小组成员：24140583 李锋 ；24140637 郑京璞

使用Python语言构建MLP模型实验

> 本实验通过原生python代码实现MLP的分类功能，如神经网络核心组件：激活函数模块、权重初始化策略、优化器实现、正则化方法，创建MLP模型架构：前向传播、反向传播、批次训练、预测与评估等过程，最终将训练后的模型保存后进行后续对测试集的标签值预测过程

一、 开发环境准备

1.1 python环境

为解决Python项目的环境问题，使用了现在推荐的方式，安装Anaconda，创建了针对该项目的Python3.12版本环境mlp_env，为了方便开发代码，安装了PyCharm作为开发工具

```
# 确认conda是否安装成功
conda --version
# 创建当前项目使用环境,指定Python版本为3.12
conda create -n mlp_env python=3.12
# 激活环境
conda activate mlp_env
# 配置国内镜像源 清华源
conda config --add channels
https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/main/
conda config --add channels
https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
# 确保配置镜像源生效
conda config --show channels
```

1.2 项目中使用到的三方模块

```
conda install numpy
conda install scikit-learn
#后续在优化环节为实现可视化效果增加了
matplotlib 和 tensorboard
```

二、MINST DATASET数据获取

2.1 MINST DATASET数据的下载

下载地址如下：https://tianchi.aliyun.com/dataset/92224

2.2 将下载的数据转换为CSV格式

（1）python实现

```python
import numpy as np
import struct

def convert_to_csv(image_file, label_file, output_file):
    with open(image_file, 'rb') as f_img, open(label_file, 'rb') as f_lbl, open(output_file, 'w') as f_out:
        # 读取图像文件的头部信息
        magic, num, rows, cols = struct.unpack('>IIII', f_img.read(16))
        # 读取标签文件的头部信息
        magic, num_labels = struct.unpack('>II', f_lbl.read(8))

        for i in range(num):
            label = ord(f_lbl.read(1))
            image = [ord(f_img.read(1)) for _ in range(rows * cols)]
            row = [str(label)] + [str(pixel) for pixel in image]
            f_out.write(','.join(row) + '\n')

# 示例使用
train_images = 'MNIST_data/train-images-idx3-ubyte'
train_labels = 'MNIST_data/train-labels-idx1-ubyte'
test_images = 'MNIST_data/t10k-images-idx3-ubyte'
test_labels = 'MNIST_data/t10k-labels-idx1-ubyte'

convert_to_csv(train_images, train_labels, 'MNIST_data/mnist_train.csv')
convert_to_csv(test_images, test_labels, 'MNIST_data/mnist_test.csv')
```

（2）运行程序文件

在代码文件夹下运行如下cmd命令行，即可得到转换后生成的文件wiki.zh.txt。

```
python csvconvert.py
```

（3）程序输出

三、MLP神经网络结构实现

3.1 定义激活函数及其导数

（1）python实现

激活函数类别有：sigmoid、relu、softmax，python代码实现如下

```
# 定义激活函数及其导数
class Activation:
    @staticmethod
    def sigmoid(x):
        """
        Sigmoid激活函数（作用是将输入映射到0-1之间，常用于二分类输出）
        :param x: 输入数据
        :return: 经过sigmoid激活后的数据
        """
        return 1 / (1 + np.exp(-x))

    @staticmethod
    def sigmoid_derivative(x):
        """
        Sigmoid激活函数的导数（Sigmoid函数的导数可表示为sigmoid(x)*(1-
sigmoid(x))）
        :param x: 输入数据
        :return: sigmoid导数计算结果
        """
        return Activation.sigmoid(x) * (1 - Activation.sigmoid(x))

    @staticmethod
    def relu(x):
        """
        ReLU激活函数
        :param x: 输入数据
        :return: 经过ReLU激活后的数据
        """
```

```
        return np.maximum(0, x)

    @staticmethod
    def relu_derivative(x):
        """
        ReLU激活函数的导数
        :param x: 输入数据
        :return: ReLU导数计算结果
        """
        return (x > 0).astype(float)

    @staticmethod
    def softmax(x):
        """
        Softmax激活函数，用于多分类问题
        :param x: 输入数据
        :return: 经过Softmax激活后的数据
        """
        exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)
```

3.2 定义权重初始化方法

（1）python代码

```
# 定义权重初始化方法
class WeightInitializer:
    @staticmethod
    def random_init(input_size, output_size):
        """
        随机初始化权重（使用np.random.randn生成正态分布的随机数，然后乘以0.01进行缩放，这是常见的权重初始化方法，防止梯度爆炸或消失。）
        :param input_size: 输入层大小
        :param output_size: 输出层大小
        :return: 初始化后的权重矩阵
        """
        return np.random.randn(input_size, output_size) * 0.01

    @staticmethod
    def xavier_init(input_size, output_size):
```

```
        """
        Xavier初始化权重（首先计算limit值，然后生成均匀分布的矩阵）
        :param input_size: 输入层大小
        :param output_size: 输出层大小
        :return: 初始化后的权重矩阵
        """
        limit = np.sqrt(6 / (input_size + output_size))
        return np.random.uniform(-limit, limit, (input_size,
output_size))
```

3.3 定义优化器

（1）python代码

优化器类别有SGD、Momentum、RMSProp、Adam，python代码如下

```
# 定义优化器
class Optimizer:
    def __init__(self, learning_rate):
        """
        优化器基类构造函数
        :param learning_rate: 学习率
        """
        self.learning_rate = learning_rate

    def update(self, weights, gradients):
        """
        更新权重的抽象方法，需要在子类中实现
        :param weights: 当前权重矩阵
        :param gradients: 梯度矩阵
        :return: 更新后的权重矩阵
        """
        raise NotImplementedError


class SGD(Optimizer):
    def update(self, weights, gradients):
        """
        随机梯度下降更新权重
        :param weights: 当前权重矩阵
```

```
        :param gradients: 梯度矩阵
        :return: 更新后的权重矩阵
        """
        return weights - self.learning_rate * gradients


class Momentum(Optimizer):
    def __init__(self, learning_rate, momentum=0.9):
        """
        动量优化器构造函数
        :param learning_rate: 学习率
        :param momentum: 动量系数
        """
        super().__init__(learning_rate)
        self.momentum = momentum
        self.v = None

    def update(self, weights, gradients):
        """
        使用动量更新权重
        :param weights: 当前权重矩阵
        :param gradients: 梯度矩阵
        :return: 更新后的权重矩阵
        """
        if self.v is None:
            self.v = np.zeros_like(weights)
        self.v = self.momentum * self.v + self.learning_rate * gradients
        return weights - self.v


class RMSProp(Optimizer):
    def __init__(self, learning_rate, rho=0.9, epsilon=1e-8):
        """
        RMSProp优化器构造函数
        :param learning_rate: 学习率
        :param rho: 衰减率
        :param epsilon: 防止除零的小常数
        """
        super().__init__(learning_rate)
        self.rho = rho
        self.epsilon = epsilon
```

```python
        self.s = None

    def update(self, weights, gradients):
        """
        使用RMSProp更新权重
        :param weights: 当前权重矩阵
        :param gradients: 梯度矩阵
        :return: 更新后的权重矩阵
        """
        if self.s is None:
            self.s = np.zeros_like(weights)
        self.s = self.rho * self.s + (1 - self.rho) * gradients ** 2
        return weights - self.learning_rate * gradients /
(np.sqrt(self.s) + self.epsilon)


class Adam(Optimizer):
    def __init__(self, learning_rate, beta1=0.9, beta2=0.999, epsilon=1e-
8):
        super().__init__(learning_rate)
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = {}  # 格式: { (层索引, 参数类型): 矩阵 }
        self.v = {}
        self.t = 0

    def update(self, layer_idx, param_type, weights, gradients):
        """
        :param layer_idx: 层索引 (从0开始)
        :param param_type: 'weight' 或 'bias'
        """
        self.t += 1
        key = (layer_idx, param_type)  # 唯一键

        if key not in self.m:
            self.m[key] = np.zeros_like(weights)
            self.v[key] = np.zeros_like(weights)

        # 更新矩估计
```

```
        self.m[key] = self.beta1 * self.m[key] + (1 - self.beta1) *
gradients
        self.v[key] = self.beta2 * self.v[key] + (1 - self.beta2) *
(gradients ** 2)

        # 偏置修正
        m_hat = self.m[key] / (1 - self.beta1 ** self.t)
        v_hat = self.v[key] / (1 - self.beta2 ** self.t)

        return weights - self.learning_rate * m_hat / (np.sqrt(v_hat) +
self.epsilon)
```

3.3 定义正则化方法

（1）python代码

```
# 定义正则化方法
class Regularizer:
    @staticmethod
    def l1(weights, lambda_):
        """
        L1正则化（计算L1正则化项，通过对权重矩阵所有元素取绝对值求和后乘以正则化系数
lambda_。用于在损失函数中增加权重稀疏性惩罚，降低模型复杂度防止过拟合）
        :param weights: 权重矩阵
        :param lambda_: 正则化系数
        :return: L1正则化项
        """
        return lambda_ * np.sum(np.abs(weights))

    @staticmethod
    def l1_derivative(weights, lambda_):
        """
        L1正则化的导数
        :param weights: 权重矩阵
        :param lambda_: 正则化系数
        :return: L1正则化导数
        """
        return lambda_ * np.sign(weights)

    @staticmethod
```

```python
    def l2(weights, lambda_):
        """
        L2正则化（2正则化通常用于防止过拟合，通过惩罚大的权重值，这里的计算是0.5乘以lambda参数，再乘以权重的平方和）
        :param weights: 权重矩阵
        :param lambda_: 正则化系数
        :return: L2正则化项
        """
        return 0.5 * lambda_ * np.sum(weights ** 2)

    @staticmethod
    def l2_derivative(weights, lambda_):
        """
        L2正则化的导数
        :param weights: 权重矩阵
        :param lambda_: 正则化系数
        :return: L2正则化导数
        """
        return lambda_ * weights

    @staticmethod
    def elastic(weights, lambda_, alpha):
        """
        弹性网络正则化（结合了L1和L2正则化，使用alpha作为权重参数，lambda_作为正则化系数。具体来说，计算alpha乘以L1正则化项加上(1-alpha)乘以L2正则化项，返回两者的加权和。）
        :param weights: 权重矩阵
        :param lambda_: 正则化系数
        :param alpha: L1正则化比例
        :return: 弹性网络正则化项
        """
        return alpha * Regularizer.l1(weights, lambda_) + (1 - alpha) * Regularizer.l2(weights, lambda_)

    @staticmethod
    def elastic_derivative(weights, lambda_, alpha):
        """
        弹性网络正则化的导数
        :param weights: 权重矩阵
        :param lambda_: 正则化系数
        :param alpha: L1正则化比例
```

```
        :return: 弹性网络正则化导数
        """
        return alpha * Regularizer.l1_derivative(weights, lambda_) + (1 -
alpha) * Regularizer.l2_derivative(weights,

                                    lambda_)
```

四、MLP模型架构

4.1 定义MLP模型初始化和模型基本结构输出

(1) python代码

```
# 定义MLP模型
class MLP:
    def __init__(self, layers, activations, weight_init='random',
optimizer='sgd', learning_rate=0.01,
                 regularization=None, lambda_=0.01, alpha=0.5,
stop_criteria=1e-6):
        logging.info("开始模型初始化过程   Initializing MLP model...")
        """
        MLP模型构造函数
        :param layers: 各层神经元数量列表
        :param activations: 各层激活函数列表
        :param weight_init: 权重初始化方法
        :param optimizer: 优化器类型
        :param learning_rate: 学习率
        :param regularization: 正则化方法
        :param lambda_: 正则化系数
        :param alpha: 弹性网络正则化中L1的比例
        :param stop_criteria: 停止训练的标准
        """
        self.layers = layers
        self.activations = activations
        self.weights = []
        self.biases = []
        self.activation_functions = {
            'sigmoid': (Activation.sigmoid,
Activation.sigmoid_derivative),
```

```
            'relu': (Activation.relu, Activation.relu_derivative),
            'softmax': (Activation.softmax, None)
        }
        self.weight_init = weight_init
        self.regularization = regularization
        self.lambda_ = lambda_
        self.alpha = alpha
        self.stop_criteria = stop_criteria

        # 初始化权重和偏置
        for i in range(len(layers) - 1):
            if weight_init == 'random':

 self.weights.append(WeightInitializer.random_init(layers[i], layers[i +
1]))
            elif weight_init == 'xavier':

 self.weights.append(WeightInitializer.xavier_init(layers[i], layers[i +
1]))
            self.biases.append(np.zeros((1, layers[i + 1])))

        # 选择优化器
        if optimizer == 'sgd':
            self.optimizer = SGD(learning_rate)
        elif optimizer == 'momentum':
            self.optimizer = Momentum(learning_rate)
        elif optimizer == 'rmsprop':
            self.optimizer = RMSProp(learning_rate)
        elif optimizer == 'adam':
            self.optimizer = Adam(learning_rate)

        # 在初始化最后添加结构输出
        self.print_model_summary()

    def print_model_summary(self):
        """打印模型结构摘要"""
        logging.info("\n{:=^60}".format(" 输出模型结构摘要信息    Model
Summary "))
        total_params = 0
        layers_info = []
```

```
        # 输入层
        layers_info.append({
            'Layer': 0,
            'Type': 'Input',
            'Neurons': self.layers[0],
            'Activation': '-',
            'Weights Shape': '-',
            'Bias Shape': '-',
            'Params': 0
        })

        # 隐藏层和输出层
        for i in range(len(self.layers) - 1):
            layer_type = 'Hidden' if i < len(self.layers) - 2 else
'Output'
            weights_shape = self.weights[i].shape if i <
len(self.weights) else '-'
            bias_shape = self.biases[i].shape if i < len(self.biases)
else '-'
            params = np.prod(weights_shape) + np.prod(bias_shape) if i <
len(self.weights) else 0

            layers_info.append({
                'Layer': i + 1,
                'Type': layer_type,
                'Neurons': self.layers[i + 1],
                'Activation': self.activations[i],
                'Weights Shape': weights_shape,
                'Bias Shape': bias_shape,
                'Params': params
            })
            total_params += params

        # 打印表格
        logging.info("{:<6} {:<8} {:<8} {:<10} {:<15} {:<12} {:
<10}".format(
            'Layer', 'Type', 'Neurons', 'Activation', 'Weights Shape',
'Bias Shape', 'Parameters'
        ))
        logging.info("-" * 70)
        for info in layers_info:
```

```
            logging.info("{:<6} {:<8} {:<8} {:<10} {:<15} {:<12} {:
<10,}".format(
                info['Layer'],
                info['Type'],
                info['Neurons'],
                info['Activation'],
                str(info['Weights Shape']),
                str(info['Bias Shape']),
                info['Params']
            ))

        # 打印汇总信息
        logging.info("\n{:=^60}".format(" Summary "))
        logging.info(f"Total layers: {len(self.layers)} (input +
{len(self.layers) - 2} hidden + output)")
        logging.info(f"Total parameters: {total_params:,}")
        logging.info(f"Weight Initialization: {self.weight_init}")
        logging.info(f"Regularization: {self.regularization if
self.regularization else 'None'}")
        if self.regularization:
            logging.info(
                f"Lambda: {self.lambda_}{' Alpha: ' + str(self.alpha) if
self.regularization == 'elastic' else ''}")
        logging.info(f"Optimizer: {type(self.optimizer).__name__} (lr=
{self.optimizer.learning_rate})")
        logging.info("=" * 60 + "\n")
```

4.2 MLP模型的前向、后向传播

（1）python代码

```
 def forward(self, X):
        """
        前向传播
        :param X: 输入数据
        :return: 输出结果和各层激活值
        """
        logging.info("开始前向传播过程   Forward propagation...")
        # activations列表的作用是存储每一层的输出，包括输入层的数据。输入层的数据X
作为第一个元素，之后每一层的输出都是基于前一层的激活值计算得到的
        activations = [X]
```

```
        logging.info(f"输入数据形状: {X.shape}")
        for i in range(len(self.layers) - 1):
            z = np.dot(activations[-1], self.weights[i]) + self.biases[i]
            activation_func =
self.activation_functions[self.activations[i]][0]
            a = activation_func(z)
            activations.append(a)
            logging.info(f"第 {i + 1} 层输入形状: {activations[-2].shape},
输出形状: {a.shape}")
        logging.info("前向传播完成   Forward propagation completed.")
        return activations[-1], activations

    def backward(self, X, y, activations):
        """
        反向传播
        :param X: 输入数据
        :param y: 真实标签
        :param activations: 各层激活值
        :return: 权重和偏置的梯度
        """
        logging.info("开始反向传播过程   Backward propagation...")
        num_samples = X.shape[0]
        weight_gradients = []
        bias_gradients = []
        output = activations[-1]

        # 计算输出层误差
        if self.activations[-1] == 'softmax':
            delta = output - y
        else:
            activation_derivative =
self.activation_functions[self.activations[-1]][1]
            delta = (output - y) * activation_derivative(output)

        # 反向传播计算梯度
        for i in range(len(self.layers) - 2, -1, -1):
            weight_gradient = np.dot(activations[i].T, delta) /
num_samples
            bias_gradient = np.sum(delta, axis=0, keepdims=True) /
num_samples
```

```
            # 添加正则化项
            if self.regularization == 'l1':
                weight_gradient +=
Regularizer.l1_derivative(self.weights[i], self.lambda_)
            elif self.regularization == 'l2':
                weight_gradient +=
Regularizer.l2_derivative(self.weights[i], self.lambda_)
            elif self.regularization == 'elastic':
                weight_gradient +=
Regularizer.elastic_derivative(self.weights[i], self.lambda_, self.alpha)

            weight_gradients.insert(0, weight_gradient)
            bias_gradients.insert(0, bias_gradient)

            if i > 0:
                activation_derivative =
self.activation_functions[self.activations[i - 1]][1]
                delta = np.dot(delta, self.weights[i].T) *
activation_derivative(activations[i])
        logging.info("反向传播完成  Backward propagation completed.")
        return weight_gradients, bias_gradients
```

4.3 MLP模型的训练、评估、损失计算、预测过程

（1）python代码

```
    def train(self, X, y, epochs=100, batch_size=32, parallel=False):
        logging.info(f"开始模型训练过程，共 {epochs} 轮，批次大小
{batch_size}")
        """
        训练模型
        :param X: 输入数据
        :param y: 真实标签
        :param epochs: 训练轮数
        :param batch_size: 批次大小
        :param parallel: 是否并行训练
        :return: 每轮的损失值
        """
        losses = []
        num_samples = X.shape[0]
```

```
        num_batches = num_samples // batch_size

        for epoch in range(epochs):
            epoch_loss = 0
            if parallel:
                # 并行训练过程
                threads = []
                for batch in range(num_batches):
                    start = batch * batch_size
                    end = start + batch_size
                    X_batch = X[start:end]
                    y_batch = y[start:end]
                    thread = threading.Thread(target=self._train_batch,
args=(X_batch, y_batch))
                    threads.append(thread)
                    thread.start()
                for thread in threads:
                    thread.join()
            else:
                # 单线程训练过程
                for batch in range(num_batches):
                    start = batch * batch_size
                    end = start + batch_size
                    X_batch = X[start:end]
                    y_batch = y[start:end]
                    loss = self._train_batch(X_batch, y_batch)
                    epoch_loss += loss

            epoch_loss /= num_batches
            losses.append(epoch_loss)
            if parallel:
                logging.info(f'######################并行模型训练过程:Epoch
{epoch + 1}/{epochs}, Loss: {epoch_loss}')
            else:
                logging.info(f'#####################单线程模型训练过程:Epoch
{epoch + 1}/{epochs}, Loss: {epoch_loss}')

            # 停止条件
            if epoch > 0 and abs(losses[-1] - losses[-2]) <
self.stop_criteria:
```

```
                logging.info(f"达到停止条件，提前结束训练，在第 {epoch + 1}
轮")
                break

        return losses

    def _train_batch(self, X_batch, y_batch):
        output, activations = self.forward(X_batch)
        weight_gradients, bias_gradients = self.backward(X_batch,
y_batch, activations)

        # 更新权重和偏置
        for i in range(len(self.weights)):
            # 更新权重
            self.weights[i] = self.optimizer.update(
                layer_idx=i,
                param_type='weight',
                weights=self.weights[i],
                gradients=weight_gradients[i]
            )
            # 更新偏置
            self.biases[i] = self.optimizer.update(
                layer_idx=i,
                param_type='bias',
                weights=self.biases[i],
                gradients=bias_gradients[i]
            )

        # 计算损失
        loss = self._compute_loss(output, y_batch)
        return loss

    def _compute_loss(self, output, y):
        """
        计算损失
        :param output: 模型输出
        :param y: 真实标签
        :return: 损失值
        """
        logging.info("计算训练损失中...")
        num_samples = y.shape[0]
```

```
        if self.activations[-1] == 'softmax':
            #计算交叉熵损失。通过取模型输出概率的对数值，与真实标签逐元素相乘求和后取
负，最后除以样本数得到平均损失。添加1e-8防止对零取对数导致数值错误。
            loss = -np.sum(y * np.log(output + 1e-8)) / num_samples
        else:
            # 计算均方差损失
            loss = np.mean((output - y) ** 2)

        # 添加正则化项
        if self.regularization == 'l1':
            reg_loss = sum([Regularizer.l1(w, self.lambda_) for w in
self.weights])
            loss += reg_loss
        elif self.regularization == 'l2':
            reg_loss = sum([Regularizer.l2(w, self.lambda_) for w in
self.weights])
            loss += reg_loss
        elif self.regularization == 'elastic':
            reg_loss = sum([Regularizer.elastic(w, self.lambda_,
self.alpha) for w in self.weights])
            loss += reg_loss
        logging.info(f"训练损失为: {loss}")
        return loss

    def predict(self, X):
        """
        预测
        :param X: 输入数据
        :return: 预测结果
        """
        logging.info("开始模型预测过程...")
        output, _ = self.forward(X)
        if self.activations[-1] == 'softmax':
            return np.argmax(output, axis=1)
        logging.info("模型预测完成，返回预测结果...")
        return output
```

4.4 MLP模型的保存、加载过程以及混淆矩阵计算过程

（1）python代码

```
    def save_model(self, file_path):
        logging.info(f"开始保存模型到文件: {file_path}")
        model_data = {
            'layers': self.layers,
            'activations': self.activations,
            'weight_init': self.weight_init,
            'regularization': self.regularization,
            'lambda_': self.lambda_,
            'alpha': self.alpha,
            'stop_criteria': self.stop_criteria
        }

        # 按层保存权重和偏置
        for i, (w, b) in enumerate(zip(self.weights, self.biases)):
            model_data[f'weight_{i}'] = w
            model_data[f'bias_{i}'] = b
        logging.info(f"模型数据保存完成，共保存了 {len(model_data)} 个参数。")
        np.savez(file_path, **model_data)

    @staticmethod
    def load_model(file_path):
        logging.info(f"开始加载模型文件: {file_path}")
        data = np.load(file_path)
        model = MLP(
            layers=data['layers'],
            activations=data['activations'],
            weight_init=data['weight_init'],
            optimizer='sgd',  # 优化器类型需重新指定
            regularization=data['regularization'],
            lambda_=data['lambda_'],
            alpha=data['alpha'],
            stop_criteria=data['stop_criteria']
        )

        # 按层加载权重和偏置
        model.weights = [data[f'weight_{i}'] for i in
range(len(model.weights))]
        model.biases = [data[f'bias_{i}'] for i in
range(len(model.biases))]
        logging.info("模型加载完成，返回模型对象...")
        return model
```

```
    def confusion_matrix(self, X, y):
        """
        计算混淆矩阵(计算混淆矩阵以评估分类的准确性。)
        :param X: 输入数据
        :param y: 真实标签
        :return: 混淆矩阵
        """
        y_pred = self.predict(X)
        if self.activations[-1] == 'softmax':
            y_true = np.argmax(y, axis=1)
        else:
            y_true = y.flatten()
        return sk_confusion_matrix(y_true, y_pred)
```

五、主函数运行

(1) python代码

```
# 加载MNIST数据集
def load_mnist_data(file_path):
    """
    加载MNIST数据集
    :param file_path: 数据集文件路径
    :return: 特征和标签
    """
    logging.info(f"开始加载MNIST数据集，文件路径：{file_path}")
    data = []
    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # 跳过标题行
        for row in reader:
            data.append([int(x) for x in row])
    data = np.array(data)
    # 特征归一化处理：将MNIST图像的像素值（data第2列及之后）除以255实现归一化
    X = data[:, 1:] / 255.0
    # 标签独热编码：用np.eye生成10维单位矩阵，根据data首列标签值索引对应one-hot向量
    y = np.eye(10)[data[:, 0]]
    logging.info(f"开始加载MNIST数据集，文件路径：{file_path}")
```

```python
    return X, y


# 示例使用
if __name__ == "__main__":
    # 分类任务（MNIST数据集）
    logging.info("分类任务（加载MNIST数据集）loading...............")

    train_path = 'MNIST_data/mnist_train.csv'    # 训练数据集路径
    test_path = 'MNIST_data/mnist_test.csv'      # 测试数据集路径

    if os.path.exists(train_path) and os.path.exists(test_path):
        X_train_mnist, y_train_mnist = load_mnist_data(train_path)
        X_test_mnist, y_test_mnist = load_mnist_data(test_path)

        # 定义MLP模型
        mlp_classification = MLP(layers=[784, 128, 10], activations=
['relu', 'softmax'], weight_init='xavier',
                                 optimizer='adam', learning_rate=0.001,
regularization='l2', lambda_=0.001)

        # 训练模型
        losses_classification = mlp_classification.train(X_train_mnist,
y_train_mnist, epochs=10, batch_size=32)

        # 保存模型
        mlp_classification.save_model('mlp_classification.npz')

        # 预测
        y_pred_mnist = mlp_classification.predict(X_test_mnist)
        y_true_mnist = np.argmax(y_test_mnist, axis=1)
        accuracy_mnist = np.mean(y_pred_mnist == y_true_mnist)
        logging.info(f"MNIST 分类准确率: {accuracy_mnist * 100:.2f}%")

        # 混淆矩阵
        conf_matrix_mnist =
mlp_classification.confusion_matrix(X_test_mnist, y_test_mnist)
        logging.info(f'MNIST 混淆矩阵:\n {conf_matrix_mnist}')
    else:
        logging.info("MNIST数据集文件未找到，请检查路径。")
```

(2) 运行程序文件

```
python mlp.py
```

(3) 程序输出

```
##第一部分日志
lifeng@lifeng ~> conda activate mlp_env
(base)
lifeng@lifeng ~> cd PythonProject/MLP/
(mlp_env)
lifeng@lifeng ~/P/MLP> python mlp.py
(mlp_env)
2025-04-11 15:06:52 - INFO - 分类任务（加载MNIST数据集）
loading...............
2025-04-11 15:06:52 - INFO - 开始加载MNIST数据集，文件路径：
MNIST_data/mnist_train.csv
2025-04-11 15:07:03 - INFO - 开始加载MNIST数据集，文件路径：
MNIST_data/mnist_train.csv
2025-04-11 15:07:03 - INFO - 开始加载MNIST数据集，文件路径：
MNIST_data/mnist_test.csv
2025-04-11 15:07:05 - INFO - 开始加载MNIST数据集，文件路径：
MNIST_data/mnist_test.csv
2025-04-11 15:07:05 - INFO - 开始模型初始化过程   Initializing MLP model...
2025-04-11 15:07:05 - INFO -
=============== 输出模型结构摘要信息    Model Summary ================
2025-04-11 15:07:05 - INFO - Layer  Type     Neurons  Activation Weights
Shape   Bias Shape   Parameters
2025-04-11 15:07:05 - INFO - --------------------------------------------
--------------------------
2025-04-11 15:07:05 - INFO - 0      Input    784      -          -
        -            0
2025-04-11 15:07:05 - INFO - 1      Hidden   128      relu       (784,
128)      (1, 128)      100,480
2025-04-11 15:07:05 - INFO - 2      Output   10       softmax    (128,
10)       (1, 10)       1,290
2025-04-11 15:07:05 - INFO -
========================= Summary ==========================
2025-04-11 15:07:05 - INFO - Total layers: 3 (input + 1 hidden + output)
2025-04-11 15:07:05 - INFO - Total parameters: 101,770
2025-04-11 15:07:05 - INFO - Weight Initialization: xavier
```

```
2025-04-11 15:07:05 - INFO - Regularization: l2
2025-04-11 15:07:05 - INFO - Lambda: 0.001
2025-04-11 15:07:05 - INFO - Optimizer: Adam (lr=0.001)
2025-04-11 15:07:05 - INFO -
============================================================


2025-04-11 15:07:05 - INFO - 开始模型训练过程，共 10 轮，批次大小 32
2025-04-11 15:07:05 - INFO - 开始前向传播过程   Forward propagation...
2025-04-11 15:07:05 - INFO - 输入数据形状：(32, 784)
2025-04-11 15:07:05 - INFO - 第 1 层输入形状：(32, 784)，输出形状：(32, 128)
2025-04-11 15:07:05 - INFO - 第 2 层输入形状：(32, 128)，输出形状：(32, 10)
2025-04-11 15:07:05 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:05 - INFO - 开始反向传播过程   Backward propagation...
2025-04-11 15:07:05 - INFO - 反向传播完成   Backward propagation completed.
2025-04-11 15:07:05 - INFO - 计算训练损失中...
2025-04-11 15:07:05 - INFO - 训练损失为：2.4800537555869373
2025-04-11 15:07:05 - INFO - 开始前向传播过程   Forward propagation...

## 第二部分日志
2025-04-11 15:07:09 - INFO - #####################单线程模型训练过程:Epoch
1/10, Loss: 0.3461509632431384
2025-04-11 15:07:13 - INFO - #####################单线程模型训练过程:Epoch
2/10, Loss: 0.2332999796296432
2025-04-11 15:07:17 - INFO - #####################单线程模型训练过程:Epoch
3/10, Loss: 0.21396469711787905
2025-04-11 15:07:22 - INFO - #####################单线程模型训练过程:Epoch
4/10, Loss: 0.20606439872107338
2025-04-11 15:07:26 - INFO - #####################单线程模型训练过程:Epoch
5/10, Loss: 0.2007185153496026
2025-04-11 15:07:31 - INFO - #####################单线程模型训练过程:Epoch
6/10, Loss: 0.1974831297414277
2025-04-11 15:07:35 - INFO - #####################单线程模型训练过程:Epoch
7/10, Loss: 0.1949759157664535
2025-04-11 15:07:40 - INFO - #####################单线程模型训练过程:Epoch
8/10, Loss: 0.19263796069351002
2025-04-11 15:07:45 - INFO - #####################单线程模型训练过程:Epoch
9/10, Loss: 0.191022098908432
2025-04-11 15:07:51 - INFO - #####################单线程模型训练过程:Epoch
10/10, Loss: 0.19009780222806535

## 第三部分日志
```

```
2025-04-11 15:07:51 - INFO - 开始保存模型到文件: mlp_classification.npz
2025-04-11 15:07:51 - INFO - 模型数据保存完成, 共保存了 11 个参数。
2025-04-11 15:07:52 - INFO - 开始模型预测过程...
2025-04-11 15:07:52 - INFO - 开始前向传播过程   Forward propagation...
2025-04-11 15:07:52 - INFO - 输入数据形状: (9999, 784)
2025-04-11 15:07:52 - INFO - 第 1 层输入形状: (9999, 784), 输出形状: (9999,
128)
2025-04-11 15:07:52 - INFO - 第 2 层输入形状: (9999, 128), 输出形状: (9999,
10)
2025-04-11 15:07:52 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:52 - INFO - MNIST 分类准确率: 96.83%
2025-04-11 15:07:52 - INFO - 开始模型预测过程...
2025-04-11 15:07:52 - INFO - 开始前向传播过程   Forward propagation...
2025-04-11 15:07:52 - INFO - 输入数据形状: (9999, 784)
2025-04-11 15:07:52 - INFO - 第 1 层输入形状: (9999, 784), 输出形状: (9999,
128)
2025-04-11 15:07:52 - INFO - 第 2 层输入形状: (9999, 128), 输出形状: (9999,
10)
2025-04-11 15:07:52 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:52 - INFO - MNIST 混淆矩阵:
 [[ 967    0    1    0    0    5    3    1    3    0]
 [   0 1129    2    1    0    0    1    0    2    0]
 [   6    6  990    5    6    0    5    7    7    0]
 [   0    1    8  964    0   26    0    7    3    1]
 [   1    0    2    0  969    0    5    0    2    3]
 [   2    1    0    0    0  885    1    0    2    1]
 [   3    3    1    0    8   43  898    0    2    0]
 [   2   12   10    0    5    1    0  988    1    8]
 [   3    4    1    7    4   14    0    2  936    3]
 [   5    6    0    7   18   10    1    5    1  956]]
```

运行过程中的截图

```
lifeng@lifeng ~> conda activate mlp_env                                   (base)
lifeng@lifeng ~> cd PythonProject/MLP/                                 (mlp_env)
lifeng@lifeng ~/P/MLP> python mlp.py                                   (mlp_env)
2025-04-11 15:06:52 - INFO - 分类任务（加载MNIST数据集）loading..............
2025-04-11 15:06:52 - INFO - 开始加载MNIST数据集，文件路径：MNIST_data/mnist_train.csv
2025-04-11 15:07:03 - INFO - 开始加载MNIST数据集，文件路径：MNIST_data/mnist_train.csv
2025-04-11 15:07:03 - INFO - 开始加载MNIST数据集，文件路径：MNIST_data/mnist_test.csv
2025-04-11 15:07:05 - INFO - 开始加载MNIST数据集，文件路径：MNIST_data/mnist_test.csv
2025-04-11 15:07:05 - INFO - 开始模型初始化过程   Initializing MLP model...
2025-04-11 15:07:05 - INFO -
=============== 输出模型结构摘要信息    Model Summary ================
2025-04-11 15:07:05 - INFO - Layer  Type    Neurons  Activation Weights Shape  Bias Shape   Parameters
2025-04-11 15:07:05 - INFO - ----------------------------------------------------------------------
2025-04-11 15:07:05 - INFO - 0      Input   784      -          -              -            0
2025-04-11 15:07:05 - INFO - 1      Hidden  128      relu       (784, 128)     (1, 128)     100,480
2025-04-11 15:07:05 - INFO - 2      Output  10       softmax    (128, 10)      (1, 10)      1,290
2025-04-11 15:07:05 - INFO -
========================= Summary ==========================
2025-04-11 15:07:05 - INFO - Total layers: 3 (input + 1 hidden + output)
2025-04-11 15:07:05 - INFO - Total parameters: 101,770
2025-04-11 15:07:05 - INFO - Weight Initialization: xavier
2025-04-11 15:07:05 - INFO - Regularization: l2
2025-04-11 15:07:05 - INFO - Lambda: 0.001
2025-04-11 15:07:05 - INFO - Optimizer: Adam (lr=0.001)
2025-04-11 15:07:05 - INFO - ============================================================

2025-04-11 15:07:05 - INFO - 开始模型训练过程，共 10 轮，批次大小 32
2025-04-11 15:07:05 - INFO - 开始前向传播过程   Forward propagation...
2025-04-11 15:07:05 - INFO - 输入数据形状：(32, 784)
2025-04-11 15:07:05 - INFO - 第 1 层输入形状：(32, 784)，输出形状：(32, 128)
2025-04-11 15:07:05 - INFO - 第 2 层输入形状：(32, 128)，输出形状：(32, 10)
2025-04-11 15:07:05 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:05 - INFO - 开始反向传播过程   Backward propagation...
2025-04-11 15:07:05 - INFO - 反向传播完成   Backward propagation completed.
2025-04-11 15:07:05 - INFO - 计算训练损失中...
2025-04-11 15:07:05 - INFO - 训练损失为：2.4800537555869373
2025-04-11 15:07:05 - INFO - 开始前向传播过程   Forward propagation...
2025-04-11 15:07:05 - INFO - 输入数据形状：(32, 784)
2025-04-11 15:07:05 - INFO - 第 1 层输入形状：(32, 784)，输出形状：(32, 128)
2025-04-11 15:07:05 - INFO - 第 2 层输入形状：(32, 128)，输出形状：(32, 10)
2025-04-11 15:07:05 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:05 - INFO - 开始反向传播过程   Backward propagation...
2025-04-11 15:07:05 - INFO - 反向传播完成   Backward propagation completed.
```


```
=============== 输出模型结构摘要信息    Model Summary ================
2025-04-11 15:07:05 - INFO - Layer  Type    Neurons  Activation Weights Shape  Bias Shape   Parameters
2025-04-11 15:07:05 - INFO - ----------------------------------------------------------------------
2025-04-11 15:07:05 - INFO - 0      Input   784      -          -              -            0
2025-04-11 15:07:05 - INFO - 1      Hidden  128      relu       (784, 128)     (1, 128)     100,480
2025-04-11 15:07:05 - INFO - 2      Output  10       softmax    (128, 10)      (1, 10)      1,290
2025-04-11 15:07:05 - INFO -
========================= Summary ==========================
2025-04-11 15:07:05 - INFO - Total layers: 3 (input + 1 hidden + output)
2025-04-11 15:07:05 - INFO - Total parameters: 101,770
2025-04-11 15:07:05 - INFO - Weight Initialization: xavier
2025-04-11 15:07:05 - INFO - Regularization: l2
2025-04-11 15:07:05 - INFO - Lambda: 0.001
2025-04-11 15:07:05 - INFO - Optimizer: Adam (lr=0.001)
2025-04-11 15:07:05 - INFO - ============================================================
```

```
2025-04-11 15:07:51 - INFO - 第 2 层输入形状: (32, 128), 输出形状: (32, 10)
2025-04-11 15:07:51 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:51 - INFO - 开始反向传播过程   Backward propagation...
2025-04-11 15:07:51 - INFO - 反向传播完成   Backward propagation completed.
2025-04-11 15:07:51 - INFO - 计算训练损失中...
2025-04-11 15:07:51 - INFO - 训练损失为: 0.5780710524474804
2025-04-11 15:07:51 - INFO - 开始前向传播过程   Forward propagation...
2025-04-11 15:07:51 - INFO - 输入数据形状: (32, 784)
2025-04-11 15:07:51 - INFO - 第 1 层输入形状: (32, 784), 输出形状: (32, 128)
2025-04-11 15:07:51 - INFO - 第 2 层输入形状: (32, 128), 输出形状: (32, 10)
2025-04-11 15:07:51 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:51 - INFO - 开始反向传播过程   Backward propagation...
2025-04-11 15:07:51 - INFO - 反向传播完成   Backward propagation completed.
2025-04-11 15:07:51 - INFO - 计算训练损失中...
2025-04-11 15:07:51 - INFO - 训练损失为: 0.1455681153000835
2025-04-11 15:07:51 - INFO - #####################单线程模型训练过程:Epoch 10/10, Loss: 0.19009780222806535
2025-04-11 15:07:51 - INFO - 开始保存模型到文件: mlp_classification.npz
2025-04-11 15:07:51 - INFO - 模型数据保存完成, 共保存了 11 个参数。
2025-04-11 15:07:52 - INFO - 开始模型预测过程...
2025-04-11 15:07:52 - INFO - 开始前向传播过程   Forward propagation...
2025-04-11 15:07:52 - INFO - 输入数据形状: (9999, 784)
2025-04-11 15:07:52 - INFO - 第 1 层输入形状: (9999, 784), 输出形状: (9999, 128)
2025-04-11 15:07:52 - INFO - 第 2 层输入形状: (9999, 128), 输出形状: (9999, 10)
2025-04-11 15:07:52 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:52 - INFO - MNIST 分类准确率: 96.83%
2025-04-11 15:07:52 - INFO - 开始模型预测过程...
2025-04-11 15:07:52 - INFO - 开始前向传播过程   Forward propagation...
2025-04-11 15:07:52 - INFO - 输入数据形状: (9999, 784)
2025-04-11 15:07:52 - INFO - 第 1 层输入形状: (9999, 784), 输出形状: (9999, 128)
2025-04-11 15:07:52 - INFO - 第 2 层输入形状: (9999, 128), 输出形状: (9999, 10)
2025-04-11 15:07:52 - INFO - 前向传播完成   Forward propagation completed.
2025-04-11 15:07:52 - INFO - MNIST 混淆矩阵:
 [[ 967    0    1    0    0    5    3    1    3    0]
 [   0 1129    2    1    0    0    1    0    2    0]
 [   6    6  990    5    6    0    5    7    7    0]
 [   0    1    8  964    0   26    0    7    3    1]
 [   1    0    2    0  969    0    5    0    2    3]
 [   2    1    0    0    0  885    1    0    2    1]
 [   3    3    1    0    8   43  898    0    2    0]
 [   2   12   10    0    5    1    0  988    1    8]
 [   3    4    1    7    4   14    0    2  936    3]
 [   5    6    0    7   18   10    1    5    1  956]]
lifeng@lifeng ~/P/MLP>
```

六、后续优化，训练过程增加训练指标的可视化

(1)python 代码

```
# 新增TensorBoard可视化类
class TensorBoardLogger:
```

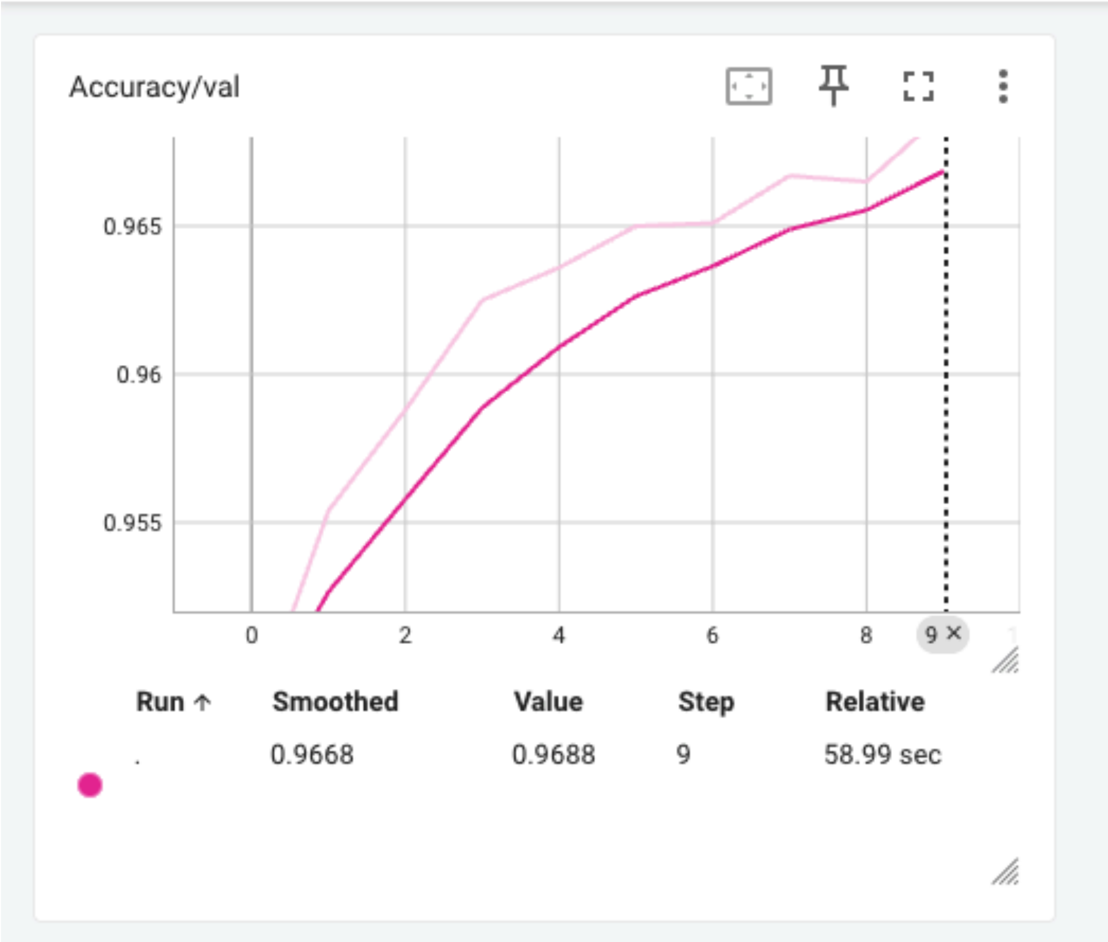
```python
    def __init__(self, log_dir="logs"):
        self.writer = SummaryWriter(log_dir)
        self.step = 0

    def log_scalar(self, tag, value):
        """记录标量数据"""
        self.writer.add_scalar(tag, value, self.step)

    def log_histogram(self, tag, values):
        """记录直方图数据"""
        self.writer.add_histogram(tag, values, self.step)

    def increment_step(self):
        self.step += 1

    def close(self):
        self.writer.close()
```

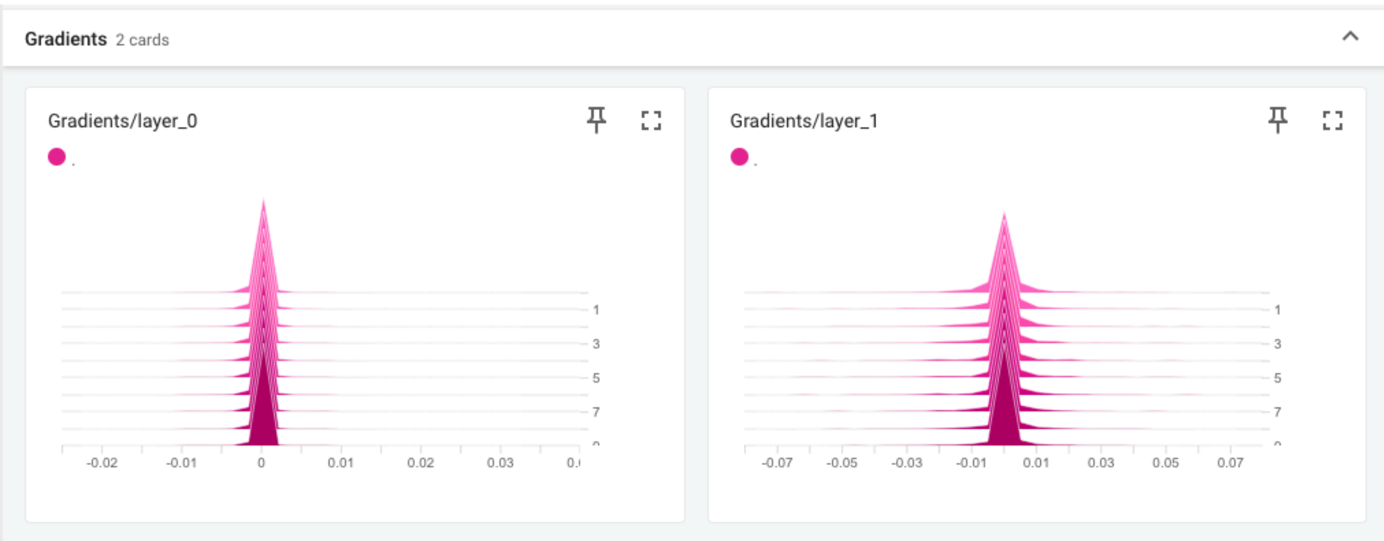
(2)运行程序文件

```
python mlp_opt_tensorboard.py
tensorboard --logdir=logs
用浏览器打开提示的URL（通常是 http://localhost:6006)
```

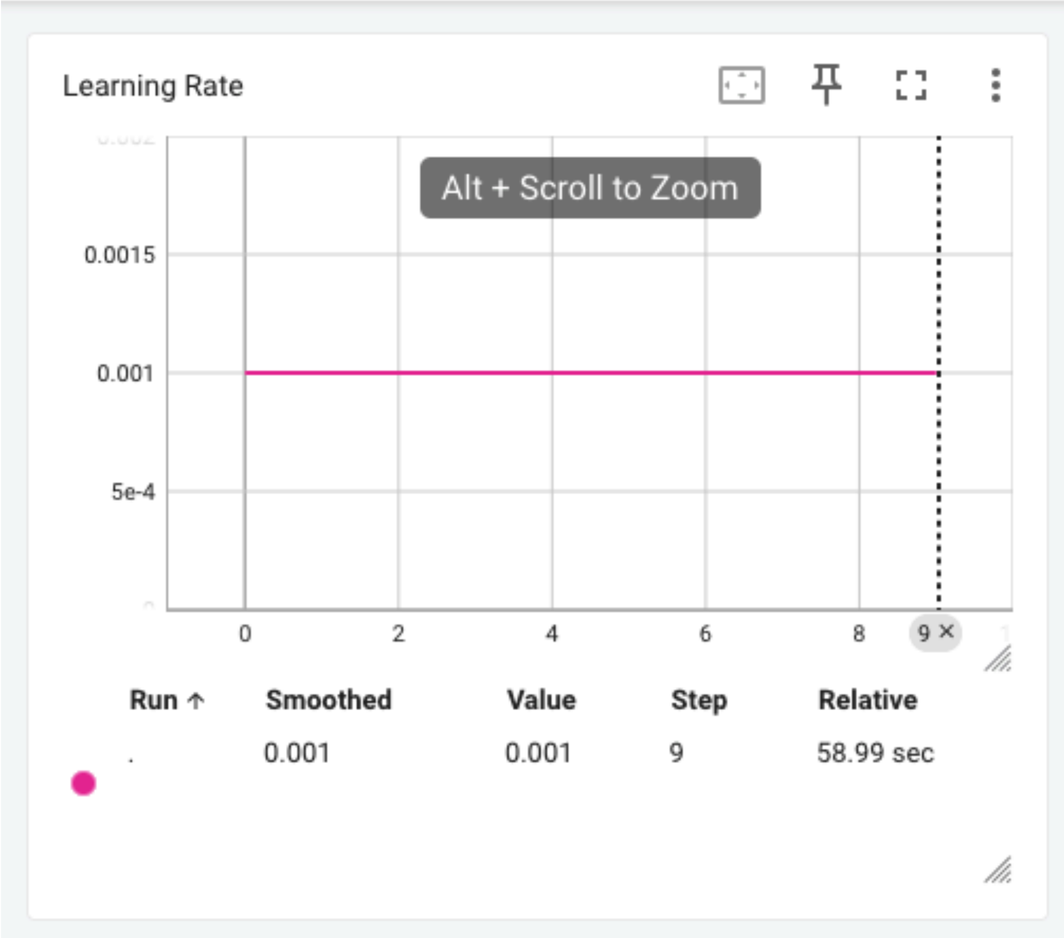
(3)程序输出

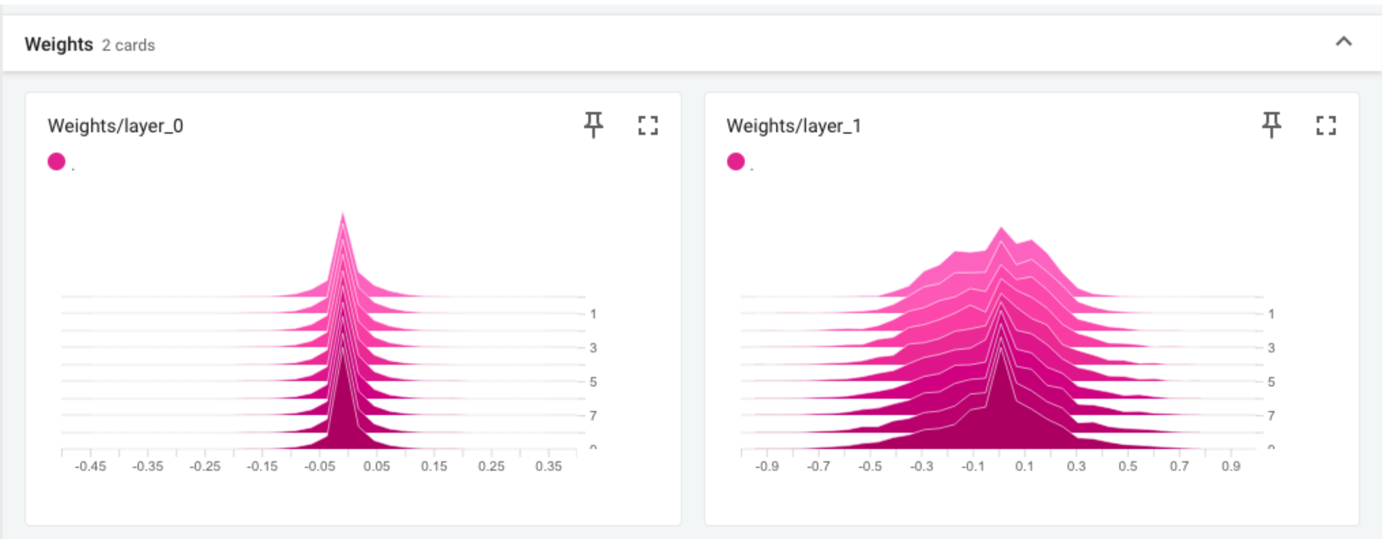
Accuracy


Accuracy/val
0.965
0.96
0.955
0
2
4
6
8
9
Run
Smoothed
Value
Step
Relative
.
0.9668
0.9688
9
58.99 sec


Gradients 2 cards
Gradients/layer_0
Gradients/layer_1
-0.02
-0.01
0
0.01
0.02
0.03
-0.07
-0.05
-0.03
-0.01
0.01
0.03
0.05
0.07
1
3
5
7

Learning Rate





七、加载测试MLP模型

(1) python代码

```
def visualize_predictions(model_path, X_test, y_test, img_shape=(28, 28)):
    """
```

```
    可视化模型预测结果
    :param model_path: 模型文件路径
    :param X_test: 测试集特征数据
    :param y_test: 测试集真实标签 (one-hot编码)
    :param img_shape: 图像尺寸
    """
    # 加载模型
    model = MLP.load_model(model_path)

    # 随机选择20个样本 (不重复)
    # np.random.seed(42)
    sample_indices = np.random.choice(len(X_test), 20, replace=False)
    X_test_samples = X_test[sample_indices]
    y_test_samples = y_test[sample_indices]

    # 生成预测结果
    y_pred = model.predict(X_test_samples)
    y_true = np.argmax(y_test_samples, axis=1)

    # 创建可视化窗口
    plt.figure(num="MLP 手写数字识别 MNIST 图像可视化",figsize=(12, 10))
    for i, (img, true_label, pred_label) in enumerate(zip(X_test_samples,
y_true, y_pred)):
        plt.subplot(5, 4, i + 1)
        plt.imshow(img.reshape(img_shape), cmap='gray')
        plt.axis('off')

        # 使用不同颜色标注正确/错误预测
        color = 'green' if true_label == pred_label else 'blue'
        plt.title(f"Index: {sample_indices[i]}\n True:
{true_label}\nPred: {pred_label}", fontweight='bold', color=color)

    plt.tight_layout()
    plt.show()

    # 加载MNIST数据集
def load_mnist_data(file_path):
    """
    加载MNIST数据集
    :param file_path: 数据集文件路径
    :return: 特征和标签
```

```
    """
    data = []
    with open(file_path, 'r') as file:
        reader = csv.reader(file)
        next(reader)  # 跳过标题行
        for row in reader:
            data.append([int(x) for x in row])
    data = np.array(data)
    X = data[:, 1:] / 255.0
    y = np.eye(10)[data[:, 0]]
    return X, y
# 在主函数中添加调用示例
if __name__ == "__main__":
    # ...原有代码...
                    random_state=42)
    test_path = 'MNIST_data/mnist_test.csv'

    if os.path.exists(test_path):
        X_test_mnist, y_test_mnist = load_mnist_data(test_path)
    # 可视化预测结果（添加在模型训练之后）
    visualize_predictions('mlp_classification.npz',
                          X_test_mnist,
                          y_test_mnist)
```

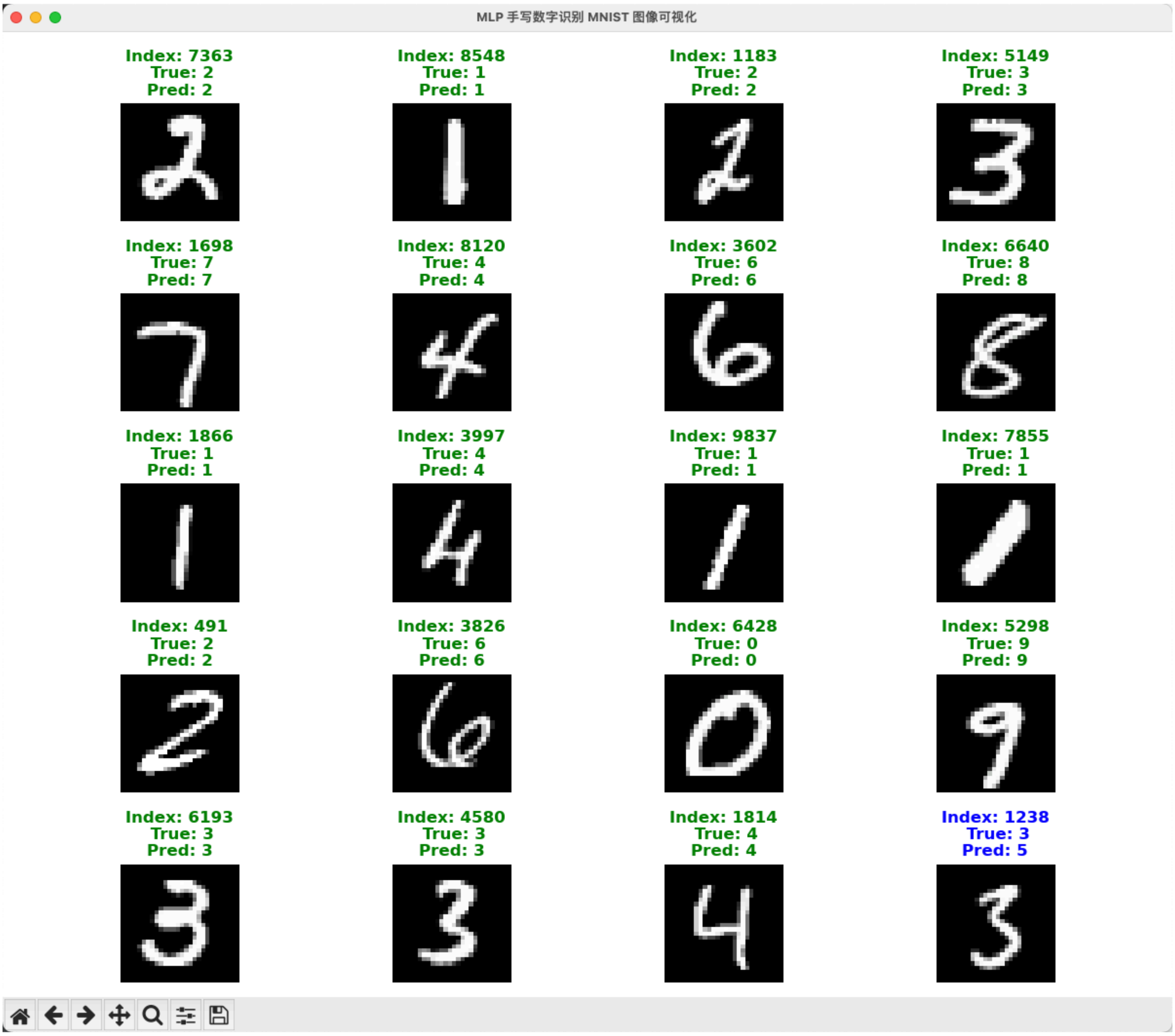
(2)运行程序文件

```
python mlp_test_show.py
```

(3)程序输出

```
2025-04-11 16:49:46 - INFO - 开始加载模型文件: mlp_classification.npz
2025-04-11 16:49:46 - INFO - 开始模型初始化过程   Initializing MLP model...
2025-04-11 16:49:46 - INFO -
=============== 输出模型结构摘要信息    Model Summary ================
2025-04-11 16:49:46 - INFO - Layer  Type     Neurons  Activation Weights
Shape    Bias Shape    Parameters
2025-04-11 16:49:46 - INFO - --------------------------------------------
--------------------------
2025-04-11 16:49:46 - INFO - 0      Input    784      -          -
        -            0
```

```
2025-04-11 16:49:46 - INFO - 1       Hidden    128       relu       (784,
128)       (1, 128)      100,480
2025-04-11 16:49:46 - INFO - 2       Output    10        softmax    (128,
10)        (1, 10)       1,290
2025-04-11 16:49:46 - INFO -
========================= Summary ==========================
2025-04-11 16:49:46 - INFO - Total layers: 3 (input + 1 hidden + output)
2025-04-11 16:49:46 - INFO - Total parameters: 101,770
2025-04-11 16:49:46 - INFO - Weight Initialization: xavier
2025-04-11 16:49:46 - INFO - Regularization: l2
2025-04-11 16:49:46 - INFO - Lambda: 0.001
2025-04-11 16:49:46 - INFO - Optimizer: SGD (lr=0.01)
2025-04-11 16:49:46 - INFO -
============================================================


2025-04-11 16:49:46 - INFO - 模型加载完成，返回模型对象...
2025-04-11 16:49:46 - INFO - 开始模型预测过程...
```



其中蓝色标记为预测错误数据。

八、Pytorch版的MLP

(1) python代码

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torch.utils.tensorboard import SummaryWriter
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import numpy as np
import itertools
```

```python
import pandas as pd
import logging

# Configure logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S')


class MNISTDataset(Dataset):
    """
    自定义数据集类，用于从 CSV 文件中加载 MNIST 数据。
    该类继承自 torch.utils.data.Dataset，需要实现 __len__ 和 __getitem__ 方
法。
    """

    def __init__(self, csv_file):
        """
        初始化函数，读取 CSV 文件并将数据转换为 PyTorch 张量。
        :param csv_file: CSV 文件的路径
        """
        self.data = pd.read_csv(csv_file)
        self.labels = torch.tensor(self.data.iloc[:, 0].values,
dtype=torch.long)
        self.images = torch.tensor(self.data.iloc[:, 1:].values,
dtype=torch.float32) / 255.0

    def __len__(self):
        """
        返回数据集的长度。
        :return: 数据集的长度
        """
        return len(self.data)

    def __getitem__(self, idx):
        """
        根据索引返回图像和对应的标签。
        :param idx: 索引
        :return: 图像和对应的标签
        """
        image = self.images[idx].view(1, 28, 28)
```

```python
        label = self.labels[idx]
        return image, label


class MLP(nn.Module):
    """
    多层感知机（MLP）模型类，继承自 nn.Module。
    该模型由多个全连接层和激活函数组成。
    """

    def __init__(self, input_size, hidden_sizes, output_size,
activation='relu'):
        """
        初始化函数，定义模型的结构。
        :param input_size: 输入层的大小
        :param hidden_sizes: 隐藏层大小的列表
        :param output_size: 输出层的大小
        :param activation: 激活函数的类型，默认为 'relu'
        """
        super(MLP, self).__init__()
        layers = []
        sizes = [input_size] + hidden_sizes + [output_size]
        for i in range(len(sizes) - 1):
            layers.append(nn.Linear(sizes[i], sizes[i + 1]))
            if i < len(sizes) - 2:
                if activation == 'relu':
                    layers.append(nn.ReLU())
                elif activation == 'sigmoid':
                    layers.append(nn.Sigmoid())
                elif activation == 'tanh':
                    layers.append(nn.Tanh())
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        """
        前向传播函数，定义模型的前向计算过程。
        :param x: 输入数据
        :return: 模型的输出
        """
        # logging.info(f"Input shape to MLP: {x.shape}")
        output = self.model(x)
```

```python
        # logging.info(f"Output shape from MLP: {output.shape}")
        return output


def initialize_weights(model, init_type='xavier'):
    """
    初始化模型的权重。
    :param model: 要初始化的模型
    :param init_type: 初始化方法的类型，默认为 'xavier'
    """
    for m in model.modules():
        if isinstance(m, nn.Linear):
            if init_type == 'xavier':
                nn.init.xavier_uniform_(m.weight)
            elif init_type == 'kaiming':
                nn.init.kaiming_uniform_(m.weight, nonlinearity='relu')
            elif init_type == 'zeros':
                nn.init.zeros_(m.weight)


def get_optimizer(model, optimizer_type='adam', lr=0.001, momentum=0.9):
    """
    根据指定的优化器类型和学习率返回优化器。
    :param model: 要优化的模型
    :param optimizer_type: 优化器的类型，默认为 'adam'
    :param lr: 学习率，默认为 0.001
    :param momentum: 动量，仅在使用 SGD 优化器时有效，默认为 0.9
    :return: 优化器对象
    """
    if optimizer_type == 'sgd':
        return optim.SGD(model.parameters(), lr=lr, momentum=momentum)
    elif optimizer_type == 'rmsprop':
        return optim.RMSprop(model.parameters(), lr=lr)
    elif optimizer_type == 'adam':
        return optim.Adam(model.parameters(), lr=lr)


def get_regularization_loss(model, reg_type='l2', reg_lambda=0.001):
    """
    计算正则化损失。
    :param model: 模型
```

```
    :param reg_type: 正则化类型，默认为 'l2'
    :param reg_lambda: 正则化系数，默认为 0.001
    :return: 正则化损失
    """
    if reg_type == 'l1':
        return reg_lambda * sum(p.abs().sum() for p in
model.parameters())
    elif reg_type == 'l2':
        return reg_lambda * sum(p.pow(2).sum() for p in
model.parameters())
    elif reg_type == 'elastic':
        l1_loss = sum(p.abs().sum() for p in model.parameters())
        l2_loss = sum(p.pow(2).sum() for p in model.parameters())
        return reg_lambda * (0.5 * l1_loss + 0.5 * l2_loss)
    return 0


def train_model(model, train_loader, test_loader, criterion, optimizer,
num_epochs, device,
                reg_type='l2', reg_lambda=0.001, writer=None):
    """
    训练模型的函数。
    :param model: 要训练的模型
    :param train_loader: 训练数据加载器
    :param test_loader: 测试数据加载器
    :param criterion: 损失函数
    :param optimizer: 优化器
    :param num_epochs: 训练的轮数
    :param device: 设备 (CPU 或 GPU)
    :param reg_type: 正则化类型，默认为 'l2'
    :param reg_lambda: 正则化系数，默认为 0.001
    :param writer: TensorBoard 写入器，默认为 None
    """
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        for i, (images, labels) in enumerate(train_loader):
            images, labels = images.to(device), labels.to(device)
            images = images.view(images.size(0), -1)
```

```
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            reg_loss = get_regularization_loss(model, reg_type,
reg_lambda)
            total_loss = loss + reg_loss
            total_loss.backward()

            # 记录各层梯度变化
            for name, param in model.named_parameters():
                if param.grad is not None:
                    writer.add_histogram(f'Gradients/{name}', param.grad,
epoch)

            optimizer.step()
            running_loss += total_loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        # 计算训练集准确率
        train_accuracy = 100 * correct / total
        avg_loss = running_loss / len(train_loader)

        # 记录训练损失和准确率
        if writer:
            writer.add_scalar('Training Loss', avg_loss, epoch)
            writer.add_scalar('Training Accuracy', train_accuracy, epoch)

        # 记录学习率
        lr = optimizer.param_groups[0]['lr']
        writer.add_scalar('Learning Rate', lr, epoch)

        # 记录各层权重变化
        for name, param in model.named_parameters():
            writer.add_histogram(f'Weights/{name}', param, epoch)

        # 在测试集上评估模型
        test_accuracy , cm= evaluate_model(model, test_loader, device,
writer, epoch)
```

```
        logging.info(
            f'Epoch {epoch + 1}/{num_epochs}, Loss: {avg_loss:.4f}, Train
Acc: {train_accuracy:.2f}%, Test Acc: {test_accuracy:.2f}%')


def evaluate_model(model, test_loader, device, writer=None, epoch=None):
    """
    评估模型在测试集上的性能。

    Args:
        model (nn.Module): 要评估的模型。
        test_loader (DataLoader): 测试数据加载器。
        device (torch.device): 评估设备（如'cuda'或'cpu'）。
        writer (SummaryWriter, optional): TensorBoard的SummaryWriter对象，
用于记录测试指标。
        epoch (int, optional): 当前训练轮数，用于在TensorBoard中记录不同轮数的
测试指标。

    Returns:
        float: 模型在测试集上的准确率
        np.ndarray: 混淆矩阵
    """
    model.eval()
    correct = 0
    total = 0
    all_labels = []
    all_preds = []
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            images = images.view(images.size(0), -1)
            # logging.info(f"Test batch input image shape:
{images.shape}, label shape: {labels.shape}")
            outputs = model(images)
            # logging.info(f"Test batch model output shape:
{outputs.shape}")
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            all_labels.extend(labels.cpu().numpy())
```

```python
            all_preds.extend(predicted.cpu().numpy())
    accuracy = 100 * correct / total
    cm = confusion_matrix(all_labels, all_preds)
    if writer and epoch is not None:
        writer.add_scalar('Test Accuracy', accuracy, epoch)
    return accuracy, cm


def plot_confusion_matrix(cm, classes):

    """
     以日志形式输出混淆矩阵
     :param cm: 混淆矩阵数据
     :param classes: 类别标签列表
     """
    logging.info("Confusion Matrix:")
    header = "     " + " ".join(f"{c:5}" for c in classes)
    logging.info(header)
    for i, row in enumerate(cm):
        row_str = f"{classes[i]:5}" + " ".join(f"{v:5}" for v in row)
        logging.info(row_str)


if __name__ == "__main__":
    train_csv_path = 'MNIST_data/mnist_train.csv'
    test_csv_path = 'MNIST_data/mnist_test.csv'

    train_dataset = MNISTDataset(train_csv_path)
    test_dataset = MNISTDataset(test_csv_path)

    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

    input_size = 28 * 28
    hidden_sizes = [128, 64]
    output_size = 10
    activation = 'relu'
    init_type = 'xavier'
    optimizer_type = 'adam'
    lr = 0.001
    num_epochs = 10
```

```
    reg_type = 'l2'
    reg_lambda = 0.001

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    model = MLP(input_size, hidden_sizes, output_size,
activation).to(device)
    logging.info(f"Model structure:\n{model}")

    initialize_weights(model, init_type)
    criterion = nn.CrossEntropyLoss()
    optimizer = get_optimizer(model, optimizer_type, lr)

    writer = SummaryWriter('runs/mlp_experiment')

    train_model(model, train_loader, test_loader, criterion, optimizer,
num_epochs, device,
                reg_type, reg_lambda, writer)

    torch.save(model.state_dict(), 'v1_torch_mlp_model.pth')

    test_accuracy, cm = evaluate_model(model, test_loader, device)

    classes = [str(i) for i in range(10)]
    plot_confusion_matrix(cm, classes)

    writer.close()
```

(2) 运行程序文件

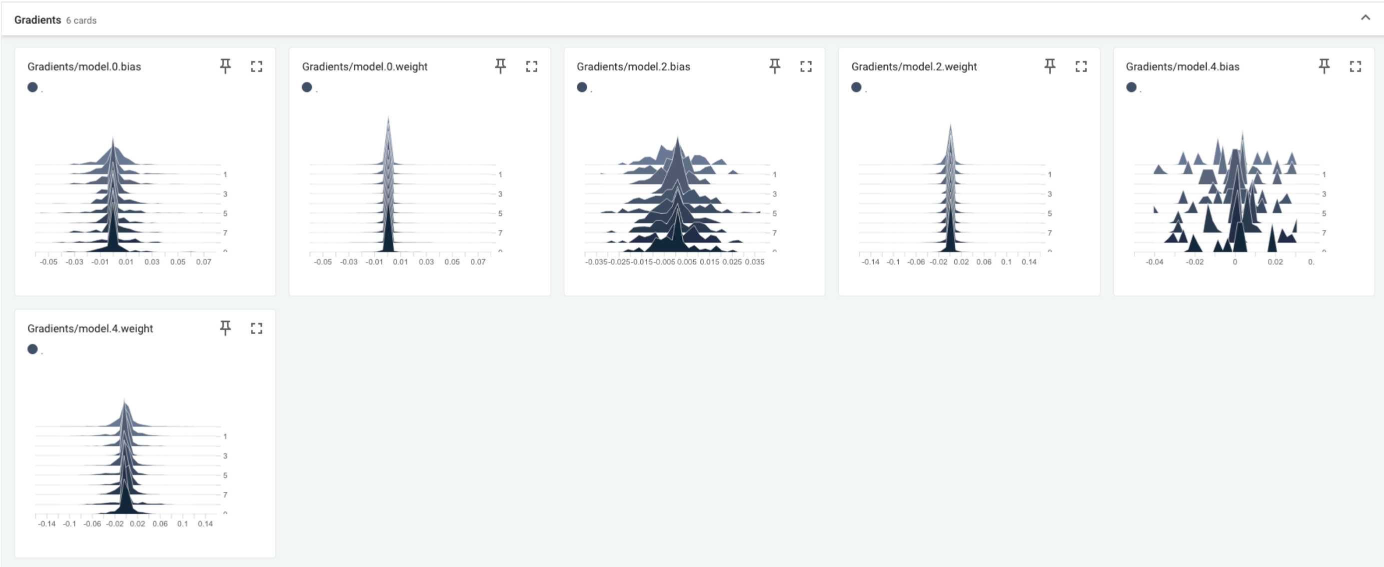
```
python v1_torch_mlp_implementation.py
```

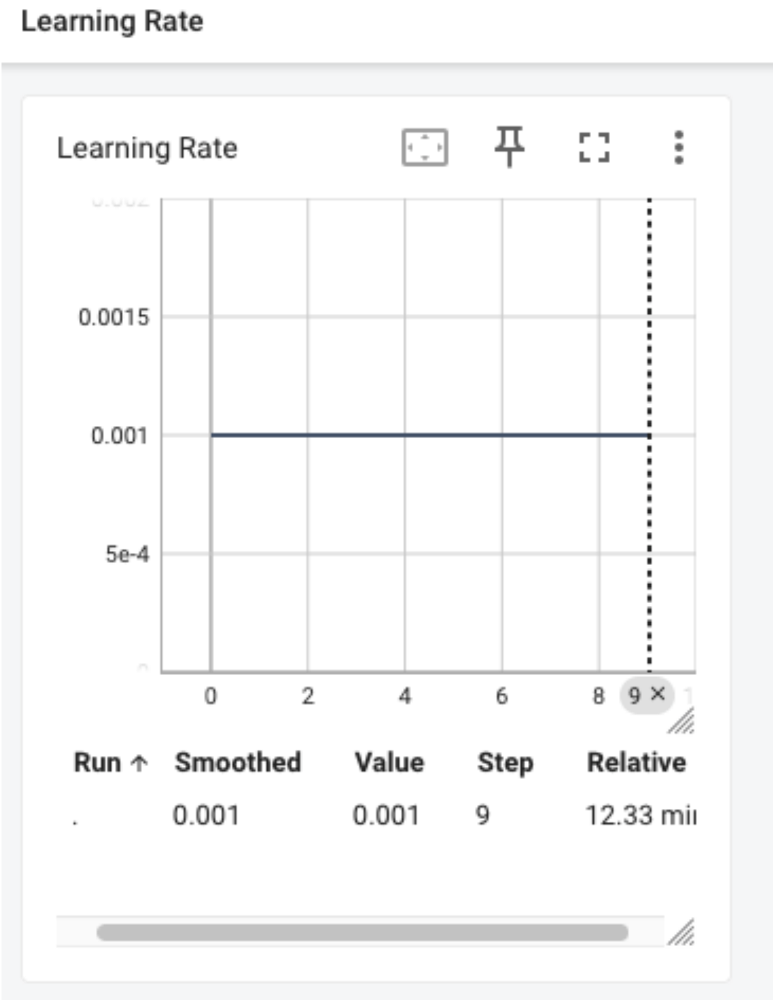
(3)程序输出

```
lifeng@lifeng ~/P/MLP> python v1_torch_mlp_implementation.py                                   (mlp_env)
2025-04-28 22:58:22 - INFO - Model structure:
MLP(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=10, bias=True)
  )
)
2025-04-28 22:59:27 - INFO - Epoch 1/10, Loss: 0.4532, Train Acc: 92.26%, Test Acc: 95.69%
2025-04-28 23:00:41 - INFO - Epoch 2/10, Loss: 0.2930, Train Acc: 95.90%, Test Acc: 96.46%
2025-04-28 23:01:57 - INFO - Epoch 3/10, Loss: 0.2627, Train Acc: 96.57%, Test Acc: 96.80%
2025-04-28 23:03:16 - INFO - Epoch 4/10, Loss: 0.2523, Train Acc: 96.82%, Test Acc: 97.03%
2025-04-28 23:04:38 - INFO - Epoch 5/10, Loss: 0.2462, Train Acc: 96.85%, Test Acc: 96.82%
2025-04-28 23:06:03 - INFO - Epoch 6/10, Loss: 0.2412, Train Acc: 96.98%, Test Acc: 96.96%
2025-04-28 23:07:27 - INFO - Epoch 7/10, Loss: 0.2377, Train Acc: 97.03%, Test Acc: 97.10%
2025-04-28 23:08:56 - INFO - Epoch 8/10, Loss: 0.2364, Train Acc: 97.05%, Test Acc: 97.01%
2025-04-28 23:10:21 - INFO - Epoch 9/10, Loss: 0.2323, Train Acc: 97.16%, Test Acc: 97.25%
2025-04-28 23:11:46 - INFO - Epoch 10/10, Loss: 0.2326, Train Acc: 97.16%, Test Acc: 97.17%
2025-04-28 23:11:47 - INFO - Confusion Matrix:
2025-04-28 23:11:47 - INFO -      0     1     2     3     4     5     6     7     8     9
2025-04-28 23:11:47 - INFO - 0      969     0     2     2     0     2     1     1     3     0
2025-04-28 23:11:47 - INFO - 1        0  1112     4     3     0     0     1     0    15     0
2025-04-28 23:11:47 - INFO - 2        4     1   994    16     5     0     0     9     3     0
2025-04-28 23:11:47 - INFO - 3        0     0     2   997     0     3     0     5     2     1
2025-04-28 23:11:47 - INFO - 4        2     0     7     0   958     0     1     1     2    11
2025-04-28 23:11:47 - INFO - 5        3     0     0    19     2   856     2     1     6     3
2025-04-28 23:11:47 - INFO - 6        8     3     5     0     4    10   919     0     9     0
2025-04-28 23:11:47 - INFO - 7        1     5     9     4     1     0     0  1002     0     5
2025-04-28 23:11:47 - INFO - 8        3     0     1    22     4     3     0     5   934     2
2025-04-28 23:11:47 - INFO - 9        5     2     0    10    11     1     0     5     0   975
```

```
lifeng@lifeng ~/P/MLP> python v1_torch_mlp_implementation.py
                                            (mlp_env)
2025-04-28 22:58:22 - INFO - Model structure:
MLP(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=10, bias=True)
  )
)
2025-04-28 22:59:27 - INFO - Epoch 1/10, Loss: 0.4532, Train Acc: 92.26%,
Test Acc: 95.69%
2025-04-28 23:00:41 - INFO - Epoch 2/10, Loss: 0.2930, Train Acc: 95.90%,
Test Acc: 96.46%
2025-04-28 23:01:57 - INFO - Epoch 3/10, Loss: 0.2627, Train Acc: 96.57%,
Test Acc: 96.80%
```

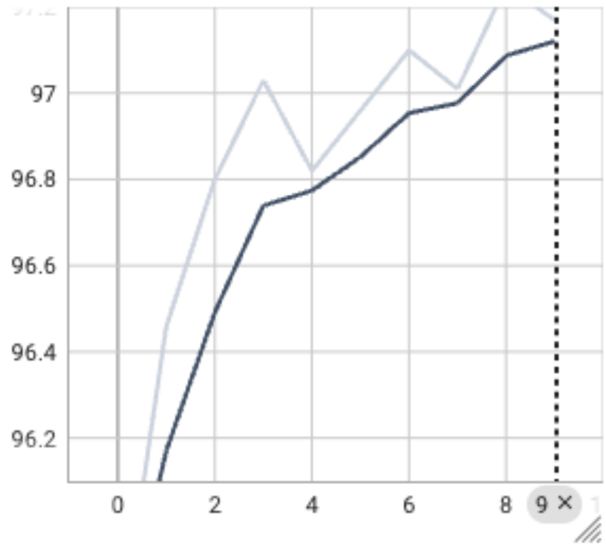
```
2025-04-28 23:03:16 - INFO - Epoch 4/10, Loss: 0.2523, Train Acc: 96.82%,
Test Acc: 97.03%
2025-04-28 23:04:38 - INFO - Epoch 5/10, Loss: 0.2462, Train Acc: 96.85%,
Test Acc: 96.82%
2025-04-28 23:06:03 - INFO - Epoch 6/10, Loss: 0.2412, Train Acc: 96.98%,
Test Acc: 96.96%
2025-04-28 23:07:27 - INFO - Epoch 7/10, Loss: 0.2377, Train Acc: 97.03%,
Test Acc: 97.10%
2025-04-28 23:08:56 - INFO - Epoch 8/10, Loss: 0.2364, Train Acc: 97.05%,
Test Acc: 97.01%
2025-04-28 23:10:21 - INFO - Epoch 9/10, Loss: 0.2323, Train Acc: 97.16%,
Test Acc: 97.25%
2025-04-28 23:11:46 - INFO - Epoch 10/10, Loss: 0.2326, Train Acc:
97.16%, Test Acc: 97.17%
2025-04-28 23:11:47 - INFO - Confusion Matrix:
2025-04-28 23:11:47 - INFO -      0      1      2      3      4      5      6
   7      8      9
2025-04-28 23:11:47 - INFO - 0      969      0      2      2      0      2
1      1      3      0
2025-04-28 23:11:47 - INFO - 1        0   1112      4      3      0      0
1      0     15      0
2025-04-28 23:11:47 - INFO - 2        4      1    994     16      5      0
0      9      3      0
2025-04-28 23:11:47 - INFO - 3        0      0      2    997      0      3
0      5      2      1
2025-04-28 23:11:47 - INFO - 4        2      0      7      0    958      0
1      1      2     11
2025-04-28 23:11:47 - INFO - 5        3      0      0     19      2    856
2      1      6      3
2025-04-28 23:11:47 - INFO - 6        8      3      5      0      4     10
919      0      9      0
```


Gradients 6 cards
Gradients/model.0.bias
Gradients/model.0.weight
Gradients/model.2.bias
Gradients/model.2.weight
Gradients/model.4.bias
Gradients/model.4.weight


Learning Rate
Learning Rate
0.0015
0.001
5e-4
0
2
4
6
8
9
Run
Smoothed
Value
Step
Relative
.
0.001
0.001
9
12.33 mi

Test Accuracy

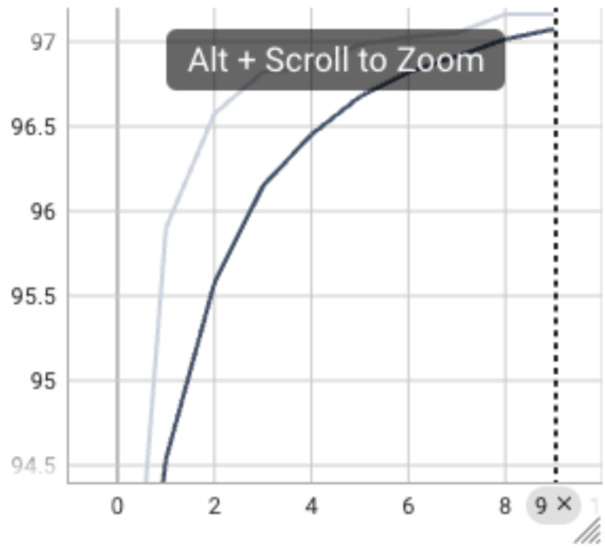
Test Accuracy



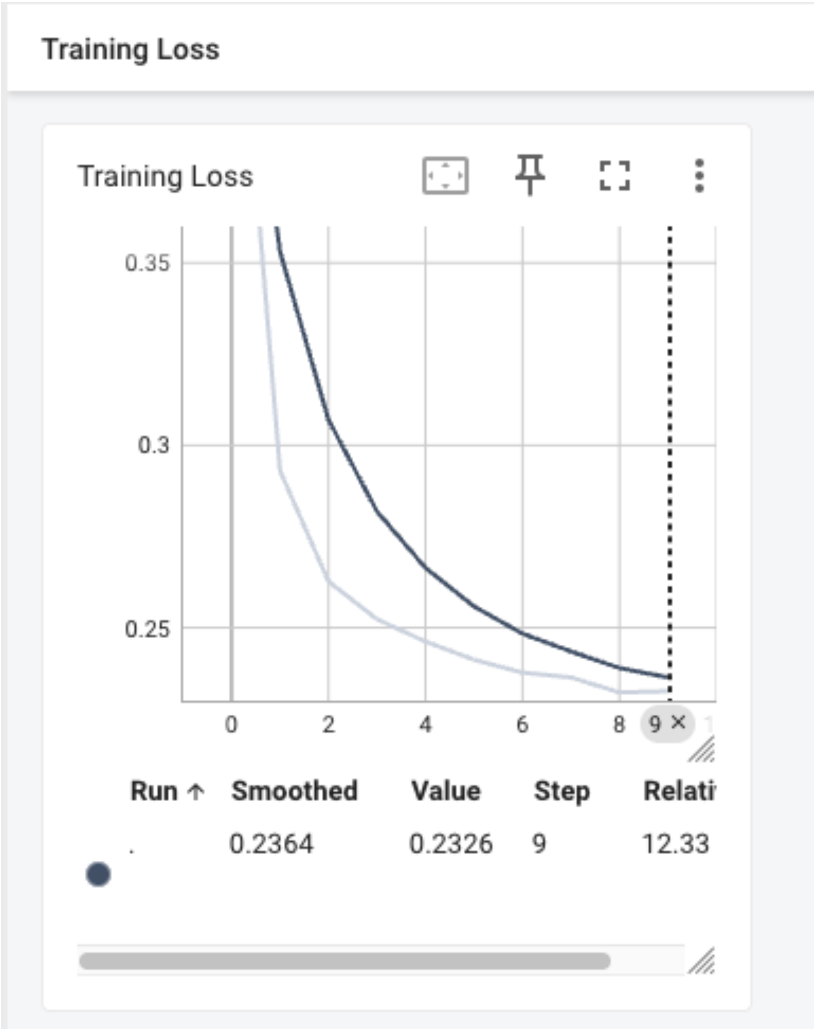
Run ↑	Smoothed	Value	Step	Relative
	97.1203	97.1697	9	12.33 min

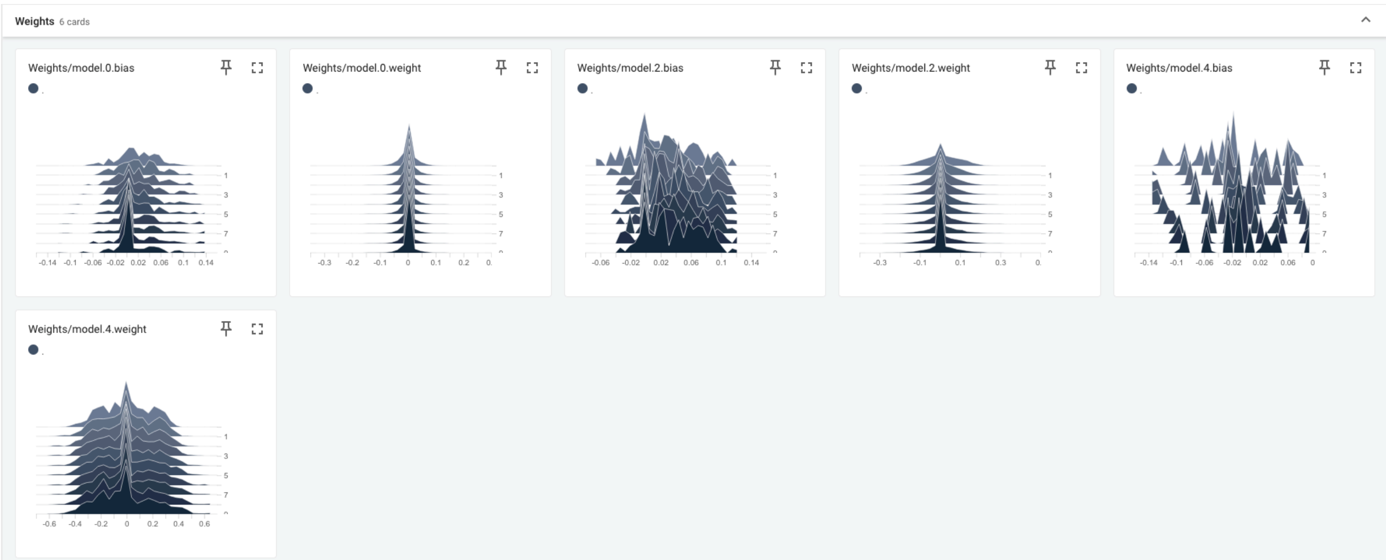
Training Accuracy

Training Accuracy



Run ↑	Smoothed	Value	Step	Relati
.	97.0739	97.1616	9	12.33





九、加载测试Pytorch版的MLP

(1) python代码

```
import torch
import matplotlib.pyplot as plt
import numpy as np
```

```python
import os
import csv
from v1_torch_mlp_implementation import MLP


class MLPTest:
    def __init__(self, model_path, test_data_path):
        """
        初始化函数，设置模型路径和测试数据路径。
        :param model_path: 保存的模型文件路径
        :param test_data_path: 测试数据集文件路径
        """
        self.model_path = model_path
        self.test_data_path = test_data_path
        self.model = self.load_model()
        self.X_test, self.y_test = self.load_mnist_data()

    def load_model(self):
        """
        加载训练好的模型。
        :return: 加载后的模型
        """
        input_size = 28 * 28
        hidden_sizes = [128, 64]
        output_size = 10
        activation ='relu'
        model = MLP(input_size, hidden_sizes, output_size, activation)
        model.load_state_dict(torch.load(self.model_path))
        model.eval()
        return model

    def load_mnist_data(self):
        """
        加载MNIST测试数据集。
        :return: 测试集特征数据和真实标签
        """
        data = []
        with open(self.test_data_path, 'r') as file:
            reader = csv.reader(file)
            next(reader)  # 跳过标题行
            for row in reader:
```

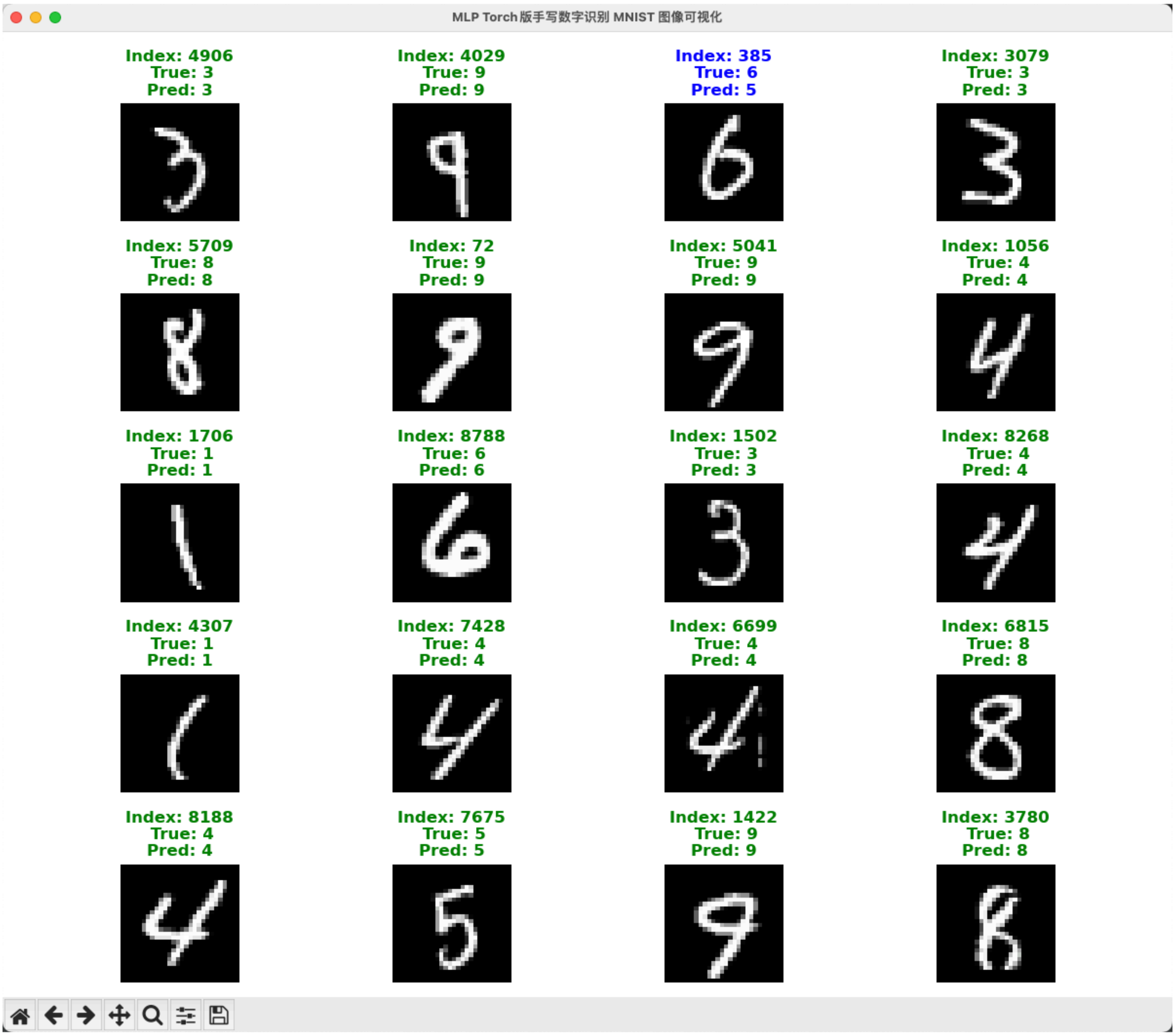
```
                data.append([int(x) for x in row])
        data = np.array(data)
        X = data[:, 1:] / 255.0
        y = np.eye(10)[data[:, 0]]
        return X, y

    def visualize_predictions(self, img_shape=(28, 28)):
        """
        可视化模型预测结果。
        :param img_shape: 图像尺寸，默认为(28, 28)
        """
        # 随机选择20个样本（不重复）
        sample_indices = np.random.choice(len(self.X_test), 20,
replace=False)
        X_test_samples = self.X_test[sample_indices]
        y_test_samples = self.y_test[sample_indices]

        # 将数据转换为PyTorch张量
        X_test_samples_tensor = torch.tensor(X_test_samples,
dtype=torch.float32)

        # 生成预测结果
        with torch.no_grad():
            outputs = self.model(X_test_samples_tensor)
        _, y_pred = torch.max(outputs, 1)
        y_true = np.argmax(y_test_samples, axis=1)

        # 创建可视化窗口
        plt.figure(num="MLP Torch版手写数字识别 MNIST 图像可视化", figsize=
(12, 10))
        for i, (img, true_label, pred_label) in
enumerate(zip(X_test_samples, y_true, y_pred.numpy())):
            plt.subplot(5, 4, i + 1)
            plt.imshow(img.reshape(img_shape), cmap='gray')
            plt.axis('off')

            # 使用不同颜色标注正确/错误预测
            color = 'green' if true_label == pred_label else 'blue'
            plt.title(f"Index: {sample_indices[i]}\n True:
{true_label}\nPred: {pred_label}", fontweight='bold',
                      color=color)
```

```python
        plt.tight_layout()
        plt.show()


if __name__ == "__main__":
    model_path ='v1_torch_mlp_model.pth'  # 替换为你的模型保存路径
    test_data_path = 'MNIST_data/mnist_test.csv'  # 替换为你的测试数据路径
    mlp_tester = MLPTest(model_path, test_data_path)
    mlp_tester.visualize_predictions()
```

(2) 运行程序文件

```
python v1_torch_mlp_test_show.py
```

(3)程序输出



十、对比两个版本的MLP

手写版MLP 10轮次训练后 准确率: 96.83% 训练损失为：0.1455681153000835

torch版MLP 10轮次训练后 准确率: 97.16% 训练损失为: 0.2326

写在最后

> 以上为MLP实现的全过程，请老师给予指导。