

2022 – 2023 第一学期 实验报告

课程编号： 2801000060， 课程名称： 机器人学导论， 主讲教师： 邱国平

学号： 2022280485	姓名： 贾苏健	专业年级： 电子信息工程
----------------	---------	--------------

Coursework 1 (20%)

教师评语/成绩



目录

I. Experimental Principles.....	4
1. Single-Layer Perceptron.....	4
2. Feature Vector.....	4
3. Training Process.....	4
4. Evaluation of Performance.....	8
5. Multiclass Classification.....	9
II. Experiment Process	10
1. Implementation Process.....	10
(1) Data Set Retrieval	10
(2) Display of Dataset Images	11
(3) Dataset Splitting	13
(4) Label Transformation	13
(5) Training Function	14
(6) Training Results and Their Visualization	15
(7) Validation Set Testing	16
(8) Validation Set Plotting Charts	17
2. Implementing the differentiation between the digit 6 and other digits in Python.....	18
3. Implementing the differentiation between the digit 6 and other digits in MATLAB.....	19
4. Implementing the differentiation between the digit 2 and digit 5 in MATLAB.....	20
5. Implementing the differentiation between the digits 1, 2, 5, and 7 in MATLAB.....	20
6. Implementing the differentiation between digits 0 - 9 in MATLAB.....	20
7. Implementing the differentiation between digits 0-9 in Python.....	20
8. Issues Encountered, Resolutions, and Some Details.....	21
(1) Data Set Loading and Display Issues	21
(2) Data Set Splitting	22
(3) MATLAB Preallocation Memory Issue	22
(4) Utilize Python Library Functions to Reduce Time Complexity When Calculating Dot Products	22



(5) Learning Rate Selection Issue	22
(6) Confusion Matrix Plotting Issue	22
9. Answers to Experimental Requirements.....	25
(1) Initial Weight Selection Issue	25
(2) Result Evaluation	25
III. Experiment Summary.....	26
1. Evaluation of Single-Layer Perceptron Performance.....	26
2. Building a Multi-Layer Perceptron for Multiclass Classification Using a Single-Layer Perceptron.....	27
3. Programming with MATLAB.....	27

I. Experimental Principles

1. Single-Layer Perceptron

A single-layer perceptron is a type of artificial neural network (ANN). Using a single-layer perceptron, we can automatically recognize and classify data, as demonstrated in this experiment where we identified and classified MNIST handwritten digits. However, it's important to note that single-layer perceptrons are primarily suitable for linearly separable problems, meaning they can classify data into two categories. In our experiment, we attempted to distinguish the digit 6 from other digits and the digit 2 from the digit 5, among other tasks. Before using a perceptron for data classification, it must undergo a training process. This training process involves using a large dataset. During training, also referred to as the learning process, we initially label the data in the training set and then continuously update the weights based on the features of the dataset to better adapt to its characteristics.

2. Feature Vector

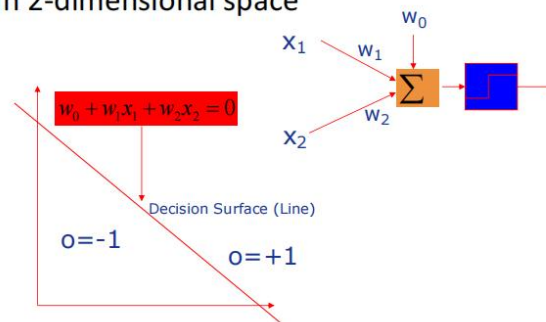
Regarding **feature vectors**, I understand them as numerical combinations that fully and uniquely represent a piece of data. For example, in the MNIST handwritten dataset, each digit image has a size of 28x28, totaling 784 pixels. The combination of these 784 values is sufficient to completely and uniquely represent each image, constituting the feature vector that distinguishes MNIST handwritten digits.

3. Training Process

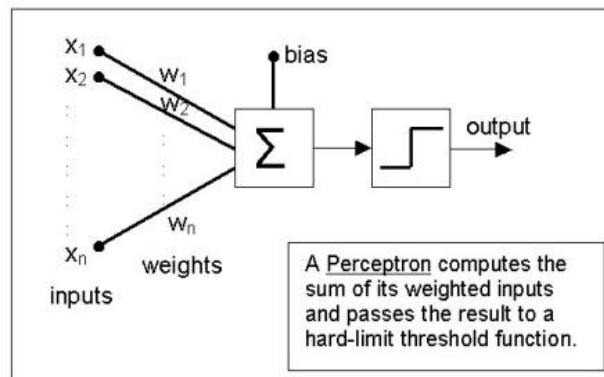
To better explain the training process, let's consider a two-dimensional feature space. Assuming we have a sufficiently large dataset, and each data point can be uniquely and completely represented by two features, denoted as x_1, x_2 . Representing these two features as coordinates, we get a point (x_1, x_2) , and the plane in which this point resides is called the **feature plane** (or **feature space** for multidimensional features). Thus, each data point in the dataset can be represented as a point in this feature plane. The goal is to recognize and classify all data points in the dataset into two categories, corresponding to the feature points on the plane.

For separating points into two categories on a plane, we can easily achieve this with a curve called the **decision curve**, and in multidimensional feature space, it is referred to as the **decision plane**. One side of the curve represents one category, and the other side represents the other category. To determine which category a point belongs to, we can substitute the point's coordinates into the function corresponding to the equation of the curve and observe the sign of the result. However, dealing with complex curves can be challenging. Therefore, in this context, we focus on studying straight lines. If a set of data points can be divided by a straight line, it is referred to as a **linearly separable problem**. This approach is also applicable in multidimensional feature space.

- In 2-dimensional space



In a two-dimensional plane, representing a straight line typically requires two parameters, such as $y = kx + b$. However, here, to ensure the equivalence of each feature, we use three parameters, expressed in the equation $Ax + By + C = 0$, corresponding to the function $f(x_1, x_2) = w_2x_2 + w_1x_1 + w_0$, where the three parameters are w_1 , w_2 , and w_0 . Our goal is to adjust these three parameters to find the most suitable values for w_1 , w_2 , and w_0 , allowing the dataset to be divided into two classes based on labels to the greatest extent. Here, we refer to w_1 and w_2 as **weights** and w_0 as **bias**.



Before searching for the most suitable weights and biases, we must first obtain crucial information: the **labels**. Labels represent the actual categories corresponding to the data. Only after obtaining the labels can we continuously adjust the weights to make the model's output, $f(x_1, x_2)$, more closely approximate the actual label values. For binary classification, we typically use a curve on a graph to divide the data into two regions, where the sign of the function value of the curve's equation determines the side to which a data point belongs. For positive and negative differentiation, we can directly use the *sign* function, where a positive value is represented as +1, and a negative value is represented as -1. This result is then compared with the true label to assess whether the perceptron's judgment is correct. To correspond with this, we usually choose labels as +1 and -1 to meet the needs of binary classification.

Once everything is prepared, we can begin training the perceptron. Our goal is to adjust the model's weights to make the output $f(x_1, x_2)$ as close as possible to the true labels or, in other words, minimize the difference between the output $f(x_1, x_2)$ and the true labels. The magnitude of this difference is measured by the **loss function** (also known as the **cost function**). The smaller the value of the loss function, the better the perceptron's perception. Since the perceptron's perception is only related to weights and biases, the loss function can be expressed as $W(w_0, w_1, w_2, \dots, w_n)$, where n is the dimensionality of the feature vector.

To represent the degree of error, we first considered the simplest *sign* function loss, which is given by:

$$W(w_0, w_1, w_2, \dots, w_n) = \text{sign}((w_0 + \sum_{i=1}^n w_i x_i)) - d$$

where d represents the true label of the data. Although such a loss function is computationally simple and easy to understand, it imposes the same penalty for all errors, regardless of the magnitude of the error. This significantly reduces the accuracy of weight updates and may encounter issues such as discontinuity and non-differentiability in the gradient. Therefore, it is not recommended for use.



1. Set the weights to small random values, e.g., in the range (-1, 1)
2. Present X , and calculate

$$R = w_0 + \sum_{i=1}^n w_i x_i \quad o = \text{sign}(R) = \begin{cases} +1, & \text{if } R > 0 \\ -1, & \text{otherwise} \end{cases}$$

3. Update the weights

$$w_i \leftarrow w_i + \eta(d - o)x_i, i = 1, 2, \dots, n$$

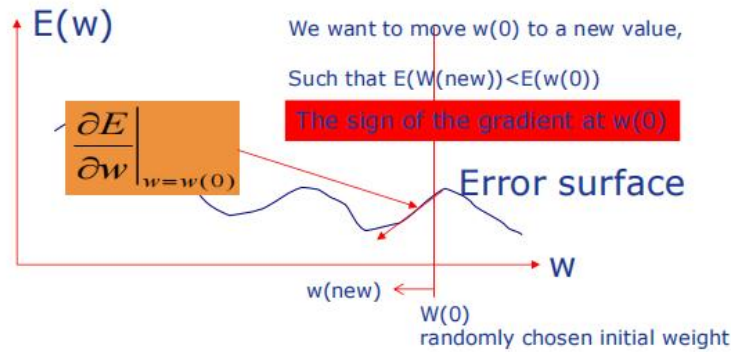
$$0 < \eta < 1 \text{ is the training rate} \quad x_0 = 1 \text{ (constant)}$$

4. Repeat by going to step 2

To facilitate more reasonable weight updates, we employ differentiable loss functions and the gradient descent algorithm. Commonly used loss functions include Mean Squared Error (MSE), Cross-Entropy Loss, Hinge Loss, Logarithmic Loss, and others. Here, we have chosen Mean Squared Error as the loss function, updating the weights through gradient descent. This process allows the model to learn appropriate weights based on the degree of error in the data, aiming to make the model's output as close as possible to the actual labels. [1]

The gradient descent process involves updating the size of the loss function, $W(w_0, w_1, w_2, \dots, w_n)$. Taking two-dimensional weights as an example, you can imagine this process in a three-dimensional space where w_i is the x-axis, w_j is the y-axis, and W is the z-axis, as if searching for the optimal path to the valley. The loss function resembles a hill, and our goal is to find the lowest point of the hill, where the loss function is minimized. By observing the slope at the current position, i.e., the gradient, we take a step in the opposite direction of the gradient, effectively updating the model's position. The learning rate determines the size of the step; an appropriate learning rate is akin to choosing an appropriate stride. We repeat this process until the slope is gentle enough, indicating a small gradient and signifying that we have reached the bottom of the hill, with the model parameters adjusted sufficiently. Gradient descent is a method that continuously observes the slope and adjusts the position along the steepest direction, ultimately

finding the model parameters that minimize the loss function value. The same principles apply to multidimensional loss functions.



4. Evaluation of Performance

The demonstration of training effectiveness can be achieved by plotting charts depicting the change in error rates over time for the training set. As for the test set, it is understood that in binary classification, the main methods for evaluation include ROC curves and confusion matrices. Therefore, the ROC curve is a graphical tool for measuring the classification performance of a model, illustrating the relationship between true positive rate and false positive rate. Meanwhile, the confusion matrix is a table used to compare the model's actual prediction results with the true labels.

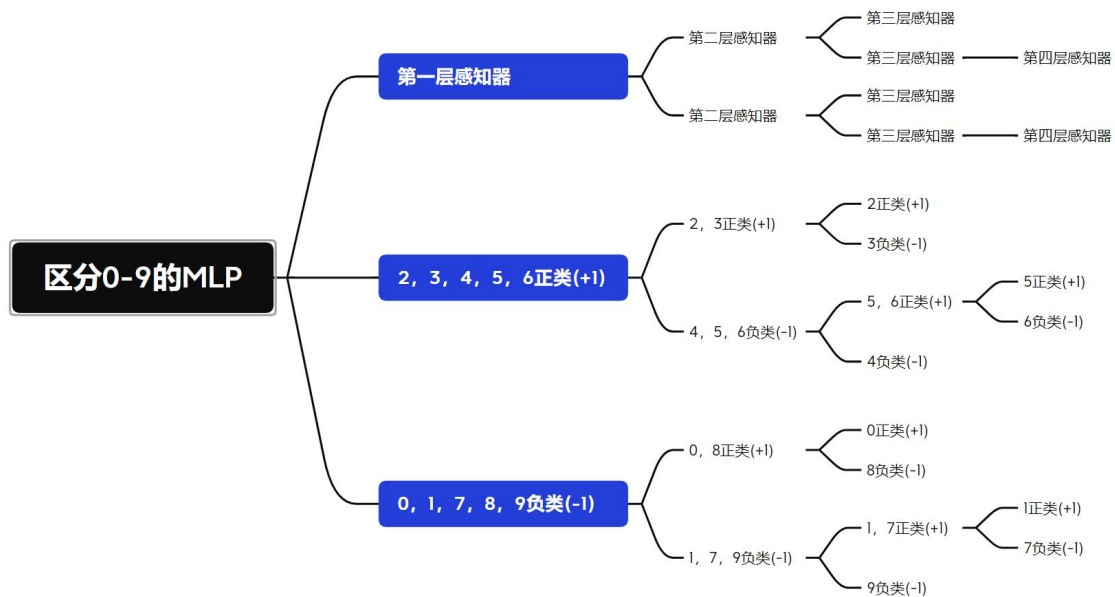
混淆矩阵		真实值	
		Positive	Negative
预测值	Positive	TP	FP
	Negative	FN	TN

Explanation of True Positive Rate and False Positive Rate. The True Positive Rate (TPR) refers to the proportion of actual positive instances correctly identified by the model, calculated as $TP/(TP + FN)$. On the other hand, the False Positive Rate (FPR) represents the proportion of instances incorrectly identified as positive by the model among actual negative samples, calculated as $FP/(FP + TN)$. On the ROC curve, the horizontal axis represents FPR, and the vertical axis represents TPR. The closer the ROC curve is to the top-left corner, the better the model's performance. The confusion matrix

provides more detailed classification results, where TP is the number of true positive instances, FP is the number of false positive instances, FN is the number of false negative instances, and TN is the number of true negative instances. Utilizing these metrics and charts comprehensively helps assess the performance of a classification model.

5. Multiclass Classification

The principles and evaluation methods of a single-layer perceptron have been explained; however, it is important to note that this type of perceptron is only suitable for binary classification problems. If our task involves achieving multiclass classification, it can be accomplished by constructing multiple single-layer perceptrons. Below is my implementation approach.



II. Experiment Process

1. Implementation Process

(1) Data Set Retrieval

Before downloading and using the dataset, it is crucial to understand its structure, which can be obtained from the following website:

<http://yann.lecun.com/exdb/mnist/>. It is important to note that in both the training image set and the test image set, the first 16 bytes contain the Magic Number and the Number of Images, not the actual image data. Similarly, in the training label set and the test label set, the first 8 bytes contain the Magic Number, not the actual label data. Therefore, it is necessary to discard these irrelevant parts when processing the data. Refer to the diagram below for detailed information:

TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....	unsigned byte	??	label
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

TRAINING SET IMAGE FILE (train-images-idx3-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....	unsigned byte	??	pixel
xxxx	unsigned byte	??	pixel

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).

TEST SET LABEL FILE (t10k-labels-idx1-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	10000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....	unsigned byte	??	label
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

TEST SET IMAGE FILE (t10k-images-idx3-ubyte):

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	10000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....	unsigned byte	??	pixel
xxxx	unsigned byte	??	pixel

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).

Python:

```
def load_mnist_images(filename):
    with open(filename, 'rb') as f:
        data = np.fromfile(f, dtype=np.uint8, count=-1)
        return data[16:].reshape(-1, 28*28) / 255.0

def load_mnist_labels(filename):
    with open(filename, 'rb') as f:
        data = np.fromfile(f, dtype=np.uint8, count=-1)
        return data[8:]
```

Matlab:

```

1 function data = load_mnist_images(filename)
2 % 数据集图像加载函数
3 fid = fopen(filename, 'rb');
4 data = fread(fid, 'uint8');
5 fclose(fid);
6 data = data(17:end) / 255.0;
7 data = reshape(data, 28*28, []).';
8 end

```

Note: It is unnecessary to transpose the images when reading them in this context.

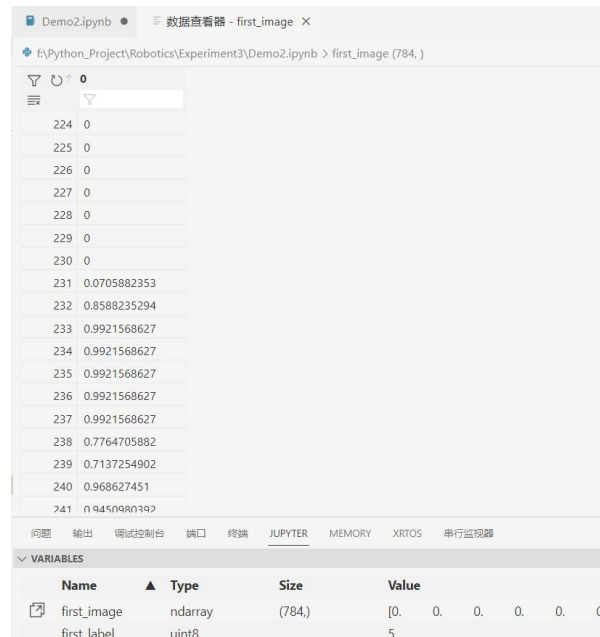
```

1 function data = load_mnist_labels(filename)
2 % 数据集标签加载函数
3 fid = fopen(filename, 'rb');
4 data = fread(fid, 'uint8');
5 fclose(fid);
6 data = data(9:end);
7 end

```

(2) Display of Dataset Images

It can be observed in the VS Code variable viewer that each image read is a one-dimensional array, as shown in the diagram below. Before displaying, it is necessary to convert the one-dimensional array with a length of 784 into a two-dimensional array of 28x28. This step can also be integrated directly into the reading function.



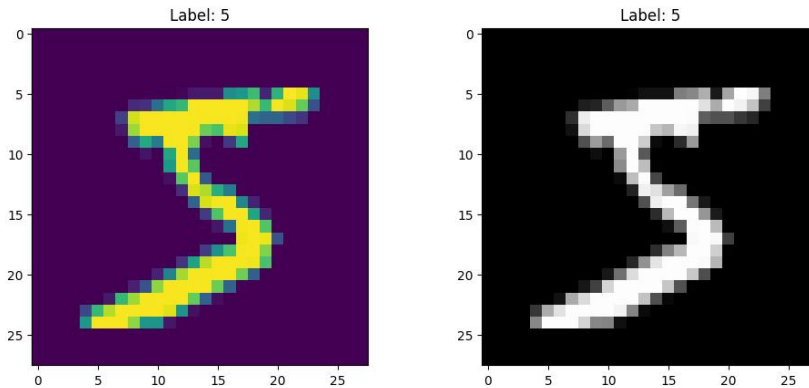
The screenshot shows the VS Code variable viewer for a Jupyter Notebook. The variable 'first_image' is displayed as a 1D array of 784 elements. The first few elements are 0, and the last few are 0.968627451 and 0.945098032. Below the array, the 'VARIABLES' section shows the variable 'first_image' as an 'ndarray' of size '(784,)' and 'first_label' as a 'uint8' of value '5'.

Name	Type	Size	Value
first_image	ndarray	(784,)	[0. 0. 0. 0. 0. 0.]
first_label	uint8		5

Grayscale images typically refer to images where each pixel has only one grayscale value, as opposed to color images with multiple color channels. For the MNIST dataset, each image is a **single-channel grayscale image**. It is appropriate to use a grayscale color map (cmap='gray') when displaying grayscale images. This is because each pixel value in a grayscale image

represents a level of grayscale between black (0) and white (255 or 1.0). By using a grayscale color map, it is easier to distinguish differences in grayscale levels when displaying the images.

If you choose to use other color maps, the images may be displayed in pseudocolor, which may not be the desired effect, especially for single-channel grayscale images. The specific display effect is shown in the diagram below.



Python:

```
# 将图像数据还原为28x28的形状
image_resaped = first_image.reshape(28, 28)

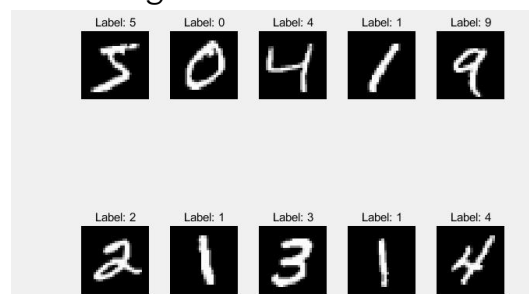
plt.imshow(image_resaped, cmap='gray') # 使用灰度颜色映射
plt.title(f"Label: {first_label}")
plt.show()
```

In the above code, `plt.imshow(image_resaped, cmap='gray')` is used to display the image using a grayscale color map. This ensures that the image is presented in black and white, reflecting the grayscale levels.

Matlab:

```
data_load
for i = 1:10
    subplot(2, 5, i);
    img = reshape(train_images_array(i, :), [28, 28]); % 将图像重新转换为 28x28 的形状
    imshow(img, 'InitialMagnification', 'fit'); % 转置图像矩阵并设置放大方式为适应窗口
    colormap(gray); % 灰度图
    title(['Label: ', num2str(train_labels_array(i))]);
end
```

Displaying the First Ten Digits in Matlab:



(3) Dataset Splitting

The MNIST official website has already provided training and test sets, and in practice, there is no need to perform dataset splitting. However, this experiment requires the division of the 60,000 data points in the training dataset into custom training and test sets. The specific requirements for CourseWork3 experiment are as follows:

guidelines:

- Split the data into a training set and a testing set.

The method to achieve this requirement is relatively simple: shuffle the dataset and then split it into two parts according to a specific ratio. Specifically, the code achieves this by splitting the shuffled dataset from the beginning. To ensure **reproducibility** of the results, I used the `random_state` random seed in the function's input variables. Here is the specific implementation of the code.

```
additional_questions.py x adline.py x
35 def split_dataset(x, y, test_size=0.2, random_state=None):
36     # 获取数据集的长度
37     data_size = len(x)
38
39     # 根据比例计算验证集的大小
40     val_size = int(test_size * data_size)
41
42     # 设置随机种子
43     if random_state is not None:
44         np.random.seed(random_state)
45
46     # 随机打乱数据集
47     indices = np.random.permutation(data_size)
48
49     # 划分数据集
50     val_indices = indices[:val_size]
51     train_indices = indices[val_size:]
52
53     # 获取划分后的数据
54     x_train, x_val = x[train_indices], x[val_indices]
55     y_train, y_val = y[train_indices], y[val_indices]
56
57     return x_train, x_val, y_train, y_val
```

(4) Label Transformation

The labels of the original dataset correspond to the digits represented by the data, i.e., 0-9. However, when performing binary classification, using the actual digits as labels is not ideal. In binary classification, we typically divide the data into two regions by drawing a curve on the image, where the sign of the curve's function values determines the side to which a data point belongs. To differentiate between positive and negative in the perceptron, we directly use the sign function, where positive values are represented as +1, and

negative values as -1. This result is compared with the true labels to determine if the perceptron's decision is correct. To align with this approach, we usually designate the labels as +1 and -1 to meet the requirements of binary classification. Below are label transformations for different experimental requirements:

Label transformation for distinguishing digit 6 from other digits.

```
170 train_label_binary = np.where(train_labels_array == 6, 1, -1)
```

Label transformation for distinguishing digit 2 from digit 5.

```
4 train_label = (train_labels_array == 6) * 2 - 1;
5 test_label = (test_labels_array == 6) * 2 - 1;
```

Label transformation for distinguishing digits 1, 2, 5, and 7.

% 提取数字 '2'、'5'、'1' 和 '7' 的样本

```
indices_2571 = find(train_labels_array == 2 | train_labels_array == 5 | train_labels_array == 1 | train_labels_array ==
```

% 将 '2' 和 '5' 视为正类 (+1)，'1' 和 '7' 视为负类 (-1)

```
labels_parent = (labels == 2 | labels == 5) * 2 - 1;
```

Label transformation for distinguishing digits 0-9 with different perceptrons in different layers.

% 将 '2.3.4.5.6' 视为正类 (+1)，'0.1.7.8.9' 视为负类 (-1)

```
labels_layer1 = (train_labels == 2 | ...
                 train_labels == 3 | ...
                 train_labels == 4 | ...
                 train_labels == 5 | ...
                 train_labels == 6) * 2 - 1;
```

% 将 '2.3' 视为正类 (+1)，'4.5.6' 视为负类 (-1)

```
binary_23456 = (labels_23456 == 2 | labels_23456 == 3) * 2 - 1;
```

% 将 '0.8' 视为正类 (+1)，'1.7.9' 视为负类 (-1)

```
binary_01789 = (labels_01789 == 0 | labels_01789 == 8) * 2 - 1;
```

% 将 '2' 视为正类 (+1)，'3' 视为负类 (-1)

```
binary_23 = (labels_23 == 2) * 2 - 1;
```

% 将 '5.6' 视为正类 (+1)，'4' 视为负类 (-1)

```
binary_456 = (labels_456 == 5 | labels_456 == 6) * 2 - 1;
```

% 将 '0' 视为正类 (+1)，'8' 视为负类 (-1)

```
binary_08 = (labels_08 == 0) * 2 - 1;
```

% 将 '1.7' 视为正类 (+1)，'9' 视为负类 (-1)

```
binary_179 = (labels_179 == 1 | labels_179 == 7) * 2 - 1;
```

% 将 '5' 视为正类 (+1)，'6' 视为负类 (-1)

```
binary_56 = (labels_56 == 5) * 2 - 1;
```

% 将 '1' 视为正类 (+1)，'7' 视为负类 (-1)

```
binary_17 = (labels_17 == 1) * 2 - 1;
```

(5) Training Function

The development of the training function is primarily based on the theoretical foundation outlined in the training principles section. I have

created two training functions, one of which utilizes the sign function as the loss function. Below is the principle and a code example:

Perceptron – Training Algorithm

The Procedure is as follows

1. Set the weights to small random values, e.g., in the range $(-1, 1)$
2. Present X_i and calculate

$$R = w_0 + \sum_{i=1}^n w_i x_i \quad o = \text{sign}(R) = \begin{cases} +1, & \text{if } R > 0 \\ -1, & \text{otherwise} \end{cases}$$

3. Update the weights

$$w_i \leftarrow w_i + \eta(d - o)x_i, i = 1, 2, \dots, n$$

$0 < \eta < 1$ is the training rate $x_0 = 1$ (constant)

4. Repeat by going to step 2

```
47 # 训练函数
48 def train(x, y, weights_init_val=None, learning_rate=0.01, epochs=100, stopping_threshold=0.018):
49     input_variables_size = len(x[0])
50     weight_i, weight_0 = initialize_weights(input_variables_size, weights_init_val)
51
52     errors = []
53
54     for epoch in range(epochs):
55         error = 0
56         for xi, target in zip(x, y):
57             dot_product = sum(xi_j * weight_i_j for xi_j, weight_i_j in zip(xi, weight_i))
58             prediction = np.sign(dot_product + weight_0)
59             update = learning_rate * (target - prediction)
60             weight_i += update * xi
61             weight_0 += update
62             error += int(update.item() != 0.0)
63
64         errors.append(error / len(y)) # 训练集的样本个数
65
66         if error / len(y) < stopping_threshold:
67             break
68
69     return errors, weight_i, weight_0
```

Gradient of ADLINE Error Functions

$$E(W) = \frac{1}{2} \sum_{k=1}^K (d(k) - o(k))^2$$

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial w_i} \left(\frac{1}{2} \sum_{k=1}^K (d(k) - o(k))^2 \right)$$

$$= \frac{1}{2} \sum_{k=1}^K \left(\frac{\partial E}{\partial w_i} (d(k) - o(k))^2 \right)$$

$$= \frac{1}{2} \sum_{k=1}^K \left(2(d(k) - o(k)) \frac{\partial E}{\partial w_i} (d(k) - o(k)) \right)$$

$$= \sum_{k=1}^K \left((d(k) - o(k)) \frac{\partial E}{\partial w_i} \left(d(k) - w_0 - \sum_{i=1}^n w_i x_i(k) \right) \right)$$

$$= \sum_{k=1}^K ((d(k) - o(k))(-x_i(k)))$$

$$= - \sum_{k=1}^K (d(k) - o(k))x_i(k)$$

```
71 def adaline_train(x, y, weights_init_val=None, learning_rate=0.01, epochs=100, stopping_threshold=0.018):
72     input_variables_size = len(x[0])
73     weight_i, weight_0 = initialize_weights(input_variables_size, weights_init_val)
74
75     errors = []
76
77     for epoch in range(epochs):
78         error = 0
79         for xi, target in zip(x, y):
80             dot_product = sum(xi_j * weight_i_j for xi_j, weight_i_j in zip(xi, weight_i))
81             prediction = dot_product + weight_0
82             update = learning_rate * (target - prediction)
83             weight_i += update * xi
84             weight_0 += update
85             error += (target - prediction) ** 2 # 使用平方损失函数
86
87         mean_squared_error = error / len(y)
88         errors.append(mean_squared_error)
89
90         if mean_squared_error < stopping_threshold:
91             break
92
93     return errors, weight_i, weight_0
```

(6) Training Results and Their Visualization

Plotting the training images is relatively simple and can be achieved using the plotting functions of matplotlib.pyplot. It's important to note that, when pre-allocating memory, if the length of the error array is fixed, the stopping threshold will not take effect. Specifically, when plotting test set images, the length of the x-axis does not change with the variation of the stopping threshold. At the end of training, the error values for untrained iterations are all 0, and these values will still be plotted in the image, making it difficult to observe the effect of the stopping threshold.

Incorrect approach: `train_errors = np.zeros(epochs) # 预分配内存`

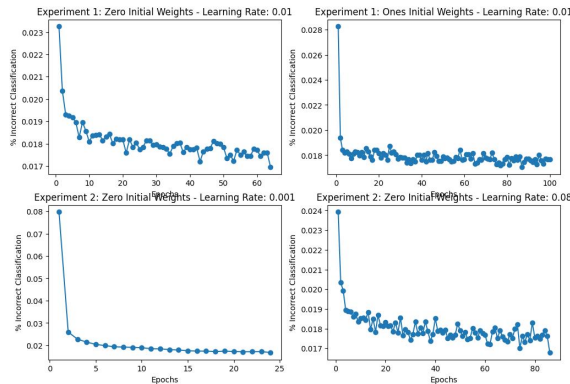
Correct approach: `errors = []`

Regarding the determination of the stopping threshold, I initially set it to a very small value, then plotted the image to observe the approximate minimum value of the error. Finally, I set the stopping threshold to this value.

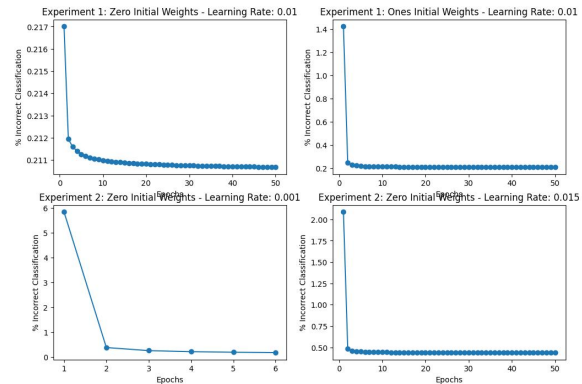
```

# 绘制训练集的错误率随时间变化的图表
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
plt.plot(range(1, len(errors1) + 1), errors1, marker='o')
plt.xlabel('Epochs')
plt.ylabel('% Incorrect Classification')
plt.title('Experiment 1: Zero Initial Weights - Learning Rate: 0.01')
  
```

Sign function as the loss function (stopping threshold at 0.017)



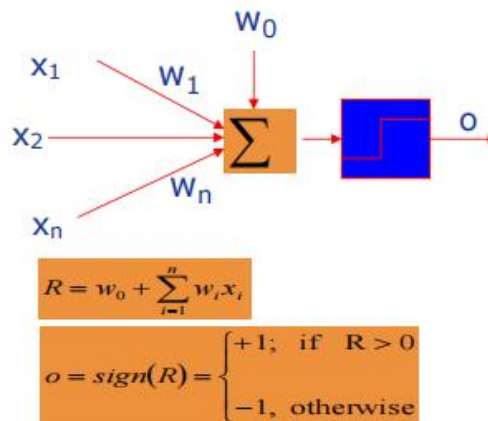
Mean Squared Error as the loss function (stopping threshold at 0.18).



(7) Validation Set Testing

The testing process for the validation set involves applying the perceptron to the data of the validation set (i.e., 784 features), computing the results, and taking the sign of the obtained values.

From a visual perspective, the perceptron can be seen as a decision plane. By inputting the coordinates of data feature points into the corresponding function, the positive or negative sign of the function value can distinguish on which side of the decision plane the feature point is located, thereby achieving binary classification.



Code:

```
# 使用训练好的参数进行预测
def predict_with_parameters(x, weight_i, weight_0):
    predictions = np.sign(np.dot(x, weight_i) + weight_0)
    return predictions

# 计算错误率
def calculate_error_rate(predictions, actual_labels):
    incorrect_samples = np.sum(predictions != actual_labels)
    error_rate = incorrect_samples / len(actual_labels)
    return error_rate
```

Results of the Single-Layer Perceptron:

```
# 在测试集上验证不同实验的错误率
predict_label_1 = predict_with_parameters(test_images_array, final_weight_i_1, final_weight_0_1)
predict_label_2 = predict_with_parameters(test_images_array, final_weight_i_2, final_weight_0_2)
predict_label_3 = predict_with_parameters(test_images_array, final_weight_i_3, final_weight_0_3)
predict_label_4 = predict_with_parameters(test_images_array, final_weight_i_4, final_weight_0_4)
error_rate_1 = calculate_error_rate(predict_label_1, test_label)
error_rate_2 = calculate_error_rate(predict_label_2, test_label)
error_rate_3 = calculate_error_rate(predict_label_3, test_label)
error_rate_4 = calculate_error_rate(predict_label_4, test_label)

print("Experiment 1 Error Rate (Zero Initial Weights):", error_rate_1)
print("Experiment 1 Error Rate (Ones Initial Weights):", error_rate_2)
print("Experiment 2 Error Rate (Learning Rate: 0.001):", error_rate_3)
print("Experiment 2 Error Rate (Learning Rate: 0.08):", error_rate_4)
```

✓ 0.0s

```
Experiment 1 Error Rate (Zero Initial Weights): 0.0185
Experiment 1 Error Rate (Ones Initial Weights): 0.0164
Experiment 2 Error Rate (Learning Rate: 0.001): 0.0192
Experiment 2 Error Rate (Learning Rate: 0.08): 0.0168
```

(8) Validation Set Plotting Charts

Based on the understanding from the theoretical part of the experiment, the assessment of the performance of a binary classification perceptron is primarily done through metrics such as the confusion matrix and ROC curve. In this experiment, we choose to use the confusion matrix for performance evaluation.

- **真正例 (True Positive, TP)** : 模型正确地预测为正例的数量。
- **真负例 (True Negative, TN)** : 模型正确地预测为负例的数量。
- **假正例 (False Positive, FP)** : 模型错误地预测为正例的数量。
- **假负例 (False Negative, FN)** : 模型错误地预测为负例的数量。

混淆矩阵的一般形式如下:

$$\begin{array}{cc} & \begin{matrix} TN & FP \end{matrix} \\ \begin{matrix} FN & TP \end{matrix} & \end{array}$$

根据混淆矩阵, 我们可以计算一些评估指标, 如准确率 (Accuracy)、精确率 (Precision)、召回率 (Recall) 和 F1 分数等。

- **准确率 (Accuracy)** : $\frac{TP+TN}{TP+TN+FP+FN}$
- **精确率 (Precision)** : $\frac{TP}{TP+FP}$
- **召回率 (Recall)** : $\frac{TP}{TP+FN}$
- **F1 分数 (F1Score)** : $2 \times \frac{Precision \times Recall}{Precision + Recall}$

Function define:

```

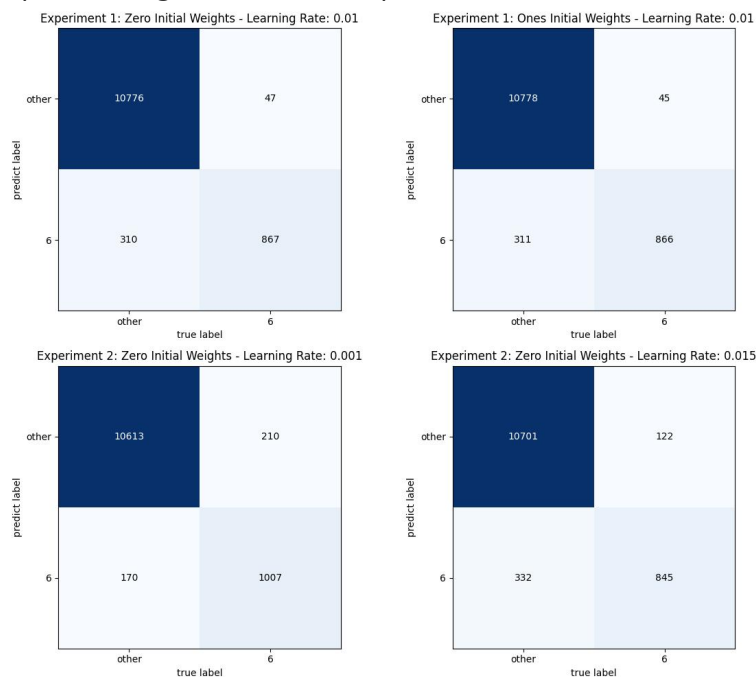
# 计算混淆矩阵
def calculate_confusion_matrix(verify, predict):
    true_positive = np.sum((verify == 1) & (predict == 1))
    false_positive = np.sum((verify == -1) & (predict == 1))
    true_negative = np.sum((verify == -1) & (predict == -1))
    false_negative = np.sum((verify == 1) & (predict == -1))

    return np.array([[true_negative, false_positive], [false_negative, true_positive]])

# 绘制混淆矩阵
def plot_confusion_matrix(confusion_matrix, title):
    plt.imshow(confusion_matrix, interpolation='nearest', cmap=plt.cm.Blues)
    plt.title(title)
    # plt.colorbar()
    # 这个 colorbar 可以在图像旁边显示颜色对应的数值
    classes = ['other', '6']
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)
    plt.xlabel('true label')
    plt.ylabel('predict label')

    # 这段代码是在绘制的矩阵中添加数字标签
    for i in range(len(classes)):
        for j in range(len(classes)):
            # 根据颜色深浅选择标签颜色
            # text_color = 'white' if confusion_matrix[i, j] > confusion_matrix.max() / 2 else 'black'
            text_color = 'white' if confusion_matrix[i, j] > confusion_matrix.mean() else 'black'
            plt.text(j, i, str(confusion_matrix[i, j]), ha='center', va='center', color=text_color)
  
```

As an example, using the Mean Squared Error as the loss function:



2. Implementing the differentiation between the digit 6 and other digits in Python.

There are two implementation approaches for this task. One involves using the sign function as the loss function, while the other involves utilizing mean squared error as the loss function. The following is an introduction to the relevant functions.



```
# Read dataset images
def load_mnist_images(filename):
# Read dataset labels
def load_mnist_labels(filename):
# Split the dataset to obtain the training set and test set
def split_dataset(x, y, test_size=0.2, random_state=None):
# Initialize weights (as required by the experiment)
def initialize_weights(input_variables_size, weights_init_val=None):
# Train using the sign function as the loss function
def train(x, y, weights_init_val=None, learning_rate=0.01, epochs=100, stopping_threshold=0.018):
# Train using mean squared error as the loss function
def adaline_train(x, y, weights_init_val, learning_rate = 0.01, epochs = 50, stopping_threshold = 0.01):
# Predict the test set with the trained function
def predict_with_parameters(x, weight_i, weight_0):
# Calculate the accuracy of predictions
def calculate_error_rate(predictions, actual_labels):
# Calculate ROC parameters: True Positive Rate and False Positive Rate
def calculate_roc_rate(verify, predict):
# Calculate the confusion matrix
def calculate_confusion_matrix(verify, predict):
# Plot the confusion matrix
def plot_confusion_matrix(confusion_matrix, title):
```

3. Implementing the differentiation between the digit 6 and other digits in MATLAB.

Function files:

```
% Read dataset images
load_mnist_images.m
% Read dataset labels
load_mnist_labels.m
% Initialize weights for the perceptron
initialize_weights.m
% Sign function as the loss function
single_perceptron.m
% Mean squared error as the loss function
adaline_train.m
```

Script files:

```
% Display the first ten images to validate data loading
show_imshow.m
% Load dataset images and labels
data_load.m
% Training with the sign function as the loss function
Single_perceptron_train.m
% Training with mean squared error as the loss function
adaline.m
```

4. Implementing the differentiation between the digit 2 and digit 5 in MATLAB.

Function files:

% Calculate the count of each label in the dataset (not referenced in script files, for debugging purposes)

`calculateSampleCount.m`

% Display images of a specific digit (not referenced in script files, for debugging purposes)

`displayDigitSamples.m`

% Display misclassified images

`displayMisclassifiedImages.m`

Script files:

`task1_SLP.m`

5. Implementing the differentiation between the digits 1, 2, 5, and 7 in MATLAB.

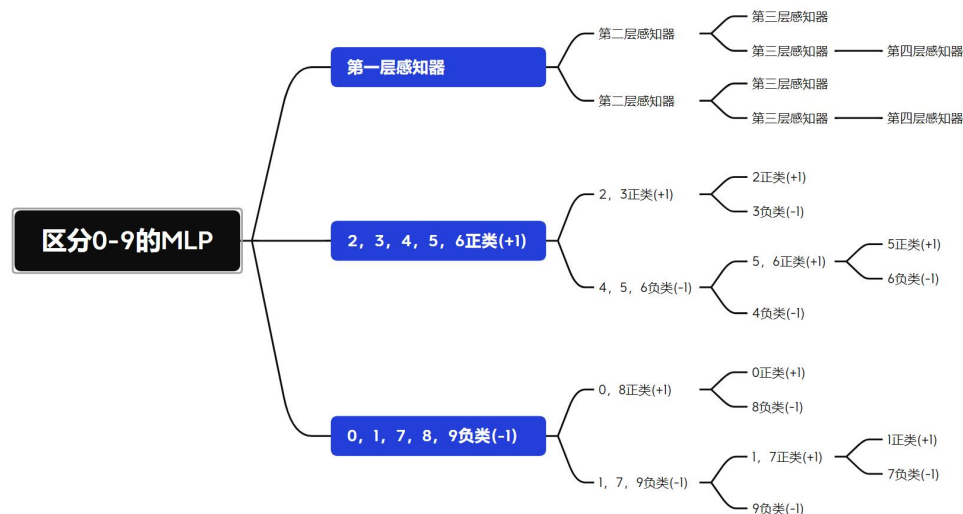
Script files:

`task2_MLP.m`

6. Implementing the differentiation between digits 0 - 9 in MATLAB.

Script files:

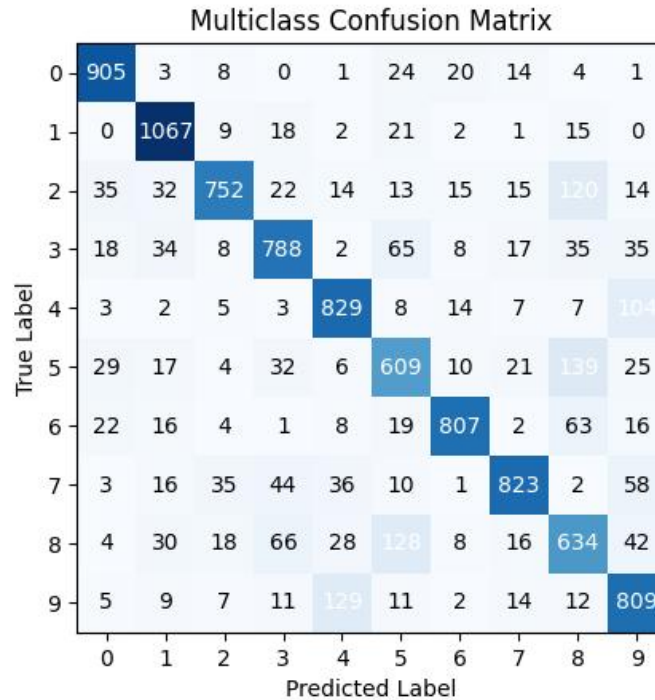
`task3.m`



7. Implementing the differentiation between digits 0-9 in Python.

This is essentially translating the MATLAB code for distinguishing digits 0-9 into Python. The implementation approach remains the same.

Additionally, there is an enhancement for visualizing test set images, and the confusion matrix is still employed for evaluation.

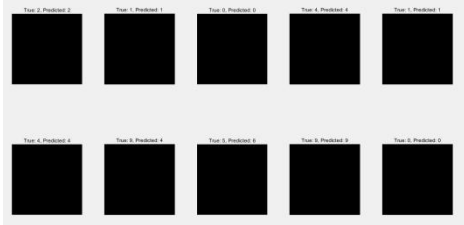


8. Issues Encountered, Resolutions, and Some Details

(1) Data Set Loading and Display Issues

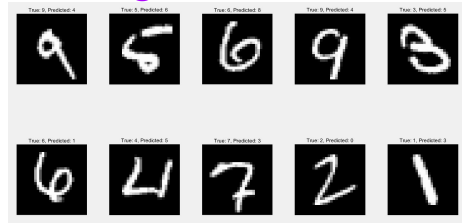
(For Python, refer to the "Implementation Process" section in the experiment report.) Regarding MATLAB display, the following is incorrect:

```
imshow(img', 'DisplayRange', [0, 255]);
```



The correct approach is:

```
imshow(img', 'InitialMagnification', 'fit');
```



(2) Data Set Splitting

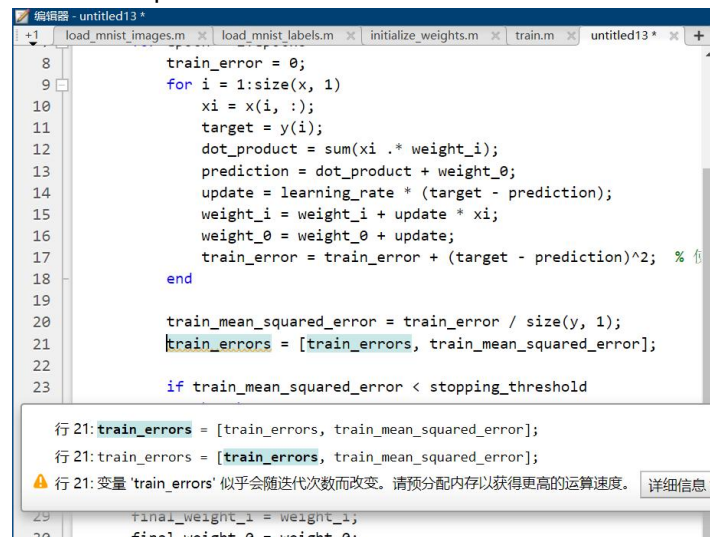
I added a random seed to ensure the reproducibility of the data set split. Setting a random seed affects the subsequent random shuffling of the data set. The code provides a random seed, and with the line `np.random.seed(random_state)`, it ensures that each time the `np.random.permutation(data_size)` step is executed, the random permutation result will be deterministic. Without setting a random seed, different random seeds will be used each time the code is run, resulting in different data set arrangements.

```
# 设置随机种子
if random_state is not None:
    np.random.seed(random_state)

# 随机打乱数据集
indices = np.random.permutation(data_size)
```

(3) MATLAB Preallocation Memory Issue

The length of `train_errors` was not defined within the function, leading to the following warning. Although the program can still run, it may cause a slowdown in computation speed.



Modify it as follows, explicitly specifying the array length before calling `train_errors`. This can improve program runtime. However, this method has a drawback: the length of `train_errors` won't change with the modification of `stopping_threshold`, causing the x-axis length of the training error change curve to remain constant, making it difficult to visually perceive the effect of `stopping_threshold`.

```
train_errors = zeros(1, epochs); % 预分配内存
```


In Python, you can include the following code when truncating to solve this problem.

```
if train_mean_squared_error < stopping_threshold:
    train_errors = train_errors[:epoch + 1]
    # 截取数组长度, 以适应变化的 stopping_threshold
    break
```

(4) Utilize Python Library Functions to Reduce Time Complexity When Calculating Dot Products

Original code effectiveness:

```
# 实验2: 不同训练速率
errors3, final_weight_i_3,
adaline_train(train_images_array, train_label, weights_init_val=1, learning_rate=0.01, stopping_threshold=0.018)
errors4, final_weight_i_4,
adaline_train(train_images_array, train_label, weights_init_val=1, learning_rate=0.08, stopping_threshold=0.018)
```

564m 31.9s

Modification:

```
# dot_product = sum(xi_j * weight_i_j for xi_j, weight_i_j in zip(xi, weight_i))
# 运行时间太长了, 我猜测是此处循环的问题, 改用以下
dot_product = np.sum(xi * weight_i)
```

(5) Learning Rate Selection Issue

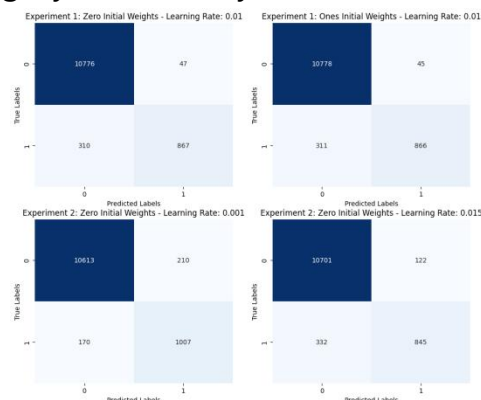
The selection of the learning rate should be realistic. For example, a learning rate of 0.08 can lead to numerical errors due to excessive values:

```
adaline_train(train_images_array, train_label, weights_init_val=1, learning_rate=0.08, stopping_threshold=0.018)
errors4, final_weight_i_4, final_weight_0_4 = \
    adaline_train(train_images_array, train_label, weights_init_val=1, learning_rate=0.08, stopping_threshold=0.018)
37m 34s

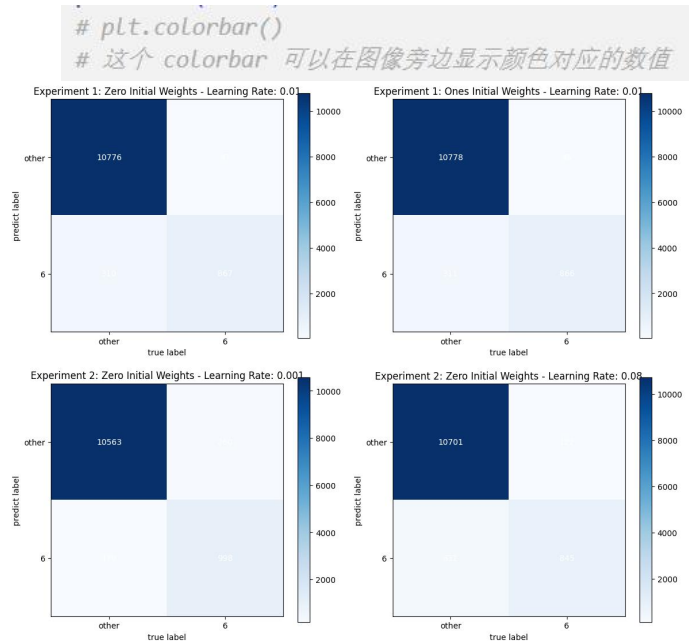
C:\Users\HP\AppData\Local\Temp\ipykernel_17928\3940178182.py:65: RuntimeWarning: overflow encountered in square
error += (target - prediction) ** 2 # 使用平方损失函数
C:\Users\HP\AppData\Local\Temp\ipykernel_17928\3940178182.py:68: RuntimeWarning: overflow encountered in scalar add
dot_product = sum(xi_j * weight_i_j for xi_j, weight_i_j in zip(xi, weight_i))
C:\Users\HP\AppData\Local\Temp\ipykernel_17928\3940178182.py:63: RuntimeWarning: invalid value encountered in multiply
weight_i += update * xi
C:\Users\HP\AppData\Local\Temp\ipykernel_17928\3940178182.py:68: RuntimeWarning: invalid value encountered in scalar multiply
dot_product = sum(xi_j * weight_i_j for xi_j, weight_i_j in zip(xi, weight_i))
```

(6) Confusion Matrix Plotting Issue

In reality, Python already provides relevant library functions for plotting confusion matrices. To gain a deeper understanding of this method, I rewrote a confusion matrix plotting function. First, let's look at the method of plotting confusion matrices using Python library functions:

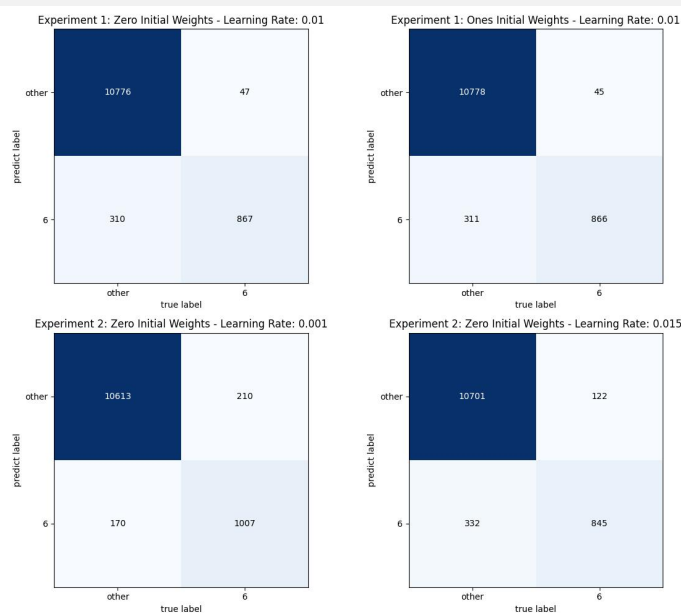


Next is my first confusion matrix plotting function, where I added the drawing of a color bar:



In addition, there is a small detail regarding font color. I attempted to use half of the maximum value and the average value of the data to differentiate colors, and found that using half of the maximum value is more reasonable.

```
# 这段代码是在绘制的矩阵中添加数字标签
for i in range(len(classes)):
    for j in range(len(classes)):
        # 根据颜色深浅选择标签颜色
        # text_color = 'white' if confusion_matrix[i, j] > confusion_matrix.max() / 2 else 'black'
        text_color = 'white' if confusion_matrix[i, j] > confusion_matrix.mean() else 'black'
        plt.text(j, i, str(confusion_matrix[i, j]), ha='center', va='center', color=text_color)
```



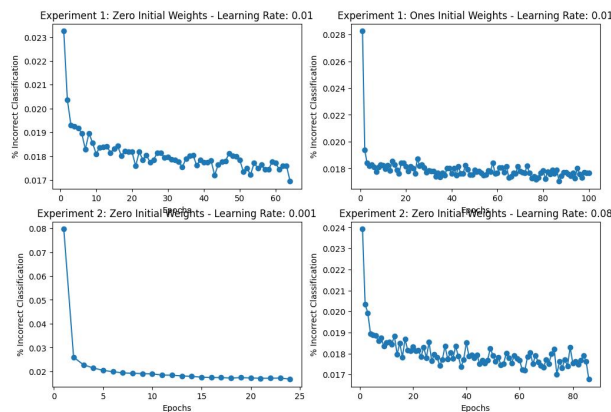
9. Answers to Experimental Requirements

(1) Initial Weight Selection Issue

In my opinion, the impact of initial weights on the results is not particularly significant when a sufficient number of training iterations are performed. This is because in our experiment, there exists a unique local minimum point between weights and the loss function. According to the rules of gradient descent, with a sufficient number of training iterations, the model will converge to this unique local minimum point, which is the most suitable combination of weights.

(2) Result Evaluation

(Mainly for the Results with Mean Squared Error as the Loss Function)



The chart below illustrates the variation of error rates over time for the four experiments. Experiment 1 primarily investigates the impact of different initial weights on performance, while Experiment 2 focuses on the effects of different learning rates. It is evident from the chart that, over time, the error rates for all experiments show a decreasing trend.

In Experiment 1, it was observed that regardless of whether the initial weights were all zeros or all ones, the error rates quickly decreased to a low level and remained stable. This result validates part of the hypothesis about initial weight selection: in our experiment, there exists a unique local minimum point between weights and the loss function. According to the rules of gradient descent, with a sufficient number of training iterations, the model will converge to this unique local minimum point, which is the most suitable combination of weights. As for the case where truncation did not occur, I speculate that it might be due to the relatively high learning rates of the two

perceptrons in Experiment 1, leading to continuous oscillation near the stopping threshold.

The results of Experiment 2 indicate that perceptrons with smaller learning rates can reach the stopping threshold in fewer training iterations and do not exhibit oscillations near this threshold. Additionally, the amplitude of oscillations is larger, possibly resulting in a greater difference from the threshold value.

III. Experiment Summary

1. Evaluation of Single-Layer Perceptron Performance

The overall conclusion of this experiment indicates that the choice of initial weights and learning rates has a significant impact on the performance of the perceptron model during training. Experiment 1 revealed the influence of different initial weights on model performance, pointing out that whether the initial weights are all zeros or all ones, the model's error rate quickly decreased and stabilized after training. This result confirms the existence of a unique local minimum point between weights and the loss function in the experiment, and the model effectively converges to the most suitable combination of weights through the gradient descent rule. Experiment 2 highlighted the crucial role of learning rates during the training process. A small learning rate helps reach the stopping threshold in fewer training iterations, and it does not induce oscillations near the threshold. Conversely, a larger learning rate may lead to frequent oscillations near the stopping threshold, or even prevent convergence. This emphasizes the importance of selecting an appropriate learning rate in real-world applications to ensure the stability and effectiveness of training. However, Experiment 1 also raised a potential issue, indicating that in certain cases, a high learning rate may cause oscillation problems, hindering model convergence. Therefore, in practical applications, careful consideration and adjustment of initial weights and learning rates are necessary to achieve optimal performance. Performance evaluation can also benefit from visualization methods such as confusion matrices.

2. Building a Multi-Layer Perceptron for Multiclass Classification Using a Single-Layer Perceptron

By combining multiple hierarchical labels and single-layer perceptrons, we can gradually divide the ten digits from 0 to 9 into different sets and then take their unique intersections, thus achieving multiclass classification. This is the approach to constructing a Multi-Layer Perceptron (MLP).

3. Programming with MATLAB

Gradually becoming familiar with MATLAB syntax and functionality, learning to manipulate data and program with MATLAB.



【References】

[1]模型评价之混淆矩阵、ROC 曲线与 AUC - 知乎 (zhihu.com)
<https://zhuanlan.zhihu.com/p/390518914>