```python
### v2.1

# Importing the necessary packages

import tensorflow as tf
import numpy as np
import scipy.misc
from tensorflow.keras.applications.resnet_v2 import ResNet50V2
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet_v2 import preprocess_input,
decode_predictions
from tensorflow.keras import layers
from tensorflow.keras.layers import Input, Add, Dense, Activation,
ZeroPadding2D, Flatten, Conv2D, AveragePooling2D, MaxPooling2D,
GlobalMaxPooling2D
from tensorflow.keras.models import Model, load_model
from resnets_utils import *
from tensorflow.keras.initializers import random_uniform,
glorot_uniform, constant, identity
from tensorflow.python.framework.ops import EagerTensor
from matplotlib.pyplot import imshow


from test_utils import summary, comparator
import public_tests

%matplotlib inline
np.random.seed(1)
tf.random.set_seed(2)

# The problem of deep neural networks is vanishing/exploding
gradients.

# Note
# In the case of deep neural networks, there is the problem of
vanishing gradients.
# The gradients of the initial layer weights are very small as
compared to the gradients of the weights of layers at the back.
# So Adam's optimization considers adaptive learning rate to be large
for the initial layers and small for the layers at
# the back. Still the deep neural networks suffer from vanishing
gradients.

# In the rare case of exploding gradients, the optimizing algorithm
may diverge.

# Due to vanishing gradients, the learning speed of weights of initial
layers is small when compared to the learning speed
# of weights of layers at the back.
```

```
# Resnets make it easier for the model to learn the identity function.
The gradient of the identity function is 1.
# So this reduces the chances of vanishing/exploding gradients which
happens as we backpropagate (multiplication of matrices)
# from the back to the initial layers.

# Two main types of blocks are used in a ResNet, depending mainly on
whether the input/output dimensions are the same or
# different. We are going to implement both of them: the "identity
block" and the "convolutional block."

# Here is where you're actually using the power of the Functional API
to create a shortcut path
```

Identity block

```
def identity_block(X, f, filters, initializer=random_uniform):
    """
    Implementation of the identity 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
    initializer -- to set up the initial weights of a layer. Equals to
random uniform initializer

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

    # Retrieve Filters
    F1, F2, F3 = filters

    # Save the input value
    X_shortcut = X
    # 'X_shortcut' and 'X' are of shape nh x nw x nc
    # Note - Tensors are immutable unlike python lists and arrays

    # First component of main path
    X = Conv2D(filters = F1, kernel_size = 1, strides = (1,1), padding
= 'valid', kernel_initializer = initializer(seed=0))(X)
    # 'X' is of shape nh x nw x F1
    # kernel_initializer refers to the method used for the initial
weight initialization of the kernel (or weights) in a
    # layer of a neural network.
    # 'valid' padding means no padding
```

```
    X = BatchNormalization(axis = 3)(X) # Default axis
    X = Activation('relu')(X)

    ## Second component of main path
    ## Set the padding = 'same'
    X = Conv2D(filters = F2, kernel_size = f, strides = (1,1), padding
= 'same', kernel_initializer = initializer(seed=0))(X)
    # 'X' is of shape nh x nw x F2
    X = BatchNormalization(axis = 3)(X)
    X = Activation('relu')(X)

    ## Third component of main path
    ## Set the padding = 'valid'
    X = Conv2D(filters = F3, kernel_size = 1, strides = (1,1), padding
= 'valid', kernel_initializer = initializer(seed=0))(X)
    # 'X' is of shape nh x nw x F3
    X = BatchNormalization(axis = 3)(X)

    ## Final step: Add shortcut value to main path, and pass it
through a RELU activation
    X = Add()([X,X_shortcut])
    # Since this is an identity block, F3 = nc because X and
X_shortcut have to be the same size
    X = Activation('relu')(X)

    return X

### you cannot edit this cell

tf.keras.backend.set_learning_phase(False)

np.random.seed(1)
tf.random.set_seed(2)
X1 = np.ones((1, 4, 4, 3)) * -1
X2 = np.ones((1, 4, 4, 3)) * 1
X3 = np.ones((1, 4, 4, 3)) * 3

X = np.concatenate((X1, X2, X3), axis = 0).astype(np.float32)

A3 = identity_block(X, f=2, filters=[4, 4, 3],
                   initializer=lambda seed=0:constant(value=1))
print('\033[1mWith training=False\033[0m\n')
A3np = A3.numpy()
print(np.around(A3.numpy()[:,(0,-1),:,:].mean(axis = 3), 5))
resume = A3np[:,(0,-1),:,:].mean(axis = 3)
print(resume[1, 1, 0])

tf.keras.backend.set_learning_phase(True)

print('\n\033[1mWith training=True\033[0m\n')
```

```
np.random.seed(1)
tf.random.set_seed(2)
A4 = identity_block(X, f=2, filters=[3, 3, 3],
                   initializer=lambda seed=0:constant(value=1))
print(np.around(A4.numpy()[:,(0,-1),:,:].mean(axis = 3), 5))

public_tests.identity_block_test(identity_block)
```

```
With training=False

[[[  0.        0.        0.        0.     ]
  [  0.        0.        0.        0.     ]]

 [[192.99992 192.99992 192.99992  96.99996]
  [ 96.99996  96.99996  96.99996  48.99998]]

 [[578.99976 578.99976 578.99976 290.99988]
  [290.99988 290.99988 290.99988 146.99994]]]
96.99996

With training=True

[[[0.      0.      0.      0.     ]
  [0.      0.      0.      0.     ]]

 [[0.40732 0.40732 0.40732 0.40732]
  [0.40732 0.40732 0.40732 0.40732]]

 [[5.00011 5.00011 5.00011 3.25955]
  [3.25955 3.25955 3.25955 2.40732]]]
All tests passed!
```

```
# Note - 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.
```

Convolutional block

```
def convolutional_block(X, f, filters, s = 2, initializer=glorot_uniform):
    """
    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
    s -- Integer, specifying the stride to be used
    initializer -- to set up the initial weights of a layer. Equals to
Glorot uniform initializer,
                   also called Xavier uniform initializer.

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

    # Retrieve Filters
    F1, F2, F3 = filters

    # Save the input value
    X_shortcut = X


    ##### MAIN PATH #####

    # First component of main path
    X = Conv2D(filters = F1, kernel_size = 1, strides = (s, s),
padding='valid', kernel_initializer = initializer(seed=0))(X)
    X = BatchNormalization(axis = 3)(X)
    X = Activation('relu')(X)
    # X is of shape ((nh-1)/2) + 1, ((nw-1)/2) + 1, F1

    ## Second component of main path
    X = Conv2D(filters = F2, kernel_size = f, strides = 1,
padding='same', kernel_initializer = initializer(seed=0))(X)
    X = BatchNormalization(axis = 3)(X)
    X = Activation('relu')(X)
    # X is of shape ((nh-1)/2) + 1, ((nw-1)/2) + 1, F2

    ## Third component of main path
    X = Conv2D(filters = F3, kernel_size = 1, strides = 1,
padding='valid', kernel_initializer = initializer(seed=0))(X)
    X = BatchNormalization(axis = 3)(X)
    # X is of shape ((nh-1)/2) + 1, ((nw-1)/2) + 1, F3

    ##### SHORTCUT PATH #####
    X_shortcut = Conv2D(filters = F3, kernel_size = 1, strides = (s,
s), padding='valid', kernel_initializer = initializer(seed=0))
(X_shortcut)
    X_shortcut = BatchNormalization(axis = 3)(X_shortcut)
    # X_shortcut is of shape ((nh-1)/2) + 1, ((nw-1)/2) + 1, F3

    # Add shortcut value to main path and pass it through a RELU
activation
    X = Add()([X, X_shortcut])
```

```
    X = Activation('relu')(X)

    return X

# Note

# In ResNet, there are 2 blocks - identity blocks and convolutional
blocks. I can understand how identity blocks result in the
# identity function. But in the case of convolutional blocks,
X_shortcut is also passed through 1 convolutional layer.
# But there is no activation function and hence there is no non-linear
transformation and this kind of represents the
# identity function. Because there is only linear transformation.

# This is not strictly identity function but it facilitates the
training of deeper networks by maintaining stronger
# gradient flows, even when the dimensions change.

### you cannot edit this cell

public_tests.convolutional_block_test(convolutional_block)

tf.Tensor(
[[[0.33485505 1.6415989  0.33789736 0.08511472 0.814965   0.        ]
  [0.17509979 1.5699672  0.2606045  0.         0.767209   0.        ]]

 [[0.         1.4983511  0.16896994 0.         0.61830646 0.        ]
  [0.         1.4502985  0.11632714 0.         0.58068544
0.        ]]], shape=(2, 2, 6), dtype=float32)
All tests passed!
```

Building of ResNet-50 model

```
# Details of ResNet-50 model

# 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 'channels' axis of the input.
# MaxPooling uses a (3,3) window and a (2,2) stride.

# Stage 2:
# The convolutional block uses three sets of filters of size
[64,64,256], "f" is 3, and "s" is 1.
# The 2 identity blocks use three sets of filters of size [64,64,256],
and "f" is 3.

# Stage 3:
# The convolutional block uses three sets of filters of size
[128,128,512], "f" is 3 and "s" is 2.
```

```
# The 3 identity blocks use three sets of filters of size
[128,128,512] and "f" is 3.

# Stage 4:
# The convolutional block uses three sets of filters of size [256,
256, 1024], "f" is 3 and "s" is 2.
# The 5 identity blocks use three sets of filters of size [256, 256,
1024] and "f" is 3.

# Stage 5:
# The convolutional block uses three sets of filters of size [512,
512, 2048], "f" is 3 and "s" is 2.
# The 2 identity blocks use three sets of filters of size [512, 512,
2048] and "f" is 3.
# The 2D Average Pooling uses a window of shape (2,2).
# The 'flatten' layer doesn't have any hyperparameters.
# The Fully Connected (Dense) layer reduces its input to the number of
classes using a softmax activation.

# Calculation of total number of layers in ResNet-50 model

# 12 Identity_blocks = 36 layers
# 4 convolutional_blocks = 12 layers
# 1 convolutional layer = 1 layer
# 1 fully connected layer = 1 layer

# Total = 50 layers

def ResNet50(input_shape = (64, 64, 3), classes = 6, training=False):
    """
    Stage-wise implementation of the architecture of the popular
ResNet50:
    CONV2D -> BATCHNORM -> RELU -> MAXPOOL -> CONVBLOCK -> IDBLOCK*2 -
> CONVBLOCK -> IDBLOCK*3
    -> CONVBLOCK -> IDBLOCK*5 -> CONVBLOCK -> IDBLOCK*2 -> AVGPOOL ->
FLATTEN -> DENSE

    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), kernel_initializer =
glorot_uniform(seed=0))(X)
    X = BatchNormalization(axis = 3)(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], s = 1)
    # After above step, suppose X is of shape nh,nw,256
    X = identity_block(X, 3, [64, 64, 256])
    # After above step, X is of shape nh,nw,256
    X = identity_block(X, 3, [64, 64, 256])
    # After above step, X is of shape nh,nw,256

    # Note - For the convolutional block, we can blindly take any
values of f,filters,s
    # But for identity_block, if input is of shape (nh,nw,nc), then
filters is [F1,F2,nc]. f can be any value

    # Use the instructions above in order to implement all of the
Stages below
    # Make sure you don't miss adding any required parameter

    ## Stage 3
    X = convolutional_block(X, f = 3, filters = [128, 128, 512], s =
2)
    # After above step, X is of shape ((nh-1)/2) + 1, ((nw-1)/2) + 1,
512 or (nh',nw',512)

    # the 3 `identity_block` with correct values of `f` and `filters`
for this stage
    X = identity_block(X, 3, [128, 128, 512])
    # After above step, X is of shape ((nh-1)/2) + 1, ((nw-1)/2) + 1,
512 or (nh',nw',512)
    X = identity_block(X, 3, [128, 128, 512])
    # After above step, X is of shape ((nh-1)/2) + 1, ((nw-1)/2) + 1,
512 or (nh',nw',512)
    X = identity_block(X, 3, [128, 128, 512])
    # After above step, X is of shape ((nh-1)/2) + 1, ((nw-1)/2) + 1,
512 or (nh',nw',512)

    # Stage 4
    # add `convolutional_block` with correct values of `f`, `filters`
and `s` for this stage
    X = convolutional_block(X, f = 3, filters = [256, 256, 1024], s =
2)
```

```
    # After above step, X is of shape ((nh'-1)/2) + 1, ((nw'-1)/2) +
1, 1024 or (nh'',nw'',1024)

    # the 5 `identity_block` with correct values of `f` and `filters`
for this stage
    X = identity_block(X, 3, [256, 256, 1024])
    # After above step, X is of shape ((nh'-1)/2) + 1, ((nw'-1)/2) +
1, 1024 or (nh'',nw'',1024)
    X = identity_block(X, 3, [256, 256, 1024])
    # After above step, X is of shape ((nh'-1)/2) + 1, ((nw'-1)/2) +
1, 1024 or (nh'',nw'',1024)
    X = identity_block(X, 3, [256, 256, 1024])
    # After above step, X is of shape ((nh'-1)/2) + 1, ((nw'-1)/2) +
1, 1024 or (nh'',nw'',1024)
    X = identity_block(X, 3, [256, 256, 1024])
    # After above step, X is of shape ((nh'-1)/2) + 1, ((nw'-1)/2) +
1, 1024 or (nh'',nw'',1024)
    X = identity_block(X, 3, [256, 256, 1024])
    # After above step, X is of shape ((nh'-1)/2) + 1, ((nw'-1)/2) +
1, 1024 or (nh'',nw'',1024)

    # Stage 5
    # add `convolutional_block` with correct values of `f`, `filters`
and `s` for this stage
    X = convolutional_block(X, f = 3, filters = [512, 512, 2048], s =
2)
    # After above step, X is of shape ((nh''-1)/2) + 1, ((nw''-1)/2) +
1, 2048 or (nh''',nw''',2048)

    # the 2 `identity_block` with correct values of `f` and `filters`
for this stage
    X = identity_block(X, 3, [512, 512, 2048])
    # After above step, X is of shape ((nh''-1)/2) + 1, ((nw''-1)/2) +
1, 2048 or (nh''',nw''',2048)
    X = identity_block(X, 3, [512, 512, 2048])
    # After above step, X is of shape ((nh''-1)/2) + 1, ((nw''-1)/2) +
1, 2048 or (nh''',nw''',2048)

    # AVGPOOL
    X = AveragePooling2D((2,2))(X)

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


    # Create model
    model = Model(inputs = X_input, outputs = X)
```

```
    return model

tf.keras.backend.set_learning_phase(True)

# The above line of code explicitly sets it to the learning/training
phase which is different from inferrence/prediction phase.

# Differences between training and prediction phase
# During training, dropout randomly sets input units to 0 at a rate of
rate at each step, which helps prevent overfitting, and
# batch normalization normalizes the input using the batch statistics.
# During inference, dropout is not applied, and batch normalization
uses the moving average and variance calculated during
# training.

# In TensorFlow 2.x and later,
# When calling model.fit(), TensorFlow automatically sets the learning
phase to training mode.
# When using model.evaluate() or model.predict(), TensorFlow sets the
learning phase to inference mode.

model = ResNet50(input_shape = (64, 64, 3), classes = 6)
print(model.summary())

### you cannot edit this cell

from outputs import ResNet50_summary

model = ResNet50(input_shape = (64, 64, 3), classes = 6)

comparator(summary(model), ResNet50_summary)

All tests passed!
```

Compiling the model

```
np.random.seed(1)
tf.random.set_seed(2)
opt = tf.keras.optimizers.Adam(learning_rate=0.00015)
model.compile(optimizer=opt, loss='categorical_crossentropy',
metrics=['accuracy'])
```

Loading the dataset

```
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)
```

Training the model

```
model.fit(X_train, Y_train, epochs = 10, batch_size = 32)
```

```
Epoch 1/10
34/34 [==============================] - 16s 63ms/step - loss: 1.8302
- accuracy: 0.3019
Epoch 2/10
34/34 [==============================] - 2s 51ms/step - loss: 1.2657 -
accuracy: 0.5259
Epoch 3/10
34/34 [==============================] - 2s 51ms/step - loss: 0.9542 -
accuracy: 0.6583
Epoch 4/10
34/34 [==============================] - 2s 51ms/step - loss: 0.6513 -
accuracy: 0.7648
Epoch 5/10
34/34 [==============================] - 2s 52ms/step - loss: 0.3943 -
accuracy: 0.8574
Epoch 6/10
34/34 [==============================] - 2s 51ms/step - loss: 0.2600 -
accuracy: 0.9009
Epoch 7/10
34/34 [==============================] - 2s 51ms/step - loss: 0.3027 -
accuracy: 0.8963
Epoch 8/10
34/34 [==============================] - 2s 51ms/step - loss: 0.2490 -
accuracy: 0.9065
Epoch 9/10
34/34 [==============================] - 2s 52ms/step - loss: 0.1940 -
accuracy: 0.9343
```

```
Epoch 10/10
34/34 [==============================] - 2s 50ms/step - loss: 0.1304 -
accuracy: 0.9528

<keras.callbacks.History at 0x7fe61c8be4f0>

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

4/4 [==============================] - 1s 11ms/step - loss: 0.6611 -
accuracy: 0.7917
Loss = 0.661126971244812
Test Accuracy = 0.7916666865348816
```

Using a pre_trained model

```
pre_trained_model = load_model('resnet50.h5')

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

4/4 [==============================] - 1s 9ms/step - loss: 0.1596 -
accuracy: 0.9500
Loss = 0.1595880389213562
Test Accuracy = 0.949999988079071

# Note

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