CONVOLUCIÓN

</h1>

Objetivos

1. Aprender acerca de la convolución

2. Determinar el tamaño de la salida

3. Aprender acerca del stride y del zero padding

Tabla de contenido

In [1]:
```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
from scipy import ndimage, misc
```

Qué es la convolución

La convolución es una operación lineal similar a una ecuación lineal, producto punto o multiplicación de matrices. Presenta varias ventajas para analizar imágenes. Preserva la relación entre elementos y requiere menos parámetros que otros métodos.

Puede ver la relación entre los diferentes métodos aprendidos:

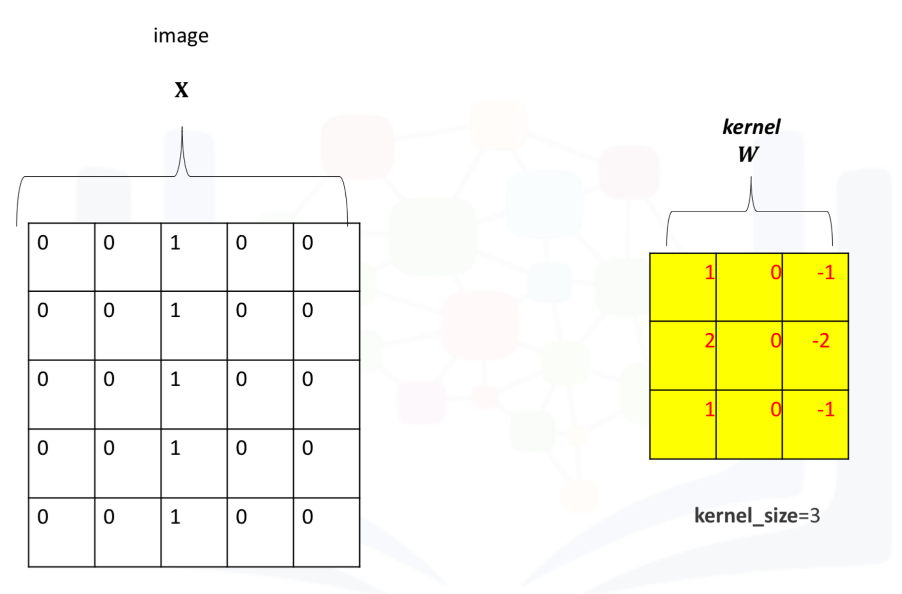
$$linear\ equation : y = wx + b$$

$$linear\ equation\ with\ multiple\ variables\ where\ \mathbf{x}\ is\ a\ vector\ \mathbf{y} = \mathbf{wx} + b$$

$$matrix\ multiplication\ where\ \mathbf{X}\ in\ a\ matrix\ \mathbf{y} = \mathbf{wX} + \mathbf{b}$$

$$convolution\ where\ \mathbf{X}\ and\ \mathbf{Y}\ is\ a\ tensor\ \mathbf{Y} = \mathbf{w} * \mathbf{X} + \mathbf{b}$$

En la convolución el parámetro w se llama kernel. Puede aplicar convolución sobre imágenes, donde X es la variable imagen y w el kernel.



Creamos un objeto convolución de 2 dimensiones usando el constructor Conv2D.

In [2]:
```
conv = nn.Conv2d(in_channels=1, out_channels=1,kernel_size=3)
conv
```

Out[2]:
```
Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1))
```

Como los parámetros en nn.Conv2D son inicializados aleatoriamente y se aprenden vía entrenamiento les damos algunos valores.

In [3]:
```
conv.state_dict()['weight'][0][0]=torch.tensor([[1.0,0,-1.0],[2.0,0,-2.0],[1.0,0.0,-1
conv.state_dict()['bias'][0]=0.0
conv.state_dict()
```

Out[3]:
```
OrderedDict([('weight',
              tensor([[[[ 1.,  0., -1.],
                        [ 2.,  0., -2.],
                        [ 1.,  0., -1.]]]])),
             ('bias', tensor([0.]))])
```

Creamos un tensor dummy para crear una imagen. La forma es (1,1,5,5) que corresponde a:

(número de entradas, número de canales, número de filas, número de columnas)

Establecemos la tercer columna en 1:

In [4]:
```
image=torch.zeros(1,1,5,5)
image[0,0,:,2]=1
image
```

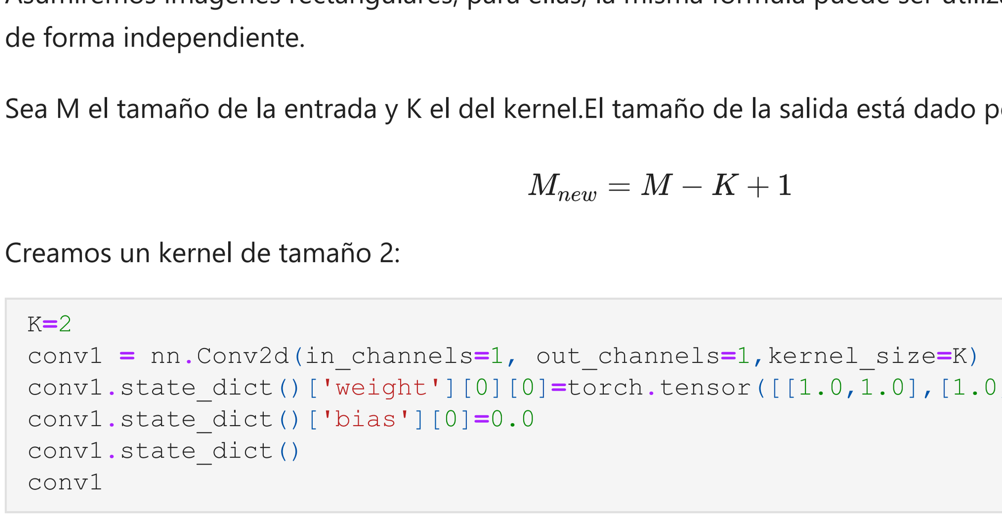
Out[4]:
```
tensor([[[[0., 0., 1., 0., 0.],
          [0., 0., 1., 0., 0.],
          [0., 0., 1., 0., 0.],
          [0., 0., 1., 0., 0.],
          [0., 0., 1., 0., 0.]]]])
```

Realizamos la convolución:

In [5]:
```
z=conv(image)
z
```

Out[5]:
```
tensor([[[[-4.,  0.,  4.],
          [-4.,  0.,  4.],
          [-4.,  0.,  4.]]]], grad_fn=<ThnnConv2DBackward>)
```

La animación siguiente ilustra el proceso.



Determinando el tamaño de la salida

Asumiremos imágenes rectangulares, para ellas, la misma fórmula puede ser utilizada en cada dimensión de forma independiente.

Sea M el tamaño de la entrada y K el del kernel.El tamaño de la salida está dado por la siguiente fórmula:

$$M_{new} = M - K + 1$$

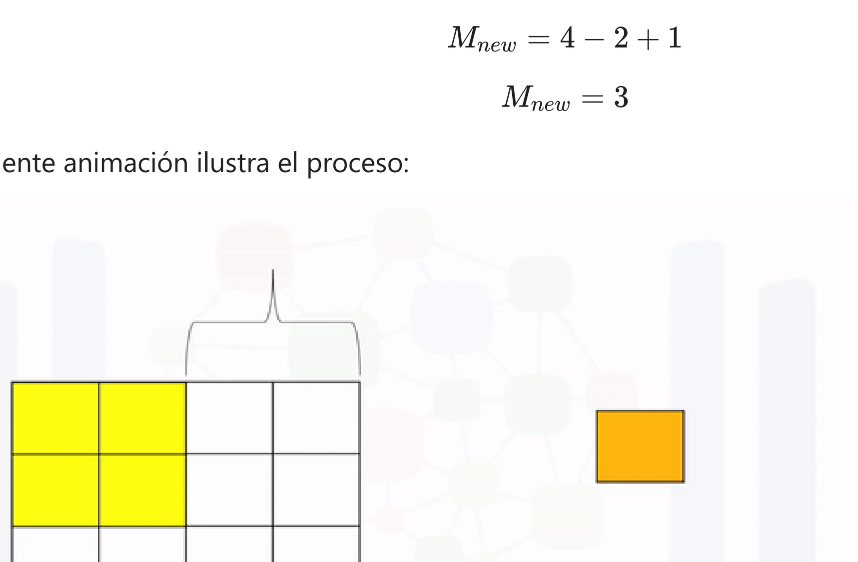
Creamos un kernel de tamaño 2:

In [6]:
```
K=2
conv1 = nn.Conv2d(in_channels=1, out_channels=1,kernel_size=K)
conv1.state_dict()['weight'][0][0]=torch.tensor([[1.0,1.0],[1.0,1.0]])
conv1.state_dict()['bias'][0]=0.0
conv1.state_dict()
conv1
```

Out[6]:
```
Conv2d(1, 1, kernel_size=(2, 2), stride=(1, 1))
```

Creamos una imagen de tamaño 2:

In [7]:
```
M=4
image1=torch.ones(1,1,M,M)
```



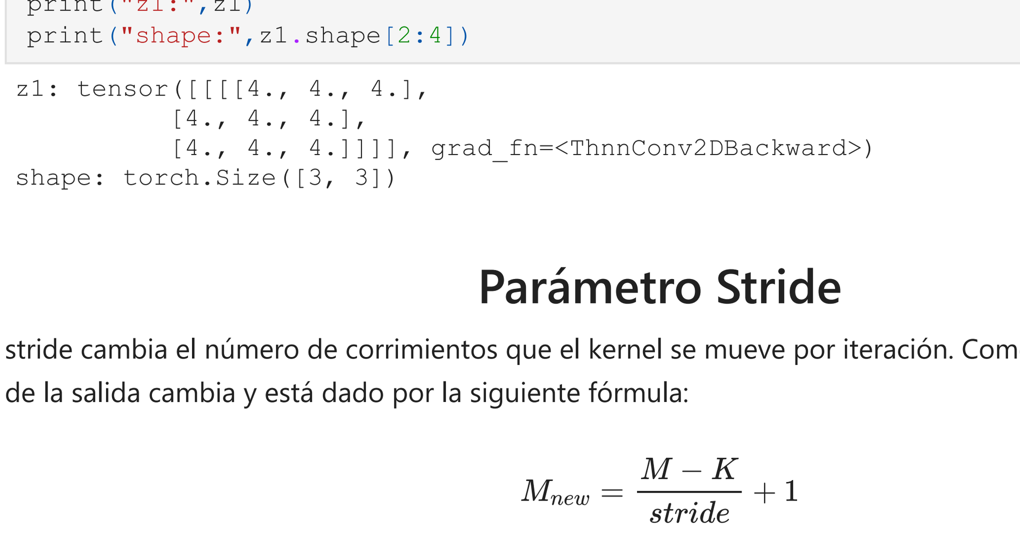
El tamaño de la salida es:

$$M_{new} = M - K + 1$$

$$M_{new} = 4 - 2 + 1$$

$$M_{new} = 3$$

La siguiente animación ilustra el proceso:



Realizamos la convolución y verificamos el tamaño:

In [8]:
```
z1=conv1(image1)
print("z1:",z1)
print("shape:",z1.shape[2:4])
```

```
z1: tensor([[[[4., 4., 4.],
          [4., 4., 4.],
          [4., 4., 4.]]]], grad_fn=<ThnnConv2DBackward>)
shape: torch.Size([3, 3])
```

Parámetro Stride

stride cambia el número de corrimientos que el kernel se mueve por iteración. Como resultado, el tamaño de la salida cambia y está dado por la siguiente fórmula:

$$M_{new} = \frac{M - K}{stride} + 1$$

Creamos un objeto convolución con stride = 2:

In [9]:
```
conv3 = nn.Conv2d(in_channels=1, out_channels=1,kernel_size=2,stride=2)

conv3.state_dict()['weight'][0][0]=torch.tensor([[1.0,1.0],[1.0,1.0]])
conv3.state_dict()['bias'][0]=0.0
conv3.state_dict()
```

Out[9]:
```
OrderedDict([('weight',
              tensor([[[[1., 1.],
                        [1., 1.]]]])),
             ('bias', tensor([0.]))])
```

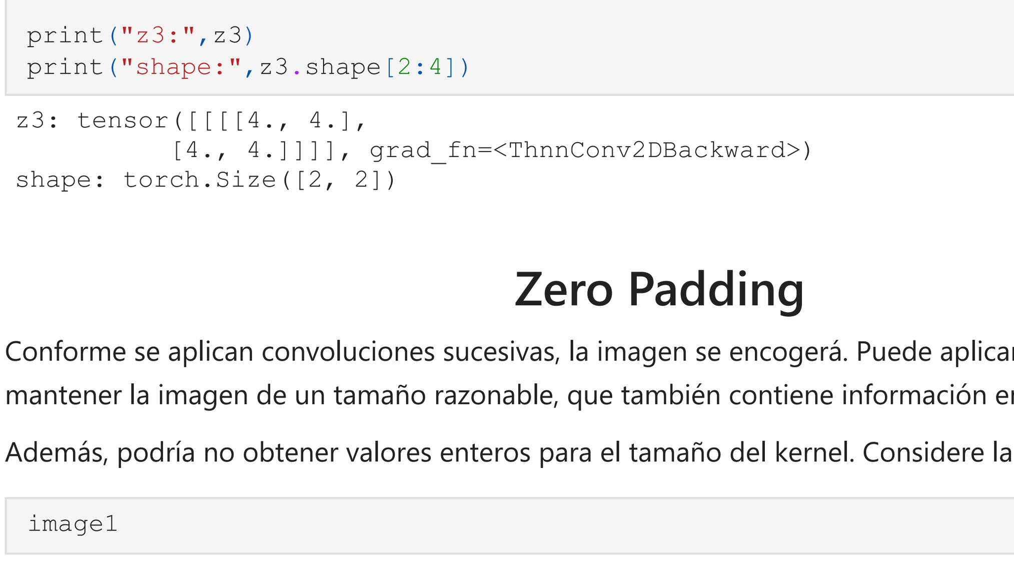
Para una imagen con tamaño 4 calculamos el tamaño de la salida:

$$M_{new} = \frac{M - K}{stride} + 1$$

$$M_{new} = \frac{4 - 2}{2} + 1$$

$$M_{new} = 2$$

La siguiente animación ilustra el proceso:



Realizamos la convolcuión y verificamos el tamaño:

In [10]:
```
z3=conv3(image1)

print("z3:",z3)
print("shape:",z3.shape[2:4])
```

```
z3: tensor([[[[4., 4.],
          [4., 4.]]]], grad_fn=<ThnnConv2DBackward>)
shape: torch.Size([2, 2])
```

Zero Padding

Conforme se aplican convoluciones sucesivas, la imagen se encogerá. Puede aplicarse el zero padding para mantener la imagen de un tamaño razonable, que también contiene información en los bordes.

Además, podría no obtener valores enteros para el tamaño del kernel. Considere la siguiente imagen:

In [11]:
```
image1
```

Out[11]:
```
tensor([[[[1., 1., 1., 1.],
          [1., 1., 1., 1.],
          [1., 1., 1., 1.],
          [1., 1., 1., 1.]]]])
```

Si kernel_size=2 y stride=1 se tiene que:

$$M_{new} = \frac{M - K}{stride} + 1$$

$$M_{new} = \frac{4 - 2}{3} + 1$$

$$M_{new} = 1.666$$

In [12]:
```
conv4 = nn.Conv2d(in_channels=1, out_channels=1,kernel_size=2,stride=3)
conv4.state_dict()['weight'][0][0]=torch.tensor([[1.0,1.0],[1.0,1.0]])
conv4.state_dict()['bias'][0]=0.0
conv4.state_dict()
z4=conv4(image1)
print("z4:",z4)
print("z4:",z4.shape[2:4])
```

```
z4: tensor([[[[4.]]]], grad_fn=<ThnnConv2DBackward>)
z4: torch.Size([1, 1])
```

Puede agregar filas y columnas de ceros alrededor de la imagen. Esto se llama padding (relleno). En el constructor Conv2D puede especificar el número de filas o columnas que quiere agregar mediante el parámetro padding.

Para una imagen cuadrada, simplemente rellena una columna extra de ceros en la primera y la última columna. Repite el proceso para las filas.Así, para una imagen cuadrada, el ancho y alto son los originales sumados a 2 veces el número de elementos de padding especificados. Puede determinar el tamaño de la salida luego de subsecuentes operaciones como se muestra en la ecuación siguiente, donde se determina el tamaño de la imagen luego del padding y de aplicar un kernel de tamaño K:

$$M' = M + 2 \times padding$$

$$M_{new} = M' - K + 1$$

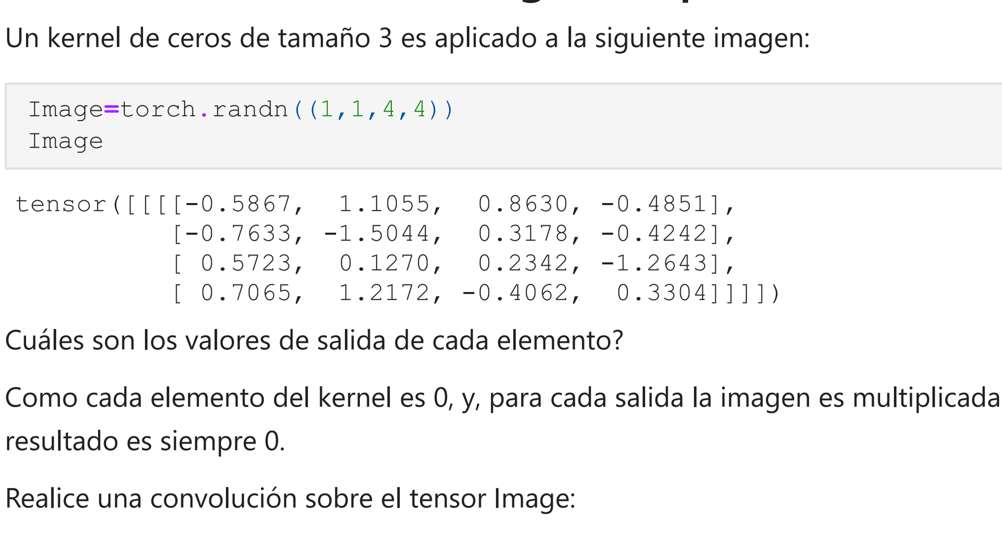
Considere el ejemplo siguiente:

In [13]:
```
conv5 = nn.Conv2d(in_channels=1, out_channels=1,kernel_size=2,stride=3,padding=1)

conv5.state_dict()['weight'][0][0]=torch.tensor([[1.0,1.0],[1.0,1.0]])
conv5.state_dict()['bias'][0]=0.0
conv5.state_dict()
z5=conv5(image1)
print("z5:",z5)
print("z5:",z4.shape[2:4])
```

```
z5: tensor([[[[1., 2.],
          [2., 4.]]]], grad_fn=<ThnnConv2DBackward>)
z5: torch.Size([1, 1])
```

El proceso se resume en la siguiente animación:



Preguntas prácticas

Un kernel de ceros de tamaño 3 es aplicado a la siguiente imagen:

In [14]:
```
Image=torch.randn((1,1,4,4))
Image
```

Out[14]:
```
tensor([[[[-0.5867,  1.1055,  0.8630, -0.4851],
          [-0.7633, -1.5044,  0.3178, -0.4242],
          [ 0.5723,  0.1270,  0.2342, -1.2643],
          [ 0.7065,  1.2172, -0.4062,  0.3304]]]])
```

Cuáles son los valores de salida de cada elemento?

Como cada elemento del kernel es 0, y, para cada salida la imagen es multiplicada por el kernel, el resultado es siempre 0.

Realice una convolución sobre el tensor Image:

In [15]:
```
conv = nn.Conv2d(in_channels=1, out_channels=1,kernel_size=3)
conv.state_dict()['weight'][0][0]=torch.tensor([[0,0,0],[0,0,0],[0,0.0,0]])
conv.state_dict()['bias'][0]=0.0
#
conv(Image)
```

Out[15]:
```
tensor([[[[0., 0.],
          [0., 0.]]]], grad_fn=<ThnnConv2DBackward>)
```

Tiene una imagen de tamaño 4. Se tiene kernel_size = 2, stride = 2. Cuál es el tamaño de la salida?

(M-K)/stride +1 (4-2)/2 +1 2