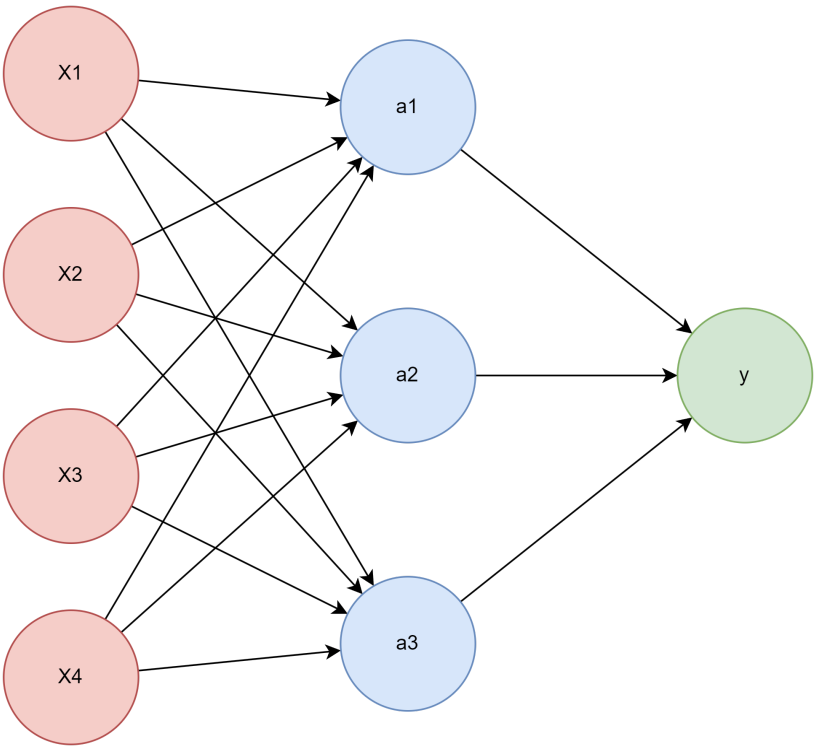
Manual Técnico

Yefer Alvarado 201731163

Requerimientos

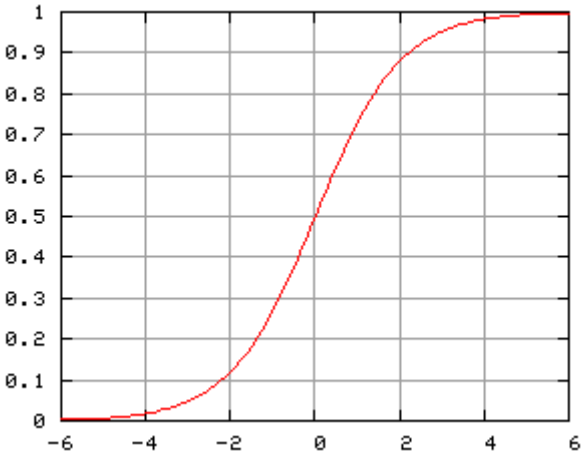
- Python 3.9
- Conocimientos de Redes Neuronales multicapa
- Librería Numpy
- Algebra Lineal
- Cálculo de Gradientes

Redes Neuronales Multicapa



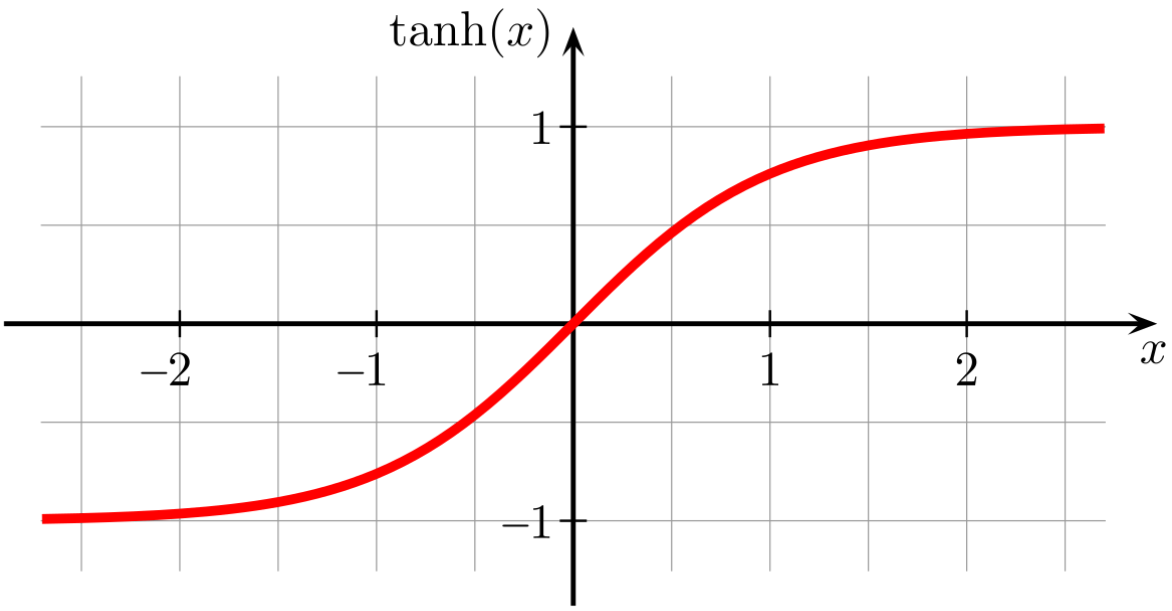
Funciones Sigmoide

Una de las razones para utilizar la función sigmoide –función Logística– es por sus propiedades matemáticas, en nuestro caso, sus derivadas. Cuando más adelante la red neuronal haga backpropagation para aprender y actualizar los pesos, haremos uso de su derivada. En esta función puede ser expresada como productos de f y 1-f . Entonces f'(t) = f(t)(1 – f(t)). Por ejemplo la función tangente y su derivada arco-tangente se utilizan normalizadas, donde su pendiente en el origen es 1 y cumplen las propiedades.



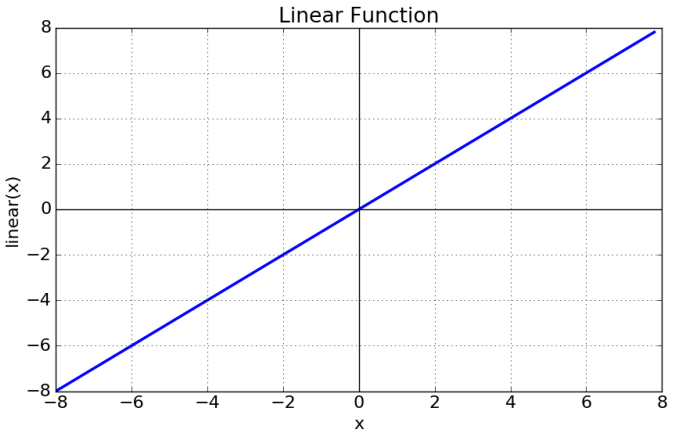
Tangente Hiperbólica

La Tangente Hiperbólica es una función de activación simétrica y continua que tiene una salida en el intervalo [-1, 1]. Esto significa que es una función simétrica en torno al origen, lo que la convierte en una opción ideal para redes neuronales que requieren una salida balanceada. Además, la Tangente Hiperbólica es una función no lineal, lo que permite a las redes neuronales modelar relaciones no lineales entre los datos.

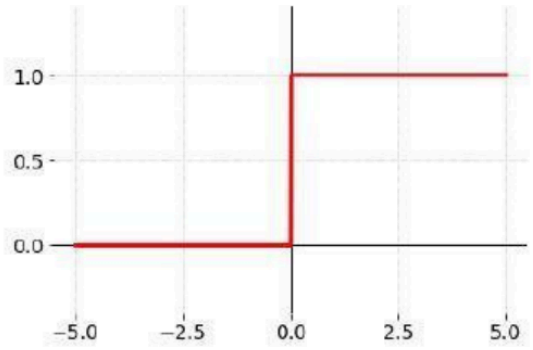


Funcion Identidad

Las transformadas lineales son básicamente la función de identidad, donde la variable dependiente tiene una relación directa y proporcional con la variable independiente. En términos prácticos, significa que la función pasa la señal sin cambios.



Funcion Step



$$f(x) = 0six < 0$$

$$f(x) = 1six \geq 0$$

Generacion de Pesos y Umbrales aleatorios

```python
def generate_values(self):
    self.weights = []
    self.biases = []

    # YA QUE LAS CAPAS OCULTAS PUEDENTE VARIAR EL EN NUMERO DE NEURONAS
    # ES NECESARIO NIVELAR LOS PESOS CON LAS ENTRADAS
    for i in range(self.hidden_layers):
        layer_weights = []
        if i == 0:
            self.weights.append(np.random.randn(self.inputs, self.neurons_hidden_layers[i]))
            self.biases.append(np.zeros((1, self.neurons_hidden_layers[i])))
            # self.biases.append(np.random.rand(1, self.neurons_hidden_layers[i]))
        else:
            self.weights.append(np.random.randn(self.neurons_hidden_layers[i-1], self.neurons_hidden_layers[i]))
            self.biases.append(np.zeros((1, self.neurons_hidden_layers[i])))
            # self.biases.append(np.random.rand(1, self.neurons_hidden_layers[i]))

    self.weights.append(np.random.randn(self.neurons_hidden_layers[-1], self.outputs))
    self.biases.append(np.zeros((1, self.outputs)))
```

Forward Propagation

Con Feedforward nos referimos al recorrido de “izquierda a derecha” que hace el algoritmo de la red, para calcular el valor de activación de las neuronas desde las entradas hasta obtener los valores de salida.

$$X = [x_0\, x_1\, x_2]$$

$$z^{layer2} = O_1 X$$

$$a^{layer2} = g(z^{layer2})$$

$$z^{layer3} = O_2 a^{layer2}$$

$$y = g(z^{layer3})$$

Backpropagation

Al hacer backpropagtion es donde el algoritmo itera para aprender! Esta vez iremos de "derecha a izquierda" en la red para mejorar la precisión de las predicciones. El algoritmo de backpropagation se divide en dos Fases: Propagar y Actualizar Pesos.

Fase 1: Propagar

Esta fase implica 2 pasos:

1.1 Hacer forward propagation de un patrón de entrenamiento (recordemos que es este es un algoritmo supervisado, y conocemos las salidas) para generar las activaciones de salida de la red.

```
def forward_pass(self, input_list):
    self.output_result = []
    self.output_result.append(self.sigmoid(np.dot(input_list, self.weights[0]) + self.biases[0]))

    for i in range(1, self.hidden_layers):
        z = np.dot(self.output_result[i-1], self.weights[i])
        self.output_result.append(self.activation_function_hidden_layers(z + self.biases[i]))

    z = np.dot(self.output_result[self.hidden_layers-1], self.weights[-1])
    self.output_result.append(self.activation_function_outputs(z + self.biases[-1]))
    return self.output_result[-1]
```

1.2 Hacer backward propagation de las salidas (activación obtenida) por la red neuronal usando las salidas "y" reales para generar los Deltas (error) de todas las neuronas de salida y de las neuronas de la capa oculta.

Fase 2: Actualizar Pesos:

Para cada <<sinapsis>> de los pesos:

2.1 Multiplicar su delta de salida por su activación de entrada para obtener el gradiente del peso.

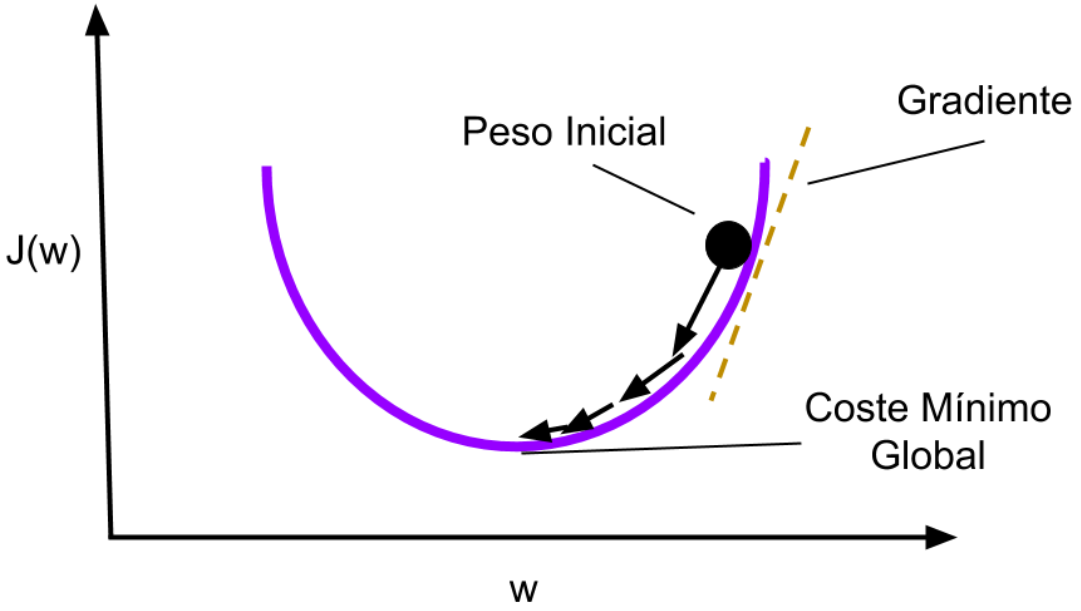
2.2 Substraer un porcentaje del gradiente de ese peso

```
def backward_pass(self, input_list, output_list):
    delta_errors = [None] * (self.hidden_layers + 1)
    # errors = y - output_res
    error = output_list - self.output_result[-1]
    # delta error
    delta_errors[-1] = error * self.derivate_function_hidden_layers(self.output_result[-1])

    for i in range(self.hidden_layers - 1, -1, -1):
        error = np.dot(delta_errors[i + 1], self.weights[i + 1].T)
        delta_errors[i] = error * self.derivate_function_hidden_layers(self.output_result[i])

    for i in range(len(self.weights)):
        self.weights[i] += np.dot(self.output_result[i].T, delta_errors[i]) * self.learning_rate
        self.biases[i] += np.sum(delta_errors[i], axis=0, keepdims=True) * self.learning_rate
```

El porcentaje que utilizaremos en el paso 2.2 tiene gran influencia en la velocidad y calidad del aprendizaje del algoritmo y es llamado “learning rate” ó tasa de aprendizaje. Si es una tasa muy grande, el algoritmo aprende más rápido pero tendremos mayor imprecisión en el resultado. Si es demasiado pequeño, tardará mucho y podría no finalizar nunca.



Si denotamos al error en el layer “l” como d(l) para nuestras neuronas de salida en layer 3 la activación menos el valor actual será (usamos la forma vectorial):

d(3) = alayer3 – y

d(2) = OT2 d(3) . g'(zlayer2)

g'(zlayer2) = alayer2 . (1 – alayer2)

El valor del costo -que es lo que queremos minimizar- de nuestra red será

J = alayer dlayer + 1