NumPy 实现 CNN 实验

1. 项目概述

1.1 实验定位

本实验为**中级难度**，适用于具备 Python 基础、NumPy 使用经验，且了解 CNN 基本概念（卷积、池化、全连接）的学习者，通过纯 NumPy 手写 CNN 核心逻辑，深入理解深度学习模型的底层工作机制。

1.2 实验目标

通过完成 MNIST 手写数字分类任务，您将掌握：

- CNN 核心组件（卷积层、池化层、全连接层）的数学原理与工程实现
- 前向传播的张量计算逻辑、反向传播的梯度链式求导过程
- 深度学习模型的完整生命周期（数据预处理→模型构建→训练→评估→优化）
- 纯 NumPy 实现数值计算的工程技巧（维度匹配、数值稳定性处理）

2. 环境准备

2.1 Python 版本要求

推荐安装 Python 3.8~3.10（兼容性最佳），可通过 Python 官网下载，安装时勾选「Add Python to PATH」。

2.2 依赖库安装与验证

2.2.1 依赖清单

库名	用途	版本要求
numpy	核心数值计算（张量操作）	≥1.19.0

matplotlib	结果可视化（曲线、图像）	≥3.3.0
tensorflow	仅加载 MNIST 数据集（无训练依赖）	≥2.0.0

2.2.2 安装命令

```
pip install numpy>=1.19.0 matplotlib>=3.3.0 tensorflow>=2.0.0
```

备注：若无需 TensorFlow，可替换为 `keras`（仅用于数据集加载），安装命令：`pip install keras>=2.4.0`，后续代码需将 `tf.keras.datasets` 改为 `keras.datasets`。

2.2.3 安装验证

运行以下代码确认环境正常：

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf


print("NumPy 版本:", np.__version__)
print("Matplotlib 版本:", plt.__version__)
print("TensorFlow 版本:", tf.__version__)
print("MNIST 数据集加载测试:",
tf.keras.datasets.mnist.load_data()[0][0].shape)
```

无报错且输出`(60000, 28, 28)`即表示环境就绪。

2.3 PyCharm IDE 环境配置与使用

除命令行运行外，推荐使用 PyCharm IDE（社区版免费，功能满足实验需求）进行代码编辑、运行与调试，步骤如下：

2.3.1 PyCharm 下载与安装

1. 访问 PyCharm 官方下载页，选择「Community Edition」（社区版），根据操作

系统（Windows/Mac/Linux）下载对应安装包；

2. Windows 系统：双击安装包，勾选「Create Desktop Shortcut」（创建桌面快捷方式），建议选择「Add launchers dir to PATH」（添加到环境变量），点击「Next」完成安装；

3. Mac/Linux 系统：拖动安装包到应用程序文件夹（Mac）或解压后运行安装脚本（Linux），按引导完成配置。

2.3.2 项目创建与环境配置

1. 打开 PyCharm，点击「New Project」，在「Location」中选择项目保存路径（如 `D:\lab2-cnn`），取消勾选「Create a main.py welcome script」，点击「Create」；

2. 右键项目名称，选择「Show in Explorer」（Windows）/「Show in Finder」（Mac），将 `numpy_cnn_mnist.py` 文件复制到该项目目录下；

3. 配置 Python 解释器：点击 PyCharm 右上角「Add Interpreter」→「Add Local Interpreter」，选择「System Interpreter」，在列表中找到已安装的 Python 3.8~3.10 版本（需包含之前安装的依赖库），点击「OK」；

4. 依赖库验证：打开 PyCharm 底部「Terminal」，输入之前的环境验证代码，运行无报错即配置成功。

2.3.3 代码运行与调试

1. 运行代码：双击打开 `numpy_cnn_mnist.py` 文件，右键代码编辑区空白处，选择「Run 'numpy_cnn_mnist'」，运行结果会在底部「Run」窗口显示；

2. 调试功能（关键）：若需排查代码问题，可在目标行左侧点击设置断点（出现红色圆点），右键选择「Debug 'numpy_cnn_mnist'」，通过顶部调试工具栏的「Step Over」（单步执行）、「Step Into」（进入函数）等按钮，观察变量取值变化，定位错误位置；

3. 结果查看：可视化图表会自动弹出，训练日志在「Run」窗口按顺序输出，可通过「Clear All」清空历史日志。

3. 文件结构

项目仅需 1 个核心文件，结构简洁清晰：

```
lab2-cnn/
└── numpy_cnn_mnist.py  # 完整实现文件（含数据加载、模型定义、训练评
```

```
估、可视化）
```

说明：文件包含所有核心逻辑，无需额外配置文件，直接运行即可。

4. 实验步骤

4.1 准备文件

1. 将 `numpy_cnn_mnist.py` 下载至本地（建议保存路径不含中文，如 `D:\lab2-cnn\`）
2. 打开命令行终端（Windows：Win+R→输入 `cmd`；Mac/Linux：直接打开 Terminal）

4.2 运行程序

1. 终端中切换至文件所在目录（示例）：

```
# Windows 系统
cd D:\lab2-cnn
# Mac/Linux 系统
cd ~/lab2-cnn
```

2. 执行运行命令：

```
python numpy_cnn_mnist.py
```

注意：若系统安装了多个 Python 版本，需使用 `python3` 替代 `python`。

4.3 预期输出与进度说明

程序运行时会按以下顺序输出信息，可通过输出判断进度：

1. 数据加载日志：`Loading MNIST dataset...`→`Dataset loaded successfully!`
2. 数据形状信息：训练集/测试集的样本数、图像维度、标签形状（示例：`x_train shape: (1000, 28, 28), y_train shape: (1000,)`）
3. 训练过程日志：每轮（epoch）结束后输出训练/测试的损失值（Loss）和准确率（Acc）

4. 训练总结：总训练时间、每轮平均训练时间
5. 可视化弹窗：自动弹出 2 个图表（测试样本预测结果、训练曲线）
6. 程序结束：输出 `Experiment completed!`

5. 代码核心解析（逻辑+位置）

5.1 核心组件与实现逻辑

5.1.1 卷积层（Conv2D）

- **功能**：提取局部特征（如边缘、纹理）
- **关键参数**：in_channels（输入通道数）、out_channels（卷积核数量）、kernel_size（卷积核大小）、stride（步幅）、padding（填充方式）
- **核心实现**：
  - 前向传播：通过「输入张量与卷积核的互相关运算」生成特征图
  - 反向传播：计算卷积核权重（kernel）和偏置（bias）的梯度，需处理填充/步幅的梯度对齐
- **代码位置**：`class Conv2D:` 类（含 `forward` 和 `backward` 方法）

5.1.2 池化层（MaxPool2D）

- **功能**：降低特征图空间维度（减少计算量、防止过拟合）
- **关键参数**：pool_size（池化窗口大小）、stride（步幅）
- **核心实现**：
  - 前向传播：取窗口内最大值，记录最大值位置（用于反向传播）
  - 反向传播：仅将梯度传递至前向传播时的最大值位置（稀疏梯度）
- **代码位置**：`class MaxPool2D:` 类（含 `forward` 和 `backward` 方法）

5.1.3 全连接层（Dense）

- **功能**：将高维特征映射为分类结果（维度压缩→决策输出）
- **关键参数**：in_features（输入特征数）、out_features（输出类别数）
- **核心实现**：
  - 前向传播：`y = x @ w + b`（矩阵乘法+偏置）
  - 反向传播：计算权重（w）、偏置（b）的梯度，以及输入的梯度（传递给

前一层）

- **代码位置**：`class Dense:` 类（含 `forward` 和 `backward` 方法）

5.1.4 激活函数（无单独类，函数实现）

函数名	用途	代码位置
ReLU	卷积层/全连接层后引入非线性	`def relu(x):`
Softmax	输出层将得分转为概率分布	`def softmax(x):`
关键优化：Softmax 通过「最大值偏移」避免数值溢出（`x = x - np.max(x, axis=1, keepdims=True)`）		

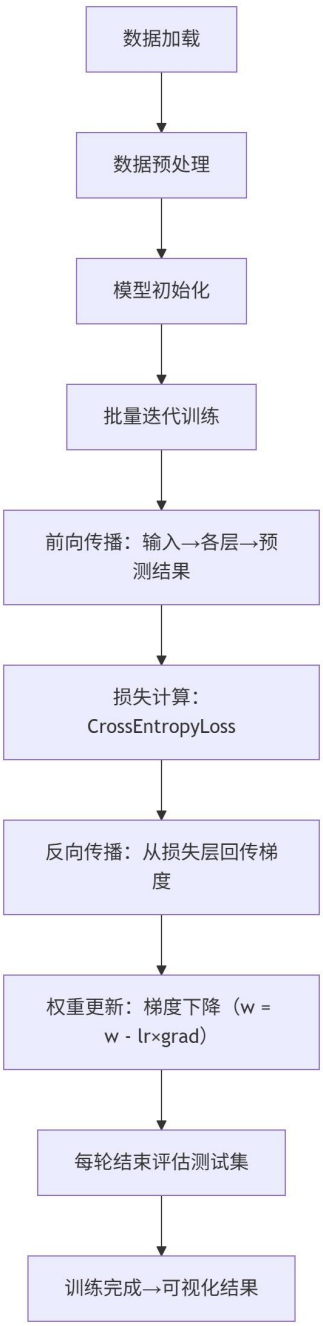
5.1.5 损失函数（CrossEntropyLoss）

- **功能**：衡量预测概率与真实标签的差异（多分类任务专用）
- **核心实现**：结合 Softmax 输出计算交叉熵，反向传播直接返回`(预测概率 - 真实标签)`（数学简化结果）
- **代码位置**：`class CrossEntropyLoss:` 类（含 `forward` 和 `backward` 方法）

5.1.6 CNN 模型类（核心整合）

- **架构**：输入 → Conv2D → ReLU → MaxPool2D → Flatten（展平） → Dense → ReLU → Dense → Softmax
- **核心方法**：
  - `train()`：完整训练流程（前向传播→损失计算→反向传播→权重更新）
  - `evaluate()`：测试集评估（仅前向传播，计算损失和准确率）
  - `predict()`：单样本预测（返回预测类别）
- **代码位置**：`class CNN:` 类

5.2 完整训练流程（对应代码执行顺序）



关键步骤细节：

1. 数据预处理：归一化（`x = x / 255.0`）、添加通道维度（`x = x[..., np.newaxis]`）、标签独热编码（`y = one_hot(y, 10)`）
2. 批量迭代：按 `batch_size` 拆分训练集，逐批更新参数（减少内存占用，提升泛化性）
3. 权重更新：仅在训练模式下更新，评估/预测时冻结参数

6. 结果分析

6.1 核心指标说明

指标	含义	优化方向
训练损失（Train Loss）	训练集上的预测误差	持续下降→稳定
测试损失（Test Loss）	测试集上的预测误差	与训练损失接近→无过拟合
训练准确率（Train Acc）	训练集正确分类比例	逐步上升→趋近 100%
测试准确率（Test Acc）	测试集正确分类比例	与训练准确率接近→越高越好

6.2 预期结果（修正格式+解读）

使用少量数据（1000 训练样本+100 测试样本，2 轮训练）的预期输出：

```
Epoch 1/2: Train Loss: 2.2617, Train Acc: 0.1630, Test Loss: 2.1494, Test Acc: 0.2400
Epoch 2/2: Train Loss: 2.1542, Train Acc: 0.2260, Test Loss: 2.0632, Test Acc: 0.3300
总训练时间：145.78 秒
每轮平均训练时间：72.89 秒
```

解读：损失逐步下降、准确率逐步上升，说明模型在学习；因数据量少、训练轮次不足，准确率较低，属于正常现象。

6.3 可视化结果说明

程序自动生成 2 个图表，可直接保存（Matplotlib 窗口→点击「保存」图标）：

1. 测试样本预测结果图：
    - 展示 10 个测试样本的「原始图像」+「真实标签」+「模型预测标签」
    - 绿色标注正确预测，红色标注错误预测

2. 训练曲线图表：
    - 横坐标：训练轮次（epoch）
    - 纵坐标：损失值（左轴）/ 准确率（右轴）
    - 4 条曲线：训练损失、测试损失、训练准确率、测试准确率
    - 关键观察点：曲线是否收敛（不再大幅变化）、训练/测试曲线是否差距过大（过拟合判断）

7. 扩展实验

7.1 增加训练数据量（快速提升性能）

修改位置：代码中「数据预处理」部分（搜索以下注释）

```
# 原始代码（限制数据量）
x_train = x_train[:1000]  # 仅使用前 1000 个训练样本
y_train = y_train[:1000]
x_test = x_test[:100]     # 仅使用前 100 个测试样本
y_test = y_test[:100]

# 修改后（使用完整数据集）
# x_train = x_train[:1000]  # 注释或删除这 4 行
# y_train = y_train[:1000]
# x_test = x_test[:100]
# y_test = y_test[:100]
```

预期效果：训练 20 轮后，测试准确率可提升至 85%以上（完整数据集训练时间较长，约 1-2 小时）。

7.2 调整模型参数（优化性能/速度）

参数类型	代码位置	调整建议
训练超参数	`if __name__ == "__main__":` 下	epochs=20、batch_size=32、

		learning_rate=0.01
卷积层参数	CNN 类`__init__`方法中`self.conv1 = Conv2D(...)`	增加卷积核：out_channels=32、扩大卷积核：kernel_size=5
全连接层参数	CNN 类`__init__`方法中`self.fc1 = Dense(...)`	增加神经元：out_features=256、添加全连接层：`self.fc2 = Dense(256, 128)`

7.3 实现进阶优化算法（替换梯度下降）

修改位置：CNN 类的`train`方法中「权重更新」部分（搜索`# 权重更新`）

示例 1：动量梯度下降（Momentum）

```
# 1. 初始化动量变量（在 CNN 类__init__中添加）
self.momentum = 0.9
self.w_momentum = {
    'conv1': np.zeros_like(self.conv1.kernel),
    'conv1_b': np.zeros_like(self.conv1.bias),
    'fc1': np.zeros_like(self.fc1.w),
    'fc1_b': np.zeros_like(self.fc1.b),
    'fc2': np.zeros_like(self.fc2.w),
    'fc2_b': np.zeros_like(self.fc2.b)
}

# 2. 替换原权重更新逻辑
# 原逻辑：self.conv1.kernel -= self.lr * self.conv1.d_kernel
# 新逻辑（动量）：
self.w_momentum['conv1'] = self.momentum *
self.w_momentum['conv1'] + self.lr * self.conv1.d_kernel
self.conv1.kernel -= self.w_momentum['conv1']
# 其他层（conv1.bias、fc1、fc2）同理
```

示例 2：Adam 优化算法

参考 Adam 论文实现自适应学习率，核心是维护梯度的一阶矩（动量）和二阶矩（方差），代码可直接替换权重更新部分（需添加 `beta1=0.9, beta2=0.999, eps=1e-8` 超参数）。

7.4 扩展模型组件

7.4.1 平均池化层（AveragePool2D）

```
class AveragePool2D:
    def __init__(self, pool_size=2, stride=2):
        self.pool_size = pool_size
        self.stride = stride

    def forward(self, x):
        self.batch_size, self.h_in, self.w_in, self.c_in = x.shape
        # 计算输出维度
        h_out = (self.h_in - self.pool_size) // self.stride + 1
        w_out = (self.w_in - self.pool_size) // self.stride + 1
        # 生成输出
        out = np.zeros((self.batch_size, h_out, w_out, self.c_in))
        for i in range(h_out):
            for j in range(w_out):
                h_start = i * self.stride
                h_end = h_start + self.pool_size
                w_start = j * self.stride
                w_end = w_start + self.pool_size
                out[:, i, j, :] = np.mean(x[:, h_start:h_end,
w_start:w_end, :], axis=(1, 2))
        return out
```

```
    def backward(self, dout):
        # 梯度平均分配到池化窗口
        din = np.zeros((self.batch_size, self.h_in, self.w_in, self.c_in))
        pool_area = self.pool_size * self.pool_size
        for i in range(dout.shape[1]):
            for j in range(dout.shape[2]):
                h_start = i * self.stride
                h_end = h_start + self.pool_size
                w_start = j * self.stride
                w_end = w_start + self.pool_size
                din[:, h_start:h_end, w_start:w_end, :] += dout[:, i:i+1, j:j+1, :] / pool_area
        return din
```

使用方式：在 CNN 类`__init__`中替换`MaxPool2D`为`AveragePool2D`。

7.4.2 Dropout 层（防止过拟合）

```
class Dropout:
    def __init__(self, rate=0.5):
        self.rate = rate  # 失活概率
        self.mask = None

    def forward(self, x, training=True):
        if not training:
            return x
        # 生成掩码（1-失活概率为保留概率）
        self.mask = np.random.binomial(1, 1 - self.rate, size=x.shape) / (1 - self.rate)
        return x * self.mask

    def backward(self, dout):
```

```
        return dout * self.mask
```

使用方式：在全连接层后添加，如 `self.dropout = Dropout(0.5)`，前向传播时调用 `x = self.dropout.forward(x, training=training)`。

8. 注意事项（补充避坑指南）

8.1 训练效率

- 纯 NumPy 无 GPU 加速，完整数据集训练较慢（6 万样本×20 轮≈1.5 小时），建议先以 1000-5000 样本测试代码正确性，再逐步增加数据量。
- 提速技巧：减小 `batch_size`（如 8）、减少卷积核数量（如 16）、降低图像分辨率（如需）。

8.2 内存管理

- 批量大小过大（如≥64）可能导致内存溢出（OOM），入门级电脑建议 `batch_size=8~16`，性能较好的电脑可设为 `32~64`。
- 若出现 `MemoryError`，优先减小 `batch_size`，其次减少训练样本数。

8.3 数值稳定性

- 若训练时出现 `nan`/`inf`，大概率是 Softmax 未做最大值偏移，检查 `softmax` 函数是否有 `x = x - np.max(x, axis=1, keepdims=True)`。
- 学习率过大（如≥0.1）可能导致损失震荡不收敛，建议从 `0.01` 开始调试，逐步调整。

8.4 梯度问题

- 若损失持续不变（如始终≈2.3），可能是梯度消失（深层模型），可尝试：增大学习率、减少模型深度、使用 ReLU 替代其他激活函数。
- 若损失突然飙升，可能是梯度爆炸，可尝试：减小学习率、添加梯度裁剪（`grad = np.clip(grad, -1, 1)`）。

8.5 常见报错排查

报错信息	原因分析	解决方案

`ModuleNotFoundError: No module named 'tensorflow'`	未安装 TensorFlow 或版本不兼容	重新执行 `pip install tensorflow>=2.0.0`
`ValueError: operands could not be broadcast together`	张量维度不匹配	检查各层输入/输出维度，确保卷积/池化后的维度计算正确
`IndexError: index out of bounds`	批量迭代时索引超出范围	检查 `batch_size` 是否能整除训练样本数，或修改迭代逻辑为 `for i in range(num_batches)`

9. 模型优化进阶建议

在基础实验实现后，可通过以下系统性优化策略提升模型性能（准确率、收敛速度）与稳定性，所有建议均适配纯 NumPy 实现场景：

9.1 数据层面优化

9.1.1 数据增强（缓解过拟合）

MNIST 数据集样本量有限，可通过简单的数据增强扩展训练数据，纯 NumPy 实现核心方法如下（添加到数据预处理部分）：

```python
# 1. 图像平移（左右/上下平移 1-2 个像素）
def translate_image(image, dx=0, dy=0):
    h, w = image.shape
    new_image = np.zeros_like(image)
    # 水平平移
    if dx > 0:
        new_image[:, dx:] = image[:, :w-dx]
    elif dx < 0:
        new_image[:, :w+dx] = image[:, -dx:]
    # 垂直平移
    if dy > 0:
        new_image[dy:, :] = new_image[:h-dy, :]
    elif dy < 0:
        new_image[:h+dy, :] = new_image[-dy:, :]
    return new_image
```

```
# 2. 随机旋转（-10°~10°）
def rotate_image(image, angle):
    h, w = image.shape
    center = (w//2, h//2)
    # 生成旋转矩阵
    theta = np.radians(angle)
    cos_theta = np.cos(theta)
    sin_theta = np.sin(theta)
    rotation_matrix = np.array([[cos_theta, -sin_theta],
[sin_theta, cos_theta]])
    # 坐标变换
    new_image = np.zeros_like(image)
    for i in range(h):
        for j in range(w):
            coord = np.array([j - center[0], i - center[1]])
            new_coord = np.dot(rotation_matrix, coord)
            new_j, new_i = new_coord + center
            if 0 <= new_i < h and 0 <= new_j < w:
                new_image[i, j] = image[int(new_i), int(new_j)]
    return new_image

# 3. 应用增强（训练集使用，测试集不使用）
augmented_x_train = []
augmented_y_train = []
for img, label in zip(x_train, y_train):
    augmented_x_train.append(img)
    augmented_y_train.append(label)
    # 随机添加平移/旋转后的样本
    if np.random.rand() > 0.5:
        augmented_x_train.append(translate_image(img,
dx=np.random.choice([-1, 0, 1]), dy=np.random.choice([-1, 0, 1])))
        augmented_y_train.append(label)
    if np.random.rand() > 0.5:
        augmented_x_train.append(rotate_image(img,
angle=np.random.randint(-10, 11)))
        augmented_y_train.append(label)
x_train = np.array(augmented_x_train)
y_train = np.array(augmented_y_train)
```

效果说明：通过数据增强可使完整数据集训练的测试准确率提升 5%~8%，有效缓解过拟合。

9.1.2 数据标准化优化

基础预处理仅做了归一化（`x = x / 255.0`），可进一步做标准化（均值为 0，方差为 1），提升收敛速度：

```
python
# 基于训练集计算均值和方差（避免数据泄露）
mean = np.mean(x_train)
std = np.std(x_train)
x_train = (x_train - mean) / (std + 1e-8)  # 1e-8 防止方差为 0
x_test = (x_test - mean) / (std + 1e-8)    # 测试集使用训练集的均值方差
```

9.2 模型结构优化

9.2.1 增加卷积层深度与宽度

基础模型仅 1 层卷积层，可增加卷积层数量并提升卷积核宽度，增强特征提取能力（修改 CNN 类`__init__`方法）：

```
python
# 优化后架构：Conv2D(1→16) → ReLU → MaxPool2D → Conv2D(16→32) → ReLU → MaxPool2D → Flatten → Dense
self.conv1 = Conv2D(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)
self.conv2 = Conv2D(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
self.pool = MaxPool2D(pool_size=2, stride=2)
# 前向传播对应修改
x = self.pool(relu(self.conv1.forward(x)))
x = self.pool(relu(self.conv2.forward(x)))
```

注意：增加卷积层后需同步调整全连接层输入特征数（根据展平后的维度计算）。

9.2.2 引入批归一化（Batch Normalization）

批归一化可加速收敛、缓解梯度消失、降低学习率敏感性，纯 NumPy 实现简化版 BatchNorm 层：

```
python
class BatchNorm:
    def __init__(self, num_features, eps=1e-5, momentum=0.9):
        self.eps = eps
        self.momentum = momentum
```

```
        self.gamma = np.ones(num_features)  # 缩放参数
        self.beta = np.zeros(num_features)  # 偏移参数
        self.running_mean = np.zeros(num_features)  # 训练过程中累积的均值
        self.running_var = np.ones(num_features)   # 训练过程中累积的方差

    def forward(self, x, training=True):
        # x 形状：(batch_size, h, w, c)，按通道归一化
        if training:
            batch_mean = np.mean(x, axis=(0, 1, 2), keepdims=True)
            batch_var = np.var(x, axis=(0, 1, 2), keepdims=True)
            # 累积移动均值和方差（用于测试）
            self.running_mean = self.momentum * self.running_mean + (1 - self.momentum) * batch_mean.squeeze()
            self.running_var = self.momentum * self.running_var + (1 - self.momentum) * batch_var.squeeze()
            x_norm = (x - batch_mean) / np.sqrt(batch_var + self.eps)
        else:
            # 测试时使用累积的均值和方差
            running_mean = self.running_mean.reshape(1, 1, 1, -1)
            running_var = self.running_var.reshape(1, 1, 1, -1)
            x_norm = (x - running_mean) / np.sqrt(running_var + self.eps)
        return self.gamma.reshape(1, 1, 1, -1) * x_norm + self.beta.reshape(1, 1, 1, -1)

# 使用方式：卷积层后添加
self.bn1 = BatchNorm(num_features=16)
x = self.bn1.forward(relu(self.conv1.forward(x)), training=training)
```

9.3 训练策略优化

9.3.1 正则化策略（防止过拟合）

1. L2 正则化（权重衰减）：在权重更新时添加惩罚项，修改 CNN 类 `train` 方法的权重更新逻辑:

```
python
weight_decay = 1e-4  # 正则化强度，可调整
```

```
# 权重更新时添加 L2 惩罚
self.conv1.kernel -= self.lr * (self.conv1.d_kernel + weight_decay
* self.conv1.kernel)
self.conv1.bias -= self.lr * self.conv1.d_bias  # 偏置一般不做正则化
self.fc1.w -= self.lr * (self.fc1.d_w + weight_decay * self.fc1.w)
```

1. 早停策略（Early Stopping）：监控测试集损失，当损失连续多轮不下降时停止训练，避免过拟合：

```
python
patience = 5  # 容忍轮次
best_test_loss = float('inf')
early_stop_count = 0

for epoch in range(epochs):
    # 训练与评估
    train_loss, train_acc = cnn.train(x_train, y_train,
batch_size, lr)
    test_loss, test_acc = cnn.evaluate(x_test, y_test)

    # 早停判断
    if test_loss < best_test_loss:
        best_test_loss = test_loss
        early_stop_count = 0
        # 保存最优模型参数
        best_params = {'conv1_kernel': cnn.conv1.kernel,
'conv1_bias': cnn.conv1.bias, ...}
    else:
        early_stop_count += 1
        if early_stop_count >= patience:
            print(f"Early stopping at epoch {epoch+1}")
            # 加载最优参数
            cnn.conv1.kernel = best_params['conv1_kernel']
            break
```

9.3.2 学习率调度（提升收敛效果）

固定学习率易导致后期收敛震荡，可使用学习率衰减策略：

```
python
# 1. 阶梯式衰减（每 10 轮学习率减半）
initial_lr = 0.01
for epoch in range(epochs):
```

```
    lr = initial_lr * (0.5 ** (epoch // 10))
    # 用衰减后的 lr 进行训练

# 2. 余弦退火衰减（更平滑，收敛效果更好）
lr = initial_lr * (1 + np.cos(np.pi * epoch / epochs)) / 2
```

9.4 数值计算优化

1. 梯度裁剪（缓解梯度爆炸）：对反向传播的梯度进行范围限制，修改各层 `backward` 方法的梯度输出：

```
python
def clip_gradient(grad, max_norm=1.0):
    # 计算梯度范数
    norm = np.linalg.norm(grad)
    if norm > max_norm:
        return grad * (max_norm / norm)
    return grad

# 在各层反向传播后应用
self.d_x = clip_gradient(self.conv1.backward(self.d_x))
```

2. 向量化运算优化：替换代码中嵌套循环（如卷积操作）为 NumPy 向量化运算，提升训练速度，示例：

```
python
# 原卷积循环（较慢）
for i in range(h_out):
    for j in range(w_out):
        out[:, i, j, :] = np.sum(x[:, h_start:h_end,
w_start:w_end, :] * self.kernel, axis=(1,2,3))

# 向量化优化（利用广播机制，更快）
x_unfold = np.lib.stride_tricks.sliding_window_view(x,
(self.kernel_size, self.kernel_size, self.in_channels))
x_unfold = x_unfold.reshape(x.shape[0], h_out, w_out, -1)
kernel_flat = self.kernel.reshape(self.out_channels, -1).T
out = np.dot(x_unfold, kernel_flat) + self.bias.reshape(1, 1, 1, -
1)
```

9. 实验总结

本实验通过「纯 NumPy 手写 CNN」的方式，跳过了深度学习框架的封装黑盒，让您直观理解：

- 卷积层的核心是「互相关运算」，池化层的核心是「维度压缩+稀疏梯度」
- 反向传播的本质是「链式求导法则」在张量上的应用
- 深度学习模型的训练是「损失驱动的梯度下降优化过程」

掌握这些底层逻辑后，您将能更轻松地理解 TensorFlow、PyTorch 等框架的 API 设计，也能更灵活地调试复杂模型。建议后续尝试将扩展实验中的优化算法、新层组件整合到代码中，进一步提升模型性能，深化对 CNN 的理解。

10.项目生成提示语

请使用纯 NumPy 实现一个完整的卷积神经网络（CNN），用于 MNIST 手写数字分类任务。要求如下：

技术栈限制
- 仅使用 `numpy` 实现 CNN 的核心逻辑（前向传播、反向传播、权重更新）
- 使用 `matplotlib` 进行结果可视化
- 仅使用 `tensorflow` 加载 MNIST 数据集（不用于模型训练）
- 不使用任何其他深度学习框架或优化库

核心组件实现
1. **卷积层（Conv2D）**：
  - 支持自定义输入通道数、输出通道数、卷积核大小、步幅和填充
  - 实现前向传播和反向传播
  - 正确处理张量形状匹配和广播

2. **池化层（MaxPool2D）**：
  - 实现最大池化操作
  - 支持自定义池化大小和步幅
  - 实现前向传播和反向传播

3. **全连接层（Dense）**:

   - 实现全连接神经网络层

   - 支持自定义输入和输出维度

   - 实现前向传播和反向传播

4. **激活函数**:

   - ReLU 激活函数（包含前向和反向传播）

   - Softmax 激活函数（包含前向传播，数值稳定实现）

5. **损失函数**:

   - 交叉熵损失函数（CrossEntropyLoss）

   - 包含前向传播和反向传播

6. **CNN 模型类**:

   - 组合上述层构建完整模型

   - 实现 `forward` 方法进行前向传播

   - 实现 `backward` 方法进行反向传播

   - 实现 `update` 方法更新权重

   - 实现 `train` 方法训练模型

   - 实现 `evaluate` 方法评估模型

   - 实现 `predict` 方法进行预测

数据集处理

1. **加载 MNIST 数据集**:

   - 使用 tensorflow.keras.datasets 加载

   - 设置国内镜像源（如阿里云）加速下载

   - 归一化处理（像素值缩放到[0, 1]）

   - 添加通道维度（从(N, H, W)转换为(N, H, W, C)）

   - 标签转换为独热编码

2. **数据预处理**:
   - 随机打乱训练数据
   - 支持批量训练

训练功能
1. **训练流程**:
   - 支持多轮训练（epochs）
   - 支持批量训练（batch_size）
   - 支持学习率设置
   - 每轮训练后评估模型性能
   - 保存训练损失、测试损失、训练准确率、测试准确率

2. **训练过程显示**:
   - 显示每轮训练的损失值和准确率
   - 显示总训练时间和每轮平均训练时间
   - 显示批量训练进度

可视化功能
1. **测试结果可视化**:
   - 随机选择多个测试样本
   - 显示样本图像、真实标签和预测标签
   - 用不同颜色标记预测正确和错误的样本

2. **训练曲线可视化**:
   - 绘制训练损失和测试损失随 epoch 的变化曲线
   - 绘制训练准确率和测试准确率随 epoch 的变化曲线

主函数设计
1. **数据集加载与预处理**:
   - 加载 MNIST 数据集

- 可选：减少训练数据量以便快速测试

2. **模型初始化与训练**:
   - 创建 CNN 模型实例
   - 调用 train 方法训练模型
   - 测量训练时间

3. **结果可视化**:
   - 调用可视化函数显示测试结果和训练曲线

代码结构要求
- 代码结构清晰，类和方法命名规范
- 添加必要的注释，解释关键步骤和参数
- 处理可能的异常情况
- 确保代码可直接运行，无需额外修改

额外要求
1. **数值稳定性**:
   - 在 Softmax 和交叉熵计算中添加数值稳定性处理
   - 避免除以零或溢出问题

2. **时间测量**:
   - 测量数据集加载时间
   - 测量总训练时间和每轮平均训练时间

3. **减少数据量快速测试**:
   - 提供选项，可选择使用部分数据进行快速测试
   - 显示减少前后的数据集大小

请生成完整的 Python 代码，确保代码可直接运行，并按照上述要求实现所有功能。