
开源软件基础大作业

基于开源仓库的卷积神经网络图像识别项目复 现和准确率优化

**Convolutional neural network image recognition project
reproduction and accuracy optimization based on open source
repository**

学 院 （系）： _____ 软件学院 _____

专 业： _____ 软件工程 _____

大连理工大学

Dalian University of Technology

● 成员及分工

姓名	学号	角色	分工占比
杨安逸	20212241418	组长	40%
夏培新	20212241119	组员	30%
陈冠宇	20212241320	组员	30%

目 录

1	项目概述	1
2	实现过程	1
2.1	数据集获取和处理	1
2.2	卷积神经网络构建	2
2.3	开源项目获取	2
3	具体实现	2
3.1	爬虫代码构建	2
3.2	数据预处理	4
3.3	开源仓库拉取	4
3.4	卷积神经网络构建	5
3.5	学习过程	6
3.6	训练和测试过程	7
4	运行结果展示	8
4.1	爬虫获取到的数据集	8
4.2	训练结果（Adam 优化器）	9
4.3	卷积神经网络模型结构	10
4.4	识别过程展示	10
5	对结果优化的几种尝试	10
5.1	数据集优化	11
5.2	激活函数的选取	12
5.3	优化器的选取	12
5.4	目前最优方案的组合	14
6	总结	14

1 项目概述

计算机视觉是一门人工智能领域的重要部分，它是指让计算机和系统能够从图像、视频和其他视觉输入中获取有意义的信息，并根据该信息采取行动或提供建议。计算机视觉的工作原理与人类视觉类似，只不过人类起步更早。计算机视觉的工作需要大量数据，它一遍又一遍地运行数据分析，直到能够辨别差异并最终识别图像为止。例如，要训练一台计算机识别汽车轮胎，需要为其输入大量的轮胎图像和轮胎相关数据，供其学习轮胎差异和识别轮胎，尤其是没有缺陷的轮胎。这个过程会用到两种关键技术：一种是机器学习，另一种是深度学习，比如卷积神经网络（CNN）。

卷积神经网络是一类包含卷积计算且具有深度结构的前馈神经网络，是深度学习的代表算法之一。卷积神经网络具有表征学习能力，能够按其阶层结构对输入信息进行平移不变分类，因此也被称为“平移不变人工神经网络”。

本项目基于开源项目 https://github.com/ffffff666/Deep_Learning-ComputerVision.git，该仓库内部有各种神经网络的学习文档和实现方法。出于对深度学习的个人兴趣，本项目基于开源项目中的方法，通过自行搭建卷积神经网络并优化，实现对狮子（lion）和狗（dog）的图片识别，这是一个二分类问题。因为狮子和狗的外形较为相像，颜色也难以区分，所以将识别正确率提升到 80% 以上是一个不小的挑战。

2 实现过程

2.1 数据集获取和处理

网络爬虫也叫做网络机器人，可以代替人们自动地在互联网中进行数据信息的采集与整理。在大数据时代，信息的采集是一项重要的工作，如果单纯靠人力进行信息采集，不仅低效繁琐，搜集的成本也会提高。我们通过使用爬虫技术，获取“狮子”和“狗”的相关图片。

由于不同网站的访问权限不同，可能会出现访问超时和图片格式错误的情况。因此在获得原始数据集后要对数据集进行筛选：去掉重复的图像，去掉格式损坏的图像，去掉无关的图像。之后对数据集打标签，在 pytorch 框架中，根据图片所属的文件夹不同，会自动为文件夹内的图片打上相应的标签，如下代码所示：

```
classess=dataset_train.classes  
class_to_idxes=dataset_train.class_to_idx
```

2.2 卷积神经网络构建

卷积神经网络（Convolutional Neural Networks, CNN）是一类包含卷积计算且具有深度结构的前馈神经网络（Feedforward Neural Networks），是深度学习（deep learning）的代表算法之一。卷积神经网络具有表征学习（representation learning）能力，能够按其阶层结构对输入信息进行平移不变分类（shift-invariant classification），因此也被称为“平移不变人工神经网络”。通过定义不同的卷积核，生成多通道的特征图像，经过池化操作后，传送到全连接层，经过 softmax 函数处理后输出分类结果。

卷积神经网络在监督学习中使用 BP 框架进行学习，其计算流程在 LeCun (1989) 中就已经确定，是最早在 BP 框架进行学习的深度算法之一。卷积神经网络中的 BP 分为三部分，即全连接层与卷积核的反向传播和池化层的反向通路。全连接层的 BP 计算与传统的前馈神经网络相同，卷积层的反向传播是一个与前向传播类似的交叉相关计算

2.3 开源项目获取

卷积神经网络在监督学习中使用 BP 框架进行学习，其计算流程在 LeCun (1989) 中就已经确定，是最早在 BP 框架进行学习的深度算法之一。卷积神经网络中的 BP 分为三部分，即全连接层与卷积核的反向传播和池化层的反向通路。全连接层的 BP 计算与传统的前馈神经网络相同，卷积层的反向传播是一个与前向传播类似的交叉相关计算

GitHub 是一个存放软件代码的网站。具有以下优势：软件开源，即编写软件的代码对所有人公开，所有人可以在现有代码的基础上进行二次开发，减少不必要的重复劳动（IT 行话简称不要重复「造轮子」）。方便团队协作。这个过程有点像是我们把文档放在石墨或语雀这类支持团队协作的平台上，而 GitHub 上存放的是代码，参与编写软件的人可以通过 Git（版本控制工具）从 GitHub 拉取或往 GitHub 上传代码

3 具体实现

3.1 爬虫代码构建

卷积神经网络在监督学习中使用 BP 框架进行学习，其计算流程在 LeCun (1989) 中就已经确定，是最早在 BP 框架进行学习的深度算法之一。卷积神经网络中的 BP 分为三部分，即全连接层与卷积核的反向传播和池化层的反向通路。全连接层的 BP 计算与传统的前馈神经网络相同，卷积层的反向传播是一个与前向传播类似的交叉相关计算

首先在网页中提取 `user-agent` 和 `cookie` 信息作为请求头，提取 `URL` 作为请求资源的地址，循环访问图片资源，在 `detail_urls[]` 中保存图片的地址。

```
headers = {
    "User-Agent": "Mozilla/5.0 (Linux; Android 6.0; Nexus 5 Build/MRA58N) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/80.0.3987.106 Mobile Safari/537.36",
    "Cookie": "BAIDUID=229A18B7534A5CEA671381D45FCDC530:FG=1; BIDUPSID=229A18B7534A5CEA671381D45FCDC530; PSTM=1592693385; BDRCSVFR[dG2JNJb_ajR]=mk3SLVN4HKm; userFrom=null; BDRCSVFR[-pGxjrCMryR]=mk3SLVN4HKm; H_WISE_SIDS=149389_148867_148211_149537_146732_138426_150175_147527_145599_148186_147715_149253_150045_149280_145607_148660_146055_110085; delPer=0; BDORZ=AE84CDB3A529C0F8A2B9DCDD1D18B695; ysm=10315; IMG_WH=626_611; __bsi=8556698095607456048_00_14_R_R_17_0303_c02f_Y",
}

detail_urls = [] # 存储图片地址
```

对于每一个在 `detail_urls` 中的 `page`，根据 `page` 中的 `url` 地址去访问图片资源，通过 `get()` 方式根据前文的 `headers` 和 `url` 请求资源，并通过 `write()` 下载到本地，这里为了方便，仅仅下载了 `.jpg` 格式的图片。

```
for i in range(1, 400, 20): # 20 页一张
    url =
    'http://image.baidu.com/search/flip?tn=baiduimage&ipn=r&ct=201326592&cl=2&lm=&st=-1&fm=result&fr=&sf=1&fmq=1592804203005_R&pv=&ic=&nc=1&z=&hd=&latest=&copyright=&se=1&showtab=0&fb=0&width=&height=&face=0&istype=2&ie=utf-8&ctd=1592804203008%5E00_1328X727&sid=&word=瓢虫&pn={}'.format(
        i) # 请求的地址
    response = requests.get(url, headers, timeout=(3, 7)) # 设置请求超时时
    间 3-7 秒
    content = response.content.decode('utf-8') # 使用 utf-8 进行解码
    detail_url = re.findall('"objURL": "(.*?)"', content, re.DOTALL) #
    re.DOTALL 忽略格式 # 匹配 objURL 的内容, 大部分为 objURL 或 URL
    detail_urls.append(detail_url) # 将获取到的图片地址保存在之前定义的列表中
    response = requests.get(url, headers=headers) # 请求网站
    content = response.content
b = 0 # 图片第几张
for page in detail_urls:
    for url in page:
        try:
            print('获取到{}张图片'.format(i))
            response = requests.get(url, headers=headers)
            content = response.content
            if url[-3:] == '.jpg':
                with open('./train_insects/瓢虫/{}.jpg'.format(b), 'wb') as
f:
                    f.write(content)
            else:
                continue
```

```
except:
    print('超时')
    b += 1
```

3.2 数据预处理

单面打印。

首先进行数据清洗，把不符合规范的图片清理掉，比如文件格式损坏的图片，和训练集不相关的图片。之后使用 `pytorch` 中的 `transform` 模块进行预处理，将图像缩放、部分垂直翻转、随机裁剪等，目的是为了增强图像。

```
transform = transforms.Compose([
    transforms.Resize(100),
    transforms.RandomVerticalFlip(),
    transforms.RandomCrop(50),
    transforms.RandomResizedCrop(150),
    transforms.ColorJitter(brightness=0.5, contrast=0.5, hue=0.5),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])
```

通过 `datasets.ImageFolder` 创建一个图像数据集对象，用来加载训练集和测试集。这个对象将会从指定的文件夹（`D:\pythonProject\train`）中加载图像，并且在加载的时候会应用之前定义的 `transform` 转换操作。

具体来说，`ImageFolder` 会假设指定的文件夹下有多个子文件夹，每个子文件夹的名字代表一个类别，子文件夹下的所有图像都属于该类别。所以在 `train` 和 `test` 文件夹下各有一个 `lion` 文件夹和 `dog` 文件夹，这个顺序要严格保证，在爬虫的时候也要注意修改文件夹路径，防止错误的图片混入。训练模型的时候，每一次都要把集合中的数据打乱，以保证训练过程中的随机性和测试过程中的泛化。

3.3 开源仓库拉取

通过 `datasets.ImageFolder` 创建一个图像数据集对象，用来加载训练集和测试集。这个对象将会从指定的文件夹（`D:\pythonProject\train`）中加载图像，并且在加载的时候会应用之前定义的 `transform` 转换操作。

首先配置本地 `git` 信息

```
$git config --global user.name ""
```

```
$git config --global user.email ""
```

执行 `copy` 命令

```
$git copy -v "https://github.com/ffffff666/Deep_Learning-ComputerVision.git"
```

把仓库克隆下来后，最好再上传到自己的仓库里

3.4 卷积神经网络构建

定义了一个名为 `ConvNet` 的卷积神经网络类的初始化函数。定义了一个二维卷积层，输入通道数为 3，输出通道数为 32，卷积核大小为 3x3。定义了一个二维最大池化层，池化核大小为 2x2。

第二个二维卷积层，输入通道数为 32，输出通道数为 64，卷积核大小为 3x3。第二个二维最大池化层，池化核大小为 2x2。第三和第四个二维卷积层，输入通道数为 64，输出通道数为 64，卷积核大小为 3x3。第三个二维最大池化层，池化核大小为 2x2。第五和第六个二维卷积层，输入通道数分别为 64 和 128，输出通道数都为 128，卷积核大小为 3x3。第四个二维最大池化层，池化核大小为 2x2。

全连接层，输入节点数为 4608，输出节点数为 512。第二个全连接层，输入节点数为 512，输出节点数为 1。具体实现如下所示，将卷积网络初始化。

```
class ConvNet(nn.Module):
    """GiveMeANamw"""
    def __init__(self):
        super(ConvNet, self).__init__()
        # 卷积层conv, 池化层pool
        self.conv1 = nn.Conv2d( in_channels: 1, out_channels: 32, kernel_size: 3)
        self.max_pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d( in_channels: 32, out_channels: 64, kernel_size: 3)
        self.max_pool2 = nn.MaxPool2d(2)
        self.conv3 = nn.Conv2d( in_channels: 64, out_channels: 64, kernel_size: 3)
        self.conv4 = nn.Conv2d( in_channels: 64, out_channels: 64, kernel_size: 3)
        self.max_pool3 = nn.MaxPool2d(2)
        self.conv5 = nn.Conv2d( in_channels: 64, out_channels: 128, kernel_size: 3)
        self.conv6 = nn.Conv2d( in_channels: 128, out_channels: 128, kernel_size: 3)
        self.max_pool4 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear( in_features: 4608, out_features: 512)
        self.fc2 = nn.Linear( in_features: 512, out_features: 1)
```

接下来定义 `ConvNet` 类的前向传播函数。首先获取输入 `x` 的批次大小。将输入 `x` 传递给第一个卷积层。对卷积层的输出应用 `ReLU` 激活函数。对激活函数的输出应用第一个最大池化层。`x = self.conv2(x)`到 `x = self.max_pool4(x)`: 这些行重复了卷积、`ReLU` 激活和最大池化的过程，构成了网络的主体部分。

`x = x.view(in_size, -1)`将 3D 的张量（通道、高度、宽度）展平为 1D 的张量，以便可以被全连接层处理（`in_size` 是批次大小，-1 表示自动计算剩余的维度）。`x = self.fc1(x)`将展平的张量传递给第一个全连接层。`x = F.relu(x)`: 对全连接层的输出应用 `ReLU` 激活函数。`x = self.fc2(x)`: 将激活函数的输出传递给第二个全连接层。`x = torch.sigmoid(x)`: 对全连接层的输出应用 `Sigmoid` 激活函数，将输出压缩到 0 和 1 之间。具体实现如下所示：


```

def forward(self, x):
    in_size = x.size(0)
    x = self.conv1(x)
    x = F.relu(x)
    x = self.max_pool1(x)
    x = self.conv2(x)
    x = F.relu(x)
    x = self.max_pool2(x)
    x = self.conv3(x)
    x = F.relu(x)
    x = self.conv4(x)
    x = F.relu(x)
    x = self.max_pool3(x)
    x = self.conv5(x)
    x = F.relu(x)
    x = self.conv6(x)
    x = F.relu(x)
    x = self.max_pool4(x)
    # 展开
    x = x.view(in_size, -1)
    x = self.fc1(x)
    x = F.relu(x)
    x = self.fc2(x)
    x = torch.sigmoid(x)
    return x

```

3.5 学习过程

这段代码定义了一个函数 `adjust_learning_rate`，它用于在训练过程中调整学习率。`adjust_learning_rate` 接受两个参数：优化器 `optimizer` 和当前的训练轮数 `epoch`。我们选择使用 `adm` 优化器，初始学习率为 0.0001。

`modellrnew = modellr * (0.1 ** (epoch // 5))`：计算新的学习率。这行代码将初始学习率 `modellr` 乘以 0.1 的 `epoch // 5` 次方。这意味着每 5 轮训练，学习率就会衰减为原来的 10%。

`for param_group in optimizer.param_groups:` 遍历优化器中的所有参数组。将每个参数组的学习率设置为新的学习率。这个函数可以在训练过程中动态地调整学习率，有助于提高模型的训练效果。使用 `adam` 优化器进行学习。

```

def adjust_learning_rate(optimizer, epoch):
    """Sets the learning rate to the initial LR decayed by 10 every 30 epochs"""
    modellrnew = modellr * (0.1 ** (epoch // 5))
    print("lr:", modellrnew)
    for param_group in optimizer.param_groups:
        param_group['lr'] = modellrnew

```

3.6 训练和测试过程

这段代码定义了一个名为 `train` 的函数，用于训练模型。先将模型设置为训练模式。然后遍历训练数据加载器，获取每个批次的数据和目标标签。将数据和目标标签移动到指定的设备（例如 GPU），并将目标标签转换为浮点数并增加一个维度。在进行新的优化步骤之前，清零所有参数的梯度。将数据传递给模型，获取模型的输出。计算模型输出和目标标签之间的二元交叉熵损失。计算损失相对于模型参数的梯度。根据计算出的梯度更新模型的参数。如果当前批次是整数倍，则打印训练进度和损失。打印当前的训练轮数、已处理的数据量、总数据量、训练进度和当前的损失值。具体实现过程如下所示：

```
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):

        data, target = data.to(device),
        target.to(device).float().unsqueeze(1)

        optimizer.zero_grad()

        output = model(data)
        # print(output)

        loss = F.binary_cross_entropy(output, target)

        loss.backward()

        optimizer.step()

        if (batch_idx + 1) % 1 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss:
{:.6f}'.format(
                epoch, (batch_idx + 1) * len(data),
                len(train_loader.dataset),
                100. * (batch_idx + 1) / len(train_loader),
                loss.item()))
```

下面定义了测试函数，用于测试模型的性能。将模型设置为评估模式。初始化测试损失和正确预测的数量。遍历测试数据加载器，获取每个批次的数据和目标标签。将数据和目标标签移动到指定的设备（例如 GPU），并将目标标签转换为浮点数并增加一个维度。将数据传递给模型，获取模型的输出。计算模型输出和目标标签之间的二元交叉熵损失，并累加到 `test_loss`。根据模型的输出计算预测值，如果输出大于或等于 0.5，则

预测为 1，否则预测为 0。计算预测正确的数量。打印测试集的平均损失和准确率。具体实现如下所示：

```
def test(model, device, test_loader):
    model.eval()

    test_loss = 0

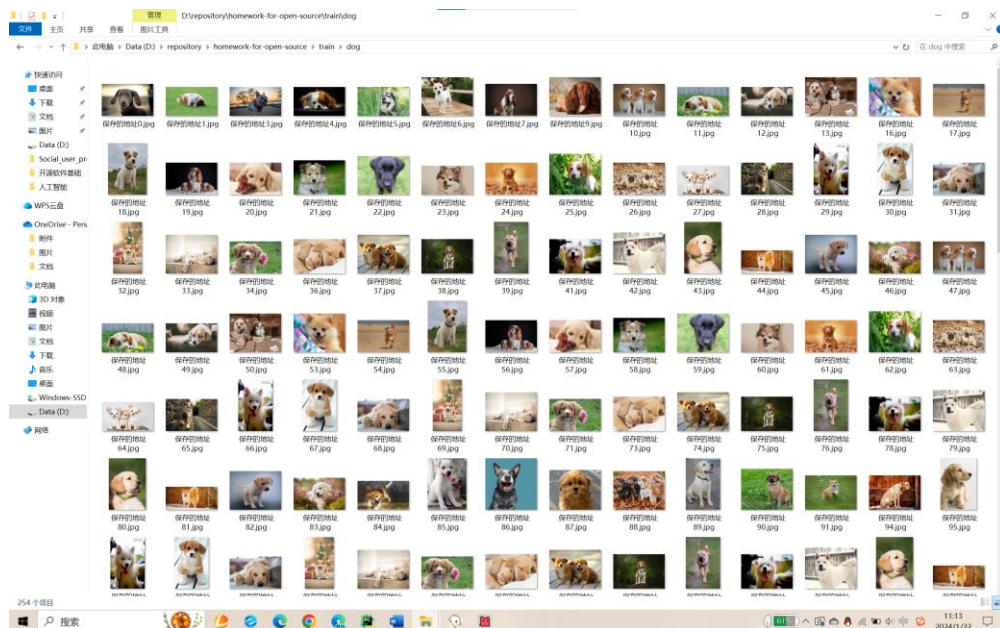
    correct = 0

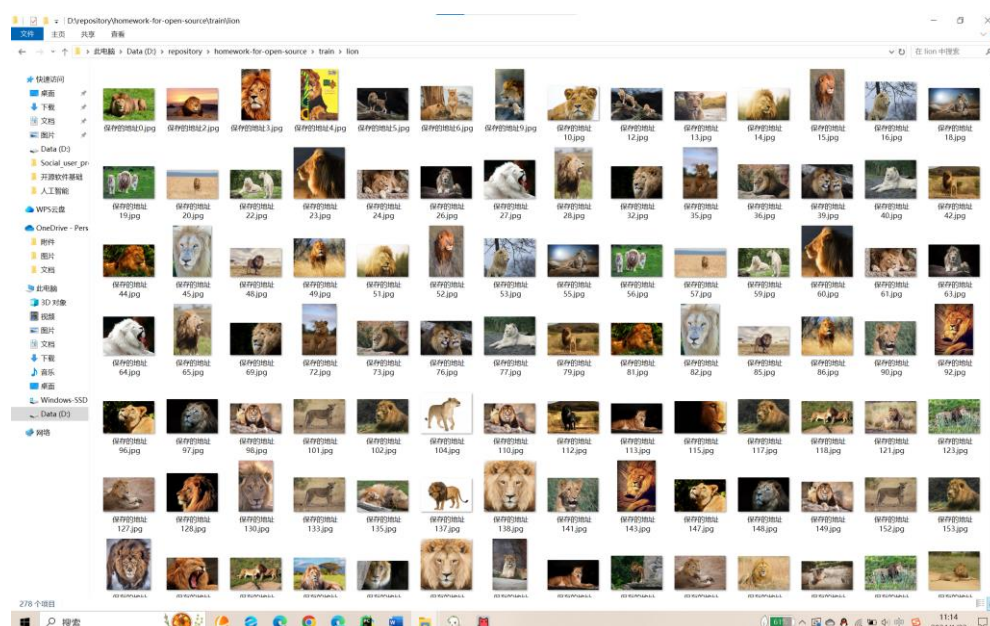
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device),
            target.to(device).float().unsqueeze(1)
            # print(target)
            output = model(data)
            # print(output)
            test_loss += F.binary_cross_entropy(output, target,
            reduction='mean').item()
            pred = torch.tensor([[1] if num[0] >= 0.5 else [0] for num in
            output]).to(device)
            correct += pred.eq(target.long()).sum().item()

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

4 运行结果展示

4.1 爬虫获取到的数据集





4.2 训练结果（Adam 优化器）

```

model_insects0.pth
model_insects_change.pth
use.py
> External Libraries
> Scratches and Consoles

106
107 optimizer = optim.Adam(model.parameters(), lr=model_lr)
108 #调整学习率
109 def adjust_learning_rate(optimizer, epoch):
110     """Sets the learning rate to the initial LR decayed by 10 ev
111     model_lrnew = model_lr * (0.1 ** (epoch // 5))
112     print("lr:", model_lrnew)
113     for param_group in optimizer.param_groups:
114         param_group['lr'] = model_lrnew
115

Run getimg x CNN x
Train Epoch: 10 [400/532 (74%)] Loss: 0.631437
Train Epoch: 10 [420/532 (78%)] Loss: 0.608351
Train Epoch: 10 [440/532 (81%)] Loss: 0.685680
Train Epoch: 10 [460/532 (85%)] Loss: 0.669232
Train Epoch: 10 [480/532 (89%)] Loss: 0.601471
Train Epoch: 10 [500/532 (93%)] Loss: 0.650591
Train Epoch: 10 [520/532 (96%)] Loss: 0.705109
Train Epoch: 10 [324/532 (100%)] Loss: 0.646256

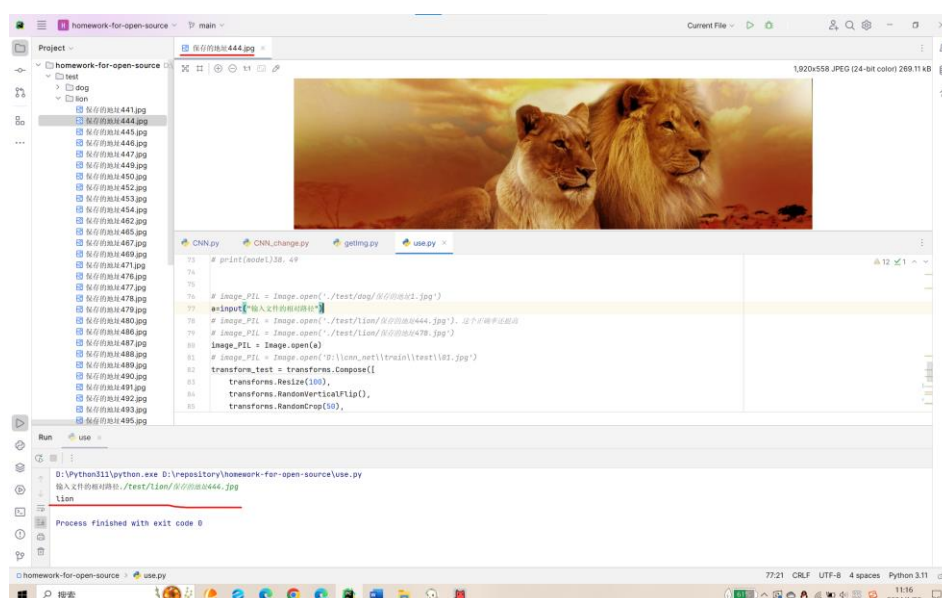
Test set: Average loss: 2.7161, Accuracy: 46/79 (58%)
    
```

4.3 卷积神经网络模型结构

```
ConvNet(  
    (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))  
    (max_pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))  
    (max_pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))  
    (max_pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    (conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))  
    (max_pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    (conv5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))  
    (conv6): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1))  
    (max_pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)  
    (fc1): Linear(in_features=4608, out_features=512, bias=True)  
    (fc2): Linear(in_features=512, out_features=1, bias=True)  
)
```

4.4 识别过程展示

在终端中输入准备分类的图片的相对地址，输出相应的类别，如下图所示。



5 对结果优化的几种尝试

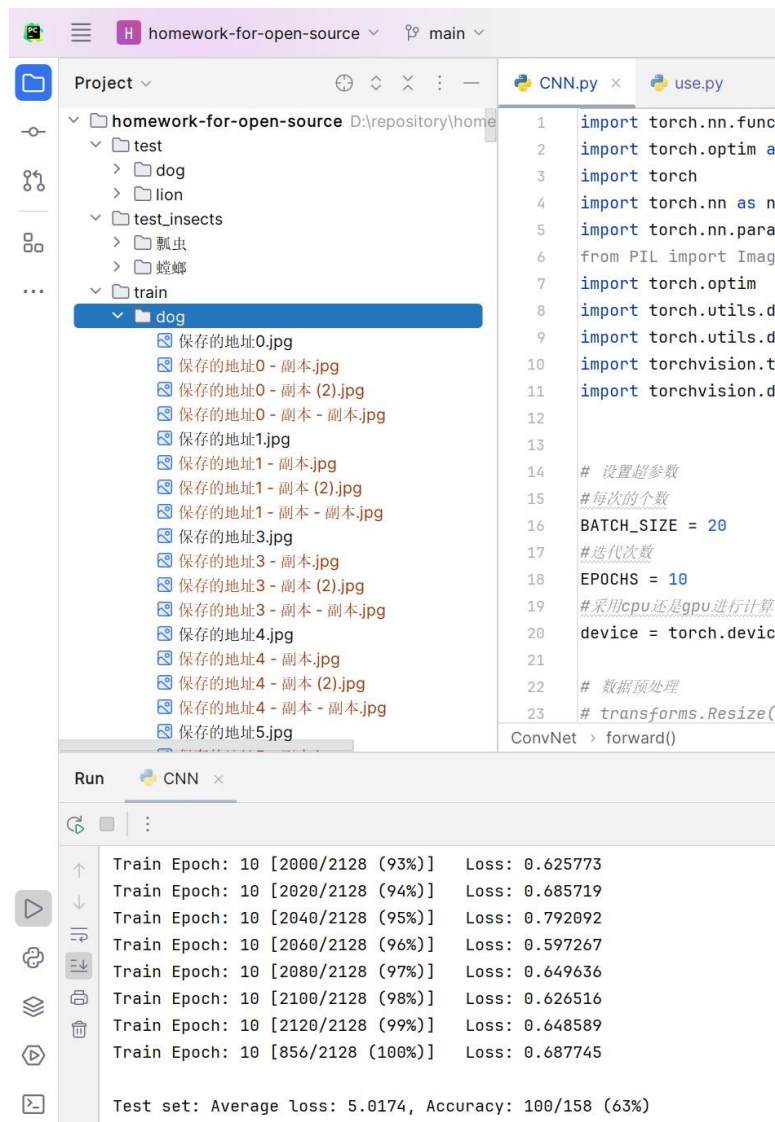
从模型训练结果可以看出，准确率为 58%，训练的结果并不是很理想，考虑下面几种优化方式。

卷积神经网络（CNN）的优化方法主要包括以下几个方向：数据处理：对数据进行清洗和增强，以提升数据质量。例如，可以通过改变光线调整、色彩填充、图片翻转等方式来增强数据。卷积方式设计：例如，使用深度分离的卷积（如 MobileNets）、二进

制卷积（如 XNOR-Net）、点组卷积和通道随机化（如 ShuffleNet）等，可以在不牺牲准确率的情况下加速 CNN 的运行，并减少内存消耗。激活函数选取：通常使用 ReLU 会在开始的时立即得到一些好的结果，但如果当 ReLU 得不到好的结果的适合，可以换成 Sigmoid 函数，如果还是不行则调整模型其它的部分，以尝试对准确率做提升。优化器选择：可以从最简单的 SGD 优化器开始，如果得到还不错的结果则对模型进行其他超参数的调优。也可以使用最为高效的 Adam 开始，切记学习率不要设置太高。

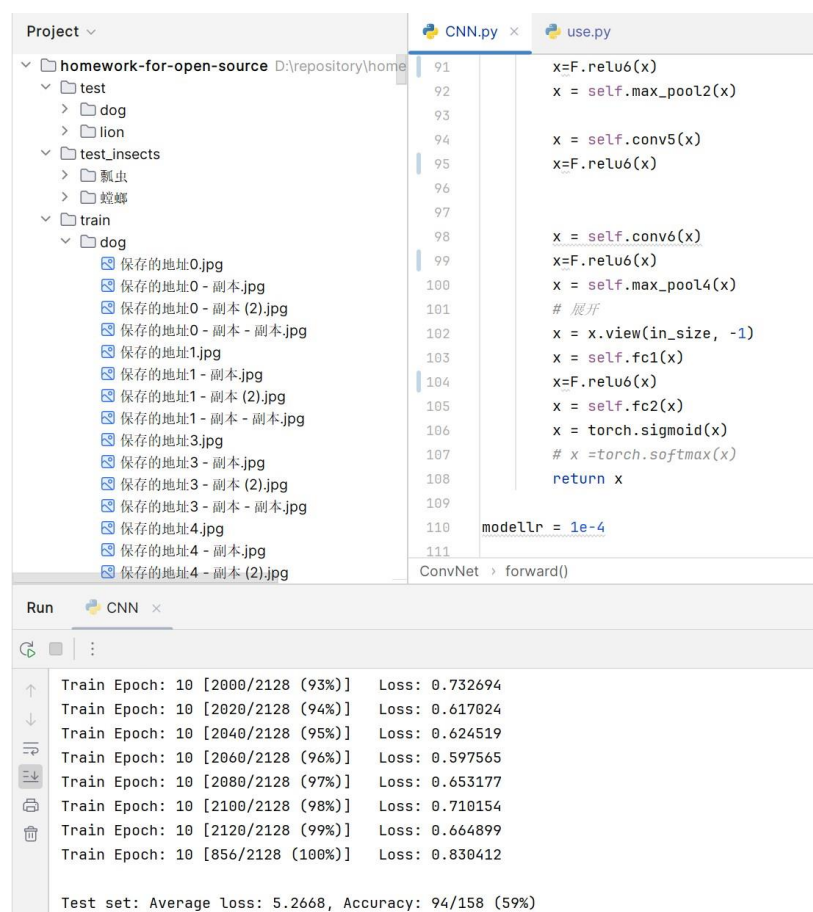
5.1 数据集优化

原始数据集训练集总共 532 张图片，测试集 125 张图片，比例近似 4:1，现在尝试将训练集扩大一倍，测试集同样扩大一倍，再次训练模型可以看到测试准确率为 63%，训练效果有所提升。



5.2 激活函数的选取

原始前向传播算法中，使用未扩展的数据集，使用 `relu` 作为激活函数，训练准确率为 58%，将 `relu` 更换为 `leaky relu` 后，训练准确率下降为 53%，更换为 `relu6` 后训练准确率为 59%。在更换激活函数这一方向上已经不会再有更多的进步。



The screenshot displays an IDE environment. On the left, a project tree shows a directory named 'homework-for-open-source' with subdirectories 'test' and 'train'. The 'train' directory contains a 'dog' subdirectory with multiple image files. The main editor shows two files: 'CNN.py' and 'use.py'. The 'use.py' file contains the following code:

```
91 x=F.relu6(x)
92 x = self.max_pool2(x)
93
94 x = self.conv5(x)
95 x=F.relu6(x)
96
97
98 x = self.conv6(x)
99 x=F.relu6(x)
100 x = self.max_pool4(x)
101 # 展开
102 x = x.view(in_size, -1)
103 x = self.fc1(x)
104 x=F.relu6(x)
105 x = self.fc2(x)
106 x = torch.sigmoid(x)
107 # x =torch.softmax(x)
108 return x
109
110 model.lr = 1e-4
111
```

Below the code editor, a 'Run' console window shows the output of the training process:

```
Train Epoch: 10 [2000/2128 (93%)] Loss: 0.732694
Train Epoch: 10 [2020/2128 (94%)] Loss: 0.617024
Train Epoch: 10 [2040/2128 (95%)] Loss: 0.624519
Train Epoch: 10 [2060/2128 (96%)] Loss: 0.597565
Train Epoch: 10 [2080/2128 (97%)] Loss: 0.653177
Train Epoch: 10 [2100/2128 (98%)] Loss: 0.710154
Train Epoch: 10 [2120/2128 (99%)] Loss: 0.664899
Train Epoch: 10 [856/2128 (100%)] Loss: 0.830412

Test set: Average loss: 5.2668, Accuracy: 94/158 (59%)
```

5.3 优化器的选取

保持原始数据集不变，激活函数不变，使用 `adam` 优化器的测试准确率为 58%，使用 `adamax` 的测试准确率为 62%，使用 `adamw` 的准确率为 63%，可以考虑把 `adamw` 优化器作为新的优化器。

大连理工大学本科毕业设计（论文）题目

```
model_insects0.pth
model_insects_change.pth
use.py
> External Libraries
> Scratches and Consoles
```

```
106
107 optimizer = optim.Adam(model.parameters(), lr=modellr)
108 #调整学习率
109 1 usage  GiveMeANamw
110 def adjust_learning_rate(optimizer, epoch):
111     """Sets the learning rate to the initial LR decayed by 10 ev
112     modellrnew = modellr * (0.1 ** (epoch // 5))
113     print("lr:", modellrnew)
114     for param_group in optimizer.param_groups:
115         param_group['lr'] = modellrnew
```

Run getimg CNN

```
Train Epoch: 10 [400/532 (74%)] Loss: 0.631437
Train Epoch: 10 [420/532 (78%)] Loss: 0.608351
Train Epoch: 10 [440/532 (81%)] Loss: 0.685680
Train Epoch: 10 [460/532 (85%)] Loss: 0.669232
Train Epoch: 10 [480/532 (89%)] Loss: 0.601471
Train Epoch: 10 [500/532 (93%)] Loss: 0.650591
Train Epoch: 10 [520/532 (96%)] Loss: 0.705109
Train Epoch: 10 [324/532 (100%)] Loss: 0.646256

Test set: Average loss: 2.7161, Accuracy: 46/79 (58%)
```

(adam)

```
model_insects0.pth
model_insects_change.pth
use.py
> External Libraries
> Scratches and Consoles
```

```
107 optimizer = optim.Adamax(model.parameters(), lr=modellr)
108 #调整学习率
109 1 usage  GiveMeANamw
110 def adjust_learning_rate(optimizer, epoch):
111     """Sets the learning rate to the initial LR decayed by 10 ev
112     modellrnew = modellr * (0.1 ** (epoch // 5))
113     print("lr:", modellrnew)
114     for param_group in optimizer.param_groups:
```

Run getimg CNN

```
Train Epoch: 10 [400/532 (74%)] Loss: 0.725361
Train Epoch: 10 [420/532 (78%)] Loss: 0.686047
Train Epoch: 10 [440/532 (81%)] Loss: 0.671771
Train Epoch: 10 [460/532 (85%)] Loss: 0.689721
Train Epoch: 10 [480/532 (89%)] Loss: 0.619814
Train Epoch: 10 [500/532 (93%)] Loss: 0.707404
Train Epoch: 10 [520/532 (96%)] Loss: 0.670007
Train Epoch: 10 [324/532 (100%)] Loss: 0.651935

Test set: Average loss: 2.6933, Accuracy: 49/79 (62%)
```

(adamax)

```
> External Libraries
> Scratches and Consoles
```

```
106
107 optimizer = optim.AdamW(model.parameters(), lr=modellr)
108 #调整学习率
109 1 usage  GiveMeANamw
110 def adjust_learning_rate(optimizer, epoch):
111     """Sets the learning rate to the initial LR decayed by 10 ev
112     modellrnew = modellr * (0.1 ** (epoch // 5))
113     print("lr:", modellrnew)
114     test()
```

Run getimg CNN

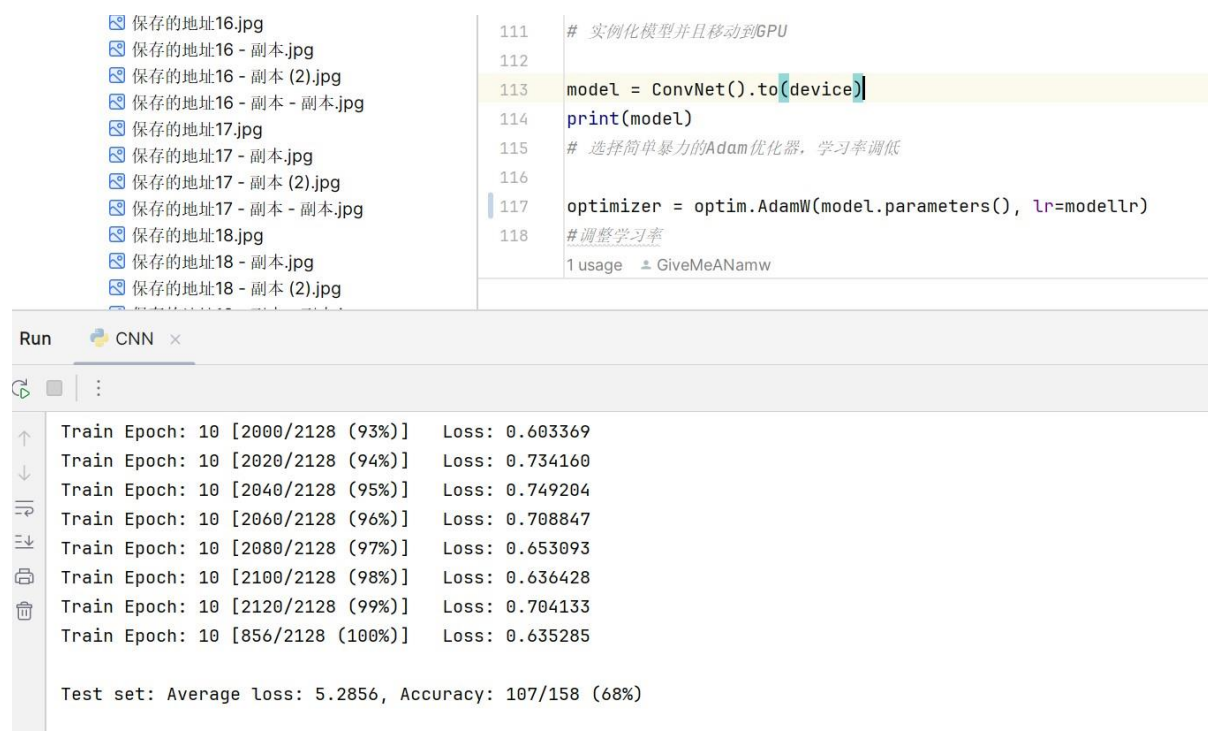
```
Train Epoch: 10 [400/532 (74%)] Loss: 0.599394
Train Epoch: 10 [420/532 (78%)] Loss: 0.643048
Train Epoch: 10 [440/532 (81%)] Loss: 0.723922
Train Epoch: 10 [460/532 (85%)] Loss: 0.684112
Train Epoch: 10 [480/532 (89%)] Loss: 0.630933
Train Epoch: 10 [500/532 (93%)] Loss: 0.613036
Train Epoch: 10 [520/532 (96%)] Loss: 0.712060
Train Epoch: 10 [324/532 (100%)] Loss: 0.682826

Test set: Average loss: 2.5919, Accuracy: 50/79 (63%)
```

(adamw)

5.4 目前最优方案的组合

现在尝试将训练集扩大 4 倍，测试集同样扩大一倍，使用 relu 作为激活函数，使用 adamw 优化器进行学习。可以看到测试准确率提升至 68%，有了较为显著的提升。



The screenshot displays a code editor with Python code for a CNN model and its training output. The code is as follows:

```
111 # 实例化模型并且移动到GPU
112
113 model = ConvNet().to(device)
114 print(model)
115 # 选择简单暴力的Adam优化器，学习率调低
116
117 optimizer = optim.AdamW(model.parameters(), lr=model_lr)
118 # 调整学习率
119 usage GiveMeANamw
```

The output shows the training progress for 10 epochs, with the loss decreasing from 0.603369 to 0.635285. The test set accuracy is 68%.

Train Epoch	Train Loss	Test Accuracy
10 [2000/2128 (93%)]	0.603369	
10 [2020/2128 (94%)]	0.734160	
10 [2040/2128 (95%)]	0.749204	
10 [2060/2128 (96%)]	0.708847	
10 [2080/2128 (97%)]	0.653093	
10 [2100/2128 (98%)]	0.636428	
10 [2120/2128 (99%)]	0.704133	
10 [856/2128 (100%)]	0.635285	

Test set: Average loss: 5.2856, Accuracy: 107/158 (68%)

6 总结

在开源软件基础这门课中，我学会了爬虫和 git 的使用方法，github 是一个很好的代码托管平台，可以让我寻找到自己感兴趣的项目示例，也可以帮助我管理存储在本地的代码，而不是凌乱的放在“新建文件夹”中。爬虫可以帮助我更方便的获取网络资源，也让我对网站的访问有了一个新的认识。美中不足的是，我对于卷积神经网络这一主题理解的还不是很透彻，只是单纯模仿着开源项目进行搭建，在优化处理部分也只是做了有关统计学习的简单修改，没有对卷积网络模型做过多的修改，这也可能是准确率不高的原因之一。对于爬虫的认知也不是很透彻，只能进行简单的爬取信息，无法做某些复杂的数据分析。