哈尔滨工业大学

<<模式识别与深度学习>>

实验 2 实验报告

(2020 春季学期)

成员 1:	1170300513 陈鋆
成员 2:	
成员 3:	

-

一、实验环境

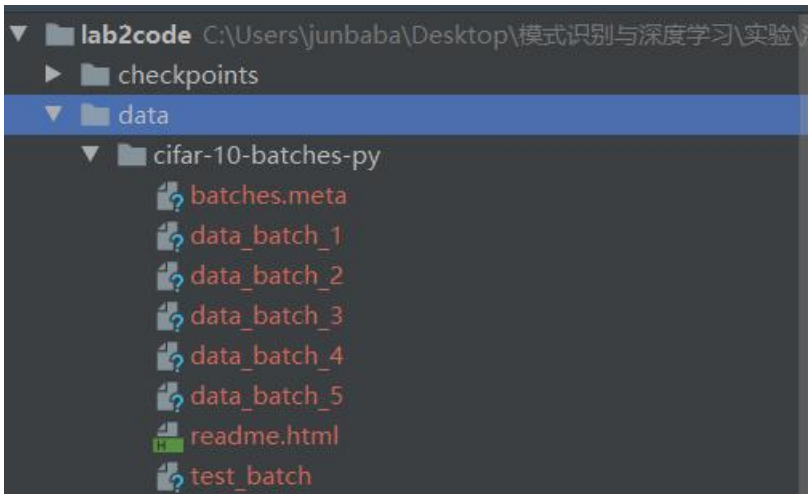
- 操作系统：win10 操作系统
- IDE：Pyrcharm Professional
- Python 运行环境：Anaconda 创建的虚拟环境 deeplearning
- 环境中必要的 Python 库：
  - Pytorch：

torch		1.3.1
torchvision		0.4.2

  - 用于可视化的 TensorboardX：

tensorboard	Tensorboard lets you watch tensors flow	2.0.0
tensorboardx		2.0
tensorflow-gpu	Metapackage for selecting a tensorflow variant.	2.1.0
tensorflow-gpu-estimator		2.1.0

- CIFAR-10 数据集存放目录：



二、实验内容

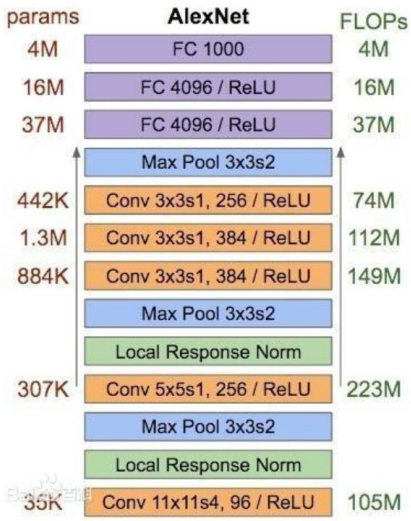
- 基于 PyTorch 实现 AlexNet 结构
- 在 Cifar-10 数据集上进行验证
- 使用 tensorboard 进行训练数据可视化

AlexNet:

2012 年，Alex 等人提出的 AlexNet 网络在 ImageNet 大赛上以远超第二名的成绩夺冠，卷积神经网络乃至深度学习重新引起了广泛的关注。

AlexNet 是在 LeNet 的基础上加深了网络的结构，学习更丰富更高维的图像特征。AlexNet 的特点：

- 更深的网络结构
- 使用层叠的卷积层，即卷积层+卷积层+池化层来提取图像的特征
- 使用 Dropout 抑制过拟合
- 使用数据增强 Data Augmentation 抑制过拟合
- 使用 Relu 替换之前的 sigmoid 的作为激活函数
- 多 GPU 训练

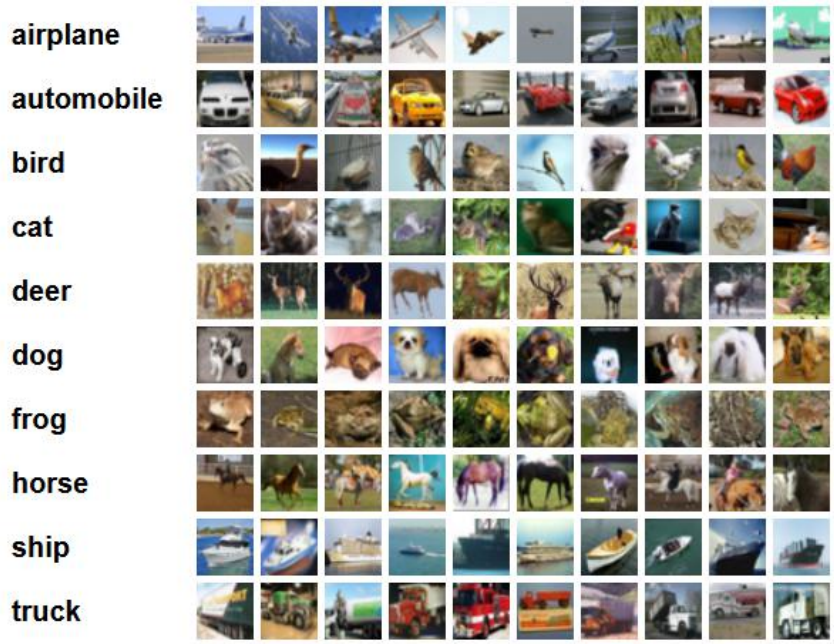


CIFAR-10：

CIFAR-10 数据集由 10 个类的 60000 个 32x32 彩色图像组成，每个类有 6000 个图像。有 5 万张训练图片和 1 万张测试图片。

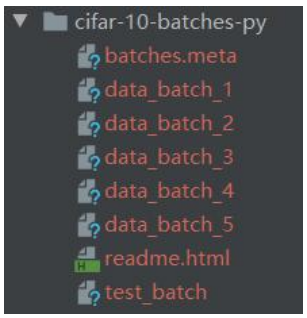
数据集分为五个训练批和一个测试批，每个测试批有 10000 个图像。测试批包含从每个类中随机选择的 1000 个图像。训练批包含随机顺序的剩余图像，但某些训练批可能包含一个类的图像多于另一个类的图像。其中，训练批次包含每个类的 5000 幅图像。

以下是数据集中的类，以及来自每个类的 10 个随机图像：



这些类是完全互斥的。汽车和卡车之间没有重叠。“汽车”包括轿车，越野车，诸如此类的东西。“卡车”只包括大卡车。也不包括皮卡。

python 版本的解压后共有 8 个文件：



Tensorboard：

Tensorboard 是 tensorflow 内置的一个可视化工具，它通过将 tensorflow 程序输出的日志文件的信息可视化使得 tensorflow 程序的理解、调试和优化更加简单高效。Tensorboard 的可视化依赖于 tensorflow 程序运行输出的日志文件，因而 tensorboard 和 tensorflow 程序在不同的进程中运行。

而在本次实验中，我们通过 TensorboardX 来代替 tensorflow 输出日志文件，从而实 Pytorch 的可视化。

三、实验代码实现

本次实验代码主要由四个结构构成：用于训练的 train.py、用于测试的 test.py、用于构造 AlexNet 模型的 Alexnet.py 以及用于构造自己的 CIFAR-10 数据集的 MyCIFAR10.py。程序执行的主体思路是：首先由 train.py 对构造的 AlexNet 模型在训练集上进行训练并保存相关的 checkpoint，再由 test.py 读取 checkpoint 并在测试集上测试手写数字识别的准确率。

Alexnet.py：

本人所构造的 Alexnet,参考了 torchvision.models.alexnet，并做出了以下改变：

1. 在 Conv2d 层中就卷积核大小、步长、填充长度进行了改变。
2. 在池化层中改变了卷积核大小，避免后续图片过小。
3. 删除了 avgpool。

```python
def __init__(self, num_classes=10):
    super(Alexnet, self).__init__()
    # 卷积层
    self.features = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),  # 缩小卷积核，步长、填充
        nn.ReLU(inplace=True),  # inplace=True，覆盖操作，节省空间
        nn.MaxPool2d(kernel_size=2),  # 32 -> 16
        nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2),  # 16 -> 8
        nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(kernel_size=2),  # 8 -> 4
    )
    # 全连接层
    self.classifier = nn.Sequential(
        nn.Dropout(),
        nn.Linear(256 * 4 * 4, 4096),
        nn.ReLU(inplace=True),
        nn.Dropout(),
        nn.Linear(4096, 4096),
        nn.ReLU(inplace=True),
        nn.Linear(4096, num_classes),
    )
```

MyCIFAR10.py：

torch.utils.data.DataLoader 不仅生成迭代数据非常方便，而且它也是经过优化的，效率十分之高（肯定比我们自己写一个要高多了），因此最好不要舍弃。因此，我的想法是根据 CIFAR-10 数据集构造一个 Dataset 的子类，使之能够作为 torch.utils.data.DataLoader 的参数，从而使数据集能被我们用于生成迭代数据进行训练与测试。

要构造 Dataset 的子类，就必须要实现两个方法：

- *_getitem_*(self, index)：根据 index 来返回数据集中标号为 index 的元素及其标签。

```python
# 继承的Dataset类需要实现两个方法之一：__getitem__(self, index)
def __getitem__(self, index):
    img, label = self.imgs[index], self.labels[index]

    if self.transform is not None:
        img = self.transform(img)

    if self.target_transform is not None:
        label = self.target_transform(label)

    return img, label
```

- *_len_*(self)：返回数据集的长度。

```
# 继承的Dataset类需要实现两个方法之一：__len__(self)
def __len__(self):
    return len(self.imgs)
```

在__init__初始化之时读取数据集，由于图像要作为数据要送入网络中，因此最后需要将其累加值从 numpy 数组转为 Tensor：

```
def __init__(self, root, train=True, transform=None, target_transform=None):
    super(MyCIFAR10, self).__init__()
    self.transform = transform
    self.target_transform = target_transform
    self.imgs = None
    self.labels = []

    # 根据CIFAR-10官网上下载的数据，训练集分为5个batch文件，每个里有10000张32*32的图片；测试集只有1个batch文件，里面有10000张32*32的图片
    train_lists = ['data_batch_1',
                   'data_batch_2',
                   'data_batch_3',
                   'data_batch_4',
                   'data_batch_5']
    test_lists = ['test_batch']
```

```
# 根据CIFAR-10官网上下载的数据，训练集分为5个batch文件，每个里有10000张32*32的图片；测试集只有1个batch文件，里面有10000张32*32的图片
train_lists = ['data_batch_1',
               'data_batch_2',
               'data_batch_3',
               'data_batch_4',
               'data_batch_5']
test_lists = ['test_batch']

# 根据train是否为True来选择测试集或训练集
if train:
    lists = train_lists
else:
    lists = test_lists

# 读取数据集，构造类中的图像集和标签
for list in lists:
    filename = os.path.join(root, list)
    with open(filename, 'rb') as f:  # 这里需要'rb' + 'latin1'才能读取
        datadict = pickle.load(f, encoding='latin1')
        X = datadict['data'].reshape(-1, 3, 32, 32)
        Y = datadict['labels']
        if self.imgs is None:
            self.imgs = np.vstack(X).reshape(-1, 3, 32, 32)
        else:
            self.imgs = np.vstack((self.imgs, X)).reshape(-1, 3, 32, 32)
        self.labels = self.labels + Y
self.imgs = torch.from_numpy(self.imgs).type(torch.FloatTensor)     # 最后需要将numpy数组转为Tensor
```

train.py：

首先，定义一个解析命令行参数的函数，使得我们能够通过命令行输入

一些训练时需要的关键常量，便于我们进行调参：

```
def get_args():
    """
    解析命令行参数
    返回参数列表
    """
    parser = OptionParser()
    parser.add_option('-e', '--epochs', dest='epochs', default=20, type='int',
                      help='number of epochs')
    parser.add_option('-b', '--batch_size', dest='batchsize', default=50,
                      type='int', help='batch size')
    parser.add_option('-l', '--lr', dest='lr', default=3e-4,
                      type='float', help='learning rate')
    (options, args) = parser.parse_args()
    return options
```

构造训练集是用上了自己定义 MyCIFRA10 类：

```
# 构造训练集
cifar10 = MyCIFAR10.MyCIFAR10('./data/cifar-10-batches-py', train=True)
train_loader = torch.utils.data.DataLoader(dataset=cifar10, batch_size=batch_size, shuffle=True)
```

剩下的训练部分都很常规，和上次实验的步骤基本相似，并且在代码中注释详尽，这里就不赘述了。主要讲一下改进的一些部分：

每一个 epoch 计算一次平均 loss，并进行可视化绘画：

```
    # 每个epoch计算一次平均Loss
    print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch + 1, epochs, trainloss / len(train_loader)))
    log.append((trainloss / len(train_loader)).item())
    # write to tensorboard
    writer.add_scalar('scalar/TrainLoss', trainloss/len(train_loader), epoch, walltime=epoch)
writer.close()
```

添加了异常，使得按下 Ctrl+C 打断训练后，能保存模型：

```
# ctrl + C 可停止训练并保存
except KeyboardInterrupt:
    print("Save.....")
    torch.save(model.state_dict(), os.path.join('./checkpoints', 'Interrupt.ckpt'))
    exit(0)
```

test.py：

关闭 dropout 开启测试模式：

```
# 关闭dropout开启测试模式
model.eval()
```

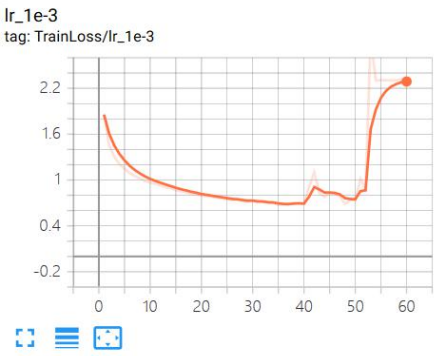
测试准确度并输出：

```
# 测试准确度
with torch.no_grad():
    correct = 0
    total = 0
    Loss = 0.0
    for images, labels in test_loader:
        # 若GPU可用，拷贝数据至GPU
        images = images.to(device)
        labels = labels.to(device)
        # 将图像输入Alexnet中并得到结果
        outputs = model(images)
        # 如果需要展示Loss，就计算并累加
        if showloss:
            loss = criterion(outputs, labels)
            Loss += loss
        # 获得概率最大的下标，即分类结果
        _, predicted = torch.max(outputs.data, 1)
        # 计算正确个数
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    # 如果需要展示Loss，则打印出Loss
    if showloss:
        print('Loss in test_loader: {:.4f}'.format(Loss / len(test_loader)))
    # 打印测试准确率
    print('Accuracy of the network on the {} test images: {} %'.format(len(images) * len(test_loader),
                                                                       100 * correct / total))
```

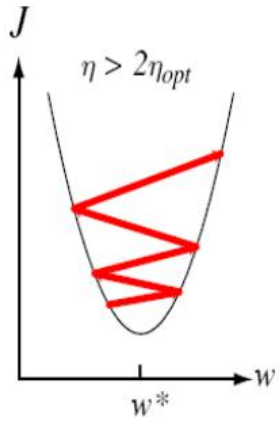
四、实验结果与分析

lr = 1e-3

首先，我将学习率 lr 定为 1e-3，训练 20 个 epoch 后，发现效果并不好，正确率仅仅只有 70%，于是我增大迭代次数 epochs，观察可视化绘制的 loss 函数：



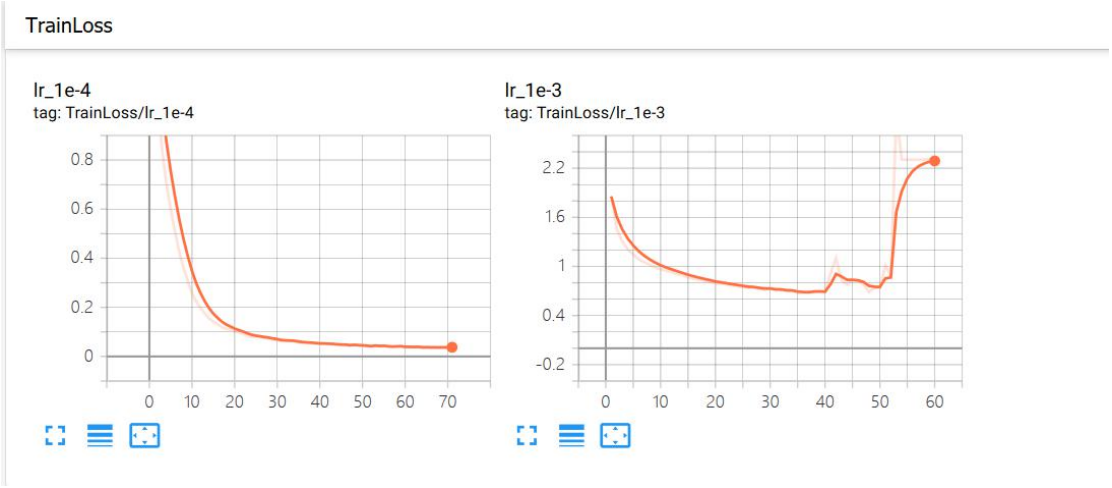
可以看到，当 epochs 增大后，loss 反而上升了。我认为这可能是学习率过大的因素，导致跳过了低点，如下图所示：



因此，我调低学习率再进行训练。

lr = 1e-4

观察可视化绘制的 loss 函数：



通过对比可以发现，lr=1e-4 确实能使得 Loss 更低，这也证实了前面学习率过大导致跳过了低点的猜想。

进行测试，准确率为 80%左右。

```
Accuracy of the network on the 10000 test images: 80.35 %
```

调整网络结构（推迟 Pooling）：

通过上面两个实验，在学习率方面继续调整以降低损失，获得更高的准确率是非常难的了。现在限制准确率主要是网络结构。

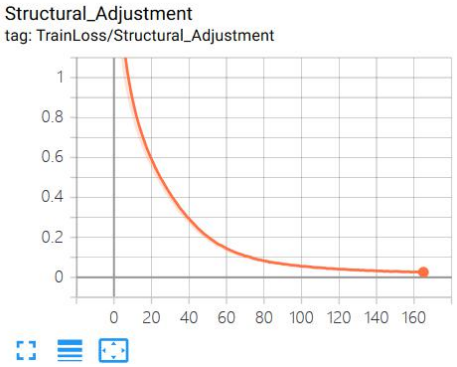
因此，我打算在网络结构上加以改进，观察其是否能继续提高性能。根据老师上课中提到的对 AlexNet 的改进思路：

1. C1: 减小卷积核尺寸
2. Pooling: 往后
3. 宽度也对性能有益

我对前文中我的 AlexNet 进行了进一步的微调，删去了第一个池化层，农民画添加了 avgpool，使得 pooling 靠后（之前已经减小过卷积核的尺寸了）：

```
# 卷积层
self.features = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),  # 缩小卷积核，步长、填充
    nn.ReLU(inplace=True),  # inplace=True，覆盖操作，节省空间
    # nn.MaxPool2d(kernel_size=2),  # 32 -> 16
    nn.Conv2d(64, 192, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=2),  # 32 -> 16
    nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
    nn.ReLU(inplace=True),
    nn.MaxPool2d(kernel_size=2),  # 16 -> 8
)
self.avgpool = nn.AdaptiveAvgPool2d((4, 4))  # 8 -> 4
# 全连接层
self.classifier = nn.Sequential(
    nn.Dropout(),
    nn.Linear(256 * 4 * 4, 4096),
    nn.ReLU(inplace=True),
    nn.Dropout(),
    nn.Linear(4096, 4096),
    nn.ReLU(inplace=True),
    nn.Linear(4096, num_classes),
)
```

损失函数曲线：

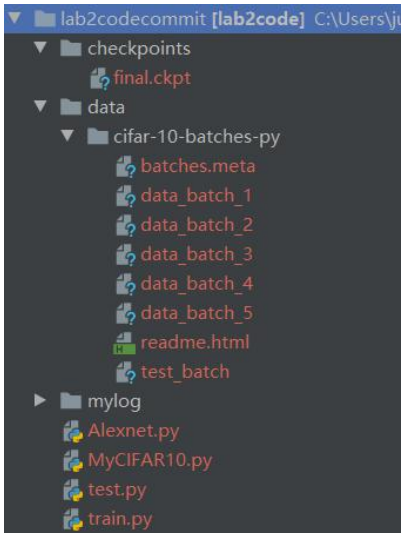


最后准确率确实有所提升，达到了 86%左右：

```
Accuracy of the network on the 10000 test images: 86.06 %
```

最后，到这一步，我认为已经到了 AlexNet 的极限，再进一步调整结构(改变层数)可能就不叫 AlexNet 了，并且时间有限，难以训练更多的结构，所以我最终的准确率为 86%左右。

最终的代码结构如下：



提交时候未提交 data 文件夹下的数据。

五、心得体会

这次实验使我收获颇多，主要有两点收获：

- 首先，因为要自己写一个 dataset 类，为了学习 torch 中对其的构建，我不得不去尝试着查看源码了解其情况。这使我戒掉了对库的了解严重依赖于搜索引擎的坏习惯，养成了 ctrl+B 查看源码实现的好习惯。

- 其次，这次实验也使我更了解了 CNN 的一些性质，并且初步掌握了构造 CNN 以及在训练中对模型进行调参的方法。