

Tensorflow 笔记：第七讲

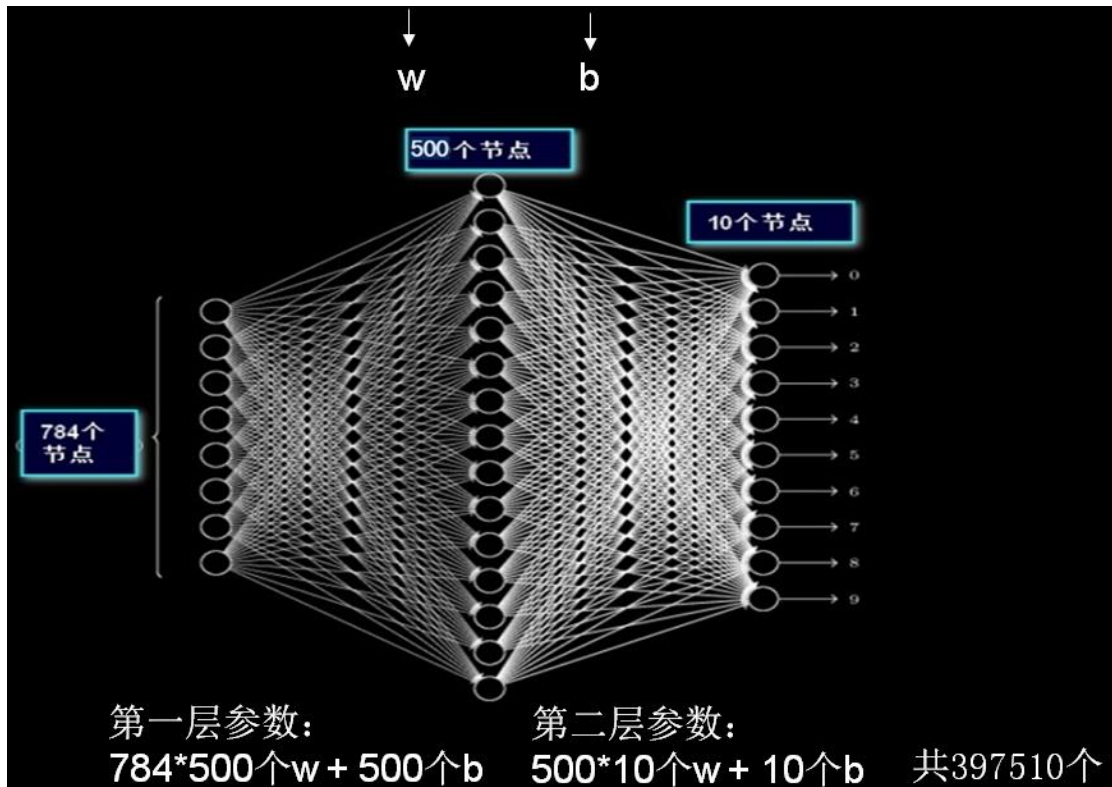
卷积神经网络

本节目标：学会使用 CNN 实现对手写数字的识别。

7.1

✓ 全连接 NN：每个神经元与前后相邻层的每一个神经元都有连接关系，输入是特征，输出为预测的结果。

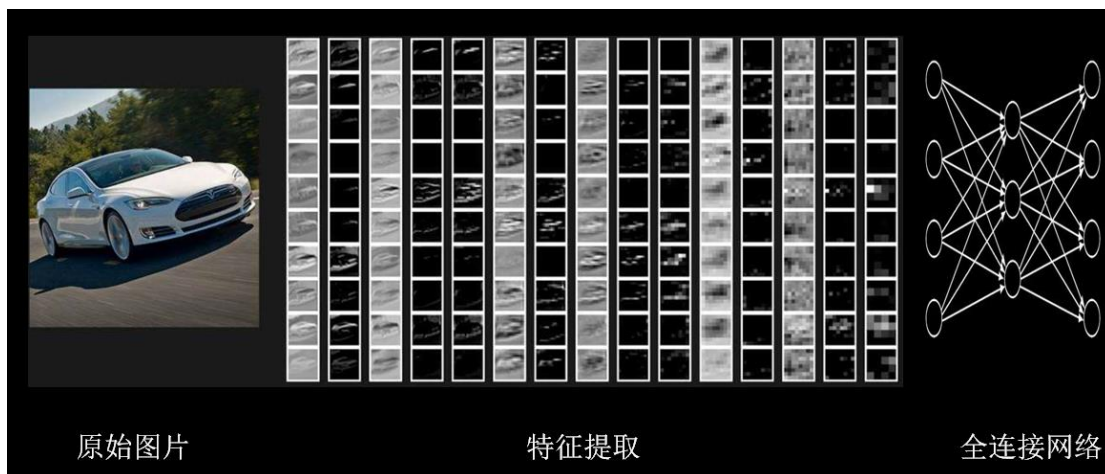
参数个数： \sum （前层 \times 后层 + 后层）



一张分辨率仅仅是 28×28 的黑白图像，就有近 40 万个待优化的参数。现实生活中高分辨率的彩色图像，像素点更多，且为红绿蓝三通道信息。

待优化的参数过多，容易导致模型过拟合。为避免这种现象，实际应用中一般不会将原始图片直接喂入全连接网络。

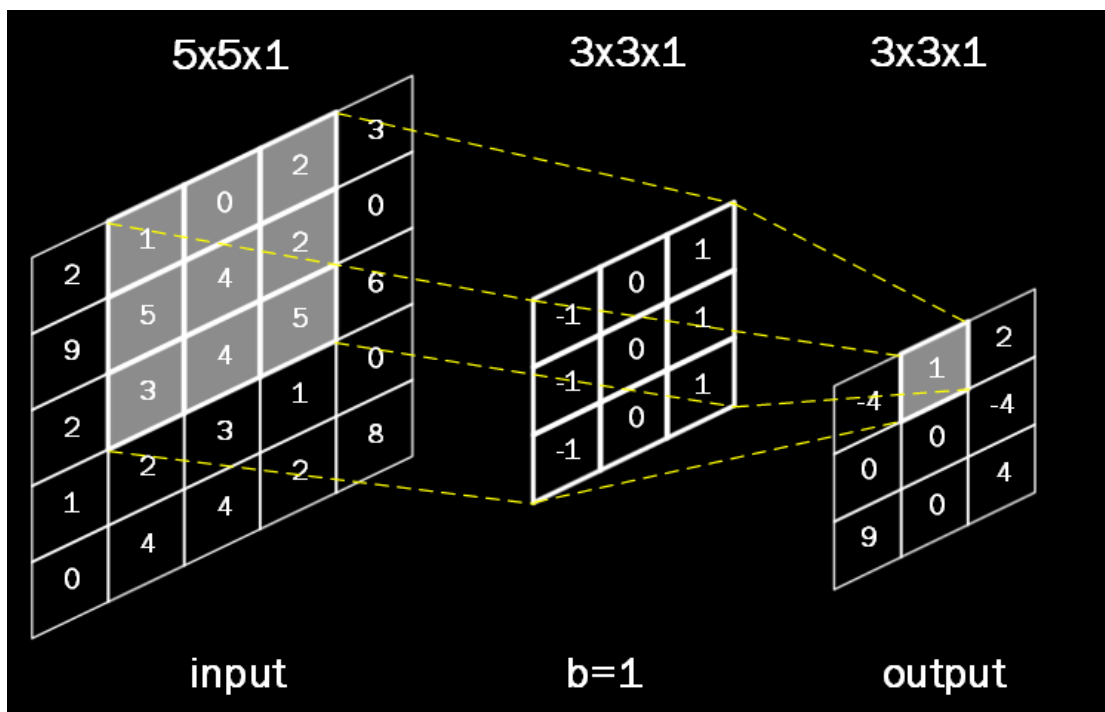
✓ 在实际应用中，会先对原始图像进行特征提取，把提取到的特征喂给全连接网络，再让全连接网络计算出分类评估值。



例：先将此图进行多次特征提取，再把提取后的计算机可读特征喂给全连接网络。

✓ 卷积 Convolutional

卷积是一种有效提取图片特征的方法。一般用一个正方形卷积核，遍历图片上的每一个像素点。图片与卷积核重合区域内相对应的每一个像素值乘卷积核内相对应点的权重，然后求和，再加上偏置后，最后得到输出图片中的一个像素值。



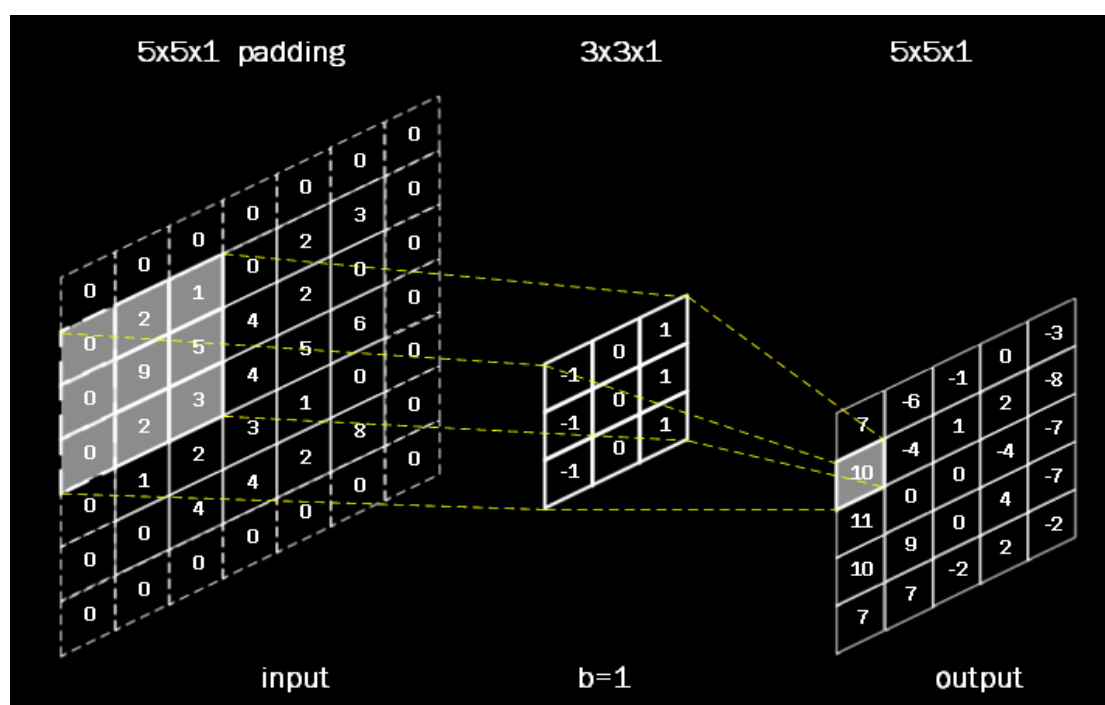
例：上面是 5x5x1 的灰度图片，1 表示单通道，5x5 表示分辨率，共有 5 行 5 列个灰度值。若用一个 3x3x1 的卷积核对此 5x5x1 的灰度图片进行卷积，偏置项

$b=1$ ，则求卷积的计算是： $(-1) \times 1 + 0 \times 0 + 1 \times 2 + (-1) \times 5 + 0 \times 4 + 1 \times 2 + (-1) \times 3 + 0 \times 4 + 1 \times 5 + 1 = 1$ （注意不要忘记加偏置 1）。

输出图片边长 = $(\text{输入图片边长} - \text{卷积核长} + 1) / \text{步长}$ ，此图为： $(5 - 3 + 1) / 1 = 3$ ，输出图片是 3×3 的分辨率，用了 1 个卷积核，输出深度是 1，最后输出的是 $3 \times 3 \times 1$ 的图片。

✓ 全零填充 Padding

有时会在输入图片周围进行全零填充，这样可以保证输出图片的尺寸和输入图片一致。



例：在前面 $5 \times 5 \times 1$ 的图片周围进行全零填充，可使输出图片仍保持 $5 \times 5 \times 1$ 的维度。这个全零填充的过程叫做 padding。

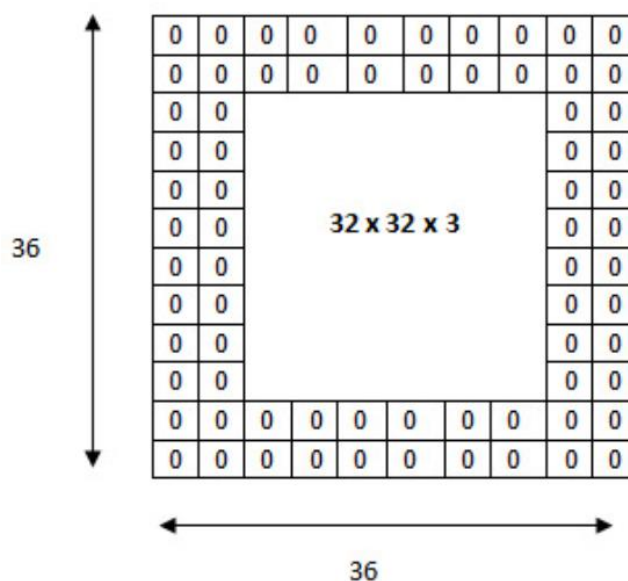
$$\text{输出数据体的尺寸} = (W - F + 2P) / S + 1$$

W ：输入数据体尺寸， F ：卷积层中神经元感知域， S ：步长， P ：零填充的数量。

例：输入是 7×7 ，滤波器是 3×3 ，步长为 1，填充为 0，那么就能得到一个 5×5 的输出。如果步长为 2，输出就是 3×3 。

如果输入量是 $32 \times 32 \times 3$ ，核是 $5 \times 5 \times 3$ ，不用全零填充，输出是 $(32 - 5 + 1) / 1 = 28$ ，如果要想输出量保持在 $32 \times 32 \times 3$ ，可以对该层加一个大小为 2 的零填充。可以根据需求计算出需要填充几层零。 $32 = (32 - 5 + 2P) / 1 + 1$ ，计算出 $P=2$ ，即需填充 2

层零。



✓ 使用 padding 和不使用 padding 的输出维度

$$\text{padding} \left\{ \begin{array}{l} \text{SAME} \\ \text{VALID (不全0填充)} \end{array} \right. \left\{ \begin{array}{l} \frac{\text{入长}}{\text{步长}} \quad (\text{向上取整}) \\ \frac{\text{入长} - \text{核长} + 1}{\text{步长}} \quad (\text{向上取整}) \end{array} \right.$$

上一行公式是使用 padding 的输出图片边长，下一行公式是不使用 padding 的输出图片边长。公式如果不能整除，需要向上取整数。如果用全零填充，也就是 padding=SAME。如果不用全零填充，也就是 padding=VALID。

✓ Tensorflow 给出的计算卷积的函数

tf.nn.conv2d (输入描述, eg. [batch, 5, 5, 1])

卷积核描述, eg. [3, 3, 1, 16]

核滑动步长, eg. [1, 1, 1, 1]

padding = 'VALID')

分辨率 通道数

行列分辨率 通道数 核个数

行步长 列步长

函数中要给出四个信息：对输入图片的描述、对卷积核的描述、对卷积核

滑动步长的描述以及是否使用 padding。

1) 对输入图片的描述: 用 batch 给出一次喂入多少张图片, 每张图片的分辨率大小, 比如 5 行 5 列, 以及这些图片包含几个通道的信息, 如果是灰度图则为单通道, 参数写 1, 如果是彩色图则为红绿蓝三通道, 参数写 3。

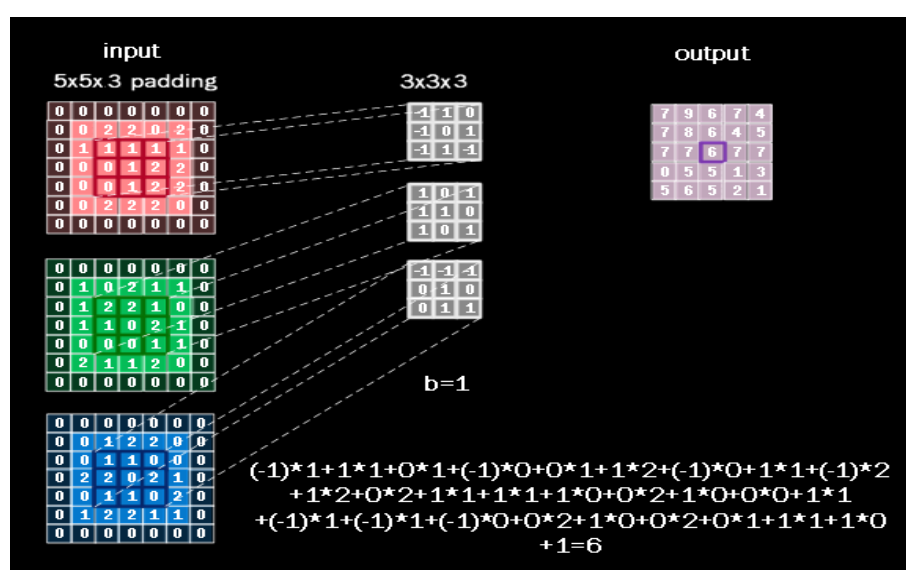
2) 对卷积核的描述: 要给出卷积核的行分辨率和列分辨率、通道数以及用了几个卷积核。比如上图描述, 表示卷积核行列分辨率分别为 3 行和 3 列, 且是 1 通道的, 一共有 16 个这样的卷积核, 卷积核的通道数是由输入图片的通道数决定的, 卷积核的通道数等于输入图片的通道数, 所以卷积核的通道数也是 1。一共有 16 个这样的卷积核, 说明卷积操作后输出图片的深度是 16, 也就是输出为 16 通道。

3) 对卷积核滑动步长的描述: 上图第二个参数表示横向滑动步长, 第三个参数表示纵向滑动步长。第一个 1 和最后一个 1 这里固定的。这句表示横向纵向都以 1 为步长。

4) 是否使用 padding: 用的是 VALID。注意这里是以字符串的形式给出 VALID。

✓ 对多通道的图片求卷积

多数情况下, 输入的图片是 RGB 三个颜色组成的彩色图, 输入的图片包含了红、绿、蓝三层数据, 卷积核的深度应该等于输入图片的通道数, 所以使用 3x3x3 的卷积核, 最后一个 3 表示匹配输入图像的 3 个通道, 这样这个卷积核有三层, 每层会随机生成 9 个待优化的参数, 一共有 27 个待优化参数 w 和一个偏置 b 。

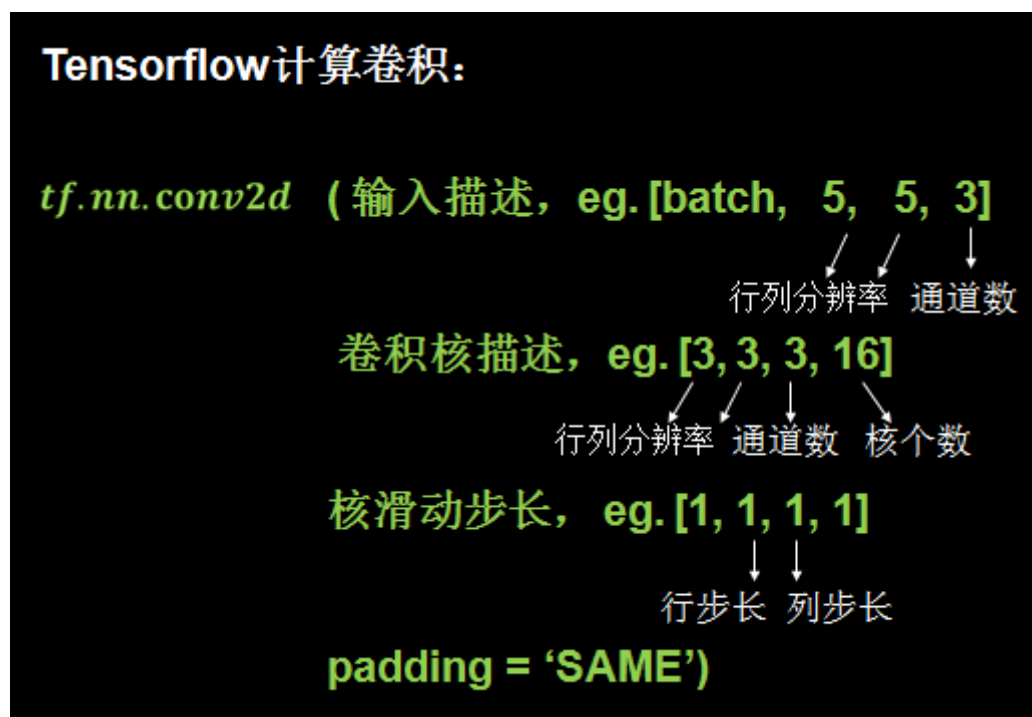


对于彩色图, 按层分解开, 可以直观表示为上面这张图, 三个颜色分量: 红

色分量、绿色分量和蓝色分量。

卷积计算方法和单层卷积核相似，卷积核为了匹配红绿蓝三个颜色，把三层的卷积核套在三层的彩色图片上，重合的 27 个像素进行对应点的乘加运算，最后的结果再加上偏置项 b ，求得输出图片中的一个值。

这个 $5 \times 5 \times 3$ 的输入图片加了全零填充，使用 $3 \times 3 \times 3$ 的卷积核，所有 27 个点与对应的待优化参数相乘，乘积求和再加上偏置 b 得到输出图片中的一个值 6。



针对上面这幅彩色图片，用 `conv2d` 函数实现可以表示为：

一次输入 `batch` 张图片，输入图片的分辨率是 5×5 ，是 3 通道的，卷积核是 $3 \times 3 \times 3$ ，一共有 16 个卷积核，这样输出的深度就是 16，核滑动横向步长是 1，纵向步长也是 1，`padding` 选择 `same`，保证输出是 5×5 分辨率。由于一共用了 16 个卷积核，所以输出图片是 $5 \times 5 \times 16$ 。

✓ 池化 Pooling

Tensorflow 给出了计算池化的函数。最大池化用 `tf.nn.max_pool` 函数，平均池化用 `tf.nn.avg_pool` 函数。

函数中要给出四个信息，对输入的描述、对池化核的描述、对池化核滑动步长的描述和是否使用 `padding`。

Tensorflow计算池化:

```
pool = tf.nn.max_pool ( 输入描述, eg. [batch, 28, 28, 6]
pool = tf.nn.avg_pool
```

池化核描述 (仅大小), eg. [1, 2, 2, 1]

池化核滑动步长, eg. [1, 2, 2, 1]

padding = 'SAME')

行列分辨率 通道数
行列分辨率
行步长 列步长

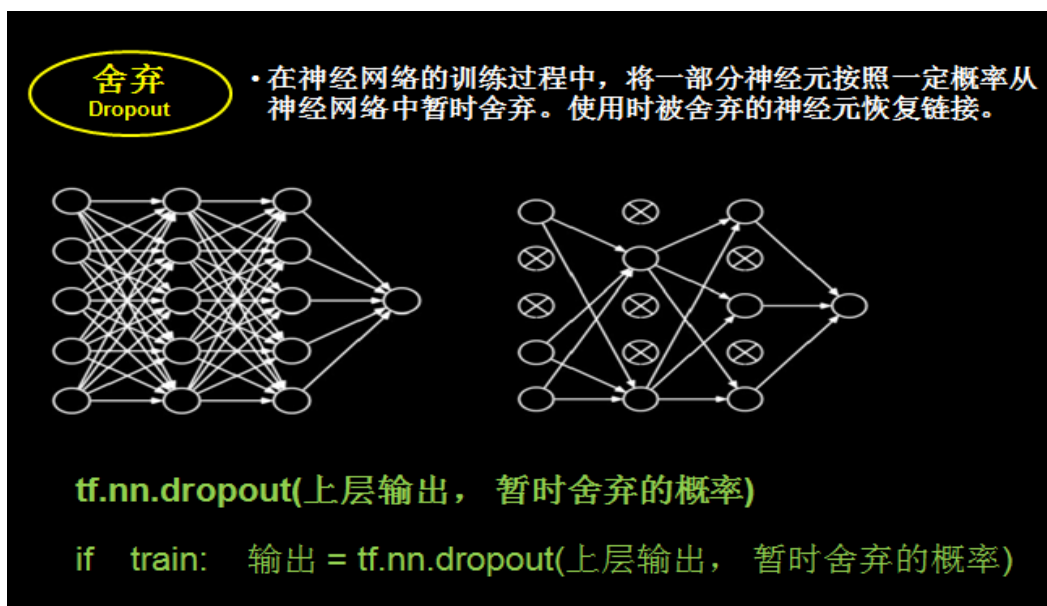
1) 对输入的描述: 给出一次输入 batch 张图片、行列分辨率、输入通道的个数。

2) 对池化核的描述: 只描述行分辨率和列分辨率, 第一个和最后一个参数固定是 1。

3) 对池化核滑动步长的描述: 只描述横向滑动步长和纵向滑动步长, 第一个和最后一个参数固定是 1。

4) 是否使用 padding: padding 可以是使用零填充 SAME 或者不使用零填充 VALID。

✓ 舍弃 Dropout



在神经网络训练过程中，为了减少过多参数常使用 **dropout** 的方法，将一部分神经元按照一定概率从神经网络中舍弃。这种舍弃是临时性的，仅在训练时舍弃一些神经元；在使用神经网络时，会把所有的神经元恢复到神经网络中。比如上面这张图，在训练时一些神经元不参加神经网络计算了。**Dropout** 可以有效减少过拟合。

Tensorflow 提供的 dropout 的函数：用 `tf.nn.dropout` 函数。第一个参数链接上一层的输出，第二个参数给出神经元舍弃的概率。

在实际应用中，常常在前向传播构建神经网络时使用 **dropout** 来减小过拟合加快模型的训练速度。

dropout 一般会放到全连接网络中。如果在训练参数的过程中，输出 `=tf.nn.dropout`（上层输出，暂时舍弃神经元的概率），这样就有指定概率的神经元被随机置零，置零的神经元不参加当前轮的参数优化。

✓ **卷积 NN：借助卷积核（kernel）提取特征后，送入全连接网络。**

卷积神经网络可以认为由两部分组成，一部分是对输入图片进行特征提取，另一部分就是全连接网络，只不过喂入全连接网络的不再是原始图片，而是经过若干次卷积、激活和池化后的特征信息。

卷积神经网络从诞生到现在，已经出现了许多经典网络结构，比如 **Lenet-5**、**Alenet**、**VGGNet**、**GoogleNet** 和 **ResNet** 等。每一种网络结构都是以卷积、激活、池化、全连接这四种操作为基础进行扩展。

Lenet-5 是最早出现的卷积神经网络，由 **Lecun** 团队首先提出，**Lenet-5** 有效解决了手写数字的识别问题。

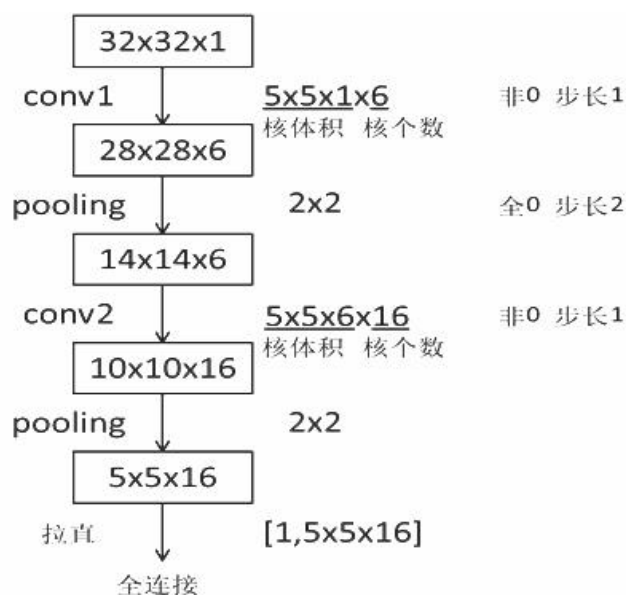
7.2

Lenet 神经网络是 Yann LeCun 等人在 1998 年提出的，该神经网络充分考虑图像的相关性。

✓ Lenet 神经网络结构为：

- ①输入为 $32 \times 32 \times 1$ 的图片大小，为单通道的输入；
- ②进行卷积，卷积核大小为 $5 \times 5 \times 1$ ，个数为 6，步长为 1，非全零填充模式；
- ③将卷积结果通过非线性激活函数；
- ④进行池化，池化大小为 2×2 ，步长为 1，全零填充模式；
- ⑤进行卷积，卷积核大小为 $5 \times 5 \times 6$ ，个数为 16，步长为 1，非全零填充模式；
- ⑥将卷积结果通过非线性激活函数；
- ⑦进行池化，池化大小为 2×2 ，步长为 1，全零填充模式；
- ⑧全连接层进行 10 分类。

Lenet 神经网络的结构图及特征提取过程如下所示：



Lenet 神经网络结构图

Lenet 神经网络的输入是 $32 \times 32 \times 1$ ，经过 $5 \times 5 \times 1$ 的卷积核，卷积核个数为 6 个，采用非全零填充方式，步长为 1，根据非全零填充计算公式：输出尺寸 = (输入尺寸 - 卷积核尺寸 + 1) / 步长 = $(32 - 5 + 1) / 1 = 28$ 。故经过卷积后输出为 $28 \times 28 \times 6$ 。经过第一层池化层，池化大小为 2×2 ，全零填充，步长为 2，由全零填充计算公式：输出尺寸 = 输入尺寸 / 步长 = $28 / 2 = 14$ ，池化层不改变深度，深度仍为 6。用同

样计算方法，得到第二层池化后的输出为 $5*5*16$ 。将第二池化层后的输出拉直送入全连接层。

✓根据 Lenet 神经网络的结构可得，Lenet 神经网络具有如下特点：

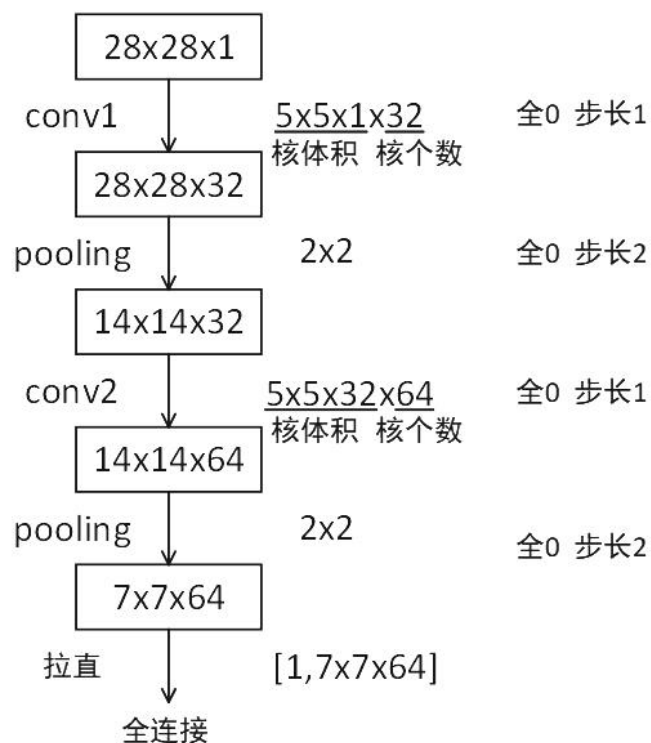
- ①卷积 (Conv)、池化 (ave-pooling)、非线性激活函数 (sigmoid) 相互交替；
- ②层与层之间稀疏连接，减少计算复杂度。

✓对 Lenet 神经网络进行微调，使其适应 Mnist 数据集：

由于 Mnist 数据集中图片大小为 $28*28*1$ 的灰度图片，而 Lenet 神经网络的输入为 $32*32*1$ ，故需要对 Lenet 神经网络进行微调。

- ①输入为 $28*28*1$ 的图片大小，为单通道的输入；
- ②进行卷积，卷积核大小为 $5*5*1$ ，个数为 32，步长为 1，全零填充模式；
- ③将卷积结果通过非线性激活函数；
- ④进行池化，池化大小为 $2*2$ ，步长为 2，全零填充模式；
- ⑤进行卷积，卷积核大小为 $5*5*32$ ，个数为 64，步长为 1，全零填充模式；
- ⑥将卷积结果通过非线性激活函数；
- ⑦进行池化，池化大小为 $2*2$ ，步长为 2，全零填充模式；
- ⑧全连接层，进行 10 分类。

Lenet 进行微调后的结构如下所示：



✓Lenet 神经网络在 Mnist 数据集上的实现，主要分为三个部分：前向传播过程（mnist_lenet5_forward.py）、反向传播过程（mnist_lenet5_backword.py）、测试过程（mnist_lenet5_test.py）。

第一，前向传播过程（mnist_lenet5_forward.py）实现对网络中参数和偏置的初始化、定义卷积结构和池化结构、定义前向传播过程。具体代码如下所示：

```
1 #coding:utf-8
2 import tensorflow as tf
3 IMAGE_SIZE = 28
4 NUM_CHANNELS = 1
5 CONV1_SIZE = 5
6 CONV1_KERNEL_NUM = 32
7 CONV2_SIZE = 5
8 CONV2_KERNEL_NUM = 64
9 FC_SIZE = 512
10 OUTPUT_NODE = 10
11
12 def get_weight(shape, regularizer):
13     w = tf.Variable(tf.truncated_normal(shape, stddev=0.1))
14     if regularizer != None: tf.add_to_collection('losses', tf.contrib.layers.l2_regularizer(regularizer)(w))
15     return w
16
17 def get_bias(shape):
18     b = tf.Variable(tf.zeros(shape))
19     return b
20
21 def conv2d(x, w):
22     return tf.nn.conv2d(x, w, strides=[1, 1, 1, 1], padding='SAME')
23
24 def max_pool_2x2(x):
25     return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
26
27 def forward(x, train, regularizer):
28     conv1_w = get_weight([CONV1_SIZE, CONV1_SIZE, NUM_CHANNELS, CONV1_KERNEL_NUM], regularizer)
29     conv1_b = get_bias([CONV1_KERNEL_NUM])
30     conv1 = conv2d(x, conv1_w)
31     relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_b))
32     pool1 = max_pool_2x2(relu1)
33
34     conv2_w = get_weight([CONV2_SIZE, CONV2_SIZE, CONV1_KERNEL_NUM, CONV2_KERNEL_NUM], regularizer)
35     conv2_b = get_bias([CONV2_KERNEL_NUM])
36     conv2 = conv2d(pool1, conv2_w)
37     relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_b))
38     pool2 = max_pool_2x2(relu2)
39
40     pool_shape = pool2.get_shape().as_list()
41     nodes = pool_shape[1] * pool_shape[2] * pool_shape[3]
42     reshaped = tf.reshape(pool2, [pool_shape[0], nodes])
43
44     fcl_w = get_weight([nodes, FC_SIZE], regularizer)
45     fcl_b = get_bias([FC_SIZE])
46     fcl = tf.nn.relu(tf.matmul(reshaped, fcl_w) + fcl_b)
47     if train: fcl = tf.nn.dropout(fcl, 0.5)
48
49     fc2_w = get_weight([FC_SIZE, OUTPUT_NODE], regularizer)
50     fc2_b = get_bias([OUTPUT_NODE])
51     y = tf.matmul(fcl, fc2_w) + fc2_b
52     return y
```

注释：

1) 定义前向传播过程中常用到的参数。

图片大小即每张图片分辨率为 28*28，故 IMAGE_SIZE 取值为 28；Mnist 数据集为灰度图，故输入图片通道数 NUM_CHANNELS 取值为 1；第一层卷积核大小为 5，卷积核个数为 32，故 CONV1_SIZE 取值为 5，CONV1_KERNEL_NUM 取值为 32；第二层卷积核大小为 5，卷积核个数为 64，故 CONV2_SIZE 取值为 5，CONV2_KERNEL_NUM

为 64；全连接层第一层为 512 个神经元，全连接层第二层为 10 个神经元，故 FC_SIZE 取值为 512，OUTPUT_NODE 取值为 10，实现 10 分类输出。

2) 把前向传播过程中，常用到的方法定义为函数，方便调用。

在 mnist_lenet5_forward.py 文件中，定义四个常用函数：权重 w 生成函数、偏置 b 生成函数、卷积层计算函数、最大池化层计算函数，其中，权重 w 生成函数和偏置 b 生成函数与之前的定义相同。

✓①卷积层计算函数描述如下：

tf.nn.conv2d(输入描述[batch, 行分辨率, 列分辨率, 通道数],
 卷积核描述[行分辨率, 列分辨率, 通道数, 卷积核个数],
 核滑动步长[1, 行步长, 列步长, 1],
 填充模式 padding)

例如：

```
tf.nn.conv2d(x=[100, 28, 28, 1], w=[5, 5, 1, 6], strides=[1, 1, 1, 1],  
             padding='SAME')
```

本例表示卷积输入 x 为 28*28*1，一个 batch_size 为 100，卷积核大小为 5*5，卷积核个数为 6，垂直方向步长为 1，水平方向步长为 1，填充方式为全零填充。

✓②最大池化层计算函数描述如下：

tf.nn.max_pool(输入描述[batch, 行分辨率, 列分辨率, 通道数],
 池化核描述[1, 行分辨率, 列分辨率, 1],
 池化核滑动步长[1, 行步长, 列步长, 1],
 填充模式 padding)

例如：

```
tf.nn.max_pool(x=[100, 28, 28, 1], ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],  
              padding='SAME')
```

本例表示卷积输入 x 为 28*28*1，一个 batch_size 为 100，池化核大小用 ksize，第一维和第四维都为 1，池化核大小为 2*2，垂直方向步长为 1，水平方向步长为 1，填充方式为全零填充。

3) 定义前向传播过程

①实现第一层卷积

```
conv1_w = get_weight([CONV1_SIZE, CONV1_SIZE, NUM_CHANNELS,  
                      CONV1_KERNEL_NUM], regularizer)
```

```
conv1_b = get_bias([CONV1_KERNEL_NUM])
```

根据先前定义的参数大小，初始化第一层卷积核和偏置项。

```
conv1 = conv2d(x, conv1_w)
```

实现卷积运算，输入参数为 `x` 和第一层卷积核参数。

```
relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_b))
```

第一层卷积的输出值作为非线性激活函数的输入值，首先通过 `tf.nn.bias_add()` 对卷积后的输出添加偏置，并过 `tf.nn.relu()` 完成非线性激活。

```
pool1 = max_pool_2x2(relu1)
```

根据先前定义的池化函数，将第一层激活后的输出值进行最大池化。

✓ `tf.nn.relu()` 用来实现非线性激活，相比 `sigmoid` 和 `tanh` 函数，`relu` 函数可以实现快速的收敛。

②实现第二层卷积

```
conv2_w = get_weight([CONV2_SIZE, CONV2_SIZE, CONV1_KERNEL_NUM,  
                      CONV2_KERNEL_NUM], regularizer)
```

```
conv2_b = get_bias([CONV2_KERNEL_NUM])
```

初始化第二层卷积层的变量和偏置项，该层每个卷积核的通道数要与上一层卷积核的个数一致。

```
conv2 = conv2d(pool1, conv2_w)
```

实现卷积运算，输入参数为上一层的输出 `pool1` 和第二层卷积核参数。

```
relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_b))
```

实现第二层非线性激活函数。

```
pool2 = max_pool_2x2(relu2)
```

根据先前定义的池化函数，将第二层激活后的输出值进行最大池化。

③将第二层池化层的输出 `pool2` 矩阵转化为全连接层的输入格式即向量形式：

```
pool_shape = pool2.get_shape().as_list()
```

根据 `.get_shape()` 函数得到 `pool2` 输出矩阵的维度，并存入 `list` 中。其中，`pool_shape[0]` 为一个 batch 值。

```
nodes = pool_shape[1] * pool_shape[2] * pool_shape[3]
```

从 list 中依次取出矩阵的长宽及深度，并求三者的乘积，得到矩阵被拉长后的长度。

```
reshaped = tf.reshape(pool2, [pool_shape[0], nodes])
```

将 pool2 转换为一个 batch 的向量再传入后续的全连接。

✓get_shape 函数用于获取一个张量的维度，并且输出张量每个维度上面的值。

例如：

```
A = tf.random_normal(shape=[3,4])
```

```
print A.get_shape()
```

输出结果为：(3, 4)

④实现第三层全连接层：

```
fc1_w = get_weight([nodes, FC_SIZE], regularizer)
```

初始化全连接层的权重，并加入正则化。

```
fc1_b = get_bias([FC_SIZE])
```

初始化全连接层的偏置项。

```
fc1 = tf.nn.relu(tf.matmul(reshaped, fc1_w) + fc1_b)
```

将转换后的 reshaped 向量与权重 fc1_w 做矩阵乘法运算，然后再加上偏置，最后再使用 relu 进行激活。

```
if train: fc1 = tf.nn.dropout(fc1, 0.5)
```

如果是训练阶段，则对该层输出使用 dropout，也就是随机的将该层输出中的一半神经元置为无效，是为了避免过拟合而设置的，一般只在全连接层中使用。

⑤实现第四层全连接层的前向传播过程：

```
fc2_w = get_weight([FC_SIZE, OUTPUT_NODE], regularizer)
```

```
fc2_b = get_bias([OUTPUT_NODE])
```

初始化全连接层对应的变量。

```
y = tf.matmul(fc1, fc2_w) + fc2_b
```

将转换后的 reshaped 向量与权重 fc2_w 做矩阵乘法运算，然后再加上偏置。

```
return y
```

返回输出值有，完成整个前向传播过程，从而实现对 Mnist 数据集的 10 分类。

第二，反向传播过程(mnist_lenet5_backward.py)，完成训练神经网络的参数。

具体代码如下所示：

```
1 #coding:utf-8
2 import tensorflow as tf
3 from tensorflow.examples.tutorials.mnist import input_data
4 import mnist_lenet5_forward
5 import os
6 import numpy as np
7
8 BATCH_SIZE = 100
9 LEARNING_RATE_BASE = 0.005
10 LEARNING_RATE_DECAY = 0.99
11 REGULARIZER = 0.0001
12 STEPS = 50000
13 MOVING_AVERAGE_DECAY = 0.99
14 MODEL_SAVE_PATH = "./model/"
15 MODEL_NAME = "mnist_model"
16
17 def backward(mnist):
18     x = tf.placeholder(tf.float32,[
19         BATCH_SIZE,
20         mnist_lenet5_forward.IMAGE_SIZE,
21         mnist_lenet5_forward.IMAGE_SIZE,
22         mnist_lenet5_forward.NUM_CHANNELS])
23     y_ = tf.placeholder(tf.float32, [None, mnist_lenet5_forward.OUTPUT_NODE])
24     y = mnist_lenet5_forward.forward(x, True, REGULARIZER)
25     global_step = tf.Variable(0, trainable=False)
26
27     ce = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=y, labels=tf.argmax(y_, 1))
28     cem = tf.reduce_mean(ce)
29     loss = cem + tf.add_n(tf.get_collection('losses'))
30
31     learning_rate = tf.train.exponential_decay(
32         LEARNING_RATE_BASE,
33         global_step,
34         mnist.train.num_examples / BATCH_SIZE,
35         LEARNING_RATE_DECAY,
36         staircase=True)
37
38     train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss, global_step=global_step)
39
40     ema = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY, global_step)
41     ema_op = ema.apply(tf.trainable_variables())
42     with tf.control_dependencies([train_step, ema_op]):
43         train_op = tf.no_op(name='train')
44
45     saver = tf.train.Saver()
46
47     with tf.Session() as sess:
48         init_op = tf.global_variables_initializer()
49         sess.run(init_op)
50
51         ckpt = tf.train.get_checkpoint_state(MODEL_SAVE_PATH)
52         if ckpt and ckpt.model_checkpoint_path:
53             saver.restore(sess, ckpt.model_checkpoint_path)
54
55         for i in range(STEPS):
56             xs, ys = mnist.train.next_batch(BATCH_SIZE)
57             reshaped_xs = np.reshape(xs, (
58                 BATCH_SIZE,
59                 mnist_lenet5_forward.IMAGE_SIZE,
60                 mnist_lenet5_forward.IMAGE_SIZE,
61                 mnist_lenet5_forward.NUM_CHANNELS))
62             _, loss_value, step = sess.run([train_op, loss, global_step], feed_dict={x: reshaped_xs, y_: ys})
63             if i % 100 == 0:
64                 print("After %d training step(s), loss on training batch is %g." % (step, loss_value))
65                 saver.save(sess, os.path.join(MODEL_SAVE_PATH, MODEL_NAME), global_step=global_step)
66
67 def main():
68     mnist = input_data.read_data_sets("./data/", one_hot=True)
69     backward(mnist)
70
71 if __name__ == '__main__':
72     main()
```


注释：

1) 定义训练过程中的超参数

规定一个 batch 的数量为 100，故 BATCH_SIZE 取值为 100；设定初始学习率为 0.005. 学习率衰减率为 0.99；最大迭代次数为 50000，故 STEPS 取值为 50000；滑动平均衰减率设置为 0.99，并规定模型保存路径以及保存的模型名称。

2) 完成反向传播过程

①给 x, y_是占位

```
x = tf.placeholder(tf.float32, [
    BATCH_SIZE,
    mnist_lenet5_forward.IMAGE_SIZE,
    mnist_lenet5_forward.IMAGE_SIZE,
    mnist_lenet5_forward.NUM_CHANNELS])
y_ = tf.placeholder(tf.float32, [None, mnist_lenet5_forward.
    OUTPUT_NODE])
```

x, y_是定义的占位符，指定参数为浮点型。由于卷积层输入为四阶张量，故 x 的占位符表示为上述形式，第一阶表示每轮喂入的图片数量，第二阶和第三阶分别表示图片的行分辨率和列分辨率，第四阶表示通道数。

✓ `x = tf.placeholder(dtype, shape, name=None)`

`tf.placeholder()` 函数有三个参数，`dtype` 表示数据类型，常用的类型为 `tf.float32`, `tf.float64` 等数值类型，`shape` 表示数据形状，`name` 表示名称。

②调用前向传播过程

```
y = mnist_lenet5_forward.forward(x, True, REGULARIZER)
```

调用前向传播网络得到维度为 10 的 tensor。

③求含有正则化的损失值

```
global_step = tf.Variable(0, trainable=False)
```

声明一个全局计数器，并输出化为 0

```
ce = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=y,
    labels=tf.argmax(y_, 1))
```

对网络最后一层的输出 y 做 softmax，求取输出属于某一类的概率，结果为一个

num_classes 大小的向量，再将此向量和实际标签值做交叉熵，返回一个向量值。

```
cem = tf.reduce_mean(ce)
```

通过 tf.reduce_mean() 函数对得到的向量求均值，得到 loss。

```
loss = cem + tf.add_n(tf.get_collection('losses'))
```

添加正则化中的 losses 值到 loss 中。

```
√ sparse_softmax_cross_entropy_with_logits(_sentinel=None,  
                                              labels=None,  
                                              logits=None,  
                                              name=None)
```

此函数的参数 logits 为神经网络最后一层的输出，它的大小为 [batch_size, num_classes]，参数 labels 表示实际标签值，大小为 [batch_size, num_classes]。

第一步是先对网络最后一层的输出做一个 softmax，输出为属于某一属性的概率向量；再将概率向量与实际标签向量做交叉熵，返回向量。

```
√ tf.reduce_mean( input_tensor,  
                  reduction_indices=None,  
                  keep_dims=False,  
                  name=None)
```

此函数表示对得到的向量求取均值。参数 input_tensor 表示要减少的张量；参数 reduction_indices 表示求取均值的维度；参数 keep_dims 含义为：如果为 true，则保留长度为 1 的缩小尺寸。name 表示操作的名称。

例如：

```
x = tf.constant([[1., 1.], [2., 2.]])  
tf.reduce_mean(x)      #表示对向量整体求均值 1.5  
tf.reduce_mean(x, 0)   #表示对向量在列上求均值[1.5, 1.5]  
tf.reduce_mean(x, 1)   #表示对向量在行上求均值[1., 2.]
```

④实现指数衰减学习率

```
√ learning_rate = tf.train.exponential_decay(  
    LEARNING_RATE_BASE,  
    global_step,
```

```
mnist.train.num_examples / BATCH_SIZE,
LEARNING_RATE_DECAY,
staircase=True)
```

`tf.train.exponential_decay` 函数中参数 `LEARNING_RATE_BASE` 表示初始学习率，参数 `LEARNING_RATE_DECAY` 表示学习率衰减速率。实现指数级的减小学习率，可以让模型在训练的前期快速接近较优解，又可以保证模型在训练后期不会有太大波动。其中，当 `staircase=True` 时，为阶梯形衰减， $(\text{global_step} / \text{decay_steps})$ 则被转化为整数；当 `staircase=False` 时，为曲线形衰减，以此根据 `staircase` 来选择不同的衰减方式。

计算公式为：

```
decayed_learning_rate=learning_rate*decay_rate(global_step/decay_
steps)
```

```
√ train_step=tf.train.GradientDescentOptimizer(learning_rate).
```

```
minimize(loss, global_step=global_step)
```

此函数的参数 `learning_rate` 为传入的学习率，构造一个实现梯度下降算法的优化器，再通过使用 `minimize` 更新存储要训练的变量的列表来减小 `loss`。

⑤实现滑动平均模型

```
√ ema = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY,
global_step)
```

```
ema_op = ema.apply(tf.trainable_variables())
```

`tf.train.ExponentialMovingAverage` 函数采用滑动平均的方法更新参数。此函数的参数 `MOVING_AVERAGE_DECAY` 表示衰减速率，用于控制模型的更新速度；此函数维护一个影子变量，影子变量初始值为变量初始值。影子变量值的更新方式如下： $\text{shadow_variable} = \text{decay} * \text{shadow_variable} + (1 - \text{decay}) * \text{variable}$ 。

其中，`shadow_variable` 是影子变量，`variable` 表示待更新的变量，`decay` 为衰减速率。`decay` 一般设为接近于 1 的数 (0.99, 0.999)，`decay` 越大模型越稳定。

⑥将 `train_step` 和 `ema_op` 两个训练操作绑定到 `train_op` 上

```
with tf.control_dependencies([train_step, ema_op]):
```

```
train_op = tf.no_op(name='train')
```

⑦实例化一个保存和恢复变量的 saver，并创建一个会话

```
saver = tf.train.Saver()
```

```
with tf.Session() as sess:
```

```
    init_op = tf.global_variables_initializer()
```

```
    sess.run(init_op)
```

创建一个会话，并通过 python 中的上下文管理器来管理这个会话，初始化计算图中的变量，并用 sess.run 实现初始化。

```
    ckpt = tf.train.get_checkpoint_state(MODEL_SAVE_PATH)
```

```
    if ckpt and ckpt.model_checkpoint_path:
```

```
        saver.restore(sess, ckpt.model_checkpoint_path)
```

通过 checkpoint 文件定位到最新保存的模型，若文件存在，则加载最新的模型。

```
    for i in range(STEPS):
```

```
        xs, ys = mnist.train.next_batch(BATCH_SIZE)
```

```
        reshaped_xs = np.reshape(xs, (
```

```
            BATCH_SIZE,
```

```
            mnist_lenet5_forward.IMAGE_SIZE,
```

```
            mnist_lenet5_forward.IMAGE_SIZE,
```

```
            mnist_lenet5_forward.NUM_CHANNELS))
```

读取一个 batch 数据，将输入数据 xs 转成与网络输入相同形状的矩阵。

```
    _, loss_value, step = sess.run([train_op, loss, global_step],
```

```
    feed_dict={x: reshaped_xs, y_: ys}))
```

喂入训练图像和标签，开始训练。

```
    if i % 100 == 0:
```

```
        print("After %d training step(s), loss on training batch is %g." %
```

```
            (step, loss_value))
```

每迭代 100 次打印 loss 信息，并保存最新的模型。

训练 Lenet 网络后，输出结果如下：

```
lab@aialab:~/CNN$ python mnist_lenet5_forward.py
lab@aialab:~/CNN$ python mnist_lenet5_backward.py
Extracting ./data/train-images-idx3-ubyte.gz
Extracting ./data/train-labels-idx1-ubyte.gz
Extracting ./data/t10k-images-idx3-ubyte.gz
Extracting ./data/t10k-labels-idx1-ubyte.gz
After 1302 training step(s), loss on training batch is 1.04771.
After 1402 training step(s), loss on training batch is 0.92521.
After 1502 training step(s), loss on training batch is 0.907243.
After 1602 training step(s), loss on training batch is 0.875127.
```

由运行结果可以看出，损失值在不断减小，且可以实现断点续训。

第三，测试过程（mnist_lenet5_test.py），对 Mnist 数据集中的测试数据进行预测，测试模型准确率。具体代码如下所示：

```
1 #coding:utf-8
2 import time
3 import tensorflow as tf
4 from tensorflow.examples.tutorials.mnist import input_data
5 import mnist_lenet5_forward
6 import mnist_lenet5_backward
7 import numpy as np
8
9 TEST_INTERVAL_SECS = 5
10
11 def test(mnist):
12     with tf.Graph().as_default() as g:
13         x = tf.placeholder(tf.float32,[
14             mnist.test.num_examples,
15             mnist_lenet5_forward.IMAGE_SIZE,
16             mnist_lenet5_forward.IMAGE_SIZE,
17             mnist_lenet5_forward.NUM_CHANNELS])
18         y_ = tf.placeholder(tf.float32, [None, mnist_lenet5_forward.OUTPUT_NODE])
19         y = mnist_lenet5_forward.forward(x,False,None)
20
21         ema = tf.train.ExponentialMovingAverage(mnist_lenet5_backward.MOVING_AVERAGE_DECAY)
22         ema_restore = ema.variables_to_restore()
23         saver = tf.train.Saver(ema_restore)
24
25         correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
26         accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
27
28     while True:
29         with tf.Session() as sess:
30             ckpt = tf.train.get_checkpoint_state(mnist_lenet5_backward.MODEL_SAVE_PATH)
31             if ckpt and ckpt.model_checkpoint_path:
32                 saver.restore(sess, ckpt.model_checkpoint_path)
33
34                 global_step = ckpt.model_checkpoint_path.split('/')[-1].split('-')[-1]
35                 reshaped_x = np.reshape(mnist.test.images,(
36                     mnist.test.num_examples,
37                     mnist_lenet5_forward.IMAGE_SIZE,
38                     mnist_lenet5_forward.IMAGE_SIZE,
39                     mnist_lenet5_forward.NUM_CHANNELS))
40                 accuracy_score = sess.run(accuracy, feed_dict={x:reshaped_x,y_:mnist.test.labels})
41                 print("After %s training step(s), test accuracy = %g" % (global_step, accuracy_score))
42             else:
43                 print('No checkpoint file found')
44                 return
45             time.sleep(TEST_INTERVAL_SECS)
46
47 def main():
48     mnist = input_data.read_data_sets("./data/", one_hot=True)
49     test(mnist)
50
51 if __name__ == '__main__':
52     main()
```

注释：

1) 在测试程序中使用的的是训练好的网络，故不使用 dropout，而是让所有神经元都参与运算，从而输出识别准确率。

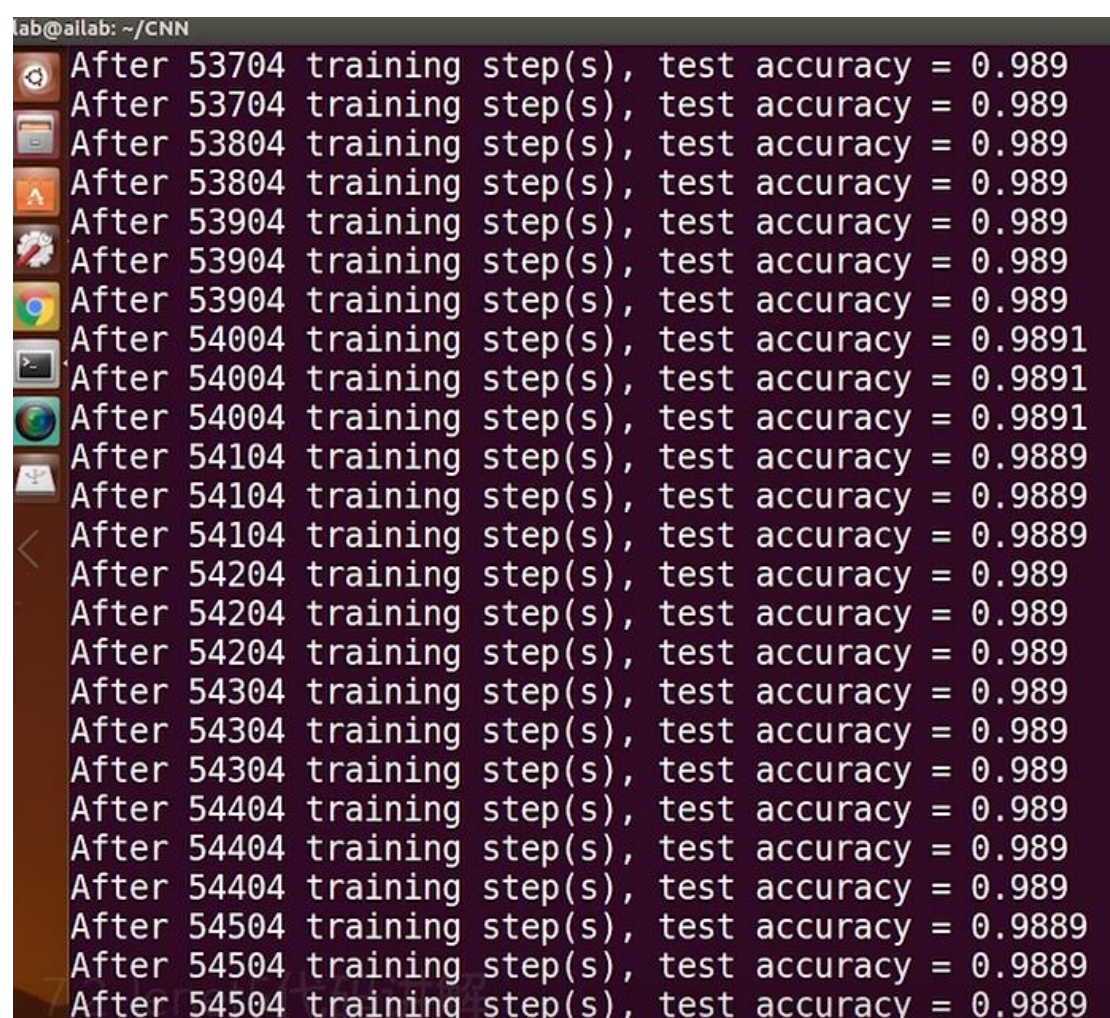
```
2) correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))  
√ tf.equal(x, y)
```

此函数用于判断函数的两个参数 x 与 y 是否相等，一般 x 表示预测值， y 表示实际值。

```
3) accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

求平均得到预测准确率。

在测试集上，输出结果如下：

A terminal window titled 'lab@allab: ~/CNN' displays the output of a training process. The output consists of 20 lines, each starting with 'After' followed by a training step number and the test accuracy. The accuracy values are mostly 0.989, with some steps showing 0.9891 or 0.9889. The steps range from 53704 to 54504 in increments of 100.

```
lab@allab: ~/CNN  
After 53704 training step(s), test accuracy = 0.989  
After 53704 training step(s), test accuracy = 0.989  
After 53804 training step(s), test accuracy = 0.989  
After 53804 training step(s), test accuracy = 0.989  
After 53904 training step(s), test accuracy = 0.989  
After 53904 training step(s), test accuracy = 0.989  
After 53904 training step(s), test accuracy = 0.989  
After 54004 training step(s), test accuracy = 0.9891  
After 54004 training step(s), test accuracy = 0.9891  
After 54004 training step(s), test accuracy = 0.9891  
After 54104 training step(s), test accuracy = 0.9889  
After 54104 training step(s), test accuracy = 0.9889  
After 54104 training step(s), test accuracy = 0.9889  
After 54204 training step(s), test accuracy = 0.989  
After 54204 training step(s), test accuracy = 0.989  
After 54204 training step(s), test accuracy = 0.989  
After 54304 training step(s), test accuracy = 0.989  
After 54304 training step(s), test accuracy = 0.989  
After 54304 training step(s), test accuracy = 0.989  
After 54404 training step(s), test accuracy = 0.989  
After 54404 training step(s), test accuracy = 0.989  
After 54404 training step(s), test accuracy = 0.989  
After 54504 training step(s), test accuracy = 0.9889  
After 54504 training step(s), test accuracy = 0.9889  
After 54504 training step(s), test accuracy = 0.9889
```

由输出结果表明，在测试集上的准确率可以达到 99%左右，Lenet 性能良好。

课程中 Lenet5 源码的全文注释

```
mnist_lenet5_forward.py
```

```
#coding:utf-8
```

```
import tensorflow as tf
```

```
# 设定神经网络的超参数
```

```
# 定义神经网络可以接收的图片的尺寸和通道数
```

```
IMAGE_SIZE = 28
```

```
NUM_CHANNELS = 1
```

```
# 定义第一层卷积核的大小和个数
```

```
CONV1_SIZE = 5
```

```
CONV1_KERNEL_NUM = 32
```

```
# 定义第二层卷积核的大小和个数
```

```
CONV2_SIZE = 5
```

```
CONV2_KERNEL_NUM = 64
```

```
# 定义第三层全连接层的神经元个数
```

```
FC_SIZE = 512
```

```
# 定义第四层全连接层的神经元个数
```

```
OUTPUT_NODE = 10
```

```
# 定义初始化网络权重函数
```

```
def get_weight(shape, regularizer):
```

```
    '''
```

```
    args:
```

```
        shape: 生成张量的维度
```



```

        regularizer: 正则化项的权重

'''

# tf.truncated_normal 生成去掉过大偏离点的正态分布随机数的张量，stddev 是指定标准差

w = tf.Variable(tf.truncated_normal(shape,stddev=0.1))

# 为权重加入 L2 正则化，通过限制权重的大小，使模型不会随意拟合训练数据中的随机噪音

if regularizer != None: tf.add_to_collection('losses', tf.contrib.layers.l2_regularizer(regularizer)(w))

return w


# 定义初始化偏置项函数

def get_bias(shape):

    '''

    args:

        shape: 生成张量的维度

    '''

    b = tf.Variable(tf.zeros(shape)) # 统一将 bias 初始化为 0

    return b


# 定义卷积计算函数

def conv2d(x,w):

    '''

    args:

        x: 一个输入 batch

        w: 卷积层的权重

    '''

    # strides 表示卷积核在不同维度上的移动步长为 1，第一维和第四维一定是 1，这是因为卷积层的步
    长只对矩阵的长和宽有效；

    # padding='SAME'表示使用全 0 填充，而'VALID'表示不填充

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


# 定义最大池化操作函数

```

```

def max_pool_2x2(x):
    '''
    args:
        x: 一个输入 batch
    '''
    # ksize 表示池化过滤器的边长为 2, strides 表示过滤器移动步长是 2, 'SAME'提供使用全 0 填充
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

#定义前向传播的过程

def forward(x, train, regularizer):
    '''
    args:
        x: 一个输入 batch
        train: 用于区分训练过程 True, 测试过程 False
        regularizer: 正则化项的权重
    '''
    # 实现第一层卷积层的前向传播过程
    conv1_w = get_weight([CONV1_SIZE, CONV1_SIZE, NUM_CHANNELS, CONV1_KERNEL_NUM],
regularizer) # 初始化卷积核

    conv1_b = get_bias([CONV1_KERNEL_NUM]) # 初始化偏置项

    conv1 = conv2d(x, conv1_w) # 实现卷积运算

    relu1 = tf.nn.relu(tf.nn.bias_add(conv1, conv1_b)) # 对卷积后的输出添加偏置, 并过 relu 非线性激活
函数

    pool1 = max_pool_2x2(relu1) # 将激活后的输出进行最大池化

    # 实现第二层卷积层的前向传播过程, 并初始化卷积层的对应变量
    conv2_w = get_weight([CONV2_SIZE, CONV2_SIZE, CONV1_KERNEL_NUM,
CONV2_KERNEL_NUM],regularizer) # 该层每个卷积核的通道数要与上一层卷积核的个数一致

```

```

conv2_b = get_bias([CONV2_KERNEL_NUM])

conv2 = conv2d(pool1, conv2_w) # 该层的输入就是上一层的输出 pool1

relu2 = tf.nn.relu(tf.nn.bias_add(conv2, conv2_b))

pool2 = max_pool_2x2(relu2)

# 将上一池化层的输出 pool2（矩阵）转化为下一层全连接层的输入格式（向量）

pool_shape = pool2.get_shape().as_list() # 得到 pool2 输出矩阵的维度, 并存入 list 中, 注意 pool_shape[0]
是一个 batch 的值

nodes = pool_shape[1] * pool_shape[2] * pool_shape[3] # 从 list 中依次取出矩阵的长宽及深度, 并求三
者的乘积就得到矩阵被拉长后的长度

reshaped = tf.reshape(pool2, [pool_shape[0], nodes]) # 将 pool2 转换为一个 batch 的向量再传入后续的
全连接

# 实现第三层全连接层的前向传播过程

fc1_w = get_weight([nodes, FC_SIZE], regularizer) # 初始化全连接层的权重, 并加入正则化

fc1_b = get_bias([FC_SIZE]) # 初始化全连接层的偏置项

# 将转换后的 reshaped 向量与权重 fc1_w 做矩阵乘法运算, 然后再加上偏置, 最后再使用 relu 进行
激活

fc1 = tf.nn.relu(tf.matmul(reshaped, fc1_w) + fc1_b)

# 如果是训练阶段, 则对该层输出使用 dropout, 也就是随机的将该层输出中的一半神经元置为无效,
是为了避免过拟合而设置的, 一般只在全连接层中使用

if train: fc1 = tf.nn.dropout(fc1, 0.5)

# 实现第四层全连接层的前向传播过程, 并初始化全连接层对应的变量

fc2_w = get_weight([FC_SIZE, OUTPUT_NODE], regularizer)

fc2_b = get_bias([OUTPUT_NODE])

y = tf.matmul(fc1, fc2_w) + fc2_b

return y

```

```
mnist_lenet5_backward.py
```

```
#coding:utf-8
```

```
import tensorflow as tf
```

```
from tensorflow.examples.tutorials.mnist import input_data
```

```
import mnist_lenet5_forward
```

```
import os
```

```
import numpy as np
```

```
# 定义训练过程中的超参数
```

```
BATCH_SIZE = 100 # 一个 batch 的数量
```

```
LEARNING_RATE_BASE = 0.005 # 初始学习率
```

```
LEARNING_RATE_DECAY = 0.99 # 学习率的衰减率
```

```
REGULARIZER = 0.0001 # 正则化项的权重
```

```
STEPS = 50000 # 最大迭代次数
```

```
MOVING_AVERAGE_DECAY = 0.99 # 滑动平均的衰减率
```

```
MODEL_SAVE_PATH="./model/" # 保存模型的路径
```

```
MODEL_NAME="mnist_model" # 模型命名
```

```
# 训练过程
```

```
def backward(mnist):
```

x, y 是定义的占位符，需要指定参数的类型，维度（要和网络的输入与输出维度一致），类似于函数的形参，运行时必须传入值

```
    x = tf.placeholder(tf.float32,[
```

```
        BATCH_SIZE,
```

```
        mnist_lenet5_forward.IMAGE_SIZE,
```

```
        mnist_lenet5_forward.IMAGE_SIZE,
```

```
        mnist_lenet5_forward.NUM_CHANNELS])
```

```
y_ = tf.placeholder(tf.float32, [None, mnist_lenet5_forward.OUTPUT_NODE])
```

```
y = mnist_lenet5_forward.forward(x, True, REGULARIZER) # 调用前向传播网络得到维度为  
10 的 tensor
```

```
global_step = tf.Variable(0, trainable=False) # 声明一个全局计数器，并输出化为 0
```

```
# 先是对网络最后一层的输出 y 做 softmax，通常是求取输出属于某一类的概率，其实就是一个  
num_classes 大小的向量，
```

```
# 再将此向量和实际标签值做交叉熵，需要说明的是该函数返回的是一个向量
```

```
ce = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=y, labels=tf.argmax(y_, 1))
```

```
cem = tf.reduce_mean(ce) # 再对得到的向量求均值就得到 loss
```

```
loss = cem + tf.add_n(tf.get_collection('losses')) # 添加正则化中的 losses
```

```
# 实现指数级的减小学习率，可以让模型在训练的前期快速接近较优解，又可以保证模型在训  
练后期不会有太大波动
```

```
# 计算公式： $\text{decayed\_learning\_rate} = \text{learning\_rate} * \text{decay\_rate}^{(\text{global\_step} / \text{decay\_steps})}$ 
```

```
learning_rate = tf.train.exponential_decay(
```

```
    LEARNING_RATE_BASE,
```

```
    global_step,
```

```
    mnist.train.num_examples / BATCH_SIZE,
```

```
    LEARNING_RATE_DECAY,
```

```
    staircase=True) # 当 staircase=True 时，(global_step/decay_steps) 则被转化为整数，以
```

```
以此来选择不同的衰减方式
```

```
# 传入学习率，构造一个实现梯度下降算法的优化器，再通过使用 minimize 更新存储要训练的  
变量的列表来减小 loss
```

```
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss,  
global_step=global_step)
```

```
# 实现滑动平均模型，参数 MOVING_AVERAGE_DECAY 用于控制模型更新的速度。训练过
```

程中会对每一个变量维护一个影子变量，这个影子变量的初始值

```
# 就是相应变量的初始值，每次变量更新时，影子变量就会随之更新
```

```
ema = tf.train.ExponentialMovingAverage(MOVING_AVERAGE_DECAY, global_step)
```

```
ema_op = ema.apply(tf.trainable_variables())
```

```
with tf.control_dependencies([train_step, ema_op]): # 将 train_step 和 ema_op 两个训练操作绑定到 train_op 上
```

```
train_op = tf.no_op(name='train')
```

```
saver = tf.train.Saver() # 实例化一个保存和恢复变量的 saver
```

```
with tf.Session() as sess: # 创建一个会话，并通过 python 中的上下文管理器来管理这个会话
```

```
init_op = tf.global_variables_initializer() # 初始化计算图中的变量
```

```
sess.run(init_op)
```

```
ckpt = tf.train.get_checkpoint_state(MODEL_SAVE_PATH) # 通过 checkpoint 文件定位到最新保存的模型
```

```
if ckpt and ckpt.model_checkpoint_path:
```

```
saver.restore(sess, ckpt.model_checkpoint_path) # 加载最新的模型
```

```
for i in range(STEPS):
```

```
xs, ys = mnist.train.next_batch(BATCH_SIZE) # 读取一个 batch 的数据
```

```
reshaped_xs = np.reshape(xs, ( # 将输入数据 xs 转换成与网络输入相同形状的矩阵
```

```
BATCH_SIZE,
```

```
mnist_lenet5_forward.IMAGE_SIZE,
```

```
mnist_lenet5_forward.IMAGE_SIZE,
```

```
mnist_lenet5_forward.NUM_CHANNELS))
```

```
# 喂入训练图像和标签，开始训练
```

```
_, loss_value, step = sess.run([train_op, loss, global_step], feed_dict={x: reshaped_xs,
```

```
y_: ys})
```

```
if i % 100 == 0: # 每迭代 100 次打印 loss 信息，并保存最新的模型
```

```
print("After %d training step(s), loss on training batch is %g." % (step,
loss_value))
```

```
saver.save(sess, os.path.join(MODEL_SAVE_PATH, MODEL_NAME),
global_step=global_step)
```

```
def main():
```

```
    mnist = input_data.read_data_sets('./data/', one_hot=True) # 读入 mnist 数据
```

```
    backward(mnist)
```

```
if __name__ == '__main__':
```

```
    main()
```

```
mnist_lenet5_test.py
```

```
#coding:utf-8
```

```
import time
```

```
import tensorflow as tf
```

```
from tensorflow.examples.tutorials.mnist import input_data
```

```
import mnist_lenet5_forward
```

```
import mnist_lenet5_backward
```

```
import numpy as np
```

```
TEST_INTERVAL_SECS = 5
```

```
def test(mnist):
```

创建一个默认图，在该图中执行以下操作（多数操作和 train 中一样，就不再重复解释，大家对照学习即可）

```
with tf.Graph().as_default() as g:
```



```

x = tf.placeholder(tf.float32,[

    mnist.test.num_examples,

    mnist_lenet5_forward.IMAGE_SIZE,

    mnist_lenet5_forward.IMAGE_SIZE,

    mnist_lenet5_forward.NUM_CHANNELS])

y_ = tf.placeholder(tf.float32, [None, mnist_lenet5_forward.OUTPUT_NODE])

y = mnist_lenet5_forward.forward(x,False,None)


ema                                                                    =

tf.train.ExponentialMovingAverage(mnist_lenet5_backward.MOVING_AVERAGE_DECAY)

ema_restore = ema.variables_to_restore()

saver = tf.train.Saver(ema_restore)


correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1)) # 判断预测值和实际值是否
相同

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32)) # 求平均得到准确率


while True:

    with tf.Session() as sess:

        ckpt                                                            =

tf.train.get_checkpoint_state(mnist_lenet5_backward.MODEL_SAVE_PATH)

        if ckpt and ckpt.model_checkpoint_path:

            saver.restore(sess, ckpt.model_checkpoint_path)


            # 根据读入的模型名字切分出该模型是属于迭代了多少次保存的

            global_step = ckpt.model_checkpoint_path.split('/')[-1].split('-')[-1]

            reshaped_x = np.reshape(mnist.test.images,(

                mnist.test.num_examples,

                mnist_lenet5_forward.IMAGE_SIZE,

                mnist_lenet5_forward.IMAGE_SIZE,

```

```

mnist_lenet5_forward.NUM_CHANNELS))

accuracy_score = sess.run(accuracy,

feed_dict={x:reshaped_x,y_:mnist.test.labels}) # 计算出测试集上准确率

print("After %s training step(s), test accuracy = %g" % (global_step,
accuracy_score))

else:

print('No checkpoint file found')

return

time.sleep(TEST_INTERVAL_SECS) # 每隔 5 秒寻找一次是否有最新的模型

def main():

mnist = input_data.read_data_sets("./data/", one_hot=True)

test(mnist)

if __name__ == '__main__':

main()

```