# Outline



# NumPy Basics

## Building basic functions with numpy

### Sigmoid function, `np.exp()`

**Exercise 2 - basic_sigmoid**

Goal: Implement the sigmoid function using `math.exp()`

Sigmoid function: $σ(x) = \frac{1}{1+e^{(-x)}}$ 1 Implementation with `math.exp()`: suitable for real number inputs, example implementation has been provided.

For example, testing with an input of `1`, the expected result is approximately `0.731`

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

    Arguments:
    x -- A scalar

    Return:
    s -- sigmoid(x)
    """
    s = 1 / (1 + math.exp(-x))

    return s
```

**Exercise 3 - sigmoid**

Goal: Implement the sigmoid function using `numpy.exp()` to apply to real numbers, vectors and matrices.

Suitable for both real numbers, vectors and matrices

Check with vector `[1 2 3]`, the result is the array of corresponding sigmoid values. => “numpy” is flexible and efficient for numerical operations, especially networks and matrices.

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

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

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

    s = 1 / (1 + np.exp(-x))

    return s
```

## Sigmoid Gradient

**Exercise 4 - Sigmoid_deravative**

Optimizing the loss function using backpropagation requires calculating the gradient of the functions. The sigmoid function is one of the functions and its gradient calculation is important in adjusting the model's weights.

Objective: Implement the gradient calculation function of the sigmoid function using `numpy`

Gradient of sigmoid $σ'(x) =σ(x)(1-σ(x))$

Check with vector `[1 2 3]`, the result is the array of corresponding gradient values.

```
def sigmoid_derivative(x):
    """
    Compute the gradient (also called the slope or derivative) of the
sigmoid function with respect to its input x.
    You can store the output of the sigmoid function into variables and
then use it to calculate the gradient.

    Arguments:
    x -- A scalar or numpy array

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

```
    s = 1 / (1 + np.exp(-x))
    ds = s * (1 - s)

    return ds
```

## Reshaping arrays

**Exercise 5 - `image2vector()`**

`image2vector()` converts a numpy3D array into a 1D vector

```
image.reshape((image.shape[0] * image.shape[1] * image.shape[2], 1))
```

converts the 3D array into a 1D vector of size `(length*height*depth,1)`

Using `reshape(-1, 1)` can help simplify the code, where `-1` automatically calculates the size needed to keep the same number of elements.

Example: Given a 3D array t_image of size `(3,3,2)`, the conversion will return a vector of size `(18,1)`

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

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

    length, height, depth = image.shape
    v = np.reshape(image, (length*height*depth, 1))

    return v
```

## Normalizing rows

**Exercise 6 - normalize_rows**

Normalization leads to better performance because gradient descent converges faster after normalization. Normalize each row vector of matrix x into a unit vector (with length equal to 1).

Use `np.linalg.norm` to calculate the quadratic norm of each row.

$\frac{x}{x_{norm}}$ divides each element in x's row by that row's corresponding norm. This turns each row into a unit vector.

```
def normalize_rows(x):
    """
    Implement a function that normalizes each row of the matrix x (to have
unit length).

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

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

    x_norm = np.linalg.norm(x, ord=2, axis=1, keepdims=True)
    x = x / x_norm

    return x
```

**Exercise 7 - softmax**

The softmax function converts a vector of values into a probability distribution. Each value in the output vector is between 0 and 1, and the sum of all values is 1.

The `np.exp()` function is applied to each element of the matrix X, the result is the exponential value of the corresponding element in X

`np.sum()` sums each row.

$\frac{x_{exp}}{x_{sum}}$ uses numpy's broadcasting to divide each element of $x_{exp}$ by the corresponding row total in $x_{sum}$.

For example: `[9 2 5 0 0][7 5 0 0 0]`

```
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
(m,n).

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

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

    x_exp = np.exp(x)
    x_sum = np.sum(x_exp, axis=1, keepdims=True)
    s = x_exp / x_sum

    return s
```

## Vectorization

### Implement the L1 and L2 loss functions

**Exercise 8 - L1**

$L_1(\hat{y}, y) = \sum_{i=0}^{m-1}|y^{(i)} - \hat{y}^{(i)}|$

np.abs() calculates the absolute difference between each pair of predicted and actual values

Sum the absolute differences to obtain the L1 loss value

For example: `y = [0.9 0.2 0.1 0.4 0.9]`, `y = ]1 0 0 1 1]`, result $L_1$ = 1.1

=> Used to evaluate model performance and adjust model parameters in machine learning problems.

```
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
    """

    loss = np.sum(np.abs(y-yhat))

    return loss
```

**Exercise 9 - L2**

L2 calculates the sum of the squares of the difference between the model's prediction and the actual value.

$L_2(\hat{y},y) = \sum_{i=0}^{m-1}(y^{(i)} - \hat{y}^{(i)})^2$

For example: `y = [0.9 0.2 0.1 0.4 0.9]`, `y = [1 0 0 1 1]`, result $L_2$ = 0.43

```
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
    """

    loss = np.sum(np.square(y-yhat))
```

```
    return loss
```