浙江大学
ZHEJIANG UNIVERSITY

人工智能基础实验报告

作业名称 实验报告五

姓　　名 试鸢

学　　号 *********

电子邮箱 **********@email.com

联系电话 **********

指导老师

2025 年 5 月 4 日

实验背景和目的

随着深度学习技术的飞速发展，神经网络在图像分类、语音识别、自然语言处理等领域取得了显著的成果。多层感知机（MLP）和卷积神经网络（CNN）是两种经典的神经网络架构，广泛应用于各种任务中。

MLP 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。它通过全连接的方式将输入数据映射到输出结果，适用于处理结构化数据。然而，在处理图像数据时，MLP 面临着一些挑战。图像数据具有高维度和复杂的空间结构，MLP 需要大量的参数来学习这些结构，导致计算成本高、训练时间长，且容易过拟合。

CNN 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。它通过局部连接和权重共享的方式减少了参数数量，同时能够提取图像的空间特征。CNN 在图像分类任务中表现出色，能够自动学习图像的层次化特征表示，具有很强的泛化能力。

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，深入理解两种模型的结构特点、性能差异以及适用场景。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。具体目标包括：

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

目　　录

1 实验原理简述

1.1 多层感知机（MLP）

多层感知机（MLP）是一种经典的前馈神经网络，其主要特点如下：

- 全连接结构：MLP 的每一层神经元都与下一层的所有神经元相连。输入层接收输入数据，经过一个或多个隐藏层的处理，最终输出层产生分类结果。
- 非线性激活函数：为了引入非线性特性，MLP 在每一层的输出上应用非线性激活函数，如 ReLU（Rectified Linear Unit）、Sigmoid 或 Tanh 等。这些激活函数使得网络能够学习复杂的非线- 性关系。
- 反向传播训练[1]：MLP 通过反向传播算法进行训练。在训练过程中，网络的输出与真实标签之间的误差通过损失函数计算，然后通过反向传播将误差逐层传递回网络的输入层，并根据梯度下降法更新网络的权重，以最小化损失函数。

在本实验中，MLP 模型的实现包括：

- SimpleMLP：单隐层 MLP，结构简单，适合初步理解 MLP 的工作原理。
- DeepMLP：多隐层 MLP，通过增加隐藏层的数量来提高模型的表达能力，同时引入 BatchNorm 和 Dropout 技术来改善训练性能和防止过拟合。
- ResidualMLP：带有残差连接的 MLP，通过残差学习解决深层网络训练中的梯度消失问题。

1.2 卷积神经网络（CNN）

卷积神经网络（CNN）是专门为处理具有网格状拓扑结构的数据（如图像）而设计的神经网络[2]，其主要特点如下：

- 局部连接：CNN 的每个神经元只与输入数据的一个局部区域连接，这种局部连接方式使得网络能够捕捉到图像的局部特征。
- 权重共享：同一特征图的所有神经元共享相同的权重，这大大减少了模型的参数量，同时提高了模型的泛化能力。
- 多层次特征提取：CNN 通过堆叠多个卷积层和池化层，逐步提取图像的层次化特征。低层卷积层通常检测简单的边缘、纹理等特征，而高层卷积层则组合这些简单特征形成更复杂的表示。
- 池化层：池化层用于降低特征图的空间维度，同时保留重要的特征信息。常用的池化操作包括最大池化（Max Pooling）和平均池化（Average Pooling）。
- 全连接层：在 CNN 的最后阶段，通常会将卷积层和池化层提取的特征图展平为一维向量，然后通过全连接层进行分类。

在本实验中，CNN 模型的实现包括：

- SimpleCNN：简单的 CNN，包含两个卷积层和一个池化层，适合初步理解 CNN 的基本结构和工作原理。
- MediumCNN：中等复杂度的 CNN，增加了更多的卷积层和池化层，并引入 BatchNorm 和 Dropout 技术来改善模型的性能。
- VGGStyleNet：基于 VGG 架构的 CNN，使用小尺寸（3×3）卷积核和 2×2 最大池化层，通过堆叠多个卷积层来增加网络深度，具有规整的结构。
- SimpleResNet：简化的 ResNet，包含残差连接，通过残差学习解决了深层网络训练中的梯度消失问题，使得训练更深的网络成为可能。

2 实验过程描述

2.1 实验环境搭建

本实验使用了 Python 编程语言和 PyTorch 深度学习框架进行模型的实现和训练。本地实验环境配置如下：

- 操作系统：Arch Linux
- Python 版本：3.10.16
- PyTorch 版本：2.7.0+cu128
- CUDA 版本：12.8
- 数据集：CIFAR-10
- 其他依赖库：NumPy，Matplotlib，scikit-learn 等

2.2 实验步骤

2.2.1 第一部分：基础 MLP 模型

2.2.1.1 了解 MLP 模型结构

查看 models/mlp.py 文件，理解三种 MLP 模型的结构：

- SimpleMLP：单隐层 MLP。
- DeepMLP：多隐层 MLP，带有 BatchNorm 和 Dropout。
- ResidualMLP：带有残差连接的 MLP。

2.2.1.2 训练和评估 MLP 模型

1. 在 train.ipynb 中训练 SimpleMLP 模型，确保将 model_type 设置为 simple_mlp。
2. 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。
3. 修改参数尝试训练 DeepMLP 模型，将 model_type 设置为 deep_mlp。

2.2.2 第二部分：基础 CNN 模型

2.2.2.1 了解 CNN 模型结构

查看 models/cnn.py 文件，理解不同 CNN 模型的结构：

- SimpleCNN：简单的 CNN，包含两个卷积层。
- MediumCNN：中等复杂度的 CNN，带有 BatchNorm 和 Dropout。
- VGGStyleNet：VGG 风格的 CNN，使用连续的 3x3 卷积。
- SimpleResNet：简化的 ResNet，包含残差连接。

2.2.2.2 训练和评估 CNN 模型

1. 在 train.ipynb 中训练 SimpleCNN 模型，确保将 model_type 设置为 simple_cnn，并将 use_data_augmentation 设置为 True。
2. 观察训练过程和卷积核可视化结果。

2.3 高级 CNN 架构探索

2.3.1 VGG 风格和 ResNet 风格网络架构

2.3.1.1 VGG 架构特点

VGG 网络[3]（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

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。

1. 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。
2. 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。

2.3.1.2 ResNet 架构及残差连接

ResNet[4]（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

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。

1. 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 。
2. 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。

2.3.1.3 Bottleneck 结构

在更深的 ResNet 变体中，常使用“瓶颈”（Bottleneck[4]）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。

2.3.2 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。

1. 运行以下代码来分析各个模型的复杂度：

```
from models import SimpleMLP, DeepMLP,ResidualMLP, SimpleCNN, MediumCNN,
VGGStyleNet,SimpleResNet

from utils import model_complexity

import torch

device = torch.device('cuda:0' if
torch.cuda.is_available() else 'cpu')

models = {
'SimpleMLP': SimpleMLP(),
'DeepMLP': DeepMLP(),
'SimpleCNN': SimpleCNN(),
'MediumCNN': MediumCNN(),
'VGGStyleNet': VGGStyleNet(),
'SimpleResNet': SimpleResNet()
}

results = {}
for name, model in models.items():
  print(f"\n分析{name}复杂度:")
  params, time = model_complexity(model,device=device)
  results[name] = {'params': params, 'time':time}
```

2. 记录并比较各个模型的参数量和推理时间。

2.3.3 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[5]
2. 跳跃连接：除了 ResNet 的残差连接，还有 DenseNet[6]的密集连接、U-Net[7]的跨层连接等
3. 特征增强：注意力机制（如 SENet[8]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[9]）、组卷积（ShuffleNet[10]）等

2.4 模型比较与分析

运行 compare.py 来对比不同模型的性能：

综合分析：根据比较结果，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 实现代码

```python
import torch
import torch.nn as nn
import torch.optim as optim
import os

# 导入项目中的模块
from models import SimpleMLP, DeepMLP, ResidualMLP, SimpleCNN, MediumCNN, VGGStyleNet, SimpleResNet
from utils import (
    load_cifar10, # 加载CIFAR-10数据集
    set_seed, # 设置随机种子
    load_model, # 加载模型
    train_model, # 训练模型
    evaluate_model, # 评估模型
    plot_training_history, # 绘制训练历史
    visualize_model_predictions, # 可视化模型预测
    visualize_conv_filters, # 可视化卷积核
    model_complexity # 分析模型复杂度
)
# 设置参数
model_type = 'simple_mlp'  # 可选: 'simple_mlp', 'deep_mlp',
                           # 'residual_mlp', 'simple_cnn',
                           # 'medium_cnn', 'vgg_style', 'resnet'
epochs = 50 # 训练轮数
learning_rate = 0.001 # 学习率
batch_size = 128 # 批数量
use_data_augmentation = True  # 是否数据增强 CNN通常受益
save_directory = f"./{model_type}ck"
visualize_filters = True  # 是否可视化卷积核
visualize_predictions = True  # 是否可视化预测结果

# 设置随机种子
set_seed()

# 检查是否有可用的GPU
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(f"使用设备: {device}")

# 加载数据
train_loader, valid_loader, test_loader, classes = load_cifar10(
    use_augmentation=use_data_augmentation,
    batch_size=batch_size
)

# 初始化选择的模型
model, model_name = load_model(model_type)

print(f"使用模型: {model_name}")
# 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
)

```

```python
# 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}")

# 如果是CNN模型并且需要可视化卷积核
if visualize_filters and model_type in ['simple_cnn', 'medium_cnn', 'vgg_style', 'resnet']:
    print("\n可视化卷积核:")
    if model_type == 'simple_cnn':
        visualize_conv_filters(trained_model, 'conv1')
    elif model_type == 'medium_cnn':
        visualize_conv_filters(trained_model, 'conv1')
    elif model_type == 'vgg_style':
        visualize_conv_filters(trained_model, 'features.0')
    else:  # resnet
        visualize_conv_filters(trained_model, 'conv1')

# 如果需要可视化模型预测
if visualize_predictions:
    print("\n可视化模型预测:")
    visualize_model_predictions(trained_model, test_loader, classes, device)

print(f"\n{model_name}的训练和评估已完成! ")
```

4 实验结果与分析

4.1 思考问题 1

1. 数据结构
   - 空间信息丢失: MLP 需要将输入图像展平为一维向量，破坏了像素间的空间局部性（如相邻像素的关联性）。这种结构忽略了图像的 2D/3D 拓扑关系，导致模型难以捕捉边缘、纹理等局部模式。
   - 平移不变性缺失: 图像中的目标无论位于哪个位置都应被识别，但 MLP 对输入顺序敏感（例如，平移后的图像会被视为完全不同的输入），需依赖大量数据隐式学习平移不变性，效率低下。
2. 参数量
   - 全连接的高计算成本：MLP 的层间全连接导致参数量爆炸。例如，处理 224x224 的 RGB 图像时，输入层到第一隐藏层（假设隐藏层大小 1024）的参数量为 $224\text{x}224\times3\times1024 \approx 154\text{M}$，远超 CNN 的局部连接方式。
   - 过拟合风险: 高参数量需要极大训练数据量来避免过拟合，而真实图像数据集通常标注成本高昂，难以满足 MLP 的需求。
3. 特征提取能力
   - 手工设计特征的局限性：传统 MLP 依赖人工设计特征（如 SIFT+HOG+MLP 组合），但图像的低级（边缘、角点）到高级特征（物体部件、整体）需分层自动提取。MLP 缺乏层级归纳偏置，难以像 CNN 那样通过卷积核分层捕获特征。
   - 全局依赖与局部感知的矛盾: MLP 理论上可通过深层网络拟合任意函数，但实际训练中难以同时兼顾局部细节（如纹理）和全局语义（如物体类别）。而 CNN 通过卷积–池化–深层的结构天然支持这一过程。

4.2 思考问题 2

1. 局部感知
   - 操作方式：卷积核（如 3x3）在图像上滑动，每次仅计算局部区域（如 3x3 像素块）的加权和，而非 MLP 的全连接。
   - 保留空间结构：像素的邻域关系（如边缘、纹理）被显式保留，避免展平操作导致的空间信息破坏。
2. 权重共享
   - 操作方式: 同一卷积核在整个图像上共享参数，无论检测目标位于图像中心还是角落。

- 平移不变性：相同模式（如猫耳）在不同位置会被同一核识别，无需 MLP 那样为每个位置学习独立参数。

3. 层次化特征提取
   - 低级→高级特征：
     - 浅层卷积：捕捉边缘、颜色等局部特征。
     - 深层卷积：组合局部特征为高级语义（如物体部件、整体类别）。
   - 空间信息传递：通过池化（如 2x2 Max Pooling）逐步降低分辨率，但保留主要空间结构。

4.3 任务 1：

```python
from torch.nn import nn

class TwoLayerMLP(nn.model):

  def __init__(self,input_dim=3*32*32):
    super(TwoLayerMLP,self).__init__()
    self.flatten = nn.Flatten()

    # 第一层
    self.fc1 = nn.Linear(input_dim, 1024)
    self.bn1 = nn.BatchNorm1d(1024)

    # 第二层
    self.fc2 = nn.Linear(1024, 512)

    # 激活和Dropout
    self.relu = nn.ReLU()
    self.dropout = nn.Dropout(dropout_rate)

  def forward(self, x):
    x = self.flatten(x)

    # 第一层
    x = self.fc1(x)
    x = self.bn1(x)
    x = self.relu(x)
    x = self.dropout(x)

    # 第二层
    x = self.fc2(x)

    return x
```

4.4 分析问题 1

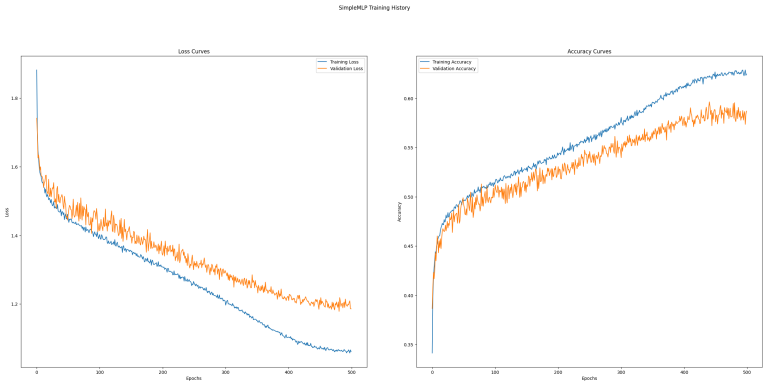

图 4-1 SimpleMLP Training History

训练损失逐渐下降并在 400epoch 后趋于平缓，准确率逐渐上升后趋于平缓。但由于 MLP 的参数量较大且难以捕捉图像的空间信息，训练过程较慢，且最终准确率较低。

由于模型结构简单，无法充分学习数据中的复杂特征，可能导致欠拟合。但是 500epochs 后训练损失和验证损失趋于平稳，说明模型已经收敛。不过模型训练准确率高于测试准确率，可能过拟合。

4.5 分析问题 2:

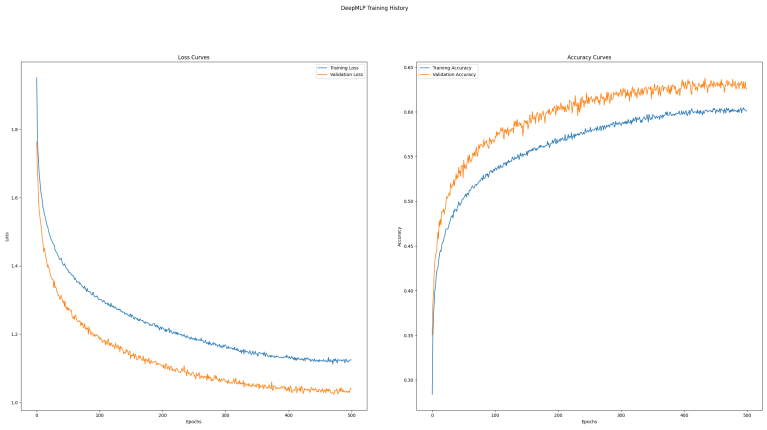

图 4-2 DeepMLP Training History

DeepMLP 由于增加了隐藏层数量和 BatchNorm、Dropout，训练过程更加稳定，损失下降更快，准确率有些许提高。同时 BatchNorm 可以加速训练过程，Dropout 可以防止过拟合。

4.6 任务 2

```python
from torch.nn import nn

class EnhancedCNN(nn.Module):

  def __init__(self):
    super(EnhancedCNN,self).__init__()
    self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
    self.bn1 = nn.BatchNorm2d(16)

    self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
    self.bn2 = nn.BatchNorm2d(32)

    self.pool1 = nn.MaxPool2d(2, 2)

    self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
    self.bn3 = nn.BatchNorm2d(64)

    self.pool2 = nn.MaxPool2d(2, 2)

    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(64 * 4 * 4, 10)
    self.bn1 = nn.BatchNorm1d(10)
    self.relu = nn.ReLU()

  def forward(self, x):
    x = self.relu(self.bn1(self.conv1(x)))
    x = self.relu(self.bn2(self.conv2(x)))
    x = self.pool1(x)

    x = self.relu(self.bn3(self.conv3(x)))
    x = self.pool2(x)

    x = self.flatten(x)
    x = self.fc1(x)

    return x
```

4.7 分析问题 3:

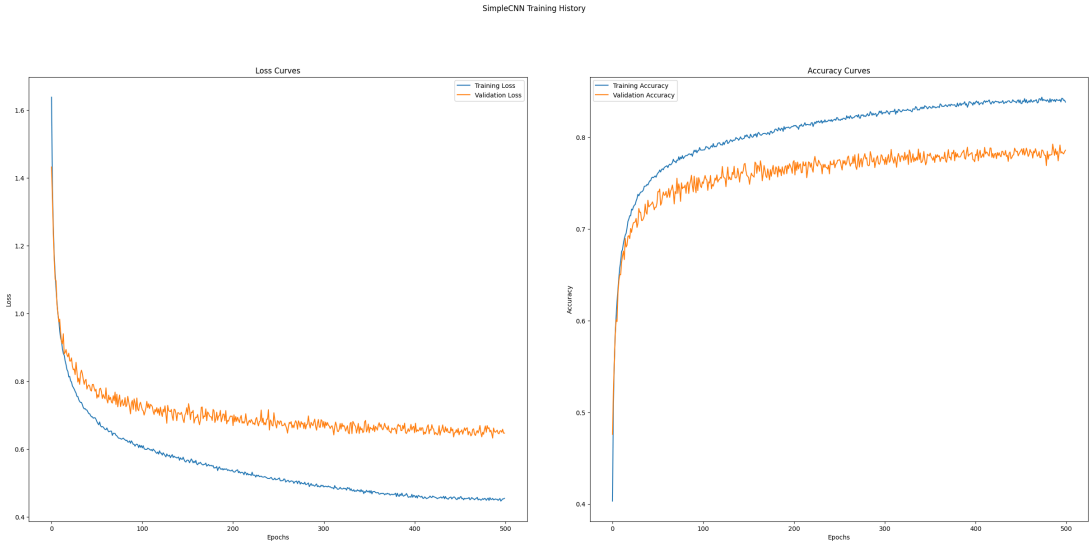

图 4-3 SimpleCNN Training History

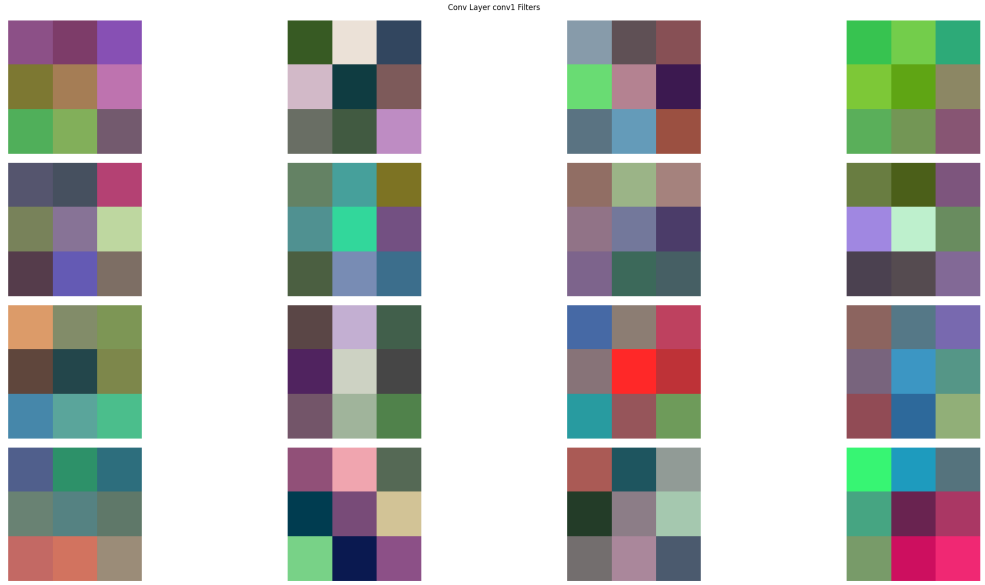

图 4-4 SimpleCNN Conv Layer conv1 Filters

观察到的卷积核模式:

1. **边缘检测**: 一些卷积核显示为明显的边缘检测器，例如垂直(1-4)、水平或对角线方向的边缘。
2. **纹理检测**: 其他卷积核可能显示为纹理检测器，能够识别图像中的特定纹理模式，如条纹、斑点或波纹。
3. **颜色和亮度**: 还有一些卷积核可能对颜色或亮度变化敏感，能够识别图像中的特定颜色区域或亮度对比。
4. **特定形状**: 某些卷积核可能专门用于识别特定的形状，如角点、圆形或直线。

这些卷积核学习到的特征与图像中的特征相对应，例如：

- **边缘**：对应于图像中物体的轮廓或边界。
- **纹理**：对应于图像中物体表面的材质或图案。
- **颜色**：对应于图像中物体的颜色区域或颜色对比。
- **形状**：对应于图像中物体的几何形状或结构。

4.8 分析问题 4：

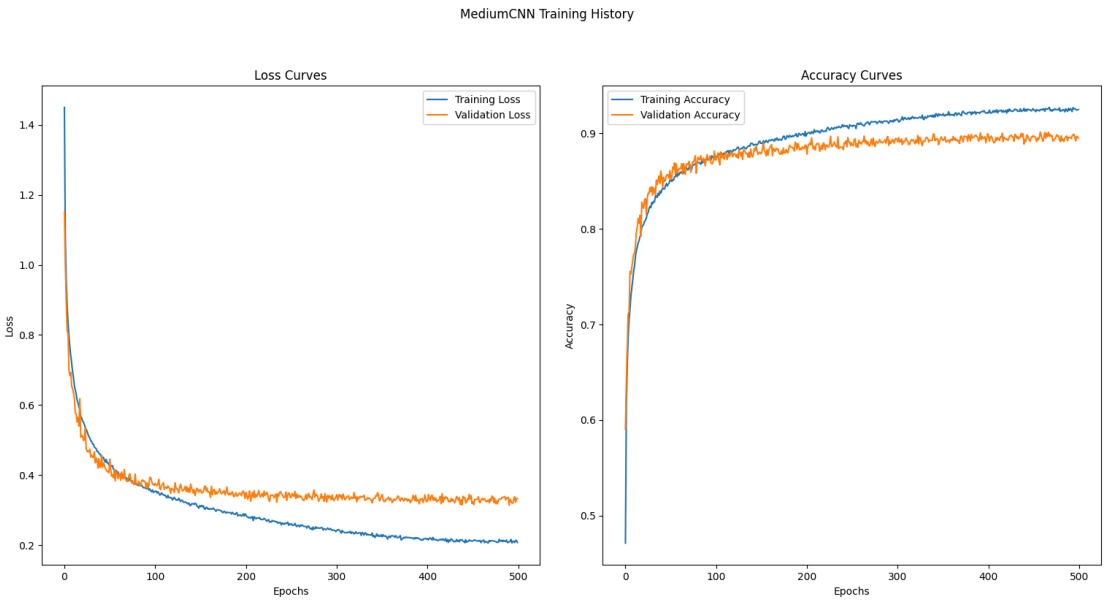

图 4-5 MediumCNN Training History

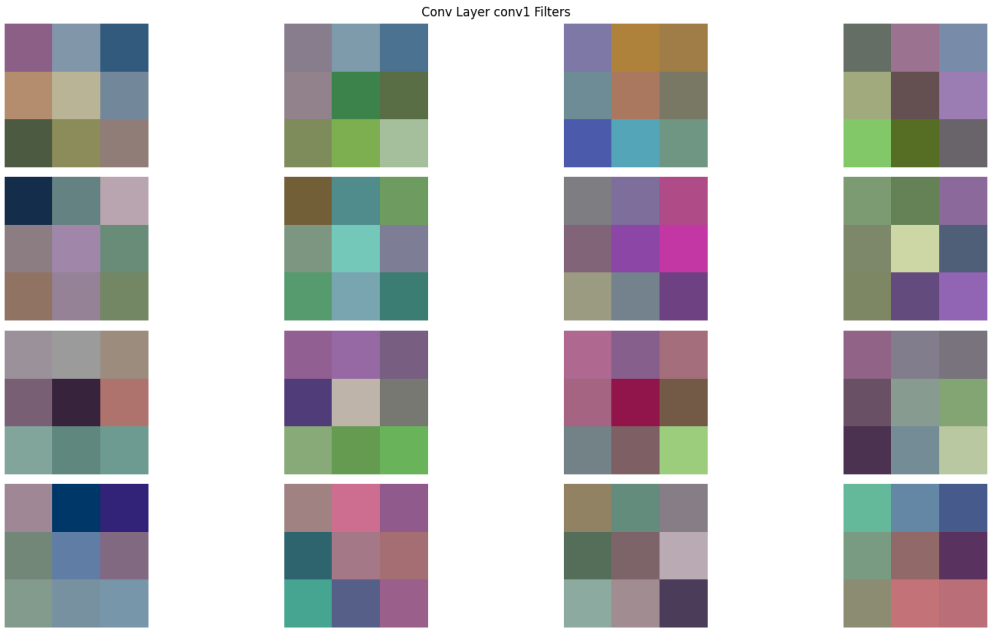

图 4-6 MediumCNN Conv Layer conv1 Filters

表 4-1 不同模型的性能比较(500epochs)

模型类型	测试准确率	验证准确率	训练时间(s)	收敛轮数
SimpleMLP	0.62	0.57	2.545	500 or more
DeepMLP	0.63	0.58	2.548	400
SimpleCNN	0.83	0.77	2.617	150
MediumCNN	0.92	0.89	3.287	100

性能差异原因

1. 空间信息利用:
   - MLP 缺陷: 展平操作破坏图像局部结构，无法有效利用像素空间相关性,导致参数量巨大且无法有效捕捉局部特征。
   - CNN 优势: CNN 通过使用卷积层实现了局部连接和权重共享。这意味着每个卷积核只与输入图像的局部区域连接，并且在整个图像上共享相同的权重。这种设计使得 CNN 能够有效地捕捉图像中的局部特征，如边缘、纹理等，同时减少了参数量和计算复杂度
2. 参数效率:
   - MLP 缺陷: 全连接导致参数量爆炸（如 CIFAR-10 的 32x32×3 输入需 3072xN 权重)。
   - CNN 优势: 权重共享和局部连接大幅减少参数（如 3x3 卷积核仅需 9 个参数/通道)。
3. 特征提取能力:
   - MLP 缺陷: 缺乏这种层次化的特征提取能力，只能通过增加网络深度来提高性能，容易导致过拟合。
   - CNN 优势: 通过卷积-池化层级自动提取低级→高级特征（边缘→纹理→物体部件)，对图像进行多尺度的特征提取，从而提高分类性能。。
4. 数据增强收益:
   - MLP 缺陷: 对输入顺序敏感，增强效果有限。
   - CNN 优势: 平移不变性使其更适合数据增强（如随机裁剪、翻转)，进一步提升泛化性和鲁棒性。

4.9 思考问题 3:

1. Bottleneck 结构的优势

- 减少参数量:

  Bottleneck 结构通过使用 1×1 卷积来减少网络的维度，从而降低模型的参数量。这有助于减少模型的复杂度和防止过拟合。对比传统 3×3 卷积堆叠，假设输入输出均为 256 通道:

  直接堆叠两个3×3卷积: 参数量 = $1256 \times 3 \times 3 \times 256 + 256 \times 3 \times 3 \times 256 = 1,179,648$。

  Bottleneck: 参数量 = $256 \times 1 \times 1 \times 64 + 64 \times 3 \times 3 \times 64 + 64 \times 1 \times 1 \times 256 = 70,400$。

  参数量减少约 16.7 倍。

- 提高计算效率:

  通过减少特征图的通道数, Bottleneck 结构降低了卷积操作的计算量, 从而提高了网络的计算效率。

  传统 3×3 卷积堆叠: $2 \times 256 \times 256 \times 3 \times 3 \times H \times W = 1,179,648 \times H \times W$。

  Bottleneck: $256 \times 64 \times 1 \times 1 \times H \times W + 64 \times 64 \times 3 \times 3 \times H \times W + 64 \times 256 \times 1 \times 1 \times H \times W = 70,400 \times H \times W$。

  计算量减少相同比例。

- 增强特征表达能力:

  1×1 卷积有助于增强特征的表达能力，促进不同通道之间的信息流动，从而提高网络的特征提取能力。

- 缓解梯度消失问题:

  在深层网络中，梯度消失是一个常见问题。Bottleneck 结构通过引入残差连接，可以在不显著增加参数量和计算复杂度的情况下增加网络的深度，有助于缓解梯度消失问题，使得深层网络的训练更加稳定。

2. 1×1 卷积的重要性

- 降维和升维:

  1×1 卷积可以在不改变特征图空间维度的情况下, 对特征图进行降维或升维操作。这在 Bottleneck 结构中尤为重要，可以在减少计算量的同时，保持特征图的表达能力。

- 特征融合:

  1×1 卷积可以对不同通道的特征进行融合, 这有助于网络学习到更加抽象和高级的特征表示。

- 控制参数量和计算复杂度:

  通过 1×1 卷积，可以灵活地控制网络的参数量和计算复杂度。在 Bottleneck 结构中，1×1 卷积用于减少和恢复特征图的通道数，从而在保持网络性能的同时，降低计算成本。

- 提高网络的灵活性:

  1×1 卷积提供了一种灵活的方式来调整网络的结构，使得网络可以更好地适应不同的任务和数据集。

3. 如何帮助控制网络的参数量和计算复杂度

- 减少参数量:

  在 Bottleneck 结构中，1×1 卷积首先将输入特征图的通道数降低，然后通过 3×3 卷积进行特征提取，最后再通过 1×1 卷积将通道数恢复。这种设计大幅减少了参数量，因为 1×1 卷积的参数量远小于 3×3 卷积。

- 降低计算复杂度:

  由于 1×1 卷积的计算量远小于 3×3 卷积，通过在 Bottleneck 结构中引入 1×1 卷积，可以显著降低网络的计算复杂度。

- 保持特征表达能力:

  尽管 1×1 卷积减少了特征图的通道数，但它通过线性组合的方式保留了重要的特征信息，从而在降低计算量的同时，保持了网络的特征表达能力。

4.10 探索问题 1:

4.10.1 残差连接的实现

在 ResidualBlock 类中，残差连接是通过将输入 x 与经过卷积和批归一化处理后的输出 out 相加来实现的。具体步骤如下:

1. 输入保存: 将输入 x 保存为 residual，以便后续与输出相加。
2. 第一层卷积和批归一化: 对输入 x 进行卷积操作（self.conv1），然后进行批归一化（self.bn1），最后通过 ReLU 激活函数（self.relu）。
3. 第二层卷积和批归一化: 对上一步的输出进行卷积操作（self.conv2），然后进行批归一化（self.bn2）。
4. 残差连接: 将经过处理的输出 out 与保存的输入 residual 通过 shortcut 路径相加。
5. 激活函数: 对相加后的输出再次通过 ReLU 激活函数（self.relu）。

4.10.2 输入和输出通道数不匹配时的处理

当输入和输出通道数不匹配时，需要通过 shortcut 路径进行调整。具体处理方式如下:

1. 判断条件: 在 ResidualBlock 的构造函数中，通过判断 stride != 1 或 in_channels != out_channels 来确定是否需要调整 shortcut 路径。
2. 卷积和批归一化: 如果需要调整，则在 shortcut 路径中添加一个卷积层（nn.Conv2d）和一个批归一化层（nn.BatchNorm2d）。卷积层的参数设置为:
   - 输入通道数: in_channels
   - 输出通道数: out_channels
   - 卷积核大小: 1x1
   - 步长: stride
   - 偏置项: False

通过这种方式，可以确保输入和输出的通道数以及特征图的大小相匹配，从而实现残差连接。

通过这种灵活的处理方式，SimpleResNet 模型可以适应不同的网络结构和参数设置，从而实现高效的残差学习。

表 4-2 模型的复杂度

模型	参数量	批(128 样本)推理时间
SimpleMLP	1,578,506	0.02ms
DeepMLP	3,809,034	0.13ms
SimpleCNN	168,650	0.13ms
MediumCNN	2,169,770	1.35ms
VGGStyleNet	3,251,018	6.40ms
SimpleResNet	175,258	1.88ms

4.11 分析问题 5:

4.11.1 性能比较

- **准确率**: ResNet 网络通常比 VGG 网络具有更高的分类准确率。这是因为残差连接使得网络能够更有效地学习到复杂的特征表示。
- **训练收敛速度**: ResNet 网络的训练收敛速度通常比 VGG 网络更快。这是因为残差连接缓解了梯度消失问题，使得网络更容易优化。
- **泛化能力**: ResNet 网络在处理未见过的数据时具有更好的泛化能力。这是因为残差连接促进了信息的流动，使得网络能够学习到更一般性的特征。

4.11.2 残差连接带来的优势

- **缓解梯度消失问题**: 残差连接允许梯度直接从输出层流向输入层，从而缓解了深层网络中的梯度消失问题。这使得网络更容易训练，并且可以构建更深的网络。
- **促进信息流动**: 残差连接促进了不同层之间的信息流动，使得网络能够更有效地学习到特征表示。这有助于提高网络的分类性能和泛化能力。
- **减少过拟合风险**: 由于残差连接的存在，ResNet 网络通常需要更少的参数来达到相同的性能水平。这减少了过拟合的风险，并提高了模型的鲁棒性。
- **模块化设计**: 残差块作为基本构建单元，使得 ResNet 网络的设计更加模块化和灵活。这使得网络的构建和扩展更加容易。

4.12 分析问题 6:

4.12.1 参数量的影响

- **内存占用**: 模型的参数量决定了其在内存中的占用空间。较大的参数量意味着更高的内存需求，这对于资源受限的设备（如移动设备、嵌入式系统）来说是一个挑战。
- **训练时间**: 参数量越大，模型的训练时间通常越长。这是因为更多的参数需要更多的计算资源和时间来优化。

- **过拟合风险**：较大的参数量增加了模型的过拟合风险。如果模型过于复杂，它可能会记住训练数据中的细节，而不是学习到一般性的特征。

4.12.2 推理时间的影响

- **实时性要求**：对于需要实时响应的应用（如自动驾驶、视频监控），推理时间是一个关键因素。如果模型的推理时间过长，它可能无法满足实时性要求。
- **用户体验**：在用户交互应用（如智能手机应用）中，推理时间直接影响用户体验。较长的推理时间可能导致用户等待时间过长，降低用户满意度。
- **资源消耗**：推理时间越长，模型对计算资源的消耗越大。这对于资源受限的设备来说是一个挑战，可能导致设备过热、电池消耗过快等问题。

4.12.3 性能与效率的平衡

在选择模型时，需要在性能（如分类准确率）和效率（如参数量、推理时间）之间找到平衡

1. 任务需求分析：首先，需要明确应用的任务需求。对于需要高精度的任务（如医学图像诊断），可能需要牺牲一些效率来获得更好的性能。而对于需要实时响应的任务（如自动驾驶），可能需要优先考虑效率。
2. 模型选择与优化：根据任务需求，选择合适的模型架构。例如，对于图像分类任务，CNN 通常比 MLP 具有更好的性能。然后，可以通过模型压缩、剪枝、量化等技术来减少参数量和推理时间，同时尽量保持性能。
3. 硬件适配：考虑目标设备的硬件特性，选择适合的模型和优化策略。例如，对于移动设备，可以选择轻量级的模型架构（如 MobileNet）或使用模型量化技术来减少计算资源的消耗。
4. 实验与评估：通过实验和评估来找到最佳的平衡点。可以使用不同的模型架构、优化策略和硬件配置进行实验，并根据性能和效率指标进行评估和比较。

4.12.4 示例分析

根据计算出的数据，我们可以对不同模型的性能和效率进行分析：

- **SimpleMLP**：参数量较小（1,578,506），推理时间较短（0.02ms），但性能可能较差。适合对性能要求不高、需要快速响应的任务。
- **DeepMLP**：参数量较大（3,809,034），推理时间较长（0.13ms），性能可能有所提升，但过拟合风险增加。适合对性能有一定要求、但对实时性要求不高的任务。
- **SimpleCNN**：参数量较小（168,650），推理时间较短（0.13ms），性能可能优于 SimpleMLP。适合对性能和效率都有要求的任务。
- **MediumCNN**：参数量较大（2,169,770），推理时间较长（1.35ms），性能可能优于 SimpleCNN。适合对性能要求较高、但对实时性要求不太高的任务。

- **VGGStyleNet**: 参数量较大 (3,251,018)，推理时间较长 (6.40ms)，性能可能较好，但对资源要求较高。适合对性能要求高、但对实时性要求不高的任务。
- **SimpleResNet**: 参数量较小 (175,258)，推理时间较短 (1.88ms)，性能可能优于 SimpleCNN 和 MediumCNN。适合对性能和效率都有较高要求的任务。

根据以上分析，如果任务对性能和效率都有较高要求，SimpleResNet 可能是一个较好的选择。它具有较小的参数量和较短的推理时间，同时性能可能优于其他轻量级模型。如果任务对性能要求极高，但对实时性要求不高，VGGStyleNet 可能是一个选择，但需要注意其较高的资源消耗。

4.13 探索问题 2:

1. 采用轻量级卷积设计

深度可分离卷积 (Depthwise Separable Convolution)

- 原理: 将传统卷积分解为深度卷积 (Depthwise Convolution) 和逐点卷积 (Pointwise Convolution)。深度卷积对每个通道单独进行卷积，逐点卷积使用 1×1 卷积核将通道进行组合。
- 优势: 显著减少参数量和计算复杂度，同时保持较好的特征提取能力。
- 应用: MobileNet 系列模型广泛采用深度可分离卷积，在移动设备上实现了高效的图像分类和目标检测。

组卷积 (Group Convolution)

- 原理: 将输入通道分成若干组，每组独立进行卷积操作，然后通过逐点卷积将各组的特征进行组合。
- 优势: 减少参数量和计算复杂度，同时促进不同通道之间的信息交流。
- 应用: ShuffleNet 系列模型使用组卷积和通道混洗 (Channel Shuffle) 技术，在保持高性能的同时提高了模型的效率。

2. 优化网络深度与宽度

平衡深度与宽度

- 深度: 更深的网络可以学习到更抽象的特征，但增加了训练难度和计算复杂度。对于移动设备，应避免过度增加网络深度。
- 宽度: 更宽的网络 (更多通道) 可以捕获更多特征，但参数量和计算复杂度也随之增加。应根据任务需求和硬件限制，合理设置网络宽度。
- 策略: 采用“窄而深”或“宽而浅”的网络结构，结合轻量级卷积设计，在保证性能的前提下减少参数量和计算复杂度。

3. 引入高效的特征增强机制

注意力机制（Attention Mechanism）

- 原理：通过学习权重来增强重要特征，抑制不重要特征。例如，SENet（Squeeze-and-Excitation Network）使用通道注意力机制，通过全局平均池化和全连接层来计算每个通道的权重。
- 优势：在不显著增加参数量和计算复杂度的情况下，提高模型的特征提取能力和分类性能。
- 应用：在移动设备模型中引入轻量级的注意力机制，如 SE 模块，可以增强模型对重要特征的关注，提高性能。

特征融合（Feature Fusion）

- 原理：将不同层或不同尺度的特征进行融合，以获得更丰富的特征表示。例如，U-Net 使用跨层连接将编码器的特征与解码器的特征进行融合。
- 优势：提高模型的表达能力和鲁棒性，特别是在处理多尺度目标时。
- 应用：在移动设备模型中，可以采用轻量级的特征融合策略，如使用 1×1 卷积进行特征降维和融合，以减少计算复杂度。

4.14 模型比较与分析

图 4-7 Model Comparison

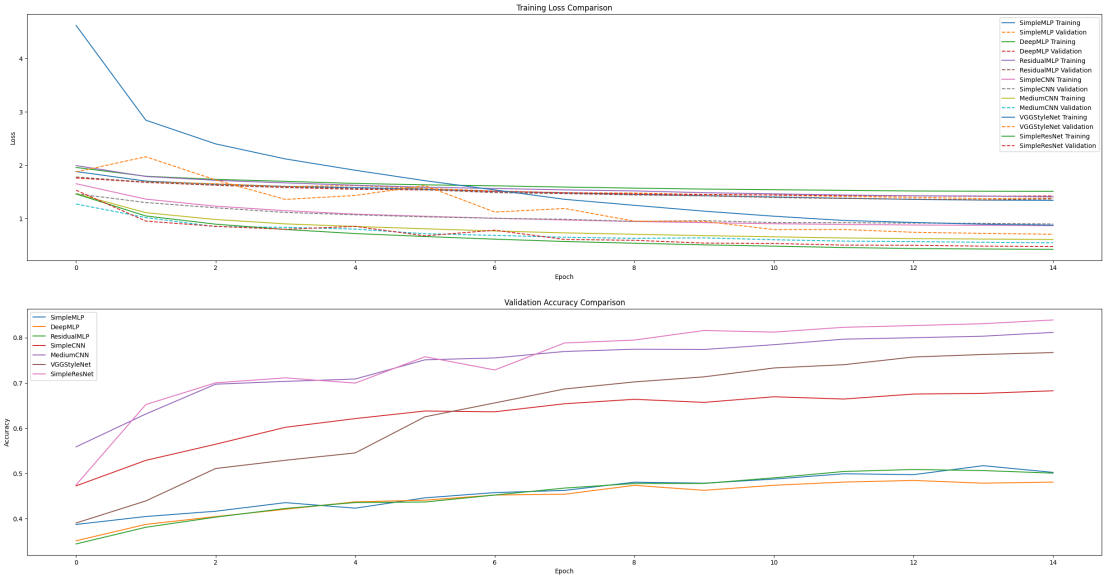

图 4-8 Training Loss Comparison and Validation Accuracy Comparison

5 创新探索任务的设计、实现与结果

5.1 模型改进:

对任一模型进行修改和改进，提高其在 CIFAR-10 上的性能。

5.1.1 改进 1: 更换网络架构为 ResNet18

```
class ResBlock(nn.Module):
    def __init__(self, inchannel, outchannel, stride=1):
        super(ResBlock, self).__init__()

        self.block = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=stride, padding=1,
bias=False),
            nn.BatchNorm2d(outchannel),
            nn.ReLU(inplace=False),
            nn.Conv2d(outchannel, outchannel, kernel_size=3, stride=1, padding=1,
bias=False),
            nn.BatchNorm2d(outchannel)
        )

        self.shortcut = nn.Sequential()
        if stride != 1 or inchannel != outchannel:
            self.shortcut = nn.Sequential(
                nn.Conv2d(inchannel, outchannel, kernel_size=1, stride=stride,
bias=False),
                nn.BatchNorm2d(outchannel)
            )

    def forward(self, x):

        out = self.block(x)
        out = out + self.shortcut(x)
        out = F.relu(out, inplace=False)

        return out

class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):

        super(ResNet, self).__init__()
        self.inchannel = 64
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.BatchNorm2d(64),
            nn.CELU(alpha=0.075, inplace=False)
        )
        self.pool1 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self.make_layer(block, 64, num_blocks[0], stride=1)
```

```
        self.layer2 = self.make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self.make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self.make_layer(block, 512, num_blocks[3], stride=2)
        self.fc = nn.Linear(512, num_classes)

    def make_layer(self, block, channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.inchannel, channels, stride))
            self.inchannel = channels
        return nn.Sequential(*layers)

    def forward(self, x):

        out = self.conv1(x)
        out = self.pool1(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.adaptive_avg_pool2d(out, (1, 1))
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

def ResNet18():
    return ResNet(ResBlock, [2, 2, 2, 2])
```

5.1.2 改进 2: 正则化

transforms.Resize((224, 224))是为了适应 ResNet18 的输入大小

```
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5), # 随机水平翻转
    transforms.Resize((224, 224)),
    transforms.RandomCrop(224, padding=28), # 随机裁剪
    Cutout(p=0.5), # 使用Cutout正则化
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

test_transform = transforms.Compose([
  transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

5.1.3 改进 3：减小输入规模

既然 CIFAR-10 数据集的输入大小是 32x32，为什么还要把它放大到 224x224 适应 ResNet 的输入呢?因此这里我们改造 ResNet18 输入部分的卷积，将 7x7 卷积改为 3x3 的卷积，同时去掉用于降维的池化层。改造后的 ResNet 结构如下：

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):

        super(ResNet, self).__init__()
        self.inchannel = 64
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.CELU(alpha=0.075, inplace=False)
        )
        self.layer1 = self.make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self.make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self.make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self.make_layer(block, 512, num_blocks[3], stride=2)
        self.fc = nn.Linear(512, num_classes)

    def make_layer(self, block, channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(block(self.inchannel, channels, stride))
            self.inchannel = channels
        return nn.Sequential(*layers)

    def forward(self, x):

        out = self.conv1(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.adaptive_avg_pool2d(out, (1, 1))
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

5.1.4 其他改进方向

1. 更换激活函数

$ReLU$是非常经典的激活函数，但是它本身还存在梯度消失、丢失负样本信息以及梯度不连续的问题。这里我们考虑将$ReLU$函数更换成$CELU$函数。$CELU$函数的数学形式为：

$$CELU(x) = \max(0, x) + \min\left(0, \alpha\left(e^{\frac{x}{\alpha}} - 1\right)\right) \tag{5-1}$$

它在输入为负数时会返回较小的负值，鼓励网络学习负样本的结果，此外，它的梯度连续，可以避免在不连续点处的跳动。

这里我们采取保守的策略，选择一个较小的α。可以将其近似看作一个梯度连续的$ReLU$。

2. 引入 StepLR 学习率调度器

观察先前训练，发现在后期模型基本收敛之后依然会出现 loss 跳动的情况。因此这里我们放弃 CosineAnnealingLR 改用 StepLR 学习率调度器，每 10 epoch 将学习率减小为原来的一半。这样可以有效保证训练后期的稳定性。

5.1.5 结果展示

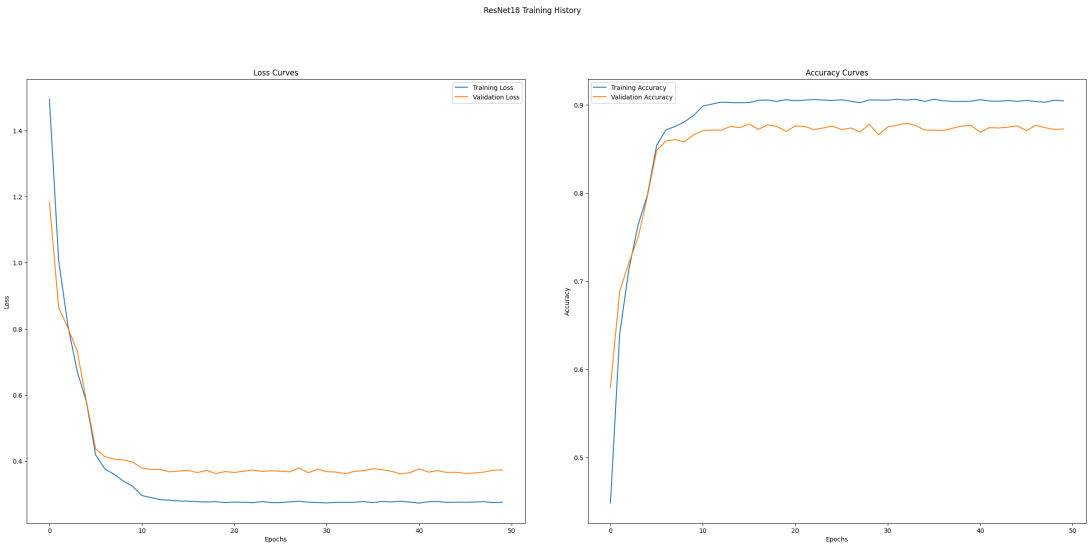

图 5-1 ResNet18 Training History

训练 50 epoch 后，ResNet18 在训练集与测试集上的准确率分别达到了 90.54% 与 87.34%。可以发现收敛速度较快并保持相对稳定，且训练集与测试集的准确率差距较小，说明模型没有出现过拟合现象。

5.2 可视化分析:

使用 netron 网站进行模型可视化，直观展现模型的结构和参数量。

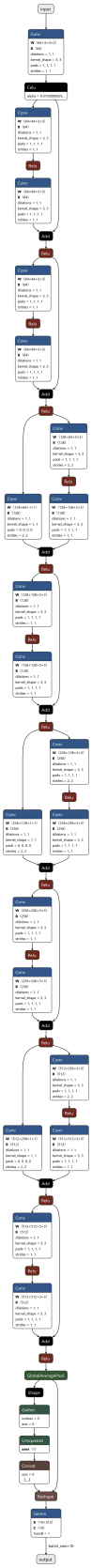

图 5-2 ResNet18 Structure

5.3 迁移学习:

使用 torchvision.models 中的 ResNet18 预训练模型进行迁移学习，冻结前几层的参数，只训练最后几层的参数。通过迁移学习，我们可以利用在大规模数据集上预训练的模型来提高在小规模数据集上的性能。

```
import torchvision.models as models

# 加载预训练的 ResNet18 模型
model = models.resnet18(pretrained=True)

# 冻结所有参数
for param in model.parameters():
    param.requires_grad = False

# 替换最后的全连接层以适应 CIFAR-10 的 10 个类别
model.fc = nn.Linear(512, 10)
model = model.to(device)

optimizer = optim.Adam(model.fc.parameters(), lr=learning_rate)  # 只优化全连接层的参数

# 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
)
```

5.4 对抗性样本:

5.5 自监督学习:

6 结论与思考

通过本次实验，我们深入理解了多层感知机（MLP）和卷积神经网络（CNN）的基本原理、实现方法以及它们在图像分类任务中的应用。以下是实验的主要结论和思考：

6.1 主要结论

1. **CNN 在图像分类任务中表现更优**：
   - **SimpleCNN** 和 **MediumCNN** 模型在 CIFAR-10 数据集上的准确率显著高于 **SimpleMLP** 和 **DeepMLP**。CNN 达到了 92% 的测试准确率，而最佳的 MLP 模型（DeepMLP）仅实现了 63% 的准确率。这表明 CNN 通过局部连接和权重共享，能够更有效地捕捉图像中的局部特征和空间结构。
2. **残差连接提升网络性能**：
   - **SimpleResNet** 模型在准确率和训练收敛速度上优于 **VGGStyleNet**。残差连接通过缓解梯度消失问题，使得网络能够更深，从而学习到更抽象的特征表示。
3. **模型复杂度与性能的权衡**：
   - **DeepMLP** 和 **VGGStyleNet** 虽然具有更高的参数量和计算复杂度，但并不一定带来更好的性能。相比之下，**SimpleResNet** 通过合理的架构设计，在保持较低参数量的同时，实现了更高的准确率。
4. **数据增强和正则化技术的重要性**：
   - 使用数据增强（如随机裁剪、水平翻转）和正则化技术（如 Dropout、BatchNorm）可以显著提高模型的泛化能力，减少过拟合风险。

6.2 思考与展望

1. **网络深度与宽度的平衡**：
   - 更深的网络能够学习到更抽象的特征，但训练难度也随之增加。更宽的网络可以捕获更多特征，但参数量也会增加。如何在深度和宽度之间找到最佳平衡点，是设计高效 CNN 架构的关键。
2. **跳跃连接的多样性**：
   - 除了 ResNet 的残差连接，DenseNet 的密集连接和 U-Net 的跨层连接等跳跃连接方式也为网络设计提供了新的思路。这些连接方式如何影响网络的性能和训练动态，值得进一步探索。
3. **特征增强机制**：

- 注意力机制（如 SENet 的通道注意力）和特征融合等技术可以进一步提高网络的特征提取能力。如何将这些机制有效地集成到现有网络架构中，是一个值得研究的方向。

4. 高效卷积设计：
   - 深度可分离卷积（如 MobileNet）和组卷积（如 ShuffleNet）等高效卷积设计可以显著减少参数量和计算复杂度，同时保持较好的性能。这些设计如何适应不同的任务需求和硬件环境，需要进一步的研究和实践。
5. 模型调优与问题解决：
   - 在实际应用中，模型的性能不仅取决于架构设计，还受到训练策略、超参数设置等因素的影响。如何系统地进行模型调优，解决训练过程中遇到的问题（如过拟合、欠拟合），是深度学习实践中的重要课题。

6.3 总结

通过本次实验，我们不仅掌握了 MLP 和 CNN 的基本原理和实现方法，还深入理解了不同网络结构对模型性能的影响。残差连接、高效卷积设计等先进技术的应用，使得 CNN 在图像分类任务中表现出色。然而，网络设计中的诸多挑战（如深度与宽度的平衡、特征增强机制的选择等）仍然需要进一步的研究和探索。希望通过不断的学习和实践，能够进一步提升深度学习模型的设计与应用能力。

参考文献

[1] Ronald J. Williams David E. Rumelhart Geoffrey E. Hinton. Learning representations by back-propagating errors. Nature, 1986, 323: 533-536

[2] Geoffrey E. Hinton Alex Krizhevsky Ilya Sutskever. ImageNet Classification with Deep Convolutional Neural Networks. NIPS, 2012, 25(2): 1097-1105

[3] Andrew Zisserman Karen Simonyan. Very Deep Convolutional Networks for Large-Scale Image Recognition. ICLR, 2015

[4] Shaoqing Ren, Jian Sun Kaiming He Xiangyu Zhang. Deep Residual Learning for Image Recognition. CVPR, 2016

[5] Sergey Zagoruyko, Nikos Komodakis. Wide Residual Networks. BMVA, 2016

[6] Laurens van der Maaten, Kilian Q. Q. Weinberger Gao Huang Zhuang Liu. Densely Connected Convolutional Networks. CVPR, 2017

[7] Thomas Brox Olaf Ronneberger Philipp Fischer. U-Net: Convolutional Networks for Biomedical Image Segmentation. MICCAI, 2015

[8] Gang Sun Jie Hu Li Shen. Squeeze-and-Excitation Networks. CVPR, 2018

[9] Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobi Adam, Hartwig Adam, Cloud TPUs Andrew G. Howard Menglong Zhu. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CVPR, 2017

[10] Mengxiao Lin, Jian Sun Xiangyu Zhang Xinyu Zhou. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. CVPR, 2018