Descenso por gradiente

En este notebook implementaremos un solo paso del método de descenso por gradiente. El método es una técnica de optimización utilizada para encontrar el mínimo de una función de manera iterativa. En el contexto de redes neuronales, se utiliza para minimizar la función de costo, que mide el error entre las predicciones del modelo y los valores reales. El proceso comienza con una estimación inicial para los parámetros del modelo, y luego, en cada paso, ajusta estos parámetros en la dirección opuesta al gradiente de la función de costo, que indica la dirección de mayor aumento. La magnitud del ajuste en cada paso se determina por un parámetro llamado tasa de aprendizaje. El proceso se repite hasta alcanzar un mínimo local o hasta que el cambio en la función de costo entre iteraciones sea insignificante, indicando que el modelo ha convergido a una solución.

gradiente

Open in Colab

@juan1rving

```
import numpy as np
```

Definimos la red neuronal

```
# función de activación
def sigmoid(h):
    return 1/(1+np.exp(-h))

# Derivada de f
def sigmoid_prime(h):
    return sigmoid(h) * (1 - sigmoid(h))

# función h lineal
def combinacion_lineal (X , W , b):
    h = np.dot(W,X) + b
    return h

# Neurona
def neurona(X,W,b):
    return sigmoid(combinacion_lineal(X,W,b))
```

Término de error

Escribe una función que calcule el término de error

$$\delta = (y - \hat{y})\, f'(h) = (y - \hat{y})\, f'\left(\sum_i w_i x_i\right)$$

$$\delta = (y - \hat{y}) f'(h) = (y - \hat{y}) f'\left(\sum_i w_i x_i\right)$$

```
def error_term(y,W,X,b):
    difference = y - neurona(X, W, b)
    h = combinacion_lineal(X, W, b)
    return difference * sigmoid_prime(h)
```

Incremento

Escribe una función para determinar el incremento a uno de los pesos

$$\Delta w_i = \eta \delta x_i$$

```
def increment(W, X, b, eta, i, y):
    return eta * error_term(y, W, X, b) * X

def update_wights(W, increment_vector):
    return W + increment_vector
```

Verificar funcionamiento

A continuación implementemos una red de ejemplo y verificaremos que está funcionando almenos un paso del método de descenso por gradiente.

```
# valores de ejemplo
learning_rate = 1.0
X = np.array([1,1])
y = 5

# Valores iniciales de los pesos
w = np.array([0.1,0.2])
b = 0

print("\t\t\t***GRADIENT DESCENT METHOD***")
print(f"Inputs: {X}")
print(f"Objective value: {y}")
print(f"Initial weights: {w}")
print(f"Learning rate: {learning_rate}\n")

salida = neurona(X, w, b)
print('Output:', salida)

residual = y - salida
print('Error:', residual)

# Calcula el incremento de los pesos
incr_vector = increment(w, X, b, learning_rate, None, y)
print('Increment vector:', incr_vector)
```

```
# Calcula el nuevo valor del los pesos
nw = update_wights(w, incr_vector)
print('New weights:', nw)

salida = neurona(X, nw, b)
nuevo_error = y - salida
print('New error:', nuevo_error)

if nuevo_error < residual:
    print("\n>>>The Gradient Descent has been successful :)")
```

```
                          ***GRADIENT DESCENT METHOD***
    Inputs: [1 1]
    Objective value: 5
    Initial weights: [0.1 0.2]
    Learning rate: 1.0

    Output: 0.574442516811659
    Error: 4.425557483188341
    Increment vector: [1.08186431 1.08186431]
    New weights: [1.18186431 1.28186431]
    New error: 4.078440380493933

    >>>The Gradient Descent has been successful :)
```

Conclusiones

En esta práctica implementamos un paso del algoritmo de descenso por gradiente para una red neuronal simple. Utilizamos la función sigmoide como función de activación y calculamos el error en términos del gradiente de la función de error con respecto a los pesos de la red. A partir de esto, ajustamos los pesos para minimizar el error.

Observamos que al actualizar los pesos con el incremento calculado, el error disminuyó, lo que indica un progreso en la dirección correcta hacia la minimización del error. Este proceso de ajuste iterativo de los pesos mediante el descenso por gradiente es fundamental en el entrenamiento de redes neuronales.