实验 2：卷积神经网络

2024 年 3 月 26 日

实验 2：卷积神经网络

1 实验简介

本实验介绍如何利用 PyTorch 搭建卷积神经网络，并基于卷积神经网络创建一个图像分类器。通过本实验，你将掌握以下技能：

- 对图像数据进行预处理
- 利用 PyTorch 搭建并训练简单的卷积神经网络
- 利用 PyTorch 从头开始搭建 ResNet

2 实验环境准备

首先，你需要安装完成本实验所需的必要组件。

2.1 安装 Miniconda

从以下链接下载 Miniconda 安装包并完成安装[Link]，安装过程中需要选中 `Register Anaconda as my default Python 3.10`。若计算机上已经安装好 Miniconda 或 Anaconda，请跳过此步骤。

2.2 安装 Jupyter Notebook

在 Anaconda Prompt 中执行 `pip install notebook`，以安装 Jupyter Notebook。

安装完成后，在 Anaconda Prompt 执行 `jupyter notebook`，检查是否能够成功启动 Notebook。

若计算机上已经安装好 Jupyter Notebook，请跳过此步骤。

2.3 安装必要的 Package

你可以使用以下代码在 Jupyter Notebook 中检查并安装必要的包，也可事先在 Anaconda Prompt 中通过 `pip` 或 `conda` 安装好。

[1]:
```
import pkgutil

# 检查 numpy 是否已安装
if pkgutil.find_loader('numpy'):
    print('numpy is available.')
else:
    !pip install numpy -i https://pypi.tuna.tsinghua.edu.cn/simple/

# 检查 matplotlib 是否已安装
if pkgutil.find_loader('matplotlib'):
    print('matplotlib is available.')
else:
    !pip install matplotlib -i https://pypi.tuna.tsinghua.edu.cn/simple/

# 检查 PIL 是否已安装
if pkgutil.find_loader('PIL'):
    print('PIL is available.')
else:
    !pip install pillow -i https://pypi.tuna.tsinghua.edu.cn/simple/

# 检查 PyTorch 是否已安装
if pkgutil.find_loader('torch'):
    print('PyTorch is available.')
else:
    !pip install torch -i https://pypi.tuna.tsinghua.edu.cn/simple/

# 检查 torchvision 是否已安装
if pkgutil.find_loader('torchvision'):
    print('torchvision is available.')
else:
    !pip install torchvision -i https://pypi.tuna.tsinghua.edu.cn/simple/
```

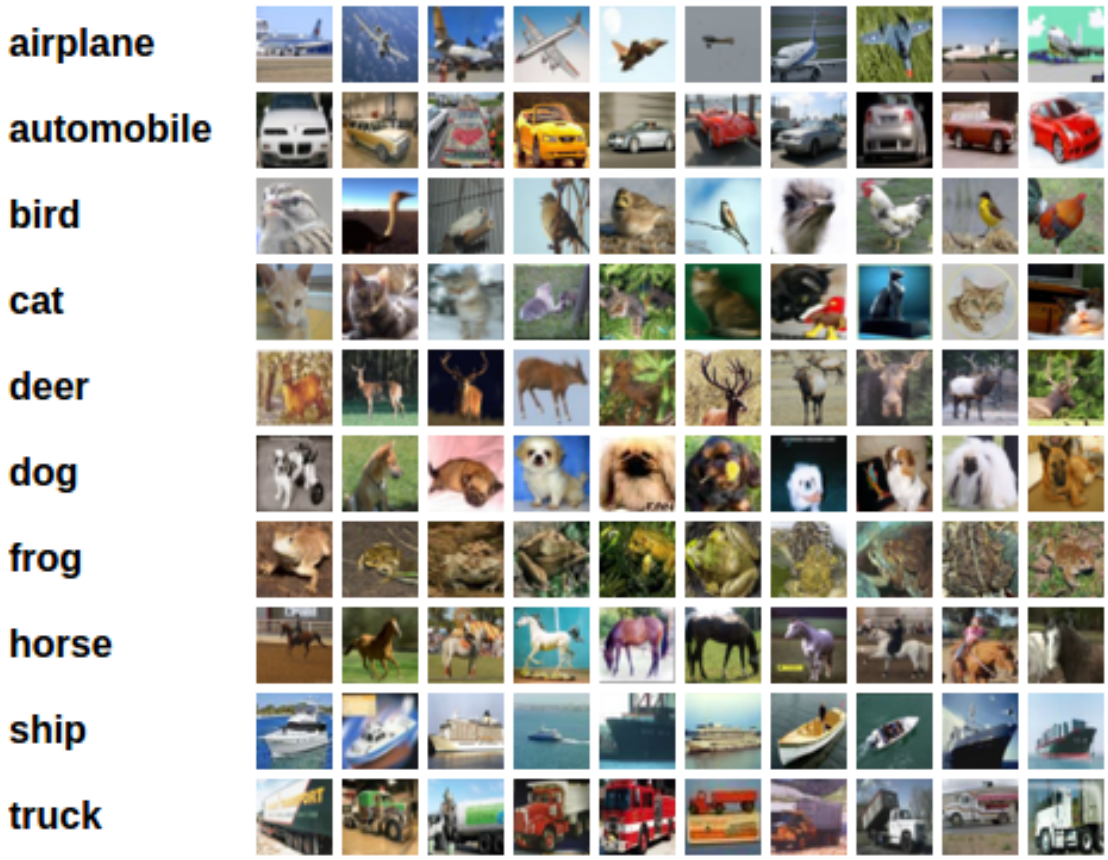
```
# 检查 torchsummary 是否已安装
if pkgutil.find_loader('torchsummary'):
    print('torchsummary is available.')
else:
    !pip install torchsummary -i https://pypi.tuna.tsinghua.edu.cn/simple/
```

```
numpy is available.
matplotlib is available.
PIL is available.
PyTorch is available.
torchvision is available.
torchsummary is available.
```

当然，你可以选择在Kaggle等平台完成实验。在此类平台上，通常已经安装好了常用的包，无需再额外执行以上代码检查包的安装情况，且能够方便地加载各类数据集。

3 CIFAR-10 数据集简介

本实验将使用 CIFAR-10 图像分类数据集。不同于 MNIST 或 FashionMNIST 数据集的是，CIFAR-10 数据集中的图片为 RGB 三通道彩色图片。每张图片的尺寸为 32×32，共包含 10 个类别，每个类别有 6000 张图片，共有 50000 张训练图片和 10000 张测试图片。



4 下载并查看数据

我们使用 `torchvision` 获取 CIFAR-10 数据集，并创建数据加载器（DataLoader）。此外，为了方便 PyTorch 进行处理，首先使用 `torchvision` 自带的数据转换器模块对图片进行适当的预处理。

[3]:
```
import torch
import torchvision
import torchvision.transforms as transforms
```

4.1 下载数据

CIFAR-10 的官方下载地址为[Link]，但下载速度较慢。你可以使用以下链接更快地完成数据集下载。

- [微云]，文件提取码：nwdmtc，在当前 Jupyter Notebook 文件所在的目录下，创建一个 `dataset` 目录，并将下载得到的文件（无需解压）放置在 `dataset` 目录；

- [天池]，下载 `Cifar10.zip` 文件，将其解压，在当前 Jupyter Notebook 文件所在的目录下，创建一个 `dataset` 目录，在刚才解压的目录中找到 `cifar-10-batches-py` 文件夹，并复制到 `datase` 目录。

如果你选择在天池、Kaggle 等平台完成实验，则不需要将数据下载到本地，而是直接挂载相应数据集即可。

4.2 定义数据转换操作

利用 `torchvision` 创建的数据集为 PILImage 图像，每个像素的取值范围为 `[0, 1]`，我们定义一个转换器，用于将图片转换为张量（Tensor），并将每个像素的取值范围标准化为 `[-1, 1]`。

`transforms.Compose` 能够定义一组数据的预处理操作，例如这里定义了转换为张量以及标准化两个预处理操作，将其应用到数据集时，将按顺序执行此处定义的处理。

[4]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

4.3 定义数据集和数据加载器

以下代码会对下载的数据集自动进行解压。请按需修改 `root` 参数。

[5]:
```
batch_size = 4

# 创建 CIFAR10 数据集的训练集，并创建训练集数据加载器
trainset = torchvision.datasets.CIFAR10(root='./dataset', train=True,␣
 ↪download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=2)

# 创建 CIFAR10 数据集的测试集，并创建测试集数据加载器
testset = torchvision.datasets.CIFAR10(root='./dataset', train=False,␣
 ↪download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,␣
 ↪shuffle=False, num_workers=2)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

定义类别标签。

[8]:
```
classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
```

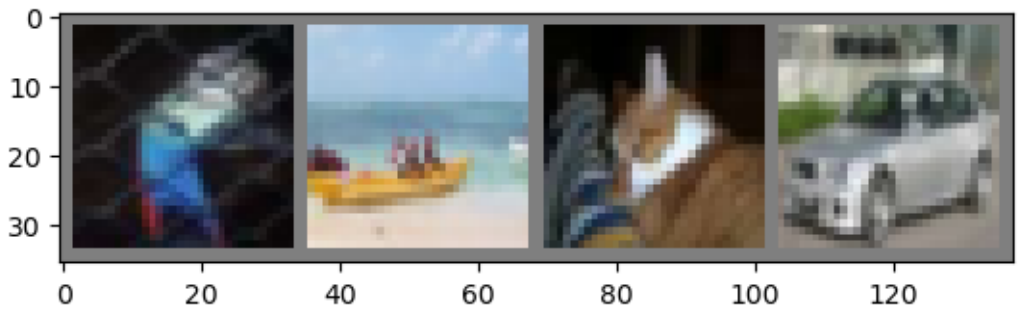
4.4 查看数据

[9]:
```
import matplotlib.pyplot as plt
import numpy as np
import os
os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"


def imshow(img):
    img = img / 2 + 0.5  # 将像素值重新映射到 [0, 1]
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))  # 将数据维度由 (3, 32, 32) 变更为
(32, 32, 3)
    plt.show()


dataiter = iter(trainloader)
images, labels = next(dataiter)

imshow(torchvision.utils.make_grid(images))
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```



```
bird  ship  cat   car
```

5 定义卷积神经网络模型

接下来，定义卷积神经网络拓扑结构、损失函数及优化方法。

5.1 定义卷积神经网络拓扑结构

在 torch.nn 模块中，提供了 Conv2d、MaxPool2d 等卷积神经网络常用的模块，因此，我们可以利用 PyTorch 方便地搭建卷积神经网络。

请检查神经网络中每层的输入、输出维度，以避免创建无效的神经网络配置。

[10]:
```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)  # 在通过全连接层之前，先将数据进行展平（batch 除外）
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
```

```
        x = self.fc3(x)
        return x


net = Net()
```

练习：请绘制上面的代码块所定义的神经网络拓扑结构。

5.2 定义损失函数及优化器

使用交叉熵损失，并采用含动量的随机梯度下降算法。

[12]:
```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

6 训练模型

训练模型通常按照以下顺序组织代码：

- 确定要遍历数据集的次数（即 epoch）；
- 每次迭代，利用 DataLoader 加载数据；
- 重置梯度（防止梯度累加）；
- 前向传播；
- 计算损失函数值；
- 反向传播；
- 利用优化器进行参数更新。

[14]:
```
for epoch in range(2):  # 为减少执行时间，这里仅遍历 2 次

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # data 是一个列表，形如 [inputs, labels]
        inputs, labels = data

        # 将梯度置零
```

```
        optimizer.zero_grad()

        # 前向传播
        outputs = net(inputs)

        # 计算损失
        loss = criterion(outputs, labels)

        # 反向传播
        loss.backward()

        # 利用优化器进行参数更新
        optimizer.step()

        # 输出训练日志
        running_loss += loss.item()
        if i % 2000 == 1999:
            print(f'Epoch: {epoch + 1}, batch: {i + 1:5d}, loss: {running_loss /
↪ 2000:.3f}')
            running_loss = 0.0

print('训练结束！')
```

```
Epoch: 1, batch:  2000, loss: 1.426
Epoch: 1, batch:  4000, loss: 1.407
Epoch: 1, batch:  6000, loss: 1.363
Epoch: 1, batch:  8000, loss: 1.334
Epoch: 1, batch: 10000, loss: 1.309
Epoch: 1, batch: 12000, loss: 1.297
Epoch: 2, batch:  2000, loss: 1.228
Epoch: 2, batch:  4000, loss: 1.251
Epoch: 2, batch:  6000, loss: 1.234
Epoch: 2, batch:  8000, loss: 1.174
Epoch: 2, batch: 10000, loss: 1.185
Epoch: 2, batch: 12000, loss: 1.146
训练结束！
```

可使用 torch.save 将训练好的模型权重存储到磁盘。

[15]:
```
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

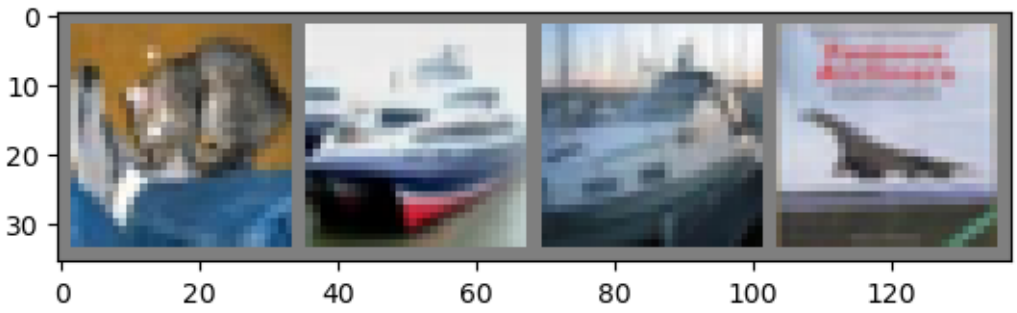
7 测试模型效果

我们已经在训练数据集上遍历 2 次，可以观察到模型损失逐步下降。但是，我们需要在训练结束后，在测试集上检查模型的效果和泛化能力。

首先，我们从测试集中取出几张图片，并查看其真实分类标签。

[16]:
```
dataiter = iter(testloader)
images, labels = next(dataiter)

imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))
```




```
GroundTruth:  cat   ship  ship  plane
```

如果你希望加载保存的模型，可执行以下代码。

[17]:
```
net = Net()
net.load_state_dict(torch.load(PATH))
```

[17]:
```
<All keys matched successfully>
```

检查神经网络模型输出。

[19]:
```
outputs = net(images)
_, predicted = torch.max(outputs, 1)
print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))
```

```
Predicted:  cat   ship  plane plane
```

上述测试结果取决于你的模型训练情况，但一般至少能够在 1 张图片上获得正确结果。接下来，我们进一步检查模型在整个测试数据集上的表现。

[20]:
```
correct = 0
total = 0

# 在测试阶段，我们不需要计算梯度，因此此处关闭梯度计算
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        # 直观地，我们取“能量”值最高的类别作为最终的预测类别
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'模型在测试集的准确率为: {100 * correct // total} %')
```

```
模型在测试集的准确率为: 58 %
```

一般而言，你得到的准确率在 50-60% 之间。相较随机猜测（准确率为 10%）而言，我们的模型已经具备一定可用性，但显然准确率不是很高。

如果你希望进一步检查在每个类别上的分类准确率，可执行以下代码块。

[21]:
```
# 准备用于记录各类别正确分类数及总数的 Dict
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# 测试模式，无需计算梯度
with torch.no_grad():
    for data in testloader:
```

```
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # 对于每个类别，检查有多少样本能够正确分类
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# 计算并输出各类别的准确率
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'类别 {classname:5s} 的准确率为:  {accuracy:.1f} %')
```

```
类别 plane 的准确率为:  70.3 %
类别 car   的准确率为:  72.4 %
类别 bird  的准确率为:  52.1 %
类别 cat   的准确率为:  42.3 %
类别 deer  的准确率为:  51.4 %
类别 dog   的准确率为:  40.6 %
类别 frog  的准确率为:  60.5 %
类别 horse 的准确率为:  71.3 %
类别 ship  的准确率为:  59.8 %
类别 truck 的准确率为:  63.8 %
```

8 模型改进

从模型测试效果中可以看出，刚才我们构建的卷积神经网络模型无法达到很高的分类准确率。我们猜想，其原因可能是训练时间太短，也可能是模型太过简单。

练习：尝试增加上述模型的训练 epoch 数量，观察能否提高准确率。

这里，我们使用 ResNet-18 作为特征提取网络，提升 CIFAR10 图片分类器的性能。

8.1 搭建 ResNet 模型

首先，我们定义 ResNet 的基本结构——ResBlock。

[75]:
```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        # 为了提高通用性，我们将输入通道数、输出通道数、步长等设置为参数
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,␣
↪stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,␣
↪stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # 如果输入输出维度不等，通过 1×1 卷积改变维度
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,␣
↪stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        residual = self.shortcut(x)
        out += residual
        out = torch.relu(out)
        return out
```

[76]:
```
res_block = ResidualBlock(16, 64)
print(res_block)

x = torch.randn(2, 16, 32, 32)
y = res_block(x)
print(y.shape)
```

```
ResidualBlock(
  (conv1): Conv2d(16, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (shortcut): Sequential(
    (0): Conv2d(16, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  )
)
torch.Size([2, 64, 32, 32])
```

接下来，定义 ResNet 网络结构。

[79]:
```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1,␣
 ↪bias=False)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, num_blocks[0], stride=1)
        self.layer2 = self.make_layer(block, 32, num_blocks[1], stride=2)
```

```
        self.layer3 = self.make_layer(block, 64, num_blocks[2], stride=2)
        self.layer4 = self.make_layer(block, 128, num_blocks[3], stride=2)
        self.avg_pool = nn.AvgPool2d(kernel_size=4)
        self.fc = nn.Linear(128, num_classes)

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

    def forward(self, x):
        out = self.conv(x)
        out = self.bn(out)
        out = self.relu(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avg_pool(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

实例化 ResNet 模型。

[80]:
```
from torchsummary import summary

resnet = ResNet(ResidualBlock, [2, 2, 2, 2])
summary(resnet, (3, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 32, 32]             432
       BatchNorm2d-2           [-1, 16, 32, 32]              32
```

ReLU-3	[-1, 16, 32, 32]	0
Conv2d-4	[-1, 16, 32, 32]	2,304
BatchNorm2d-5	[-1, 16, 32, 32]	32
ReLU-6	[-1, 16, 32, 32]	0
Conv2d-7	[-1, 16, 32, 32]	2,304
BatchNorm2d-8	[-1, 16, 32, 32]	32
ResidualBlock-9	[-1, 16, 32, 32]	0
Conv2d-10	[-1, 16, 32, 32]	2,304
BatchNorm2d-11	[-1, 16, 32, 32]	32
ReLU-12	[-1, 16, 32, 32]	0
Conv2d-13	[-1, 16, 32, 32]	2,304
BatchNorm2d-14	[-1, 16, 32, 32]	32
ResidualBlock-15	[-1, 16, 32, 32]	0
Conv2d-16	[-1, 32, 16, 16]	4,608
BatchNorm2d-17	[-1, 32, 16, 16]	64
ReLU-18	[-1, 32, 16, 16]	0
Conv2d-19	[-1, 32, 16, 16]	9,216
BatchNorm2d-20	[-1, 32, 16, 16]	64
Conv2d-21	[-1, 32, 16, 16]	512
BatchNorm2d-22	[-1, 32, 16, 16]	64
ResidualBlock-23	[-1, 32, 16, 16]	0
Conv2d-24	[-1, 32, 16, 16]	9,216
BatchNorm2d-25	[-1, 32, 16, 16]	64
ReLU-26	[-1, 32, 16, 16]	0
Conv2d-27	[-1, 32, 16, 16]	9,216
BatchNorm2d-28	[-1, 32, 16, 16]	64
ResidualBlock-29	[-1, 32, 16, 16]	0
Conv2d-30	[-1, 64, 8, 8]	18,432
BatchNorm2d-31	[-1, 64, 8, 8]	128
ReLU-32	[-1, 64, 8, 8]	0
Conv2d-33	[-1, 64, 8, 8]	36,864
BatchNorm2d-34	[-1, 64, 8, 8]	128
Conv2d-35	[-1, 64, 8, 8]	2,048
BatchNorm2d-36	[-1, 64, 8, 8]	128
ResidualBlock-37	[-1, 64, 8, 8]	0
Conv2d-38	[-1, 64, 8, 8]	36,864
BatchNorm2d-39	[-1, 64, 8, 8]	128

```
              ReLU-40             [-1, 64, 8, 8]               0
            Conv2d-41             [-1, 64, 8, 8]          36,864
      BatchNorm2d-42             [-1, 64, 8, 8]             128
    ResidualBlock-43             [-1, 64, 8, 8]               0
            Conv2d-44            [-1, 128, 4, 4]          73,728
      BatchNorm2d-45            [-1, 128, 4, 4]             256
              ReLU-46            [-1, 128, 4, 4]               0
            Conv2d-47            [-1, 128, 4, 4]         147,456
      BatchNorm2d-48            [-1, 128, 4, 4]             256
            Conv2d-49            [-1, 128, 4, 4]           8,192
      BatchNorm2d-50            [-1, 128, 4, 4]             256
    ResidualBlock-51            [-1, 128, 4, 4]               0
            Conv2d-52            [-1, 128, 4, 4]         147,456
      BatchNorm2d-53            [-1, 128, 4, 4]             256
              ReLU-54            [-1, 128, 4, 4]               0
            Conv2d-55            [-1, 128, 4, 4]         147,456
      BatchNorm2d-56            [-1, 128, 4, 4]             256
    ResidualBlock-57            [-1, 128, 4, 4]               0
        AvgPool2d-58            [-1, 128, 1, 1]               0
            Linear-59                  [-1, 10]           1,290
================================================================
Total params: 701,466
Trainable params: 701,466
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 3.41
Params size (MB): 2.68
Estimated Total Size (MB): 6.09
----------------------------------------------------------------
```

8.2 损失函数及优化器

使用交叉熵损失函数及带动量的 SGD 优化器。

[90]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(resnet.parameters(), lr=1e-2, momentum=0.9,␣
↪weight_decay=5e-4)
```

8.3 训练模型

由于模型更加复杂，模型训练的时间也相应增长。在无 GPU 的环境下，执行完以下代码通常需要 15 分钟左右，可以根据实际情况调整 epoch 等训练超参数。例如，你可以将 epoch 设置为 5，但改变超参可能影响最终模型的准确率。

思考：batch_size 对收敛速度是否有影响？为什么？

[91]:
```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=32,␣
↪shuffle=True, num_workers=2)

for epoch in range(10):  # 为减少执行时间，这里仅遍历 10 次

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # data 是一个列表，形如 [inputs, labels]
        inputs, labels = data

        # 将梯度置零
        optimizer.zero_grad()

        # 前向传播
        outputs = resnet(inputs)

        # 计算损失
        loss = criterion(outputs, labels)

        # 反向传播
        loss.backward()

        # 利用优化器进行参数更新
        optimizer.step()
```

```
        # 输出训练日志
        running_loss += loss.item()
        if i % 100 == 99:
            print(f'Epoch: {epoch + 1}, batch: {i + 1:5d}, loss: {running_loss /
 ↪ 100:.3f}')
            running_loss = 0.0

print('训练结束！')
```

```
Epoch: 1, batch:   100, loss: 2.312
Epoch: 1, batch:   200, loss: 2.290
Epoch: 1, batch:   300, loss: 2.302
Epoch: 1, batch:   400, loss: 2.304
Epoch: 1, batch:   500, loss: 2.303
Epoch: 1, batch:   600, loss: 2.284
Epoch: 1, batch:   700, loss: 2.162
Epoch: 1, batch:   800, loss: 2.083
Epoch: 1, batch:   900, loss: 2.039
Epoch: 1, batch:  1000, loss: 1.961
Epoch: 1, batch:  1100, loss: 1.931
Epoch: 1, batch:  1200, loss: 1.927
Epoch: 1, batch:  1300, loss: 1.970
Epoch: 1, batch:  1400, loss: 1.908
Epoch: 1, batch:  1500, loss: 1.845
Epoch: 2, batch:   100, loss: 1.736
Epoch: 2, batch:   200, loss: 1.735
Epoch: 2, batch:   300, loss: 1.737
Epoch: 2, batch:   400, loss: 1.701
Epoch: 2, batch:   500, loss: 1.670
Epoch: 2, batch:   600, loss: 1.632
Epoch: 2, batch:   700, loss: 1.636
Epoch: 2, batch:   800, loss: 1.629
Epoch: 2, batch:   900, loss: 1.579
Epoch: 2, batch:  1000, loss: 1.592
Epoch: 2, batch:  1100, loss: 1.551
Epoch: 2, batch:  1200, loss: 1.514
Epoch: 2, batch:  1300, loss: 1.508
```

```
Epoch: 2, batch:  1400, loss: 1.469
Epoch: 2, batch:  1500, loss: 1.495
Epoch: 3, batch:   100, loss: 1.404
Epoch: 3, batch:   200, loss: 1.366
Epoch: 3, batch:   300, loss: 1.360
Epoch: 3, batch:   400, loss: 1.398
Epoch: 3, batch:   500, loss: 1.345
Epoch: 3, batch:   600, loss: 1.350
Epoch: 3, batch:   700, loss: 1.391
Epoch: 3, batch:   800, loss: 1.351
Epoch: 3, batch:   900, loss: 1.309
Epoch: 3, batch:  1000, loss: 1.296
Epoch: 3, batch:  1100, loss: 1.305
Epoch: 3, batch:  1200, loss: 1.216
Epoch: 3, batch:  1300, loss: 1.261
Epoch: 3, batch:  1400, loss: 1.273
Epoch: 3, batch:  1500, loss: 1.251
Epoch: 4, batch:   100, loss: 1.155
Epoch: 4, batch:   200, loss: 1.192
Epoch: 4, batch:   300, loss: 1.177
Epoch: 4, batch:   400, loss: 1.160
Epoch: 4, batch:   500, loss: 1.139
Epoch: 4, batch:   600, loss: 1.136
Epoch: 4, batch:   700, loss: 1.136
Epoch: 4, batch:   800, loss: 1.108
Epoch: 4, batch:   900, loss: 1.126
Epoch: 4, batch:  1000, loss: 1.096
Epoch: 4, batch:  1100, loss: 1.088
Epoch: 4, batch:  1200, loss: 1.073
Epoch: 4, batch:  1300, loss: 1.095
Epoch: 4, batch:  1400, loss: 1.040
Epoch: 4, batch:  1500, loss: 1.066
Epoch: 5, batch:   100, loss: 1.000
Epoch: 5, batch:   200, loss: 1.005
Epoch: 5, batch:   300, loss: 1.003
Epoch: 5, batch:   400, loss: 0.961
Epoch: 5, batch:   500, loss: 0.964
```

```
Epoch: 5, batch:   600, loss: 0.994
Epoch: 5, batch:   700, loss: 1.001
Epoch: 5, batch:   800, loss: 0.977
Epoch: 5, batch:   900, loss: 0.956
Epoch: 5, batch:  1000, loss: 0.945
Epoch: 5, batch:  1100, loss: 0.969
Epoch: 5, batch:  1200, loss: 0.946
Epoch: 5, batch:  1300, loss: 0.912
Epoch: 5, batch:  1400, loss: 0.953
Epoch: 5, batch:  1500, loss: 0.920
Epoch: 6, batch:   100, loss: 0.822
Epoch: 6, batch:   200, loss: 0.834
Epoch: 6, batch:   300, loss: 0.845
Epoch: 6, batch:   400, loss: 0.842
Epoch: 6, batch:   500, loss: 0.864
Epoch: 6, batch:   600, loss: 0.840
Epoch: 6, batch:   700, loss: 0.886
Epoch: 6, batch:   800, loss: 0.840
Epoch: 6, batch:   900, loss: 0.816
Epoch: 6, batch:  1000, loss: 0.826
Epoch: 6, batch:  1100, loss: 0.826
Epoch: 6, batch:  1200, loss: 0.835
Epoch: 6, batch:  1300, loss: 0.850
Epoch: 6, batch:  1400, loss: 0.858
Epoch: 6, batch:  1500, loss: 0.849
Epoch: 7, batch:   100, loss: 0.732
Epoch: 7, batch:   200, loss: 0.746
Epoch: 7, batch:   300, loss: 0.719
Epoch: 7, batch:   400, loss: 0.757
Epoch: 7, batch:   500, loss: 0.749
Epoch: 7, batch:   600, loss: 0.724
Epoch: 7, batch:   700, loss: 0.759
Epoch: 7, batch:   800, loss: 0.731
Epoch: 7, batch:   900, loss: 0.749
Epoch: 7, batch:  1000, loss: 0.773
Epoch: 7, batch:  1100, loss: 0.722
Epoch: 7, batch:  1200, loss: 0.753
```

```
Epoch: 7, batch:  1300, loss: 0.746
Epoch: 7, batch:  1400, loss: 0.730
Epoch: 7, batch:  1500, loss: 0.762
Epoch: 8, batch:   100, loss: 0.674
Epoch: 8, batch:   200, loss: 0.661
Epoch: 8, batch:   300, loss: 0.660
Epoch: 8, batch:   400, loss: 0.624
Epoch: 8, batch:   500, loss: 0.658
Epoch: 8, batch:   600, loss: 0.685
Epoch: 8, batch:   700, loss: 0.700
Epoch: 8, batch:   800, loss: 0.663
Epoch: 8, batch:   900, loss: 0.662
Epoch: 8, batch:  1000, loss: 0.692
Epoch: 8, batch:  1100, loss: 0.693
Epoch: 8, batch:  1200, loss: 0.687
Epoch: 8, batch:  1300, loss: 0.682
Epoch: 8, batch:  1400, loss: 0.638
Epoch: 8, batch:  1500, loss: 0.666
Epoch: 9, batch:   100, loss: 0.569
Epoch: 9, batch:   200, loss: 0.567
Epoch: 9, batch:   300, loss: 0.584
Epoch: 9, batch:   400, loss: 0.564
Epoch: 9, batch:   500, loss: 0.580
Epoch: 9, batch:   600, loss: 0.579
Epoch: 9, batch:   700, loss: 0.635
Epoch: 9, batch:   800, loss: 0.599
Epoch: 9, batch:   900, loss: 0.670
Epoch: 9, batch:  1000, loss: 0.596
Epoch: 9, batch:  1100, loss: 0.627
Epoch: 9, batch:  1200, loss: 0.640
Epoch: 9, batch:  1300, loss: 0.644
Epoch: 9, batch:  1400, loss: 0.627
Epoch: 9, batch:  1500, loss: 0.638
Epoch: 10, batch:   100, loss: 0.511
Epoch: 10, batch:   200, loss: 0.496
Epoch: 10, batch:   300, loss: 0.520
Epoch: 10, batch:   400, loss: 0.565
```

```
Epoch: 10, batch:   500, loss: 0.567
Epoch: 10, batch:   600, loss: 0.532
Epoch: 10, batch:   700, loss: 0.589
Epoch: 10, batch:   800, loss: 0.574
Epoch: 10, batch:   900, loss: 0.580
Epoch: 10, batch:  1000, loss: 0.581
Epoch: 10, batch:  1100, loss: 0.534
Epoch: 10, batch:  1200, loss: 0.570
Epoch: 10, batch:  1300, loss: 0.552
Epoch: 10, batch:  1400, loss: 0.553
Epoch: 10, batch:  1500, loss: 0.575
```

训练结束!

8.4 测试模型效果

[92]:
```python
correct = 0
total = 0

# 由于 ResNet 使用了批正则化，测试阶段应将其关闭，故需使用 eval()
resnet.eval()
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = resnet(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'模型在测试集的准确率为: {100 * correct // total} %')
```

```
模型在测试集的准确率为: 72 %
```

一般而言，模型经过大约 10 个 epoch 的训练，准确率就可以达到 70% 以上，大大超过了实验开始阶段搭建的简易模型。

[93]:
```python
# 保存模型权重
PATH = './resnet18.pth'
```

```
torch.save(resnet.state_dict(), PATH)
```

9 使用数据集外的图片进行测试

最后，你可以将模型应用于其他图片，例如你的手机相册中的图片或互联网图片。

探索：更换几张与本实验手册不同的图片，测试分类效果。

[131]:
```
from PIL import Image
import numpy as np
import requests

classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')
PATH = './resnet18.pth'

model = ResNet(ResidualBlock, [2, 2, 2, 2])
model.load_state_dict(torch.load(PATH))  # 加载模型权重
model.eval()

# 读取要预测的图片
url = 'https://www.toopic.cn/public/uploads/image/20200411/20200411154304_97063.
 ↪jpg'
response = requests.get(url, stream=True)
img = Image.open(response.raw)  # 读取图像
img
```

[131]:

[132]:
```
transform = transforms.Compose(
    [transforms.Resize(32),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

img1 = transform(img)

# 为图片增加一个 batch_size 维度
img1 = img1.unsqueeze(0)

output = model(img1)
prob = F.softmax(output, dim=1)
value, predicted = torch.max(output.data, 1)
print('Predicted: ', classes[predicted])
```

```
Predicted:  dog
```

探索：尝试修改代码，在 **GPU** 上完成模型训练和推理。