机器学习系列（5）

卷积神经网络CNN之--原理及Python实现

卷积神经网络的原理解释：

- 计算机视觉
- 边缘检测
- Padding
- 卷积步长
- 单卷积层
- 池化层
- 卷积神经网络示例
- 为什么选择CNN

Python实现：

- 见文章内容

申明

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

一、卷积神经网络(CNN)的原理和步骤

1.计算机视觉：

- 计算机视觉是一门研究如何使机器学会“看”的学科，让机器能够从图像或多维图像中感知。其应用广泛，比如人脸识别、自动驾驶、风格转换等。在本系列先前的个人笔记中，我们将深度神经网络应用于识别猫，虽然取得了不错的效果，但是训练分类器的时间较长，本次我们将介绍卷积神经网络（CNN）,这种流行的网络结构广泛应用于计算机视觉领域，并取得了很好的效果。

2.边缘检测：

- 在本系列之前的识别猫的笔记中，我们提到了应用深度全连接的神经网络进行分类，当图像像素比较高时，将会遇到非常大的麻烦。举例说明，如$Figure1$所示，若图片大小为$1000*1000*3$，第一层隐藏层神经元个数为1000，那么$W^{[1]}$的维度将是$1000*3M = 30$亿，与数据量相比大很多，很容易发生过拟合，而且巨大的参数数量，将导致对内存的需求急剧增大，社图片的边长为n,那么这种需求是0（n^2）级的，这显然不适合于大规模计算。卷积神经网络通过卷积运算，大大减小了深度学习的计算量，而且在效果上表现更突出。

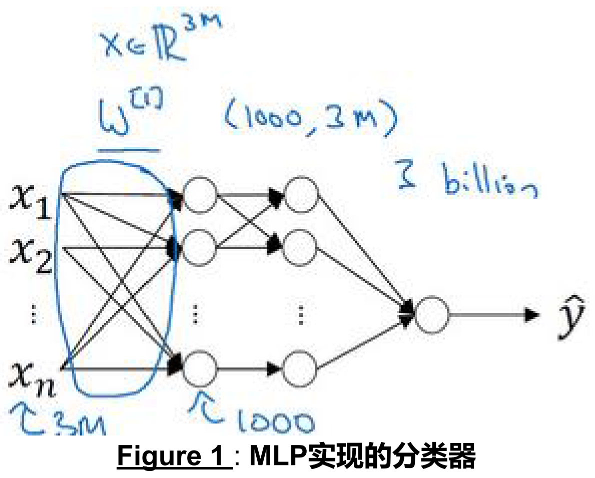


Figure 1 : **MLP实现的分类器**

- 边缘检测是卷积神经网络的一个概念，它的作用是提取出图片中物体的边缘信息（可以认为是一种滤波），进而网络可以根据提取到的特征进行分类。例如$Figure2$所示，边缘检测通过两个边缘检测器（滤波器，也叫核，即kernel），分别检测出了图片的垂直边缘和水平边缘，这就是边缘检测的简单示例。那么边缘检测是如何作用的又是如何实现的，举个例子来说，见$Figure3$,假设图片中像素值越大的部位越亮，反之越暗，那么经过一个垂直检测器（figure 3中）后，过滤出来的新图片(figure 3右)检测出了原来图片（figure 3左）的明暗交界边缘。事实上改变过滤器中的值的大小和分布可以构造出形式各样的检测器。

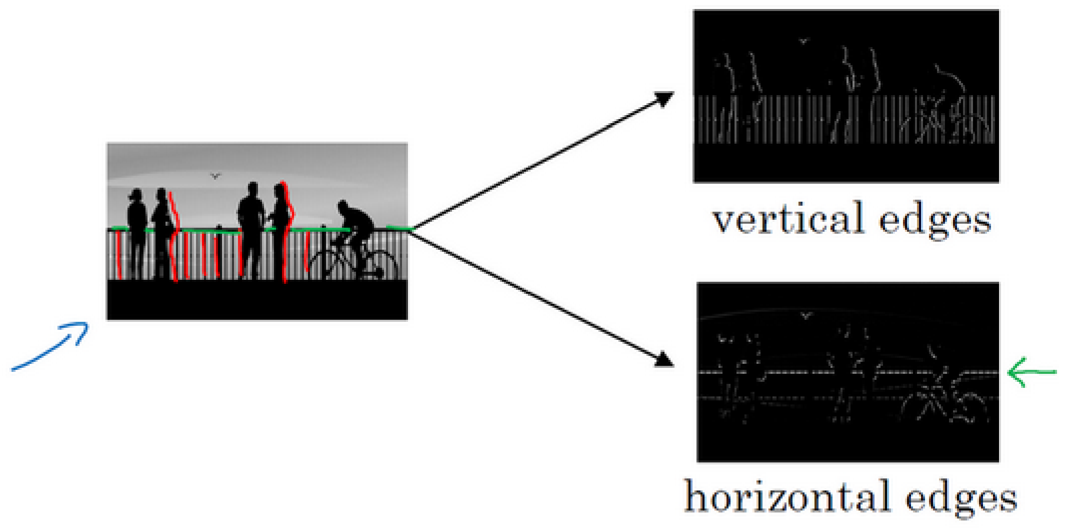


Figure 2 : **边缘特征**

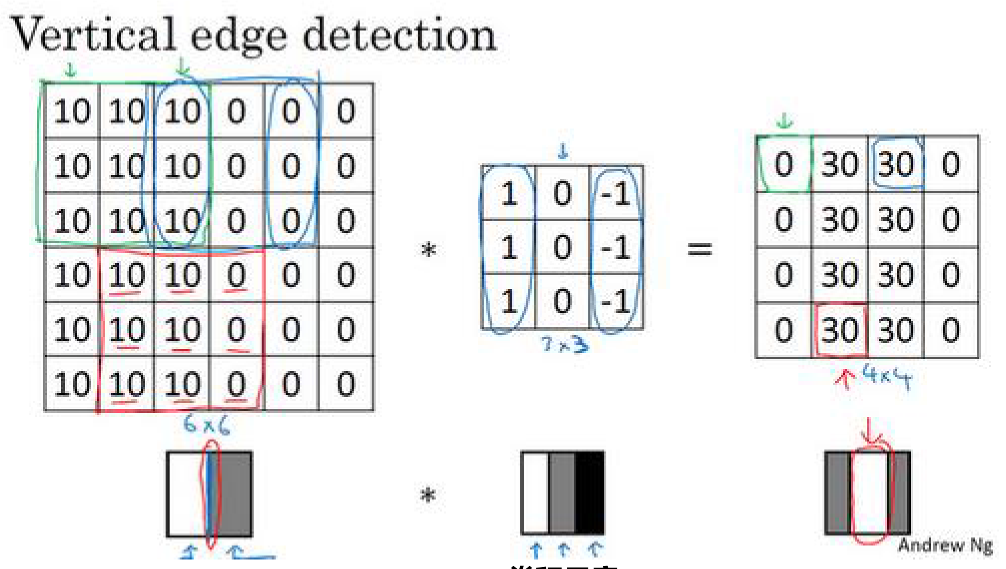


Figure 3 : **卷积示意**

- 那么这个过滤的操作是什么呢？其实就是“卷积运算”（和数学上定义的卷积稍有不同，在数学定义中卷积运算在对应元素相乘之前，会有一个对Kernel先反向，在CNN中我们不进行反向，原因是其不影响对边界特征的提取，关于此点，可参见个人专题笔记《机器学习中的数学（1）--卷积运算》），以图$Figure3$说明这里“卷积”的含义，图中左边是一个图片的像素矩阵，中间是过滤器，右边是新检测出的边缘图片，左边有三种颜色标记的框，经过“卷积”运算后，对应在右边相应颜色的输出位上，看出来了吧，这里的卷积指的是矩阵的元素乘积求和，然后移动Kernel再次求和，最后拼成新的矩阵。所以改变过滤器，可改变输出的新图形（称为特征）。在卷积神经网络中，我们把过滤器中的值当作参数让网络去学习，这样可以自动给出具体的值，并且可以检测任何角度的边缘特征，通过这种计算方式，我们提取特征的方式从全连接变成了卷积，参数数量减小为(过滤器的大小$J*K$*过滤器的个数L,时间复杂度降为$O(J*K*L)$，不再是输入维数的平方量级，这大大减小了深度学习的计算量。

Padding:

- Padding,顾名思义为填充。前面提到，我们在做特征提取时，用到的是卷积运算，由于其运算特点，容易发现其对整张图片的边沿部分的数值在计算中使用较少，而在其它区域，其数值不止使用较多，这意味这对于边沿部分提取到的特征较少，而且可以发现，每次进行特征提取后，图像的大小减小了，但你可能不想让某次的图像减小。为了解决这两个问题，CNN引入了Padding思想，即在卷积运算之前，图像外沿先填充新的像素点，这样边缘信息能被更好的提取，新出来的特征图像大小也会更大，通过合适选择填充像素圈数（设为p）和Kernel的大小，可以控制输出特征图像的大小，常见的有等维度输出，半维度输出等。如图$Figure4$,对一个$6*6$大小的图像，经过$3*3$大小的Kernel,其输出图像维度为$4*4$，而当我们在外圈填充一层像素之后，图像先变成了$8*8$,然后经过$3*3$的kernel，输出图像依然是$6*6$，这就是padding的作用。

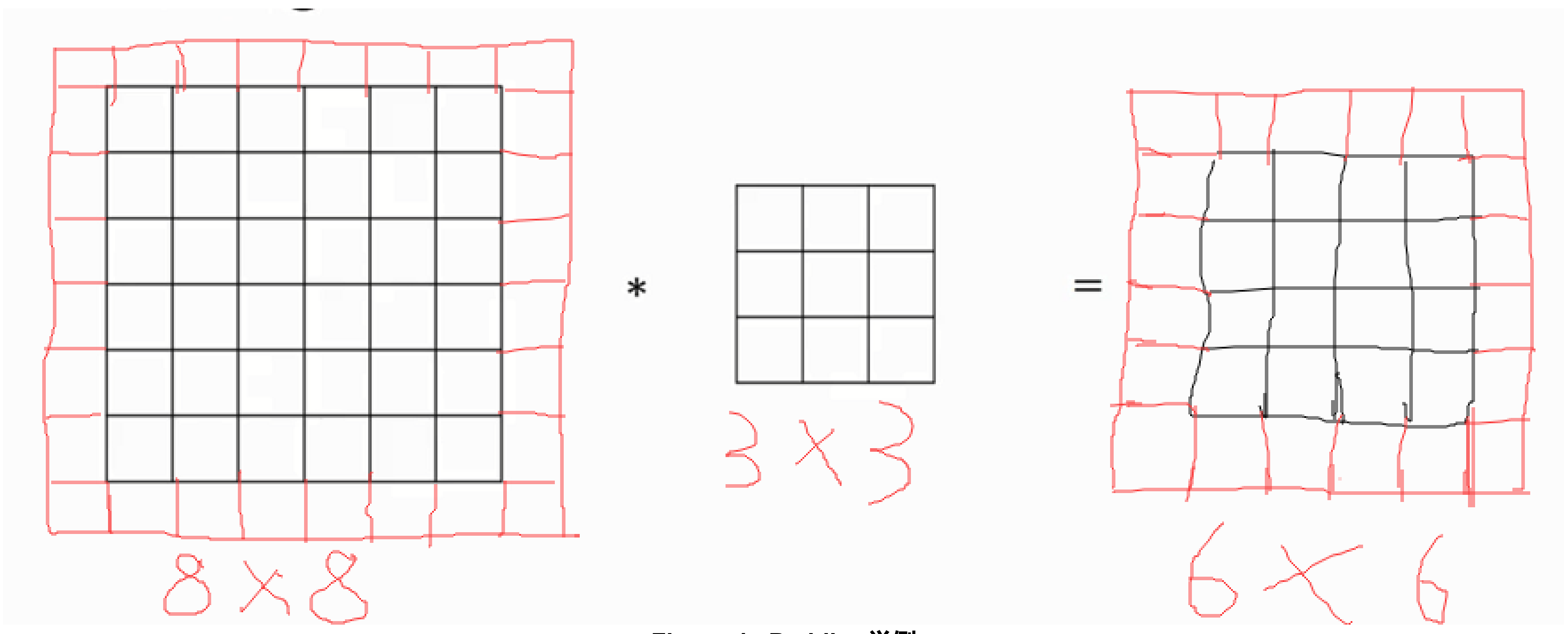


Figure 4 : **Padding举例**

卷积步长:

- 在以上的描述中，Kernel每次移动的步长都是1，如果把每次移动的步长定义为卷积步长（Stride，简略为S表示），那么提取出来的特征图像与步长为1时大小不一样，例如当S为2时,输入输出大小的的关系如图$Figure5$。

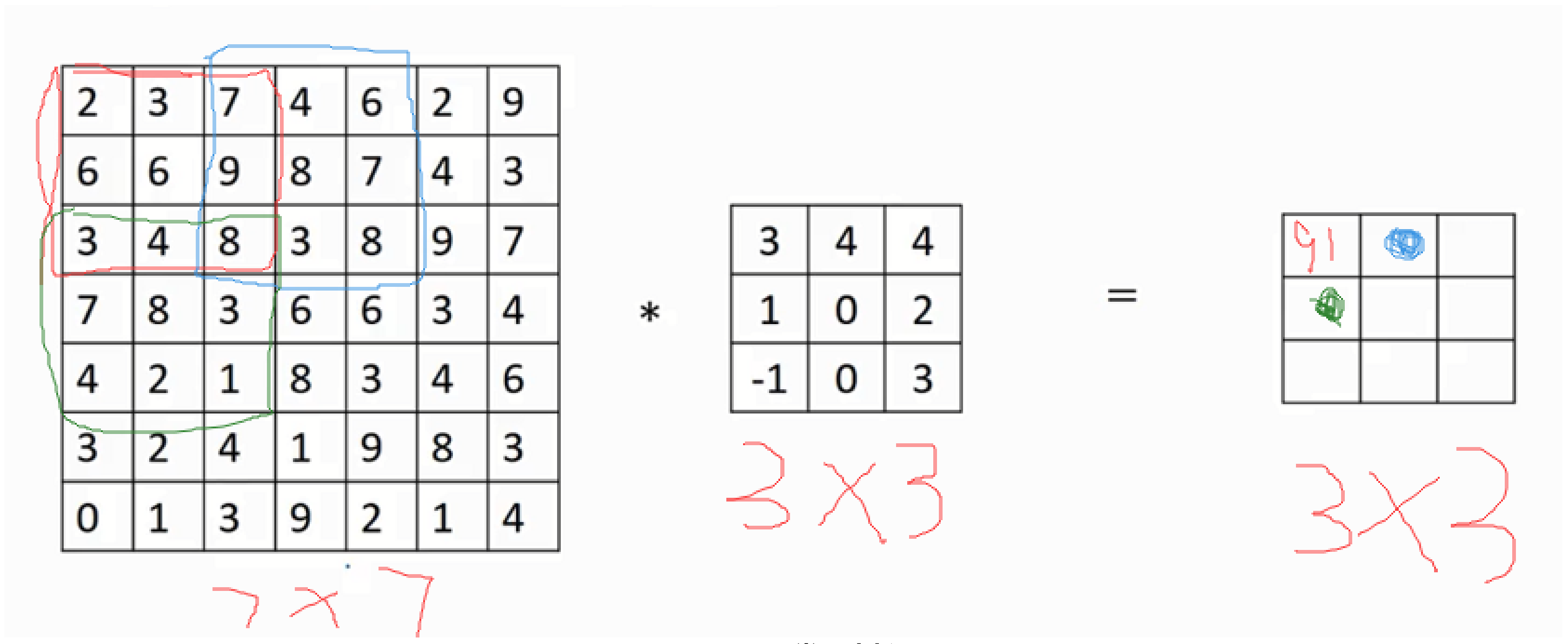


Figure 5 : **卷积步长**

- 一般的，容易得到特征矩阵的大小与原图像和Kernel的关系可用下式表示（假设原图像大小为$n*n$,Kernel的大小为$f*f$,Padding的大小为P,那么输出的大小为$\frac{n+2p-f}{2}+1*\frac{n+2p-f}{2}+1$. **三维卷积**： - 三维卷积即是一维卷积和二维卷积的推广，事实上N维空间上的卷积形式是一样的，这里特地介绍三

维卷积是为了描述RGB图像的边缘提取，黑白图像的维度可以用二维平面表示，而彩色图像需要用三维空间来表示，因为彩色图像有三个通道，所以在做卷积运算时，kernel的维度为从二维增加到三维，第三个维度的大小与彩色图像的必须通道数相同，$Figure6$是三维卷积的演示，注意输出不是$4*4*3$，而是$4*4*1$。

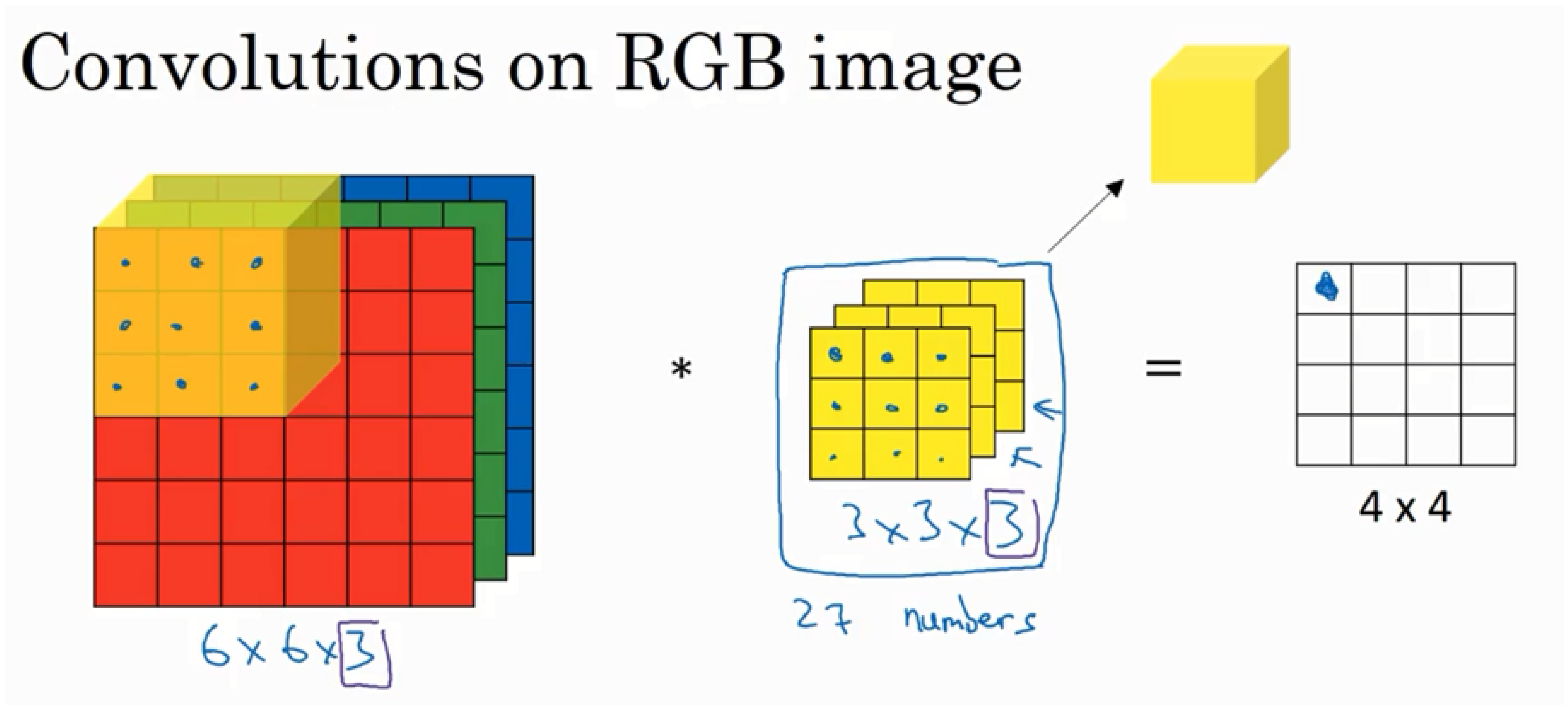


Figure 6 : **三维卷积示意**

单卷积层:

- 以上是卷积神经网络的基础描述，下面介绍单层卷积神经网络的具体实现。以图$Figure7$为例，单层神经网络在三维卷积的基础上增加了一个Kernel个数的维度，显然输出最后一个维度也自然成了三维Kernel的个数K，和全连接神经网络一样，我们也加入了激活函数。单卷积层相当于在提取图像的特征。从这里可知，不管输入图像多大，参数的数量只由Kernel的大小和个数确定，因此参数数量远远小于全连接神经网络。

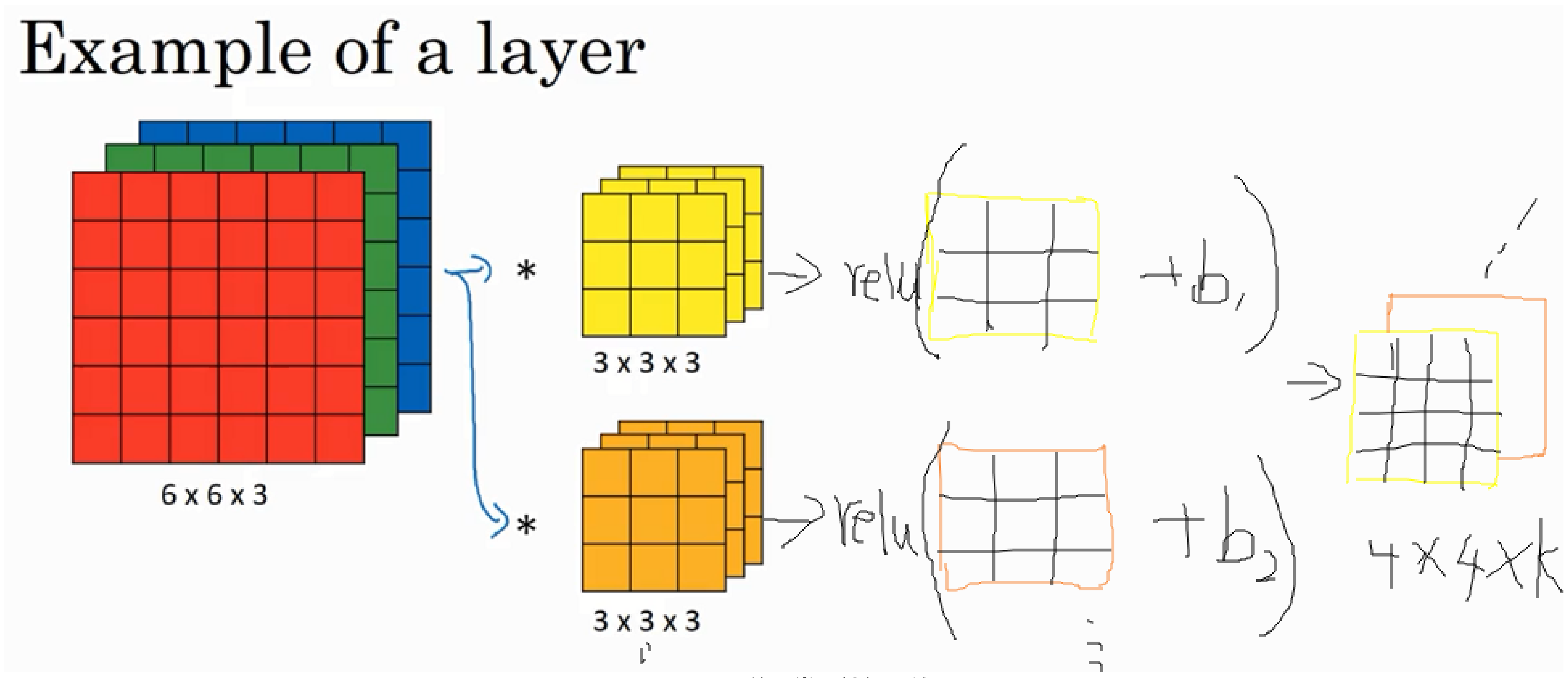


Figure 7 : **单层卷积神经网络**

池化层pooling:

- 池化层是一层对图像进行另一种处理的层，直观来说，这一层的作用是提取区域特征，从实际效果上看，这种方式确实提升了CNN的"看"的能力。$Figure8$是池化的一个示例，此例中使用的是最大池化（Max pooling），即取对应区域的最大值，此外常用的还有平均值池化等。池化层也有对应的Kernel，步长等参数，但是不同于卷积层的Kernel，这些参数并不需要由网络学习得到，我们给定Kernel的大小和步长后，计算是确定的（卷积层的Kernel参数还有每个Kernel里的数字，池化层没有），所以池化层不会额外增加新的参数，且由于其能缩小图像尺寸，反而能加快训练。当输入图像是三维的时候，kernel也对应增加为3维，同样输出也增加为3维（不同于卷积运算，因为在卷积层，kernel的个数是可以改变的，这增加输出的一个维度，而在池化层，kernel的个数确定是1，所以为了能使池化的输出作为卷积的输入，取最大值时，不会在三维空间取，而是在二维平面取完后，在第三个维度拼接）。

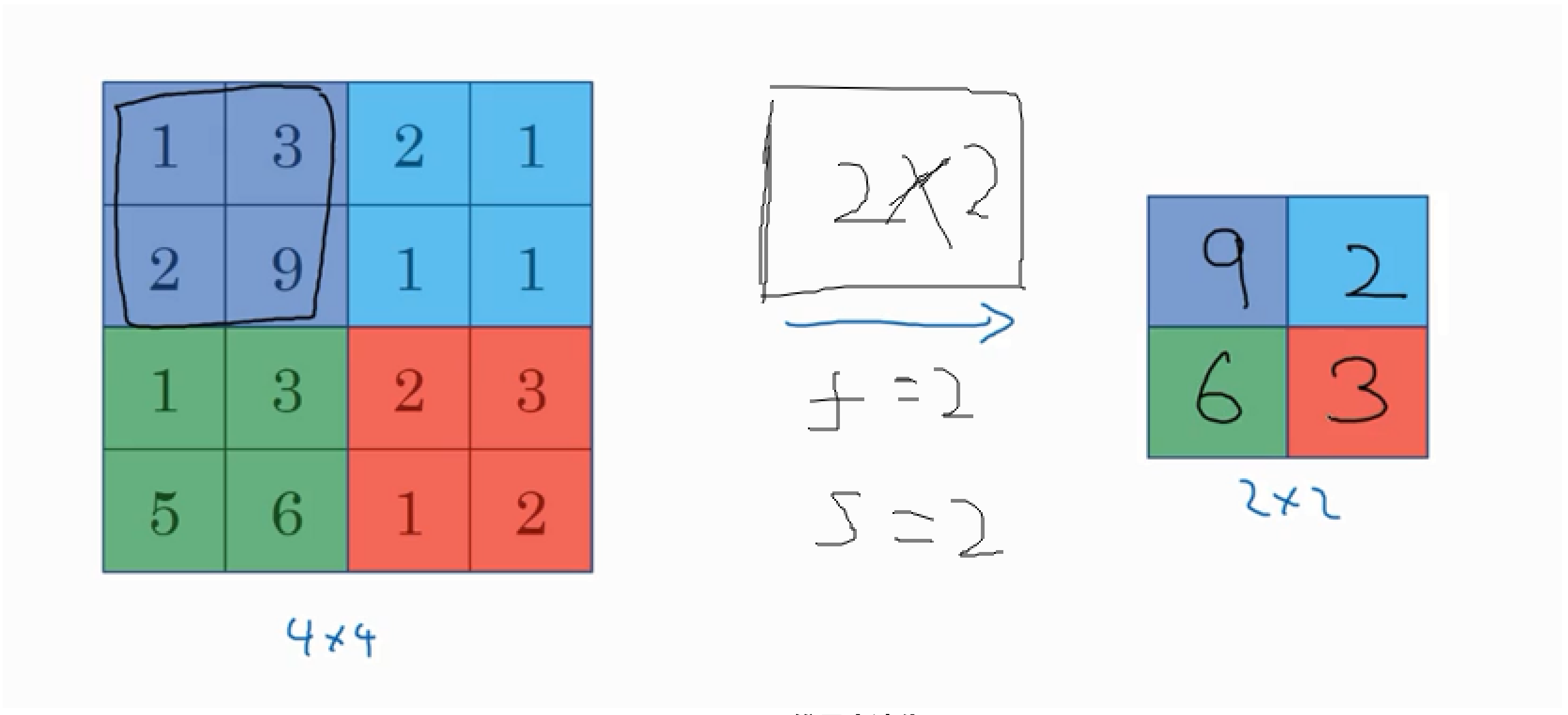


Figure 8 : **二维最大池化**

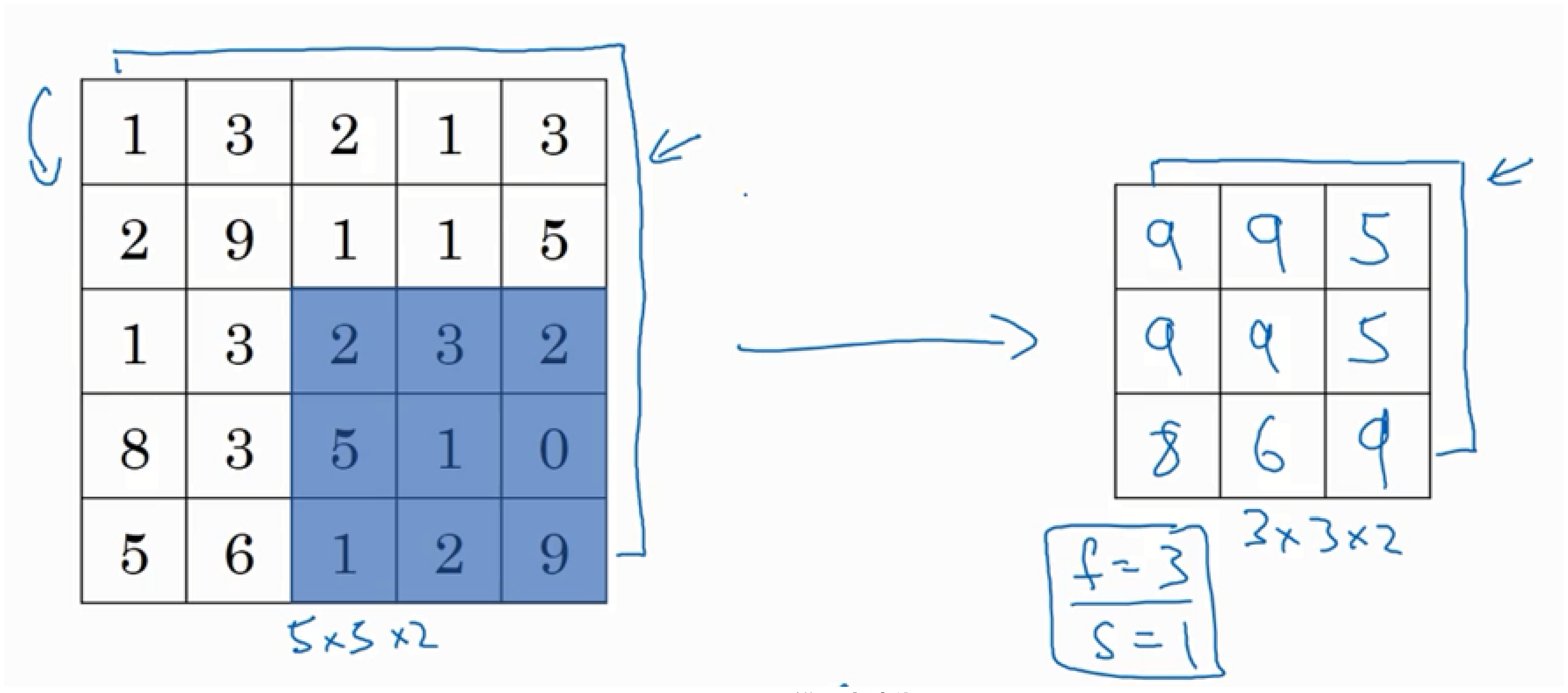


Figure 9 : **三维最大池化**

卷积神经网络示例:

- 卷积神经网络由输入层、卷积层、池化层、全连接层，输出层复合而成，除了输入输出层固定为1层外，其它层均不固定，以下就是一个经典的卷积网络LeNet-5的例子，如$Figure 10$所示。

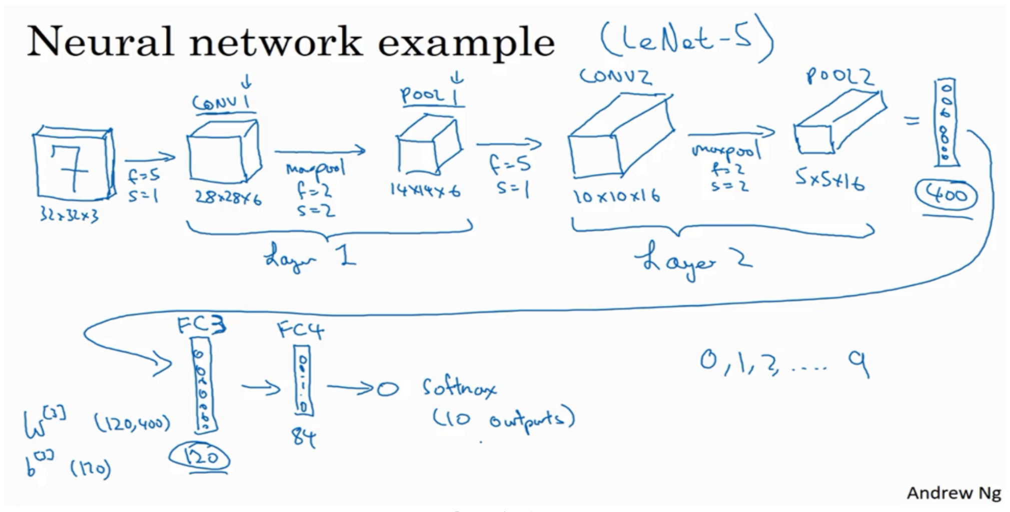


Figure 10 : **卷积神经网络LeNet-5**

为什么选择CNN:

- 参数共享机制：在卷积神经网络中，参数是kernel的值，这些值对所有的区域都是一样的，因此参数的数量可以做到比较少，能有效的防止过拟合。
- 稀疏连接机制：在卷积网络的输出中，每一个“小格子”的输出只与输入图像和其对应的部分有关，而已其它部分无关，计算量较小。
- 高级特征提取：不断使用卷积池化的过程相当于不断提取更高级的特征。

二、卷积神经网络的python实现

Welcome to Course 4's first assignment! In this assignment, you will implement convolutional (CONV) and pooling (POOL) layers in numpy, including both forward propagation and (optionally) backward propagation.

Notation:

- Superscript $[l]$ denotes an object of the l^{th} layer.
    - Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.
- Superscript (i) denotes an object from the i^{th} example.
    - Example: $x^{(i)}$ is the i^{th} training example input.
- Lowerscript i denotes the i^{th} entry of a vector.
    - Example: $a^{[l]}_i$ denotes the i^{th} entry of the activations in layer l, assuming this is a fully connected (FC) layer.
- n_H, n_W and n_C denote respectively the height, width and number of channels of a given layer. If you want to reference a specific layer l, you can also write $n_H^{[l]}$, $n_W^{[l]}$, $n_C^{[l]}$.
- $n_{H_{prev}}$, $n_{W_{prev}}$ and $n_{C_{prev}}$ denote respectively the height, width and number of channels of the previous layer. If referencing a specific layer l, this could also be denoted $n_H^{[l-1]}$, $n_W^{[l-1]}$, $n_C^{[l-1]}$.

We assume that you are already familiar with `numpy` and/or have completed the previous courses of the specialization. Let's get started!

1 - Packages

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

- numpy (www.numpy.org) is the fundamental package for scientific computing with Python.
- matplotlib (http://matplotlib.org) 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.

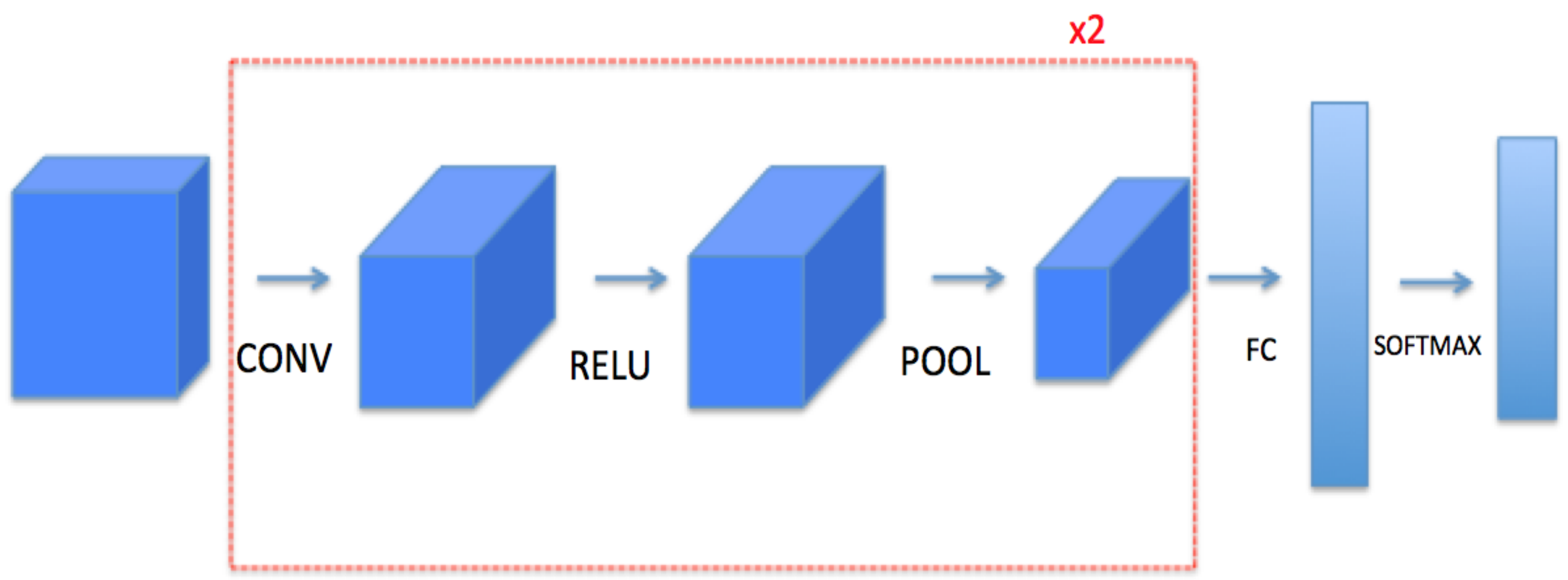
In [1]:
```
import numpy as np
import h5py
import matplotlib.pyplot as plt

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

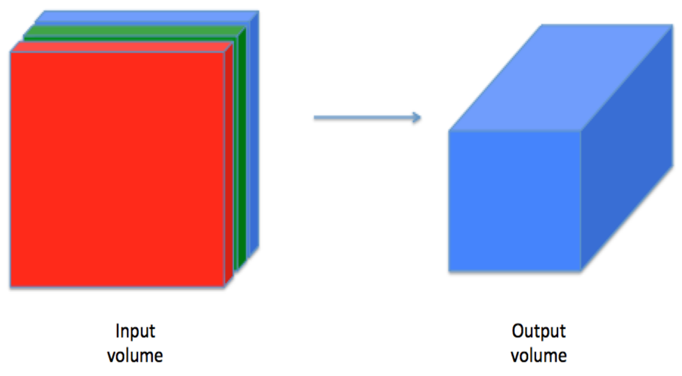
This notebook will ask you to implement these functions from scratch in `numpy`. In the next notebook, you will use the TensorFlow equivalents of these functions to build the following model:



Note that for every forward function, there is its corresponding backward equivalent. Hence, at every step of your forward module you will store some parameters in a cache. These parameters are used to compute gradients during backpropagation.

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:

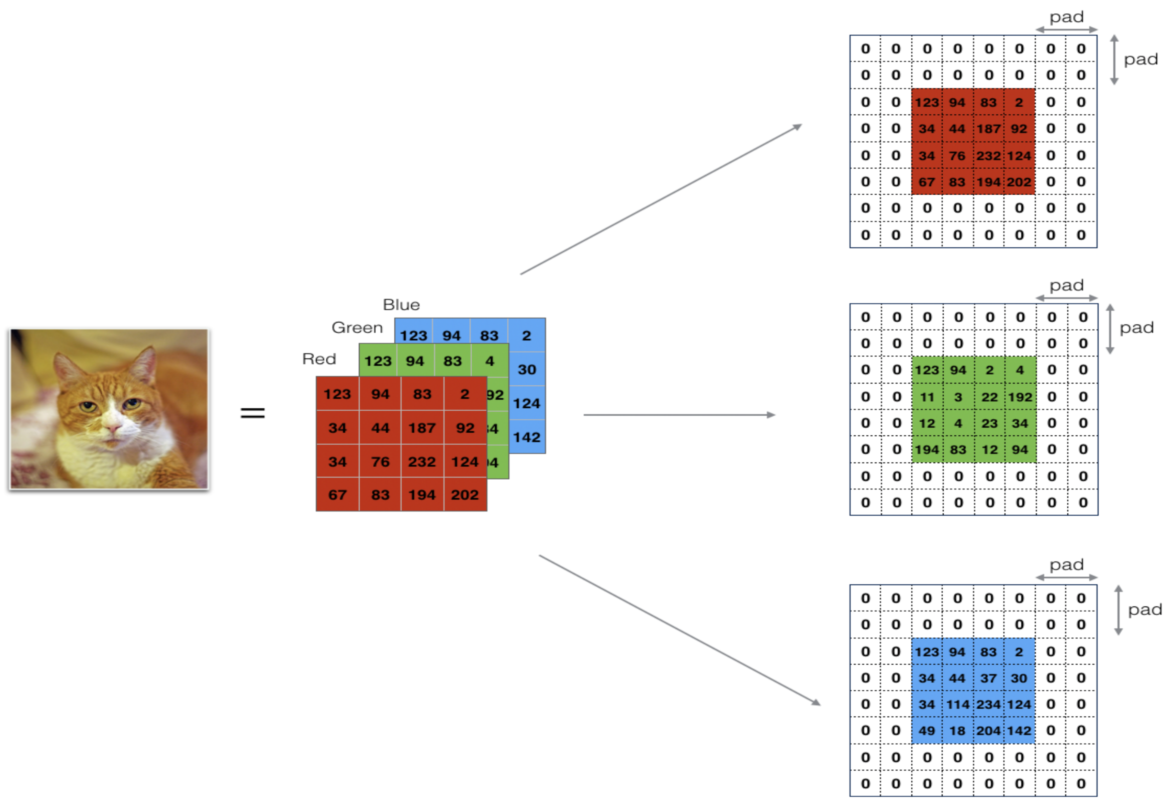


Figure 1 : **Zero-Padding**
Image (3 channels, RGB) with a padding of 2.

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.

Exercise: Implement the following function, which pads all the images of a batch of examples X with zeros. Use np.pad (https://docs.scipy.org/doc/numpy/reference/generated/numpy.pad.html). Note if you want to pad the array "a" of shape $(5, 5, 5, 5, 5)$ with $\text{pad} = 1$ for the 2nd dimension, $\text{pad} = 3$ for the 4th dimension and $\text{pad} = 0$ for the rest, you would do:

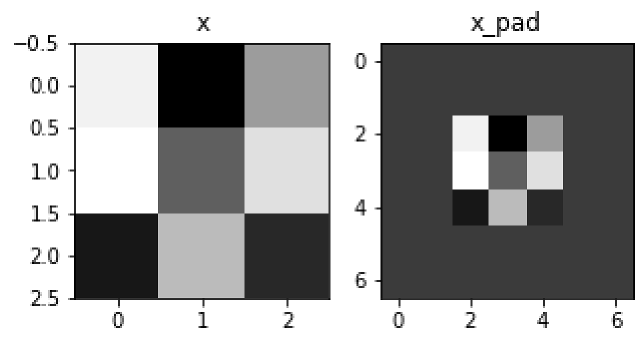
$$a = \text{np.pad}(a, ((0,0), (1,1), (0,0), (3,3), (0,0)), '\text{constant}', \text{constant\_values} = (..,..))$$

In [2]:
```
# 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)
    """

    ### START CODE HERE ### (≈ 1 line)
    X_pad = np.pad(X, ((0, 0),(pad, pad),(pad, pad),(0, 0)), 'constant', constant_values=0)
    ### END CODE HERE ###

    return X_pad
```

In [3]:
```
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.]]
```

Out[3]:
```
<matplotlib.image.AxesImage at 0x7fef958ebc50>
```



Expected Output:

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)

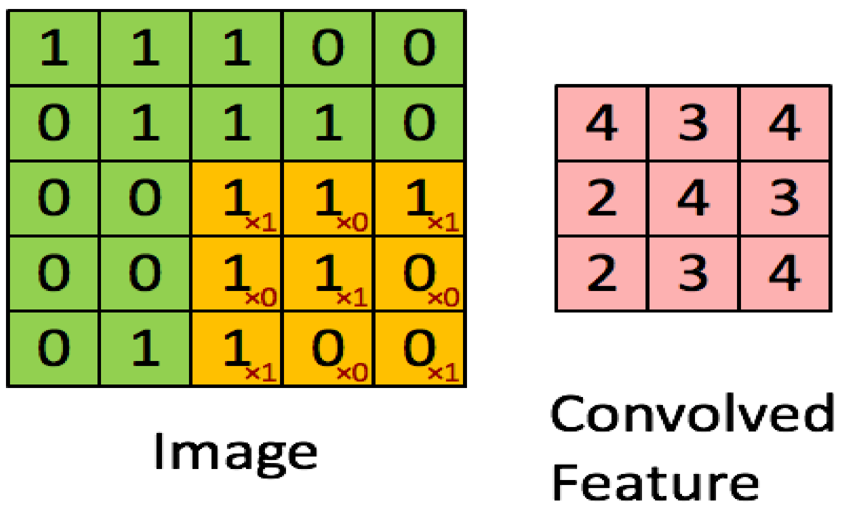


Figure 2 : **Convolution operation**
with a filter of 2x2 and a stride of 1 (stride = amount you move the window each time you slide)

In a computer vision application, each value in the matrix on the left corresponds to a single pixel value, and we convolve a 3x3 filter with the image by multiplying its values element-wise with the original matrix, then summing them up. In this first step of the exercise, you will implement a single step of convolution, corresponding to applying a filter to just one of the positions to get a single real-valued output.

Later in this notebook, you'll apply this function to multiple positions of the input to implement the full convolutional operation.

Exercise: Implement conv_single_step(). Hint (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.sum.html).

In [4]:
```
# GRADED FUNCTION: conv_single_step

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, result of convolving the sliding window (W, b) on a slice x of the input data
    """

    ### START CODE HERE ### (≈ 2 lines of code)
    # Element-wise product between a_slice and W. Add bias.
    s = np.multiply(a_slice_prev, W) + b
    # Sum over all entries of the volume s
    Z = np.sum(s)
    ### END CODE HERE ###

    return Z
```

In [5]:
```
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)
```

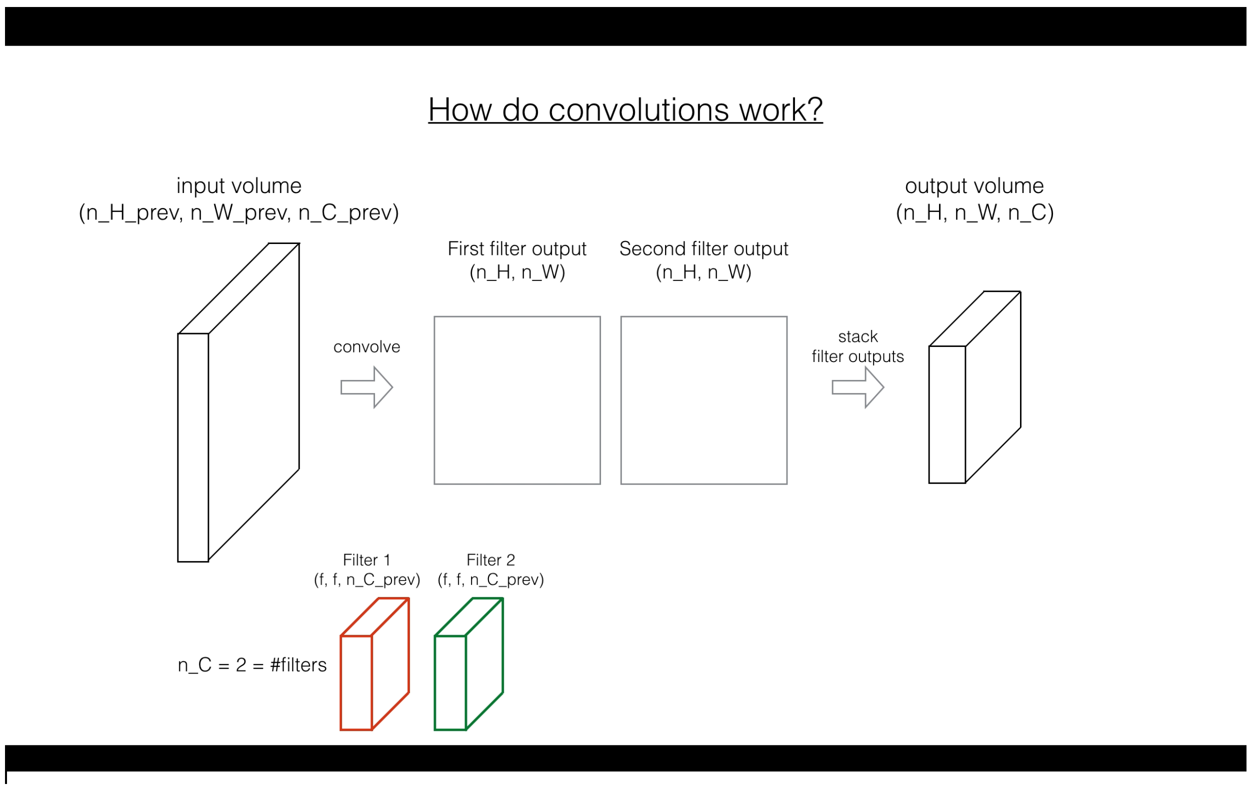
```
Z = -23.1602122025
```

Expected Output:

Z	-23.1602122025

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:



Exercise: Implement the function below to convolve the filters W on an input activation A_prev. This function takes as input A_prev, the activations output by the previous layer (for a batch of m inputs), F filters/weights denoted by W, and a bias vector denoted by b, where each filter has its own (single) bias. Finally you also have access to the hyperparameters dictionary which contains the stride and the padding.

Hint:

1. To select a 2x2 slice at the upper left corner of a matrix "a_prev" (shape (5,5,3)), you would do:

   ```
   a_slice_prev = a_prev[0:2,0:2,:]
   ```

   This will be useful when you will define `a_slice_prev` below, using the `start/end` indexes you will define.
2. To define a_slice you will need to first define its corners `vert_start`, `vert_end`, `horiz_start` and `horiz_end`. This figure may be helpful for you to find how each of the corner can be defined using h, w, f and s in the code below.

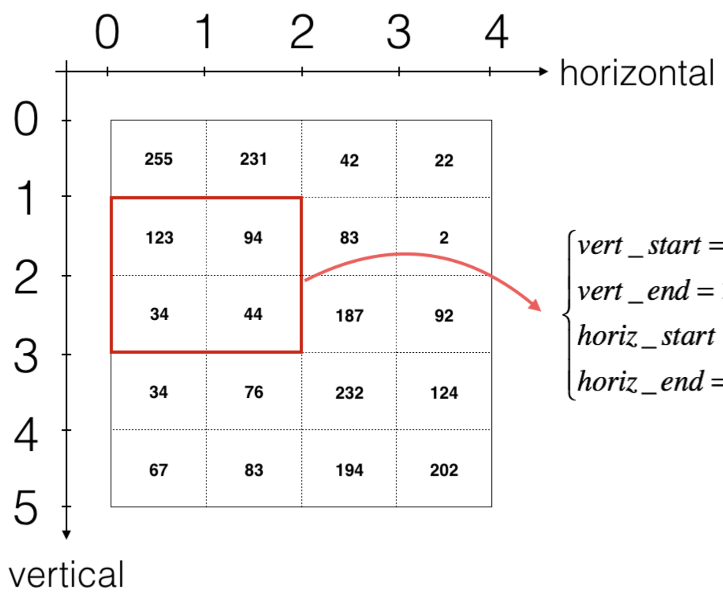


Figure 3 : **Definition of a slice using vertical and horizontal start/end (with a 2x2 filter)**
This figure shows only a single channel.

Reminder: The formulas relating the output shape of the convolution to the input shape is:

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_C = \text{number of filters used in the convolution}$$

For this exercise, we won't worry about vectorization, and will just implement everything with for-loops.

In [6]:
```python
# GRADED FUNCTION: conv_forward

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
    """

    ### START CODE HERE ###
    # 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 = 1 + int((n_H_prev + 2 * pad - f) / stride)
    n_W = 1 + int((n_W_prev + 2 * pad - f) / stride)

    # 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 (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. (≈1 line)
                    Z[i, h, w, c] = np.sum(np.multiply(a_slice_prev, W[:, :, :, c]) + b[:, :, :, c])

    ### END CODE HERE ###

    # 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
```

In [7]:
```python
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": 1}

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

```
Z's mean = 0.155859324889
cache_conv[0][1][2][3] = [-0.20075807  0.18656139  0.41005165]
```

Expected Output:

Z's mean	0.155859324889
cache_conv[0][1][2][3]	[-0.20075807 0.18656139 0.41005165]

Finally, CONV layer should also contain an activation, in which case we would add the following line of code:

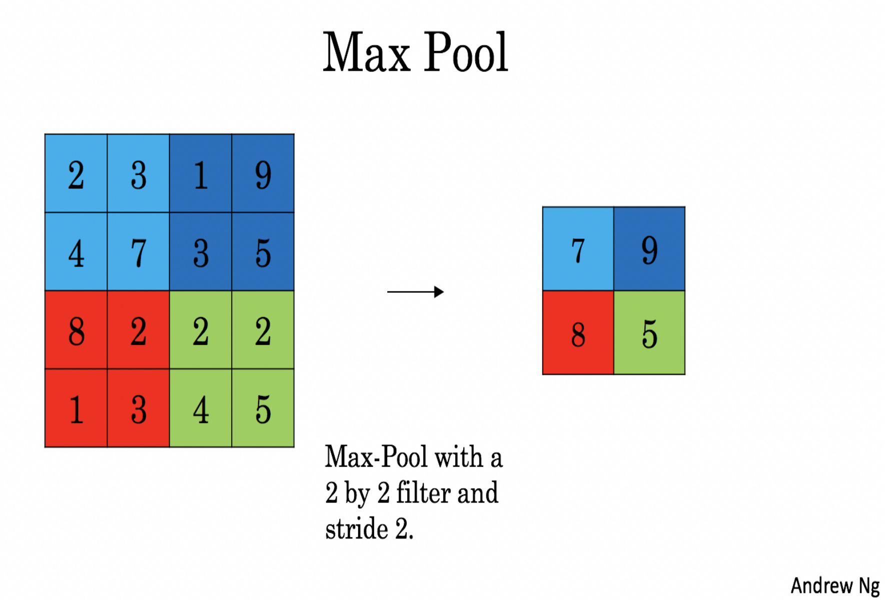
```
# Convolve the window to get back one output neuron
Z[i, h, w, c] = ...
# Apply activation
A[i, h, w, c] = activation(Z[i, h, w, c])
```

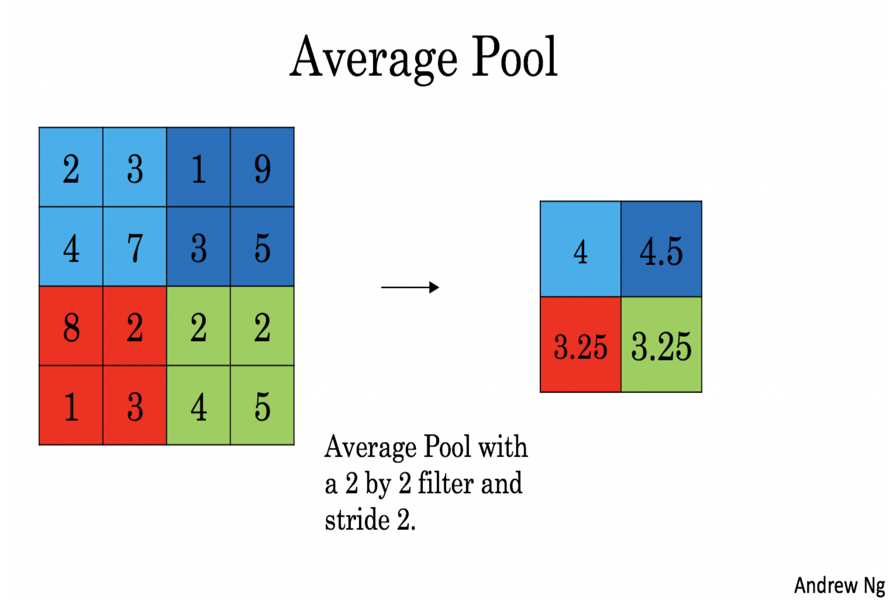
You don't need to do it here.

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, 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.





These pooling layers have no parameters for backpropagation to train. However, they have hyperparameters such as the window size f. This specifies the height and width of the fxf window you would compute a max or average over.

4.1 - Forward Pooling

Now, you are going to implement MAX-POOL and AVG-POOL, in the same function.

Exercise: Implement the forward pass of the pooling layer. Follow the hints in the comments below.

Reminder: As there's no padding, the formulas binding the output shape of the pooling to the input shape is:

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

In [8]:
```python
# GRADED FUNCTION: 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))

    ### START CODE HERE ###
    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)

    ### END CODE HERE ###

    # 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
```

In [9]:
```python
np.random.seed(1)
A_prev = np.random.randn(2, 4, 4, 3)
hparameters = {"stride" : 1, "f": 4}

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  1.6924546   2.10025514]]]


 [[[ 1.19891788  1.51981682  2.18557541]]]]

mode = average
A = [[[[-0.09498456  0.11180064 -0.14263511]]]


 [[[-0.09525108  0.28325018  0.33035185]]]]
```

Expected Output:

```
A =    [[[[ 1.74481176 1.6924546 2.10025514]]]
       [[[ 1.19891788 1.51981682 2.18557541]]]]

A =    [[[[-0.09498456 0.11180064 -0.14263511]]]
       [[[-0.09525108 0.28325018 0.33035185]]]]
```

Congratulations! You have now implemented the forward passes of all the layers of a convolutional network.

The remainer of this notebook is optional, and will not be graded.

5 - Backpropagation in convolutional neural networks (OPTIONAL / UNGRADED)

In modern deep learning frameworks, you only have to implement the forward pass, and the framework takes care of the backward pass, so most deep learning engineers don't need to bother with the details of the backward pass. The backward pass for convolutional networks is complicated. If you wish however, you can work through this optional portion of the notebook to get a sense of what backprop in a convolutional network looks like.

When in an earlier course you implemented a simple (fully connected) neural network, you used backpropagation to compute the derivatives with respect to the cost to update the parameters. Similarly, in convolutional neural networks you can to calculate the derivatives with respect to the cost in order to update the parameters. The backprop equations are not trivial and we did not derive them in lecture, but we briefly presented them below.

5.1 - Convolutional layer backward pass

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

5.1.1 - Computing dA:

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

$$dA += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c \times dZ_{hw} \tag{1}$$

Where W_c is a filter and dZ_{hw} is a scalar corresponding to the gradient of the cost with respect to the output of the conv layer Z at the hth row and wth column (corresponding to the dot product taken at the ith stride left and jth stride down). Note that at each time, we multiply the the same filter W_c by a different dZ when updating dA. We do so mainly because when computing the forward propagation, each filter is dotted and summed by a different a_slice. Therefore when computing the backprop for dA, we are just adding the gradients of all the a_slices.

In code, inside the appropriate for-loops, this formula translates into:

```
da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] += W[:,:,:,c] * dZ[i, h, w, c]
```

5.1.2 - Computing dW:

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

$$dW_c += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw} \tag{2}$$

Where a_{slice} corresponds to the slice which was used to generate the acitivation Z_{ij}. 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.

In code, inside the appropriate for-loops, this formula translates into:

```
dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
```

5.1.3 - Computing db:

This is the formula for computing db with respect to the cost for a certain filter W_c:

$$db = \sum_h \sum_w dZ_{hw} \tag{3}$$

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.

In code, inside the appropriate for-loops, this formula translates into:

```
db[:,:,:,c] += dZ[i, h, w, c]
```

Exercise: Implement the `conv_backward` function below. You should sum over all the training examples, filters, heights, and widths. You should then compute the derivatives using formulas 1, 2 and 3 above.

In [10]:
```
def conv_backward(dZ, cache):
    """
    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)
    """

    ### START CODE HERE ###
    # 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[i, pad:-pad, pad:-pad, :]
    ### END CODE HERE ###

    # 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
```

In [11]:
```
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 = 9.60899067587
dW_mean = 10.5817412755
db_mean = 76.3710691956
```

Expected Output:

dA_mean	9.60899067587
dW_mean	10.5817412755
db_mean	76.3710691956

5.2 Pooling layer - backward pass

Next, let's implement the backward pass for the pooling layer, starting with the MAX-POOL layer. Even though a pooling layer has no parameters for backprop to update, you still need to backpropagation the gradient through the pooling layer in order to compute gradients for layers that came before the pooling layer.

5.2.1 Max pooling - backward pass

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

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

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). You'll see later that the backward pass for average pooling will be similar to this but using a different mask.

Exercise: Implement `create_mask_from_window()`. This function will be helpful for pooling backward. Hints:

- np.max() () may be helpful. It computes the maximum of an array.
- If you have a matrix X and a scalar x: `A = (X == x)` will return a matrix A of the same size as X such that:

```
A[i,j] = True if X[i,j] = x
A[i,j] = False if X[i,j] != x
```

- Here, you don't need to consider cases where there are several maxima in a matrix.

In [12]:
```
def create_mask_from_window(x):
    """
    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.
    """

    ### START CODE HERE ### (≈1 line)
    mask = (x == np.max(x))
    ### END CODE HERE ###

    return mask
```

In [13]:
```
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]]
```

Expected Output:

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

Why do we keep track of the position of the max? It's because this is the input value that ultimately influenced the output, and therefore the cost. Backprop is computing gradients with respect to the cost, so anything that influences the ultimate cost should have a non-zero gradient. So, backprop will "propagate" the gradient back to this particular input value that had influenced the cost.

5.2.2 - Average pooling - backward pass

In max pooling, for each input window, all the "influence" on the output came from a single input value--the max. 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:

$$dZ = 1 \quad \rightarrow \quad dZ = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix} \tag{5}$$

This implies that each position in the dZ matrix contributes equally to output because in the forward pass, we took an average.

Exercise: Implement the function below to equally distribute a value dz through a matrix of dimension shape. Hint (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.ones.html)

In [14]:
```
def distribute_value(dz, shape):
    """
    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
    """

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

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

    # Create a matrix where every entry is the "average" value (≈1 line)
    a = np.ones(shape) * average
    ### END CODE HERE ###

    return a
```

In [15]:
```
a = distribute_value(2, (2,2))
print('distributed value =', a)
```

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

Expected Output:

distributed_value =	[[0.5 0.5] [0.5 0.5]]

5.2.3 Putting it together: Pooling backward

You now have everything you need to compute backward propagation on a pooling layer.

Exercise: Implement the `pool_backward` function in both modes (`"max"` and `"average"`). You will once again use 4 for-loops (iterating over training examples, height, width, and channels). You should use an `if/elif` statement to see if the mode is equal to `'max'` or `'average'`. If it is equal to 'average' you should use the `distribute_value()` function you implemented above to create a matrix of the same shape as `a_slice`. Otherwise, the mode is equal to '`max`', and you will create a mask with `create_mask_from_window()` and multiply it by the corresponding value of dZ.

```
In  [16]: def pool_backward(dA, cache, mode = "max"):
              """
              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
              """

              ### START CODE HERE ###

              # 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_like(A_prev)

              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 * dA[i, vert_start, horiz_start, c]

                              elif mode == "average":

                                  # Get the value a from dA (≈1 line)
                                  da = dA[i, vert_start, horiz_start, 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)

              ### END CODE ###

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

              return dA_prev
```

In [17]:
```
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 =  0.145713902729
dA_prev[1,1] =  [[ 0.          0.        ]
 [ 5.05844394 -1.68282702]
 [ 0.          0.        ]]

mode = average
mean of dA =  0.145713902729
dA_prev[1,1] =  [[ 0.08485462  0.2787552 ]
 [ 1.26461098 -0.25749373]
 [ 1.17975636 -0.53624893]]
```

Expected Output:

mode = max:

mean of dA =	0.145713902729
dA_prev[1,1] =	[[0. 0.] [5.05844394 -1.68282702] [0. 0.]]

mode = average

mean of dA =	0.145713902729
dA_prev[1,1] =	[[0.08485462 0.2787552] [1.26461098 -0.25749373] [1.17975636 -0.53624893]]

Congratulations !

Congratulation on completing this assignment. You now understand how convolutional neural networks work. You have implemented all the building blocks of a neural network. In the next assignment you will implement a ConvNet using TensorFlow.