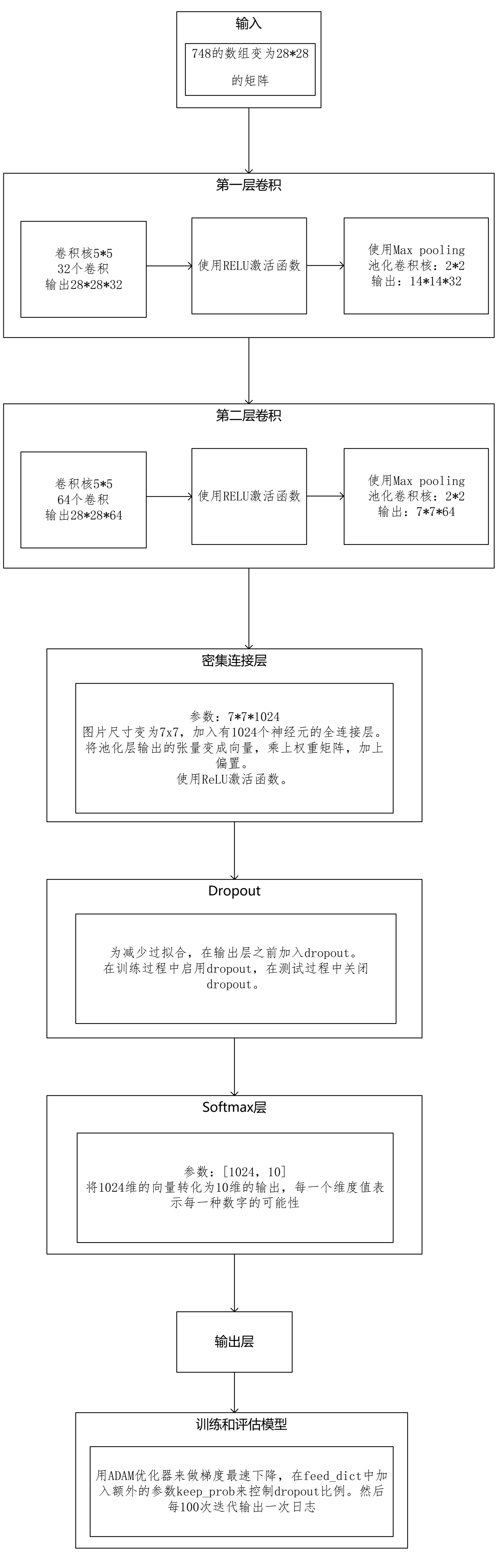
手写数字识别实验报告

2016010308 计64 嵇天颖

一、模型结构图及流程分析

（一）模型结构图（training1.py）



（二）流程分析（training1.py）

(1)权重初始化

- 为了创建这个模型，需要创建大量的权重和偏置项。
- 为了打破对称性以及避免0梯度，模型中权重在初始化时应该加入少量的噪声。使用一个较小的正数来初始化偏置项，以避免神经元节点输出恒为0的问题。
- 为了不在建立模型的时候反复做初始化操作，定义两个函数用于初始化：

```
#权重初始化
def weight_variable(shape):
    initial = tf.truncated_normal(shape,stddev=0.1)
    return tf.Variable(initial)
def bias_variable(shape):
    initial = tf.constant(0.1, shape = shape)
    return tf.Variable(initial)
```

(2)卷积和池化

- 卷积使用1步长，0边距的模板，保证输出和输入是同一个大小；
- 池化用简单传统的2x2大小的模板做max pooling。

```
#卷积和池化
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1], padding='SAME')
```

(3)第一层卷积

- 第一层卷积由一个卷积和一个池化完成。
- 卷积在每个5x5的patch中算出32个特征。卷积的权重张量形状是[5，5，1，32]，前两个维度是patch的大小，接着是输入的通道数目，最后是输出的通道数目。而对于每一个输出通道都有一个对应的偏置量。
- 将x变成一个4d向量，其第2、第3维对应图片的宽、高，最后一维代表图片的颜色通道数，灰度图颜色通道数为1。
- 将x_image和权值向量进行卷积，加上偏置项，然后应用ReLU激活函数，最后进行max pooling池化。

```
#第一层卷积
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])


#将x变成一个4d向量，其第2、第3维对应图片的宽、高，最后一维灰度图颜色通道数为1
x_image = tf.reshape(x,[-1,28,28,1])

#将x_image和权值向量进行卷积，加上偏置项，然后应用ReLU激活函数，最后进行池化
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

(4)第二层卷积

为了构建更深的网络，第二层中，每个5x5的patch会得到64个特征。

```
#第二层卷积
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)
```

(5)密集连接层

- 将图片尺寸减小到7x7，加入一个有1024个神经元的全连接层，用于处理整个图片。
- 将池化层输出的张量reshape成一些向量，乘上权重矩阵，加上偏置，再对其使用ReLU激活函数。

```
#密集连接层
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
```

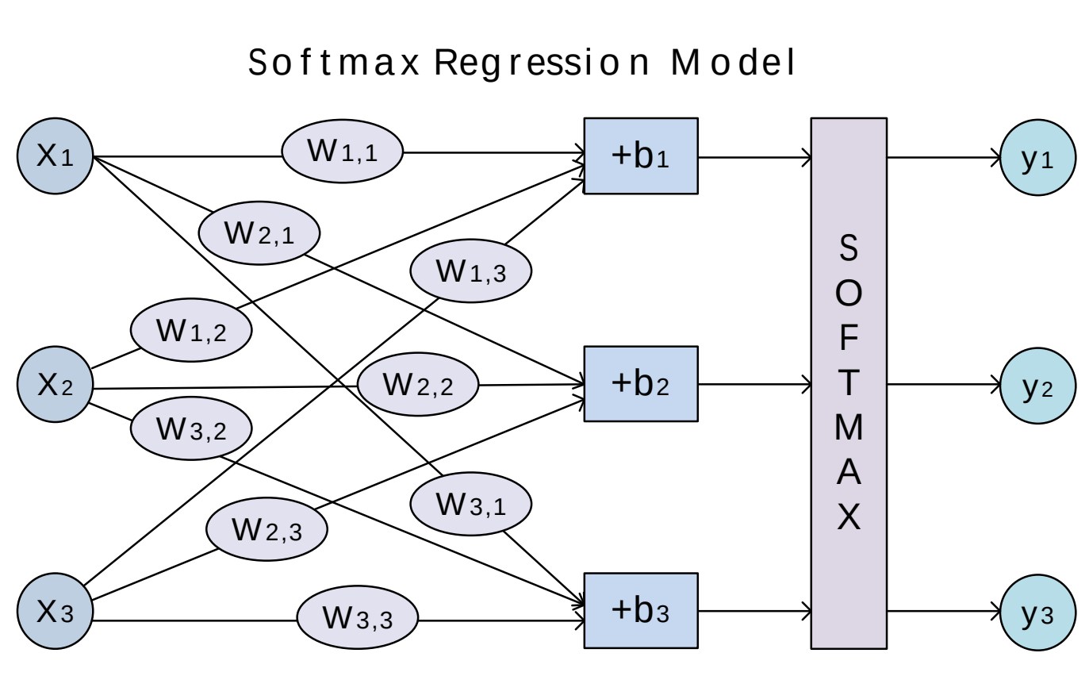
(6)Dropout

- 为了减少过拟合，故在输出层之前加入dropout。
- 用一个placeholder来代表一个神经元的输出在dropout中保持不变的概率。这样可以在训练过程中启用dropout，在测试过程中关闭dropout。

```
#Dropout
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

(7)输出层

- 在输出层的最后，利用Softmax回归模型，添加一个softmax层，给不同对象分配概率。



- 将1024维向量变为10维向量输出，每一维表示一种数字的概率

```
#输出层
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

(8)训练与测试模型

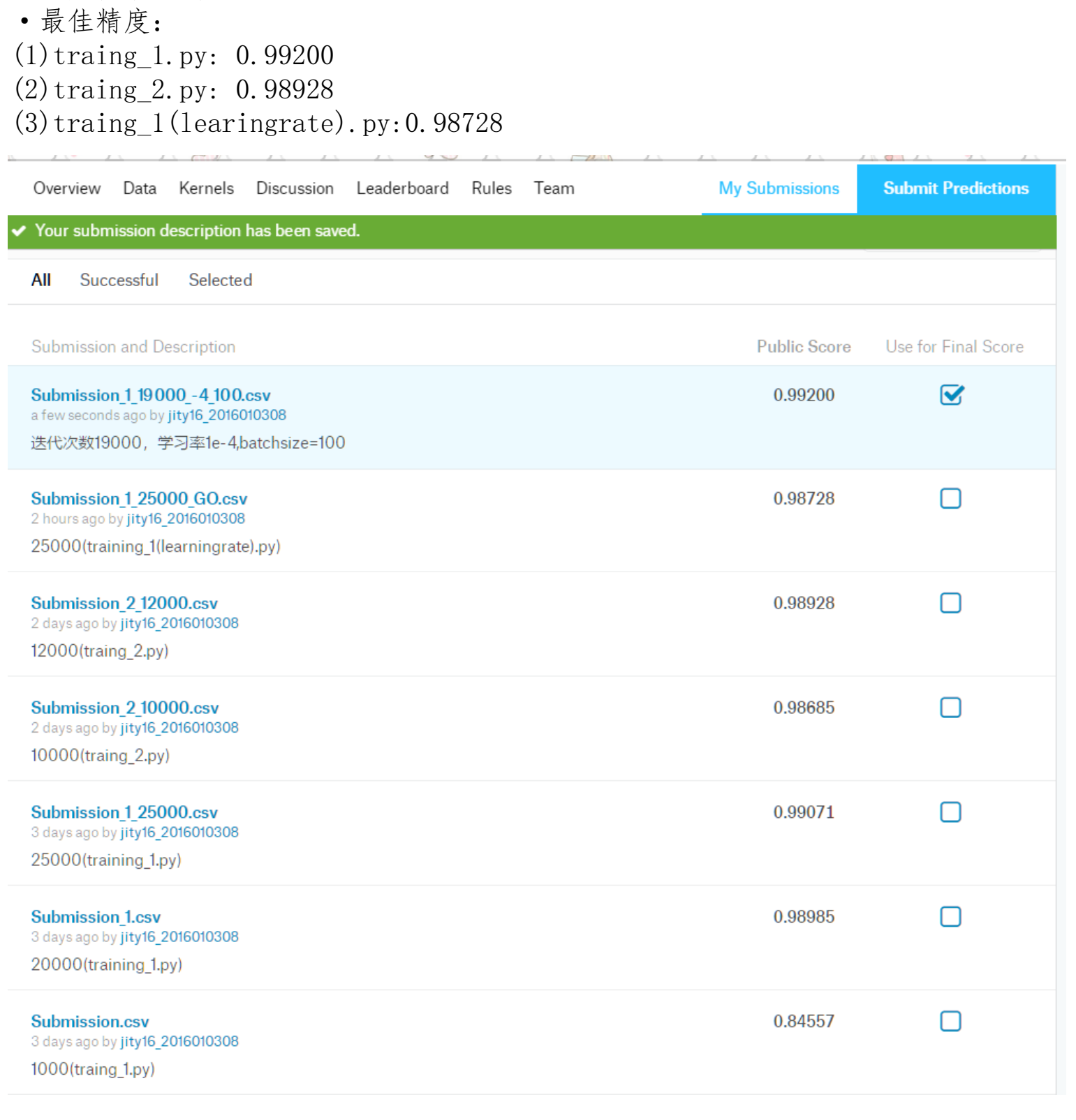
- 使用ADAM优化器来做梯度最速下降，在feed_dict中加入额外的参数keep_prob来控制dropout比例。
- 每100次迭代输出一次日志。

```
#训练和评估模型
MAX_ITERATION = 25000;
BATCH_SIZE = 50;

cross_entropy = -tf.reduce_sum(y_*tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))

sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

print('Starting Training Process...')
for i in range(MAX_ITERATION):
  batch = train.next_batch(BATCH_SIZE)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
        x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))

  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
print("Finishing Training Process...")

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: validation.images, y_: validation.labels, keep_prob: 1.0}))
```

二、实验结果

- 账号：jity16_2016010308
- 提交次数：5次
- 最佳精度：

(1)traing_1.py：0.99200
(2)traing_2.py：0.98928
(3)traing_1(learingrate).py:0.98728



✓ Your submission description has been saved.

Submission and Description	Public Score	Use for Final Score
Submission_1_19000_-4_100.csv a few seconds ago by jity16_2016010308 迭代次数19000，学习率1e-4,batchsize=100	0.99200	☑
Submission_1_25000_GO.csv 2 hours ago by jity16_2016010308 25000(training_1(learningrate).py)	0.98728	☐
Submission_2_12000.csv 2 days ago by jity16_2016010308 12000(traing_2.py)	0.98928	☐
Submission_2_10000.csv 2 days ago by jity16_2016010308 10000(traing_2.py)	0.98685	☐
Submission_1_25000.csv 3 days ago by jity16_2016010308 25000(training_1.py)	0.99071	☐
Submission_1.csv 3 days ago by jity16_2016010308 20000(training_1.py)	0.98985	☐
Submission.csv 3 days ago by jity16_2016010308 1000(traing_1.py)	0.84557	☐

三、试比较实验中使用的不同参数效果，并分析原因

(1)学习方式的差别：

- training_1.py

```
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

- training_1(learningrate).py

```
LEARNING_RATE = 1e-4;
train_step = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(cross_entropy)
```

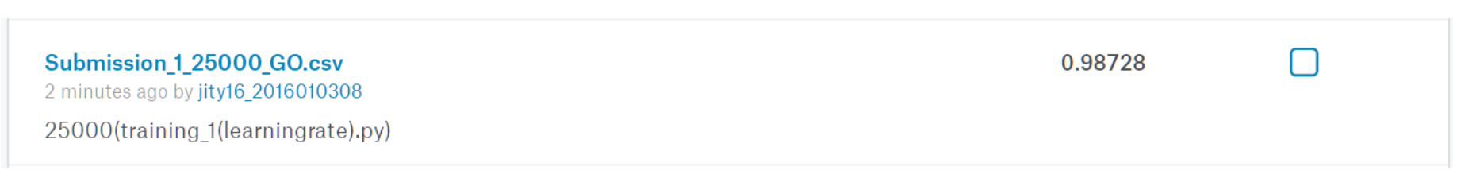
- **两个py文件采用了不同的学习率函数：**

- 参考：
https://stats.stackexchange.com/questions/184448/difference-between-gradientdescentoptimizer-and-adamoptimizer-tensorflow

- 我在上面网址的论坛中看到关于AdamOptimizer与GradientDescentOptimizer的孰优孰劣的讨论，于是我用了两个py文件来测试，这两分代码除了Optimizer优化器不同，其余都相同。

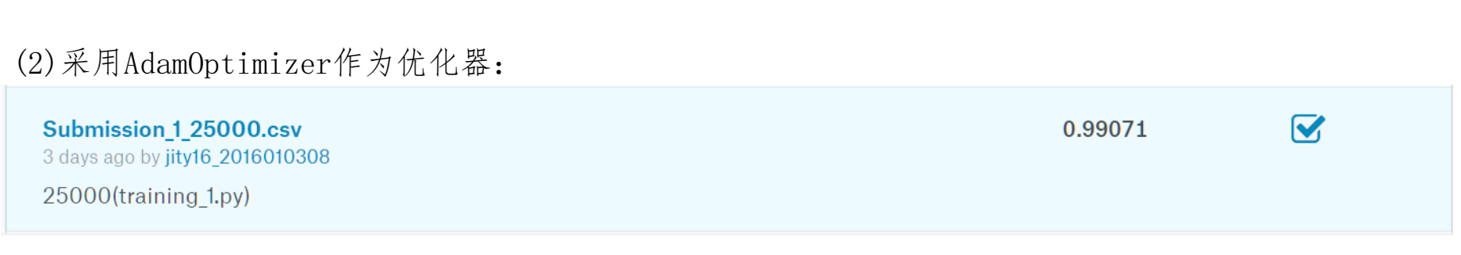
- 第一个文件参照了tensorflow的官网教程，使用了AdamOptimizer；第二个文件采用了算法更为简单的GradientDescentOptimizer来计算学习率。

结果如下：
(1)采用GradientDescentOptimizer作为优化器：



Submission and Description	Public Score	Use for Final Score
Submission_1_25000_GO.csv 2 minutes ago by jity16_2016010308 25000(training_1(learningrate).py)	0.98728	☐

(2)采用AdamOptimizer作为优化器：



Submission and Description	Public Score	Use for Final Score
Submission_1_25000.csv 3 days ago by jity16_2016010308 25000(training_1.py)	0.99071	☑

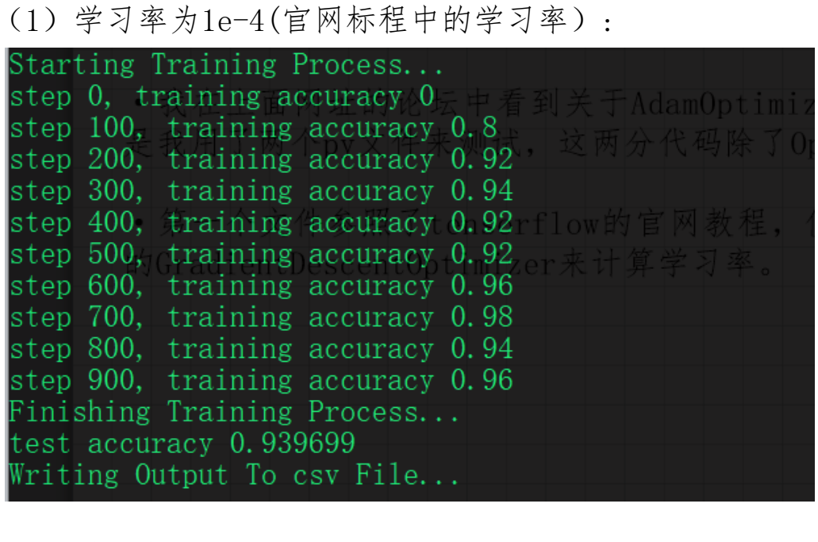
结果发现：AdamOptimizer作为优化器的时候，正确率更高。

原因：tf.train.AdamOptimizer 优化器使用了Kingma and Ba's Adam 算法来控制学习率，要比简单版本的 tf.train.GradientDescentOptimizer优化器好。在每个训练步骤中，AdamOptimizer需要对每个参数执行更多的计算，以保持移动平均值和方差，并计算缩放梯度；并且为每个参数保留更多的状态(约为三倍的大小)。该模型存储每个参数的平均值和方差。

(2) 学习率的差别

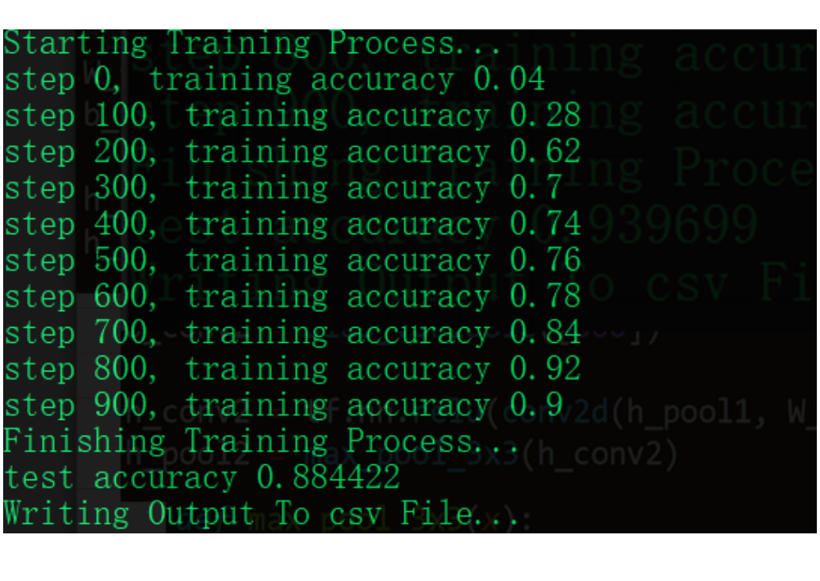
- **采用不同学习率，为了节省时间，这里运行时都采取较小的1000次迭代：**

（1）学习率为1e-4(官网标程中的学习率)：



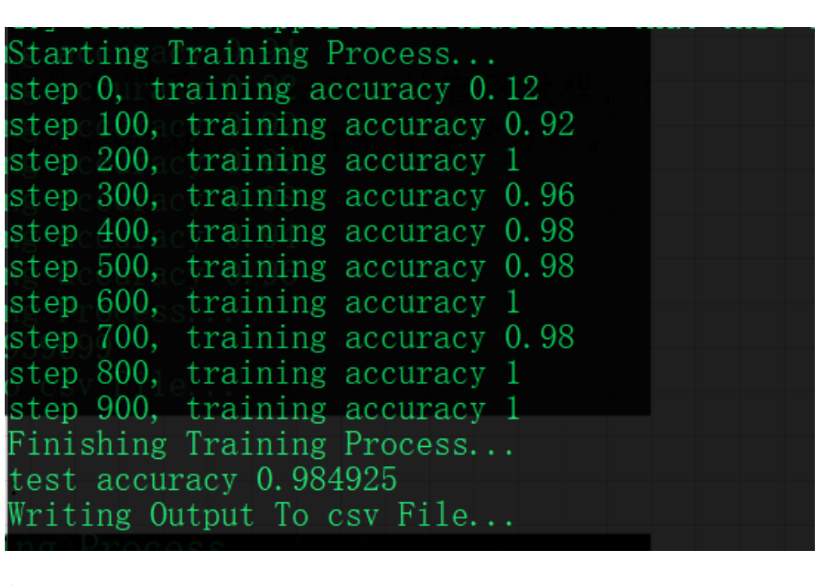
```
Starting Training Process...
step 0, training accuracy 0
step 100, training accuracy 0.8
step 200, training accuracy 0.92
step 300, training accuracy 0.94
step 400, training accuracy 0.92
step 500, training accuracy 0.92
step 600, training accuracy 0.96
step 700, training accuracy 0.98
step 800, training accuracy 0.94
step 900, training accuracy 0.96
Finishing Training Process...
test accuracy 0.939699
Writing Output To csv File...
```

（2）学习率为1e-5:



```
Starting Training Process...
step 0, training accuracy 0.04
step 100, training accuracy 0.28
step 200, training accuracy 0.62
step 300, training accuracy 0.7
step 400, training accuracy 0.74
step 500, training accuracy 0.76
step 600, training accuracy 0.78
step 700, training accuracy 0.84
step 800, training accuracy 0.92
step 900, training accuracy 0.9
Finishing Training Process...
test accuracy 0.884422
Writing Output To csv File...
```

(3) 学习率为1e-3:

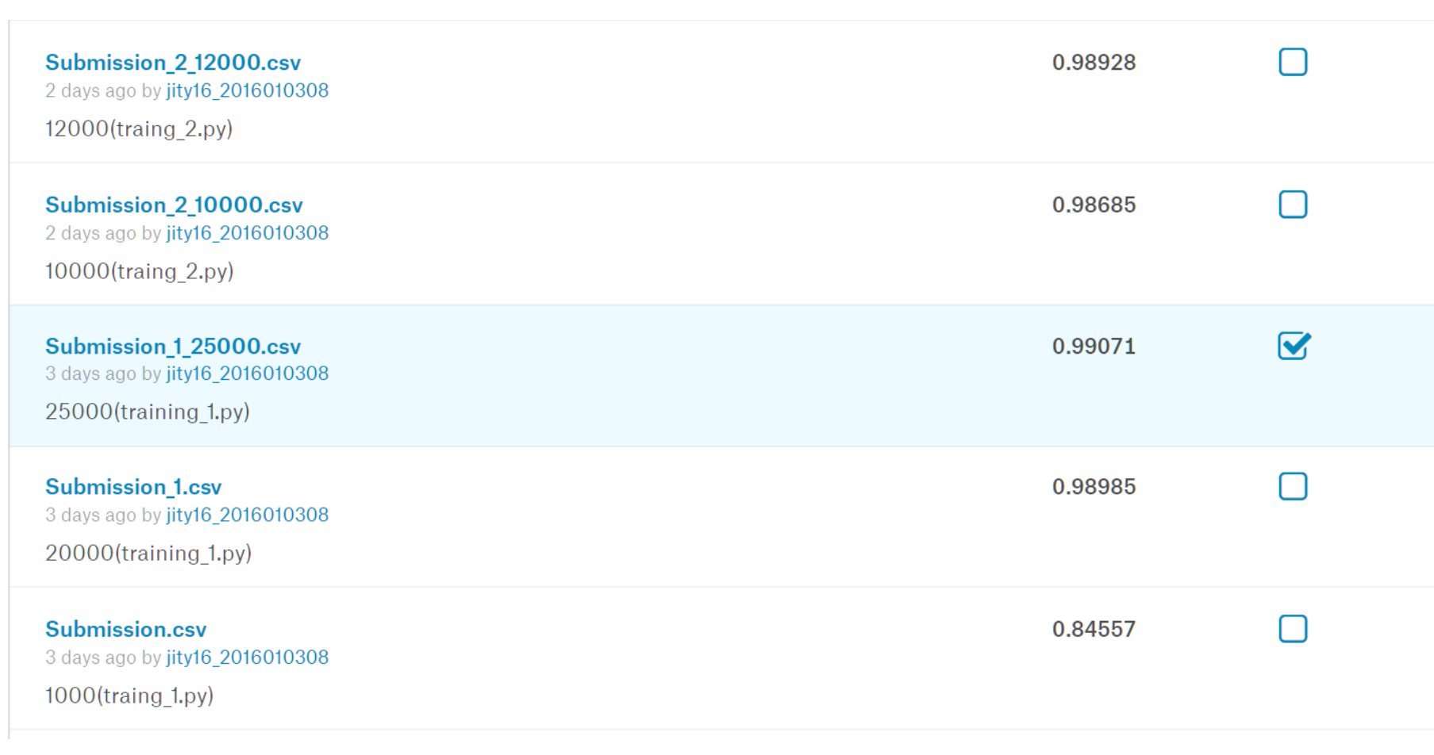


```
Starting Training Process...
step 0, training accuracy 0.12
step 100, training accuracy 0.92
step 200, training accuracy 1
step 300, training accuracy 0.96
step 400, training accuracy 0.98
step 500, training accuracy 0.98
step 600, training accuracy 1
step 700, training accuracy 0.98
step 800, training accuracy 1
step 900, training accuracy 1
Finishing Training Process...
test accuracy 0.984925
Writing Output To csv File...
```

结果：在学习率为1e-3时最大。
原因：
（1）但是这个结果与实际中学习率与正确率的一般关系相悖，是因为，实际中，学习率和迭代次数关系很大，所以并不能表明这是最佳的学习率。这个实验只能表现出学习率对正确率是有显著的影响的。
（2）学习率控制着基于损失梯度调整神经网络权值的速度。学习率越小，沿着损失梯度下降的速度越慢。学习率较小的时候，可以尽量避免错过局部最优解，但它也意味着我们要花更多时间来收敛。

(3)迭代次数的差别：

下图中各次提交命名为（Submission_文件（training_1.py或者training_2.py)_迭代次数）



Submission and Description	Public Score	Use for Final Score
Submission_2_12000.csv 2 days ago by jity16_2016010308 12000(traing_2.py)	0.98928	☐
Submission_2_10000.csv 2 days ago by jity16_2016010308 10000(traing_2.py)	0.98685	☐
Submission_1_25000.csv 3 days ago by jity16_2016010308 25000(training_1.py)	0.99071	☑
Submission_1.csv 3 days ago by jity16_2016010308 20000(training_1.py)	0.98985	☐
Submission.csv 3 days ago by jity16_2016010308 1000(traing_1.py)	0.84557	☐

结果：在其他参数相同的条件下，迭代次数的增加可以提高正确率。
原因：迭代是重复反馈的动作，神经网络中我们希望通过迭代进行多次的训练以到达所需的目标或结果。每一次迭代得到的结果都会被作为下一次迭代的初始值。一个迭代 = 一个正向通过+一个反向通过

(4)批尺寸的差别：

- 批量大小将决定我们一次训练的样本数目。batch_size将影响到模型的优化程度和速度。
- batchsize 的正确选择是为了在内存效率和内存容量之间寻找最佳平衡。

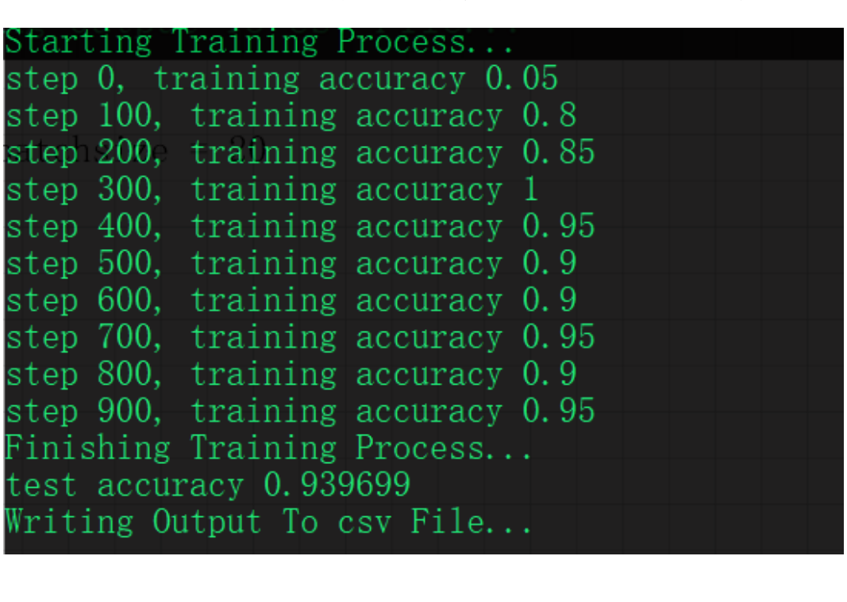
（1）batchsize = 50 (官网标程值)

```
Starting Training Process...
step 0, training accuracy 0
step 100, training accuracy 0.8
step 200, training accuracy 0.92
step 300, training accuracy 0.94
step 400, training accuracy 0.92
step 500, training accuracy 0.92
step 600, training accuracy 0.96
step 700, training accuracy 0.98
step 800, training accuracy 0.94
step 900, training accuracy 0.96
Finishing Training Process...
test accuracy 0.939699
Writing Output To csv File...
```

（2）batchsize = 100(运行速度很慢)

```
Starting Training Process...
step 0, training accuracy 0.1
step 100, training accuracy 0.82
step 200, training accuracy 0.89
step 300, training accuracy 0.92
step 400, training accuracy 0.94
step 500, training accuracy 0.96
step 600, training accuracy 0.95
step 700, training accuracy 0.95
step 800, training accuracy 0.96
step 900, training accuracy 0.96
Finishing Training Process...
test accuracy 0.959799
Writing Output To csv File...
```

（3）batchsize = 20(很快运行完毕，结果和50的时候差不多)



```
Starting Training Process...
step 0, training accuracy 0.05
step 100, training accuracy 0.8
step 200, training accuracy 0.85
step 300, training accuracy 1
step 400, training accuracy 0.95
step 500, training accuracy 0.9
step 600, training accuracy 0.9
step 700, training accuracy 0.95
step 800, training accuracy 0.9
step 900, training accuracy 0.95
Finishing Training Process...
test accuracy 0.939699
Writing Output To csv File...
```

- 适当的增加Batchsize 的优点：

1. 通过并行化提高内存利用率。
2. 单次epoch的迭代次数减少，提高运行速度。
3. 适当的增加Batch_Size，梯度下降方向准确度增加，训练震动的幅度减小。

(三) 更换连接方式

在training_2.py中修改了卷积层和池化层。修改处如下：

```
#卷积和池化
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1], padding='SAME')

def max_pool_3x3(x):
    return tf.nn.max_pool(x, ksize=[1, 3, 3, 1], strides=[1, 3, 3, 1], padding='SAME')
```

- 第一层卷积核变为4*4，生成20 个隐含层。

```
#第一层卷积
W_conv1 = weight_variable([4, 4, 1, 20])
b_conv1 = bias_variable([20])
```

- 第二个卷积核仍然用5*5 的，但是卷积层变为800。池化采用3*3，这样，得到5*5*800 的张量。

```
#第二层卷积
W_conv2 = weight_variable([5, 5, 20, 800])
b_conv2 = bias_variable([800])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_3x3(h_conv2)
```

- 修改密集连接层，更改7*7*64的输入为5*5*800的输入。

```
#密集连接层
W_fc1 = weight_variable([5 * 5 * 800, 150])
b_fc1 = bias_variable([150])

h_pool3_flat = tf.reshape(h_pool2, [-1, 5 * 5 * 800])
h_fc1 = tf.nn.relu(tf.matmul(h_pool3_flat, W_fc1) + b_fc1)
```

- 将5*5*800 个元素转化为150 个元素，得到输出层。

```
#输出层
W_fc2 = weight_variable([150, 10])
b_fc2 = bias_variable([10])

y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

结果：在迭代次数为12000次的时候，正确率与traing_1.py迭代为20000次的大致持平，但是由于训练版本二耗时很大，所以没有进行更高迭代次数的训练。

四、问题思考

1）实验训练什么时候停止是最合适的？简要陈述你的实现方式，并试分析固定迭代次数与通过验证集调整等方法的优缺点。

固定迭代次数优点是在验证集和测试集上都会呈现出稳定的结果，不会出现较大的偏差；缺点是固定迭代次数难以寻找到最适合的迭代次数。通过验证集调整可能会出现为了验证准确率最高而调整到某一个迭代次数，但是实际测试时可能反而正确率会下降，但是不断调整可以找到适合的点。

2）实验参数的初始化是怎么做的？不同的方法适合哪些地方？（现有的初始化方法为零均值初始化，高斯分布初始化，正交初始化等）

- 我的做法：权重在初始化时加入少量的噪声来打破对称性以及避免0梯度。由于使用的是ReLU神经元，因此比较好的做法是用一个较小的正数来初始化偏置项，以避免神经元节点输出恒为0的问题。
- 正交初始化：理想的情况是，特征值绝对值为1。则无论步数增加多少，梯度都在数值计算的精度内。把转移矩阵初始化为单位正交阵，可以避免在训练一开始就发生梯度爆炸/消失现象。
- 高斯分布初始化：在数据符合正态分布时较好。
- 零均值初始化：消除指标之间的量纲影响，进行数据标准化处理，以解决数据指标之间的可比性。

3） 过拟合是深度学习常见的问题，有什么方法可以方式训练过程陷入过拟合。

- 为了减少过拟合，我们在输出层之前加入dropout。
- 我们用一个placeholder来代表一个神经元的输出在dropout中保持不变的概率。这样我们可以在训练过程中启用dropout，在测试过程中关闭dropout。
- TensorFlow的tf.nn.dropout操作除了可以屏蔽神经元的输出外，还会自动处理神经元输出值的scale。所以用dropout的时候可以不用考虑Scale。

4） 试分析CNN（卷积神经网络）相对于全连接神经网络的优点 。

- CNN的特征提取层参数是通过训练数据学习得到的，所以其避免了人工特征提取，而是从训练数据中进行学习；其次同一特征图的神经元共享权值，减少了网络参数，这也是卷积网络相对于全连接网络的一大优势。共享局部权值这一特殊结构更接近于真实的生物神经网络，使CNN在图像处理、语音识别领域有着独特的优越性，另一方面权值共享同时降低了网络的复杂性，且多维输入信号（语音、图像）可以直接输入网络的特点避免了特征提取和分类过程中数据重排的过程。
- CNN是一种特殊的深层的神经网络模型，它的特殊性体现在两个方面，一方面它的神经元的连接是非全连接的，另一方面同一层中某些神经元之间的连接的权重是共享的。它的非全连接和权值共享的网络结构使之更类似于生物神经网络，降低了网络模型的复杂度，减少了权值的数量。

五、心得体会

本次实验是第一次尝试tensorflow和cnn，收获很大。

training_1.py是在官网标程的指导下完成的，training_2.py是对卷积层和池化层连接方式的改动。

官网上的标程算法准确率很高，采用其他方式进行的连接的正确率相对于标程并没有很大的提高，training_2.py的修改可能在高迭代次数时超过training_1.py，但是由于training_2.py要求的存储空间很大，可能迭代次数过大时会出现无法训练的状况。由于training_2.py要求的时间成本较高，所以并没有运行迭代20000次的情况。

对参数的修改和比较过程中，收获很多，真切地感受到了参数的影响，在阅读一些资料的过程中，逐渐了解了影响的原因。

参考资料：

（1）http://www.tensorfly.cn/
（2）https://www.jianshu.com/p/6766fbcd43b9
（3）https://blog.csdn.net/mydear_11000/article/details/53197891
（4）https://blog.csdn.net/yhhyhhyhhyhh/article/details/54574021