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。

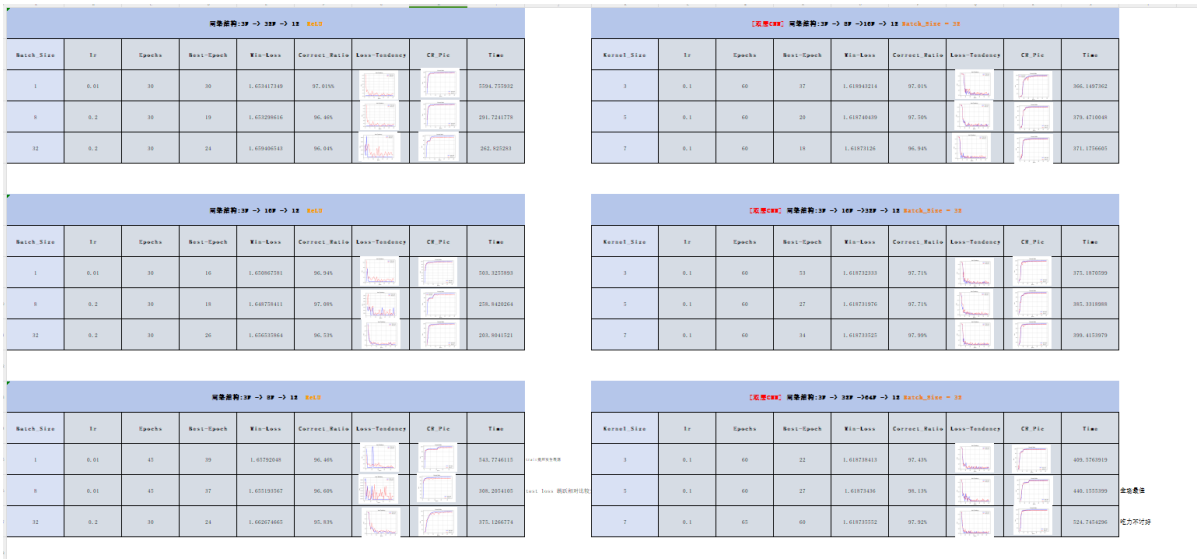
main.py：主体部分。实现了7个函数或类。

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。



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的顺序，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。

实验中遇到的问题

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。

但是Loss的数值不影响实验的正确性，而且Loss的趋势也还是同样的，所以我xls文件里的loss就不用信了。