Red neuronal simple

En este notebook programaremos una red neuronal simple utilizando numpy. El objetivo es que comprendamos la diferencia con el perceptrón. A diferencia del ejercicio anterior en este usaremos funciones de numpy.

- *numpy.dot(a, b, out=None)* Dot product of two arrays. Parameters: a array_like First argument, b array_like Second argument.

@juan1rving

```
import numpy as np
```

Calcular producto punto

El primer paso es calcular el logit, *h*, a partir del producto punto. La fórmula explícita es la siguiente:

$$h = WX + b$$

```
def linear_combination(X , W , b):
    h = np.dot(W, X) + b
    return h
```

Función de activación

Para este ejemplo utilizaremos la función sigmoide como función de activación.

$$f(x) = \frac{1}{1 + e^{-x}}$$

```
def sigmoid(h:np.float64) -> np.float64:
    """
    Compute de sigmoid function for h

    Parameters:
    h (numpy.float64): Number to which sigmoid function is applied

    Returns:
    numpy.float64: Output to sigmoid function
    """
    sg = 1 / (1 + np.exp(-h))
    return sg
```

Neurona simple

Implementación de la neurona simple

```
def neuron(W:np.ndarray, X:np.ndarray, b:float, activation) -> np.float64:
    """Fuction that simulates a simple neural network

    Args:
        W (np.ndarray): Model parameter matrix
        X (np.ndarray): Neuron inputs
        b (float): bias
        activation: activation function

    Returns:
        np.float64: Neural Network prediction
    """
    h = linear_combination(W, X, b)
    return activation(h)
```

Probar la red

Ahora definamos unos pesos y veamos el resultado de una pasada frontal (fordward pass).

```
# Definamos unos pesos y sesgo
inputs = np.array([0.7, -0.3])
weights = np.array([0.1, 0.8])
bias = -0.1
activation_function = sigmoid
output = neuron(weights, inputs, bias, activation_function)

print("\t\t\t***SIMPLE NEURAL NETWORK WITH NUMPY***")
print(f"Inputs: {inputs}")
print(f"Weights: {weights}")
print(f"Bias: {bias}")
print(f'Output with sigmoid function: {output}')
```

```
                            ***SIMPLE NEURAL NETWORK WITH NUMPY***
Inputs: [ 0.7 -0.3]
Weights: [0.1 0.8]
Bias: -0.1
Output with sigmoid function: 0.4329070950345457
```

```
def tanh(h:np.float64):
    """
    Compute the hyperbolic tangent for h.

    Parameters:
    h (numpy.float64): Input number.
```

```
    Returns:
    numpy.np.float64: Hyperbolic tangent evaluation
    """
    return np.tanh(h)


activation_function = tanh
output = neuron(weights, inputs, bias, activation_function)

print("\t\t\t***SIMPLE NEURAL NETWORK WITH NUMPY***")
print(f"Inputs: {inputs}")
print(f"Weights: {weights}")
print(f"Bias: {bias}")
print(f'Output with hyperbolic tangent function: {output}')
```

```
                            ***SIMPLE NEURAL NETWORK WITH NUMPY***
Inputs: [ 0.7 -0.3]
Weights: [0.1 0.8]
Bias: -0.1
Output with hyperbolic tangent function: -0.26362483547220333
```

Contesta: ¿La tanh y sigmoide producen las mismas salidas?

No, ambas funciones arrojan diferentes resultados a su salida. Si bien, ambas reciben el mismo parámetro h a sus entradas, dichas funciones se definen de manera diferente. Además de que el rango de sigmoide(h) pertenece a [0,1], mientras que el de tanh(h) pertenece a [-1,1]

Conclusiones

En esta práctica, observamos cómo diferentes funciones de activación afectan la salida de una red neuronal simple implementada con NumPy. Al emplear la función sigmoide como función de activación, los resultados estaban restringidos en el rango de 0 a 1, mientras que al utilizar la función tangente hiperbólica, los valores de salida se encontraban en el rango de -1 a 1.

Esto resalta la influencia significativa que tiene la elección de la función de activación en el comportamiento y la capacidad de la red neuronal para modelar relaciones entre las entradas y las salidas. Además, ilustra cómo diferentes funciones de activación pueden ser más adecuadas para diferentes tipos de problemas o datos.