Rede Neural Convolucional: Passo a Passo

Bem vindo a primeira tarefa do curso 3! Netsa tarefa você irá implementar as camadas convolucionais (CONV) e pooling (POOL) em numpy, incluindo a propagação para frente e a programação para trás (opcional).

Notação:

- Sobrescrito $[l]$ indica um objeto da $l^{ésima}$ camada.
  - Exemplo: $a^{[4]}$ é a ativação da 4^a camada. $W^{[5]}$ e $b^{[5]}$ são os parâmetros da 5^a camada.
- Sobrescrito (i) indica um objeto do $i^{ésimo}$ exemplo.
  - Exemplo: $x^{(5)}$ é o 5^o exemplo de treinamento.
- Subscrito i indica a $i^{ésima}$ entrada de um vetor.
  - Exemplo: $a^{[l]}_3$ indica a a^a entrada da camada de ativação l, assumindo que esta é uma camada totalmente conectada (FC).
- n_H, n_W e n_C indicam respectivamente a altura, largur e número de canais de uma dada camada. Se você quer se referenciar a uma camada específica l, você pode escrever $n_H^{[l]}$, $n_W^{[l]}$, $n_C^{[l]}$.
- $n_{H_{prev}}$, $n_{W_{prev}}$ e $n_{C_{prev}}$ indicam respectivamente a altura, largura e número de canais de uma camada anterior. Para inidicar uma camada específica l, você pode escrever $n_H^{[l-1]}$, $n_W^{[l-1]}$, $n_C^{[l-1]}$.

Estamos assumindo que você já está acostumado com `numpy`. Vamos começar!

1 - Pacotes

Vamos importar os pacotes necessários para esta tarefa.

- numpy (www.numpy.org) é o principal pacote para computação científica do Python.
- matplotlib (http://matplotlib.org) é uma biblioteca para plotar gráficos em Python.
- np.random.seed(1) é utilizada para manter as chamadas de funções aleatórias consistentes.

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

%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # ajusta o padrão para os tamanhos de figuras
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

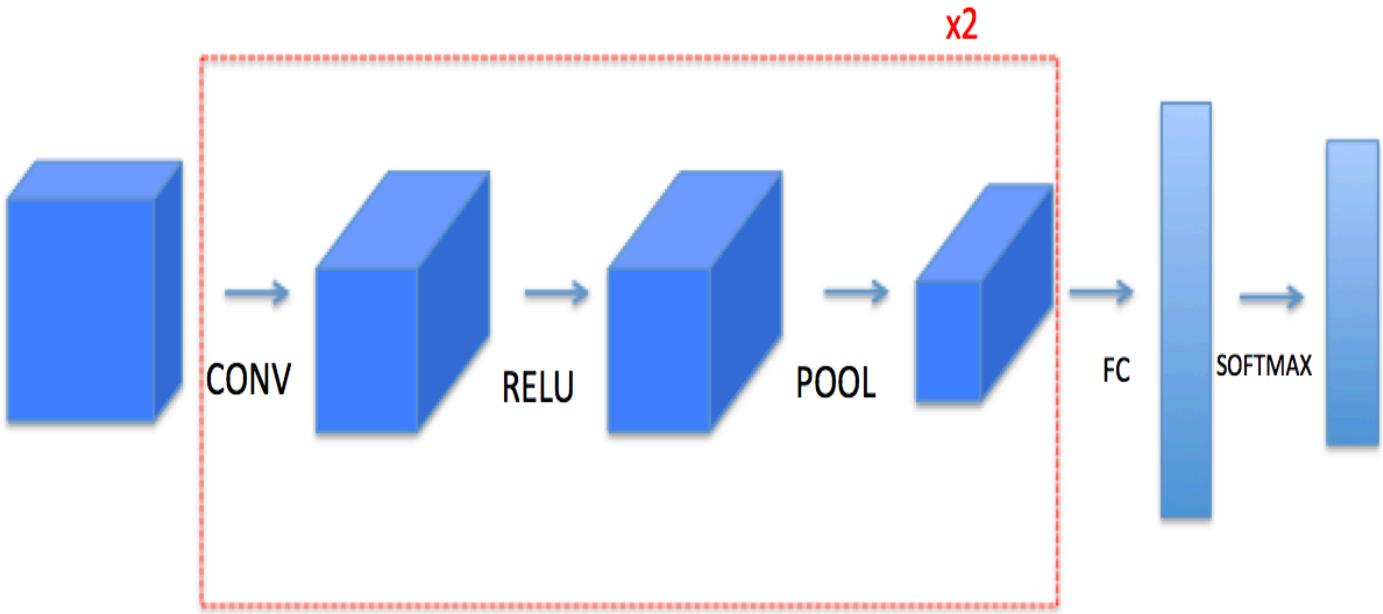
```
/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

2 - Descrição da tarefa

Voc6e irá implementar os blocos, um a um, de uma rede neural convolucional! Cada função que você implementar terá instruções detalhadas para auxiliá-lo nos passos necessários para a implementação:

- Funções da convolução, incluem:
  - Padding com zeros.
  - Kernel de convolução.
  - Convolução para frente.
  - Convolução para trás (opcional).
- Funções de Pooling, incluindo:
  - Pooling para frente.
  - Criar uma máscara.
  - Distribuir valores.
  - Pooling para trás (opcional).

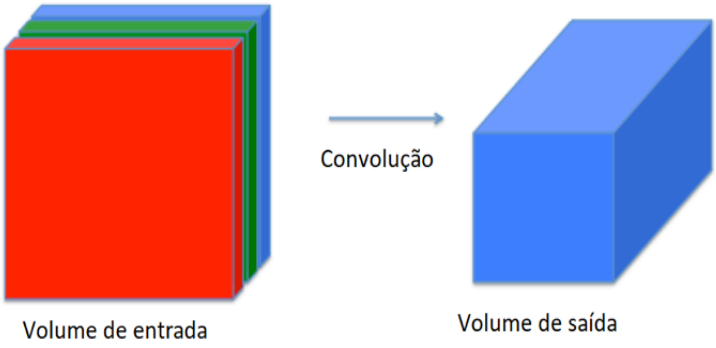
Este notebook irá guiá-lo para implementar estas funções em `numpy` do zero. Na próxima tarefa você irá utilizar as funções equivalentes do tensorFlow para construir o modelo abaixo:



Nota para cada função para frente existe uma equivalente para trás. Portanto, em cada etapado seu modelo para frente você deve armazenar parâmetros em uma cache. Estes parâmetros serão utilizados para determinar os gradientes na propagação para trás.

3 - Rede Neural Convolucional

Embora os frameworks de programação tornem fácil o uso de convoluções, ele continuam sendo um dos conceitos mais dificeis de se entender em Aprendizado Profundo. Uma camada de convolução transforma um volume de entrada em um volme de saída de tamanho diferente, conforme mostrado abaixo:



Nesta parte, você irá construir cada um dos processos de uma camada de convolução. Primeiro você irá implementar duas funç/òes de auxilio: uma para fazer o padding usando zeros e a outra para computar a convolução propriamente.

3.1 - Padding com zeros

O processo de padding com zeros em volta da borda de uma imagem:

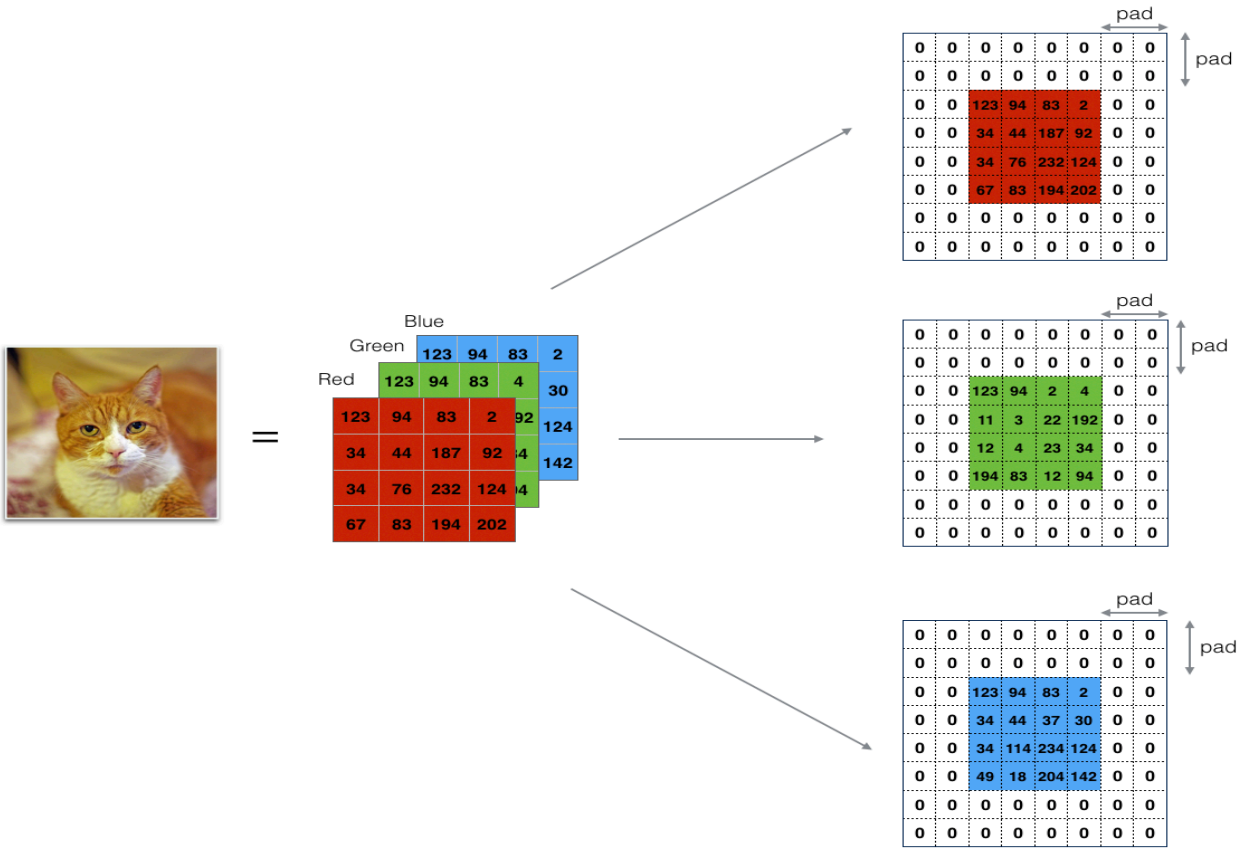


Figure 1 : **Padding com zeros**
Imagem (3 canais, RGB) com um padding de tamanho 2 (p=2).

Os principais beneficios do processo de padding são:

- Ele permite que se utlize uma camada CONV sem que se reduza o tamanho da saída (altura e largura do volume). Isto é importante para o caso de uso de redes mais profundas, caso não se utilize o padding a altura e largura podem ficar muito pequenas. Um caso especial importante é a convolução "same", onde a altura e largura na saída são exatamente iguais as da entrada.
- Ela auxilia na manutenção das informações da borda da imagem, sem o padding, poucos valores nas camadas seguintes seriam afetados por pixels nas bordas da imagem.

Exercício: Implemente a seguinte função que adiciona zeros em todas as imagens de um batch X de exemplos. Use np.pad (https://docs.scipy.org/doc/numpy/reference/generated/numpy.pad.html). Note que, se você deseja fazer um pad em um array "a" no formato $(5, 5, 5, 5, 5)$ com `p = 1` para a 2^a dimensão, `p = 3` para a 4^a dimensão e `p = 0` para o restante, voce deve fazer a seguinte chamada:

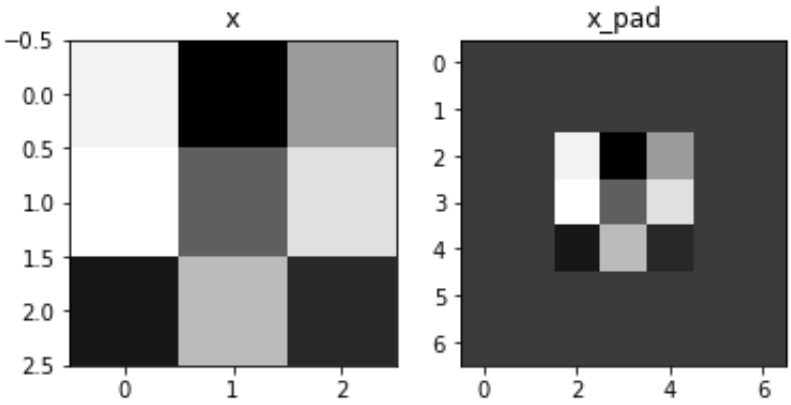
```
a = np.pad(a, ((0,0), (1,1), (0,0), (3,3), (0,0)), 'constant', constant_values = (..,..))
```

In [7]:
```
# FUNÇÃO DE AVALIAÇÃO: zero_pad

def zero_pad(X, pad):
    """
    Faça um pad de zeros em todas as imagens de uma base de dados X
. O padding deve ser aplicado na altura e largura das imagens
    conforme ilustrado na figura 1.

    Argumentos:
    X -- um array numpy no formato (m, n_H, n_W, n_C) representando
um batch de m imagens
    pad -- valor inteiro, número de linhas/colunas a ser adicionada
s com zeros em cada imagem.

    Returns:
    X_pad -- m imagens do batch no formato (m, n_H+pad, n_W+pad, n_
C)
    """
    ### INICIE SEU CÓDIGO AQUI ### (≈ 1 linha)
    X_pad = np.pad(X, ((0, 0), (pad, pad), (pad, pad), (0, 0)), 'co
nstant', constant_values=(0, 0))
    ### TÉRMINO DO CÓDIGO ###

    return X_pad
```

In [8]:
```
np.random.seed(1)
x = np.random.randn(4, 3, 3, 2)
x_pad = zero_pad(x, 2)
print ("formato de x =", x.shape)
print ("formato de x com padding =", 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])
```

```
formato de x = (4, 3, 3, 2)
formato de x com padding = (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[8]: <matplotlib.image.AxesImage at 0x118dd8f98>
```



Saída esperada:

formato de x:	(4, 3, 3, 2)
formato de x com padding:	(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 - Uma etapa da convolução

Nesta parte você irá implementar uma etapa do processo de convolução. Você aplica o filtro em uma posição da imagem de entrada. Esta função será utilizada para construir a unidade de convolução que:

- Recebe como entrada um volume
- Aplica o filtro em cada posição da entrada
- Retorna como saída um novo volume (normalmente de tamanho diferente)

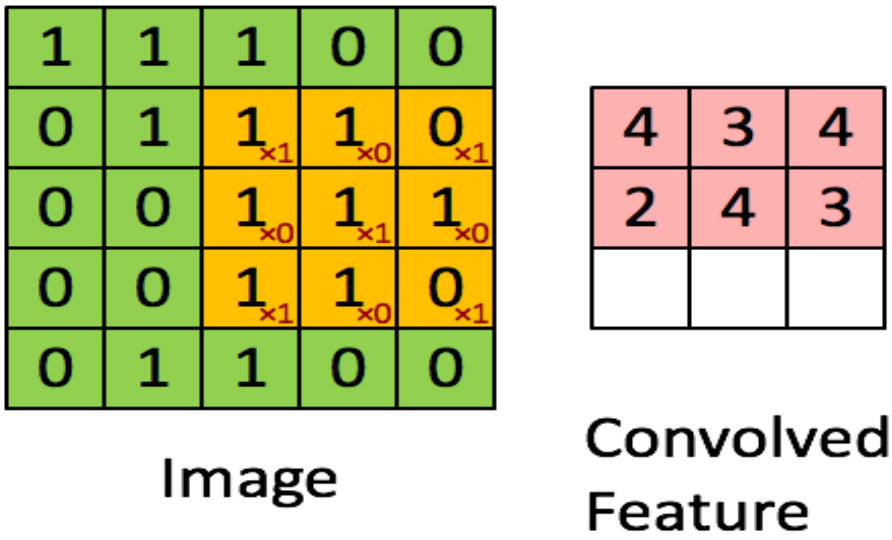


Figura 2 : **Operação de Convolução**
com um filtro de tamanho 3x3 e um **stride** de 1 (stride = quanto o filtro é movido para cada operação)

Em uma aplicação de visão computacional, cada valor na matriz à esuqerda corresponde ao valor de um pixel da imagem. Fazemos a convolução com um filtro 3x3 multiplicando seus valores elemento a elemento com os valores da imagem e então somamos o resultado das multiplicações e adicionando um bias. Nesta primeira etapado exercício você irá implementar um etapa simples do processo de convolução que corresponde a aplicação do filtro em uma única posição da imagem e obtendo como saída um valor real.
Mais tarde utilizaremos esta função para fazer o processo de convolução em toda a imagem de entrada.

Exercício: Implemente conv_single_step(). Dica (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.sum.html).

In [13]:
```
# FUNÇÃO DE AVALIAÇÃO: conv_single_step

def conv_single_step(a_slice_prev, W, b):
    """
    Aplica um filtro definido pelos parâmetros W sobre uma parte da imagem de entrada (a_slice_prev) da saída da camada de
    ativacao anterior.

    Argumentos:
    a_slice_prev -- parte dos dados de entrado no formato (f, f, n_C_prev)
    W -- parâmetros do filtro a ser utilizado -matriz no formato (f, f, n_C_prev)
    b -- parâmetro de Bias - possui formato (1, 1, 1)

    Retorna:
    Z -- um valor escalar, resultado da convolução pelo filtro W, e adiciona o bias bno final
    """

    ### INICIE O SEU CÓDIGO AQUI ### (≈ 3 linhas de código)
    # produto elemento a elemento entre a parte da imagem de entrada e o filtro.
    s = np.multiply(a_slice_prev, W)
    # Soma dos valores dos produtos feitos acima.
    Z = np.sum(s)
    # Adiciona o Bias b em Z. Defina b como um float() de forma que Z seja um valor escalar.
    Z = Z + b
    ### TÉRMINO DO CÓDIGO ###

    return Z
```

In [14]:
```
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)
```

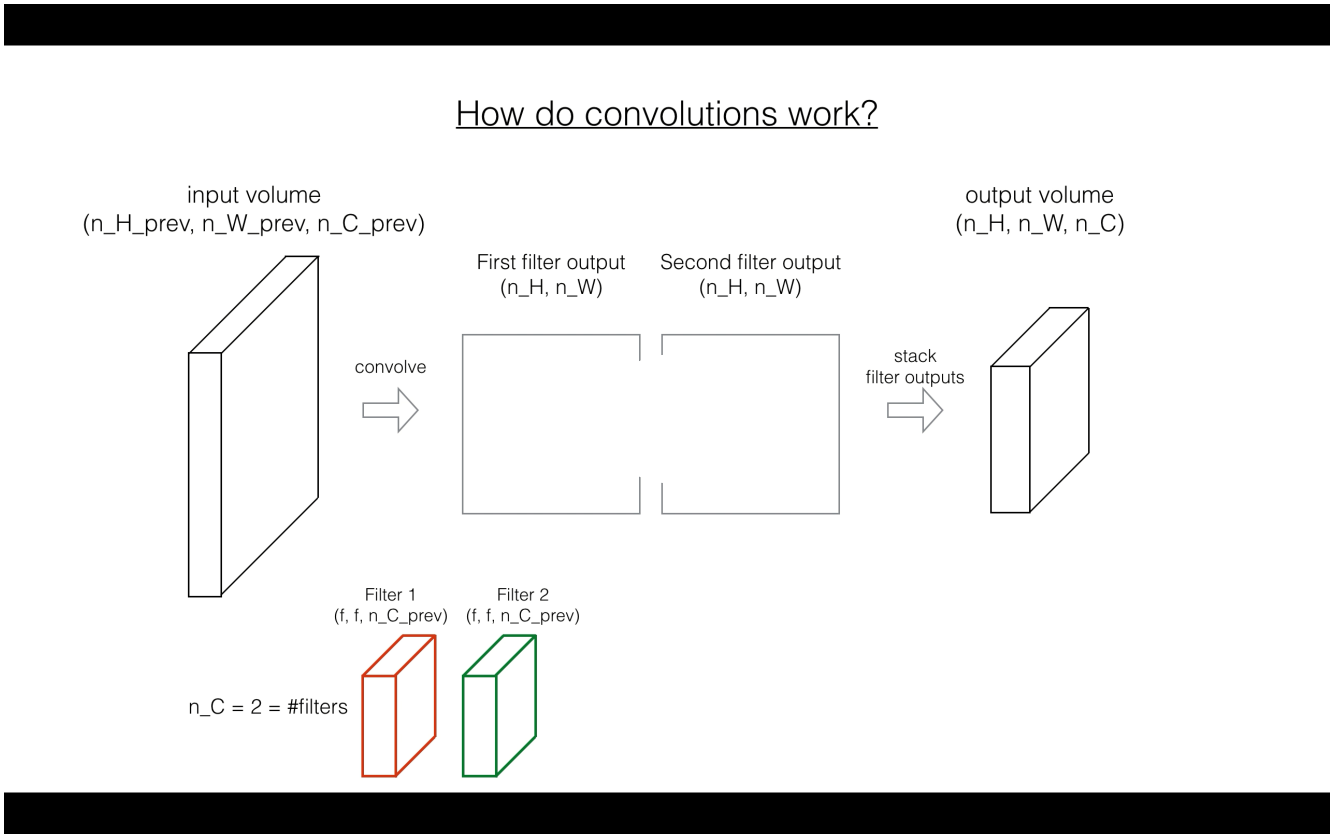
```
Z = [[[-6.99908945]]]
```

Saída esperada:

Z	-6.99908945068

3.3 - Redes Neurais Convolucionais - Passo para frente

No passo para frente você irá pegar vários filtros e fazer a convolução deles com a entrada. Cada convolução retorna uma matriz em 2D de saída. Você irá empilhar estas matrizes para obter um volume em 3D:



Exercício: Implemente a função abaixo para fazer a convolução dos filtros W sobre a entrada da ativação A_prev. Esta função recebe como entrada A_prev, a saída da ativação da camada anterior (para um batch de m exemplos), F filtros/pesos indicados por W, e um vetor de bias indicado como b, onde cada filtro possui seu próprio bias. Por fim, você ainda tem acesso ao dicionário de hiper-parâmentros contendo o valor de stride e padding.

Dica:

1. Para selecionar uma fatia 2x2 do canto superior esquerdo da matriz "a_prev" (formato (5,5,3)), você deve fazer:

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

   Isto será útil para você definir `a_slice_prev` abaixo, utilizando os índices `inicio/fim` que você definir.
2. Para definir a_slice você primeiro precisa definir os cantos `vert_inicio`, `vert_fim`, `horiz_inicio` e `horiz_fim`. A figura abaixo deve ajudá-lo a encontrar cada um dos cantos definidos usando h, w, f e s no código abaixo.

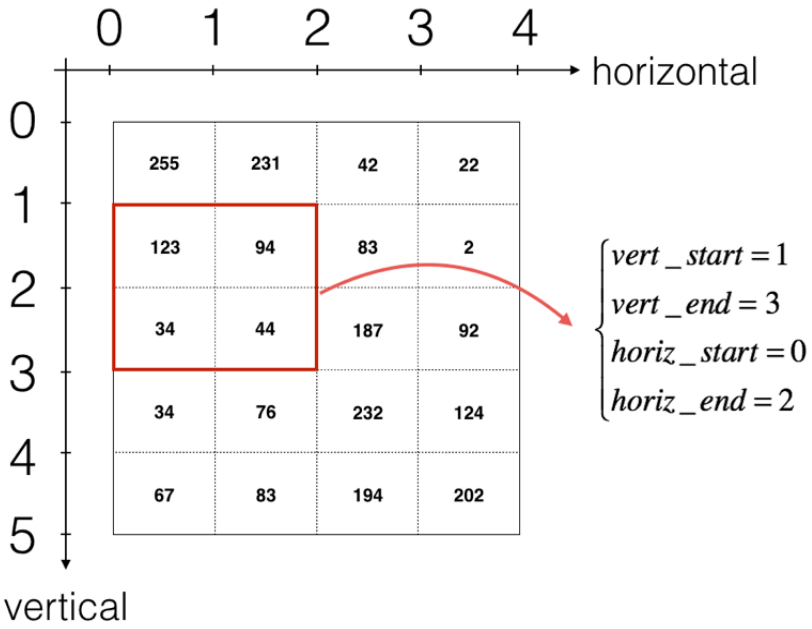


Figure 3 : **Definição de uma fatia utilizando inicio/fim horizontal e vertical (com um filtro 2x2)**
A figura mostra apenas um canal.

Lembre-se: As fórmulas relacionadas com o formato da saída da convolução são:

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$

$$n_C = \text{número de filtros utilizados na convolução}$$

Para este exercício não iremos nos preocupar com vetorização e implementaremos tudo usando loops.

In [108]:
```
# FUNÇÃO DE AVALIAÇÃO: conv_forward

def conv_forward(A_prev, W, b, hparameters):
    """
    Implementa a etapa de propagação para frente para a função de c
onvolução.

    Argumentos:
    A_prev -- saída da camada de ativação anterior, um array numpy
no formato (m, n_H_prev, n_W_prev, n_C_prev)
    W -- Pesos, array numpy no formato (f, f, n_C_prev, n_C)
    b -- Biases, array numpy no formato (1, 1, 1, n_C)
    hparameters -- dicionário python contendo os valores de "stride
```

```
" e "pad"

    Retorna:
    Z -- saída da convolução, um array numpy no formato (m, n_H, n_
W, n_C)
    cache -- cache dos valores necessários para a propagação para t
rás.
    """

    ### INICIE SEU CÓDIGO AQUI ###
    # recupere as dimensões de A_prev (≈1 linha)
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # recupere as dimensões de W (≈1 linha)
    (f, f, n_C_prev, n_C) = W.shape

    # recupere as informações de stride e pad de "hparameters" (≈2
linhas)
    stride = hparameters["stride"]
    pad = hparameters["pad"]

    # Compute a dimensão do volume de saida CONV utilizando a fórmu
la dada acima. Dica: use int() para floor. (≈2 linhas)
    n_H = int(((n_H_prev - f + (2 * pad)) / stride) + 1)
    n_W = int(((n_W_prev - f + (2 * pad)) / stride) + 1)

    # Inicialize o volume de saída Z com zeros. (≈1 linha)
    Z = np.zeros((m, n_H, n_W, n_C))

    # Crie A_prev_pad fazendo o padding de A_prev. (~1 linha)
    A_prev_pad = zero_pad(A_prev, pad)

    for i in range(m):                                  # Faça o loop
sobre todos os exemplos de treinamento
        a_prev_pad = A_prev_pad[i, :, :, :]  # Selecione o i-ésimo
exemplo já com o padding
        for h in range(n_H):                           # Faça um loop s
obre o eixo vertical do volume de saída
            for w in range(n_W):                       # Faça um loop s
obre o eixo horizontal do volume de saída
                for c in range(n_C):                   # Faça um loop
sobre os canais (= #filtros) do volume de saída

                    # Encontre os cantos da fatia atual (≈4 linhas)
                    vert_start = h
                    vert_end = vert_start + f
                    horiz_start = w
                    horiz_end = horiz_start + f

                    # Use os cantos para definir a fatia em 3D do a
_prev_pad (Veja a dica na célula acima). (≈1 linha)
                    a_slice_prev = a_prev_pad[vert_start: vert_end,
horiz_start: horiz_end, :]
```

```
                    # Faça a convolução da fatia em 3D com o filtro
W e adicione o bias b
                    # para obter a saída do neurônio. (≈1 linha)
                    Z[i, h, w, c] = conv_single_step(a_slice_prev,
W[:, :, :, c], b[:, :, :, c])

    ### TÉRMINO DO CÓDIGO AQUI ###

    # Verificando o formato da saída para saber se está correto.
    assert(Z.shape == (m, n_H, n_W, n_C))

    # Salva a informação na cache para a propagação para trás
    cache = (A_prev, W, b, hparameters)

    return Z, cache
```

In [109]:
```
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 = -0.007801972093158612
Z[3,2,1] = [ 0.10709871 -0.03102354 -0.52995452  0.98611224  0.657
33641 -0.84239368
 -0.04608241  0.08802027]
cache_conv[0][1][2][3] = [-0.20075807  0.18656139  0.41005165]
```

Saída esperada:

média dos Zs	0.0489952035289
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]

Finalmente, a camada CONV deve ter uma função de ativação, portanto iremos adicionar a seguinte linha de código:

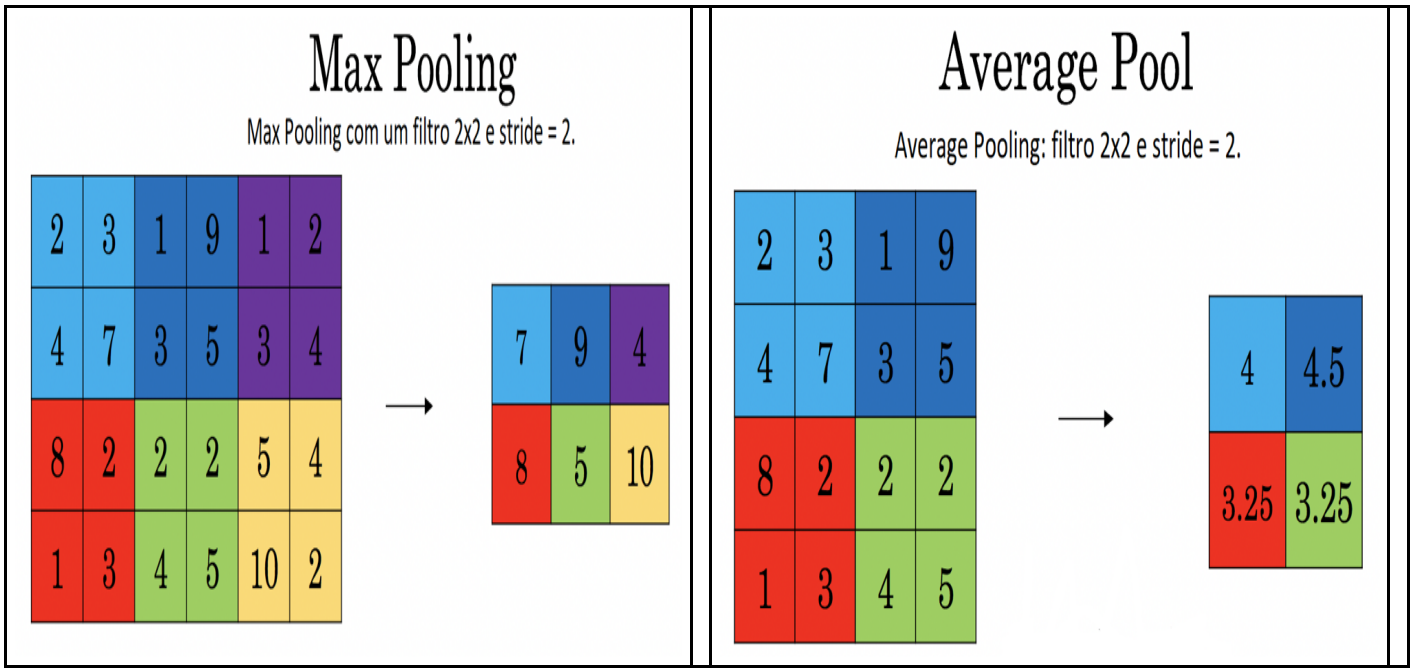
```
# Convolução da janela para obter a saída do neurônio
Z[i, h, w, c] = ...
# Aplique a ativação
A[i, h, w, c] = activation(Z[i, h, w, c])
```

Você não precisa fazer isto aqui.

4 - Camada de Pooling

A camada de pooling (POOL) reduz as dimensões de altura e largura da entrada. Ele auxilia na redução da computação e também torna os detectores de características mais invariantes de sua posição na entrada. Os dois tipos de pooling são:

- Camada de Max-pooling: utiliza uma janela de tamanho (f, f) sobre a entrada e armazena o maior valor encontrado na janela.
- Camada Average-pooling: utiliza uma janela de tamanho (f, f) sobre a entrada e armazena a média dos valores encontrados na janela.



As camadas de pooling não possuem parâmetros a serem treinados pelo processo de propagação para trás. Porém, elas possuem hiper-parâmetros como o tamanho da janela$f$. Este valor especifica a altura e largura da janela utilizada para determinar o máximo ou a média.

4.1 - Pooling para frente

Agora você irá implementar as duas funções: MAX-POOL e AVG-POOL, na mesma função.

Exercício: Implemente o passo para frente da camada de pooling. Siga as dicas abaixo:

Dicas: Como não existe padding, as fórmulas que definem o formato da saída do pooling são:

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

In [110]:
```
# FUNÇÃO DE AVALIAÇÃO: pool_forward
```

```
def pool_forward(A_prev, hparameters, mode = "max"):
    """
    Implementa o passo para frente da camada de pooling.

    Argumentos:
    A_prev -- dados de entrada, um array numpy no formato (m, n_H_p
rev, n_W_prev, n_C_prev)
    hparameters -- dicionário python contendo "f" e "stride"
    mode -- o modo de pooling desejado, definido como uma string ("
max" ou "average")

    Retorna:
    A -- saída da camada de pooling, um array numpy array no format
o (m, n_H, n_W, n_C)
    cache -- cache utilizada na passo de propagação para trás da ca
mada de pooling, contém a entrada e hparameters
    """

    # Recupera as dimensões do formato da entrada
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Recupera os hiper-parâmetros de "hparameters"
    f = hparameters["f"]
    stride = hparameters["stride"]

    # Define as dimensões da saída
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    # Inicializa a matriz de saída
    A = np.zeros((m, n_H, n_W, n_C))

    ### INICIE O SEU CÓDIGO AQUI ###
    for i in range(m):                               # loop sobre os ex
emplos de treinamento
        for h in range(n_H):                         # loop no eixo ver
tical do volume de saída
            for w in range(n_W):                     # loop no eixo hor
izontal do volume de saída
                for c in range (n_C):                # loop sobre os ca
nais do volume de saída

                    # Determine os cantos da fatia atual (≈4 linhas
)
                    vert_start = h
                    vert_end = vert_start + f
                    horiz_start = w
                    horiz_end = horiz_start + f

                    # Use os cantos para definir a fatia no i-ésimo
exemplo de treinamento de A_prev, no canal c. (≈1 linha)
                    a_prev_slice = A_prev[i, : , : , c]
```

```
                    # Compute a operação de pooling desejada na fat
ia. Use um comando if para diferenciar os modos.
                    # Use np.max/np.mean (2 linhas).
                    if mode == "max":
                        A[i, h, w, c] = np.max(a_prev_slice[vert_st
art:vert_end, horiz_start:horiz_end])
                    elif mode == "average":
                        A[i, h, w, c] = np.mean(a_prev_slice[vert_s
tart:vert_end, horiz_start:horiz_end])

    ### TÉRMINO DO CÓDIGO ###

    # Armazene a entrada e os hparameters na "cache" para fazer a p
ropagação para trás.
    cache = (A_prev, hparameters)

    # Verifica se as dimensões da saída estão corretas.
    assert(A.shape == (m, n_H, n_W, n_C))

    return A, cache
```

In [111]:
```
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("modo = max")
print("A =", A)
print()
A, cache = pool_forward(A_prev, hparameters, mode = "average")
print("modo = média")
print("A =", A)
```

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


 [[[1.13162939 1.51981682 2.18557541]]]]

modo = média
A = [[[[ 0.02105773 -0.20328806 -0.40389855]]]


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

Saída esperada:

A =	[[[[1.74481176 0.86540763 1.13376944]]] [[[1.13162939 1.51981682 2.18557541]]]]
A =	[[[[0.02105773 -0.20328806 -0.40389855]]] [[[-0.22154621 0.51716526 0.48155844]]]]

Parabéns! Você implementou os passos da propagação para frente de uma rede convolucional.

O restante deste notebook é opcional e não será avaliado.

5 - Propagação para trás em uma rede convolucional (Opcional)

Nos frameworks mais novos para aprendizado profundo você deve apenas implementar o passo para frente, logo, a maioria dos profissionais que atuam em aprendizado profundo não se preocupam com o passo para trás. O passo para trás nas redes convolucionais é complicado. Porém, se você desejar, você pode trabalhar nesta parte opcional da tarefa que irá te dar uma idéia de como é feita a propagação para trás em uma rede convolucional.

Já foi visto como se implementa o passo para trás em uma rede totalmente conectada, o passo para trás é utilizado para computar as derivadas com relação ao custo para atualizar os parâmetros. Da mesma forma, nas redes convolucionais você pode calcular as derivadas com relação ao custo para a tualizar os parâmetros da rede. As equações da propagação para trás não são triviais e nós não trabalhamos com elas na aula, mas elas serão apresentadas abaixo.

5.1 - Camada convolucional - passo para trás

Vamos começar implementando o passo para trás de uma camada CONV.

5.1.1 - Computando dA:

Esta é a fórmula para se determinar dA com relação ao custo para um certo filtro W_c e um dado exemplo de treinamento:

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

Onde W_c é um filtro e dZ_{hw} é um escalar que corresponde ao gradiente do custo com relação a saída da camada CONV Z na h-ésima linha e w-ésima coluna (correspondendo ao produto escalar (dot product) feito no i-ésimo stride para a esquerda e j-ésimo stride para baixo). Note que a cada vez, multiplicamos o mesmo filtro W_c por um valor diferente de dZ quando atualizamos o valor de dA. Fazemos isto porque, quando computamos a propagação para frente cada filtro faz o produto e soma para uma fatia diferente. Logo, quando computamos a propagação para trás para dA, estamos simplesmente adicionando os gradientes de todas as fatias.

Em código, dentro do loop apropriado, esta fórmula resolve isto:

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

5.1.2 - Computando dW:

Esta é a fórmula para computar dW_c (dW_c é a derivda de um filtro) com relação ao custo:

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

Onde a_{slice} corresponde a fatia que foi utilizada para gerar a ativação Z_{ij}. Logo, isto acaba dando o gradiente para W com relação aquela fatia. Como ele está relacionado ao mesmo W, nós simplesmente adicionamos todos os gradientes para obter dW.

Em código, dentro do loop apropriado, esta fórmula resolve isto:

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

5.1.3 - Computando db:

Esta fórmula determina db com relação ao custo para um determinado filtro W_c:

$$db = \sum_{h} \sum_{w} dZ_{hw} \tag{3}$$

Como já foi visto anteriormente nas redes neurais básicas, db é determinado somando-se dZ. Neste caso você irá somar sobre todos os gradientes da saida da CONV (Z) com relação ao custo.

Em código, dentro do loop apropriado, esta fórmula resolve isto:

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

Exercício: Implemente a função `conv_backward` abaixo. Você deveria somar sobre todos os exemplos de treinamentos, filtros, alturas e larguras. Na sequência você deve computar as derivadas usando as equações 1, 2 e 3 acima.

In [114]:
```
def conv_backward(dZ, cache):
    """
    Implemente a propagação para trás para a função de convolução

    Argumentos:
    dZ -- gradiente do custo com relação a saída da camada CONV (Z)
, array numpy no formato (m, n_H, n_W, n_C)
    cache -- cache dos valores necessários para a função conv_backw
ard(), saídas da conv_forward()

    Retorna:
    dA_prev -- gradiente do custo com relação a entrada da camada C
ONV (A_prev),
               array numpy no formato (m, n_H_prev, n_W_prev, n_C_p
rev)
    dW -- gradiente do custo com relação aos pesos da camada CONV (
W)
          array numpy no formato (f, f, n_C_prev, n_C)
```

```
    db -- gradiente do custo com relação ao bias da camada CONV (b)
          array numpy no formato (1, 1, 1, n_C)
    """

    ### INICIE SEU CÓDIGO AQUI ###
    # Recupere a informação da "cache"
    (A_prev, W, b, hparameters) = cache

    # Recupere as dimensões de A_prev
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Recupere as dimensões de W
    (f, f, n_C_prev, n_C) = W.shape

    # Recupere as informações de "hparameters"
    stride = hparameters["stride"]
    pad = hparameters["pad"]

    # Recupere as informações de dZ
    (m, n_H, n_W, n_C) = dZ.shape

    # Inicialize dA_prev, dW, db com os formatos correspondentes
    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))

    # Faça o padding em A_prev e dA_prev
    A_prev_pad = zero_pad(A_prev, pad)
    dA_prev_pad = zero_pad(dA_prev, pad)

    for i in range(m):                       # loop sobre os exempl
os de treinamento

        # selecione o i-ésimo exemplo de treinamento de A_prev_pad
e dA_prev_pad
        a_prev_pad = A_prev_pad[i, :, :, :]
        da_prev_pad = dA_prev_pad[i, :, :, :]

        for h in range(n_H):                   # loop sobre o eixo
vertical do volume de saída
            for w in range(n_W):               # loop sobre o eixo
horizontal do volume de saída
                for c in range(n_C):           # loop sobre os cana
is do volume de saída

                    # Encontre os cantos da fatia atual
                    vert_start = h
                    vert_end = vert_start + f
                    horiz_start = w
                    horiz_end = horiz_start + f

                    # Use os cantos para definir a fatia de a_prev_
pad
```

```
                    a_slice = a_prev_pad[vert_start:vert_end, horiz
_start:horiz_end, :]

                    # Atualize os gradientes para a janela e os par
âmetros do filtro utilizando as fórmulas acima.
                    da_prev_pad[vert_start:vert_end, horiz_start:ho
riz_end, :] += W[:,:,:,c] * dZ[i, h, w, c]
                    dW[:,:,:,c] += a_slice * dZ[i, h, w, c]
                    db[:,:,:,c] += dZ[i, h, w, c]

        # Defina o i-ésimo exemplo de treinamento dA_prev para da_p
rev_pad sem o padding (Dica: use X[pad:-pad, pad:-pad, :])
        dA_prev[i, :, :, :] = da_prev_pad[pad:-pad, pad:-pad, :]
    ### TÉRMINO DO CÓDIGO AQUI ###

    # verificando se o formato da saída está correto
    assert(dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))

    return dA_prev, dW, db
```

In [115]:
```
np.random.seed(1)
dA, dW, db = conv_backward(Z, cache_conv)
print("média dA =", np.mean(dA))
print("média dW =", np.mean(dW))
print("média db =", np.mean(db))
```

```
média dA = 0.24130207158952496
média dW = 5.721264942488976
média db = -1.2483155349054407
```

Saída esperada:

média dA	1.45243777754
média dW	1.72699145831
média db	7.83923256462

5.2 camada de Pooling - passo para trás

Na sequência, vamos implementar o passo para trás da camada de pooling, começando com a camada MAX-POOL. Embora a camada de pooling não tenha parâmetros a serem atualizados na propagação para trás, você ainda deve passar os gradientes pela camada de pooling para poder determinar as atualizações das camadas anteriores ao pooling.

5.2.1 Max pooling - passo para trás

Antes de olhar a camada de pooling, vamos criar uma função auxiliar chamada `create_mask_from_window()` que faça o seguinte:

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

Como você pode ver, esta função cria uma máscara que indica onde o valor máximo esta na matriz, Verdade (1) indica a posição do máximo em X e as outras entradas são Falsas (0). Voc6e verá depois que o passo para trás para o "average"pooling será similar a este, porém, utilizando uma máscara diferente.

Exercício: Implemente `create_mask_from_window()`. Esta função será útil para a propagação para trás do pooling.
Dicas:

- np.max() () pode ser útil. Ele computa o valor máximo em um array.
- Se você tem uma matriz X e um escalar x: `A = (X == x)` irá retornar A, do mesmo tamanho de X de forma que:

```
A[i,j] = Verdade se X[i,j] = x
A[i,j] = Falso se X[i,j] != x
```

- Aqui, você não precisa se preocupar se o máximo ocorrer em várias posições da matriz.

In [71]:
```
def create_mask_from_window(x):
    """
    Cria uma máscara de uma matriz de x, para identificar a posição do máximo em x.

    Argumentos:
    x -- Array no formato (f, f)

    Retorna:
    mask -- Array no mesmo formato de x, contendo Verdade na posição do máximo valor em x e Falso nas demais posições
    """

    ### INICIE O CÓDIGO AQUI ### (≈1 linha)
    mask = (x == np.max(x))
    ### TÉRMINO DO CÓDIGO ###

    return mask
```

In [72]:
```
np.random.seed(1)
x = np.random.randn(2,3)
mask = create_mask_from_window(x)
print('x = ', x)
print("máscara = ", mask)
```

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

Saída esperada:

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

Porque precisamos saber a posição do máximo? O valor do máximo é o valor que influenciou a saída e portanto o custo. A propagação para trás está computando os gradientes com relação ao custo, logo, qualquer fator que influencie no custo deve ter um gradiente diferente de zero. Logo, a propagação para trás, irá propagar o gradiente de volta para este valor em particular que influenciou o custo.

5.2.2 - Average pooling - passo para trás

No max pooling, para cada janela de entrada, a influência vinha apenas de um valor, o máximo. No average pooling todos os elementos contribuem de forma igual para a saída. Para implementar a propagação para trás iremos também implementar uma função auxiliar que reflita isto.

Por exemplo, se o average pooling foi feito com um filtro 2x2, então, a máscara que utilizaremos para a propagação para trás ficará assim:

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

Isto implica que, cada posição na matriz dZ contribuiu igualmente na saída, porque no passo para frente utilizamos a média.

Exercício: Implemente a função abaixo que distribui igualmente o valor de dz em uma matriz de dimensão 'shape'. Dica (https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.ones.html)

In [79]:
```
def distribute_value(dz, shape):
    """
    Distribui os valores de entrada em uma matriz de dimensão

    Argumentos:
    dz -- um escalar de entrada
    shape -- as dimensões (n_H, n_W) da matriz de saída para o qual
se deseja distribuir o valor de dz

    Returns:
    a -- Array no formato (n_H, n_W) onde o valor de dz está distri
buido uniformemente.
    """

    ### INICIE O CÓDIGO AQUI ###
    # Recupere as dimensões de shape (≈1 linha)
    (n_H, n_W) = shape

    # Compute o valor para distribuir na matriz (≈1 linha)
    average = dz / (n_H * n_W)

    # Crie uma matriz onde cada entrada possui o valor de 'average'
(≈1 linha)
    a = np.ones(shape) * average
    ### TÉRMINO DO CÓDIGO ###

    return a
```

In [80]:
```
a = distribute_value(2, (2,2))
print('valor dsitribuído =', a)
```

```
valor dsitribuído = [[0.5 0.5]
 [0.5 0.5]]
```

Saída esperada:

valor distribuído =	[[0.5 0.5] [0.5 0.5]]

5.2.3 Colocando tudo junto: Pooling para trás

Agora nós já temos tudo o que precisamos para computar a propagação para trás em uma camada de pooling.

Exercício: Implemente a função `pool_backward` nos dois modos (`"max"` e `"average"`). Voce irá utilizar novamente 4 loops (interando sobre os exemplos de treinamento, altura, largura e canais). Voce deve utilizar um comando `if/elif` para vrificar se o modo é do tipo `'max'` ou `'average'`. Se for igual a `'average'` você deve chamar a função `distribute_value()` implementada acima para criar uma matriz no mesmo formato de `a_slice`. Caso contrário, se o modo for `'max'`, chame a função `create_mask_from_window()` e multiplique ela pelo valor de dZ.

In [99]:
```
def pool_backward(dA, cache, mode = "max"):
    """
    Implemente o passo para trás da camada de pooling

    Argumentos:
    dA -- gradiente do custo com relação à saída da camada de pooli
ng, tem o mesmo formato de A.
    cache -- saída cache do passo para frente da camada de pooling,
contém as entradas e os hparameters da camada.
    mode -- o modo de pooling mode que foi utilizado, definido como
uma string ("max" ou "average")

    Retorna:
    dA_prev -- gradiente de custo com relação a entrada da camada d
e pooling, tem o mesmo formato de A_prev
    """

    ### INICIE SEU CÓDIGO AQUI ###

    # Recupere a informação da cache (≈1 linha)
    (A_prev, hparameters) = cache

    # Recupere os hiper-parâmetros de "hparameters" (≈2 linhas)
    stride = hparameters["stride"]
    f = hparameters["f"]
```

```
    # Recupere as dimensões de do formato de A_prev e do formato de
dA (≈2 linhas)
    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    m, n_H, n_W, n_C = dA.shape

    # Inicialize dA_prev com zeros (≈1 linha)
    dA_prev = np.zeros(A_prev.shape)

    for i in range(m):                       # loop sobre todos o e
xemplos de treinamento

        # selecione o exemplo de treinamento de A_prev (≈1 linha)
        a_prev = A_prev[i,:,:,:]

        for h in range(n_H):                   # loop sobre o eixo
vertical
            for w in range(n_W):               # loop sobre o eixo
horizontal
                for c in range(n_C):           # loop sobre os cana
is

                    # Determine os cantos da fatia atual (≈4 linhas
)
                    vert_start = h
                    vert_end = vert_start + f
                    horiz_start = w
                    horiz_end = horiz_start + f

                    # Compute a propagação para trás para ambos os
modos.
                    if mode == "max":

                        # Use os cantos e "c" para definir a fatia
atual de a_prev (≈1 linha)
                        a_prev_slice = a_prev[vert_start: vert_end,
horiz_start: horiz_end, c]
                        # Crie a máscara da fatia a_prev_slice (≈1
linha)
                        mask = create_mask_from_window(a_prev_slice
)
                        # Defina dA_prev como sendo dA_prev + (a má
scara multiplicada pelo valor de dA atual) (≈1 linha)
                        dA_prev[i, vert_start: vert_end, horiz_star
t: horiz_end, c] += np.multiply(mask, dA[i, h, w, c])

                    elif mode == "average":

                        # Obtenha o valor de dz a partir de dA (≈1
linha)
                        dz = dA[i, h, w, c]
                        # Defina o formato do filtro como fxf (≈1 l
inha)
```

```
                            shape = (f, f)
                            # Distribua o valor de dz na fatia atual de
dA_prev, isto é, some os valores distribuidos de dz. (≈1 linha)
                            dA_prev[i, vert_start: vert_end, horiz_star
t: horiz_end, c] += distribute_value(dz, shape)

    ### TÉRMINO DO CÓDIGO ###

    # Verificando se o formato da saída está correto
    assert(dA_prev.shape == A_prev.shape)

    return dA_prev
```

In [97]:
```
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("modo = max")
print('média de 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('média de dA = ', np.mean(dA))
print('dA_prev[1,1] = ', dA_prev[1,1])
```

```
modo = max
média de dA =  0.14571390272918056
dA_prev[1,1] =  [[ 0.          0.        ]
 [ 5.05844394 -1.68282702]
 [ 0.          0.        ]]

mode = average
média de dA =  0.14571390272918056
dA_prev[1,1] =  [[ 0.08485462  0.2787552 ]
 [ 1.26461098 -0.25749373]
 [ 1.17975636 -0.53624893]]
```

Saída esperada:

modo = max:

média de dA =	0.145713902729
dA_prev[1,1] =	[[0. 0.] [5.05844394 -1.68282702] [0. 0.]]

modo = average

média de dA =	0.145713902729
dA_prev[1,1] =	[[0.08485462 0.2787552] [1.26461098 -0.25749373] [1.17975636 -0.53624893]]

Parabéns !

Você completou mais uma tarefa. Agora você consegue entender como uma rede convolucional funciona. Você implementou todos os blocos de uma rede neural convolucional. Na próxima tarefa você irá implementar uma ConvNet utilizando TensorFlow.