Residual Networks

Welcome to the first assignment of this week! You'll be building a very deep convolutional network, 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. allow also very deep networks to be trained.

For the assignment, you'll use Keras.

By the end of this assignment, you'll be able to:

- Implement the basic building blocks of ResNets in a deep neural network using Keras
- Put together these building blocks to implement and train a state-of-the-art neural network for image classification
- Pre-condition a dataset using Keras

For the assignment, you'll need Kera.

Before coming into the problem, run the cell below to load the required packages.

Table of Content

- 1 - Packages
- 2 - The Problem of Very Deep Neural Networks
- 3 - Building a Residual Network
  - 3.1 - The Identity Block
  - 3.2 - The Convolutional Block
  - 3.3 - The Convolution Block
  - 3.4 - The Convolutional Block
- 4 - Building Your First Residual Network (50 layers)
  - 4.1 - Load the "SIGNS" dataset
  - 4.2 - Run the cell of cell magic to load extensions
  - 4.3 - Overview of the model

1 - Packages

```
import numpy as np
import tensorflow as tf
import keras.backend as K
from keras.layers import Input, Add, Dense, Activation, ZeroPadding2D, BatchNormalization, Flatten, Conv2D, AveragePooling2D, Max
from keras.models import Model, load_model, model_from_json
from keras.preparedata import *
from keras.initializers import glorot_uniform, constant, identity
from keras.utils import layer_utils, to_categorical
from keras.utils.vis_utils import model_to_dot
from keras.utils import plot_model
%matplotlib inline
%load_ext autoreload
%autoreload 2
np.random.seed(1)
```

2 - The Problem of Very Deep Neural Networks

Last week, you built your first convolutional neural network architecture: first manually with numpy, then using TensorFlow and Keras.

In recent years, neural networks have become deeper, with state-of-the-art networks evolving from having 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). When it comes to the deep networks, however, there are also some difficulties to training them.

- Vanishing / exploding gradient, vanishing/exploding feature is a huge hurdle for training deep networks, to overcome this problem, a new architecture called a residual network is used.

- More specifically, having skipped shortcuts, as you'll acknowledge from the first hidden layer to the first hidden layer, you are multiplying by the weight matrix at every step, and then you apply the activation function, so if you have very deep networks, if you multiply by even slightly greater than one or even slightly less than one, you'll end up with either very large gradients or very small gradients or vanishing gradients.

- So by training, you might therefore also be magnitude-wise more of the gradient for the shallower layers because to very very rapidly or training gradients, you'll end up with

Figure 1. Vanishing gradient
The speed of learning decreases very rapidly for the deeper layers when you apply gradient descent to train the network.

Next to work? You are going to implement a skip connection in your network.

3 - Building a Residual Network

In the lecture, it was mentioned that a residual block allows the model to learn layers:

Figure 2. Residual block. 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 taking the Residual Block on the right, it allows you to express more complex functions.

The identity block and the convolutional block are the building blocks of residual networks. The identity block is the standard block used in ResNets, and it corresponds to an input activation (say $a^{[l]}$) is the same as the output activation (say $a^{[l+2]}$). To make this work, there are some details that you'll need to add in a residual block.

In this exercise, you'll implement both the identity block and the convolutional block.

You'll make ample use of this kind of building, depending mainly on whether the input/output dimensions are the same or different. You are going to implement both of them: the "identity block" and the "convolutional block".

3.1 - 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 to a ResNet's identity block, here is an alternative diagram showing the individual steps:

Figure 3. Identity block. Skip connection "skips over" 3 layers.

The upper path is the "main path." The lower path is the "shortcut." In this diagram, notice the CONV2D and BatchNorm layers in each layer. To speed up training, a 2D fully connected layer has been added to the shortcut path to ensure you can train at a reasonable speed. Finally, the relu activation function is added after adding the two lines together.

Here are the details of the steps in the identity block. In this example, we are using two layers.

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". Call this layer output for this component X. Then apply BatchNorm to X. Then apply the activation function relu.
- The second CONV2D has F_2 filters of shape (f,f) and a stride of (1,1). Its padding is "same". Call this layer output for this component X. Then apply BatchNorm to X. Then apply the activation function relu.
- The third CONV2D has F_3 filters of shape (1,1) and a stride of (1,1). Its padding is "valid". Call this layer output for this component X. Then apply BatchNorm to X. Then apply the activation function relu.

Shortcut component of main path:

- The shortcut path has no any layer. Its input X is added to the output of the main path. Then apply the activation function relu.

Final component of main path:

- The relu activation function is applied to the output of the shortcut path.

Finally:

- The two paths are added, and the output from the first layer $a^{[l]}$ is added together.
- Then the output of the two paths is fed through the activation function relu.

Exercise 1 - identity_block

Implement the ResNet identity block. The first component of the main path has been implemented for you already: first, you should finish the missing details in the second component of the main path.

- A keyword for the third component is mainpath3.
- To implement Conv2D(filters = F1, kernel_size = 1, strides = (1,1), padding = 'valid', name = 'conv', kernel_initializer = glorot_uniform(seed=0)) use the syntax: X = Conv2D(filters = F1, kernel_size = 1, strides = (1,1), padding = 'valid', name = 'conv', kernel_initializer = glorot_uniform(seed=0))(X)
- For the activation, use the syntax: X = Activation('relu')(X)
- To add the value passed forward by the shortcut: Add() use the syntax: X = Add([X_shortcut, X])

We've provided the function load_dataset() to load the data. Let's see what it looks like:

```
# 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 tuple 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, strides = (1,1), padding = 'valid', name = 'conv', kernel_initializer = glorot_uniform
    X = BatchNormalization(axis = 3, name = bn_name_base + '2a')(X)
    X = Activation('relu')(X)

    # Second component of main path
    X = Conv2D(filters = F2, kernel_size = (f,f), strides = (1,1), padding = 'same', name = 'conv', kernel_initializer = glorot_uni
    X = BatchNormalization(axis = 3, name = bn_name_base + '2b')(X)
    X = Activation('relu')(X)

    # Third component of main path
    X = Conv2D(filters = F3, kernel_size = 1, strides = (1,1), padding = 'valid', name = 'conv', kernel_initializer = glorot_uni
    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
    X = Add([X, X_shortcut])
    X = Activation('relu')(X)

    return X
```

```
tf.reset_default_graph()
with tf.Session() as sess:
    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")
    print("output = " + str(A.eval(feed_dict={A_prev: X})))
```

```
output = [0.94822985 0.         0.48162925 1.07744026 0.26285798 0.
  2.69565306]
```

3.2 - The Convolutional Block

The ResNet "convolutional block" is the second block type. 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 you'll add a CONV2D layer in the shortcut path. The details are:

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 2x, you can use a CONV2D with "stride" of (2,2). 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.

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". Call this layer output for this component X. Then apply BatchNorm to X. Then apply the activation function relu.
- The second CONV2D has F_2 filters of shape (f,f) and a stride of (1,1). Its padding is "same". Call this layer output for this component X. Then apply BatchNorm to X. Then apply the activation function relu.
- The third CONV2D has F_3 filters of shape (1,1) and a stride of (1,1). Its padding is "valid". Call this layer output for this component X. Then apply BatchNorm to X. Then apply the activation function relu.

Shortcut component of main path:

- The CONV2D layer in the shortcut path has F_3 filters of shape (1,1) and a stride of (s,s). Its padding is "valid". Call this layer output for this component X. Then apply BatchNorm to X. Then apply the activation function relu.

Final component of main path:

- The relu activation function is applied to the output of the shortcut path.

Finally:

- 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. Finally, the shortcut path is added to the main path, and the combined result is fed through the activation function relu.

Remember that these functions accept a shape attribute seed that can be any value you want, but that in this notebook must set to 0 for **grading purposes**.

Hence, when you are actually using these functions, you'll use them like this:

Exercise 2 - convolutional_block

Implement the convolutional block. The first component of the main path is already implemented; then it's your turn to implement the rest! As before, always use 0 as the seed for the function glorot_uniform.

- A keyword for the third component is mainpath3.
- To implement Conv2D(filters = F1, kernel_size = 1, strides = (s,s), padding = 'valid', name = 'conv', kernel_initializer = glorot_uniform(seed=0)) use the syntax: X = Conv2D(filters = F1, kernel_size = 1, strides = (s,s), padding = 'valid', name = 'conv', kernel_initializer = glorot_uniform(seed=0))(X)
- For the activation, use the syntax: X = Activation('relu')(X)
- To add the value passed forward by the shortcut: Add() use the syntax: X = Add([X_shortcut, X])

```
# 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 tuple 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


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

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

    # Shortcut path
    X_shortcut = Conv2D(filters = F3, kernel_size = 1, strides = (s,s), padding='valid', name = conv_name_base + '1', kernel_initia
    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
    X = Add([X, X_shortcut])
    X = Activation('relu')(X)

    return X
```

```
tf.reset_default_graph()
with tf.Session() as sess:
    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")
    print("output = " + str(A.eval(feed_dict={A_prev: X})))
```

```
output = [0.09018463 1.048872   0.         1.1741092  0.35004437 0.
  1.2744384 ]
```

4 - Building Your First ResNet Model (50 layers)

You now have all 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).
  - BatchNorm is applied to the channel 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 and "s" is 1.
  - The identity block uses three set of filters of size [64,64,256], "f" is 3 and "s" is 1.
- Stage 3:
  - The convolutional block uses three set of filters of size [128,128,512], "f" is 3 and "s" is 2.
  - The identity block uses three set of filters of size [128,128,512], "f" is 3 and "s" is 1.
- Stage 4:
  - The convolutional block uses three set of filters of size [256, 256, 1024], "f" is 3 and "s" is 2.
  - The identity block uses three set of filters of size [256, 256, 1024], "f" is 3 and "s" is 1.
- Stage 5:
  - The convolutional block uses three set of filters of size [512, 512, 2048], "f" is 3 and "s" is 2.
  - The identity block uses three set of filters of size [512, 512, 2048], "f" is 3 and "s" is 1.
- The "AveragePooling" step gives a shape of (1,1,2048)
- FLATTEN flattens its input. No number to be specified.
- The "Dense" layer applies a fully connected layer to the last dimension of its input. Dense(classes) where classes is the number of classes.

Suppose that this model is used for multi-class classification. Stage 5's output is then fed to a dense layer with softmax activation, which has 2048 hidden units and 6 output units, since there are 6 classes. Stage 5's output is also fed to dense layer to train with your SIGNS dataset.

Stage 5's output is also fed to dense layer to train with your SIGNS dataset

Then the rest is just the function:

- Average pooling two-dimensional
- Flatten

Here are some other functions we used in the code below:

- Conv2D: Keras convolution layer.
- BatchNormalization: Keras batch normalization layer.
- ZeroPadding: Keras layer for adding zeros at the boundaries of a tensor.
- MaxPooling2D: Max pooling operation for 2D spatial data.
- AveragePooling2D: Average pooling operation for spatial data.
- Flatten: Flattens the input, does not affect the batch size.
- Dense: Fully connected layer.
- Addition: Keras layer for adding two inputs.
- Activation: Applies an activation function.

```
# GRADED FUNCTION: ResNet50

def ResNet50(input_shape = (64, 64, 3), classes = 6):
    """
    Implementation of the popular ResNet50 :
    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')

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

    # AVGPOOL
    X = AveragePooling2D(pooling_size=(2,2), padding='same')(X)

    # 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
```

The following lines allow you to load the trained ResNet50 model that we built in the previous cell.

```
model = ResNet50(input_shape = (64, 64, 3), classes = 6)
model.compile("adam", "categorical_crossentropy", metrics=["accuracy"])
```

```
Using TensorFlow backend.
```

```
model.summary()
```

```
__Layer (type)__                    __Output Shape__         __Param #__   __Connected to__
input_1 (InputLayer)                (None, 64, 64, 3)        0             
__zero_padding2d_1 (ZeroPad__       (None, 70, 70, 3)        0             input_1[0][0]
conv1 (Conv2D)                      (None, 32, 32, 64)       9472          zero_padding2d_1[0][0]
bn_conv1 (BatchNormalization)       (None, 32, 32, 64)       256           conv1[0][0]
activation_1 (Activation)           (None, 32, 32, 64)       0             bn_conv1[0][0]
max_pooling2d_1 (MaxPooling2D)      (None, 15, 15, 64)       0             activation_1[0][0]
__convolutional_block_1 (Co__       (None, 8, 8, 256)        37888         max_pooling2d_1[0][0]
__bn2a_branch1 (BatchNormali__      (None, 8, 8, 256)        1024          convolutional_block_1[0][0]
__bn2a_branch2a (BatchNormali__     (None, 8, 8, 64)         256           convolutional_block_1[0][0]
activation_2 (Activation)           (None, 8, 8, 64)         0             bn2a_branch2a[0][0]
__bn2a_branch2b (BatchNormali__     (None, 8, 8, 64)         256           activation_2[0][0]
activation_3 (Activation)           (None, 8, 8, 64)         0             bn2a_branch2b[0][0]
__bn2a_branch2c (BatchNormali__     (None, 8, 8, 256)        1024          activation_3[0][0]
__bn2a_branch1[0][0]__              bn2a_branch2c[0][0]      
activation_4 (Activation)           (None, 8, 8, 256)        0             
__convolutional_block_2 (Co__       (None, 8, 8, 256)        65536         activation_4[0][0]
__bn2b_branch2a (BatchNormali__     (None, 8, 8, 64)         256           convolutional_block_2[0][0]
activation_5 (Activation)           (None, 8, 8, 64)         0             bn2b_branch2a[0][0]
__bn2b_branch2b (BatchNormali__     (None, 8, 8, 64)         256           activation_5[0][0]
activation_6 (Activation)           (None, 8, 8, 64)         0             bn2b_branch2b[0][0]
__bn2b_branch2c (BatchNormali__     (None, 8, 8, 256)        1024          activation_6[0][0]
__bn2b_branch2c[0][0]__             activation_4[0][0]       
activation_7 (Activation)           (None, 8, 8, 256)        0             
__convolutional_block_3 (Co__       (None, 8, 8, 256)        65536         activation_7[0][0]
__bn2c_branch2a (BatchNormali__     (None, 8, 8, 64)         256           convolutional_block_3[0][0]
activation_8 (Activation)           (None, 8, 8, 64)         0             bn2c_branch2a[0][0]
__bn2c_branch2b (BatchNormali__     (None, 8, 8, 64)         256           activation_8[0][0]
activation_9 (Activation)           (None, 8, 8, 64)         0             bn2c_branch2b[0][0]
__bn2c_branch2c (BatchNormali__     (None, 8, 8, 256)        1024          activation_9[0][0]
__bn2c_branch2c[0][0]__             activation_7[0][0]       
activation_10 (Activation)          (None, 8, 8, 256)        0             
__convolutional_block_4 (Co__       (None, 4, 4, 512)        131072        activation_10[0][0]
__bn3a_branch2a (BatchNormali__     (None, 4, 4, 128)        512           convolutional_block_4[0][0]
activation_11 (Activation)          (None, 4, 4, 128)        0             bn3a_branch2a[0][0]
__bn3a_branch2b (BatchNormali__     (None, 4, 4, 128)        512           activation_11[0][0]
activation_12 (Activation)          (None, 4, 4, 128)        0             bn3a_branch2b[0][0]
__bn3a_branch2c (BatchNormali__     (None, 4, 4, 512)        2048          activation_12[0][0]
__bn3a_branch1 (BatchNormali__      (None, 4, 4, 512)        2048          convolutional_block_4[0][0]
__bn3a_branch2c[0][0]__             bn3a_branch1[0][0]       
activation_13 (Activation)          (None, 4, 4, 512)        0             
__convolutional_block_5 (Co__       (None, 4, 4, 512)        262144        activation_13[0][0]
__bn3b_branch2a (BatchNormali__     (None, 4, 4, 128)        512           convolutional_block_5[0][0]
activation_14 (Activation)          (None, 4, 4, 128)        0             bn3b_branch2a[0][0]
__bn3b_branch2b (BatchNormali__     (None, 4, 4, 128)        512           activation_14[0][0]
activation_15 (Activation)          (None, 4, 4, 128)        0             bn3b_branch2b[0][0]
__bn3b_branch2c (BatchNormali__     (None, 4, 4, 512)        2048          activation_15[0][0]
__bn3b_branch2c[0][0]__             activation_13[0][0]      
activation_16 (Activation)          (None, 4, 4, 512)        0             
__convolutional_block_6 (Co__       (None, 4, 4, 512)        262144        activation_16[0][0]
__bn3c_branch2a (BatchNormali__     (None, 4, 4, 128)        512           convolutional_block_6[0][0]
activation_17 (Activation)          (None, 4, 4, 128)        0             bn3c_branch2a[0][0]
__bn3c_branch2b (BatchNormali__     (None, 4, 4, 128)        512           activation_17[0][0]
activation_18 (Activation)          (None, 4, 4, 128)        0             bn3c_branch2b[0][0]
__bn3c_branch2c (BatchNormali__     (None, 4, 4, 512)        2048          activation_18[0][0]
__bn3c_branch2c[0][0]__             activation_16[0][0]      
activation_19 (Activation)          (None, 4, 4, 512)        0             
__convolutional_block_7 (Co__       (None, 4, 4, 512)        262144        activation_19[0][0]
__bn3d_branch2a (BatchNormali__     (None, 4, 4, 128)        512           convolutional_block_7[0][0]
activation_20 (Activation)          (None, 4, 4, 128)        0             bn3d_branch2a[0][0]
__bn3d_branch2b (BatchNormali__     (None, 4, 4, 128)        512           activation_20[0][0]
activation_21 (Activation)          (None, 4, 4, 128)        0             bn3d_branch2b[0][0]
__bn3d_branch2c (BatchNormali__     (None, 4, 4, 512)        2048          activation_21[0][0]
__bn3d_branch2c[0][0]__             activation_19[0][0]      
activation_22 (Activation)          (None, 4, 4, 512)        0             
__convolutional_block_8 (Co__       (None, 2, 2, 1024)       524288        activation_22[0][0]
__bn4a_branch2a (BatchNormali__     (None, 2, 2, 256)        1024          convolutional_block_8[0][0]
activation_23 (Activation)          (None, 2, 2, 256)        0             bn4a_branch2a[0][0]
__bn4a_branch2b (BatchNormali__     (None, 2, 2, 256)        1024          activation_23[0][0]
activation_24 (Activation)          (None, 2, 2, 256)        0             bn4a_branch2b[0][0]
__bn4a_branch2c (BatchNormali__     (None, 2, 2, 1024)       4096          activation_24[0][0]
__bn4a_branch1 (BatchNormali__      (None, 2, 2, 1024)       4096          convolutional_block_8[0][0]
__bn4a_branch2c[0][0]__             bn4a_branch1[0][0]       
activation_25 (Activation)          (None, 2, 2, 1024)       0             
__convolutional_block_9 (Co__       (None, 2, 2, 1024)       1048576       activation_25[0][0]
__bn4b_branch2a (BatchNormali__     (None, 2, 2, 256)        1024          convolutional_block_9[0][0]
activation_26 (Activation)          (None, 2, 2, 256)        0             bn4b_branch2a[0][0]
__bn4b_branch2b (BatchNormali__     (None, 2, 2, 256)        1024          activation_26[0][0]
activation_27 (Activation)          (None, 2, 2, 256)        0             bn4b_branch2b[0][0]
__bn4b_branch2c (BatchNormali__     (None, 2, 2, 1024)       4096          activation_27[0][0]
__bn4b_branch2c[0][0]__             activation_25[0][0]      
activation_28 (Activation)          (None, 2, 2, 1024)       0             
__convolutional_block_10 (Co__      (None, 2, 2, 1024)       1048576       activation_28[0][0]
__bn4c_branch2a (BatchNormali__     (None, 2, 2, 256)        1024          convolutional_block_10[0][0]
activation_29 (Activation)          (None, 2, 2, 256)        0             bn4c_branch2a[0][0]
__bn4c_branch2b (BatchNormali__     (None, 2, 2, 256)        1024          activation_29[0][0]
activation_30 (Activation)          (None, 2, 2, 256)        0             bn4c_branch2b[0][0]
__bn4c_branch2c (BatchNormali__     (None, 2, 2, 1024)       4096          activation_30[0][0]
__bn4c_branch2c[0][0]__             activation_28[0][0]      
activation_31 (Activation)          (None, 2, 2, 1024)       0             
__convolutional_block_11 (Co__      (None, 2, 2, 1024)       1048576       activation_31[0][0]
__bn4d_branch2a (BatchNormali__     (None, 2, 2, 256)        1024          convolutional_block_11[0][0]
activation_32 (Activation)          (None, 2, 2, 256)        0             bn4d_branch2a[0][0]
__bn4d_branch2b (BatchNormali__     (None, 2, 2, 256)        1024          activation_32[0][0]
activation_33 (Activation)          (None, 2, 2, 256)        0             bn4d_branch2b[0][0]
__bn4d_branch2c (BatchNormali__     (None, 2, 2, 1024)       4096          activation_33[0][0]
__bn4d_branch2c[0][0]__             activation_31[0][0]      
activation_34 (Activation)          (None, 2, 2, 1024)       0             
__convolutional_block_12 (Co__      (None, 2, 2, 1024)       1048576       activation_34[0][0]
__bn4e_branch2a (BatchNormali__     (None, 2, 2, 256)        1024          convolutional_block_12[0][0]
activation_35 (Activation)          (None, 2, 2, 256)        0             bn4e_branch2a[0][0]
__bn4e_branch2b (BatchNormali__     (None, 2, 2, 256)        1024          activation_35[0][0]
activation_36 (Activation)          (None, 2, 2, 256)        0             bn4e_branch2b[0][0]
__bn4e_branch2c (BatchNormali__     (None, 2, 2, 1024)       4096          activation_36[0][0]
__bn4e_branch2c[0][0]__             activation_34[0][0]      
activation_37 (Activation)          (None, 2, 2, 1024)       0             
__convolutional_block_13 (Co__      (None, 2, 2, 1024)       1048576       activation_37[0][0]
__bn4f_branch2a (BatchNormali__     (None, 2, 2, 256)        1024          convolutional_block_13[0][0]
activation_38 (Activation)          (None, 2, 2, 256)        0             bn4f_branch2a[0][0]
__bn4f_branch2b (BatchNormali__     (None, 2, 2, 256)        1024          activation_38[0][0]
activation_39 (Activation)          (None, 2, 2, 256)        0             bn4f_branch2b[0][0]
__bn4f_branch2c (BatchNormali__     (None, 2, 2, 1024)       4096          activation_39[0][0]
__bn4f_branch2c[0][0]__             activation_37[0][0]      
activation_40 (Activation)          (None, 2, 2, 1024)       0             
__convolutional_block_14 (Co__      (None, 1, 1, 2048)       2097152       activation_40[0][0]
__bn5a_branch2a (BatchNormali__     (None, 1, 1, 512)        2048          convolutional_block_14[0][0]
activation_41 (Activation)          (None, 1, 1, 512)        0             bn5a_branch2a[0][0]
__bn5a_branch2b (BatchNormali__     (None, 1, 1, 512)        2048          activation_41[0][0]
activation_42 (Activation)          (None, 1, 1, 512)        0             bn5a_branch2b[0][0]
__bn5a_branch2c (BatchNormali__     (None, 1, 1, 2048)       8192          activation_42[0][0]
__bn5a_branch1 (BatchNormali__      (None, 1, 1, 2048)       8192          convolutional_block_14[0][0]
__bn5a_branch2c[0][0]__             bn5a_branch1[0][0]       
activation_43 (Activation)          (None, 1, 1, 2048)       0             
__convolutional_block_15 (Co__      (None, 1, 1, 2048)       4194304       activation_43[0][0]
__bn5b_branch2a (BatchNormali__     (None, 1, 1, 512)        2048          convolutional_block_15[0][0]
activation_44 (Activation)          (None, 1, 1, 512)        0             bn5b_branch2a[0][0]
__bn5b_branch2b (BatchNormali__     (None, 1, 1, 512)        2048          activation_44[0][0]
activation_45 (Activation)          (None, 1, 1, 512)        0             bn5b_branch2b[0][0]
__bn5b_branch2c (BatchNormali__     (None, 1, 1, 2048)       8192          activation_45[0][0]
__bn5b_branch2c[0][0]__             activation_43[0][0]      
activation_46 (Activation)          (None, 1, 1, 2048)       0             
__convolutional_block_16 (Co__      (None, 1, 1, 2048)       4194304       activation_46[0][0]
__bn5c_branch2a (BatchNormali__     (None, 1, 1, 512)        2048          convolutional_block_16[0][0]
activation_47 (Activation)          (None, 1, 1, 512)        0             bn5c_branch2a[0][0]
__bn5c_branch2b (BatchNormali__     (None, 1, 1, 512)        2048          activation_47[0][0]
activation_48 (Activation)          (None, 1, 1, 512)        0             bn5c_branch2b[0][0]
__bn5c_branch2c (BatchNormali__     (None, 1, 1, 2048)       8192          activation_48[0][0]
__bn5c_branch2c[0][0]__             activation_46[0][0]      
activation_49 (Activation)          (None, 1, 1, 2048)       0             
__average_pooling2d_1 (Av__         (None, 1, 1, 2048)       0             activation_49[0][0]
flatten_1 (Flatten)                 (None, 2048)             0             
__fc6 (Dense)__                     (None, 6)                12294         flatten_1[0][0]
```

```
Total params: 23,587,712
Trainable params: 23,534,592
Non-trainable params: 53,120
```

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

Let's see how this model (trained on 1 epoch) performs.

```
model.fit(X_train_orig, Y_train_orig, epochs = 1, batch_size = 32)
```

```
Epoch 1/1
1080/1080 [==============================] - 3,1,1,1
```

```
BATCH_SUBMISSION_ID_INCORRECT (Made: 0, Fix: 1000)  ADD_MISSING_ATTRIBUTE(S)
BATCH_SUBMISSION_ID_INCORRECT (Made: 0, Fix: 1000)  ADD_MISSING_ATTRIBUTE(S)
ADD_AT (100)                                        ADD_MISSING_ATTRIBUTE(S)
```