

Part 2：卷积神经网络

代码基本结构

这部分由于可以调用PyTorch的库，代码形式比较简单。

所有代码存放在Part3文件夹下，有三个文件组成。

Part3

- CNNmodel.py
- Draw.py
- main.py

CNNmodel.py: 内部实现了两个CNNmodel，一个用了两层卷积层，一个用了一层卷积层。

Draw.py: 实现对数据的可视化处理。

main.py: 主体部分。实现了7个函数或类。

- 1. timethis(), 是一个装饰器。添加了@timethis注解的函数会自动打印函数的运行时间。
- 2. CustomDataset, 自定义数据集类，届时用于创建训练数据集和测试数据集。
- 3. createData(), 利用DataLoader返回trainLoader和testLoader
- 4. train(), 训练模型，反向传播优化模型。
- 5. validate(), 测试模型，不进行反向传播。
- 6. MyTrain(), 主函数。设置超参，创建模型，训练和验证，最终保存Loss, Acc曲线以及模型。
- 7. AnaTrain(), MyTrain的复制品，用于自动化测试各种结构的CNN网络。

网络结构改进与对比

对比总述

所有结果记录在了[CNN网络结构对比.xls](#)之中。（PDF可能需要手动打开文件）

单卷积层: 1F -> 1F -> 1F -> 1F									
Batch Size	lr	Epochs	Best-Epoch	Min-Loss	Correct_Ratio	Loss-Tendency	CE_Pic	Time	
1	0.01	30	30	1.435877340	97.61%			1086.772802	
5	0.2	30	30	1.435289610	96.48%			201.731176	
10	0.2	30	24	1.435805103	96.94%			363.932081	
双卷积层: 1F -> 1F -> 1F -> 1F -> 1F									
Batch Size	lr	Epochs	Best-Epoch	Min-Loss	Correct_Ratio	Loss-Tendency	CE_Pic	Time	
1	0.01	30	30	1.435867780	96.94%			1081.922893	
5	0.2	30	18	1.435738411	97.08%			276.942804	
10	0.2	30	26	1.435215864	96.74%			203.8941721	
双卷积层: 1F -> 1F -> 1F -> 1F -> 1F -> 1F									
Batch Size	lr	Epochs	Best-Epoch	Min-Loss	Correct_Ratio	Loss-Tendency	CE_Pic	Time	
1	0.01	47	39	1.435702040	96.48%			743.7748115	~80%~80%
5	0.01	47	37	1.435138357	96.68%			886.2014105	~80%~80%
10	0.2	30	24	1.435274802	95.93%			375.1262774	
双卷积层: 1F -> 1F -> 1F -> 1F -> 1F -> 1F -> 1F									
Batch Size	lr	Epochs	Best-Epoch	Min-Loss	Correct_Ratio	Loss-Tendency	CE_Pic	Time	
1	0.1	60	22	1.438738410	97.43%			639.1870019	~80%~80%
5	0.1	60	27	1.43873436	96.13%			440.1333359	~80%~80%
7	0.1	60	60	1.438733332	97.92%			724.7416250	~80%~80%

总共对比了两种CNN网络结构：单卷积层和双卷积层（仿LeNet5）

单卷积层下对比了不同的BatchSize，不同的卷积核数量。

双卷积层下对比了不同的KernalSize，不同的卷积核数量。

CNN内部结构对比

单双卷积层对比

- 1. 单层收敛更快（5epoch左右），双层需要10epoch往上，但这也与我的lr设定有关。
- 2. 最佳分类由双卷积层产生，为98.13%；单层最佳为97.08%。
- 3. 双卷积层的Loss较为稳定，单卷积层Loss经常性抖动。

单卷积层BatchSize

- 1. 测试了三种BatchSize(1,8,32)，三者正确率几乎相同，误差不超过1%。均在96%~97%之间。
- 2. 训练时间上，BatchSize1是BatchSize8的2倍，是BatchSize32的3倍。
- 3. BatchSize需要用更小的学习率学习，否则根本无法训的动。

单卷积层卷积核数量

- 1. 测试了8/16/32三种卷积和数量，平均效果最好的是16个卷积核，但与其他两者差距不超过±0.5%。最佳效果也是在16个卷积核时取到，为97.8%。
- 2. 卷积核数量越多，训练时间越长，但影响程度相比BatchSize来说影响的小。

双卷积层KernalSize

- 1. KernalSize从3变到7，网络结构比较简单时，训练时长小幅上升（不超过10%）。较复杂时（3Feature Map → 32 FM → 64FM），时长增加了30%。
- 2. 对于三种网络结构，KernalSize为5平均表现最好，正确率最高也是由5时取到，为98.13%。
- 3. 大KernalSize更早收敛，Acc趋于稳定。

双卷积卷积核数量

- 1. 卷积核数量最多时（先32后64）时取得最佳结果，98.13%。
- 2. 三种卷积和数量最终结果相差并不大，约1%，但多卷积核训练时间要增长10%~30%。

对比手写多层感知机

	多层感知机	CNN
正确率上限	90%	98%
正确率范围	85%~90%	95%~98%
过拟合现象	高，Train与Test相差10%	低，Train与Test相差2%
收敛轮数	慢，至少需要100epochs	快，约15个epoch
训练速度	慢，基本需要1000s以上	快，约100s
Learning Rate	1e-3级别	1e-1级别（可能与网络结构有关）
小BatchSize	准确率高于大BatchSize	准确率不如大BatchSize

毕竟CNN的复杂程度要比多层感知机高，而且CNN本身也包含有一部分多层感知机，同时CNN模仿了人类识别物体从低级特征到高级特征顺序，分类效果比多层感知机效果好是自然的。

对CNN网络设计的理解

卷积层

说卷积核是对特征的提取，靠前的卷积层提取低级特征，靠后的卷积层从低级特征中提取高级特征。

我认为卷积核的作用实际上是和传统CV中的算子相似，至少从运算方式上看，卷积核与算子是一样的。算子对原图片运算之后，得到了某种特征的图片，例如Sobel算子得到了边缘，卷积核应该亦如此。区别在于算子很明确自身要提取何种特征，但卷积核却需要BP优化自身权值。也就是卷积核在不断优化自身所提取到的权值，最终收敛到一个针对输入最有效的特征。

如果人类本身能清晰地解释如何对猫狗分类，对汉字分类了，那么可能卷积层就不需要进行BP了。因为人可以上来就指定好Kernal的权值，也就是某某算子，提取出指定的特征，从而完成任务。

池化层

池化层最大的作用就是削减特征数量，例如将28*28的图片转化成了14*14的图片，这大大降低了网络的复杂程度以及计算量。

ReLU和池化有异曲同工之妙，ReLU放弃了负值，保留了具有“刺激其他神经元”正值输出。

池化的思想与计算机图形学中的Spatial Pyramid Model有些像。SPM为每一张图片采用Avg或者Max的方式生成分辨率更小的图片，当需要小图片时用生成的小分辨率图片进行平铺，得到的效果比直接用大图片压缩更好。

池化层也起到了一定抑制过拟合的作用，因为它抛弃了一定的特征。对于图片共有的特征来说，它的权值会更大些，因而更不容易抛弃。对于部分图片特有的特征，它的权值在其它训练集的BP下更容易在Pooling时被放弃。

搭建网络

一般来说CNN都包含若干CONV+RELU+POOL的结构。针对网络的复杂程度，再确定该结构的数量。

最后用全连接层链接所有特征，输出值传入分类器。

至于复杂程度如何判断，这又是个问题。因为人都无法解释自己是如何解决问题的，自然也不知道问题的难度是多少。

但至少还是有一部分问题是可以判别出难度的，比如说分类猫狗就肯定比分类哈士奇和狼要简单。因为后者处理的分类对象更为相似，可以想象到要找出能够区分两者的特征是比较困难的。因此要初始化一个比较大，特征提取很多的CNN网络来处理分类哈士奇和狼的问题。

Bonus：减少过拟合的尝试

使用L2正则化

做法

在optimizer中设置一个小于1的weight_decay参数，在每次更新之后，所有参数都要乘上weight_decay。

```
optimizer = optim.SGD(model.parameters(), lr=lr, weight_decay=1e-3)。
```

目的

减小参数大小从而降低模型复杂度，最终提高模型泛化能力。

使用后效果

对于卷积核个数8->16、卷积核3*3的双卷积层网络，经过三次测试，发现正确率从徘徊在96.8%~97%上升到了徘徊在97.3%~97.5%，最高正确率从97.01%提高了97.78%。

对于卷积核个数16->32、卷积核5*5的双卷积层网络，经过三次测试，发现正确率从徘徊在98.4~98.5%下降到了徘徊97.8%~98.1%，最高正确率从98.6%下降到了98.1%

但是不论哪种CNN模型，中途出现了突然模型崩坏的情况，正确率突然降低至8.33%。

```
-----Epoch: 19 -----
Train Loss: 0.002604587934911251, Train CR: 100.00 %
Test Loss: 0.09956305474042892, Test CR: 97.36%
-----Epoch: 20 -----
Train Loss: 0.002594045363366604, Train CR: 100.00 %
Test Loss: 0.09787531197071075, Test CR: 97.36%
-----Epoch: 21 -----
Train Loss: 2.2233152389526367, Train CR: 21.33 %
Test Loss: 2.485306739807129, Test CR: 8.33%
-----Epoch: 22 -----
Train Loss: 2.4861958026885986, Train CR: 7.85 %
Test Loss: 2.485023260116571, Test CR: 8.33%
-----Epoch: 23 -----
```

使用Dropout

做法

在卷积层中添加 `nn.Dropout()`

dropout在每个训练批次中，忽略一定比例的神经元，让他们不工作，间接降低了模型的复杂程度。

集成学习里会用好几个模型对同一个目标进行预测，少数服从多数决定最终结果。因为每个模型都有不同程度的过拟合，综合多个模型结果相当于取了平均，有效降低了过拟合。

dropout与集成学习有相似的效果。每次隐藏一部分神经元就相当于在训练一个不同的网络，训练结束后的最终模型就相当于多个网络模型的平均。

此外由于两个神经元不一定每次都在一个dropout过后的网络中出现，有依赖关系的节点在dropout时可能会失联，它们所产生的特征不会次次都被学习，所以dropout也在一定概率上强迫模型去学习一些更鲁棒的特征。

效果

选用了卷积核个数(16->24)，卷积核大小5*5的双卷积层网络。原本最高正确率98.6%(无weight decay)，采用dropout后最高正确率高达99.03%，而且基本徘徊在98.7%左右。可见dropout的效果确实好。

```
Train Loss: 0.017324743792414665 , Train CR: 99.52 %  
Test Loss: 0.10977295786142349, Test CR: 97.22%  
-----Epoch: 19 -----  
Train Loss: 0.01999112218618393 , Train CR: 99.35 %  
Test Loss: 0.05791625380516052, Test CR: 98.75%  
-----Epoch: 20 -----  
Train Loss: 0.012144063590407672 , Train CR: 99.62 %  
Test Loss: 0.0626891478895141, Test CR: 99.03%  
-----Epoch: 21 -----  
Train Loss: 0.012947842478752136 , Train CR: 99.55 %
```

实验中遇到的问题及解决

Pytorch的CrossEntropy

我自己手写的CrossEntropy在计算loss的时候是没有问题的：正确率到90%左右时，loss在0.6~0.7左右，这还是比较合理的。

但是用Pytorch的CrossEntropy时，我发现哪怕我的正确率已经来到了98%，loss都仍然能达到1.6~1.7。

我怀疑Pytorch的CE是有问题的，例如下面这个例子。

```
x = torch.tensor([[0.01, 1.0, 0.001],  
                  [0.33, 0.33, 0.33],  
                  [0.7, 0.3, 0.001]])  
  
y = torch.tensor([1,  
                  2,  
                  0])  
  
loss_fn = torch.nn.CrossEntropyLoss(reduction='none')  
print(loss_fn(x,y))
```

运算结果为

```
tensor([0.5538, 1.0986, 0.7735])
```

设目标值one-hot为t，预测值为y_hat，按照公式

$$Loss = \sum t_i \log(\hat{y}_i)$$

结果应该为 [0, 1.0986, 0.357]。很显然pytorch给出的结果第一个和第三个都是错误的，第二个loss最大的情况下却正确。

解决方案

实际上是因为pytorch的CrossEntropy自带Softmax，因此我没必要再CNNmodel里加入一层Softmax。这也解释了为什么Loss在三个概率相等时与我手算是一样得了。

由于解决这个问题解决的比较的晚，所以我xls文件中记录的loss都相对来说比较大。不过我测试了有无Softmax层都不会对反向传播产生影响，只是会影响Loss的大小，所以也Loss数值错了无伤大雅，至少趋势是正确的。

(这印证了我在手写神经网络中提出的“Softmax和CrossEntropy是捆绑的”论点，现在看来pytorch也是这么干的。只有捆绑了才能优雅的求出关于Loss的梯度)