

姓名和学号？	刘瑜，22020006076
本实验属于哪门课程？	中国海洋大学24秋《软件工程原理与实践》
实验名称？	实验3：卷积神经网络
博客地址？	24秋《软件工程原理与实践》实验3：卷积神经网络
Github仓库地址？	Yulia2233/software_project(github.com)

一、实验内容

【第一部分：代码练习】

(一) MNIST 数据集分类

1.加载数据

```
[1] import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy

# 一个函数，用来计算模型中有多少参数
def get_n_params(model):
    np=0
    for p in list(model.parameters()):
        np += p.nelement()
    return np

# 使用CPU训练，可以在菜单“代码执行工具”->“更改运行时类型”里进行设置
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

个人解读和想法：导入了 PyTorch 和其他用于深度学习的库，并定义了一个函数来计算模型的参数总数。它还根据系统是否支持 GPU，自动选择在 GPU 或 CPU 上进行训练，从而为深度学习模型的训练过程做好准备。

```
[2] input_size = 28*28 # MNIST上的图像尺寸是 28x28
output_size = 10 # 类别为 0 到 9 的数字，因此为十类

train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data', train=True, download=True,
        transform=transforms.Compose(
            [transforms.ToTensor(),
             transforms.Normalize((0.1307,), (0.3081,))])),
    batch_size=64, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('./data', train=False, transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))])),
    batch_size=1000, shuffle=True)

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%[#####] 9912422/9912422 [00:00<00:00, 52045430.21it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

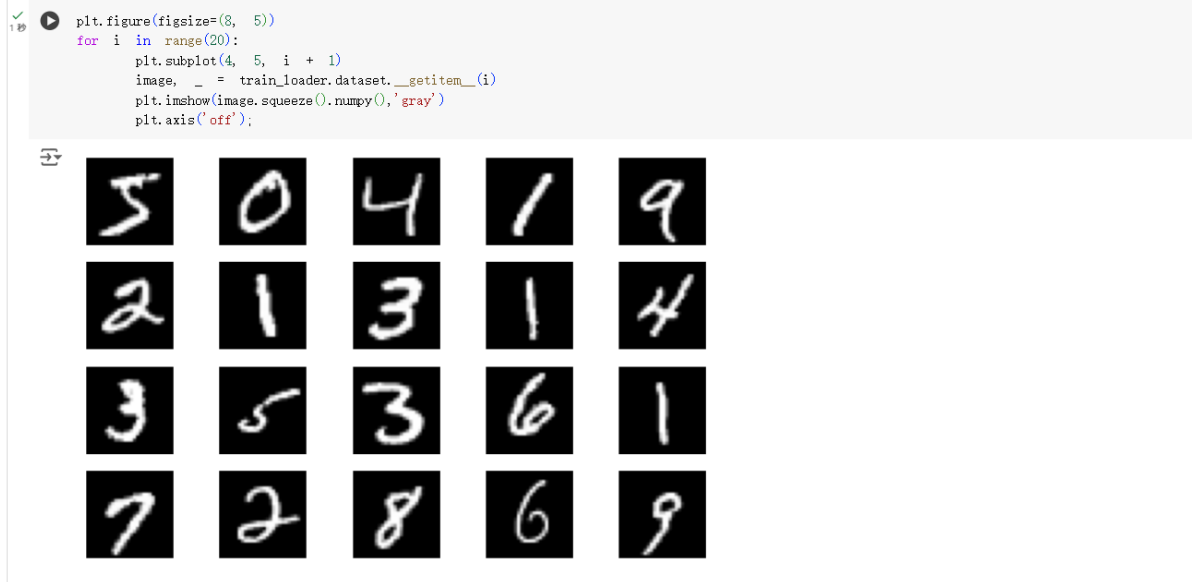
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%[#####] 28881/28881 [00:00<00:00, 47879720.88it/s]Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
```

个人解读和想法：加载并预处理 MNIST 数据集，将图像转换为张量并标准化后，分别创建了用于训练和测试的数据加载器，设置了批量大小和是否随机打乱数据，准备为深度学习模型提供输入。



个人解读和想法：使用 `matplotlib` 将训练数据中的前 20 张 MNIST 图像可视化。通过 `train_loader` 依次获取每张图片，将其转换为 `NumPy` 数组并以灰度图显示，排成 4 行 5 列的网格布局，同时隐藏坐标轴。

2.创建网络

```
[4] class FC2Layer(nn.Module):
    def __init__(self, input_size, n_hidden, output_size):
        # nn.Module子类的函数必须在构造函数中执行父类的构造函数
        # 下式等价于nn.Module.__init__(self)
        super(FC2Layer, self).__init__()
        self.input_size = input_size
        # 这里直接用 Sequential 就定义了网络，注意要和下面 CNN 的代码区分开
        self.network = nn.Sequential(
            nn.Linear(input_size, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, output_size),
            nn.LogSoftmax(dim=1)
        )
    def forward(self, x):
        # view一般出现在model类的forward函数中，用于改变输入或输出的形状
        # x.view(-1, self.input_size) 的意思是多维的数据展成二维
        # 代码指定二维数据的列数为 input_size=784，行数 -1 表示我们不想算，电脑会自己计算对应的数字
        # 在 DataLoader 部分，我们可以看到 batch_size 是64，所以得到 x 的行数是64
        # 大家可以加一行代码，print(x.cpu().numpy().shape)
        # 训练过程中，就会看到 (64, 784) 的输出，和我们的预期是一致的

        # forward 函数的作用是，指定网络的运行过程，这个全连接网络可能看不啥意义，
        # 下面的CNN网络可以看出 forward 的作用。
        x = x.view(-1, self.input_size)
        return self.network(x)
```

```
[4] class CNN(nn.Module):
    def __init__(self, input_size, n_feature, output_size):
        # 执行父类的构造函数，所有的网络都要这么写
        super(CNN, self).__init__()
        # 下面是网络里典型结构的一些定义，一般就是卷积和全连接
        # 池化、ReLU一类的不用在这里定义
        self.n_feature = n_feature
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=n_feature, kernel_size=5)
        self.conv2 = nn.Conv2d(n_feature, n_feature, kernel_size=5)
        self.fc1 = nn.Linear(n_feature*4*4, 50)
        self.fc2 = nn.Linear(50, 10)

    # 下面的 forward 函数，定义了网络的结构，按照一定顺序，把上面构建的一些结构组织起来
    # 意思就是，conv1, conv2 等等的，可以多次重用
    def forward(self, x, verbose=False):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = x.view(-1, self.n_feature*4*4)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.log_softmax(x, dim=1)
        return x
```

个人解读和想法：定义了两个神经网络模型一个是简单的 **全连接两层网络 (FC2Layer)**，另一个是基于卷积的 **卷积神经网络 (CNN)**。两者都继承自 `nn.Module`，并且分别定义了网络的层次结构和前向传播逻辑。

1. FC2Layer:

- 包含两层隐藏层的全连接神经网络。
- 使用 `nn.Sequential` 定义了一个由线性层和 ReLU 激活函数组成的网络，并在最后一层使用 `LogSoftmax` 进行输出。
- `forward` 函数将输入展平为 2D (64, 784) 的形状，并通过定义好的网络进行前向传播。

2. CNN:

- 包含两个卷积层，每个卷积层后跟随 ReLU 激活和最大池化层。
- 两个卷积层之后是两个全连接层，并最终输出使用 `LogSoftmax` 进行分类。
- `forward` 函数描述了输入经过卷积、激活、池化、展平、全连接层的完整过程。

```
[5] # 训练函数
def train(model):
    model.train()
    # 主里从train_loader里, 64个样本一个batch为单位提取样本进行训练
    for batch_idx, (data, target) in enumerate(train_loader):
        # 把数据送到GPU中
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train: [0/0] ([:.0f]%) \t Loss: {:.6f}'.format(
                batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

```
[5]
def test(model):
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        # 把数据送到GPU中
        data, target = data.to(device), target.to(device)
        # 把数据送入模型, 得到预测结果
        output = model(data)
        # 计算本次batch的损失, 并加到 test_loss 中
        test_loss += F.nll_loss(output, target, reduction='sum').item()
        # get the index of the max log-probability, 最后一层输出10个数,
        # 值最大的那个即对应着分类结果, 然后把分类结果保存在 pred 里
        pred = output.data.max(1, keepdim=True)[1]
        # 将 pred 与 target 相比, 得到正确预测结果的数量, 并加到 correct 中
        # 这里需要注意一下 view_as, 意思是把 target 变成维度和 pred 一样的意思
        correct += pred.eq(target.data.view_as(pred)).cpu().sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: 0/0 ([:.0f]%) \n'.format(
        test_loss, correct, len(test_loader.dataset),
        accuracy))
```

个人解读和想法：这段代码定义了模型的训练和测试函数。`train` 函数通过批量加载训练数据，计算损失并使用反向传播和优化器更新模型参数，每隔 100 个批次输出一一次训练进度；`test` 函数在测试数据上评估模型性能，计算平均损失并统计分类准确率，最终输出测试结果。

3.在小型全连接网络上训练

```
[6] n_hidden = 8 # number of hidden units

model_fnn = FC2Layer(input_size, n_hidden, output_size)
model_fnn.to(device)
optimizer = optim.SGD(model_fnn.parameters(), lr=0.01, momentum=0.5)
print('Number of parameters: {}'.format(get_n_params(model_fnn)))

train(model_fnn)
test(model_fnn)
```

Number of parameters: 6442
Train: [0/60000 (0%)] Loss: 2.325240
Train: [6400/60000 (11%)] Loss: 2.160522
Train: [12800/60000 (21%)] Loss: 1.707147
Train: [19200/60000 (32%)] Loss: 1.173558
Train: [25600/60000 (43%)] Loss: 0.684810
Train: [32000/60000 (53%)] Loss: 0.536066
Train: [38400/60000 (64%)] Loss: 0.630924
Train: [44800/60000 (75%)] Loss: 0.315473
Train: [51200/60000 (85%)] Loss: 0.449926
Train: [57600/60000 (96%)] Loss: 0.430666
Test set: Average loss: 0.4126, Accuracy: 8779/10000 (88%)

个人解读和想法：初始化了一个具有 8 个隐藏单元的全连接神经网络 (FC2Layer)，并使用 SGD 优化器进行训练。首先将模型移到 GPU 或 CPU 上，然后通过 `get_n_params` 打印模型的参数总数。接着，调用 `train` 函数对模型进行训练，并使用 `test` 函数在测试集上评估模型的性能。

4.在卷积神经网络上训练

```
[7] # Training settings
n_features = 6 # number of feature maps

model_cnn = CNN(input_size, n_features, output_size)
model_cnn.to(device)
optimizer = optim.SGD(model_cnn.parameters(), lr=0.01, momentum=0.5)
print('Number of parameters: {}'.format(get_n_params(model_cnn)))

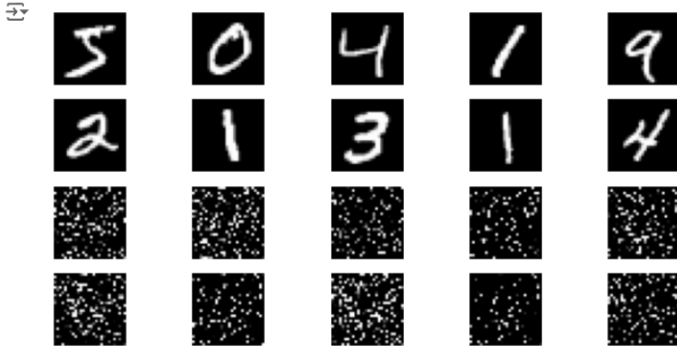
train(model_cnn)
test(model_cnn)
```

Number of parameters: 6422
Train: [0/60000 (0%)] Loss: 2.287000
Train: [6400/60000 (11%)] Loss: 1.167713
Train: [12800/60000 (21%)] Loss: 0.688154
Train: [19200/60000 (32%)] Loss: 0.173318
Train: [25600/60000 (43%)] Loss: 0.589576
Train: [32000/60000 (53%)] Loss: 0.259786
Train: [38400/60000 (64%)] Loss: 0.183031
Train: [44800/60000 (75%)] Loss: 0.181049
Train: [51200/60000 (85%)] Loss: 0.223813
Train: [57600/60000 (96%)] Loss: 0.325131
Test set: Average loss: 0.1615, Accuracy: 9489/10000 (95%)

个人解读和想法：这段代码初始化了一个具有 6 个特征图的卷积神经网络 (CNN)，并使用 SGD 优化器进行训练。首先将模型移到 GPU 或 CPU 上，然后打印模型的参数总数。接着，调用 `train` 函数对模型进行训练，并使用 `test` 函数在测试集上评估其性能。

5. 打乱像素顺序再次在两个网络上训练与测试

```
✓ 1秒 # 这里解释一下 torch.randperm 函数，给定参数n，返回一个从0到n-1的随机整数排列
perm = torch.randperm(784)
plt.figure(figsize=(8, 4))
for i in range(10):
    image, _ = train_loader.dataset.__getitem__(i)
    # permute pixels
    image_perm = image.view(-1, 28*28).clone()
    image_perm = image_perm[:, perm]
    image_perm = image_perm.view(-1, 1, 28, 28)
    plt.subplot(4, 5, i + 1)
    plt.imshow(image.squeeze().numpy(), 'gray')
    plt.axis('off')
    plt.subplot(4, 5, i + 11)
    plt.imshow(image_perm.squeeze().numpy(), 'gray')
    plt.axis('off')
```



个人解读和想法：

1. torch.randperm(784):

- 生成一个从 0 到 783 的随机整数排列，用于随机打乱 MNIST 图像的像素顺序。

2. 可视化原始图像和像素打乱后的图像：

- 从 `train_loader` 中加载前 10 张图像。
- 对每张图像，首先展示其原始版本，然后通过随机排列像素（`image_perm = image_perm[:, perm]`）展示打乱后的版本。
- 使用 `plt.subplot` 将每对图像（原始和打乱）排成 4 行 5 列的网格布局进行展示。

这样，用户可以直观地比较原始图像与其像素打乱后的版本。

```
✓ 0秒 [9] # 对每个 batch 里的数据，打乱像素顺序的函数
def perm_pixel(data, perm):
    # 转化为二维矩阵
    data_new = data.view(-1, 28*28)
    # 打乱像素顺序
    data_new = data_new[:, perm]
    # 恢复为原来4维的 tensor
    data_new = data_new.view(-1, 1, 28, 28)
    return data_new

# 训练函数
def train_perm(model, perm):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        # 像素打乱顺序
        data = perm_pixel(data, perm)

        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train: [0/0] ([:.0f%])\tLoss: {:.6f}'.format(
                batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

```
# 测试函数
def test_perm(model, perm):
    model.eval()
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)

        # 像素打乱顺序
        data = perm_pixel(data, perm)

        output = model(data)
        test_loss += F.nll_loss(output, target, reduction='sum').item()
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: 0/0 ({:.0f}%) \n'.format(
        test_loss, correct, len(test_loader.dataset),
        accuracy))
```

```
✓ [10] perm = torch.randperm(784)
20 秒 n_hidden = 8 # number of hidden units

model_fnn = FC2Layer(input_size, n_hidden, output_size)
model_fnn.to(device)
optimizer = optim.SGD(model_fnn.parameters(), lr=0.01, momentum=0.5)
print('Number of parameters: {}'.format(get_n_params(model_fnn)))

train_perm(model_fnn, perm)
test_perm(model_fnn, perm)
```

```
➡ Number of parameters: 6442
Train: [0/60000 (0%)] Loss: 2.329277
Train: [6400/60000 (11%)] Loss: 1.870516
Train: [12800/60000 (21%)] Loss: 1.115754
Train: [19200/60000 (32%)] Loss: 0.806669
Train: [25600/60000 (43%)] Loss: 0.699961
Train: [32000/60000 (53%)] Loss: 0.468831
Train: [38400/60000 (64%)] Loss: 0.576345
Train: [44800/60000 (75%)] Loss: 0.383968
Train: [51200/60000 (85%)] Loss: 0.215671
Train: [57600/60000 (96%)] Loss: 0.493921

Test set: Average loss: 0.4131, Accuracy: 8831/10000 (88%)
```

个人解读和想法：

perm_pixel 函数：

- 这个函数接受图像数据 `data` 和像素打乱顺序的索引 `perm`。
- 它首先将数据从 4D 转为 2D，然后根据 `perm` 打乱像素顺序，最后恢复为原来的 4D 张量格式。

训练函数 train_perm：

- 训练过程中，每次提取数据时，都会对图像像素进行打乱（调用 `perm_pixel` 函数）。
- 然后进行前向传播、计算损失、反向传播并更新权重。每 100 个批次输出一一次训练进度和损失。

测试函数 test_perm：

- 在测试过程中，同样对图像像素进行打乱。
- 计算测试集上的平均损失和准确率，最后输出测试结果。

模型初始化和训练：

- 初始化了一个具有 8 个隐藏单元的全连接神经网络（`model_fnn`），并使用随机梯度下降（SGD）优化器。
- 使用 `train_perm` 进行训练，并通过 `test_perm` 在打乱像素顺序后的测试集上评估模型的性能。

```
[11] perm = torch.randperm(784)
n_features = 6 # number of feature maps

model_cnn = CNN(input_size, n_features, output_size)
model_cnn.to(device)
optimizer = optim.SGD(model_cnn.parameters(), lr=0.01, momentum=0.5)
print('Number of parameters: {}'.format(get_n_params(model_cnn)))

train_perm(model_cnn, perm)
test_perm(model_cnn, perm)

Number of parameters: 6422
Train: [0/60000 (0%)] Loss: 2.285706
Train: [6400/60000 (11%)] Loss: 2.246545
Train: [12800/60000 (21%)] Loss: 2.133343
Train: [19200/60000 (32%)] Loss: 1.671137
Train: [25600/60000 (43%)] Loss: 1.386889
Train: [32000/60000 (53%)] Loss: 0.786171
Train: [38400/60000 (64%)] Loss: 1.009412
Train: [44800/60000 (75%)] Loss: 0.998665
Train: [51200/60000 (85%)] Loss: 0.810229
Train: [57600/60000 (96%)] Loss: 0.724922

Test set: Average loss: 0.6130, Accuracy: 8172/10000 (82%)
```

个人解读和想法：首先，定义了一个具有 6 个特征图的卷积神经网络，并使用随机梯度下降（SGD）作为优化器，学习率设为 0.01，动量为 0.5。在训练过程中，通过 `train_perm` 函数每个批次都对图像像素顺序进行打乱，以模拟图像数据的变化。然后，使用 `test_perm` 函数在测试集上评估模型性能，计算并输出模型的损失和准确率。这种方法有助于测试模型对像素顺序变化的鲁棒性，即在像素位置打乱的情况下，模型能否仍然准确地进行分类。

（二）CIFAR10 数据集分类

1.初始化数据集

```
[1] import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

# 使用GPU训练，可以在菜单“代码执行工具”->“更改运行时类型”里进行设置
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

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

# 注意下面代码中：训练的 shuffle 是 True，测试的 shuffle 是 false
# 训练时可以打乱顺序增加多样性，测试是没有必要
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                       download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch_size=8,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

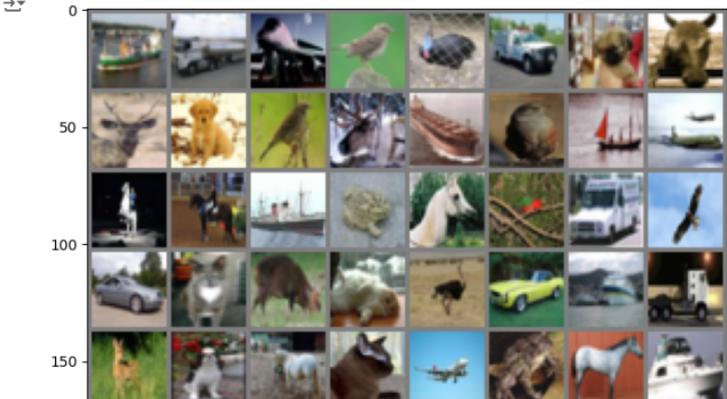
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:03<00:00, 44892207.29it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

个人解读和想法：加载了 CIFAR-10 数据集，并为后续的深度学习任务准备了训练和测试数据。首先，它通过 `torchvision.transforms.Compose` 创建了一个数据预处理流水线，对图像进行标准化处理，使其像素值在 [-1, 1] 范围内。然后，使用 `torchvision.datasets.CIFAR10` 下载并加载 CIFAR-10 数据集，包含 60,000 张 32x32 彩色图像，分为 10 个类别。训练集的加载器（`trainloader`）设置了批大小为 64，数据顺序在训练时会打乱（`shuffle=True`），以增加模型的泛化能力。测试集的加载器（`testloader`）则批量大小为 8，并且数据顺序保持不变（`shuffle=False`）。这些数据将被送入神经网络进行训练和评估。同时，定义了一个包含 CIFAR-10 所有类别的列表，方便后续的标签解析。

2.展示图片

```
def imshow(img):
    plt.figure(figsize=(8,8))
    img = img / 2 + 0.5 # 转换到 [0,1] 之间
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# 得到一组图像
images, labels = next(iter(trainloader))
# 展示图像
imshow(torchvision.utils.make_grid(images))
# 展示第一行图像的标签
for j in range(8):
    print(classes[labels[j]])
```



个人解读和想法：首先，它将输入的图像张量转换到 [0,1] 范围，然后通过 NumPy 转换并调整维度以便通过 matplotlib 可视化。接着，代码从训练数据加载器中提取一批图像，并使用 imshow 函数以网格形式展示这批图像。最后，通过迭代显示每个图像的标签，对应 CIFAR-10 中的类别名称。

3.定义网络，损失函数和优化器

```
[2] class Net(nn.Module):
    def __init__(self):
        super(Net, self).__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 = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# 网络放到GPU上
net = Net().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)
```

个人解读和想法：定义了一个卷积神经网络 Net，继承自 nn.Module。网络结构包括两个卷积层 conv1 和 conv2，每个卷积层后都有一个 ReLU 激活函数和最大池化层。卷积层之后，特征图被展平成一维，接着通过三个全连接层 fc1、fc2 和 fc3，输出对应于 CIFAR-10 的10个类别。损失函数使用交叉熵损失 nn.CrossEntropyLoss，优化器采用 Adam 算法，学习率为 0.001。网络模型被移动到 GPU 设备（如果可用）以加速训练。

4.训练网络

```
[3] for epoch in range(10): # 重复多轮训练
    for i, (inputs, labels) in enumerate(trainloader):
        inputs = inputs.to(device)
        labels = labels.to(device)
        # 优化器梯度归零
        optimizer.zero_grad()
        # 正向传播 + 反向传播 + 优化
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # 输出统计信息
        if i % 100 == 0:
            print('Epoch: %d Minibatch: %5d loss: %.3f' %(epoch + 1, i + 1, loss.item()))

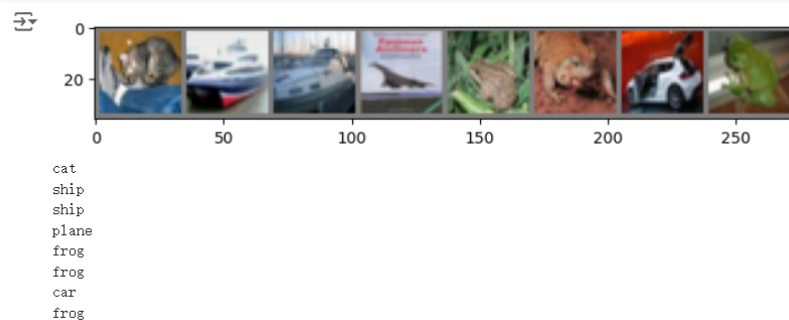
print('Finished Training')
```

```
Epoch: 1 Minibatch: 1 loss: 2.292
Epoch: 1 Minibatch: 101 loss: 1.848
Epoch: 1 Minibatch: 201 loss: 1.633
Epoch: 1 Minibatch: 301 loss: 1.583
Epoch: 1 Minibatch: 401 loss: 1.503
Epoch: 1 Minibatch: 501 loss: 1.585
Epoch: 1 Minibatch: 601 loss: 1.475
Epoch: 1 Minibatch: 701 loss: 1.540
Epoch: 2 Minibatch: 1 loss: 1.688
Epoch: 2 Minibatch: 101 loss: 1.624
Epoch: 2 Minibatch: 201 loss: 1.587
Epoch: 2 Minibatch: 301 loss: 1.475
Epoch: 2 Minibatch: 401 loss: 1.399
Epoch: 2 Minibatch: 501 loss: 1.443
Epoch: 2 Minibatch: 601 loss: 1.140
Epoch: 2 Minibatch: 701 loss: 1.451
Epoch: 3 Minibatch: 1 loss: 1.217
Epoch: 3 Minibatch: 101 loss: 1.158
Epoch: 3 Minibatch: 201 loss: 1.508
```

个人解读和想法：进行模型的训练，循环10个训练周期（epochs）。在每个周期内，遍历 `trainloader` 中的所有数据批次（minibatch）。每次迭代中，输入图像和标签都被转移到GPU设备上。然后，优化器的梯度被归零，执行正向传播得到输出，再计算损失，通过反向传播计算梯度并更新模型参数。每训练100个小批次，打印当前的周期数、批次数量和损失值。训练结束后，输出“Finished Training”提示。

5.取出测试集中的图片并查看模型识别情况

```
[7] # 得到一组图像
images, labels = next(iter(testloader))
# 展示图像
imshow(torchvision.utils.make_grid(images))
# 展示图像的标签
for j in range(8):
    print(classes[labels[j]])
```



```
✓ 0 秒 [8] outputs = net(images.to(device))
_, predicted = torch.max(outputs, 1)

# 展示预测的结果
for j in range(8):
    print(classes[predicted[j]])

cat
ship
ship
plane
deer
frog
cat
bird
```

个人解读和想法：从测试数据集中获取一批图像并使用 `imshow` 函数进行展示，同时输出这些图像的真实标签。然后，模型 `net` 对这些图像进行预测，使用 `torch.max` 函数获取模型的预测结果中概率最大的类别索引。最后，代码将每张图像的预测结果输出，与真实标签进行对比，展示模型对这批测试图像的分类预测。

6.查看模型整体表现

```
✓ 14 秒 [9] correct = 0
total = 0

for data in testloader:
    images, labels = data
    images, labels = images.to(device), labels.to(device)
    outputs = net(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

Accuracy of the network on the 10000 test images: 62 %
```

个人解读和想法：用于计算模型在测试数据集上的准确率。首先，遍历 `testloader` 中的所有测试图像和对应标签，将它们转移到 GPU 设备上。然后，模型对每组测试图像进行预测，并通过 `torch.max` 获取预测结果中概率最大的类别索引。接着，统计模型预测正确的数量 `correct` 和总样本数 `total`。最后，输出模型在10000张测试图像上的分类准确率，以百分比的形式显示。

（三）使用 VGG16 对 CIFAR10 分类

1.定义 dataloader

```

✓ [18] import torch
1秒 import torchvision
import torchvision.transforms as transforms
import numpy as np
from torch.utils.data import Subset

# 使用GPU训练
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

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

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)

# 只使用20%的数据
num_train_samples = int(len(trainset) * 0.1)
num_test_samples = int(len(testset) * 0.1)

# 随机选择子集
train_indices = np.random.choice(len(trainset), num_train_samples, replace=False)
test_indices = np.random.choice(len(testset), num_test_samples, replace=False)

# 创建子集
trainset = Subset(trainset, train_indices)
testset = Subset(testset, test_indices)

# 创建 DataLoader
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=128, shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

```

个人解读和想法：首先，定义了不同的图像预处理方式：训练集采用随机裁剪、水平翻转和归一化，测试集只进行归一化。然后，从 CIFAR-10 数据集中加载训练和测试数据，并分别随机选择其中 10% 的数据作为子集来进行训练和测试。最后，使用 `torch.utils.data.DataLoader` 创建训练和测试数据加载器，确保训练数据按批次打乱加载，测试数据保持顺序加载。

2.VGG 网络定义

```

✓ [19] class VGG(nn.Module):
0秒 def __init__(self):
    super(VGG, self).__init__()
    self.cfg = [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M']
    self.features = self._make_layers(self.cfg)
    self.classifier = nn.Linear(512, 10)

    def forward(self, x):
        out = self.features(x)
        out = out.view(out.size(0), -1)
        out = self.classifier(out)
        return out

    def _make_layers(self, cfg):
        layers = []
        in_channels = 3
        for x in cfg:
            if x == 'M':
                layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
            else:
                layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
                           nn.BatchNorm2d(x),
                           nn.ReLU(inplace=True)]
                in_channels = x
        layers += [nn.AvgPool2d(kernel_size=1, stride=1)]
        return nn.Sequential(*layers)

```

个人解读和想法：定义了一个基于 VGG 网络结构的卷积神经网络。类 `VGG` 继承自 `nn.Module`，其中 `cfg` 配置了卷积层的通道数和池化层位置。`features` 部分通过 `_make_layers` 函数根据 `cfg` 创建卷积层、批归一化层和 ReLU 激活层的组合，以及最大池化层 `M`。这些层逐步提取特征图。`classifier` 是一个全连接层，用于将512维特征映射到10个类别。`forward` 函数执行前向传播，提取特征后展平数据并通过分类器进行预测。

3.网络训练

```
[20] # 网络放到GPU上
net = VGG().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

[21] for epoch in range(5): # 重复多轮训练
    for i, (inputs, labels) in enumerate(trainloader):
        inputs = inputs.to(device)
        labels = labels.to(device)
        # 优化器梯度归零
        optimizer.zero_grad()
        # 正向传播 + 反向传播 + 优化
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # 输出统计信息
        if i % 100 == 0:
            print('Epoch: %d Minibatch: %5d loss: %.3f' % (epoch + 1, i + 1, loss.item()))

    print('Finished Training')
```

Epoch: 1 Minibatch:	1	loss: 2.516
Epoch: 2 Minibatch:	1	loss: 1.913
Epoch: 3 Minibatch:	1	loss: 1.583
Epoch: 4 Minibatch:	1	loss: 1.611
Epoch: 5 Minibatch:	1	loss: 1.336
Finished Training		

个人解读和想法：首先，模型被移到 GPU 上运行，并定义了交叉熵损失函数 `nn.CrossEntropyLoss` 和 Adam 优化器，学习率为 0.001。训练循环执行 5 个周期（epochs），在每个周期中，遍历所有训练数据。对于每个数据批次，输入图像和标签被转移到 GPU 上，接着通过以下步骤：优化器梯度归零、前向传播计算输出、计算损失、反向传播更新梯度、优化器更新参数。每训练100个小批次，打印当前的周期数、批次数和损失值。训练结束后输出“Finished Training”提示。

4.测试验证准确率

```
correct = 0
total = 0

for data in testloader:
    images, labels = data
    images, labels = images.to(device), labels.to(device)
    outputs = net(images)
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %.2f %%' % (
    100 * correct / total))
```

Accuracy of the network on the 10000 test images: 50.90 %

个人解读和想法：遍历 `testloader` 中的所有测试图像，将图像和标签转移到 GPU 上。模型对每组测试图像进行前向传播，使用 `torch.max` 获取预测结果中概率最高的类别索引。随后，统计正确预测的样本数量 `correct` 和总测试样本数量 `total`。最后，计算并打印模型在整个测试集上的准确率，以百分比的形式保留两位小数。

【第二部分：问题总结】

1. DataLoader 里面 shuffle 取不同值有什么区别？

在 PyTorch 的 `DataLoader` 中，`shuffle` 参数的作用是控制数据是否在每个 epoch 之前进行随机打乱。

设置 `shuffle=True` 时，数据会在每个 epoch 开始前被随机打乱，这通常用于训练集，因为打乱数据顺序能够提高模型的泛化能力，减少过拟合的可能性。通过随机化数据顺序，模型不会依赖于样本的排列方式，从而能更全面地学习数据的特征。虽然打乱数据集需要额外的计算，稍微增加了训练时间，但整体影响微乎其微。

`shuffle=False` 时，数据按照固定的顺序提供给模型，这在评估或测试阶段尤其有用，因为此时我们希望模型在每次评估时都处理相同的顺序数据，从而获得一致的评估结果。此外，对于某些任务，如时间序列模型，保持数据顺序是至关重要的。在测试集中，通常会设置 `shuffle=False`。

2.transform 里，取了不同值，这个有什么区别？

在 PyTorch 中，`transform` 是用来对输入数据进行预处理和数据增强的工具，通常用于图像数据。通过在数据加载时使用不同的 `transform` 操作，可以改变输入数据的表现形式，从而影响模型的训练效果。`transform` 里的不同操作和参数能够在训练过程中发挥不同的作用。

比如，`transform_train` 和 `transform_test` 使用了不同的变换操作：

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])

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

在 `transform_train` 中，使用了 `RandomCrop` 和 `RandomHorizontalFlip`，这两者是常见的数据增强操作。`RandomCrop` 会在图像的不同位置随机裁剪出一个 32x32 像素的子区域，并通过填充来保持图像大小不变。这有助于模型更好地泛化，因为它增加了图像的变化性，防止模型仅学习到特定像素位置的特征。`RandomHorizontalFlip` 会随机水平翻转图像，从而增强数据集的多样性。因为真实世界中的物体可能在不同方向出现，这些数据增强操作能帮助模型在不同的情况下更好地识别物体。

`transform_test` 中没有使用这些随机数据增强操作，仅有 `ToTensor` 和 `Normalize`。`ToTensor` 将图像数据从 PIL 图像或 NumPy 数组转换为 PyTorch 的张量格式，方便后续计算。`Normalize` 则是对图像的每个通道（RGB）进行归一化处理，使其值分布在一个较小的范围内，这有助于加快模型的收敛速度并提高数值稳定性。

`transform_train` 和 `transform_test` 的区别主要体现在数据增强操作上。训练时，`RandomCrop` 和 `RandomHorizontalFlip` 等随机变换增加了数据的多样性，帮助模型更好地泛化。而在测试阶段，为了获得一致的评估结果，没有使用这些随机操作，仅进行了图像转换和归一化，确保测试集输入数据的标准化和稳定性。

3.epoch 和 batch 的区别？

Epoch（周期） 指的是整个训练数据集通过模型一次的过程。每经过一次完整的训练集，称为一个 epoch。通常，训练会进行多个 epoch，以便模型能逐步优化。

Batch（小批次） 指的是将训练数据集分成小的子集，每个子集用于一次模型的前向传播和反向传播。在每个 batch 上计算梯度并更新模型参数，而不是在整个数据集上进行。这种方式提高了计算效率，并帮助避免内存溢出。

4.1x1的卷积和 FC 有什么区别？主要起什么作用？

4x1 的卷积和全连接层（FC，Fully Connected Layer） 在结构和作用上有所不同。**4x1 卷积** 是一个局部连接的卷积操作，通常用于处理序列数据或图像的某一维度，主要通过卷积核在输入数据上滑动，以捕捉局部特征（如时间序列中的局部模式）。它的权重在整个输入数据中共享，减少了参数数量。

而 **全连接层（FC）** 是一个将每个输入节点与每个输出节点完全连接的层，在每个节点上都有独立的权重，通常用于最后的分类或回归任务，能够学习全局的特征组合。

4x1 卷积 主要用于局部特征提取和降维，参数共享减少了计算量，而 **全连接层** 主要用于全局特征的组合和最终的决策。

5.residual learning 为什么能够提升准确率？

Residual learning (残差学习) 通过引入**跳跃连接** (skip connections)，允许模型直接学习输入与输出之间的差异，而不是直接学习完整的映射。这样，网络可以更容易地优化和训练深层网络，避免了深度网络在训练过程中常见的梯度消失或梯度爆炸问题。残差结构使得信息可以在层与层之间更顺畅地流动，从而减轻了深层网络的优化难度，提高了模型的准确性。此外，残差学习还能够更好地捕捉到低级和高级特征的组合，进一步提升模型的泛化能力。

6.代码练习二里，网络和1989年 Lecun 提出的 LeNet 有什么区别？

网络结构复杂度：

- **LeNet** 主要由两个卷积层和两个全连接层组成，结构相对较简单，适用于较小的输入图像（如 28x28 的手写数字图像）。LeNet 的卷积层使用了较小的卷积核（通常是 5x5），并且每个卷积层后面都有池化层。
- **网络 (Net)** 也包含两个卷积层和三个全连接层，但在设计上，卷积层的数量较少，且使用了更大的卷积核（5x5），而且在最后一个全连接层之前使用了两次最大池化。

池化层的使用：

- **LeNet** 使用了平均池化 (Average Pooling) 来减少特征图的大小。
- **网络** 使用的是最大池化 (Max Pooling)，这种池化方式更常见，它保留了特征图中的最强信息。

全连接层的结构：

- **LeNet** 的全连接层较少，只有两个全连接层，且输出的节点较少。
- **网络** 有三个全连接层，输出节点更多，且输出的类别数为 10（对应 CIFAR-10 的类别数），LeNet 则主要用于分类 28x28 的手写数字（10 个类别）。

网络的目标和应用：

- **LeNet** 主要用于手写数字识别（如 MNIST 数据集），输入的图像较小，且较为简单。
- **网络** 设计用于处理 CIFAR-10 数据集，这些图像较大且包含更多的类别（10 个类别），需要更复杂的特征提取能力。

激活函数的使用：

- **LeNet** 使用了 Sigmoid 和 Tanh 激活函数。
- **网络** 使用了 ReLU 激活函数，因为 ReLU 能够更有效地缓解梯度消失问题。

7.代码练习二里，卷积以后 feature map 尺寸会变小，如何应用 Residual Learning？

在卷积神经网络中，经过卷积层和池化层后，特征图 (feature map) 的尺寸通常会变小，这可能导致在深层网络中丧失细节信息。**Residual Learning (残差学习)** 通过引入跳跃连接 (skip connections)，允许原始输入与卷积层的输出相加，从而保持更多的细节信息。在每个残差块中，网络学习的是输入和输出之间的“残差”，而不是直接学习输出。这样，即使特征图尺寸变小，网络仍然可以通过残差连接保留原始输入的信息，从而减轻梯度消失问题，帮助模型更好地训练并提高准确性。

8.有什么方法可以进一步提升准确率？

首先，**增加网络的复杂度** 是一种常用策略，可以通过加深网络的层数或增加每层的神经元数量，来让模型学习到更丰富的特征。其次，**数据增强** 也是有效的方法，利用随机裁剪、翻转、旋转、颜色变化等技术，可以在不增加数据量的情况下生成更多多样化的训练样本，从而提高模型的泛化能力。其次，**正则化技术**（如 L2 正则化、Dropout 等）可以防止模型过拟合，促使其在测试数据上表现更好。最后，调

优化器（如使用自适应学习率的AdamW等）以及**学习率调度**，也可以帮助模型在训练过程中更快、更稳健地收敛，从而进一步提升准确率。

二、问题总结与体会

（一）问题总结：

1.第二段代码练习中，代码出现错误，修改后如下：

```
# 得到一组图像

images, labels = next(iter(testloader))

# 展示图像

imshow(torchvision.utils.make_grid(images))

# 展示图像的标签

for j in range(8):

    print(classes[labels[j]])
```

修改行为：images, labels = next(iter(testloader))

2.第三段代码练习中，代码运行时间过长。

首先，减少数据集：

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_test)

# 只使用10%的数据
num_train_samples = int(len(trainset) * 0.1)
num_test_samples = int(len(testset) * 0.1)

# 随机选择子集
train_indices = np.random.choice(len(trainset), num_train_samples, replace=False)
test_indices = np.random.choice(len(testset), num_test_samples, replace=False)

# 创建子集
trainset = Subset(trainset, train_indices)
testset = Subset(testset, test_indices)

# 创建 DataLoader
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True,
num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=128, shuffle=False,
num_workers=2)
```

其次，减少训练轮数：

```
for epoch in range(5): # 重复多轮训练
    for i, (inputs, labels) in enumerate(trainloader):
        inputs = inputs.to(device)
        labels = labels.to(device)
        # 优化器梯度归零
        optimizer.zero_grad()
        # 正向传播 + 反向传播 + 优化
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # 输出统计信息
        if i % 100 == 0:
            print('Epoch: %d Minibatch: %5d loss: %.3f' %(epoch + 1, i + 1,
loss.item()))

print('Finished Training')
```

(二) 体会

我通过实验加深了对卷积神经网络架构的理解，包括卷积层、池化层和全连接层的搭配使用。同时，我在训练过程中尝试了数据增强、正则化等技术，发现这些方法可以有效提高模型的泛化能力和准确率。此外，实验中我还遇到了一些问题，如数据加载时间过长和内存占用较高等，通过减少数据集规模、优化批量大小等措施，我成功提升了训练效率。这次实验帮助我掌握了如何优化神经网络模型，同时也让我意识到模型调优和实验设置的重要性。这不仅提升了我对深度学习的实际应用能力，还让我对卷积神经网络的工作原理有了更深的理解。