A convolution layer transforms an input volume into an output volume of different size.

- Convolution functions, including:
  - Zero Padding
  - Convolve window
  - Convolution forward
  - Convolution backward (optional)
- Pooling functions, including:
  - Pooling forward
  - Create mask
  - Distribute value
  - Pooling backward (optional)

1. Convolution

Zero Padding surrounds the image with zeros so the size is the same and border pixels aren't missed.

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

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

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

    #(≈ 1 line)
    # X_pad = None
    # YOUR CODE STARTS HERE
    X_pad = np.pad(X, ((0,0), (pad,pad), (pad,pad), (0,0)), mode='constant', constant_values=0)

    # YOUR CODE ENDS HERE

    return X_pad
```

Single Step of Convolution in which a filter is applied to a single position of the input basically squishes it together : In a computer vision application, each value in the matrix on the left corresponds to a single pixel value. You convolve a 3x3 filter with the image by multiplying its values element-wise with the original matrix, then summing them up and adding a bias.

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

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

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

    #(≈ 3 lines of code)
    # Element-wise product between a_slice_prev and W. Do not add the bias yet.
    s = np.multiply(a_slice_prev, W)
    # Sum over all entries of the volume s.
    Z = np.sum(s)
    # Add bias b to Z. Cast b to a float() so that Z results in a scalar value.
    Z = Z + float(b)

    return Z
```

Forward Pass you will take many filters and convolve them on the input. Each 'convolution' gives you a 2D matrix output. You will then stack these outputs to get a 3D volume.

```
def conv_forward(A_prev, W, b, hparameters):
    """
    Implements the forward propagation for a convolution function

    Arguments:
    A_prev -- output activations of the previous layer,
```

```
        numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    W -- Weights, numpy array of shape (f, f, n_C_prev, n_C)
    b -- Biases, numpy array of shape (1, 1, 1, n_C)
    hparameters -- python dictionary containing "stride" and "pad"

    Returns:
    Z -- conv output, numpy array of shape (m, n_H, n_W, n_C)
    cache -- cache of values needed for the conv_backward() function
    """

    # Retrieve dimensions from A_prev's shape (≈1 line)
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

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

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

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

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

    # Create A_prev_pad by padding A_prev
    A_prev_pad = np.pad(A_prev, ((0,0), (pad,pad), (pad,pad), (0,0)), mode="constant",
constant_values=0)

    for i in range(m):               # loop over the batch of training examples
        a_prev_pad = A_prev_pad[i]     # Select ith training example's padded activation
        for h in range(n_H):           # loop over vertical axis of the output volume
            # Find the vertical start and end of the current "slice" (≈2 lines)
            vert_start = h * stride
            vert_end = vert_start + f

            for w in range(n_W):       # loop over horizontal axis of the output volume
                # Find the horizontal start and end of the current "slice" (≈2 lines)
                horiz_start = w * stride
                horiz_end = horiz_start + f

                for c in range(n_C):   # loop over channels (= #filters) of the output volume
```

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

                # Convolve the (3D) slice with the correct filter W and bias b, to get back one output neuron. (≈3 line)
                weights = W[:, :, :, c]
                biases = b[:, :, :, c]
                Z[i, h, w, c] = np.sum(np.multiply(a_slice_prev, weights)) + float(biases)

    # Save information in "cache" for the backprop
    cache = (A_prev, W, b, hparameters)

    return Z, cache
```

2. Pooling

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

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

Pool Forward

```
def pool_forward(A_prev, hparameters, mode = "max"):
    """
    Implements the forward pass of the pooling layer

    Arguments:
    A_prev -- Input data, numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
    hparameters -- python dictionary containing "f" and "stride"
    mode -- the pooling mode you would like to use, defined as a string ("max" or "average")

    Returns:
    A -- output of the pool layer, a numpy array of shape (m, n_H, n_W, n_C)
```

```
    cache -- cache used in the backward pass of the pooling layer, contains the input and
hparameters
    """

    # Retrieve dimensions from the input shape
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Retrieve hyperparameters from "hparameters"
    f = hparameters["f"]
    stride = hparameters["stride"]

    # Define the dimensions of the output
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    # Initialize output matrix A
    A = np.zeros((m, n_H, n_W, n_C))

    for i in range(m):       # loop over the training examples
        for h in range(n_H):  # loop on the vertical axis of the output volume
            # Find the vertical start and end of the current "slice" (≈2 lines)
            vert_start = h * stride
            vert_end = vert_start + f

            for w in range(n_W):    # loop on the horizontal axis of the output volume
                # Find the vertical start and end of the current "slice" (≈2 lines)
                horiz_start = w * stride
                horiz_end = horiz_start + f

                for c in range (n_C):   # loop over the channels of the output volume

                    # Use the corners to define the current slice on the ith training example of A_prev,
channel c. (≈1 line)
                    a_prev_slice = A_prev[i, vert_start:vert_end, horiz_start:horiz_end, c]

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

```
    # Store the input and hparameters in "cache" for pool_backward()
    cache = (A_prev, hparameters)

    # Making sure your output shape is correct
    #assert(A.shape == (m, n_H, n_W, n_C))

    return A, cache
```

CONVOLUTIONAL MODEL APPLICATION

```
def happyModel():
    """
    Implements the forward propagation for the binary classification model:
    ZEROPAD2D -> CONV2D -> BATCHNORM -> RELU -> MAXPOOL -> FLATTEN -> DENSE

    Note that for simplicity and grading purposes, you'll hard-code all the values
    such as the stride and kernel (filter) sizes.
    Normally, functions should take these values as function parameters.

    Arguments:
    None

    Returns:
    model -- TF Keras model (object containing the information for the entire training process)
    """
    model = tf.keras.Sequential([
            ## ZeroPadding2D with padding 3, input shape of 64 x 64 x 3
            tfl.ZeroPadding2D(padding=3, input_shape=(64, 64, 3)),
            ## Conv2D with 32 7x7 filters and stride of 1
            tfl.Conv2D(filters=32, kernel_size=(7,7), strides=1, padding="valid"),
            ## BatchNormalization for axis 3
            tfl.BatchNormalization(axis=3),
            ## ReLU
            tfl.ReLU(),
            ## Max Pooling 2D with default parameters
            tfl.MaxPooling2D(pool_size=(2,2), strides=2),
            ## Flatten layer
            tfl.Flatten(),
            ## Dense layer with 1 unit for output & 'sigmoid' activation
            tfl.Dense(units=1, activation="sigmoid")
        ])

    return model
```

FORWARD PROPAGATION WITH TENSORFLOW KERAS

```
def convolutional_model(input_shape):
    """
    Implements the forward propagation for the model:
    CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN ->
DENSE

    Note that for simplicity and grading purposes, you'll hard-code some values
    such as the stride and kernel (filter) sizes.
    Normally, functions should take these values as function parameters.

    Arguments:
    input_img -- input dataset, of shape (input_shape)

    Returns:
    model -- TF Keras model (object containing the information for the entire training process)
    """

    input_img = tf.keras.Input(shape=input_shape)
    ## CONV2D: 8 filters 4x4, stride of 1, padding 'SAME'
    Z1 = tfl.Conv2D(filters=8, kernel_size=(4,4), strides=1, padding="same")(input_img)
    ## RELU
    A1 = tfl.ReLU()(Z1)
    ## MAXPOOL: window 8x8, stride 8, padding 'SAME'
    P1 = tfl.MaxPooling2D(pool_size=(8,8), strides=8, padding="same")(A1)
    ## CONV2D: 16 filters 2x2, stride 1, padding 'SAME'
    Z2 = tfl.Conv2D(filters=16, kernel_size=(2,2), strides=1, padding="same")(P1)
    ## RELU
    A2 = tfl.ReLU()(Z2)
    ## MAXPOOL: window 4x4, stride 4, padding 'SAME'
    P2 = tfl.MaxPooling2D(pool_size=(4,4), strides=4, padding="same")(A2)
    ## FLATTEN
    F = tfl.Flatten()(P2)
    ## Dense layer
    ## 6 neurons in output layer. Hint: one of the arguments should be "activation='softmax'"
    outputs = tfl.Dense(units=6, activation="softmax")(F)

    model = tf.keras.Model(inputs=input_img, outputs=outputs)
    return model
```

RESIDUAL NETWORKS

Deep Neural Networks

In recent years, neural networks have become much 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 shallower layers, closer to the input) to very complex features (at the deeper layers, closer to the output).
- 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 prohibitively slow.
- More specifically, during gradient descent, as you backpropagate 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," from gaining very large values).
- During training, you might therefore see the magnitude (or norm) of the gradient for the shallower layers decrease to zero very rapidly as training proceeds, as shown below:

Building a Residual Network

In ResNets, a "shortcut" or a "skip connection" allows the model to skip layers:

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.

The lecture mentioned 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.

On that note, there is also some evidence that the ease of learning an identity function accounts for ResNets' remarkable performance even more than skip connections help with vanishing gradients.

Two main types of blocks are used in a ResNet, 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."

1. **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]).

```
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. 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', kernel_initializer =
initializer(seed=0))(X)
    X = BatchNormalization(axis = 3)(X) # Default axis
    X = Activation('relu')(X)

    ### START CODE HERE
    ## Second component of main path (≈3 lines)
    ## Set the padding = 'same'
    X = Conv2D(filters=F2, kernel_size=(f,f), strides=(1,1), padding="same",
kernel_initializer=initializer(seed=0))(X)
    X = BatchNormalization(axis=3)(X)
    X = Activation("relu")(X)

    ## Third component of main path (≈2 lines)
    ## Set the padding = 'valid'
    X = Conv2D(filters=F3, kernel_size=(1,1), strides=(1,1), padding="valid",
kernel_initializer=initializer(seed=0))(X)
    X = BatchNormalization(axis=3)(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

2. **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 there is a CONV2D layer in the shortcut path.

```
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 glorot_uniform(seed=0)
    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)
```

```
    ### START CODE HERE

    ## Second component of main path (≈3 lines)
    X = Conv2D(filters=F2, kernel_size=(f,f), strides=(1,1), padding="same",
kernel_initializer=initializer(seed=0))(X)
    X = BatchNormalization(axis=3)(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",
kernel_initializer=initializer(seed=0))(X)
    X = BatchNormalization(axis=3)(X)

    ##### SHORTCUT PATH ##### (≈2 lines)
    X_shortcut = Conv2D(filters=F3, kernel_size=(1,1), strides=(s,s), padding="valid",
kernel_initializer=initializer(seed=0))(X_shortcut)
    X_shortcut = BatchNormalization(axis=3)(X_shortcut)

    ### END CODE HERE

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

    return X
```

RESNET MODEL

```
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)
X = identity_block(X, 3, [64, 64, 256])
X = identity_block(X, 3, [64, 64, 256])

### START CODE HERE

# 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 (≈4 lines)
# `convolutional_block` with correct values of `f`, `filters` and `s` for this stage
X = convolutional_block(X, f=3, filters=[128, 128, 512], s=2)

# the 3 `identity_block` with correct values of `f` and `filters` for this stage
X = identity_block(X, f=3, filters=[128, 128, 512])
X = identity_block(X, f=3, filters=[128, 128, 512])
X = identity_block(X, f=3, filters=[128, 128, 512])

# Stage 4 (≈6 lines)
# 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)

# the 5 `identity_block` with correct values of `f` and `filters` for this stage
X = identity_block(X, f=3, filters=[256, 256, 1024])
X = identity_block(X, f=3, filters=[256, 256, 1024])
X = identity_block(X, f=3, filters=[256, 256, 1024])
X = identity_block(X, f=3, filters=[256, 256, 1024])
```

```
X = identity_block(X, f=3, filters=[256, 256, 1024])

# Stage 5 (≈3 lines)
# 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)

# the 2 `identity_block` with correct values of `f` and `filters` for this stage
X = identity_block(X, f=3, filters=[512, 512, 2048])
X = identity_block(X, f=3, filters=[512, 512, 2048])

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

### END CODE HERE

# 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
```

Transfer Learning with MobileNetV2

Packages

In [ ]:

```
### v2.1
```

In [1]:

```
import matplotlib.pyplot as plt
import json
import numpy as np
import os
import tensorflow as tf
import tensorflow.keras.layers as tfl

from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras.layers.experimental.preprocessing import RandomFlip,
RandomRotation
```

Create the Dataset and Split it into Training and Validation Sets

When training and evaluating deep learning models in Keras, generating a dataset from image files stored on disk is simple and fast. Call `image_data_set_from_directory()` to read from the directory and create both training and validation datasets.

If you're specifying a validation split, you'll also need to specify the subset for each portion. Just set the training set to `subset='training'` and the validation set to `subset='validation'`.

You'll also set your seeds to match each other, so your training and validation sets don't overlap

Preprocess and Augment Training Data

You may have encountered `dataset.prefetch` in a previous TensorFlow assignment, as an important extra step in data preprocessing.

Using `prefetch()` prevents a memory bottleneck that can occur when reading from disk. It sets aside some data and keeps it ready for when it's needed, by creating a source dataset from your input data, applying a transformation to preprocess it, then iterating over the dataset one element at a time. Because the iteration is streaming, the data doesn't need to fit into memory.

You can set the number of elements to prefetch manually, or you can use `tf.data.experimental.AUTOTUNE` to choose the parameters automatically. Autotune prompts `tf.data` to tune that value dynamically at runtime, by tracking the time spent in each operation and feeding those times into an optimization algorithm. The optimization algorithm tries to find the best allocation of its CPU budget across all tunable operations.

To increase diversity in the training set and help your model learn the data better, it's standard practice to augment the images by transforming them, i.e., randomly flipping and rotating them. Keras' Sequential API offers a straightforward method for these kinds of data augmentations, with built-in, customizable preprocessing layers. These layers are saved with the rest of your model and can be re-used later. Ahh, so convenient!

```
def data_augmenter():
    '''
    Create a Sequential model composed of 2 layers
    Returns:
        tf.keras.Sequential
    '''
    data_augmentation = tf.keras.Sequential([
        RandomFlip("horizontal"),  # Randomly flips images horizontally
        RandomRotation(0.2)        # Randomly rotates images up to 20% of total range
    ])
```

return data_augmentation

Take a look at how an image from the training set has been augmented with simple transformations:

From one cute animal, to 9 variations of that cute animal, in three lines of code. Now your model has a lot more to learn from.

What you should remember:

- When calling image_data_set_from_directory(), specify the train/val subsets and match the seeds to prevent overlap
- Use prefetch() to prevent memory bottlenecks when reading from disk
- Give your model more to learn from with simple data augmentations like rotation and flipping.
- When using a pretrained model, it's best to reuse the weights it was trained on.

MobileNetV2 was trained on ImageNet and is optimized to run on mobile and other low-power applications. It's 155 layers deep (just in case you felt the urge to plot the model yourself, prepare for a long journey!) and very efficient for object detection and image segmentation tasks, as well as classification tasks like this one. The architecture has three defining characteristics:

- Depthwise separable convolutions
- Thin input and output bottlenecks between layers
- Shortcut connections between bottleneck layers

What you should remember:

- MobileNetV2's unique features are:
  - Depthwise separable convolutions that provide lightweight feature filtering and creation
  - Input and output bottlenecks that preserve important information on either end of the block
- Depthwise separable convolutions deal with both spatial and depth (number of channels) dimensions

alpaca_model

In [69]:

```
# UNQ_C2
# GRADED FUNCTION
def alpaca_model(image_shape=IMG_SIZE, data_augmentation=data_augmenter()):
    ''' Define a tf.keras model for binary classification out of the
MobileNetV2 model
    Arguments:
        image_shape -- Image width and height
        data_augmentation -- data augmentation function
    Returns:
    Returns:
        tf.keras.model
```

```
    '''

    input_shape = image_shape + (3,)

    ### START CODE HERE

base_model_path="imagenet_base_model/without_top_mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_160_no_top.h5"

    base_model = tf.keras.applications.MobileNetV2(input_shape=input_shape,
                                                   include_top=False, # <== Important!!!!
                                                   weights=base_model_path)

    # freeze the base model by making it non trainable
    base_model.trainable = False

    # create the input layer (Same as the imageNetv2 input size)
    inputs = tf.keras.Input(shape=input_shape)

    # apply data augmentation to the inputs
    x = data_augmentation(inputs)

    # data preprocessing using the same weights the model was trained on
    x = preprocess_input(x)

    # set training to False to avoid keeping track of statistics in the batch norm layer
    x = base_model(x, training=False)

    # add the new Binary classification layers
    # use global avg pooling to summarize the info in each channel
    x = GlobalAveragePooling2D()(x)
    # include dropout with probability of 0.2 to avoid overfitting
    x = Dropout(0.2)(x)

    # use a prediction layer with one neuron (as a binary classifier only needs one)
    outputs = Dense(1, activation='linear')(x)

    ### END CODE HERE

    model = tf.keras.Model(inputs, outputs)
```

```
    return model
```

The base learning rate has been set for you, so you can go ahead and compile the new model and run it for 5 epochs:

In [120]:

```
base_learning_rate = 0.001
model2.compile(optimizer=tf.keras.optimizers.Adam(lr=base_learning_rate),
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

In [121]:

```
initial_epochs = 5
history = model2.fit(train_dataset, validation_data=validation_dataset,
epochs=initial_epochs)
```

Fine-tuning the Model

You could try fine-tuning the model by re-running the optimizer in the last layers to improve accuracy. When you use a smaller learning rate, you take smaller steps to adapt it a little more closely to the new data. In transfer learning, the way you achieve this is by unfreezing the layers at the end of the network, and then re-training your model on the final layers with a very low learning rate. Adapting your learning rate to go over these layers in smaller steps can yield more fine details - and higher accuracy.

```
# UNQ_C3
base_model = model2.layers[4]
base_model.trainable = True
# Let's take a look to see how many layers are in the base model
print("Number of layers in the base model: ", len(base_model.layers))

# Fine-tune from this layer onwards
fine_tune_at = 120

### START CODE HERE

# Freeze all the layers before the `fine_tune_at` layer
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable = False

# Define a BinaryCrossentropy loss function. Use from_logits=True
loss_function = BinaryCrossentropy(from_logits=True)
# Define an Adam optimizer with a learning rate of 0.1 * base_learning_rate
optimizer = Adam(learning_rate=0.1 * base_learning_rate)
```

```
# Use accuracy as evaluation metric
metrics= ["accuracy"]

### END CODE HERE

model2.compile(loss=loss_function,
              optimizer = optimizer,
              metrics=metrics)
```

```
fine_tune_epochs = 5
total_epochs =  initial_epochs + fine_tune_epochs

history_fine = model2.fit(train_dataset,
                         epochs=total_epochs,
                         initial_epoch=history.epoch[-1],
                         validation_data=validation_dataset)
```

What you should remember:

- To adapt the classifier to new data: Delete the top layer, add a new classification layer, and train only on that layer
- When freezing layers, avoid keeping track of statistics (like in the batch normalization layer)
- Fine-tune the final layers of your model to capture high-level details near the end of the network and potentially improve accuracy

YOLO

"You Only Look Once" (YOLO) is a popular algorithm because it achieves high accuracy while also being able to run in real time. This algorithm "only looks once" at the image in the sense that it requires only one forward propagation pass through the network to make predictions. After non-max suppression, it then outputs recognized objects together with the bounding boxes.

2.1 - Model Details

Inputs and outputs

- The **input** is a batch of images, and each image has the shape (608, 608, 3)
- The **output** is a list of bounding boxes along with the recognized classes. Each bounding box is represented by 6 numbers $(p_c, b_x, b_y, b_h, b_w, c)$ as explained above. If you expand c into an 80-dimensional vector, each bounding box is then represented by 85 numbers.

Anchor Boxes

- Anchor boxes are chosen by exploring the training data to choose reasonable height/width ratios that represent the different classes. For this assignment, 5 anchor boxes were chosen for you (to cover the 80 classes), and stored in the file './model_data/yolo_anchors.txt'
- The dimension of the encoding tensor of the second to last dimension based on the anchor boxes is $(m, n_H, n_W, anchors, classes)$
- The YOLO architecture is: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).

```
def yolo_filter_boxes(boxes, box_confidence, box_class_probs, threshold = .6):

    """Filters YOLO boxes by thresholding on object and class confidence.


    Arguments:

        boxes -- tensor of shape (19, 19, 5, 4)

        box_confidence -- tensor of shape (19, 19, 5, 1)

        box_class_probs -- tensor of shape (19, 19, 5, 80)

        threshold -- real value, if [ highest class probability score < threshold],

                    then get rid of the corresponding box


    Returns:
```

scores -- tensor of shape (None,), containing the class probability score for selected boxes

boxes -- tensor of shape (None, 4), containing (b_x, b_y, b_h, b_w) coordinates of selected boxes

classes -- tensor of shape (None,), containing the index of the class detected by the selected boxes

Note: "None" is here because you don't know the exact number of selected boxes, as it depends on the threshold.

For example, the actual output size of scores would be (10,) if there are 10 boxes.

"""

START CODE HERE

Step 1: Compute box scores

##(≈ 1 line)

box_scores = box_confidence * box_class_probs

Step 2: Find the box_classes using the max box_scores, keep track of the corresponding score

##(≈ 2 lines)

IMPORTANT: set axis to -1

box_classes = tf.math.argmax(box_scores, axis=-1)

box_class_scores = tf.math.reduce_max(box_scores, axis=-1)

Step 3: Create a filtering mask based on "box_class_scores" by using "threshold". The mask should have the

same dimension as box_class_scores, and be True for the boxes you want to keep (with probability >= threshold)

(≈ 1 line)

filtering_mask = box_class_scores >= threshold

```
    # Step 4: Apply the mask to box_class_scores, boxes and box_classes

    ## (≈ 3 lines)

    scores = tf.boolean_mask(box_class_scores, filtering_mask)

    boxes = tf.boolean_mask(boxes, filtering_mask)

    classes = tf.boolean_mask(box_classes, filtering_mask)

    ### END CODE HERE


    return scores, boxes, classes
```

Non-max Suppression

Even after filtering by thresholding over the class scores, you still end up with a lot of overlapping boxes. A second filter for selecting the right boxes is called non-maximum suppression (NMS).

Intersection over Union

```
def iou(box1, box2):

    """Implement the intersection over union (IoU) between box1 and box2


    Arguments:

    box1 -- first box, list object with coordinates (box1_x1, box1_y1, box1_x2, box_1_y2)

    box2 -- second box, list object with coordinates (box2_x1, box2_y1, box2_x2, box2_y2)

    """

    (box1_x1, box1_y1, box1_x2, box1_y2) = box1

    (box2_x1, box2_y1, box2_x2, box2_y2) = box2

    ### START CODE HERE

    # Calculate the (yi1, xi1, yi2, xi2) coordinates of the intersection of box1 and box2. Calculate
its Area.
```

```
##(≈ 7 lines)

xi1 = max(box1_x1, box2_x1)  # Leftmost x coordinate of intersection

yi1 = max(box1_y1, box2_y1)  # Topmost y coordinate of intersection

xi2 = min(box1_x2, box2_x2)  # Rightmost x coordinate of intersection

yi2 = min(box1_y2, box2_y2)  # Bottommost y coordinate of intersection

inter_width = max(xi2 - xi1, 0)  # Ensure non-negative width

inter_height = max(yi2 - yi1, 0)  # Ensure non-negative height

inter_area = inter_width * inter_height  # Area of intersection


# Calculate the Union area by using Formula: Union(A,B) = A + B - Inter(A,B)

## (≈ 3 lines)

box1_area = (box1_x2 - box1_x1) * (box1_y2 - box1_y1)  # Area of box 1

box2_area = (box2_x2 - box2_x1) * (box2_y2 - box2_y1)  # Area of box 2

union_area = box1_area + box2_area - inter_area  # Union(A, B) = A + B - Intersection


# compute the IoU

iou = inter_area / union_area if union_area > 0 else 0  # Avoid division by zero

### END CODE HERE


return iou
```

YOLO Non-max Suppression

This process effectively eliminates boxes that overlap significantly with the selected boxes, leaving only the "best" candidates.

```
def yolo_non_max_suppression(scores, boxes, classes, max_boxes = 10, iou_threshold = 0.5):
```

```
    """
    Applies Non-max suppression (NMS) to set of boxes

    Arguments:
    scores -- tensor of shape (None,), output of yolo_filter_boxes()
    boxes -- tensor of shape (None, 4), output of yolo_filter_boxes() that have been scaled to the
image size (see later)
    classes -- tensor of shape (None,), output of yolo_filter_boxes()
    max_boxes -- integer, maximum number of predicted boxes you'd like
    iou_threshold -- real value, "intersection over union" threshold used for NMS filtering

    Returns:
    scores -- tensor of shape (None, ), predicted score for each box
    boxes -- tensor of shape (None, 4), predicted box coordinates
    classes -- tensor of shape (None, ), predicted class for each box

    Note: The "None" dimension of the output tensors has obviously to be less than max_boxes.
Note also that this
    function will transpose the shapes of scores, boxes, classes. This is made for convenience.
    """
    boxes = tf.cast(boxes, dtype=tf.float32)
    scores = tf.cast(scores, dtype=tf.float32)

    nms_indices = []
    classes_labels = tf.unique(classes)[0] # Get unique classes
```

```
    for label in classes_labels:
        filtering_mask = classes == label

    #### START CODE HERE

        # Get boxes for this class
        # Use tf.boolean_mask() with 'boxes' and `filtering_mask`
        boxes_label = tf.boolean_mask(boxes, filtering_mask)

        # Get scores for this class
        # Use tf.boolean_mask() with 'scores' and `filtering_mask`
        scores_label = tf.boolean_mask(scores, filtering_mask)

        if tf.shape(scores_label)[0] > 0:  # Check if there are any boxes to process

            # Use tf.image.non_max_suppression() to get the list of indices corresponding to boxes
you keep
            ##(≈ 5 lines)
            nms_indices_label = tf.image.non_max_suppression(
                boxes_label, scores_label, max_boxes, iou_threshold
            )

            # Get original indices of the selected boxes
            selected_indices = tf.squeeze(tf.where(filtering_mask), axis=1)

            # Append the resulting boxes into the partial result
```

```
        # Use tf.gather() with 'selected_indices' and `nms_indices_label`
        nms_indices.append(tf.gather(selected_indices, nms_indices_label))

    # Flatten the list of indices and concatenate
    # Use tf.concat() with 'nms_indices' and `axis=0`
    nms_indices = tf.concat(nms_indices, axis=0)

    # Use tf.gather() to select only nms_indices from scores, boxes and classes
    ##(≈ 3 lines)
    scores = tf.gather(scores, nms_indices)
    boxes = tf.gather(boxes, nms_indices)
    classes = tf.gather(classes, nms_indices)

    ### END CODE HERE

    # Sort by scores and return the top max_boxes
    sort_order = tf.argsort(scores, direction='DESCENDING').numpy()
    scores = tf.gather(scores, sort_order[0:max_boxes])
    boxes = tf.gather(boxes, sort_order[0:max_boxes])
    classes = tf.gather(classes, sort_order[0:max_boxes])

    return scores, boxes, classes
```

Wrapping Up the Filtering

It's time to implement a function taking the output of the deep CNN (the 19x19x5x85 dimensional encoding) and filtering through all the boxes using the functions you've just implemented.

```python
def yolo_boxes_to_corners(box_xy, box_wh):
    """Convert YOLO box predictions to bounding box corners."""
    box_mins = box_xy - (box_wh / 2.)
    box_maxes = box_xy + (box_wh / 2.)

    return tf.keras.backend.concatenate([
        box_mins[..., 1:2],  # y_min
        box_mins[..., 0:1],  # x_min
        box_maxes[..., 1:2],  # y_max
        box_maxes[..., 0:1]  # x_max
    ])
```

In [11]:

```python
# UNQ_C4 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: yolo_eval

def yolo_eval(yolo_outputs, image_shape = (720, 1280), max_boxes=10,
score_threshold=.6, iou_threshold=.5):
    """
    Converts the output of YOLO encoding (a lot of boxes) to your predicted
boxes along with their scores, box coordinates and classes.

    Arguments:
    yolo_outputs -- output of the encoding model (for image_shape of (608,
608, 3)), contains 4 tensors:
                    box_xy: tensor of shape (None, 19, 19, 5, 2)
                    box_wh: tensor of shape (None, 19, 19, 5, 2)
                    box_confidence: tensor of shape (None, 19, 19, 5, 1)
                    box_class_probs: tensor of shape (None, 19, 19, 5, 80)
    image_shape -- tensor of shape (2,) containing the input shape, in this
notebook we use (608., 608.) (has to be float32 dtype)
    max_boxes -- integer, maximum number of predicted boxes you'd like
    score_threshold -- real value, if [ highest class probability score <
threshold], then get rid of the corresponding box
    iou_threshold -- real value, "intersection over union" threshold used for
NMS filtering

    Returns:
    scores -- tensor of shape (None, ), predicted score for each box
    boxes -- tensor of shape (None, 4), predicted box coordinates
    classes -- tensor of shape (None,), predicted class for each box
    """
```

```
    ### START CODE HERE
    # Retrieve outputs of the YOLO model (≈1 line)
    box_xy, box_wh, box_confidence, box_class_probs = yolo_outputs

    # Convert boxes to be ready for filtering functions (convert boxes box_xy
and box_wh to corner coordinates)
    boxes = yolo_boxes_to_corners(box_xy, box_wh)

    # Use one of the functions you've implemented to perform Score-filtering
with a threshold of score_threshold (≈1 line)
    scores, boxes, classes = yolo_filter_boxes(boxes, box_confidence,
box_class_probs, score_threshold)

    # Scale boxes back to original image shape.
    boxes = scale_boxes(boxes, image_shape)

    # Use one of the functions you've implemented to perform Non-max
suppression with
    # maximum number of boxes set to max_boxes and a threshold of
iou_threshold (≈1 line)
    scores, boxes, classes = yolo_non_max_suppression(scores, boxes, classes,
max_boxes, iou_threshold)
    ### END CODE HERE

    return scores, boxes, classes
```

Run the YOLO on an Image

Let the fun begin! You will create a graph that can be summarized as follows:

`yolo_model.input` is given to `yolo_model`. The model is used to compute the output `yolo_model.output` `yolo_model.output` is processed by `yolo_head`. It gives you `yolo_outputs` `yolo_outputs` goes through a filtering function, `yolo_eval`. It outputs your predictions: `out_scores`, `out_boxes`, `out_classes`.
Now, we have implemented for you the `predict(image_file)` function, which runs the graph to test YOLO on an image to compute `out_scores`, `out_boxes`, `out_classes`.

The code below also uses the following function:

> image, image_data = preprocess_image("images/" + image_file, model_image_size = (608, 608))

which opens the image file and scales, reshapes and normalizes the image. It returns the outputs:
image: a python (PIL) representation of your image used for drawing boxes. You won't need to use it.

image_data: a numpy-array representing the image. This will be the input to the CNN.

In [ ]:

```
def predict(image_file):
    """
    Runs the graph to predict boxes for "image_file". Prints and plots the
predictions.

    Arguments:
    image_file -- name of an image stored in the "images" folder.

    Returns:
    out_scores -- tensor of shape (None, ), scores of the predicted boxes
    out_boxes -- tensor of shape (None, 4), coordinates of the predicted boxes
    out_classes -- tensor of shape (None, ), class index of the predicted
boxes

    Note: "None" actually represents the number of predicted boxes, it varies
between 0 and max_boxes.
    """

    # Preprocess your image
    image, image_data = preprocess_image("images/" + image_file,
model_image_size = (608, 608))

    yolo_model_outputs = yolo_model(image_data)
    yolo_outputs = yolo_head(yolo_model_outputs, anchors, len(class_names))

    out_scores, out_boxes, out_classes = yolo_eval(yolo_outputs,
[image.size[1],  image.size[0]], 10, 0.3, 0.5)

    # Print predictions info
    print('Found {} boxes for {}'.format(len(out_boxes), "images/" +
image_file))
    # Generate colors for drawing bounding boxes.
    colors = get_colors_for_classes(len(class_names))
    # Draw bounding boxes on the image file
    #draw_boxes2(image, out_scores, out_boxes, out_classes, class_names,
colors, image_shape)
    draw_boxes(image, out_boxes, out_classes, class_names, out_scores)
    # Save the predicted bounding box on the image
    image.save(os.path.join("out", image_file), quality=100)
    # Display the results in the notebook
    output_image = Image.open(os.path.join("out", image_file))
    imshow(output_image)
```

```
return out_scores, out_boxes, out_classes
```

Summary for YOLO

- Input image (608, 608, 3)
- The input image goes through a CNN, resulting in a (19,19,5,85) dimensional output.
- After flattening the last two dimensions, the output is a volume of shape (19, 19, 425):
    - Each cell in a 19x19 grid over the input image gives 425 numbers.
    - 425 = 5 x 85 because each cell contains predictions for 5 boxes, corresponding to 5 anchor boxes, as seen in lecture.
    - 85 = 5 + 80 where 5 is because $(p_c, b_x, b_y, b_h, b_w)$ has 5 numbers, and 80 is the number of classes we'd like to detect
- You then select only few boxes based on:
    - Score-thresholding: throw away boxes that have detected a class with a score less than the threshold
    - Non-max suppression: Compute the Intersection over Union and avoid selecting overlapping boxes
- This gives you YOLO's final output.

What you should remember:

- YOLO is a state-of-the-art object detection model that is fast and accurate
- It runs an input image through a CNN, which outputs a 19x19x5x85 dimensional volume.
- The encoding can be seen as a grid where each of the 19x19 cells contains information about 5 boxes.
- You filter through all the boxes using non-max suppression. Specifically:
    - Score thresholding on the probability of detecting a class to keep only accurate (high probability) boxes
    - Intersection over Union (IoU) thresholding to eliminate overlapping boxes
- Because training a YOLO model from randomly initialized weights is non-trivial and requires a large dataset as well as lot of computation, previously trained model parameters were used in this exercise. If you wish, you can also try fine-tuning the YOLO model with your own dataset, though this would be a fairly non-trivial exercise.

Image Segmentation with U-Net

1.
Contracting path (Encoder containing downsampling steps):

Images are first fed through several convolutional layers which reduce height and width, while growing the number of channels.

The contracting path follows a regular CNN architecture, with convolutional layers, their activations, and pooling layers to downsample the image and extract its features. In detail, it consists of the repeated application of two 3 x 3 same padding convolutions, each followed by a rectified linear unit (ReLU) and a 2 x 2 max pooling operation with stride 2 for downsampling. At each downsampling step, the number of feature channels is doubled.

```
def conv_block(inputs=None, n_filters=32, dropout_prob=0, max_pooling=True):
    """
    Convolutional downsampling block

    Arguments:
        inputs -- Input tensor
        n_filters -- Number of filters for the convolutional layers
        dropout_prob -- Dropout probability
        max_pooling -- Use MaxPooling2D to reduce the spatial dimensions of the output volume
    Returns:
        next_layer, skip_connection --  Next layer and skip connection outputs
    """

    conv = Conv2D(filters=n_filters,  # Number of filters
                  kernel_size=3,       # 3x3 kernel
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(inputs)

    conv = Conv2D(filters=n_filters,
                  kernel_size=3,
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(conv)

    # if dropout_prob > 0 add a dropout layer, with the variable dropout_prob as parameter
    if dropout_prob > 0:
         ### START CODE HERE
        conv = Dropout(rate=dropout_prob)(conv)
         ### END CODE HERE
```

```
    # if max_pooling is True add a MaxPooling2D with 2x2 pool_size
    if max_pooling:
        ### START CODE HERE
        next_layer = MaxPooling2D(pool_size=(2, 2))(conv)
        ### END CODE HERE

    else:
        next_layer = conv

    skip_connection = conv

    return next_layer, skip_connection
def conv_block(inputs=None, n_filters=32, dropout_prob=0, max_pooling=True):
    """
    Convolutional downsampling block

    Arguments:
        inputs -- Input tensor
        n_filters -- Number of filters for the convolutional layers
        dropout_prob -- Dropout probability
        max_pooling -- Use MaxPooling2D to reduce the spatial dimensions of the output volume
    Returns:
        next_layer, skip_connection --  Next layer and skip connection outputs
    """

    conv = Conv2D(filters=n_filters,  # Number of filters
                  kernel_size=3,       # 3x3 kernel
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(inputs)

    conv = Conv2D(filters=n_filters,
                  kernel_size=3,
                  activation='relu',
                  padding='same',
                  kernel_initializer='he_normal')(conv)

    # if dropout_prob > 0 add a dropout layer, with the variable dropout_prob as parameter
    if dropout_prob > 0:
         ### START CODE HERE
        conv = Dropout(rate=dropout_prob)(conv)
         ### END CODE HERE
```

```
    # if max_pooling is True add a MaxPooling2D with 2x2 pool_size
    if max_pooling:
        ### START CODE HERE
        next_layer = MaxPooling2D(pool_size=(2, 2))(conv)
        ### END CODE HERE

    else:
        next_layer = conv

    skip_connection = conv

    return next_layer, skip_connection
```

2.

Decoder (Upsampling Block)

The decoder, or upsampling block, upsamples the features back to the original image size. At each upsampling level, you'll take the output of the corresponding encoder block and concatenate it before feeding to the next decoder block.

```
def upsampling_block(expansive_input, contractive_input, n_filters=32):
    """
    Convolutional upsampling block

    Arguments:
        expansive_input -- Input tensor from previous layer
        contractive_input -- Input tensor from previous skip layer
        n_filters -- Number of filters for the convolutional layers
    Returns:
        conv -- Tensor output
    """

    up = Conv2DTranspose(filters=n_filters,   # Number of filters
                         kernel_size=3,       # Kernel size (3x3)
                         strides=(2, 2),      # Stride (2x2) for upsampling
                         padding='same')(expansive_input)

    # Merge the previous output and the contractive_input
    merge = concatenate([up, contractive_input], axis=3)
    conv = Conv2D(filters=n_filters,
```

```
                 kernel_size=3,
                 activation='relu',
                 padding='same',
                 kernel_initializer='he_normal')(merge)

    conv = Conv2D(filters=n_filters,
                 kernel_size=3,
                 activation='relu',
                 padding='same',
                 kernel_initializer='he_normal')(conv)
    ### END CODE HERE

    return conv
```

3.

```
def unet_model(input_size=(96, 128, 3), n_filters=32, n_classes=23):
    """
    Unet model

    Arguments:
        input_size -- Input shape
        n_filters -- Number of filters for the convolutional layers
        n_classes -- Number of output classes
    Returns:
        model -- tf.keras.Model
    """
    inputs = Input(input_size)
    # Contracting Path (encoding)
    # Add a conv_block with the inputs of the unet_ model and n_filters
    ### START CODE HERE
    cblock1 = conv_block(inputs, n_filters)
    # Chain the first element of the output of each block to be the input of the next conv_block.
    # Double the number of filters at each new step
    cblock2 = conv_block(cblock1[0], n_filters * 2)
    cblock3 = conv_block(cblock2[0], n_filters * 4)
    cblock4 = conv_block(cblock3[0], n_filters * 8, dropout_prob=0.3)  # Apply dropout
    # Include a dropout_prob of 0.3 for this layer, and avoid the max_pooling layer
    cblock5 = conv_block(cblock4[0], n_filters * 16, dropout_prob=0.3, max_pooling=False)  # No
pooling in bottleneck
    ### END CODE HERE

    # Expanding Path (decoding)
    # Add the first upsampling_block.
```

```
    # Use the cblock5[0] as expansive_input and cblock4[1] as contractive_input and n_filters * 8
    ### START CODE HERE
    ublock6 = upsampling_block(cblock5[0], cblock4[1], n_filters * 8)
    # Chain the output of the previous block as expansive_input and the corresponding
contractive block output.
    # Note that you must use the second element of the contractive block i.e before the
maxpooling layer.
    # At each step, use half the number of filters of the previous block
    ublock7 = upsampling_block(ublock6, cblock3[1], n_filters * 4)
    ublock8 = upsampling_block(ublock7, cblock2[1], n_filters * 2)
    ublock9 = upsampling_block(ublock8, cblock1[1], n_filters)
    ### END CODE HERE

    conv9 = Conv2D(n_filters,
                 3,
                 activation='relu',
                 padding='same',
                 # set 'kernel_initializer' same as above exercises
                 kernel_initializer='he_normal')(ublock9)

    # Add a Conv2D layer with n_classes filter, kernel size of 1 and a 'same' padding
    ### START CODE HERE
    conv10 = Conv2D(n_classes,  # Number of output classes
                   kernel_size=1,
                   activation='linear',  # Softmax for multi-class segmentation
                   padding='same')(conv9)
    ### END CODE HERE

    model = tf.keras.Model(inputs=inputs, outputs=conv10)

    return model
```

3.5 - Set Model Dimensions

In [15]:

```
img_height = 96
img_width = 128
num_channels = 3

unet = unet_model((img_height, img_width, num_channels))
```

Loss Function

In semantic segmentation, you need as many masks as you have object classes. In the dataset you're using, each pixel in every mask has been assigned a single integer probability that it belongs to a certain class, from 0 to num_classes-1. The correct class is the layer with the higher probability.

This is different from categorical crossentropy, where the labels should be one-hot encoded (just 0s and 1s). Here, you'll use sparse categorical crossentropy as your loss function, to perform pixel-wise multiclass prediction. Sparse categorical crossentropy is more efficient than other loss functions when you're dealing with lots of classes.

In [17]:

```
unet.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
             metrics=['accuracy'])
```

Dataset Handling

Below, define a function that allows you to display both an input image, and its ground truth: the true mask. The true mask is what your trained model output is aiming to get as close to as possible.

In [18]:

```
def display(display_list):
   plt.figure(figsize=(15, 15))

   title = ['Input Image', 'True Mask', 'Predicted Mask']

   for i in range(len(display_list)):
       plt.subplot(1, len(display_list), i+1)
       plt.title(title[i])
       plt.imshow(tf.keras.preprocessing.image.array_to_img(display_list[i]))
       plt.axis('off')
   plt.show()
```

In [19]:

```
for image, mask in image_ds.take(1):
   sample_image, sample_mask = image, mask
   print(mask.shape)
display([sample_image, sample_mask])
```

Training

EPOCHS = 15
VAL_SUBSPLITS = 5
BUFFER_SIZE = 500
BATCH_SIZE = 32

tf.keras.utils.set_random_seed(1)
tf.config.experimental.enable_op_determinism()

```
train_dataset = processed_image_ds.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
print(processed_image_ds.element_spec)

unet = unet_model((img_height, img_width, num_channels))
unet.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=['accuracy']
)
```

```
model_history = unet.fit(train_dataset, epochs=EPOCHS)
```

Create Predicted Masks

Now, define a function that uses `tf.argmax` in the axis of the number of classes to return the index with the largest value and merge the prediction into a single image:

In [22]:

```
def create_mask(pred_mask):
    pred_mask = tf.argmax(pred_mask, axis=-1)
    pred_mask = pred_mask[..., tf.newaxis]
    return pred_mask[0]
```

Plot Model Accuracy

Let's see how your model did!

In [23]:

```
plt.plot(model_history.history["accuracy"])
```

Show Predictions

Next, check your predicted masks against the true mask and the original input image:

In [24]:

```
def show_predictions(dataset=None, num=1):
    """
    Displays the first image of each of the num batches
    """
    if dataset:
        for image, mask in dataset.take(num):
            pred_mask = unet.predict(image)
            display([image[0], mask[0], create_mask(pred_mask)])
    else:
        display([sample_image, sample_mask,
             create_mask(unet.predict(sample_image[tf.newaxis, ...]))])
show_predictions(train_dataset, 6)
```

Face Recognition

Face Verification "Is this the claimed person?" For example, at some airports, you can pass through customs by letting a system scan your passport and then verifying that you (the person carrying the passport) are the correct person. A mobile phone that unlocks using your face is also using face verification. This is a 1:1 matching problem.

Face Recognition "Who is this person?" For example, the video lecture showed a [face recognition video]() of Baidu employees entering the office without needing to otherwise identify themselves. This is a 1:K matching problem.

Using a ConvNet to Compute Encodings

The FaceNet model takes a lot of data and a long time to train. So following the common practice in applied deep learning, you'll load weights that someone else has already trained. The network architecture follows the Inception model from [Szegedy *et al.*.]()

The key things to be aware of are:

- This network uses 160x160 dimensional RGB images as its input. Specifically, a face image (or batch of m face images) as a tensor of shape $(m, n_H, n_W, n_C) = (m, 160, 160, 3)$
- The input images are originally of shape 96x96, thus, you need to scale them to 160x160. This is done in the `img_to_encoding()` function.
- The output is a matrix of shape
- $(m, 128)$
- (m,128) that encodes each input face image into a 128-dimensional vector

The Triplet Loss

Important Note: Since you're using a pretrained model, you won't actually need to implement the triplet loss function in this assignment. *However*, the triplet loss is the main ingredient of the face recognition algorithm, and you'll need to know how to use it for training your own FaceNet model, as well as other types of image similarity problems. Therefore, you'll implement it below, for fun and edification. :)

```
def triplet_loss(y_true, y_pred, alpha = 0.2):
    """
    Implementation of the triplet loss as defined by formula (3)

    Arguments:
    y_true -- true labels, required when you define a loss in Keras, you don't need it in this
function.
    y_pred -- python list containing three objects:
```

anchor -- the encodings for the anchor images, of shape (None, 128)
positive -- the encodings for the positive images, of shape (None, 128)
negative -- the encodings for the negative images, of shape (None, 128)

Returns:
loss -- real number, value of the loss
"""

anchor, positive, negative = y_pred[0], y_pred[1], y_pred[2]

START CODE HERE
#(≈ 4 lines)
# Step 1: Compute the (encoding) distance between the anchor and the positive
pos_dist = tf.reduce_sum(tf.square(tf.subtract(anchor, positive)), axis=-1)
# Step 2: Compute the (encoding) distance between the anchor and the negative
neg_dist = tf.reduce_sum(tf.square(tf.subtract(anchor, negative)), axis=-1)
# Step 3: subtract the two previous distances and add alpha.
basic_loss = tf.add(tf.subtract(pos_dist, neg_dist), alpha)
# Step 4: Take the maximum of basic_loss and 0.0. Sum over the training examples.
loss = tf.reduce_sum(tf.maximum(basic_loss, 0.0))
END CODE HERE

return loss

Applying the Model

You're building a system for an office building where the building manager would like to offer facial recognition to allow the employees to enter the building.

You'd like to build a face verification system that gives access to a list of people. To be admitted, each person has to swipe an identification card at the entrance. The face recognition system then verifies that they are who they claim to be.

5.1 - Face Verification

Now you'll build a database containing one encoding vector for each person who is allowed to enter the office. To generate the encoding, you'll use `img_to_encoding(image_path, model)`, which runs the forward propagation of the model on the specified image.

Run the following code to build the database (represented as a Python dictionary). This database maps each person's name to a 128-dimensional encoding of their face.

In [ ]:

```
#tf.keras.backend.set_image_data_format('channels_last')
```

```python
def img_to_encoding(image_path, model):
    img = tf.keras.preprocessing.image.load_img(image_path, target_size=(160,
160))
    img = np.around(np.array(img) / 255.0, decimals=12)
    x_train = np.expand_dims(img, axis=0)
    embedding = model.predict_on_batch(x_train)
    return embedding / np.linalg.norm(embedding, ord=2)
```

In [ ]:

```python
database = {}
database["danielle"] = img_to_encoding("images/danielle.png", FRmodel)
database["younes"] = img_to_encoding("images/younes.jpg", FRmodel)
database["tian"] = img_to_encoding("images/tian.jpg", FRmodel)
database["andrew"] = img_to_encoding("images/andrew.jpg", FRmodel)
database["kian"] = img_to_encoding("images/kian.jpg", FRmodel)
database["dan"] = img_to_encoding("images/dan.jpg", FRmodel)
database["sebastiano"] = img_to_encoding("images/sebastiano.jpg", FRmodel)
database["bertrand"] = img_to_encoding("images/bertrand.jpg", FRmodel)
database["kevin"] = img_to_encoding("images/kevin.jpg", FRmodel)
database["felix"] = img_to_encoding("images/felix.jpg", FRmodel)
database["benoit"] = img_to_encoding("images/benoit.jpg", FRmodel)
database["arnaud"] = img_to_encoding("images/arnaud.jpg", FRmodel)
```

Load the images of Danielle and Kian:

In [ ]:

```python
danielle = tf.keras.preprocessing.image.load_img("images/danielle.png",
target_size=(160, 160))
kian = tf.keras.preprocessing.image.load_img("images/kian.jpg",
target_size=(160, 160))
```

In [ ]:

```python
np.around(np.array(kian) / 255.0, decimals=12).shape
```

In [ ]:

```python
kian
```

In [ ]:

```python
np.around(np.array(danielle) / 255.0, decimals=12).shape
```

In [ ]:

```python
danielle
```

Now, when someone shows up at your front door and swipes their ID card (thus giving you their name), you can look up their encoding in the database, and use it to check if the person standing at the front door matches the name on the ID.

def verify(image_path, identity, database, model):

"""

Function that verifies if the person on the "image_path" image is "identity".

Arguments:

image_path -- path to an image

identity -- string, name of the person you'd like to verify the identity. Has to be an employee who works in the office.
database -- python dictionary mapping names of allowed people's names (strings) to their encodings (vectors).
model -- your Inception model instance in Keras

Returns:
dist -- distance between the image_path and the image of "identity" in the database.
door_open -- True, if the door should open. False otherwise.
"""
START CODE HERE
# Step 1: Compute the encoding for the image. Use img_to_encoding() see example above. (≈ 1 line)
encoding = img_to_encoding(image_path, model)
# Step 2: Compute distance with identity's image (≈ 1 line)
dist = np.linalg.norm(encoding - database[identity])
# Step 3: Open the door if dist < 0.7, else don't open (≈ 3 lines)
if dist < 0.7:
print("✅ It's " + str(identity) + ", welcome in!")
door_open = True
else:
print("❌ It's not " + str(identity) + ", access denied!")
door_open = False
END CODE HERE
return dist, door_open

who_is_it

Implement `who_is_it()` with the following steps:

- Compute the target encoding of the image from `image_path`
- Find the encoding from the database that has smallest distance with the target encoding.
- Initialize the `min_dist` variable to a large enough number (100). This helps you keep track of the closest encoding to the input's encoding.
- Loop over the database dictionary's names and encodings. To loop use for (name, db_enc) in `database.items()`.
- Compute the L2 distance between the target "encoding" and the current "encoding" from the database. If this distance is less than the min_dist, then set min_dist to dist, and identity to name.

In [25]:

```
# UNQ_C3(UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# GRADED FUNCTION: who_is_it

def who_is_it(image_path, database, model):
```

```
    """
    Implements face recognition for the office by finding who is the person on 
the image_path image.
    
    Arguments:
        image_path -- path to an image
        database -- database containing image encodings along with the name of 
the person on the image
        model -- your Inception model instance in Keras
    
    Returns:
        min_dist -- the minimum distance between image_path encoding and the 
encodings from the database
        identity -- string, the name prediction for the person on image_path
    """
    
    ### START CODE HERE 
    
    ## Step 1: Compute the target "encoding" for the image. Use 
img_to_encoding() see example above. ## (≈ 1 line)
    encoding = img_to_encoding(image_path, model)
    
    ## Step 2: Find the closest encoding ##
    
    # Initialize "min_dist" to a large value, say 100 (≈1 line)
    min_dist = 100
    
    # Loop over the database dictionary's names and encodings.
    for (name, db_enc) in database.items():
        
        # Compute L2 distance between the target "encoding" and the current 
db_enc from the database. (≈ 1 line)
        dist = np.linalg.norm(encoding - db_enc)

        # If this distance is less than the min_dist, then set min_dist to 
dist, and identity to name. (≈ 3 lines)
        if dist < min_dist:
            min_dist = dist
            identity = name
    ### END CODE HERE
    
    if min_dist > 0.7:
        print("Not in the database.")
    else:
        print ("it's " + str(identity) + ", the distance is " + str(min_dist))
```

```
    return min_dist, identity
```

Younes is at the front door and the camera takes a picture of him ("images/camera_0.jpg"). Let's see if your `who_it_is()` algorithm identifies Younes.

In [26]:

```
### YOU CANNOT EDIT THIS CELL

# BEGIN UNIT TEST
# Test 1 with Younes pictures
who_is_it("images/camera_0.jpg", database, FRmodel)

# Test 2 with Younes pictures
test1 = who_is_it("images/camera_0.jpg", database, FRmodel)
assert np.isclose(test1[0], 0.5992946)
assert test1[1] == 'younes'

# Test 3 with Younes pictures
test2 = who_is_it("images/younes.jpg", database, FRmodel)
assert np.isclose(test2[0], 0.0)
assert test2[1] == 'younes'
# END UNIT TEST
```

Deep Learning & Art: Neural Style Transfer

Transfer Learning

Neural Style Transfer (NST) uses a previously trained convolutional network, and builds on top of that. The idea of using a network trained on a different task and applying it to a new task is called transfer learning.

```
def compute_content_cost(content_output, generated_output):
    """
    Computes the content cost

    Arguments:
    a_C -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing content of the image C
    a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing content of the image G

    Returns:
    J_content -- scalar that you compute using equation 1 above.
    """
```

```
    a_C = content_output[-1]
    a_G = generated_output[-1]

    ### START CODE HERE

    # Retrieve dimensions from a_G (≈1 line)
    _, n_H, n_W, n_C = a_G.get_shape().as_list()

    # Reshape 'a_C' and 'a_G' (≈2 lines)
    # DO NOT reshape 'content_output' or 'generated_output'
    a_C_unrolled = tf.reshape(a_C, shape=[-1, n_H * n_W, n_C])  # Reshape to (batch_size,
height*width, channels)
    a_G_unrolled = tf.reshape(a_G, shape=[-1, n_H * n_W, n_C])

    # compute the cost with tensorflow (≈1 line)
    J_content = tf.reduce_sum(tf.square(tf.subtract(a_C_unrolled, a_G_unrolled))) / (4 * n_H *
n_W * n_C)

    ### END CODE HERE

    return J_content
```

What you should remember:

- The content cost takes a hidden layer activation of the neural network, and measures how different $a(C)$ and $a(G)$ are.
- When you minimize the content cost later, this will help make sure G has similar content as C

```
def gram_matrix(A):
    """
    Argument:
    A -- matrix of shape (n_C, n_H*n_W)

    Returns:
    GA -- Gram matrix of A, of shape (n_C, n_C)
    """
    ### START CODE HERE

    #(≈1 line)
    GA = tf.matmul(A, tf.transpose(A))

    ### END CODE HERE

    return GA
```

```
def compute_layer_style_cost(a_S, a_G):
    """
    Arguments:
    a_S -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing style of
the image S
    a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer activations representing style of
the image G

    Returns:
    J_style_layer -- tensor representing a scalar value, style cost defined above by equation (2)
    """
    ### START CODE HERE

    # Retrieve dimensions from a_G (≈1 line)
    _, n_H, n_W, n_C = a_G.get_shape().as_list()

    # Reshape the tensors from (1, n_H, n_W, n_C) to (n_C, n_H * n_W) (≈2 lines)
    a_S = tf.reshape(tf.transpose(a_S, perm=[0,3,1,2]), [n_C, n_H * n_W])
    a_G = tf.reshape(tf.transpose(a_G, perm=[0,3,1,2]), [n_C, n_H * n_W])

    # Computing gram_matrices for both images S and G (≈2 lines)
    GS = gram_matrix(a_S)
    GG = gram_matrix(a_G)

    # Computing the loss (≈1 line)
    J_style_layer = tf.reduce_sum(tf.square(tf.subtract(GS, GG))) / (4 * (n_C ** 2) * (n_H * n_W)
** 2)

    ### END CODE HERE

    return J_style_layer
def compute_style_cost(style_image_output, generated_image_output,
STYLE_LAYERS=STYLE_LAYERS):
    """
    Computes the overall style cost from several chosen layers

    Arguments:
    style_image_output -- our tensorflow model
    generated_image_output --
    STYLE_LAYERS -- A python list containing:
                        - the names of the layers we would like to extract style from
                        - a coefficient for each of them

    Returns:
```

```
    J_style -- tensor representing a scalar value, style cost defined above by equation (2)
    """

    # initialize the overall style cost
    J_style = 0

    # Set a_S to be the hidden layer activation from the layer we have selected.
    # The last element of the array contains the content layer image, which must not be used.
    a_S = style_image_output[:-1]

    # Set a_G to be the output of the choosen hidden layers.
    # The last element of the list contains the content layer image which must not be used.
    a_G = generated_image_output[:-1]
    for i, weight in zip(range(len(a_S)), STYLE_LAYERS):
        # Compute style_cost for the current layer
        J_style_layer = compute_layer_style_cost(a_S[i], a_G[i])

        # Add weight * J_style_layer of this layer to overall style cost
        J_style += weight[1] * J_style_layer

    return J_style
```

What you should remember:

- The style of an image can be represented using the Gram matrix of a hidden layer's activations.
- You get even better results by combining this representation from multiple different layers.
- This is in contrast to the content representation, where usually using just a single hidden layer is sufficient.
- Minimizing the style cost will cause the image G to follow the style of the image S.

```
@tf.function()
def total_cost(J_content, J_style, alpha = 10, beta = 40):
    """
    Computes the total cost function

    Arguments:
    J_content -- content cost coded above
    J_style -- style cost coded above
    alpha -- hyperparameter weighting the importance of the content cost
    beta -- hyperparameter weighting the importance of the style cost

    Returns:
    J -- total cost as defined by the formula above.
```

""""
START CODE HERE

#(≈1 line)
J = alpha * J_content + beta * J_style

START CODE HERE

return J

What you should remember:

- The total cost is a linear combination of the content cost $J_{content}(C,G)$ and the style cost $J_{style}(S,G)$
- α and β are hyperparameters that control the relative weighting between content and style.

Solving the Optimization Problem

Finally, you get to put everything together to implement Neural Style Transfer!

Here's what your program be able to do:

1. Load the content image
2. Load the style image
3. Randomly initialize the image to be generated
4. Load the VGG19 model
5. Compute the content cost
6. Compute the style cost
7. Compute the total cost
8. Define the optimizer and learning rate

Here are the individual steps in detail.

5.1 Load the Content Image

Run the following code cell to load, reshape, and normalize your "content" image C (the Louvre museum picture):

In [ ]:

```
content_image = np.array(Image.open("images/louvre_small.jpg").resize((img_size, img_size)))
content_image = tf.constant(np.reshape(content_image, ((1,) + content_image.shape)))

print(content_image.shape)
imshow(content_image[0])
plt.show()
```

5.2 Load the Style Image

Now load, reshape and normalize your "style" image (Claude Monet's painting):

In [ ]:

```
style_image =  np.array(Image.open("images/monet.jpg").resize((img_size, img_size)))
style_image = tf.constant(np.reshape(style_image, ((1,) + style_image.shape)))

print(style_image.shape)
imshow(style_image[0])
plt.show()
```

5.3 Randomly Initialize the Image to be Generated

Now, you get to initialize the "generated" image as a noisy image created from the content_image.

- The generated image is slightly correlated with the content image.
- By initializing the pixels of the generated image to be mostly noise but slightly correlated with the content image, this will help the content of the "generated" image more rapidly match the content of the "content" image.

In [ ]:

```
generated_image = tf.Variable(tf.image.convert_image_dtype(content_image, tf.float32))
noise = tf.random.uniform(tf.shape(generated_image), -0.25, 0.25)
generated_image = tf.add(generated_image, noise)
generated_image = tf.clip_by_value(generated_image, clip_value_min=0.0, clip_value_max=1.0)

print(generated_image.shape)
imshow(generated_image.numpy()[0])
plt.show()
```

5.4 - Load Pre-trained VGG19 Model

Next, as explained in part(2), define a function which loads the VGG19 model and returns a list of the outputs for the middle layers.

In [ ]:

```
def get_layer_outputs(vgg, layer_names):
    """ Creates a vgg model that returns a list of intermediate output values."""
    outputs = [vgg.get_layer(layer[0]).output for layer in layer_names]

    model = tf.keras.Model([vgg.input], outputs)
    return model
```

Now, define the content layer and build the model.

In [ ]:

```
content_layer = [('block5_conv4', 1)]

vgg_model_outputs = get_layer_outputs(vgg, STYLE_LAYERS + content_layer)
```

Save the outputs for the content and style layers in separate variables.

In [ ]:

```
content_target = vgg_model_outputs(content_image)   # Content encoder
style_targets = vgg_model_outputs(style_image)      # Style encoder
```

5.5 - Compute Total Cost

5.5.1 - Compute the Content image Encoding (a_C)

You've built the model, and now to compute the content cost, you will encode your content image using the appropriate hidden layer activations. Set this encoding to the variable `a_C`. Later in the assignment, you will need to do the same for the generated image, by setting the variable `a_G` to be the appropriate hidden layer activations. You will use layer `block5_conv4` to compute the encoding. The code below does the following:

1. Set a_C to be the tensor giving the hidden layer activation for layer "block5_conv4" using the content image.

In [ ]:

```
# Assign the content image to be the input of the VGG model.
# Set a_C to be the hidden layer activation from the layer we have selected
preprocessed_content =  tf.Variable(tf.image.convert_image_dtype(content_image, tf.float32))
a_C = vgg_model_outputs(preprocessed_content)
```

5.5.2 - Compute the Style image Encoding (a_S)

The code below sets a_S to be the tensor giving the hidden layer activation for `STYLE_LAYERS` using our style image.

In [ ]:

```
# Assign the input of the model to be the "style" image
preprocessed_style =  tf.Variable(tf.image.convert_image_dtype(style_image, tf.float32))
a_S = vgg_model_outputs(preprocessed_style)
```

Below are the utils that you will need to display the images generated by the style transfer model.

In [ ]:

```
def clip_0_1(image):
    """
    Truncate all the pixels in the tensor to be between 0 and 1

    Arguments:
```

```
    image -- Tensor
    J_style -- style cost coded above

    Returns:
    Tensor
    """
    return tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0)

def tensor_to_image(tensor):
    """
    Converts the given tensor into a PIL image

    Arguments:
    tensor -- Tensor

    Returns:
    Image: A PIL image
    """
    tensor = tensor * 255
    tensor = np.array(tensor, dtype=np.uint8)
    if np.ndim(tensor) > 3:
        assert tensor.shape[0] == 1
        tensor = tensor[0]
    return Image.fromarray(tensor)
```

train_step

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.01)

@tf.function()
def train_step(generated_image):
  with tf.GradientTape() as tape:
    # In this function you must use the precomputed encoded images a_S and a_C

    # Compute a_G as the vgg_model_outputs for the current generated image
    #(1 line)
    a_G = vgg_model_outputs(generated_image)

    # Compute the style cost
    #(1 line)
    J_style = compute_style_cost(a_S, a_G)

    #(2 lines)
    # Compute the content cost
```

```
        J_content = compute_content_cost(a_C, a_G)
        # Compute the total cost
        J = total_cost(J_content, J_style, alpha=10, beta=40)

    grad = tape.gradient(J, generated_image)

    optimizer.apply_gradients([(grad, generated_image)])
    generated_image.assign(clip_0_1(generated_image))
    # For grading purposes
    return J
```

Train the Model

Run the following cell to generate an artistic image. It should take about 3min on a GPU for 2500 iterations. Neural Style Transfer is generally trained using GPUs.

If you increase the learning rate you can speed up the style transfer, but often at the cost of quality.

In [*]:

```
# Show the generated image at some epochs
# Uncomment to reset the style transfer process. You will need to compile the
train_step function again
epochs = 2501
for i in range(epochs):
   train_step(generated_image)
   if i % 250 == 0:
       print(f"Epoch {i} ")
   if i % 250 == 0:
       image = tensor_to_image(generated_image)
       imshow(image)
       image.save(f"output/image_{i}.jpg")
       plt.show()
```

```
# Show the 3 images in a row
fig = plt.figure(figsize=(16, 4))
ax = fig.add_subplot(1, 3, 1)
imshow(content_image[0])
ax.title.set_text('Content image')
ax = fig.add_subplot(1, 3, 2)
imshow(style_image[0])
ax.title.set_text('Style image')
ax = fig.add_subplot(1, 3, 3)
imshow(generated_image[0])
ax.title.set_text('Generated image')
plt.show()
```