Arul Kumar ARK

225229103

Lab1 : Python Functions and Numpy

Part-I: Write a method for sigmoid function

Write a function, mysigmoid(x), that takes the real number x and returns the sigmoid value using math.exp().

```
In [1]: import numpy as np
```

```
In [2]: import math
```

```
In [3]: def mysigmoid(n):
            print(math.exp(n))
```

Call mysigmoid() with x=4 and print the sigmoid value of 4.

```
In [4]: x=4
```

```
In [5]: mysigmoid(x)

        54.598150033144236
```

Now, find the sigmoid values for x=[1, 2, 3]. Observe the results.

```
In [6]: x=[1, 2, 3]
```

```
In [7]: for x in x:
            mysigmoid(x)

        2.718281828459045
        7.38905609893065
        20.085536923187668
```

Rewrite mysigmoid() using np.exp() function.

```
In [8]: def mysigmoid(n):
            print(np.exp(n))
```

Now call your function with x=[1, 2, 3] and observe the results

```
In [9]: x=[1,2,3]
```

```
In [10]: for x in x:
             mysigmoid(x)

         2.718281828459045
         7.38905609893065
         20.085536923187668
```

Understand the difference between scalar and vector.

```
In [ ]:
```

Part-II: Gradient or derivative of sigmoid function

We compute gradients to optimize loss functions using backpropagation. s_derivative = s * (1 – s), where s = sigmoid(X) Write a function, sig_derivative(s) that returns the gradient of s. You should call your earlier function, mysigmoid() to compute the sigmoid value.

```
In [11]: def sig_derivative(s):
             return s*(1.0 - s)
```

In [12]:
```
x = 4
```

In [13]:
```
mysigmoid(sig_derivative(x))
```

```
6.14421235332821e-06
```

Part-III: Write a method image_to_vector()

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

Image Representation: an image is represented by a 3D array of shape (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) . In other words, you "unroll", or reshape, the 3D array into a 1D vector.

1). Now, implement image_to_vector() that takes an input of shape (length, height, 3) and returns a vector of shape (length*height*3, 1). For example, if you would like to reshape an array v of 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

In [14]:
```
def image_to_vector():
    length, height =  int(input("Img len : ")),int(input("Img ht : "))
    return (length*height*3,1)
```

In [16]:
```
image_to_vector()
```

```
Img len : 3
Img ht : 4
```

Out[16]:
```
(36, 1)
```

2). Test with a 3x3x2 array of values of RGB color values.

In [17]:
```
def image_to_vector():
    length, height, depth =  int(input("Img len : ")),int(input("Img ht : ")),int(input("Img dth : "))
    return (length*height*depth,1)
```

In [18]:
```
image_to_vector()
```

```
Img len : 3
Img ht : 3
Img dth : 2
```

Out[18]:
```
(18, 1)
```

Part-IV: Write a method normalizeRows()

In [25]:
```
def normalizeRows(x):
    x_n =np.linalg.norm(x)
    x = x / x_n
    return x
```

In [26]:
```
x =np.array([[0,3,4],[1,6,4]])
normalizeRows(x)
```

Out[26]:
```
array([[0.        , 0.33968311, 0.45291081],
       [0.1132277 , 0.67936622, 0.45291081]])
```

Part-V: Multiplication and Vectorization Operations

In deep learning, you deal with very large datasets. So multiplication takes long execution time than vectorization.

1). For the following two vectors, find the multiplication value and the dot product. First compute multiplication and dot product manually. Then, you can use np.multiply() and np.dot() functions.

a). x1 = [9, 2, 5] x2 = [7, 2, 2]

In [27]:
```
x1 = [9, 2, 5]
x2 = [7, 2, 2]
np.multiply(x1,x2)
```

Out[27]:
```
array([63,  4, 10])
```

```
In [28]: np.dot(x1,x2)

Out[28]: 77
```

b). x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0, 4, 5, 7] x2 = [7, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0, 8, 5, 3]

```
In [31]: x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0, 4, 5, 7]
         x2 = [7, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0, 8, 5, 3]

In [32]: np.multiply(x1,x2)

Out[32]: array([63,  4, 10,  0,  0, 63, 10,  0,  0,  0, 81,  4, 25,  0,  0, 32, 25,
                21])

In [33]: np.dot(x1,x2)

Out[33]: 338
```

2) Create two random vectors of N elements, perform multiplication and vectorization operations and print the respective running times. Is running time of vectorization is less?.

```
In [42]: import time

In [46]: x1 = [9, 2, 5, 0, 0, 7, 5, 0, 0, 0, 9, 2, 5, 0, 0, 4, 5, 7]
         x2 = [7, 2, 2, 9, 0, 9, 2, 5, 0, 0, 9, 2, 5, 0, 0, 8, 5, 3]

         tic = time.process_time()
         doc = np.dot(x1,x2)
         toc = time.process_time()

         print(str((toc-tic)))

         0.0

In [47]: tic = time.process_time()
         doc = np.multiply(x1,x2)
         toc = time.process_time()
         print(str((toc-tic)))

         0.0
```

Part-VI: Implement L1 and L2 loss functions

1). Write a method loss_l1(y, ypred) that takes the actual value y and predicted value ypred and returns l1 loss value. Test your function with the following vectors. y = np.array([1, 0, 0, 1, 1]) ypred = np.array([.9, 0.2, 0.1, .4, .9]) // Your output will be 1.1

```
In [34]: def loss_l1(y, ypred) :
             loss = np.sum(abs(y-ypred))
             return loss

In [35]: y = np.array([1, 0, 0, 1, 1])
         ypred = np.array([.9, 0.2, 0.1, .4, .9])

In [36]: loss_l1(y, ypred)

Out[36]: 1.1
```

2). Write a method loss_l2(y, ypred) that takes the actual value y and predicted value ypred and returns l2 loss value. Test your function with the following vectors. y = np.array([1, 0, 0, 1, 1]) ypred = np.array([.9, 0.2, 0.1, .4, .9]) // Your output will be 0.43

```
In [38]: def loss_l2(y, ypred):
             loss = np.dot(y - ypred,y - ypred)
             return loss

In [39]: y = np.array([1, 0, 0, 1, 1])
         ypred = np.array([.9, 0.2, 0.1, .4, .9])

In [40]: loss_l2(y, ypred)

Out[40]: 0.43

In [ ]:
```