Deep Neural Network for Image Classification: Application

- Packages
- Dataset
- Model Architecture
- Two-Layer Neural Network
- L-Layer Neural Network
- Results Analysis
- Test with Your Own Image

1 - Packages

Let's first import all the packages that you will need during this assignment.

- numpy is the fundamental package for scientific computing with Python.
- matplotlib is a library to plot graphs in Python.
- np.random.seed(1) is used to keep all the random function calls consistent. It will help us grade your work.

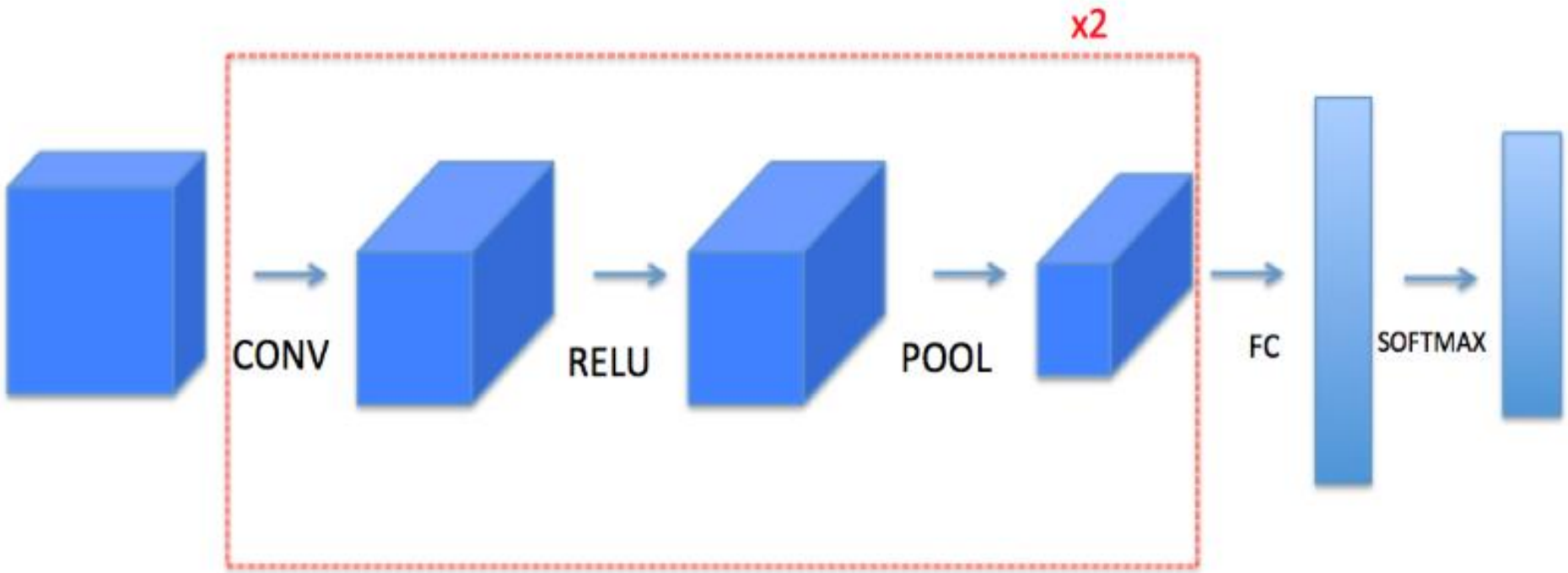
```
import numpy as np
import h5py
import matplotlib.pyplot as plt

%matplotlib inline
# set default size of plots
plt.rcParams['figure.figsize'] = (5.0, 4.0)
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

2 - Outline of the Assignment

You will be implementing the building blocks of a convolutional neural network! Each function you will implement will have detailed instructions that will walk you through the steps needed:

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

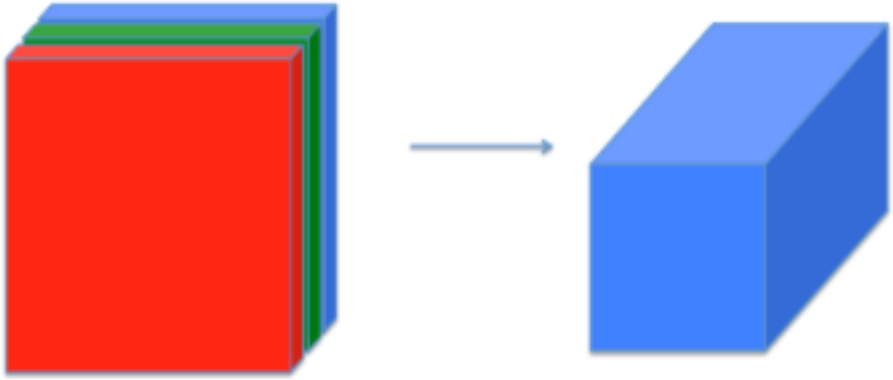
- This notebook will ask you to implement these functions from scratch in numpy.



3 - Convolutional Neural Networks

Although programming frameworks make convolutions easy to use, they remain one of the hardest concepts to understand in Deep Learning.

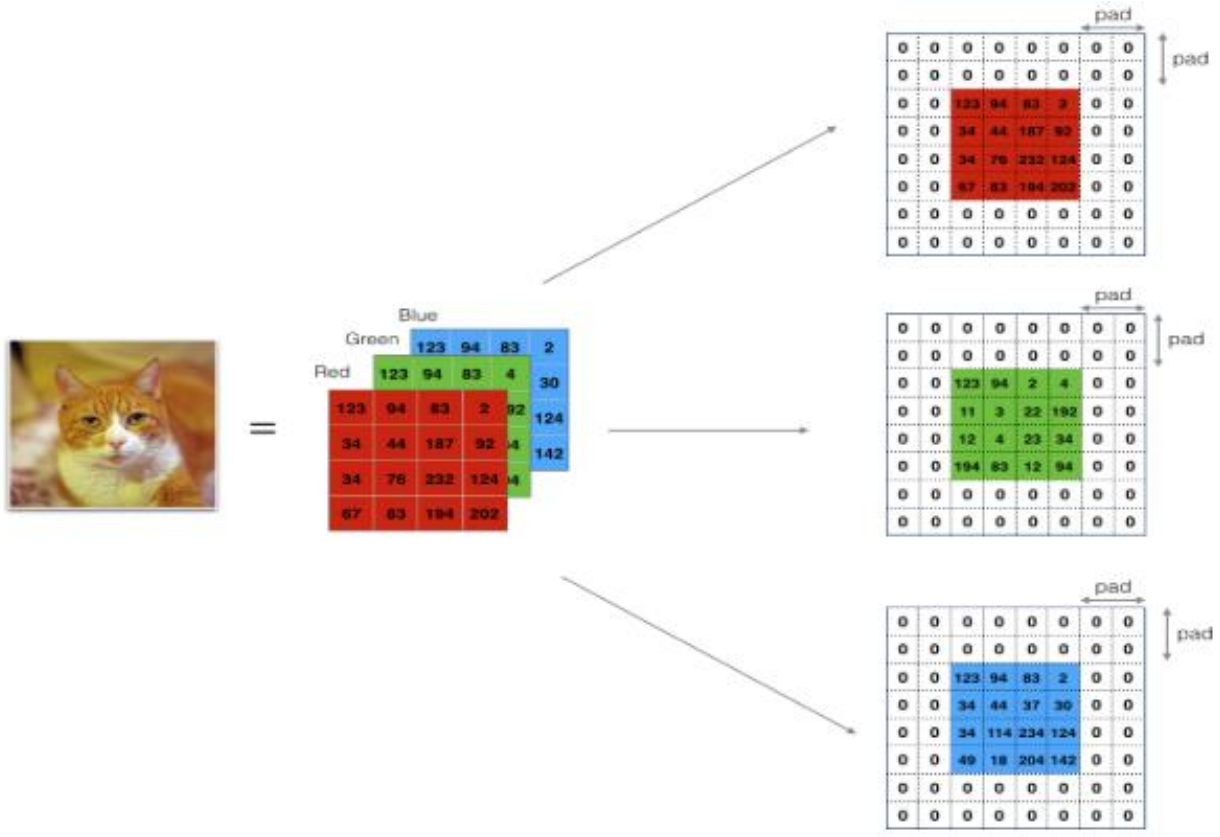
A convolution layer transforms an input volume into an output volume of different size, as shown below.



In this part, you will build every step of the convolution layer. You will first implement two helper functions: one for zero padding and the other for computing the convolution function itself.

3.1 - Zero-Padding

Zero-padding adds zeros around the border of an image:



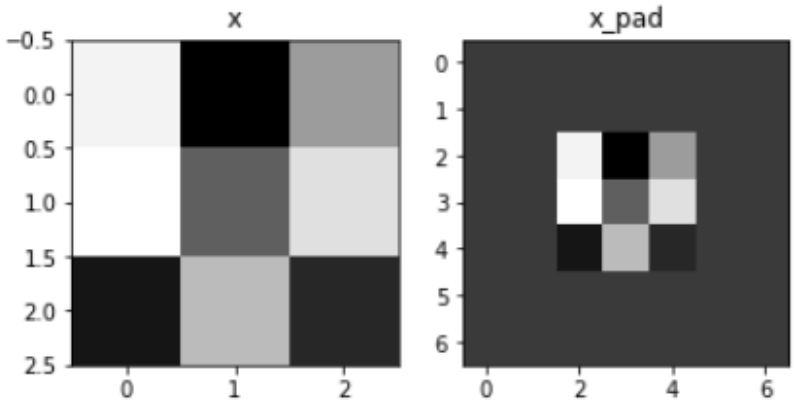
The main benefits of padding are the following:

- It allows you to use a CONV layer without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since otherwise the height/width would shrink as you go to deeper layers. An important special case is the "same" convolution, in which the height/width is exactly preserved after one layer.
- It helps us keep more of the information at the border of an image. Without padding, very few values at the next layer would be affected by pixels as the edges of an image.

```python
# GRADED FUNCTION: zero_pad

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)
    """

    X_pad = np.pad(X, ((0,0),(pad,pad),(pad,pad),(0,0)), 'constant', constant_values=0)

    return X_pad
```

```
np.random.seed(1)
x = np.random.randn(4, 3, 3, 2)
x_pad = zero_pad(x, 2)
print ("x.shape =", x.shape)
print ("x_pad.shape =", x_pad.shape)
print ("x[1,1] =", x[1,1])
print ("x_pad[1,1] =", x_pad[1,1])

fig, axarr = plt.subplots(1, 2)
axarr[0].set_title('x')
axarr[0].imshow(x[0,:,:,0])
axarr[1].set_title('x_pad')
axarr[1].imshow(x_pad[0,:,:,0])
```

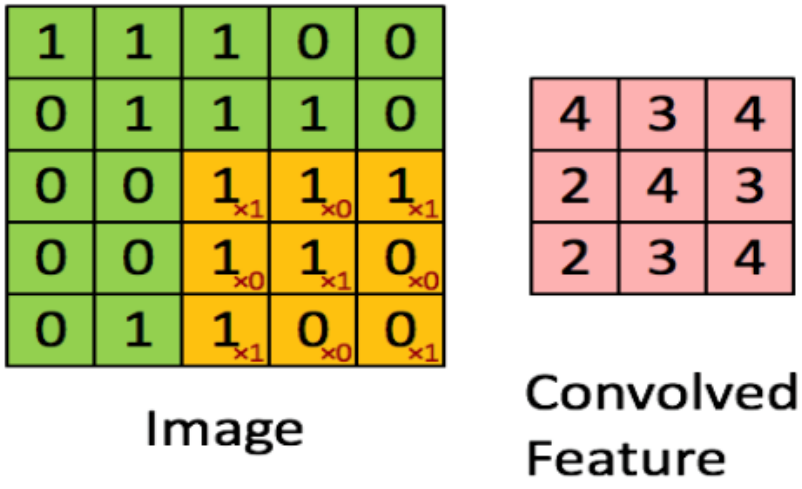
```
x.shape = (4, 3, 3, 2)
x_pad.shape = (4, 7, 7, 2)
x[1,1] = [[ 0.90085595 -0.68372786]
 [-0.12289023 -0.93576943]
 [-0.26788808  0.53035547]]
x_pad[1,1] = [[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
```



3.2 - Single step of convolution

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

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



FUNCTION: conv_single_step

Apply one filter defined by parameters W on a single slice (a_slice_prev) of the output activation of the previous layer.

Arguments:

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

Returns:

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

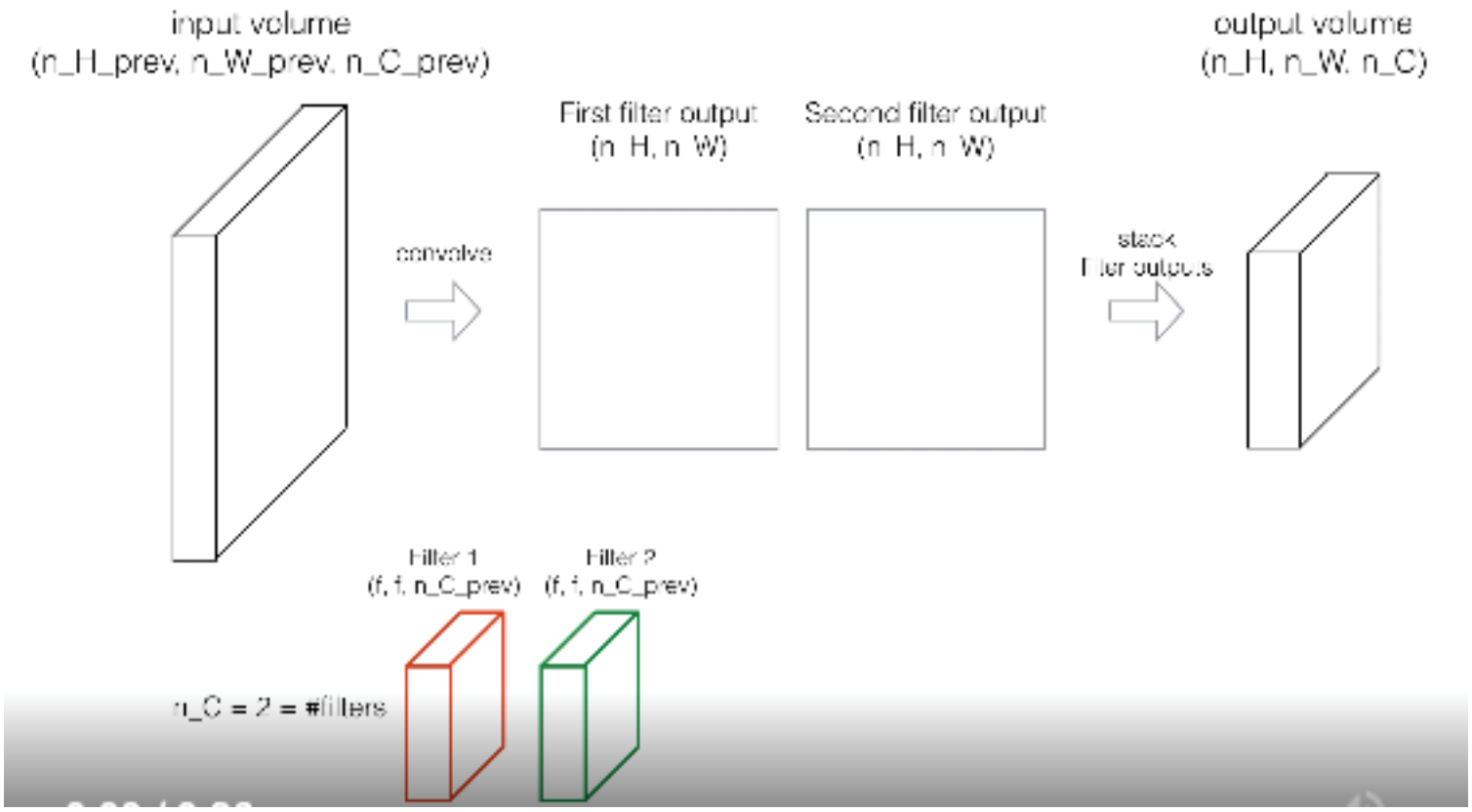
```python
# GRADED FUNCTION: conv_single_step

def conv_single_step(a_slice_prev, W, b):

    # Element-wise product between a_slice and W.
    # Do not add the bias yet.
    s = np.multiply(a_slice_prev, W)

    # Sum over all entries of the volume s.
    Z = np.sum(s)

    # Add bias b to Z. Cast b to a float() so that Z
    # results in a scalar value.
    Z = Z+np.float(b)

    return Z
```

```
np.random.seed(1)
a_slice_prev = np.random.randn(4, 4, 3)
W = np.random.randn(4, 4, 3)
b = np.random.randn(1, 1, 1)

Z = conv_single_step(a_slice_prev, W, b)
print("Z =", Z)
```

```
Z = -6.999089450680221
```

3.3 - Convolutional Neural Networks - Forward pass

- In the 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:



FUNCTION: conv_forward

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

```
def conv_forward(A_prev, W, b, hparameters):


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

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

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

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

```
    for i in range(m):                       # loop over the batch of training examples
        a_prev_pad = A_prev_pad[i]           # Select ith training example's padded activation
        for h in range(n_H):                 # loop over vertical axis of the output volume
            for w in range(n_W):             # loop over horizontal axis of the output volume
                for c in range(n_C):         # loop over channels (= #filters) of the output volume

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

                    # Use the corners to define the (3D) slice of a_prev_pad
                    a_slice_prev = a_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :]

                    # Convolve the (3D) slice with the correct filter W and bias b,
                    # to get back one output neuron. (≈1 line)
                    Z[i, h, w, c] = conv_single_step(a_slice_prev, W[:,:,:,c], b[:,:,:,c])

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

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

    return Z, cache
```

```
np.random.seed(1)
A_prev = np.random.randn(10,4,4,3)
W = np.random.randn(2,2,3,8)
b = np.random.randn(1,1,1,8)
hparameters = {"pad" : 2,
               "stride": 2}

Z, cache_conv = conv_forward(A_prev, W, b, hparameters)
print("Z's mean =", np.mean(Z))
print("Z[3,2,1] =", Z[3,2,1])
print("cache_conv[0][1][2][3] =", cache_conv[0][1][2][3])
```

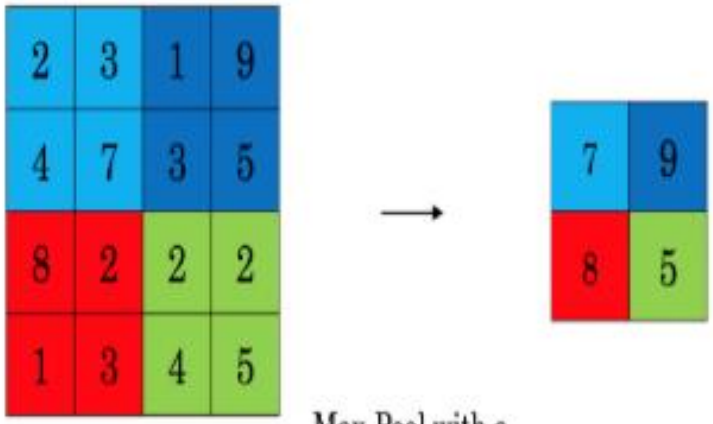
```
Z's mean = 0.0489952035288557 94
Z[3,2,1] = [-0.61490741 -6.7439236  -2.55153897  1.75698377  3.56208902  0.53036437
  5.18531798  8.75898442]
cache_conv[0][1][2][3] = [-0.20075807  0.18656139  0.41005165]
```

4 - Pooling layer

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

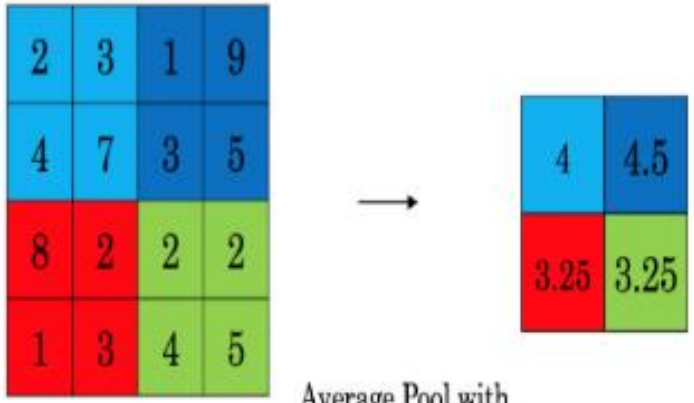
- Max-pooling layer: slides an (f,ff,f) window over the input and stores the max value of the window in the output.
- Average-pooling layer: slides an (f,ff,f) window over the input and stores the average value of the window in the output.

Max Pool



Max-Pool with a 2 by 2 filter and stride 2.

Average Pool



Average Pool with a 2 by 2 filter and stride 2.

4.1 - Forward Pooling

- Now, you are going to implement MAX-POOL and AVG-POOL, in the same function.
- As there's no padding, the formulas binding the output shape of the pooling to the input shape is:

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

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

$$n_C = n_{C_{prev}}$$

FUNCTION: pool_forward

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

```
def pool_forward(A_prev, hparameters, mode = "max"):

    # 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
            for w in range(n_W):             # loop on the horizontal axis of the output volume
                for c in range (n_C): # loop over the channels of the output volume
                    # Find the corners of the current "slice" (≈4 lines)
                    vert_start = h*stride
                    vert_end = vert_start+f
                    horiz_start = w*stride
                    horiz_end = horiz_start+f

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

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

    # 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
```

```
np.random.seed(1)
A_prev = np.random.randn(2, 4, 4, 3)
hparameters = {"stride" : 2, "f": 3}

A, cache = pool_forward(A_prev, hparameters)
print("mode = max")
print("A =", A)
print()
A, cache = pool_forward(A_prev, hparameters, mode = "average")
print("mode = average")
print("A =", A)
```

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


 [[[1.13162939 1.51981682 2.18557541]]]]

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


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

5.1 - Convolutional layer backward pass

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

Computing dA:

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

$$dA+ = \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw}$$

Computing dW:

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

$$dW_c += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw}$$

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

Computing db:

- This is the formula for computing dbdb with respect to the cost for a certain filter WcWc:

$$db = \sum_{h} \sum_{w} dZ_{hw}$$

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

Function: conv_backward

Implement the backward propagation for a convolution function

Arguments:

- dZ -- gradient of the cost with respect to the output of the conv layer (Z), numpy array of shape (m, n_H, n_W, n_C)
- cache -- cache of values needed for the conv_backward(), output of conv_forward()

Returns:

- dA_prev -- gradient of the cost with respect to the input of the conv layer (A_prev),
- numpy array of shape (m, n_H_prev, n_W_prev, n_C_prev)
- dW -- gradient of the cost with respect to the weights of the conv layer (W)
- numpy array of shape (f, f, n_C_prev, n_C)
- db -- gradient of the cost with respect to the biases of the conv layer (b) numpy array of shape (1, 1, 1, n_C)

```python
def conv_backward(dZ, cache):

    # Retrieve information from "cache"
    (A_prev, W, b, hparameters) = cache

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

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

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

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

    # Initialize dA_prev, dW, db with the correct shapes
    dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))
    dW = np.zeros((f, f, n_C_prev, n_C))
    db = np.zeros((1, 1, 1, n_C))

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

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

        # select ith training example from A_prev_pad and dA_prev_pad
        a_prev_pad = A_prev_pad[i]
        da_prev_pad = dA_prev_pad[i]

        for h in range(n_H):           # loop over vertical axis of the output volume
            for w in range(n_W):       # loop over horizontal axis of the output volume
                for c in range(n_C): # loop over the channels of the output volume

                    # Find the corners of the current "slice"
                    vert_start = h*stride
                    vert_end = vert_start+f
                    horiz_start = w*stride
                    horiz_end = horiz_start+f

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

                    # Update gradients for the window and the filter's parameters using
                    # the code formulas given above
                    da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] +=
                                            W[:,:,:,c] * dZ[i, h, w, c]
                    dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
                    db[:,:,:,c] += dZ[i, h, w, c]

        # Set the ith training example's dA_prev to the unpaded da_prev_pad
        # (Hint: use X[pad:-pad, pad:-pad, :])
        dA_prev[i, :, :, :] = da_prev_pad[pad:-pad,pad:-pad,:]

    # Making sure your output shape is correct
    assert(dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))

    return dA_prev, dW, db
```

```
np.random.seed(1)
dA, dW, db = conv_backward(Z, cache_conv)
print("dA_mean =", np.mean(dA))
print("dW_mean =", np.mean(dW))
print("db_mean =", np.mean(db))
```

```
dA_mean = 1.4524377775388075
dW_mean = 1.7269914583139097
db_mean = 7.839232564616838
```

5.2 Max pooling - backward pass

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

$$X = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \quad \rightarrow \quad M = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$$

- As you can see, this function creates a "mask" matrix which keeps track of where the maximum of the matrix is. True (1) indicates the position of the maximum in X, the other entries are False (0).

Function: create_mask_from_window

Creates a mask from an input matrix x, to identify the max entry of x.

Arguments:

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

Returns:

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

```
def create_mask_from_window(x):

    mask = (x==np.max(x))

    return mask
```

```
np.random.seed(1)
x = np.random.randn(2,3)
mask = create_mask_from_window(x)
print('x = ', x)
print("mask = ", mask)
```

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

5.3 Average pooling - backward pass

- In average pooling, every element of the input window has equal influence on the output. So to implement backprop, you will now implement a helper function that reflects this.
- For example if we did average pooling in the forward pass using a 2x2 filter, then the mask you'll use for the backward pass will look like:

$$dZ = 1 \quad \rightarrow \quad dZ = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}$$

Function: distribute_value

Distributes the input value in the matrix of dimension shape

Arguments:

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

Returns:

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

```
def distribute_value(dz, shape):

    # Retrieve dimensions from shape (≈1 line)
    (n_H, n_W) = shape

    # Compute the value to distribute on the matrix (≈1 line)
    average = 1/(n_H*n_W)

    # Create a matrix where every entry is the "average" value (≈1 line)
    a = np.ones((n_H, n_W))*dz*average

    return a
```

```
a = distribute_value(2, (2,2))
print('distributed value =', a)
```

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

5.4 Putting it together: Pooling backward

Function: pool_backward

Implements the backward pass of the pooling layer

Arguments:

- dA -- gradient of cost with respect to the output of the pooling layer, same shape as A
- cache -- cache output from the forward pass of the pooling layer, contains the layer's input and hparameters
- mode -- the pooling mode you would like to use, defined as a string ("max" or "average")

Returns:

- dA_prev -- gradient of cost with respect to the input of the pooling layer, same shape as A_prev

```python
def pool_backward(dA, cache, mode = "max"):

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

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

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

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

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

    # select training example from A_prev (≈1 line)
    a_prev = A_prev[i]

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

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

```
                    # Compute the backward propagation in both modes.
                    if mode == "max":

                        # Use the corners and "c" to define the current slice from a_prev
                        # (≈1 line)
                        a_prev_slice = a_prev[vert_start:vert_end,horiz_start:horiz_end,c]
                        # Create the mask from a_prev_slice (≈1 line)
                        mask = create_mask_from_window(a_prev_slice)
                        # Set dA_prev to be dA_prev + (the mask multiplied by the correct
                        # entry of dA) (≈1 line)
                        dA_prev[i, vert_start: vert_end, horiz_start: horiz_end, c] +=
                                                (mask * a_prev_slice)

                    elif mode == "average":

                        # Get the value a from dA (≈1 line)
                        da = np.sum(a_prev[horiz_start:horiz_end, vert_start:vert_end, c])
                        # Define the shape of the filter as fxf (≈1 line)
                        shape = (f,f)
                        # Distribute it to get the correct slice of dA_prev. i.e.
                        # Add the distributed value of da. (≈1 line)
                        dA_prev[i, vert_start: vert_end, horiz_start: horiz_end, c] +=
                                                 distribute_value(da, shape)

    # Making sure your output shape is correct
    assert(dA_prev.shape == A_prev.shape)

    return dA_prev
```

```
np.random.seed(1)
A_prev = np.random.randn(5, 5, 3, 2)
hparameters = {"stride" : 1, "f": 2}
A, cache = pool_forward(A_prev, hparameters)
dA = np.random.randn(5, 4, 2, 2)

dA_prev = pool_backward(dA, cache, mode = "max")
print("mode = max")
print('mean of dA = ', np.mean(dA))
print('dA_prev[1,1] = ', dA_prev[1,1])
print()
dA_prev = pool_backward(dA, cache, mode = "average")
print("mode = average")
print('mean of dA = ', np.mean(dA))
print('dA_prev[1,1] = ', dA_prev[1,1])
```

```
mode = max
mean of dA =  0.14571390272918056
dA_prev[1,1] =  [[0.         0.        ]
 [6.63920871 1.48408832]
 [0.         0.        ]]

mode = average
mean of dA =  0.14571390272918056
dA_prev[1,1] =  [[-0.18169846 -0.31723859]
 [ 0.20725933  0.52015979]
 [ 0.38895779  0.83739837]]
```

References

- https://github.com/Kulbear/deep-learning-coursera/blob/master/Convolutional%20Neural%20Networks/Convolution%20model%20-%20Step%20by%20Step%20-%20v1.ipynb