**

**HUNAN UNIVERSITY**

人工智能

实验报告

|  |  |  |
| --- | --- | --- |
| **题 目：** | | 实验四：深度学习算法及应用 |
|  |  |  |
|  | 学生姓名： | 朱钇冰 |
|  | 学生学号： | 201708010803 |
|  | 专业班级： | 计科1706 |
|  | **完成日期：** | 2019年12月28日 |

**目 录**

[一、内容 2](#_Toc524883658)

[二、目的 2](#_Toc524883659)

[三、实验设计思想和流程 2](#_Toc524883660)

[四、主要数据结构及符号说明 7](#_Toc524883661)

[五、实验环境以及实验过程与结果分析 8](#_Toc524883662)

[六、实验体会和思考题 20](#_Toc524883663)

[附录（源代码及注释） 21](#_Toc524883664)

## 一、内容

安装开源深度学习工具设计并实现一个深度学习模型，它能够学习识别图像中的数字序列。然后使用数据训练它：你可以使用人工合成的数据（推荐），或直接使用现实数据。

## 二、目的

1、了解深度学习的基本原理

2、能够使用深度学习开源工具识别图像中的数字

3、了解图像识别的基本原理

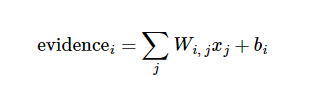
## 三、实验设计思想和流程

**核心算法思想：**

**1、softmax回归**

为了得到一张给定图片属于某个特定数字类的证据（evidence），我们对图片像素值进行加权求和。如果这个像素具有很强的证据说明这张图片不属于该类，那么相应的权值为负数，相反如果这个像素拥有有利的证据支持这张图片属于这个类，那么权值是正数。

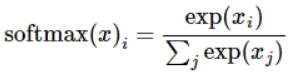
我们也需要加入一个额外的偏置量（bias），因为输入往往会带有一些无关的干扰量。因此对于给定的输入图片 x 它代表的是数字 i 的证据可以表示为其中wi描述代表权重，bi描述代表数字 i 类的偏置量，j 代表给定图片 x 的像素索引用于像素求和。



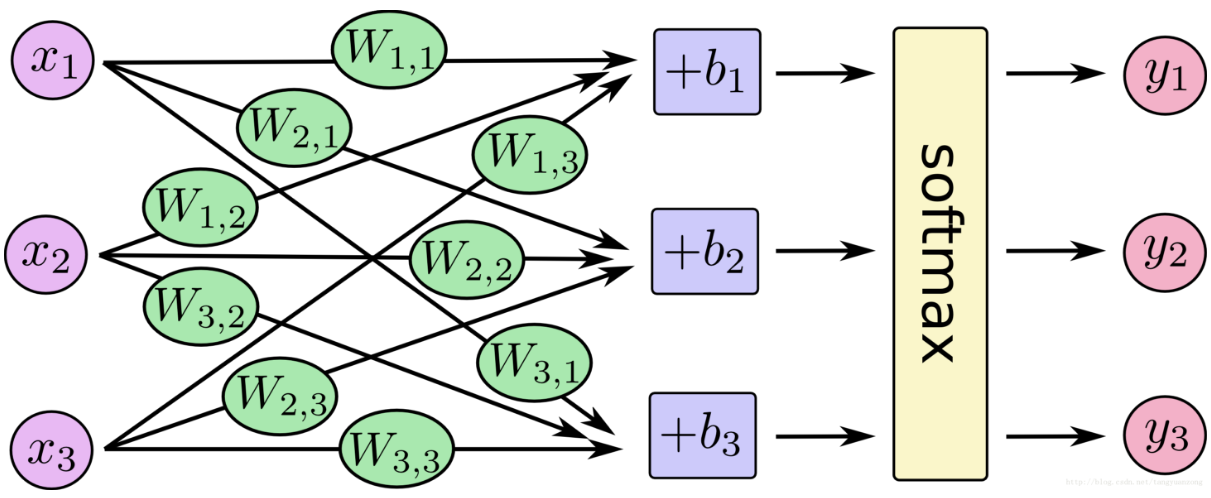
然后用softmax函数可以把这些证据转换成概率 y：



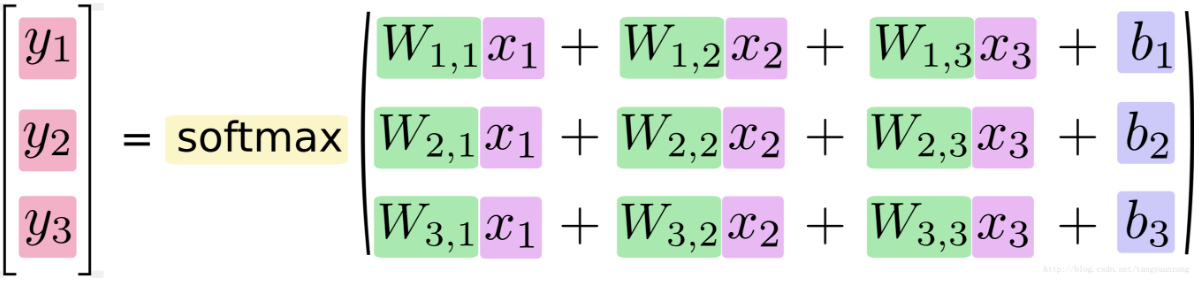
这里的softmax可以看成是一个激励（activation）函数或者链接（link）函数，把定义的线性函数的输出转换成想要的格式，也就是关于10个数字类的概率分布。因此，给定一张图片，它对于每一个数字的吻合度可以被softmax函数转换成为一个概率值。softmax函数可以定义为：



模型：对于输入的xs加权求和，再分别加上一个偏置量，最后再输入到softmax函数中：



写成的xingshi1可以得到



**2、CNN卷积神经网络**

图像处理中，往往把图像表示为像素的向量，比如一个 1000×1000 的图像，可以表示为一个10^6 的向量，就如同在手写字神经网络中，输入层为28×28 = 784 维的向量。如果隐含层的节点个数与输入层一样，即也是10^6 时，那么输入层到隐含层的参数数据为 10^6×10^6 =10^12，参数的个数太多了，要想在正常的时间内训练完，基本是不可能的。所以要想处理1000×1000的图像分类，就得首先想办法减少参数的个数，也就是只基于深度神经网络（DNN）已经很难训练或者有没有更加优秀的算法可以专门处理这种图像分类呢？ 这就是卷积神经网络，convolutional neural network ，简称为 CNN 。

**卷积：**应用了两种技术： 局部连接、权值共享

减少了权重参数，具体这个操作为卷积操作，这个操作有3个重要的超参数：

步长（Stride）、零填充（Zero-padding）、深度（Depth）

在卷积操作中，涉及到一种特殊的操作，叫做求内积，它是两个同型矩阵对应的元素相乘，然后求和。

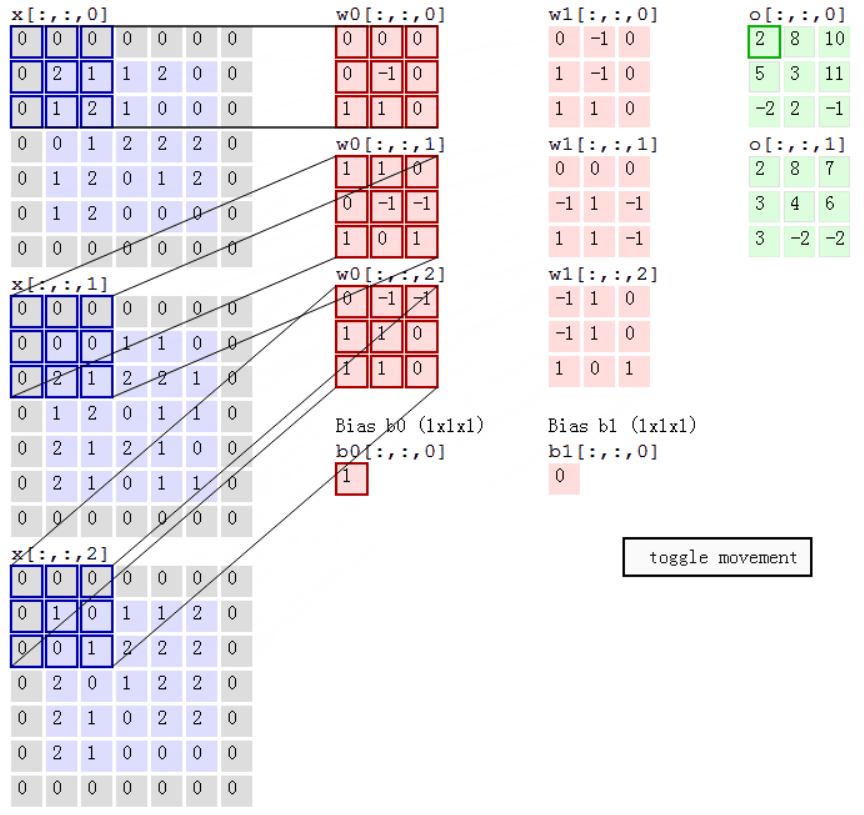
例：对于A矩阵：1、0、1、0、1、0、1、0、1与B矩阵：1、0、1、0、1、0、1、0、1

求内积：1\*1 + 0\*0 + 1\*1 + 0\*0 + 1\*1 + 0\*0 + 1\*1 + 0\*0 + 1\*1 = 5

对于如下矩阵进行卷积：

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 0 | 0 | + | 卷积核 | | | => | 结果 | | |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 4 | 3 | 4 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 2 | 4 | 3 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 2 | 3 | 4 |
| 0 | 1 | 1 | 0 | 0 |  | | |  | | |

如果用多个卷积核进行卷积操作，图中的输入为 7 × 7 × 3，可以看到还做了一层零填充（Zero-padding），这是CNN中另一个重要的超参数，用到了两个过滤核：w0和w1，这在CNN中称为深度（Depth），是CNN三个超参数介绍的最后一个，分别用两个过滤核w0,和w1做了一次卷积操作，对应的得到两个卷积结果。



**补0（zero padding）**

我们的图片通过卷积操作，再通过池化操作后，如果我们再添加层，图片会越变越小。这个时候我们就会引出“Zero Padding”（补零），它可以帮助我们保证每次经过卷积或池化输出后图片的大小不变。如果原始图片大小为4\*4，采用3\*3的卷积核，那么我们可以加入1的zero padding。得到下图，这样卷积完毕后仍然得到6\*6的矩阵

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |

**池化：**Pooling层能起到降低上一层输入的特征的维数的作用，但是同时能保持其最重要的信息，Pooling操作分多种：最大池化，平均池化，求和池化等。

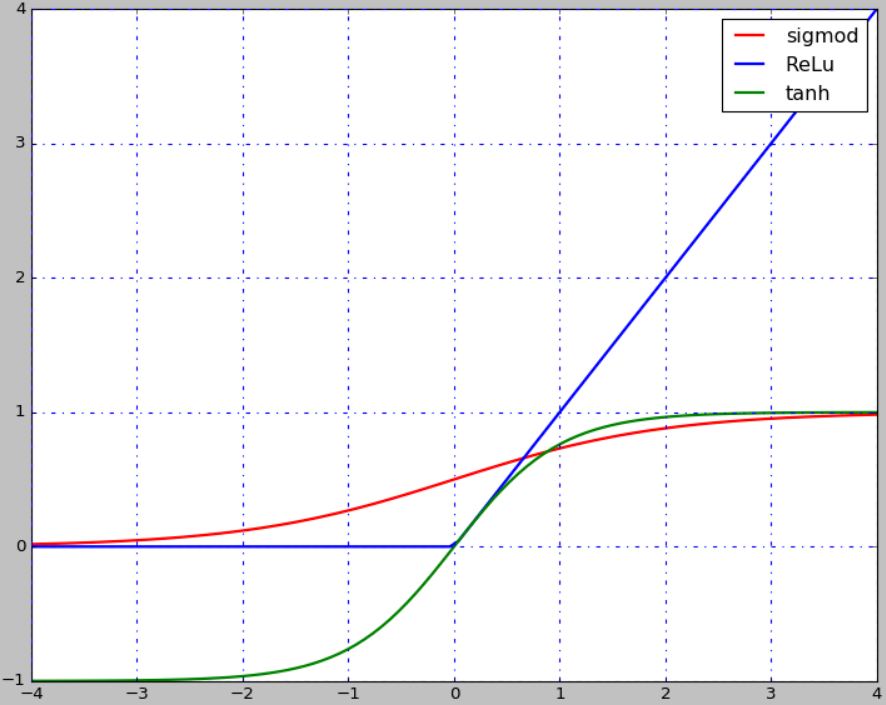
以最大池化为例，池化一般在ReLU操作之后，首先定义一个相邻区域，然后求出这个区域的最大值，再选定一个步长2，依次遍历完图像，如下图所示

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 2 | 4 | => |  | |
| 5 | 6 | 7 | 8 | 6 | 8 |
| 3 | 2 | 1 | 0 | 3 | 3 |
| 1 | 2 | 3 | 4 |  | |

Pooling的好处是减少参数,就有平移不变性（(translation invariant）

**激活层:**在每次卷积操作之后一般都会经过一个非线性层，也是激活层

现在一般选择是ReLu,层次越深，相对于其他的函数效果较好，还有Sigmod,tanh函数等



sigmod和tanh都存在饱和的问题，如上图所示，当x轴上的值较大时，对应的梯度几乎为0，若是利用BP反向传播算法， 可能造成梯度消失的情况，也就学不到东西了

ReLU函数对CNN的实际意义:可以看出CNN中的卷积操作是线性操作，对应元素相乘，然后再求和，又知道在现实世界中，大部分的数据都是非线性的，所以有必要引入一个非线性的激活函数，对于单通道图上取值范围为0－255，其中255为白色，0是黑色，对原图像卷积后的图像，因为卷积后的结果有可能为负值，所以经过ReLU后，所有的值为非负值，不再出现有问题的负值。

**Dropout层**

dropout是一种防止模型过拟合的技术，这项技术也很简单，但是很实用。它的基本思想是在训练的时候随机的dropout(丢弃)一些神经元的激活，这样可以让模型更鲁棒，因为它不会太依赖某些局部的特征（因为局部特征有可能被丢弃）

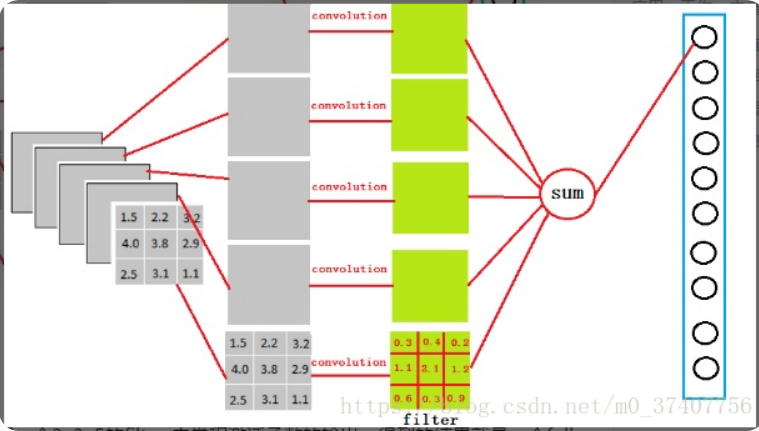
**全连接层 Fully connected layer**

卷积取的是局部特征，全连接就是把以前的局部特征重新通过权值矩阵组装成完整的图。

因为用到了所有的局部特征，所以叫全连接

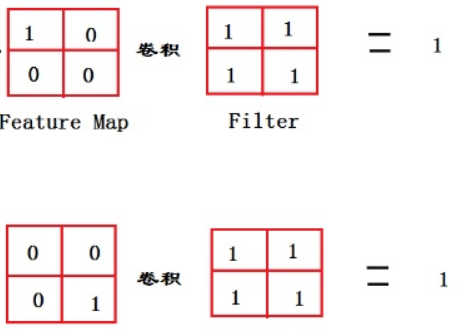
例如经过卷积，relu后得到3x3x5的输出。

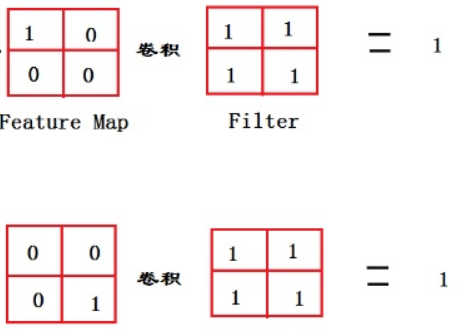
把3x3x5的输出，转换成1x4096的形式: 可以理解为在中间做了一个卷积



用一个3x3x5的filter 去卷积激活函数的输出，得到的结果就是一个fully connected layer 的一个神经元的输出，这个输出就是一个值。因为我们有4096个神经元。我们实际就是用一个3x3x5x4096的卷积层去卷积激活函数的输出。

把特征representation整合到一起，输出为一个值。就是大大减少特征位置对分类带来的影响。





1在不同的位置，输出的feature值相同，但是位置不同。

对于电脑来说，特征值相同，但是特征值位置不同，那分类结果也可能不一样。

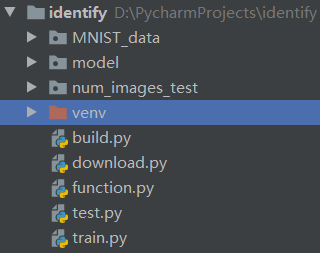
这时全连接层filter的作用就相当于1在哪不管，我只要有，于是我让filter去把这个喵找到，

实际就是把feature map 整合成一个值，这个值大，有1，这个值小，那就可能没1

和这个1在哪关系不大了，鲁棒性有大大增强。

因为空间结构特性被忽略了，所以全连接层不适合用于在方位上找Pattern的任务，比如segmentation。

## 四、主要数据结构及符号说明

工程文件夹

数据集

模型文件夹

手写测试

相关库文件

构建CNN网络.py

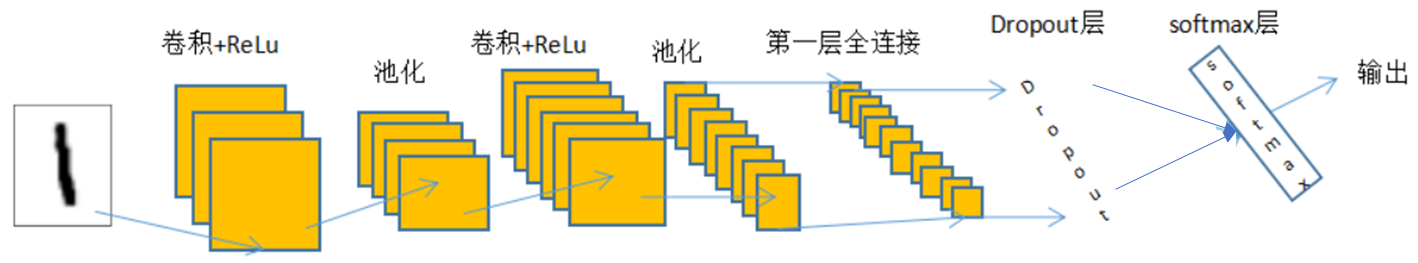
下载数据集.py

功能函数声明.py

测试手写图片.py

训练生成模型.py

函数流程如下：



两次卷积加池化，一层全连接，加入dropout后softmax函数输出

## 五、实验环境以及实验过程与结果分析

实验环境：

硬件：计算机

软件：操作系统：WINDOWS

应用软件：Pycharm、Anaconda、Tensorflow

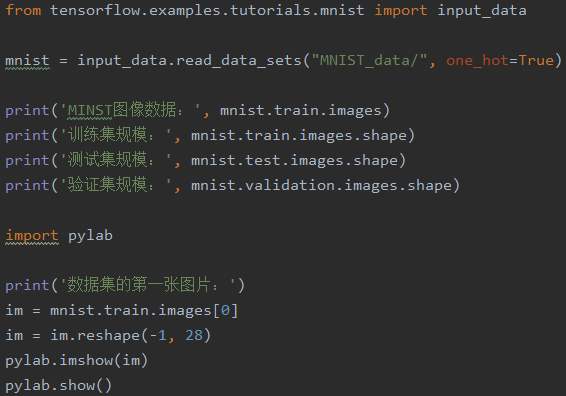
实验过程：

**1.安装Pycharm、Anaconda、Tensorflow相关软件**

**2.获取数据源：MNIST数据集**

MNIST数据集是一个开源的手写数据库。它提供了大量的数据样本作为训练集和验证集。这个数据集拥有 60000 个训练样本，和 10000 个测试样本。

运行download.py即可自动下载数据集到对应目录下。

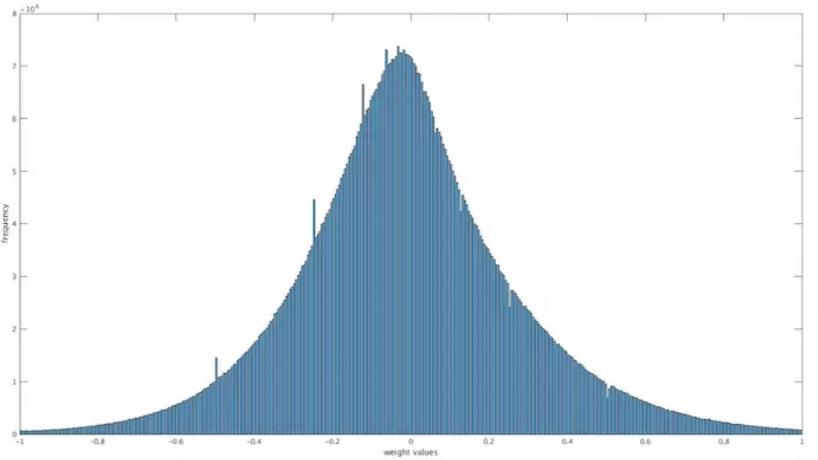


**3.定义辅助函数：用来创建 权重、偏置量、卷积核和池化层。function.py**

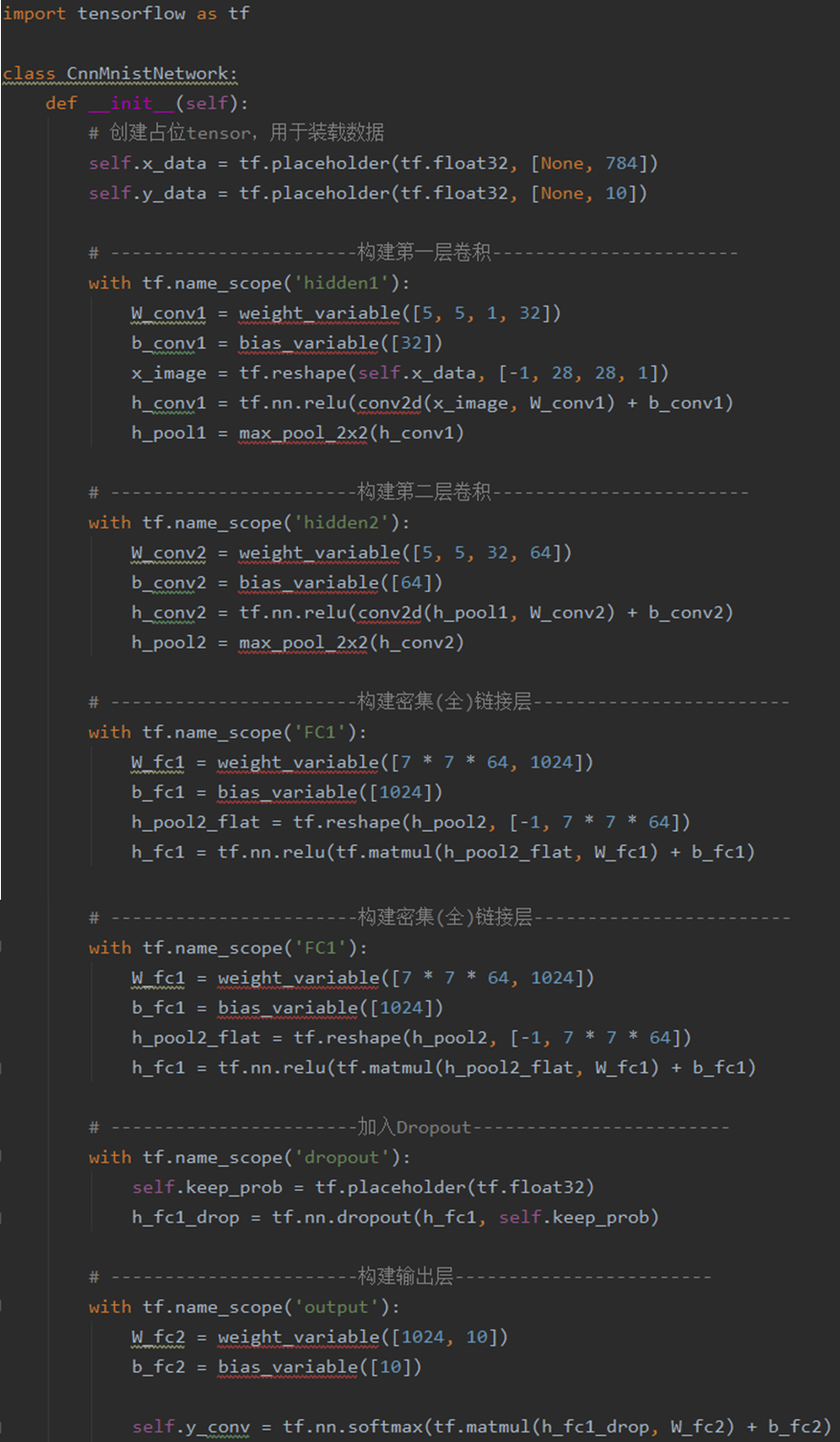


其中，权重 weight 的生成：

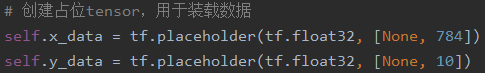
从一个正太分布（该正太分布的均值为 mean，默认为0）中产生随机数，stddev 表示标准差，就是图中横坐标上的取值，也就是说，这个函数会从 [mean ± 2stddev] 的范围内产生随机数，如果产生的随机数不在这个范围内，就会再次随机产生，直到随机数落在这个范围内。使用这种方式产生的 weight 不容易出现梯度消失或者爆炸的问题。



**4.构建CNN卷积神经网络build.py**



**（1）创建数据输入的占位符**



需要先创建两个占位变量来容纳待会儿输入的数据，然后把它们带着到网络中进行运算。

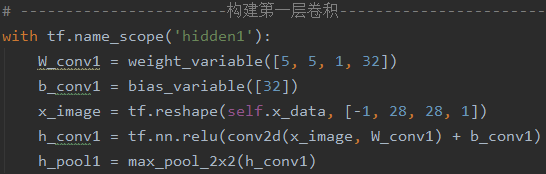
**① x\_data** 是用来容纳训练数据的，它的 shape 形状在这里被莫名其妙的定义为 [None, 784]。其实这里是有学问的，且听 CoorChice 慢慢道来。

第一维定义为 None 表示不确定，后面会被实际的数值替代。这样做是因为我们一开始并不知道会有多少张图片数据会被输入。或者当我们采取 mini-batch 的梯度下降策略时，可以自由的设置 batch 的大小。

第二个维度定义为 784，这完全是因为我们数据集中的图片大小被统一为了 28\*28 。

**② y\_data** 是用来容纳训练数据的标签的，它的 shape 之所以被定义为 [None, 10] ，是因为它的第一维为 None 与 x\_data 具有相同理由，而第二维为 10 是因为我们总共有 0~9 共 10 种类别的数字。

**（2）构建第一层网络**



**①权值 w 的构建**：

W\_conv1 = weight\_variable([5, 5, 1, 32])

传入的数组表示 w 的形状，其实就定义了该层卷积核的大小和数量：

每个卷积核的大小为 5x5

输入通道数为 1，因为用的图片是灰度图。如果是用不带透明通道的 rgb 彩色图该值就设为 3，如果在带了透明通道的 rgba 彩色图该值就设为 4

该层输出通道数为 32，即该层有 32 个卷积核。对于隐藏层中每个卷积层中的卷积核大小如何的确定，最靠谱的方法是用一些公开的网络模型，是经过反复尝试论证出来的，较为可靠。

**②偏置量 b 的构建**：

b\_conv1 = bias\_variable([32])

b 的大小和卷积核个数相对应就行了。什么意思呢？就是每个卷积核和输入卷积后，再加上一个偏置量就好。wx+b

**③改变输入数据的形状**：

x\_image = tf.reshape(self.x\_data, [-1, 28, 28, 1])

这行代码作用是改变输入数据的形状，让它可以和我们的卷积核进行卷积。因为输入的数据 x\_data 是一个 Nx784 的张量，所以需要先变为 M x 28 x 28 x 1 的形状才能进行运算。

第一个维度的 -1 表示大小待定，优先满足后面 3 个维度，最后再计算第一个维度的大小。也就是这样的：N x 784 / (28 x 28 x 1) 。

第二、第三维度实际上表示每张输入图片的大小需改为 28 x 28。

第四维度表示通道数，灰度为 1，rgb 为 3，rgba 为 4 。

**④构建卷积并加上激活函数**：

h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)

包含了 3 个操作：

conv2d(x\_image, W\_conv1)，将输入与该层的所有卷积核进行卷积运算

conv2d(x\_image, W\_conv1) + b\_conv1，每次卷积后加上一个偏置量

tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)，最后加上一个激活函数，增加非线性的变换

此处使用的激活函数是比较流行的 ReLu 函数，它的的函数式：y = max(x, 0)

ReLu激活函数的好处在于，由于它在第一象限就是 x，所以能够大量的减少计算，从而加速收敛。同时它天生就能减小梯度消失发生的可能性，不过梯度爆炸还是可能会发生。

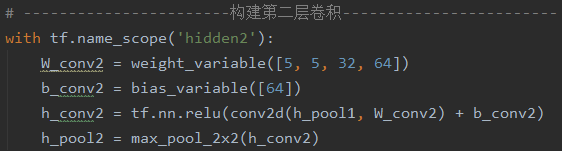
**⑤池化**：

h\_pool1 = max\_pool\_2x2(h\_conv1)

最后，再加上一个池化层，能够压缩权值数量，减小最后模型的大小，提高模型的泛化性。

这里使用了一个 2x2 的 max\_pooling，且步长取 1。max\_pooling 实际就是取一个 2x2 张量中的最大值，这样能够过滤掉一些并不是很重要的噪声。

**（3）构建第二层网络**



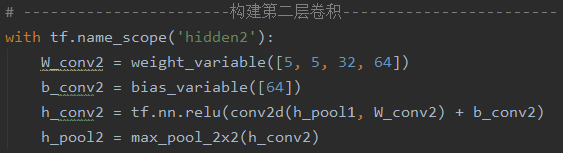
第二层和第一层构建的套路基本是一样的。

需要注意第二层中，w 的输入通道数为上一层最后的输出通道数，也就是 hpool1 的输出，这里直接算出了是 32，因为上面已经定义好。

如果不确定的时候，可以通过这种方式来确定输入通道数：in = h\_pool1.get\_shape()[-1].value

[-1] 表示不管 h\_pool1 的形状如何，都取它最后一维的大小。

**（4）构建全链接层**



**①定义 w 和 b**

W\_fc1 = weight\_variable([7 \* 7 \* 64, 1024])

上面这行代码，将fc1 的 w 形状设定为 [7 \* 7 \* 64, 1024]。第一个维度大小为 7 \* 7 \* 64，因为经过前面一层的卷积层的池化层后，输出的就是一个 7 \* 7 \* 64 的张量，所以这里的输入就是上一层的输出。嗯，这一点在整个神经网络里都是这样。

第二个维度大小为 1024。这是我们可以自己随便设定的，它表示该全链接层的神经元个数，数量越多计算耗时越长，但是数量过少，对前面提取出来的特征的分类效果又不够好。

因此，我们的全链接层拥有的特征数量就是 7 x 7 x 64 x 1024。数量还是比较惊人的。

b\_fc1 = bias\_variable([1024])

同样，b 的数量需要对应于 w 的最后一个维度，也就是一个神经元对应一个偏置量b。

**②变形输入**

h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7 \* 7 \* 64])

为了能和上面定义的全链接层特征张量相乘，需要把输入的张量变形。其实对于每一个 7 x 7 x64 的输入张量而言，就是将它们展平成一个一维的向量。

第一个维度取 -1 同上面提到过的意思一样，最后确定这个维度。实际上就是最后一个池化层输出的数量。

**③构建线性函数，加上 ReLu 函数**

h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat, W\_fc1) + b\_fc1)

上面的代码还是按照 wx + b 的线性公式构建函数，然后加上 ReLu 函数增加非线性变化。

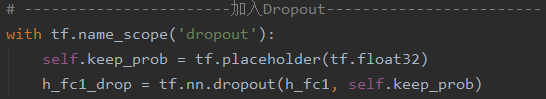
当一个来自最后一层池化层的 [7 x 7 x 64] 的输出经过全链接层后，就被平铺成了一个 [1 x 1024] 的向量。相当于把前面分散的特征全部链接在了一起，这也就是为什么说前面的卷积核是从局部观察，而全链接层是从全局的视野去观察的，因为它这一层整合所有前面的特征。整合了所有的特征，就可以进行后续的分类操作了。

**全连接总结：**

全链接层增加模型的复杂度，因为增加了很多神经元来扩充特征集。也因此，它有助于提升模型的准确率。

但随着特征数量的爆炸式增加，训练速度必然会变慢。而且如果全链接层设置的神经元数量过多，会出现过拟合的现象。所以，需要适当的设置，不能一味的贪多。

**（5）加入 Dropout**



在全链接层之后，往往会跟着 Dropout 操作。由于在神经网络中，神经元的个数非常爆炸，往往会产生过拟合的问题，特别是引入了全链接层这种操作之后。所以我们需要做些什么来让过拟合发生的概率减小一些。Dropout 就是一种很流行的方案。

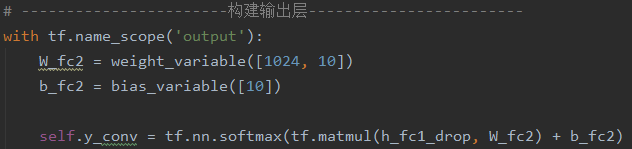
h\_fc1\_drop = tf.nn.dropout(h\_fc1, self.keep\_prob)

这行代码的第二个参数，可以动态的传入一个数值，表示每个神经元有多大的概率失效，其实就是不参与计算。

形象点描述就是这样一个过程。每个神经元进行运算前，都按照设置的 keep\_prob 概率决定它要不要参与计算。比如 keep\_prob=0.5 的话，表示每个神经元有 50% 的概率失效。

Dropout 操作能够一定程度上加快训练速度，同时降低过拟合的可能性。

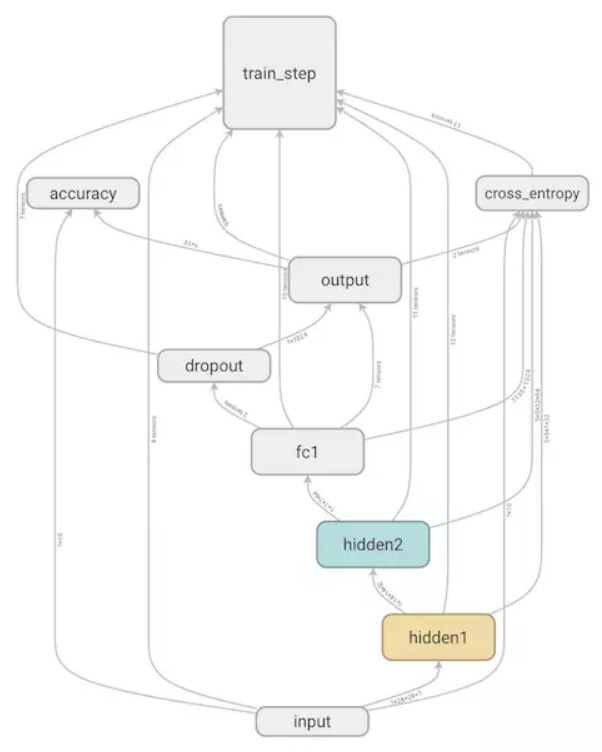
**（6）构建输出层**



输出层的函数结构仍然是 wx+b 的线性结构。

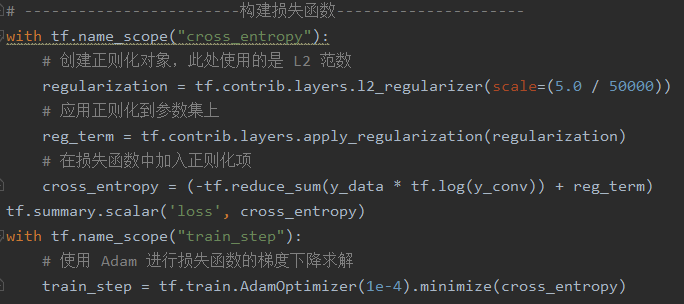
输出的 w 的形状。第一个维度不用说，就是上一层输出的结果，这里上一层是从全链接层输出出来 1024 个神经元。第二个维度是我们分类的总类目数，由于识别的是 0-9 的手写数字，所以总共有 10 个类别。

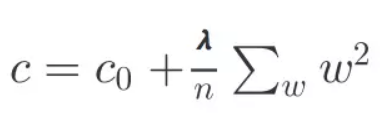
构建好线性函数后，在加入一个非线性的激活函数。在分类场景中， Softmax 是一个在输出层被广泛使用的激活函数。直接输出各个分类的概率。



**4.训练生成模型train.py**

**（1）构建损失函数**



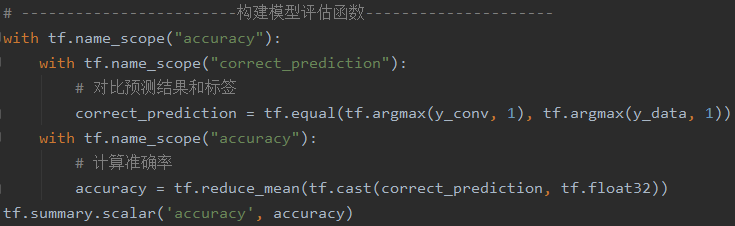
选择使用 L2范式，加入正则化后的 Loss 公式如下：

c0就是原本的损失函数部分，这里就是交叉熵，这部分又被称作是经验风险。后面的一部分就是L2正则化式了，它实际就是把每个权重平方后求和，然后除以 w 的数量，在乘一个重要度系数。正则化的部分又叫作结构风险，因为它是基于 w 计算出的一个数值，加在交叉熵上，从而每次增大交叉熵的值，也就是增大梯度，达到惩罚loss的效果。它一定程度上削弱了网络中特征值的作用，从而使模型的泛化性提高，也就能进一步的避免过拟合发生的可能。

第八行构建出一个 交叉熵 损失函数，第十二行对损失函数做梯度下降获得 tensor。

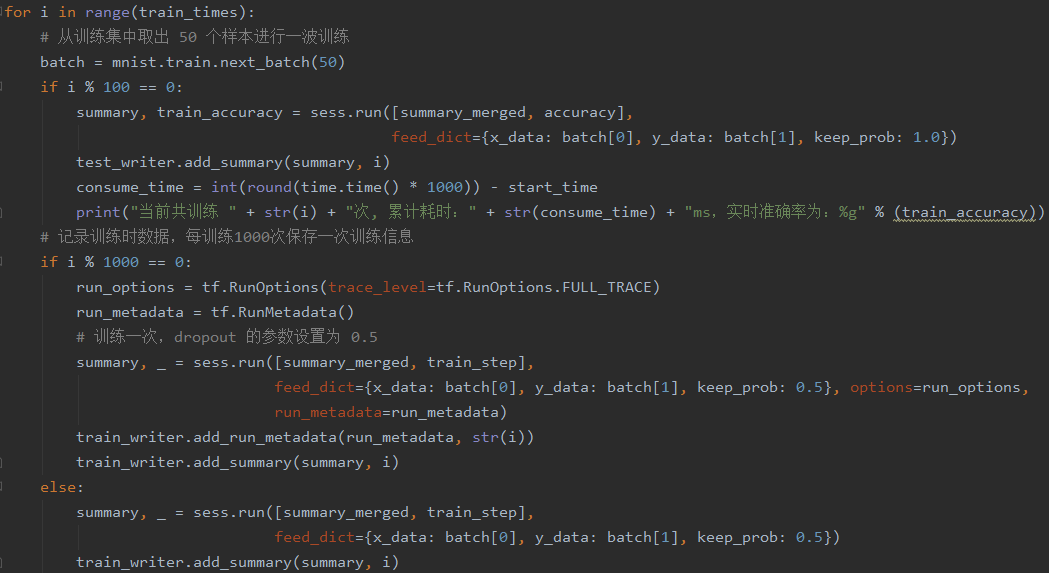
这里使用了 Adam优化算法 能够为不同的参数动态的计算不同的自适应学习率，这与 SGD 恒定不变的学习率有区别。这种优化算法使得各个参数的变化比较平稳，计算消耗的内存更小，收敛也会更快一点。

**（2）构建评估函数**



构建了一个用于评估模型准确率的 accuracy tensor。第一行实际就是比较了一下预测值和真实值，结果是一个 bool 向量，在第二行中转化为浮点数，求个平均就是准确率了。

**（3）开始训练**



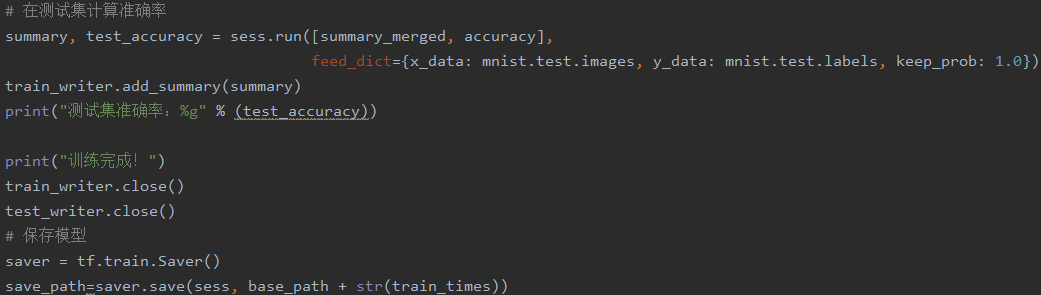
首先从训练集中随机的取出 50 个样本作为一次训练的输入。

因为训练集有60000个样本，训练一次太耗时了，

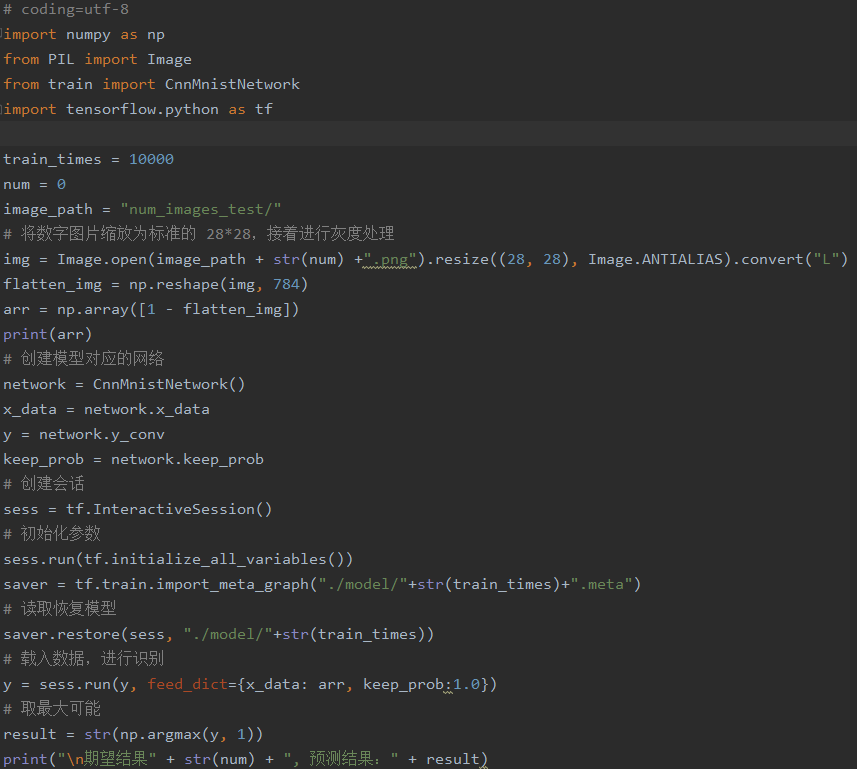
这么做理论上准确率没有全集训练高，但是也能达到 99% 的准确率，却能节省巨量的时间，这点理论上的准确率还是可以适当的舍弃的。

接着就调用 sess.run(train\_step) 开始一次训练，注意此处由于加了 dropout，所以每次 feed\_dict 中需要设定它的值。

最后使用测试集测试正确率并保存模型



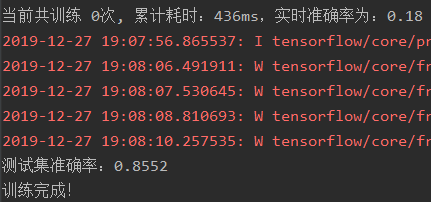
**5.测试手写图片test.py**

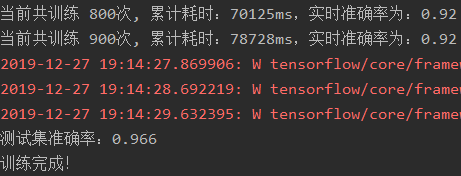


首先从指定目录中读取输入的测试图片，将其缩放为28\*28，然后创建模型对应的神经网络，读取保存的训练好的模型，最后使用模型进行预测。

结果分析：

分别展示训练100、1000、10000次时的正确率



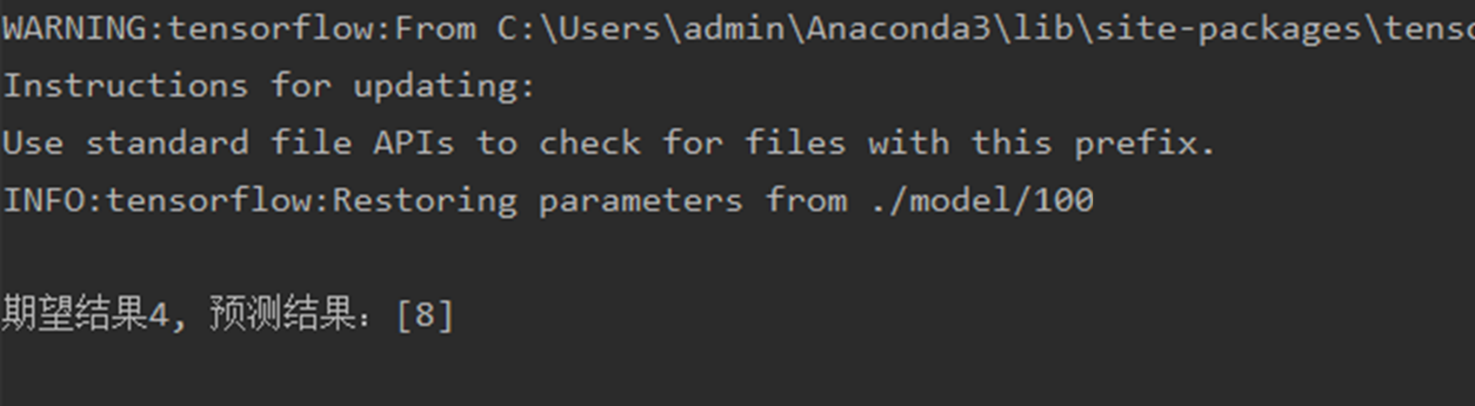




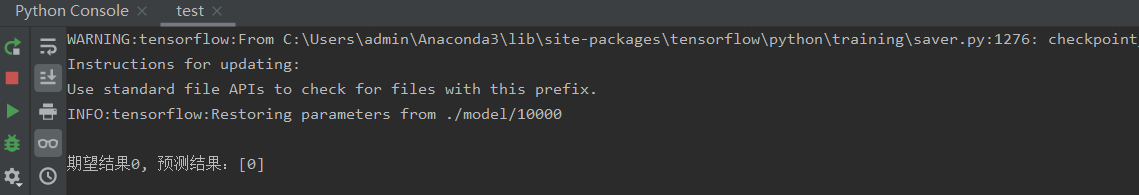
保存的模型：



当使用较小的训练次数时：识别出错



当使用较大的训练次数时：识别正确



## 六、实验体会和思考题

实验体会：

这次实验中主要要求使用python解决问题，与前一次的要求相比更加严格。在这次实验过程中，首先要求了解卷积神经网络集构造与其基本组成：卷积、池化、全连接、dropout、softmax输出，在实际编写代码过程中，我遇到的一个较为复杂的问题是保存训练模型与读取模型，总是会由于未给出正确的模型路径导致模型读取失败。

通过具体的实验代码编写，在课本知识的基础上我更加进一步的了解了相关操作的具体python实现，得到了很大的收获。

思考题：

**Learning Rate（学习率）**

学习率决定了权值更新的速度，设置得太大会使结果超过最优值，太小会使下降速度过慢。仅靠人为干预调整参数需要不断修改学习率。

**Weight decay（权值衰减率）**

在实际应用中，为了避免网络的过拟合，必须对价值函数（Cost function）加入一些正则项，在SGD中加入这一正则项对这个Cost function进行规范化：

公式基本思想就是减小不重要的参数对最后结果的影响，网络中有用的权重则不会受到Weight decay影响。在机器学习或者模式识别中，会出现overfitting，而当网络逐渐overfitting时网络权值逐渐变大，因此，为了避免出现overfitting,会给误差函数添加一个惩罚项，常用的惩罚项是所有权重的平方乘以一个衰减常量之和。其用来惩罚大的权值。

weightdecay（权值衰减）的使用既不是为了提高收敛精确度也不是为了提高收敛速度，其最终目的是防止过拟合。在损失函数中，weight decay是放在正则项（regularization）前面的一个系数，正则项一般指示模型的复杂度，所以weight decay的作用是调节模型复杂度对损失函数的影响，若weight decay很大，则复杂的模型损失函数的值也就大。

**Momentum （激励值）**

动量来源于牛顿定律，基本思想是为了找到最优加入“惯性”的影响，当误差曲面中存在平坦区域，SGD就可以更快的学习。

momentum是梯度下降法中一种常用的加速技术。对于一般的SGD，其表达式为

其中β即momentum系数，通俗的理解上面式子就是，如果上一次的momentum（即v）与这一次的负梯度方向是相同的，那这次下降的幅度就会加大，所以这样做能够达到加速收敛的过程。

normalization（batch normalization）。batch normalization的是指在神经网络中激活函数的前面，将按照特征进行normalization，这样做的好处有三点：

1、提高梯度在网络中的流动。Normalization能够使特征全部缩放到[0,1]，这样在反向传播时候的梯度都是在1左右，避免了梯度消失现象。

2、提升学习速率。归一化后的数据能够快速的达到收敛。

3、减少模型训练对初始化的依赖

**Learning Rate Decay （学习衰减率）**

该参数是为了提高SGD寻优能力，具体就是每次迭代的时候减少学习率的大小。

## 附录（源代码及注释）

**为引用方便将function直接加入build.py&train.py**

**download.py**

from tensorflow.examples.tutorials.mnist import input\_data

mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)

print('MINST图像数据：', mnist.train.images)

print('训练集规模：', mnist.train.images.shape)

print('测试集规模：', mnist.test.images.shape)

print('验证集规模：', mnist.validation.images.shape)

import pylab

print('数据集的第一张图片：')

im = mnist.train.images[0]

im = im.reshape(-1, 28)

pylab.imshow(im)

pylab.show()

**build.py**

import tensorflow as tf

# 定义一个用于创建 权重 变量的函数

def weight\_variable(shape):

initial = tf.truncated\_normal(shape, stddev=0.1)

var = tf.Variable(initial)

# 记录每一个权重，因为后面要使用正则化

# 至于原因，后面具体再说

tf.add\_to\_collection(tf.GraphKeys.WEIGHTS, var)

return var

# 定义一个用于创建 偏置量 变量的函数

def bias\_variable(shape):

# 初始化值为0.1

initial = tf.constant(0.1, shape=shape)

return tf.Variable(initial)

# 构建卷积函数

# 该卷积核每次在长、宽上移动一个步长，padding采用"SAME"策略

def conv2d(x, W):

return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

# 该池化核大小为2x2，长、宽上步长为2，padding采用"SAME"策略

def max\_pool\_2x2(x):

return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

class CnnMnistNetwork:

def \_\_init\_\_(self):

# 创建占位tensor，用于装载数据

self.x\_data = tf.placeholder(tf.float32, [None, 784])

self.y\_data = tf.placeholder(tf.float32, [None, 10])

# -----------------------构建第一层卷积-----------------------

with tf.name\_scope('hidden1'):

W\_conv1 = weight\_variable([5, 5, 1, 32])

b\_conv1 = bias\_variable([32])

x\_image = tf.reshape(self.x\_data, [-1, 28, 28, 1])

h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)

h\_pool1 = max\_pool\_2x2(h\_conv1)

# -----------------------构建第二层卷积------------------------

with tf.name\_scope('hidden2'):

W\_conv2 = weight\_variable([5, 5, 32, 64])

b\_conv2 = bias\_variable([64])

h\_conv2 = tf.nn.relu(conv2d(h\_pool1, W\_conv2) + b\_conv2)

h\_pool2 = max\_pool\_2x2(h\_conv2)

# -----------------------构建密集(全)链接层------------------------

with tf.name\_scope('FC1'):

W\_fc1 = weight\_variable([7 \* 7 \* 64, 1024])

b\_fc1 = bias\_variable([1024])

h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7 \* 7 \* 64])

h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat, W\_fc1) + b\_fc1)

# -----------------------加入Dropout------------------------

with tf.name\_scope('dropout'):

self.keep\_prob = tf.placeholder(tf.float32)

h\_fc1\_drop = tf.nn.dropout(h\_fc1, self.keep\_prob)

# -----------------------构建输出层------------------------

with tf.name\_scope('output'):

W\_fc2 = weight\_variable([1024, 10])

b\_fc2 = bias\_variable([10])

self.y\_conv = tf.nn.softmax(tf.matmul(h\_fc1\_drop, W\_fc2) + b\_fc2)

**train.py**

# coding=utf-8

import build

import time

import tensorflow as tf

from tensorflow.examples.tutorials.mnist import input\_data

from function import \*

from build import CnnMnistNetwork

train\_times = 1000

base\_path = "model/"

save\_path = base\_path + str(train\_times) + "/"

# 读取数据

mnist = input\_data.read\_data\_sets("MNIST\_data/", one\_hot=True)

# 创建网络

network = CnnMnistNetwork()

x\_data = network.x\_data

y\_data = network.y\_data

y\_conv = network.y\_conv

keep\_prob = network.keep\_prob

# ----------------------功能函数--------------------------

# 定义一个用于创建 权重 变量的函数

def weight\_variable(shape):

initial = tf.truncated\_normal(shape, stddev=0.1)

var = tf.Variable(initial)

# 记录每一个权重，因为后面要使用正则化

# 至于原因，后面具体再说

tf.add\_to\_collection(tf.GraphKeys.WEIGHTS, var)

return var

# 定义一个用于创建 偏置量 变量的函数

def bias\_variable(shape):

# 初始化值为0.1

initial = tf.constant(0.1, shape=shape)

return tf.Variable(initial)

# 构建卷积函数

# 该卷积核每次在长、宽上移动一个步长，padding采用"SAME"策略

def conv2d(x, W):

return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

# 该池化核大小为2x2，长、宽上步长为2，padding采用"SAME"策略

def max\_pool\_2x2(x):

return tf.nn.max\_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# --------------------------------------------------------

# ------------------------构建损失函数---------------------

with tf.name\_scope("cross\_entropy"):

# 创建正则化对象，此处使用的是 L2 范数

regularization = tf.contrib.layers.l2\_regularizer(scale=(5.0 / 50000))

# 应用正则化到参数集上

reg\_term = tf.contrib.layers.apply\_regularization(regularization)

# 在损失函数中加入正则化项

cross\_entropy = (-tf.reduce\_sum(y\_data \* tf.log(y\_conv)) + reg\_term)

tf.summary.scalar('loss', cross\_entropy)

with tf.name\_scope("train\_step"):

# 使用 Adam 进行损失函数的梯度下降求解

train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)

# ------------------------构建模型评估函数---------------------

with tf.name\_scope("accuracy"):

with tf.name\_scope("correct\_prediction"):

# 对比预测结果和标签

correct\_prediction = tf.equal(tf.argmax(y\_conv, 1), tf.argmax(y\_data, 1))

with tf.name\_scope("accuracy"):

# 计算准确率

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

tf.summary.scalar('accuracy', accuracy)

# 创建会话

sess = tf.InteractiveSession()

# 合并 summary

summary\_merged = tf.summary.merge\_all()

train\_writer = tf.summary.FileWriter(save\_path + "graph/train", sess.graph)

test\_writer = tf.summary.FileWriter(save\_path + "graph/test")

start\_time = int(round(time.time() \* 1000))

# 初始化参数

sess.run(tf.initialize\_all\_variables())

for i in range(train\_times):

# 从训练集中取出 50 个样本进行一波训练

batch = mnist.train.next\_batch(50)

if i % 100 == 0:

summary, train\_accuracy = sess.run([summary\_merged, accuracy],

feed\_dict={x\_data: batch[0], y\_data: batch[1], keep\_prob: 1.0})

test\_writer.add\_summary(summary, i)

consume\_time = int(round(time.time() \* 1000)) - start\_time

print("当前共训练 " + str(i) + "次, 累计耗时：" + str(consume\_time) + "ms，实时准确率为：%g" % (train\_accuracy))

# 记录训练时数据，每训练1000次保存一次训练信息

if i % 1000 == 0:

run\_options = tf.RunOptions(trace\_level=tf.RunOptions.FULL\_TRACE)

run\_metadata = tf.RunMetadata()

# 训练一次，dropout 的参数设置为 0.5

summary, \_ = sess.run([summary\_merged, train\_step],

feed\_dict={x\_data: batch[0], y\_data: batch[1], keep\_prob: 0.5}, options=run\_options,

run\_metadata=run\_metadata)

train\_writer.add\_run\_metadata(run\_metadata, str(i))

train\_writer.add\_summary(summary, i)

else:

summary, \_ = sess.run([summary\_merged, train\_step],

feed\_dict={x\_data: batch[0], y\_data: batch[1], keep\_prob: 0.5})

train\_writer.add\_summary(summary, i)

# 在测试集计算准确率

summary, test\_accuracy = sess.run([summary\_merged, accuracy],

feed\_dict={x\_data: mnist.test.images, y\_data: mnist.test.labels, keep\_prob: 1.0})

train\_writer.add\_summary(summary)

print("测试集准确率：%g" % (test\_accuracy))

print("训练完成！")

train\_writer.close()

test\_writer.close()

# 保存模型

saver = tf.train.Saver()

save\_path=saver.save(sess, base\_path + str(train\_times))

**test.py**

# coding=utf-8

import numpy as np

from PIL import Image

from train import CnnMnistNetwork

import tensorflow.python as tf

train\_times = 1000

num = 0

image\_path = "num\_images\_test/"

# 将数字图片缩放为标准的 28\*28，接着进行灰度处理

img = Image.open(image\_path + str(num) +".png").resize((28, 28), Image.ANTIALIAS).convert("L")

flatten\_img = np.reshape(img, 784)

arr = np.array([1 - flatten\_img])

print(arr)

# 创建模型对应的网络

network = CnnMnistNetwork()

x\_data = network.x\_data

y = network.y\_conv

keep\_prob = network.keep\_prob

# 创建会话

sess = tf.InteractiveSession()

# 初始化参数

sess.run(tf.initialize\_all\_variables())

saver = tf.train.import\_meta\_graph("./model/"+str(train\_times)+".meta")

# 读取恢复模型

saver.restore(sess, "./model/"+str(train\_times))

# 载入数据，进行识别

y = sess.run(y, feed\_dict={x\_data: arr, keep\_prob:1.0})

# 取最大可能

result = str(np.argmax(y, 1))

print("\n期望结果" + str(num) + ", 预测结果：" + result)