实验 2-卷积神经网络实现

学号：1190201215
姓名：冯开来

一、 实验目的：

基于 Pytorch 实现 AlexNet 结构，在 Caltech101 数据集上进行验证，使用 tensorboard 进行训练数据可视化。

二、 实验环境：

（在文件 requirements.txt 已经体现）
Python=3.8; GPU: NVIDIA GeForce MX450; 图片输入 resize 后为 109×109

```
28  requests-oauthlib==1.3.1
29  rsa==4.8
30  six @ file:///tmp/build/80754af9/six_1644875935023/work
31  tensorboard==2.9.0
32  tensorboard-data-server==0.6.1
33  tensorboard-plugin-wit==1.8.1
34  tensorboardX==2.5
35  torch==1.11.0
36  torchaudio==0.11.0
37  torchvision==0.12.0
38  typing_extensions @ file:///opt/conda/conda-bld/typing_extensions_1647553014482/work
39  urllib3 @ file:///C:/ci/urllib3_1650640043075/work
40  Werkzeug==2.1.2
41  win-inet-pton @ file:///C:/ci/win_inet_pton_1605306167264/work
```

(requirements.txt 部分截图)

三、 实验内容：

1. 超参数定义

```
parser = argparse.ArgumentParser("AlexNet")
parser.add_argument("--device", default="cuda")
parser.add_argument("--batch-size", default=30)
parser.add_argument("--epochs", default=10, type=int)
parser.add_argument("--seed", default=2)
parser.add_argument("--learning-rate", default=0.001)
parser.add_argument("--output-path", default="./model")  # ./model
parser.add_argument("--data-path", default="./caltech-101/")
parser.add_argument("--img-size", default=109)
parser.add_argument("--test", action="store_true", help="Only run test")

args = parser.parse_args()
print(args)
```

这里默认由 GPU 进行加速训练。其中 seed 为构造数据集时用的随机种子；output-path 是存放模型结构，优化器等信息；data-path 是下载的数据的路径；--test 参数表示当这个 test 为真的时候，我们只验证 test 集的正确率。

2. 数据集构造

因为本次实验没有现成划分好的数据集，我们需要自己读入图片和标签，并按照 8：1：1 的比例划分 train，val，test 集进行训练、验证和测试。caltech-101/101_ObjectCategories 的子文件夹存放了各个类别的图片，所以我在构造的过程中，使用两层循环，外循环用来构造 lable 数组，值为 0,1,…100。内循环用来构造图片并以张量的形式存入 imgs 这个 list 中。

```
        img_dir = os.path.join(args.data_path, '101_ObjectCategories/')
        label = -1
        for path in os.listdir(img_dir):
            if path == 'BACKGROUND_Google':
                continue
            path_name = os.path.join(img_dir, path)
            label += 1
            for name in os.listdir(path_name):
                file_name = os.path.join(path_name, name)
                self.labels.append(label)
                img = self.transforms(Image.open(file_name).convert('RGB'))
```

当然在构造图片的时候，我们调用了 transform 用来对图片进行预处理，其中的重点就是将图片 resize 为 109×109 的大小。

```
def __init__(self, args):
    self.imgs = []
    self.labels = []
    self.transforms = transforms.Compose([
        transforms.Resize((args.img_size, args.img_size)),
        transforms.ToTensor(),
    ])
```

最后这个数据集的构造继承了 Dataset 类，并重写了 init，getitem 和 len 函数，其中 getitem 返回的是 img 和 label 的字典类型。

3. 搭建网络结构

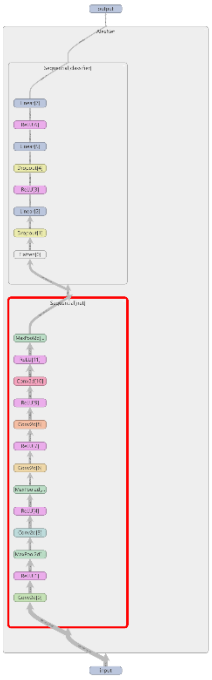
AlexNet 网络包含 5 个卷积层和 3 个全连接层（其中最后一个全连接层也就是 softmax 输出层）。因为显卡的原因，我调整了部分网络参数。其参数数量如下表所示：

层数	定义	参数数量
C1	8 个 5×5×3 卷积核，步长 2，填充 2	(5×5×3+1)×8=608
C2	16 个 3×3×8 卷积核，步长 1，填充 1	(3×3×8+1)×16=1168
C3	32 个 3×3×16 卷积核，步长 1，填充 1	(3×3×16+1)×32=4640
C4	64 个 3×3×32 卷积核，步长 1，填充 1	(3×3×32+1)×64=18496
C5	128 个 3×3×64 卷积核，步长 1，填充 1	(9×64+1)×128=73856
FC6	卷积核 6×6×6×128，512 个神经元，偏置	(27648+1)×512=14156288
FC7	全连接层，256 个神经元，偏置参数	(512+1)×256=131328
Output	全连接层，101 个神经元	101×256=25856

可以发现，这里参数一共大概有608+1168+4640+18496+73856+14156288+131328+25856=14412240。大约是 1 千万个参数，这比原本的 AlexNet 的 6 千万参数小了 6 倍。网络的具体实现方法如下图所示：

```
self.net = nn.Sequential(
    # 输入 (3, 109, 109) 输出 (8, 55, 55)
    nn.Conv2d(in_channels=3, out_channels=8, kernel_size=(5, 5), stride=(2, 2), padding=2),  #
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),  # (55-3+0)/2+1=27
    # 输入 (8, 27, 27) 输出 (16, 27, 27)
    nn.Conv2d(in_channels=8, out_channels=16, kernel_size=(3, 3), stride=(1, 1), padding=1),  #
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),  # (27-3+0)/2+1=13
    # 输入 (16, 13, 13) 输出 (32, 13, 13)
    nn.Conv2d(in_channels=16, out_channels=32, kernel_size=(3, 3), stride=(1, 1), padding=1),
    nn.ReLU(),
    # 输入 (32, 13, 13) 输出 (64, 13, 13)
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=(3, 3), stride=(1, 1), padding=1),
    nn.ReLU(),
    # 输入 (64, 13, 13) 输出 (128, 6, 6)
    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=(3, 3), stride=(1, 1), padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2)  # (13-3+0)/2+1=6
)

self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Dropout(p=0.5),
    nn.Linear(128 * 6 * 6, 512),
    nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Linear(256, 101)
)
```



（网络结构图，放大应该还是能看清的…）

4. 优化器和损失函数

本次实验选用的优化器有 AdamW。不过相比上次实验，这一次使用了学习率中的 scheduler = MultiStepLR()函数，实现了在训练一定 epoch 后能够调整 lr，本次实验为在 20 和 30 个 epoch 之后减少 10 倍。

损失函数是交叉熵损失函数 CrossEntropyLoss()。

5. 训练过程、测试过程和 tensorboard 可视化

训练过程的流程大致是放入 model 进行训练，然后计算 loss，将 loss 反向传播，更新网络参数，然后继续训练直到收敛。在每次进行过一轮 epoch 后，放入 val 集检验正确率。然后将正确率最高的模型，优化器，epoch 写入.pth 作为模型文件进行保存。

```
if val_accur > best_accur and args.output_path:
    best_accur = val_accur
    output_dir = Path(args.output_path) / "AlexNet.pth"
    state = {
        'net': alexnet.state_dict(),
        'optimizer': optimizer.state_dict(),
        'epoch': epoch+1
    }
    torch.save(state, output_dir)
```

测试过程则是不需要 train 和 val，而是将之前保存好的最优模型 load_to_dict 并且在 test 集上跑出正确率即可。

```
    # Only run on test
    if args.test:
        print("*****Starting testing*****\n")

        test_num = 0
        test_accur = 0.0
        load_dir = Path(args.output_path) / "AlexNet.pth"
        checkpoint = torch.load(load_dir)
        alexnet.load_state_dict(checkpoint['net'])
        optimizer.load_state_dict(checkpoint['optimizer'])
        alexnet.eval()
```

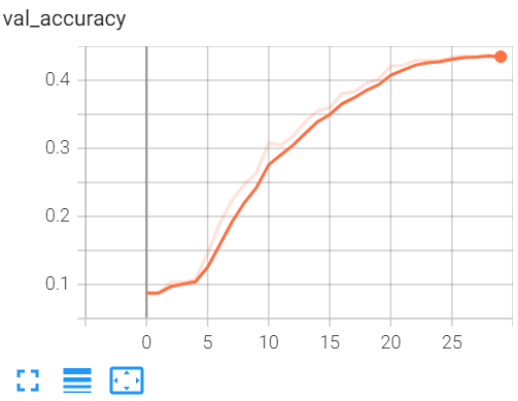
```
PS D:\python_projects\AlexNet> python main.py --test
Namespace(batch_size=30, data_path='./caltech-101/', device='cuda', epochs=30, i

*****Starting testing*****

Test Accuracy: 51.90%

*****Testing ends*****
```

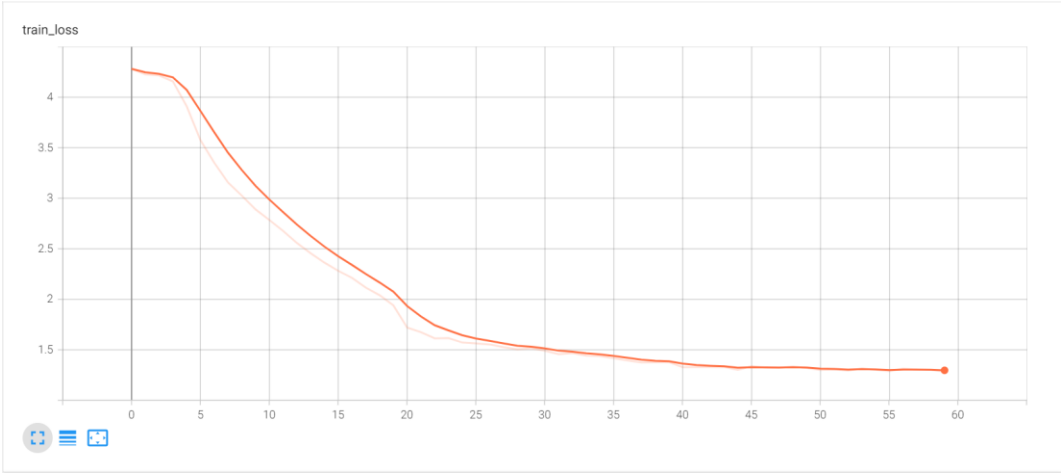
对于 tensorboard 可视化，本次实验只将网络结构（上文中出现）和 train 集的 loss 以及 val 集的正确率写入了文件。最后在终端运行 tensorboard --logdir=路径名。即可得到结果。

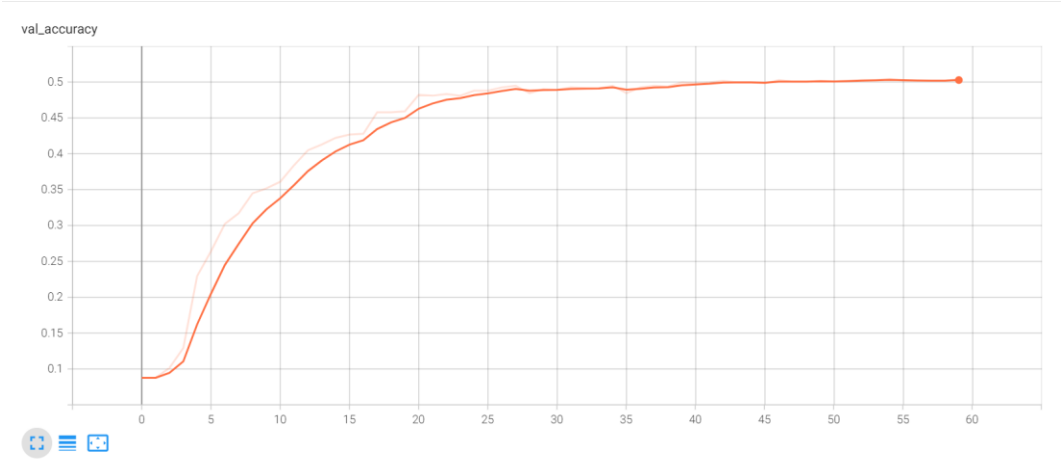


四、 实验结果与分析：

第一个实验参数作为对照组，依次改变参数的大小判断其对于结果的影响。因为次数太多，我这里就每个参数集就找了其中一次实验作为报告内容。

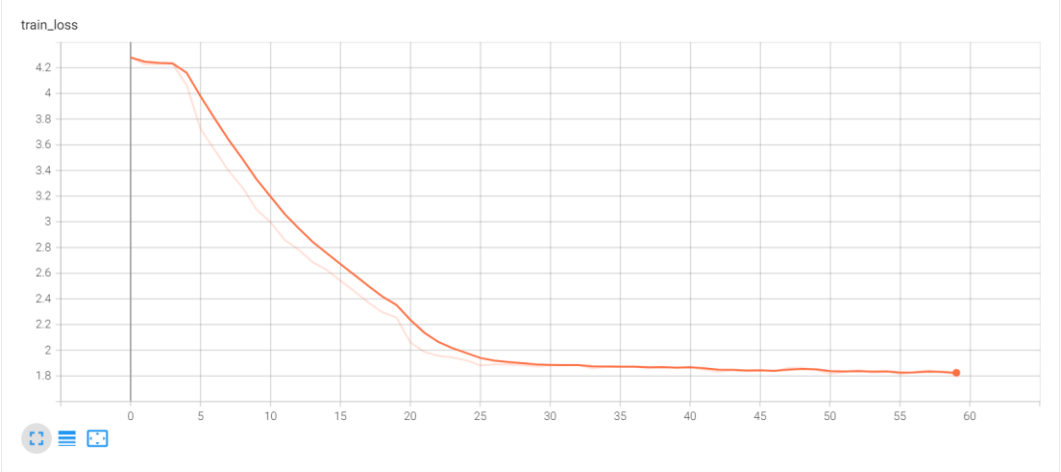
1. 参数：batch_size=30; epochs=60; lr=0，001（epoch20 和 40 之后×0.1）最优结果正确率：50.4%；损失：1.287

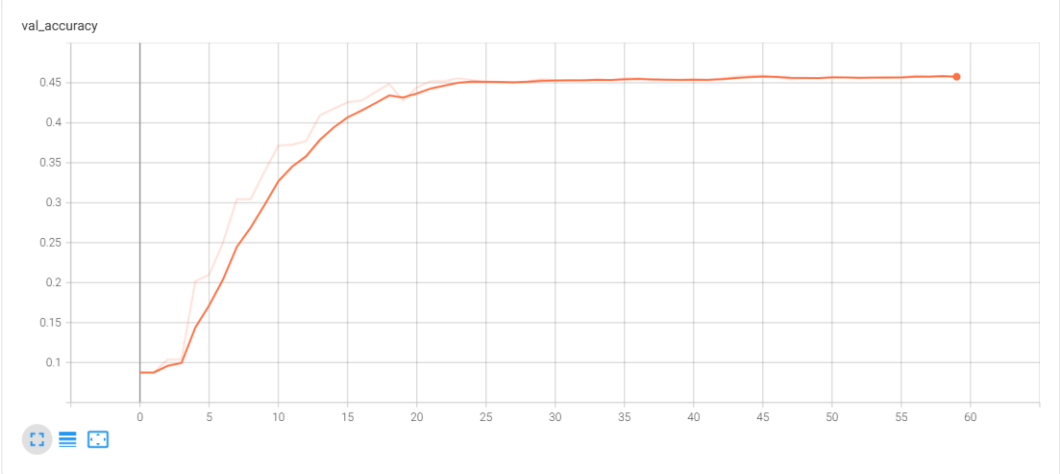




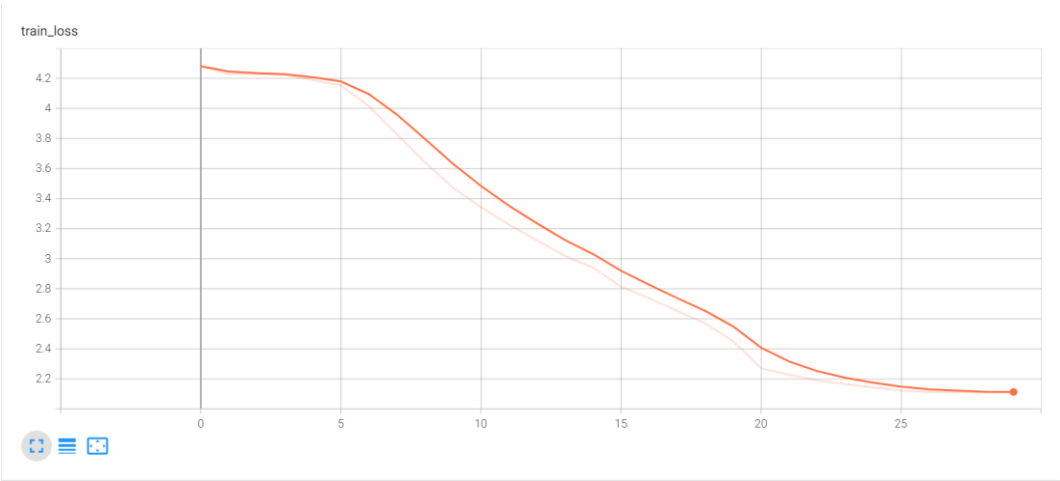
第一次参数的设置几乎是我所有实验中效果最好的，正确率最高的（可能因为 resize 和网络结构的不一样，我的正确率并没有 60，70 那么高）所以在接下来的比较中，我主要对比了 epoch 和 lr 对于效果的影响。

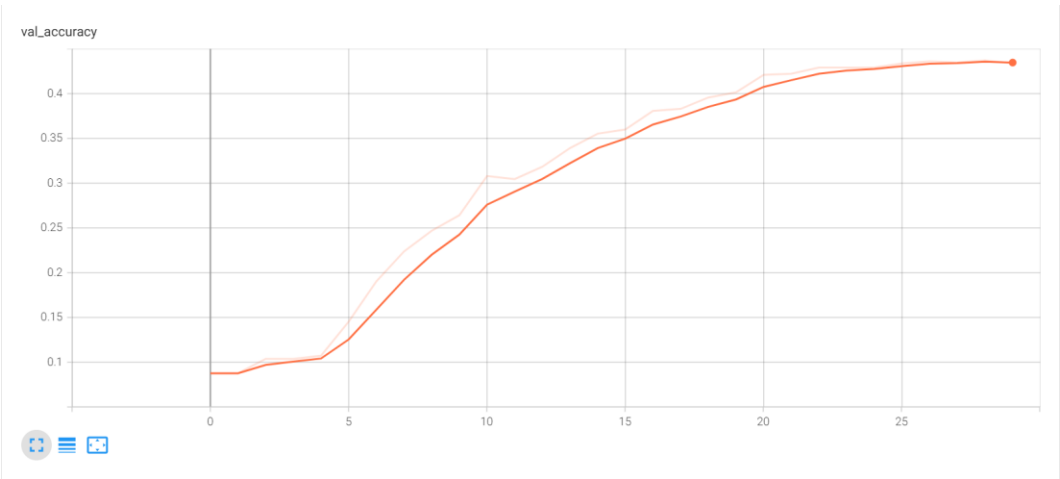
2. 参数：batch_size=30; epochs=60; lr=0.001(epoch20 和 25 之后×0.1)
最优结果正确率：45.67%；损失：1.813





3.参数：batch_size=30；epochs=30；lr=0.001（在 epoch20 后×0.1）
最优结果正确率：44.29%；损失：1.949361





纵观以上实验（当然现实不止做了这么多），可以看到： epochs 的提高可以显著提升效果；而对于学习率调整就很有意思，像第二次实验过早的设置了 lr 缩小为原来的 1/10，所以导致最后并没有收敛。而在实际实验中（报告中并没有体现），还出现了过晚的设置 lr 缩小导致先有一个较好的结果最后好几个 epoch 却一直在这个较好结果左右震荡，这显然是 lr 设置过大导致的。