实验一神经网络报告

江辉--171250506

2020-3-10

南京大学

软件学院

目录

一、实验目的

本次实验使用 python 构建一个 CNN 神经网络完成 MNIST“手写体识别”任务，

二、内容描述

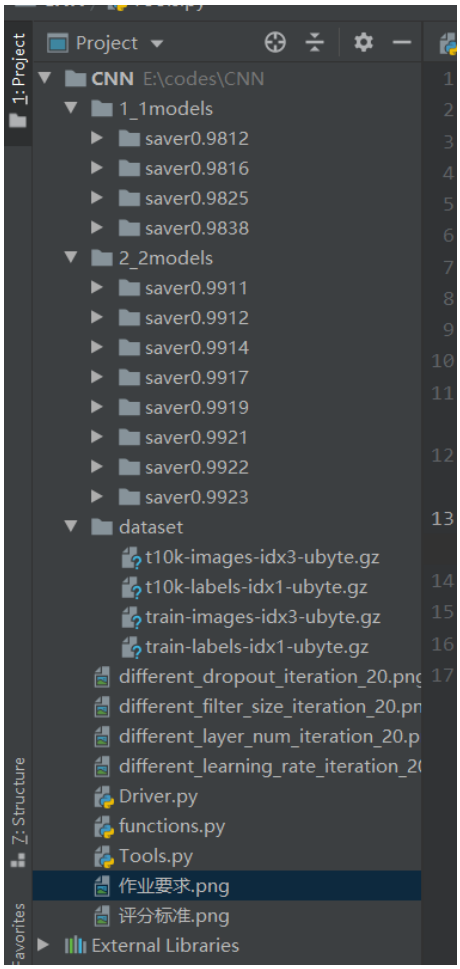
本程序用 python 编写，使用 TensorFlow 框架实现 CNN 神经网络。预先定义好各个层卷积层的卷积核大小、池化层函数和全连接层的网络大小，运用 Adam 优化算法来学习，使用 dropout 来避免过拟合。并且通过对比试验观察不同参数、结构的网络执行的效率和准确率等情况。

1、环境及配置

本项目推荐使用 pycharm 打开
最好配好 TensorFlow-gpu+ cuDNN+CUDA，这样可以用 gpu 训练
我的 gpu 配置是 GTX1050，大概是 3-4s 左右一次迭代，一次训练约 1-2min，(4w 训练集)，不同环境运行时可能不同，仅供参考

2、项目目录描述

1_1models 存放了使用 1 层卷积层 + 1 层全连接层训练的结果
2_2models 存放了使用 2 层卷积层 + 2 层全连接层训练的结果
saver 后面的数字是对应存档的测试集准确率
dataset 存放了 MNIST 的数据文件
然后下面是我用 pyplot 画出来的几个对比试验结果图



Driver.py 里写了 main 函数和几个对照试验的参数，我把对应的对照试验都封装成函数了，可以直接在 main 函数里调用运行
functions.py 里写了整个 model 的所有函数，所有模型相关的函数入口和参数都以注释的形式写在这个文件里了
Toos.py 里有我写的用来测试是否使用 gpu 的函数，和一个我写的用 matplotlib.pyplot 将结果可视化的函数，如果需要跑程序的话可以参考使用
difference_xxxxx.png 是我完成各个对照试验的结果可视化后的折线图

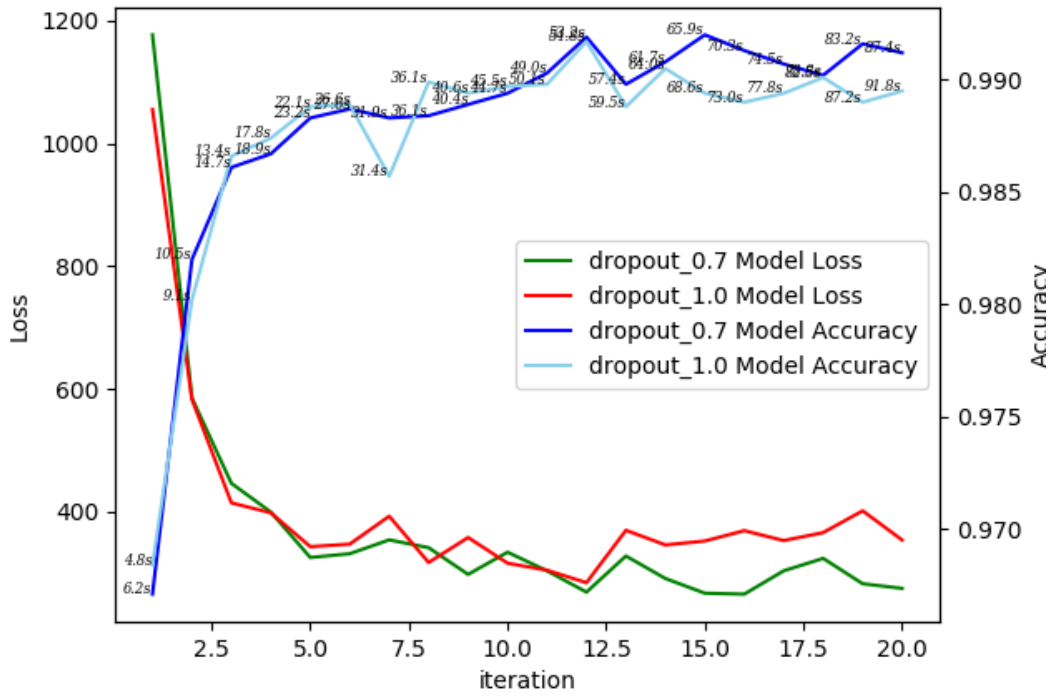
三、 运行用例和截图

1、不同的 KEEP_PROB 值的对照试验

右图为输入的用例数据，下图为相应展示的结果折线图，折线图里点上的数字是运行的时间(s)，可以看到，**keep_prob 为 0.7 和为 1.0 的情况**都可以在测试集上达到 **99%**以上的正确率，且运行时间上没有很大差别（一次迭代约 5s，keep_prob 为 1.0 时平均运行时间稍长一些），但是当不使用 dropout（即 keep_prob=1.0）时，训练的效果会有一定波动，且平均准确率略低于使用了 dropout 的情况(keep_prob=0.7)，而且**不使用 dropout 的 loss 曲线**在迭代次数多了之后有一点向上升高的趋势，开始变得**过拟合**，而使用了 dropout 的则没有发现这个趋势

```
def different_dropout():
    """
    不同的keep_prob值的对照试验
    采用2层卷积 + 2层全连接的结构
    第一个是keep_prob = 0.7的情况
    第二个是keep_prob = 1.0的情况

    """
    conv1_shape = [5, 5, 1, 20]
    conv2_shape = [5, 5, 20, 50]
    fc1_shape = [7 * 7 * 50, 500]
    fc2_shape = [500, 10]
    learning_rate = 0.001
    keep_prob = 0.7  # -------关键参数--0.7
    max_iteration = 20
    result1 = _2_2_model(conv1_shape, conv2_shape, fc1_shape, fc2_shape,
                         learning_rate=learning_rate, keep=keep_prob,
                         max_iteration=max_iteration, save_result=True)
    conv1_shape = [5, 5, 1, 20]
    conv2_shape = [5, 5, 20, 50]
    fc1_shape = [7 * 7 * 50, 500]
    fc2_shape = [500, 10]
    learning_rate = 0.001
    keep_prob = 1.0  # -------关键参数--1.0
    max_iteration = 20
    result2 = _2_2_model(conv1_shape, conv2_shape, fc1_shape, fc2_shape,
                         learning_rate=learning_rate, keep=keep_prob,
                         max_iteration=max_iteration, save_result=True)
    plot(result1, result2, "dropout_0.7", "dropout_1.0", "different_dropout")
```

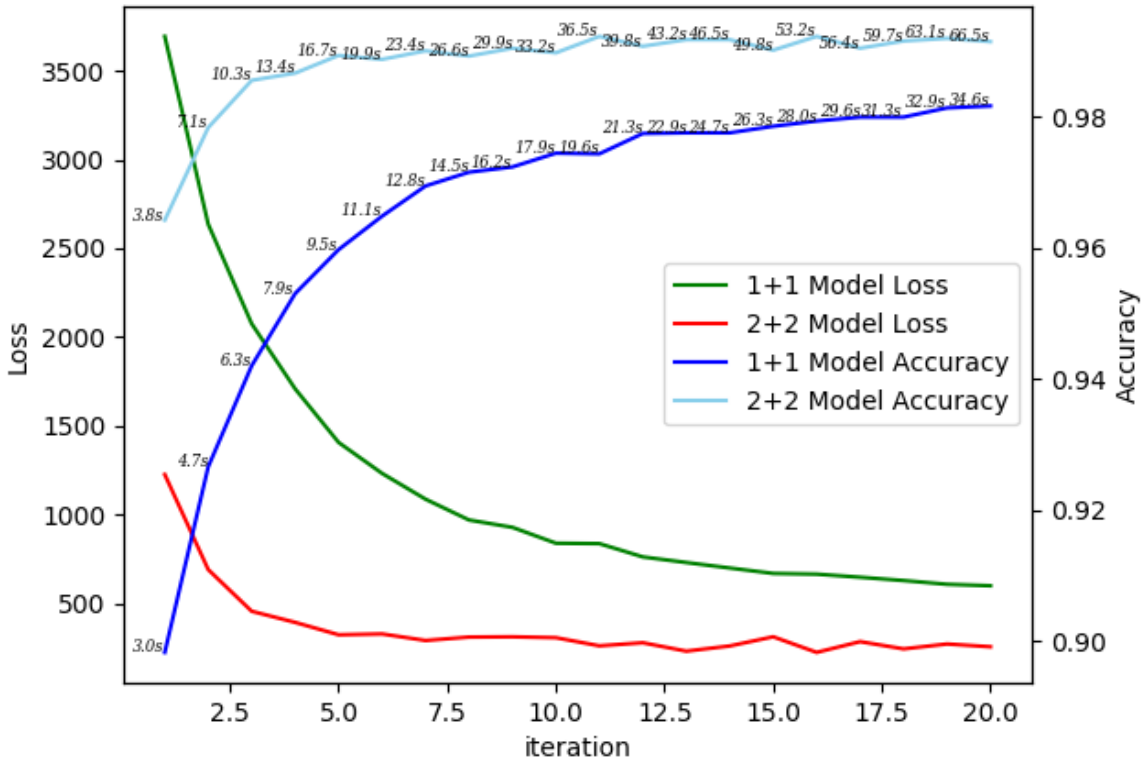


2、不同的网络层结构的对照试验

右图为输入的用例数据，下图为相应展示的结果折线图，折线图里点上的数字是运行的时间(s)，可以看到，当采用 **1 层卷积 + 1 层全连接**的结构时结果缓慢上升，接近稳定在 **98%**附近，而采用 **2 层卷积 + 2 层全连接**的结构则很快就稳定在了 **99%**的准确率。

运行时间上有较大差别，**1 层卷积 + 1 层全连接** 20 次迭代总时间 34s，而 **2 层卷积 + 2 层全连接**则花费了 66s，是前者的 2 倍，因此准确率和时间需要作出权衡

```python
def different_layer_num():
    """
    不同的网络层结构的对照试验
    第一个是采用1层卷积 + 1层全连接的结构的情况
    第二个是采用2层卷积 + 2层全连接的结构的情况
    """
    conv1_shape = [3, 3, 1, 20]      # -------关键参数--卷积层1
    fc1_shape = [14 * 14 * 20, 10]   # -------关键参数--全连接层1
    learning_rate = 0.001
    keep_prob = 0.7
    max_iteration = 20
    result1 = _1_1_model(conv1_shape, fc1_shape,
                         learning_rate=learning_rate, keep=keep_prob,
                         max_iteration=max_iteration, save_result=True)
    conv1_shape = [5, 5, 1, 20]      # -------关键参数--卷积层1
    conv2_shape = [3, 3, 20, 50]     # -------关键参数--卷积层2
    fc1_shape = [7 * 7 * 50, 500]    # -------关键参数--全连接层1
    fc2_shape = [500, 10]            # -------关键参数--全连接层2
    learning_rate = 0.001
    keep_prob = 0.7
    max_iteration = 20
    result2 = _2_2_model(conv1_shape, conv2_shape, fc1_shape, fc2_shape,
                         learning_rate=learning_rate, keep=keep_prob,
                         max_iteration=max_iteration, save_result=True)
    plot(result1, result2, "1+1", "2+2", "different_layer_num")
```

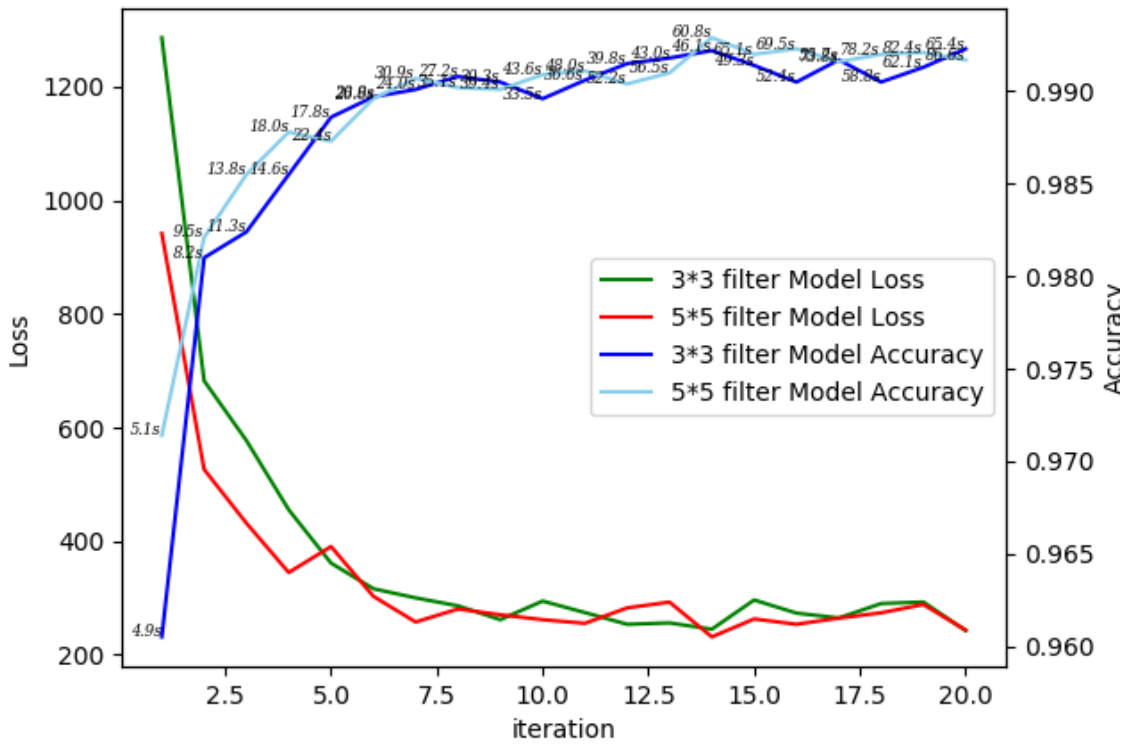


3、不同的卷积核大小的对照试验

右图为输入的用例数据，下图为相应展示的结果折线图，折线图里点上的数字是运行的时间(s)，可以看到，采用 **3*3** 的卷积核在刚开始时效率会稍微比用 **5*5** 的卷积核差一些，但是在几次迭代后两者就没有什么差别了，最终都达到了 99% 的准确率，不过 3*3 的收敛稳定性稍差一些。

在**运行时间上**，**3*3** 的卷积核在 20 次迭代总用时 **65s**，而 **5*5** 的卷积核用时 **86s**，因此在这个对比条件下效率和准确率也是负相关的关系。

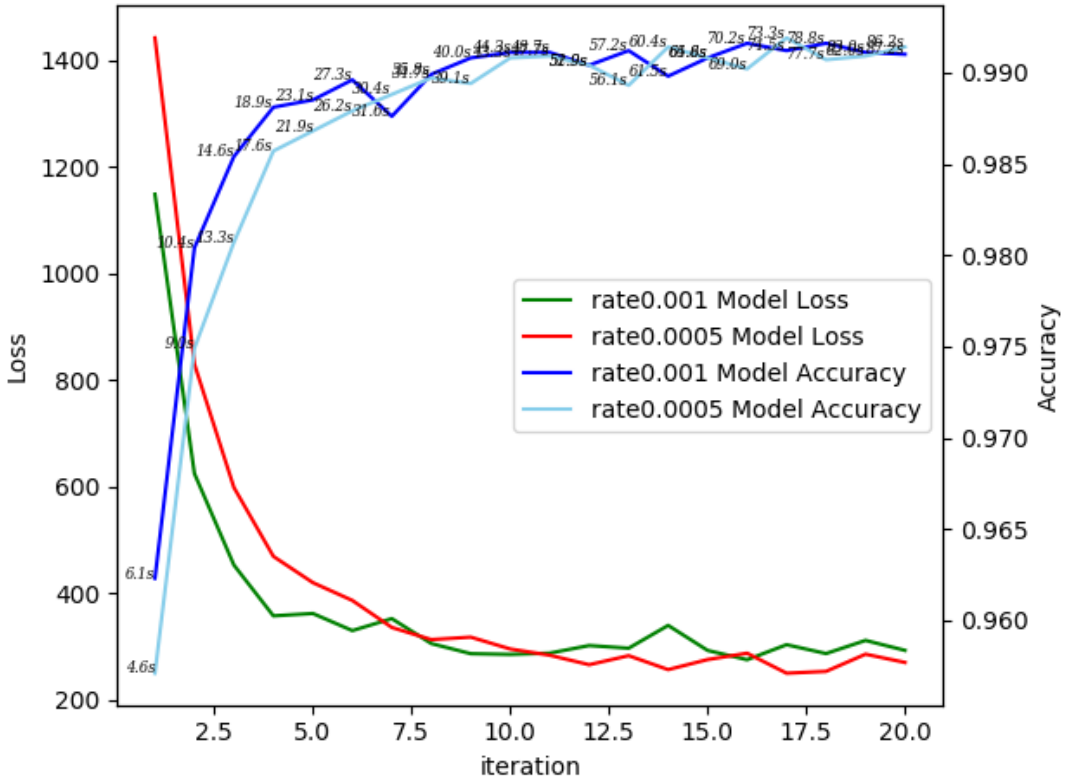
```
def different_filter_size():
    """
    不同的卷积核大小的对照试验
    第一个是采用 3 * 3 的卷积核的情况
    第二个是采用 5 * 5 的卷积核的情况
    """
    conv1_shape = [3, 3, 1, 20]     # -------关键参数--卷积核，3 * 3
    conv2_shape = [3, 3, 20, 50]    # -------关键参数--卷积核，3 * 3
    fc1_shape = [7 * 7 * 50, 500]
    fc2_shape = [500, 10]
    learning_rate = 0.001
    keep_prob = 0.7
    max_iteration = 20
    result1 = _2_2_model(conv1_shape, conv2_shape, fc1_shape, fc2_shape,
                         learning_rate=learning_rate, keep=keep_prob,
                         max_iteration=max_iteration, save_result=True)
    conv1_shape = [5, 5, 1, 20]     # -------关键参数--卷积核，5 * 5
    conv2_shape = [5, 5, 20, 50]    # -------关键参数--卷积核，5 * 5
    fc1_shape = [7 * 7 * 50, 500]
    fc2_shape = [500, 10]
    learning_rate = 0.001
    keep_prob = 0.7
    max_iteration = 20
    result2 = _2_2_model(conv1_shape, conv2_shape, fc1_shape, fc2_shape,
                         learning_rate=learning_rate, keep=keep_prob,
                         max_iteration=max_iteration, save_result=True)
    plot(result1, result2, "3*3 filter", "5*5 filter",
"different_filter_size")
```



4、不同的学习速率大小的对照试验

右图为输入的用例数据，下图为相应展示的结果折线图，折线图里点上的数字是运行的时间(s)，可以看到，采用 **0.0005 的学习速率**和**采用 0.001 的学习速率**在刚开始的收敛速度上有一点区别，但在经过几次迭代后两者几乎就没有区别了，即使在运行时间上两者也没有区别。我试过 0.01 的学习率，报错了，在某处计算时出现了 nan，这也说明在学习率合适的情况下，小学习率只是影响刚开始的速率，这使得学习效率下降

```
def different_learning_rate():
    """
    不同的学习速率大小的对照试验
    第一个是采用 0.001 的速率的情况
    第二个是采用 0.0005 的速率的情况
    """
    conv1_shape = [5, 5, 1, 20]
    conv2_shape = [5, 5, 20, 50]
    fc1_shape = [7 * 7 * 50, 500]
    fc2_shape = [500, 10]
    learning_rate = 0.001   # -------关键参数--学习率，0.001
    keep_prob = 0.7
    max_iteration = 2
    result1 = _2_2_model(conv1_shape, conv2_shape, fc1_shape, fc2_shape,
                         learning_rate=learning_rate, keep=keep_prob,
                         max_iteration=max_iteration, save_result=True)
    conv1_shape = [5, 5, 1, 20]
    conv2_shape = [5, 5, 20, 50]
    fc1_shape = [7 * 7 * 50, 500]
    fc2_shape = [500, 10]
    learning_rate = 0.0005  # -------关键参数--学习率，0.0005
    keep_prob = 0.7
    max_iteration = 2
    result2 = _2_2_model(conv1_shape, conv2_shape, fc1_shape, fc2_shape,
                         learning_rate=learning_rate, keep=keep_prob,
                         max_iteration=max_iteration, save_result=True)
    plot(result1, result2, "0.001", "rate0.0005", "different_learning_rate")
```



四、 思路方法

详细实现在六、过程分析描述中，也可以在 Driver 代码中执行 custom()方法，然后进入 terminal 输入 tensorboard --logdir=log，进入 TensorBoard 查看流程图

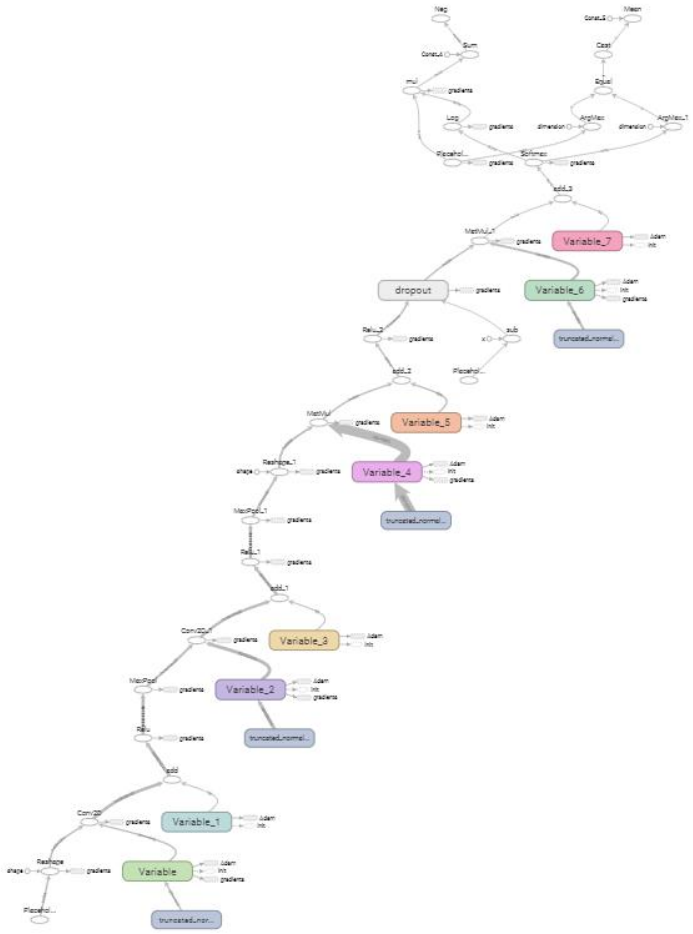
1、 首先通过 TensorFlow 提供的数据集读取函数读取数据

2、 对数据集进行 batch=400 的分割

3、 构建 LeNet-5 结构的 CNN 网络，即 2 层(卷积层+池化层) + 2 层全连接层

4、 对中间层采用 relu 激活，对输出层采用 softmax 激活

5、 使用 adam 优化学习算法进行后向传播学习参数



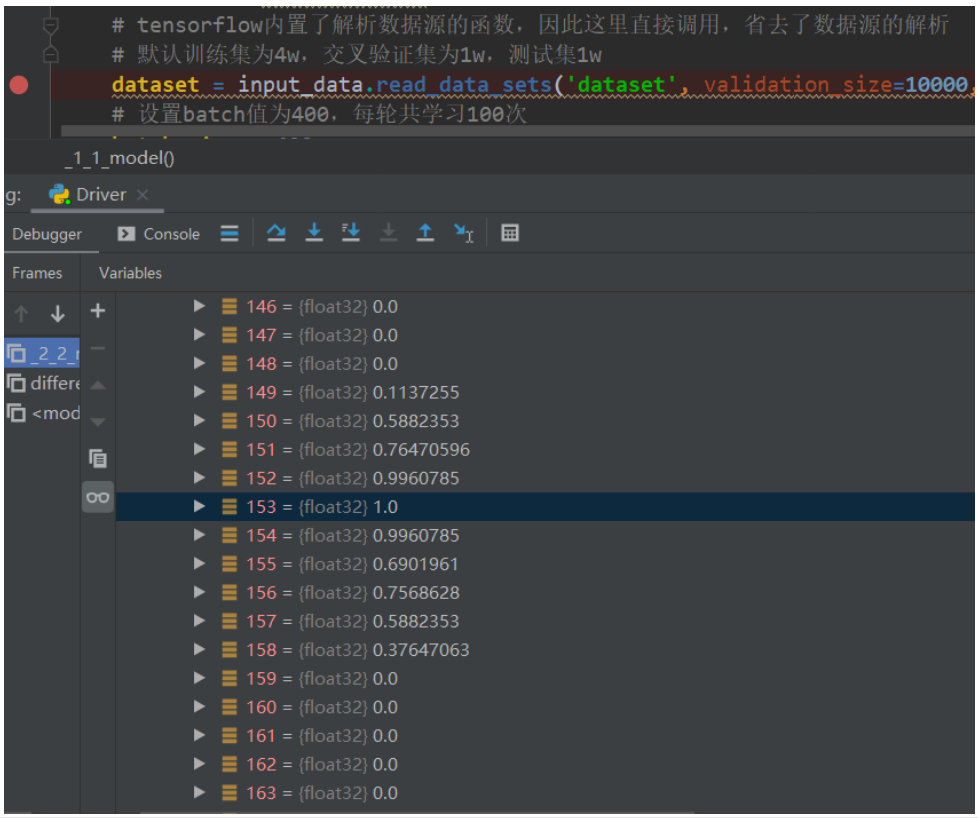
五、 假设

假设输入的数据已经去噪和归一化，这也是我在载入 MNIST 数据集后简单观测到的结果

六、过程分析描述

1、数据预处理

TensorFlow 已经提供了读取 MNIST 数据的方法，因此我直接调用了它的方法。我本来想对数据进行去噪、归一化等操作，但是发现里面数据已经是归一化的数据，且看了几张图的数据之后我没有找到噪声点，所以就省去了这两个步骤。图中可以看到数组里的片段中数据∈[0, 1.0]，且非常集中，周围没有噪声



2、数据分割

MNIST 数据集里有 6w 数据，我选择训练集为 4w，交叉验证集 1w，测试集 1w

并以 batch=400 将其分割为 100 个小数据集，每次迭代将学习 100 次

```
50      # tensorflow内置了解析数据源的函数，因此这里直接调用，省去了数据源的解析
51      # 默认训练集为4w，交叉验证集为1w，测试集1w
52      dataset = input_data.read_data_sets('dataset', validation_size=10000
53      # 设置batch值为400，每轮共学习100次
54      batch_size = 400
55      n_batch = dataset.train.num_examples // batch_size
56
```

3、构造 CNN 网络层和训练

我主要采用的是 LeNet-5 结构，即 2 层(卷积层+池化层) + 2 层全连接层的 CNN 神经网络
使用 5 * 5 的卷积核，中间层使用 rule 激活，且每次卷积后进行 2 * 2 池化(步长为 2)，即缩小一半，相关的 shape 参数和初始化各层参数的方法会在后面说明

```
# ------------------卷积层----------------------------

# 初始化第一层卷积核 W 和第一层偏差值 b
W_conv1, b_conv1 = init_layer_variable(conv1_shape)
# 把输入的图片矩阵x和权值向量进行卷积，再加上偏置值得到Z
Z_conv1 = conv2d(x_image, W_conv1) + b_conv1
# 通过relu激活
A_conv1 = tf.nn.relu(Z_conv1)
# 进行max_pooling 池化层
A_pool1 = max_pool(A_conv1)

# 初始化第二层卷积核 W 和第二层偏差值 b
W_conv2, b_conv2 = init_layer_variable(conv2_shape)
# 把第一个池化层结果和权值向量进行卷积，再加上偏置值
Z_conv2 = conv2d(A_pool1, W_conv2) + b_conv2
# 通过relu激活
A_conv2 = tf.nn.relu(Z_conv2)
# 进行max_pooling 池化层
A_pool2 = max_pool(A_conv2)
```

经过两次卷积后的激活值进行 2 次全连接和 relu 激活，最后经过 softmax 函数输出 10 个预测值，并计算代价函数(交叉熵)，学习时采用 adam 优化算法，其实也可以用 momentum 或 RMSprop，不过 adam 已经是二者结合的方式，所以就不做多的对比测试了

```
# ----------------全连接层-------------------------

# 初始化第一个全连接层的权值并将池化层扁平化
W_fc1, b_fc1 = init_layer_variable(fc1_shape)
A_pool2_flat = tf.reshape(A_pool2, [-1, fc1_shape[0]])
# 求第一个全连接层的输出
Z_fc1 = tf.matmul(A_pool2_flat, W_fc1) + b_fc1
A_fc1 = tf.nn.relu(Z_fc1)
A_fc1_drop = tf.nn.dropout(A_fc1, keep_prob)

# 初始化第二个全连接层
W_fc2, b_fc2 = init_layer_variable(fc2_shape)

# 用softmax计算输出
prediction = tf.nn.softmax(tf.matmul(A_fc1_drop, W_fc2) + b_fc2)

# 计算交叉熵代价函数
cross_entropy = -tf.reduce_sum(y * tf.log(prediction))
# 用AdamOptimizer进行优化
train_step = tf.train.AdamOptimizer(learning_rate).minimize(cross_entropy)
# 结果存放在一个bool列表中(argmax函数返回一维张量中最大的值所在的位置)
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(prediction, 1))
# 求准确率，输出tf.cast将bool转换为float
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

4、结果保存和分析

每次迭代我都会保存数据，用以在后面通过图片打印出来方便查看和分析，主要保存的数据有**代价值、准确率、运行时间**，前面所有的分析和展示都是基于这 3 个数据指标

```
# 训练并输出
with tf.Session() as sess:
    result = [[], [], []]
    start_time = time.process_time()
    sess.run(tf.global_variables_initializer())
    acc = 0.
    for epoch in range(1, max_iteration + 1):
        for batch in range(n_batch):
            x_batch, y_batch = dataset.train.next_batch(batch_size)
            # 进行迭代训练
            sess.run(train_step,
                     feed_dict={x: x_batch, y: y_batch,
                                keep_prob: keep})
        # 测试数据计算出准确率
        loss = sess.run(cross_entropy,
                     feed_dict={x: dataset.test.images,
                                y: dataset.test.labels,
                                keep_prob: 1.0})
        acc = sess.run(accuracy,
                     feed_dict={x: dataset.test.images,
                                y: dataset.test.labels,
                                keep_prob: 1.0})
        end_time = time.process_time()
        result[0].append(loss)
        result[1].append(acc)
        result[2].append(end_time - start_time)
        print('Iter:%d, %s s, Loss=%f,Test Accuracy=%f'
              % (epoch, end_time - start_time, loss, acc))
    if save_result:
        tf.train.Saver().save(sess,
                     save_path="2_2models/saver" + str(acc) + "/model")
return result
```

七、 重要函数

1、卷积核、池化核、全连接参数的初始化

我将参数初始化封装了一个方法，只需要传入对应的 shape 即可获得随机的参数张量

```
def init_layer_variable(shape):
    """
    初始化权值 W 和偏差值 b
    :param shape: 如果是卷积核则格式为
        [height, weight, in_channel, out_channel]，否则为 [in, out]
    :return 标准差为0.1的初始化权值W和初始值为0.1的偏差值b
    """
    weight = tf.truncated_normal(shape, stddev=0.1)
    bias = tf.constant(0.1, shape=[shape[len(shape) - 1]])
    return tf.Variable(weight), tf.Variable(bias)
```

2、卷积计算

我默认使用步长为 1，1 的卷积方式
填充时采用了保留边界的 SAME 填充

```
def conv2d(input, filter, strides=None, padding="SAME"):
    """
    进行卷积计算
    :param input: 输入的张量
    :param filter: 卷积核(过滤器)
    :param strides: 步长 [batch, height, weight, channel]
    :param padding: 填充方式，默认为"SAME"
    :return: 卷积结果
    """
    if strides is None:
        strides = [1, 1, 1, 1]
    return tf.nn.conv2d(input, filter, strides=strides, padding=padding)
```

3、池化

我采用了 2 * 2 的池化核，2 为步长，即将卷积输出缩小为原来的一半

```
def max_pool(value, kernel_size=None, strides=None, padding="SAME"):
    """
    池化，默认将输入的宽高缩小到一半
    :param value: 输入
    :param kernel_size: 池化卷积核形状 [batch=1, height=2, weight=2, channel=1]
    :param strides: 步长 [batch=1, height=2, weight=2, channel=1]
    :param padding: 填充方式，默认为"SAME"
    :return: 输出
    """
    if strides is None:
        strides = [1, 2, 2, 1]
    if kernel_size is None:
        kernel_size = [1, 2, 2, 1]
    return tf.nn.max_pool(value, ksize=kernel_size, strides=strides, padding=padding)
```

八、核心参数

1、5*5 的卷积核

我在所有对照实验里主要采用 5 * 5 的卷积核，第一层输出 20 个通道，第二层输出 50 个通道

```
conv1_shape = [5, 5, 1, 20]      # -------关键参数--卷积核，5 * 5
conv2_shape = [5, 5, 20, 50]     # -------关键参数--卷积核，5 * 5
fc1_shape = [7 * 7 * 50, 500]
fc2_shape = [500, 10]
learning_rate = 0.001
keep_prob = 0.7
max_iteration = 20
```

2、2*2 的池化核

所有对照试验中的池化核都是采用默认的 2 * 2 池化

```
def max_pool(value, kernel_size=None, strides=None, padding="SAME"):
    """
    池化，默认将输入的宽高缩小到一半
    :param value: 输入
    :param kernel_size: 池化卷积核形状 [batch=1, height=2, weight=2, channel=1]
    :param strides: 步长 [batch=1, height=2, weight=2, channel=1]
    :param padding: 填充方式，默认为"SAME"
    :return: 输出
    """
    if strides is None:
        strides = [1, 2, 2, 1]
    if kernel_size is None:
        kernel_size = [1, 2, 2, 1]
    return tf.nn.max_pool(value, ksize=kernel_size, strides=strides, padding=padding)
```

3、2 层的全连接网络

我在所有对照实验里主要采用卷积层输出 7*7*50 个节点后进行全连接到 50 个节点，再输出 10 个预测值的全连接层

```
conv1_shape = [5, 5, 1, 20]      # -------关键参数--卷积层1
conv2_shape = [3, 3, 20, 50]     # -------关键参数--卷积层2
fc1_shape = [7 * 7 * 50, 500]    # -------关键参数--全连接层1
fc2_shape = [500, 10]            # -------关键参数--全连接层2
learning_rate = 0.001
keep_prob = 0.7
max_iteration = 20
```

4、其余超参数

为避免折磨我的 gpu，我选择了每组模型最多跑 20 次迭代，实际上这也跑出了不错的结果
学习率主要使用 0.001，我尝试了 0.01 但是报错，某处计算出了 nan
训练中的保留率主要设置为 0.7

```
conv1_shape = [5, 5, 1, 20]
conv2_shape = [5, 5, 20, 50]
fc1_shape = [7 * 7 * 50, 500]
fc2_shape = [500, 10]
learning_rate = 0.001    # -------关键参数--学习率，0.001
keep_prob = 0.7
max_iteration = 20
```

九、实验中出现的问题和相应的解决办法

1、刚开始时想解析 MNIST 提供的源文件，但是后在搜索资料的时候发现 TensorFlow 已经提供了解析数据源的方法，所以就直接调用了
2、刚开始时想通过 excel 绘制表格来将结果进行可视化展示，后来发现每次都要操作很麻烦，而且展示的效果也不是很好，就去学习了 pyplot 并写了展示并保存图片的方法
3、刚开始时看到安装 TensorFlow 要使用 gpu 的话要安装些麻烦的包，图方便就直接使用 cpu 跑程序时，发现实在太慢，即使只使用很小的网络速度也十分感人，最后还是不得已去装了 TensorFlow-gpu+ cuDNN+CUDA，最后效果很满意

十、对实验的评价和感觉

这次实验前我正在自学 CNN，正好有这个机会上手实践代码，体会到了从输入到卷积到池化到全连接再到输出的全过程，也体会到了 gpu 对机器学习计算的超强能力。这也是我第一次接触 TensorFlow 框架，以前学自学机器学习的时候都是手撸代码，实在难写，这次也算学了一个工具，本来也是想手写实现的，因为学期太紧，还有其他课程，如果时间不是那么紧张的话还是希望尝试一下手动实现 CNN 网络的。不过可惜的是写这次实验代码中因为直接使用了 TensorFlow 中对数据源的读取，没有体验到解析数据源的步骤，以后有时间的话还是得再回头写写。最后就是关于 tensorboard 的使用，刚开始做可视化时按照以前的习惯用了 pyplot。后来才知道有更方便的 tensorboard 可以用，后续的学习中一定要去学会用这些方便的工具