Josue Santana Robledo Corona 325073061

Maestría en Ciencias de la Robótica e Inteligencia Artificial

Algoritmos Bio-Inspirados

Centro Universitario de Ciencias Exactas e Ingeniería.

Mtro. Carlos Alberto López Franco

1. Introducción

Este reporte detalla la implementación de este algoritmo crucial. A diferencia de los métodos anteriores que no requieren derivadas, el descenso y ascenso de gradiente aprovecha la pendiente del terreno para encontrar la dirección de mayor descenso.

2. Descripción del código

El código implementa el gradiente descendente y ascendente, con el objetivo de encontrar el mínimo o máximo local de la función objetivo.

3. Componentes del Código

Variables y parámetros iniciales

```
Rango = [-4, 8]  # Rango de visualización (no utilizado en el algoritmo)
presicion = 0.0001  # Precisión para el criterio de parada
x_inicial = 3  # Punto inicial para comenzar la optimización
paso = 0.2  # Tasa de aprendizaje (learning rate)
max_iter = 300  # Número máximo de iteraciones permitidas

# Listas para almacenar el historial de la optimización
puntos_x = []  # Almacenará los valores de x en cada iteración
puntos_y = []  # Almacenará los valores de f(x) en cada iteración
iteraciones = [0]  # Almacenará el número de iteración
```

Función Objetivo

```
def funcion(x):
  return np.sin(x)
```

Cambiamos la función por una que sirva tanto para el descendente como para el asce ndente, ya que la función cuadrática que usábamos anteriormente solo subiría.

Función Derivada

```
def derivada(x):
  return np.cos(x)
```

4. Algoritmo Principal

El algoritmo sigue estos pasos en cada iteración:

Cálculo del gradiente: Se calcula la derivada de la función en el punto actual

Actualización de parámetros:

x = x - (paso * gradiente)
Este es el caso para el gradiente descendente

Si se quiere un gradiente ascendente, se usa la siguiente formula
x = x + (paso * gradiente)

Registro de resultados: Se almacenan los valores de x y f(x) para visualización

Criterio de parada: El ciclo continúa hasta que el gradiente sea menor que la precisión establecida o se alcance el máximo de iteraciones

```
if abs(gradiente) > presicion:
  gradiente = derivada(x)  # Calcular el gradiente en el punto actual
  x = x + (paso * gradiente)  # Actualizar x: movimiento en dirección opuesta al gradiente

  # Almacenar historial
  puntos_x.append(x)
  puntos_y.append(funcion(x))
  iteraciones.append(i + 1)  # +1 porque i comienza en 0

  # Mostrar información de la iteración actual
  print(f"{i + 1:<12}{x:<20.8f}{funcion(x):<25.8f}{gradiente:<20.8f}")
else:
  break  # Salir del bucle si se alcanza la convergencia
```

El alma del código ocurre aquí, lo que estamos haciendo es que al entrar en cada ciclo validamos si aún nos falta para llegar a la precisión, de esta forma establecemos 2 formas de parar el ciclo, una es llegando a la precisión y la otra por ciclos, en cada ciclo calculamos el gradiente, guardamos los puntos y las iteraciones, imprimimos la tabla y volvemos al ciclo.

5. **Resultados Obtenidos**

Primero expondremos los resultados del gradiente descendente

43	4.71221493	-0.99999998	-0.00021756
44	4.71224974	-0.99999999	-0.00017405
45	4.71227759	-0.99999999	-0.00013924
46	4.71229987	-1.00000000	-0.00011139
47	4.71231769	-1.00000000	-0.00008911

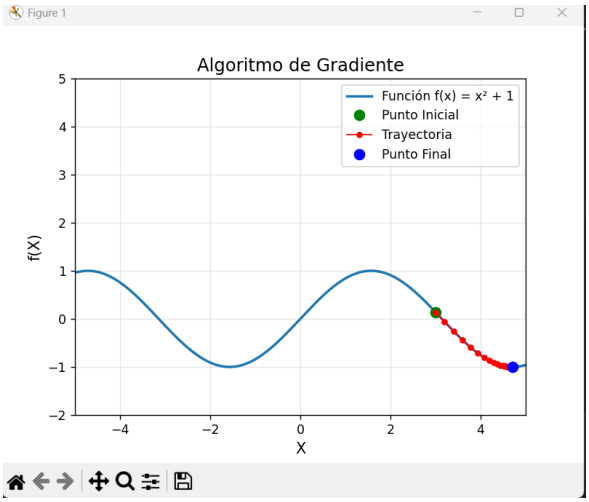

Imagen 1
Gráfica resultante de gradiente descendente

Y ahora los resultados del gradiente ascendente

43	1.57092296	0.99999999	-0.00015829
44	1.57089763	0.99999999	-0.00012663
45	1.57087737	1.00000000	-0.00010131
46	1.57086116	1.00000000	-0.00008104

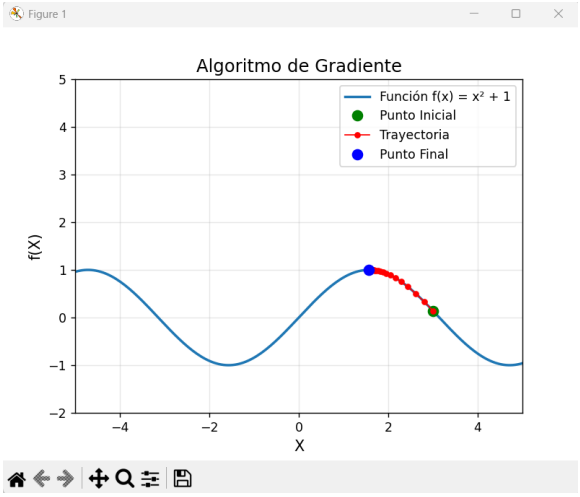

Imagen 1
Gráfica resultante de gradiente ascendente

6. Ventajas del Método

Solo requiere el cálculo del gradiente en cada paso, lo que hace computacionalmente barata.

7. Limitaciones

Solo puede calcular un mínimo o máximo local, pero no ve el panorama completo, de modo que no sabe si se trata de un mínimo global.

8. Conclusiones

El código implementa correctamente el algoritmo de descendente gradiente, al principio creía que estaría un poco más fácil de implementar, se veía como el primer algoritmo, si tuve que consultar fuentes y ejemplos para hacer mi propia versión, pero al final se llegó al resultado, gracias a esto aprendí más respecto al gradiente, y veo la importancia que tendrá para el DeepLearning y porque es tan utilizado.

9. Código

```
import matplotlib.pyplot as plt

import numpy as np


# Parámetros iniciales del algoritmo

Rango = [-4, 8]  # Rango de visualización (no utilizado en el algoritmo)

presicion = 0.0001  # Precisión para el criterio de parada

x_inicial = 3  # Punto inicial para comenzar la optimización

paso = 0.2  # Tasa de aprendizaje (learning rate)

max_iter = 300  # Número máximo de iteraciones permitidas


# Listas para almacenar el historial de la optimización

puntos_x = []  # Almacenará los valores de x en cada iteración

puntos_y = []  # Almacenará los valores de f(x) en cada iteración

iteraciones = [0]  # Almacenará el número de iteración



# Función objetivo que queremos minimizar: f(x) = x^2 + 1

def funcion(x):

   return np.sin(x)



# Derivada de la función objetivo: f'(x) = 2x
```

```
def derivada(x):
    return np.cos(x)



# Implementación del algoritmo de descenso de gradiente
def gradiente_descendente(x_inicial, paso, max_iter):
    x = x_inicial  # Inicializar x con el valor inicial
    puntos_x.append(x)  # Guardar el punto inicial
    puntos_y.append(funcion(x))  # Guardar el valor de la función en el punto inicial

    gradiente = derivada(x)  # Calcular el gradiente en el punto inicial

    # Encabezado de la tabla de resultados
    print(f"{'Iteracion':<12}{'X':<20}{'f(x)':<25}{'Gradiente':<20}")
    print("-" * 80)

    # Bucle principal de optimización
    for i in range(max_iter):
        # Verificar si el gradiente es mayor que la precisión (criterio de parada)
        # NOTA: Esto debería usar el valor absoluto del gradiente para funcionar correctamente
        if abs(gradiente) > presicion:  # CORRECCIÓN: Debería ser abs(gradiente)
            gradiente = derivada(x)  # Calcular el gradiente en el punto actual
            x = x - (paso * gradiente)  # Actualizar x: movimiento en dirección opuesta al gradiente

            # Almacenar historial
            puntos_x.append(x)
            puntos_y.append(funcion(x))
            iteraciones.append(i + 1)  # +1 porque i comienza en 0
```

```
        # Mostrar información de la iteración actual
        print(f"{i + 1:<12}{x:<20.8f}{funcion(x):<25.8f}{gradiente:<20.8f}")
    else:
        break  # Salir del bucle si se alcanza la convergencia

  return puntos_x, puntos_y, iteraciones



# Ejecutar el algoritmo de descenso de gradiente
gradiente_descendente(x_inicial, paso, max_iter)

# Preparar datos para visualización
x = np.linspace(-10, 10, 400)  # Crear 400 puntos en el rango [-10, 10]
y_func = funcion(x)  # Calcular f(x) para cada punto

# Crear visualización gráfica
plt.plot(x, y_func, label="Función f(x) = x^2 + 1", linewidth=2)  # Graficar la función
plt.plot(x_inicial, funcion(x_inicial), 'go', markersize=8, label="Punto Inicial")  # Punto inicial
plt.plot(puntos_x, puntos_y, "ro-", label="Trayectoria", markersize=4, linewidth=1)  # Trayectoria
plt.plot(puntos_x[-1], puntos_y[-1], "bo", markersize=8, label="Punto Final")  # Punto final

# Configuración del gráfico
plt.title("Algoritmo de Gradiente", fontsize=14)
plt.xlabel("X", fontsize=12)
plt.ylabel("f(X)", fontsize=12)
plt.xlim(-5, 5)
plt.ylim(-2, 5)
plt.legend()
plt.grid(True, alpha=0.3)
```

```
plt.show()
```