哈尔滨工业大学

<<模式识别与深度学习>>

实验 2 基于 Pytorch 的 Alexnet 实现

(2020 春季学期)

学院：计算机科学与技术

学号：1170300909

姓名：武磊

指导老师：左旺孟

日期：2020.4.26

一、实验目的

1. 使用 Pytorch 实现 Alexnet，并在 CIFAR10 数据集上验证
2. 熟悉卷积神经网络的基本原理和设计方法
3. 了解 Batch Normalization 和 Dropout 在 CNN 中的作用
4. 参考其他网络结构，尝试对 Alexnet 网络结构进行调整

二、实验环境配置

1. 硬件：

   CPU ： Inter Core i7-7500U 2.70GHz

   GPU ： NVIDIA GeForce 940MX
2. 软件：

   操作系统:WIN10

   软件：

   Python 3.6 (基于 Anaconda)

   Pytorch 1.2.0

   CUDA 10.1 V10.1.105

   cudnn v7.6.3 for CUDA 10.1
3. 开发 IDE

   Jupyter NoteBook

   Pycharm

三、实验过程

1. 数据导入

首先在 Cifar 数据官网下载 cifar10 数据集解压，关于数据文件的说明可以参考官网：https://www.cs.toronto.edu/~kriz/cifar.html

数据处理和数据读取包 MyDataset：

1) Class Cifar10：继承 torch.utils.data.Dataset 类

   __init__(): 读取 batch 文件，加载训练集和测试集数据

   __getitem__(): 根据 index 获取对应标号的数据和相应标签

   __len__(): 获取数据大小

2) Class Totensor():将数据转换成 torch.tensor.().float()

```
# 如果直接使用 lambda 表达式用于 tensor 转换，在 dataloader num_workers>0
# 时调用多线程取数据时或报错
train_dataset = Mydataset.Cifar10(root, train=True,
                    transform=lambda x: torch.tensor(x,dtype=torch.float))
test_dataset = Mydataset.Cifar10(root, train=False,
                    transform=lambda x: torch.tensor(x,dtype=torch.float))
```

3) 也可以直接使用 kaggle 的 kernel 进行网络训练，获得超强算力！

2. 设计定义网络

1) 简单复现 Alexnet

我们参考 Alexnet 的结构设计出具有 5*CNN+3*Fully Connected 的一共八

层的网络。其中五层卷积层作为 feature extractor，用于从图片中提取特征，3 层全连接层根据卷积层提取的特征向量来对输入图片进行分类。

卷积层设计：

a) 卷积核选择：考虑到 cifar10 数据集的图片特征，我们选用的是小尺寸的卷积核，我们主要考虑了 5*5 和 3*3 的卷积核。

b) 为了避免网络层数增加导致的 Internal Covariate Shift 现象，在 Activate Function 之前添加了 Batch Norm 层。

c) Activate Function：Sigmoid 和 Relu，其中 Sigmoid 函数再添加 BN 层之后效果明显上升（归一化输入，使得卷积后的输出值主要集中在激活函数的感受域）。
总体比较，Relu 性能优于 Sigmoid。

d) 池化层：采用最大池化，叠加池化和非叠加池化均存在。

全连接层设计：

a) 考虑 Cifar 数据只需要对图片分是 10 类，远小于 Imagenet 中的 1000 类，大幅删减全连接层中的节点个数。

b) 最后 output 10 个值

```
(0): Conv2d(3, 96, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): BatchNorm2d(96, eps=1e-05, momentum=0.9, affine=True, track_running_stats=True)
(2): ReLU()
(3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
from torchsummary import summary
summary(model,input_size=(3,32,32))
# output
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 96, 32, 32]           2,688
       BatchNorm2d-2           [-1, 96, 32, 32]             192
              ReLU-3           [-1, 96, 32, 32]               0
         MaxPool2d-4           [-1, 96, 16, 16]               0
            Conv2d-5          [-1, 128, 16, 16]         307,328
       BatchNorm2d-6          [-1, 128, 16, 16]             256
              ReLU-7          [-1, 128, 16, 16]               0
         MaxPool2d-8          [-1, 128, 15, 15]               0
            Conv2d-9          [-1, 256, 13, 13]         295,168
      BatchNorm2d-10          [-1, 256, 13, 13]             512
             ReLU-11          [-1, 256, 13, 13]               0
           Conv2d-12          [-1, 256, 13, 13]         590,080
      BatchNorm2d-13          [-1, 256, 13, 13]             512
             ReLU-14          [-1, 256, 13, 13]               0
           Conv2d-15          [-1, 128, 13, 13]         295,040
      BatchNorm2d-16          [-1, 128, 13, 13]             256
             ReLU-17          [-1, 128, 13, 13]               0
```

```
        MaxPool2d-18            [-1, 128, 6, 6]               0
           Linear-19                  [-1, 128]         589,952
      BatchNorm1d-20                  [-1, 128]             256
             ReLU-21                  [-1, 128]               0
           Linear-22                   [-1, 64]           8,256
      BatchNorm1d-23                   [-1, 64]             128
             ReLU-24                   [-1, 64]               0
           Linear-25                   [-1, 10]             650
================================================================
Total params: 2,091,274
Trainable params: 2,091,274
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 5.92
Params size (MB): 7.98
Estimated Total Size (MB): 13.91
----------------------------------------------------------------
```

3. 选择代价函数，超参数，优化器

   1) 模型的超参数

   Alexnet 的网络比实验一中的多层感知器复杂许多，涉及到的网络层类型有卷积层，池化层，全连接层，Batch_Norm 层，还有 Batch size，学习率的选择。需要考虑的参数非常多，所以我们有选择性的进行了实验。

   网络结构：网络的设计是复杂的，同时卷积层，池化层的选择，蚕食的选择也是复杂的，这里只能给出实验的感性认知：

   对于同样的数据，参数越多，模型越复杂的神经网络不能带来明显的性能提升，反而会造成实验的过拟合

   Lr：学习率对于网络的学习速度和效果非常关键，较长使用的学习率一般是 0.005,0.001,0.0001 等。一些同学采用了小学习率，多 epoch 的策略进行学习，epoch 的数量可能高达几百。但是这种方式往往很难有效。

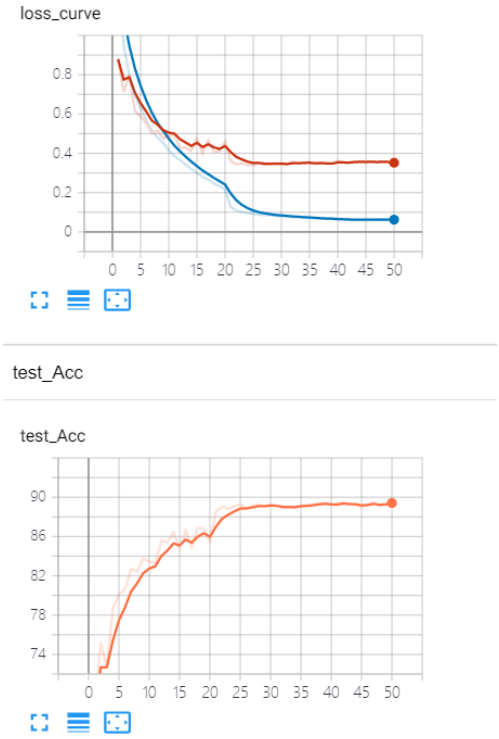
   可以采用 lr_scheduler 对学习率进行动态设置，多步长的设置。

   Batch size：文献中说大的 Batch size 能够每个 epoch 学习的时间（允许 GPU 做更大规模的并行计算），但是 Batch size 增加也会导致实际的学习率降低，因为由下列公式可知每次 mini batch gradient decent 每次更新的梯度大小和 Batch size 成反比，即 Batch size 增大能够起到类似于减小学习率的效果，这会导致收敛的 epoch 增多。

$$\boldsymbol{g}_t \leftarrow \nabla f_{\mathcal{B}_t}(\boldsymbol{x}_{t-1}) = \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}_t}\nabla f_i(\boldsymbol{x}_{t-1})$$

综上，理论上存在一个以收敛为目标用时最短的 Batch size。

大 Batch 一定程度上能够稳定，小 Batch 则能够提供更多的随机性。而这个特点在实际训练的过程中被体现出来：我们发现在前期使用小 batch 训练，而在后期使用一个较大的 batch 训练，会在出现一个效果的跃变。



2) 代价函数选择
分类问题，采用交叉熵作为代价函数。

3) 优化器选择
主要可以选择 SGD 和 Ada 两大类，常用的有 SGDM 和 Adam，但是 Adam 在实际使用的过程中在前期可以快速收敛，但是在后期由于学习率过低（Adam 算法会考虑之前梯度的类和，实际学习率会随着训练的增加而减小），导致不收敛的现象出现。
我在实验中主要使用的是 SGDM，因为可以根据实验观察结果，更好的设置学习率。
SGDM 的学习率动态设置也可以使用 lr_scheduler 进行动态设置，如 Multistep 保证在学习前期能够以大步长到达“低洼”地区，在学习后期也不会应为学习率过大出现震荡的现象。

四、结果分析

网络	Epoch	Batchsize	Lr	优化器	Accuracy
Pure	50	128	0.001	SGDM	75.56%
增加 BN	50	128	0.001	SGDM	80.30%
增加 Dp 模块	50	128	0.001	SGDM	82.95%
增加 数据增广 (水平翻转)	50	128	0.001	SGDM	86.01%
增加 多 batch 训练	50	32 (20 epoch) 128 (20 epoch) 256 (10 epoch)	0.001	SGDM	88.63%
Best	50	32 (20 epoch) 128 (20 epoch) 256 (10 epoch)	0.05 0.005 0.0005	SGDM	89.49%

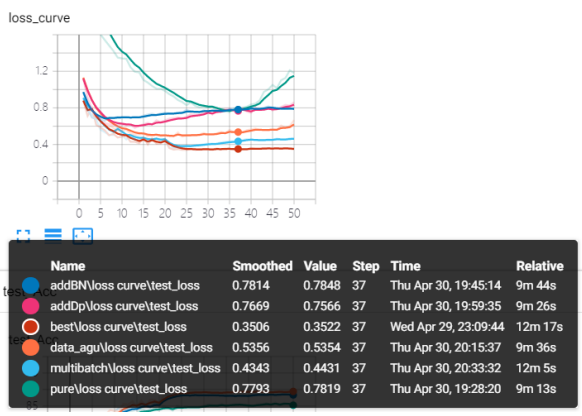
说明：

所有的训练的测试数据都进行了归一化处理。

1) Pure 版网络结构仅仅单纯实现了（卷积-池化-激活）-（全连接-激活）层结构。此时训练的初始 loss 非常大，test_acc 会从 10%开始上升，最后上升到 75%左右的准确率，之后就会到达瓶颈
2) 添加了 Batch Norm 之后，第一个 epoch 的 test_acc 就到达了 65%，说明批标准化能够在训练前期起到更好的初始化和加速收敛的效果。BN 之后，可以观察到 train_loss 快速下降，但是 test_loss 却很早的进入瓶颈，导致测试准确率一直卡在 80%（测试准确率和 test_loss 关系并不绝对，实验中也观察到了 testloss 增加，准确率却继续上升的情况）
3) 添加 Dropout（仅在全连接层），第一个 epoch 的 test_acc 就到达了 60%左右，train_loss 下降的速度明显比单纯添加 BN 的降低了，但是 test_loss 却获得了很好的持续下降。这印证了 Dropout 能够一定程度抑制模型对训练集的过拟合。
4) 数据增广：这是 Alexnet 论文中的思路。通过随机改变训练样本们可以降低模型对某些属性的依赖，提高模型的泛化能力。Torchvision.transforms 中提供了许多数据增广的方式可供选择。
   使用数据增广之后性能大于能够提升 2%-3%。
5) 多阶段 batch 训练：基本想法是先使用小 batch 训练一定数量 epoch 之后再换成大 batch 继续训练。实验发现在转换 batch 的时候会使得测试效果变好，出现 test_acc 的跃变。
   我的解释是小 batch 的随机性使得模型在开始训练的时候能够选择到更好的优化方向，而大 batch 则帮助模型稳定收敛。

还有许多其他的技巧，比如 multistepLR 和动态调整学习率的方法，但是都没有带来明显的性能提升，后续实验会接着探索。

还应该尝试不同的优化器。



五、总结

1. 卷积层的设计需要考虑的输入图片的特征。同时卷积技术和思想不单单适用于图片特征提取，CNN 还可以用于处理 word embedding 后的词序列预测等等。最关键的是 CNN 技术中 **Shared weight**，**Pattern detector** 的思想。
2. DNN 能够对图片很好的分类，但是在 DNN 分类结果中具有很高置信度的图片不一定属于该图片。DNNs are easily fooled!
3. 多看论文，理论指导实验。

六、参考

【优化方法 SGD】https://fyubang.com/2019/08/10/optimizer_sgd/
【batch 实现对比】https://www.jianshu.com/p/71f31c105879
【kaggle notebook】https://www.kaggle.com/motoight/kernel7731672045