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：Building a Convolutional Neural Network Model and Application, using Keras to build a residual network		学号：201900301174
日期：2021-11-01	班级： 智能 19	姓名： 韩旭
Email：hanx@mail.sdu.edu.cn		

实验目的：

Introduction

In this assignment you will understand the architecture of **Convolutional Neural Networks** and get practice with training these models on data.

- you will be given two subtasks: Building a Convolutional Neural Network Model and Application, using Keras to build a residual network **(Optional)**.

实验软件和硬件环境：

Termius

Jupyter notebook

RTX3090

Xeon Gold 6226R

实验原理和方法：

- **Experiments**

  There are ### START CODE HERE / ### END CODE HERE tags denoting the start and end of code sections you should fill out. Take care to not delete or modify these tags, or your assignment may not be properly graded.

  - **Q1: Convolutional model -Step by Step (50 points)**

    The IPython Notebook Convolutional model -Step by Step.ipynb will walk you through implementing the CNN step by step.

  - **Q2: Convolutional model - Application (50 points)**

    The IPython Notebook Convolutional model - Application.ipynb will walk you through implementing the CNN application.

  - **Q3: Residual Networks (50 points) (Optional)**

    The IPython Notebook Residual Networks.ipynb will walk you through implementing the Residual Networks.

  - **See the code file for details.**

ResNet 测试准确度经过调参达到 0.975，高于预训练保存的 h5 模型的测试准确度！

```python
preds = model2.evaluate(X_test, Y_test)
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```

executed in 496ms, finished 17:14:59 2021-11-03

```
120/120 [==============================] - 0s 4ms/step
Loss = 0.14251829956968626
Test Accuracy = 0.975
```

一, Convolutional Neural Networks: Step by Step

Zero padding:

Exercise: Implement the following function, which pads all the images of a batch of examples X with zeros. Use np.pad. Note if you want to pad the array "a" of shape $(5,5,5,5,5)$ with `pad = 1` for the 2nd dimension, `pad = 3` for the 4th dimension and `pad = 0` for the rest, you would do:

```python
a = np.pad(a, ((0,0), (1,1), (0,0), (3,3), (0,0)), 'constant', constant_values = (..,..))
```

Code: just follow the hint

```python
def zero_pad(X, pad):
    """
    Pad with zeros all images of the dataset X. The padding is applied to the height and width of an image,
    as illustrated in Figure 1.

    Argument:
    X -- python numpy array of shape (m, n_H, n_W, n_C) representing a batch of m images
    pad -- integer, amount of padding around each image on vertical and horizontal dimensions

    Returns:
    X_pad -- padded image of shape (m, n_H + 2*pad, n_W + 2*pad, n_C)
    """

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

    return X_pad
```

Single step of convolution:

1.3.2 3.2 - Single step of convolution

In this part, implement a single step of convolution, in which you apply the filter to a single position of the input. This will be used to build a convolutional unit, which:

- Takes an input volume
- Applies a filter at every position of the input
- Outputs another volume (usually of different size)

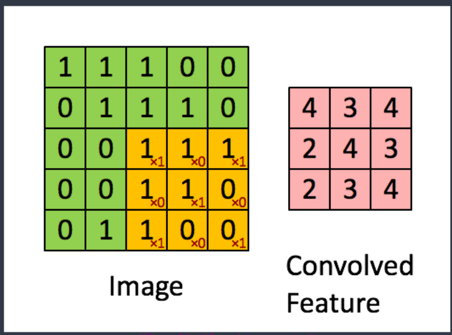


Figure 2 : **Convolution operation**
with a filter of 2x2 and a stride of 1 (stride = amount you move the window each time you slide)

In a computer vision application, each value in the matrix on the left corresponds to a single pixel value, and we convolve a 3x3 filter with the image by multiplying its values element-wise with the original matrix, then summing them up. In this first step of the exercise, you will implement a single step of convolution, corresponding to applying a filter to just one of the positions to get a single real-valued output.

Later in this notebook, you'll apply this function to multiple positions of the input to implement the full convolutional operation.

Exercise: Implement conv_single_step(). Hint.

Code: follow the hint, multiply input with weight and then add bias.

```
def conv_single_step(a_slice_prev, W, b):
    """
    Apply one filter defined by parameters W on a single slice (a_slice_prev) of the output activation
    of the previous layer.

    Arguments:
    a_slice_prev -- slice of input data of shape (f, f, n_C_prev)
    W -- Weight parameters contained in a window - matrix of shape (f, f, n_C_prev)
    b -- Bias parameters contained in a window - matrix of shape (1, 1, 1)

    Returns:
    Z -- a scalar value, result of convolving the sliding window (W, b) on a slice x of the input data
    """

    ### START CODE HERE ### (≈ 2 lines of code)
    # 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)+np.float(b)
    ### END CODE HERE ###

    return Z
```

The result is correct.

executed in 8ms, finished 11:30:17 2021-11-03

```
Z = -6.999089450680221
```

Expected Output:

Z	-6.99908945068

Convolution neural networks – Forward pass

Exercise: Implement the function below to convolve the filters W on an input activation A_prev. This function takes as input A_prev, the activations output by the previous layer (for a batch of m inputs), F filters/weights denoted by W, and a bias vector denoted by b, where each filter has its own (single) bias. Finally you also have access to the hyperparameters dictionary which contains the stride and the padding.

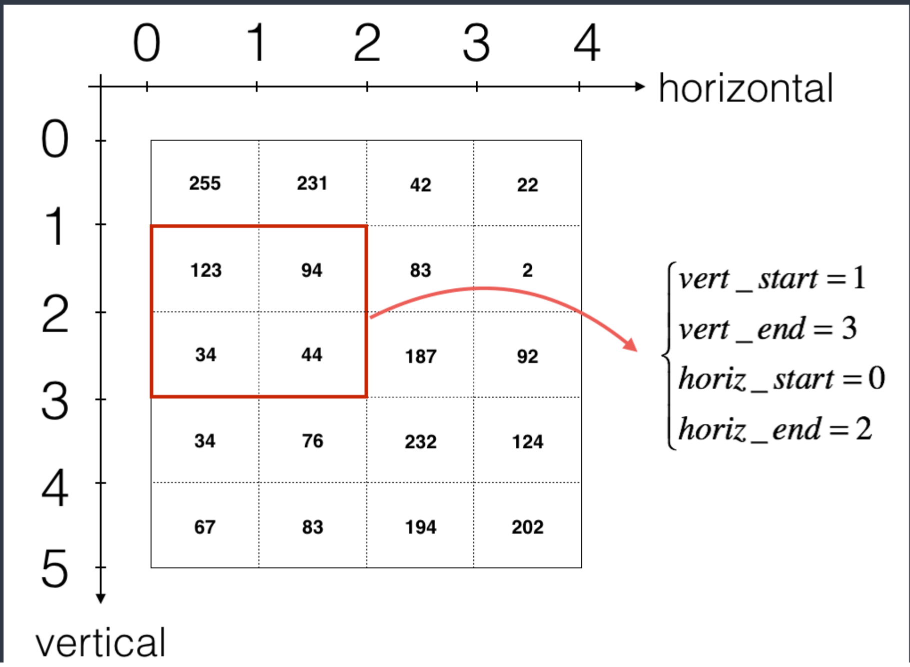
Hint:

1. To select a 2x2 slice at the upper left corner of a matrix "a_prev" (shape (5,5,3)), you would do:

```
a_slice_prev = a_prev[0:2,0:2,:]
```

This will be useful when you will define `a_slice_prev` below, using the `start/end` indexes you will define.

2. To define a_slice you will need to first define its corners `vert_start`, `vert_end`, `horiz_start` and `horiz_end`. This figure may be helpful for you to find how each of the corner can be defined using h, w, f and s in the code below.



Reminder: The formulas relating the output shape of the convolution to the input shape is:

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_C = \text{number of filters used in the convolution}$$

For this exercise, we won't worry about vectorization, and will just implement everything with for-loops.

Code: follow the hint

```
### START CODE HERE ###
# Retrieve dimensions from A_prev's shape (≈1 line)
(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_H_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])
```

```python
# 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 = h*stride
                vert_end = vert_start+f
                horiz_start = w*stride
                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])

### END CODE HERE ###
```

The result is correct.

executed in 19ms, finished 11:30:17 2021-11-03

```
Z's mean = 0.048995203528855794
Z[3,2,1] = [-0.61490741 -6.7439236  -2.55153897  1.75698377  3.56208902  0.53036437
  5.18531798  8.75898442]
cache_conv[0][1][2][3] = [-0.20075807  0.18656139  0.41005165]
```

Expected Output:

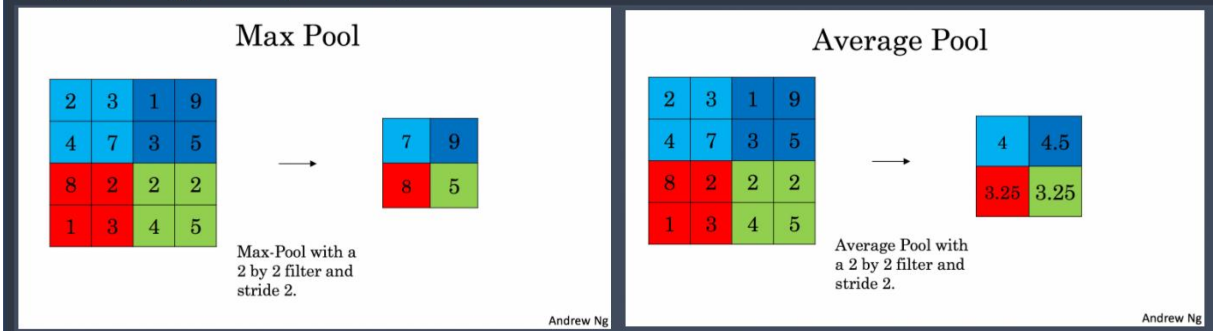
Z's mean	0.0489952035289
Z[3,2,1]	[-0.61490741 -6.7439236 -2.55153897 1.75698377 3.56208902 0.53036437 5.18531798 8.75898442]
cache_conv[0][1][2][3]	[-0.20075807 0.18656139 0.41005165]

Pooling layer

1.4 4 - Pooling layer

The pooling (POOL) layer reduces the height and width of the input. It helps reduce computation, as well as helps make feature detectors more invariant to its position in the input. The two types of pooling layers are:

- Max-pooling layer: slides an (f, f) window over the input and stores the max value of the window in the output.
- Average-pooling layer: slides an (f, f) window over the input and stores the average value of the window in the output.



These pooling layers have no parameters for backpropagation to train. However, they have hyperparameters such as the window size f. This specifies the height and width of the fxf window you would compute a max or average over.

1.4.1 4.1 - Forward Pooling

Now, you are going to implement MAX-POOL and AVG-POOL, in the same function.

Exercise: Implement the forward pass of the pooling layer. Follow the hints in the comments below.

Reminder: As there's no padding, the formulas binding the output shape of the pooling to the input shape is:

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

Code: follow the hint

```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 = h*stride
                vert_end = vert_start+f
                horiz_start = w*stride
                horiz_end = horiz_start+f

                # Use the corners to define the current slice on the ith training example of A_prev, channel c. (≈1 line)
                a_prev_slice = A_prev[i, vert_start:vert_end, horiz_start:horiz_end, c] # don't forget channel c.

                # Compute the pooling operation on the slice. Use an if statment to differentiate the modes. Use np.max/np.mean.
                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)

### END CODE HERE ###
```

```
mode = max
A = [[[[1.74481176 0.86540763 1.13376944]]]


 [[[1.13162939 1.51981682 2.18557541]]]]

mode = average
A = [[[[ 0.02105773 -0.20328806 -0.40389855]]]


 [[[-0.22154621  0.51716526  0.48155844]]]]
```

Expected Output:

A =	[[[[1.74481176 0.86540763 1.13376944]]] [[[1.13162939 1.51981682 2.18557541]]]]
A =	[[[[0.02105773 -0.20328806 -0.40389855]]] [[[-0.22154621 0.51716526 0.48155844]]]]

Backpropagation in convolutional neural networks:

1.5 5 - Backpropagation in convolutional neural networks

In modern deep learning frameworks, you only have to implement the forward pass, and the framework takes care of the backward pass, so most deep learning engineers don't need to bother with the details of the backward pass. The backward pass for convolutional networks is complicated. If you wish however, you can work through this optional portion of the notebook to get a sense of what backprop in a convolutional network looks like.

When in an earlier course you implemented a simple (fully connected) neural network, you used backpropagation to compute the derivatives with respect to the cost to update the parameters. Similarly, in convolutional neural networks you can to calculate the derivatives with respect to the cost in order to update the parameters. The backprop equations are not trivial and we did not derive them in lecture, but we briefly presented them below.

1.5.1 5.1 - Convolutional layer backward pass

Let's start by implementing the backward pass for a CONV layer.

1.5.1.1 5.1.1 - Computing dA:

This is the formula for computing dA with respect to the cost for a certain filter W_c and a given training example:

Where W_c is a filter and dZ_{hw} is a scalar corresponding to the gradient of the cost with respect to the output of the conv layer Z at the hth row and wth column (corresponding to the dot product taken at the ith stride left and jth stride down). Note that at each time, we multiply the the same filter W_c by a different dZ when updating dA. We do so mainly because when computing the forward propagation, each filter is dotted and summed by a different a_slice. Therefore when computing the backprop for dA, we are just adding the gradients of all the a_slices.

In code, inside the appropriate for-loops, this formula translates into:

```
da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] += W[:,:,:,c] * dZ[i, h, w, c]
```

1.5.1.2 5.1.2 - Computing dW:

This is the formula for computing dW_c (dW_c is the derivative of one filter) with respect to the loss:

Where a_{slice}corresponds to the slice which was used to generate the acitivation Z_{ij}. Hence, this ends up giving us the gradient for W with respect to that slice. Since it is the same W, we will just add up all such gradients to get dW.

In code, inside the appropriate for-loops, this formula translates into:

```
dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
```

1.5.1.3 5.1.3 - Computing db:

This is the formula for computing db with respect to the cost for a certain filter W_c:

As you have previously seen in basic neural networks, db is computed by summing dZ. In this case, you are just summing over all the gradients of the conv output (Z) with respect to the cost.

In code, inside the appropriate for-loops, this formula translates into:

```
db[:,:,:,c] += dZ[i, h, w, c]
```

Exercise: Implement the `conv_backward` function below. You should sum over all the training examples, filters, heights, and widths. You should then compute the derivatives using formulas 1, 2 and 3 above.

Code: follow the hint

```
### START CODE HERE ###
# Retrieve information from "cache"
(A_prev, W, b, hparameters) = cache

# Retrieve dimensions from A_prev's shape
(m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

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

# Retrieve information from "hparameters"
stride = hparameters['stride']
pad = hparameters['pad']

# Retrieve dimensions from dZ's shape
(m, n_H, n_W, n_C) = dZ.shape

# Initialize dA_prev, dW, db with the correct shapes
dA_prev = np.zeros(A_prev.shape)
dW = np.zeros(W.shape)
db = np.zeros([1, 1, 1, n_C])
```

```python
# Pad A_prev and dA_prev
A_prev_pad = zero_pad(A_prev,pad)
dA_prev_pad = zero_pad(dA_prev,pad)

for i in range(m):                       # loop over the training examples

    # 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 = h*stride
                vert_end = vert_start+f
                horiz_start = w*stride
                horiz_end = horiz_start+f

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

                # Update gradients for the window and the filter's parameters using the code formulas given above
                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 ###
```

The result is correct.

```
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

Pooling layer - backward pass

1.6 5.2 Pooling layer - backward pass

Next, let's implement the backward pass for the pooling layer, starting with the MAX-POOL layer. Even though a pooling layer has no parameters for backprop to update, you still need to backpropagation the gradient through the pooling layer in order to compute gradients for layers that came before the pooling layer.

1.6.1 5.2.1 Max pooling - backward pass

Before jumping into the backpropagation of the pooling layer, you are going to build a helper function called `create_mask_from_window()` which does the following:

As you can see, this function creates a "mask" matrix which keeps track of where the maximum of the matrix is. True (1) indicates the position of the maximum in X, the other entries are False (0). You'll see later that the backward pass for average pooling will be similar to this but using a different mask.

Exercise: Implement `create_mask_from_window()`. This function will be helpful for pooling backward. Hints:

- np.max() may be helpful. It computes the maximum of an array.
- If you have a matrix X and a scalar x: `A = (X == x)` will return a matrix A of the same size as X such that:

```
A[i,j] = True if X[i,j] = x
A[i,j] = False if X[i,j] != x
```

- Here, you don't need to consider cases where there are several maxima in a matrix.

Code: follow the hint

```python
def create_mask_from_window(x):
    """
    Creates a mask from an input matrix x, to identify the max entry of x.

    Arguments:
    x -- Array of shape (f, f)

    Returns:
    mask -- Array of the same shape as window, contains a True at the position corresponding to the max entry of x.
    """

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

    return mask
```

The result is correct.

```
x =  [[ 1.62434536 -0.61175641 -0.52817175]
 [-1.07296862  0.86540763 -2.3015387 ]]
mask =  [[ True False False]
 [False False False]]
```

Expected Output:

x =	[[1.62434536 -0.61175641 -0.52817175] [-1.07296862 0.86540763 -2.3015387]]
mask =	[[True False False] [False False False]]

Average pooling - backward pass

1.6.2 5.2.2 - Average pooling - backward pass

In max pooling, for each input window, all the "influence" on the output came from a single input value--the max. In average pooling, every element of the input window has equal influence on the output. So to implement backprop, you will now implement a helper function that reflects this.

For example if we did average pooling in the forward pass using a 2x2 filter, then the mask you'll use for the backward pass will look like:

This implies that each position in the dZ matrix contributes equally to output because in the forward pass, we took an average.

Exercise: Implement the function below to equally distribute a value dz through a matrix of dimension shape. Hint

Code: follow the hint

```python
def distribute_value(dz, shape):
    """
    Distributes the input value in the matrix of dimension shape

    Arguments:
    dz -- input scalar
    shape -- the shape (n_H, n_W) of the output matrix for which we want to distribute the value of dz

    Returns:
    a -- Array of size (n_H, n_W) for which we distributed the value of dz
    """

    ### START CODE HERE ###
    # 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 ###

    return a
```

The result is correct.

```
distributed value = [[0.5 0.5]
 [0.5 0.5]]
```

Expected Output:

distributed_value = [[0.5 0.5] [0.5 0.5]]

Putting it together: Pooling backward

1.6.3 5.2.3 Putting it together: Pooling backward

You now have everything you need to compute backward propagation on a pooling layer.

Exercise: Implement the `pool_backward` function in both modes (`"max"` and `"average"`). You will once again use 4 for-loops (iterating over training examples, height, width, and channels). You should use an `if/elif` statement to see if the mode is equal to `'max'` or `'average'`. If it is equal to 'average' you should use the `distribute_value()` function you implemented above to create a matrix of the same shape as `a_slice`. Otherwise, the mode is equal to '`max`', and you will create a mask with `create_mask_from_window()` and multiply it by the corresponding value of dZ.

Code: follow the hint

```python
### START CODE HERE ###

# Retrieve information from cache (≈1 line)
(A_prev, hparameters) = cache

# Retrieve hyperparameters from "hparameters" (≈2 lines)
stride = hparameters['stride']
f = hparameters['f']

# Retrieve dimensions from A_prev's shape and dA's shape (≈2 lines)
m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
m, n_H, n_W, n_C = dA.shape

# Initialize dA_prev with zeros (≈1 line)
dA_prev = np.zeros(A_prev.shape)
```

```python
    for h in range(n_H):                   # loop on the vertical axis
        for w in range(n_W):               # loop on the horizontal axis
            for c in range(n_C):           # loop over the channels (depth)

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

### END CODE ###
```

The result is correct.

```
mode = max
mean of dA =  0.14571390272918056
dA_prev[1,1] =  [[ 0.          0.        ]
 [ 5.05844394 -1.68282702]
 [ 0.          0.        ]]

mode = average
mean of dA =  0.14571390272918056
dA_prev[1,1] =  [[ 0.08485462  0.2787552 ]
 [ 1.26461098 -0.25749373]
 [ 1.17975636 -0.53624893]]
```

Expected Output:

mode = max:

mean of dA =	0.14571390272918056
dA_prev[1,1] =	[[0. 0.] [10.11330283 -0.49726956] [0. 0.]]

mode = average

mean of dA =	0.14571390272918056
dA_prev[1,1] =	[[2.59843096 -0.27835778] [7.96018612 -1.95394424] [5.36175516 -1.67558646]]

二, Convolutional model — Application

Create placeholders:

1.1.1 1.1 - Create placeholders

TensorFlow requires that you create placeholders for the input data that will be fed into the model when running the session.

Exercise: Implement the function below to create placeholders for the input image X and the output Y. You should not define the number of training examples for the moment. To do so, you could use "None" as the batch size, it will give you the flexibility to choose it later. Hence X should be of dimension **[None, n_H0, n_W0, n_C0]** and Y should be of dimension **[None, n_y]**. Hint.

Code: follow the hint

```python
def create_placeholders(n_H0, n_W0, n_C0, n_y):
    """
    Creates the placeholders for the tensorflow session.

    Arguments:
    n_H0 -- scalar, height of an input image
    n_W0 -- scalar, width of an input image
    n_C0 -- scalar, number of channels of the input
    n_y -- scalar, number of classes

    Returns:
    X -- placeholder for the data input, of shape [None, n_H0, n_W0, n_C0] and dtype "float"
    Y -- placeholder for the input labels, of shape [None, n_y] and dtype "float"
    """

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

    return X, Y
```

Result is correct.

```
X = Tensor("Placeholder:0", shape=(?, 64, 64, 3), dtype=float32)
Y = Tensor("Placeholder_1:0", shape=(?, 6), dtype=float32)
```

Expected Output

X = Tensor("Placeholder:0", shape=(?, 64, 64, 3), dtype=float32)

Y = Tensor("Placeholder_1:0", shape=(?, 6), dtype=float32)

Initialize parameters:

1.1.2 1.2 - Initialize parameters

You will initialize weights/filters $W1$ and $W2$ using `tf.contrib.layers.xavier_initializer(seed = 0)`. You don't need to worry about bias variables as you will soon see that TensorFlow functions take care of the bias. Note also that you will only initialize the weights/filters for the conv2d functions. TensorFlow initializes the layers for the fully connected part automatically. We will talk more about that later in this assignment.

Exercise: Implement initialize_parameters(). The dimensions for each group of filters are provided below. Reminder - to initialize a parameter W of shape [1,2,3,4] in Tensorflow, use:

```python
W = tf.get_variable("W", [1,2,3,4], initializer = ...)
```

Code: follow the hint

```python
def initialize_parameters():
    """
    Initializes weight parameters to build a neural network with tensorflow. The shapes are:
                        W1 : [4, 4, 3, 8]
                        W2 : [2, 2, 8, 16]
    Returns:
    parameters -- a dictionary of tensors containing W1, W2
    """

    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 ###

    parameters = {"W1": W1,
                  "W2": W2}

    return parameters
```

Result is correct.

```
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]

Forward propagation:

1.1.3 1.2 - Forward propagation

In TensorFlow, there are built-in functions that carry out the convolution steps for you.

- **tf.nn.conv2d(X,W1, strides = [1,s,s,1], padding = 'SAME'):** given an input X and a group of filters $W1$, this function convolves $W1$'s filters on X. The third input ([1,f,f,1]) represents the strides for each dimension of the input (m, n_H_prev, n_W_prev, n_C_prev). You can read the full documentation here
- **tf.nn.max_pool(A, ksize = [1,f,f,1], strides = [1,s,s,1], padding = 'SAME'):** given an input A, this function uses a window of size (f, f) and strides of size (s, s) to carry out max pooling over each window. You can read the full documentation here
- **tf.nn.relu(Z1):** computes the elementwise ReLU of Z1 (which can be any shape). You can read the full documentation here.
- **tf.contrib.layers.flatten(P)**: given an input P, this function flattens each example into a 1D vector it while maintaining the batch-size. It returns a flattened tensor with shape [batch_size, k]. You can read the full documentation here.
- **tf.contrib.layers.fully_connected(F, num_outputs):** given a the flattened input F, it returns the output computed using a fully connected layer. You can read the full documentation here.

In the last function above (`tf.contrib.layers.fully_connected`), the fully connected layer automatically initializes weights in the graph and keeps on training them as you train the model. Hence, you did not need to initialize those weights when initializing the parameters.

Exercise:

Implement the `forward_propagation` function below to build the following model: `CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLYCONNECTED`. You should use the functions above.

In detail, we will use the following parameters for all the steps: - Conv2D: stride 1, padding is "SAME" - ReLU - Max pool: Use an 8 by 8 filter size and an 8 by 8 stride, padding is "SAME" - Conv2D: stride 1, padding is "SAME" - ReLU - Max pool: Use a 4 by 4 filter size and a 4 by 4 stride, padding is "SAME" - Flatten the previous output. - FULLYCONNECTED (FC) layer: Apply a fully connected layer without an non-linear activation function. Do not call the softmax here. This will result in 6 neurons in the output layer, which then get passed later to a softmax. In TensorFlow, the softmax and cost function are lumped together into a single function, which you'll call in a different function when computing the cost.

Code: follow the hint

```python
### 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, num_outputs=6, activation_fn=None)
### END CODE HERE ###
```

Result is correct.

```
Z3 = [[-0.44670227 -1.5720876  -1.5304923  -2.3101304  -1.2910438   0.46852064]
 [-0.17601591 -1.5797201  -1.4737016  -2.616721   -1.0081065   0.5747785 ]]
```

Expected Output:

Z3 =	[[-0.44670227 -1.5720876 -1.5304923 -2.3101304 -1.2910438 0.46852064] [-0.17601591 -1.5797201 -1.4737016 -2.616721 -1.0081065 0.5747785]]

Compute cost:

1.1.4 1.3 - Compute cost

Implement the compute cost function below. You might find these two functions helpful:

- **tf.nn.softmax_cross_entropy_with_logits(logits = Z3, labels = Y):** computes the softmax entropy loss. This function both computes the softmax activation function as well as the resulting loss. You can check the full documentation here.
- **tf.reduce_mean:** computes the mean of elements across dimensions of a tensor. Use this to sum the losses over all the examples to get the overall cost. You can check the full documentation here.

** Exercise**: Compute the cost below using the function above.

Code: follow the hint

```python
def compute_cost(Z3, Y):
    """
    Computes the cost

    Arguments:
    Z3 -- output of forward propagation (output of the last LINEAR unit), of shape (6, number of examples)
    Y -- "true" labels vector placeholder, same shape as Z3

    Returns:
    cost - Tensor of the cost function
    """

    ### 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
```

Result is correct.

```
cost = 2.9103396
```

Expected Output:

cost = 2.9103398

Model:

1.2 1.4 Model

Finally you will merge the helper functions you implemented above to build a model. You will train it on the SIGNS dataset.

You have implemented `random_mini_batches()` in the Optimization programming assignment of course 2. Remember that this function returns a list of mini-batches.

Exercise: Complete the function below.

The model below should:

- create placeholders
- initialize parameters
- forward propagate
- compute the cost
- create an optimizer

Finally you will create a session and run a for loop for num_epochs, get the mini-batches, and then for each mini-batch you will optimize the function. Hint for initializing the variables

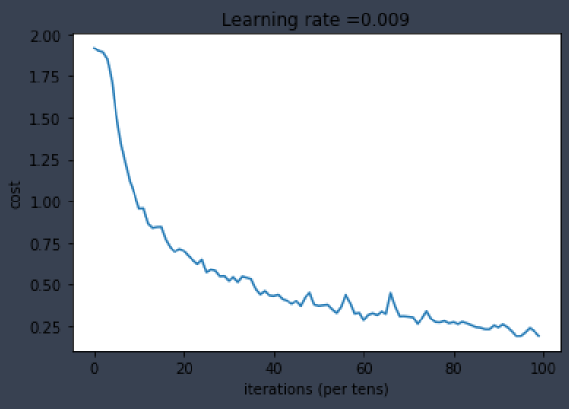
Code: follow the hint

```
# 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 minimizes the cost.
### START CODE HERE ### (1 line)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)
### END CODE HERE ###
```

```
# Do the training loop
for epoch in range(num_epochs):

    minibatch_cost = 0.
    num_minibatches = int(m / minibatch_size) # number of minibatches of size minibatch_size in the train set
    seed = seed + 1
    minibatches = random_mini_batches(X_train, Y_train, minibatch_size, seed)

    for minibatch in minibatches:

        # Select a minibatch
        (minibatch_X, minibatch_Y) = minibatch
        # IMPORTANT: The line that runs the graph on a minibatch.
        # Run the session to execute the optimizer and the cost, the feedict should contain a minibatch for (X,Y).
        ### START CODE HERE ### (1 line)
        _ , temp_cost = sess.run([optimizer,cost],feed_dict={X:minibatch_X,Y:minibatch_Y})
        ### END CODE HERE ###

        minibatch_cost += temp_cost / num_minibatches
```

Result is correct.

```
Cost after epoch 0: 1.917929
Cost after epoch 5: 1.506757
Cost after epoch 10: 0.955359
Cost after epoch 15: 0.845802
Cost after epoch 20: 0.701174
Cost after epoch 25: 0.572085
Cost after epoch 30: 0.521668
Cost after epoch 35: 0.532902
Cost after epoch 40: 0.431018
Cost after epoch 45: 0.401331
Cost after epoch 50: 0.370733
Cost after epoch 55: 0.365420
Cost after epoch 60: 0.283705
Cost after epoch 65: 0.323672
Cost after epoch 70: 0.307099
Cost after epoch 75: 0.294215
Cost after epoch 80: 0.273980
Cost after epoch 85: 0.243726
Cost after epoch 90: 0.242174
Cost after epoch 95: 0.191030
```




```
Tensor("Mean_1:0", shape=(), dtype=float32)
Train Accuracy: 0.9138889
Test Accuracy: 0.7916667
```

Expected output: although it may not match perfectly, your expected output should be close to ours and your cost value should decrease.

Cost after epoch 0 =	1.917920
Cost after epoch 5 =	1.532475
Train Accuracy =	0.86851853
Test Accuracy =	0.73333335

```python
import imageio
from PIL import Image
fname = "images/thumbs_up.jpg"
my_image = np.array(Image.fromarray(imageio.imread(fname)).resize((64,64)))
plt.imshow(my_image)
```

executed in 453ms, finished 11:41:14 2021-11-03

```
<matplotlib.image.AxesImage at 0x7fbd9c7e80f0>
```

三, Residual Networks

The identity block:

Here're the individual steps.

First component of main path:

- The first CONV2D has F_1 filters of shape (1,1) and a stride of (1,1). Its padding is "valid" and its name should be `conv_name_base + '2a'`. Use 0 as the seed for the random initialization.
- The first BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2a'`.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Second component of main path:

- The second CONV2D has F_2 filters of shape (f, f) and a stride of (1,1). Its padding is "same" and its name should be `conv_name_base + '2b'`. Use 0 as the seed for the random initialization.
- The second BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2b'`.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Third component of main path:

- The third CONV2D has F_3 filters of shape (1,1) and a stride of (1,1). Its padding is "valid" and its name should be `conv_name_base + '2c'`. Use 0 as the seed for the random initialization.
- The third BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2c'`. Note that there is no ReLU activation function in this component.

Final step:

- The shortcut and the input are added together.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Exercise: Implement the ResNet identity block. We have implemented the first component of the main path. Please read over this carefully to make sure you understand what it is doing. You should implement the rest.

- To implement the Conv2D step: See reference
- To implement BatchNorm: See reference (axis: Integer, the axis that should be normalized (typically the channels axis))
- For the activation, use: `Activation('relu')(X)`
- To add the value passed forward by the shortcut: See reference

Code: follow the hint

```python
### START CODE HERE ###

# 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', kernel_initializer = glorot_uniform(seed=0)
X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
X = Activation('relu')(X)

# 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_initializer = glorot_uniform(seed=0
X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)

# Final step: Add shortcut value to main path, and pass it through a RELU activation (≈2 lines)
X = Add()([X_shortcut,X])
X = Activation('relu')(X)

### END CODE HERE ###
```

Result is correct.

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

Expected Output:

out [0.94822985 0. 1.1610144 2.747859 0. 1.36677]

The convolutional block:

First component of main path:

- The first CONV2D has F_1 filters of shape (1,1) and a stride of (s,s). Its padding is "valid" and its name should be `conv_name_base + '2a'`.
- The first BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2a'`.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Second component of main path:

- The second CONV2D has F_2 filters of (f,f) and a stride of (1,1). Its padding is "same" and it's name should be `conv_name_base + '2b'`.
- The second BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2b'`.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Third component of main path:

- The third CONV2D has F_3 filters of (1,1) and a stride of (1,1). Its padding is "valid" and it's name should be `conv_name_base + '2c'`.
- The third BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '2c'`. Note that there is no ReLU activation function in this component.

Shortcut path:

- The CONV2D has F_3 filters of shape (1,1) and a stride of (s,s). Its padding is "valid" and its name should be `conv_name_base + '1'`.
- The BatchNorm is normalizing the channels axis. Its name should be `bn_name_base + '1'`.

Final step:

- The shortcut and the main path values are added together.
- Then apply the ReLU activation function. This has no name and no hyperparameters.

Exercise: Implement the convolutional block. We have implemented the first component of the main path; you should implement the rest. As before, always use 0 as the seed for the random initialization, to ensure consistency with our grader.

- Conv Hint
- BatchNorm Hint (axis: Integer, the axis that should be normalized (typically the features axis))
- For the activation, use: `Activation('relu')(X)`
- Addition Hint

Code: follow the hint

```
### START CODE HERE ###

# 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', kernel_initializer = glorot_uniform(seed=0)
X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
X = Activation('relu')(X)

# 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_initializer = glorot_uniform(seed=0
X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)

##### 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_unifor
X_shortcut = BatchNormalization(axis = 3, name = bn_name_base + '1')(X_shortcut)

# Final step: Add shortcut value to main path, and pass it through a RELU activation (≈2 lines)
X = Add()([X_shortcut,X])
X = Activation('relu')(X)

### END CODE HERE ###
```

Result is correct.

```
out = [0.09018463 1.2348977  0.46822017 0.0367176  0.         0.65516603]
```

Expected Output:

out [0.09018463 1.23489773 0.46822017 0.0367176 0. 0.65516603]

Building your first ResNet50 model:

The details of this ResNet-50 model are:

- Zero-padding pads the input with a pad of (3,3)
- Stage 1:
  - The 2D Convolution has 64 filters of shape (7,7) and uses a stride of (2,2). Its name is "conv1".
  - BatchNorm is applied to the channels axis of the input.
  - MaxPooling uses a (3,3) window and a (2,2) stride.
- Stage 2:
  - The convolutional block uses three set of filters of size [64,64,256], "f" is 3, "s" is 1 and the block is "a".
  - The 2 identity blocks use three set of filters of size [64,64,256], "f" is 3 and the blocks are "b" and "c".
- Stage 3:
  - The convolutional block uses three set of filters of size [128,128,512], "f" is 3, "s" is 2 and the block is "a".
  - The 3 identity blocks use three set of filters of size [128,128,512], "f" is 3 and the blocks are "b", "c" and "d".
- Stage 4:
  - The convolutional block uses three set of filters of size [256, 256, 1024], "f" is 3, "s" is 2 and the block is "a".
  - The 5 identity blocks use three set of filters of size [256, 256, 1024], "f" is 3 and the blocks are "b", "c", "d", "e" and "f".
- Stage 5:
  - The convolutional block uses three set of filters of size [512, 512, 2048], "f" is 3, "s" is 2 and the block is "a".
  - The 2 identity blocks use three set of filters of size [256, 256, 2048], "f" is 3 and the blocks are "b" and "c".
- The 2D Average Pooling uses a window of shape (2,2) and its name is "avg_pool".
- The flatten doesn't have any hyperparameters or name.
- The Fully Connected (Dense) layer reduces its input to the number of classes using a softmax activation. Its name should be `'fc' + str(classes)`.

Exercise: Implement the ResNet with 50 layers described in the figure above. We have implemented Stages 1 and 2. Please implement the rest. (The syntax for implementing Stages 3-5 should be quite similar to that of Stage 2.) Make sure you follow the naming convention in the text above.

You'll need to use this function:

- Average pooling see reference

Here're some other functions we used in the code below:

- Conv2D: See reference
- BatchNorm: See reference (axis: Integer, the axis that should be normalized (typically the features axis))
- Zero padding: See reference
- Max pooling: See reference
- Fully conected layer: See reference
- Addition: See reference

Code: follow the hint

```
### START CODE HERE ###

# 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')

# 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')

# 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, [256, 256, 2048], stage=5, block='b')
X = identity_block(X, 3, [256, 256, 2048], stage=5, block='c')

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

### END CODE HERE ###
```

Model.fit:

```
Epoch 1/2
1080/1080 [==============================] - 24s 22ms/step - loss: 2.0567 - acc: 0.4657
Epoch 2/2
1080/1080 [==============================] - 19s 17ms/step - loss: 0.6299 - acc: 0.7907


<keras.callbacks.History at 0x7f8c4d2b6cf8>
```

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.

Model.evauate:

```
120/120 [==============================] - 1s 10ms/step
Loss = 2.737965456644694
Test Accuracy = 0.16666666666666666
```

Expected Output:

Test Accuracy between 0.16 and 0.25

After change some params and add some tricks, test acc can achieve 0.975!!!

```python
from keras import optimizers
from keras.callbacks import ReduceLROnPlateau,EarlyStopping
model2 = ResNet50(input_shape = (64, 64, 3), classes = 6)
optimizer=optimizers.Adam(lr=0.001)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.00001, mode='auto', verbose=1)
earlystop=EarlyStopping(monitor='val_acc', min_delta=0.0001, patience=5, verbose=1, mode='auto')
model2.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
model2.fit(X_train, Y_train,
           epochs = 50,
           validation_split=0.1,
           batch_size = 32,
           callbacks=[reduce_lr]
          )
```

```
Epoch 00044: reducing learning rate to 1e-05.
972/972 [==============================] - 17s 17ms/step - loss: 0.0687 - acc: 0.9805 - val_loss: 0.3217 - val_acc: 0.9444
Epoch 45/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0583 - acc: 0.9835 - val_loss: 0.3175 - val_acc: 0.9444
Epoch 46/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0641 - acc: 0.9835 - val_loss: 0.3127 - val_acc: 0.9444
Epoch 47/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0558 - acc: 0.9887 - val_loss: 0.3145 - val_acc: 0.9444
Epoch 48/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0557 - acc: 0.9794 - val_loss: 0.3131 - val_acc: 0.9537
Epoch 49/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0485 - acc: 0.9866 - val_loss: 0.3107 - val_acc: 0.9537
Epoch 50/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0376 - acc: 0.9938 - val_loss: 0.3091 - val_acc: 0.9444

<keras.callbacks.History at 0x7f6d4423e358>
```

```python
preds = model2.evaluate(X_test, Y_test)
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```

xecuted in 496ms, finished 17:14:59 2021-11-03

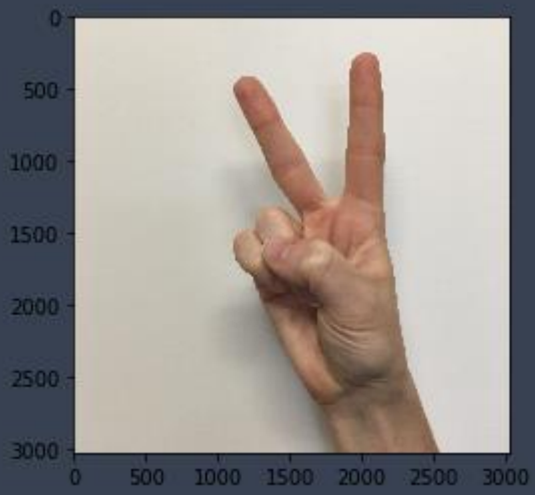
```
120/120 [==============================] - 0s 4ms/step
Loss = 0.14251829956968626
Test Accuracy = 0.975
```

Load ResNet50.h5:

```
120/120 [==============================] - 2s 14ms/step
Loss = 0.5301783204078674
Test Accuracy = 0.8666666626930237
```

Test on my own image:

```
Input image shape: (1, 64, 64, 3)
class prediction vector [p(0), p(1), p(2), p(3), p(4), p(5)] =
[[1. 0. 0. 0. 0. 0.]]
```



Model paeams:

```
Total params: 23,600,006
Trainable params: 23,546,886
Non-trainable params: 53,120
```

结论分析与体会：

1， ResNet50 简单调参后对此数据集可以达到 0.975 的测试准确度。

2， 在手写网络的时候一定要注意维度对齐。

3， 要熟练掌握 BP 的实现原理以及推导公式。

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，没有调参的网络经过 2epoch 训练后准确度太低，需要调参而且适当使用小 trick。这里使用学习率递减的 Adam 优化器，epoch 设置为 50，而且训练时分出 10%的数据用于验证，最后测试准确度最高从 0.167 修改到 0.975。

```
from keras import optimizers
from keras.callbacks import ReduceLROnPlateau,EarlyStopping
model2 = ResNet50(input_shape = (64, 64, 3), classes = 6)
optimizer=optimizers.Adam(lr=0.001)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.00001, mode='auto', verbose=1)
earlystop=EarlyStopping(monitor='val_acc', min_delta=0.0001, patience=5, verbose=1, mode='auto')
model2.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
model2.fit(X_train, Y_train,
           epochs = 50,
           validation_split=0.1,
           batch_size = 32,
           callbacks=[reduce_lr]
          )
```

```
Epoch 00044: reducing learning rate to 1e-05.
972/972 [==============================] - 17s 17ms/step - loss: 0.0687 - acc: 0.9805 - val_loss: 0.3217 - val_acc: 0.9444
Epoch 45/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0583 - acc: 0.9835 - val_loss: 0.3175 - val_acc: 0.9444
Epoch 46/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0641 - acc: 0.9835 - val_loss: 0.3127 - val_acc: 0.9444
Epoch 47/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0558 - acc: 0.9887 - val_loss: 0.3145 - val_acc: 0.9444
Epoch 48/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0557 - acc: 0.9794 - val_loss: 0.3131 - val_acc: 0.9537
Epoch 49/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0485 - acc: 0.9866 - val_loss: 0.3107 - val_acc: 0.9537
Epoch 50/50
972/972 [==============================] - 17s 17ms/step - loss: 0.0376 - acc: 0.9938 - val_loss: 0.3091 - val_acc: 0.9444


<keras.callbacks.History at 0x7f6d4423e358>
```

```python
preds = model2.evaluate(X_test, Y_test)
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```

xecuted in 496ms, finished 17:14:59 2021-11-03

```
120/120 [==============================] - 0s 4ms/step
Loss = 0.14251829956968626
Test Accuracy = 0.975
```

第二次跑，测试准确度 0.95

```
Epoch 45/50
972/972 [==============================] - 17s 18ms/step - loss: 6.3114e-04 - acc: 1.0000 - val_loss: 0.3321 - val_acc: 0.9259
Epoch 46/50
972/972 [==============================] - 17s 18ms/step - loss: 0.0184 - acc: 0.9990 - val_loss: 0.3306 - val_acc: 0.9259
Epoch 47/50
972/972 [==============================] - 17s 18ms/step - loss: 8.2819e-04 - acc: 1.0000 - val_loss: 0.3274 - val_acc: 0.9352
Epoch 48/50
972/972 [==============================] - 17s 18ms/step - loss: 8.3620e-04 - acc: 1.0000 - val_loss: 0.3256 - val_acc: 0.9352
Epoch 49/50
972/972 [==============================] - 17s 18ms/step - loss: 8.7940e-04 - acc: 1.0000 - val_loss: 0.3219 - val_acc: 0.9352
Epoch 50/50
972/972 [==============================] - 17s 18ms/step - loss: 5.1903e-04 - acc: 1.0000 - val_loss: 0.3198 - val_acc: 0.9352


<keras.callbacks.History at 0x7f56847b04a8>
```

```python
#after change some params and use some tricks.
preds = model2.evaluate(X_test, Y_test)
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```

xecuted in 504ms, finished 18:37:22 2021-11-03

```
120/120 [==============================] - 0s 4ms/step
Loss = 0.16242215298116208
Test Accuracy = 0.95
```