浙江大学

实 验 报 告

姓名与学号 曾成宽 3240105591

年级与专业 2024 农业工程

所在学院 求是学院

提交日期 2024 年 5 月 13 日

目录

§1 实验目的与背景介绍

1.1 实验目的

本实验的实验目的是掌握 MLP 和 CNN 的基本原理和实现方法，了解不同网络结构对模型性能的影响，学习深度学习模型训练、评估和可视化的方法，同时通过对比实验，理解不同模型在图像分类任务中的优缺点，培养深度学习模型调优和问题解决的能力。

1.2 实验背景

深度学习模型在计算机视觉、自然语言处理等领域取得了显著成果。其中，基础的全连接网络（MLP）和具有局部感知特性的卷积网络（CNN）是两类代表性架构。”本实验旨在通过对多层感知机（MLP）和卷积神经网络（CNN）的实现、 训练和评估，帮助学生深入理解两种模型的结构特点、性能差异以及适用场景。学生将从基础模型开始，逐步探索更复杂的网络架构，最终通过对比分析，掌握深度学习模型设计与评估的关键技能。

§2 实验原理与实验过程

2.1 实验原理

2.1.1 基础 MLP 模型

2.1.1.1 多层感知机(MLP)

多层感知机是一种前馈神经网络，由输入层、一个或多个隐藏层和输出层组成。MLP 的主要特点是：

1. 每层神经元与下一层全连接
2. 使用非线性激活函数（如 ReLU、Sigmoid 等）
3. 通过反向传播算法进行训练

2.1.1.2 卷积神经网络(CNN)

卷积神经网络是为处理具有网格状拓扑结构的数据而设计的神经网络，主要包含卷积层、池化层和全连接层。CNN 的主要特点是：

1. 局部连接：每个神经元只与输入数据的一个局部区域连接。
2. 权重共享：同一特征图的所有神经元共享相同的权重。
3. 多层次特征提取：低层检测边缘等简单特征，高层组合这些特征形成更复杂的表示。

2.2 高级 CNN 架构

2.2.1 VGG 架构

VGG 网络（由 Visual Geometry Group 开发）是一种非常简洁而有效的 CNN 架构，在 2014 年 ImageNet 挑战赛中取得了优异成绩。其主要特点包括：

1. 简单统一的设计：使用小尺寸（3×3）卷积核和 2×2 最大池化层

2. 深度堆叠：通过堆叠多个相同配置的卷积层增加网络深度

3. 结构规整：遵循"卷积层组-池化层"的模式，随着网络深入，特征图尺寸减小而通道数增加

2.2.2 ResNet 架构

ResNet（残差网络）由微软研究院的 He 等人在 2015 年提出，是解决"深度退化问题"的突破性架构。其核心创新是引入了残差连接（skip connection）：

1. 残差连接：通过快捷连接（shortcut connection）将输入直接加到输出上，形成恒等映射路径。

2. 残差学习：网络不再直接学习输入到输出的映射 F(x)，而是学习残差 F(x)-x。

3. 深度扩展：残差连接有效缓解了梯度消失问题，使得训练非常深的网络成为可能。

2.2.3 Bottleneck 结构

在更深的 ResNet 变体中，常使用"瓶颈"（Bottleneck）结构来降低计算复杂度：

- 使用 1×1 卷积降低通道数（降维）
- 使用 3×3 卷积进行特征提取
- 再使用 1×1 卷积恢复通道数（升维）

这种设计大幅减少参数量和计算量，同时保持或提高性能。

2.2.4 理解高级 CNN 设计理念

随着深度学习的发展，CNN 架构设计也变得更加精细和高效。以下是一些重要的设计理念：

1. 网络深度与宽度平衡：更深的网络能学习更抽象的特征，但也更难训练；更宽的网络（更多通道）能捕获更多特征，但参数量增加。

2. 跳跃连接：除了 ResNet 的残差连接，还有 DenseNet 的密集连接、UNet 的跨层连接等。

3. 特征增强：注意力机制（如 SENet 的通道注意力）、特征融合等。

4. 高效卷积设计：深度可分离卷积（MobileNet）、组卷积（ShuffleNet）等。

2.2 实验过程

2.2.1 了解与评估基础 MLP 模型

首先我们查看 models/mlp.py 文件，理解三种 MLP 模型的结构： SimpleMLP ，MLP，DeepMLP。然后通过代码实现一个具有两个隐藏层的 MLP 模型。第一隐藏层有 128 个神经元，第二隐藏层有 64 个神经元，输出层对应 10 个类别。 使用 ReLU 激活函数，并添加 BatchNorm 和 Dropout(0.3)。

随后我们通过给定代码来训练和评估 MLP 模型。首先在 train.ipynb 中训练 SimpleMLP 模型，确保将 model_type 设置为'simple_mlp'。然后观察训练过程中的损失和准确率变化，以及最终在测试集上的性能。最后修改参数尝试训练 DeepMLP 模型。

2.2.2 了解与评估基础 CNN 模型

首先我们查看 models/cnn.py 文件，理解四种 CNN 模型的结构： SimpleCNN ，MediumCNN ，VGGStyleNet，SimpleResNet。然后修改一个 simpleCNN 代码，添加一个额外的卷积层和 BatchNorm。新的卷积层在第二个池化层之后，卷积核数量为 64，卷积核大小为 3x3。

随后我们通过给定代码来训练和评估 MLP 模型。首先在 train.ipynb 中训练 SimpleMLP 模型，确保将 model_type 设置为'simple_cnn'。并将 use_data_augmentation 设置为 True，观察训练过程和卷积核可视化结果。最后继续训练 MediumCNN 模型，将 model_type 设置为'medium_cnn'。

2.2.3 了解 VGG 架构

在我们的实现中，VGGStyleNet 采用了简化版的 VGG 设计理念，包含三个卷积块，每个块包含两个卷积层和一个池化层。

我们首先在 train.ipynb 中训练 SimpleMLP 模型，确保将 model_type 设置为'vgg_style'，并将 use_data_augmentation 设置为 True。 然后观察网络的训练过程和性能。特别注意其收敛速度和最终准确率。

2.2.4 了解 ResNet 架构及残差连接

在我们的实现中，SimpleResNet 使用了基本的残差块，每个残差块包含两个 3×3 的卷积层和一个跳跃连接。

首先我们在 train.ipynb 中训练 SimpleMLP 模型，确保将 model_type 设置为'resnet'，并将 use_data_augmentation 设置为 True。然后观察网络的训练过程和性能，特别是深度对训练稳定性的影响。

2.2.5 模型复杂度分析

不同 CNN 架构在性能和效率之间存在权衡。现在我们将通过分析不同模型的参数量和推理时间来理解这种权衡。

首先我们运行一段代码来分析各个模型的复杂度，然后记录并比较各个模型的参数量和推理时间。

2.2.6 模型比较与分析

运行 compare.py 来对比不同模型的性能，并根据比较结果，综合分析不同类型模型（MLP 和 CNN）以及不同复杂度模型的性能差异。考虑以下几点：

1. 测试准确率
2. 参数量
3. 推理时间
4. 训练收敛速度
5. 过拟合/欠拟合情况

§3 代码实现

3.1 MLP 模型构建

```
import torch
import torch.nn as nn
import torch.nn.functional as F
# 定义一个简单的多层感知机（MLP）模型，包含一个隐藏层

class SimpleMLP(nn.Module):
"""单隐层 MLP 模型"""
初始化模型结构
def __init__(self, input_dim=3*32*32, hidden_dim=512, output_dim=10):
super(SimpleMLP, self).__init__()    # 调用父类构造函数初始化
self.flatten = nn.Flatten()    # 将多维输入张量展平成一维（例如 [B, 3, 32, 32] -> [B, 3072]）
self.fc1 = nn.Linear(input_dim, hidden_dim)    # 第一个全连接层：输入 -> 隐藏层
self.relu = nn.ReLU()    # ReLU 激活函数
self.fc2 = nn.Linear(hidden_dim, output_dim)    # 第二个全连接层：隐藏层 -> 输出层（类别分数）

def forward(self, x):
x = self.flatten(x)    # 将输入展平为二维张量 [B, 3072]
x = self.relu(self.fc1(x))
x = self.fc2(x)
return x    #返回模型输出
```

3.2 CNN 模型构建

```
import torch
import torch.nn as nn
import torch.nn.functional as F
class SimpleCNN(nn.Module):    # 定义一个简单的卷积神经网络（CNN）模型
"""简单的 CNN 模型，包含两个卷积层和两个全连接层"""
def __init__(self):
super(SimpleCNN, self).__init__()
self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)    # 第一个卷积层
self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)    # 第二个卷积层
self.pool = nn.MaxPool2d(2, 2)
self.fc1 = nn.Linear(32 * 8 * 8, 128)
```

```
self.fc2 = nn.Linear(128, 10)
self.relu = nn.ReLU()     # ReLU 激活函数

def forward(self, x):
x = self.pool(self.relu(self.conv1(x))) # 输出大小: 16x16x16
x = self.pool(self.relu(self.conv2(x))) # 输出大小: 8x8x32
x = x.view(-1, 32 * 8 * 8)    # 将多维特征图展平成一维向量，准备送入全连接层
x = self.relu(self.fc1(x))
x = self.fc2(x)
return x
```

3.3 模型的训练和评估

```
# 计算模型复杂度
print("\n 分析模型复杂度:")
model_complexity(model, device=device)

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# 可以添加学习率调度器
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=epochs)

# 确保 checkpoints 目录存在
os.makedirs(save_directory, exist_ok=True)

# 训练模型
trained_model, history = train_model(
    model, train_loader, valid_loader, criterion, optimizer, scheduler,
    num_epochs=epochs, device=device, save_dir=save_directory
)

# 绘制训练历史
plot_training_history(history, title=f"{model_name} Training History")

# 在测试集上评估模型
print("\n 在测试集上评估模型:")
test_loss, test_acc = evaluate_model(trained_model, test_loader, criterion, device, classes)

print(f"{model_name} 最终测试准确率: {test_acc:.4f}")
```

3.4 模型的比较和分析

```
# 训练和评估每个模型
for model_name, model in models.items():
print(f"\n 开始训练 {model_name}...")

# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=15)


# 计算模型复杂度
print(f"\n 分析 {model_name} 复杂度...")
num_params, inference_time = model_complexity(model, device=device)


# 训练模型
_, history = train_model(
model, train_loader, valid_loader, criterion, optimizer, scheduler,
num_epochs=15, device=device, save_dir='./checkpoints'
)


# 在测试集上评估模型
test_loss, test_acc = evaluate_model(model, test_loader, criterion, device)


print(f"{model_name} 测试准确率: {test_acc:.4f}")


# 存储结果
results[model_name] = {
'history': history,
'test_acc': test_acc,
'params': num_params,
'inf_time': inference_time
}


# 比较模型性能
model_names = list(results.keys())
test_accs = [results[name]['test_acc'] for name in model_names]
params = [results[name]['params'] / 1e6 for name in model_names] # 转换为百万
inf_times = [results[name]['inf_time'] * 1000 for name in model_names] # 转换为毫秒
```

§4 实验结果和分析

思考问题 1　MLP 在处理图像数据时面临哪些挑战？请从数据结构、参数量和特征提取能力三个角度分析。

数据结构角度：

图像的本质是二维或三维结构（如 32×32×3 的彩色图像），具有局部空间相关性（邻近像素之间有联系）。然而，MLP 的输入需要展平成一维向量（如 32×32×3 = 3072 维向量），这会破坏图像的空间结构。结果是模型失去了对像素位置和邻近关系的感知，难以有效提取局部模式（如边缘、角点）。

参数量角度：

将图像展平成一维后，每个输入节点都与每个隐藏层节点连接，形成全连接网络。多层 MLP 堆叠后参数量会进一步激增，容易导致训练缓慢，需要大量内存，且更容易过拟合。

特征提取能力角度：

MLP 的每个神经元都处理整个输入的线性加权和，没有“局部感知”机制。不同于 CNN 使用卷积核提取图像局部特征（边缘、纹理、图案等），MLP 无法识别图像中的空间模式。同时，CNN 通过多层卷积可以逐步构建从低级到高级的特征层次，而 MLP 缺乏这种层次化特征提取能力。

思考问题 2: CNN 相比 MLP 在处理图像时具有哪些优势？解释卷积操作如何保留图像的空间信息。

相比于 MLP,CNN 的空间信息得以保留，且参数更少，局部感知能力更强，具有多层次表示的特点，且具有平移不变性。

卷积操作中，卷积核在图像的每个小区域滑动，只处理该区域内的像素。这意味着每个卷积输出值都对应图像中的一个具体位置和局部区域，不会打乱空间结构。并且由于卷积核在整

张图上滑动，同一组参数在不同位置使用。不仅减少参数数量，还让模型能识别相同的图案无论它出现在图像的哪个位置。此外，卷积层输出的特征图仍保留二维结构（如从 32×32 图像得到 30×30 特征图），不像 MLP 那样将图像展平。这样可以在后续层中继续利用空间信息，提取更复杂的结构。

任务 1 在下面的代码块中，实现一个具有两个隐藏层的 MLP 模型。第一隐藏层有 128 个神经元，第二隐藏层有 64 个神经元，输出层对应 10 个类别。使用 ReLU 激活函数，并添加 BatchNorm 和 Dropout(0.3)。

```
import torch.nn as nn
class TwoLayerMLP(nn.Module):
    def __init__(self, input_dim=3*32*32):
        super(TwoLayerMLP, self).__init__()
        self.flatten = nn.Flatten()  # 将输入展平为一维向量
        # 第一个隐藏层：128 个神经元
        self.layer1 = nn.Sequential(
            nn.Linear(input_dim, 128),  # 全连接层
            nn.BatchNorm1d(128),        # 批量归一化
            nn.ReLU(),                  # ReLU 激活函数
            nn.Dropout(0.3)             # Dropout 正则化
        )
        # 第二个隐藏层：64 个神经元
        self.layer2 = nn.Sequential(
            nn.Linear(128, 64),         # 全连接层
            nn.BatchNorm1d(64),         # 批量归一化
```

```
            nn.ReLU(),                      # ReLU 激活函数
            nn.Dropout(0.3)                 # Dropout 正则化
        )
        # 输出层：10 个类别
        self.out = nn.Linear(64, 10)    # 输出层不需要激活函数

    def forward(self, x):
        x = self.flatten(x)             # 先展平输入
        x = self.layer1(x)              # 通过第一隐藏层
        x = self.layer2(x)              # 通过第二隐藏层
        x = self.out(x)                 # 输出层
        return x
```

分析问题 1：训练过程中，损失和准确率曲线表现如何？是否出现过拟合或欠拟合？简要分析可能的原因。

训练损失和验证损失逐轮下降，训练准确率和验证准确率逐步提升。第 1 轮训练准确率：0.4045，验证准确率：0.4768，而第 15 轮训练准确率：0.6960，验证准确率：0.6846，说明没有出现过拟合和欠拟合的现象。

分析问题 2：对比 SimpleMLP 和 DeepMLP 的性能，增加网络深度对性能有何影响？

从数据上看，deepmlp 第 1 轮训练准确率：0.2840，验证准确率：0.3494，而第 15 轮训练准确率：0.4598，验证准确率：0.4868，说明增加网络深度会降低性能。

任务 2 修改下面的 SimpleCNN 代码，添加一个额外的卷积层和 BatchNorm。新的卷积层应该在第二个池化层之后，卷积核数量为 64，卷积核大小为 3x3。

```
import torch.nn as nn
class EnhancedCNN(nn.Module):
    def __init__(self):
        super(EnhancedCNN, self).__init__()
        # 第一个卷积块：输入 3 通道，输出 16 通道，3x3 卷积核，padding=1 保持尺寸
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),  # 卷积层
            nn.BatchNorm2d(16),                          # 批量归一化（加速训练）
            nn.ReLU()                                    # ReLU 激活函数
        )
        # 第二个卷积块：输入 16 通道，输出 32 通道
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU()
        )

        # 新增的第三个卷积块：输入 32 通道，输出 64 通道（任务要求）
        self.conv3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=3, padding=1),  # 3x3 卷积核
            nn.BatchNorm2d(64),                          # 新增的 BatchNorm
            nn.ReLU()
        )
```

```
        # 公共池化层：2x2 最大池化，步长 2（尺寸减半）
        self.pool = nn.MaxPool2d(2, 2)
        # 展平层：将多维特征图转换为一维向量
        self.flatten = nn.Flatten()
        # 全连接层（假设输入图像为 32x32）：
        # 经过 3 次池化后尺寸：32 → 16 → 8 → 4，最终特征图尺寸 4x4x64=1024
        self.fc = nn.Sequential(
            nn.Linear(64 * 4 * 4, 256),  # 输入 1024 维，输出 256 维
            nn.ReLU(),                   # 全连接层后加 ReLU
            nn.Linear(256, 10)           # 输出 10 分类
        )
    def forward(self, x):
        # 第一卷积块 + 池化
        x = self.conv1(x)       # 输出尺寸：[batch, 16, 32, 32]
        x = self.pool(x)        # 池化后：[batch, 16, 16, 16]
        # 第二卷积块 + 池化
        x = self.conv2(x)       # 输出尺寸：[batch, 32, 16, 16]
        x = self.pool(x)        # 池化后：[batch, 32, 8, 8]
        # 新增的第三卷积块 + 池化
        x = self.conv3(x)       # 输出尺寸：[batch, 64, 8, 8]
        x = self.pool(x)        # 池化后：[batch, 64, 4, 4] ← 关键修改点
        # 全连接部分
        x = self.flatten(x)     # 展平为[batch, 64*4*4=1024]
        x = self.fc(x)          # 通过全连接层输出 10 分类
        return x
```

分析问题 3：卷积核可视化显示了什么模式？这些模式与图像中的哪些特征可能对应？

边缘检测器，对应图像中的边缘、轮廓（如物体边界、纹理过渡区域）。

颜色敏感核，对应彩色物体的色块分界。

斑点检测器，对应局部小尺度结构。

低频/高频滤波器,对应细节和模糊处理。

分析问题 4: CNN 模型相比 MLP 在 CIFAR-10 上的性能有何不同？为什么会有这样的差异？

在 CIFAR-10 数据集上，CNN 模型通常显著优于 MLP 模型，因为 CNN 可以通过局部连接+参数共享高效提取空间特征，用层次化架构模拟视觉处理流程，它的平移不变性也能减少对数据增强的依赖。

思考问题 3：分析 Bottleneck 结构的优势。为什么 1×1 卷积在深度 CNN 中如此重要？它如何帮助控制网络的参数量和计算复杂度？

Bottleneck 结构是深度 CNN（如 ResNet、MobileNet 等）中的关键设计，其核心是通过 1×1 卷积实现通道维度的压缩与扩展，其通过先降维处理再升维的方式从而显著降低参数量和计算量，同时保持甚至提升模型性能。

探索问题 1：查看 models/cnn.py 中的 SimpleResNet 实现，分析残差连接是如何实现的。如果输入和输出通道数不匹配，代码是如何处理的？

在 SimpleResNet 的中，残差连接通过 ResidualBlock 模块实现，其核心思想是将输

入 x 与卷积层的输出 F(x) 相加（F(x) + x），从而缓解深层网络的梯度消失问题。通道不匹配时，动态创建包含 1x1 卷积 + BN 的 shortcut 分支来解决这一问题。

分析问题 5: VGG 风格和 ResNet 风格网络的性能比较。残差连接带来了哪些优势？

可以解决梯度消失/爆炸，并且允许训练极深层网络，提升特征复用效率，抑制网络退化。

分析问题 6: 参数量和推理时间如何影响模型的实用性？如何在性能和效率之间找到平衡？

参数量会影响内存占用和训练成本，同时参数量过多和过少会导致模型过拟合或欠拟合，进而影响模型的实用性；推理时间则会影响能耗，计算延迟和服务成本。要在性能和效率之间找到平衡，就应该找到需求的边界，在资源允许的范围内选择最大的模型，然后通过优化技术将其压缩到部署平台上。

探索问题 2: 如果你要为移动设备设计一个 CNN 模型，应该考虑哪些因素来权衡性能和效率？请提出至少三条具体的设计原则。

降低模型复杂度：使用轻量级网络结构如 MobileNet、EfficientNet-Lite、SqueezeNet 或 ShuffleNet，这些网络专为移动和嵌入式设备设计，参数量和计算量都很小。

模型量化和剪枝：减少模型大小与计算需求，将模型的权重和激活从 32-bit 浮点数降低到 8-bit 整数，极大减少存储和加速推理速度，几乎不影响准确率。同时移除模型中不重要的连接或神经元，减小模型规模，提高执行效率。

限制输入尺寸与操作分辨率，控制输入图像的分辨率（例如从 224x224 降到 160x160），

可显著减少计算量，但仍需确保足够的信息保留。

分析不同类型模型（MLP 和 CNN）以及不同复杂度模型的性能差异。考虑以下几点： 1. 测试准确率 2. 参数量 3. 推理时间 4. 训练收敛速度 5. 过拟合/欠拟合情况

1.CNN 在图像数据上测试准确率优于 MLP。

2.CNN 通过参数共享（卷积核）大幅降低参数量，适合高维数据；而 MLP 参数量与输入维度强相关，不适合图像等高维数据。

3.MLP 推理速度通常快于 CNN（参数量相同时），但实际中 CNN 通过优化（如 Winograd 算法）可缩小差距。

4.CNN 通常收敛更快。

5.欠拟合常见于简单 MLP，而 CNN 和复杂 MLP 则可能过拟合。

§5 创新探索

5.1 迁移学习

探索如何利用预训练模型提高 CIFAR-10 的分类性能。

方法一：微调全模型（Fine-tuning）适用于数据集较大、希望模型能完全适应新任务。所有参数都是可训练的。

```
import torch
import torchvision.models as models
import torch.nn as nn

# 加载预训练模型
model = models.resnet18(pretrained=True)

# 修改最后的全连接层（ResNet 的分类头是 fc 层）
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, 10)  # CIFAR-10 有 10 个类别

# 全部参数参与训练（fine-tune）
for param in model.parameters():
    param.requires_grad = True
```

方法二：冻结特征提取层，只训练分类器（Feature extraction）更快、参数更少、泛化性好。适合小数据集或设备受限的情况。

```
# 冻结所有层的参数
for param in model.parameters():
    param.requires_grad = False

# 替换并训练最后一层
model.fc = nn.Linear(model.fc.in_features, 10)
```

5.2 自监督学习

实现一个简单的自监督学习方法，并评估其效果。

在预训练阶段（自监督学习），我们用对比学习训练一个编码器；在评估阶段（监督线性分类）我们冻结编码器，用其输出作为特征训练线性分类器。代码如下：

```
#依赖导入
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchvision.models import resnet18
import random


#自定义数据生成“正样本对”
class ContrastiveTransform:
    def __init__(self, base_transform):
        self.base_transform = base_transform

    def __call__(self, x):
        xi = self.base_transform(x)
        xj = self.base_transform(x)
        return xi, xj

contrast_transform = transforms.Compose([
```

```
        transforms.RandomResizedCrop(32),
        transforms.RandomHorizontalFlip(),
        transforms.ColorJitter(0.4, 0.4, 0.4, 0.1),
        transforms.RandomGrayscale(p=0.2),
        transforms.ToTensor(),
])

train_dataset = torchvision.datasets.CIFAR10(
        root='./data', train=True, download=True,
        transform=ContrastiveTransform(contrast_transform)
)

train_loader = DataLoader(train_dataset, batch_size=256, shuffle=True, num_workers=2)



#定义编码器+投影头
class SimCLRModel(nn.Module):
        def __init__(self, base_encoder, projection_dim=128):
                super(SimCLRModel, self).__init__()
                self.encoder = base_encoder
                self.encoder.fc = nn.Identity()   # 去掉原来的分类头
                self.projector = nn.Sequential(
                        nn.Linear(512, 512),
                        nn.ReLU(),
                        nn.Linear(512, projection_dim)
                )

        def forward(self, x):
                h = self.encoder(x)         # 特征
```

```
        z = self.projector(h)     # 投影空间
        return F.normalize(z, dim=1)


#对比损失函数（Nt-Xent）
def contrastive_loss(z_i, z_j, temperature=0.5):
    batch_size = z_i.size(0)
    z = torch.cat([z_i, z_j], dim=0)    # [2B, D]
    similarity = F.cosine_similarity(z.unsqueeze(1), z.unsqueeze(0), dim=2)

    labels = torch.arange(batch_size).to(z.device)
    labels = torch.cat([labels, labels], dim=0)

    mask = torch.eye(batch_size * 2, dtype=torch.bool).to(z.device)
    logits = similarity / temperature
    logits = logits.masked_fill(mask, -1e9)    # 避免自身匹配

    loss = F.cross_entropy(logits, labels)
    return loss


#预训练模型（自监督学习）
def contrastive_loss(z_i, z_j, temperature=0.5):
    batch_size = z_i.size(0)
    z = torch.cat([z_i, z_j], dim=0)    # [2B, D]
    similarity = F.cosine_similarity(z.unsqueeze(1), z.unsqueeze(0), dim=2)

    labels = torch.arange(batch_size).to(z.device)
    labels = torch.cat([labels, labels], dim=0)
```

```
        mask = torch.eye(batch_size * 2, dtype=torch.bool).to(z.device)
        logits = similarity / temperature
        logits = logits.masked_fill(mask, -1e9)   # 避免自身匹配

        loss = F.cross_entropy(logits, labels)
        return loss


#评估：用冻结的编码器做线性分类
# 冻结编码器，提取特征
model.eval()
features = []
labels = []

eval_transform = transforms.Compose([
        transforms.ToTensor(),
])

test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=eval_transform)
test_loader = DataLoader(test_dataset, batch_size=256, shuffle=False)

with torch.no_grad():
        for x, y in test_loader:
                x = x.to(device)
                h = model.encoder(x)
                features.append(h.cpu())
                labels.append(y)
```

```
features = torch.cat(features)
labels = torch.cat(labels)

# 训练一个线性分类器
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

clf = LogisticRegression(max_iter=1000)
clf.fit(features, labels)
preds = clf.predict(features)
acc = accuracy_score(labels, preds)
print(f"Linear evaluation accuracy: {acc * 100:.2f}%")
```

§6 结论与思考

在本次实验中，我们通过对比 MLP（多层感知机）与 CNN（卷积神经网络）在 MNIST 手写数字识别任务中的表现，深入理解了两种神经网络模型的结构特点与适用场景。实验开始前，我们首先思考了为什么要对这两类模型进行对比。MLP 作为基础的前馈神经网络，结构相对简单，但在处理图像等高维数据时，可能存在参数过多、特征提取能力有限的问题。而 CNN 则通过引入卷积层和池化层，能够有效提取局部特征，减少参数数量，更适合图像类任务。在模型搭建与训练过程中，我们观察到：

在训练集上的表现： MLP 和 CNN 都能达到较高的准确率，但随着 epoch 的增加，CNN 的准确率上升更快且更稳定。

在测试集上的泛化能力： CNN 的表现显著优于 MLP，表明其在防止过拟合、提升模型泛化方面具有更大优势。

在训练速度与收敛速度方面： CNN 由于参数共享与局部连接机制，在保证性能的同时训练效率也较高。

模型复杂度与性能平衡： 尽管 CNN 的结构相对复杂，但其出色的表现证明了结构设计对于特定任务的重要性。

结论：在处理图像数据时，MLP 存在一定局限性，尤其在特征提取能力和模型泛化能力方面，而 CNN 在图像识别任务中具有明显优势，不仅能更有效地提取图像的局部特征，还能显著提升模型性能。同时我意识到了模型的结构设计对最终性能影响巨大，应根据任务特性选择合适的模型类型。

§7 参考文献

1. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

2. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. CVPR.

3. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprintarXiv:1409.1556.

4. PyTorch 文档： https://pytorch.org/docs/stable/index.html5. CS231n: Convolutional Neural Networks for Visual Recognition：https://cs231n.github.io/