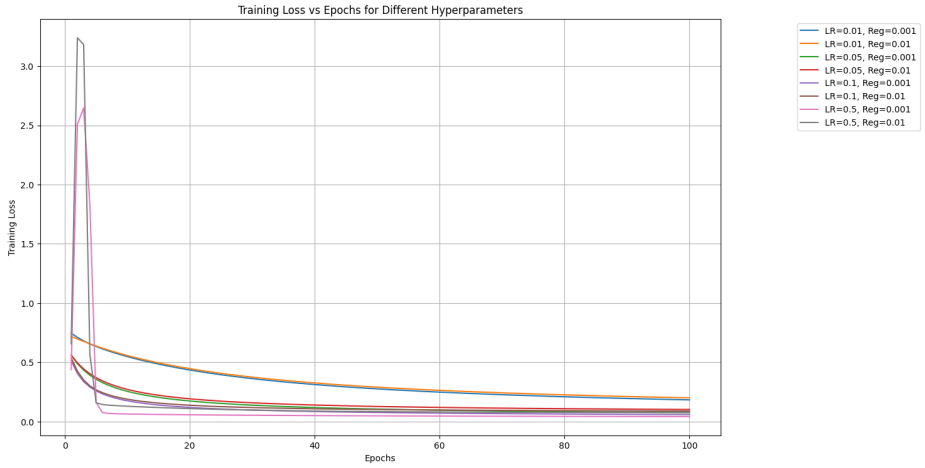


Figure 3: Training Loss vs Epochs for digits 4 and 6

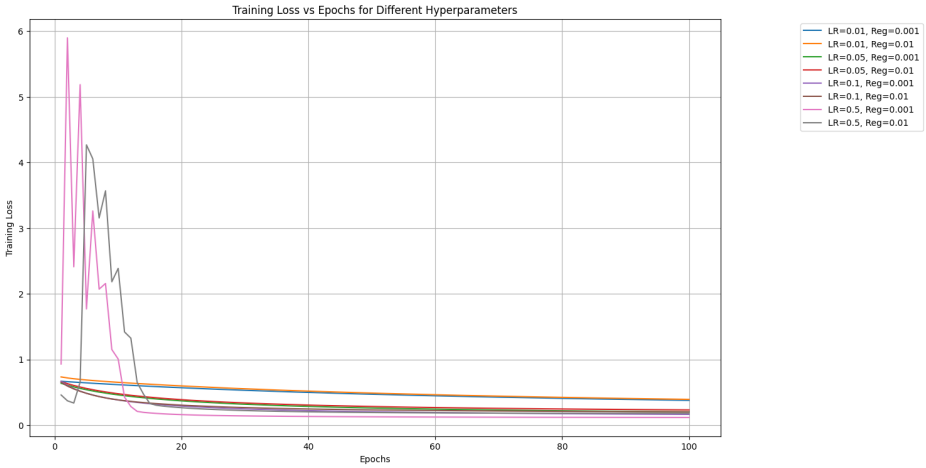


Figure 4: Training Loss vs Epochs for digits 4 and 9

**Analysis**

The training loss for all digit pairs generally follows a steady decline, indicating successful optimization. Key observations include:

- **Digits 0 and 1:** Rapid convergence with learning rates around 0.1 and smooth stabilization with higher regularization strengths.
- **Digits 2 and 7:** Some instability at higher learning rates (e.g., LR = 0.5) but overall steady convergence.
- **Digits 4 and 6:** Higher visual similarity between these digits requires careful tuning of hyperparameters, with LR = 0.1 providing the best balance.
- **Digits 4 and 9:** High learning rates cause spikes initially, but regularization (Reg = 0.01) smoothens the loss curve, ensuring better generalization.

Lower learning rates (LR = 0.01) provide stability across all digit pairs but at the cost of slower convergence. Higher regularization helps stabilize training and prevents overfitting, especially for digit pairs with high visual similarity.

## 3.2 Training and Validation Accuracy vs Epochs

The following figures illustrate training and validation accuracy vs. epochs for the best hyperparameter settings for different digit pairs:

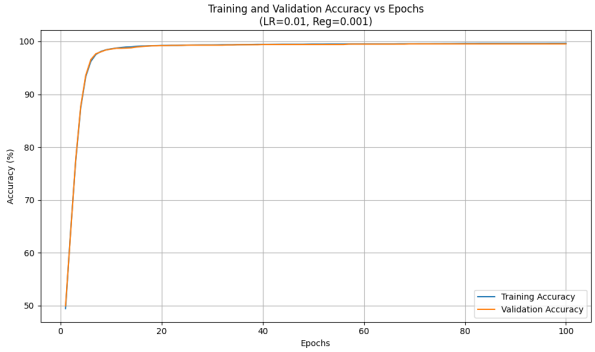


Figure 5: Training and Validation Accuracy vs. Epochs for digits 0 and 1

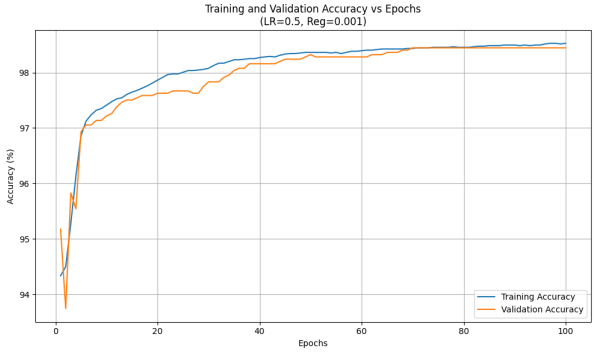


Figure 6: Training and Validation Accuracy vs. Epochs for digits 2 and 7

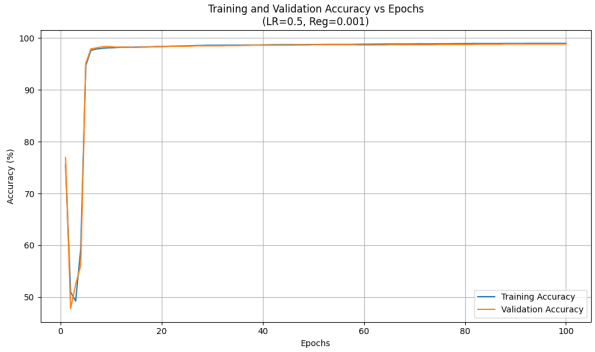


Figure 7: Training and Validation Accuracy vs. Epochs for digits 4 and 6

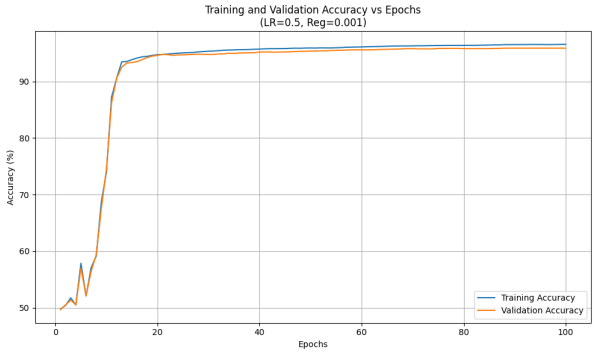


Figure 8: Training and Validation Accuracy vs. Epochs for digits 4 and 9

**Analysis**

Training and validation accuracy curves provide insights into the model's generalization ability and optimization performance:

- **Digits 0 and 1:** Training and validation accuracies converge to near 100%, indicating excellent generalization with minimal overfitting.
- **Digits 2 and 7:** Slightly higher training accuracy than validation accuracy suggests mild overfitting, but the overall performance is strong.
- **Digits 4 and 6:** The higher visual similarity between digits increases classification complexity, but the model achieves high accuracy with stable convergence.
- **Digits 4 and 9:** Minimal gap between training and validation accuracy highlights effective regularization and learning rate tuning for this challenging pair.

Higher learning rates (e.g., LR = 0.5) enable rapid convergence, while regularization (Reg = 0.001) ensures stable training and prevents overfitting across all digit pairs.

## 3.4 Final Results and Discussion

The performance of the linear classifier for different digit pairs was evaluated based on the best hyperparameters and the corresponding validation accuracy. The results are summarized in Table 1.

| Digit Pair | Learning Rate (LR) | Regularization Strength (Reg) | Best Validation Accuracy (%) |
|---|---|---|---|
| 0 and 1 | 0.01 | 0.001 | 99.64 |
| 2 and 7 | 0.5 | 0.001 | 98.45 |
| 4 and 6 | 0.5 | 0.001 | 98.85 |
| 4 and 9 | 0.5 | 0.001 | 95.89 |

Table 1: Best Hyperparameters and Validation Accuracy for Each Digit Pair

**Analysis of Digit Pairs**

For digits 0 and 1, they achieved the highest validation accuracy (99.64%) among all pairs. The pair is visually distinct, with digit 0 being a closed loop and digit 1 being a vertical stroke, making them easy to classify. Misclassification is minimal due to the clear geometric separation in feature space. The model performs well even with a low learning rate (LR = 0.01), as the simplicity of this task requires less aggressive optimization.

Digit pairs 2 and 7 achieved a validation accuracy of 98.45%, slightly lower than the simpler pairs. Misclassifications occur when 7s have incomplete strokes or 2s have elongated tails, as these features overlap in some cases. A higher learning rate (LR = 0.5) is required for effective convergence, but regularization (Reg = 0.001) ensures stability. The structural similarity between digits 2 and 7 introduces additional challenges for the linear classifier.

Digit pairs 4 and 6 achieved a validation accuracy of 98.85%, despite the visual similarity between the digits. Misclassifications occur when 6s have incomplete loops or 4s have rounded edges, as these traits lead to overlapping features. The model benefits from a higher learning rate (LR = 0.5) and strong regularization (Reg = 0.001), allowing it to generalize well without overfitting. This pair demonstrates the capability of linear classifiers to distinguish between subtle structural differences with proper hyperparameter tuning.

Digit pairs 4 and 9 achieved the lowest validation accuracy (95.89%) among the digit pairs. Misclassifications arise from overlapping features, such as closed loops in 4s and open-ended loops in 9s. The structural overlap challenges the linear classifier, as linear decision boundaries struggle to separate such features effectively. While a high learning rate (LR = 0.5) aids convergence, the inherent difficulty of this task limits the achievable accuracy.

**General Observations**

- The linear classifier performs exceptionally well for digit pairs with distinct structural features (e.g., 0 and 1).

- For digit pairs with overlapping features (e.g., 4 and 9), performance is lower due to the limitations of linear decision boundaries.
- Higher learning rates (e.g., LR = 0.5) lead to faster convergence but require careful regularization (Reg = 0.001) to prevent overfitting.
- Regularization is crucial in stabilizing the model and ensuring generalization across all digit pairs.
- The simplicity of the model makes it computationally efficient, but non-linear methods might be required for more complex pairs.

### Model Visualizations

The following figures illustrate some examples for each digit pair. The labels consist of two variables being $P$ representing prediction values and $T$ representing true values, where they are output in boolean values 0 and 1 for each digit class. If $P$ and $T$ values match, then the label is green, and red for vice-versa. In here, it can be observed that digits 4 and 9 has the most mismatches out of all the digit pairs.

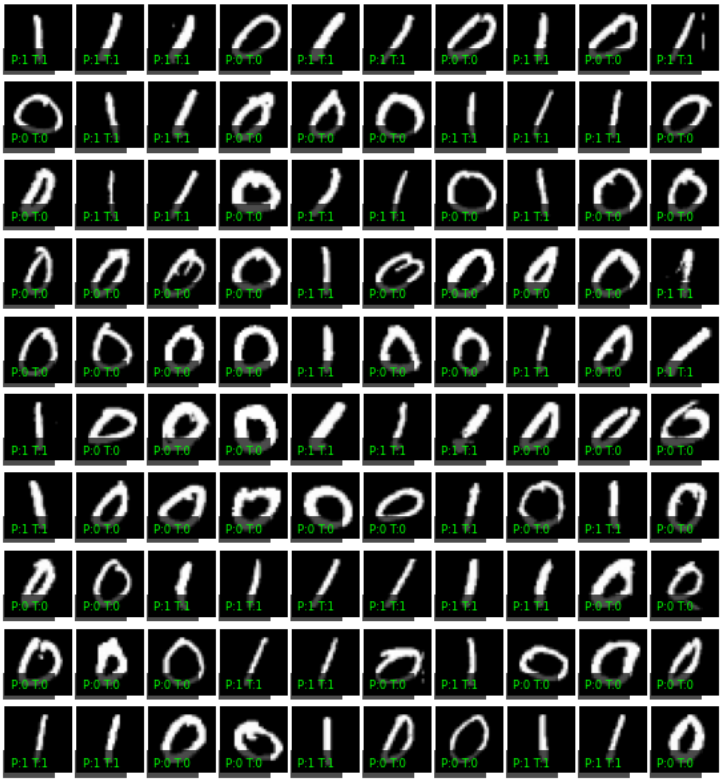

Figure 9: First 100 images for digits 0 and 1

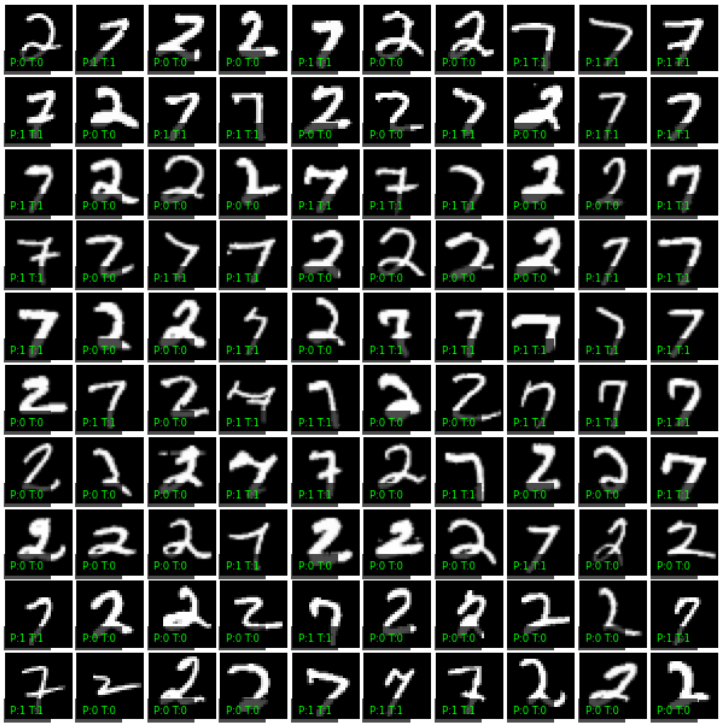

Figure 10: First 100 images for digits 2 and 7

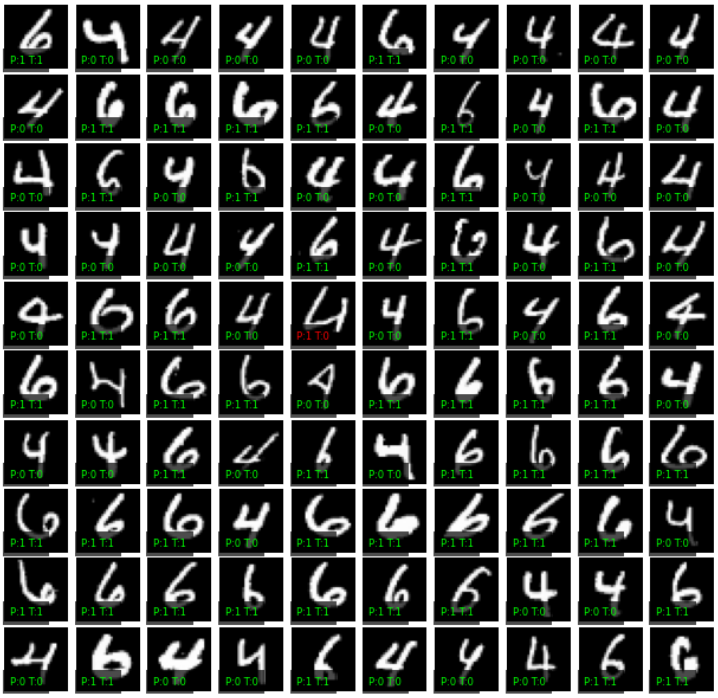

Figure 11: First 100 images for digits 4 and 6

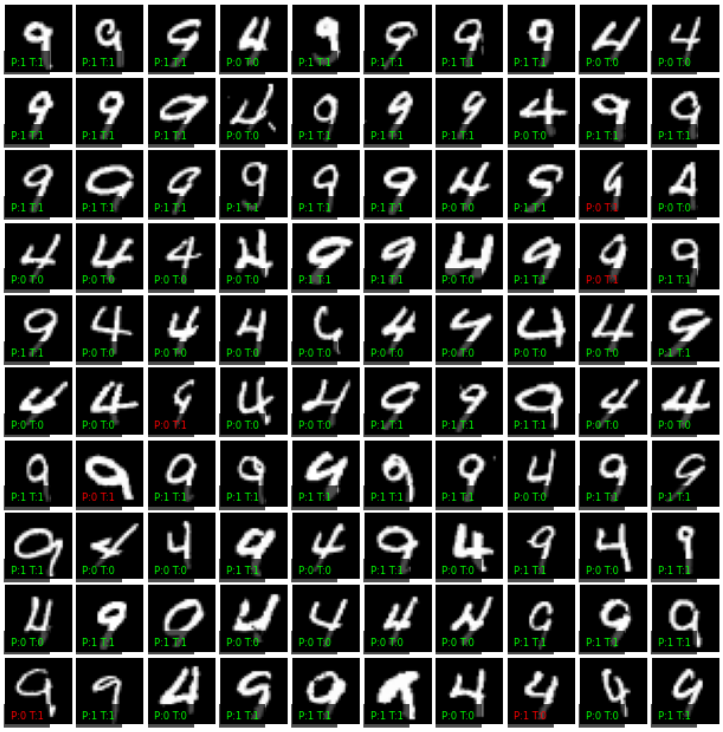

Figure 12: First 100 images for digits 4 and 9

## 4. Conclusion

This project successfully demonstrated the application of linear classifiers to classify handwritten digits from the MNIST dataset. The methodology involved data preprocessing, designing a linear classifier, implementing a regularized cross-entropy loss function, and optimizing the model using gradient descent. Hyperparameter tuning was conducted to identify the best combination of learning rate and regularization strength for each digit pair.

Key results indicate that the linear classifier performs exceptionally well for digit pairs with distinct structural features, achieving validation accuracies as high as 99.64% for digits 0 and 1. However, its performance declines for digit pairs with significant feature overlap, such as 4 and 9, where the best validation accuracy was 95.89%. These results highlight the limitations of linear classifiers in handling complex classification tasks with non-linear decision boundaries.

Despite its simplicity and computational efficiency, the linear classifier has inherent limitations:

- It struggles to distinguish digit pairs with overlapping structural features due to the constraints of linear decision boundaries.
- The model's performance depends heavily on hyperparameter tuning, especially for challenging digit pairs.
- Misclassifications arise primarily in cases where the digits share visually similar features, indicating the need for more expressive models.

Future work could focus on addressing these limitations by incorporating non-linear methods such as kernel-based techniques or neural networks. Additionally, data augmentation strategies can be explored to improve the classifier's robustness to variations in handwriting styles. Finally, extending the model to handle multi-class classification using techniques like one-vs-rest or one-vs-one can broaden its applicability to the entire MNIST dataset.

In conclusion, this project highlights the strengths and weaknesses of linear classifiers in digit classification and sets a foundation for exploring more advanced methods for challenging classification tasks.