南開大學
Nankai University

南开大学

计算机学院和密码与网络空间安全学院

《深度学习及应用》课程作业

Lab02：卷积神经网络

姓名：陆皓喆

学号：2211044

专业：信息安全

指导教师：李重仪

2025 年 5 月 13 日

目录

1 实验目的

1. 掌握 PyTorch 框架基础算子（Convolution, Pooling, Activation layers）；
2. 学会使用 PyTorch 搭建简单的卷积神经网络来训练 Cifar10 或者 Cifar100 数据集；
3. 了解经典的卷积神经网络结构。

2 实验要求

1. 老师提供的原始版本卷积网络结构（可用 print(net) 打印，复制文字或截图皆可）、在 Cifar 10 或者 Cifar 100 验证集上的训练 loss 曲线以及准确度曲线图；
2. 个人实现的**微型 ResNet 网络结构**、在 Cifar 10 或者 Cifar 100 验证集上的训练 loss 曲线以及准确度曲线图；
3. 个人实现的**微型 DenseNet 网络结构**、在 Cifar 10 或者 Cifar 100 验证集上的训练 loss 曲线以及准确度曲线图；
4. 个人实现的**带有 SE 结构的微型 ResNet 网络结构**、在 Cifar 10 或者 Cifar 100 验证集上的训练 loss 曲线以及准确度曲线图。

3 实验原理

简单来说，本次实验的任务就是，通过设计不同类型的 CNN 网络，来完成在 cifar10 和 cifar100 数据集上的分类任务。

cifar 数据集被广泛运用于计算机视觉的训练集和验证集，cifar10 数据集一共拥有 10 种类别，相对来说分类较为简单；cifar100 在 cifar10 的基础上，拥有 100 种类型，相对来说分类难度较大，对神经网络的要求会比较高。

目前已经存在许多 CNN 网络，通过课程的学习和论文的阅读，我大概了解到以下这些 CNN 网络：VGGNet[1]、ResNet[2]、ResNeXt[3]、DenseNet[4]、Res2Net[5]、Self-Calibrated Network[6]、SKNet[7]、SENet[8]、CA[9] 和 Non-Local Net[10] 等。

由于时间有限，本次实验只能根据作业要求，挑选部分经典的网络进行复现。

4 实验环境

本次实验本人继续延续之前的实验，在 4 卡 A100 的 GPU（linux）上进行实验。首先，通过以下命令行创建虚拟环境并进入，然后安装我们的需求库就可以了。

```
python -m venv CNN
source CNN/bin/activate
pip install -r requirements.txt
```

我们运行 ___INIT___.py，输出以下信息：

```
[notice] A new release of pip is available: 24.3.1 -> 25.1.1
[notice] To update, run: pip install --upgrade pip
(CNN) (base) root@tomorin:/home/Deep-Learning/CNN# python3 /home/Deep-Learning/CNN/__init__.py
Is GPU available?:  True
Using PyTorch version: 2.7.0+cu126  Device: cuda
Current GPU device id: 0
Current GPU device name: NVIDIA A100-PCIE-40GB
(CNN) (base) root@tomorin:/home/Deep-Learning/CNN#
```

图 4.1: 实验环境配置

这样，我们就完成了本次实验环境的配置。

5 数据集选取

本次实验，实验文档提供给我们的是 cifar10 数据集，但是我认为该数据集太小，可能不能体现出我后续设计出的 CNN 网络的提升，所以我选择使用 **cifar10 数据集**和 **cifar100 数据集**。相比于 cifar10 数据集，cifar100 在类别上，每个类有 **600** 张大小为 32×32 的彩色图像，其中 **500** 张作为训练集，**100** 张作为测试集。如图5.2所示，展示了 cifar100 的 100 种类别。

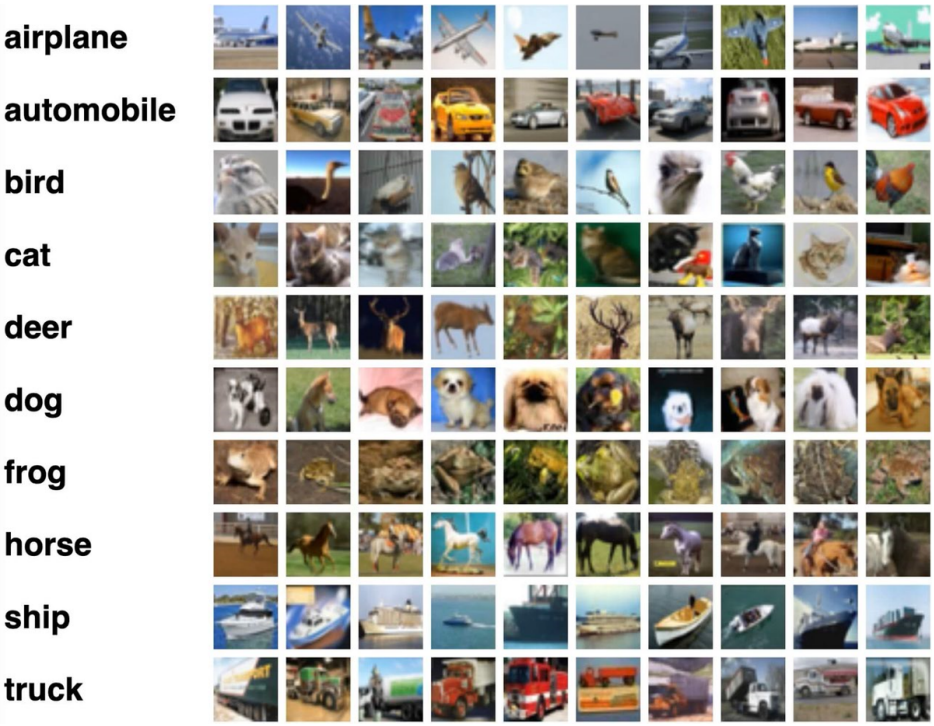

图 5.2: cifar100 数据集概览

由于 cifar10 数据集只有十种类别，而 cifar100 拥有 100 种分类，所以显而易见，后者的分类难度要远远大于前者。**我预估该 cifar100 上的训练效果可能会较差，所以这也是我选取两个数据集同步进行测试的主要原因**。

相应的，由于修改数据集后，我们需要对原先的网络结构进行一定的修改，使得最后输出的是 100 个类别。这一部分，我们在后续章节中进行介绍。

6 复现——运行并测试 CNN Base 代码

6.1 展示原始版本卷积网络结构

我们使用 print(net) 来对我们的网络结构进行打印，具体网络结构如下所示。

```
BaseCNN(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
  ↪  ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

我们发现，该经典的卷积神经网络主要由**卷积层**、**池化层**和**全连接层**构成。

首先，**第一个卷积层**运用 5×5 的卷积核，将 3 通道的输入图像映射成 6 通道的特征图；**最大池化层**采用 2×2 的窗口，能对特征图进行下采样，进而减少参数数量；然后**第二个卷积层**把把 6 通道的特征图转换为 16 通道的特征图。后面是三个全连接层，**第一个全连接层**把经过展平处理的 400 维特征向量映射成 120 维的特征向量；**第二个全连接层**将 120 维的特征向量映射为 84 维的特征向量；**最后一个全连接层**把 84 维的特征向量映射到 10 个输出类别，这样就可以完成我们的 10 分类任务了。

6.2 修改结构用来适配 cifar100

从上面的 CNN 框架，我们可以看出，实际上上述框架是针对于 cifar10 来设计的，为了适配 cifar100 的分类任务，我们还需要对原先的网络做一定的改进。

本人把最后一层的输出修改为 100，这样才可以完成对应的 100 分类任务。修改后的网络结构如下所示：

```
BaseCNN(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
  ↪  ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=100, bias=True)
)
```

6.3 CNN Base 实验结果展示

通过上述的修改，我们就可以开始我们的训练了。训练的脚本基本沿用了上一次实验的 train.py，main.py 中编写了函数，对我们的基础 CNN 模型进行训练与测试。我们在此首先固定选用 SGD 模型

作为优化模型，选定学习率为 0.001，momentum 为 0.9。此处优化参数的修改对实验结果的影响留到后面再进行分析与实验。下面本人分别展示在两个数据集上的训练结果。

6.3.1 cifar10 数据集

展示一下本人的训练日志，如图6.3所示。

```
(CNN) (base) root@tomorin:/home/Deep-Learning/CNN# python3 /home/Deep-Learning/CNN/main_cifar10.py
Epoch 1/50, Train Loss: 1.6568, Train Accuracy: 0.3919, Valid Loss: 1.4452, Valid Accuracy: 0.4733, Time: 22.06s
Epoch 2/50, Train Loss: 1.3779, Train Accuracy: 0.5045, Valid Loss: 1.3208, Valid Accuracy: 0.5267, Time: 21.06s
Epoch 3/50, Train Loss: 1.2504, Train Accuracy: 0.5531, Valid Loss: 1.2451, Valid Accuracy: 0.5615, Time: 21.71s
Epoch 4/50, Train Loss: 1.1702, Train Accuracy: 0.5843, Valid Loss: 1.1976, Valid Accuracy: 0.5740, Time: 21.66s
Epoch 5/50, Train Loss: 1.1034, Train Accuracy: 0.6092, Valid Loss: 1.1368, Valid Accuracy: 0.5962, Time: 21.85s
Epoch 6/50, Train Loss: 1.0476, Train Accuracy: 0.6294, Valid Loss: 1.1035, Valid Accuracy: 0.6102, Time: 21.51s
Epoch 7/50, Train Loss: 0.9988, Train Accuracy: 0.6470, Valid Loss: 1.0952, Valid Accuracy: 0.6197, Time: 22.04s
Epoch 8/50, Train Loss: 0.9595, Train Accuracy: 0.6614, Valid Loss: 1.0725, Valid Accuracy: 0.6230, Time: 21.88s
Epoch 9/50, Train Loss: 0.9149, Train Accuracy: 0.6765, Valid Loss: 1.0908, Valid Accuracy: 0.6153, Time: 22.12s
Epoch 10/50, Train Loss: 0.8820, Train Accuracy: 0.6878, Valid Loss: 1.0735, Valid Accuracy: 0.6351, Time: 22.22s
Epoch 11/50, Train Loss: 0.8519, Train Accuracy: 0.6970, Valid Loss: 1.0587, Valid Accuracy: 0.6365, Time: 22.14s
Epoch 12/50, Train Loss: 0.8221, Train Accuracy: 0.7077, Valid Loss: 1.0887, Valid Accuracy: 0.6265, Time: 21.44s
Epoch 13/50, Train Loss: 0.7942, Train Accuracy: 0.7197, Valid Loss: 1.1002, Valid Accuracy: 0.6289, Time: 22.26s
Epoch 14/50, Train Loss: 0.7719, Train Accuracy: 0.7253, Valid Loss: 1.0701, Valid Accuracy: 0.6359, Time: 22.03s
Epoch 15/50, Train Loss: 0.7531, Train Accuracy: 0.7324, Valid Loss: 1.1208, Valid Accuracy: 0.6316, Time: 21.66s
Epoch 16/50, Train Loss: 0.7261, Train Accuracy: 0.7405, Valid Loss: 1.0906, Valid Accuracy: 0.6352, Time: 22.21s
Epoch 17/50, Train Loss: 0.7076, Train Accuracy: 0.7475, Valid Loss: 1.0889, Valid Accuracy: 0.6429, Time: 21.83s
Epoch 18/50, Train Loss: 0.6858, Train Accuracy: 0.7545, Valid Loss: 1.1473, Valid Accuracy: 0.6289, Time: 21.76s
Epoch 19/50, Train Loss: 0.6686, Train Accuracy: 0.7614, Valid Loss: 1.1246, Valid Accuracy: 0.6361, Time: 21.98s
Epoch 20/50, Train Loss: 0.6514, Train Accuracy: 0.7672, Valid Loss: 1.1592, Valid Accuracy: 0.6403, Time: 21.92s
Epoch 21/50, Train Loss: 0.6365, Train Accuracy: 0.7731, Valid Loss: 1.2075, Valid Accuracy: 0.6284, Time: 21.20s
Epoch 22/50, Train Loss: 0.6180, Train Accuracy: 0.7765, Valid Loss: 1.1782, Valid Accuracy: 0.6299, Time: 21.53s
Epoch 23/50, Train Loss: 0.6042, Train Accuracy: 0.7820, Valid Loss: 1.2181, Valid Accuracy: 0.6255, Time: 20.96s
Epoch 24/50, Train Loss: 0.5869, Train Accuracy: 0.7891, Valid Loss: 1.2230, Valid Accuracy: 0.6321, Time: 21.56s
Epoch 25/50, Train Loss: 0.5734, Train Accuracy: 0.7952, Valid Loss: 1.2675, Valid Accuracy: 0.6293, Time: 21.17s
Epoch 26/50, Train Loss: 0.5602, Train Accuracy: 0.7977, Valid Loss: 1.2983, Valid Accuracy: 0.6259, Time: 21.24s
Epoch 27/50, Train Loss: 0.5441, Train Accuracy: 0.8007, Valid Loss: 1.3174, Valid Accuracy: 0.6269, Time: 20.96s
Epoch 28/50, Train Loss: 0.5287, Train Accuracy: 0.8100, Valid Loss: 1.3234, Valid Accuracy: 0.6252, Time: 20.73s
Epoch 29/50, Train Loss: 0.5146, Train Accuracy: 0.8139, Valid Loss: 1.3747, Valid Accuracy: 0.6195, Time: 20.65s
Epoch 30/50, Train Loss: 0.5042, Train Accuracy: 0.8184, Valid Loss: 1.3776, Valid Accuracy: 0.6223, Time: 20.53s
Epoch 31/50, Train Loss: 0.4925, Train Accuracy: 0.8234, Valid Loss: 1.4011, Valid Accuracy: 0.6150, Time: 20.44s
Epoch 32/50, Train Loss: 0.4828, Train Accuracy: 0.8258, Valid Loss: 1.4712, Valid Accuracy: 0.6182, Time: 20.78s
Epoch 33/50, Train Loss: 0.4687, Train Accuracy: 0.8293, Valid Loss: 1.4949, Valid Accuracy: 0.6141, Time: 20.62s
Epoch 34/50, Train Loss: 0.4605, Train Accuracy: 0.8347, Valid Loss: 1.5278, Valid Accuracy: 0.6116, Time: 21.74s
Epoch 35/50, Train Loss: 0.4506, Train Accuracy: 0.8360, Valid Loss: 1.5737, Valid Accuracy: 0.6084, Time: 21.69s
Epoch 36/50, Train Loss: 0.4413, Train Accuracy: 0.8397, Valid Loss: 1.5899, Valid Accuracy: 0.6117, Time: 21.57s
Epoch 37/50, Train Loss: 0.4330, Train Accuracy: 0.8432, Valid Loss: 1.6368, Valid Accuracy: 0.6091, Time: 21.38s
Epoch 38/50, Train Loss: 0.4188, Train Accuracy: 0.8474, Valid Loss: 1.6057, Valid Accuracy: 0.6087, Time: 21.42s
Epoch 39/50, Train Loss: 0.4105, Train Accuracy: 0.8516, Valid Loss: 1.6772, Valid Accuracy: 0.6136, Time: 21.08s
Epoch 40/50, Train Loss: 0.4040, Train Accuracy: 0.8544, Valid Loss: 1.7391, Valid Accuracy: 0.6025, Time: 21.09s
Epoch 41/50, Train Loss: 0.3959, Train Accuracy: 0.8557, Valid Loss: 1.7855, Valid Accuracy: 0.6012, Time: 20.86s
Epoch 42/50, Train Loss: 0.3876, Train Accuracy: 0.8593, Valid Loss: 1.7710, Valid Accuracy: 0.6111, Time: 20.80s
Epoch 43/50, Train Loss: 0.3731, Train Accuracy: 0.8647, Valid Loss: 1.7751, Valid Accuracy: 0.6116, Time: 20.72s
Epoch 44/50, Train Loss: 0.3670, Train Accuracy: 0.8666, Valid Loss: 1.8784, Valid Accuracy: 0.6096, Time: 21.09s
Epoch 45/50, Train Loss: 0.3677, Train Accuracy: 0.8651, Valid Loss: 1.8901, Valid Accuracy: 0.6093, Time: 21.04s
Epoch 46/50, Train Loss: 0.3534, Train Accuracy: 0.8714, Valid Loss: 1.9025, Valid Accuracy: 0.6025, Time: 21.45s
Epoch 47/50, Train Loss: 0.3466, Train Accuracy: 0.8739, Valid Loss: 1.9548, Valid Accuracy: 0.6032, Time: 20.78s
Epoch 48/50, Train Loss: 0.3427, Train Accuracy: 0.8745, Valid Loss: 1.9891, Valid Accuracy: 0.5979, Time: 21.44s
Epoch 49/50, Train Loss: 0.3329, Train Accuracy: 0.8786, Valid Loss: 2.0245, Valid Accuracy: 0.6032, Time: 21.16s
Epoch 50/50, Train Loss: 0.3336, Train Accuracy: 0.8798, Valid Loss: 2.0957, Valid Accuracy: 0.6061, Time: 20.78s
Training Finished
Model Saved
```

图 6.3: cnn-base 在 cifar10 上的训练结果

训练结束，我将训练结果和训练可视化结果存储到了 results/cifar10/文件夹下，在此展示一下训练 loss 曲线和准确度曲线，如图6.4所示。

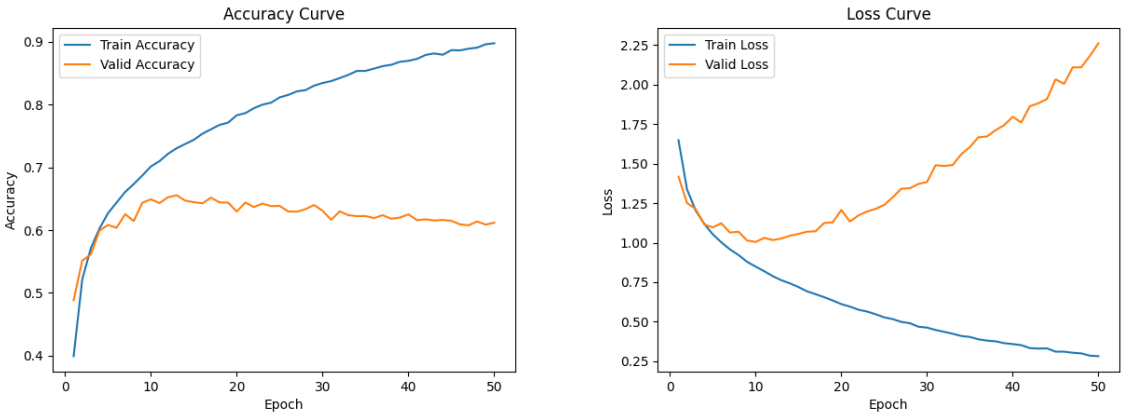

图 6.4: CNN 基础模型在 cifar10 上的准确率和损失展示

可以发现，在小数据集上，我们的训练准确率达到了 0.8798 左右，验证准确率也到了 0.6061。但是根据图像可以看出，我们的验证准确率从大概**第十轮**开始就开始下降了，从最高的 0.6429 左右下降到了 0.6061，说明我们的数据集过于简单了，导致我们的 CNN 网络出现了**过拟合**的情况！

6.3.2 cifar100 数据集

展示一下本人的训练日志，如图6.5所示。

```
(CNN) (base) root@tomorin:/home/Deep-Learning/CNN# python3 /home/Deep-Learning/CNN/main_cifar100.py
Epoch 1/50, Train Loss: 3.9745, Train Accuracy: 0.0894, Valid Loss: 3.6471, Valid Accuracy: 0.1383, Time: 23.09s
Epoch 2/50, Train Loss: 3.4710, Train Accuracy: 0.1670, Valid Loss: 3.3551, Valid Accuracy: 0.1933, Time: 21.40s
Epoch 3/50, Train Loss: 3.2494, Train Accuracy: 0.2114, Valid Loss: 3.2129, Valid Accuracy: 0.2207, Time: 21.73s
Epoch 4/50, Train Loss: 3.1058, Train Accuracy: 0.2379, Valid Loss: 3.1406, Valid Accuracy: 0.2336, Time: 20.99s
Epoch 5/50, Train Loss: 2.9958, Train Accuracy: 0.2568, Valid Loss: 3.0585, Valid Accuracy: 0.2462, Time: 21.25s
Epoch 6/50, Train Loss: 2.9155, Train Accuracy: 0.2746, Valid Loss: 2.9969, Valid Accuracy: 0.2615, Time: 21.16s
Epoch 7/50, Train Loss: 2.8430, Train Accuracy: 0.2901, Valid Loss: 2.9640, Valid Accuracy: 0.2707, Time: 21.37s
Epoch 8/50, Train Loss: 2.7837, Train Accuracy: 0.3001, Valid Loss: 2.9316, Valid Accuracy: 0.2805, Time: 21.33s
Epoch 9/50, Train Loss: 2.7256, Train Accuracy: 0.3116, Valid Loss: 2.9334, Valid Accuracy: 0.2812, Time: 21.06s
Epoch 10/50, Train Loss: 2.6845, Train Accuracy: 0.3233, Valid Loss: 2.8965, Valid Accuracy: 0.2842, Time: 21.48s
Epoch 11/50, Train Loss: 2.6364, Train Accuracy: 0.3333, Valid Loss: 2.9179, Valid Accuracy: 0.2912, Time: 21.30s
Epoch 12/50, Train Loss: 2.6029, Train Accuracy: 0.3365, Valid Loss: 2.8764, Valid Accuracy: 0.2887, Time: 21.71s
Epoch 13/50, Train Loss: 2.5609, Train Accuracy: 0.3459, Valid Loss: 2.8568, Valid Accuracy: 0.2979, Time: 21.57s
Epoch 14/50, Train Loss: 2.5340, Train Accuracy: 0.3503, Valid Loss: 2.8297, Valid Accuracy: 0.3019, Time: 20.93s
Epoch 15/50, Train Loss: 2.4984, Train Accuracy: 0.3573, Valid Loss: 2.8347, Valid Accuracy: 0.3012, Time: 20.77s
Epoch 16/50, Train Loss: 2.4732, Train Accuracy: 0.3621, Valid Loss: 2.8375, Valid Accuracy: 0.3016, Time: 21.02s
Epoch 17/50, Train Loss: 2.4480, Train Accuracy: 0.3680, Valid Loss: 2.8093, Valid Accuracy: 0.3057, Time: 20.60s
Epoch 18/50, Train Loss: 2.4244, Train Accuracy: 0.3715, Valid Loss: 2.8365, Valid Accuracy: 0.3051, Time: 20.82s
Epoch 19/50, Train Loss: 2.4038, Train Accuracy: 0.3761, Valid Loss: 2.8293, Valid Accuracy: 0.3025, Time: 21.22s
Epoch 20/50, Train Loss: 2.3808, Train Accuracy: 0.3794, Valid Loss: 2.8288, Valid Accuracy: 0.3100, Time: 20.48s
Epoch 21/50, Train Loss: 2.3557, Train Accuracy: 0.3867, Valid Loss: 2.8357, Valid Accuracy: 0.3126, Time: 20.62s
Epoch 22/50, Train Loss: 2.3388, Train Accuracy: 0.3887, Valid Loss: 2.8457, Valid Accuracy: 0.3085, Time: 20.44s
Epoch 23/50, Train Loss: 2.3202, Train Accuracy: 0.3938, Valid Loss: 2.8374, Valid Accuracy: 0.3097, Time: 20.29s
Epoch 24/50, Train Loss: 2.3084, Train Accuracy: 0.3959, Valid Loss: 2.8930, Valid Accuracy: 0.3031, Time: 20.01s
Epoch 25/50, Train Loss: 2.2868, Train Accuracy: 0.4016, Valid Loss: 2.9195, Valid Accuracy: 0.3045, Time: 19.97s
Epoch 26/50, Train Loss: 2.2738, Train Accuracy: 0.4004, Valid Loss: 2.9222, Valid Accuracy: 0.3051, Time: 20.12s
Epoch 27/50, Train Loss: 2.2574, Train Accuracy: 0.4072, Valid Loss: 2.8668, Valid Accuracy: 0.3070, Time: 20.03s
Epoch 28/50, Train Loss: 2.2473, Train Accuracy: 0.4098, Valid Loss: 2.8577, Valid Accuracy: 0.3084, Time: 20.16s
Epoch 29/50, Train Loss: 2.2355, Train Accuracy: 0.4123, Valid Loss: 2.8995, Valid Accuracy: 0.3097, Time: 20.56s
Epoch 30/50, Train Loss: 2.2151, Train Accuracy: 0.4167, Valid Loss: 2.8970, Valid Accuracy: 0.3100, Time: 20.80s
Epoch 31/50, Train Loss: 2.2071, Train Accuracy: 0.4198, Valid Loss: 2.9374, Valid Accuracy: 0.3007, Time: 21.23s
Epoch 32/50, Train Loss: 2.1925, Train Accuracy: 0.4183, Valid Loss: 2.8640, Valid Accuracy: 0.3124, Time: 21.40s
Epoch 33/50, Train Loss: 2.1805, Train Accuracy: 0.4234, Valid Loss: 2.9254, Valid Accuracy: 0.3018, Time: 21.55s
Epoch 34/50, Train Loss: 2.1649, Train Accuracy: 0.4251, Valid Loss: 2.9213, Valid Accuracy: 0.3074, Time: 21.54s
Epoch 35/50, Train Loss: 2.1570, Train Accuracy: 0.4293, Valid Loss: 2.9433, Valid Accuracy: 0.3047, Time: 21.13s
Epoch 36/50, Train Loss: 2.1496, Train Accuracy: 0.4309, Valid Loss: 2.9078, Valid Accuracy: 0.3138, Time: 21.19s
Epoch 37/50, Train Loss: 2.1338, Train Accuracy: 0.4321, Valid Loss: 2.9158, Valid Accuracy: 0.3071, Time: 21.50s
Epoch 38/50, Train Loss: 2.1275, Train Accuracy: 0.4335, Valid Loss: 2.9890, Valid Accuracy: 0.2974, Time: 21.54s
Epoch 39/50, Train Loss: 2.1227, Train Accuracy: 0.4332, Valid Loss: 2.9759, Valid Accuracy: 0.3031, Time: 21.45s
Epoch 40/50, Train Loss: 2.1054, Train Accuracy: 0.4397, Valid Loss: 2.9794, Valid Accuracy: 0.3025, Time: 21.60s
Epoch 41/50, Train Loss: 2.1023, Train Accuracy: 0.4392, Valid Loss: 2.9422, Valid Accuracy: 0.3103, Time: 21.03s
Epoch 42/50, Train Loss: 2.0899, Train Accuracy: 0.4440, Valid Loss: 3.0231, Valid Accuracy: 0.3007, Time: 21.80s
Epoch 43/50, Train Loss: 2.0870, Train Accuracy: 0.4443, Valid Loss: 2.9760, Valid Accuracy: 0.3066, Time: 21.83s
Epoch 44/50, Train Loss: 2.0782, Train Accuracy: 0.4440, Valid Loss: 3.0109, Valid Accuracy: 0.3023, Time: 21.77s
Epoch 45/50, Train Loss: 2.0669, Train Accuracy: 0.4470, Valid Loss: 3.0118, Valid Accuracy: 0.3018, Time: 21.82s
Epoch 46/50, Train Loss: 2.0609, Train Accuracy: 0.4486, Valid Loss: 3.0185, Valid Accuracy: 0.2990, Time: 21.48s
Epoch 47/50, Train Loss: 2.0541, Train Accuracy: 0.4479, Valid Loss: 3.0445, Valid Accuracy: 0.2994, Time: 21.56s
Epoch 48/50, Train Loss: 2.0433, Train Accuracy: 0.4517, Valid Loss: 3.0187, Valid Accuracy: 0.2975, Time: 21.26s
Epoch 49/50, Train Loss: 2.0358, Train Accuracy: 0.4548, Valid Loss: 3.0295, Valid Accuracy: 0.3010, Time: 21.83s
Epoch 50/50, Train Loss: 2.0270, Train Accuracy: 0.4558, Valid Loss: 3.0312, Valid Accuracy: 0.3043, Time: 21.27s
Training Finished
Model Saved
(CNN) (base) root@tomorin:/home/Deep-Learning/CNN#
```

图 6.5: cnn-base 在 cifar100 上的训练结果

训练结束，我将训练结果和训练可视化结果存储到了 results/cifar100/文件夹下，在此展示一下训练 loss 曲线和准确度曲线，如图6.6所示。

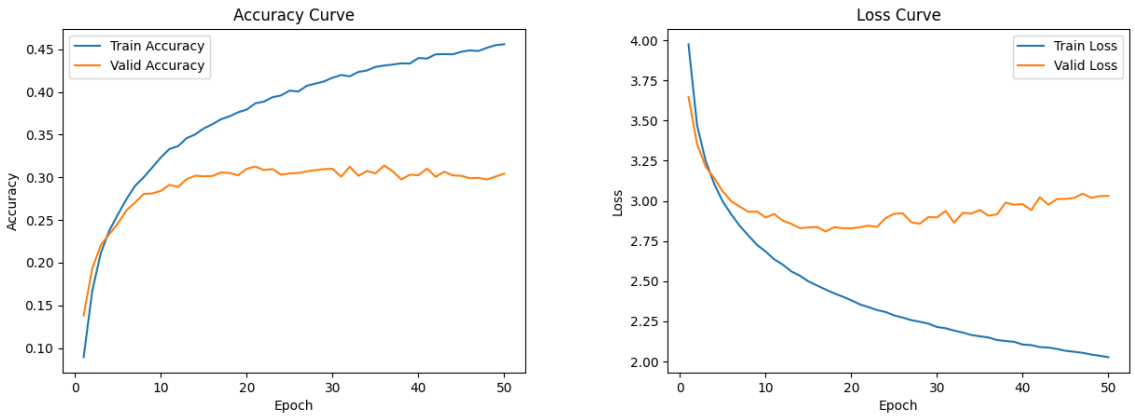

图 6.6: CNN 基础模型在 cifar100 上的准确率和损失展示

我们发现，实际上正确率并不是特别高，训练准确率只达到了 **0.4352** 左右，测试准确率也只有

0.2987。实际上，通过观察训练日志，我发现本次训练还出现了轻微的过拟合的情况，但相比前面的cifar10，并没有那么明显。

通过以上的训练结果，我们可以发现，**仅仅通过简单的 CNN 网络，我们是无法很好地完成 cifar100 数据集的分类的**。

7 实现微型 ResNet 网络结构

在实现 ResNet[2] 前，本人有必要先通过论文的阅读，对该 CNN 模型有一个简单的了解。

7.1 ResNet 主要原理介绍

根据论文阅读和实验指导手册的查看，我发现 ResNet 相较于传统的卷积神经网络（CNN）模型，其核心优化在于引入了“残差学习”（Residual Learning）框架，有效地解决了深度 CNN 训练中的“退化问题”（Degradation Problem）。如图7.7所示，展示了详细的网络结构。

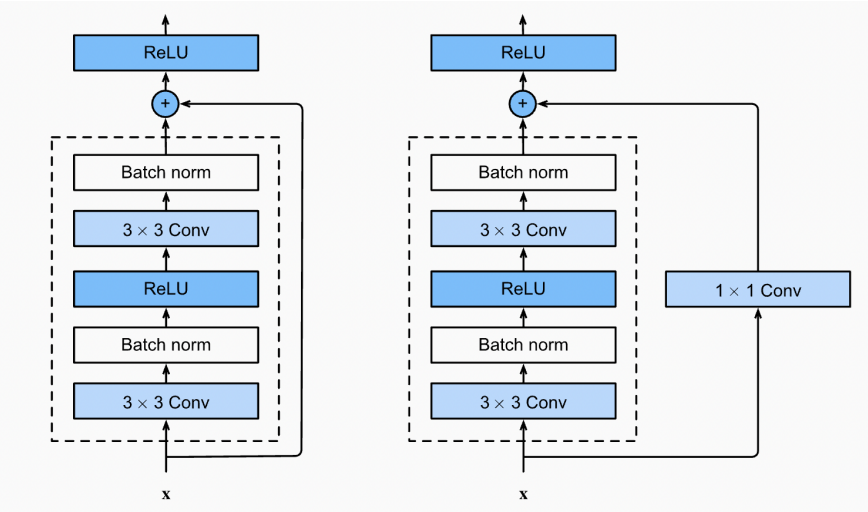

图 7.7: ResNet 具体结构 [2]

我发现，实际上 ResNet 的优化之处在于：

1. **残差块与跳跃连接:** 该网络的设计通过一个“跳跃连接”或“快捷连接”直接将输入 x 加到这些层的输出上，可以显著减少参数数量和计算复杂度；
2. **解决梯度消失/爆炸问题:** 跳跃连接允许梯度能够更直接地反向传播到较浅的层，这样可以较好地解决梯度消失或者爆炸的问题；
3. **简化学习目标:** 即使网络非常深，通过 ResNet 的优化，某些层也可以轻松地“什么都不做”，直接传递信息。
4. **实现更深的网络结构:** 通过跳跃连接的模块，我们可以设计出更深的网络，具有更强的特征提取能力；
5. **批量归一化的有效使用:**ResNet 的残差块中广泛使用了批量归一化，有利于稳定训练过程、加速收敛、获得一个更好的结果。

根据论文中对于 ResNet 架构的介绍（如图7.8所示），我们可以发现：实际上 ResNet 一共有五种参数大小的版本，分别为 ResNet-18、ResNet-34、ResNet-50、ResNet-101 和 ResNet-152。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

图 7.8: ResNet 主要架构 [2]

下面我们具体解释一下**残差块 Residual Block** 和**瓶颈 Bottleneck** 的具体结构，如图7.9所示。

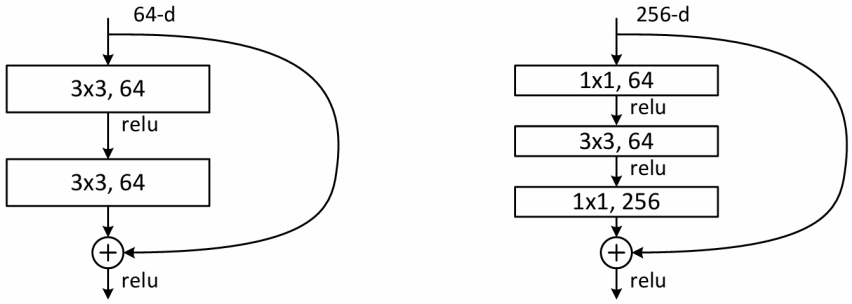

图 7.9: ResNet 的两种基础部分框架 [2]

1. **Residual Block：**包含两个卷积层，每个卷积层后通常会跟一个批量归一化层（Batch Normalization）和 ReLU 激活函数。残差块的输入通过一个跳跃连接（Skip Connection）直接添加到第二个卷积层的输出上。跳跃连接允许输入数据直接跳过一个或多个卷积层，从而减少了梯度消失的风险；

2. **Bottleneck：**首先，应用一个 1x1 的卷积层，这有助于减少特征图的通道数，这一步不包括激活函数。接着应用一个 3x3 的卷积层，这是特征提取的主要步骤，这一步通常会跟一个批量归一化层和 ReLU 激活函数。最后再次应用一个 1x1 的卷积层，将通道数恢复到原始的通道数，这一步也通常不包括激活函数。瓶颈设计不仅减少了计算量，还有助于提高网络的性能。最重要的是，这样的设计可以减少过拟合的风险。

7.2 ResNet 具体实现

了解了 ResNet 的具体原理之后，我们就可以开始具体对其进行实现了。此处，我准备将五种 ResNet 均进行复现。首先，我们需要根据上面的两种基础结构，进行网络的设计。设计完两种基础网络后，我们再根据五种不同的网络结构去进行不同的设计。

由于篇幅原因，此处仅仅展示我的**总体代码设计思路**和**部分核心代码**。具体的代码，详见我的代码文件。首先，根据论文中的数据和网络结构，我编写了两个基础框架 Residual Block 和 Bottleneck。

对于 Residual Block 部分，我们设计了 shortcut，默认情况下，它是一个空的 nn.Sequential()，意味着如果输入和输出的维度（通道数和尺寸）相同，它不执行任何操作，直接将输入 x 传递过去。但是卷积操作改变了输入的空间维度或者通道数的时候，那么 shortcut 路径也会通过一个 1x1 的卷积层来调整输入 x 的维度，使其能够与 self.left 的输出相加。对于 Bottleneck 来说，基本设计思路与上面相同，仅仅修改参数，此处就不再过多阐述。

对于五种模型的设计，实际上是差不多的，仅仅修改了部分参数。我们根据图7.8中的五种 layer 的参数，分别设计了不同的层结构。

```
self.layer1 = self.make_layer(ResidualBlock, 64, 2, stride=1)
self.layer2 = self.make_layer(ResidualBlock, 128, 2, stride=2)
self.layer3 = self.make_layer(ResidualBlock, 256, 2, stride=2)
self.layer4 = self.make_layer(ResidualBlock, 512, 2, stride=2)
self.fc = nn.Linear(512, num_classes)
```

如上述代码所示，是本人设计的 ResNet18 的 layer 结构，就是完全按照7.8中的参数来进行设计的。那么其他的设计也就很简单了，我们只需要更换函数的两个参数值即可，ResNet18 和 ResNet34 选用的是 BasicBlock，其余的三个模型使用的都是 Bottleneck。对于 num_classes 参数，我们在不同的 main 函数文件中，调用不同的值，若测试 cifar10 数据集，则规定该参数为 10；同理，若测试 cifar100 数据集，则规定该参数为 100。

展示一下具体的函数调用代码：

```
if args.model == 'cnn_base':
    model = BaseCNN(num_of_last_layer=10)
elif args.model == 'cnn_resnet18':
    model = ResNet_18(num_classes=10, ResidualBlock=BasicBlock)
elif args.model == 'cnn_resnet34':
    model = ResNet_34(num_classes=10, ResidualBlock=BasicBlock)
elif args.model == 'cnn_resnet50':
    model = ResNet_50(num_classes=10, ResidualBlock=Bottleneck)
elif args.model == 'cnn_resnet101':
    model = ResNet_101(num_classes=10, ResidualBlock=Bottleneck)
elif args.model == 'cnn_resnet152':
    model = ResNet_152(num_classes=10, ResidualBlock=Bottleneck)
```

7.3 ResNet 训练结果展示

编写完我们的五种 ResNet 的代码后，我们分别对这五个模型在 cifar10 和 cifar100 上进行测试，得到十组不同的结果。由于篇幅过长，此处本人仅选取其中的四组进行展示与分析，其他的请移步我的代码文件进行查看。本人选取的是 cifar10-ResNet18，cifar10-ResNet152，cifar100-ResNet34 和 cifar100-ResNet101。

7.3.1 在 cifar10 数据集上测试 ResNet18

展示一下我们的训练结果可视化的准确率和损失，如图7.10所示。

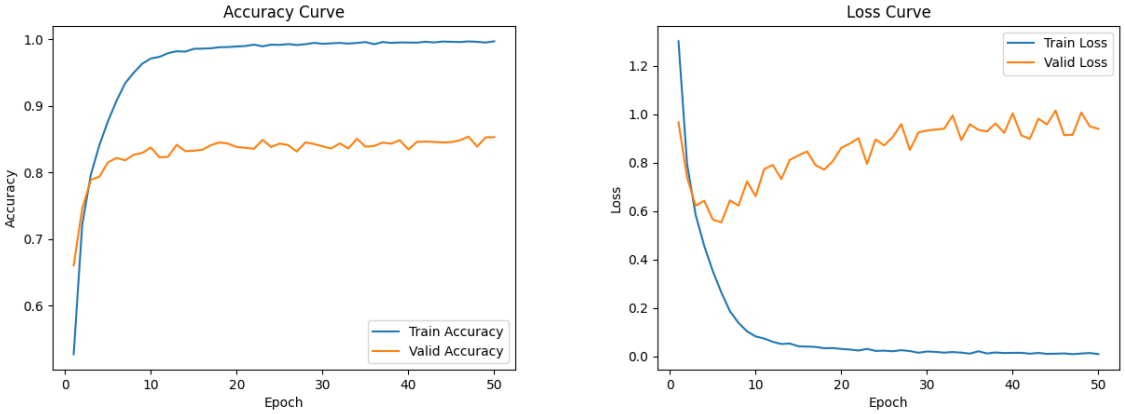

图 7.10: ResNet18 模型在 cifar10 上的准确率和损失展示

本人发现，通过该网络进行训练，训练准确率可以达到 0.9970，验证准确率可以达到 0.8529，而且通过图像来看，几乎没有发生过拟合的情况。从损失来看，训练损失越来越小，收敛到 0.0098，而验证损失则不断增大。说明该模型在验证集上表现不是特别好。

7.3.2 在 cifar10 数据集上测试 ResNet152

展示一下我们的训练结果可视化的准确率和损失，如图7.11所示。

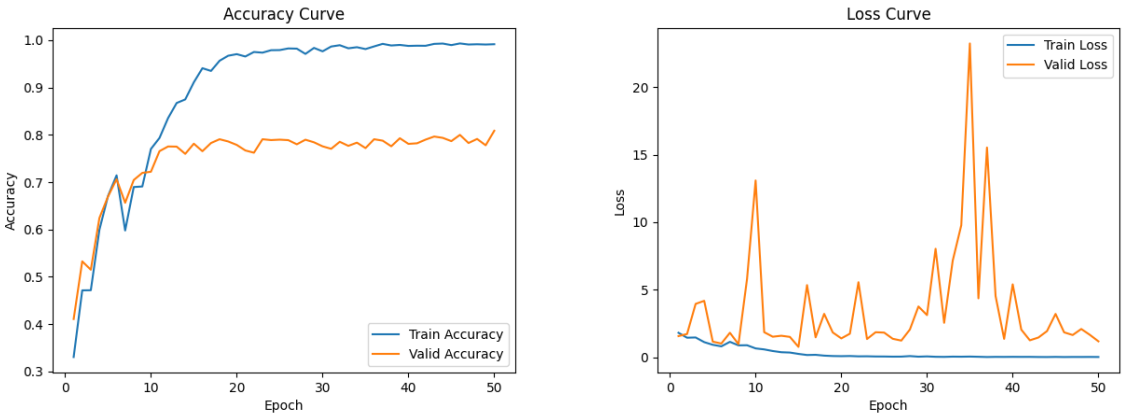

图 7.11: ResNet152 模型在 cifar10 上的准确率和损失展示

我们可以发现，由于我们使用较大的网络来适配 cifar10 这个小数据集，所以实际上很大程度地发生了过拟合的现象。实际上随着训练损失越来越小的情况下，我们的验证损失和验证准确率并没有获得很大的提升。

7.3.3 在 cifar100 数据集上测试 ResNet34

展示一下我们的训练结果可视化的准确率和损失，如图7.12所示。

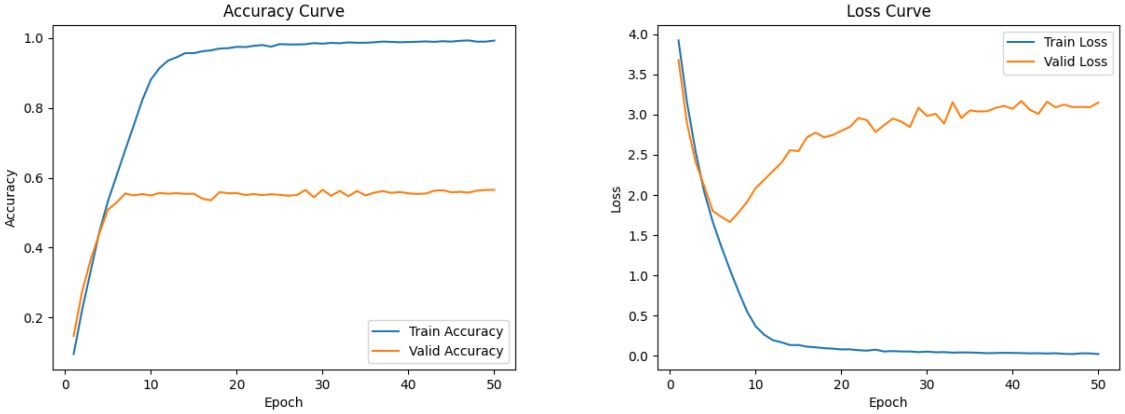

图 7.12: ResNet34 模型在 cifar100 上的准确率和损失展示

我们发现，随着训练准确率的上升和训练损失的下降，我们的验证集上的表现并没有特别好，基本上处于一个平稳的状态。可以注意到，我们的验证损失实际上是随着训练的进行变得越来越大的。

7.3.4 在 cifar100 数据集上测试 ResNet101

展示一下我们的训练结果可视化的准确率和损失，如图7.13所示。

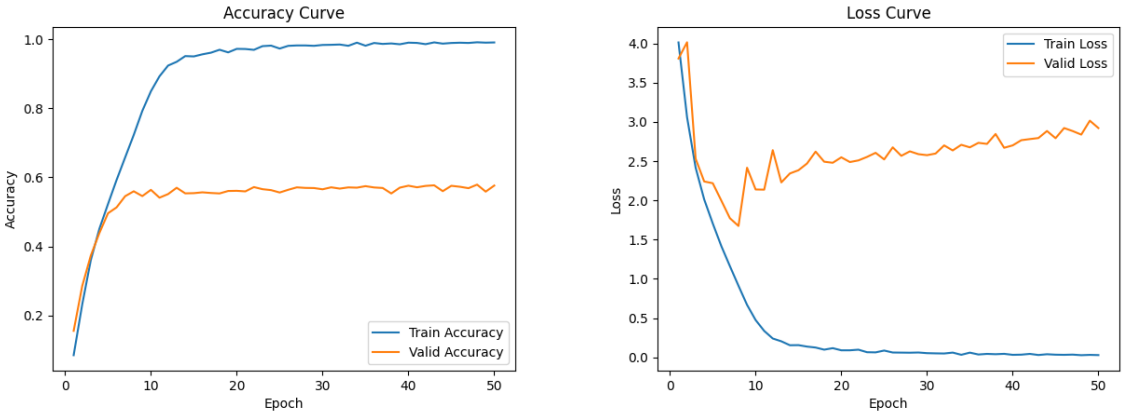

图 7.13: ResNet101 模型在 cifar100 上的准确率和损失展示

我们更换成 ResNet-101 网络，在 cifar100 上进行实验，但是实际上效果与上一组结果差不多。

8 实现微型 DenseNet 网络结构

下面，我们开始完成 DenseNet[4] 的复现。首先，我们还是需要先阅读一下论文。由于篇幅原因，此处我仅展示论文网络设计的核心步骤和创新点。

8.1 DenseNet 主要原理介绍

首先，展示一下 DenseNet 的整体网络架构，如图8.14所示。

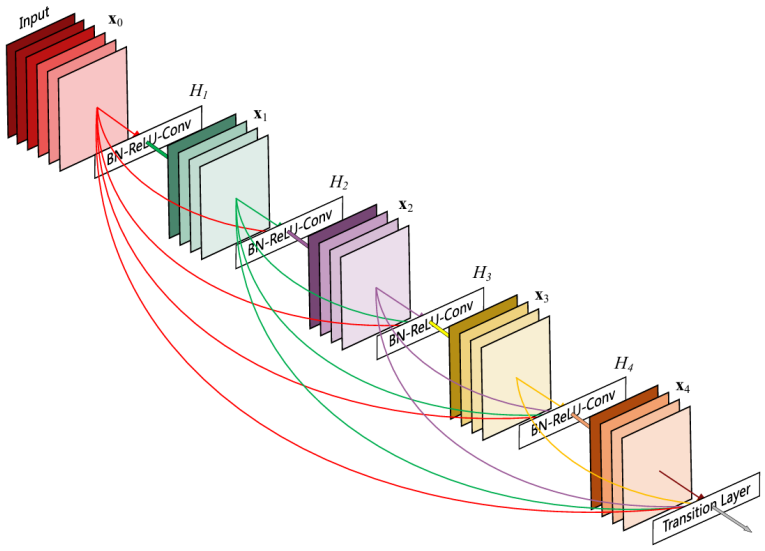

图 8.14: DenseNet 主要架构 [4]

我们发现，相比于其他网络，DenseNet 具有以下特点：

1. **密集连接：** 每一层的输入并不仅仅是其直接前一层的输出，而是其前面所有层的输出的拼接；
2. **特征图的累积：** 每一层都将前面所有层的特征图进行拼接，这种机制使得网络能够学习到更丰富、更多样化的特征，并且特征可以在网络中更深远地传播；
3. **稠密块：** 一个完整的 DenseNet 通常由多个稠密块构成，具有稠密连接的特点；
4. **过渡层：** 在稠密块之间，DenseNet 会使用过渡层，通常包含一个批量归一化层、一个 1×1 卷积层和一个平均池化层。

我们进一步查看其网络结构中的参数，如图8.15所示。我发现实际上与我们前面实现的 ResNet 基本类似，所以此处不再展开介绍。

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1\times1 \text{ conv} \\ 3\times3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

图 8.15: DenseNet 部分核心数据

8.2 DenseNet 具体实现

基本上，DenseNet 的实现和前面提到的 ResNet 类似，就是更换了一下对应的层的结构。具体的实现可以看我的代码，对于该网络，我们设计了 DenseBlock 基础结构，采用了论文中提出的瓶颈层 (bottleneck layer) 的设计。我们按照封装式的子类设计，通过修改 num_blocks 数组的值，来适配每一种网络的参数结构。

由于代码实现较为复杂，此处仅仅展示我们的部分设计代码和设计思路。

```
def forward(self, x):
    out = self.conv1(x)
    out = self.trans1(self.dense1(out))
    out = self.trans2(self.dense2(out))
    out = self.trans3(self.dense3(out))
    out = self.dense4(out)
    out = F.relu(self.bn(out))
    out = F.avg_pool2d(out, 4)
    out = out.view(out.size(0), -1)
    out = self.linear(out)
    return out
```

这段代码是我们设计的末端处理，在网络的末端，通过批归一化、ReLU 激活、平均池化（起到全局平均池化的作用）、展平和全连接层，将学习到的深度卷积特征转换为最终的分类预测。

我们通过编写下面的代码来完成各模型的调用。

```
class DenseNet_121(nn.Module):
    def __init__(self, num_classes):
        super(DenseNet_121, self).__init__()
```

```
        self.model = _DenseNet(DenseBlock, [6, 12, 24, 16], growth_rate=32,
        ↪  reduction=0.5, num_classes=num_classes)

    def forward(self, x):
        return self.model(x)
```

如上所示，我们调用 DenseNet121 模型，就可以使用 [6, 12, 24, 16] 这个数组来完成 num_blocks 的赋值，这样就可以对我们的模型的参数进行一个赋值的操作。

8.3 DenseNet 训练结果展示

编写完我们的四种 DenseNet 的代码后，我们分别对这四个模型在 cifar10 和 cifar100 上进行测试，得到八组不同的结果。由于篇幅过长，此处本人仅选取其中的四组进行展示与分析，其他的请移步我的代码文件进行查看。本人选取的是 cifar10-DenseNet121，cifar10-DenseNet264，cifar100-DenseNet169 和 cifar100-DenseNet201。

8.3.1 在 cifar10 数据集上测试 DenseNet121

展示一下我们的训练结果可视化的准确率和损失，如图8.16所示。

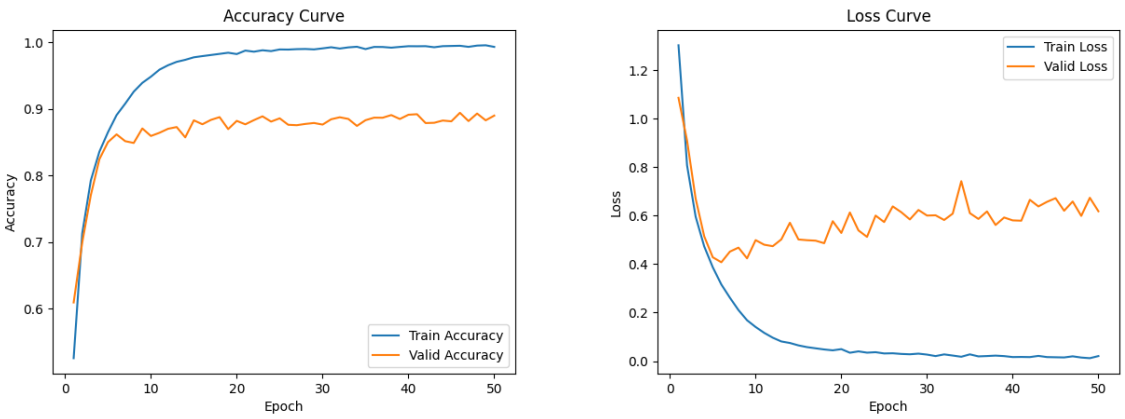

图 8.16: DenseNet121 模型在 cifar10 上的准确率和损失展示

我们发现，实际上随着训练准确率的上升，我们的验证准确率基本上稳定在 0.89，对应的验证集损失值也基本上在 0.6 左右。

8.3.2 在 cifar10 数据集上测试 DenseNet264

展示一下我们的训练结果可视化的准确率和损失，如图8.17所示。

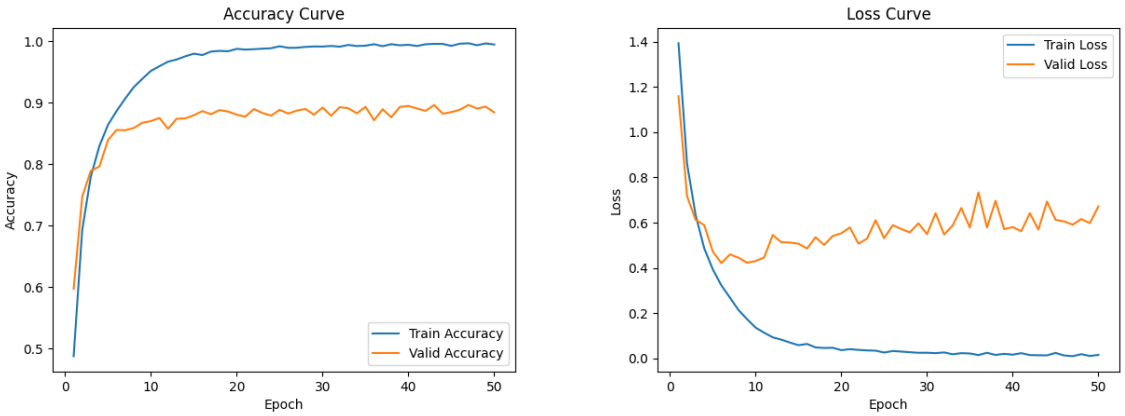

图 8.17: DenseNet264 模型在 cifar10 上的准确率和损失展示

我们在训练集的准确率上达到了 0.9945，但是损失值还是在一个较大的值上进行滑动。

8.3.3 在 cifar100 数据集上测试 DenseNet169

展示一下我们的训练结果可视化的准确率和损失，如图8.18所示。

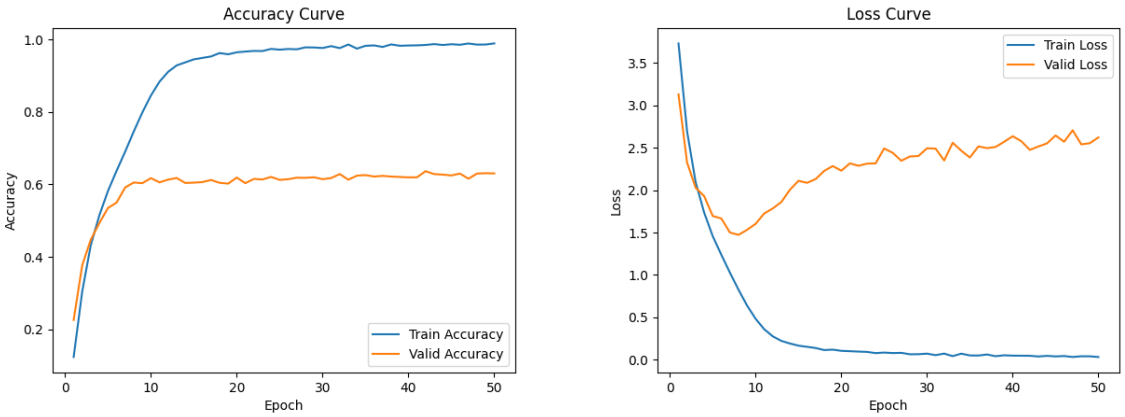

图 8.18: DenseNet169 模型在 cifar100 上的准确率和损失展示

我们发现，实际上 DenseNet 在 cifar100 上的表现也是非常好的，基本达到了 0.9893 的训练准确率。

8.3.4 在 cifar100 数据集上测试 DenseNet201

展示一下我们的训练结果可视化的准确率和损失，如图8.19所示。

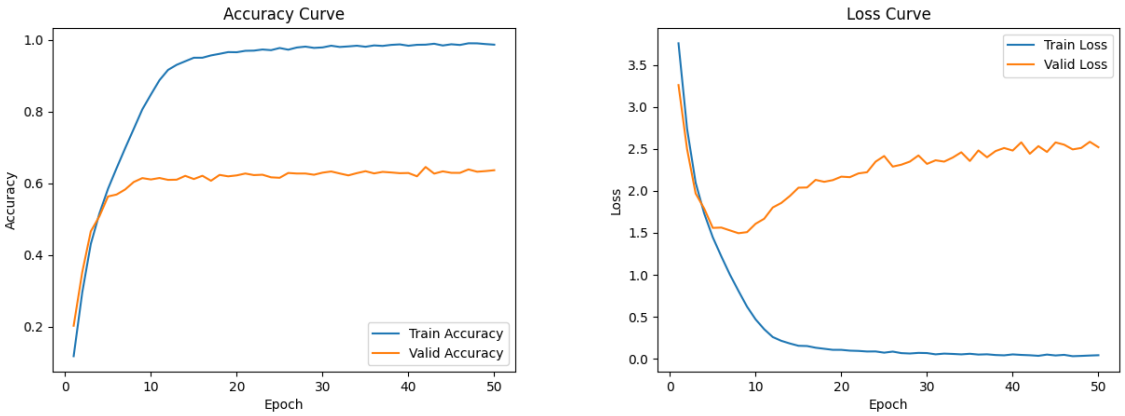

图 8.19: DenseNet201 模型在 cifar100 上的准确率和损失展示

同理，这个测试结果也与 DenseNet169 差不多。

9 实现带有 SE 结构的微型 ResNet 网络结构

我们首先阅读论文 Squeeze-and-Excitation Networks[8]，了解一下 SE-ResNet 的主要架构。

9.1 SE-ResNet 主要原理介绍

经过文献的阅读，SE-ResNet 的具体结构如图9.20所示。

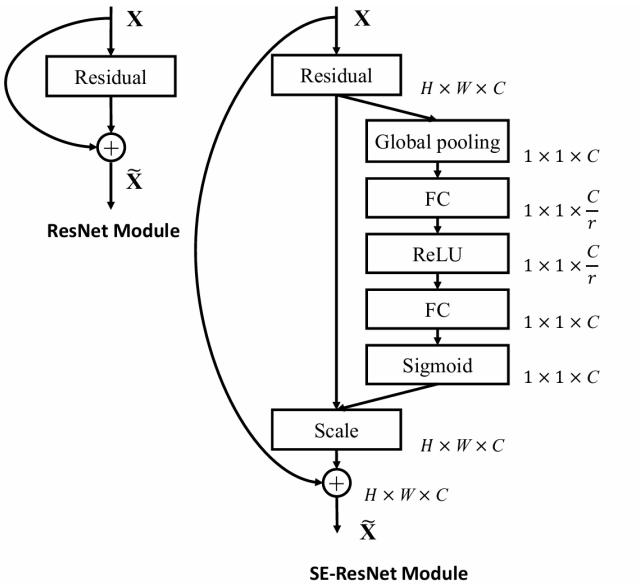

图 9.20: SE-ResNet 主要结构 [8]

SE-ResNet(Squeeze-and-Excitation ResNet)相较于原始的 ResNet,其核心优化在于引入了**“Squeeze-and-Excitation”（SE）模块**，从而显著提升了网络的性能和表征能力。

通过上网查阅博客，我了解到，SE 模块的目的是让网络能够显式地建模特征通道之间的相互依赖关系，并根据这种关系自适应地重新校准（recalibrate）通道的特征响应。简单来说，就是让网络学会“关注”信息量更丰富的特征通道，并“抑制”信息量较少的特征通道。

具体来说，我们通常使用全局平均池化将每个二维的特征通道（$H \times W$）压缩成一个单一的数值。对于一个输入特征图 $U \in \mathbb{R}^{H \times W \times C}$，其第 c 个通道为 $u_c \in \mathbb{R}^{H \times W}$，Squeeze 操作通过下式计算得到通道描述符 $z \in \mathbb{R}^C$，其中 z_c 是 z 的第 c 个元素：

$$z_c = F_{sq}(u_c) = \frac{1}{H \times W} \sum_{i=1}^{H} \sum_{j=1}^{W} u_c(i, j)$$

这个描述符 z（维度为 $1 \times 1 \times C$ 或等效的长度为 C 的向量）可以看作是每个通道特征的全局概要。

一般的，SE-ResNet 相比于 ResNet 有以下优势：

1. **提升表征能力：**通过显式建模通道间的依赖关系，SE-ResNet 能够学习到更具判别力的特征表示。它使得网络能够自适应地增强有用的特征通道，抑制无用的特征通道。
2. **显著的性能提升：**SE-ResNet 在增加相对较少计算成本的情况下，能够取得比原始 ResNet 更高的准确率。
3. **轻量级且易于集成：**SE 模块本身的参数量和计算量相对较小，尤其是通过在 Excitation 操作中使用降维技巧（缩减比例 r）。
4. **增强特征辨识度：**通过对不同通道赋予不同的权重，网络可以更关注对当前任务最重要的信息，从而提高特征的辨识度。
5. **自适应特征校准：**SE 模块是数据驱动的，能够根据输入数据的特性动态调整不同通道的权重，具有更好的适应性。

9.2 SE-ResNet 具体实现

具体而言，其实现方法是首先通过全局平均池化来获取到每一个通道中的信息，然后根据此通过一个两层的 MLP，在经过 sigmod 计算出每一个通道的重要性，并通过这种门控机制，筛选更为有效的通道。下面我们展示一下该网络与 ResNet 的不同之处，也展示一下我的设计思路和部分代码。

```
# SE 模块：全局池化后 FC，再利用 sigmoid 得到注意力权重
se = self.globalAvgPool(out)
se = se.view(se.size(0), -1)
se = self.fc1(se)
se = self.relu(se)
se = self.fc2(se)
se = self.sigmoid(se)
se = se.view(se.size(0), se.size(1), 1, 1)
out = out * se
```

此处我们设计了对应的 SE 结构，和图9.20表示的基本上差不多，首先进行池化操作，然后通过 FC 操作、ReLU 操作、FC 操作和 Sigmoid 操作得到注意力权重，然后调整为我们需要的规模就可以了。其他部分的 SE 结构设计基本上与此处类似，所以此处由于篇幅原因就不再介绍和展示了。

9.3 SE-ResNet 训练结果展示

编写完我们的五种 DenseNet 的代码后，我们分别对这四个模型在 cifar10 和 cifar100 上进行测试，得到八组不同的结果。由于篇幅过长，此处本人仅选取其中的四组进行展示与分析，其他的请移步我的代码文件进行查看。本人选取的是 cifar10-SEResNet18，cifar10-SEResNet152，cifar100-SEResNet34 和 cifar100-SEResNet101。

9.3.1 在 cifar10 上测试 SE-ResNet18

展示一下我们的训练结果可视化的准确率和损失，如图9.21所示。

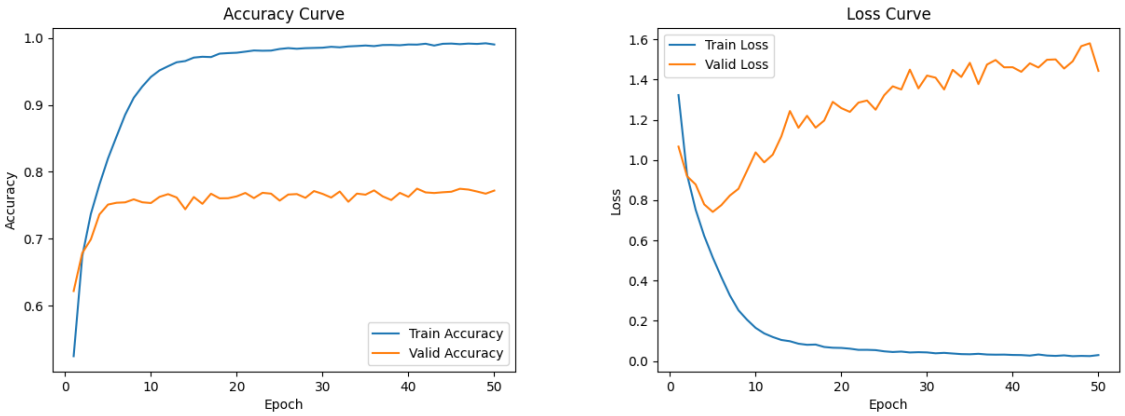

图 9.21: SE-ResNet18 模型在 cifar10 上的准确率和损失展示

9.3.2 在 cifar10 上测试 SE-ResNet152

展示一下我们的训练结果可视化的准确率和损失，如图9.22所示。

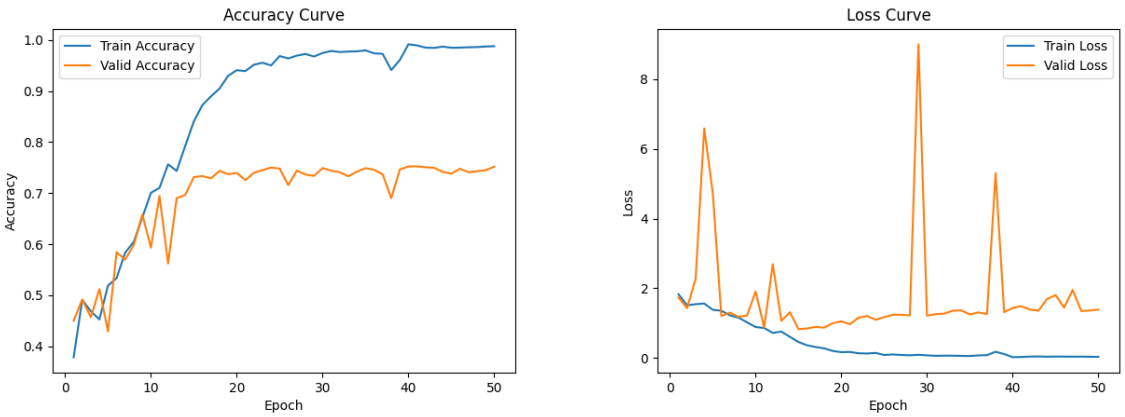

图 9.22: SE-ResNet152 模型在 cifar10 上的准确率和损失展示

9.3.3 在 cifar100 上测试 SE-ResNet34

展示一下我们的训练结果可视化的准确率和损失，如图9.23所示。

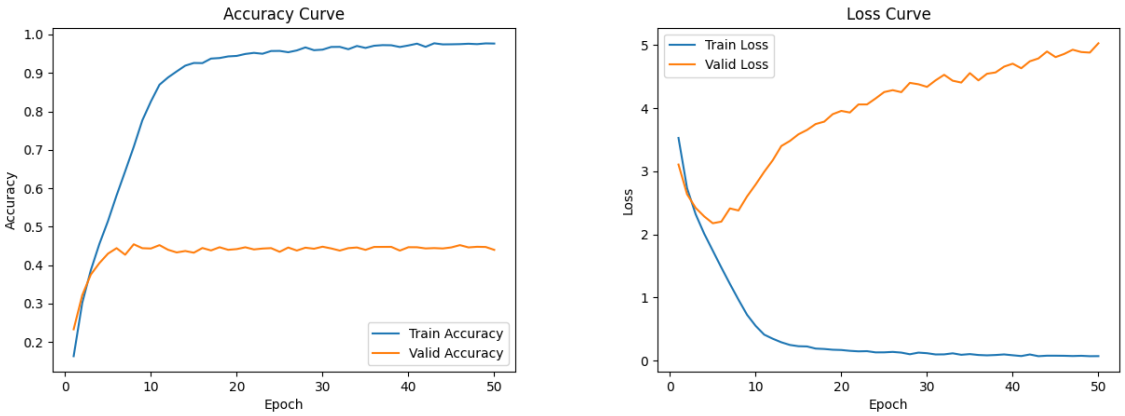

图 9.23: SE-ResNet34 模型在 cifar100 上的准确率和损失展示

9.3.4 在 cifar100 上测试 SE-ResNet101

展示一下我们的训练结果可视化的准确率和损失，如图9.24所示。

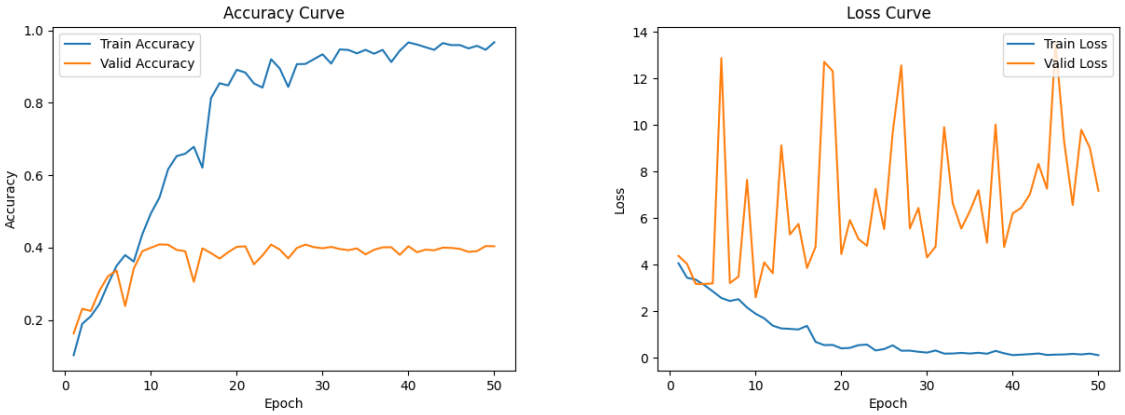

图 9.24: SE-ResNet101 模型在 cifar100 上的准确率和损失展示

10 实验结果与分析

由于前面的各小节中都已经展示了对应的准确率和损失的图像，所以此处不再分别展示对应的图像，可以参考仓库中的 results 文件夹中的图像。此处仅对我们前面实现和测试的全部网络进行总结和对比。

10.1 cifar10 数据集的测试结果

首先，展示一下相对简单的数据集，也就是我们的 cifar10 数据集，我们展示一下我们编写的各类模型在该数据集上的测试结果，如表1所示。

模型名称	训练准确率	验证准确率	训练损失	验证损失
CNN Base	87.98%	60.61%	33.36%	209.57%
CNN DenseNet121	99.32%	89.00%	2.08%	61.78%
CNN DenseNet169	99.35%	**89.20%**	1.98%	**60.44%**
CNN DenseNet201	**99.59%**	88.16%	1.25%	66.83%
CNN DenseNet264	99.45%	88.42%	1.55%	67.20%
CNN ResNet18	99.70%	85.29%	0.98%	93.93%
CNN ResNet34	99.73%	84.76%	**0.84%**	95.86%
CNN ResNet50	99.19%	84.16%	2.47%	97.54%
CNN ResNet101	99.38%	84.98%	1.77%	91.43%
CNN ResNet152	99.15%	80.86%	2.47%	118.45%
CNN SE-ResNet18	99.02%	77.19%	2.95%	77.19%
CNN SE-ResNet34	99.15%	76.96%	2.58%	149.40%
CNN SE-ResNet50	99.04%	77.89%	3.01%	126.94%
CNN SE-ResNet101	99.02%	76.62%	3.02%	130.17%
CNN SE-ResNet152	98.76%	75.19%	3.71%	138.96%

表 1: cifar10 数据集上的各项模型测试

首先，我们普遍发现，在 cifar10 数据集上的测试结果大部分准确率都很高，训练准确率基本上达到了 99%，验证准确率也有部分达到了 90% 左右，基本上可以看出 DenseNet 的表现较为好，其中表现最好的是 DenseNet169，验证准确率比较高。另外，我们还可以看出，随着网络结构变得越来越复杂，实际上我们的训练结果并不是越复杂就越好的，有一些复杂的模型，可能在简单的 cifar10 数据集上的表现并不是很好。

另外，我们从表中可以看到，随着模型的复杂性的增大，我们出现了过拟合的情况，并不是模型越大，我们的训练效果就越好。这一点在前面的章节中就已经提到过了。

10.2 cifar100 数据集的测试结果

然后，再展示一下我们相对大型的数据集的测试结果，也就是我们的 cifar100 数据集。测试结果如表2所示。

模型名称	训练准确率	验证准确率	训练损失	验证损失
CNN Base	45.58%	30.43%	202.70%	303.12%
CNN DenseNet121	98.64%	63.96%	4.25%	246.38%
CNN DenseNet169	98.93%	63.04%	3.33%	262.09%
CNN DenseNet201	98.62%	63.60%	4.18%	252.00%
CNN DenseNet264	98.82%	**64.59%**	3.88%	**244.14%**
CNN ResNet18	99.06%	53.05%	2.85%	369.46%
CNN ResNet34	**99.26%**	56.57%	**2.28%**	314.99%
CNN ResNet50	99.20%	57.03%	3.28%	290.49%
CNN ResNet101	99.10%	57.63%	2.81%	292.20%
CNN ResNet152	98.85%	58.62%	3.69%	276.00%
CNN SE-ResNet18	97.89%	44.54%	6.29%	495.93%
CNN SE-ResNet34	97.65%	43.98%	7.23%	502.76%
CNN SE-ResNet50	96.95%	41.71%	10.21%	500.61%
CNN SE-ResNet101	96.69%	40.36%	11.18%	716.88%
CNN SE-ResNet152	80.97%	33.76%	66.71%	3041.05%

表 2: cifar100 数据集上的各项模型测试

可以看出，我们的随着数据集的变大，我们的验证准确率已经越来越低了，基本上最好的也只能达到 60% 左右，可以看到我们的 DenseNet264 表现相对较好，因为我们的网络结构很大，这样可以使我们的训练结果更好一些。另外，随着模型的复杂度的增大，我们发现基本上没有出现过拟合的情况，这也与我们数据集的复杂程度有着很大的联系。

11 各模型在训练过程中的不同

对于我们的基础模型，也就是没有跳跃连接的卷积网络来说，由于每一层的输出仅仅依赖于上一层的输出，所以可能会导致出现**梯度消失或者爆炸问题**。比如我们可以从上面的基础模型的训练结果看出，实际上发生了很大的梯度消失情况。实际上在 cifar10 数据集上的表现是还行的，但是在 cifar100 这种大型数据集上的表现就不是很好了。

对于 ResNet 来说，我们引入了**残差连接**，网络可以学习到与输入之间的残差，而不是直接学习到完整的映射。这样实际上可以缓解我们之前的梯度消失或者爆炸的问题，可以使我们的训练过程更加稳定一些。

对于 DenseNet 来说，我们引入了**密集网络**，利用跳跃连接，将每一层的输入都与之前所有层的输出进行连接。也就是说 DenseNet 中的每一层都接收前面所有层的输出作为我们的输入。这样来说，在训练的时候，我们提升了特征的利用率，避免了冗余的学习，但是对于 DenseNet 的参数，我们也是比较难以确定的。

最后，对于 SE-ResNet 来说，实际上引入了 **Squeeze-and-Excitation** 模块，SE 模块通过学习通道之间的关系，会自动调整每个通道的权重，增强重要特征通道的表达能力。

我们从前面展示的可视化图像中也可以发现，实际上 ResNet 的训练图像是十分稳定的，且收敛轮数很小，基本上在第五轮都可以完成收敛。但是对于大规模的 DenseNet 可能就会出现一些训练上的波动问题。包括对于 cifar100 数据集上的基础模型的训练，验证准确率仅仅只有 30% 左右，说明我们的梯度消失问题还是很严重的。

以上就是总结出的各模型在训练过程中的不同，实际上在前面的章节中我们都已经在其中涉及到了，此处只是做了一个总结，具体的内容还是在前面更加详细一些。

12 总结与体会

本次实验，我们通过编写 CNN 基础网络、ResNet、DenseNet 和 SE-ResNet 等 15 种 CNN 网络，在 cifar10 和 cifar100 两个数据集上进行训练和测试，并将结果进行了归纳与总结。

通过本次实验，我理解并掌握了以下深度学习的网络编写技巧：

1. 我学会了编写正常的基础 CNN 结构，输出了对应的网络结构；
2. 通过编写 ResNet 网络结构，学习了残差连接结构，了解了其原理；
3. 通过编写 DenseNet 网络结构，学习了稠密连接结构，与前面的方法做了对比，了解了其优势和不足之处；
4. 通过 SE 结构的阅读和调研，成功复现出 SE-ResNet 结构，并进行测试，得到了不错的结果；
5. 对于获得的训练准确率和损失值的图像，我可以很好地对其进行详细和准确的分析；
6. 对于遇到的很多过拟合问题，我们可以尝试降低模型参数数量，或者提前停止训练，或者减少训练轮数等方法来缓解我们的过拟合问题。

通过本次实验，我基本上了解了 CNN 的一些常见的网络结构，并且可以通过论文的面熟对其进行复现，熟悉了各类算子的编写手段，对网络结构的了解也更加深刻了。希望在后续的实验中，还可以学习更多的深度学习的知识。

13 文件目录结构

本次实验的代码文件目录结构如下所示：

```
codes ..... 项目总目录
├── .gitignore ..... gitignore 文件
├── load_data_cifar10.py ..... 用于加载 cifar10 的数据
├── load_data_cifar100.py ..... 用于加载 cifar100 的数据
├── main_cifar10.py ..... 主函数，用于测试 cifar10
├── main_cifar100.py ..... 主函数，用于测试 cifar100
├── plot.py ..... 用于画图
├── README.md ..... 项目 README 文档
├── requirements.txt ..... 项目需求库
├── train.py ..... 编写了训练函数
├── __init__.py ..... 项目初始化，用于查看 GPU
├── results ..... 训练结果存储地址，包括各类模型和各类数据集
├── model ..... 存放我们的 CNN 模型实现
│   ├── CNN_Base.py ..... CNN 基础代码
│   ├── CNN_DenseNet.py ..... 实现的 DensenNet
│   ├── CNN_ResNet.py ..... 实现的 ResNet
│   └── CNN_SE_ResNet.py ..... 实现的 SE 结构的 ResNet
└── data ..... 主要存放实验使用的数据
```

本次实验的有关代码和文件，都已经上传至我的个人 github 中。您可以通过访问**此链接**来查阅我的代码文件。

参考文献

[1] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[3] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks, 2017.

[4] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[5] Shang-Hua Gao, Ming-Ming Cheng, Kai Zhao, Xin-Yu Zhang, Ming-Hsuan Yang, and Philip Torr. Res2net: A new multi-scale backbone architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(2):652–662, February 2021.

[6] Jiang-Jiang Liu, Qibin Hou, Ming-Ming Cheng, Changhu Wang, and Jiashi Feng. Improving convolutional networks with self-calibrated convolutions. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10093–10102, 2020.

[7] Xiang Li, Wenhai Wang, Xiaolin Hu, and Jian Yang. Selective kernel networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 510–519, 2019.

[8] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks, 2019.

[9] Qibin Hou, Daquan Zhou, and Jiashi Feng. Coordinate attention for efficient mobile network design. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13708–13717, 2021.

[10] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks, 2018.