Laboratorio 2

Bienvenidos de nuevo al segundo laboratorio de Deep Learning y Sistemas inteligentes. Espero que este laboratorio sirva para consolidar sus conocimientos del tema de Redes Neuronales Convolucionales.

Este laboratorio consta de dos partes. En la primera trabajaremos una Red Neuronal Convolucional paso-a-paso. En la segunda fase, usaremos PyTorch para crear una nueva Red Neuronal Convolucional, con la finalidad de que no solo sepan que existe cierta función sino también entender qué hace en un poco más de detalle.

Para este laboratorio estaremos usando una herramienta para Jupyter Notebooks que facilitará la calificación, no solo asegurándo que ustedes tengan una nota pronto sino también mostrandoles su nota final al terminar el laboratorio.

Espero que esta vez si se muestren los *marks*. De nuevo me discupo si algo no sale bien, seguiremos mejorando conforme vayamos iterando. Siempre pido su comprensión y colaboración si algo no funciona como debería.

Al igual que en el laboratorio pasado, estaremos usando la librería de Dr John Williamson et al de la University of Glasgow, además de ciertas piezas de código de Dr Bjorn Jensen de su curso de Introduction to Data Science and System de la University of Glasgow para la visualización de sus calificaciones.

```
# # Una vez instalada la librería por favor, recuerden volverla a comentar.
# !pip install -U --force-reinstall --no-cache https://github.com/johnhw/jhwutils/zipball/master
# !pip install scikit-image
```

```
import numpy as np
import copy
import matplotlib.pyplot as plt
import scipy
from PIL import Image
import os
#from IPython import display
#from base64 import b64decode


# Other imports
from unittest.mock import patch
from uuid import getnode as get_mac

from jhwutils.checkarr import array_hash, check_hash, check_scalar, check_string
import jhwutils.image_audio as ia
import jhwutils.tick as tick

###
tick.reset_marks()

%matplotlib inline
```

```
# Seeds
seed_ = 2023
np.random.seed(seed_)
```

```
# Hidden cell for utils needed when grading (you can/should not edit this)
# Celda escondida para utlidades necesarias, por favor NO edite esta celda

```

Información del estudiante en dos variables

- carne_1 : un string con su carne (e.g. "12281"), debe ser de al menos 5 caracteres.
- firma_mecanografiada_1: un string con su nombre (e.g. "Albero Suriano") que se usará para la declaracion que este trabajo es propio (es decir, no hay plagio)
- carne_2 : un string con su carne (e.g. "12281"), debe ser de al menos 5 caracteres.
- firma_mecanografiada_2: un string con su nombre (e.g. "Albero Suriano") que se usará para la declaracion que este trabajo es propio (es decir, no hay plagio)

```
# carne_1 =
# firma_mecanografiada_1 =
# carne_2 =
# firma_mecanografiada_2 =
# YOUR CODE HERE
carne_1 = "20117";
firma_mecanografiada_1 = "Yongbum Park" ;
```

```
8 carne_2 = "20679";
9 firma_mecanografiada_2 = "Oscar Fernando Lopez";
```

```
1 # Deberia poder ver dos checkmarks verdes [0 marks], que indican que su información básica está OK
2
3 with tick.marks(0):
4     assert(len(carne_1)>=5 and len(carne_2)>=5)
5
6 with tick.marks(0):
7     assert(len(firma_mecanografiada_1)>0 and len(firma_mecanografiada_2)>0)
```

✓ [0 marks]

✓ [0 marks]

Dataset a Utilizar

Para este laboratorio seguriemos usando el dataset de Kaggle llamado Cats and Dogs image classification. Por favor, descarguenlo y ponganlo en una carpeta/folder de su computadora local.

Parte 1 - Construyendo una Red Neuronal Convolucional

Créditos: La primera parte de este laboratorio está tomado y basado en uno de los laboratorios dados dentro del curso de "Convolutional Neural Networks" de Andrew Ng

Muchos framework en la actualidad hacen que las operaciones de convolución sean fáciles de usar, pero no muchos entienden realmente este concepto, que es uno de los más interesantes de entender en Deep Learning. Una capa convolucional transforma el volumen de un input a un volumen de un output que es de un tamaño diferente.

En esta sección, ustedes implementaran una capa convolucional paso a paso. Primero empezaremos por hacer unas funciones de padding con ceros y luego otra para computar la convolución.

Algo muy importante a **notar** es que para cada función *forward*, hay una equivalente en *backward*. Por ello, en cada paso de su modulo de forward, deberán guardar algunos datos que se usarán durante el cálculo de gradientes en el backpropagation

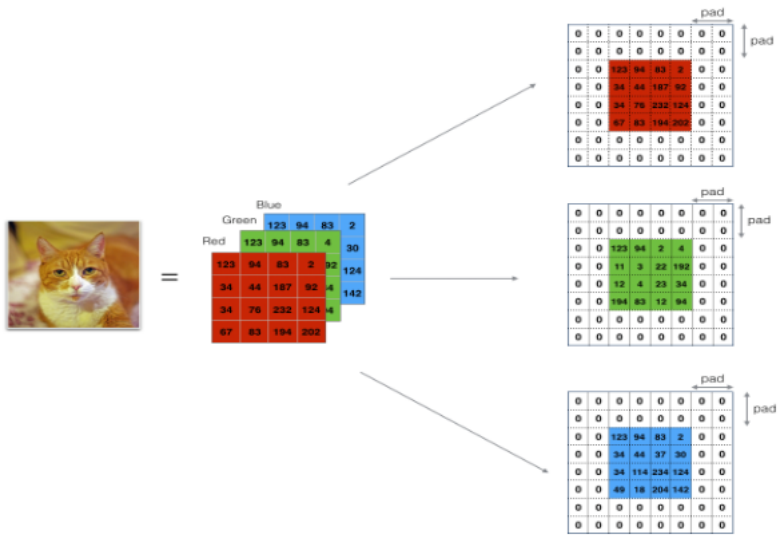
Ejercicio 1

Ahora construiremos una función que se encargue de hacer *padding*, que como vimos en la clase es hacer un tipo de marco sobre la imagen. Este "marco" suele ser de diferentes tipos que lo que debe buscarse es que no tengan significacia dentro de la imagen, usualmente es cero, pero puede ser otro valor que no afecte en los cálculos.

Para este laboratorio, usaremos cero, y en este caso se le suele llamar *zero-padding* el cual agrega ceros alrededor del borde de la imagen.

Algo interesante a notar, es que este borde se agrega sobre cada uno de los canales de color de la imagen. Es decir, en una imagen RGB se agregará sobre la matriz de rojos, otro sobre la matriz de verdes y otro más sobre la matriz de azules.

Como se puede ver en la siguiente imagen.



Crédito de imagen al autor, imagen tomada del curso "Convolutional Neural Networks" de Andrew Ng

Recordemos que el agregar padding nos permite:

- Usar una capa convolucional sin necesariamente reducir el alto y ancho de los volumenes de entrada. Esto es importante para cuando se crean modelos/redes profundas, dado que de esta manera evitamos reducir demasiado la entrada mientras se avanza en profundidad.
- Ayuda a obtener más información de los bordes de la imagen. Sin el padding, muy pocos valores serán afectados en la siguiente capa por los pixeles de las orillas

Ahora sí, el **ejercicio** como tal:

Implemente la siguiente función, la cual agregará el padding de ceros a todas las imagenes de un grupo (batch) de tamaño X. Para eso se usará *np.pad*.

Nota: Si se quiere agregar padding a un array "a" de tamaño (5,5,5,5,5) con un padding de tamaño diferente para cada dimensión, es decir, pad=1 para la segunda dimensión, pad=3 para la cuarta dimensión, y pad=0 para el resto, esto se puede hacer de la siguiente manera

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

```
def zero_pad(X, pad):
    """
    Agrega padding de ceros a todas las imagenes en el dataset X. El padding es aplicado al alto y ancho de una imagen,
    como se mostró en la figura anterior.

    Argument:
    X: Array (m, n_H, n_W, n_C) representando el batch de imagens
    pad: int, cantidad de padding

    Returns:
    X_pad: Imagen con padding agregado, (m, n_H + 2*pad, n_W + 2*pad, n_C)
    """

    # Aprox 1 línea de código
    # X_pad =
    # YOUR CODE HERE
    X_pad = np.pad(X, ((0, 0), (pad, pad), (pad, pad), (0, 0)), mode='constant', constant_values=0)
    # raise NotImplementedError()

    return X_pad
```

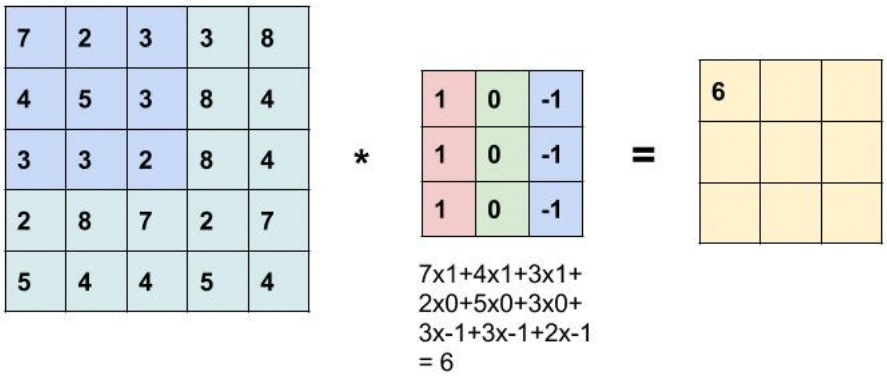
```
np.random.seed(seed_)
x = np.random.randn(4, 3, 3, 2)
x_pad = zero_pad(x, 2)

print ("x.shape =\n", x.shape)
print ("x_pad.shape =\n", x_pad.shape)
print ("x[1,1] =\n", x[1,1])
print ("x_pad[1,1] =\n", x_pad[1,1])

# Mostrar imagen

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])


with tick.marks(5):
    assert(check_hash(x_pad, ((4, 7, 7, 2), -1274.231087426035)))
```

```
x.shape =
 (4, 3, 3, 2)
x_pad.shape =
 (4, 7, 7, 2)
x[1,1] =
 [[ 0.64212494 -0.18117553]
 [ 0.77174916  0.74152348]
 [ 1.32476273  0.43928671]]
x_pad[1,1] =
 [[0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]
 [0. 0.]]
```

Ejercicio 2

Ahora, es momento de implementar un solo paso de la convolución, en esta ustedes aplicaran un filtro/kernel a una sola posición del input. Esta será usada para construir una unidad convolucional, la cual:

- Tomará una matriz (volumen) de input
- Aplicará un filtro a cada posición del input
- Sacará otra matriz (volumen) que será usualmente de diferente tamaño



Crédito de la imagen al autor. Tomada de https://medium.datadriveninvestor.com/convolutional-neural-networks-3b241a5da51e

En la anterior imagen, estamos viendo un filtro de 3x3 con un stride de 1 (recuerden que stride es la cantidad que se mueve la ventana). Además, lo que usualmente se hace con esta operación es una **multiplicación element-wise** (en clase les dije que era un producto punto, pero realmente es esta operación), para luego sumar la matriz y agregar un bias. Ahora, primero implementaran un solo paso de la convolución en el cual deberán aplicar un filtro a una sola posición y obtendrán un flotante como salida.

Ejercicio: Implemente la función conv_single_step()

Probablemente necesite esta función

Considre que la variable "b" será pasada como un numpy.array. Se se agrega un escalar (flotante o entero) a un np.array, el resultado será otro np.array. En el caso especial de cuando un np.array contiene un solo valor, se puede convertir a un flotante

```
def conv_single_step(a_slice_prev, W, b):
    """
    Aplica un filtro definido en el parámetro W a un solo paso de
    slice (a_slice_prev) de la salida de activación de una capa previa

    Arguments:
    a_slice_prev: Slice shape (f, f, n_C_prev)
    W: Pesos contenidos en la ventana. Shape (f, f, n_C_prev)
    b: Bias contenidos en la ventana. Shape (1, 1, 1)

    Returns:
    Z: Un escalar, resultado de convolving la ventana (W, b)
```

```
    """

    # Aprox 2-3 lineas de codigo
    # Multiplicación element-wise
    # Z =
    # YOUR CODE HERE
    # raise NotImplementedError()
    s = a_slice_prev * W
    Z = np.sum(s)
    Z = Z + float(b)

    return Z
```

```
np.random.seed(seed_)
a_slice_prev = np.random.randn(4, 3, 3)
W = np.random.randn(4, 3, 3)
b = np.random.randn(1, 1, 1)
Z = conv_single_step(a_slice_prev, W, b)
print("Z =", Z)

with tick.marks(5):
    assert check_scalar(Z, '0x92594c5b')
```

```
Z = 17.154767154043057
C:\Users\omen\AppData\Local\Temp\ipykernel_29328\3930102447.py:22: DeprecationWarning: Conversion of an array with ndim > 0 to a sc
  Z = Z + float(b)
```

✓ [5 marks]

Ejercicio 3

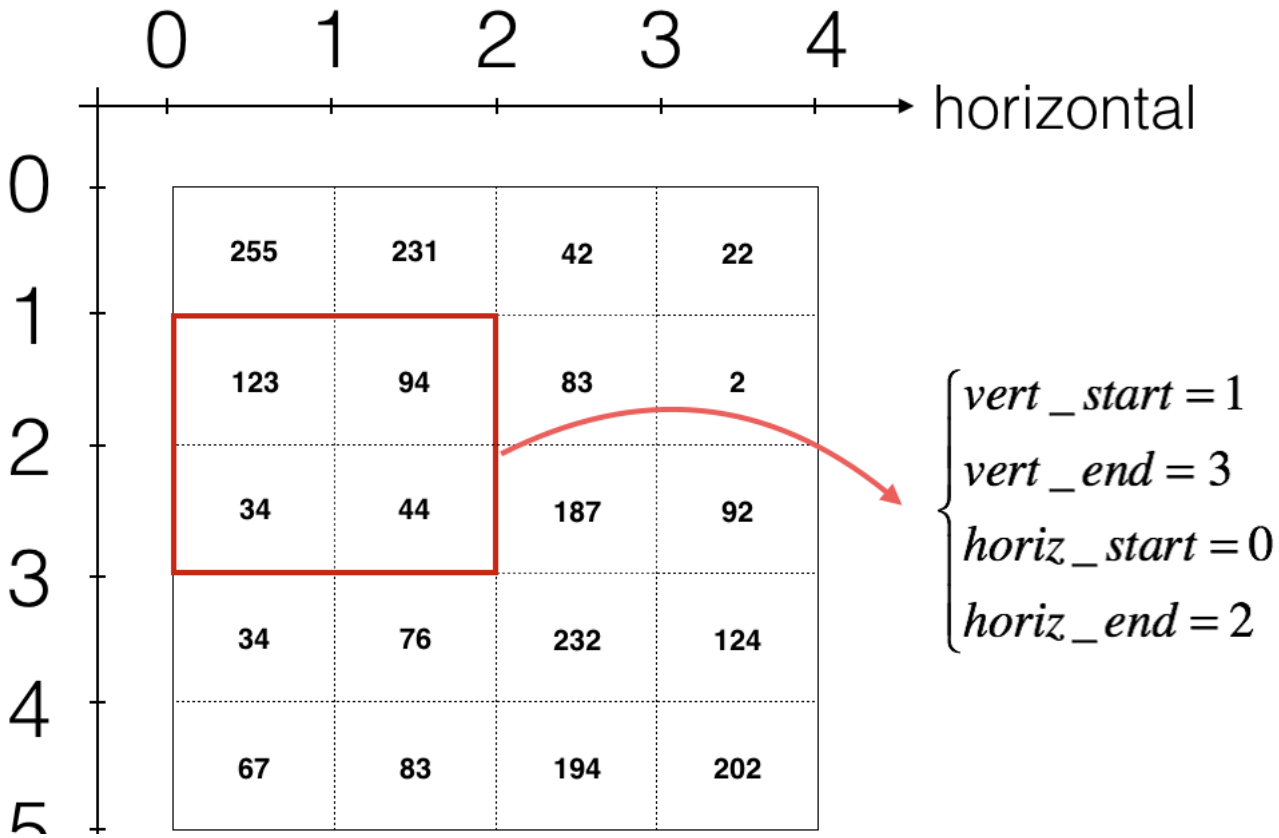
Ahora pasaremos a construir el paso de forward. En este, se tomará muchos filtros y los convolucionaran con los inputs. Cada "convolución" les dará como resultado una matriz 2D, las cuales se "stackearan" en una salida que será entonces de 3D.

Ejercicio: Implemente la función dada para convolucionar los filtros "W" con el input dado "A_prev". Esta función toma los siguientes inputs:

- A_prev, la salida de las activaciones de la capa previa (para un batch de m inputs)
- W, pesos. Cada uno tendra un tamaño de fxf
- b, bias, donde cada filtro tiene su propio bias (uno solo)
- hparameters, hiperparámetros como stride y padding

Considere lo siguiente: a. Para seleccionar una ventana (slice) de 2x2 en la esquina superior izquierda de una matriz a_prev, deberian hacer algo como `a_slice_prev = a_prev[0:2, 0:2, :]` Noten como esto da una salida 3D, debido a que tiene alto, ancho (de 2) y profundo (de 3 por los canales RGB). Esto le puede ser de utilidad cuando defina `a_slide_prev` en la función, usando los índices de `start/end`.

b. Oara definir `a_slice` necesitará primero definir las esquinas `vert_start, vert_end, horiz_start, horiz_end`. La imagen abajo puede resultar útil para entender como cada esquina puede ser definida usando h,w,f y s en el código.




```
def conv_forward(A_prev, W, b, hparameters):
    """
    Implementa la parte de forward propagation para una función de convolución

    Arguments:
    A_prev: Salida de activación de la capa previa layer,
            Shape (m, n_H_prev, n_W_prev, n_C_prev)
    W: Pesos, shape (f, f, n_C_prev, n_C)
    b: Biases, shape (1, 1, 1, n_C)
    hparameters: Dictionario  con "stride" y "pad"

    Returns:
    Z:  conv output, shape (m, n_H, n_W, n_C)
    cache: cache de valores necesarios para conv_backward()
    """

    # Aprox 1 linea para:
    # Obtener las dimensiones de A_prev
    # (m, n_H_prev, n_W_prev, n_C_prev) =
    # YOUR CODE HERE
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    # raise NotImplementedError()

    # Aprox 1 linea para:
    # Obtener las dimensiones de W
    # (f, f, n_C_prev, n_C) =
    # YOUR CODE HERE
    (f, f, n_C_prev, n_C) = W.shape
    # raise NotImplementedError()

    # Aprox 2 lineas para:
    # Obtener datos de "hparameters"
    # stride =
    # pad =
    # YOUR CODE HERE
    stride = hparameters["stride"]
    pad = hparameters["pad"]
    # raise NotImplementedError()

    # Hint: use int() to apply the 'floor' operation. (≈2 lines)
    # Aprox 2 lineas para:
    # Calcular las dimensiones del Conv output, usando las formulas dadas arriba
    # n_H =
    # n_W
    # YOUR CODE HERE
    n_H = int((n_H_prev - f + 2 * pad) / stride) + 1
    n_W = int((n_W_prev - f + 2 * pad) / stride) + 1
    # raise NotImplementedError()
```

```
    # Aprox 1 linea para
    # Iniciarlizar el volumen de salida Z con ceros
    # Z =
    # YOUR CODE HERE
    Z = np.zeros((m, n_H, n_W, n_C))
    # raise NotImplementedError()

    # Creamos A_prev_pad al agregar padding a A_prev
    A_prev_pad = zero_pad(A_prev, pad)

    # En este bloque de codigo deberan:
    # 1. Iterar sobre el batch de entrenamiento (m)
    # 2. Obtener a_prev_pad, para seleccionar el ith ejemplo con padding de entrenamiento
    # 3. Iterar sobre el axis vertical de la salida (n_H)
    # 4 y 5. Encontrar el inicio y final vertical de la ventana actual (slice)
    #        Es decir:
    #        vert_start =
    #        vert_end =
    # 6. Iterar sobre el axis horizontal de la salida (n_W)
    # 7 y 8. Encontrar el inicio y final horizontal de la ventana actual (slice)
    #        Es decir:
    #        horiz_start =
    #        horiz_end =
    # 9. Iterar sobre los canales (= # filtros) de la salida (n_C)
    # 10. Usar las esquinas para definir el slice (3D) de a_prev_pad.
    #     a_slice_prev =
    # 11. Convolucione el slice (3D) con el filtro correcto W, y bias b, para regresar la salida
    #     weights =
    #     biases =
    #     Z[i, h, w, c] =
    # YOUR CODE HERE

    for i in range(m):
        a_prev_pad = A_prev_pad[i, :, :, :]

        for h in range(n_H):
            vert_start = h * stride
            vert_end = vert_start + f

            for w in range(n_W):
                horiz_start = w * stride
                horiz_end = horiz_start + f

                for c in range(n_C):
                    a_slice_prev = a_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :]

                    weights = W[:, :, :, c]
                    biases = b[:, :, :, c]
                    Z[i, h, w, c] = conv_single_step(a_slice_prev, weights, biases)
    # raise NotImplementedError()

    # Asegurandose que la salida sea con la forma correcta
    assert(Z.shape == (m, n_H, n_W, n_C))

    # Guardando información en el "cache" para el backpropagation
    cache = (A_prev, W, b, hparameters)

    return Z, cache
```

```
np.random.seed(seed_)
A_prev = np.random.randn(10,7,7,5)
W = np.random.randn(3,3,5,8)
b = np.random.randn(1,1,1,8)
hparameters = {"pad" : 1,
               "stride": 1}

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

with tick.marks(5):
    assert check_hash(Z, ((10, 7, 7, 8), 2116728.6653762255))

with tick.marks(5):
    assert check_hash(cache_conv[0], ((10, 7, 7, 5), -12937.39104655015))

with tick.marks(5):
    assert check_scalar(np.mean(Z), '0xb416d11a')
```

```
Z's mean =
 0.3337664511415829
Z[3,2,1] =
 [ 4.1749337   9.00045401  1.79056239 -2.293963   -0.5189402  10.78547069
  1.42173371 13.11009042]
cache_conv[0][1][2][3] =
 [ 1.88596049  0.30974852 -0.3170466   0.481606   -0.43747686]
C:\Users\omen\AppData\Local\Temp\ipykernel_29328\3930102447.py:22: DeprecationWarning: Conversion of an array with ndim > 0 to a sc
  Z = Z + float(b)
```

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

También deberíamos agregar una función de activación a la salida de la forma, que teniendo al salida Z

```
Z[i, h, w, c] = ...
```

Deberiamos aplicar la activación de forma que:

```
A[i, h, w, c] = activation(Z[i, h, w, c])
```

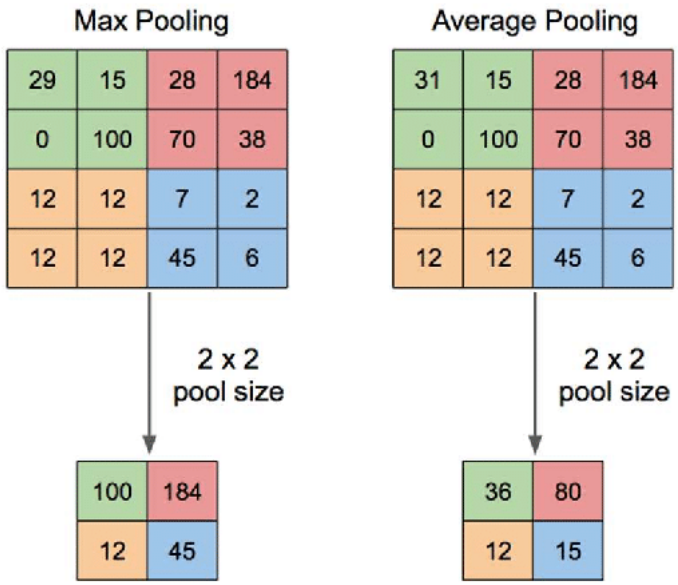
Pero esto no lo haremos acá

Ejercicio 4

Ahora lo que necesitamos es realizar la parte de "Pooling", la cual reducirá el alto y ancho del input. Este ayudará a reducir la complejidad computacional, así como también ayudará a detectar features más invariantes en su posición del input. Recuerden que hay dos tipos más comunes de pooling.

- Max-pooling: Mueve una ventana de (fxf) sobre un input y guarda el valor máximo de cada ventana en su salida
- Average-pooling: Mueve una ventana de (fxf) sobre un input y guarda el valor promedio de cada ventana en su salida

Estas capas de pooling no tienen parámetros para la parte de backpropagation al entrenar. Pero, estas tienen hiperparámetros como el tamaño de la ventana (f). Este especifica el alto y ancho de la ventana.



Crédito de imagen al autor, imagen tomada dehttps://www.researchgate.net/figure/Illustration-of-Max-Pooling-and-Average-Pooling-Figure-2-above-shows-an-example-of-max_fig2_333593451

Ejercicio: Implemente la función del paso forwarding de la capa de la capa de pooling.

Considere que como no hay padding, las formulas para los tamaños del output son:

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

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

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

```python
def pool_forward(A_prev, hparameters, mode = "max"):
    """
    Implementa el paso forward de una capa de pooling

    Arguments:
    A_prev: Input shape (m, n_H_prev, n_W_prev, n_C_prev)
    hparameters: Diccionario con "f" and "stride"
    mode: String para el modo de pooling a usar ("max" or "average")

    Returns:
    A: Salida shape (m, n_H, n_W, n_C)
    cache: Cache usado en el backward, contiene input e hiperparametros
    """




    # Obtenemos las dimensiones del input
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Obtenemos los hiperparametros
    f = hparameters["f"]
    stride = hparameters["stride"]

    # Definimos las dimensiones de salida
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    # Init la matriz de salida A
    A = np.zeros((m, n_H, n_W, n_C))

    # En este bloque de codigo debera;
    # 1. Iterar sobre los ejemplos de entrada (m)
    # 2. Iterar sobre los el eje vertical de la salida (n_H)
    # 3. Encontrar el inicio y fin vertical de la ventana actual
    #    vert_start =
    #    vert_end =
    # 4. Iterar sobre el eje horizontal de la salida (n_W)
    # 5. Encontrar el inicio y fin horizontal de la ventana actual
    #    horiz_start =
    #    horiz_end =
    # 6. Iterar sobre los canales de salida (n_C)
    # 7. Encontrar las orillas para definir el slice(ventana) actual en el ith ejemplo de training de A_prev, canal c
    #    a_prev_slice =
    # 8. Calcular el pooling dependiendo del modo (mode) - Use np.max y np.mean
    #    A[i, h, w, c] =
    # YOUR CODE HERE
    for i in range(m):
        for h in range(n_H):
            vert_start = h * stride
            vert_end = vert_start + f

            for w in range(n_W):
                horiz_start = w * stride
                horiz_end = horiz_start + f

                for c in range(n_C):
                    a_prev_slice = A_prev[i, vert_start:vert_end, horiz_start:horiz_end, c]

                    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)
    # raise NotImplementedError()

    # Guardar el input e hiperparametos en el "cache" para el pool_backward()
    cache = (A_prev, hparameters)

    # Asegurarse que la salida tiene la forma correcta
    assert(A.shape == (m, n_H, n_W, n_C))

    return A, cache
```

```python
np.random.seed(seed_)
A_prev = np.random.randn(2, 5, 5, 3)
hparameters = {"stride" : 1, "f": 3}

A, cache = pool_forward(A_prev, hparameters)
with tick.marks(5):
    assert check_hash(A, ((2, 3, 3, 3), 2132.191781663462))
```

```
print("mode = max")
print("A.shape = " + str(A.shape))
print("A =\n", A)
print()
A, cache = pool_forward(A_prev, hparameters, mode = "average")
with tick.marks(5):
    assert check_hash(A, ((2, 3, 3, 3), -14.942132313028413))
print("mode = average")
print("A.shape = " + str(A.shape))
print("A =\n", A)
```

✓ [5 marks]

```
mode = max
A.shape = (2, 3, 3, 3)
A =
 [[[[2.65440726 2.09732919 0.89256196]
   [2.65440726 2.09732919 0.89256196]
   [2.65440726 1.44060519 0.77174916]]

  [[1.5964877  2.39887598 0.89256196]
   [1.5964877  2.39887598 0.89256196]
   [1.5964877  2.39887598 0.77174916]]

  [[1.5964877  2.39887598 1.23583026]
   [1.5964877  2.39887598 1.36481958]
   [1.5964877  2.39887598 1.36481958]]]


 [[[1.84392163 2.43182155 1.29498747]
   [1.84392163 1.84444871 1.29498747]
   [0.6411132  1.84444871 1.16961103]]

  [[1.05650003 1.83680466 1.29498747]
   [1.05650003 0.9194503  1.29498747]
   [1.60523445 0.9194503  1.16961103]]

  [[1.06265456 1.83680466 0.68596995]
   [1.84881089 0.9194503  0.91014646]
   [1.84881089 1.42664748 1.06769765]]]]
```

✓ [5 marks]

```
mode = average
A.shape = (2, 3, 3, 3)
A =
 [[[[-0.03144582  0.21101766 -0.4691968 ]
   [-0.19309428  0.11749016 -0.32066469]
   [ 0.03682201  0.07413032 -0.36460992]]

  [[-0.58916194  0.45332745 -0.92209295]
   [ 0.01933338  0.23001555 -0.80282417]
   [ 0.33096648 -0.05773358 -0.55515521]]

  [[-0.19306801  0.61727733 -0.75579122]
   [ 0.34757347  0.47452468 -0.55854075]
   [ 0.52805193 -0.10908417 -0.5041339 ]]]


 [[[ 0.41867593  0.27110615  0.24018433]
   [-0.08325311  0.13111052  0.36317349]
   [-0.35974293 -0.13195187  0.30872263]]

  [[ 0.13066225  0.09595298 -0.31301579]
   [-0.36030628 -0.08070726  0.1281678 ]
   [-0.190839   -0.07153563  0.25708761]]
```

¡Muy bien terminamos la parte del paso forward!

Ejercicio 5

Antes de empezar con el ejercicio 5, debemos clarificar unas cuantas cosas.

Por ello, es momento de pasar a hacer el paso de backward propagation. En la mayoría de frameworks/librerías de la actualidad, solo deben implementarse el paso forward, y estas librerías se encargan de hacer el paso de backward. El backward puede ser complicado para una CNN.

Durante la semana pasada implementamos el backpropagation de una Fully Connected para calcular las derivadas con respecto de un costo. De similar manera, en CNN se debe calcular la derivada con respecto del costo para actualizar parámetros. Las ecuaciones de backpropagation no son triviales, por ello trataremos de entenderlas mejor acá

Calculando dA

Esta es la formula para calcular dA con respecto de costo para un filtro dado W_c y un ejemplo de entrenamiento dado

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

Donde W_c es un filtro y dZ_{hw} es un escalar correspondiente a la gradiente del costo con respecto de la salida de la capa convolucional Z en la h-th fila y la w-th columna (correspondiente al producto punto tomado de la i-th stride en la izquierda y el j-th stride inferior). Noten que cada vez que multiplicamos el mismo filtro W_c por una dZ difrente cuando se actualiza dA. Esto lo hacemos principalmente cuando calculamos el paso forward, cada filtro es multiplicado (punto) y sumado por un differente a_slice. Entonces cuando calculamos el backpropagation para dA, estamos agregando todas las gradientes de a_slices.

En código, dentro del ciclo-for apropiado, esta formula se transforma en:

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

Calculando dW

Esta es la formula para calcular dW_c (dW_c es la derivada de un solo filtro) con respecto de la perdida

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

Donde a_{slice} corresponde al slice que se usó para generar la activación de dZ_{ij}. Entonces, esto termina dandos la gradiente de W con respecto de ese slice (ventana). Debido a que el mismo W, solo agregamos todas las gradientes para obtener dW

En código, dentro del ciclo-for apropiado, esta formula se transforma en:

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

Calculando db

Esta es la formula para calcular db con respecto del costo para un filtro dado Wc

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

Como hemos previamente visto en una red neuronal básica, db es calculada al sumar dZ. En este caso, solo sumaremos sobre todas las gradientes de la salida conv (Z) con respecto del costo.

En código, dentro del ciclo-for apropiado, esta formula se transforma en:

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

Después de este preambulo, ahora pasemos al **ejercicio**. Deberá implementar la función `conv_backward`. Deberá sumar sobre todos los datos de entrenamiento, filtros, altos, y anchos. Luego, deberá calcular las derivadas usando las formulas 1-3, de arriba.

```
def conv_backward(dZ, cache):
    """
    Implementa el backprogation para una función de convolución

    Arguments:
    dZ: Gradiente de costo shape (m, n_H, n_W, n_C)
    cache: Cache de valores output de conv_forward()

    Returns:
    dA_prev: Gradiente de costos con respecto de la entrada de la capa conv (A_prev), shape (m, n_H_prev, n_W_prev, n_C_prev)
    dW: Gradiente de costo con respecto de los de los pesos (W), shape (f, f, n_C_prev, n_C)
    db: Gradiente de costo con respecto de los biases de la capa conv (b), shape (1, 1, 1, n_C)
    """

    # Obtener info del "cache"
    (A_prev, W, b, hparameters) = cache

    # Obtener dimensiones de A_prev
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Obtener dimensiones de W
    (f, f, n_C_prev, n_C) = W.shape

    # Obtener información de los hiperparametros (hparameters)
    stride = hparameters["stride"]
    pad = hparameters["pad"]

    # Obtener dimensiones de dZ
    (m, n_H, n_W, n_C) = dZ.shape

    # Init variables
    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))




    # Agregar padding a  A_prev y dA_prev
    A_prev_pad = zero_pad(A_prev, pad)
    dA_prev_pad = zero_pad(dA_prev, pad)
```

```
    # Iteramos sobre los datos de entrenamiento
    for i in range(m):

        # Aprox 2 lineas para
        # seleccionar el i-th ejemplo de entrenamiento de A_prev y dA_prev_pad
        # a_prev_pad =
        # da_prev_pad =
        # YOUR CODE HERE
        a_prev_pad = A_prev_pad[i, :, :, :]
        da_prev_pad = dA_prev_pad[i, :, :, :]
        # raise NotImplementedError()

        # Repetimos los loops de los pasos forward
        # Iteramos sobre el eje vertical (n_H)
        for h in range(n_H):
            # Iteramos sobre el eje horizontal (n_W)
            for w in range(n_W):
                # Iteramos sobre los canales (n_C)
                for c in range(n_C):

                    # Aprox 4 linea para
                    # Encontrar las orillas de la ventana actual (slice) similar a como se hizo antes
                    # vert_start =
                    # vert_end =
                    # horiz_start =
                    # horiz_end =
                    # YOUR CODE HERE
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f
                    # raise NotImplementedError()

                    # Aprox 1 linea de codigo para
                    # Usar las orillas para definir el slice (ventana) de a_prev_pad
                    # a_slice =
                    # YOUR CODE HERE
                    a_slice = a_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :]
                    # raise NotImplementedError()

                    # Update gradients for the window and the filter's parameters using the code formulas given above
                    # Aprox 3 lineas para
                    # Actualizar gradientes para la ventana y los params del filtro usando las formulas de arriba
                    # da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] +=
                    # dW[:,:,:,c] +=
                    # db[:,:,:,c] +=
                    # YOUR CODE HERE
                    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]
                    # raise NotImplementedError()

        # Aprox 1 linea para
        # Settear el dA_prev del i-th ejemplo de entrenamiento a un da_prev_pad sin padding
        # Considere usar X[pad:-pad, pad:-pad, :]
        # dA_prev[i, :, :, :] =
        # YOUR CODE HERE
        dA_prev[i, :, :, :] = da_prev_pad[pad:-pad, pad:-pad, :]
        # raise NotImplementedError()

    # Asegurandose que la forma es la correcta
    assert(dA_prev.shape == (m, n_H_prev, n_W_prev, n_C_prev))

    return dA_prev, dW, db
```

```
np.random.seed(seed_)

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))

with tick.marks(5):
    assert(check_hash(dA, ((10, 7, 7, 5), 5720525.244018247)))

with tick.marks(5):
    assert(check_hash(dW, ((3, 3, 5, 8), -2261214.2801842494)))

with tick.marks(5):
    assert(check_hash(db, ((1, 1, 1, 8), 11211.666220998337)))
```

```
dA_mean = 1.7587047105903002
dW_mean = -30.84696464944312
db_mean = 163.5455610593757
```

✓ [5 marks]

✓ [5 marks]

✓ [5 marks]

Ejercicio 6

Es momento de hacer el paso backward para la **capa pooling**. Vamos a empezar con la versión max-pooling. Noten que incluso aunque las capas de pooling no tienen parámetros para actualizar en backpropagation, aun se necesita pasar el gradiente en backpropagation por las capas de pooling para calcular los gradientes de las capas que vinieron antes de la capa de pooling

Max-pooling paso Backward

Antes de ir al backpropagation de la capa de pooling, vamos a crear una función de apoyo llamada `create_mask_from_window()` que hará lo siguiente

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

Como pueden observar, esta función creará una matriz "máscara" que ayudará a llevar tracking de donde está el valor máximo. El valor 1 indica la posición del máximo de una matriz X, las demás posiciones son 0. Veremos más adelante que el paso backward con average-pooling es similar pero con diferente máscara

Ejercicio: Implemente la función `create_mask_from_window()`.

Hints:

- np.max() puede ser de ayuda.
- Si tienen una matriz X y un escalar x: `A = (X==x)` devolverá una matriz A del mismo tamaño de X tal que:

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

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

- En este caso, no considere casos donde hay varios máximos en una matriz

```
def create_mask_from_window(x):
    """
    Crea una máscara para el input x, para identificar máximos

    Arguments:
    x: Array, shape (f, f)

    Returns:
    mask: Array de la misma dimensión de la ventana, y con 1 donde está el máximo

    """
    # Aprox 1 linea para
    # mask =
    # YOUR CODE HERE
    mask = (x == np.max(x)).astype(int)
    # raise NotImplementedError()

    return mask
```

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

with tick.marks(5):
    assert(check_hash(mask, ((2, 3), 2.5393446629166316)))
```

```
x =  [[ 0.71167353 -0.32448496 -1.00187064]
 [ 0.23625079 -0.10215984 -1.14129263]]
mask =  [[1 0 0]
 [0 0 0]]
```

✓ [5 marks]

Es válido preguntarse ¿por qué hacemos un seguimiento de la posición del máximo? Es porque este es el valor de entrada que finalmente influyó en la salida y, por lo tanto, en el costo.

Backprop está calculando gradientes con respecto al costo, por lo que todo lo que influya en el costo final debe tener un gradiente distinto de cero. Entonces, backprop "propagará" el gradiente de regreso a este valor de entrada particular que influyó en el costo.

Average-pooling paso Backward

En max-pooling, para cada ventana de entrada, toda la "influencia" en la salida provino de un solo valor de entrada: el máximo. En la agrupación promedio, cada elemento de la ventana de entrada tiene la misma influencia en la salida. Entonces, para implementar backprop, ahora implementaramos una función auxiliar que refleje esto.

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

Esto implica que cada posición en la matriz contribuye por igual a la salida porque en el pase hacia adelante tomamos un promedio.

Ejercicio: Implemente la función para distribuir de igual manera el valor dz en una matriz del mismo tamaño de "shape"

```
def distribute_value(dz, shape):
    """
    Distribuye la entrada en una matriz de la misma dimensión de shape

    Arguments:
    dz: Input
    shape: La forma (n_H, n_W) de la salida

    Returns:
    a: Array, shape (n_H, n_W)
    """

    # Aprox 3 lieneas para
    # (n_H, n_W) =
    # average =
    # a =
    # YOUR CODE HERE
    (n_H, n_W) = shape
    average = dz / (n_H * n_W)
    a = np.ones(shape) * average
    # raise NotImplementedError()

    return a
```

```
a = distribute_value(5, (7,7))
print('valor distribuido =', a)

with tick.marks(5):
    assert check_scalar(a[0][0], '0x23121715')
```

```
valor distribuido = [[0.10204082 0.10204082 0.10204082 0.10204082 0.10204082 0.102046
  0.10204082]
 [0.10204082 0.10204082 0.10204082 0.10204082 0.10204082 0.10204082
  0.10204082]
 [0.10204082 0.10204082 0.10204082 0.10204082 0.10204082 0.10204082
  0.10204082]
 [0.10204082 0.10204082 0.10204082 0.10204082 0.10204082 0.10204082
  0.10204082]
 [0.10204082 0.10204082 0.10204082 0.10204082 0.10204082 0.10204082
  0.10204082]
 [0.10204082 0.10204082 0.10204082 0.10204082 0.10204082 0.10204082
  0.10204082]
 [0.10204082 0.10204082 0.10204082 0.10204082 0.10204082 0.10204082
  0.10204082]]
```

✓ [5 marks]

Ejercicio 7

Ahora tienen todo lo necesario para calcular el backpropagation en una capa de agrupación.

Ejercicio: Implementen la función `pool_backward` en ambos modos ("max" y "average"). Una vez más, usarán 4 loops-for (iterando sobre ejemplos de entrenamiento, altura, ancho y canales). Debe usar una instrucción `if/elif` para ver si el modo es igual a 'máximo' o 'promedio'. Si es igual a 'promedio', debe usar la función `distribuir_valor ()` que se creo anteriormente para crear una matriz de la misma forma que "a_slice". De lo contrario, el modo es igual a 'max', y creará una máscara con `create_mask_from_window()` y la multiplicará por el valor correspondiente de `dZ`.

```
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
    """


    # Obtener info del cache
    (A_prev, hparameters) = cache

    # Obtener info de "hparameters"
    stride = hparameters["stride"]
    f = hparameters["f"]

    # Dimensiones de A_prev y dA
    m, n_H_prev, n_W_prev, n_C_prev = A_prev.shape
    m, n_H, n_W, n_C = dA.shape

    # Init dA_prev
    dA_prev = np.zeros(A_prev.shape)

    # Iterar sobre los ejemplos de entrenamiento
    for i in range(m):

        # Aprox 1 linea para
        # seleccionar el ejemplo de entrenamiento de A_prev
        # a_prev =
        # YOUR CODE HERE
        a_prev = A_prev[i, :, :, :]
        # raise NotImplementedError()

        # Iterar sobre lo vertical (n_H)
        for h in range(n_H):
            # Iterar sobre lo horizontal (n_W)
            for w in range(n_W):
                # Iterar sobre los canales (n_c)
                for c in range(n_C):

                    # Aprox 4 linea para
                    # Encontrar las orillas de la ventana actual (slice) similar a como se hizo antes
                    # vert_start =
                    # vert_end =
                    # horiz_start =
                    # horiz_end =
                    # YOUR CODE HERE
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f
                    # raise NotImplementedError()

                    # Calcular backward prop para ambos modos
                    if mode == "max":

                        # Aprox 3 lineas para
                        # Usar las orillas y "c" para definir el slice actual de a_prev
                        # a_prev_slice =
                        # Crear una mascara desde a_prev_slice
                        # mask =
                        # Setear dA_prev para ser dA_prev + (la mascara multiplciada por la entrada correcta de dA)
                        # dA_prev[i, vert_start:vert_end, horiz_start:horiz_end, c] +=
                        # YOUR CODE HERE
                        a_prev_slice = a_prev[vert_start:vert_end, horiz_start:horiz_end, c]
                        mask = create_mask_from_window(a_prev_slice)
                        dA_prev[i, vert_start:vert_end, horiz_start:horiz_end, c] += mask * dA[i, h, w, c]
                        # raise NotImplementedError()

                    elif mode == "average":
                        # Aprox 3 lineas para
                        # Obtener los valores de dA
                        # da =
                        # Definir la forma del filtro fxf
                        # shape =
                        # Distribuirlo para obtener el tamaño correcto de dA_prev (sume el valod distribuido de da)
                        # YOUR CODE HERE
```

```
                        da = dA[i, h, w, c]
                        shape = (f, f)
                        dA_prev[i, vert_start:vert_end, horiz_start:horiz_end, c] += distribute_value(da, shape)
                        # raise NotImplementedError()


    # Asegurandose que la forma de la salida sea correcta
    assert(dA_prev.shape == A_prev.shape)

    return dA_prev
```

```
np.random.seed(seed_)
A_prev = np.random.randn(5, 5, 3, 2)
hparameters = {"stride" : 1, "f": 2}
A, cache = pool_forward(A_prev, hparameters)
print(A.shape)
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()
with tick.marks(5):
    assert(check_hash(dA_prev, ((5, 5, 3, 2), 1166.727871556145)))

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])
with tick.marks(5):
    assert(check_hash(dA_prev, ((5, 5, 3, 2), 1131.4343089227643)))
```

```
(5, 4, 2, 2)
mode = max
mean of dA =  0.10390017715645054
dA_prev[1,1] =  [[ 1.24312631  0.        ]
 [ 0.         -1.05329248]
 [-1.03592891  0.        ]]
```

✓ [5 marks]

```
mode = average
mean of dA =  0.10390017715645054
dA_prev[1,1] =  [[ 0.31078158  0.10580814]
 [ 0.30923847 -0.2046901 ]
 [-0.00154311 -0.31049824]]
```

✓ [5 marks]

Ejercicio 8

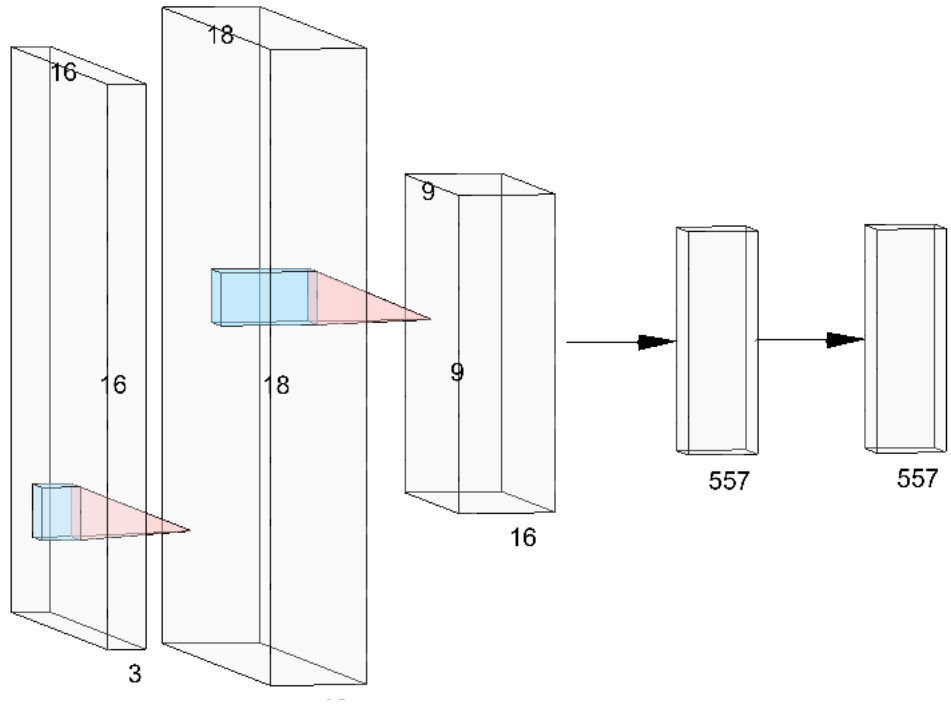
Hemos hecho todas las partes "a mano", es momento entonces de unir todo para intentar predecir nuevamente gatitos y perritos.

Tengan en cuenta que volveremos a usar el mismo método de la última vez pero ahora bajaremos significativamente la resolución de las imagenes para agilizar el proceso de entrenamiento. Esto, logicamente, afectará al resultado final, pero no se preocupen, está bien si les sale una pérdida excesivamente alta y un accuracy demasiado bajo, lo importante de este paso es que entiendan como todo se entrelaza. Además, muchas de las funciones para terminar de enlanzar todo les son dadas, no está de más que las lean y entiendan que sucede, pero ustedes deberán implementar varias también.

Para esta parte esparemos haciendo una arquitectura bastante simple, siendo esta algo como sigue

Input (64x64x3) --> Conv Layer (16 filters, 3x3, stride 1) --> ReLU Activation --> Max Pooling (2x2, stride 2) --> Fully Connected Layer (2 classes) --> Softmax Activation --> Output (Probs para Gato y Perro)

Se podría visualizar de una forma como la que se muestra en la siguiente imagen

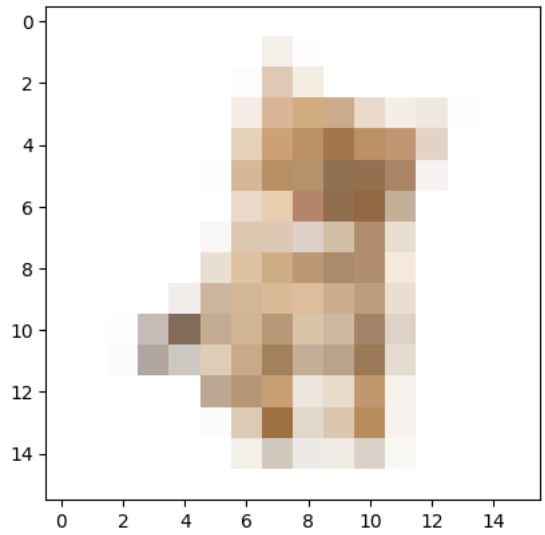



```
# Por favor cambien esta ruta a la que corresponda en sus maquinas
data_dir = './images/'

train_images = []
train_labels = []
test_images = []
test_labels = []

def read_images(folder_path, label, target_size, color_mode='RGB'):
    for filename in os.listdir(folder_path):
        image_path = os.path.join(folder_path, filename)
        # Use PIL to open the image
        image = Image.open(image_path)

        # Convert to a specific color mode (e.g., 'RGB' or 'L' for grayscale)
        image = image.convert(color_mode)

        # Resize the image to the target size
        image = image.resize(target_size)

        # Convert the image to a numpy array and add it to the appropriate list
        if label == "cats":
            if 'train' in folder_path:
                train_images.append(np.array(image))
                train_labels.append(0)  # Assuming 0 represents cats
            else:
                test_images.append(np.array(image))
                test_labels.append(0)  # Assuming 0 represents cats
        elif label == "dogs":
            if 'train' in folder_path:
                train_images.append(np.array(image))
                train_labels.append(1)  # Assuming 1 represents dogs
            else:
                test_images.append(np.array(image))
                test_labels.append(1)  # Assuming 1 represents dogs
# Call the function for both the 'train' and 'test' folders
train_cats_path = os.path.join(data_dir, 'train', 'cats')
train_dogs_path = os.path.join(data_dir, 'train', 'dogs')
test_cats_path = os.path.join(data_dir, 'test', 'cats')
test_dogs_path = os.path.join(data_dir, 'test', 'dogs')


# Read images
target_size = (16, 16)
read_images(train_cats_path, "cats", target_size)
read_images(train_dogs_path, "dogs", target_size)
read_images(test_cats_path, "cats", target_size)
read_images(test_dogs_path, "dogs", target_size)

# Convert the lists to numpy arrays
train_images = np.array(train_images)
```

```
train_labels = np.array(train_labels)
test_images = np.array(test_images)
test_labels = np.array(test_labels)

# Reshape the labels
train_labels = train_labels.reshape((1, len(train_labels)))
test_labels = test_labels.reshape((1, len(test_labels)))
```

```
# Ejemplo de una imagen
# Sí, ahora se ve menos el pobre gatito :(
index = 25
plt.imshow(train_images[index])
print ("y = " + str(train_labels[0][index]) + ", es una imagen de un " + 'gato' if train_labels[0][index]==0 else 'perro' +  "'.")
```

```
y = 0, es una imagen de un gato
```




```
np.random.seed(seed_)

def one_hot_encode(labels, num_classes):
    """
    Convierte labels categoricos a vectores

    Arguments:
    labels: Array shape (m,)
    num_classes: Número de clases

    Returns:
    one_hot: Array, shape (m, num_classes)
    """
    m = labels.shape[0]
    one_hot = np.zeros((m, num_classes))
    one_hot[np.arange(m), labels] = 1
    return one_hot


def relu(Z):
    """
    Aplica ReLU como funcion de activación al input

    Arguments:
    Z: Input

    Returns:
    A: Output array, mismo shape de Z
    cache: Contiena a Z para usar en el backprop
    """
    A = np.maximum(0, Z)
    cache = Z
    return A, cache


def relu_backward(dA, cache):
    """
    Calcula la derivada del costo con respecto del input de ReLU

    Arguments:
    dA: Gradiente del costo con respecto del output de ReLU
    cache: Z del paso forward

```

```
    Returns:
    dZ: Gradiente del costo con respecto del input
    """
    Z = cache
    dZ = np.multiply(dA, Z > 0)
    return dZ


def softmax(Z):
    """
    Aplica softmax al input

    Arguments:
    Z: Input array, shape (m, C), m = # de ejemplso, C = # de clases

    Returns:
    A : Salida con softmax
    """
    e_Z = np.exp(Z - np.max(Z, axis=1, keepdims=True))
    A = e_Z / np.sum(e_Z, axis=1, keepdims=True)
    return A


def softmax_backward(A):
    """
    Calcula la derivada de softmax

    Arguments:
    A: Salida del softmax

    Returns:
    dA: Gradiente del costo con respecto de la salida del softmax
    """
    dA = A * (1 - A)
    return dA

def initialize_parameters(n_H, n_W, n_C):
    """
    Inicializa los parametros de la CNN

    Arguments:
    n_H: Alto de las imagenes
    n_W: Ancho de las imagenes
    n_C: Canales

    Returns:
    parameters: Diccionario con los filtros y biases
    """
    parameters = {}

    # First convolutional layer
    parameters['W1'] = np.random.randn(3, 3, n_C, 16) * 0.01
    parameters['b1'] = np.zeros((1, 1, 1, 16))

    # Second convolutional layer - Not used to much time on training
    #parameters['W2'] = np.random.randn(3, 3, 16, 32) * 0.01
    #parameters['b2'] = np.zeros((1, 1, 1, 32))

    # Fully connected layer
    parameters['W3'] = np.random.randn(1296, 2) * 0.01
    parameters['b3'] = np.zeros((1, 2))

    return parameters

def conv_layer_forward(A_prev, W, b, hparameters_conv):
    """
    Forward pass de una capa convolucional

    Arguments:
    A_prev: Matriz previa
    W: Filtro
    b: Biases
    hparameters_conv: hiperparametros

    Returns:
    A: Nueva matriz de datos
    cache: Cache con la info de ReLU y la convolucional
    """
    Z, cache_conv = conv_forward(A_prev, W, b, hparameters_conv)
    A, cache_relu = relu(Z)
    cache = (cache_conv, cache_relu)
    return A, cache
```

```
126
127 def pool_layer_forward(A_prev, hparameters_pool, mode='max'):
128     """
129     Llama a la función realizada previamente - Ver docstring de pool_forward
130     """
131     A, cache = pool_forward(A_prev, hparameters_pool, mode)
132     return A, cache
133
134 def fully_connected_layer_forward(A2, A_prev_flatten, W, b):
135     """
136     Forward pass de fully connected
137
138     Arguments:
139     A2: Matriz previa no aplanada
140     A_prev_flatten: Matriz previa aplanada
141     W: Filtro
142     b: Biases
143
144     Returns:
145     A: Nueva matriz de datos
146     cache: Cache con la info de ReLU y la convolucional
147     """
148     Z = np.dot(A_prev_flatten, W) + b
149     A = softmax(Z) # cache_fc = softmax(Z)
150     cache = (A2, A_prev_flatten, W, b, Z, A)
151     return A, cache
152
153 def simple_cnn_model(image_array, parameters):
154     """
155     Implementa un modelo simple de CNN para predeciri si una imagen es un perrito o un gatito
156
157     Arguments:
158     image_array: Imagenes, shape (m, n_H, n_W, n_C)
159     parameters: Diccionario con los filtros y pesos de cada capa
160
161     Returns:
162     A_last: Salida de la ultima capa (Probabilidades softmax para ambas clases) cat and dog)
163     caches: Lista de caches con lo necesario para backward prop
164     """
165     # Retrieve the filter weights and biases from the parameters dictionary
166     W1 = parameters['W1']
167     b1 = parameters['b1']
168
169     #W2 = parameters['W2']
170     #b2 = parameters['b2']
171
172     W3 = parameters['W3']
173     b3 = parameters['b3']
174
175     # Define the hyperparameters for the CNN
176     hparameters_conv = {"stride": 1, "pad": 2}
177     hparameters_pool = {"f": 2, "stride": 2}
178
179     # Aprox 2 lineas para Forward propagation
180     # Asegurese de usar las funciones dadas y de usar el mode='max' para la pooling layer
181     # A1, cache0 =
182     # A2, cache1 =
183     # YOUR CODE HERE
184     A1, cache0 = conv_layer_forward(image_array, W1, b1, hparameters_conv)
185     A2, cache1 = pool_layer_forward(A1, hparameters_pool, mode='max')
186     # raise NotImplementedError()
187
188     # Flatten the output of the second convolutional layer
189     A2_flatten = A2.reshape(A2.shape[0], -1)
190
191     # Aprox 1 linea para Fully connected layer
192     # De nuevo, asegurense de usar la funcion dada
193     # A_last, cache2 =
194     # YOUR CODE HERE
195     A_last, cache2 = fully_connected_layer_forward(A2, A2_flatten, W3, b3)
196     # raise NotImplementedError()
197
198     # Cache values needed for backward propagation
199     caches = [cache0, cache1, cache2]
200
201     return A_last, caches
202
203 def compute_cost(A_last, Y):
204     """
205     Calcula el costo de cross-entroy para las probabilidades predichas
206
207     Arguments:
```

```
    A_last: Prob predichas, shape (m, 2)
    Y: Labels verdaders, shape (m, 2)

    Returns:
    cost: cross-entropy cost
    """
    m = Y.shape[0]
    cost = -1/m * np.sum(Y * np.log(A_last + 1e-8))
    return cost

def fully_connected_layer_backward(dA_last, cache, m):
    """
    Calcula el backward pass de la fully connected

    Arguments:
    dA_last: Matriz de valores
    cache: cache util
    m: Cantidad de obs

    Returns:
    dA_prev: Derivada de la matriz
    dW: Derivada del costo con respecto del filtro
    db: Derivada del costo con respecto de bias
    """
    A_prev_unflatten, A_prev_flatten, W, b, Z, A_last = cache
    dZ = dA_last * softmax_backward(A_last)
    dW = np.dot(A_prev_flatten.T, dZ) / m
    db = np.sum(dZ, axis=0, keepdims=True) / m
    dA_prev_flatten = np.dot(dZ, W.T)
    dA_prev = dA_prev_flatten.reshape(A_prev_flatten.shape)
    return dA_prev, dW, db

def pool_layer_backward(dA, cache, mode='max'):
    """
    Llama al metodo antes definido - Ver docstring de pool_backward
    """
    return pool_backward(dA, cache, mode)

def conv_layer_backward(dA, cache):
    """
    Llama al metodo antes definido - Ver docstring de conv_backward
    """
    dZ = relu_backward(dA, cache[1])
    dA_prev, dW, db = conv_backward(dZ, cache[0])
    return dA_prev, dW, db

def backward_propagation(A_last, Y, caches):
    """
    Implemente la parte de backward prop de nuestro modelo

    Arguments:
    A_last: Probabilidades predichas, shape (m, 2)
    Y: Labels verdaderas, shape (m, 2)
    caches: Lista de caches con info para el back prop

    Returns:
    gradients: Diccionario con gradientes de filtros y biases para cada capa
    """
    m = Y.shape[0]
    gradients = {}
    # Compute the derivative of the cost with respect to the softmax output
    dZ3 = A_last - Y


    # Aprox 1 linea para hacer backprog en la fully connected layer y guardarlo en el diccionario de gradientes
    # gradients['dA2_flatten'], gradients['dW3'], gradients['db3'] =
    # Recuerden usar los metodos definidos previamente
    # N.B.: Vean a que posición del array corresponde cada caché
    # YOUR CODE HERE
    gradients['dA2_flatten'], gradients['dW3'], gradients['db3'] = fully_connected_layer_backward(dZ3, caches[2], m)
    # raise NotImplementedError()

    # Reshape dA2_flatten to match the shape of dA2
    dA2 = gradients['dA2_flatten'].reshape(caches[2][0].shape)

    # Backpropagation through the second convolutional layer and pooling layer
    # Aprox 2 lineas para hacer el backprop en la pooling y la convolucional y guardarlo en el diccionario de grads
    # gradients['dA1_pool'] =
    # gradients['dA1'], gradients['dW1'], gradients['db1'] =
    # Recuerden usar los metodos definidos previamente
    # N.B.: Vean a que posición del array corresponde cada caché
    # YOUR CODE HERE
```

```
    gradients['dA1_pool'] = pool_layer_backward(dA2, caches[1], mode='max')
    gradients['dA1'], gradients['dW1'], gradients['db1'] = conv_layer_backward(gradients['dA1_pool'], caches[0])
    # raise NotImplementedError()

    # Backpropagation through the first convolutional layer and pooling layer
    # Removed due high processing times
    # gradients['dA0'], gradients['dW1'], gradients['db1'] = conv_layer_backward(gradients['dA1_pool'], caches[0])

    return gradients

def update_parameters(parameters, gradients, learning_rate=0.01):
    """
    Actualiza los filtros y biases usando gradiente descendiente

    Arguments:
    parameters: Diccionario con filtros y biases de cada layer
    gradients: Diccionario con gradientes de filtros y biases de cada layer
    learning_rate: learning rate para gradient descent (default: 0.01)

    Returns:
    parameters: Parametros actualizados despues de un paso en la grad descent
    """
    # Aprox 4 lineas para calculo de W1, b1, W3,b3 (si, no hay continuidad en la numeracion :P)
    # parameters['W1'] -=
    # parameters['b1'] -=
    # parameters['W3'] -=
    # parameters['b3'] -=
    # YOUR CODE HERE
    parameters['W1'] -= learning_rate * gradients['dW1']
    parameters['b1'] -= learning_rate * gradients['db1']
    parameters['W3'] -= learning_rate * gradients['dW3']
    parameters['b3'] -= learning_rate * gradients['db3']
    # raise NotImplementedError()

    return parameters
```

```
np.random.seed(seed_)

# Initialize the parameters of the CNN model
parameters = initialize_parameters(n_H=target_size[0], n_W=target_size[1], n_C=3)

# Training loop
# Noten como estamos usando bien poquitas epocas, pero es para que no les tome más de 10 minutos entrenar la mini red
num_epochs = 5
learning_rate = 0.01
```

```
np.random.seed(seed_)

# Combine the train_images and test_images into one array
X_train = train_images #np.concatenate((train_images, test_images), axis=0)

# Combine the train_labels and test_labels into one array
Y_train_labels = train_labels #np.concatenate((train_labels, test_labels), axis=0)

# Convert labels to one-hot encoding
num_classes = 2
Y_train = one_hot_encode(Y_train_labels, num_classes)

```

```
np.random.seed(seed_)

for epoch in range(num_epochs):
    # Forward propagation
    A_last, caches = simple_cnn_model(X_train, parameters)

    # Compute the cost
    cost = compute_cost(A_last, Y_train)

    # Backward propagation
    gradients = backward_propagation(A_last, Y_train, caches)

    # Update parameters using gradient descent
    parameters = update_parameters(parameters, gradients, learning_rate)

    # Print the cost every few epochs - Removed not used due high times
    #if epoch % 10 == 0:
    print(f"Epoch {epoch+1}, Cost: {cost}")

print("Training completed.")
```

```
C:\Users\omen\AppData\Local\Temp\ipykernel_29328\3930102447.py:22: DeprecationWarning: Conversion of an array with ndim > 0 to a sc
  Z = Z + float(b)
Epoch 1, Cost: 2370.7655299381468
Epoch 2, Cost: 10260.319168811471
Epoch 3, Cost: 10260.319168811471
Epoch 4, Cost: 10260.319168811471
Epoch 5, Cost: 10260.319168811471
Training completed.
```

```
with tick.marks(10):
    assert check_scalar(cost, '0xd574bb64')
```

✓ [10 marks]

```
# Testing the model using the test dataset
def test_model(X_test, Y_test, parameters):
    """
    Testea el modelo CNN usando el dataset

    Arguments:
    X_test: Imagenes, shape (m, n_H, n_W, n_C)
    Y_test: Labels verdaders para probas las imagenes, shape (m, num_classes)
    parameters: Diccionario con filtros y biases de cada capa

    Returns:
    accuracy: Accuracy
    """
    # Forward propagation
    A_last, _ = simple_cnn_model(X_test, parameters)

    # Convert softmax output to predicted class labels (0 for cat, 1 for dog)
    predictions = np.argmax(A_last, axis=1)

    # Convert true labels to class labels
    true_labels = np.argmax(Y_test, axis=1)

    # Calculate accuracy
    accuracy = np.mean(predictions == true_labels) * 100
    return accuracy

# Test the model on the test dataset
accuracy = test_model(test_images, one_hot_encode(test_labels, 2), parameters)
print(f"Test Accuracy: {accuracy:.2f}%")
```

```
C:\Users\omen\AppData\Local\Temp\ipykernel_29328\3930102447.py:22: DeprecationWarning: Conversion of an array with ndim > 0 to a sc
  Z = Z + float(b)
Test Accuracy: 0.00%
```

```
with tick.marks(10):
    assert check_scalar(accuracy, '0x75c2e82a')
```

✓ [10 marks]

Entonces, como podemos ver el modelo creado es **realmente** malo. Pero para fines didácticos cumple con su cometido 😁

NOTA: Conteste como txt, pdf, comentario en la entrega o en este mismo notebook:

- ¿Por qué creen que es el mal rendimiento de este modelo?
- ¿Qué pueden hacer para mejorarlo?
- ¿Cuáles son las razones para que el modelo sea tan lento?

Ahora pasemos a ver como hacer algo mejor para el mismo tipo de tarea usando PyTorch 😊

Parte 2 - Usando PyTorch

Muy bien, hemos entendido ahora de mejor manera todo lo que sucede dentro de una red neuronal convolucional. Pasamos desde definir los pasos de forward, hasta incursionar en cómo realizar los pasos de backpropagation y al final, vimos una forma simple pero academicamente efectiva para entender lo que sucede dentro de una CNN.

Ahora, subamos un nivel y pasemos a ver como PyTorch nos ayuda a crear una CNN básica pero efectiva. Pero antes, es necesario que definamos la unidad básica de PyTorch, esta es conocida como Tensor 🤔

Tensor

En PyTorch, un tensor es una estructura de datos fundamental que representa matrices multidimensionales o matrices n-dimensionales. Los tensores son similares a las matrices o arrays de NumPy, pero tienen características y funcionalidades adicionales que están optimizadas para tareas de Deep Learning, incluida la diferenciación automática para el cálculo de gradientes durante la retropropagación.

En PyTorch, se puede crear un tensor para representar datos numéricos, como imágenes, sonidos o cualquier otro dato numérico que pueda necesitar. Los tensores se pueden manipular mediante operaciones matemáticas como la suma, la resta y la multiplicación, lo que los hace esenciales para construir y entrenar modelos de Deep Learning.

Los tensores en PyTorch y los arrays de NumPy comparten similitudes y pueden realizar operaciones similares. Sin embargo, los tensores de PyTorch están diseñados específicamente para tareas de Deep Learning y ofrecen algunas funcionalidades adicionales optimizadas para la diferenciación automática durante backpropagation. Aquí hay algunas operaciones comunes que los tensores pueden hacer:

- **Operaciones matemáticas**: Los tensores admiten operaciones matemáticas estándar, como suma, resta, multiplicación, división y operaciones por elementos, como multiplicación por elementos, división por elementos, etc.
- **Reshaping**: Los tensores se pueden remodelar para cambiar sus dimensiones, lo que le permite convertir un tensor 1D en un tensor 2D, o viceversa.
- Operaciones de reducción: Los tensores admiten operaciones de reducción como sumar a lo largo de dimensiones específicas, calcular la media, la varianza, el máximo, el mínimo, etc.
- **Element-wise Functions**: Puede aplicar funciones matemáticas elementales como exponencial, logaritmo, seno, coseno, etc., a los tensores.
- **Broadcasting**: Los tensores admiten la difusión, lo que le permite realizar operaciones en tensores con diferentes formas.
- **Indexing y Slicing**: Puede acceder a elementos específicos o secciones de un tensor mediante operaciones de indexación y división.
- **Concatenación y splitting**: Los tensores se pueden concatenar a lo largo de dimensiones específicas y puede dividir un tensor en varios tensores más pequeños.
- **Transposición**: los tensores se pueden transponer para cambiar el orden de sus dimensiones.
- **Aceleración de GPU**: Los tensores se pueden mover y operar fácilmente en GPU, lo que permite cálculos más rápidos para modelos de aprendizaje profundo a gran escala.

Además, se diferencian de los arrays de Numpy en:

- **Diferenciación automática**: Una de las diferencias clave es la función de diferenciación automática de PyTorch, que permite que los tensores realicen un seguimiento de las operaciones realizadas en ellos y calculen automáticamente los gradientes durante la retropropagación. Esta característica es crucial para entrenar redes neuronales utilizando algoritmos de optimización basados en gradientes.
- **Compatibilidad con GPU**: Si bien las matrices NumPy están diseñadas para el cálculo numérico basado en CPU, los tensores PyTorch se pueden mover y operar fácilmente en GPU, lo que permite un cálculo más rápido para modelos de aprendizaje profundo a gran escala.
- **Gráfico computacional dinámico**: PyTorch crea un gráfico computacional (como el que vimos en la primera clase) dinámico, lo que significa que el gráfico se construye sobre la marcha a medida que se realizan las operaciones. Esto permite una mayor flexibilidad en la definición de modelos complejos en comparación con los gráficos de cálculo estáticos utilizados en marcos como TensorFlow.
- **Integración de Deep Learning**: PyTorch se usa ampliamente en la comunidad de aprendizaje profundo debido a su estrecha integración con marcos de aprendizaje profundo. Muchas bibliotecas de aprendizaje profundo, como torchvision y torchtext, se construyen sobre PyTorch.

Después de todo este texto (sí, yo sé, es mucho texto 😶), vamos a empezar ahora a definir nuestra CNN con PyTorch, para luego medir su performance. Empecemos por traer de vuelta parte del código que teníamos la otra vez.

No está de más recordarles en que **se recomienda el uso de ambientes virtuales**

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import Dataset, DataLoader, random_split
from PIL import Image
import torch.utils.data as data
import random
from torchvision.datasets import ImageFolder



# Seed all possible
seed_ = 2023
random.seed(seed_)
np.random.seed(seed_)
torch.manual_seed(seed_)

# If using CUDA, you can set the seed for CUDA devices as well
if torch.cuda.is_available():
    torch.cuda.manual_seed(seed_)
    torch.cuda.manual_seed_all(seed_)

```

```
import torch.backends.cudnn as cudnn
cudnn.deterministic = True
cudnn.benchmark = False
```

```
!pip install torchvision
```

```
Requirement already satisfied: torchvision in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8
Requirement already satisfied: torch==2.0.1 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra
Requirement already satisfied: numpy in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\loc
Requirement already satisfied: requests in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_q
Requirement already satisfied: sympy in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\loc
Requirement already satisfied: jinja2 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\lo
Requirement already satisfied: filelock in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\
Requirement already satisfied: networkx in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra8p0\
Requirement already satisfied: typing-extensions in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n
Requirement already satisfied: certifi>=2017.4.17 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5
Requirement already satisfied: idna<4,>=2.5 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.1
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2k
Requirement already satisfied: mpmath>=0.19 in c:\users\omen\appdata\local\packages\pythonsoftwarefoundation.python.3.10_qbz5n2kfra

[notice] A new release of pip is available: 23.0.1 -> 23.2.1
[notice] To update, run: C:\Users\omen\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\pytho
```

```
class CatsAndDogsDataset(Dataset):
    def __init__(self, data_dir, target_size=(28, 28), color_mode='RGB', train=True, transform=None):
        self.data_dir = data_dir
        self.target_size = target_size
        self.color_mode = color_mode
        self.classes = ['cats', 'dogs']
        self.train = train
        self.image_paths, self.labels = self.load_image_paths_and_labels()
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]
        image = Image.open(image_path)
        image = image.convert(self.color_mode)
        image = image.resize(self.target_size)

        if self.transform is not None:
            image = self.transform(image)

        label = torch.tensor(self.labels[idx], dtype=torch.long)

        return image, label

    def load_image_paths_and_labels(self):
        image_paths = []
        labels = []
        for class_idx, class_name in enumerate(self.classes):
            class_path = os.path.join(self.data_dir, 'train' if self.train else 'test', class_name)
            for filename in os.listdir(class_path):
                image_path = os.path.join(class_path, filename)
                image_paths.append(image_path)
                labels.append(class_idx)
        return image_paths, labels
```

```
# Define the CNN model
class CNNClassifier(nn.Module):
    def __init__(self, input_channels, image_size, num_classes):
        super(CNNClassifier, self).__init__()
        # Formula to calculate the size of the next layers:
        # output_size = (input_size - kernel_size + 2 * padding) / stride + 1
        # For pooling layers, output_size = input_size / kernel_size

        self.conv_layers = nn.Sequential(
            # The 16 represents the number of filters used in the first convolutional layer.
            # Each filter will generate one feature map, resulting in a total of 16 feature maps
            # as the output of this layer.
            # Increasing the number of filters can help the model learn more complex and abstract features,
            # but it also increases the number of parameters in the model and may require more computational resources.
            nn.Conv2d(input_channels, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Output size after the first convolution and pooling
```

```
            # output_size = (image_size - 3 + 2 * 1) / 1 + 1 = (image_size - 1) / 1 + 1 = image_size / 2

            # Here 16 denotes the number of input channels or feature maps from the previous layer.
            # In this case, the output of the first convolutional layer (with 16 filters)
            # serves as the input to the second convolutional layer
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Output size after the second convolution and pooling
            # output_size = (image_size / 2 - 3 + 2 * 1) / 1 + 1 = (image_size / 2 - 1) / 1 + 1 = image_size / 4

            # Aprox 3 lineas para completar:
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Output size after the third convolution and pooling
            # output_size = (image_size / 4 - 3 + 2 * 1) / 1 + 1 = (image_size / 4 - 1) / 1 + 1 = image_size / 8
        )

        # Calculate the output size after convolutions and pooling
        #   Since we have 3 max pooling layers with kernel_size=2 and stride=2
        #   2^3 = 8
        output_size_after_conv = image_size // 8
        self.fc_layers = nn.Sequential(
            # The value 64 comes from the number of output channels (or feature maps) of the last convolutional layer.
            # The output_size_after_conv represents the spatial size (height and width) of the feature maps after
            #   passing through the convolutional and max-pooling layers.
            # The 128 This is the number of output units (neurons) in the fully connected layer.
            #   It determines the dimensionality of the representation learned by this layer.
            #   The choice of 128 units is a hyperparameter that can be adjusted based on the complexity
            #   of the task and the available computational resources.
            nn.Linear(64 * output_size_after_conv * output_size_after_conv, 128),
            # Aprox 2 lineas para complementar
            nn.ReLU(),
            nn.Linear(128,2)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        # Reshape
        x = x.view(x.size(0), -1)
        x = self.fc_layers(x)
        return x
```

```
# Set the parameters
input_channels = 3  # RGB images have 3 channels
image_size = 64     # Size of the input images (assuming square images)
num_classes = 2     # Number of classes (cat and dog)
output_size = 2
batch_size = 32
```

```
# Create the CNN model
model = CNNClassifier(input_channels, image_size, num_classes)
```

```
# Check if CUDA is available and move the model to the GPU if possible
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print("Using device:", device)
```

```
Using device: cpu
```

```
# Print the model architecture
print(model)
```

```
CNNClassifier(
  (conv_layers): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc_layers): Sequential(
    (0): Linear(in_features=4096, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=2, bias=True)
```

```
        )
    )
```

```
# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
# Define data transformations
transform = transforms.Compose([
    transforms.Resize((image_size, image_size)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # Normalize to range [-1, 1]
])

train_dataset = CatsAndDogsDataset(data_dir=data_dir, target_size=(image_size, image_size), train=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

test_dataset = CatsAndDogsDataset(data_dir=data_dir, target_size=(image_size, image_size), train=False, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```
# Training loop
num_epochs = 50
losses = []  # List to store losses per epoch

# Estimated time in training = 5 min
for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    # Calculate the average loss for this epoch
    epoch_loss = total_loss / len(train_loader)
    losses.append(epoch_loss)

    print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {total_loss / len(train_loader)}")

```

```
Epoch 1/50, Loss: 0.7085405323240492
Epoch 2/50, Loss: 0.6793268885877397
Epoch 3/50, Loss: 0.672454704840978
Epoch 4/50, Loss: 0.6453045242362552
Epoch 5/50, Loss: 0.6212782363096873
Epoch 6/50, Loss: 0.6036208205752902
Epoch 7/50, Loss: 0.5497165570656458
Epoch 8/50, Loss: 0.5069848315583335
Epoch 9/50, Loss: 0.47104638980494606
Epoch 10/50, Loss: 0.42449313236607444
Epoch 11/50, Loss: 0.3667708494597011
Epoch 12/50, Loss: 0.2743135591348012
Epoch 13/50, Loss: 0.20001775978340042
Epoch 14/50, Loss: 0.13239809518886936
Epoch 15/50, Loss: 0.08384558279067278
Epoch 16/50, Loss: 0.08744736781550778
Epoch 17/50, Loss: 0.05568400863558054
Epoch 18/50, Loss: 0.04625098306375245
Epoch 19/50, Loss: 0.04691910076265534
Epoch 20/50, Loss: 0.031214885258426268
Epoch 21/50, Loss: 0.010848335753608909
Epoch 22/50, Loss: 0.006220981310535636
Epoch 23/50, Loss: 0.004077760732292922
Epoch 24/50, Loss: 0.0023784524851685595
Epoch 25/50, Loss: 0.0016892649388561647
Epoch 26/50, Loss: 0.001614093463609202
Epoch 27/50, Loss: 0.0013479461275791335
Epoch 28/50, Loss: 0.0011579820533774586
Epoch 29/50, Loss: 0.0009311509800479851
Epoch 30/50, Loss: 0.0009270413267788374
Epoch 31/50, Loss: 0.0008178018324542791
Epoch 32/50, Loss: 0.0006996023310219041
Epoch 33/50, Loss: 0.0006238809324309437
Epoch 34/50, Loss: 0.0006185303112336745
Epoch 35/50, Loss: 0.000558226796985966
Epoch 36/50, Loss: 0.0005171389978689452
Epoch 37/50, Loss: 0.00046524042247458256
Epoch 38/50, Loss: 0.00043319125042115856
Epoch 39/50, Loss: 0.0004072540484937943
Epoch 40/50, Loss: 0.00037653246626076807
```

```
Epoch 41/50, Loss: 0.00035759177424349927
Epoch 42/50, Loss: 0.0003352656090606211
Epoch 43/50, Loss: 0.0003439895954215899
Epoch 44/50, Loss: 0.00031686189484187507
Epoch 45/50, Loss: 0.00027880768619878736
Epoch 46/50, Loss: 0.0002659567366612868
Epoch 47/50, Loss: 0.0002552206901277208
Epoch 48/50, Loss: 0.0002446055372678933
Epoch 49/50, Loss: 0.00022509118126537133
Epoch 50/50, Loss: 0.00022111718216264207
```

Como pueden observar, ahora somos capaces de usar más epocas y esto es más eficiente (si estan en el device de CUDA). Con un tiempo aproximado de 5 minutos, podemos usar 50 epocas de entrenamiento. Ahora ya podemos considerar entrenar mucho más nuestro modelo para que se vuelva mejor (aunque esto no siempre pase, pero sí podemos entrenarlo con más epocas 😅)

```
print(losses[len(losses)-1])
```

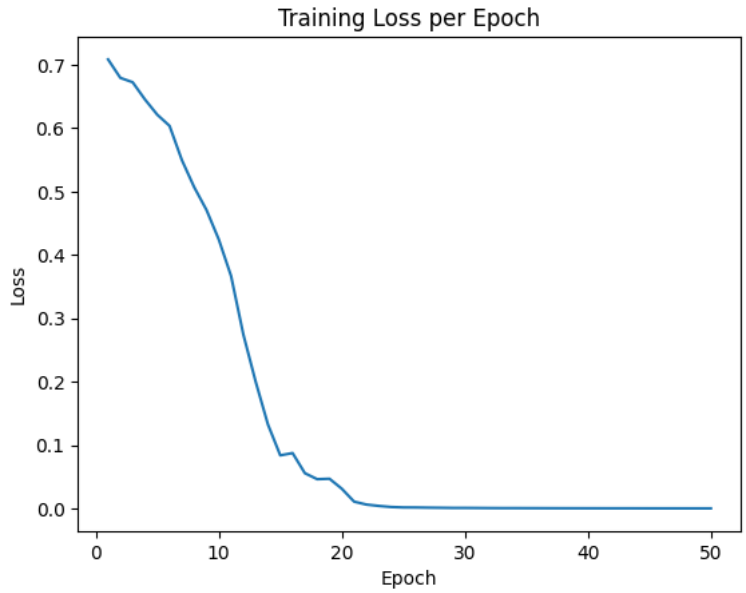
```
0.00022111718216264207
```

```
with tick.marks(5):
    assert 0.7 < losses[0] and 0.71 > losses[0]

with tick.marks(5):
    assert 0.0002 < losses[len(losses)-1] and losses[len(losses)-1] < 0.0003
```

✓ [5 marks]

✓ [5 marks]

```
# Plot the losses per epoch
plt.plot(range(1, num_epochs + 1), losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss per Epoch')
plt.show()
```



Como se puede apreciar en la gráfica, vemos como la perdida (loss) va disminuyendo conforme vamos entrenando. Esto hace mucho sentido y resulta poderosísimo, debido a que con este tipo de comportamiento podemos asegurar que nuestro modelo está funcionado al menos con el comportamiento esperado.

Cabe mencionar que algunas veces también se grafica la métrica de desempeño (accuracy, f1, etc) en estos casos para monitorear el overfitting.

```
# Evaluation
# Note eval(), this is used to remove some techniques that helps with reducing overfitting like dropout
model.eval()
correct = 0
total = 0
with torch.no_grad():
```

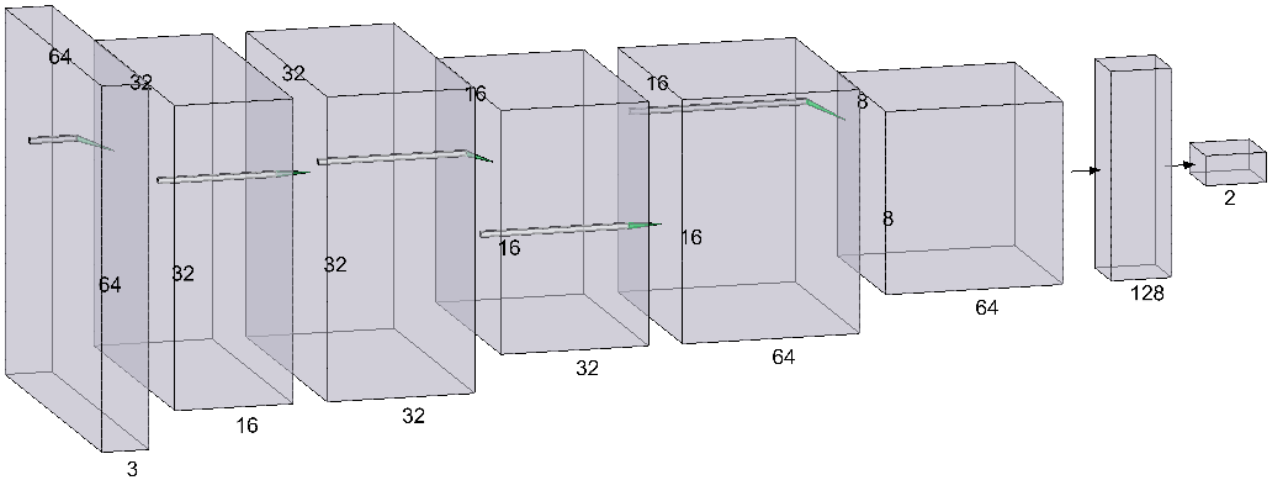
```
    for inputs, labels in test_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy_model = 100 * correct / total
print('Accuracy on test set: {:.2f}%'.format(accuracy_model))
```

```
Accuracy on test set: 63.57%
```

```
with tick.marks(10):
    assert 60 < accuracy_model and 68 > accuracy_model
```

✓ [10 marks]

Algunas otras veces necesitamos validar nuestra arquitectura, para ello es útil poder tener a vista los tamaños de las capas que vamos generando. En este caso hemos hecho una arquitectura como esta:



Pero también una visualización numérica es util. Para ello podemos usar la librería "torchsummary" que la podemos instalar como cualquier otro paquete. Recuerden volver a comentar la linea de abajo una vez hayan instalado la librería.

```
# !pip install torchsummary
```

```
Collecting torchsummary
  Downloading torchsummary-1.5.1-py3-none-any.whl (2.8 kB)
Installing collected packages: torchsummary
Successfully installed torchsummary-1.5.1

[notice] A new release of pip is available: 23.0.1 -> 23.2.1
[notice] To update, run: C:\Users\omen\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\pytho
```

```
from torchsummary import summary

# Con esta gráfica podemos observar las dimensiones de cada capa, así como los parámetros que está
# optimizando dentro de la misma
summary(model, (input_channels, image_size, image_size))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 16, 64, 64]             448
              ReLU-2          [-1, 16, 64, 64]               0
         MaxPool2d-3          [-1, 16, 32, 32]               0
            Conv2d-4          [-1, 32, 32, 32]           4,640
              ReLU-5          [-1, 32, 32, 32]               0
         MaxPool2d-6          [-1, 32, 16, 16]               0
            Conv2d-7          [-1, 64, 16, 16]          18,496
              ReLU-8          [-1, 64, 16, 16]               0
         MaxPool2d-9            [-1, 64, 8, 8]               0
           Linear-10                 [-1, 128]         524,416
             ReLU-11                 [-1, 128]               0
           Linear-12                   [-1, 2]             258
================================================================
Total params: 548,258
```

```
Trainable params: 548,258
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 1.97
Params size (MB): 2.09
Estimated Total Size (MB): 4.11
----------------------------------------------------------------
```

NOTA: Conteste como txt, pdf, comentario en la entrega o en este mismo notebook:

- ¿Qué haría para mejorar el rendimiento del modelo? R// Mejoraria la cantidad de capas y de épocas que se utilizaron para realizar este entrenamiento..
- ¿Qué haría para disminuir las posibilidades de overfitting? R// Se mejoraria el modelo con respecto a los valores de los pixeles que se brindan.

Calificación

Asegúrese de que su notebook corra sin errores (quite o resuelva los raise NotImplementedError()) y luego reinicie el kernel y vuelva a correr todas las celdas para obtener su calificación correcta

```
1
2 print()
3 print("La fraccion de abajo muestra su rendimiento basado en las partes visibles de este laboratorio")
4 tick.summarise_marks() #
5
```

```
La fraccion de abajo muestra su rendimiento basado en las partes visibles de este lat
```

110 / 110 marks (100.0%)