華中科技大學

《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

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。

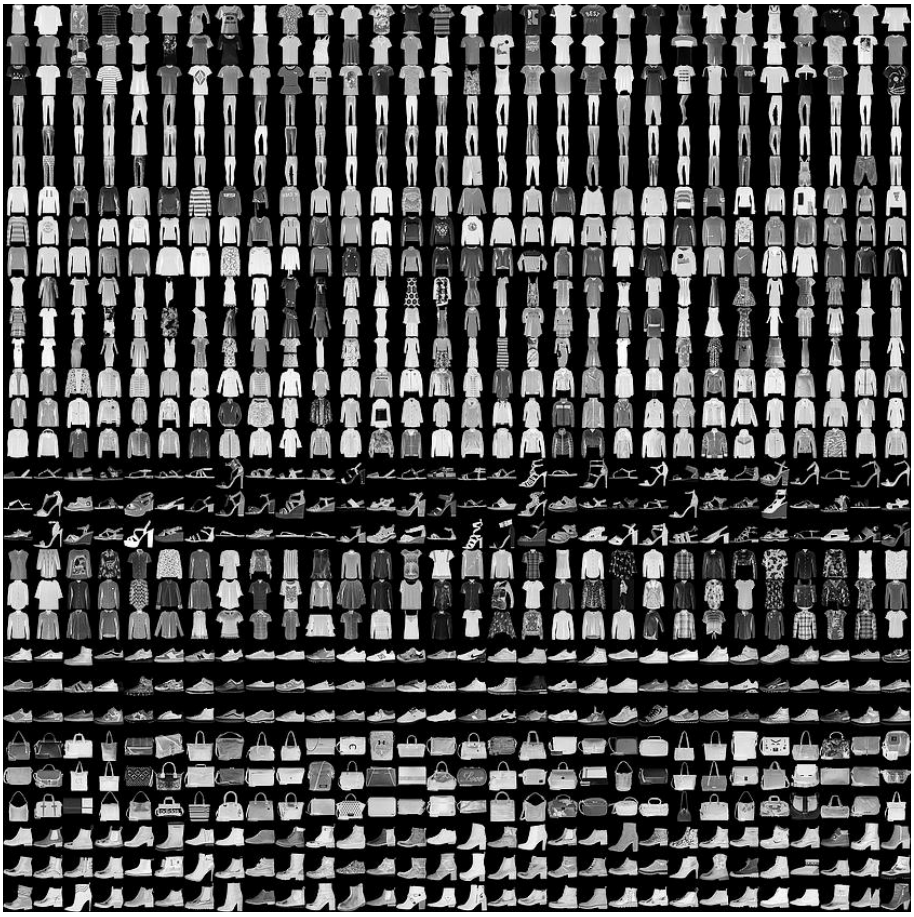


图 1--Fashion-MNIST 数据集

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

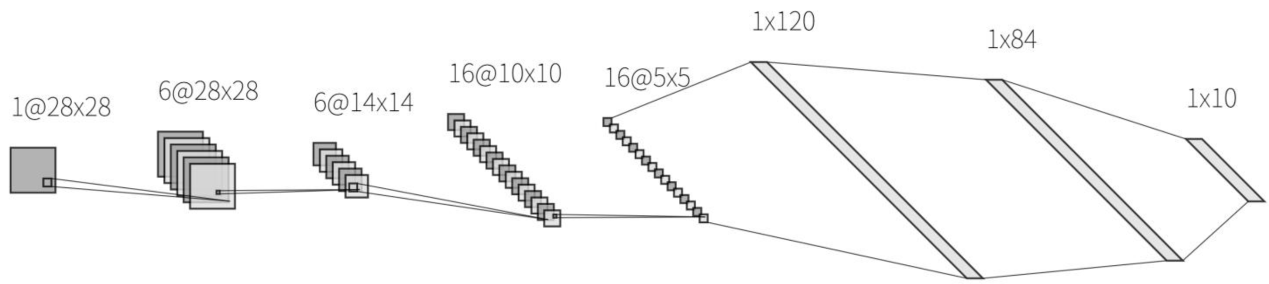


图 3--cnn-by-myself 网络结构（1）

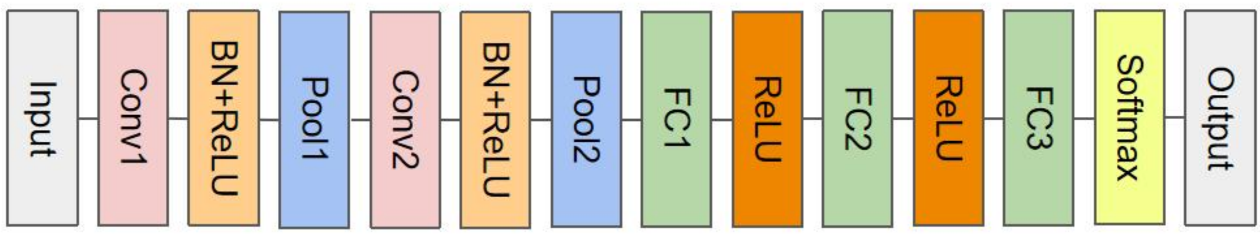


图 4--cnn-by-myself 网络结构（2）

(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：$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{10} e^{z_k}}$

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：$W\sim N(0,\sqrt{2/n_{in}})$
   - 偏置项初始化为 0
2. 参数更新方法：
   - 使用带动量的随机梯度下降（SGD）：$v = momentum * v - lr * grad$
   - 动量系数：0.9
   - 学习率通过实验确定
3. 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
4. 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
5. 损失函数：
   - 交叉熵损失：$L = -\sum_{i=1}^{10} y_i \log(p_i)$

3.2 cnn-by-pytorch

3.2.1 网络结构

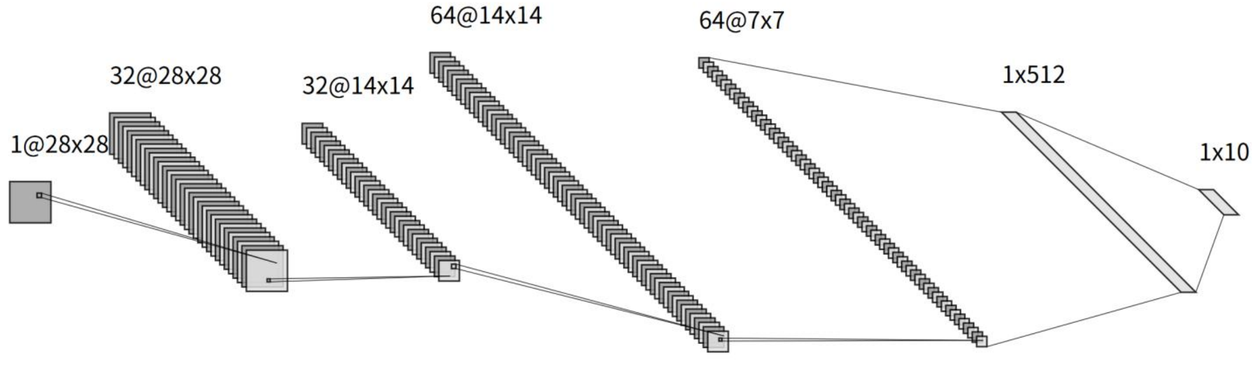


图 5--cnn-by-pytorch 网络结构

(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：$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{10} e^{z_k}}$

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：$W \sim N(0, \sqrt{2/n_{in}})$
   - 全连接层采用 Xavier 初始化：$W \sim U(-\sqrt{6/(n_{in}+n_{out})}, \sqrt{6/(n_{in}+n_{out})})$
   - 偏置项初始化为 0
2. 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
3. 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
4. 损失函数：
   - 交叉熵损失：$L = -\sum_{i=1}^{10} y_i \log(p_i)$

4 实验结果与分析

4.2 cnn-by-myself

4.1.1 超参数

Parameter	Value
max_steps	5000
batch_size	64
learning_rate	0.0005

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。

```
2025-04-20 13:33:44.029, Step: 1/5000, Loss: 3.2231, Accuracy: 0.0469
2025-04-20 13:33:44.695, Step: 2/5000, Loss: 3.1009, Accuracy: 0.0781
2025-04-20 13:33:45.305, Step: 3/5000, Loss: 2.8735, Accuracy: 0.1250
2025-04-20 13:33:45.952, Step: 4/5000, Loss: 3.2798, Accuracy: 0.0625
2025-04-20 13:33:46.568, Step: 5/5000, Loss: 3.2397, Accuracy: 0.0625
2025-04-20 13:33:47.190, Step: 6/5000, Loss: 2.9717, Accuracy: 0.0938
2025-04-20 13:33:47.821, Step: 7/5000, Loss: 2.7894, Accuracy: 0.1250
2025-04-20 13:33:48.459, Step: 8/5000, Loss: 2.8940, Accuracy: 0.1094
2025-04-20 13:33:49.097, Step: 9/5000, Loss: 2.6142, Accuracy: 0.1719
2025-04-20 13:33:49.701, Step: 10/5000, Loss: 2.6209, Accuracy: 0.1250
2025-04-20 13:33:50.293, Step: 11/5000, Loss: 2.6717, Accuracy: 0.0781
2025-04-20 13:33:50.893, Step: 12/5000, Loss: 2.1869, Accuracy: 0.2812
2025-04-20 13:33:51.507, Step: 13/5000, Loss: 2.6852, Accuracy: 0.1562
2025-04-20 13:33:52.104, Step: 14/5000, Loss: 2.3925, Accuracy: 0.1875
2025-04-20 13:33:52.689, Step: 15/5000, Loss: 2.3279, Accuracy: 0.2656
2025-04-20 13:33:53.285, Step: 16/5000, Loss: 2.3660, Accuracy: 0.1875
2025-04-20 13:33:53.946, Step: 17/5000, Loss: 2.1608, Accuracy: 0.3125
2025-04-20 13:33:54.642, Step: 18/5000, Loss: 2.1059, Accuracy: 0.2031
```

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

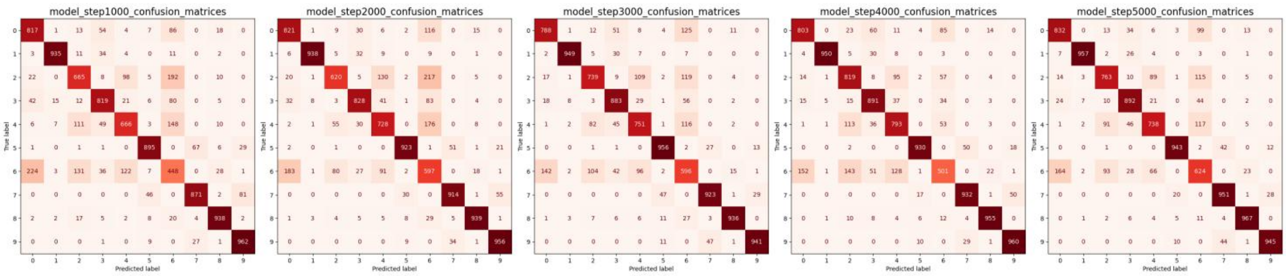


图 9--cnn-by-myself 混淆矩阵

4.2 cnn-by-pytorch

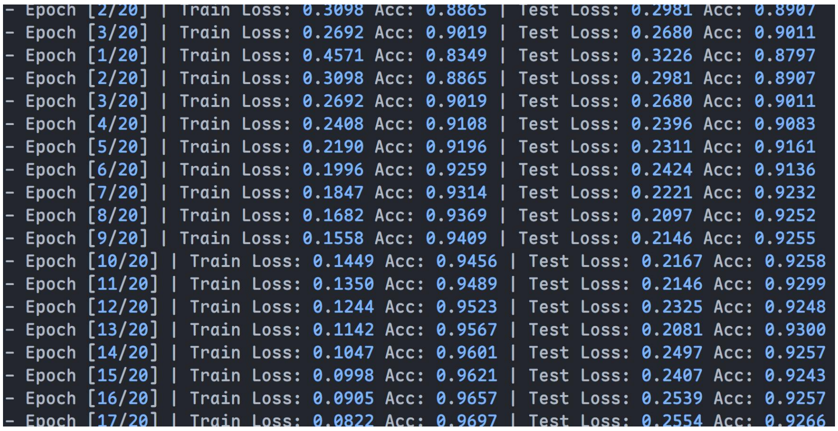
4.2.1 超参数

Parameter	*Value*
epochs	*20*
batch_size	*128*
learning_rate	*0.001*

表 4--cnn-by-pytorch 超参数

4.2.2 训练过程

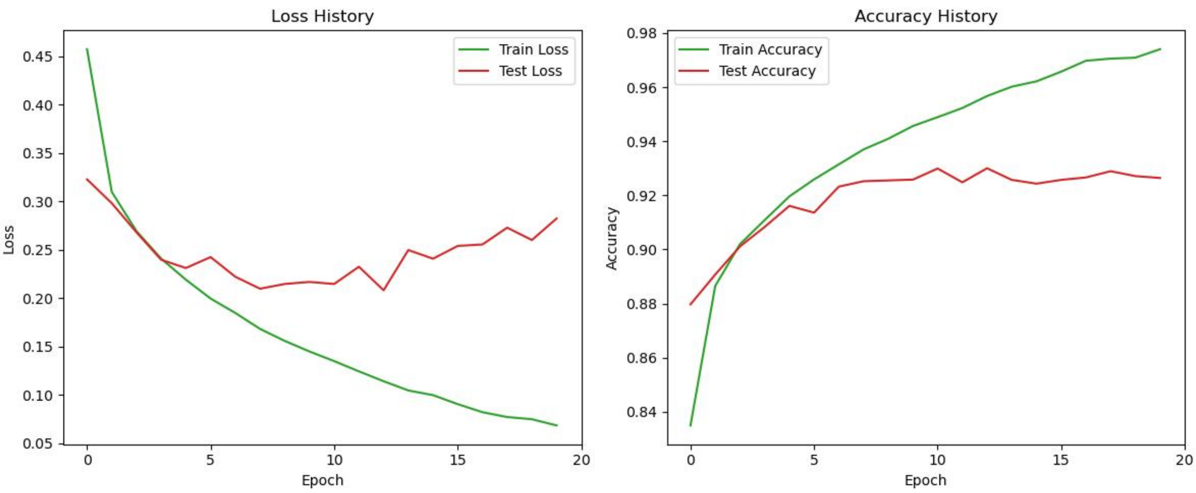
每个 epoch 保存一次模型，并保存训练日志(cnn-by-pytorch\logs 目录下)，每个 epoch 绘制一个点，训练过程在 Fashion-MNIST-Chiale\cnn-by-pytorch\demo.ipynb 目录下。



```
- Epoch [2/20] | Train Loss: 0.3098 Acc: 0.8865 | Test Loss: 0.2981 Acc: 0.8907
- Epoch [3/20] | Train Loss: 0.2692 Acc: 0.9019 | Test Loss: 0.2680 Acc: 0.9011
- Epoch [1/20] | Train Loss: 0.4571 Acc: 0.8349 | Test Loss: 0.3226 Acc: 0.8797
- Epoch [2/20] | Train Loss: 0.3098 Acc: 0.8865 | Test Loss: 0.2981 Acc: 0.8907
- Epoch [3/20] | Train Loss: 0.2692 Acc: 0.9019 | Test Loss: 0.2680 Acc: 0.9011
- Epoch [4/20] | Train Loss: 0.2408 Acc: 0.9108 | Test Loss: 0.2396 Acc: 0.9083
- Epoch [5/20] | Train Loss: 0.2190 Acc: 0.9196 | Test Loss: 0.2311 Acc: 0.9161
- Epoch [6/20] | Train Loss: 0.1996 Acc: 0.9259 | Test Loss: 0.2424 Acc: 0.9136
- Epoch [7/20] | Train Loss: 0.1847 Acc: 0.9314 | Test Loss: 0.2221 Acc: 0.9232
- Epoch [8/20] | Train Loss: 0.1682 Acc: 0.9369 | Test Loss: 0.2097 Acc: 0.9252
- Epoch [9/20] | Train Loss: 0.1558 Acc: 0.9409 | Test Loss: 0.2146 Acc: 0.9255
- Epoch [10/20] | Train Loss: 0.1449 Acc: 0.9456 | Test Loss: 0.2167 Acc: 0.9258
- Epoch [11/20] | Train Loss: 0.1350 Acc: 0.9489 | Test Loss: 0.2146 Acc: 0.9299
- Epoch [12/20] | Train Loss: 0.1244 Acc: 0.9523 | Test Loss: 0.2325 Acc: 0.9248
- Epoch [13/20] | Train Loss: 0.1142 Acc: 0.9567 | Test Loss: 0.2081 Acc: 0.9300
- Epoch [14/20] | Train Loss: 0.1047 Acc: 0.9601 | Test Loss: 0.2497 Acc: 0.9257
- Epoch [15/20] | Train Loss: 0.0998 Acc: 0.9621 | Test Loss: 0.2407 Acc: 0.9243
- Epoch [16/20] | Train Loss: 0.0905 Acc: 0.9657 | Test Loss: 0.2539 Acc: 0.9257
- Epoch [17/20] | Train Loss: 0.0822 Acc: 0.9697 | Test Loss: 0.2554 Acc: 0.9266
```

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

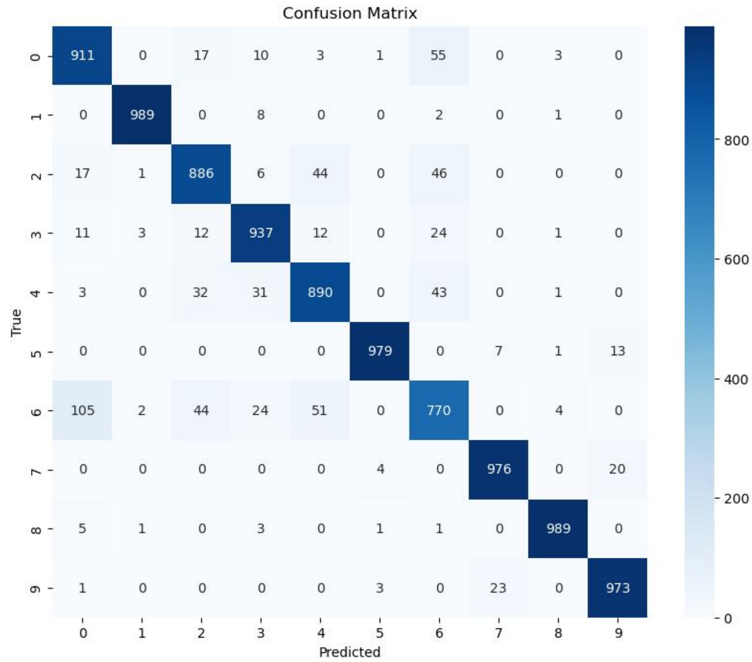


图 12--cnn-by-pytorch 混淆矩阵（best model）

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。

2. 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。

2. 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。

3. 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。

4. 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_s
ize) * 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) * v
ar
        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_s
tart: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_siz
e)
        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}")
```