**Convolutional Neural Networks: Step by Step**

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

By the end of this notebook, you'll be able to:

* Explain the convolution operation
* Apply two different types of pooling operation
* Identify the components used in a convolutional neural network (padding, stride, filter, ...) and their purpose
* Build a convolutional neural network

**Notation**:

* Superscript [�] denotes an object of the ��ℎ layer.
  + Example: �[4] is the 4�ℎ layer activation. �[5] and �[5] are the 5�ℎ layer parameters.
* Superscript (�) denotes an object from the ��ℎ example.
  + Example: �(�) is the ��ℎ training example input.
* Subscript � denotes the ��ℎ entry of a vector.
  + Example: ��[�] denotes the ��ℎ entry of the activations in layer �, assuming this is a fully connected (FC) layer.
* ��, �� and �� denote respectively the height, width and number of channels of a given layer. If you want to reference a specific layer �, you can also write ��[�], ��[�], ��[�].
* ������, ������ and ������ denote respectively the height, width and number of channels of the previous layer. If referencing a specific layer �, this could also be denoted ��[�−1], ��[�−1], ��[�−1].

You should be familiar with numpy and/or have completed the previous courses of the specialization. Let's get started!

**Table of Contents**

* [1 - Packages](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#1)
* [2 - Outline of the Assignment](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#2)
* [3 - Convolutional Neural Networks](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#3)
  + [3.1 - Zero-Padding](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#3-1)
    - [Exercise 1 - zero\_pad](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#ex-1)
  + [3.2 - Single Step of Convolution](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#3-2)
    - [Exercise 2 - conv\_single\_step](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#ex-2)
  + [3.3 - Convolutional Neural Networks - Forward Pass](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#3-3)
    - [Exercise 3 - conv\_forward](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#ex-3)
* [4 - Pooling Layer](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#4)
  + [4.1 - Forward Pooling](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#4-1)
    - [Exercise 4 - pool\_forward](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#ex-4)
* [5 - Backpropagation in Convolutional Neural Networks (OPTIONAL / UNGRADED)](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5)
  + [5.1 - Convolutional Layer Backward Pass](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5-1)
    - [5.1.1 - Computing dA](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5-1-1)
    - [5.1.2 - Computing dW](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5-1-2)
    - [5.1.3 - Computing db](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5-1-3)
      * [Exercise 5 - conv\_backward](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#ex-5)
  + [5.2 Pooling Layer - Backward Pass](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5-2)
    - [5.2.1 Max Pooling - Backward Pass](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5-2-1)
      * [Exercise 6 - create\_mask\_from\_window](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#ex-6)
    - [5.2.2 - Average Pooling - Backward Pass](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5-2-2)
      * [Exercise 7 - distribute\_value](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#ex-7)
    - [5.2.3 Putting it Together: Pooling Backward](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#5-2-3)
      * [Exercise 8 - pool\_backward](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/#ex-8)

**1 - Packages**

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

* [numpy](https://github.com/berkayalan/neural-networks-and-deep-learning/blob/590b5b40199f358e911da44f99f6c3e00b0c2934/www.numpy.org) is the fundamental package for scientific computing with Python.
* [matplotlib](http://matplotlib.org/) is a library to plot graphs in Python.
* np.random.seed(1) is used to keep all the random function calls consistent. This helps to grade your work.

In [1]:

**import** numpy **as** np

**import** h5py

**import** matplotlib.pyplot **as** plt

**from** public\_tests **import** **\***

**%matplotlib** inline

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

plt**.**rcParams['image.interpolation'] **=** 'nearest'

plt**.**rcParams['image.cmap'] **=** 'gray'

**%load\_ext** autoreload

**%autoreload** 2

np**.**random**.**seed(1)

**2 - Outline of the Assignment**

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

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

**3 - Convolutional Neural Networks**

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

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

**3.1 - Zero-Padding**

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

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

The main benefits of padding are:

* 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 at the edges of an image.

**Exercise 1 - zero\_pad**

Implement the following function, which pads all the images of a batch of examples X with zeros. [Use np.pad](https://docs.scipy.org/doc/numpy/reference/generated/numpy.pad.html). Note if you want to pad the array "a" of shape (5,5,5,5,5) with 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))

In [17]:

*# GRADED FUNCTION: zero\_pad*

**def** zero\_pad(X, pad):

"""

Pad with zeros all images of the dataset X. The padding is applied to the height and width of an image,

as illustrated in Figure 1.

Argument:

X -- python numpy array of shape (m, n\_H, n\_W, n\_C) representing a batch of m images

pad -- integer, amount of padding around each image on vertical and horizontal dimensions

Returns:

X\_pad -- padded image of shape (m, n\_H + 2 \* pad, n\_W + 2 \* pad, n\_C)

"""

*#(≈ 1 line)*

*# X\_pad = None*

*# YOUR CODE STARTS HERE*

X\_pad **=** np**.**pad(X, ((0,0), (pad,pad), (pad,pad),(0,0)), mode**=**'constant', constant\_values **=** (0,0))

*# YOUR CODE ENDS HERE*

**return** X\_pad

In [18]:

np**.**random**.**seed(1)

x **=** np**.**random**.**randn(4, 3, 3, 2)

x\_pad **=** zero\_pad(x, 3)

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])

zero\_pad\_test(zero\_pad)

x.shape =

(4, 3, 3, 2)

x\_pad.shape =

(4, 9, 9, 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.]

[0. 0.]

[0. 0.]]

x.shape =

(4, 3, 3, 2)

x\_pad.shape =

(4, 9, 9, 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.]

[0. 0.]

[0. 0.]]

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

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

All tests passed!

A black and white squares

Description automatically generated

**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. You 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 2 - conv\_single\_step**

Implement conv\_single\_step().

[Hint](https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.sum.html).

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

In [38]:

*# GRADED FUNCTION: conv\_single\_step*

**def** conv\_single\_step(a\_slice\_prev, W, b):

"""

Apply one filter defined by parameters W on a single slice (a\_slice\_prev) of the output activation

of the previous layer.

Arguments:

a\_slice\_prev -- slice of input data of shape (f, f, n\_C\_prev)

W -- Weight parameters contained in a window - matrix of shape (f, f, n\_C\_prev)

b -- Bias parameters contained in a window - matrix of shape (1, 1, 1)

Returns:

Z -- a scalar value, the result of convolving the sliding window (W, b) on a slice x of the input data

"""

*#(≈ 3 lines of code)*

*# Element-wise product between a\_slice\_prev and W. Do not add the bias yet.*

*# s = None*

*# Sum over all entries of the volume s.*

*# Z = None*

*# Add bias b to Z. Cast b to a float() so that Z results in a scalar value.*

*# Z = None*

*# YOUR CODE STARTS HERE*

s **=** np**.**multiply(a\_slice\_prev,W)

z **=** np**.**sum([s])

Z **=** np**.**sum([z,np**.**squeeze(b)])

*# YOUR CODE ENDS HERE*

**return** Z

In [39]:

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)

conv\_single\_step\_test(conv\_single\_step)

**assert** (type(Z) **==** np**.**float64), "You must cast the output to numpy float 64"

**assert** np**.**isclose(Z, **-**6.999089450680221), "Wrong value"

Z = -6.999089450680221

All tests passed!

**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 3 - conv\_forward**

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.

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.

1. 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 corners 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 are:

��=⌊������−�+2×���������⌋+1��=⌊������−�+2×���������⌋+1��=number of filters used in the convolution

For this exercise, don't worry about vectorization! Just implement everything with for-loops.

**Additional Hints (if you're stuck):**

* 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.

In [40]:

a\_slice\_prev**.**shape

Out[40]:

(4, 4, 3)

In [45]:

*# GRADED FUNCTION: conv\_forward*

**def** conv\_forward(A\_prev, W, b, hparameters):

"""

Implements the forward propagation for a convolution function

Arguments:

A\_prev -- output activations of the previous layer,

numpy array of shape (m, n\_H\_prev, n\_W\_prev, n\_C\_prev)

W -- Weights, numpy array of shape (f, f, n\_C\_prev, n\_C)

b -- Biases, numpy array of shape (1, 1, 1, n\_C)

hparameters -- python dictionary containing "stride" and "pad"

Returns:

Z -- conv output, numpy array of shape (m, n\_H, n\_W, n\_C)

cache -- cache of values needed for the conv\_backward() function

"""

*# Retrieve dimensions from A\_prev's shape (≈1 line)*

*# (m, n\_H\_prev, n\_W\_prev, n\_C\_prev) = A\_prev.shape*

*# Retrieve dimensions from W's shape (≈1 line)*

*# (f, f, n\_C\_prev, n\_C) = None*

*# Retrieve information from "hparameters" (≈2 lines)*

*# stride = hparameters["stride"]*

*# pad = None*

*# Compute the dimensions of the CONV output volume using the formula given above.*

*# Hint: use int() to apply the 'floor' operation. (≈2 lines)*

*# n\_H = ((n\_H\_prev-f+(2\*pad))/stride)+1*

*# n\_W = None*

*# Initialize the output volume Z with zeros. (≈1 line)*

*# Z = np.zeros((m, n\_H, n\_W, n\_C), dtype=int)*

*# Create A\_prev\_pad by padding A\_prev*

*# A\_prev\_pad = None*

*# for i in range(None): # loop over the batch of training examples*

*# a\_prev\_pad = None # Select ith training example's padded activation*

*# for h in range(None): # loop over vertical axis of the output volume*

*# Find the vertical start and end of the current "slice" (≈2 lines)*

*# vert\_start = None*

*# vert\_end = None*

*# for w in range(None): # loop over horizontal axis of the output volume*

*# Find the horizontal start and end of the current "slice" (≈2 lines)*

*# horiz\_start = None*

*# horiz\_end = None*

*# for c in range(None): # loop over channels (= #filters) of the output volume*

*# Use the corners to define the (3D) slice of a\_prev\_pad (See Hint above the cell). (≈1 line)*

*# a\_slice\_prev = None*

*# Convolve the (3D) slice with the correct filter W and bias b, to get back one output neuron. (≈3 line)*

*# weights = None*

*# biases = None*

*# Z[i, h, w, c] = None*

*# YOUR CODE STARTS HERE*

(m, n\_H\_prev, n\_W\_prev, n\_C\_prev) **=** A\_prev**.**shape

(f, f, n\_C\_prev, n\_C) **=** W**.**shape

stride **=** hparameters["stride"]

pad **=** hparameters["pad"]

n\_H **=** int((n\_H\_prev**+**(2**\***pad)**-**f)**/**stride)**+**1

n\_W **=** int((n\_W\_prev**+**(2**\***pad)**-**f)**/**stride)**+**1

Z **=** np**.**zeros((m, n\_H, n\_W, n\_C))

A\_prev\_pad **=** zero\_pad(A\_prev, pad)

**for** i **in** range(m): *# loop over the batch of training examples*

a\_prev\_pad **=** A\_prev\_pad[i] *# Select ith training example's padded activation*

**for** h **in** range(n\_H): *# loop over vertical axis of the output volume*

*# Find the vertical start and end of the current "slice" (≈2 lines)*

vert\_start **=** stride **\*** h

vert\_end **=** vert\_start **+** f

**for** w **in** range(n\_W): *# loop over horizontal axis of the output volume*

*# Find the horizontal start and end of the current "slice" (≈2 lines)*

horiz\_start **=** stride **\*** w

horiz\_end **=** horiz\_start **+** f

**for** c **in** range(n\_C): *# loop over channels (= #filters) of the output volume*

*# Use the corners to define the (3D) slice of a\_prev\_pad (See Hint above the cell). (≈1 line)*

a\_slice\_prev **=** a\_prev\_pad[vert\_start:vert\_end,horiz\_start:horiz\_end,:]

*# Convolve the (3D) slice with the correct filter W and bias b, to get back one output neuron. (≈3 line)*

weights **=** W[:, :, :, c]

biases **=** b[:, :, :, c]

Z[i, h, w, c] **=** conv\_single\_step(a\_slice\_prev, weights, biases)

*# YOUR CODE ENDS HERE*

*# Save information in "cache" for the backprop*

cache **=** (A\_prev, W, b, hparameters)

**return** Z, cache

In [46]:

np**.**random**.**seed(1)

A\_prev **=** np**.**random**.**randn(2, 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)

z\_mean **=** np**.**mean(Z)

z\_0\_2\_1 **=** Z[0, 2, 1]

cache\_0\_1\_2\_3 **=** cache\_conv[0][1][2][3]

print("Z's mean =\n", z\_mean)

print("Z[0,2,1] =\n", z\_0\_2\_1)

print("cache\_conv[0][1][2][3] =\n", cache\_0\_1\_2\_3)

conv\_forward\_test\_1(z\_mean, z\_0\_2\_1, cache\_0\_1\_2\_3)

conv\_forward\_test\_2(conv\_forward)

Z's mean =

0.5511276474566768

Z[0,2,1] =

[-2.17796037 8.07171329 -0.5772704 3.36286738 4.48113645 -2.89198428

10.99288867 3.03171932]

cache\_conv[0][1][2][3] =

[-1.1191154 1.9560789 -0.3264995 -1.34267579]

First Test: All tests passed!

Second Test: All tests passed!

Finally, a CONV layer should also contain an activation, in which case you 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, however.

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

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

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

**4.1 - Forward Pooling**

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

**Exercise 4 - pool\_forward**

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:

��=⌊������−�������⌋+1

��=⌊������−�������⌋+1

��=������

In [51]:

*# GRADED FUNCTION: pool\_forward*

**def** pool\_forward(A\_prev, hparameters, mode **=** "max"):

"""

Implements the forward pass of the pooling layer

Arguments:

A\_prev -- Input data, numpy array of shape (m, n\_H\_prev, n\_W\_prev, n\_C\_prev)

hparameters -- python dictionary containing "f" and "stride"

mode -- the pooling mode you would like to use, defined as a string ("max" or "average")

Returns:

A -- output of the pool layer, a numpy array of shape (m, n\_H, n\_W, n\_C)

cache -- cache used in the backward pass of the pooling layer, contains the input and hparameters

"""

*# Retrieve dimensions from the input shape*

(m, n\_H\_prev, n\_W\_prev, n\_C\_prev) **=** A\_prev**.**shape

*# Retrieve hyperparameters from "hparameters"*

f **=** hparameters["f"]

stride **=** hparameters["stride"]

*# Define the dimensions of the output*

n\_H **=** int(1 **+** (n\_H\_prev **-** f) **/** stride)

n\_W **=** int(1 **+** (n\_W\_prev **-** f) **/** stride)

n\_C **=** n\_C\_prev

*# Initialize output matrix A*

A **=** np**.**zeros((m, n\_H, n\_W, n\_C))

*# for i in range(None): # loop over the training examples*

*# for h in range(None): # loop on the vertical axis of the output volume*

*# Find the vertical start and end of the current "slice" (≈2 lines)*

*# vert\_start = None*

*# vert\_end = None*

*# for w in range(None): # loop on the horizontal axis of the output volume*

*# Find the vertical start and end of the current "slice" (≈2 lines)*

*# horiz\_start = None*

*# horiz\_end = None*

*# for c in range (None): # loop over the channels of the output volume*

*# Use the corners to define the current slice on the ith training example of A\_prev, channel c. (≈1 line)*

*# a\_prev\_slice = None*

*# Compute the pooling operation on the slice.*

*# Use an if statement to differentiate the modes.*

*# Use np.max and np.mean.*

*# if mode == "max":*

*# A[i, h, w, c] = None*

*# elif mode == "average":*

*# A[i, h, w, c] = None*

*# YOUR CODE STARTS HERE*

**for** i **in** range(m): *# loop over the batch of training examples*

a\_prev\_slice **=** A\_prev[i] *# Select ith training example's padded activation*

**for** h **in** range(n\_H): *# loop over vertical axis of the output volume*

*# Find the vertical start and end of the current "slice" (≈2 lines)*

vert\_start **=** stride **\*** h

vert\_end **=** vert\_start **+** f

**for** w **in** range(n\_W): *# loop over horizontal axis of the output volume*

*# Find the horizontal start and end of the current "slice" (≈2 lines)*

horiz\_start **=** stride **\*** w

horiz\_end **=** horiz\_start **+** f

**for** c **in** range(n\_C): *# loop over channels (= #filters) of the output volume*

*# Use the corners to define the (3D) slice of a\_prev\_pad (See Hint above the cell). (≈1 line)*

a\_slice\_prev **=** a\_prev\_slice[vert\_start:vert\_end,horiz\_start:horiz\_end,c]

**if** mode **==** "max":

A[i, h, w, c] **=** np**.**max(a\_slice\_prev)

**elif** mode **==** "average":

A[i, h, w, c] **=** np**.**mean(a\_slice\_prev)

**else**:

print(mode**+** "-type pooling layer NOT Defined")

*# YOUR CODE ENDS HERE*

*# Store the input and hparameters in "cache" for pool\_backward()*

cache **=** (A\_prev, hparameters)

*# Making sure your output shape is correct*

*#assert(A.shape == (m, n\_H, n\_W, n\_C))*

**return** A, cache

In [52]:

*# 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, mode **=** "max")

print("mode = max")

print("A.shape = " **+** str(A**.**shape))

print("A[1, 1] =\n", A[1, 1])

A, cache **=** pool\_forward(A\_prev, hparameters, mode **=** "average")

print("mode = average")

print("A.shape = " **+** str(A**.**shape))

print("A[1, 1] =\n", A[1, 1])

pool\_forward\_test(pool\_forward)

mode = max

A.shape = (2, 3, 3, 3)

A[1, 1] =

[[1.96710175 0.84616065 1.27375593]

[1.96710175 0.84616065 1.23616403]

[1.62765075 1.12141771 1.2245077 ]]

mode = average

A.shape = (2, 3, 3, 3)

A[1, 1] =

[[ 0.44497696 -0.00261695 -0.31040307]

[ 0.50811474 -0.23493734 -0.23961183]

[ 0.11872677 0.17255229 -0.22112197]]

All tests passed!

**Expected output**

mode = max

A.shape = (2, 3, 3, 3)

A[1, 1] =

[[1.96710175 0.84616065 1.27375593]

[1.96710175 0.84616065 1.23616403]

[1.62765075 1.12141771 1.2245077 ]]

mode = average

A.shape = (2, 3, 3, 3)

A[1, 1] =

[[ 0.44497696 -0.00261695 -0.31040307]

[ 0.50811474 -0.23493734 -0.23961183]

[ 0.11872677 0.17255229 -0.22112197]]

In [53]:

*# 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[0] =\n", A[0])

print()

A, cache **=** pool\_forward(A\_prev, hparameters, mode **=** "average")

print("mode = average")

print("A.shape = " **+** str(A**.**shape))

print("A[1] =\n", A[1])

mode = max

A.shape = (2, 2, 2, 3)

A[0] =

[[[1.74481176 0.90159072 1.65980218]

[1.74481176 1.6924546 1.65980218]]

[[1.13162939 1.51981682 2.18557541]

[1.13162939 1.6924546 2.18557541]]]

mode = average

A.shape = (2, 2, 2, 3)

A[1] =

[[[-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[0] =

[[[1.74481176 0.90159072 1.65980218]

[1.74481176 1.6924546 1.65980218]]

[[1.13162939 1.51981682 2.18557541]

[1.13162939 1.6924546 2.18557541]]]

mode = average

A.shape = (2, 2, 2, 3)

A[1] =

[[[-0.17313416 0.32377198 -0.34317572]

[ 0.02030094 0.14141479 -0.01231585]]

[[ 0.42944926 0.08446996 -0.27290905]

[ 0.15077452 0.28911175 0.00123239]]]

**What you should remember**:

* A convolution extracts features from an input image by taking the dot product between the input data and a 3D array of weights (the filter).
* The 2D output of the convolution is called the feature map
* A convolution layer is where the filter slides over the image and computes the dot product
  + This transforms the input volume into an output volume of different size
* Zero padding helps keep more information at the image borders, and is helpful for building deeper networks, because you can build a CONV layer without shrinking the height and width of the volumes
* Pooling layers gradually reduce the height and width of the input by sliding a 2D window over each specified region, then summarizing the features in that region

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

The remainder of this notebook is optional, and will not be graded. If you carry on, just remember to hit the Submit button to submit your work for grading first.

**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 were not derived in lecture, but are briefly presented below.

**5.1 - Convolutional Layer Backward Pass**

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

**5.1.1 - Computing dA:**

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

(1)��+=∑ℎ=0��∑�=0����×��ℎ�

Where �� is a filter and ��ℎ� 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, you multiply the the same filter �� 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, you 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 ��� (��� is the derivative of one filter) with respect to the loss:

(2)���+=∑ℎ=0��∑�=0��������×��ℎ�

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

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

dW[:,:,:,c] \mathrel{**+**}**=** a\_slice **\*** dZ[i, h, w, c]

**5.1.3 - Computing db:**

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

(3)��=∑ℎ∑���ℎ�

As you have previously seen in basic neural networks, db is computed by summing ��. 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 5 - conv\_backward**

Implement the conv\_backward function below. You should sum over all the training examples, filters, heights, and widths. You should then compute the derivatives using formulas 1, 2 and 3 above.

In [54]:

**def** conv\_backward(dZ, cache):

"""

Implement the backward propagation for a convolution function

Arguments:

dZ -- gradient of the cost with respect to the output of the conv layer (Z), numpy array of shape (m, n\_H, n\_W, n\_C)

cache -- cache of values needed for the conv\_backward(), output of conv\_forward()

Returns:

dA\_prev -- gradient of the cost with respect to the input of the conv layer (A\_prev),

numpy array of shape (m, n\_H\_prev, n\_W\_prev, n\_C\_prev)

dW -- gradient of the cost with respect to the weights of the conv layer (W)

numpy array of shape (f, f, n\_C\_prev, n\_C)

db -- gradient of the cost with respect to the biases of the conv layer (b)

numpy array of shape (1, 1, 1, n\_C)

"""

*# Retrieve information from "cache"*

(A\_prev, W, b, hparameters) **=** cache

*# Retrieve dimensions from A\_prev's shape*

(m, n\_H\_prev, n\_W\_prev, n\_C\_prev) **=** A\_prev**.**shape

*# Retrieve dimensions from W's shape*

(f, f, n\_C\_prev, n\_C) **=** W**.**shape

*# Retrieve information from "hparameters"*

stride **=** hparameters["stride"]

pad **=** hparameters["pad"]

*# Retrieve dimensions from dZ's shape*

(m, n\_H, n\_W, n\_C) **=** dZ**.**shape

*# Initialize dA\_prev, dW, db with the correct shapes*

dA\_prev **=** np**.**zeros(A\_prev**.**shape)

dW **=** np**.**zeros(W**.**shape)

db **=** np**.**zeros(b**.**shape) *# b.shape = [1,1,1,n\_C]*

*# Pad A\_prev and dA\_prev*

A\_prev\_pad **=** zero\_pad(A\_prev, pad)

dA\_prev\_pad **=** zero\_pad(dA\_prev, pad)

**for** i **in** range(m): *# loop over the training examples*

*# select ith training example from A\_prev\_pad and dA\_prev\_pad*

a\_prev\_pad **=** A\_prev\_pad[i]

da\_prev\_pad **=** dA\_prev\_pad[i]

**for** h **in** range(n\_H): *# loop over vertical axis of the output volume*

**for** w **in** range(n\_W): *# loop over horizontal axis of the output volume*

**for** c **in** range(n\_C): *# loop over the channels of the output volume*

*# Find the corners of the current "slice"*

vert\_start **=** stride **\*** h

vert\_end **=** vert\_start **+** f

horiz\_start **=** stride **\*** w

horiz\_end **=** horiz\_start **+** f

*# Use the corners to define the slice from a\_prev\_pad*

a\_slice **=** a\_prev\_pad[vert\_start:vert\_end,horiz\_start:horiz\_end,:]

*# Update gradients for the window and the filter's parameters using the code formulas given above*

da\_prev\_pad[vert\_start:vert\_end, horiz\_start:horiz\_end, :] **+=** W[:,:,:,c] **\*** dZ[i, h, w, c]

dW[:,:,:,c] **+=** a\_slice **\*** dZ[i, h, w, c]

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

*# Set the ith training example's dA\_prev to the unpadded da\_prev\_pad (Hint: use X[pad:-pad, pad:-pad, :])*

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

In [55]:

*# 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))

**assert** type(dA) **==** np**.**ndarray, "Output must be a np.ndarray"

**assert** type(dW) **==** np**.**ndarray, "Output must be a np.ndarray"

**assert** type(db) **==** np**.**ndarray, "Output must be a np.ndarray"

**assert** dA**.**shape **==** (10, 4, 4, 3), f"Wrong shape for dA {dA**.**shape} != (10, 4, 4, 3)"

**assert** dW**.**shape **==** (2, 2, 3, 8), f"Wrong shape for dW {dW**.**shape} != (2, 2, 3, 8)"

**assert** db**.**shape **==** (1, 1, 1, 8), f"Wrong shape for db {db**.**shape} != (1, 1, 1, 8)"

**assert** np**.**isclose(np**.**mean(dA), 1.4524377), "Wrong values for dA"

**assert** np**.**isclose(np**.**mean(dW), 1.7269914), "Wrong values for dW"

**assert** np**.**isclose(np**.**mean(db), 7.8392325), "Wrong values for db"

print("\033[92m All tests passed.")

dA\_mean = 1.4524377775388075

dW\_mean = 1.7269914583139097

db\_mean = 7.839232564616838

All tests passed.

**Expected Output**:

|  |  |
| --- | --- |
| dA\_mean | 1.45243777754 |
| dW\_mean | 1.72699145831 |
| db\_mean | 7.83923256462 |

**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 backpropagate the gradient through the pooling layer in order to compute gradients for layers that came before the pooling layer.

**5.2.1 Max Pooling - Backward Pass**

Before jumping into the backpropagation of the pooling layer, you are going to build a helper function called create\_mask\_from\_window() which does the following:

(4)�=[1342]→�=[0010]

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 is similar to this, but uses a different mask.

**Exercise 6 - create\_mask\_from\_window**

Implement create\_mask\_from\_window(). This function will be helpful for pooling backward. Hints:

* [np.max()](https://notebooks.githubusercontent.com/view/ipynb?browser=chrome&bypass_fastly=true&color_mode=auto&commit=590b5b40199f358e911da44f99f6c3e00b0c2934&device=unknown_device&docs_host=https%3A%2F%2Fdocs.github.com&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f6265726b6179616c616e2f6e657572616c2d6e6574776f726b732d616e642d646565702d6c6561726e696e672f353930623562343031393966333538653931316461343466393966366333653030623063323933342f436f6e766f6c7574696f6e5f6d6f64656c5f537465705f62795f537465705f76312e6970796e62&logged_in=false&nwo=berkayalan%2Fneural-networks-and-deep-learning&path=Convolution_model_Step_by_Step_v1.ipynb&platform=windows&repository_id=326138598&repository_type=Repository&version=121) may be helpful. It computes the maximum of an array.
* If you have a matrix X and a scalar x: A = (X == x) will return a matrix A of the same size as X such that:

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

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

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

In [56]:

**def** create\_mask\_from\_window(x):

"""

Creates a mask from an input matrix x, to identify the max entry of x.

Arguments:

x -- Array of shape (f, f)

Returns:

mask -- Array of the same shape as window, contains a True at the position corresponding to the max entry of x.

"""

*# (≈1 line)*

*# mask = None*

*# YOUR CODE STARTS HERE*

mask **=** (x **==** np**.**max(x))

*# YOUR CODE ENDS HERE*

**return** mask

np**.**random**.**seed(1)

In [57]:

np**.**random**.**seed(1)

x **=** np**.**random**.**randn(2, 3)

mask **=** create\_mask\_from\_window(x)

print('x = ', x)

print("mask = ", mask)

x **=** np**.**array([[**-**1, 2, 3],

[2, **-**3, 2],

[1, 5, **-**2]])

y **=** np**.**array([[**False**, **False**, **False**],

[**False**, **False**, **False**],

[**False**, **True**, **False**]])

mask **=** create\_mask\_from\_window(x)

**assert** type(mask) **==** np**.**ndarray, "Output must be a np.ndarray"

**assert** mask**.**shape **==** x**.**shape, "Input and output shapes must match"

**assert** np**.**allclose(mask, y), "Wrong output. The True value must be at position (2, 1)"

print("\033[92m All tests passed.")

x = [[ 1.62434536 -0.61175641 -0.52817175]

[-1.07296862 0.86540763 -2.3015387 ]]

mask = [[ True False False]

[False False False]]

All tests passed.

**Expected Output:**

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

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

**5.2.2 - Average Pooling - Backward Pass**

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

For example if we did average pooling in the forward pass using a 2x2 filter, then the mask you'll use for the backward pass will look like:(5)��=1→��=[1/41/41/41/4]

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

**Exercise 7 - distribute\_value**

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

[Hint](https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.ones.html)

In [58]:

**def** distribute\_value(dz, shape):

"""

Distributes the input value in the matrix of dimension shape

Arguments:

dz -- input scalar

shape -- the shape (n\_H, n\_W) of the output matrix for which we want to distribute the value of dz

Returns:

a -- Array of size (n\_H, n\_W) for which we distributed the value of dz

"""

*# Retrieve dimensions from shape (≈1 line)*

(n\_H, n\_W) **=** shape

*# Compute the value to distribute on the matrix (≈1 line)*

average **=** np**.**prod(shape)

*# Create a matrix where every entry is the "average" value (≈1 line)*

a **=** (dz**/**average)**\***np**.**ones(shape)

*# YOUR CODE STARTS HERE*

*# YOUR CODE ENDS HERE*

**return** a

In [59]:

a **=** distribute\_value(2, (2, 2))

print('distributed value =', a)

**assert** type(a) **==** np**.**ndarray, "Output must be a np.ndarray"

**assert** a**.**shape **==** (2, 2), f"Wrong shape {a**.**shape} != (2, 2)"

**assert** np**.**sum(a) **==** 2, "Values must sum to 2"

a **=** distribute\_value(100, (10, 10))

**assert** type(a) **==** np**.**ndarray, "Output must be a np.ndarray"

**assert** a**.**shape **==** (10, 10), f"Wrong shape {a**.**shape} != (10, 10)"

**assert** np**.**sum(a) **==** 100, "Values must sum to 100"

print("\033[92m All tests passed.")

distributed value = [[0.5 0.5]

[0.5 0.5]]

All tests passed.

**Expected Output**:

|  |  |
| --- | --- |
| distributed\_value = | [[ 0.5 0.5] [ 0.5 0.5]] |

**5.2.3 Putting it Together: Pooling Backward**

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

**Exercise 8 - pool\_backward**

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.

In [60]:

**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

"""

*# Retrieve information from cache (≈1 line)*

(A\_prev, hparameters) **=** cache

*# Retrieve hyperparameters from "hparameters" (≈2 lines)*

stride **=** hparameters["stride"]

f **=** hparameters["f"]

*# Retrieve dimensions from A\_prev's shape and dA's shape (≈2 lines)*

m, n\_H\_prev, n\_W\_prev, n\_C\_prev **=** A\_prev**.**shape

m, n\_H, n\_W, n\_C **=** dA**.**shape

*# Initialize dA\_prev with zeros (≈1 line)*

dA\_prev **=** np**.**zeros(A\_prev**.**shape)

**for** i **in** range(m): *# loop over the training examples*

*# select training example from A\_prev (≈1 line)*

a\_prev **=** A\_prev[i,:,:,:]

**for** h **in** range(n\_H): *# loop on the vertical axis*

**for** w **in** range(n\_W): *# loop on the horizontal axis*

**for** c **in** range(n\_C): *# loop over the channels (depth)*

*# Find the corners of the current "slice" (≈4 lines)*

vert\_start **=** h **\*** stride

vert\_end **=** h **\*** stride **+** f

horiz\_start **=** w **\*** stride

horiz\_end **=** w **\*** stride **+** f

*# Compute the backward propagation in both modes.*

**if** mode **==** "max":

*# Use the corners and "c" to define the current slice from a\_prev (≈1 line)*

a\_prev\_slice **=** a\_prev[ vert\_start:vert\_end, horiz\_start:horiz\_end, c ]

*# Create the mask from a\_prev\_slice (≈1 line)*

mask **=** create\_mask\_from\_window( a\_prev\_slice )

*# Set dA\_prev to be dA\_prev + (the mask multiplied by the correct entry of dA) (≈1 line)*

dA\_prev[i, vert\_start:vert\_end, horiz\_start:horiz\_end, c] **+=** mask **\*** dA[i, h, w, c]

**elif** mode **==** "average":

*# Get the value da from dA (≈2 line)*

da **=** dA[i, h, w, c]

*# Define the shape of the filter as fxf (≈1 line)*

shape **=** (f,f)

*# Distribute it to get the correct slice of dA\_prev. i.e. Add the distributed value of da. (≈1 line)*

dA\_prev[i, vert\_start: vert\_end, horiz\_start: horiz\_end, c] **+=** distribute\_value(da, shape)

*# Making sure your output shape is correct*

**assert**(dA\_prev**.**shape **==** A\_prev**.**shape)

**return** dA\_prev

In [61]:

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)

print(A**.**shape)

print(cache[0]**.**shape)

dA **=** np**.**random**.**randn(5, 4, 2, 2)

dA\_prev1 **=** pool\_backward(dA, cache, mode **=** "max")

print("mode = max")

print('mean of dA = ', np**.**mean(dA))

print('dA\_prev1[1,1] = ', dA\_prev1[1, 1])

print()

dA\_prev2 **=** pool\_backward(dA, cache, mode **=** "average")

print("mode = average")

print('mean of dA = ', np**.**mean(dA))

print('dA\_prev2[1,1] = ', dA\_prev2[1, 1])

**assert** type(dA\_prev1) **==** np**.**ndarray, "Wrong type"

**assert** dA\_prev1**.**shape **==** (5, 5, 3, 2), f"Wrong shape {dA\_prev1**.**shape} != (5, 5, 3, 2)"

**assert** np**.**allclose(dA\_prev1[1, 1], [[0, 0],

[ 5.05844394, **-**1.68282702],

[ 0, 0]]), "Wrong values for mode max"

**assert** np**.**allclose(dA\_prev2[1, 1], [[0.08485462, 0.2787552],

[1.26461098, **-**0.25749373],

[1.17975636, **-**0.53624893]]), "Wrong values for mode average"

print("\033[92m All tests passed.")

(5, 4, 2, 2)

(5, 5, 3, 2)

mode = max

mean of dA = 0.14571390272918056

dA\_prev1[1,1] = [[ 0. 0. ]

[ 5.05844394 -1.68282702]

[ 0. 0. ]]

mode = average

mean of dA = 0.14571390272918056

dA\_prev2[1,1] = [[ 0.08485462 0.2787552 ]

[ 1.26461098 -0.25749373]

[ 1.17975636 -0.53624893]]

All tests passed.

**Expected Output**:

mode = max:

|  |  |
| --- | --- |
| **mean of dA =** | 0.145713902729 |
| dA\_prev[1,1] = | [[ 0. 0. ] [ 5.05844394 -1.68282702] [ 0. 0. ]] |

mode = average

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

**Congratulations**! You've completed the assignment and its optional portion. You now understand how convolutional neural networks work, and have implemented all the building blocks of a neural network. In the next assignment you will implement a ConvNet using TensorFlow. Nicely done! See you there.