

**《机器学习导论》课程设计报告**

**选题： 衣服图像分类--Fashion-MNIST**

**课程名称： 机器学习导论**

**专业班级： 计算机本硕博2301班**

**姓 名： 王家乐**

**学 号： U202315763**

**指导教师： 李钦宾**

**完成日期： 2025.5.8**

**计算机科学与技术学院**

**目录**

[1 问题定义与理解 3](#_Toc1608)

[2 数据分析及处理 4](#_Toc7256)

[2.1 数据集介绍 4](#_Toc5008)

[2.2 数据集加载 4](#_Toc3938)

[2.3 数据预处理 5](#_Toc3697)

[3 模型构建 6](#_Toc13540)

[3.1 cnn-by-myself (仿照LeNet) 6](#_Toc9497)

[3.1.1 网络结构 6](#_Toc32385)

[3.1.2 参数量 7](#_Toc13856)

[3.1.3 关键技术 8](#_Toc15849)

[3.2 cnn-by-pytorch 8](#_Toc27661)

[3.2.1 网络结构 8](#_Toc23764)

[3.2.2 参数量 10](#_Toc30168)

[3.2.3 关键技术 10](#_Toc4596)

[4 实验结果与分析 11](#_Toc19613)

[4.2 cnn-by-myself 11](#_Toc25178)

[4.1.1 超参数 11](#_Toc14310)

[4.1.2 训练过程 11](#_Toc31525)

[4.1.3 损失&准确率 11](#_Toc3375)

[4.1.4 测试结果 12](#_Toc3438)

[4.2 cnn-by-pytorch 12](#_Toc19596)

[4.2.1 超参数 12](#_Toc26318)

[4.2.2 训练过程 13](#_Toc2982)

[4.2.3 损失&准确率 13](#_Toc4857)

[4.2.4 测试结果 13](#_Toc26975)

[4.3 模型比较 14](#_Toc30525)

[5 结论与改进 15](#_Toc22499)

[5.1 结论 15](#_Toc7909)

[5.2 改进方向 15](#_Toc14318)

[附录-cnn-by-myself关键代码 16](#_Toc26581)

# 1 问题定义与理解

随着深度学习技术的迅速发展，卷积神经网络（Convolutional Neural Network, CNN）在图像识别和计算机视觉领域取得了显著成果。为了深入理解CNN的基本原理和实际应用，本课程设计选取了Fashion-MNIST作为实验数据集，围绕图像分类任务展开研究与实现。  
 Fashion-MNIST是由服饰电商公司Zalando Research发布的图像数据集，该数据集包含10类不同的服饰类别（如T恤、裤子、鞋子等），每类包含6000张训练图像和1000张测试图像。所有图像均为灰度图，尺寸为28x28像素，格式统一，适合用于深度学习模型的快速训练与测试。该数据集因其规模适中、预处理简单且分类任务明确，成为图像分类入门项目中常用的基准数据集。



图1--Fashion-MNIST数据集

本课程设计旨在基于 Fashion-MNIST 数据集，构建卷积神经网络（CNN）模型，实现对服饰图像的自动分类。为了对比不同实现方式，采用了两种技术路线进行对比设计：

一是手动实现的 CNN 模型：不依赖高层框架，手动构建卷积、池化、激活等模块，以加深对CNN内部机制的理解。

二是基于PyTorch框架的模型实现：利用PyTorch这一主流深度学习框架构建CNN模型，从工程实践角度提高开发效率与模型性能。

# 2 数据分析及处理

## 2.1 数据集介绍

整个数据集包含60000张训练图像和10000张测试图像，所有图像均为灰度图，尺寸为28x28像素，文件格式为.gz的二进制格式，包含10 类服饰图像，类别标签如下：

| Label | Class |
| --- | --- |
| 0 | T-shirt/top |
| 1 | Trouser |
| 2 | Pullover |
| 3 | Dress |
| 4 | Coat |
| 5 | Sandal |
| 6 | Shirt |
| 7 | Sneaker |
| 8 | Bag |
| 9 | Ankle boot |

表1--数据标签及类别

## 2.2 数据集加载

使用以下代码从.gz文件中加载图像并转换为1\*784的向量：

def load\_mnist(path, kind='train'):

    """Load MNIST data from `path`"""

    labels\_path = f'{path}/{kind}-labels-idx1-ubyte.gz'

    images\_path = f'{path}/{kind}-images-idx3-ubyte.gz'

    with gzip.open(labels\_path, 'rb') as lbpath:

        magic, n = struct.unpack('>II', lbpath.read(8))

        labels = np.frombuffer(lbpath.read(), dtype=np.uint8)

    with gzip.open(images\_path, 'rb') as imgpath:

        magic, num, rows, cols = struct.unpack('>IIII', imgpath.read(16))

        images = np.frombuffer(imgpath.read(), dtype=np.uint8).reshape(len(labels), 784)

    return images, labels

显示前30个示例图像：



图2--示例图像

## 2.3 数据预处理

归一化：将像素值缩放到[0, 1]区间。

cnn-by-myself中使用np.eye( )将类别标签转换为one-hot编码。

*# 数据预处理*

train\_images = train\_images / 255.0  *# 归一化*

train\_labels = np.eye(10)[train\_labels]  *# one-hot编码*

cnn-by-pytorch中使用DataLoader处理数据。

*# 创建数据加载器*

train\_dataset = TensorDataset(x\_train, y\_train)

test\_dataset = TensorDataset(x\_test, y\_test)

train\_loader =DataLoader(train\_dataset,batch\_size=128,shuffle=True,num\_workers=4)

test\_loader = DataLoader(test\_dataset,batch\_size=256,shuffle=False,num\_workers=4)

# 3 模型构建

## 3.1 cnn-by-myself (仿照LeNet)

### 3.1.1 网络结构

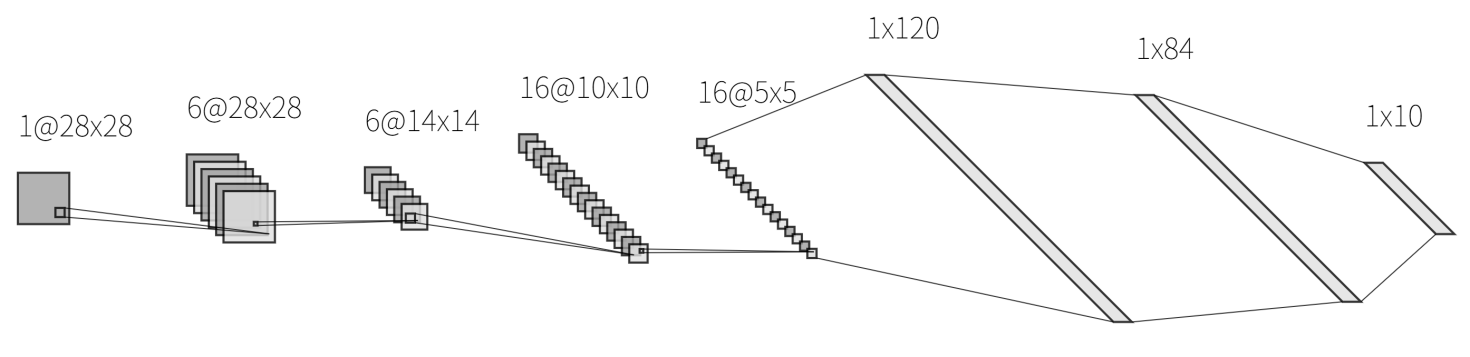


图3--cnn-by-myself网络结构（1）

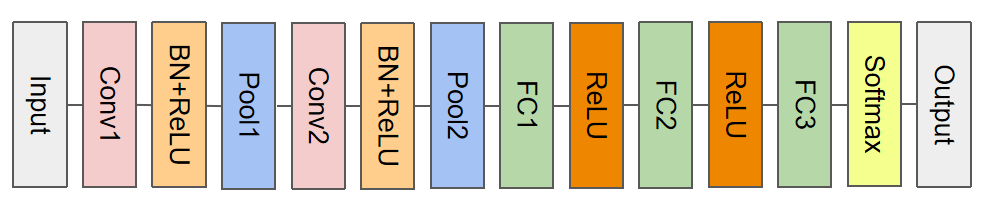


图4--cnn-by-myself网络结构（2）

(1) 输入层

·输入尺寸：1×28×28（单通道灰度图像）

(2) 卷积层1（Conv\_1）

·输入通道：1

·输出通道：6

·卷积核尺寸：5×5

·步长（stride）：1

·填充（padding）：2（保持特征图尺寸不变）

·初始化方法：He初始化

(3) 批归一化层1（BN\_1）

·特征通道数：6

·γ（缩放因子）：6

·β（平移因子）：6

(4) 最大池化层1（Pool\_1）

·池化尺寸：2×2

·步长：2

·输出尺寸：6×14×14

(5) 卷积层2（Conv\_2）

·输入通道：6

·输出通道：16

·卷积核尺寸：5×5

·步长（stride）：1

·填充（padding）：0

(6) 批归一化层2（BN\_2）

·特征通道数：16

·γ（缩放因子）：16

·β（平移因子）：16

(7) 最大池化层2（Pool\_2）

·池化尺寸：2×2

·步长：2

·输出尺寸：16×5×5

(8) 全连接层1（FC\_1）

·输入维度：16×5×5 = 400

·输出维度：120

(9) 全连接层2（FC\_2）

·输入维度：120

·输出维度：84

(10) 全连接层3（FC\_3）

·输入维度：84

·输出维度：10（对应10个类别）

(11) Softmax输出层

·输出形式：10维概率分布​

·Softmax函数：

### 3.1.2 参数量

Layer (type)         Output Shape         Param #

===================================================

Conv2D               (6, 28, 28)          156

BatchNorm            (6, 28, 28)          12

ReLU                 (6, 28, 28)          0

MaxPool2D            (6, 14, 14)          0

Conv2D               (16, 10, 10)         2416

BatchNorm            (16, 10, 10)         32

ReLU                 (16, 10, 10)         0

MaxPool2D            (16, 5, 5)           0

FullyConnected       (120,)               48120

ReLU                 (120,)               0

FullyConnected       (84,)                10164

ReLU                 (84,)                0

FullyConnected       (10,)                850

===================================================

Total params: 61750

### 3.1.3 关键技术

1. 参数初始化：

·卷积层和全连接层采用He初始化：

·偏置项初始化为0

1. 参数更新方法：

·使用带动量的随机梯度下降（SGD）：

·动量系数：0.9

·学习率通过实验确定

1. 正则化策略：

·批归一化：每个卷积层后接BN层

·隐含正则化：ReLU的稀疏激活特性

1. 维度变换：

·使用im2col技巧加速卷积运算

·池化层采用最大池化保留显著特征

1. 损失函数：

·交叉熵损失：

## 3.2 cnn-by-pytorch

### 3.2.1 网络结构

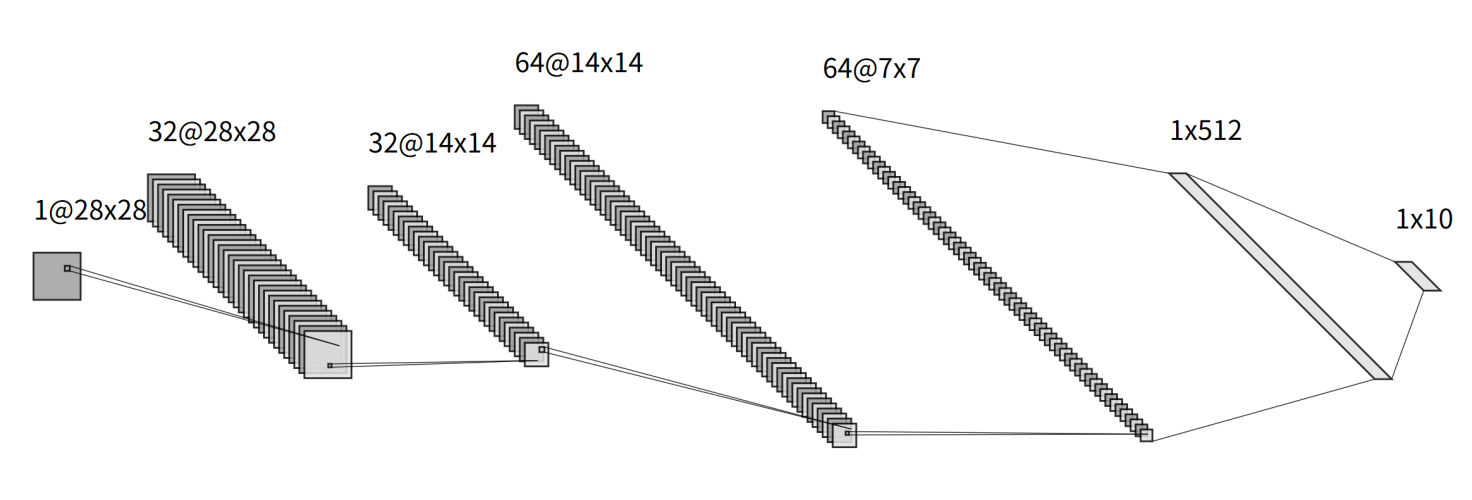


图5--cnn-by-pytorch网络结构

(1) 输入层  
 ·输入尺寸：1×28×28（单通道灰度图像）

(2) 卷积层1（Conv\_1）  
 ·输入通道：1  
 ·输出通道：32  
 ·卷积核尺寸：3×3  
 ·步长（stride）：1  
 ·填充（padding）：1  
 ·初始化方法：He初始化

(3) 批归一化层1（BN\_1）  
 ·特征通道数：32  
 ·γ（缩放因子）：32  
 ·β（平移因子）：32

(4) 最大池化层1（Pool\_1）  
 ·池化尺寸：2×2  
 ·步长：2  
 ·输出尺寸：32×14×14

(5) 卷积层2（Conv\_2）  
 ·输入通道：32  
 ·输出通道：64  
 ·卷积核尺寸：3×3  
 ·步长（stride）：1  
 ·填充（padding）：1

(6) 批归一化层2（BN\_2）  
 ·特征通道数：64  
 ·γ（缩放因子）：64  
 ·β（平移因子）：64

(7) 最大池化层2（Pool\_2）  
 ·池化尺寸：2×2  
 ·步长：2  
 ·输出尺寸：64×7×7

(8) 全连接层1（FC\_1）  
 ·输入维度：64×7×7 = 3136  
 ·输出维度：512

(9) 全连接层2（FC\_2）  
 ·输入维度：512

·输出维度：10

(10) Softmax输出层  
 ·输出形式：10维概率分布​

·Softmax函数：

### 3.2.2 参数量

Layer (type)         Output Shape         Param #

===================================================

Conv2D*\_*1(32, 28, 28)320 

BatchNorm1          (32, 28, 28)          64

ReLU                (32, 28, 28)          0

MaxPool2D\_1         (32, 14, 14)          0

Conv2D*\_*2(64, 14, 14)18496

BatchNorm2          (64, 14, 14)          128

ReLU                (64, 14, 14)          0

MaxPool2D\_2         (64, 7, 7)            0

Flatten             (3136,)               0

FC1                 (512,)                1606144

Dropout             (512,)                0

FC2                 (10,)                 5130

===================================================

Total params: 1,630,282

### 3.2.3 关键技术

1. 参数初始化：

·卷积层采用He初始化：

·全连接层采用Xavier初始化：

·偏置项初始化为0

1. 参数更新方法：

·使用Adam优化器

·动量系数：β1=0.9，β2=0.999

·学习率通过实验确定

1. 正则化策略：

·批归一化：每个卷积层后接BN层

·Dropout：全连接层后设置0.5丢弃率

· 双重正则机制：BN减少内部协变量偏移 + Dropout防止过拟合

1. 损失函数：

·交叉熵损失：

# 4 实验结果与分析

## 4.2 cnn-by-myself

### 4.1.1 超参数

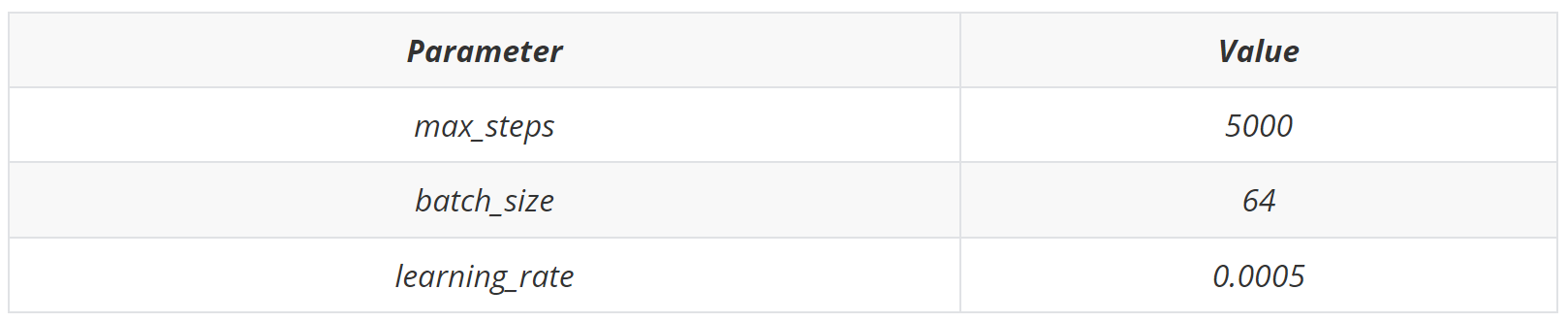


表2--cnn-by-myself超参数

### 4.1.2 训练过程

每1000steps保存一次模型，并保存训练日志(cnn-by-myself\logs目录下)，每个step绘制一个点，训练过程在Fashion-MNIST-Chiale\cnn-by-myself\demo.ipynb 目录下。

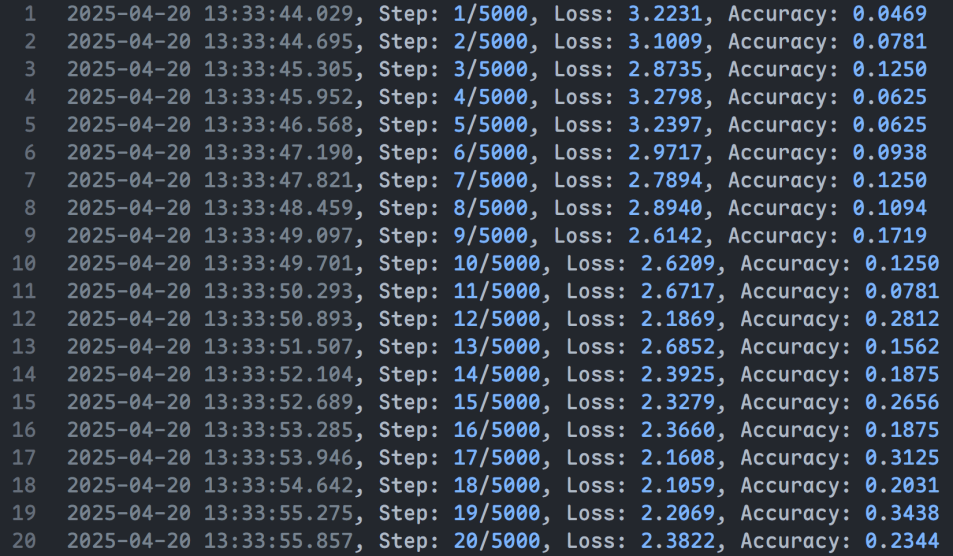


图6--cnn-by-myself训练过程

### 4.1.3 损失&准确率

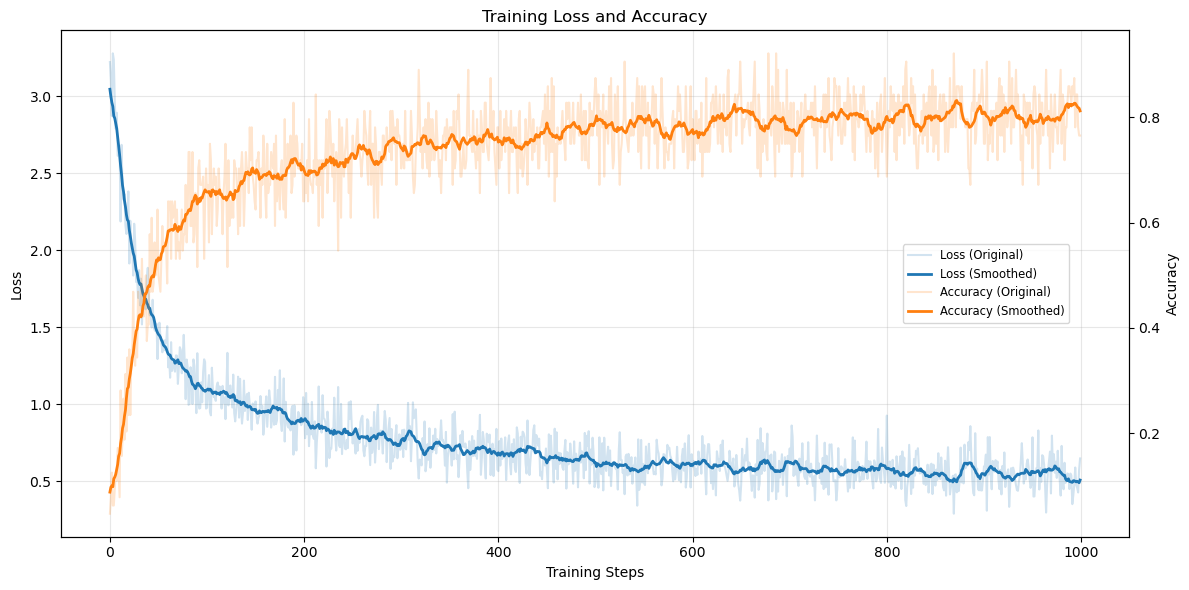


图7--1000steps损失&准确率

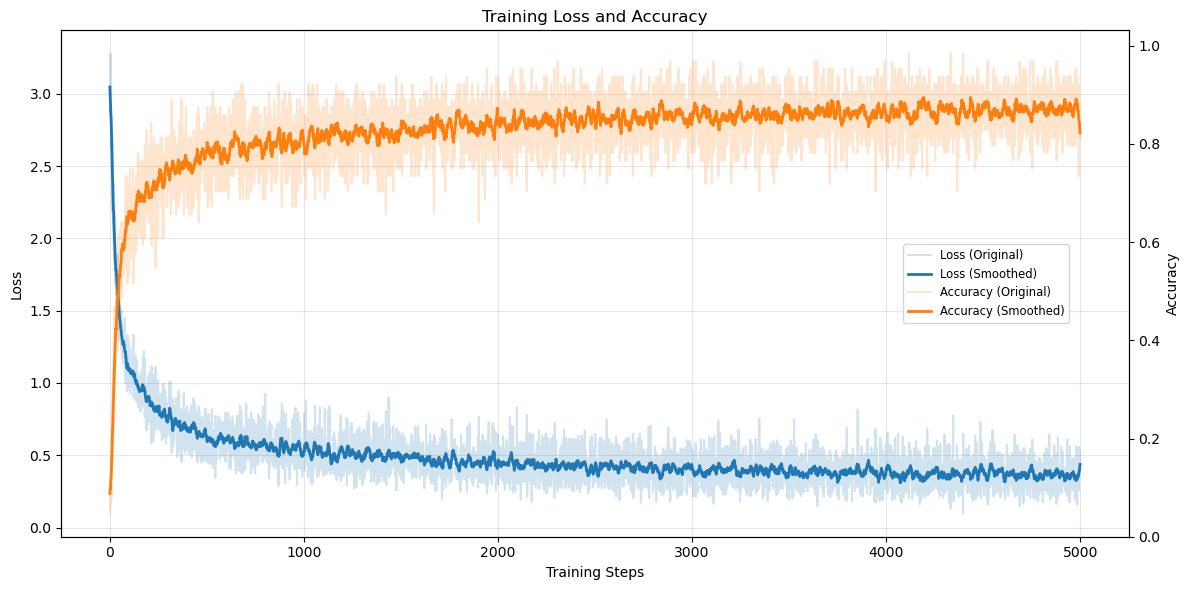


图8--5000steps损失&准确率

### 4.1.4 测试结果

|  |  |
| --- | --- |
| Model | Accuracy |
| model\_step1000.npz | 0.8016 |
| model\_step2000.npz | 0.8264 |
| model\_step3000.npz | 0.8462 |
| model\_step4000.npz | 0.8534 |
| model\_step5000.npz | 0.8612 |

表3--cnn-by-myself测试结果

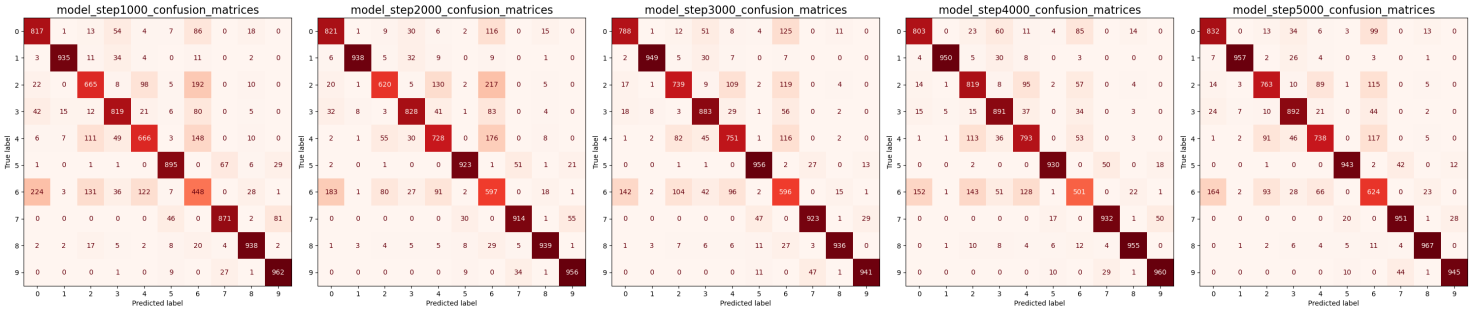


图9--cnn-by-myself混淆矩阵

## 4.2 cnn-by-pytorch

### 4.2.1 超参数

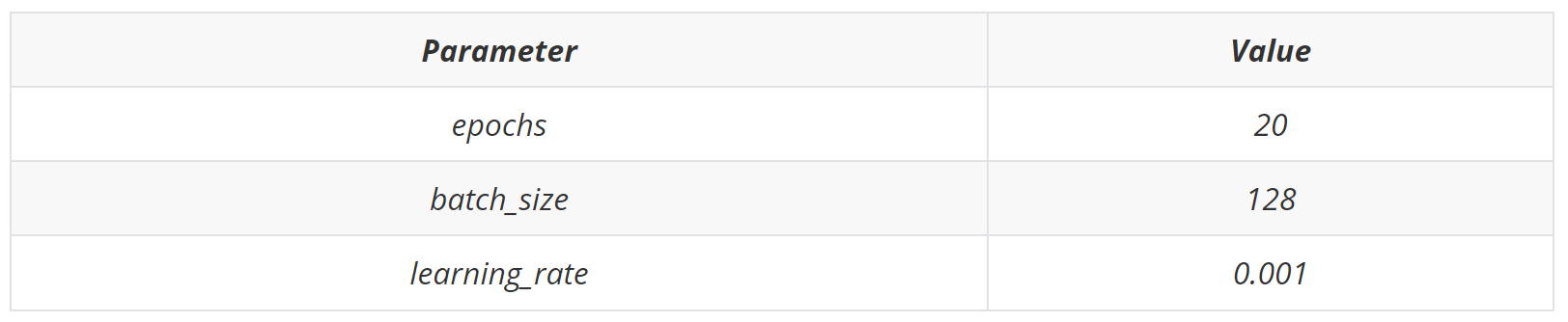


表4--cnn-by-pytorch超参数

### 4.2.2 训练过程

每个epoch保存一次模型，并保存训练日志(cnn-by-pytorch\logs目录下)，每个epoch绘制一个点，训练过程在Fashion-MNIST-Chiale\cnn-by-pytorch\demo.ipynb 目录下。

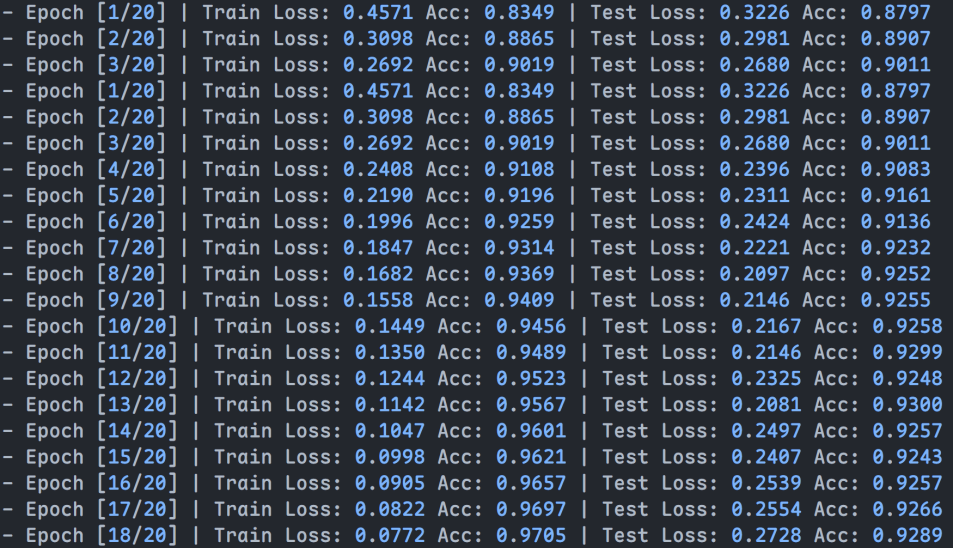


图10--cnn-by-pytorch训练过程

### 4.2.3 损失&准确率

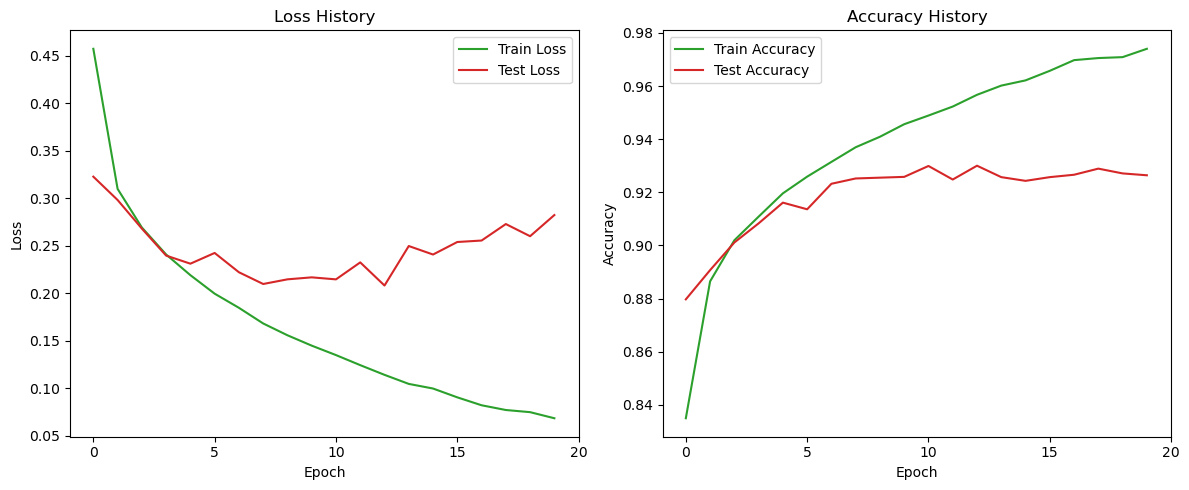
s

图11--损失&准确率

### 4.2.4 测试结果

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Epoch | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Accuracy | 0.8797 | 0.8907 | 0.9011 | 0.9083 | 0.9161 | 0.9136 | 0.9232 | 0.9252 | 0.9255 | 0.9258 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Epoch | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| Accuracy | 0.9299 | 0.9248 | 0.9300 | 0.9257 | 0.9243 | 0.9257 | 0.9266 | 0.9289 | 0.9271 | 0.9264 |

表5--cnn-by-pytorch测试结果

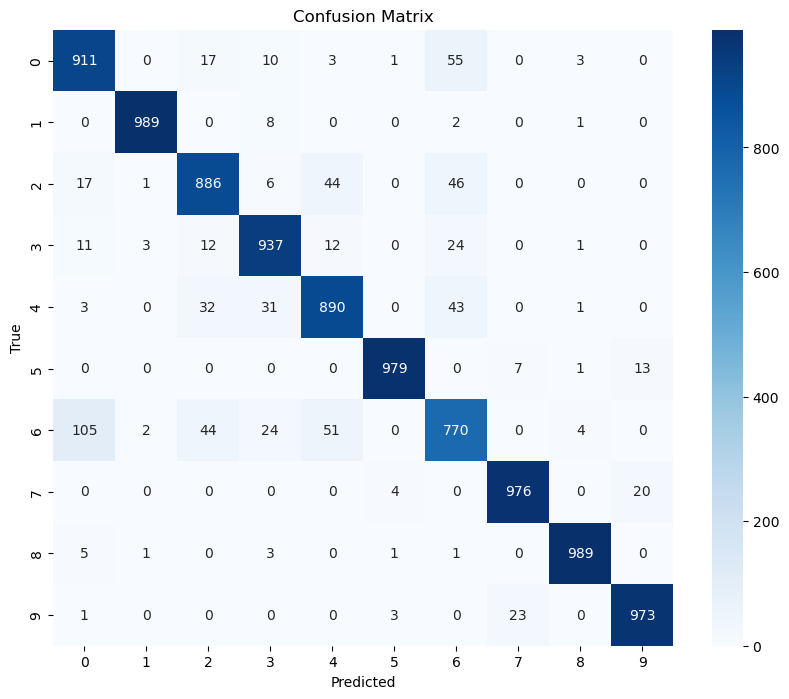


图12--cnn-by-pytorch混淆矩阵（best model）

## 4.3 **模型比较**

|  | cnn-by-myself | cnn-by-pytorch |
| --- | --- | --- |
| 实现方式 | 纯 Python + NumPy 实现 | 使用 PyTorch 框架实现 |
| 准确率 | 约 86.1% | 约 93.0% |
| 训练效率 | 较慢，需手动更新参数 | 较快，自动反向传播 |
| 灵活性 | 不易扩展，调试复杂 | 模块化，易于扩展与调试 |
| 学习价值 | 高，深入理解底层原理 | 高，掌握框架用法与实际应用 |

表6--模型比较

# 5 结论与改进

## 5.1 结论

1. 技术路线对比

手动实现模型（cnn-by-myself）：通过纯Python与NumPy实现仿LeNet网络，准确率达86.1%。虽然性能低于框架实现，但通过手动编写卷积、池化、反向传播等核心模块，深入理解了CNN的底层计算逻辑与参数更新机制。

PyTorch实现模型（cnn-by-pytorch）：基于PyTorch框架构建更复杂的网络结构（如更大卷积核通道、Dropout层），准确率提升至93.0%。得益于框架的自动微分与高效计算，开发效率显著提高，且模型扩展性更强。

1. 性能差异分析

PyTorch模型性能更优的关键因素包括：更深的网络设计（如双3×3卷积层）、Adam优化器的自适应学习率调整、Dropout正则化抑制过拟合，以及批量数据加载的并行化加速。

手动实现受限于计算效率与简化设计（如较小的参数量），导致模型表达能力不足，但通过动量SGD与批归一化仍实现了较高的基线性能。

## 5.2 改进方向

1. 模型架构优化

手动模型可引入残差连接（ResNet思想）或增加卷积层深度，以提升特征提取能力。

PyTorch模型可尝试引入注意力机制（如SE模块）或替换为更先进架构（如ResNet、MobileNet），进一步优化分类精度。

1. 训练策略改进

数据增强：从混淆矩阵可以看出，模型最大的错误在于将shirt分类为T-shirt，可能由于两者相似度较高，需要对训练图像添加旋转、平移、噪声等增强操作，提升模型泛化能力。

学习率调度：采用余弦退火或动态调整学习率策略，避免训练后期陷入局部最优。

正则化强化：在PyTorch模型中尝试L2权重衰减或标签平滑（Label Smoothing），降低过拟合风险。

1. 工程实践优化

为手动实现模型引入GPU加速（如CuPy库），缩短训练时间。

在PyTorch中集成TensorBoard可视化工具，实时监控训练过程并分析模型行为。

1. 扩展研究

探索多模型集成（如投票法或加权平均），结合两种实现方式的优势提升整体性能。

迁移学习应用：基于预训练模型（如VGG、ResNet）进行微调，验证其在Fashion-MNIST上的迁移效果。

# 附录-cnn-by-myself关键代码

def im2col(input\_data, filter\_h, filter\_w, stride=1, pad=0):

    N, C, H, W = input\_data.shape

    out\_h = (H + 2 \* pad - filter\_h) // stride + 1

    out\_w = (W + 2 \* pad - filter\_w) // stride + 1

    img = np.pad(input\_data, [(0, 0), (0, 0), (pad, pad), (pad, pad)], 'constant')

    col = np.zeros((N, C, filter\_h, filter\_w, out\_h, out\_w))

    for y in range(filter\_h):

        y\_max = y + stride \* out\_h

        for x in range(filter\_w):

            x\_max = x + stride \* out\_w

            col[:, :, y, x, :, :] = img[:, :, y:y\_max:stride, x:x\_max:stride]

    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N \* out\_h \* out\_w, -1)

    return col

def col2im(col, input\_shape, filter\_h, filter\_w, stride=1, pad=0):

    N, C, H, W = input\_shape

    out\_h = (H + 2 \* pad - filter\_h) // stride + 1

    out\_w = (W + 2 \* pad - filter\_w) // stride + 1

    col = col.reshape(N, out\_h, out\_w, C, filter\_h, filter\_w).transpose(0, 3, 4, 5, 1, 2)

    img = np.zeros((N, C, H + 2 \* pad + stride - 1, W + 2 \* pad + stride - 1))

    for y in range(filter\_h):

        y\_max = y + stride \* out\_h

        for x in range(filter\_w):

            x\_max = x + stride \* out\_w

            img[:, :, y:y\_max:stride, x:x\_max:stride] += col[:, :, y, x, :, :]

    return img[:, :, pad:H + pad, pad:W + pad]

class Conv2D:

    def \_\_init\_\_(self, in\_channels, out\_channels, kernel\_size, stride=1, padding=0):

        self.in\_channels = in\_channels

        self.out\_channels = out\_channels

        self.kernel\_size = kernel\_size

        self.stride = stride

        self.padding = padding

*# He 初始化*

        self.weights = np.random.randn(out\_channels, in\_channels, kernel\_size, kernel\_size) \* np.sqrt(2 / (in\_channels \* kernel\_size \* kernel\_size))

        self.bias = np.zeros((out\_channels, 1))

        self.dweights = np.zeros\_like(self.weights)

        self.dbias = np.zeros\_like(self.bias)

*# 动量项*

        self.v\_weights = np.zeros\_like(self.weights)

        self.v\_bias = np.zeros\_like(self.bias)

    def forward(self, X):

        self.X = X

        N, C, H, W = X.shape

        F, \_, HH, WW = self.weights.shape

        H\_out = 1 + (H + 2 \* self.padding - HH) // self.stride

        W\_out = 1 + (W + 2 \* self.padding - WW) // self.stride

        col = im2col(X, HH, WW, self.stride, self.padding)

        col\_W = self.weights.reshape(F, -1).T

        out = np.dot(col, col\_W) + self.bias.T

        out = out.reshape(N, H\_out, W\_out, -1).transpose(0, 3, 1, 2)

        return out

    def backward(self, dout):

        N, F, H\_out, W\_out = dout.shape

        \_, C, HH, WW = self.weights.shape

        H = (H\_out - 1) \* self.stride + HH - 2 \* self.padding

        W = (W\_out - 1) \* self.stride + WW - 2 \* self.padding

        dout = dout.transpose(0, 2, 3, 1).reshape(-1, F)

        col = im2col(self.X, HH, WW, self.stride, self.padding)

        self.dweights = np.dot(col.T, dout).transpose(1, 0).reshape(F, C, HH, WW)

        self.dbias = np.sum(dout, axis=0, keepdims=True).T

        dcol = np.dot(dout, self.weights.reshape(F, -1))

        dX = col2im(dcol, self.X.shape, HH, WW, self.stride, self.padding)

        return dX

    def update\_params(self, learning\_rate, momentum=0.9):

        self.v\_weights = momentum \* self.v\_weights - learning\_rate \* self.dweights

        self.weights += self.v\_weights

        self.v\_bias = momentum \* self.v\_bias - learning\_rate \* self.dbias

        self.bias += self.v\_bias

class BatchNormalization:

    def \_\_init\_\_(self, channels):

        self.gamma = np.ones((1, channels, 1, 1))

        self.beta = np.zeros((1, channels, 1, 1))

        self.dgamma = np.zeros\_like(self.gamma)

        self.dbeta = np.zeros\_like(self.beta)

        self.moving\_mean = np.zeros((1, channels, 1, 1))

        self.moving\_var = np.ones((1, channels, 1, 1))

        self.eps = 1e-5

        self.momentum = 0.9

    def forward(self, x, train\_flg=True):

        self.train\_flg = train\_flg

        N, C, H, W = x.shape

        if train\_flg:

            mu = x.mean(axis=(0, 2, 3), keepdims=True)

            xc = x - mu

            var = np.mean(xc \*\* 2, axis=(0, 2, 3), keepdims=True)

            std = np.sqrt(var + self.eps)

            xn = xc / std

            self.xc = xc

            self.xn = xn

            self.std = std

            self.moving\_mean = self.momentum \* self.moving\_mean + (1 - self.momentum) \* mu

            self.moving\_var = self.momentum \* self.moving\_var + (1 - self.momentum) \* var

        else:

            xc = x - self.moving\_mean

            xn = xc / np.sqrt(self.moving\_var + self.eps)

        out = self.gamma \* xn + self.beta

        return out

    def backward(self, dout):

        N, C, H, W = dout.shape

        dbeta = dout.sum(axis=(0, 2, 3), keepdims=True)

        dgamma = np.sum(self.xn \* dout, axis=(0, 2, 3), keepdims=True)

        dxn = self.gamma \* dout

        dxc = dxn / self.std

        dstd = -np.sum((dxn \* self.xc) / (self.std \*\* 2), axis=(0, 2, 3), keepdims=True)

        dvar = 0.5 \* dstd / self.std

        dxc += (2.0 / (N \* H \* W)) \* self.xc \* dvar

        dmu = np.sum(dxc, axis=(0, 2, 3), keepdims=True)

        dx = dxc - dmu / (N \* H \* W)

        self.dgamma = dgamma

        self.dbeta = dbeta

        return dx

    def update\_params(self, learning\_rate):

        self.gamma -= learning\_rate \* self.dgamma

        self.beta -= learning\_rate \* self.dbeta

class ReLU:

    def \_\_init\_\_(self):

        self.mask = None

    def forward(self, x):

        self.mask = (x <= 0)

        out = x.copy()

        out[self.mask] = 0

        return out

    def backward(self, dout):

        dout[self.mask] = 0

        dx = dout

        return dx

class MaxPool2D:

    def \_\_init\_\_(self, pool\_size=2, stride=2):

        self.pool\_size = pool\_size

        self.stride = stride

    def forward(self, X):

        self.X = X

        N, C, H, W = X.shape

        H\_out = 1 + (H - self.pool\_size) // self.stride

        W\_out = 1 + (W - self.pool\_size) // self.stride

        out = np.zeros((N, C, H\_out, W\_out))

        self.arg\_max = np.zeros((N, C, H\_out, W\_out), dtype=np.int64)

        for i in range(N):

            for c in range(C):

                for h in range(H\_out):

                    for w in range(W\_out):

                        h\_start = h \* self.stride

                        h\_end = h\_start + self.pool\_size

                        w\_start = w \* self.stride

                        w\_end = w\_start + self.pool\_size

                        out[i, c, h, w] = np.max(X[i, c, h\_start:h\_end, w\_start:w\_end])

                        self.arg\_max[i, c, h, w] = np.argmax(X[i, c, h\_start:h\_end, w\_start:w\_end])

        return out

    def backward(self, dout):

        N, C, H\_out, W\_out = dout.shape

        \_, \_, H, W = self.X.shape

        dX = np.zeros\_like(self.X)

        for i in range(N):

            for c in range(C):

                for h in range(H\_out):

                    for w in range(W\_out):

                        h\_start = h \* self.stride

                        h\_end = h\_start + self.pool\_size

                        w\_start = w \* self.stride

                        w\_end = w\_start + self.pool\_size

                        idx = self.arg\_max[i, c, h, w]

                        dX[i, c, h\_start + idx // self.pool\_size, w\_start + idx % self.pool\_size] = dout[i, c, h, w]

        return dX

class FullyConnected:

    def \_\_init\_\_(self, input\_size, output\_size):

*# He 初始化*

        self.weights = np.random.randn(input\_size, output\_size) \* np.sqrt(2 / input\_size)

        self.bias = np.zeros((1, output\_size))

        self.dweights = np.zeros\_like(self.weights)

        self.dbias = np.zeros\_like(self.bias)

*# 动量项*

        self.v\_weights = np.zeros\_like(self.weights)

        self.v\_bias = np.zeros\_like(self.bias)

    def forward(self, X):

        self.X = X

        return np.dot(X, self.weights) + self.bias

    def backward(self, dout):

        dX = np.dot(dout, self.weights.T)

        self.dweights = np.dot(self.X.T, dout)

        self.dbias = np.sum(dout, axis=0, keepdims=True)

        return dX

    def update\_params(self, learning\_rate, momentum=0.9):

        self.v\_weights = momentum \* self.v\_weights - learning\_rate \* self.dweights

        self.weights += self.v\_weights

        self.v\_bias = momentum \* self.v\_bias - learning\_rate \* self.dbias

        self.bias += self.v\_bias

class Softmax:

    def forward(self, X):

        exp\_X = np.exp(X - np.max(X, axis=1, keepdims=True))

        return exp\_X / np.sum(exp\_X, axis=1, keepdims=True)

    def backward(self, y\_pred, y\_true):

        N = y\_pred.shape[0]

        return (y\_pred - y\_true) / N

class CNN:

    def \_\_init\_\_(self):

        self.conv1 = Conv2D(in\_channels=1, out\_channels=6, kernel\_size=5, padding=2)

        self.bn1 = BatchNormalization(6)

        self.relu1 = ReLU()

        self.pool1 = MaxPool2D(pool\_size=2, stride=2)

        self.conv2 = Conv2D(in\_channels=6, out\_channels=16, kernel\_size=5, padding=0)

        self.bn2 = BatchNormalization(16)

        self.relu2 = ReLU()

        self.pool2 = MaxPool2D(pool\_size=2, stride=2)

        self.fc1 = FullyConnected(input\_size=5 \* 5 \* 16, output\_size=120)

        self.relu3 = ReLU()

        self.fc2 = FullyConnected(input\_size=120, output\_size=84)

        self.relu4 = ReLU()

        self.fc3 = FullyConnected(input\_size=84, output\_size=10)

        self.softmax = Softmax()

    def forward(self, X):

        X = X.reshape(-1, 1, 28, 28)

        out = self.conv1.forward(X)

        out = self.bn1.forward(out)

        out = self.relu1.forward(out)

        out = self.pool1.forward(out)

        out = self.conv2.forward(out)

        out = self.bn2.forward(out)

        out = self.relu2.forward(out)

        out = self.pool2.forward(out)

        out = out.reshape(out.shape[0], -1)

        out = self.fc1.forward(out)

        out = self.relu3.forward(out)

        out = self.fc2.forward(out)

        out = self.relu4.forward(out)

        out = self.fc3.forward(out)

        out = self.softmax.forward(out)

        return out

    def backward(self, y\_pred, y\_true):

        dout = self.softmax.backward(y\_pred, y\_true)

        dout = self.fc3.backward(dout)

        dout = self.relu4.backward(dout)

        dout = self.fc2.backward(dout)

        dout = self.relu3.backward(dout)

        dout = self.fc1.backward(dout)

        dout = dout.reshape(-1, 16, 5, 5)

        dout = self.pool2.backward(dout)

        dout = self.relu2.backward(dout)

        dout = self.bn2.backward(dout)

        dout = self.conv2.backward(dout)

        dout = self.pool1.backward(dout)

        dout = self.relu1.backward(dout)

        dout = self.bn1.backward(dout)

        dout = self.conv1.backward(dout)

        return dout

    def update\_params(self, learning\_rate, momentum=0.9):

        self.conv1.update\_params(learning\_rate, momentum)

        self.bn1.update\_params(learning\_rate)

        self.conv2.update\_params(learning\_rate, momentum)

        self.bn2.update\_params(learning\_rate)

        self.fc1.update\_params(learning\_rate, momentum)

        self.fc2.update\_params(learning\_rate, momentum)

        self.fc3.update\_params(learning\_rate, momentum)

    def save\_model(self, filename):

        model\_params = {

            'conv1\_weights': self.conv1.weights,

            'conv1\_bias': self.conv1.bias,

            'bn1\_gamma': self.bn1.gamma,

            'bn1\_beta': self.bn1.beta,

            'conv2\_weights': self.conv2.weights,

            'conv2\_bias': self.conv2.bias,

            'bn2\_gamma': self.bn2.gamma,

            'bn2\_beta': self.bn2.beta,

            'fc1\_weights': self.fc1.weights,

            'fc1\_bias': self.fc1.bias,

            'fc2\_weights': self.fc2.weights,

            'fc2\_bias': self.fc2.bias,

            'fc3\_weights': self.fc3.weights,

            'fc3\_bias': self.fc3.bias,

        }

        np.savez(filename, \*\*model\_params)

    def load\_model(self, filename):

        model\_params = np.load(filename)

        self.conv1.weights = model\_params['conv1\_weights']

        self.conv1.bias = model\_params['conv1\_bias']

        self.bn1.gamma = model\_params['bn1\_gamma']

        self.bn1.beta = model\_params['bn1\_beta']

        self.conv2.weights = model\_params['conv2\_weights']

        self.conv2.bias = model\_params['conv2\_bias']

        self.bn2.gamma = model\_params['bn2\_gamma']

        self.bn2.beta = model\_params['bn2\_beta']

        self.fc1.weights = model\_params['fc1\_weights']

        self.fc1.bias = model\_params['fc1\_bias']

        self.fc2.weights = model\_params['fc2\_weights']

        self.fc2.bias = model\_params['fc2\_bias']

        self.fc3.weights = model\_params['fc3\_weights']

        self.fc3.bias = model\_params['fc3\_bias']

    def print\_model(self):

        layers = [

            (self.conv1, 'Conv2D', 'conv1'),

            (self.bn1, 'BatchNorm', 'bn1'),

            (self.relu1, 'ReLU', 'relu1'),

            (self.pool1, 'MaxPool2D', 'pool1'),

            (self.conv2, 'Conv2D', 'conv2'),

            (self.bn2, 'BatchNorm', 'bn2'),

            (self.relu2, 'ReLU', 'relu2'),

            (self.pool2, 'MaxPool2D', 'pool2'),

            (self.fc1, 'FullyConnected', 'fc1'),

            (self.relu3, 'ReLU', 'relu3'),

            (self.fc2, 'FullyConnected', 'fc2'),

            (self.relu4, 'ReLU', 'relu4'),

            (self.fc3, 'FullyConnected', 'fc3'),

        ]

        current\_shape = (1, 28, 28)  *# 输入形状: (channels, height, width)*

        total\_params = 0

        print("Layer (type)         Output Shape         Param #")

        print("===================================================")

        for layer\_info in layers:

            layer\_obj, layer\_type, layer\_name = layer\_info

            params = 0

            output\_shape = current\_shape

            if layer\_type == 'Conv2D':

                in\_channels, H\_in, W\_in = current\_shape

                padding = layer\_obj.padding

                kernel\_size = layer\_obj.kernel\_size

                stride = layer\_obj.stride

                out\_channels = layer\_obj.out\_channels

                H\_out = (H\_in + 2 \* padding - kernel\_size) // stride + 1

                W\_out = (W\_in + 2 \* padding - kernel\_size) // stride + 1

                output\_shape = (out\_channels, H\_out, W\_out)

                params = (in\_channels \* kernel\_size\*\*2) \* out\_channels + out\_channels

                layer\_desc = f"Conv2D"

            elif layer\_type == 'BatchNorm':

                channels = current\_shape[0]

                params = 2 \* channels  *# gamma和beta*

                layer\_desc = f"BatchNorm"

                output\_shape = current\_shape

            elif layer\_type == 'ReLU':

                layer\_desc = "ReLU"

                params = 0

                output\_shape = current\_shape

            elif layer\_type == 'MaxPool2D':

                pool\_size = layer\_obj.pool\_size

                stride = layer\_obj.stride

                channels, H\_in, W\_in = current\_shape

                H\_out = (H\_in - pool\_size) // stride + 1

                W\_out = (W\_in - pool\_size) // stride + 1

                output\_shape = (channels, H\_out, W\_out)

                layer\_desc = f"MaxPool2D"

                params = 0

            elif layer\_type == 'FullyConnected':

                if len(current\_shape) == 3:

                    input\_dim = current\_shape[0] \* current\_shape[1] \* current\_shape[2]

                else:

                    input\_dim = current\_shape[0]

                output\_dim = layer\_obj.weights.shape[1]

                params = input\_dim \* output\_dim + output\_dim

                output\_shape = (output\_dim,)

                layer\_desc = f"FullyConnected"

*# 格式化输出*

            print(f"{layer\_desc.ljust(20)} {str(output\_shape).ljust(20)} {params}")

            total\_params += params

            current\_shape = output\_shape

        print("===================================================")

        print(f"Total params: {total\_params}")