Python Basics with Numpy (optional assignment)

Welcome to your first assignment. This exercise gives you a brief introduction to Pytl
Even if you've used Python before, this will help familiarize you with the functions we
need.

Instructions:

- You will be using Python 3.
- Avoid using for-loops and while-loops, unless you are explicitly told to do so.
- After coding your function, run the cell right below it to check if your result is cor

After this assignment you will:

- Be able to use iPython Notebooks
- Be able to use numpy functions and numpy matrix/vector operations
- Understand the concept of "broadcasting"
- Be able to vectorize code

Let's get started!

Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not
the following:

1. You have not added any extra `print` statement(s) in the assignment.
2. You have not added any extra code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless speci
   instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating
   variables.

If you do any of the following, you will get something like, `Grader Error: Grader feedback not found` (or similarly unexpected) error upon submitting your assignm
Before asking for help/debugging the errors in your assignment, check for these first
is the case, and you don't remember the changes you have made, you can get a fres
of the assignment by following these instructions.

Table of Contents

About iPython Notebooks

iPython Notebooks are interactive coding environments embedded in a webpage. Yo using iPython notebooks in this class. You only need to write code between the # you here comment. After writing your code, you can run the cell by either pressing "SHIFT"+"ENTER" or by clicking on "Run Cell" (denoted by a play symbol) in the uppe the notebook.

We will often specify "(≈ X lines of code)" in the comments to tell you about how mu you need to write. It is just a rough estimate, so don't feel bad if your code is longer shorter.

In [1]:
```
### v1.2
```

Exercise 1

Set test to `"Hello World"` in the cell below to print "Hello World" and run the two c below.

In [7]:
```
# (≈ 1 line of code)
# test =
# YOUR CODE STARTS HERE
test = 'Hello World'

# YOUR CODE ENDS HERE
```

In [8]:
```
print ("test: " + test)
```

```
test: Hello World
```

Expected output: test: Hello World

What you need to remember :

- Run your cells using SHIFT+ENTER (or "Run cell")
- Write code in the designated areas using Python 3 only
- Do not modify the code outside of the designated areas

1 - Building basic functions with numpy

Numpy is the main package for scientific computing in Python. It is maintained by a community (www.numpy.org). In this exercise you will learn several key numpy funct such as `np.exp`, `np.log`, and `np.reshape`. You will need to know how to use thes functions for future assignments.

1.1 - sigmoid function, np.exp()

Before using `np.exp()`, you will use `math.exp()` to implement the sigmoid functic will then see why `np.exp()` is preferable to `math.exp()`.

Exercise 2 - basic_sigmoid

Build a function that returns the sigmoid of a real number x. Use `math.exp(x)` for t exponential function.

Reminder: $sigmoid(x) = \frac{1}{1+e^{-x}}$ is sometimes also known as the log function. It is a non-linear function used not only in Machine Learning (Logistic Regre but also in Deep Learning.

No description has been provided for this image

To refer to a function belonging to a specific package you could call it using `package_name.function()`. Run the code below to see an example with `math.exp`

In [9]:
```
import math
from public_tests import *
```

```python
# GRADED FUNCTION: basic_sigmoid

def basic_sigmoid(x):
    """
    Compute sigmoid of x.

    Arguments:
    x -- A scalar

    Return:
    s -- sigmoid(x)
    """
    # (≈ 1 line of code)
    # s =
    # YOUR CODE STARTS HERE

    s = 1/(1+math.exp(-x))
    # YOUR CODE ENDS HERE

    return s
```

In [10]:
```python
print("basic_sigmoid(1) = " + str(basic_sigmoid(1)))

basic_sigmoid_test(basic_sigmoid)
```

```
basic_sigmoid(1) = 0.7310585786300049
 All tests passed.
```

Actually, we rarely use the "math" library in deep learning because the inputs of the functions are real numbers. In deep learning we mostly use matrices and vectors. Th why numpy is more useful.

In [11]:
```python
### One reason why we use "numpy" instead of "math" in Deep Learning ###

x = [1, 2, 3] # x becomes a python list object
basic_sigmoid(x) # you will see this give an error when you run it, becau
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-f93a844e80fe> in <module>
      2
      3 x = [1, 2, 3] # x becomes a python list object
----> 4 basic_sigmoid(x)
# you will see this give an error when you run it, because x is a vector.

<ipython-input-9-9168aaafe663> in basic_sigmoid(x)
     18     # YOUR CODE STARTS HERE
     19
---> 20     s = 1/(1+math.exp(-x))
     21     # YOUR CODE ENDS HERE
     22

TypeError: bad operand type for unary -: 'list'
```

In fact, if $ x = (x_1, x_2, ..., x_n)$ is a row vector then `np.exp(x)` will apply the exponential function to every element of x. The output will thus be: `np.exp(x) = (e^{x_1}, e^{x_2}, ..., e^{x_n})`

In [12]:
```
import numpy as np

# example of np.exp
t_x = np.array([1, 2, 3])
print(np.exp(t_x)) # result is (exp(1), exp(2), exp(3))
```

```
[ 2.71828183  7.3890561  20.08553692]
```

Furthermore, if x is a vector, then a Python operation such as $s = x + 3$ or $s = \fr
{x}$ will output s as a vector of the same size as x.

In [13]:
```
# example of vector operation
t_x = np.array([1, 2, 3])
print (t_x + 3)
```

```
[4 5 6]
```

Any time you need more info on a numpy function, we encourage you to look at the documentation.

You can also create a new cell in the notebook and write `np.exp?` (for example) to g
quick access to the documentation.

Exercise 3 - sigmoid

Implement the sigmoid function using numpy.

Instructions: x could now be either a real number, a vector, or a matrix. The data structures we use in numpy to represent these shapes (vectors, matrices...) are calle numpy arrays. You don't need to know more for now. $$ \text{For } x \in \mathbb{R} \text{, } sigmoid(x) = sigmoid\begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \\ \end{pmatrix} \begin{pmatrix} \frac{1}{1+e^{-x_1}} \\ \frac{1}{1+e^{-x_2}} \\ ... \\ \frac{1}{1 x_n}} \\ \end{pmatrix}\tag{1} $$

In [14]:
```
# GRADED FUNCTION: sigmoid

def sigmoid(x):
    """
    Compute the sigmoid of x

    Arguments:
    x -- A scalar or numpy array of any size

    Return:
    s -- sigmoid(x)
    """

    # (≈ 1 line of code)
    # s = 
    # YOUR CODE STARTS HERE
    s = 1/(1+np.exp(-x))

    # YOUR CODE ENDS HERE
```

```
    return s
```

In [15]:
```
t_x = np.array([1, 2, 3])
print("sigmoid(t_x) = " + str(sigmoid(t_x)))

sigmoid_test(sigmoid)
```

```
sigmoid(t_x) = [0.73105858 0.88079708 0.95257413]
 All tests passed.
```

1.2 - Sigmoid Gradient

As you've seen in lecture, you will need to compute gradients to optimize loss functi
using backpropagation. Let's code your first gradient function.

Exercise 4 - sigmoid_derivative

Implement the function sigmoid_grad() to compute the gradient of the sigmoid funct
respect to its input x. The formula is:

$$sigmoid\_derivative(x) = \sigma'(x) = \sigma(x) (1 - \sigma(x))\tag{2}$$

You often code this function in two steps:

1. Set s to be the sigmoid of x. You might find your sigmoid(x) function useful.
2. Compute $\sigma'(x) = s(1-s)$

In [16]:
```
# GRADED FUNCTION: sigmoid_derivative

def sigmoid_derivative(x):
    """
    Compute the gradient (also called the slope or derivative) of the sig
    You can store the output of the sigmoid function into variables and t

    Arguments:
    x -- A scalar or numpy array

    Return:
    ds -- Your computed gradient.
    """

    #(≈ 2 lines of code)
    # s =
    # ds =
    # YOUR CODE STARTS HERE
    s = 1/(1 + np.exp(-x))
    ds = s * (1 - s)

    # YOUR CODE ENDS HERE

    return ds
```

In [17]:
```
t_x = np.array([1, 2, 3])
print ("sigmoid_derivative(t_x) = " + str(sigmoid_derivative(t_x)))
```

```
sigmoid_derivative_test(sigmoid_derivative)
```

```
sigmoid_derivative(t_x) = [0.19661193 0.10499359 0.04517666]
 All tests passed.
```

1.3 - Reshaping arrays

Two common numpy functions used in deep learning are np.shape and np.reshape().

- X.shape is used to get the shape (dimension) of a matrix/vector X.
- X.reshape(...) is used to reshape X into some other dimension.

For example, in computer science, an image is represented by a 3D array of shape $ height, depth = 3)$. However, when you read an image as the input of an algorithm convert it to a vector of shape $(length*height*3, 1)$. In other words, you "unroll", o reshape, the 3D array into a 1D vector.

No description has been provided for this image

Exercise 5 - image2vector

Implement `image2vector()` that takes an input of shape (length, height, 3) and ret vector of shape (length*height*3, 1). For example, if you would like to reshape an ar shape (a, b, c) into a vector of shape (a*b,c) you would do:

```
v = v.reshape((v.shape[0] * v.shape[1], v.shape[2])) # v.shape[0] = a ;
v.shape[1] = b ; v.shape[2] = c
```

- Please don't hardcode the dimensions of image as a constant. Instead look up th quantities you need with `image.shape[0]`, etc.
- You can use v = v.reshape(-1, 1). Just make sure you understand why it works.

In [28]:
```
# GRADED FUNCTION:image2vector

def image2vector(image):
    """
    Argument:
    image -- a numpy array of shape (length, height, depth)

    Returns:
    v -- a vector of shape (length*height*depth, 1)
    """

    # (≈ 1 line of code)
    # v =
    # YOUR CODE STARTS HERE

    v = np.reshape(image, (image.shape[0] * image.shape[1] * image.shape[
    # YOUR CODE ENDS HERE

    return v
```

In [29]:
```
# This is a 3 by 3 by 2 array, typically images will be (num_px_x, num_px
t_image = np.array([[[ 0.67826139,  0.29380381],
                     [ 0.90714982,  0.52835647],
                     [ 0.4215251 ,  0.45017551]],

                   [[ 0.92814219,  0.96677647],
                    [ 0.85304703,  0.52351845],
                    [ 0.19981397,  0.27417313]],

                   [[ 0.60659855,  0.00533165],
                    [ 0.10820313,  0.49978937],
                    [ 0.34144279,  0.94630077]]])

print ("image2vector(image) = " + str(image2vector(t_image)))

image2vector_test(image2vector)
```

```
image2vector(image) = [[0.67826139]
 [0.29380381]
 [0.90714982]
 [0.52835647]
 [0.4215251 ]
 [0.45017551]
 [0.92814219]
 [0.96677647]
 [0.85304703]
 [0.52351845]
 [0.19981397]
 [0.27417313]
 [0.60659855]
 [0.00533165]
 [0.10820313]
 [0.49978937]
 [0.34144279]
 [0.94630077]]
All tests passed.
```

1.4 - Normalizing rows

Another common technique we use in Machine Learning and Deep Learning is to nor our data. It often leads to a better performance because gradient descent converges after normalization. Here, by normalization we mean changing x to $ \frac{x}{\| x\|} (dividing each row vector of x by its norm).

For example, if $$x = \begin{bmatrix} 0 & 3 & 4 \\ 2 & 6 & 4 \\ \end{bmatrix}\tag{3 then $$\| x\| = \text{np.linalg.norm(x, axis=1, keepdims=True)} = \begin{bmatrix} \sqrt{56} \\ \end{bmatrix}\tag{4} $$ and $$ x_normalized = \frac{x}{\| x\|} = \begin{bmatrix} 0 & \frac{3}{5} & \frac{4}{5} \\ \frac{2}{\sqrt{56}} & \frac{6}{\sqrt{56}} & \frac{4}{\sqrt{56}} \\ \end{bmatrix}\tag{5}$$

Note that you can divide matrices of different sizes and it works fine: this is called broadcasting and you're going to learn about it in part 5.

With `keepdims=True` the result will broadcast correctly against the original x.

`axis=1` means you are going to get the norm in a row-wise manner. If you need the in a column-wise way, you would need to set `axis=0`.

numpy.linalg.norm has another parameter `ord` where we specify the type of norma to be done (in the exercise below you'll do 2-norm). To get familiar with the types of normalization you can visit numpy.linalg.norm

Exercise 6 - normalize_rows

Implement normalizeRows() to normalize the rows of a matrix. After applying this fur an input matrix x, each row of x should be a vector of unit length (meaning length 1)

In [30]:
```
# GRADED FUNCTION: normalize_rows

def normalize_rows(x):
    """
    Implement a function that normalizes each row of the matrix x (to hav

    Argument:
    x -- A numpy matrix of shape (n, m)

    Returns:
    x -- The normalized (by row) numpy matrix. You are allowed to modify
    """

    #(≈ 2 lines of code)
    # Compute x_norm as the norm 2 of x. Use np.linalg.norm(..., ord = 2,
    # x_norm =
    # Divide x by its norm.
    # x =
    # YOUR CODE STARTS HERE
    x_norm = np.linalg.norm(x, axis=1, keepdims=True)
    x /= x_norm

    # YOUR CODE ENDS HERE

    return x
```

In [31]:
```
x = np.array([[0., 3., 4.],
              [1., 6., 4.]])
print("normalizeRows(x) = " + str(normalize_rows(x)))

normalizeRows_test(normalize_rows)
```

```
normalizeRows(x) = [[0.          0.6         0.8        ]
 [0.13736056 0.82416338 0.54944226]]
 All tests passed.
```

Note: In normalize_rows(), you can try to print the shapes of x_norm and x, and ther the assessment. You'll find out that they have different shapes. This is normal given x_norm takes the norm of each row of x. So x_norm has the same number of rows bu column. So how did it work when you divided x by x_norm? This is called broadcastir we'll talk about it now!

Exercise 7 - softmax

Implement a softmax function using numpy. You can think of softmax as a normalizir function used when your algorithm needs to classify two or more classes. You will lea more about softmax in the second course of this specialization.

Instructions:

- $\text{for } x \in \mathbb{R}^{1\times n} \text{, }$

\begin{align*} softmax(x) &= softmax\left(\begin{bmatrix} x_1 && x_2 && ... && x_ \end{bmatrix}\right) \\&= \begin{bmatrix} \frac{e^{x_1}}{\sum_{j}e^{x_j}} && \frac{e^{x_2}}{\sum_{j}e^{x_j}} && ... && \frac{e^{x_n}}{\sum_{j}e^{x_j}} \end{bmatrix} \end{align*}

- $\text{for a matrix } x \in \mathbb{R}^{m \times n} \text{, x_{ij} maps to tł element in the i^{th} row and j^{th} column of x, thus we have: }$

\begin{align*} softmax(x) &= softmax\begin{bmatrix} x_{11} & x_{12} & x_{13} & & x_{1n} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2n} \\ \vdots & \vdots & \vdots \ddots & \vdots \\ x_{m1} & x_{m2} & x_{m3} & \dots & x_{mn} \end{bmatrix} \\ \ \begin{bmatrix} \frac{e^{x_{11}}}{\sum_{j}e^{x_{1j}}} & \frac{e^{x_{12}}}{\sum_{j}e^{x_{1j}}} & \frac{e^{x_{13}}}{\sum_{j}e^{x_{1j}}} & \dots & \frac{e^{x_{1n}}}{\sum_{j}e^{x_{1j}}} \\ \frac{e^{x_{21}}}{\sum_{j}e^{x_{2 \frac{e^{x_{22}}}{\sum_{j}e^{x_{2j}}} & \frac{e^{x_{23}}}{\sum_{j}e^{x_{2 \dots & \frac{e^{x_{2n}}}{\sum_{j}e^{x_{2j}}} \\ \vdots & \vdots & \vdots & \ddc \vdots \\ \frac{e^{x_{m1}}}{\sum_{j}e^{x_{mj}}} & \frac{e^{x_{m2}}}{\sum_{ e^{x_{mj}}} & \frac{e^{x_{m3}}}{\sum_{j}e^{x_{mj}}} & \dots & \frac{e^{x_ {\sum_{j}e^{x_{mj}}} \end{bmatrix} \\ \\ &= \begin{pmatrix} softmax\text{(first x)} \\ softmax\text{(second row of x)} \\ \vdots \\ softmax\text{(last row of x)} \\ \end{pmatrix} \end{align*}

Notes: Note that later in the course, you'll see "m" used to represent the "number o training examples", and each training example is in its own column of the matrix. Als feature will be in its own row (each row has data for the same feature).
Softmax should be performed for all features of each training example, so softmax w performed on the columns (once we switch to that representation later in this course

However, in this coding practice, we're just focusing on getting familiar with Python, we're using the common math notation $m \times n$
where m is the number of rows and n is the number of columns.

In [32]:
```
# GRADED FUNCTION: softmax

def softmax(x):
    """Calculates the softmax for each row of the input x.

    Your code should work for a row vector and also for matrices of shape
```

```
    Argument:
    x -- A numpy matrix of shape (m,n)

    Returns:
    s -- A numpy matrix equal to the softmax of x, of shape (m,n)
    """

    #(≈ 3 lines of code)
    # Apply exp() element-wise to x. Use np.exp(...).
    # x_exp = ...

    # Create a vector x_sum that sums each row of x_exp. Use np.sum(...,
    # x_sum = ...

    # Compute softmax(x) by dividing x_exp by x_sum. It should automatica
    # s = ...

    # YOUR CODE STARTS HERE
    x_exp = np.exp(x)
    x_sum = np.sum(x_exp, axis=1, keepdims=True)
    s = x_exp / x_sum

    # YOUR CODE ENDS HERE

    return s
```

In [33]:
```
t_x = np.array([[9, 2, 5, 0, 0],
                [7, 5, 0, 0 ,0]])
print("softmax(x) = " + str(softmax(t_x)))

softmax_test(softmax)
```

```
softmax(x) = [[9.80897665e-01 8.94462891e-04 1.79657674e-02 1.21052389e-04
  1.21052389e-04]
 [8.78679856e-01 1.18916387e-01 8.01252314e-04 8.01252314e-04
  8.01252314e-04]]
All tests passed.
```

Notes

- If you print the shapes of x_exp, x_sum and s above and rerun the assessment c will see that x_sum is of shape (2,1) while x_exp and s are of shape (2,5). **x_exp/** works due to python broadcasting.

Congratulations! You now have a pretty good understanding of python numpy and ha implemented a few useful functions that you will be using in deep learning.

What you need to remember:

- np.exp(x) works for any np.array x and applies the exponential function to every coordinate
- the sigmoid function and its gradient
- image2vector is commonly used in deep learning
- np.reshape is widely used. In the future, you'll see that keeping your matrix/vect dimensions straight will go toward eliminating a lot of bugs.
- numpy has efficient built-in functions

- broadcasting is extremely useful

2 - Vectorization

In deep learning, you deal with very large datasets. Hence, a non-computationally-op function can become a huge bottleneck in your algorithm and can result in a model t takes ages to run. To make sure that your code is computationally efficient, you will u vectorization. For example, try to tell the difference between the following implemer of the dot/outer/elementwise product.

In [34]:
```
import time

x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### CLASSIC DOT PRODUCT OF VECTORS IMPLEMENTATION ###
tic = time.process_time()
dot = 0

for i in range(len(x1)):
    dot += x1[i] * x2[i]
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000 *

### CLASSIC OUTER PRODUCT IMPLEMENTATION ###
tic = time.process_time()
outer = np.zeros((len(x1), len(x2))) # we create a len(x1)*len(x2) matrix

for i in range(len(x1)):
    for j in range(len(x2)):
        outer[i,j] = x1[i] * x2[j]
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " + str(100

### CLASSIC ELEMENTWISE IMPLEMENTATION ###
tic = time.process_time()
mul = np.zeros(len(x1))

for i in range(len(x1)):
    mul[i] = x1[i] * x2[i]
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n ----- Computation

### CLASSIC GENERAL DOT PRODUCT IMPLEMENTATION ###
W = np.random.rand(3,len(x1)) # Random 3*len(x1) numpy array
tic = time.process_time()
gdot = np.zeros(W.shape[0])

for i in range(W.shape[0]):
    for j in range(len(x1)):
        gdot[i] += W[i,j] * x1[j]
toc = time.process_time()
print ("gdot = " + str(gdot) + "\n ----- Computation time = " + str(1000
```

```
dot = 278
 ----- Computation time = 0.09455100000001093ms
outer = [[81. 18. 18. 81.  0. 81. 18. 45.  0.  0. 81. 18. 45.  0.  0.]
 [18.  4.  4. 18.  0. 18.  4. 10.  0.  0. 18.  4. 10.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  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.]
 [63. 14. 14. 63.  0. 63. 14. 35.  0.  0. 63. 14. 35.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  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.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [81. 18. 18. 81.  0. 81. 18. 45.  0.  0. 81. 18. 45.  0.  0.]
 [18.  4.  4. 18.  0. 18.  4. 10.  0.  0. 18.  4. 10.  0.  0.]
 [45. 10. 10. 45.  0. 45. 10. 25.  0.  0. 45. 10. 25.  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.]]
 ----- Computation time = 0.2098999999999851ms
elementwise multiplication = [81.  4. 10.  0.  0. 63. 10.  0.  0.  0. 81.  4.
0.  0.]
 ----- Computation time = 0.2425619999999462ms
gdot = [23.39697336 25.458709   23.10153112]
 ----- Computation time = 0.2602899999994085ms
```

In [35]:

```python
x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0]
x2 = [9, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0]

### VECTORIZED DOT PRODUCT OF VECTORS ###
tic = time.process_time()
dot = np.dot(x1,x2)
toc = time.process_time()
print ("dot = " + str(dot) + "\n ----- Computation time = " + str(1000 *

### VECTORIZED OUTER PRODUCT ###
tic = time.process_time()
outer = np.outer(x1,x2)
toc = time.process_time()
print ("outer = " + str(outer) + "\n ----- Computation time = " + str(100

### VECTORIZED ELEMENTWISE MULTIPLICATION ###
tic = time.process_time()
mul = np.multiply(x1,x2)
toc = time.process_time()
print ("elementwise multiplication = " + str(mul) + "\n ----- Computation

### VECTORIZED GENERAL DOT PRODUCT ###
tic = time.process_time()
dot = np.dot(W,x1)
toc = time.process_time()
print ("gdot = " + str(dot) + "\n ----- Computation time = " + str(1000 *
```

```
dot = 278
 ----- Computation time = 0.39745700000004547ms
outer = [[81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  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]
 [63 14 14 63  0 63 14 35  0  0 63 14 35  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  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  0  0  0  0  0  0  0  0  0  0  0]
 [81 18 18 81  0 81 18 45  0  0 81 18 45  0  0]
 [18  4  4 18  0 18  4 10  0  0 18  4 10  0  0]
 [45 10 10 45  0 45 10 25  0  0 45 10 25  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]]
 ----- Computation time = 0.23406299999995106ms
elementwise multiplication = [81  4 10  0  0 63 10  0  0  0 81  4 25  0  0]
 ----- Computation time = 0.07430399999999695ms
gdot = [23.39697336 25.458709   23.10153112]
 ----- Computation time = 1.2471759999999055ms
```

As you may have noticed, the vectorized implementation is much cleaner and more efficient. For bigger vectors/matrices, the differences in running time become even b

Note that `np.dot()` performs a matrix-matrix or matrix-vector multiplication. This different from `np.multiply()` and the `*` operator (which is equivalent to `.*` in M Octave), which performs an element-wise multiplication.

2.1 Implement the L1 and L2 loss functions

Exercise 8 - L1

Implement the numpy vectorized version of the L1 loss. You may find the function ab (absolute value of x) useful.

Reminder:

- The loss is used to evaluate the performance of your model. The bigger your los more different your predictions ($ \hat{y} $) are from the true values (y). In d learning, you use optimization algorithms like Gradient Descent to train your mo to minimize the cost.
- L1 loss is defined as: $$\begin{align*} & L_1(\hat{y}, y) = \sum_{i=0}^{m-1}|y - \hat{y}^{(i)}| \end{align*}\tag{6}$$

In [62]:
```
# GRADED FUNCTION: L1

def L1(yhat, y):
    """
    Arguments:
    yhat -- vector of size m (predicted labels)
    y -- vector of size m (true labels)
```

```
    Returns:
    loss -- the value of the L1 loss function defined above
    """

    #(≈ 1 line of code)
    # loss =
    # YOUR CODE STARTS HERE
    loss = np.sum(np.abs(yhat-y))
    # YOUR CODE ENDS HERE

    return loss
```

In [63]:
```
yhat = np.array([.9, 0.2, 0.1, .4, .9])
y = np.array([1, 0, 0, 1, 1])
print("L1 = " + str(L1(yhat, y)))

L1_test(L1)
```

```
L1 = 1.1
 All tests passed.
```

Exercise 9 - L2

Implement the numpy vectorized version of the L2 loss. There are several way of implementing the L2 loss but you may find the function np.dot() useful. As a reminde = [x_1, x_2, ..., x_n]$, then `np.dot(x,x)` = $\sum_{j=1}^n x_j^{2}$.

- L2 loss is defined as $$\begin{align*} & L_2(\hat{y},y) = \sum_{i=0}^{m-1}(y \hat{y}^{(i)})^2 \end{align*}\tag{7}$$

In [64]:
```
# GRADED FUNCTION: L2

def L2(yhat, y):
    """
    Arguments:
    yhat -- vector of size m (predicted labels)
    y -- vector of size m (true labels)

    Returns:
    loss -- the value of the L2 loss function defined above
    """

    #(≈ 1 line of code)
    # loss = ...
    # YOUR CODE STARTS HERE
    loss = np.sum((y - yhat) ** 2)

    # YOUR CODE ENDS HERE

    return loss
```

In [65]:
```
yhat = np.array([.9, 0.2, 0.1, .4, .9])
y = np.array([1, 0, 0, 1, 1])

print("L2 = " + str(L2(yhat, y)))
```

```
L2_test(L2)
```

```
L2 = 0.43
 All tests passed.
```

Congratulations on completing this assignment. We hope that this little warm-up exe
helps you in the future assignments, which will be more exciting and interesting!

What to remember:

- Vectorization is very important in deep learning. It provides computational efficie
  and clarity.
- You have reviewed the L1 and L2 loss.
- You are familiar with many numpy functions such as np.sum, np.dot, np.multiply
  np.maximum, etc...