计算机科学与技术学院神经网络与深度学习课程实验报告

实验题目：了解卷积神经网络的架构		学号：201900150221
日期：10.28	班级： 19 智能	姓名： 张进华
Email：zjh15117117428@163.com		

实验目的：

了解卷积神经网络的架构网络并通过数据训练这些模型获得实践。实现两个子任务：构建卷积神经网络模型和应用，使用 Keras 构建残差网络（可选）

实验软件和硬件环境：

Visual studio Code python 3.9.7

Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz   2.59 GHz

实验步骤

步骤一 Convolutional Neural Networks: Step by Step

一个卷积层将一个输入转换成一个不同大小的输出体积，首先实现两个辅助函数：一个用于零填充，另一个用于计算卷积函数本身。

1.1 - Zero-Padding

填充的主要好处如下：

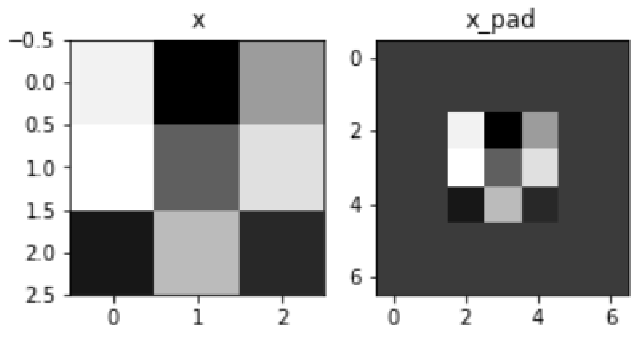
允许使用 CONV 层而不必缩小体积的高度和宽度。 这对于构建更深的网络很重要，否则当进入更深的层时，高度/宽度会缩小。 一个重要的特殊情况是“相同”卷积，其中高度/宽度在一层之后完全保留。

在图像的边界保留更多信息。 如果没有填充，下一层的值很少会受到作为图像边缘的像素的影响。

零填充的实现代码关键部分如下：

```
### START CODE HERE ### (≈ 1 line)
X_pad = np.pad(X,((0,0),(pad,pad),(pad,pad),(0,0)),'constant',constant_values =(0))
### END CODE HERE ###
```

```
x.shape = (4, 3, 3, 2)
x_pad.shape = (4, 7, 7, 2)
x[1,1] = [[ 0.90085595 -0.68372786]
 [-0.12289023 -0.93576943]
 [-0.26788808  0.53035547]]
x_pad[1,1] = [[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
<matplotlib.image.AxesImage at 0x20d3e8bb0a0>
```



1.2 - Single step of convolution

实现一个卷积步骤，将过滤器应用到输入的单个位置，用于构建一个卷积单元，实现代码关键部分如下：

```
    # Element-wise product between a_slice and W. Do not add the bias yet.
    s = np.multiply(a_slice_prev,W)
    # Sum over all entries of the volume s
    Z = np.sum(s)
    # Add bias b to Z. Cast b to a float() so that Z results in a scalar value
    Z = Z+b
    ### END CODE HERE ###
```

1.3 - Convolutional Neural Networks - Forward pass

在前向传递中，采用许多过滤器并将它们与输入进行卷积。 每个“卷积”都会提供一个 2D 矩阵输出，将堆叠这些输出以获得 3D 张量，其中输出部分的矩阵维度计算公式如下：

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_C = \text{number of filters used in the convolution}$$

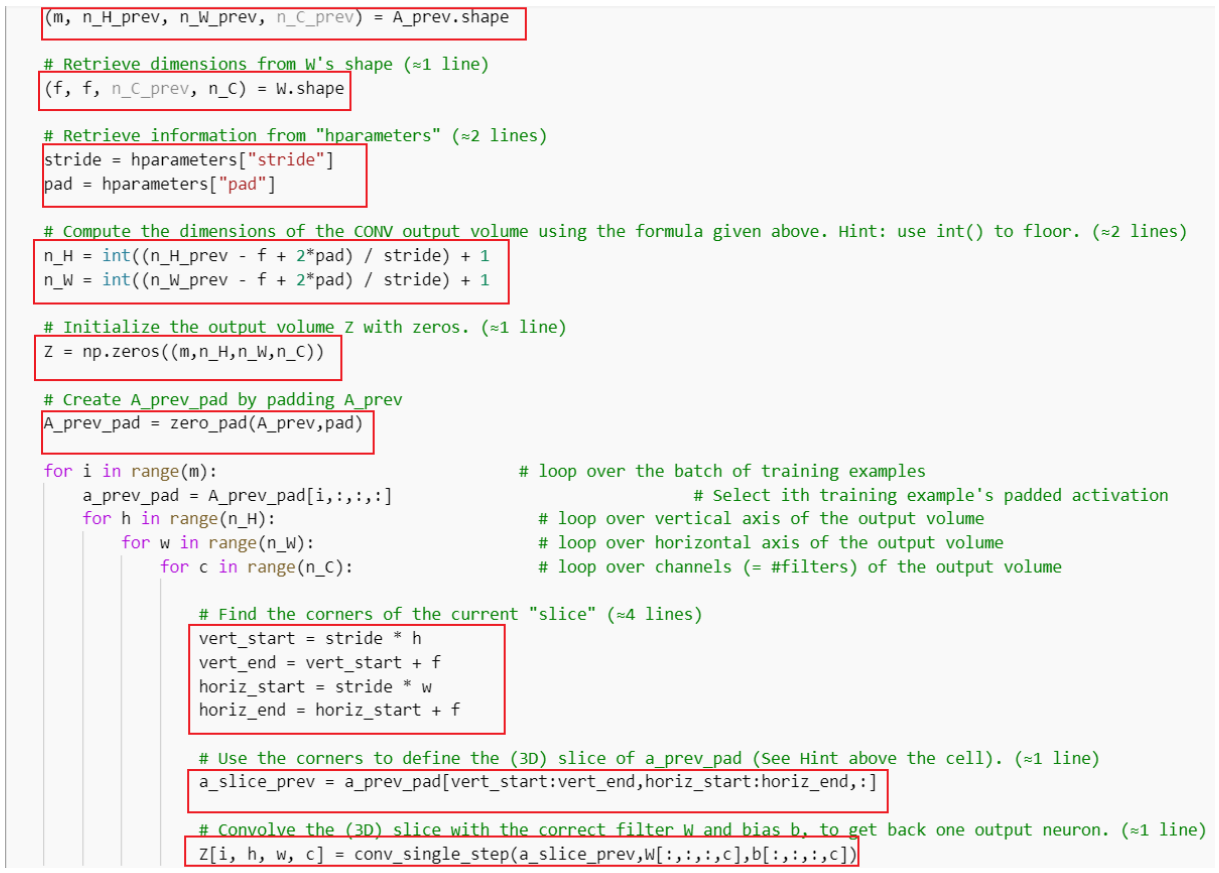
erything with for-loops

开始 for loop，4 个 for 循环分别遍历

①样本，②输出图像的高度，③输出图像的宽度，④每个通道，即当前层卷积核的个数

先定位当前的切片位置，得到 vert_start 和 vert_end。

定位好之后，在输入的 a 图像中取出要进行加卷积的部分，并执行单步卷积，保存到结果 Z 中，实现代码关键部分如下：



```
(m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

# Retrieve dimensions from W's shape (≈1 line)
(f, f, n_C_prev, n_C) = W.shape

# Retrieve information from "hparameters" (≈2 lines)
stride = hparameters["stride"]
pad = hparameters["pad"]

# Compute the dimensions of the CONV output volume using the formula given above. Hint: use int() to floor. (≈2 lines)
n_H = int((n_H_prev - f + 2*pad) / stride) + 1
n_W = int((n_W_prev - f + 2*pad) / stride) + 1

# Initialize the output volume Z with zeros. (≈1 line)
Z = np.zeros((m,n_H,n_W,n_C))

# Create A_prev_pad by padding A_prev
A_prev_pad = zero_pad(A_prev,pad)

for i in range(m):                               # loop over the batch of training examples
    a_prev_pad = A_prev_pad[i,:,:,:]                               # Select ith training example's padded activation
    for h in range(n_H):                           # loop over vertical axis of the output volume
        for w in range(n_W):                       # loop over horizontal axis of the output volume
            for c in range(n_C):                   # loop over channels (= #filters) of the output volume

                # Find the corners of the current "slice" (≈4 lines)
                vert_start = stride * h
                vert_end = vert_start + f
                horiz_start = stride * w
                horiz_end = horiz_start + f

                # Use the corners to define the (3D) slice of a_prev_pad (See Hint above the cell). (≈1 line)
                a_slice_prev = a_prev_pad[vert_start:vert_end,horiz_start:horiz_end,:]

                # Convolve the (3D) slice with the correct filter W and bias b, to get back one output neuron. (≈1 line)
                Z[i, h, w, c] = conv_single_step(a_slice_prev,W[:,:,:,c],b[:,:,:,c])
```

1.4 - Pooling layer

池化（POOL）层减少了输入的高度和宽度，有助于减少计算。

1.4.1 - Forward Pooling

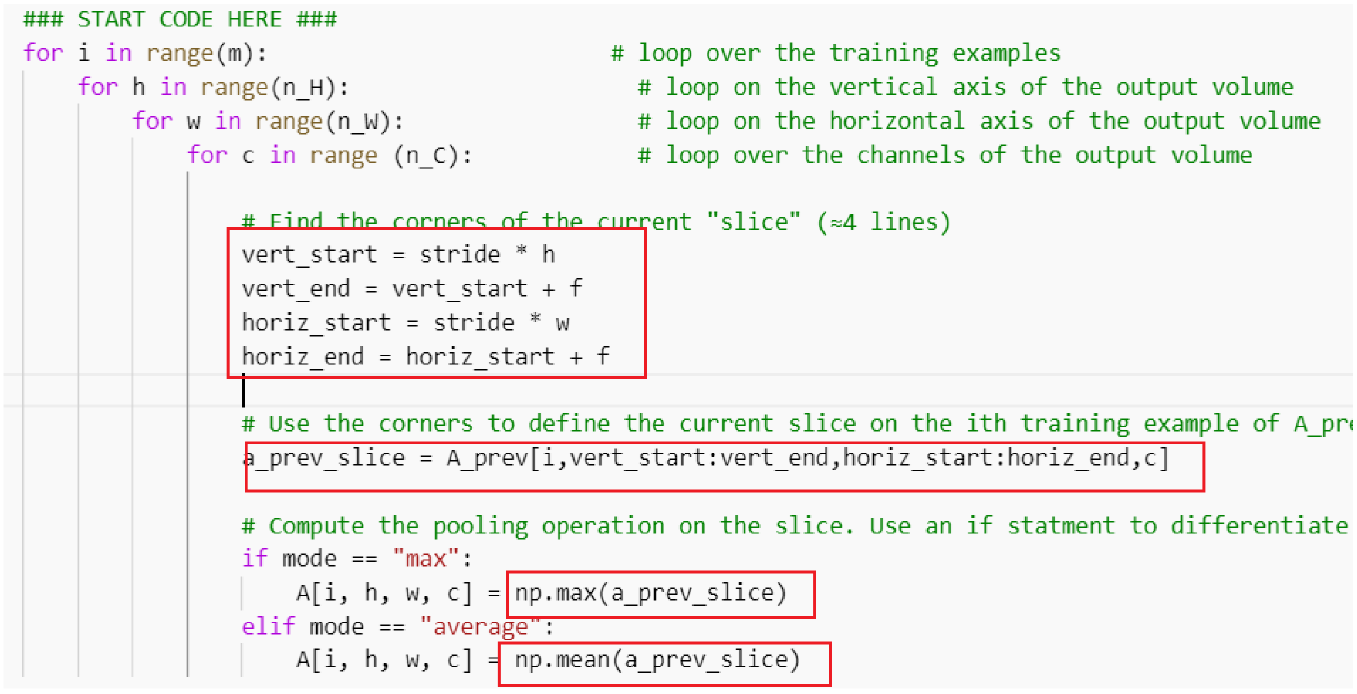
由于没有填充，将池化的输出形状绑定到输入形状的公式为：

$$n_H = \lfloor \frac{n_{H_{prev}} - f}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f}{stride} \rfloor + 1$$

$$n_C = n_{C_{prev}}$$

池化层前向传播过程实现代码如下，有两种模式，最大池化和平均池化，开始 for loop，4 个 for 循环分别遍历①样本，②输出图像的高度，③输出图像的宽度，④每个通道，即当前层卷积核的个数，先定位当前的切片位置，得到 vert_start 和 vert_end。定位好之后，在输入的 a 图像中取出要进行池化操作的部分，并执行池化操作，保存到结果 Z 中，实现代码关键部分如下：



```python
### START CODE HERE ###
for i in range(m):                         # loop over the training examples
    for h in range(n_H):                     # loop on the vertical axis of the output volume
        for w in range(n_W):                 # loop on the horizontal axis of the output volume
            for c in range (n_C):            # loop over the channels of the output volume

                # Find the corners of the current "slice" (≈4 lines)
                vert_start = stride * h
                vert_end = vert_start + f
                horiz_start = stride * w
                horiz_end = horiz_start + f

                # Use the corners to define the current slice on the ith training example of A_pre
                a_prev_slice = A_prev[i,vert_start:vert_end,horiz_start:horiz_end,c]

                # Compute the pooling operation on the slice. Use an if statment to differentiate
                if mode == "max":
                    A[i, h, w, c] = np.max(a_prev_slice)
                elif mode == "average":
                    A[i, h, w, c] = np.mean(a_prev_slice)
```

1.5 - Backpropagation in convolutional neural networks

了解卷积网络中的反向传播

1.5.1 - Convolutional layer backward pass

实现 CONV 层的反向传递

1.5.1.1 - Computing dA:

相对于特定过滤器 W_c 和给定训练示例的成本计算 dA 的公式：

$$dA += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw}$$

1.5.1.2 - Computing dW:

关于损失计算 dW_c（dW_c 是一个滤波器的导数）的公式：

$$dW_c += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw}$$

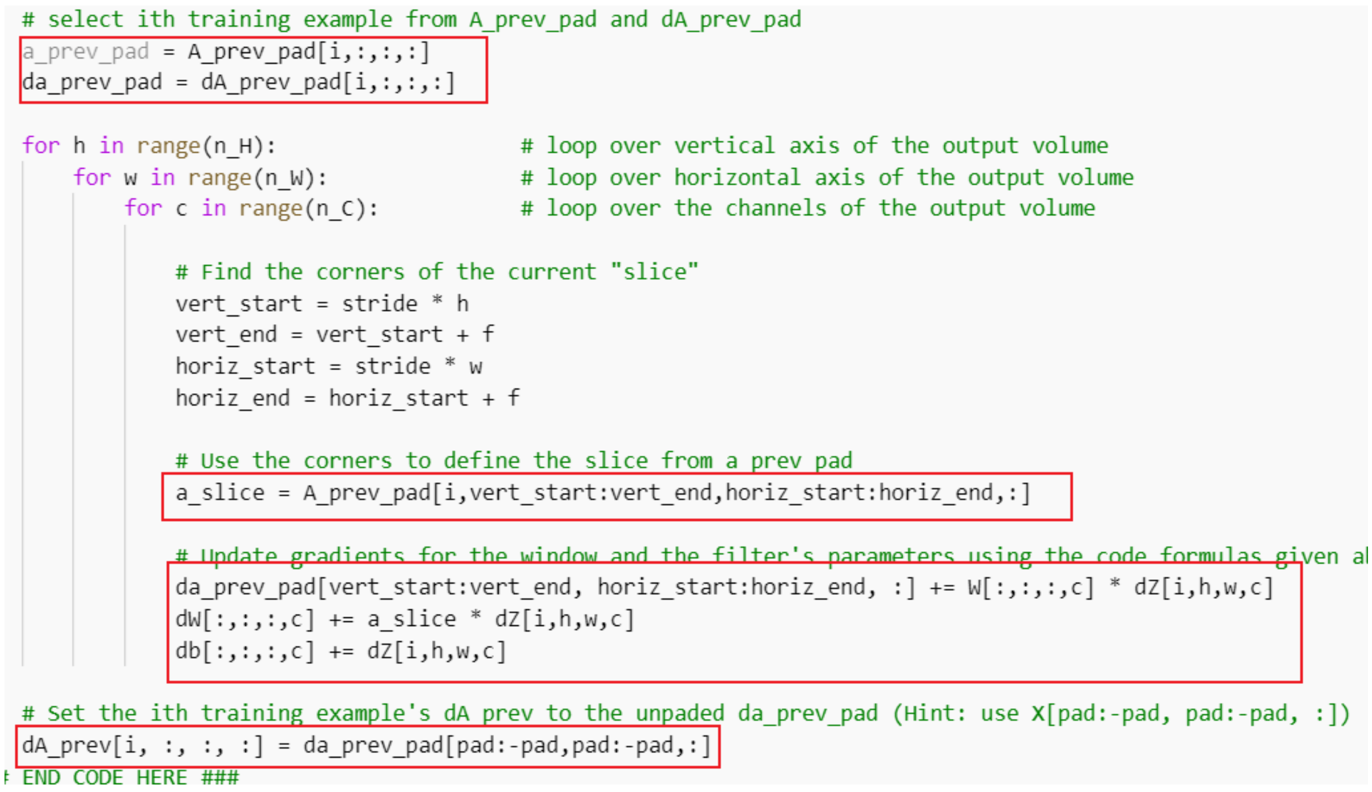
a_slice 对应于用于生成激活 Z_{ij}的切片。因此，提供了 W 相对于该切片的梯度。由于是相同的 W，将所有这些梯度相加得到 dW。

1.5.1.3 - Computing db:

计算 db 相对于某个过滤器 W_c 的成本的公式：

$$db = \sum_h \sum_w dZ_{hw}$$

根据公式实现代码如下：



```
# select ith training example from A_prev_pad and dA_prev_pad
a_prev_pad = A_prev_pad[i,:,:,:]
da_prev_pad = dA_prev_pad[i,:,:,:]

for h in range(n_H):                    # loop over vertical axis of the output volume
    for w in range(n_W):                # loop over horizontal axis of the output volume
        for c in range(n_C):            # loop over the channels of the output volume

            # Find the corners of the current "slice"
            vert_start = stride * h
            vert_end = vert_start + f
            horiz_start = stride * w
            horiz_end = horiz_start + f

            # Use the corners to define the slice from a_prev_pad
            a_slice = A_prev_pad[i,vert_start:vert_end,horiz_start:horiz_end,:]

            # Update gradients for the window and the filter's parameters using the code formulas given al
            da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] += W[:,:,:,c] * dZ[i,h,w,c]
            dW[:,:,:,c] += a_slice * dZ[i,h,w,c]
            db[:,:,:,c] += dZ[i,h,w,c]

# Set the ith training example's dA_prev to the unpaded da_prev_pad (Hint: use X[pad:-pad, pad:-pad, :])
dA_prev[i, :, :, :] = da_prev_pad[pad:-pad,pad:-pad,:]
# END CODE HERE ###
```

```
dA_mean = 1.4524377775388075
dW_mean = 1.7269914583139097
db_mean = 7.839232564616838
```

** Expected Output: **

dA_mean	1.45243777754
dW_mean	1.72699145831
db_mean	7.83923256462

1.5.2 Pooling layer - backward pass

实现池化层的反向传递，从 MAX-POOL 层开始。 虽然池化层没有要更新的反向传播参数，但需要通过池化层反向传播梯度，以便计算池化层之前的层的梯度。

1.5.2.1 Max pooling - backward pass

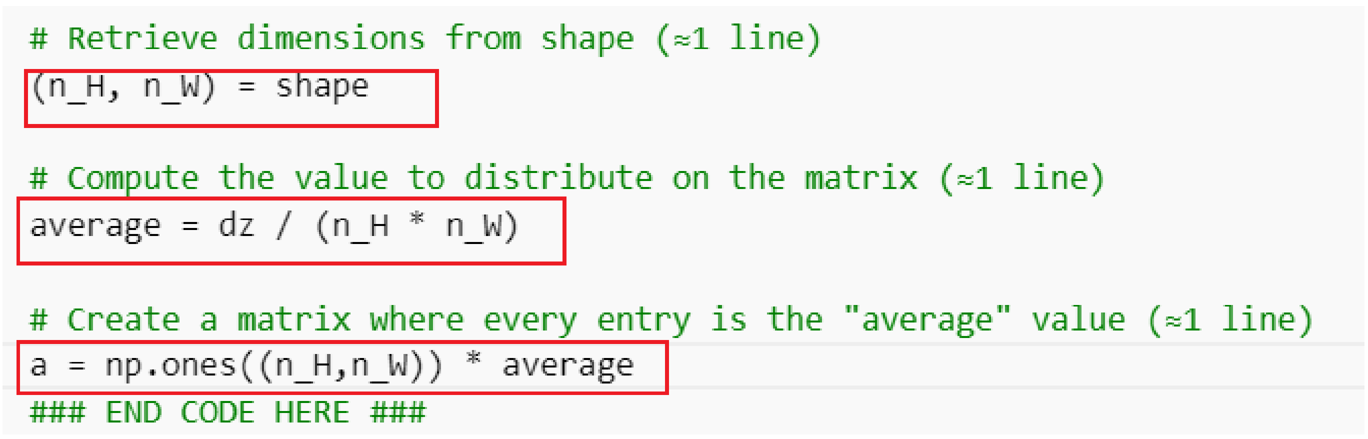
在进入池化层的反向传播之前，构建一个名为 create_mask_from_window()的辅助函数，执行以下操作：创建了一个“掩码”矩阵，用于跟踪矩阵的最大值在哪里。 True （1）表示最大值在 X 中的位置，其他条目为 False (0)。 平均池化的反向传递将与此类似，但使用不同的掩码，实现代码关键部分如下：

```
### START CODE HERE ### (≈1 line)
mask = (x == np.max(x))
### END CODE HERE ###
```

为什么要跟踪最大值的位置？ 这是因为这是最终影响输出和成本的输入值。 反向传播是根据成本计算梯度，所以任何影响最终成本的东西都应该有一个非零梯度。 因此，反向传播会将梯度“传播”会影响成本的特定输入值。

1.5.2.2 - Average pooling - backward pass

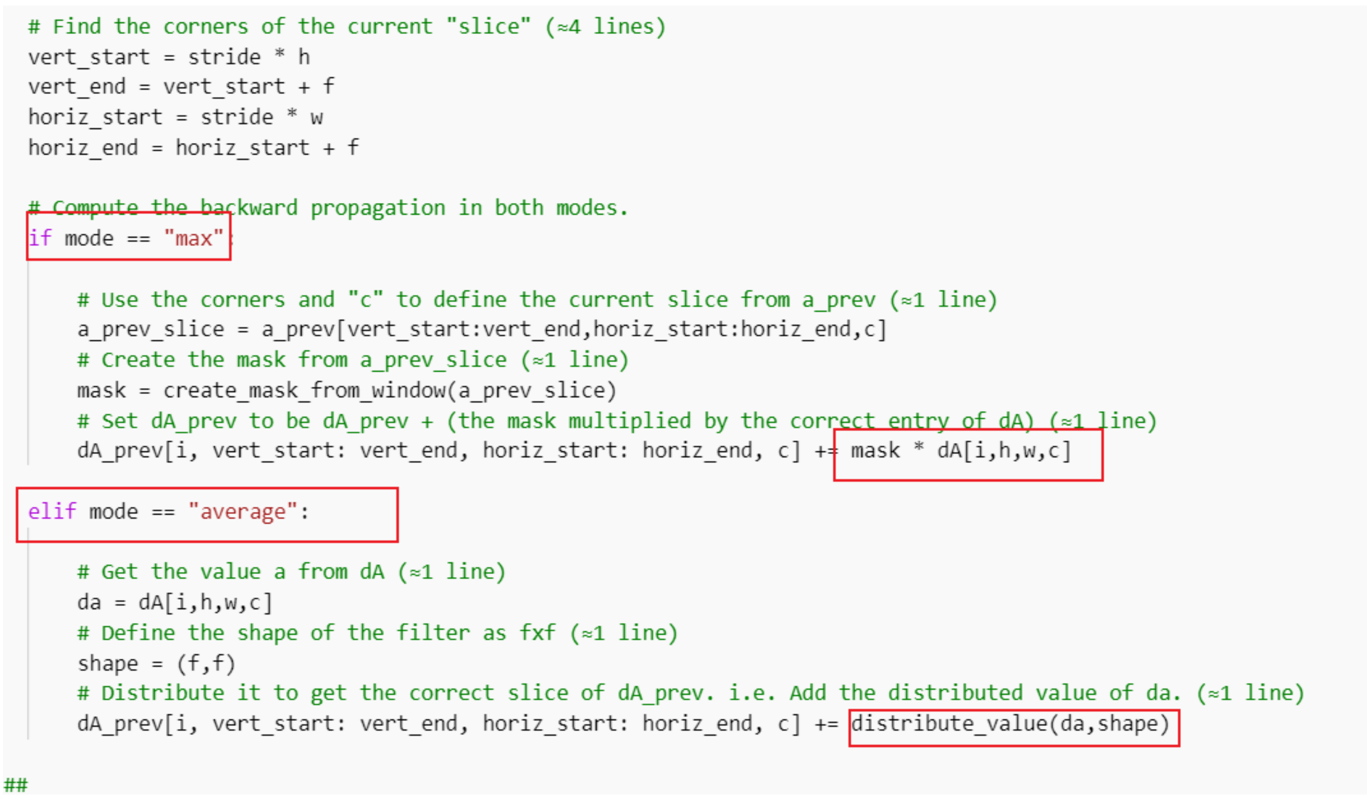
在最大池化中，对于每个输入窗口，对输出的所有“影响”都来自单个输入值——最大值。 在平均池化中，输入窗口的每个元素对输出都有相同的影响。 因此，要实现反向传播，实现一个反映这一点的辅助函数，关键代码如下：



```
# Retrieve dimensions from shape (≈1 line)
(n_H, n_W) = shape

# Compute the value to distribute on the matrix (≈1 line)
average = dz / (n_H * n_W)

# Create a matrix where every entry is the "average" value (≈1 line)
a = np.ones((n_H,n_W)) * average
### END CODE HERE ###
```

1.5.2.3 Putting it together: Pooling backward

在两种模式（max 和 average）下实现 pool_backward（） 函数。 使用 4 个 for 循环（迭代训练示例、高度、宽度和通道）。如果它等于 average，使用上面实现的 distribute_value()函数来创建一个与 a_slice 形状相同的矩阵。 否则，模式等于 max，使用 create_mask_from_window()创建一个掩码并将其乘以 dZ 的相应值，实现代码关键部分如下：

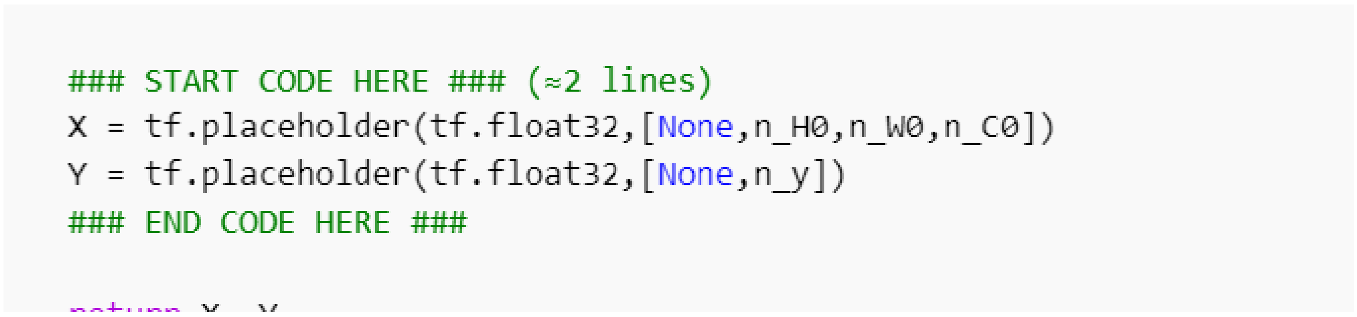


```
# Find the corners of the current "slice" (≈4 lines)
vert_start = stride * h
vert_end = vert_start + f
horiz_start = stride * w
horiz_end = horiz_start + f

# Compute the backward propagation in both modes.
if mode == "max":

    # Use the corners and "c" to define the current slice from a_prev (≈1 line)
    a_prev_slice = a_prev[vert_start:vert_end,horiz_start:horiz_end,c]
    # Create the mask from a_prev_slice (≈1 line)
    mask = create_mask_from_window(a_prev_slice)
    # Set dA_prev to be dA_prev + (the mask multiplied by the correct entry of dA) (≈1 line)
    dA_prev[i, vert_start: vert_end, horiz_start: horiz_end, c] += mask * dA[i,h,w,c]

elif mode == "average":

    # Get the value a from dA (≈1 line)
    da = dA[i,h,w,c]
    # Define the shape of the filter as fxf (≈1 line)
    shape = (f,f)
    # Distribute it to get the correct slice of dA_prev. i.e. Add the distributed value of da. (≈1 line)
    dA_prev[i, vert_start: vert_end, horiz_start: horiz_end, c] += distribute_value(da,shape)

##
```

步骤 2 Convolutional Neural Networks: Application

2.1 – Create placeholders

TensorFlow 要求在运行会话时输入模型的输入数据创建占位符，实现代码如下：



```
### START CODE HERE ### (≈2 lines)
X = tf.placeholder(tf.float32,[None,n_H0,n_W0,n_C0])
Y = tf.placeholder(tf.float32,[None,n_y])
### END CODE HERE ###

return X, Y
```

2.2 - Initialize parameters

初始化权重 W1 和 W2，使用 tf.get_variable(name, shape=None, initializer=None)来定义变量时，可以利用变量初始化函数来实现对 initializer 的赋值。在神经网络中，最常权重赋值方式是**正态随机赋值**和 Xavier **赋值**。

常用变量初始化函数如下

正态 tf.random_normal_initializer(shape,mean=0.0,stddev=1.0,dtype=tf.float32,name=None))

均匀 tf.random_uniform_initializer(shape,minval=0,maxval=None,dtype=tf.float32)

常量 tf.constant_initializer(obj,shape)

全 0 tf.zeros_initializer(shape,dtype)

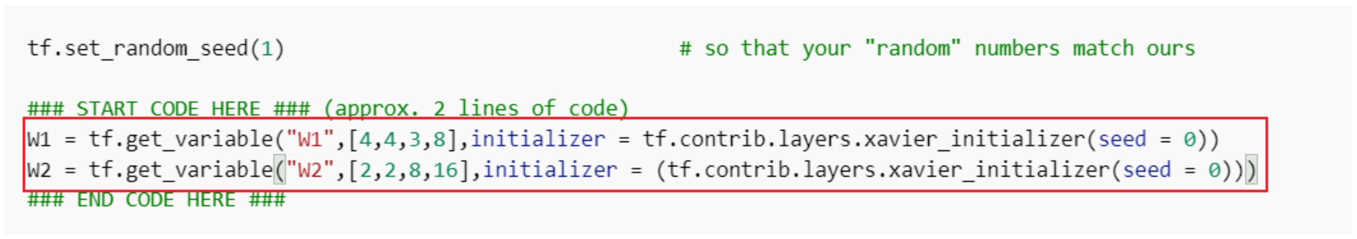
全 1 tf.ones_initializer(shape,dtype)

Xavier 初始化器

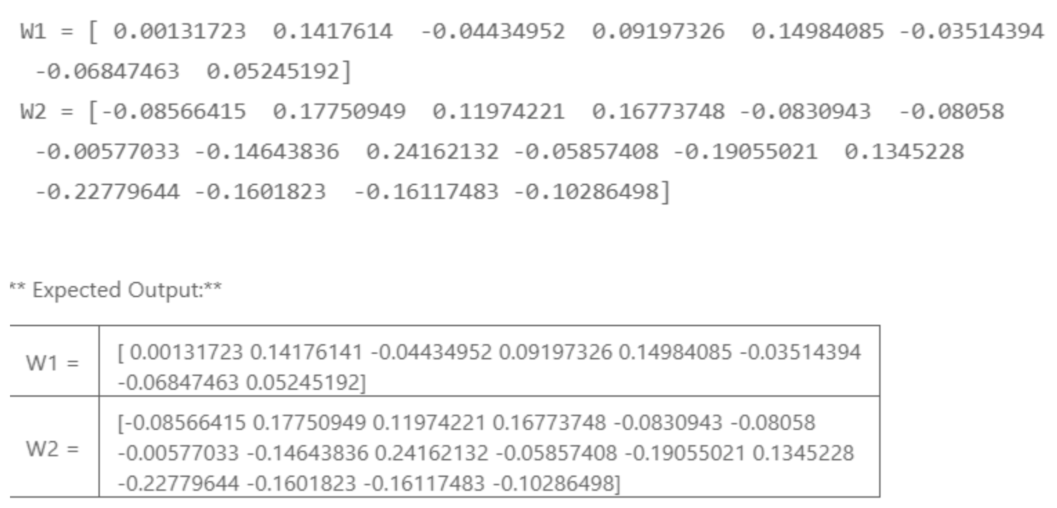
如果深度学习的权重初始化的太小，那么信号将在每一层传递时逐渐缩小而难以产生作用，但如果初始化得太大，那信号将在每层传递时逐渐放大并导致发散和失效，Xavier 初始化器就是让权重不大不小，刚好合适。会根据某一层网络的输入、输出节点的数量自动调整最合适的分布。

tf.contrib.layers.xavier_initializer(uniform=True, seed=None, dtype=tf.float32)

uniform 参数：使用 uniform 或者 normal 分布来随机初始化，实现代码如下：



```
tf.set_random_seed(1)                             # so that your "random" numbers match ours

### START CODE HERE ### (approx. 2 lines of code)
W1 = tf.get_variable("W1",[4,4,3,8],initializer = tf.contrib.layers.xavier_initializer(seed = 0))
W2 = tf.get_variable("W2",[2,2,8,16],initializer = (tf.contrib.layers.xavier_initializer(seed = 0)))
### END CODE HERE ###
```



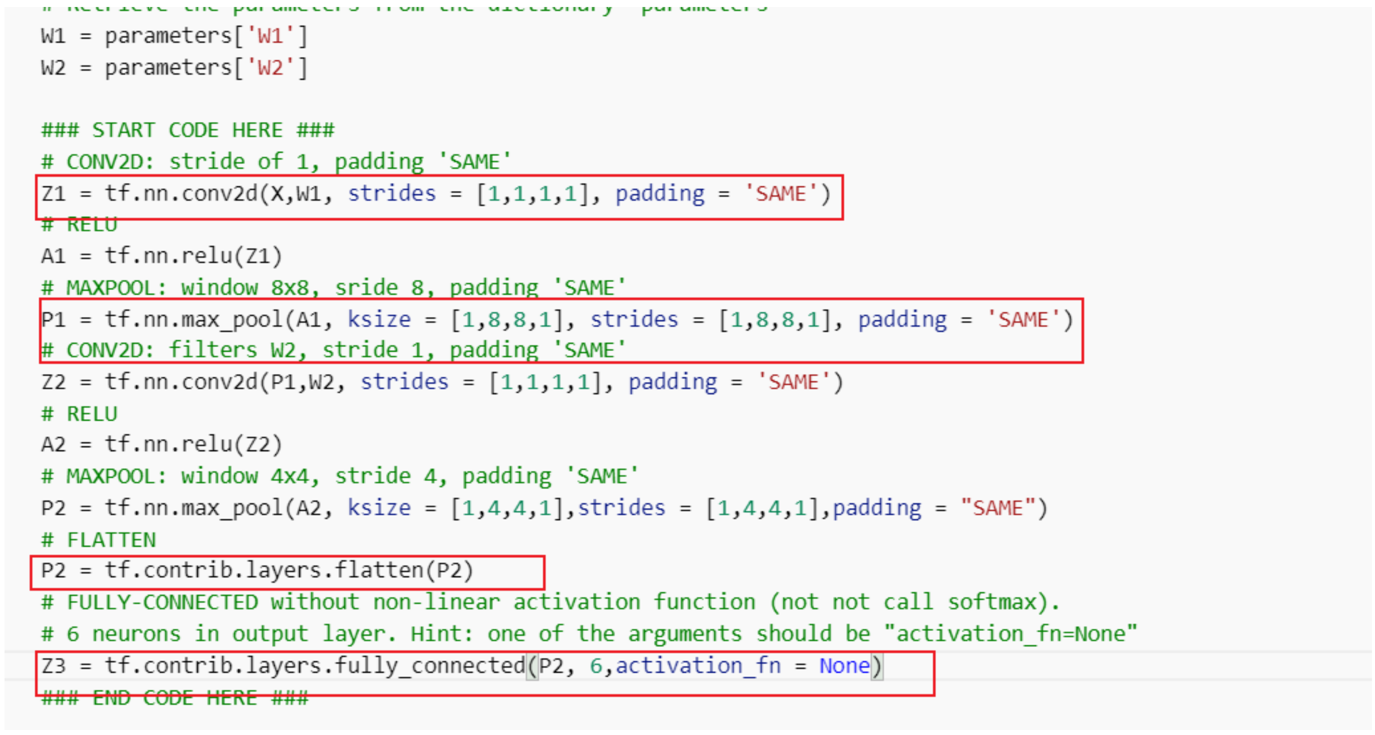
```
W1 = [ 0.00131723  0.1417614  -0.04434952  0.09197326  0.14984085 -0.03514394
 -0.06847463  0.05245192]
W2 = [-0.08566415  0.17750949  0.11974221  0.16773748 -0.0830943  -0.08058
 -0.00577033 -0.14643836  0.24162132 -0.05857408 -0.19055021  0.1345228
 -0.22779644 -0.1601823  -0.16117483 -0.10286498]
```

** Expected Output:**

W1 =	[0.00131723 0.14176141 -0.04434952 0.09197326 0.14984085 -0.03514394 -0.06847463 0.05245192]
W2 =	[-0.08566415 0.17750949 0.11974221 0.16773748 -0.0830943 -0.08058 -0.00577033 -0.14643836 0.24162132 -0.05857408 -0.19055021 0.1345228 -0.22779644 -0.1601823 -0.16117483 -0.10286498]

2.3 - Forward propagation

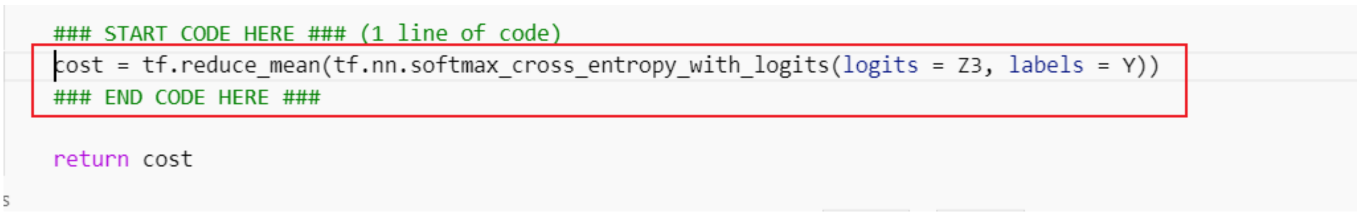
由提示实现前向传播函数，关键代码如下：



```
W1 = parameters['W1']
W2 = parameters['W2']

### START CODE HERE ###
# CONV2D: stride of 1, padding 'SAME'
Z1 = tf.nn.conv2d(X,W1, strides = [1,1,1,1], padding = 'SAME')
# RELU
A1 = tf.nn.relu(Z1)
# MAXPOOL: window 8x8, sride 8, padding 'SAME'
P1 = tf.nn.max_pool(A1, ksize = [1,8,8,1], strides = [1,8,8,1], padding = 'SAME')
# CONV2D: filters W2, stride 1, padding 'SAME'
Z2 = tf.nn.conv2d(P1,W2, strides = [1,1,1,1], padding = 'SAME')
# RELU
A2 = tf.nn.relu(Z2)
# MAXPOOL: window 4x4, stride 4, padding 'SAME'
P2 = tf.nn.max_pool(A2, ksize = [1,4,4,1],strides = [1,4,4,1],padding = "SAME")
# FLATTEN
P2 = tf.contrib.layers.flatten(P2)
# FULLY-CONNECTED without non-linear activation function (not not call softmax).
# 6 neurons in output layer. Hint: one of the arguments should be "activation_fn=None"
Z3 = tf.contrib.layers.fully_connected(P2, 6,activation_fn = None)
### END CODE HERE ###
```

2.4 - Compute cost

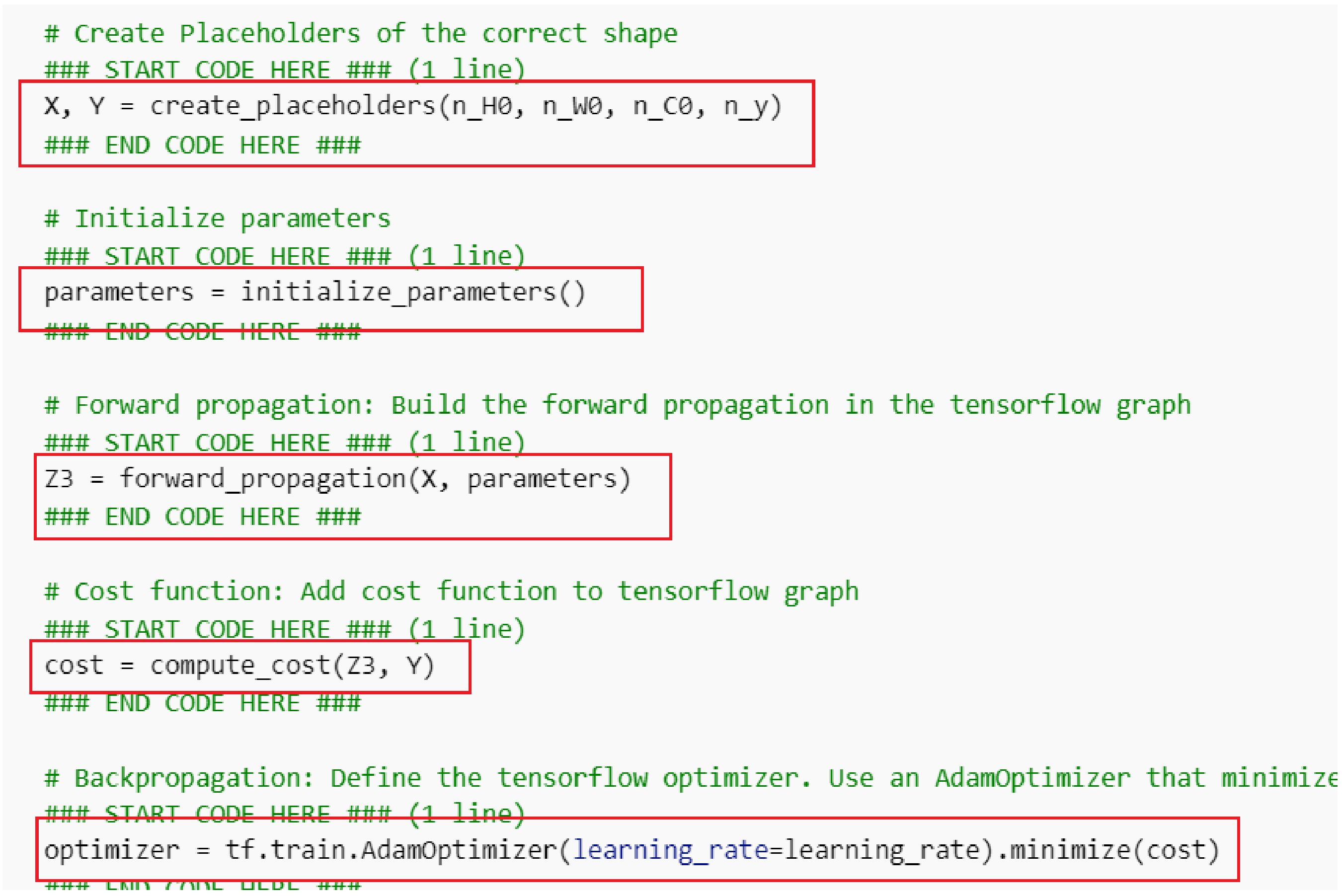
计算 softmax 的损失函数，这个函数既计算 softmax 的激活，也计算其损失，实现代码如下：



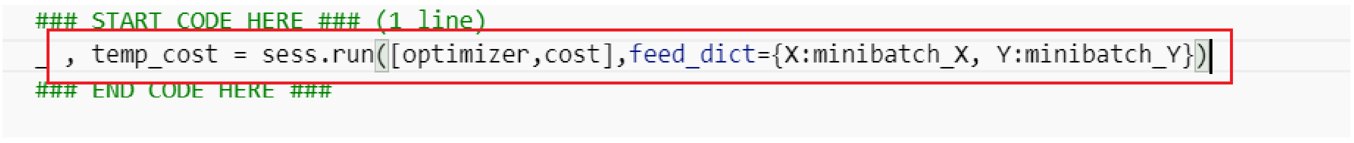
```
    ### START CODE HERE ### (1 line of code)
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = Z3, labels = Y))
    ### END CODE HERE ###

    return cost
```

2.5 Model

最后，合并在上面实现的辅助函数以构建模型，在 SIGNS 数据集上训练它，实现代码如下：



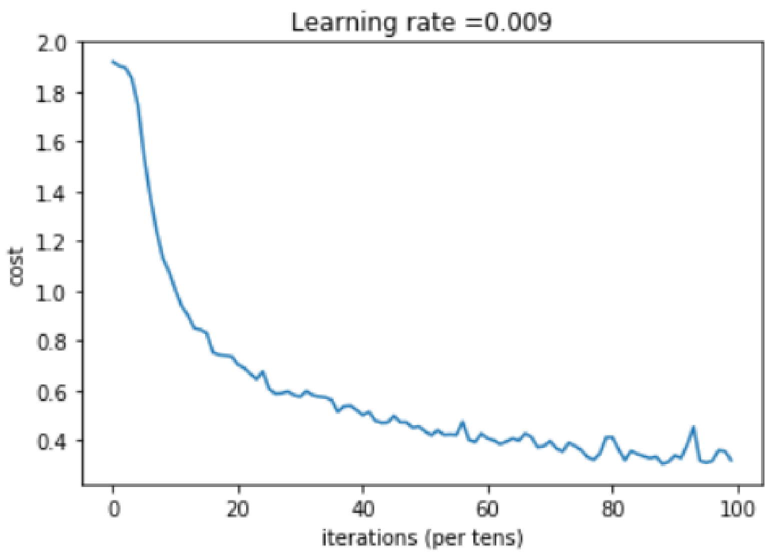
```
    # Create Placeholders of the correct shape
    ### START CODE HERE ### (1 line)
    X, Y = create_placeholders(n_H0, n_W0, n_C0, n_y)
    ### END CODE HERE ###

    # Initialize parameters
    ### START CODE HERE ### (1 line)
    parameters = initialize_parameters()
    ### END CODE HERE ###

    # Forward propagation: Build the forward propagation in the tensorflow graph
    ### START CODE HERE ### (1 line)
    Z3 = forward_propagation(X, parameters)
    ### END CODE HERE ###

    # Cost function: Add cost function to tensorflow graph
    ### START CODE HERE ### (1 line)
    cost = compute_cost(Z3, Y)
    ### END CODE HERE ###

    # Backpropagation: Define the tensorflow optimizer. Use an AdamOptimizer that minimize
    ### START CODE HERE ### (1 line)
    optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
    ### END CODE HERE ###
```



```
  ### START CODE HERE ### (1 line)
  _ , temp_cost = sess.run([optimizer,cost],feed_dict={X:minibatch_X, Y:minibatch_Y})
  ### END CODE HERE ###
```

完成并构建了一个模型，此时效果如下：

/>



```
Tensor("Mean_1:0", shape=(), dtype=float32)
Train Accuracy: 0.88703704
Test Accuracy: 0.76666665
```

可以看到最终在训练集上的准确率是 0.887，测试集上的准确率是 0.767

步骤三 sidual Networks

使用残差网络（ResNets）构建非常深的卷积网络，理论上，非常深的网络可以表示非常复杂的功能，但在实践中，很难训练。

非常深的网络的主要好处是可以表示非常复杂的功能，还可以学习许多不同抽象级别的特征，从边缘（在较低层）到非常复杂的特征（在更深层）。然而，使用更深的网络并不总是有帮助，训练的一个巨大障碍是梯度消失。

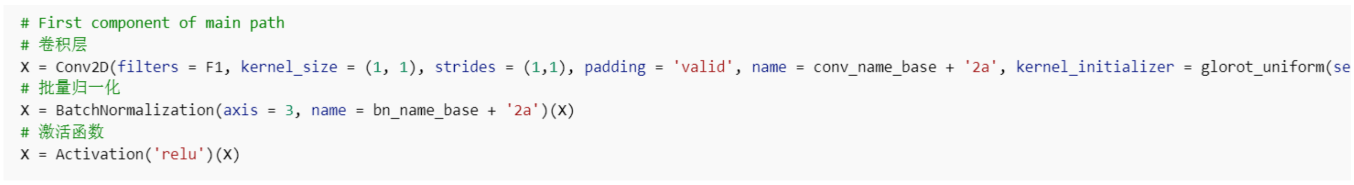
非常深的网络通常有一个梯度信号，会很快变为零，从而使梯度下降慢得难以忍受。更具体地说，在梯度下降过程中，当你从最后一层反向传播到第一层时，你在每一步都乘以权重矩阵，因此梯度可以呈指数快速下降到零（或者，在极少数情况下，呈指数增长快速并“爆炸”以获取非常大的值）。

因此，在训练期间，可能会看到，随着训练的进行，较早层的梯度幅度（或范数）会非常迅速地减小为零，而在 ResNets 中，“shortcut”或“skip connection”允许梯度直接反向传播到较早的层

3.1 - The identity block

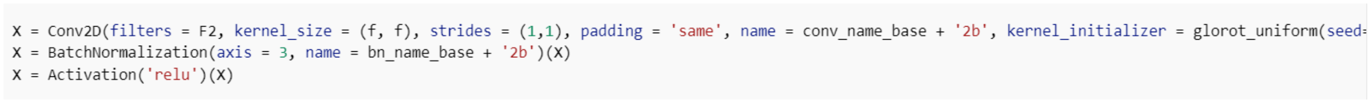
The identity block 是 ResNets 中使用的标准块，对应于输入激活（比如 a^[l]）与输出激活（比如 a^[l+2]）。为了充实 ResNet 的身份块中发生的不同步骤，实现代码如下：

第一层实现代码如下：



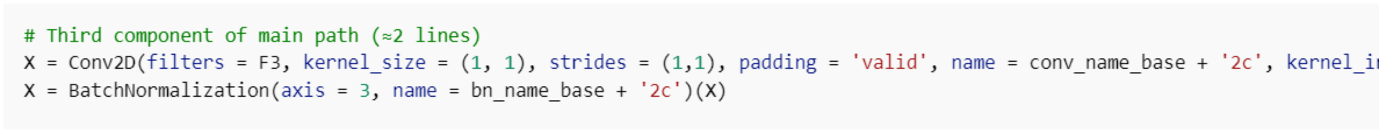
```
# First component of main path
# 卷积层
X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2a', kernel_initializer = glorot_uniform(se
# 批量归一化
X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
# 激活函数
X = Activation('relu')(X)
```

第二层实现代码如下：



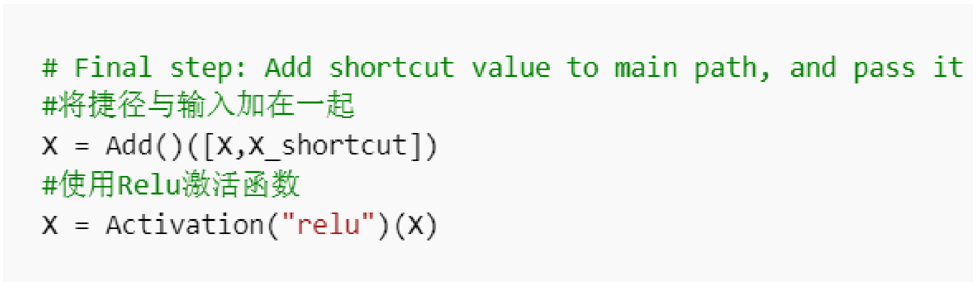
```
X = Conv2D(filters = F2, kernel_size = (f, f), strides = (1,1), padding = 'same', name = conv_name_base + '2b', kernel_initializer = glorot_uniform(seed=
X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
X = Activation('relu')(X)
```

第三层实现代码如下：

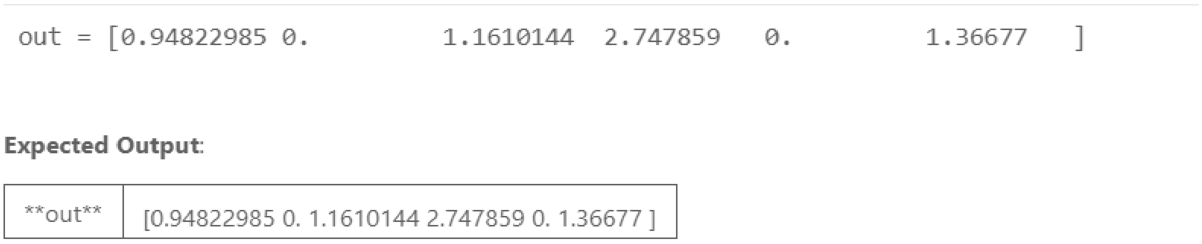


```
# Third component of main path (≈2 lines)
X = Conv2D(filters = F3, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2c', kernel_i
X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)
```

最后相加并同 Relu,实现代码如下：



```
# Final step: Add shortcut value to main path, and pass it
#将捷径与输入加在一起
X = Add()([X,X_shortcut])
#使用Relu激活函数
X = Activation("relu")(X)
```



```
out = [0.94822985 0.         1.1610144  2.747859   0.         1.36677    ]
```

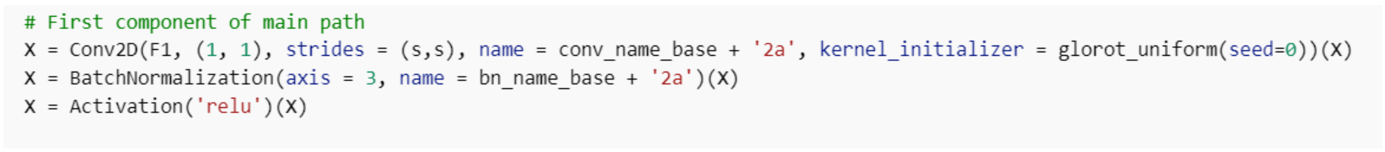
Expected Output:

out	[0.94822985 0. 1.1610144 2.747859 0. 1.36677]

3.2 - The convolutional block

已经实现了 ResNet 身份块，接下来，ResNet“卷积块”是另一种类型的块。 当输入和输出维度不匹配时，可以使用这种类型的块，与 identity block 的区别在于快捷路径中有一个 CONV2D 层。

第一层实现代码如下：



```
# First component of main path
X = Conv2D(F1, (1, 1), strides = (s,s), name = conv_name_base + '2a', kernel_initializer = glorot_uniform(seed=0))(X)
X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
X = Activation('relu')(X)
```

第二层实现代码如下：

```
# Second component of main path (≈3 lines)
X = Conv2D(filters=F2, kernel_size=(f,f), strides = (1,1), padding="same",name = conv_name_base + '2b'
X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
X = Activation('relu')(X)
```

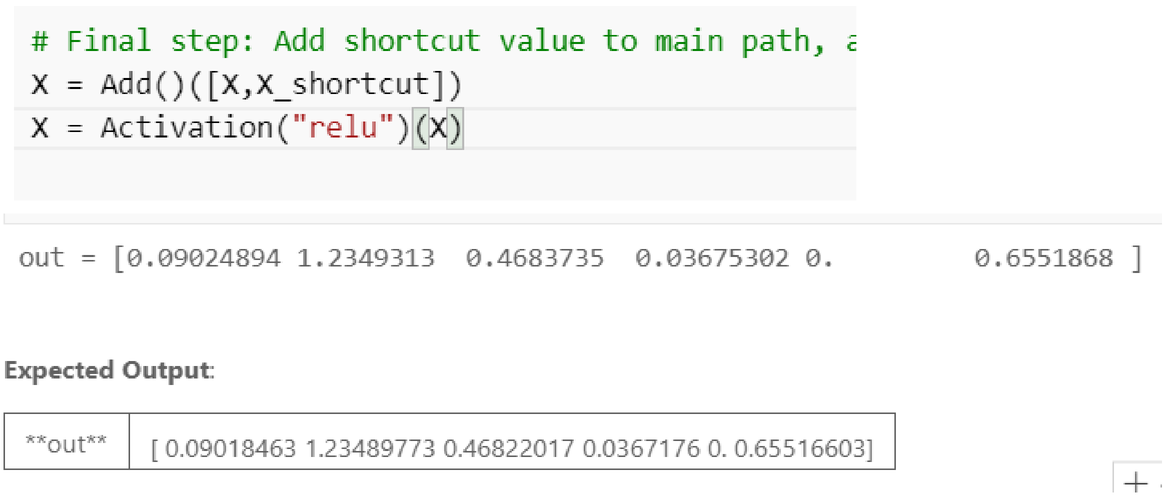
第三层实现代码如下：

```
# Third component of main path (≈2 lines)
X = Conv2D(filters=F3, kernel_size=(1,1), strides = (1,1), padding="same",name = conv_name_base + '2c',
X = Activation('relu')(X)
```

SHORTCUT PATH 实现代码如下：

```
##### SHORTCUT PATH #### (≈2 lines)
X_shortcut =  Conv2D(filters=F3, kernel_size=(1,1), strides=(s,s), padding="valid",
                name=conv_name_base+"1", kernel_initializer=glorot_uniform(seed=0))(X_shortcut)
X_shortcut = BatchNormalization(axis=3,name=bn_name_base+"1")(X_shortcut)
```

最后一步相加并通过 Relu 层，实现代码如下：



```
# Final step: Add shortcut value to main path, a
X = Add()([X,X_shortcut])
X = Activation("relu")(X)
```

```
out = [0.09024894 1.2349313  0.4683735  0.03675302 0.         0.6551868 ]
```

Expected Output:

out	[0.09018463 1.23489773 0.46822017 0.0367176 0. 0.65516603]

3.3 – Building your first ResNet model (50 layers)

构建卷积神经网络。现在已经构建好需要用到的残差块了，现在根据这个 50 层的神经网络原理图构建卷积神经网络。

第一步实现代码如下：

```
# Stage 1
X = Conv2D(64, (7, 7), strides = (2, 2), name = 'conv1', kernel_initializer = glorot_uniform(seed=0))(X)
X = BatchNormalization(axis = 3, name = 'bn_conv1')(X)
X = Activation('relu')(X)
X = MaxPooling2D((3, 3), strides=(2, 2))(X)
```

第二步实现代码如下：

```
# Stage 2
X = convolutional_block(X, f = 3, filters = [64, 64, 256], stage = 2, block='a', s = 1)
X = identity_block(X, 3, [64, 64, 256], stage=2, block='b')
X = identity_block(X, 3, [64, 64, 256], stage=2, block='c')
```

第三部实现代码如下：

```
# Stage 3 (≈4 lines)
X = convolutional_block(X, f = 3, filters = [128, 128, 512], stage = 3, block='a', s = 2)
X = identity_block(X, 3, [128,128,512], stage=3, block='b')
X = identity_block(X, 3, [128,128,512], stage=3, block='c')
X = identity_block(X, 3, [128,128,512], stage=3, block='d')
```

第四步实现代码如下：

```
# Stage 4 (≈6 lines)
X = convolutional_block(X, f = 3, filters = [256, 256, 1024], stage = 4, block='a', s = 2)
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='b')
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='c')
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='d')
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='e')
X = identity_block(X, 3, [256, 256, 1024], stage=4, block='f')
```

第五步实现代码如下：

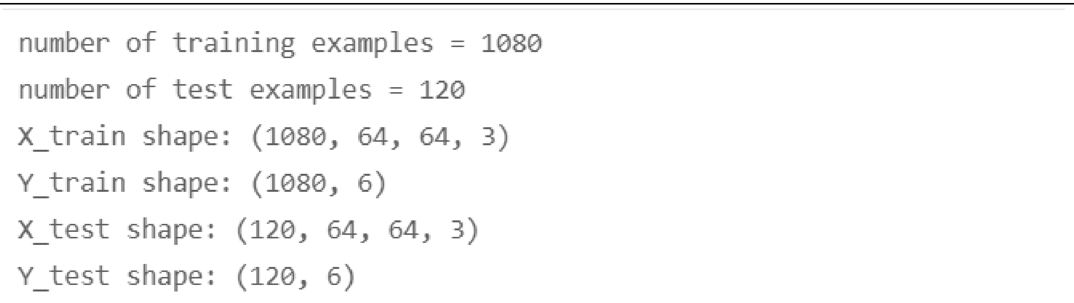
```
# Stage 5 (≈3 lines)
X = convolutional_block(X, f = 3, filters = [512, 512, 2048], stage = 5, block='a', s = 2)
X = identity_block(X, 3, [512, 512, 2048], stage=5, block='b')
X = identity_block(X, 3, [512, 512, 2048], stage=5, block='c')
```

均值池化部分代码如下：

```
# AVGPOOL (≈1 line). Use "X = AveragePooling2D(...)(X)"
X = AveragePooling2D(pool_size=(2,2),padding="same")(X)
```

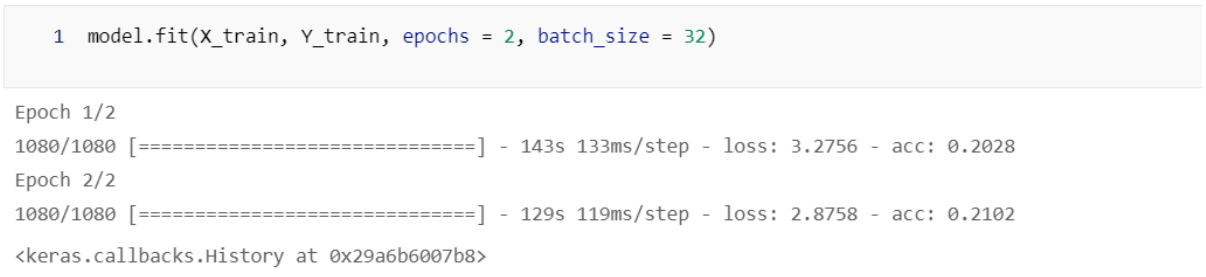
3.4 构建模型、训练、测试

建立模型并导入数据集，得到手势数据集维度如下：



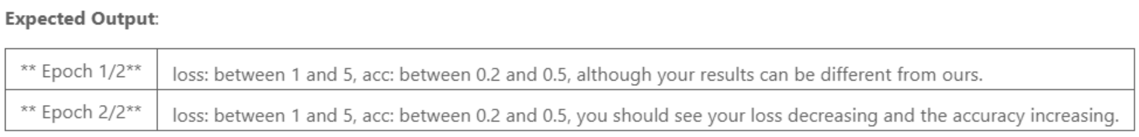
```
number of training examples = 1080
number of test examples = 120
X_train shape: (1080, 64, 64, 3)
Y_train shape: (1080, 6)
X_test shape: (120, 64, 64, 3)
Y_test shape: (120, 6)
```

训练模型过程如下：



```
model.fit(X_train, Y_train, epochs = 2, batch_size = 32)
```

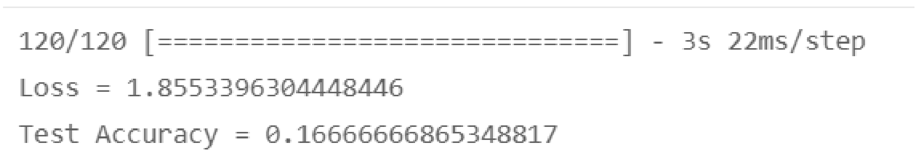
```
Epoch 1/2
1080/1080 [==============================] - 143s 133ms/step - loss: 3.2756 - acc: 0.2028
Epoch 2/2
1080/1080 [==============================] - 129s 119ms/step - loss: 2.8758 - acc: 0.2102
<keras.callbacks.History at 0x29a6b6007b8>
```



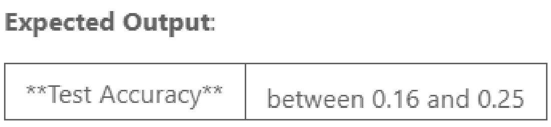
Expected Output:

** Epoch 1/2**	loss: between 1 and 5, acc: between 0.2 and 0.5, although your results can be different from ours.
** Epoch 2/2**	loss: between 1 and 5, acc: between 0.2 and 0.5, you should see your loss decreasing and the accuracy increasing.

最后测试效果如下：



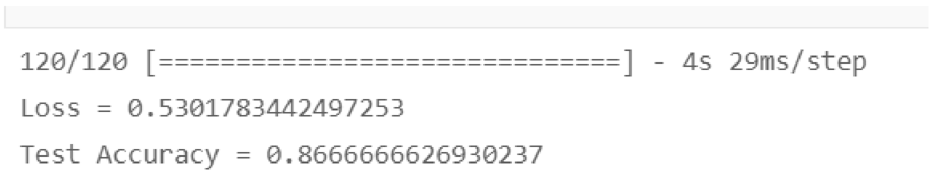
```
120/120 [==============================] - 3s 22ms/step
Loss = 1.8553396304448446
Test Accuracy = 0.16666666865348817
```



Expected Output:

Test Accuracy	between 0.16 and 0.25

然后使用训练好的模型进行测试，实现效果如下：



```
120/120 [==============================] - 4s 29ms/step
Loss = 0.5301783442497253
Test Accuracy = 0.8666666626930237
```

可以发现使用别人训练很长时间的权重的模型效果比刚才自己只两次的模型效果好很多，在测试集上的准确率达到了 0.867

结论分析与体会：

Convolutional Neural Networks

学习了卷积神经网络的构建方法，了解了卷积神经网络主要有卷积模块和池化模块组成。

卷积模块，包含了以下函数：

使用 0 扩充边界、卷积窗口、前向卷积、反向卷积（可选）

池化模块，包含了以下函数：

前向池化、创建掩码、值分配、反向池化（可选）

有了这两个模块后，根据网络结构原理图，构建了 CNN 神经网络。

CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLYCONNECTED

构建了 CNN 模型，包括以下步骤

创建占位符、初始化参数、前向传播、计算成本、反向传播、创建优化器、创建一个 session 运行模型

Residual Networks

非常深的“普通”网络在实践中不起作用，因为它们由于梯度消失而难以训练，skip-connections 有助于解决渐变消失问题，还使 ResNet 块很容易学习恒等映射函数。

就实验过程中遇到和出现的问题，你是如何解决和处理的，自拟 1—3 道问答题：

1.tensorflow 需要用到对应版本才能够很好的使用，对于 python 环境，使用 anconda 可能会更为方便快捷，建立专门的虚拟环境，运行会更加方便一些，直接改自己的配置，又没法做其他课的实验，还得修改回来