**REPORT :** PYTHON BASICS WITH NUMPY

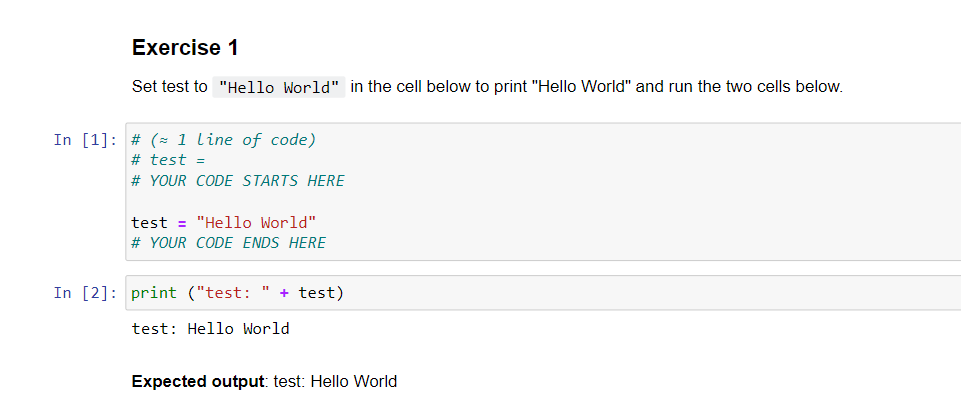
In this assignment, I will be provided an introduction to Python and Numpy, focusing on several fundamental concepts required for deep learning, such as vectorization, broadcasting, and efficient computation using Numpy functions.

With the following table of contents :

* About iPython Notebooks
* Exercise 1
* 1 - Building basic functions with numpy
  + - sigmoid function, np.exp()
* Exercise 2 - basic\_sigmoid
* Exercise 3 - sigmoid
  + - Sigmoid Gradient
* Exercise 4 - sigmoid\_derivative
  + - Reshaping arrays
* Exercise 5 - image2vector
  + - Normalizing rows
* Exercise 6 - normalize\_rows
* Exercise 7 - softmax
* - Vectorization
  + Implement the L1 and L2 loss functions
* Exercise 8 - L1
* Exercise 9 - L2

**Exercise 1 :**

The first task will be to get acquainted with ipython notebook by set a variable to "Hello World" and print it.

**1 - Building basic functions with numpy**

Numpy is the main package for scientific computing in Python. It is maintained by a large community ([www.numpy.org](http://www.numpy.org/)). In this exercise I will learn several key numpy functions such as np.exp, np.log, and np.reshape.

### - sigmoid function, np.exp()

The sigmoid function is defined as :



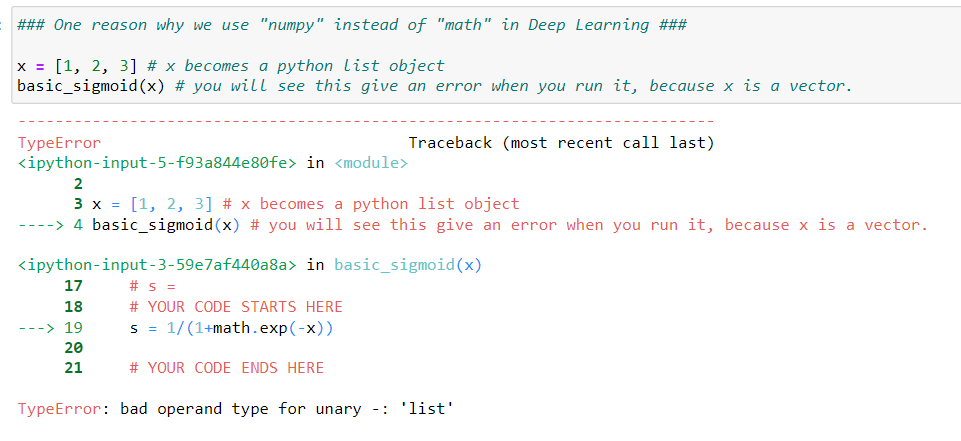
Before using **np.exp()**, we will first implement the sigmoid function using **math.exp()** and then compare it to **np.exp()** to understand why **np.exp()** is generally preferred.

### **Exercise 2 - basic\_sigmoid**

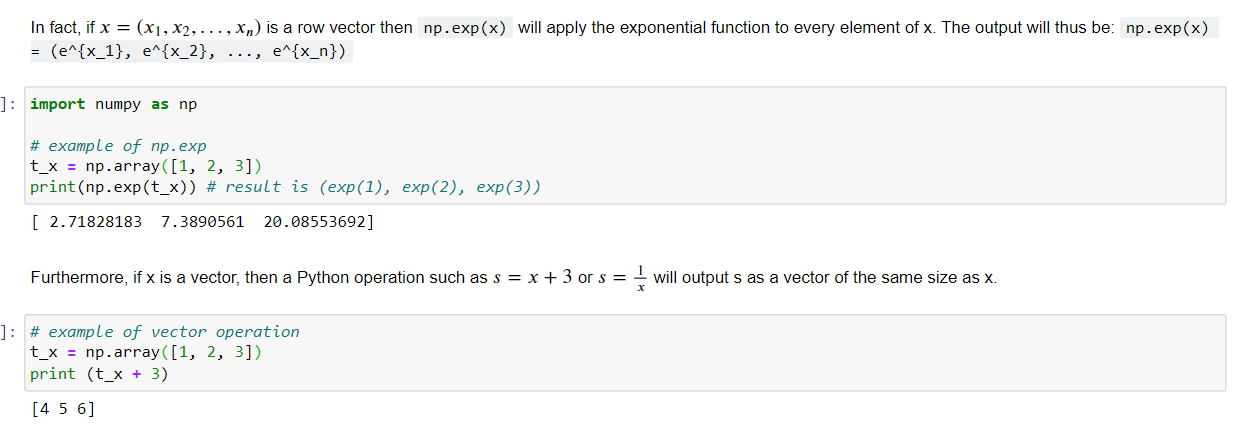
Implemented the sigmoid function using math.exp().



While the math library is suitable for scalar computations, it falls short when dealing with vector or matrix operations, which are common in deep learning. For instance, passing a list or numpy array to basic\_sigmoid will result in an error:

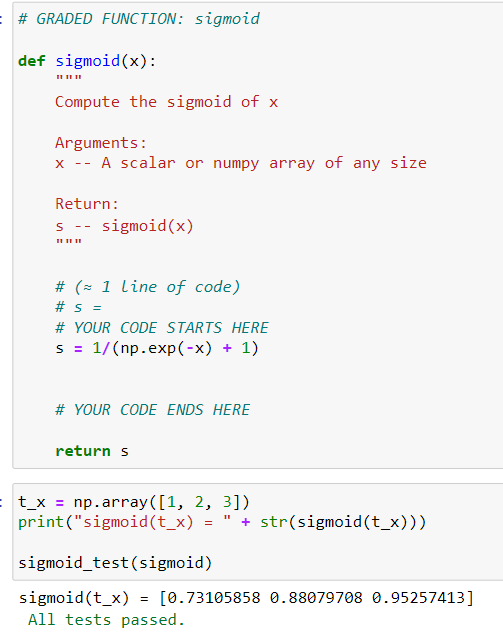


This error occurs because math.exp() cannot handle vectorized operations. Instead, we should use the numpy library, which can efficiently perform element-wise operations on arrays.



### **Exercise 3 - sigmoid**

Implement the sigmoid function using numpy.



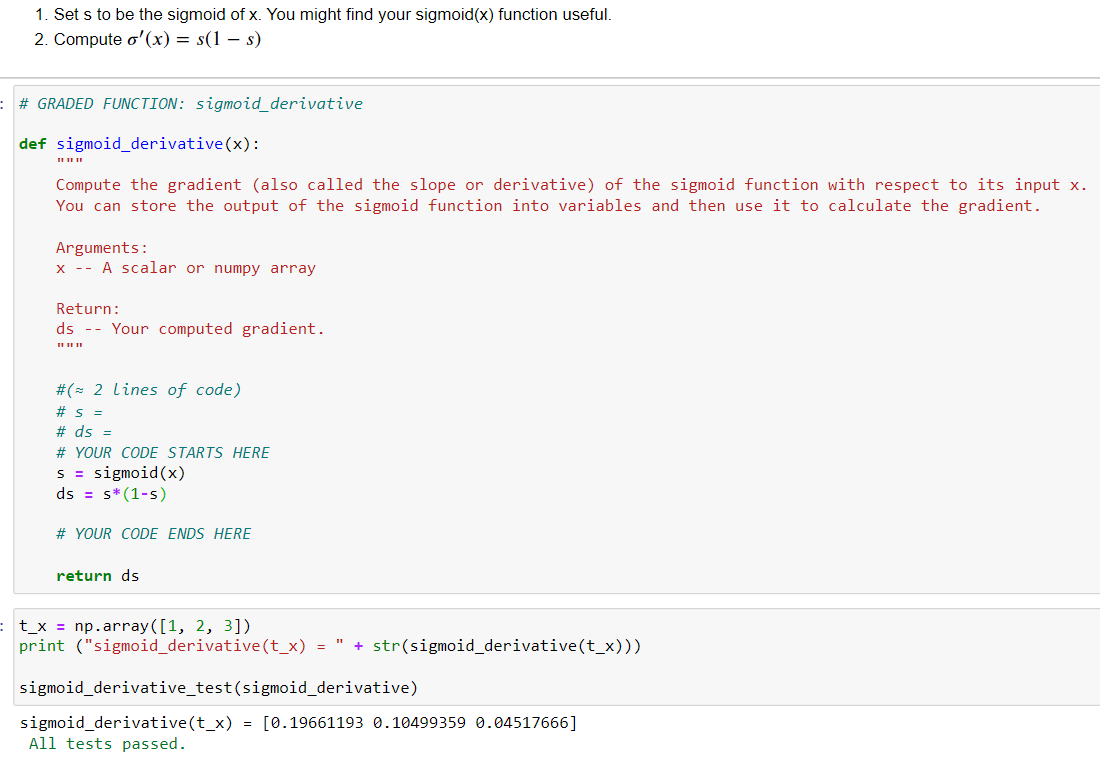
### **1.2 - Sigmoid Gradient**

In neural networks and deep learning, computing gradients is a fundamental task necessary.

**Exercise 4 - sigmoid\_derivative**

Implement the function sigmoid\_grad() to compute the gradient of the sigmoid function with respect to its input x. The formula is:





### **1.3 - Reshaping arrays**

Two essential functions in the numpy library for this purpose 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 (𝑙𝑒𝑛𝑔𝑡ℎ,ℎ𝑒𝑖𝑔ℎ𝑡,𝑑𝑒𝑝𝑡ℎ=3)(𝑙𝑒𝑛𝑔𝑡ℎ,ℎ𝑒𝑖𝑔ℎ𝑡,𝑑𝑒𝑝𝑡ℎ=3). However, when you read an image as the input of an algorithm you convert it to a vector of shape (𝑙𝑒𝑛𝑔𝑡ℎ∗ℎ𝑒𝑖𝑔ℎ𝑡∗3,1)(𝑙𝑒𝑛𝑔𝑡ℎ∗ℎ𝑒𝑖𝑔ℎ𝑡∗3,1). In other words, you "unroll", or reshape, the 3D array into a 1D vector.

### **Exercise 5 - image2vector**

This exercise focuses on implementing a function, image2vector, which reshapes a 3D image array into a 1D vector.

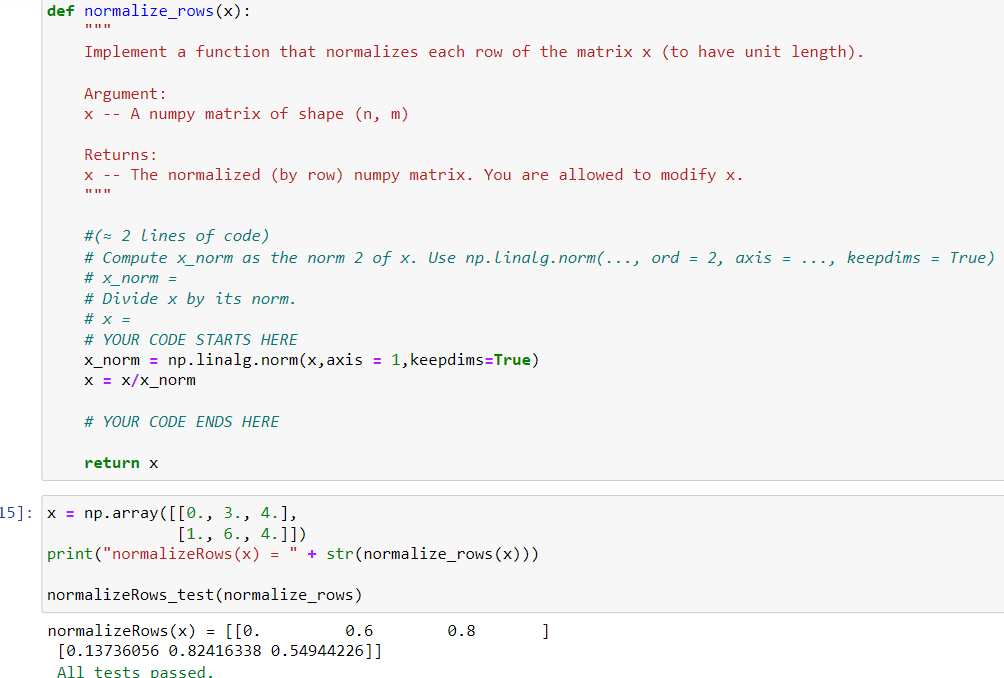


### **1.4 - Normalizing rows**

Normalizing data often leads to improved performance, especially in gradient-based optimization algorithms like gradient descent.

**Exercise 6 - normalize\_rows**

In this exercise, we implement a function normalize\_rows() that normalizes each row of a matrix to have unit length, i.e., a length of 1.

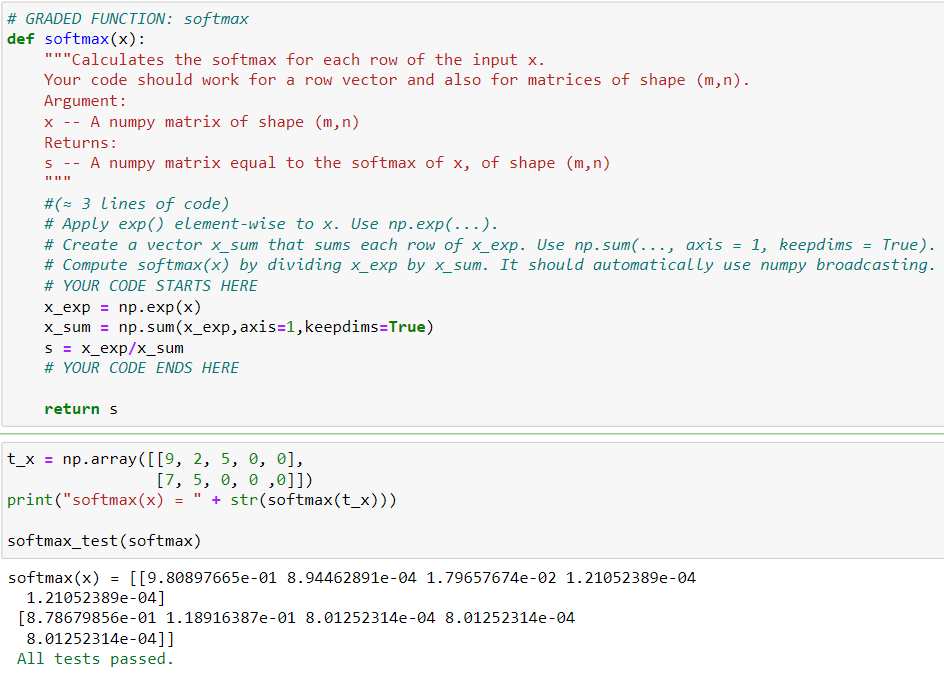


np.linalg.norm(): This function computes the norm of a matrix or vector. We use it to compute the norm of each row of the input matrix.

### **Exercise 7 - softmax**

Softmax is a mathematical function that converts a vector of numbers into a probability distribution. It's commonly used in machine learning algorithms for multi-class classification problems. In this exercise, we implement a softmax function using numpy.

Softmax is applied to each row of the input matrix x, transforming the raw scores into probabilities.



Calculating Exponentials:

In the softmax function, we first apply the exponential function element-wise to the input matrix x. This is done to transform the raw scores in x into positive values, preparing them for the softmax transformation. Each element of x is transformed into its exponential counterpart using the np.exp() function, which computes the exponential of each element in the input array. The resulting array, denoted as x\_exp, contains the exponential values corresponding to each element of the original matrix x.

Summing Exponentials:

After obtaining the matrix x\_exp containing the exponential values, we proceed to calculate the sum of each row of x\_exp. This step is crucial for computing the softmax probabilities as it represents the denominator of the softmax function. By summing across rows (axis=1), we obtain a column vector where each element corresponds to the sum of exponentials for the respective row in the original matrix x\_exp. Setting keepdims=True ensures that the resulting vector maintains the same dimensions as the input matrix x\_exp

Computing Softmax Probabilities:

Finally, the softmax probabilities are computed by dividing each element of the matrix x\_exp by the corresponding sum stored in x\_sum. This step normalizes the exponential values, ensuring that the resulting probabilities sum up to 1 for each row. Numpy's broadcasting feature enables efficient element-wise division, automatically aligning dimensions between x\_exp and x\_sum to perform the operation across the entire matrix.

By following these steps, the softmax function transforms the input matrix x into a matrix of softmax probabilities s, where each row represents the probability distribution over different classes.

**What you need to remember:**

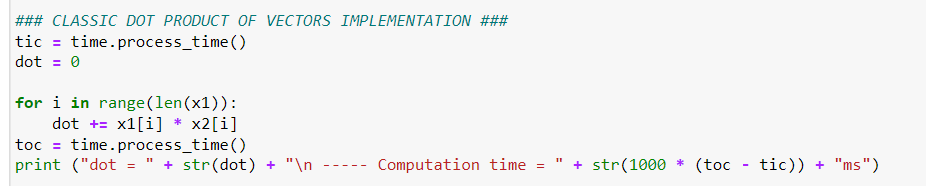
* np.exp(x) works for any np.array x and applies the exponential function to every coordinate
* the sigmoid function and its gradient
* image2vector is commonly used in deep learning
* np.reshape is widely used. In the future, you'll see that keeping your matrix/vector dimensions straight will go toward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

**2 – Vectorization**

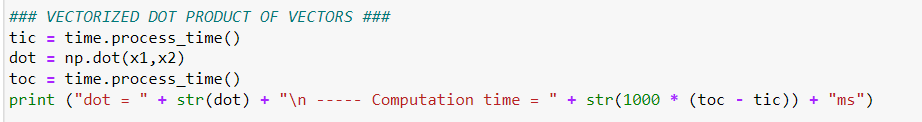
The provided code demonstrates the difference between classic (non-vectorized) implementations and vectorized implementations for various operations such as dot product, outer product, element-wise multiplication, and general dot product.

Dot Product of Vectors:

* Classic implementation: Using a loop to iterate over each element of the vectors and compute the dot product.

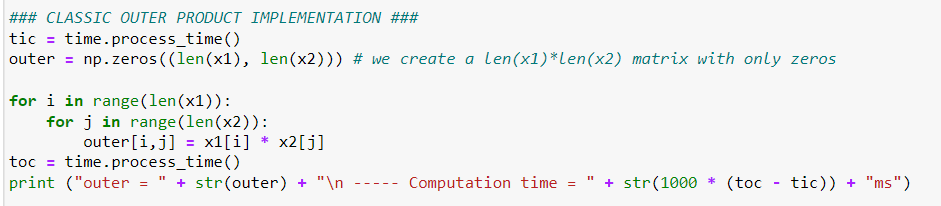


* Vectorized implementation: Utilizing NumPy's np.dot() function to directly compute the dot product of the vectors.

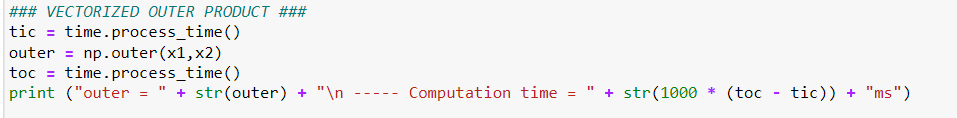


Outer Product:

* Classic implementation: Using nested loops to iterate over each element of the vectors and compute the outer product.

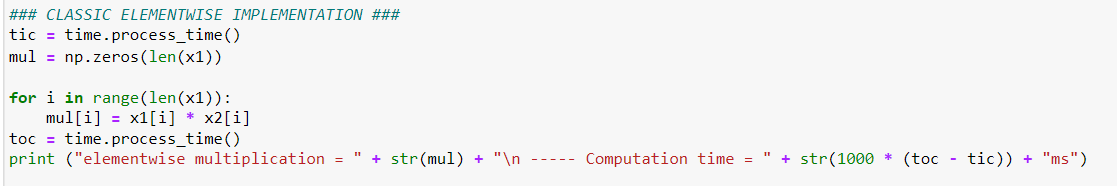


* Vectorized implementation: Utilizing NumPy's np.outer() function to directly compute the outer product.

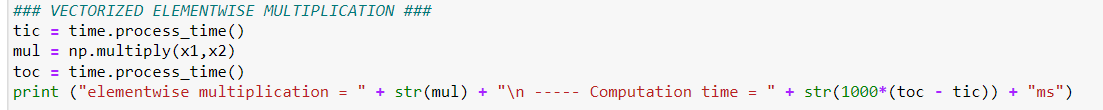


Element-wise Multiplication:

* Classic implementation: Using a loop to iterate over each element of the vectors and perform element-wise multiplication.

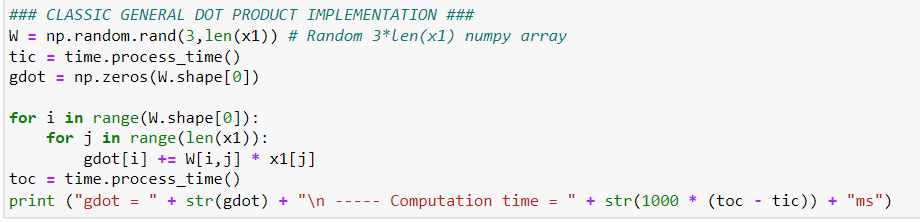


* Vectorized implementation: Utilizing NumPy's element-wise multiplication operation (np.multiply() or \* operator) to perform element-wise multiplication directly.

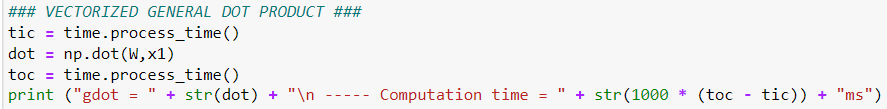


General Dot Product:

* Classic implementation: Using nested loops to iterate over each element of the matrix and compute the dot product with a vector.



* Vectorized implementation: Utilizing NumPy's np.dot() function to perform matrix-vector multiplication directly.



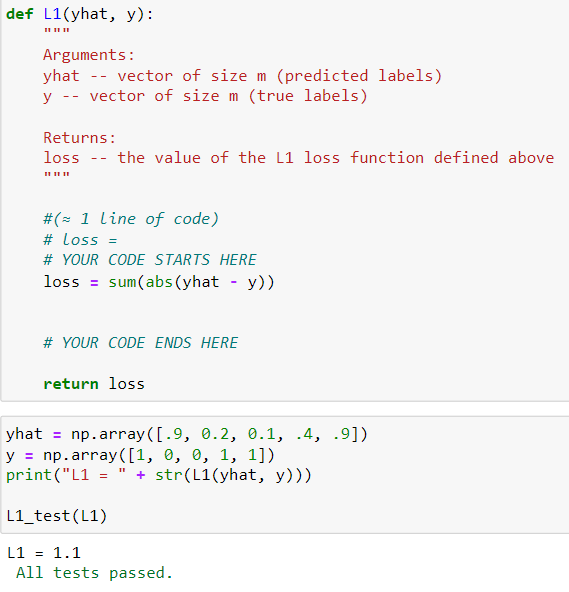
The vectorized implementations are more efficient and concise compared to their classic counterparts, especially for large datasets. Vectorization allows operations to be performed in parallel, leveraging optimized libraries and hardware acceleration, leading to significant improvements in computation time.

In summary, when working with large datasets in deep learning, leveraging vectorized operations provided by libraries like NumPy is essential for improving computational efficiency and reducing execution time.

**2.1 Implement the L1 and L2 loss functions**

**Exercise 8 - L1**

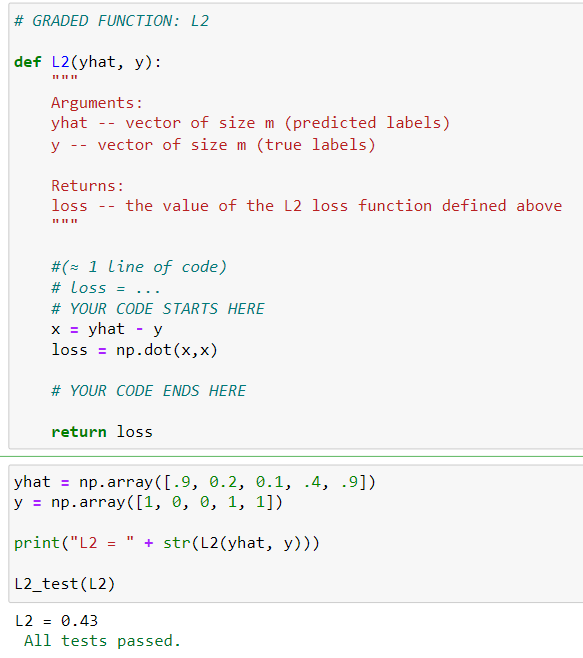
The L1 loss function computes the absolute differences between predicted labels **(yhat)** and true labels **(y)** and then sums up these differences.



This function calculates the L1 loss using NumPy's vectorized operations, which makes it efficient for large datasets. The np.abs() function is used to compute the absolute differences element-wise between yhat and y, and np.sum() sums up these differences to obtain the final loss value.

**Exercise 9 - L2**

The L2 loss function computes the squared differences between predicted labels (yhat) and true labels (y) and then sums up these squared differences.



This function calculates the L2 loss using NumPy's vectorized operations. It first computes the element-wise differences between yhat and y, then squares these differences, and finally sums up the squared differences to obtain the final loss value.

**What to remember:**

* Vectorization is very important in deep learning. It provides computational efficiency 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...