专业：求是科学班（计算机）
姓名：　蒋仕彪
学号：　3170102587
日期：　2020/1/15

浙江大学实验报告

课程名称：　计算机视觉　　指导老师：　宋明黎　　成绩：

实验名称：　HW#5: 利用 CNN 进行手写数字识别

一、实验目的和要求

- 框架：TensorFlow https://github.com/tensorflow/tensorflow（已包含下面网络结构与数据集）
- 数据集：The Mnist Database of handwritten digits http://yann.lecun.com/exdb/mnist/
- 网络结构：LeNet-5 http://yann.lecun.com/exdb/lenet/

1. 具体任务：

   利用上述数据集、网络结构以及选定的 TensorFlow 框架实现手写数字的识别

   参考链接：

   ①https://www.tensorflow.org/versions/r0.12/tutorials/mnist/pros/index.html

   ②http://wiki.jikexueyuan.com/project/tensorflowzh/tutorials/mnist_beginners.html

   ③http://blog.csdn.net/kkk584520/article/details/51477537

2. 提交报告（个人实现过程+结果）

二、实验内容和原理

2.1 基本环境和实现方法的选取

应实验要求，我选择 python 作为语言，选择 Tensorflow(CPU) 作为深度学习的框架。

由于网络的搭建和调试需要大量的测试，把代码直接写在 python 文件里很不方便——每次都要重新编译运行，错误行的定位也很麻烦。于是我选择在 Jupyter Notebook 环境下写 python。

2.2 数据集的配置和预处理

MNIST 是深度学习入门最经典的数据集之一，所以 tensorflow 直接将其封装在框架里。

从官网下载数据集后，我们就可以直接在 tensorflow 的帮助下（它已经帮我们写好读入接口了），很方便地把数据读入进来。如图：

```
import tensorflow.examples.tutorials.mnist.input_data as input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

One-hot = True 的意思是：读入的 label 不是直接的答案，而是编码成 10 维的向量。只有正确答案的位置处会被标为 1，其余位置会被标为 0。接着我们设定一些基本参数，如下图：

```
print (mnist)
CLASS_NUMBERS = 10 # The MNIST dataset has 10 classes, varying from 0 to 9.
IMAGE_SIZE = 28    # The MNIST images are always 28x28 pixels.
IMAGE_PIXELS = IMAGE_SIZE * IMAGE_SIZE
```

```
Datasets(train=<tensorflow.contrib.learn.python.learn.datasets.mnist.DataSet object at 0x000001F39D83CF60>, validation=<tensorflow.contrib.learn.python.learn.datasets.mnist.DataSet object at 0x000001F39CDA04A8>, test=<tensorflow.contrib.learn.python.learn.datasets.mnist.DataSet object at 0x000001F398B447F0>)
```

IMAGE_SIZE 是每个图像固定的长和宽。每张图像是固定的 28*28 的灰度图。

CLASS_NUMBERS 是我们分类器的输出范围。这里是识别数字 0~9，显然有 10 个。

MNIST 的数据集被划成了三部分：训练集、验证集、测试集，如图：

```
# minist.train.* 训练集
# minist.validation.* 验证集
# minist.test.* 测试集
print (mnist.train.images.shape)
print (mnist.train.labels.shape)
print (mnist.validation.images.shape)
print (mnist.test.images.shape)
```

```
(55000, 784)
(55000, 10)
(5000, 784)
(10000, 784)
```

可以发现，训练集一共有 55000 幅图片，验证集有 5000 幅，测试集有 10000 集。训练时的逻辑是：**每次在训练集上跑 epoch 训练；如果想看训练效果，可以以一定的频率把当前模型带到验证集里去验证；最终提交模型时，再在测试集里跑结果观察“客观”的正确率**。

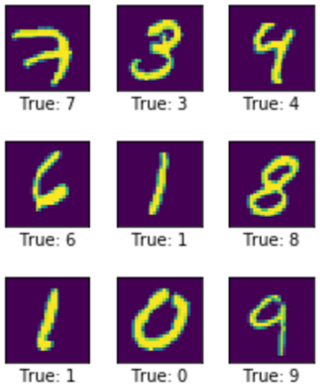
为了方便之后的操作，我对数据又进行了简单的处理——把压成一维的图像 reshape 成两维；增加一份“把 one-hot 的编码缩减成正确答案”的数据。

```
def refineData(data):
    return data.images.reshape((len(data.images), IMAGE_SIZE, IMAGE_SIZE)),
    data.labels, np.array([np.argmax(data.labels[k]) for k in range(len(data.labels))])

train_images, train_labels, train_answers = refineData(mnist.train)
validation_images, validation_labels, validation_answers = refineData(mnist.validation)
test_images, test_labels, test_answers = refineData(mnist.test)
# refine the data.
# _images: IMAGE_PIXELS
# _labels: one-hot ground truth
# _answers: single-value ground truth
```

2.3 数据集可视化分析

MNIST 里的数字图像究竟是怎么样的呢？我们可以用 python 强大的绘图功能来很方便地生成。

```
# Check ground truth or prediction
def plot_images(images, cls_true, cls_pred = None):
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.6)

    for i, ax in enumerate(axes.flat):
        ax.imshow(images[i].reshape(IMAGE_SIZE, IMAGE_SIZE))

        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i], cls_pred[i])

        ax.set_xlabel(xlabel)

        ax.set_xticks([])
        ax.set_yticks([])

    plt.show()
plot_images(train_images[0:9], train_answers[0:9])
```



在上述函数里，我传入一部分的图像和它们的 ground truth，试图可视化出这些图像表示的实际内容。在 plt 的 subplot 的帮助下能生成很好看的可视化信息。结果展示在右侧。

2.4 Lenet5 网络搭建

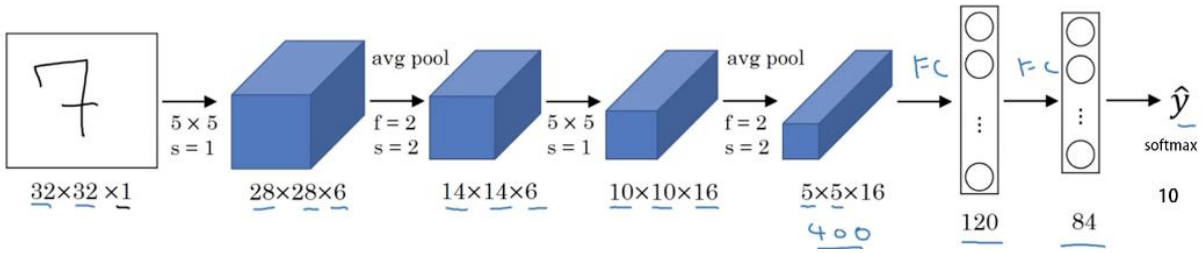
由于以前接触过深度学习相关内容，做这次实验时，我决定用刚学的 **slim** 来搭建网络。slim 是 tf 里封装性比较“中等”的框架。它不像 keras 那么高层而无脑，所涉及到的网络层均是 tf 里比较基础的。它通过函数的参数共享机制来极大地简化代码。

可以在网络搭建之前先定义一个基本的参数共享机制：

```
def net_arg_scope(weight_decay=0.0005):
    with slim.arg_scope([slim.conv2d, slim.fully_connected],
                        activation_fn=tf.nn.relu,
                        biases_initializer=tf.constant_initializer(0.1),
                        weights_regularizer=slim.l2_regularizer(weight_decay)):
        with slim.arg_scope([slim.conv2d], padding='SAME'):
            with slim.arg_scope([slim.max_pool2d], padding='VALID') as arg_sc:
                return arg_sc
```

只要我们在后面搭网络时用 with 把 net_arg_scope 套起来，接下来的函数会默认使用这些参数。在上述代码里，中括号[]表示适用的函数名称，A=B 就表示 A 参数默认取成 B。在这里，我默认卷积层用 same 处理卷积核不整除边界的情况，而 pooling 层用 valid 处理（缺的情况要舍去，不能补 0）。

网络具体要怎么搭建，只要参考 Lenet5 的结构照搬即可。



注意 Lenet5 本来是为 32*32 的图像设计的，而 MNIST 的图像是 28*28。我在运用时**直接取了原来 Lenet5 的卷积核大小**，这样每层过后的图像大小都是比标准的 Lenet5 小一些。

运用 slim 后代码得到了极度简化。下图可以很清晰地看到各层的情况：

```
with tf.variable_scope(scope, 'Lenet5', [inputs], reuse = tf.AUTO_REUSE) as sc:
    with slim.arg_scope([slim.conv2d, slim.fully_connected, slim.max_pool2d]):
        net = slim.conv2d(inputs, 6, [5, 5], 1, padding='VALID', scope='conv1')
        print (net.shape) # 24*24*6
        net = slim.max_pool2d(net, [2, 2], 2, scope='pool1')
        print (net.shape) # 12*12*6
        net = slim.conv2d(net, 6, [5, 5], 1, padding='VALID', scope='conv2')
        print (net.shape) # 8*8*16
        net = slim.max_pool2d(net, [2, 2], 2, scope='pool2')
        print (net.shape) # 4*4*16
        shape = net.get_shape().as_list()
        net = tf.reshape(net,[-1, shape[1] * shape[2] * shape[3]])
        print (net.shape) # 256

    with slim.arg_scope([slim.fully_connected],
                        weights_initializer=trunc_normal(0.005),
                        biases_initializer=tf.constant_initializer(0.1)):
        net = slim.fully_connected(net, 120, scope='fc1')
        net = slim.fully_connected(net, num_classes, scope='fc2')
        print (net.shape)
```

2.5 网络训练

Tf 有一个比较厉害的机制叫做 **placeholder**。我个人的理解是，这可以更方便地实现 batch 训练（比如每次可以动态调整 batch 的大小）。我们可以声明一种类型叫做“占位符”，该变量在定义时其实还没有真实出现（我们只是定义了一个“外壳”）。我们可以根据这个外壳去定义其他的变量和结构。等待需要用到这个数据时，我们再用真实数据把这个外壳填满。如下图：

```
x = tf.placeholder(tf.float32, shape=(None, IMAGE_SIZE, IMAGE_SIZE, 1), name = "input_data")
y = tf.placeholder(tf.float32, shape=(None, CLASS_NUMBERS), name = "output_data")
is_training = tf.placeholder(tf.bool, name="is_training")
```

X 表示输入网络的结构（batch，size，size，channel），Y 表示输出网络的结构（batch，class），is_training 表示这次跑是在训练还是在测试。在训练和测试时，大部分情况都是一样的，只有微小的区别。比如训练时为了防止过拟合会进行 dropout，但是测试时不能 dropout。网络结构搭建时会依据这个变量讨论。

接下来就要思考深度学习时具体怎么根据数据优化模型参数。在分类问题中，我们常用**交叉熵函数**作为损失函数，在 tf 上即对应 softmax_cross_entropy_with_logits。它要传入 labels 和 logits 两个参数，分别表示真实标签和网络给出的预测。注意这两个值都是多维的（因为标签是 one-hot 过的），而损失函数的结果必然是单个的值，所以我们要做一步 reduce_mean 取一个平均值。最后我们要选择一个**优化器**并给出学习率，使模型在梯度下降后可以改进自己。我采用了 Adam 优化器（它是目前比较鲁棒的一个优化器）。

```
outputs = alexnet(inputs = x, num_classes = CLASS_NUMBERS)
loss = tf.nn.softmax_cross_entropy_with_logits(labels = y, logits = outputs)
cost = tf.reduce_mean(loss)
optm = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

我们称每训练完整批为一个 epoch。由于训练数据往往很大，我们不可能也做不到把整个 epoch 放在一起训练。为了加快速度，我们通常把训练集分成好几部分，每一部分称为一个 batch。所以网络训练时分为 epoch 和 batch 两个阶段。实际上我们往 placeholder 里传数据的时候都是以 batch 为准的：在当前这个 batch 的训练任务里，我们把对应数量的图像传进网络让其计算，再用 BP 去求出导数丢到优化器里去优化。具体的流程可以看下面这份代码。这里有两个 sess.run()，一个是算 loss，一个是优化。

```
for epoch in range(learning_epoches):
    number = int(len(train_images) / batch_size)
    for cur_batch in range(number):
        with slim.arg_scope(net_arg_scope()):
            now_images =   train_images[batch_size * cur_batch : batch_size * (cur_batch + 1)]
            now_answers = train_answers[batch_size * cur_batch : batch_size * (cur_batch + 1)]
            now_labels =   train_labels[batch_size * cur_batch : batch_size * (cur_batch + 1)]

            now_images = now_images.reshape(batch_size, IMAGE_SIZE, IMAGE_SIZE, 1)

            feeds = {x: now_images, y: now_labels, is_training: True}
            sess.run(optm, feed_dict = feeds)
            now_loss = sess.run(cost, feed_dict = feeds)
            step += 1
```

有一个小细节是：原来的 images 是 [?,28,28] 的，但是在网络训练时**还会有一个通道数**，所以要把它 reshape 一下，扩展一个大小为 1 的维度。

还有一个比较重要的东西是学习率的设置。一种比较好的解决方案是：学习率随着训练的推进动态缩小。MNIST 训练集的正确率本来就很高了，所以我直接取了 learning_rate=0.001 这个定值。

在 Jupyter Notebook 上测试时，一个经典的坑就是**图的“残留”**。因为它会实时保留之前运行的结果，当你第二次去跑代码的时候，可能会遇到因为上一次图的残留而编译错误的情况。解决的方法有很多，比如每次重新跑这个 Kernel，或者在代码开头用 tf.reset_default_graph() 把之前的图给清除掉。

2.6 网络测试和保存

训练了一段时间的网络后，我们需要把数据丢进验证集里跑，观察它的正确率。所以我们首先要考虑的是如何检验在一批数据下的正确率。正确率的设定代码如下：

```
correct = tf.equal(tf.arg_max(outputs, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct, "float"))
```

tf.arg_max 是找到一个向量中最大值的下标，在这里就计算出了网络的预测值（output 输出的本来 one-hot，经过了这一步操作后就得到了具体的值）。然后我们在拿它和真实标签作比较。最后我们“把每一个图像是否相等”的这个向量求平均值，即得到了正确率。

我在程序里的约定是：每做完一个 epoch 就把整个验证集丢进网路跑一下观察正确率。如果该正确率比以往所有的都要高，我就把这个模型保存下来。计算正确率时只要把验证集喂给 placeholder，然后调用之前的 accuracy 即可得到。具体的代码如下：

```
now_images =   validation_images
now_answers = validation_answers
now_labels =   validation_labelsXW
now_images = now_images.reshape(len(validation_images), IMAGE_SIZE, IMAGE_SIZE, 1)

test_feeds = {x: now_images, y: now_labels, is_training: False}
ACRate = sess.run(accuracy, feed_dict = test_feeds)
AC_list.append(ACRate)

print ("Epoch %d Accuracy: %.5f" % (epoch, ACRate))
if ACRate > best_ACRate:
    best_ACRate = ACRate
    savename = savedir + "best__accuracy_" + str(best_ACRate) + ".ckpt"
    saver.save(sess = sess, save_path = savename)
    print (" [%s] SAVED." % (savename))
```

上述代码里，AC_list 记录了每次的正确率，供后面的分析使用。

当然在训练完所有 epoch 的时候（我固定跑了 20 个 epoch），我们还要把测试集丢进现在的这个网络看看“真实正确率”是怎么样的。这里的步骤和丢验证集是类似的。

```
plt.plot([k for k in range(len(AC_list))], AC_list, 'o-', linewidth=2)

now_images =   test_images
now_answers =  test_answers
now_labels =   test_labels
now_images = now_images.reshape(len(test_images), IMAGE_SIZE, IMAGE_SIZE, 1)
input_size = len(test_images)
test_feeds = {x: now_images, y: now_labels, is_training: False}
ACRate = sess.run(accuracy, feed_dict = test_feeds)

print ("Actual AC Rate in test dataset: %.5f" % (ACRate))

time_end = time.time()
print('totally cost',time_end-time_start, "s")
```

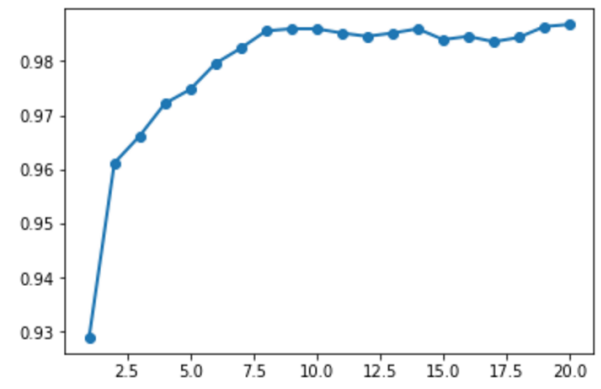
为了加强可视化分析，我们可以把之前每个 epoch 里的正确率收集起来，并用 plt 画出折线图，观察正确率的变化。此外，**训练时间**也是衡量一个网络是否优秀的重要标准。我们可以在训练前和训练后加一个时间戳，分析本次训练和测试一共花了多久。MNIST 数据集特别小，一般都是以秒为单位的。

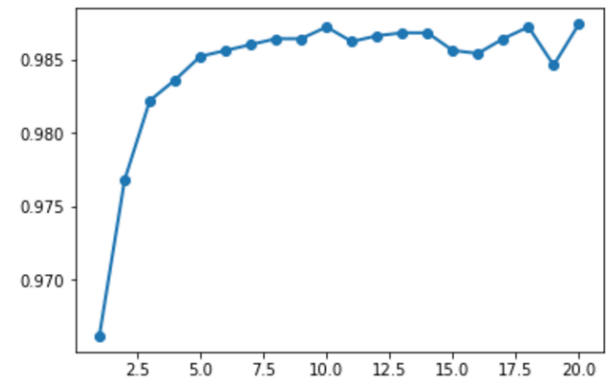
三、成果展示

3.1 训练结果展示

```
Epoch 0 Accuracy: 0.92900
 [./Lenet5/best__accuracy_0.929.ckpt] SAVED.
Epoch 1 Accuracy: 0.96120
 [./Lenet5/best__accuracy_0.9612.ckpt] SAVED.
Epoch 2 Accuracy: 0.96620
 [./Lenet5/best__accuracy_0.9662.ckpt] SAVED.
Epoch 3 Accuracy: 0.97220
 [./Lenet5/best__accuracy_0.9722.ckpt] SAVED.
Epoch 4 Accuracy: 0.97480
 [./Lenet5/best__accuracy_0.9748.ckpt] SAVED.
Epoch 5 Accuracy: 0.97960
 [./Lenet5/best__accuracy_0.9796.ckpt] SAVED.
Epoch 6 Accuracy: 0.98240
 [./Lenet5/best__accuracy_0.9824.ckpt] SAVED.
Epoch 7 Accuracy: 0.98560
 [./Lenet5/best__accuracy_0.9856.ckpt] SAVED.
Epoch 8 Accuracy: 0.98600
 [./Lenet5/best__accuracy_0.986.ckpt] SAVED.
Epoch 9 Accuracy: 0.98600
Epoch 10 Accuracy: 0.98520
Epoch 11 Accuracy: 0.98460
Epoch 12 Accuracy: 0.98520
Epoch 13 Accuracy: 0.98600
Epoch 14 Accuracy: 0.98400
Epoch 15 Accuracy: 0.98460
Epoch 16 Accuracy: 0.98360
Epoch 17 Accuracy: 0.98440
Epoch 18 Accuracy: 0.98640
 [./Lenet5/best__accuracy_0.9864.ckpt] SAVED.
Epoch 19 Accuracy: 0.98680
 [./Lenet5/best__accuracy_0.9868.ckpt] SAVED.
Actual AC Rate in test dataset: 0.98680
totally cost 681.3585848808289s
```

```
Epoch 0 Accuracy: 0.96620
 [./AlexNet/best__accuracy_0.9662.ckpt] SAVED.
Epoch 1 Accuracy: 0.97680
 [./AlexNet/best__accuracy_0.9768.ckpt] SAVED.
Epoch 2 Accuracy: 0.98220
 [./AlexNet/best__accuracy_0.9822.ckpt] SAVED.
Epoch 3 Accuracy: 0.98360
 [./AlexNet/best__accuracy_0.9836.ckpt] SAVED.
Epoch 4 Accuracy: 0.98520
 [./AlexNet/best__accuracy_0.9852.ckpt] SAVED.
Epoch 5 Accuracy: 0.98560
 [./AlexNet/best__accuracy_0.9856.ckpt] SAVED.
Epoch 6 Accuracy: 0.98600
 [./AlexNet/best__accuracy_0.986.ckpt] SAVED.
Epoch 7 Accuracy: 0.98640
 [./AlexNet/best__accuracy_0.9864.ckpt] SAVED.
Epoch 8 Accuracy: 0.98640
Epoch 9 Accuracy: 0.98720
 [./AlexNet/best__accuracy_0.9872.ckpt] SAVED.
Epoch 10 Accuracy: 0.98620
Epoch 11 Accuracy: 0.98660
Epoch 12 Accuracy: 0.98680
Epoch 13 Accuracy: 0.98680
Epoch 14 Accuracy: 0.98560
Epoch 15 Accuracy: 0.98540
Epoch 16 Accuracy: 0.98640
Epoch 17 Accuracy: 0.98720
Epoch 18 Accuracy: 0.98460
Epoch 19 Accuracy: 0.98740
 [./AlexNet/best__accuracy_0.9874.ckpt] SAVED.
Actual AC Rate in test dataset: 0.98750
totally cost 268.436886548996 s
```





左上图是 Lenet5 训练完后反馈的信息。它总共花了 681 秒，最终正确率是 98.7%。

右上图是“类 Alenet”[1]的网络训练完后的结果。总共花了 268 秒，最终征率是 98.8%。

左下图和右下图是这两个网络在 20 个 epoch 里验证集正确率的折线图（**注意右图 y 坐标大于左图**）。

[1]为了对比 Lenet5 的正确率，我就手动实现了另一个网络。它和 lenet5 的最大区别就是：减少了一层卷积和 pooling 层，并把其中一个全连接层换成了卷积层（减少参数）。新的这个网络的卷积层和 pooling 的参数我是参考 Alexnet 的。为什么叫做“类 Alexnet”呢？因为 Alexnet 是作用在 224*224 的图像上的，而这里图像只有 28*28，很快就“卷没了”，所以我只选取了 Alexnet 里的一层卷积层和一层 pooling 层。

3.2 结果分析

1. 最明显的一个结论：MNIST 的数据集很小，训练结果也很优秀。特别先进的网络体现不出自己的优势，大家都是 98~99%的正确率。

2. MNIST 数据集的数据一致性很高。验证集里能跑多少正确率，测试集里也能跑多少。还有一个原因是输入图像较小，几乎不会出现过拟合的情况。

3. Lenet5 的收敛速度比较慢。上图展示了它在第一个 epoch 后只有 92%的正确率，而另一个网络直接有 96+%的正确率。为了确认这件事（确保不是随机因素造成的），我后来每做完一个 batch 后就测试一下，发现 Lenet5 第一个 batch 的正确率只有 80%左右，而另一个网络有 94%左右。我对此的解释是，Lenet5 有两层全连接层，参数比较多，所以收敛速度变慢了。

4. 大概是和上一条一样的原因，Lenet5 的训练速度比较慢。

四、感想和收获

我以前用 pytorch 训练过一些网络，对于这次的作业就有迷之自信，觉得很快就能完成。事实上，我用 tf 手动实现了一遍后，依然遇到了不少问题（主要是编译问题哈哈）。最主要的是这两个问题：

① **在 placeholder 的时候，不能预设 Tensor 的格式**。在 tf 的框架里，所有带确定参数都保存成了 Tensor 的计算图模型，我就想当然地把 placeholder 里传入的数据也设成 Tensor 了。结果 python 的报错我没看懂，我一直以为是我 placeholder 的定义和喂给的数据不一致，起劲地在那里改，甚至把两个 Tensor 的名字都写成一样了>_<。网上搜资料也没搜到类似的。最后发现是 Tensor 的格式不对，改成最基本的 numpy 格式就行了。我猜每次数据都要用 numpy 格式喂，在进入网络后自动转化成 Tensor 格式了。

② **placeholder 的第一位设置成 None 就很难 reshape 了**。这个问题应该很经典，但是我却在上面花了超过半小时的时间思考和修正。主要问题是这样的：首先为了 batch 的动态性（包括我们用的测试数据的 batch 与训练可能不一样），网络接受的第一维数据（图像个数）一般会在 placeholder 里设置成 None（意思是不确定，None 可以匹配所有数）。而 Lenet5 结构里有一层需要全连接，之前的数据维度是 [?, 4, 4, 16] 的，**我们想把它先变成 [?, 256] 再进行全连接**。但是在网络定义时，**该数据的第一位其实是 None，而 reshape 的时候不支持 None**。遇到了这个问题后，我首先想到的解决方案是：把第一维改成一个固定的值，就可以随意 reshape 了。但是这会导致一个不便利的情况：等到我们要验证和测试的时候，我们必须把 batch_size 调成和训练时的一样一批一批测，最后再求个平均。这样显然不 general 也不优美。经过多次尝试，我发现可以在 reshape 的时候把至多一维设置成-1。代码运行时会自动给它分配合适的大小。所以上述问题可以简单地改为 net = tf.reshape(net, [-1, 后三维的长度]。

虽然仅仅是个 MNIST，看到代码跑起来后还是很开心的。