机器学习系列（6）

计算机视觉--ResNets、Inception原理及ResNets的Python实现

计算机视觉中的网络及方法：

- 经典网络
- 残差网络
- ResNets有效原因
- 1×1卷积
- 谷歌Inception网络
- 迁移学习
- 数据扩充

Python实现：

- 见文章内容

申明

本文原理解释及公式推导部分均由LSayhi完成，供学习参考，可传播；代码实现部分的框架由Coursera提供，由LSayhi完成，详细数据及代码可在github查阅。 https://github.com/LSayhi/Neural-network-and-Deep-learning (https://github.com/LSayhi/Neural-network-and-Deep-learning)

一、计算机视觉中的网络及方法

1.经典网络：

- 在深度学习的发展中，诞生了许多经典网络模型，这些网络给后来的网络结构提供了重大的参考，有的现在已经不怎么使用了，但是不管怎么样，这些网络的提出，不论在当时还是现在都有一定的意义，比如LeNet5、AlexNet、VGGNet等。这些网络一般是由卷积层和池化层以及输入输出层复合而成，通常，在一个或若干个卷积层之后，会连接一个池化层，在网络的最后几层，一般是将卷积结构转成全连接结构，然后再输出层使用softmax进行分类，随着网络的发展，网络的深度也越来越深，参数也越来越多。

2.残差网络：

- 残差网络($ResidualNetworks$,简称$ResNets$)是由残差块等复合而成的网络结构，利用残差网络可以将网络层数做到很深，以取得更好的表现。我们知道，在网路的训练中，可能会遇到梯度消失或梯度爆炸的现象，导致无法成功训练，在现代网络中，通常网络的层数很大，因此发生梯度消失和梯度爆炸的可能性大大增加。残差网络可以有效防止这些现象的发生。

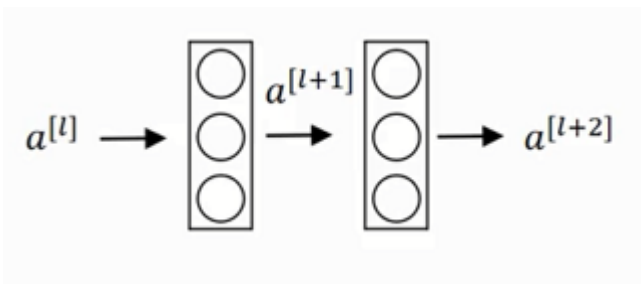


Figure 1 : **plain network**

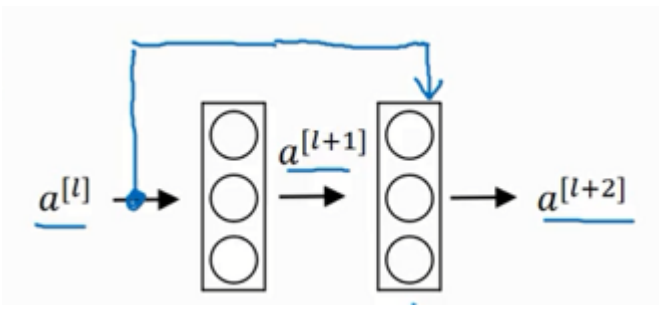


Figure 2 : **skip connection**

- 残差块与跳跃连接。残差块是组成残差网络的重要子系统。如图$Figure1$示例，这是一个一般的网络结构，每一层都依次和下一层连接，第L层的输出取决于第$L-1$的的输出和两层之间的连接方式。残差网络在这个基础上，加入了跳跃连接（$skipconnection$）,如图$Figure2$所示，在$a^{[l+2]}=g(z^{[l+2]}+a^{[l]})$,在普通的网络中，是没有$a^{[l]}$这一项的,这个加入的项是从第$l$层直接连接过来的，中间隔着其它层，因此我们把这个连接称作跳跃连接。如图$Figure3$所示，这是一个残差网络的结构示意图，随着层数的增加，普通的深度网络和残差网络的$loss$的比较。

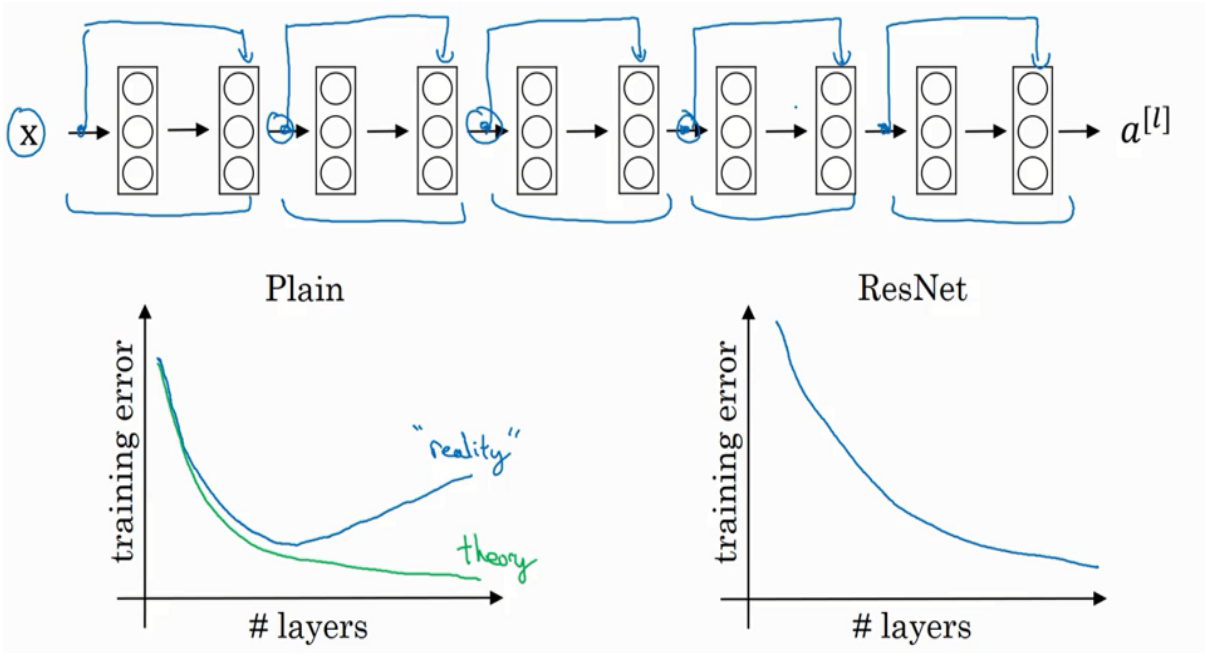


Figure 3 : **residual network 示例**

3.残差网络有效原因:

- 一个网络的深度越深，它在训练集上的表现就有所减弱，这让我们不能把网络做得太深，然而，构造更深的网络可以得到更高级的特征，为了解决这一矛盾，残差网络就发挥了作用。通常，我们希望能做到在加深网络的深度同时不减弱网络的性能。因为一个在训练集上表现良好的网络是其在开发集和测试集取得优异表现的基础。残差网络能够很容易学习到恒等式并且不影响网络性能，至少不降低性能，有些时候反而能学习到前馈网络很难学习到的映射关系，能提升网络性能。这一点可以通过一下例子直观理解，见$Figure 4$,当W和b为0时，残差网络模块就学习到了恒等式。

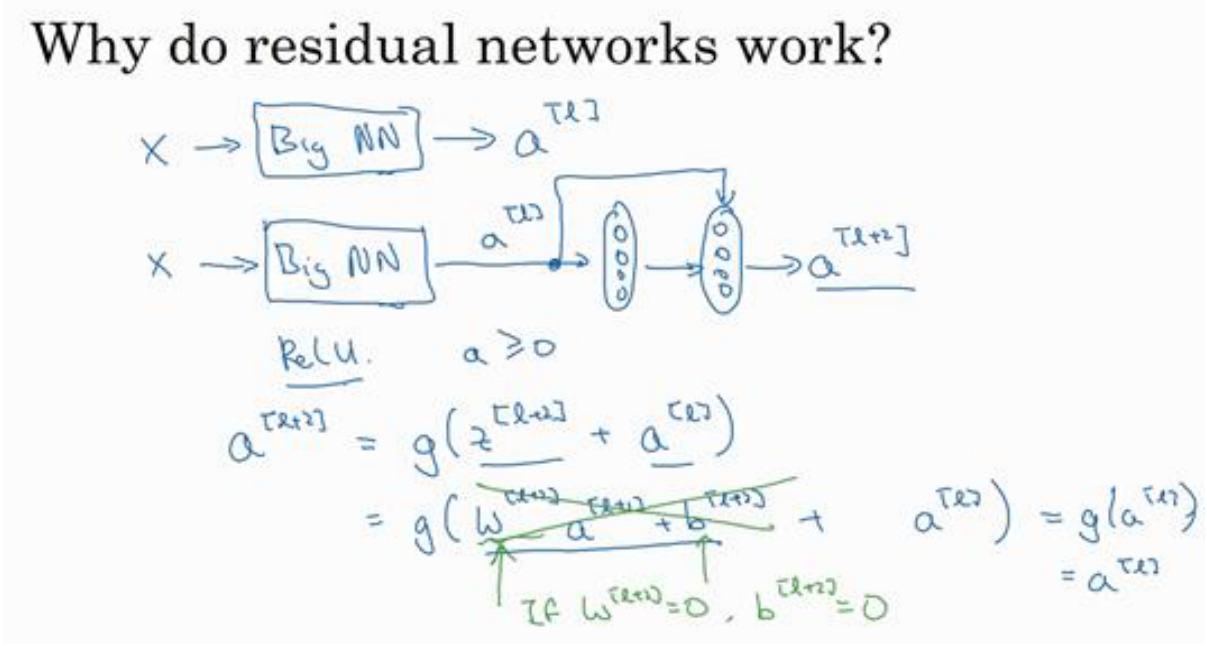


Figure 4 : **学习恒等式**

4.1×1卷积:

- $1 * 1$卷积，即是卷积核的大小是$1 * 1 * Nc$大小的的卷积操作，也称network in network。当Nc=1时，如$Figure 5$所示，相当于把每一个像素点乘以卷积核中的值，当Nc>1时（即图像的通道数大于1时），其相当于对Nc个通道进行加权求和，如图$Figure 6$,当卷积核的数量为filters时，输出最后一个维度则等于filters。$1 * 1$卷积可以改变图像的通道数，通常是减小或保持不变，当然也可以增加，这取决于卷积核的个数，举个例子说明$1 * 1$卷积的作用，如图$Figure 7$,卷积核数量为192时，输出通道数保持不变，通道数为32时，输出通道数减小为32。

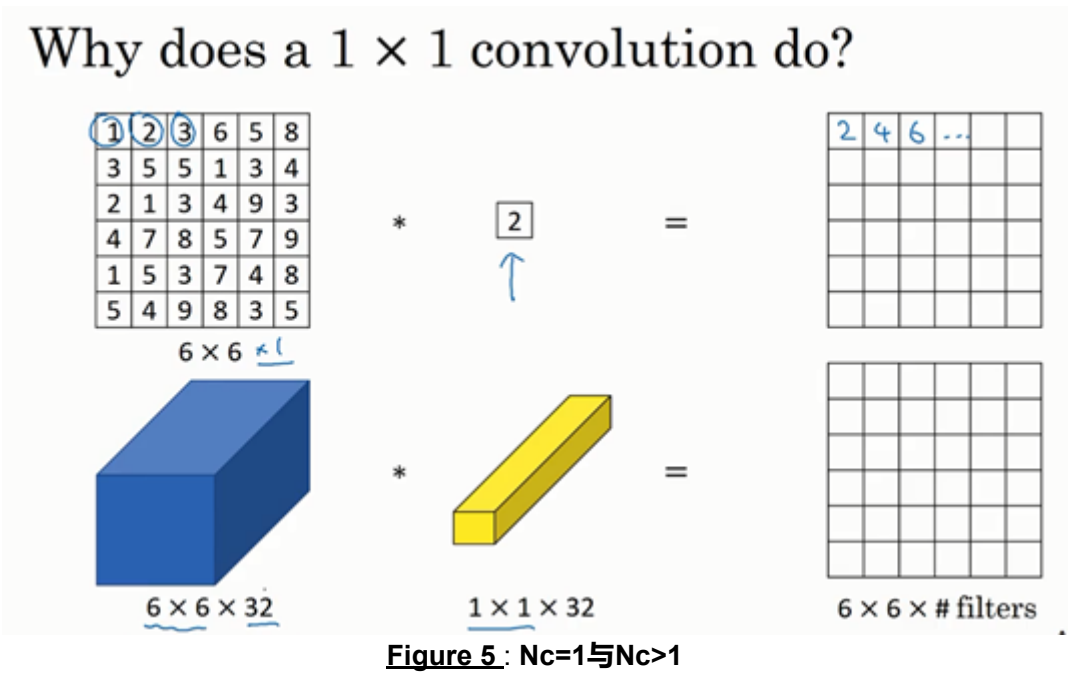


Figure 5 : **Nc=1与Nc>1**

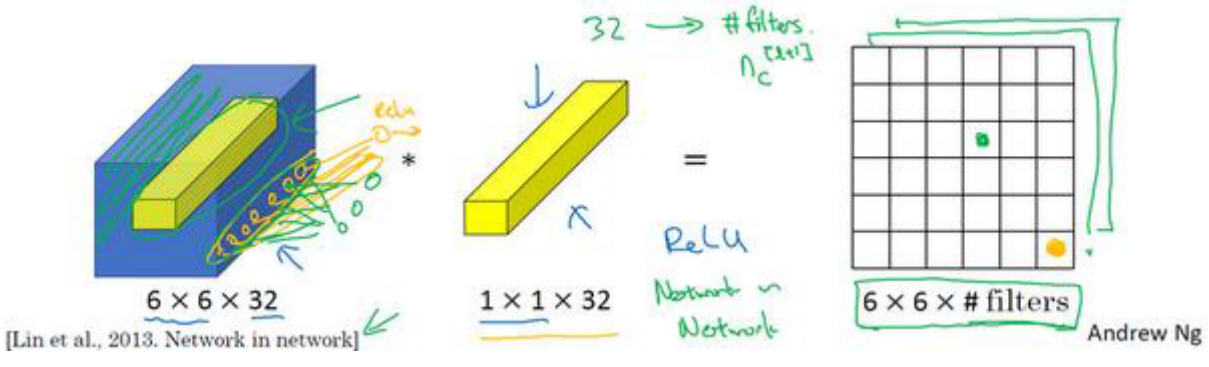


Figure 6 : **多个卷积核**

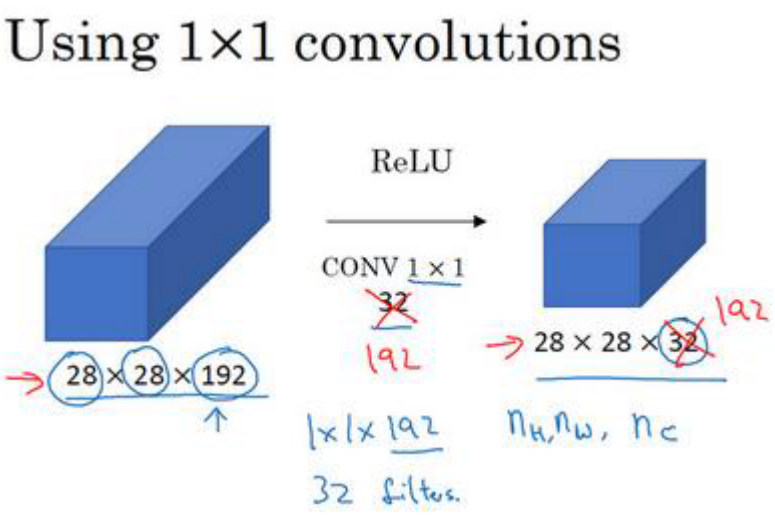


Figure 7 : **改变卷积核数量以改变通道大小**

谷歌Inception网络:

- 谷歌Inception网络基本思想是让网络自己决定需要什么样卷积核以及是否需要池化操作。我们知道。卷积核的大小需要我们去确定，比如$1*1, 3*3, 5*5$等，但是我们不好判断哪种最合适，以及池化层是否需要在某个位置添加，Inception网络就是让网络学习参数，从而决定采用什么样的过滤器的组合。$Figure 8$给出了一个Inception网络的例子,通过使用same卷积，这些输出保持相同维度，只有通道数的不同。

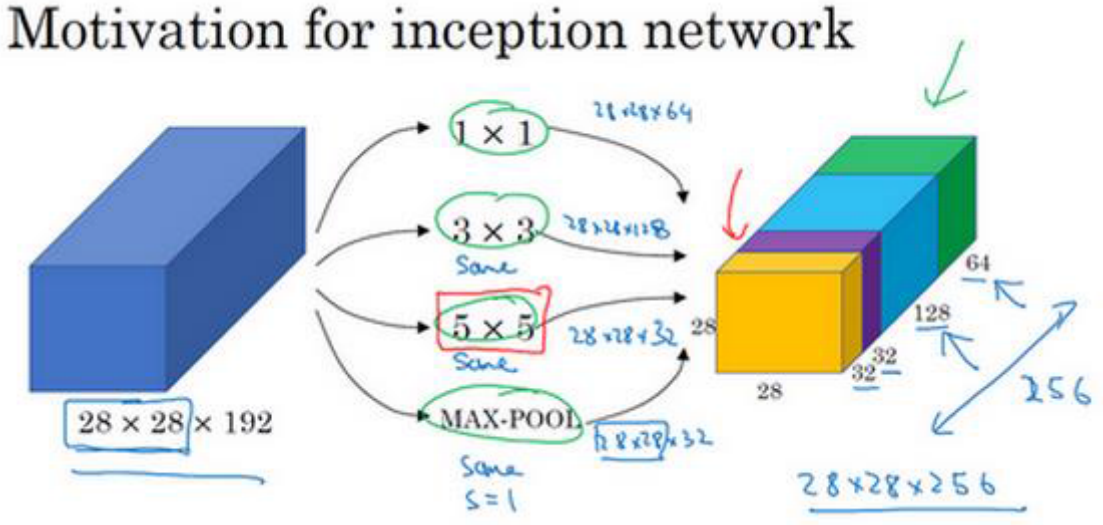


Figure 8 : **Inception网络**

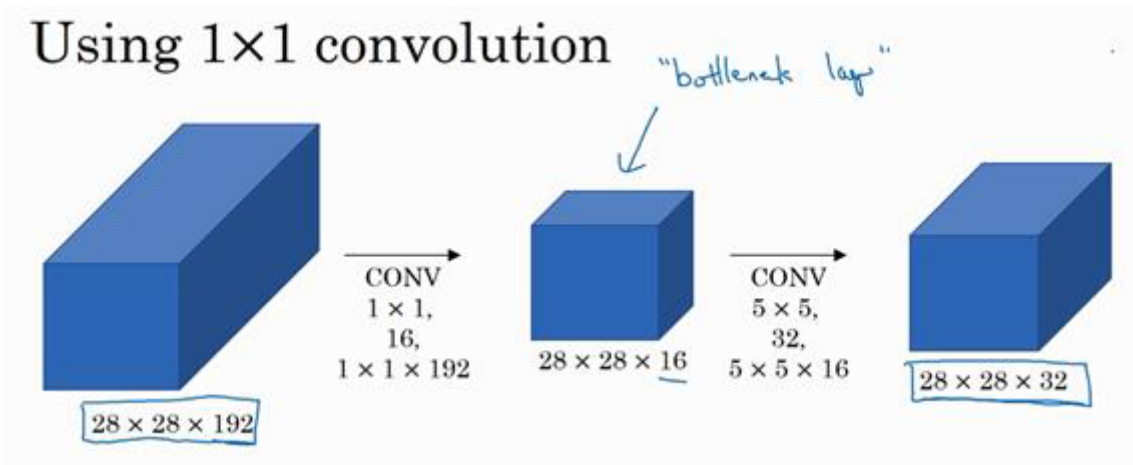


Figure 9 : **使用1*1卷积减小计算成本**

- 不难发现，Inception网络有一个缺点就是需要更多的参数，因此会增加计算成本。那么，此时如果在输入输出之间先经过一个$1*1$卷积,就可以先降低通道数，从而使得计算量下降，这就是$1*1$卷积在Inception网络中的应用，$Figure 9$是一个示例。其可以用在单个Inception模块中，如图$Figure 10$所示是一个完整的单个Inception模块（类似resnet的单模块），然后其可以组成$Figure 11$所示的完整Inception网络，网络由多个Inception模块组成，当然还可以加上一些分支用来对提取到的特征进行预测分类等。

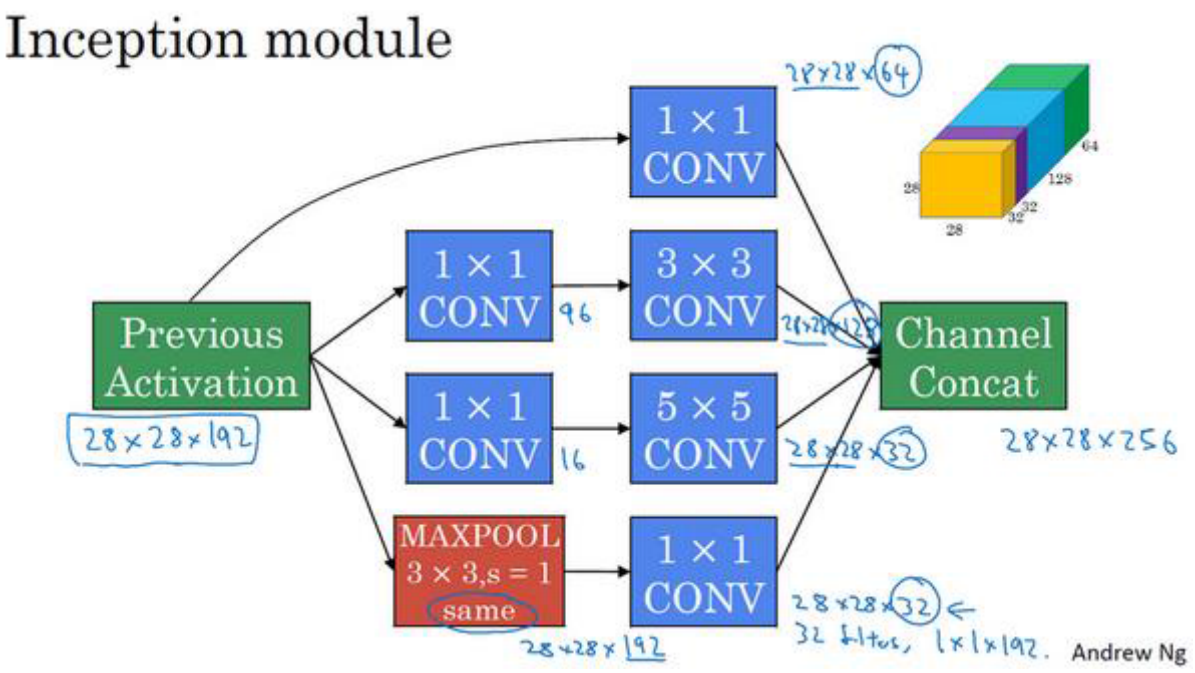


Figure 10 : **Inception模块**

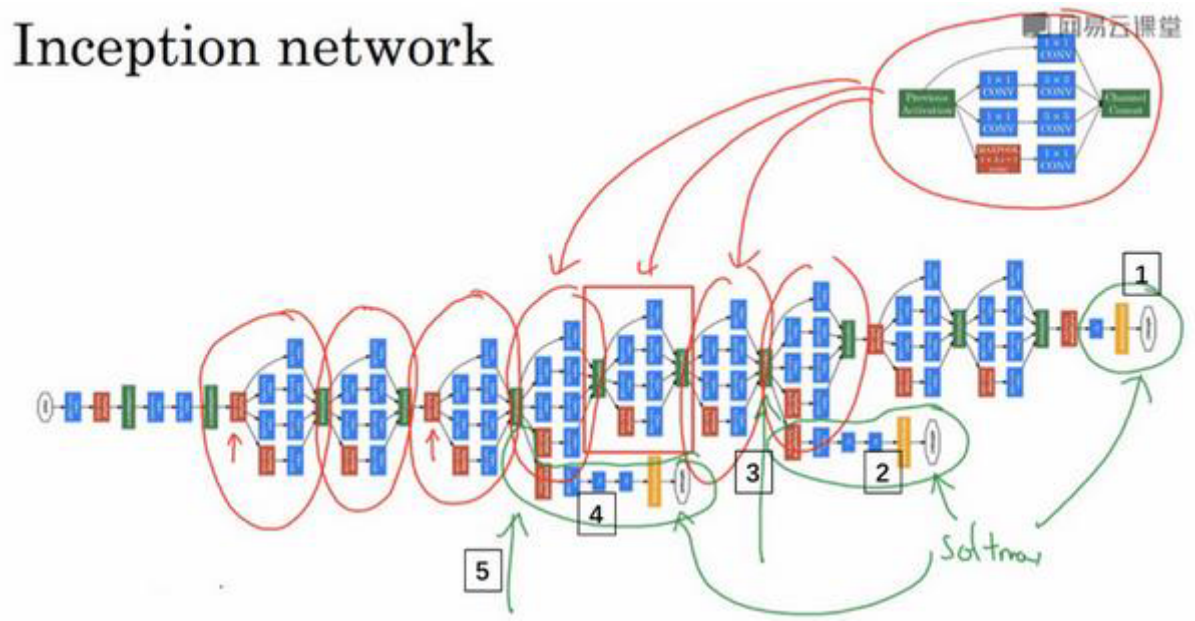


Figure 11 : **Inception网络**

迁移学习:

- 众所周知，训练一个有效的深度网络需要很长的时间和一些调参技巧，如果遇到一个问题就直接训练模型，那么花费的时间将相当长，而且，有时候由于数据量少，也很难通过数据训练出有效的网络。这时迁移学习就派上了用场，迁移学习可以允许我们使用已经提出的网络模型和其训练好的参数，稍作修改，就很可能解决这两个问题。具体来说，假如你要训练一个猫狗猪分类器，你可以从开源网站上下载一个分类器，假设这个分类器可以识别100种动物类别，那么你可以将最后的softmax层从100个输出改为3个，然后保持其它参数不变，只训练和softmax层有关的参数，这样就可以训练出一个良好的识别猫狗猪的网络模型。当然，如果你要识别的是20种动物，你可能需要更改更多的参数，比如说将网络最后几层的参数都进行训练，然后softmax20个输出。能使用迁移学习的原因是卷积神经网络的每一步操作都相当于在提取图像特征，参数即是提取特征的方法。

数据扩充:

- 在训练网络时，可能会遇到训练集不够的情况，使得网络很难训练，这时就可以考虑数据扩充方法，扩充训练集，从而更好地训练网络。在计算机视觉领域，常见的数据扩充手段有图像镜像、图像颠倒、图像扭曲、图像变色等等，如图$Figure 12$。将这些操作应用在原始数据集上，可以产生很多新的图像，把原始图像数据集和新产生的图像数据集合并为一个新的数据集，这就是计算机视觉中的数据扩充。

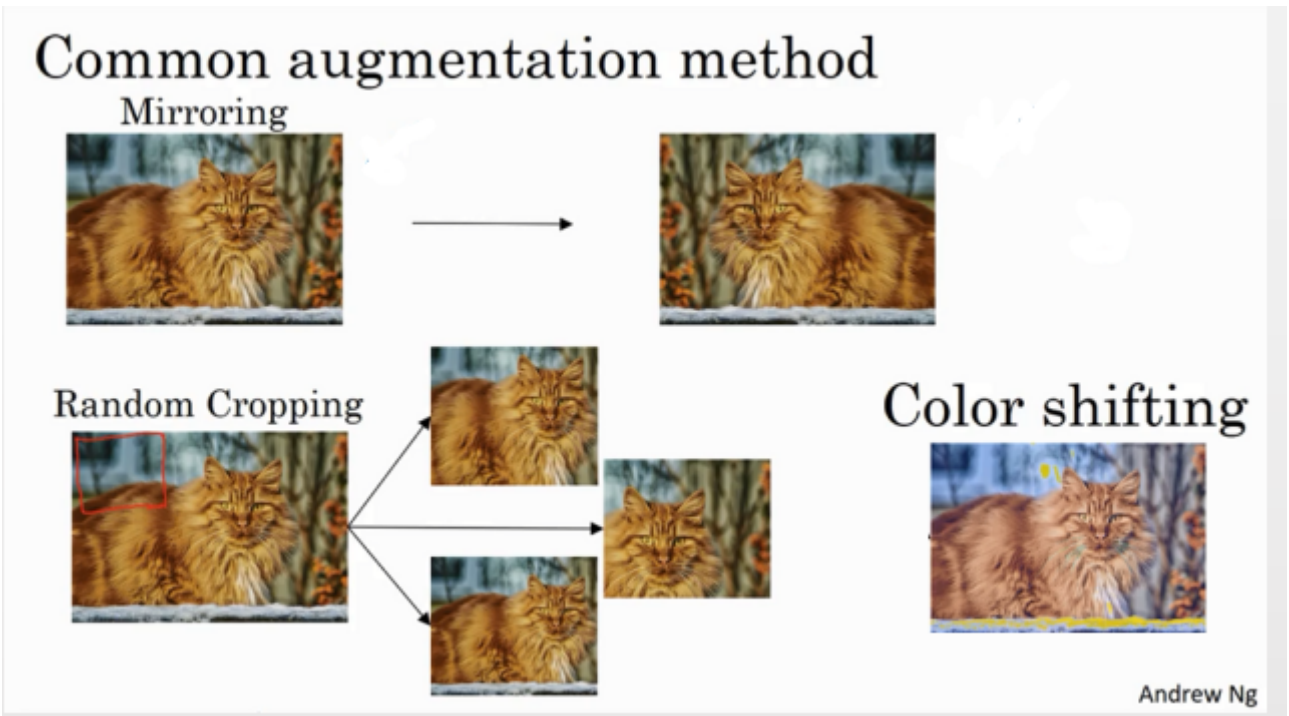


Figure 12 : **数据扩充**

二、ResNets的python实现

Welcome to the second assignment of this week! You will learn how to build very deep convolutional networks, using Residual Networks (ResNets). In theory, very deep networks can represent very complex functions; but in practice, they are hard to train. Residual Networks, introduced by He et al. (https://arxiv.org/pdf/1512.03385.pdf), allow you to train much deeper networks than were previously practically feasible.

In this assignment, you will:

- Implement the basic building blocks of ResNets.
- Put together these building blocks to implement and train a state-of-the-art neural network for image classification.

This assignment will be done in Keras.

Before jumping into the problem, let's run the cell below to load the required packages.

In [1]:
```
import numpy as np
from keras import layers
from keras.layers import Input, Add, Dense, Activation, ZeroPadding2D, BatchNormalization, Flatten, Conv2D, AveragePooling2D, MaxPooli
from keras.models import Model, load_model
from keras.preprocessing import image
from keras.utils import layer_utils
from keras.utils.data_utils import get_file
from keras.applications.imagenet_utils import preprocess_input
import pydot
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
from keras.utils import plot_model
from resnets_utils import *
from keras.initializers import glorot_uniform
import scipy.misc
from matplotlib.pyplot import imshow
%matplotlib inline

import keras.backend as K
K.set_image_data_format('channels_last')
K.set_learning_phase(1)
```

```
Using TensorFlow backend.
```

1 - The problem of very deep neural networks

Last week, you built your first convolutional neural network. In recent years, neural networks have become deeper, with state-of-the-art networks going from just a few layers (e.g., AlexNet) to over a hundred layers.

The main benefit of a very deep network is that it can represent very complex functions. It can also learn features at many different levels of abstraction, from edges (at the lower layers) to very complex features (at the deeper layers). However, using a deeper network doesn't always help. A huge barrier to training them is vanishing gradients: very deep networks often have a gradient signal that goes to zero quickly, thus making gradient descent unbearably slow. More specifically, during gradient descent, as you backprop from the final layer back to the first layer, you are multiplying by the weight matrix on each step, and thus the gradient can decrease exponentially quickly to zero (or, in rare cases, grow exponentially quickly and "explode" to take very large values).

During training, you might therefore see the magnitude (or norm) of the gradient for the earlier layers descrease to zero very rapidly as training proceeds:

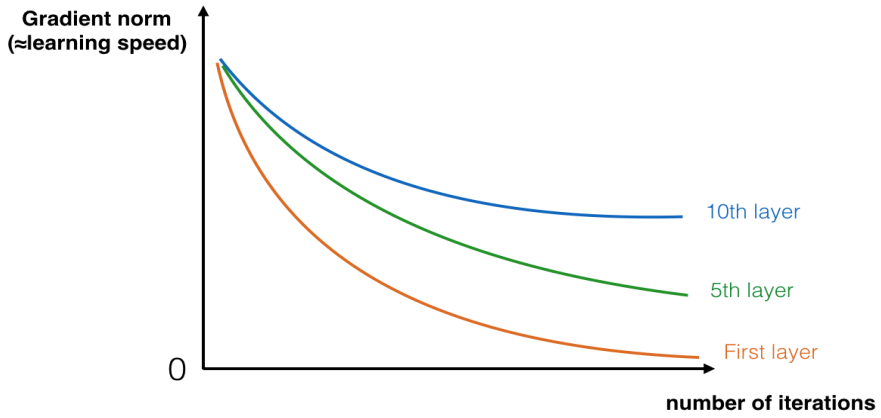


Figure 1 : **Vanishing gradient**
The speed of learning decreases very rapidly for the early layers as the network trains

You are now going to solve this problem by building a Residual Network!

2 - Building a Residual Network

In ResNets, a "shortcut" or a "skip connection" allows the gradient to be directly backpropagated to earlier layers:

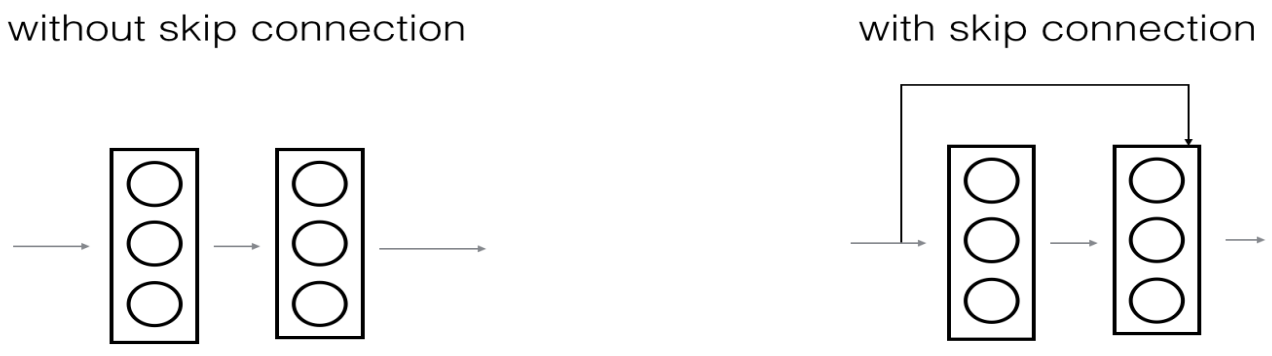


Figure 2 : A ResNet block showing a **skip-connection**

The image on the left shows the "main path" through the network. The image on the right adds a shortcut to the main path. By stacking these ResNet blocks on top of each other, you can form a very deep network.

We also saw in lecture that having ResNet blocks with the shortcut also makes it very easy for one of the blocks to learn an identity function. This means that you can stack on additional ResNet blocks with little risk of harming training set performance. (There is also some evidence that the ease of learning an identity function--even more than skip connections helping with vanishing gradients--accounts for ResNets' remarkable performance.)

Two main types of blocks are used in a ResNet, depending mainly on whether the input/output dimensions are same or different. You are going to implement both of them.

2.1 - The identity block

The identity block is the standard block used in ResNets, and corresponds to the case where the input activation (say $a^{[l]}$) has the same dimension as the output activation (say $a^{[l+2]}$). To flesh out the different steps of what happens in a ResNet's identity block, here is an alternative diagram showing the individual steps:

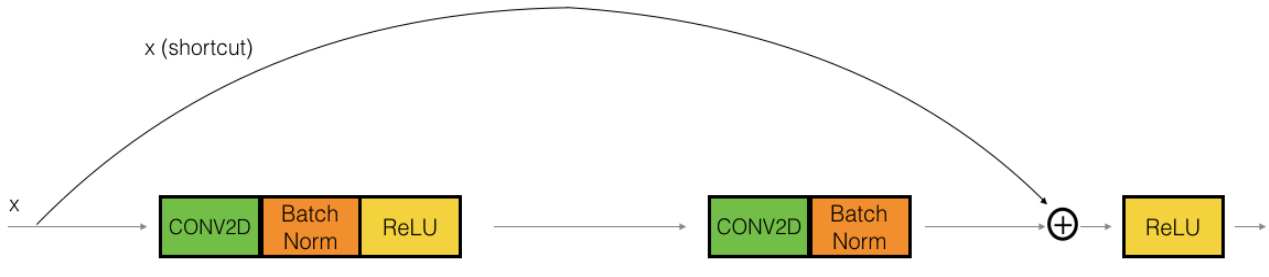


Figure 3 : **Identity block.** Skip connection "skips over" 2 layers.

The upper path is the "shortcut path." The lower path is the "main path." In this diagram, we have also made explicit the CONV2D and ReLU steps in each layer. To speed up training we have also added a BatchNorm step. Don't worry about this being complicated to implement--you'll see that BatchNorm is just one line of code in Keras!

In this exercise, you'll actually implement a slightly more powerful version of this identity block, in which the skip connection "skips over" 3 hidden layers rather than 2 layers. It looks like this:

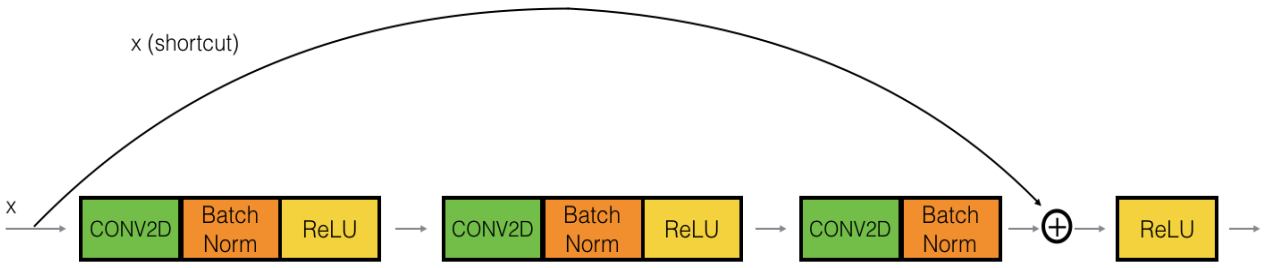


Figure 4 : **Identity block.** Skip connection "skips over" 3 layers.

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 (https://keras.io/layers/convolutional/#conv2d)
- To implement BatchNorm: See reference (https://faroit.github.io/keras-docs/1.2.2/layers/normalization/) (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 (https://keras.io/layers/merge/#add)

In [2]:
```
# GRADED FUNCTION: identity_block

def identity_block(X, f, filters, stage, block):
    """
    Implementation of the identity block as defined in Figure 3

    Arguments:
    X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
    f -- integer, specifying the shape of the middle CONV's window for the main path
    filters -- python list of integers, defining the number of filters in the CONV layers of the main path
    stage -- integer, used to name the layers, depending on their position in the network
    block -- string/character, used to name the layers, depending on their position in the network

    Returns:
    X -- output of the identity block, tensor of shape (n_H, n_W, n_C)
    """

    # defining name basis
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    # Retrieve Filters
    F1, F2, F3 = filters

    # Save the input value. You'll need this later to add back to the main path.
    X_shortcut = X

    # First component of main path
    X = Conv2D(filters = F1, kernel_size = (1, 1), strides = (1,1), padding = 'valid', name = conv_name_base + '2a', kernel_initialize
    X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    ### 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
    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_initialize
    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,X_shortcut])
    X = Activation('relu')(X)

    ### END CODE HERE ###

    return X
```

In [4]:
```
tf.reset_default_graph()

with tf.Session() as test:
    np.random.seed(1)
    A_prev = tf.placeholder("float", [3, 4, 4, 6])
    X = np.random.randn(3, 4, 4, 6)
    A = identity_block(A_prev, f = 2, filters = [2, 4, 6], stage = 1, block = 'a')
    test.run(tf.global_variables_initializer())
    out = test.run([A], feed_dict={A_prev: X, K.learning_phase(): 0})
    print("out = " + str(out[0][1][1][0]))
```

```
out = [ 0.94822985  0.          1.16101444  2.747859    0.          1.36677003]
```

Expected Output:

out [0.94822985 0. 1.16101444 2.747859 0. 1.36677003]

2.2 - The convolutional block

You've implemented the ResNet identity block. Next, the ResNet "convolutional block" is the other type of block. You can use this type of block when the input and output dimensions don't match up. The difference with the identity block is that there is a CONV2D layer in the shortcut path:

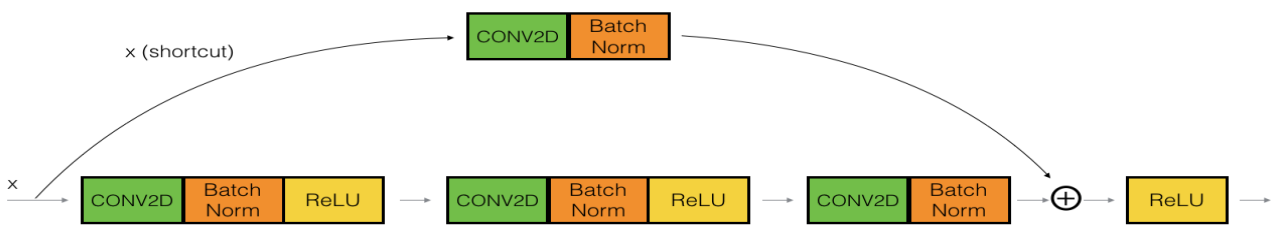


Figure 4 : **Convolutional block**

The CONV2D layer in the shortcut path is used to resize the input x to a different dimension, so that the dimensions match up in the final addition needed to add the shortcut value back to the main path. (This plays a similar role as the matrix W_s discussed in lecture.) For example, to reduce the activation dimensions's height and width by a factor of 2, you can use a 1x1 convolution with a stride of 2. The CONV2D layer on the shortcut path does not use any non-linear activation function. Its main role is to just apply a (learned) linear function that reduces the dimension of the input, so that the dimensions match up for the later addition step.

The details of the convolutional block are as follows.

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 (https://keras.io/layers/convolutional/#conv2d)
- BatchNorm Hint (https://keras.io/layers/normalization/#batchnormalization) (axis: Integer, the axis that should be normalized (typically the features axis))
- For the activation, use: `Activation('relu')(X)`
- Addition Hint (https://keras.io/layers/merge/#add)

In [11]:
```
# GRADED FUNCTION: convolutional_block

def convolutional_block(X, f, filters, stage, block, s = 2):
    """
    Implementation of the convolutional block as defined in Figure 4

    Arguments:
    X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
    f -- integer, specifying the shape of the middle CONV's window for the main path
    filters -- python list of integers, defining the number of filters in the CONV layers of the main path
    stage -- integer, used to name the layers, depending on their position in the network
    block -- string/character, used to name the layers, depending on their position in the network
    s -- Integer, specifying the stride to be used

    Returns:
    X -- output of the convolutional block, tensor of shape (n_H, n_W, n_C)
    """

    # defining name basis
    conv_name_base = 'res' + str(stage) + block + '_branch'
    bn_name_base = 'bn' + str(stage) + block + '_branch'

    # Retrieve Filters
    F1, F2, F3 = filters

    # Save the input value
    X_shortcut = X


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

    ### START CODE HERE ###

    # Second component of main path (≈3 lines)
    X = Conv2D(F2, (f, f), strides = (1, 1), name = conv_name_base + '2b',padding='same', 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(F3, (1, 1), strides = (1, 1), name = conv_name_base + '2c',padding='valid', kernel_initializer = glorot_uniform(seed=0)
    X = BatchNormalization(axis = 3, name = bn_name_base + '2c')(X)

    ##### SHORTCUT PATH #### (≈2 lines)
    X_shortcut = Conv2D(F3, (1, 1), strides = (s, s), name = conv_name_base + '1',padding='valid', kernel_initializer = glorot_uniform
    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, X_shortcut])
    X = Activation('relu')(X)

    ### END CODE HERE ###

    return X
```

In [12]:
```
tf.reset_default_graph()

with tf.Session() as test:
    np.random.seed(1)
    A_prev = tf.placeholder("float", [3, 4, 4, 6])
    X = np.random.randn(3, 4, 4, 6)
    A = convolutional_block(A_prev, f = 2, filters = [2, 4, 6], stage = 1, block = 'a')
    test.run(tf.global_variables_initializer())
    out = test.run([A], feed_dict={A_prev: X, K.learning_phase(): 0})
    print("out = " + str(out[0][1][1][0]))
```

```
out = [ 0.09018463  1.23489773  0.46822017  0.0367176   0.          0.65516603]
```

Expected Output:

out	[0.09018463 1.23489773 0.46822017 0.0367176 0. 0.65516603]

3 - Building your first ResNet model (50 layers)

You now have the necessary blocks to build a very deep ResNet. The following figure describes in detail the architecture of this neural network. "ID BLOCK" in the diagram stands for "Identity block," and "ID BLOCK x3" means you should stack 3 identity blocks together.

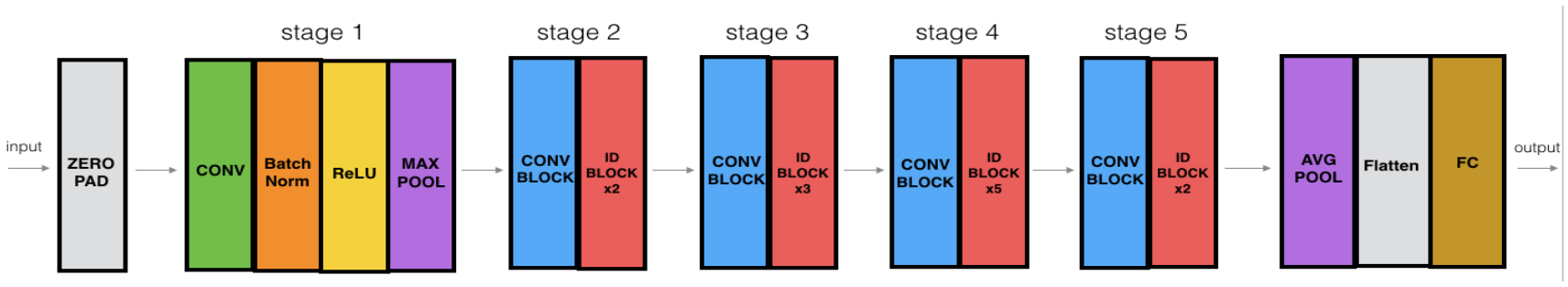


Figure 5 : **ResNet-50 model**

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 (https://keras.io/layers/pooling/#averagepooling2d)

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

- Conv2D: See reference (https://keras.io/layers/convolutional/#conv2d)
- BatchNorm: See reference (https://keras.io/layers/normalization/#batchnormalization) (axis: Integer, the axis that should be normalized (typically the features axis))
- Zero padding: See reference (https://keras.io/layers/convolutional/#zeropadding2d)
- Max pooling: See reference (https://keras.io/layers/pooling/#maxpooling2d)
- Fully conected layer: See reference (https://keras.io/layers/core/#dense)
- Addition: See reference (https://keras.io/layers/merge/#add)

In [15]:
```
# GRADED FUNCTION: ResNet50

def ResNet50(input_shape = (64, 64, 3), classes = 6):
    """
    Implementation of the popular ResNet50 the following architecture:
    CONV2D -> BATCHNORM -> RELU -> MAXPOOL -> CONVBLOCK -> IDBLOCK*2 -> CONVBLOCK -> IDBLOCK*3
    -> CONVBLOCK -> IDBLOCK*5 -> CONVBLOCK -> IDBLOCK*2 -> AVGPOOL -> TOPLAYER

    Arguments:
    input_shape -- shape of the images of the dataset
    classes -- integer, number of classes

    Returns:
    model -- a Model() instance in Keras
    """

    # Define the input as a tensor with shape input_shape
    X_input = Input(input_shape)


    # Zero-Padding
    X = ZeroPadding2D((3, 3))(X_input)

    # 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)

    # 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')

    ### 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), strides=(2,2))(X)

    ### END CODE HERE ###

    # output layer
    X = Flatten()(X)
    X = Dense(classes, activation='softmax', name='fc' + str(classes), kernel_initializer = glorot_uniform(seed=0))(X)


    # Create model
    model = Model(inputs = X_input, outputs = X, name='ResNet50')

    return model
```

Run the following code to build the model's graph. If your implementation is not correct you will know it by checking your accuracy when running `model.fit(...)` below.

In [16]:
```
model = ResNet50(input_shape = (64, 64, 3), classes = 6)
```

As seen in the Keras Tutorial Notebook, prior training a model, you need to configure the learning process by compiling the model.

In [17]:
```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

The model is now ready to be trained. The only thing you need is a dataset.

Let's load the SIGNS Dataset.

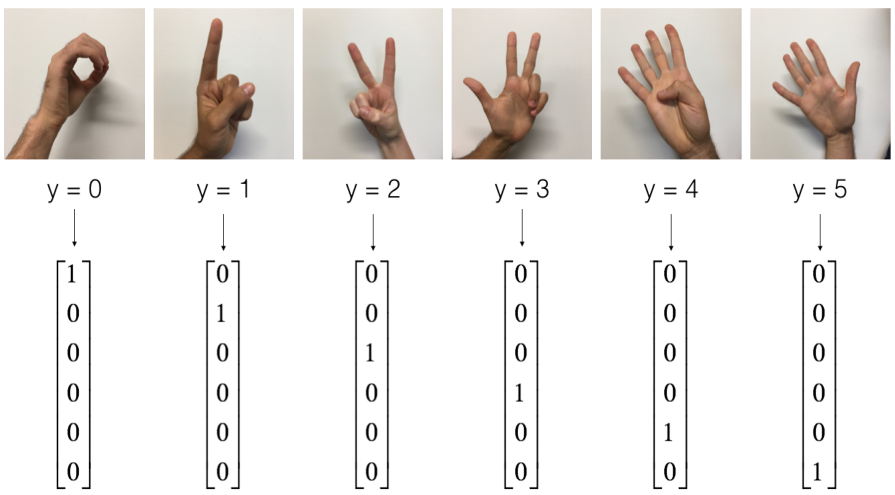


Figure 6 : **SIGNS dataset**

In [18]:
```
X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes = load_dataset()

# Normalize image vectors
X_train = X_train_orig/255.
X_test = X_test_orig/255.

# Convert training and test labels to one hot matrices
Y_train = convert_to_one_hot(Y_train_orig, 6).T
Y_test = convert_to_one_hot(Y_test_orig, 6).T

print ("number of training examples = " + str(X_train.shape[0]))
print ("number of test examples = " + str(X_test.shape[0]))
print ("X_train shape: " + str(X_train.shape))
print ("Y_train shape: " + str(Y_train.shape))
print ("X_test shape: " + str(X_test.shape))
print ("Y_test shape: " + str(Y_test.shape))
```

```
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)
```

Run the following cell to train your model on 2 epochs with a batch size of 32. On a CPU it should take you around 5min per epoch.

In [19]:
```
model.fit(X_train, Y_train, epochs = 2, batch_size = 32)#因为是用的CPU，epochs设不了太大，效果肯定是不行的。
```

```
Epoch 1/2
1080/1080 [==============================] - 361s 334ms/step - loss: 2.9576 - acc: 0.2324
Epoch 2/2
1080/1080 [==============================] - 324s 300ms/step - loss: 2.3586 - acc: 0.3657
```

Out[19]:
```
<keras.callbacks.History at 0x1cae8ac9cc0>
```

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.

Let's see how this model (trained on only two epochs) performs on the test set.

In [20]:
```
preds = model.evaluate(X_test, Y_test)
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```

```
120/120 [==============================] - 14s 113ms/step
Loss = 13.4317459742
Test Accuracy = 0.166666667163
```

Expected Output:

Test Accuracy	between 0.16 and 0.25

For the purpose of this assignment, we've asked you to train the model only for two epochs. You can see that it achieves poor performances. Please go ahead and submit your assignment; to check correctness, the online grader will run your code only for a small number of epochs as well.

After you have finished this official (graded) part of this assignment, you can also optionally train the ResNet for more iterations, if you want. We get a lot better performance when we train for ~20 epochs, but this will take more than an hour when training on a CPU.

Using a GPU, we've trained our own ResNet50 model's weights on the SIGNS dataset. You can load and run our trained model on the test set in the cells below. It may take ≈1min to load the model.

In [24]:
```
#model = load_model('ResNet50.h5') #这里要读入模型，但我没有模型的文件，先用刚刚训练的吧，虽然效果不行
```

In [25]:
```
preds = model.evaluate(X_test, Y_test)
print ("Loss = " + str(preds[0]))
print ("Test Accuracy = " + str(preds[1]))
```

```
120/120 [==============================] - 11s 94ms/step
Loss = 13.4317459742
Test Accuracy = 0.166666667163
```

ResNet50 is a powerful model for image classification when it is trained for an adequate number of iterations. We hope you can use what you've learnt and apply it to your own classification problem to perform state-of-the-art accuracy.

Congratulations on finishing this assignment! You've now implemented a state-of-the-art image classification system!

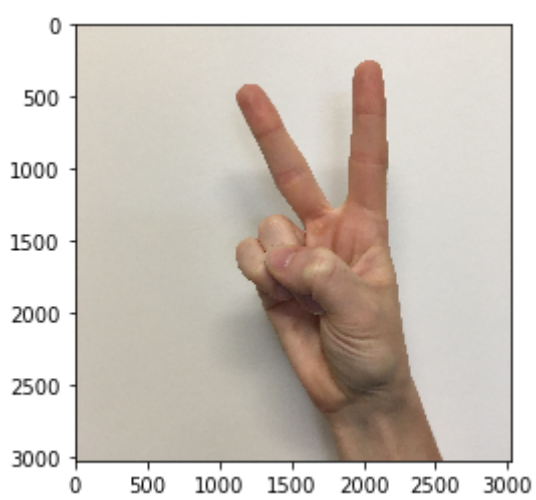
4 - Test on your own image (Optional/Ungraded)

If you wish, you can also take a picture of your own hand and see the output of the model. To do this:

1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub.
2. Add your image to this Jupyter Notebook's directory, in the "images" folder
3. Write your image's name in the following code
4. Run the code and check if the algorithm is right!

In [26]:
```
img_path = 'images/my_image.jpg'
img = image.load_img(img_path, target_size=(64, 64))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
print('Input image shape:', x.shape)
my_image = scipy.misc.imread(img_path)
imshow(my_image)
print("class prediction vector [p(0), p(1), p(2), p(3), p(4), p(5)] = ")
print(model.predict(x))#用的cpu训练的差的模型，预测不准，这里先这样吧
```

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



You can also print a summary of your model by running the following code.

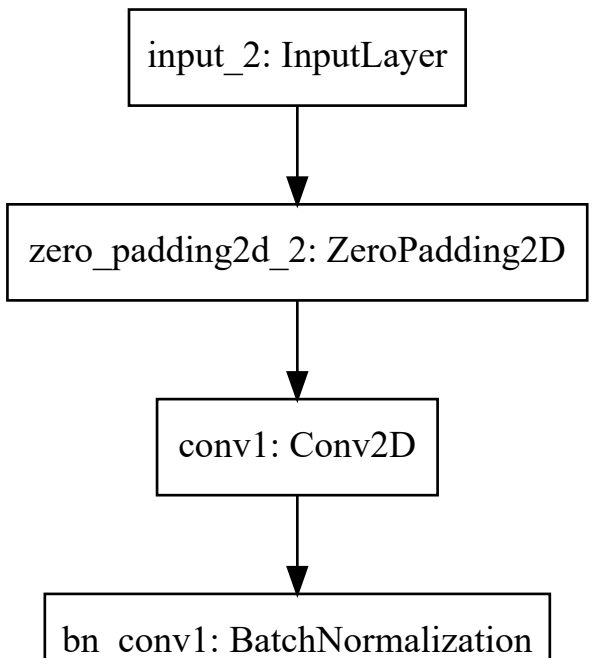
In [27]:
```
model.summary()
```

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	(None, 64, 64, 3)	0	
zero_padding2d_2 (ZeroPadding2D	(None, 70, 70, 3)	0	input_2[0][0]
conv1 (Conv2D)	(None, 32, 32, 64)	9472	zero_padding2d_2[0][0]
bn_conv1 (BatchNormalization)	(None, 32, 32, 64)	256	conv1[0][0]
activation_53 (Activation)	(None, 32, 32, 64)	0	bn_conv1[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 15, 15, 64)	0	activation_53[0][0]
res2a_branch2a (Conv2D)	(None, 15, 15, 64)	4160	max_pooling2d_2[0][0]
bn2a_branch2a (BatchNormalizati	(None, 15, 15, 64)	256	res2a_branch2a[0][0]
activation_54 (Activation)	(None, 15, 15, 64)	0	bn2a_branch2a[0][0]

Finally, run the code below to visualize your ResNet50. You can also download a .png picture of your model by going to "File -> Open...-> model.png".

In [28]:
```
plot_model(model, to_file='model.png')
SVG(model_to_dot(model).create(prog='dot', format='svg'))
```

Out[28]:



What you should remember:

- Very deep "plain" networks don't work in practice because they are hard to train due to vanishing gradients.
- The skip-connections help to address the Vanishing Gradient problem. They also make it easy for a ResNet block to learn an identity function.
- There are two main type of blocks: The identity block and the convolutional block.
- Very deep Residual Networks are built by stacking these blocks together.

References

This notebook presents the ResNet algorithm due to He et al. (2015). The implementation here also took significant inspiration and follows the structure given in the github repository of Francois Chollet:

- Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun - Deep Residual Learning for Image Recognition (2015) (https://arxiv.org/abs/1512.03385)
- Francois Chollet's github repository: https://github.com/fchollet/deep-learning-models/blob/master/resnet50.py (https://github.com/fchollet/deep-learning-models/blob/master/resnet50.py)