G03_Project2A

April 28, 2021

Pràctica: Project 2A
Autores: Bo Miquel Nordfeldt, Joan Muntaner, Helena Antich
Fecha: Abril 2021

1 Convolutional Neural Networks: Coding the layers in Numpy

In this exercise, you will implement convolutional (CONV) and pooling (POOL) layers in numpy, including both forward propagation and (optionally) backward propagation.

Notation: - Superscript $[l]$ denotes an object of the l^{th} layer. - Example: $a^{[4]}$ is the 4^{th} layer activation. $W^{[5]}$ and $b^{[5]}$ are the 5^{th} layer parameters.

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

1.1 1 - Packages

Let's first import all the packages that you will need during this coding exercise. - numpy is the fundamental package for scientific computing with Python. - matplotlib is a library to plot graphs in Python. - np.random.seed(1) is used to keep all the random function calls consistent. It will help us grade your work.

[4]:
```
# IMPORTACIÓN DE LIBRERÍAS

import numpy as np
import h5py # It lets you store huge amounts of numerical data, and easily␣
 ↪manipulate that data from NumPy.
import matplotlib.pyplot as plt
```

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

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1.2 2 - Outline of the Exercise

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

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

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

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

1.3 3 - Convolutional Neural Networks

Although programming frameworks make convolutions easy to use, they remain one of the hardest concepts to understand in Deep Learning. A convolution layer transforms an input volume into an output volume of different size, as shown below.

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

1.3.1 3.1 - Zero-Padding

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

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

The main benefits of padding are the following:

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

Exercise: Implement the following function, which pads all the images of a batch of examples X with zeros. Use np.pad. Note if you want to pad the array "a" of shape $(5, 5, 5, 5, 5)$ with `pad = 1` for the 2nd dimension, `pad = 3` for the 4th dimension and `pad = 0` for the rest, you would do:

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

[5]:
```
# FUNCIÓN AUXILIAR PARA REALIZAR PADDING

def zero_pad(X, pad):
    """
    DEFINICIÓN:
    Esta función realiza padding con ceros a todas las imágenes del dataset que␣
↪se le pase por parámetro.
    El número de columnas y filas que se quiera añadir a las imágenes (es decir␣
↪el padding) también se debe pasar
    por parámetro.

    PARÁMETROS:
    X -- Array de NumPy que representa el dataset de imágenes. El array tiene␣
↪el siguiente shape: (m, n_H, n_W, n_C)
    donde m es el número de imágenes que hay por batch, n_H la altura de la␣
↪imagen (es decir el número de filas de la
    imagen), n_W el ancho de la imagen (es decir el número de columnas de la␣
↪imagen) y n_C el número de canales de la
    imagen.
    pad -- Número entero que especifica la cantidad de relleno alrededor de␣
↪cada imagen en dimensiones verticales y horizontales

    DEVUELVE:
    X_pad -- Array de NumPy que representa el dataset de imágenes con padding.␣
↪Las dimensiones de este array son las siguientes:
    (m, n_H + 2 * pad, n_W + 2 * pad, n_C)

    """
```

```
    # Para realizar el padding de las imágenes utilizamos la función de NumPy: np.pad. Esta función permite aumentar el tamaño
    # de cualquier dimensión de una array. Para ello se debe especificar cuanto se quiere aumentar una dimensión tanto por su inicio
    # como por su final. Por ejemplo, si tenemos una matriz llamada "ejemplo" de dimensión 5x4 y queremos que sea cuadrada podemos
    # añadirle una columna al final (es decir una quinta columna), de la siguiente manera:
    # ejemplo_pad = np.pad(ejemplo, ((0, 0), (0, 1)), 'constant', constant_values = (0, 0))
    # Si queremos añadirle una columna al inicio (es decir una primera columna y desplazar el resto de columnas 1 índice) debemos
    # hacer: ejemplo_pad = np.pad(ejemplo, ((0, 0), (1, 0)), 'constant', constant_values = (0, 0)). El parámetro constant_values
    # indica a np.pad con que valor rellenar las nuevas columnas/filas del array.

    # Por tanto, si queremos hacer padding de ceros de nuestras imágenes, es decir aumentar el número de filas y columnas de la
    # imagen, debemos aumentar la altura (n_H) y la anchura de la imagen (n_W) "pad" veces tanto al inicio como al final de cada una
    # de las dimensiones y rellenar esas filas y columnas con ceros (constant_values = (0, 0))

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

    # Se devuelve el dataset de imágenes con padding
    return X_pad
```

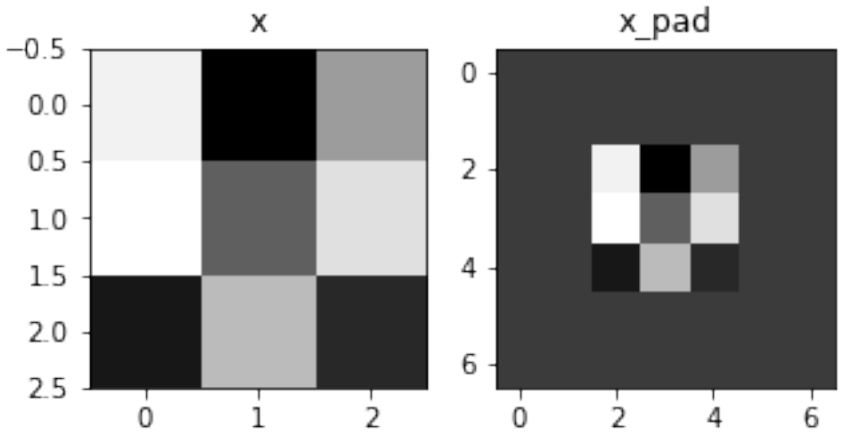
[6]:
```
# COMPROBACIÓN DE QUE zero_pad FUNCIONA

np.random.seed(1)
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])

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

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

[6]: `<matplotlib.image.AxesImage at 0x7f930909ce50>`



Expected Output:

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

```
[ 0.  0.]
[ 0.  0.]
[ 0.  0.]
[ 0.  0.]
[ 0.  0.]]
```

1.3.2 3.2 - Single step of convolution

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

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

Figure 2 : **Convolution operation** with a filter of 3x3 and a stride of 1 (stride = amount you move the window each time you slide)

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

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

Exercise: Implement conv_single_step(). Hint.

Note: The variable b will be passed in as a numpy array. If we add a scalar (a float or integer) to a numpy array, the result is a numpy array. In the special case when a numpy array contains a single value, we can cast it as a float to convert it to a scalar.

[7]:
```
# FUNCIÓN AUXILIAR PARA REALIZAR UNA CONVOLUCIÓN ENTRE UN FILTRO Y UN TROZO DE␣
↪UNA IMAGEN DE IGUAL TAMAÑO QUE EL FILTRO

def conv_single_step(a_slice_prev, W, b):
    """
    DEFINICIÓN:
    Esta función realiza la operación de convolución a un trozo de una imagen.␣
↪Para ello recibe por
    parámetro el trozo de la imagen, un filtro de convolución y el bias.

    PARÁMETROS:
    a_slice_prev -- Un trozo de la imagen que proviene de la capa anterior de␣
↪la red. Su shape es el siguiente:
    (f, f, n_C_prev). Las f's son la altura y la anchura del filtro de␣
↪convolución (las dos dimensiones son
    iguales) y n_C_prev es el número de canales que tiene la imagen de la capa␣
↪anterior de la red.
```

```
    W -- Filtro de convolución de tamaño (f, f, n_C_prev), donde las f's son la altura y la anchura del filtro
    y n_C_prev es el número de canales de la imagen anterior. Como estamos en una red neuronal, este filtro
    de convolución también son los pesos entre la capa actual y la anterior.
    b -- Matriz de bias. Su shape es de: (1, 1, 1)

    DEVUELVE:
    Z -- Un valor escalar resultante de convolucionar el trozo de la imagen con el filtro y luego aplicar
    el bias.
    """

    # Para realizar la convolución primero se debe multiplicar el trozo de la imagen con el filtro de convolución
    # elemento a elemento:
    s = np.multiply(a_slice_prev, W)

    # Seguidamente se deben sumar todos los elementos de la matriz resultante de multiplicar a_slice_prev por W
    # elemento a elemento:
    Z = np.sum(s)

    # Al valor escalar resultante le sumamos el bias (transformadao anteriormente en "float"):
    Z += float(b)

    # Finalmente devolvemos un valor escalar ("float") que es el resultado de la convolución entre un trozo de la
    # imagen y el filtro de convolución:
    return Z
```

[8]:
```
# COMPROBACIÓN DE QUE conv_single_step FUNCIONA

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

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

```
Z = -6.999089450680221
```

Expected Output:

Z

-6.99908945068

1.3.3 3.3 - Convolutional Neural Networks - Forward pass

In the forward pass, you will take many filters and convolve them on the input. Each 'convolution' gives you a 2D matrix output. You will then stack these outputs to get a 3D volume:

Exercise: Implement the function below to convolve the filters `W` on an input activation `A_prev`. This function takes the following inputs: * `A_prev`, the activations output by the previous layer (for a batch of m inputs); * Weights are denoted by `W`. The filter window size is `f` by `f`. * The bias vector is `b`, where each filter has its own (single) bias.

Finally you also have access to the hyperparameters dictionary which contains the stride and the padding.

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

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

Notice how this gives a 3D slice that has height 2, width 2, and depth 3. Depth is the number of channels.
This will be useful when you will define `a_slice_prev` below, using the `start/end` indexes you will define. 2. To define a_slice you will need to first define its corners `vert_start`, `vert_end`, `horiz_start` and `horiz_end`. This figure may be helpful for you to find out how each of the corner can be defined using h, w, f and s in the code below.

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

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

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

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

$$n_C = \text{number of filters used in the convolution}$$

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

Additional Hints if you're stuck

- You will want to use array slicing (e.g.`varname[0:1,:,3:5]`) for the following variables: `a_prev_pad` ,`W`, `b`
  Copy the starter code of the function and run it outside of the defined function, in separate cells.
  Check that the subset of each array is the size and dimension that you're expecting.

- To decide how to get the vert_start, vert_end; horiz_start, horiz_end, remember that these are indices of the previous layer.
  Draw an example of a previous padded layer (8 x 8, for instance), and the current (output layer) (2 x 2, for instance).
  The output layer's indices are denoted by `h` and `w`.

- Make sure that `a_slice_prev` has a height, width and depth.
- Remember that `a_prev_pad` is a subset of `A_prev_pad`.
  Think about which one should be used within the for loops.

[9]:
```
# CONVOLUCIÓN FORWARD

def conv_forward(A_prev, W, b, hparameters):
    """
    DEFINICIÓN:
    Esta función implementa la convolución completa de una imagen, para ello␣
↪recibe por parámetro
    el conjunto de imágenes resultantes de la activación de la capa anterior,␣
↪los filtros de convolución (es
    decir los pesos de las conexiones entre la capa anterior y la capa actual),␣
↪el bias y los hiper-parámetros
    de la capa (en forma de diccionario).

    PARÁMETROS:
    A_prev -- Conjunto de imágenes resultantes de la activación de la capa␣
↪previa. Es una array de NumPy con
    los siguientes tamaños (m, n_H_prev, n_W_prev, n_C_prev) donde m es el␣
↪número de imágenes por batch,
    n_H_prev es la altura de las imágenes en la capa anterior, n_W_prev la␣
↪anchura de las imágenes de la capa
    anterior y n_C_prev el número de canales de las imágenes de la capa␣
↪anterior.
    W -- Filtros de convolución (los pesos entre capas). Son una array de NumPy␣
↪con los siguientes tamaños
    (f, f, n_C_prev, n_C), donde las f's son la altura y la anchura de los␣
↪filtros, n_C_prev es el número de canales
    de las imágenes de la capa anterior y n_C son el número de canales actuales.
    b -- Array de NumPy con los bias de la capa de convolución. Tiene los␣
↪siguientes tamaños: (1, 1, 1, n_C)
    hparameters -- Diccionario con los hiper-parámetros 'stride' y 'pad'.

    DEVUELVE:
    Z -- Array de NumPy con el conjunto de imágenes convolucionadas. Tiene los␣
↪siguientes tamaños: (m, n_H, n_W, n_C),
    donde m es el número de imágenes por batch, n_H es la altura de la imágen␣
↪actual, n_W su anchura actual y n_C
    el número de canales actuales.
```

```
    cache -- Estructura de datos que guarda los valores de las variables␣
↪necesarias para el backpropagation: (A_prev, W, b, hparameters)
    """

    # Primero se obtienen las dimensiones del conjunto de imágenes resultantes␣
↪de la activación de la capa anterior
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Seguidamente se obtienen las dimensiones de los filtros de convolución:
    (f, f, n_C_prev, n_C) = W.shape

    # A continuación se obtienen los hiper-parámetros del diccionario␣
↪hparameters:
    stride = hparameters["stride"]
    pad = hparameters["pad"]

    # Luego se calculan, mediante las expresiones vistas en la teoria, la␣
↪altura y la anchura de las imágenes
    # resultantes de la convolución de esta capa (se utiliza int para aplicar␣
↪la función suelo):
    n_H = int((n_H_prev - f + 2 * pad) / stride) + 1
    n_W = int((n_W_prev - f + 2 * pad) / stride) + 1

    # Se inicializan todos los elementos a cero de la array multidimensional de␣
↪salida de la función:
    Z = np.zeros((m, n_H, n_W, n_C))

    # Se realiza padding al conjunto de imágenes de la capa anterior:
    A_prev_pad = zero_pad(A_prev, pad)

    # A continuación se van a generar las nuevas imágenes (las imágenes de la␣
↪capa actual) a partir de la convolución
    # entre las imágenes resultantes de la activación de la capa previa y los␣
↪filtros de convolución. Para
    # construir estas nuevas imágenes se iterará sobre el batch y se irá␣
↪selaccionando cada imagen previa. Seguidamente
    # se seleccionarán trozos de la imagen anterior y se convolucionarán␣
↪obteniendo así un valor escalar. Este
    # escalar se colocará en su posición correspondiente de la nueva imagen.

    # Se itera sobre cada imagen del lote de imágenes resultantes de la␣
↪activación de la capa previa.
    for i in range(m):

        # Una vez seleccionada una imagen previa "i", se itera sobre el eje␣
↪vertical de la imagen que dará lugar a
```

```
        # esta imagen "i" cuando sea convolucionada.
        for h in range(n_H):

            # Seguidamente se itera sobre el eje horizontal de la imagen que␣
↪dará lugar esta imagen "i" cuando
            # sea convolucionada.
            for w in range(n_W):

                # Se genera una ventana que se utilizará para seleccionar que␣
↪trozo de la imagen previa se debe
                # convolucionar para generar el valor de la posición actual␣
↪(h,w) de la nueva imagen. Para
                # construir esta ventana se deben definir sus esquinas. La␣
↪esquina vertical inicial se define
                # como la fila "h" de la nueva imagen por el "stride". La␣
↪esquina vertical final se define como
                # la esquina vertical inicial más el tamaño del filtro (ya que␣
↪el trozo de imagen que se debe
                # seleccionar debe ser del tamaño del filtro de convolución).␣
↪La esquina horizontal inicial
                # se define como la columna "w" de la nueva imagen por el␣
↪"stride". Finalmente la esquina
                # horizontal final se define como a esquina horizontal inicial␣
↪más el tamaño del filtro.
                vert_start = stride * h
                vert_end = vert_start + f
                horiz_start = stride * w
                horiz_end = horiz_start + f

                # Bucle sobre los canales de la imagen de salida
                for c in range(n_C):

                    # Se utiliza la ventana previamente definida para␣
↪seleccionar un trozo (en 3D) de la imagen
                    # "i" tras haberle realizado el padding.
                    a_slice_prev = A_prev_pad[i, vert_start:vert_end,␣
↪horiz_start:horiz_end, :]

                    # Se convoluciona el volumen seleccionado, por cada canal␣
↪que tiene esta capa de convolución,
                    # utilizando la función conv_single_step anteriormente␣
↪definida. De esta manera se genera el
                    # valor de la posición (h,w) de la nueva imagen en cada uno␣
↪de sus canales.
                    weights = W[:, :, :, c]
                    biases = b[:, :, :, c]
```

```
                    Z[i, h, w, c] = conv_single_step(a_slice_prev, weights,␣
↪biases)

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

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

    return Z, cache
```

[10]:
```
# COMPROBACIÓN DE QUE conv_forward FUNCIONA

np.random.seed(1)
A_prev = np.random.randn(10,5,7,4)
W = np.random.randn(3,3,4,8)
b = np.random.randn(1,1,1,8)
hparameters = {"pad" : 1,
               "stride": 2}

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])
```

```
Z's mean =
 0.6923608807576933
Z[3,2,1] =
 [-1.28912231  2.27650251  6.61941931  0.95527176  8.25132576  2.31329639
 13.00689405  2.34576051]
cache_conv[0][1][2][3] =
 [-1.1191154   1.9560789  -0.3264995  -1.34267579]
```

Expected Output:

```
Z's mean =
 0.692360880758
Z[3,2,1] =
 [ -1.28912231   2.27650251   6.61941931   0.95527176   8.25132576
   2.31329639  13.00689405   2.34576051]
cache_conv[0][1][2][3] = [-1.1191154   1.9560789  -0.3264995  -1.34267579]
```

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

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

You don't need to do it here.

1.4 4 - Pooling layer

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

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

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

1.4.1 4.1 - Forward Pooling

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

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

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

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

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

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

[11]:
```
# POOLING FORWARD

def pool_forward(A_prev, hparameters, mode = "max"):
    """
    DEFINICIÓN:
    Esta función implementa la capa de pooling, para ello recibe por parámetro␣
 ↪el conjunto de imágenes resultantes
    de la activación de la capa anterior, los hiper-parámetros de la capa (en␣
 ↪forma de diccionario) y el modo de la
    capa.

    PARÁMETROS:
```

```
    A_prev -- Conjunto de imágenes resultantes de la activación de la capa␣
↪previa. Es una array de NumPy con
    los siguientes tamaños (m, n_H_prev, n_W_prev, n_C_prev) donde m es el␣
↪número de imágenes por batch,
    n_H_prev es la altura de las imágenes en la capa anterior, n_W_prev la␣
↪anchura de las imágenes de la capa
    anterior y n_C_prev el número de canales de las imágenes de la capa␣
↪anterior.
    hparameters -- Diccionario con los hiper-parámetros 'stride' y 'pad'.
    mode -- String que permite seleccionar la operación de "max-pooling" o la␣
↪de "average-pooling".

    DEVUELVE:
    A -- Array multidimensional de tamaño (m, n_H, n_W, n_C) (donde m es el␣
↪número de imágenes por batch, n_H es la
    altura de la imágen actual, n_W su anchura actual y n_C el número de␣
↪canales actuales) que representa un
    conjunto de imágenes que han pasado la capa de pooling.
    cache -- Estructura de datos que guarda los valores de las variables␣
↪necesarias para el backpropagation: (A, hparameters)
    """

    # Primero se obtienen las dimensiones del conjunto de imágenes resultantes␣
↪de la activación de la capa anterior:
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # A continuación se obtienen los hiper-parámetros del diccionario␣
↪hparameters:
    f = hparameters["f"]
    stride = hparameters["stride"]

    # Luego se calculan, mediante las expresiones vistas en la teoria, la␣
↪altura y la anchura de las imágenes
    # resultantes del pooling de esta capa (se utiliza int para aplicar la␣
↪función suelo):
    n_H = int(1 + (n_H_prev - f) / stride)
    n_W = int(1 + (n_W_prev - f) / stride)
    n_C = n_C_prev

    # Se inicializan todos los elementos de la array multidimensional de salida␣
↪de la función a cero:
    A = np.zeros((m, n_H, n_W, n_C))

    # A continuación se van a generar las nuevas imágenes (las imágenes de la␣
↪capa actual) a partir de realizar la
```

```
   # operación de pooling a todas las imágenes resultantes de la activación de
↪la capa previa. Para construir
   # estas nuevas imágenes se debe iterar sobre el batch e ir seleccionando
↪cada imagen previa. Seguidamente
   # se seleccionan trozos de la imagen anterior y se les aplica la operación
↪de pooling obteniendo así un valor
   # escalar. Este escalar se colocará en su posición correspondiente de la
↪nueva imagen.

  # Se itera sobre cada imagen del lote de imágenes resultantes de la
↪activación de la capa previa.
   for i in range(m):

       # Una vez seleccionada una imagen previa "i", se itera sobre el eje
↪vertical de la imagen que dará lugar
       # esta imagen "i" cuando se le aplique la operación de padding.
       for h in range(n_H):

           # Seguidamente se itera sobre el eje horizontal de la imagen que
↪dará lugar a esta imagen "i" cuando
           # sea convolucionada.
           for w in range(n_W):

               # Se genera la misma ventana que en la operación de convolución
↪para seleccionar el trozo de la
               # imagen.
               vert_start = stride * h
               vert_end = vert_start + f
               horiz_start = stride * w
               horiz_end = horiz_start + f

               # Bucle sobre los canales de la imagen de salida
               for c in range (n_C):

                   # Se utilizan los bordes previamente definidos para
↪seleccionar un trozo de la imagen previa
                   # en uno de sus canales (es decir que el trozo tiene 2
↪dimensiones).
                   a_prev_slice = A_prev[i, vert_start:vert_end, horiz_start:
↪horiz_end, c]

                   # Se realiza la operación de pooling dependiendo del modo
↪seleccionado. Si el modo es "max"
                   # se coge el mayor valor del trozo de la imagen y se guarda
↪dentro de la posición correspondiente
```

```
                    # de la nueva imagen. Si el modo es "average" se realiza una media de todo el trozo del la imagen
                    # y luego se guarda dentro de la posición correspondiente de la nueva imagen.
                    if mode == "max":
                        A[i, h, w, c] = np.max(a_prev_slice)

                    elif mode == "average":
                        A[i, h, w, c] = np.mean(a_prev_slice)

    # Store the input and hparameters in "cache" for pool_backward()
    cache = (A_prev, hparameters)

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

    return A, cache
```

[12]:
```
# COMPROBACIÓN DE QUE pool_forward FUNCIONA

# Case 1: stride of 1
np.random.seed(1)
A_prev = np.random.randn(2, 5, 5, 3)
hparameters = {"stride" : 1, "f": 3}

A, cache = pool_forward(A_prev, hparameters)
print("mode = max")
print("A.shape = " + str(A.shape))
print("A =\n", A)
print()
A, cache = pool_forward(A_prev, hparameters, mode = "average")
print("mode = average")
print("A.shape = " + str(A.shape))
print("A =\n", A)
```

```
mode = max
A.shape = (2, 3, 3, 3)
A =
 [[[[1.74481176 0.90159072 1.65980218]
    [1.74481176 1.46210794 1.65980218]
    [1.74481176 1.6924546  1.65980218]]

   [[1.14472371 0.90159072 2.10025514]
    [1.14472371 0.90159072 1.65980218]
    [1.14472371 1.6924546  1.65980218]]

   [[1.13162939 1.51981682 2.18557541]
```

```
   [1.13162939 1.51981682 2.18557541]
   [1.13162939 1.6924546  2.18557541]]]


 [[[1.19891788 0.84616065 0.82797464]
   [0.69803203 0.84616065 1.2245077 ]
   [0.69803203 1.12141771 1.2245077 ]]

  [[1.96710175 0.84616065 1.27375593]
   [1.96710175 0.84616065 1.23616403]
   [1.62765075 1.12141771 1.2245077 ]]

  [[1.96710175 0.86888616 1.27375593]
   [1.96710175 0.86888616 1.23616403]
   [1.62765075 1.12141771 0.79280687]]]]

mode = average
A.shape = (2, 3, 3, 3)
A =
 [[[[-3.01046719e-02 -3.24021315e-03 -3.36298859e-01]
    [ 1.43310483e-01  1.93146751e-01 -4.44905196e-01]
    [ 1.28934436e-01  2.22428468e-01  1.25067597e-01]]

  [[-3.81801899e-01  1.59993515e-02  1.70562706e-01]
    [ 4.73707165e-02  2.59244658e-02  9.20338402e-02]
    [ 3.97048605e-02  1.57189094e-01  3.45302489e-01]]

  [[-3.82680519e-01  2.32579951e-01  6.25997903e-01]
    [-2.47157416e-01 -3.48524998e-04  3.50539717e-01]
    [-9.52551510e-02  2.68511000e-01  4.66056368e-01]]]


 [[[-1.73134159e-01  3.23771981e-01 -3.43175716e-01]
    [ 3.80634669e-02  7.26706274e-02 -2.30268958e-01]
    [ 2.03009393e-02  1.41414785e-01 -1.23158476e-02]]

  [[ 4.44976963e-01 -2.61694592e-03 -3.10403073e-01]
    [ 5.08114737e-01 -2.34937338e-01 -2.39611830e-01]
    [ 1.18726772e-01  1.72552294e-01 -2.21121966e-01]]

  [[ 4.29449255e-01  8.44699612e-02 -2.72909051e-01]
    [ 6.76351685e-01 -1.20138225e-01 -2.44076712e-01]
    [ 1.50774518e-01  2.89111751e-01  1.23238536e-03]]]]
```

** Expected Output**

```
mode = max
A.shape = (2, 3, 3, 3)
A =
```

```
 [[[[ 1.74481176  0.90159072  1.65980218]
   [ 1.74481176  1.46210794  1.65980218]
   [ 1.74481176  1.6924546   1.65980218]]

  [[ 1.14472371  0.90159072  2.10025514]
   [ 1.14472371  0.90159072  1.65980218]
   [ 1.14472371  1.6924546   1.65980218]]

  [[ 1.13162939  1.51981682  2.18557541]
   [ 1.13162939  1.51981682  2.18557541]
   [ 1.13162939  1.6924546   2.18557541]]]


 [[[ 1.19891788  0.84616065  0.82797464]
   [ 0.69803203  0.84616065  1.2245077 ]
   [ 0.69803203  1.12141771  1.2245077 ]]

  [[ 1.96710175  0.84616065  1.27375593]
   [ 1.96710175  0.84616065  1.23616403]
   [ 1.62765075  1.12141771  1.2245077 ]]

  [[ 1.96710175  0.86888616  1.27375593]
   [ 1.96710175  0.86888616  1.23616403]
   [ 1.62765075  1.12141771  0.79280687]]]]

mode = average
A.shape = (2, 3, 3, 3)
A =
 [[[[ -3.01046719e-02  -3.24021315e-03  -3.36298859e-01]
   [  1.43310483e-01   1.93146751e-01  -4.44905196e-01]
   [  1.28934436e-01   2.22428468e-01   1.25067597e-01]]

  [[ -3.81801899e-01   1.59993515e-02   1.70562706e-01]
   [  4.73707165e-02   2.59244658e-02   9.20338402e-02]
   [  3.97048605e-02   1.57189094e-01   3.45302489e-01]]

  [[ -3.82680519e-01   2.32579951e-01   6.25997903e-01]
   [ -2.47157416e-01  -3.48524998e-04   3.50539717e-01]
   [ -9.52551510e-02   2.68511000e-01   4.66056368e-01]]]


 [[[ -1.73134159e-01   3.23771981e-01  -3.43175716e-01]
   [  3.80634669e-02   7.26706274e-02  -2.30268958e-01]
   [  2.03009393e-02   1.41414785e-01  -1.23158476e-02]]

  [[  4.44976963e-01  -2.61694592e-03  -3.10403073e-01]
   [  5.08114737e-01  -2.34937338e-01  -2.39611830e-01]
   [  1.18726772e-01   1.72552294e-01  -2.21121966e-01]]
```

```
  [[  4.29449255e-01   8.44699612e-02  -2.72909051e-01]
   [  6.76351685e-01  -1.20138225e-01  -2.44076712e-01]
   [  1.50774518e-01   2.89111751e-01   1.23238536e-03]]]]
```

[13]:
```
# COMPROBACIÓN DE QUE pool_forward FUNCIONA

# Case 2: stride of 2
np.random.seed(1)
A_prev = np.random.randn(2, 5, 5, 3)
hparameters = {"stride" : 2, "f": 3}

A, cache = pool_forward(A_prev, hparameters)
print("mode = max")
print("A.shape = " + str(A.shape))
print("A =\n", A)
print()

A, cache = pool_forward(A_prev, hparameters, mode = "average")
print("mode = average")
print("A.shape = " + str(A.shape))
print("A =\n", A)
```

```
mode = max
A.shape = (2, 2, 2, 3)
A =
 [[[[1.74481176 0.90159072 1.65980218]
    [1.74481176 1.6924546  1.65980218]]

   [[1.13162939 1.51981682 2.18557541]
    [1.13162939 1.6924546  2.18557541]]]


 [[[1.19891788 0.84616065 0.82797464]
    [0.69803203 1.12141771 1.2245077 ]]

   [[1.96710175 0.86888616 1.27375593]
    [1.62765075 1.12141771 0.79280687]]]]

mode = average
A.shape = (2, 2, 2, 3)
A =
 [[[[-0.03010467 -0.00324021 -0.33629886]
    [ 0.12893444  0.22242847  0.1250676 ]]

   [[-0.38268052  0.23257995  0.6259979 ]
    [-0.09525515  0.268511    0.46605637]]]
```

```
 [[[-0.17313416  0.32377198 -0.34317572]
   [ 0.02030094  0.14141479 -0.01231585]]

  [[ 0.42944926  0.08446996 -0.27290905]
   [ 0.15077452  0.28911175  0.00123239]]]]
```

Expected Output:

```
mode = max
A.shape = (2, 2, 2, 3)
A =
 [[[[ 1.74481176  0.90159072  1.65980218]
   [ 1.74481176  1.6924546   1.65980218]]

  [[ 1.13162939  1.51981682  2.18557541]
   [ 1.13162939  1.6924546   2.18557541]]]


 [[[ 1.19891788  0.84616065  0.82797464]
   [ 0.69803203  1.12141771  1.2245077 ]]

  [[ 1.96710175  0.86888616  1.27375593]
   [ 1.62765075  1.12141771  0.79280687]]]]

mode = average
A.shape = (2, 2, 2, 3)
A =
 [[[[-0.03010467 -0.00324021 -0.33629886]
   [ 0.12893444  0.22242847  0.1250676 ]]

  [[-0.38268052  0.23257995  0.6259979 ]
   [-0.09525515  0.268511    0.46605637]]]


 [[[-0.17313416  0.32377198 -0.34317572]
   [ 0.02030094  0.14141479 -0.01231585]]

  [[ 0.42944926  0.08446996 -0.27290905]
   [ 0.15077452  0.28911175  0.00123239]]]]
```

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

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

1.5 5 - Backpropagation in convolutional neural networks (OPTIONAL / UNGRADED)

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

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

1.5.1 5.1 - Convolutional layer backward pass

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

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

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

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

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

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

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

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

Where a_{slice} corresponds to the slice which was used to generate the activation Z_{ij}. Hence, this ends up giving us the gradient for W with respect to that slice. Since it is the same W, we will just add up all such gradients to get dW.

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

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

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

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

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

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

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

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

[14]:
```
# BACKPROPAGATION DE LA CAPA DE CONVOLUCIÓN

def conv_backward(dZ, cache):
    """
    DEFINICIÓN:
    Esta función implementa el backpropagation de la capa de convolución, para␣
↪ello recibe por parámetro el gradiente
    del coste con respecto al output de la capa de convolución y el cache de␣
↪esta capa.

    PARÁMETROS:
    dZ -- Gradiente del coste respecto al output de la capa de convolución (Z).␣
↪Es un array de NumPy con las siguientes
    dimensiones (m, n_H, n_W, n_C), donde m es el número de imágenes por batch,␣
↪n_H es la altura de las imágenes de la capa
    en cuestión, n_W la anchura de las imágenes de esta capa y n_C el número de␣
↪canales de las imágenes.
    cache -- Estructura de datos guardados a la hora de realizar la convolución␣
↪forward para poder ejecutar el backpropagation
    de forma más sencilla. Los datos guardados son: (A_prev, W, b, hparameters).
↪

    DEVUELVE:
    dA_prev -- Gradiente del coste respecto al input de la capa de convolución␣
↪(A_prev). Es un array de NumPy con las siguientes
    dimensiones (m, n_H_prev, n_W_prev, n_C_prev), donde m es el número de␣
↪imágenes por batch, n_H_prev es la altura de las imágenes
    resultantes de la activación de la capa previa, n_W_prev la anchura de las␣
↪imágenes de esa capa y n_C_prev el número de canales
```

```
    de cada imagen de la capa previa.
    dW -- Gradiente del coste respeto a los pesos de la capa de convolución (es␣
↪decir el gradiente del coste de los valores de los
    filtros de convolución). Es un array de NumPy con las siguientes␣
↪dimensiones (f, f, n_C_prev, n_C) , donde las f's son la altura
    y la anchura de los filtros, n_C_prev es el número de canales de las␣
↪imágenes de la capa anterior y n_C son el número de canales
    de las imágenes de la capa actual.
    db -- Gradiente del coste respecto al bias de la capa de convolución (b) Es␣
↪una array de NumPy con las siguientes dimensiones:
    (1, 1, 1, n_C), donde n_C son el número de canales de las imágenes de la␣
↪capa actual.
    """

    # Se obiene la información del cache
    (A_prev, W, b, hparameters) = cache

    # Se obtienen las dimensiones de las imágenes previas a la capa de␣
↪convolución
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape

    # Se obtienen las dimensiones de los filtros de convolución
    (f, f, n_C_prev, n_C) = W.shape

    # Se obtienen los valores de los hiper-parámetros
    stride = hparameters["stride"]
    pad = hparameters["pad"]

    # Se obtienen las dimensiones del gradiente del coste con respecto al␣
↪output de la capa de convolución (Z)
    (m, n_H, n_W, n_C) = dZ.shape

    # Se inicializa dA_prev, dW y db con valores random y con sus dimensiones␣
↪correspondientes
    dA_prev = np.random.randn(m, n_H_prev, n_W_prev, n_C_prev)
    dW = np.random.randn(f, f, n_C_prev, n_C)
    db = np.random.randn(1, 1, 1, n_C)

    # Se realiza padding a las imágenes de la capa previa y a su gradiente de␣
↪coste
    A_prev_pad = zero_pad(A_prev, pad)
    dA_prev_pad = zero_pad(dA_prev, pad)

    # Se itera sobre cada imagen del batch.
    for i in range(m):
```

```
        # Se selecciona la imagen "i" previa (con padding) y su correspondiente␣
↪gradiente de coste
        a_prev_pad = A_prev_pad[i, :, :, :]
        da_prev_pad = dA_prev_pad[i, :, :, :]

        # Se itera sobre el eje vertical de la imagen "i" post-convolución
        for h in range(n_H):

            # Se itera sobre el eje horizontal de la imagen "i" post-convolución
            for w in range(n_W):

                # Se itera sobre los canales de la imagen "i" post-convolución
                for c in range(n_C):

                    # Se genera la misma ventana que en la operación de␣
↪convolución (forward) para seleccionar un trozo de la
                    # imagen.
                    vert_start = stride * h
                    vert_end = vert_start + f
                    horiz_start = stride * w
                    horiz_end = horiz_start + f

                    # Se utiliza la ventana para seleccionar un trozo de la␣
↪imagen previa a la convolución
                    a_slice = a_prev_pad[vert_start:vert_end, horiz_start:
↪horiz_end, :]

                    # Se actualizan los gradientes mediante las expresiones␣
↪explicadas en la teoría de este trabajo
                    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]

        # Se le deshace el padding al gradiente del coste con respecto al input␣
↪de la capa de convolución
        dA_prev[i, :, :, :] = da_prev_pad[pad:-pad, pad:-pad, :]

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

    return dA_prev, dW, db
```

[15]:
```
# COMPROBACIÓN DE QUE conv_backward FUNCIONA

# We'll run conv_forward to initialize the 'Z' and 'cache_conv",
```

```
# which we'll use to test the conv_backward function
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)

# Test conv_backward
dA, dW, db = conv_backward(Z, cache_conv)
print("dA_mean =", np.mean(dA))
print("dW_mean =", np.mean(dW))
print("db_mean =", np.mean(db))
```

```
dA_mean = 1.457943469708148
dW_mean = 1.7519334429380524
db_mean = 7.259465017141471
```

** Expected Output: **

dA_mean

1.45243777754

dW_mean

1.72699145831

db_mean

7.83923256462

1.6 5.2 Pooling layer - backward pass

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

1.6.1 5.2.1 Max pooling - backward pass

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

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

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

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

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

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

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

[16]:
```
# FUNCIÓN AUXILIAR PARA CREAR UNA MÁSCARA DE UN TROZO DE LA IMAGEN

def create_mask_from_window(x):
    """
    DEFINICIÓN:
    Esta función genera una máscara de booleanos que identifica con un 1 el␣
↪valor más alto de una matriz.
    Por ello se le pasa por parámetro una matriz que simboliza un trozo de una␣
↪imagen.

    PARÁMETROS:
    x -- Array de dos dimensiones con tamaño: (f, f), donde las f's son el␣
↪tamaño de los filtros de convolución.

    DEVUELVE:
    mask -- Array del mismo tamaño que el array de entrada que contiene un 1 en␣
↪la posición donde se encuentra
    el valor más alto del array de entrada y 0's en el resto de posiciones.
    """

    # Se obtiene el valor máximo del array de entrada.
    max = np.max(x)

    # Se pone a "true" la posición donde se encuentra el valor máximo del array␣
↪de entrada. El resto de posiciones
    # se queda a "false".
    mask = (x == max)

    # Se devuelve la máscara
    return mask
```

[17]:
```
# COMPROBACIÓN DE QUE create_mask_from_window FUNCIONA

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

```
print("mask = ", mask)
```

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

Expected Output:

x =

[[1.62434536 -0.61175641 -0.52817175] [-1.07296862 0.86540763 -2.3015387]]

mask =

[[True False False] [False False False]]

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

1.6.2 5.2.2 - Average pooling - backward pass

In max pooling, for each input window, all the "influence" on the output came from a single input value–the max. In average pooling, every element of the input window has equal influence on the output. So to implement backprop, you will now implement a helper function that reflects this.

For example if we did average pooling in the forward pass using a 2x2 filter, then the mask you'll use for the backward pass will look like:

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

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

Exercise: Implement the function below to equally distribute a value dz through a matrix of dimension shape. Hint

[20]:
```
# FUNCIÓN AUXILIAR PARA CREAR UNA MÁSCARA DE UN TROZO DE LA IMAGEN

def distribute_value(dz, shape):
    """
    DEFINICIÓN:
    Esta función distribuye (como si se realizará la inversa de la operación de la media) el valor de entrada
    por toda una matriz.

    PARÁMETROS:
    dz -- Valor de entrada, es un escalar
```

```
    shape -- Dimensiones de la matriz de salida (n_H, n_W), donde n_H es la␣
↪altura del trozo de la matriz que se
    le pasa por parámetro y n_W es la anchura del trozo de la matriz

    DEVUELVE:
    a -- Array de tamaño (n_H, n_W) por donde se le ha distribuído todo el␣
↪valor de dz
    """

    # Se obtiene los tamaños de la matriz de salida
    (n_H, n_W) = shape

    # Se divide dz entre todo el tamaño de la matriz de salida para obtener el␣
↪valor de cada elemento de la nueva matriz
    average = dz / (n_H * n_W)

    # Se genera la matriz de salida la cual contiene en cada uno de sus␣
↪elementos el "average"
    a = np.ones(shape) * average

    # Se devuelve la matriz
    return a
```

[21]:
```
a = distribute_value(2, (2,2))
print('distributed value =', a)
```

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

Expected Output:

distributed_value =

[[0.5 0.5]
 [0.5 0.5]]

1.6.3 5.2.3 Putting it together: Pooling backward

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

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

[22]:
```
def pool_backward(dA, cache, mode = "max"):
    """
```

```
    DEFINICIÓN:
    Esta función implementa el backpropagation de la capa de pooling, para ello␣
↪recibe por parámetro el gradiente
    del coste respecto al output de la capa de pooling, el cache de esa misma␣
↪capa y el modo de operación de
    la capa.

    PARÁMETROS:
    dA -- Gradiente del coste respecto al output de la capa de pooling. Este␣
↪gradiente tiene el mismo tamaño
    que las imágenes post-pooling es decir: (m, n_H, n_W, n_C), donde m es el␣
↪número de imágenes por batch, n_H
    es la altura de las imágenes post-pooling, n_W la anchura de las imágenes␣
↪post-pooling y n_C el número de
    canales de las imágenes.
    cache -- Estructura de datos guardados a la hora de realizar la operación␣
↪de pooling forward para poder
    ejecutar el backpropagation de forma más sencilla. Los datos guardados son:␣
↪(A, hparameters), donde A son las
    imágenes post-pooling y hparameters son los hiper-parámetros.
    mode -- El modo de pooling que se desea utilizar, puede ser "max" o␣
↪"average".

    DEVUELVE:
    dA_prev -- Gradiente del coste respecto al input de la capa de pooling␣
↪(A_prev). Es un array de NumPy con las siguientes
    dimensiones (m, n_H_prev, n_W_prev, n_C_prev), donde m es el número de␣
↪imágenes por batch, n_H_prev es la altura de las imágenes
    resultantes de la activación de la capa previa, n_W_prev la anchura de las␣
↪imágenes de esa capa y n_C_prev el número de canales
    de cada imagen de la capa previa.
    """

    # Se obiene la información del cache
    (A_prev, hparameters) = cache

    # Se obtienen los valores de los hiper-parámetros
    f = hparameters["f"]
    stride = hparameters["stride"]

    # Se obtienen las dimensiones de las imágenes pre-pooling y del gradiente␣
↪del coste post-pooling.
    (m, n_H_prev, n_W_prev, n_C_prev) = A_prev.shape
    (m, n_H, n_W, n_C) = dA.shape
```

```
    # Se inicializa el gradiente del coste respecto al input de la capa de␣
↪pooling con ceros y con el
    # tamaño de las imágenes de entrada de esta capa.
    dA_prev = np.zeros((m, n_H_prev, n_W_prev, n_C_prev))

    # Se itera sobre cada imagen del batch.
    for i in range(m):

        # Se selecciona una imagen pre-pooling de todo el lote
        a_prev = A_prev[i, :, :, :]

        # Se itera sobre el eje vertical de la imagen "i"
        for h in range(n_H):

            # Se itera sobre el eje horizontal de la imagen "i"
            for w in range(n_W):

                # Se itera sobre los canales de la imagen "i"
                for c in range(n_C):

                    # Se genera la misma ventana que en la operación de␣
↪convolución (forward) para seleccionar un trozo de la
                    # imagen.
                    vert_start = stride * h
                    vert_end = vert_start + f
                    horiz_start = stride * w
                    horiz_end = horiz_start + f

                    # Se calcula el backpropagation del modo seleccionado
                    if mode == "max":

                        # Se selecciona un trozo, en cada canal, de la imagen␣
↪pre-pooling utilizando la ventana anteriormente definida
                        a_prev_slice = a_prev[vert_start:vert_end, horiz_start:
↪horiz_end, c]

                        # Con ese trozo se genera una máscara que contiene un 1␣
↪en la posición donde el trozo de la imagen tiene el valor
                        # más alto. En el resto de posiciones se guardan 0s.
                        mask = create_mask_from_window(a_prev_slice)

                        # Se multiplica elemento a elemento la máscara y el␣
↪gradiente del coste respecto al output de la capa de pooling y
                        # luego se suma al gradiente del coste respecto al␣
↪input de la capa de pooling. De esta manera se colocan en dA_prev
```

```
                        # los gradientes del coste respecto al output de la␣
↪capa de pooling en las posiciones donde estaban los valores más
                        # altos de la imagen.
                        dA_prev[i, vert_start:vert_end, horiz_start:horiz_end,␣
↪c] += np.multiply(mask, dA[i, h, w, c])


                    elif mode == "average":

                        # Se obtiene el gradiente del coste de la posición de␣
↪la imagen en cuestión
                        da = dA[i, h, w, c]

                        # Se define el tamaño del filtro
                        shape = (f, f)

                        # Se distribuye el gradiente del coste respecto al␣
↪output de la capa de pooling por toda la ventana definida anteriormente
                        # (la cual tiene el tamaño del filtro)
                        dA_prev[i, vert_start: vert_end, horiz_start:␣
↪horiz_end, c] += distribute_value(da, shape)


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

    return dA_prev
```

[23]:
```
np.random.seed(1)
A_prev = np.random.randn(5, 5, 3, 2)
hparameters = {"stride" : 1, "f": 2}
A, cache = pool_forward(A_prev, hparameters)
dA = np.random.randn(5, 4, 2, 2)

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

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

```
 [ 5.05844394 -1.68282702]
 [ 0.          0.        ]]

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

Expected Output:

mode = max:

mean of dA =

0.145713902729

dA_prev[1,1] =

[[0. 0.] [5.05844394 -1.68282702] [0. 0.]]

mode = average

mean of dA =

0.145713902729

dA_prev[1,1] =

[[0.08485462 0.2787552] [1.26461098 -0.25749373] [1.17975636 -0.53624893]]

1.6.4 Congratulations !

Congratulations on completing this assignment. You now understand how convolutional neural networks work. You have implemented all the building blocks of a neural network.