南開大學
Nankai University

网络空间安全学院

机器学习实验报告

LeNet-5 手写数字识别

姓名：武桐西

学号：2112515

专业：信息安全

2024 年 1 月 15 日

目录

1 实验目的

本次实验要求使用 Python 实现 LeNet-5 神经网络，在 MNIST 手写数字识别数据集[1]上进行训练与测试，完成 0 – 9 十个手写数字体的识别（分类）任务。

实验要求：

1. 代码使用 Python 实现；
2. 手动实现网络前向传播与梯度反向传播，不能使用自动微分框架，如 PyTorch、TensorFlow、Caffe 等深度学习框架；
3. 数据读取可以使用 PIL、OpenCV-python 等库；
4. 矩阵运算可以使用 NumPy 等库。

2 实验环境

本实验在 Windows 系统下，使用`Python 3.9.7`解释器。所使用的相关的包主要有：

```
numpy                   1.20.3  # 矩阵计算
scikit-learn            0.24.2  # 混淆矩阵
matplotlib              3.4.3   # 绘图
```

3 实验原理

3.1 卷积（Convolution）

相比于传统的机器学习方法，以卷积运算作为机器学习中的特征提取器能够保持图像的位置信息，而传统方法会丢失图像的位置信息（一般会将所有特征“拉成”一个向量），同时由于卷积运算是线性的，基本运算只有乘法和加法，因此其计算简便，性能较好（特别是采用 GPU），因此以其为基本算子的卷积神经网络（CNN）在计算机视觉中表现突出，应用十分广泛。

与数学上的卷积定义略有区别，卷积神经网络中的基本的卷积运算如图3.1所示。

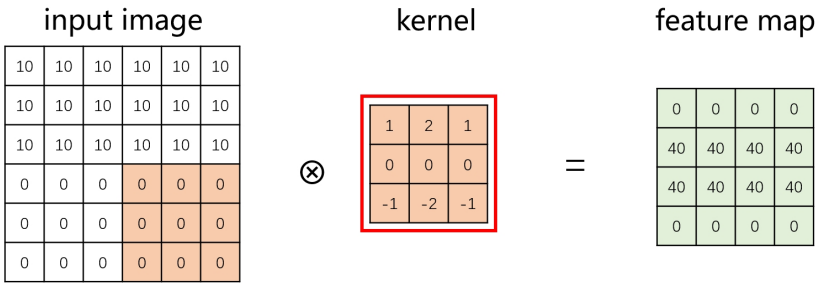

图 3.1: 卷积算子

实际使用中，一般会有多个卷积核（过滤器），进行多通道多核卷积，如图3.2所示。

[1]MNIST 手写数字识别数据集

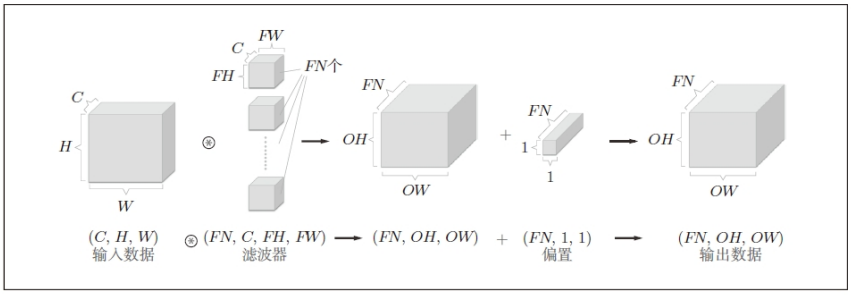

图 3.2: 卷积运算的处理流（带有偏置项）

卷积的特征提取效果实例如图3.3所示[2]。

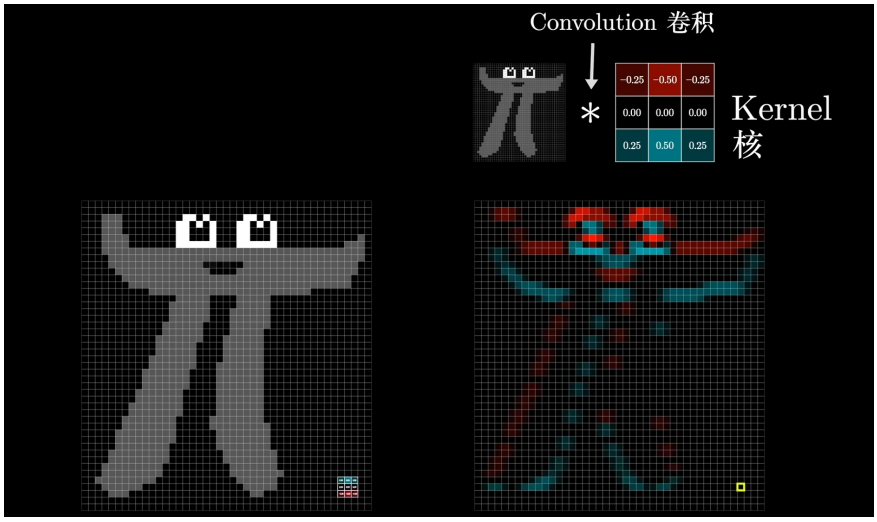

图 3.3: 卷积核作水平边缘检测器

除了标准的卷积运算外，还有反卷积（转置卷积）、空洞卷积、分组卷积、深度可分离卷积等，可参考此处，这里不再赘述。

卷积运算的参数一般包括：

1. 卷积核的大小`kernel_size`：$filter\_H \times filter\_W$；
2. `stride`：步长参数；
3. `padding`：补零参数；
4. `bias`：是否包含偏置参数；
5. 卷积通道数。

3.2 基于梯度的误差反向传播算法

基于梯度的误差反向传播算法是通过计算损失函数对网络参数的梯度，然后利用梯度信息更新参数，从而最小化网络的预测误差。整个过程可以分为前向传播和反向传播两个阶段：

1. **前向传播（Forward Propagation）：**

[2] BiliBili 3Blue1Brown 什么是卷积?

- 输入样本通过神经网络进行前向计算，得到模型的预测输出。
- 计算预测输出与实际标签之间的损失函数值。

2. **反向传播（Backward Propagation）：**

- 计算损失函数对网络参数的梯度。
- 利用梯度信息和优化算法（如梯度下降）来更新网络参数，减小损失函数值。
- 重复上述步骤，直至达到训练停止的条件。

通过反向传播，网络能够学习调整权重和偏置，以最小化损失函数，从而提高对输入数据的预测性能。

3.3 LeNet-5

LeNet-5[3] 是由 Yann LeCun 在 1998 年提出的深度卷积神经网络架构，主要应用于手写数字识别。该网络结构包含了卷积层、池化层和全连接层，是深度学习领域的经典模型之一。其网络架构如图所示。

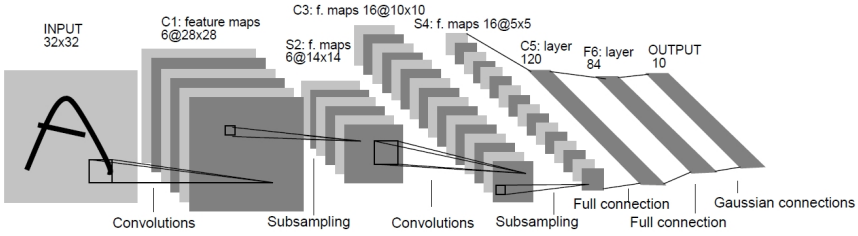

图 3.4: LeNet-5 模型

4 LeNet-5 的实现

使用 NumPy 实现 LeNet-5。

4.1 代码结构

```
LeNet-5
├── MNIST // MNIST 数据集
│   ├── train-images-idx3-ubyte // 训练集图像
│   ├── train-labels-idx1-ubyte // 训练集标签
│   ├── t10k-images-idx3-ubyte // 测试集图像
│   └── t10k-labels-idx1-ubyte // 测试集标签
├── mnist.py // 数据集加载
├── functions.py // 一些函数
├── utils.py // 辅助函数（用于优化等）
├── layers.py // LeNet-5 的层的封装
├── optim.py // 优化器
└── model.py // LeNet-5 模型架构
```

```
├── trainer.py // 训练器（Trainer）封装
├── visualization.py // 结果可视化
└── main.py // 主程序
```

4.2 LeNet-5 模型架构

本实验中实现的 LeNet-5 模型与 Yann Lecun 论文 [3] 中的模型架构基本一致，但略有区别。

4.2.1 模型架构

本实验中的 LeNet-5 模型由 7 层构成（不计算输出层 SoftmaxWithLoss），主要有 2 个卷积层，2 个池化层，3 个全连接层组成，具体如下：

1. `conv1`：$6 \times 5 \times 5$ 的卷积层[3]；后接一个 *ReLU* 激活函数层`relu1`。
2. `pool1`：最大池化层（MaxPooling），`kernel_size=2, stride=2`。
3. `conv2`：$16 \times 5 \times 5$ 的卷积层；后接一个 *ReLU* 激活函数层`relu2`。
4. `pool2`：最大池化层（MaxPooling），`kernel_size=2, stride=2`。
5. `fc1`：全连接层（Fully Connected Layer），`in_features=16*4*4, out_features=120`；后接一个 *ReLU* 激活函数层`relu3`。
6. `fc2`：全连接层（Fully Connected Layer），`in_features=120, out_features=84`；后接一个 *ReLU* 激活函数层`relu4`。
7. `fc3`：全连接层（Fully Connected Layer），`in_features=84, out_features=10`（类别数`num_classes`）。
8. **输出层** SoftmaxWithLoss：输出层 SoftmaxWithLoss 仅用于模型的训练阶段，推理阶段实则不需要；该层采用 *Softmax* 函数将`fc3`的输出转换为概率形式（之和为 1），然后连接损失层，在这里损失采用交叉熵损失（Cross Entropy Loss），实际上是因为这两个层连用在误差反向传播时的梯度有比较简单的表达形式[4]（后面会详细说明）。

上面的表述中，未注明的情况下，卷积层的参数默认`stride=1, padding=0, bias=True`，池化层的参数默认`padding=0`，全连接层参数默认`bias=True`。

4.2.2 重点说明

与原论文 [3] 的区别：

1. **输入图像大小不同：**本实验中，输入图像的大小为 $1 \times 28 \times 28$，而原论文中的输入图像大小为 $1 \times 32 \times 32$，因此在`fc1`中的输入参数`in_features`不同，本实验中为 $16 \times 4 \times 4$，原论文为 $16 \times 5 \times 5$。
2. **池化层不同：**原论文中的池化层采用的类似于均值池化，但实际上并不是均值池化，原论文中的做法是将输入特征中与池化核对应的部分相加，并乘一个可学习的系数，再加一个可学习的偏置，因此原论文中的池化层也是有参数需要学习的；而本实验中，池化层采用最大池化（MaxPooling），取对应感受野中的最大值，因此并无可学习的参数。

[3] 大小含义为 $C \times H \times W$，其中 C 代表通道数，H 代表高度，W 代表宽度，以下同

[4] 实际上是人们为了有一个简单的梯度表达而构造的

3. **激活函数不同：**原论文中采用的激活函数层是 *Sigmoid* 函数；而本实验中采用的是现在深度学习中更为常用的 *ReLU* 函数（修正线性单元）。

4.3 一些优化与加速

众所周知，神经网络模型的训练一般需要消耗比较大的算力[5]，训练时间一般比较长。本次实验中，采用了一些优化技巧与方法，进行模型的加速。但是，本实验中并没有实现利用 GPU 进行加速。

4.3.1 批处理（mini-batch）

若将数据一个一个的 “喂给” 模型，则模型的吞吐率会变得非常低。因此，可以采用批处理（mini-batch）的方式进行模型的训练，每次输入一批数据（可由`batch_size`参数控制）进行计算，这样不但可以提高吞吐率，而且由于 NumPy 等科学计算库会对大矩阵的乘法进行优化与加速，因此采用批处理进行模型的训练能够优化计算，大大地提高模型的训练速度，极大缩短训练时间。除此之外，采用批处理在计算损失并进行梯度反向传播时，得到的损失和梯度比较稳定，单个数据造成的误差与噪声被减少，因此也能够提高模型的表现性能（Performance），加快模型的收敛速度。

从代码的角度来讲，采用批处理需要将矩阵增加一个维度（一般是第一个位置上的维度，这样做其实是为了方便矩阵的运算），表示 N 个数据作为一批。

4.3.2 利用计算图进行梯度反向传播

利用**计算图**（Computational Graph）计算梯度，而不是使用数值微分方法，一方面能够提高计算效率，节省大量计算量，同时其精度也一般会比后者更高，而且无需考虑数值微分中的计算机浮点数问题。

利用计算机计算梯度，一种比较简单同时也比较适用于计算机的实现是**数值微分**方法。

采用数值微分方法虽然代码比较简单，但是也有需要注意的问题，比如计算机的精度、计算溢出的处理（包括上溢和下溢）等需要特殊对待，这些通常都是由于计算机的浮点数处理而引起的。

除了上述问题之外，虽然数值微分对于一些复杂函数（比如一些非初等函数）的计算非常简单粗暴、简便易行（从代码编写的角度），但是由于在机器学习算法中，采用的函数一般都比较简单（一般是线性的函数，主要运算为乘法和加法，以及一些激活函数等非线性函数），可以认为是一些比较简单的函数的加减乘除运算以及函数复合运算构成的，而这样的函数一般能够通过微分的运算法则以及复合函数求导法则来进行分析上的解析求解。因此，如果能够从数学分析的结论入手，应该能够节约大量的计算量，并且保证求解微分的精度不低于数值微分的结果。

经过上面的分析，我们可以发现，机器学习算法适合于使用数学分析中的微分结论直接求得。每一步的函数梯度基本上可以使用简单的数学分析结论直接写出，而整个梯度反向传播可以在此基础上使用复合函数求导法则进行传递和计算。为了使用复合函数求导法则，引入了**计算图**（Computational Graph，如图4.5所示）的概念，并用其进行梯度的反向传播。

[5]这也是为什么神经网络的提出很早，但是由于当时的算力不足而导致没有火起来。

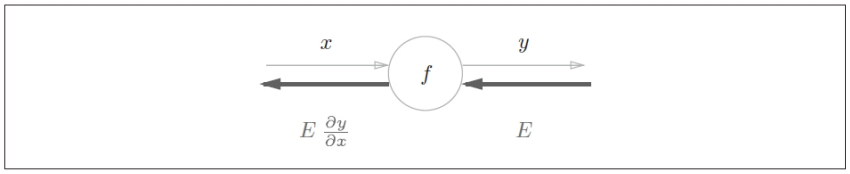

图 4.5: 计算图的反向传播

使用计算图进行梯度反向传播，可以节约大量的计算资源（计算图每一步一般只有简单的乘法、加法等线性运算，而数值微分需要带入原函数计算），同时也能获得更高的精度。因此，采用计算图可以大大提高模型的训练速度，缩短模型训练时间。

4.3.3 卷积算子加速：im2col 优化

直接进行卷积运算，虽然从代码的角度非常容易实现，但是需要用到大量的`for`循环，由于`for`循环在 Python 中运行非常慢（特别是与 NumPy 连用时），同时 LeNet-5 网络本身就是一种经典的卷积神经网络（CNN），包含大量的卷积运算，因此大量的`for`循环的使用无疑会成为模型训练速度的关键瓶颈。因此，对卷积算子进行优化与加速无疑会大大提高模型的训练速度。

常见的对卷积进行加速计算的方法一般是 FFT（快速傅里叶变换）。但是注意到本次实验我们是基于 NumPy 这样科学计算库，这个库封装了对于线性代数等矩阵计算（特别是大矩阵计算）的极致优化与加速，因此我们可以尝试将卷积算子转化为矩阵之间的乘法运算，利用 NumPy 的矩阵计算加速，也可以实现对卷积算子的极致优化与加速。

可以使用 im2col（image to column）的思想[6]（这实际上也是 Caffe、Chainer 等深度学习框架中对卷积层的优化方式）来实现卷积算子到矩阵的转换。这个函数的作用是将输入特征（在 CNN 中一般是一个 4 维的矩阵，形状为 $N \times C \times H \times W$）展开成一个二维的矩阵，以使其适应与卷积核（过滤器），如图4.6所示。

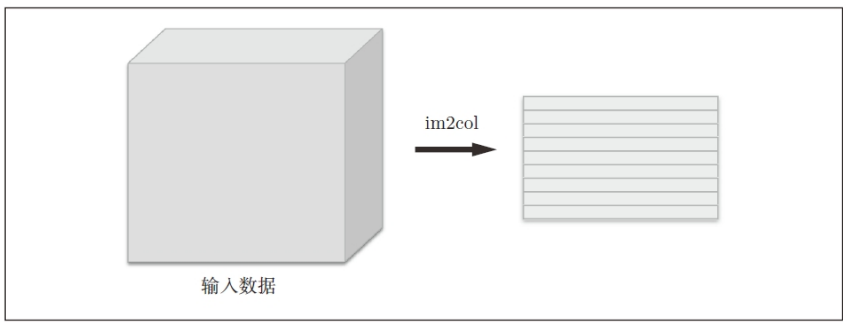

图 4.6: im2col 示意图

之后，只需要把每个卷积核（过滤器）展开为一维向量，C 个卷积核展开后合成一个二维矩阵，然后将 im2col 的结果与 C 个卷积核展开的二维矩阵进行矩阵乘法运算即可，如图4.7所示。

[6] [卷积算子加速] im2col 优化 - 知乎 (zhihu.com)

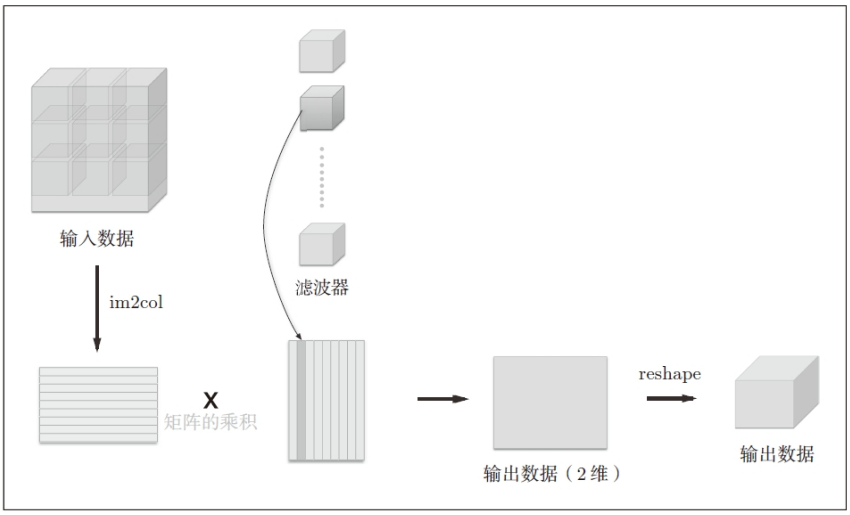

图 4.7: im2col 进行卷积算子加速

需要注意的是，虽然使用 im2col 函数将输入特征展开成二维矩阵会得到比原输入数据更大的张量，会占用更多的内存，但是之后使用 NumPy 对矩阵运算进行优化与加速仍是值得的（Tradeoff），这是由于矩阵运算已被高度优化。

im2col 函数的具体实现如代码1所示。

代码 1: im2col 函数

```
def im2col(input_data, filter_h, filter_w, stride, padding):
    """
    Convert image to column
    :param input_data: [N, C, H, W]
    :param filter_h:
    :param filter_w:
    :param stride:
    :param padding:
    :return: [N*out_h*out_w, C*filter_h*filter_w]
    """
    N, C, H, W = input_data.shape
    out_h = (H + 2*padding - filter_h) // stride + 1
    out_w = (W + 2*padding - filter_w) // stride + 1

    img = np.pad(input_data, [(0,0), (0, 0), (padding, padding), (padding,
        ↪ padding)], 'constant')
    col = np.zeros((N, C, out_h, out_w, filter_h, filter_w))

    for y in  range(out_h):
        y_begin = y*stride
        for x in  range(out_w):
            x_begin = x*stride
```

```
                col[:, :, y, x, :, :] = img[:, :, y_begin:(y_begin+filter_h),
                    ↪ x_begin:(x_begin+filter_w)]

        col = col.transpose(0, 2, 3, 1, 4, 5).reshape(N*out_h*out_w,
            ↪ C*filter_h*filter_w)
        return col
```

在进行梯度反向传播时，需要用到 im2col 函数的逆函数 col2im，采用其逆操作即可，如代码2所示。

代码 2: col2im 函数

```
    def col2im(col, input_shape, filter_h, filter_w, stride, padding):
        """
        Convert column to image
        :param col: [N*out_h*out_w, C*filter_h*filter_w]
        :param input_shape: [N, C, H, W]
        :param filter_h:
        :param filter_w:
        :param stride:
        :param padding:
        :return: [N, C, H, W]
        """
        N, C, H, W = input_shape
        out_h = (H + 2 * padding - filter_h) // stride + 1
        out_w = (W + 2 * padding - filter_w) // stride + 1

        # col: [N, C, out_h, out_w, filter_h, filter_w]
        col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 1, 2,
            ↪ 4, 5)

        img = np.zeros((N, C, H + 2*padding, W + 2*padding))
        for y in  range(out_h):
            y_begin = y*stride
            for x in  range(out_w):
                x_begin = x*stride
                img[:, :, y_begin:(y_begin+filter_h), x_begin:(x_begin+filter_w)] =
                    ↪ col[:, :, y, x, :, :]

        return img[:, :, padding:(H+padding), padding:(W+padding)]
```

注：实际上这两个辅助函数（im2col 与 col2im）不仅可以在**卷积层**使用并加速优化，而且也可以在**池化层**进行使用并加速优化，详见后文。

4.4 其他注意事项

4.4.1 参数的初始化

在机器学习或深度学习中，模型的参数的初始化是一个非常重要的问题。若参数初始化不恰当，则会影响模型的收敛速度以及最终的模型表现（Performance），甚至使得模型不收敛。

显然，不能将参数的值初始化为 0（或者更严格的说，不能初始化为相同的值），这是因为在误差反向传播法中，所有的权重值都会进行相同的更新，权重被更新为相同的值，并拥有了**对称**的值（重复的值）。这使得神经网络拥有许多不同的权重的意义丧失了。为了防止“**权重均一化**”（严格地讲，是为了瓦解权重的对称结构），必须**随机生成初始值**。

在权重的初始化方面，也有一些研究成果，如 Xavier 初始值 [1]、He 初始值 [2] 等。

在本次实验中，进行了简化，采用在正态分布中随机采用的方式进行参数的随机初始化。卷积层的 W 权重的初始化如代码**??**所示。

代码 3: 卷积层的 W 权重随机初始化

```
WEIGHT_INIT_STD = 0.01
...
self.W = np.random.randn(out_channels, in_channels, *kernel_size) * WEIGHT_INIT_STD
```

4.4.2 $Softmax$ 数值计算溢出问题

$Softmax$ 函数经常用于分类问题的输出层中（通常和交叉熵损失结合使用，构成 SoftmaxWithLoss 层），其函数解析式如公式1所示。

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} \quad k = 1, 2, 3, \cdots, n \tag{1}$$

显然，所有的 y_k 相加得 1，$Softmax$ 函数的意义可以看作模型将输入分类为每一类的概率，选取概率最大的那一类作为模型的分类结果。

在使用计算机计算 $Softmax$ 函数时，一方面可以使用 NumPy 处理多维矩阵形式，另一方面也需要注意一些问题。最重要的问题就是计算机的**溢出**问题。由于需要计算 exp() 函数，因此计算结果可能会溢出，变成`inf`等。

注意到以下事实：

$$\begin{aligned} y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} &= \frac{\mathrm{C}\exp(a_k)}{\mathrm{C}\sum_{i=1}^{n} \exp(a_i)} \\ &= \frac{\exp(a_k + \log \mathrm{C})}{\sum_{i=1}^{n} \exp(a_i + \log \mathrm{C})} \\ &= \frac{\exp(a_k + \mathrm{C}')}{\sum_{i=1}^{n} \exp(a_i + \mathrm{C}')} \end{aligned} \tag{2}$$

因此，公式2表明，在进行 $Softmax$ 的指数运算时，加上（或者减去）一个常数并不影响结果。为了防止溢出，一般会使用输入信号中的**最大值**。

$Softmax$ 函数的实现如代码4所示，这里考虑是否进行批处理两种情况。

代码 4: $Softmax$ 函数

```
    def softmax(x):
```

```
    """softmax实现时，为防止溢出，减去最大值"""
    if x.ndim == 2:  # [N, num_classes]
        x_max = np. max(x, axis=1, keepdims=True)  # x_max: [N, 1]
        y = np.exp(x - x_max)  # in case of overflow
        y = y / np. sum(y, axis=1, keepdims=True)
        return y
    else:  # [num_classes]
        x_max = np. max(x)
        y = np.exp(x - x_max)  # in case of overflow
        y = y / np. sum(y)
        return y
```

4.4.3 数据集随机重排

在进行批处理（mini-batch）版本的模型训练时，每次会一次性输入一批数据，在这里可以将数据集随机重排，使得每次选择的一批数据都是随机的，这样可以更好地利用多个数据消除或减少误差与噪声，使得模型训练效果更好。

数据集随机重排如代码5所示。

代码 5: $dataset_shuffle$ 函数

```
def dataset_shuffle(imgs, labels):
    """
    Shuffle Dataset
    :param imgs:
    :param labels:
    :return:
    """
    permutation = np.random.permutation(imgs.shape[0])
    imgs = imgs[permutation]
    labels = labels[permutation]
    return imgs, labels
```

4.5 数据集的加载

由于 MNIST 数据集一般是以二进制的形式保存的，因此根据其保存格式（图像与标签），可以采用代码6来读取并加载。

代码 6: load_dataset 函数

```
def load_dataset(file_path, is_img=True):
    """
    :param file_path:
    :param is_img: image or label
```

```
        :return data:
        For image, data: [n, C, H, W]
        For label, data: [n, 1]
        """
        if is_img:
            offset = 16  # 偏移量为16
            data_size = (IMAGE_CHANNEL, IMAGE_HEIGHT, IMAGE_WIDTH)
        else:
            offset = 8  # 偏移量为8
            data_size = (1, )
        with  open(file_path, 'rb') as  file:
            data = np.frombuffer( file.read(), np.uint8, offset=offset)
        data = data.reshape(-1, *data_size)
        return data
```

在加载数据集时，可以对数据集进行一些变换和操作（Transform），比如将标签转为 One-hot 编码（独热编码），进行数据归一化（转换到 $[0, 1]$ 上），数据白化（减去均值并除以标准差）等。

本次实验中主要实现了**独热编码**（One-hot encoding）与**数据归一化**，详细请参见源代码，这里不再赘述。

图像加载后，可以使用`display_img`函数（调用 PIL 库）来显示并查看图像。

4.6 卷积层的实现

对于每一个层（Layer）的实现，主要包括三部分（后续不再重复）：

1. **初始化：**记录相关参数的值；对于有参数的层，初始化参数。
2. **前向传播：**`forward`，对输入特征进行计算，返回计算结果。
3. **反向传播：**`backward`，对从上游来的损失梯度进行反向传播，计算当前层的参数的梯度（有参数的层），计算并返回损失对输入特征的梯度。

本实验的实现中，所有的层（Layers）中，只有卷积层 Convolution 和全连接层 Linear 具有权重参数。

参数的初始化部分已在前面提到过，这里包括之后不再重复。

4.6.1 前向传播

利用前面提到的 im2col 进行卷积算子的优化与加速，将其规约为矩阵的乘法运算，如代码7所示。

代码 7: 卷积层前向传播

```
    def forward(self, x):
        """
        Forward Propagation
        :param x: [N, C, H, W]
        :return y: [N, out_channels, out_h, out_w]
```

```
        """
        N, C, H, W = x.shape
        filter_h, filter_w = self.kernel_size[0], self.kernel_size[1]
        out_h = (H + 2*self.padding - filter_h) // self.stride + 1
        out_w = (W + 2*self.padding - filter_w) // self.stride + 1

        # col: [N*out_h*out_w, C*filter_h*filter_w]
        col = im2col(x, filter_h, filter_w, stride=self.stride, padding=self.padding)
        # col_W: [in_channels*filter_h*filter_w, out_channels]
        col_W = self.W.reshape(self.out_channels, -1).T

        y = np.matmul(col, col_W)
        if self.bias:
            y += self.b

        y = y.reshape(N, out_h, out_w, self.out_channels).transpose(0, 3, 1, 2)

        # Store Variables
        self.x = x
        self.col = col
        self.col_W = col_W

        return y
```

其中，对于高度和宽度分别为 H 和 W 的输入特征，经过大小为 $FH \times FW$ 的卷积核的卷积运算后，得到的输出特征的大小（OH 和 OW）的计算可由公式3进行计算。

$$\begin{aligned} OH &= \frac{H + 2 \times padding - FH}{stride} + 1 \\ OW &= \frac{W + 2 \times padding - FW}{stride} + 1 \end{aligned} \tag{3}$$

4.6.2 反向传播

在进行损失梯度的反向传播时，由于已经将卷积运算规约为矩阵的乘法和加法运算（线性运算，可以理解为**仿射变换**），因此这部分的损失梯度的反向传播也同样可以规约为**矩阵线性运算（仿射变换）**的损失梯度反向传播。

仿射变换（矩阵线性运算）的计算图如图4.8所示。

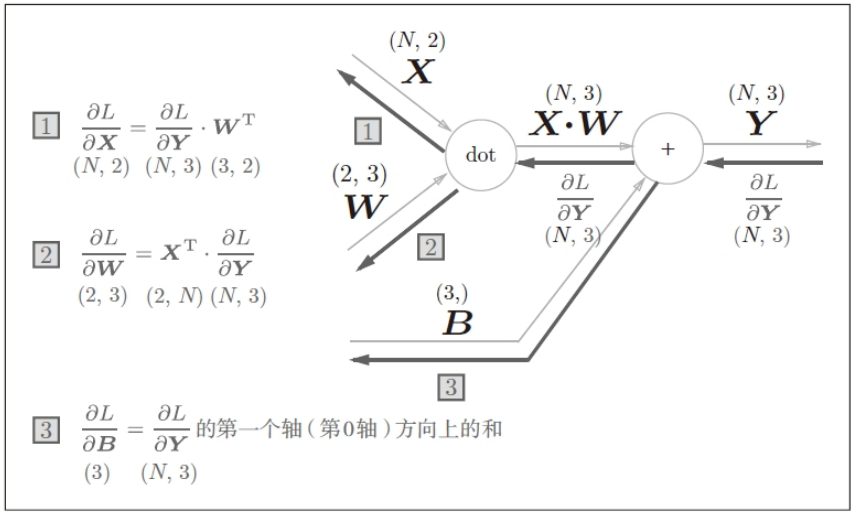

图 4.8: 仿射变换（矩阵线性运算）的计算图

除此之外，还需要用到 col2im 函数，将先前 im2col 变换的结果还原，该部分如代码8所示。

代码 8: 卷积层反向传播

```python
def backward(self, dout):
    """
    Backward Propagation
    :param dout: [N, out_channels, out_h, out_w]
    :return dx: [N, C, H, W]
    """
    filter_h, filter_w = self.kernel_size[0], self.kernel_size[1]

    # dout: [N*out_h*out_w, out_channels]
    dout = dout.transpose(0, 2, 3, 1).reshape(-1, self.out_channels)

    # db: [out_channels]
    if self.bias:
        db = np. sum(dout, axis=0)
    else:
        db = None

    # dW: [in_channels*filter_h*filter_w, out_channels]
    dW = np.matmul(self.col.T, dout)
    # dW: [out_channels, in_channels, filter_h, filter_w]
    dW = dW.transpose(1, 0).reshape(self.out_channels, self.in_channels, filter_h,
        ↪ filter_w)

    # dcol: [N*out_h*out_w, in_channels*filter_h*filter_w]
    dcol = np.matmul(dout, self.col_W.T)
```

```
25
26          # dx: [N, C, H, W]
27          dx = col2im(dcol, self.x.shape, filter_h, filter_w, stride=self.stride,
                ↪ padding=self.padding)
28
29          # Store Gradients
30          self.dW = dW
31          self.db = db
32
33          return dx
```

4.7 池化层的实现

本实验中的池化层采用最大池化（MaxPooling），即将输入特征的对应感受野部分取最大值作为输出。

池化层的实现与卷积层类似，在前向传播时也需要用到 im2col 辅助函数进行优化与加速，只不过矩阵的乘法运算需要改为沿轴axis=1取最大值；在反向传播时使用 col2im 函数还原，同时损失的梯度只能沿取最大值的位置向后传递，其他位置被截断（为 0）。

需要注意的是，**池化层并不含可学习的参数**。

4.7.1 前向传播

代码 9: MaxPooling 前向传播

```
1   def forward(self, x):
2       """
3       Forward Propagation
4       :param x: [N, C, H, W]
5       :return y: [N, C, out_h, out_w]
6       """
7       N, C, H, W = x.shape
8       pool_h, pool_w = self.kernel_size[0], self.kernel_size[1]
9       out_h = (H + 2*self.padding - pool_h) // self.stride + 1
10      out_w = (W + 2*self.padding - pool_w) // self.stride + 1
11
12      # col: [N*out_h*out_w, C*pool_h*pool_w]
13      col = im2col(x, pool_h, pool_w, stride=self.stride, padding=self.padding)
14      # col: [N*out_h*out_w*C, pool_h*pool_w]
15      col = col.reshape(-1, pool_h*pool_w)
16
17      arg_max = np.argmax(col, axis=1)
18      y = np. max(col, axis=1)
19      y = y.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)
```

```
20
21          # Store Variables
22          self.x = x
23          self.arg_max = arg_max
24
25          return y
```

4.7.2 反向传播

代码 10: MaxPooling 反向传播

```
1       def backward(self, dout):
2           """
3           Backward Propagation
4           :param dout: [N, C, out_h, out_w]
5           :return dx: [N, C, H, W]
6           """
7           N, C, out_h, out_w = dout.shape
8           pool_h, pool_w = self.kernel_size[0], self.kernel_size[1]
9
10          # dout: [N*out_h*out_w*C]
11          dout = dout.transpose(0, 2, 3, 1).reshape(-1)
12
13          dx = np.zeros((N*out_h*out_w*C, pool_h*pool_w))
14          dx[np.arange(self.arg_max.size), self.arg_max.flatten()] = dout
15
16          # dx: [N*out_h*out_w, C*pool_h*pool_w]
17          dx = dx.reshape(N*out_h*out_w, C*pool_h*pool_w)
18          # dx: [N, C, H, W]
19          dx = col2im(dx, self.x.shape, pool_h, pool_w, stride=self.stride,
                ↪ padding=self.padding)
20
21          return dx
```

4.8 全连接层的实现

全连接层实际上就是仿射变换（矩阵的线性运算），因此正向传播即为矩阵的乘法和加法（偏置项 b）运算，而反向传播同样可以使用图4.8中的仿射层的计算图来进行。

需要注意的一点是，在正向传播时，该层需要保存输入特征的形状，在进行计算时，会将输入特征转换为二维矩阵并进行计算[7]，在反向传播时，最后需要将得到的损失梯度进行`reshape`操作，转换

[7]如果是使用 PyTorch，则在卷积层与全连接层衔接时，通常会有一个函数，如 view 函数，用于将输入特征展平成一个 batch_size 行 n_features 列的二维矩阵

为原输入特征的形状。

其他部分基本与前面相同，详情请参见源代码，这里不再赘述。

4.9 激活函数层的实现

本实验中，激活函数层采用 *ReLU* 函数（Rectified Linear Unit, 修正线性单元函数）。其定义如公式4所示。

$$ReLU(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{4}$$

其对应的梯度如公式5所示。

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{5}$$

其计算图如图4.9所示。

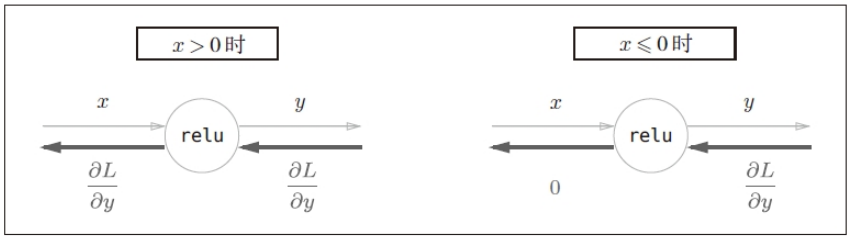

图 4.9: ReLU 层计算图

据此，可以写出 ReLU 层的前向传播和反向传播，如代码11所示。

代码 11: ReLU 层

```
class ReLU:
    """
    ReLU Activation Layer
    """
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)

        y = x.copy()
        y[self.mask] = 0

        return y

```

```
        def backward(self, dout):
            dx = dout
            dx[self.mask] = 0
            return dx
```

4.10 SoftmaxWithLoss 层的实现

这一层实际上是两个层的封装，前一个层使用 $Softmax$ 函数将输出转换为概率形式，后一个层在此基础上进行损失的计算，一般采用交叉熵损失（Cross Entropy Loss），原因已在前面陈述，这里不再重复。

SoftmaxWithLoss 层用来作为输出层，只用于训练过程中的损失梯度反向传播，而在模型的推理过程中则不需要 SoftmaxWithLoss 这一层。

4.10.1 前向传播

$Softmax$ 函数如公式1所示。

交叉熵损失的计算如公式6所示。

$$E = -\sum_{k} t_k \log(y_k) \tag{6}$$

据此，可以给出计算交叉熵损失的代码，如代码12所示。

代码 12: 交叉熵损失函数

```
def cross_entropy_loss(y, t, epsilon=1e-7):
    """
    Cross Entropy Loss
    :param y: output
    :param t: ground truth labels
    :param epsilon: = 1e-7  防止log的真数为0
    :return:
    """
    if y.ndim == 1:
        y = y.reshape(1, -1)
        t = t.reshape(1, -1)

    if t.shape[1] == y.shape[1]:  # one-hot encoding
        # Convert to NON one-hot encoding(True Label)
        t = np.argmax(t, axis=1)

    batch_size = y.shape[0]
    return -np. sum(np.log(y[np.arange(batch_size), t] + epsilon)) / batch_size
```

其中的epsilon参数的作用主要是为了防止 log 的真数为 0。

因此，SoftmaxWithLoss 层的前向传播即为两个函数的复合，如代码13所示。

代码 13: SoftmaxWithLoss 层前向传播

```
def forward(self, x, t):
    """
    Forward Propagation
    :param x: Output of model
    :param t: Ground-Truth Label
    :return loss: scalar
    """
    self.y = softmax(x)
    self.t = t
    self.loss = cross_entropy_loss(self.y, t)
    return self.loss
```

4.10.2 反向传播

对于反向传播，正如我们前面提到的，其损失梯度具有一个比较简单的表达形式，其计算图如图4.10所示。

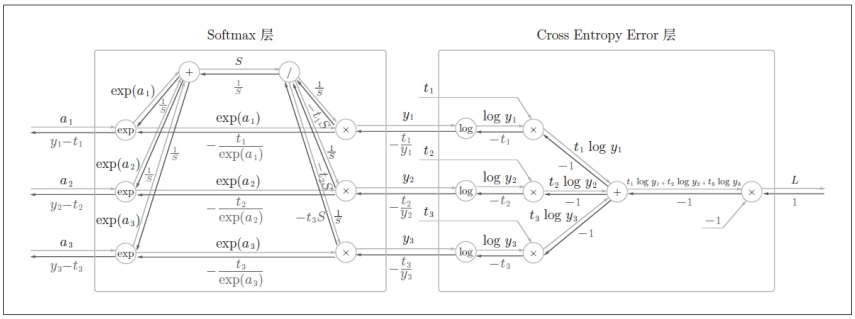

图 4.10: SoftmaxWithLoss 层的计算图

基于此，SoftmaxWithLoss 层的反向传播如代码14所示。

代码 14: SoftmaxWithLoss 层反向传播

```
def backward(self, dout=1):
    """
    Backward Propagation
    :param dout: = 1
    :return dx: [N, num_classes]
    """
    batch_size = self.y.shape[0]

    if self.t.size == self.y.size:  # one-hot encoding
        dx = self.y - self.t
```

```
        else:  # NON one-hot encoding
            dx = self.y.copy()
            dx[np.arange(batch_size), self.t.flatten()] -= 1

        return dx
```

4.11 优化器的实现

优化器主要负责根据每一层的梯度对模型的参数按照一定的算法进行更新迭代，使得模型的损失尽可能小。

本次实验中，实现了两个优化器：

1. **SGD:** 随机梯度下降（Stochastic Gradient Descent）
2. **Adam:** 自适应矩估计（Adaptive Moment Estimation）

4.11.1 SGD

随机梯度下降法的原理是函数的梯度方向是函数上升最快的的方向，那么对于损失函数而言，梯度的负方向就是损失函数下降最快的方向，因此每次参数的更新都沿着梯度的负方向进行。

模型的参数为 θ_t，学习率为 η，损失函数为 $J(\theta_t; x^{(i)}, y^{(i)})$，则其数学表达形式如公式7所示。

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla J(\theta_t; x^{(i)}, y^{(i)}) \tag{7}$$

SGD 的实现如代码15所示。

代码 15: SGD 优化器

```
class SGD:
    """
    Stochastic Gradient Descent
    """
    def __init__(self, model_layers, lr=0.01):
        """
        Stochastic Gradient Descent
        :param model_layers:
        :param lr: learning rate
        """
        self.model_layers = model_layers
        self.lr = lr

    def step(self):
        """
        Update Model Parameters
        :return:
```

```
        """
        for layer in self.model_layers.values():
            if is_weighted_layer(layer):
                layer.W -= self.lr * layer.dW
                if layer.bias:
                    layer.b -= self.lr * layer.db
```

4.11.2 Adam

Adam 是一种自适应学习率的优化算法，结合了梯度的一阶矩估计和二阶矩估计。Adam 融合了 Momentum 和 AdaGrad 二者的优点。

Adam 优化器的参数更新表达式如公式8所示。

$$
\begin{aligned}
m_t &= \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \\
v_t &= \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \cdot \hat{m}_t
\end{aligned}
\tag{8}
$$

其中，g_t 是损失函数的梯度，m_t 是梯度的一阶矩估计，v_t 是梯度的二阶矩估计，β_1 和 β_2 是衰减率（通常取 0.9 和 0.999），η 是学习率，ε 是一个小的常数，防止除零错误。

相比于 SGD，Adam 能够实现**鞍点逃逸**等，避免陷入局部最优解，具有一定的优势。

根据公式8，Adam 优化器的实现如代码16所示。

代码 16: Adam 优化器

```
class Adam:
    """
    Adam
    """
    def __init__(self, model_layers, lr=0.001, beta1=0.9, beta2=0.999):
        """
        Adam
        :param model_layers:
        :param lr: learning rate
        :param beta1:
        :param beta2:
        """
        self.model_layers = model_layers
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
```

```
            self.t = 0
            self.m = {}
            self.v = {}

            for layer_name, layer in self.model_layers.items():
                if is_weighted_layer(layer):
                    self.m[layer_name] = {}
                    self.v[layer_name] = {}
                    self.m[layer_name]['W'] = np.zeros_like(layer.W)
                    self.v[layer_name]['W'] = np.zeros_like(layer.W)
                    if layer.bias:
                        self.m[layer_name]['b'] = np.zeros_like(layer.b)
                        self.v[layer_name]['b'] = np.zeros_like(layer.b)

        def step(self):
            """
            Update Model Parameters
            :return:
            """
            self.t += 1
            for layer_name, layer in self.model_layers.items():
                if is_weighted_layer(layer):
                    self.m[layer_name]['W'] = self.beta1 * self.m[layer_name]['W'] + (1
                        ↪ - self.beta1) * layer.dW
                    self.v[layer_name]['W'] = self.beta2 * self.v[layer_name]['W'] + (1
                        ↪ - self.beta2) * (layer.dW ** 2)
                    m_w = self.m[layer_name]['W'] / (1 - self.beta1 ** self.t)
                    v_w = self.v[layer_name]['W'] / (1 - self.beta2 ** self.t)
                    layer.W -= self.lr * m_w / (np.sqrt(v_w) + 1e-7)
                    if layer.bias:
                        self.m[layer_name]['b'] = self.beta1 * self.m[layer_name]['b']
                            ↪ + (1 - self.beta1) * layer.db
                        self.v[layer_name]['b'] = self.beta2 * self.v[layer_name]['b']
                            ↪ + (1 - self.beta2) * (layer.db ** 2)
                        m_b = self.m[layer_name]['b'] / (1 - self.beta1 ** self.t)
                        v_b = self.v[layer_name]['b'] / (1 - self.beta2 ** self.t)
                        layer.b -= self.lr * m_b / (np.sqrt(v_b) + 1e-7)
```

在后面的实验验证中，我们也会发现，在 MNIST 手写数字识别任务中（LeNet-5 网络）使用 Adam 优化器要明显由于使用 SGD 优化器。

4.12 LeNet-5 模型

根据前面的铺垫，将各个层（Layer）组合起来，构建 LeNet-5 模型。

使用 Python 的 OrderedDict（有序字典）来保存各个层，由于输出层 SoftmaxWithLoss 只在模型训练时使用，因此输出层进行单独保存。

本实验中，LeNet-5 的基本架构如17所示。

代码 17: LeNet-5 模型架构

```
class LeNet5:
    """
    LeNet-5
    """
    def __init__(self, img_channels=1, num_classes=10):
        """
        LeNet-5
        :param img_channels: int = 1
        :param num_classes: int = 10
        """
        self.layers = OrderedDict()

        self.layers['conv1'] = Convolution(img_channels, 6, kernel_size=5)
        self.layers['relu1'] = ReLU()
        self.layers['pool1'] = MaxPooling(kernel_size=2, stride=2)

        self.layers['conv2'] = Convolution(6, 16, kernel_size=5)
        self.layers['relu2'] = ReLU()
        self.layers['pool2'] = MaxPooling(kernel_size=2, stride=2)

        self.layers['fc1'] = Linear(16*4*4, 120)
        self.layers['relu3'] = ReLU()

        self.layers['fc2'] = Linear(120, 84)
        self.layers['relu4'] = ReLU()

        self.layers['fc3'] = Linear(84, num_classes)

        # 输出层 SoftmaxWithLoss 仅用于训练阶段，推理阶段不需要
        self.output_layer = SoftmaxWithLoss()
```

除此之外，封装好预测函数`predict`、计算损失函数`loss`、反向传播函数`backward`、获取准确率函数`get_accuracy`、获取混淆矩阵（Confusion Matrix）函数`get_confusion_matrix`等，详情参见源代码，这里不再赘述。

4.13 训练器（Trainer）封装

将模型的训练过程与测试过程封装为一个训练器类（Trainer），作为训练模型的代理。

在 Trainer 中封装模型**超参数**以及一些其他的**训练相关参数**，对外提供训练接口，并提供**日志**输出（训练过程中的**损失**、训练集与测试集上的**准确率**以及训练所用的**时间**等），提供获取训练结果与评估结果、**混淆矩阵**、模型参数等的接口，同时也支持**加载模型参数**（参数加载在其他模块也做了逐层封装，这意味着可以支持**预训练** Pre-Trained 模型）。

4.14 其他事项

4.14.1 模型参数的获取与加载

逐层封装，实现了模型参数的获取（保存）与加载（设置）功能，允许用户将训练好的模型的参数打印或保存下来，同时也允许用户直接加载一个已训练好的模型的参数，这意味着可以支持**预训练** Pre-Trained 模型的使用。

5 实验结果

5.1 实验设置

本实验在笔者的个人电脑上使用 CPU 运行，主要的训练参数如表1所示。

表 1: 训练参数

参数项	参数值
Optimizer	Adam
lr	0.001
beta1	0.9
beta2	0.999
Epoch	50
Batch Size	64
Random Shuffle	True
Normalized	True

其中，Random Shuffle 指的是是否使用训练集的随机重排，Normalized 指的是是否将图像数据归一化（包括训练集与测试集）。

注意：实际上，除此之外，笔者还多次尝试了使用 SGD 优化器，但是其效果非常不理想，模型训练 200 个 Epoch，其准确率仍然只有 10% 左右，因此在本报告中不再过多说明。

5.2 实验结果及分析

按照表1中的训练参数对 LeNet-5 模型进行训练，其训练过程中的损失如图5.11所示，训练过程中训练集与测试集上的准确率的变化如图5.12所示。

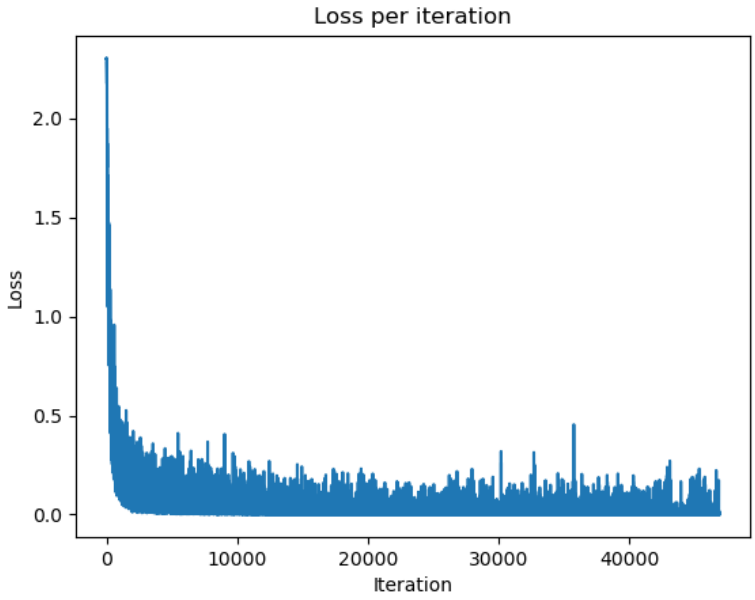

图 5.11: 训练过程中的损失变化

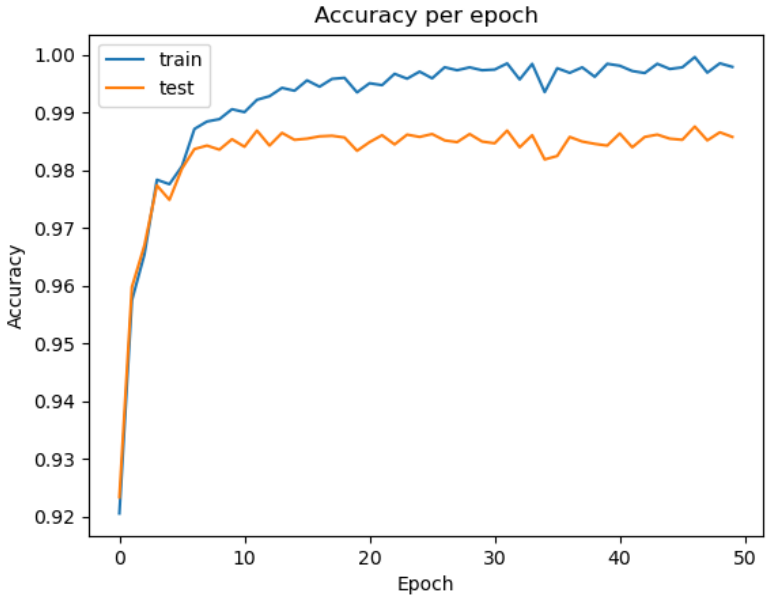

图 5.12: 训练过程中训练集与测试集上准确率的变化

模型在训练过程中的损失刚开始随着迭代次数急剧下降，随后缓慢下降，最终波动式地趋于平稳，说明模型逐渐收敛。

可以看到模型的训练效果非常好，在第一个 Epoch 迭代后，模型在训练集与测试集上的准确率就已经**均**达到了 92% 以上；在随后的 49 个 Epoch 中，模型在训练集与测试集上的准确率同步上升，在第 20 个 Epoch 后基本趋于稳定。最终模型的测试集准确率能够稳定在 99.8% 左右，测试集准确率能够稳定在 98.6% 左右，模型表现非常好。

由于模型在训练集与测试集上的准确率同步上升并最终趋于稳定，因此并没有出现过拟合现象（当

然也没有出现欠拟合现象），因此本次训练的 LeNet-5 模型在 MNIST 手写数字识别分类任务中取得了非常好的表现。

除此之外，打印模型训练完成后在训练集与测试集上的混淆矩阵，如图5.13和图5.14所示。

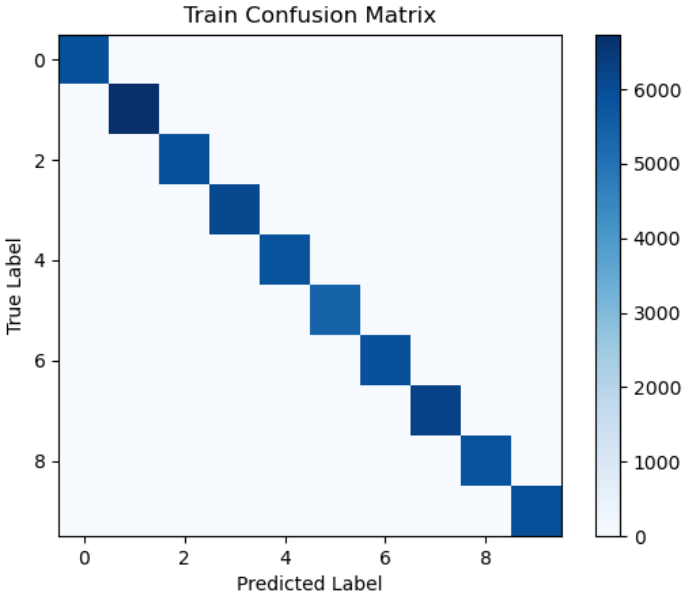

图 5.13: 训练集混淆矩阵

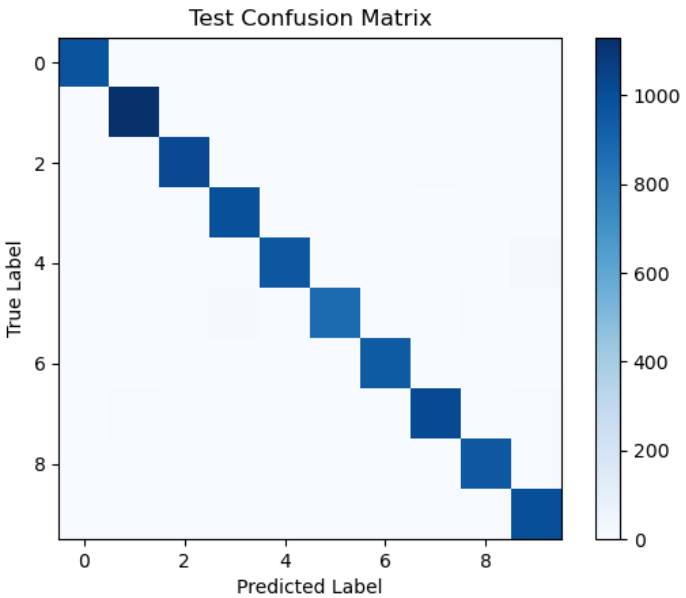

图 5.14: 测试集混淆矩阵

可以看到，模型在训练集与测试集上的混淆矩阵深蓝色均集中在主对角线上，其他区域颜色非常浅，说明模型的预测效果非常准确，能够准确分类每一个手写数字。

其实，对于那些 LeNet-5 模型分类错误的手写数字图像单独查看，可以发现其非常难以辨别，即使是人也很难分辨，比如某些书写非常近似的 0 与 6，1 与 7 等，因此，模型在训练集与测试集上取

得这样一个高准确率已经非常能够证明其手写数字识别能力。

5.3 训练时间分析

训练过程中，根据日志输出（带有训练时间信息），可以看到每一轮 Epoch（包括一轮 Epoch 的训练以及一次在测试集上的测试）用时大概 1 分钟左右，训练完 50 个 Epoch 大约需要不到 1 小时的时间，虽然这个训练速度仍不及使用 PyTorch 框架并在GPU上进行训练的速度更快，但是仍然算得上比较快速，这得益于之前的一些优化与加速，说明这些优化与加速非常成功。

6 未来展望

1. 后续会尝试使用并行化技术（如移植到 GPU 上进行计算）进行训练，进一步提高模型的训练速度。
2. 未来可能会从数学理论与实验验证上分析为什么 LeNet-5 在 MNIST 上使用 SGD 的效果如此之差，为什么 Adam 的效果要明显由于 SGD。
3. 本项目代码计划于实验截至后在GitHub开源。

7 结语

本次实验笔者完整又比较完美地基于 NumPy 实现了 LeNet-5，并采用了一些优化与加速地技巧，实现了模型的快速训练，并取得了较好的模型表现（Performance），因此可以认为本次实验比较成功。

除此之外，笔者在实验中发现的 LeNet-5 在 MNIST 上使用 SGD 的效果远不及 Adam，值得注意和分析，笔者计划未来对此进行详细研究。

8 致谢

这一学期的《机器学习》课程让我收获颇丰，在此特别感谢谢晋老师以及其他教学组老师的细致讲解，感谢助教学长学姐的耐心指导！

参考文献

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] 斋藤康毅. 深度学习入门：基于 *Python* 的理论与实现. 图灵程序设计丛书. 人民邮电出版社, 7 2018.