华中科技大学

课 程 实 验 报 告

《计算机视觉》实验二
基于卷积神经网络的 CIFAR-10 数据集分类

专业班级：CS2003
学　　号：U202015374
姓　　名：张隽翊
指导教师：李贤芝
报告日期：2023 年 4 月 3 日

计算机科学与技术学院

目 录

一、实验内容

1.1 实验题目

基于卷积神经网络的 CIFAR-10 数据集分类。

1.2 实验内容

1.2.1 实验任务

设计一个卷积神经网络，在 CIFAR-10 数据集上实现分类任务。

1.2.2 注意事项

（1）不能直接导入现有的 CNN 网络，比如 VGG、ResNet 等，可以以现有网络为基础进行改进。

（2）可以尝试不同的卷积神经网络设计、使用不同的激活函数等，观察网络性能。

（3）深度学习框架任选。

（4）实验报告包括网络设计、在 10 类测试集上的平均准确率截图、每一类的准确率截图以及必要的分析等。

1.3 实验过程

1.3.1 开发环境

本次实验中使用的环境配置如下：

（1）操作系统版本：Ubuntu 22.04.2 x86_64

（2）深度学习框架：PyTorch 版本 2.0.0 + CUDA12.1

（3）环境管理工具：Anaconda

（4）编程环境：PyCharm Professional + Visual Studio Code

1.3.2 数据准备

直接使用 PyTorch 内置的 CIFAR-10 数据集。由于 CIFAR-10 数据集共包含 60000 张 32x32 的图片，取 50000 张作为训练集和验证集，10000 张作为测试集。进一步地，取 49000 张作为训练集，1000 张作为验证集。

取 batch_size=128，用 DataLoader 加载数据，并对数据进行归一化处理和增强处理。数据增强我尝试了 torchvision.transforms.ColorJitter 和 torchvision.transforms.RandomHorizontalFlip，前者随机更改图像的亮度、对比度、饱和度和色调，后者以给定的概率随机水平翻转给定的图像。最终我选择了效果较为明显的后者，即 RandomHorizontalFlip。

综上，数据处理部分的代码如下。

```
# from lab2.configs import DATA_DIR
```

```
from configs import DATA_DIR

NUM_TRAIN = 49000

transform_train = T.Compose([
    # T.ColorJitter(0.5),
    T.RandomHorizontalFlip(),
    T.ToTensor(),
    T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = T.Compose([
    T.ToTensor(),
    T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

batch_size = 128

# 训练集
cifar10_train = dset.CIFAR10(root=DATA_DIR, train=True, download=True,
transform=transform_train)
loader_train = DataLoader(cifar10_train, batch_size=batch_size,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN))
)

# 验证集
cifar10_val = dset.CIFAR10(root=DATA_DIR, train=True, download=True,
transform=transform_train)
loader_val = DataLoader(cifar10_val, batch_size=batch_size,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN,
50000)))

# 测试集
cifar10_test = dset.CIFAR10(root=DATA_DIR, train=False, download=True,
transform=transform_test)
loader_test = DataLoader(cifar10_test, batch_size=batch_size)
```

至此，用于训练和测试的数据集已经处理完毕。

在开始训练前，简单预览一下数据集中的图片。这里我参考了斯坦福大学计算机视觉课程（CS231n）2022 年春季实验二（assignments2）中的部分代码，加载已经下载好的 CIFAR-10 数据集，代码如下：

```
from data_utils import load_CIFAR10
```

```
cifar10_dir = os.path.join(DATA_DIR, 'cifar-10-batches-py')
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

然后可视化数据集中的数据，为 CIFAR-10 中的每一类别展示几张图片。

```
from configs import cifar10_classes, num_classes

samples_per_class = 7

for y, cls in enumerate(cifar10_classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

输出结果如图 1.1 所示。

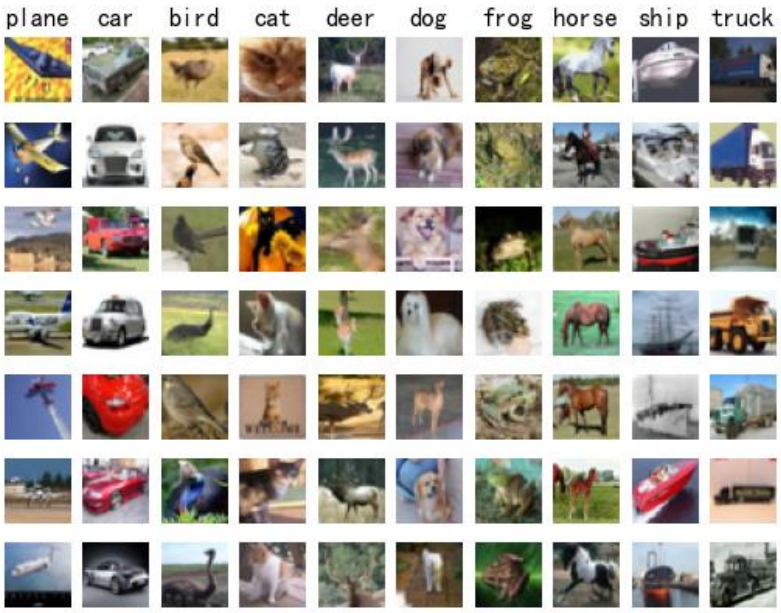

图 1. 1 CIFAR-10 数据集预览

1.4 模型训练

1.4.1 模型准备

本实验中我尝试了多种模型，包括 LeNet、AlexNet、VGGNet、ResNet 这几种经典的卷积神经网络模型，并比较了其中部分网络的性能，观察了引入批量正则化与否对网络性能的影响。

1.4.2 LeNet

LeNet 是由 Yan LeCun 等人于 1998 年提出的一种简单的卷积神经网络架构，一般指 LeNet-5。LeNet 是最早的卷积神经网络之一，推动了深度学习的发展。

LeNet-5 由 7 层组成，除输入以外每隔一层都可以训练参数。LeNet-5 的架构图如图 1.2 所示。

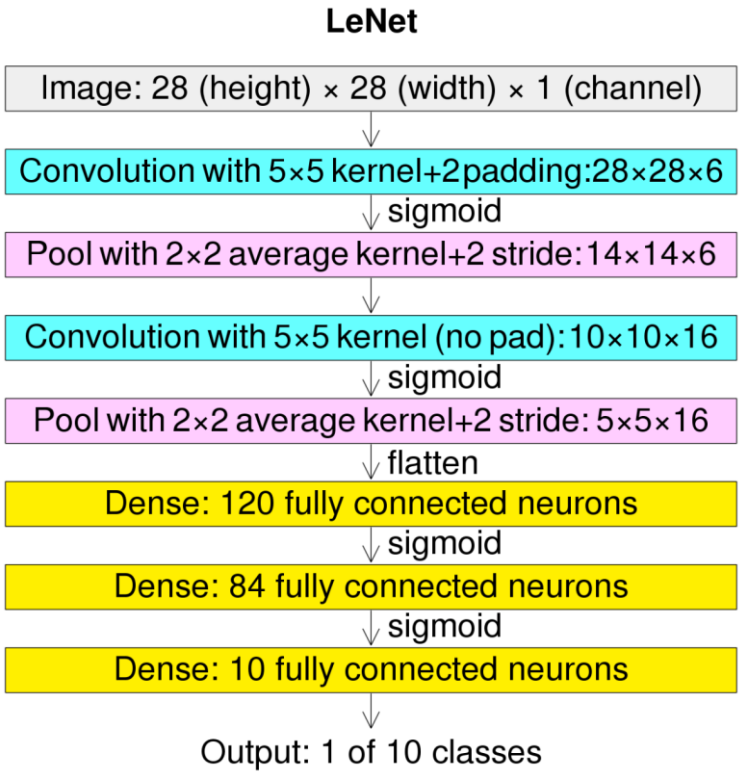

图 1. 2 LeNet5 架构图

参照上述架构图，编写 lenet_utils.py 文件，如下所示。

```
import torch.nn as nn
import torch.nn.functional as F


class LeNet(nn.Module):
    def __init__(self, num_classes=10):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)
```

```
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, num_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

定义模型部分的代码如下。损失函数采用 torch.nn.CrossEntropyLoss，后续模型也采用这个设置，因为 CIFAR-10 属于多分类问题；优化器采用 torch.optim.SGD 即随机梯度下降，学习率调整采用 ReduceLROnPlateu。

```
from lenet_utils import LeNet

model = LeNet()
criterion = nn.CrossEntropyLoss()
# optimizer = optim.Adam(model.parameters(), weight_decay=1e-4)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
lr_scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
patience=5, verbose=True)
```

然后训练模型，记录训练过程的一些参数，保留最优模型。

```
from train import train
from configs import generate_save_dir

best_model, train_loss_history, train_acc_history, val_acc_history = train(
    model, criterion, optimizer, lr_scheduler, loader_train, loader_val,
    epochs=10, save_dir=generate_save_dir('lenet5.pth', False)
)
```

训练过程中损失 loss 和准确率 acc 的变化如图 1.3 所示。

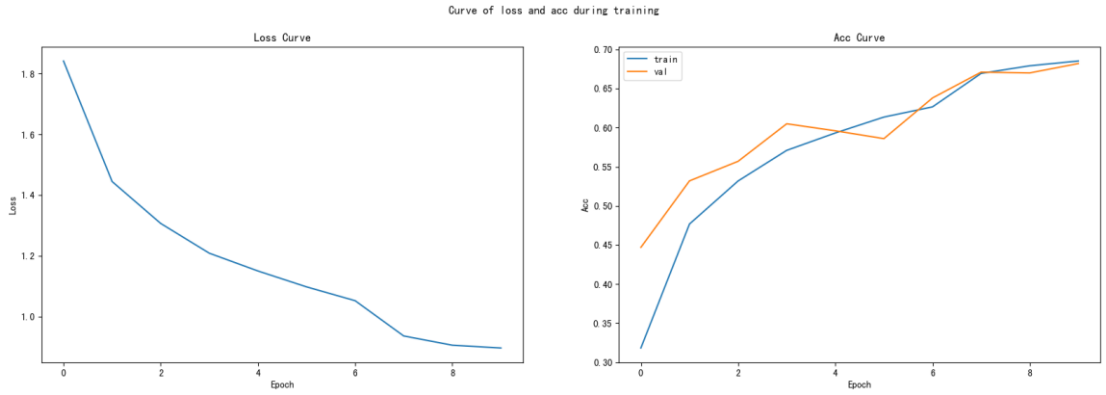

图 1. 3 LeNet 训练记录

在测试集上检验模型效果，使用的代码如下。

```
from train import check_accuracy


_ = check_accuracy(best_model, loader_test)
```

最终结果如图 1.4 所示。

```
Checking accuracy on test set
Got 6514 / 10000 correct (65.14)
Class   Accuracy
----------------
plane     |  68.0
car       |  76.6
bird      |  54.6
cat       |  47.7
deer      |  58.6
dog       |  49.5
frog      |  76.1
horse     |  70.1
ship      |  79.0
truck     |  71.2
```

图 1. 4 LeNet 模型结果

可以看出，LeNet 模型比较简单，参数较少，训练速度较快，但对较复杂的图片准确率不高。

1.4.3 AlexNet

AlexNet 是划时代的杰作，它一共包含 8 层，前 5 层是卷积层，后面有一些最大池化层，最后是 3 层全连接层。

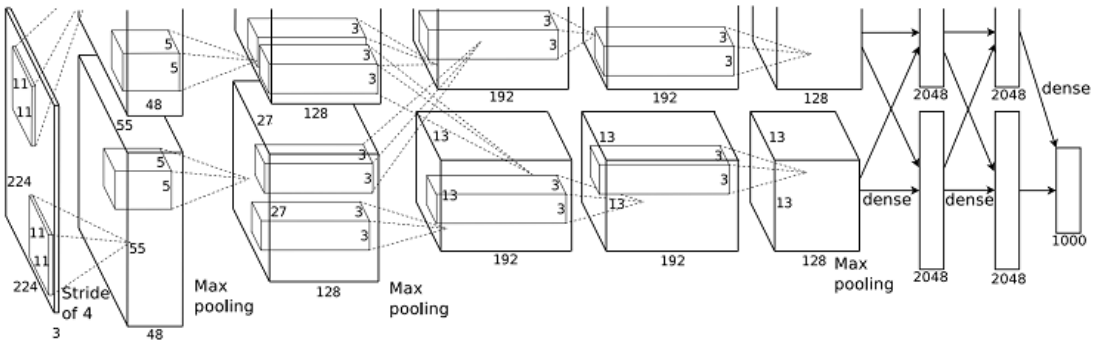

图 1. 5 AlexNet 架构图

参照上述架构图，编写 alexnet_utils.py 文件，如下所示。

```
import torch.nn as nn
from configs import num_classes


class AlexNet(nn.Module):
    def __init__(self, num_classes=num_classes):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),
        )
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 2 * 2, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), 256 * 2 * 2)
        x = self.classifier(x)
        return x
```

然后训练模型，记录训练过程的一些参数，保留最优模型。

```
from alexnet_utils import AlexNet

model = AlexNet()
```

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), weight_decay=1e-4)
lr_scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='max',
patience=5, verbose=True)
```

训练过程中损失 loss 和准确率 acc 的变化如图 1.6 所示。

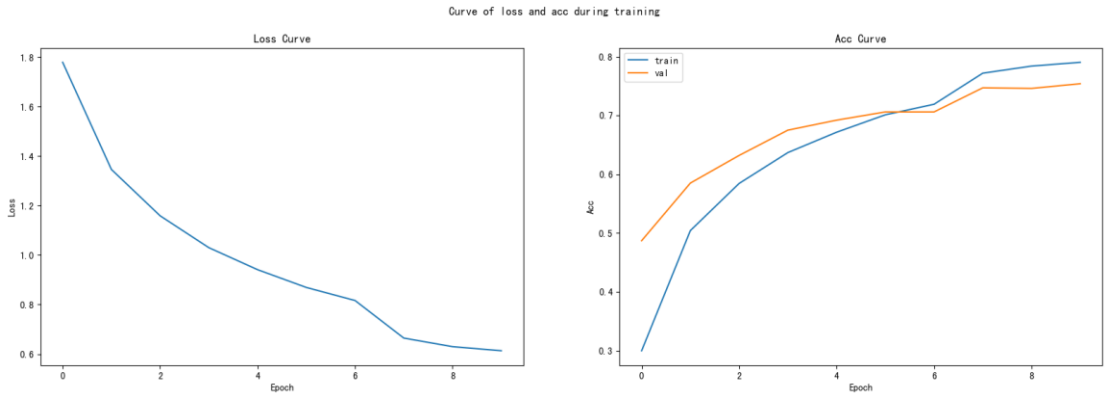

图 1. 6 AlexNet 训练记录

在测试集上检验模型结果，最终结果如图 1.7 所示。

```
Checking accuracy on test set
Got 7356 / 10000 correct (73.56)
Class    Accuracy
----------------
plane    |  78.5
car      |  83.4
bird     |  62.2
cat      |  55.1
deer     |  67.1
dog      |  59.4
frog     |  83.1
horse    |  78.7
ship     |  84.3
truck    |  83.8
```

图 1. 7 AlexNet 模型结果

可以看出，AlexNet 的准确率相比 LeNet 有了一定的提升，但仍然不算太高。

1.4.4 简易 ResNet

发现简单模型效果一般后，我转向使用更为复杂的模型，例如残差神经网络 ResNet。残差神经网络是由微软研究院的何恺明、张祥雨、任少卿、孙剑等人提出的。它的主要贡献是发现了在增加网络层数的过程中，随着训练精度逐渐趋于饱和，继续增加层数，训练精度就会出现下降的现象，而这种下降不是由过拟合造成的。他们将这一现象称之为“退化现象（Degradation）”，并针对退化现象发明了 “快捷连接（Shortcut connection）”，极大的消除了深度过大的神经网络训练困难问题。神经网络的“深度”首次突破了 100 层、最大的神经网络甚至超过了 1000 层。

ResNet 网络是由很多相同的模块堆叠起来的，为了保证代码具有可读性和

可扩展性，ResNet 在设计时采用了模块化设计，其残差单元就是这样设计的，如图 1.8 所示。

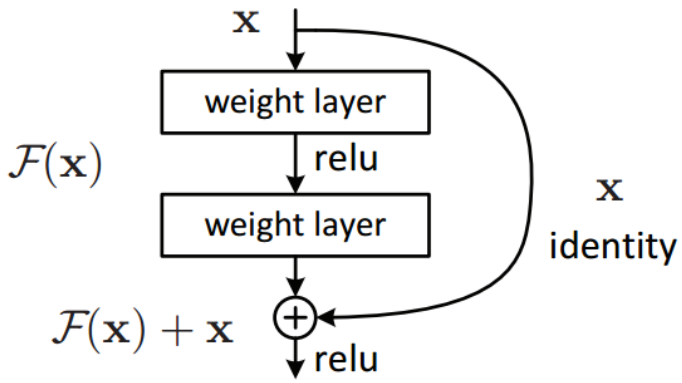

图 1. 8 ResNet 残差模块

秉承这个思想，我参照 ResNet9 架构编写了简易的 ResNet 模型，网络结构如图 1.9 所示。

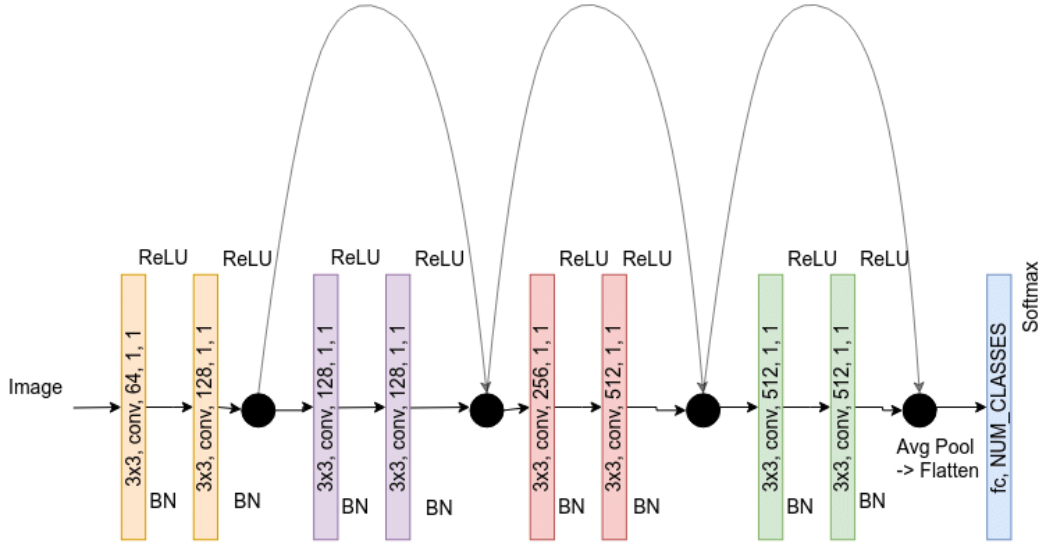

图 1. 9 ResNet9 架构图

编写 demo_utils.py 文件，如下所示。

```
import torch.nn as nn


def conv_block(in_channels, out_channels, pool=False):
    layers = [
        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True)
    ]
    if pool:
        layers.append(nn.MaxPool2d(2))
    return nn.Sequential(*layers)
```

```
class ResNet9(nn.Module):
    def __init__(self, in_channels=3, num_classes=10):
        super(ResNet9, self).__init__()

        self.conv1 = conv_block(in_channels, 64)
        self.conv2 = conv_block(64, 128, pool=True)
        self.res1 = nn.Sequential(conv_block(128, 128), conv_block(128, 128))

        self.conv3 = conv_block(128, 256, pool=True)
        self.conv4 = conv_block(256, 512, pool=True)
        self.res2 = nn.Sequential(conv_block(512, 512), conv_block(512, 512))

        self.classifier = nn.Sequential(nn.MaxPool2d(4),
                                        nn.Flatten(),
                                        nn.Dropout(0.2),
                                        nn.Linear(512, num_classes))

    def forward(self, x):
        out = self.conv1(x)
        out = self.conv2(out)
        out = self.res1(out) + out
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.res2(out) + out
        out = self.classifier(out)
        return out
```

训练过程中损失 loss 和准确率 acc 的变化如图 1.10 所示。

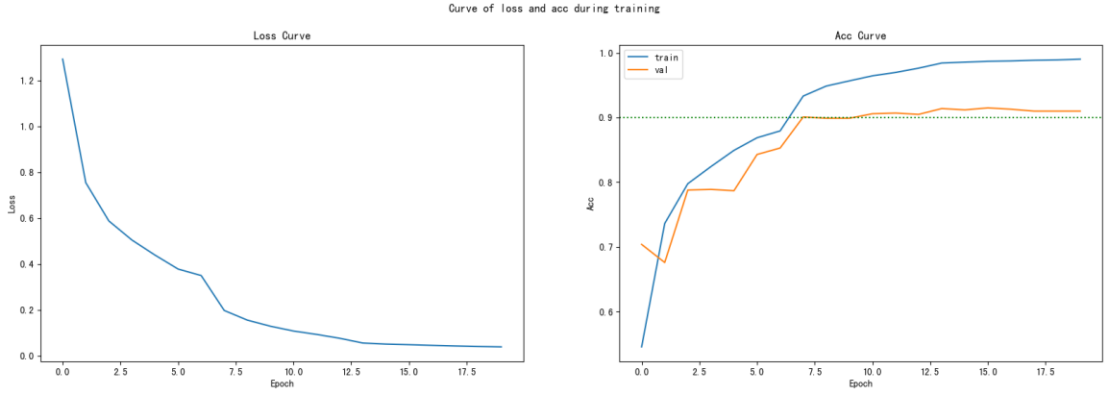

图 1. 10 ResNet9 训练记录

在测试集上检验模型结果，最终结果如图 1.11 所示。

```
Checking accuracy on test set
Got 9164 / 10000 correct (91.64)
Class   Accuracy
----------------
plane    |  94.0
car      |  96.3
bird     |  88.5
cat      |  83.8
deer     |  92.3
dog      |  86.3
frog     |  93.6
horse    |  92.3
ship     |  95.0
truck    |  94.3
```

图 1. 11 ResNet9 模型结果

结果表明，该模型的准确率相当高，且收敛速度也很快，在 20 个 epoch 就达到了 90%以上的准确率。这也表明了残差结构具有十分优良的性能。

1.4.5 VGGNet

VGG 相比 AlexNet 的一个改进是采用连续的几个3×3卷积核代替 AlexNet 中的大卷积核（$11 \times 11, 7 \times 7, 5 \times 5$）。对于给定的感受野，采用堆积的小卷积核优于采用大卷积核，因为多层非线性层可以增加网络深度来保证学习更加复杂的模式，同时使用的参数更少、代价更小。

VGG 网络的结构如图 1.12 所示。

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

图 1. 12 VGG 网络架构图

以 VGG16 和 VGG19 为例，VGG16 包含了 16 个隐藏层（13 个卷积层和 3 个全连接层），如上图中 D 列所示；VGG19 包含了 19 个隐藏层（16 个卷积层

和 3 个全连接层），如上图中 E 列所示。VGG 网络的结构非常一致，全部使用的是3×3的卷积和2×2的最大池化。

VGG 的优点在于结构简洁统一，采用几个小卷积层替代了大卷积层从而减少了参数数量，验证了通过不断加深网络结构可以提升网络的性能。但相对的，网络层数加深带来的后果是 VGG 会耗费更多的计算资源，并使用了更多的参数，导致更多存储占用。

实验中，我修改了 VGG 的输入和输出，使之能够用于 CIFAR-10 数据集，如下所示。

```
class VGG(nn.Module):

    def __init__(self, features, num_classes=1000, init_weights=True):
        super(VGG, self).__init__()
        self.features = features  # backbone used to extract features
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 1 * 1, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            self._initialize_weights()
```

我也对比了使用预训练与否对模型性能的影响，如图 1.13 所示。

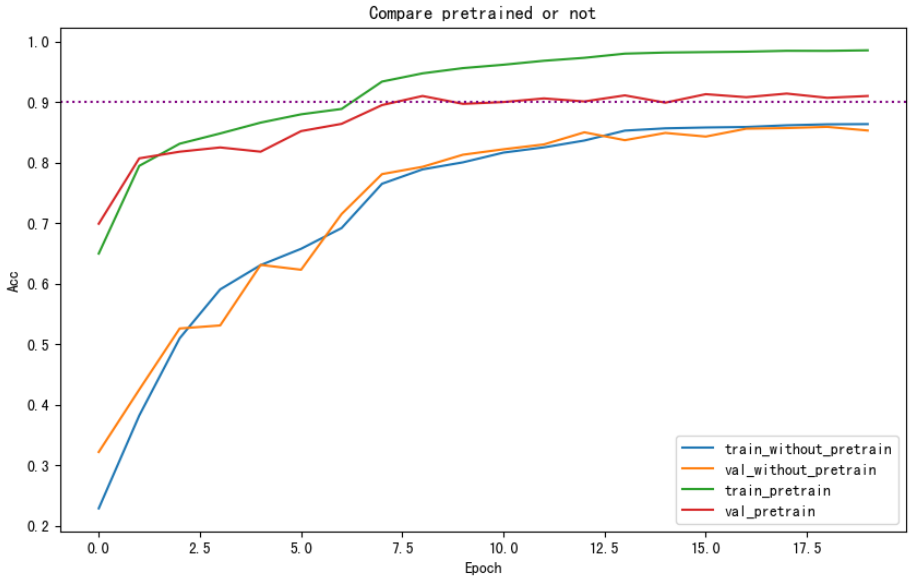

图 1. 13 VGG19 使用预训练与否的性能对比

从图上可以看出，使用了预训练的模型在有限的迭代次数中获得了更好的性能。

1.4.6 ResNet

实验中，我修改了 ResNet 的输入和输出，使之能够用于 CIFAR-10 数据集。我还对比了不同架构的网络对模型性能的影响，如图 1.14 所示。

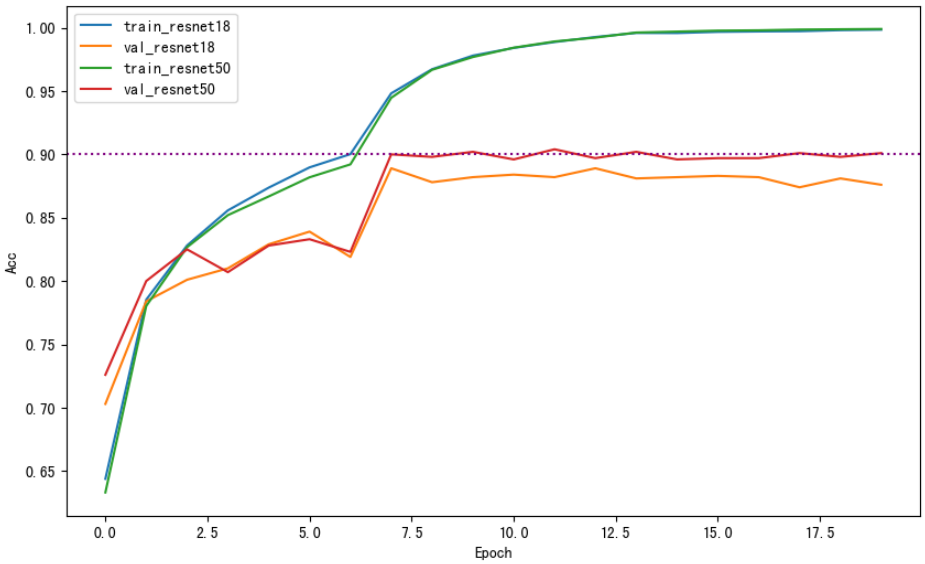

图 1. 14 ResNet18 和 ResNet50 性能对比

从图上可以看出，ResNet50 和 ResNet18 在训练阶段的准确率相差不大，但在验证阶段 ResNet50 的准确率要高于 ResNet18，这可能是由于前者具有更多的参数。

1.5 可视化特征图

输入的原始图像经过每次卷积层得到的数据称为特征图。可视化卷积核是为了看模型提取哪些特征，可视化特征图是为了看模型提取到的特征是什么样子的。

获取特征图的方法有很多种，可以从输入开始，逐层做前向传播，直到想要的特征图处将其返回。尽管这种方法可行，但有些麻烦了。PyTorch 中提供了一个专用的接口使得网络在前向传播的过程中能够获取到特征图，即 hook。

```
import torchvision.utils as vutil
# from configs import IMAGE_DIR

IMAGE_DIR = os.path.join(os.path.dirname(DATA_DIR), 'images')

DEMO_DIR = None


def hook_func(module, input, output):
    """
```

```
    Hook function of register_forward_hook

    Parameters:
    -----------
    module: module of neural network
    input: input of module
    output: output of module
    """
    image_name = get_image_name_for_hook(module)
    data = output.clone().detach()
    data = data.permute(1, 0, 2, 3)
    vutil.save_image(data, image_name, pad_value=0.5)


def get_image_name_for_hook(module):
    """
    Generate image filename for hook function

    Parameters:
    -----------
    module: module of neural network
    """
    os.makedirs(DEMO_DIR, exist_ok=True)
    base_name = str(module).split('(')[0]
    index = 0
    image_name = '.'  # '.' is surely exist, to make first loop condition True
    while os.path.exists(image_name):
        index += 1
        image_name = os.path.join(
            DEMO_DIR, '%s_%d.png' % (base_name, index))
    return image_name
```

以已经训练好的 ResNet50 网络为例，展示其中卷积层的特征图，如图 1.15 所示。

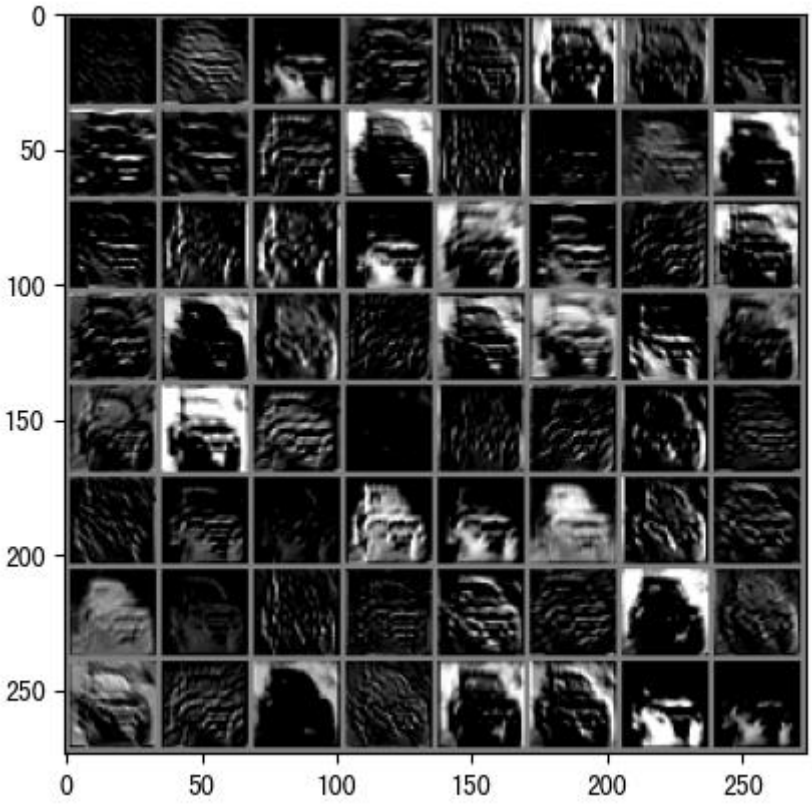

图 1. 15 ResNet50 卷积层特征图

从特征图名称看出，这是 CIFAR-10 数据集中标签为“1”的类，即 Car。从提取到的特征图来看，有一些确实能够看出汽车的轮廓，而有些则接近全黑。

1.6 心得体会

这次实验中，我在实验一探索的基础上尝试了一些更加复杂的处理操作，并取得了不错的结果。以下是本实验中用于提高模型性能和减少训练时间采用的技巧总结：

（1）数据归一化：通过减去平均值并除以每个通道的像素的标准差来归一化图像张量。规格化数据可以避免来自任何一个通道的像素值对损失和梯度的影响程度不同的问题。

（2）数据增强：从训练数据集中加载图像时，我使用了随机转换。具体来说就是以 50%的概率水平翻转图像。

（3）批处理归一化：在卷积层后添加一个 BN 层，对前一层的输出进行归一化。批处理归一化类似于数据归一化，但它应用于一个层的输出，且平均值和标准差是其学习参数。

（4）学习率调整：不使用固定的学习率，而是使用 lr_scheduler，它会在每次训练后改变学习率。学习率的选择是深度学习中一个困扰人们许久的问题，学习速率设置过小，会极大降低收敛速度，增加训练时间；学习率太大，可能导致参数在最优解两侧来回振荡。但是当我们选定了一个合适的学习率后，经过许多

轮的训练后，可能会出现准确率震荡或 loss 不再下降等情况，说明当前学习率已不能满足模型调优的需求。此时我们就可以通过一个适当的学习率衰减策略来改善这种现象，提高我们的精度。这种设置方式在 PyTorch 中被称为 scheduler。

（5）权重衰减：将权重衰减添加到优化器中，这是一种正则化技术，通过向损失函数中添加额外的项来防止权重变得过大。

（6）Dropout：在模型的最后加入一些 Dropout 层，可以在一定程度上抑制模型的过拟合。

（6）Adam 优化器：将优化器由 SGD（随机梯度下降）变为 Adam，它使用了动量和自适应学习率等技术来实现更快的训练，类似于 RMSprop。

（7）预训练模型：为了提高训练效率，减少不必要的重复劳动，PyTorch 官方也提供了一些预训练好的模型供我们使用。合理利用预训练模型可以加快训练速度。

在实验过程中，我查阅了经典神经网络的相关资料，阅读了许多出色的代码，搭建模型和修改模型的能力得到了提升。受限于算力，我没能尝试更多的结构搭配和模型架构，但我惊叹于每一种已经被提出的网络结构，感叹前人智慧的同时也从中学习技巧和精髓，“站在巨人的肩膀上”，不断提升自身能力，争取未来为计算机视觉研究贡献自己的一份力量。

附录 实验源代码

参见提交文件的 src 目录，包含一个 notebook——lab2.ipynb 和若干 Python 源文件。

参考文献

1 Numpy 官方文档. URL：https://numpy.org/doc/stable/index.html

2 PyTorch 官方文档. URL：https://pytorch.org/docs/stable/index.html

3 Stanford CS231n 课程. URL: http://cs231n.stanford.edu/

4 网络结构可视化. URL: https://dgschwend.github.io/netscope/#/preset/vgg-16