Universidad Nacional del Altiplano - Puno

Facultad de Ingeniería Estadística e Informática

Métodos de Optimización

Trabajo Encargado - N° 008

Docente: Fred Torres Cruz

Autor: Juan Carlos Anquise Vargas

Código: 191062

6 de noviembre de 2024

1. ENUNCIADO Y CÓDIGO PYTHON

1.1. ENUNCIADO

Resumen de la sección 2.2 - Gradientes y jacobianos (págs. 21-27) Esta sección repasa las definiciones básicas y los conceptos relacionados con la diferenciación de funciones multivaluadas, centrándose en las propiedades esenciales para la comprensión de los modelos diferenciables. Derivada: La derivada de una función en un punto se define como la tasa de cambio de la función en ese punto. Geométricamente, se puede entender como la pendiente de la recta tangente que pasa por un punto. Esta pendiente indica cómo evoluciona la función en una vecindad cercana al punto. Derivada direccional: Generaliza el concepto de derivada a funciones con múltiples variables, y permite calcular la tasa de cambio de la función en una dirección particular. Se define como el límite del cambio de la función dividido por la magnitud del desplazamiento en esa dirección. Gradiente: Para funciones con múltiples variables, el gradiente es un vector que apunta en la dirección del mayor aumento de la función. Sus componentes son las derivadas parciales de la función con respecto a cada variable. Conocer el gradiente permite calcular todas las posibles derivadas direccionales. Jacobiano: Es una generalización del gradiente para funciones que tienen múltiples salidas. Se representa como una matriz donde cada fila corresponde a una salida y cada columna a una variable de entrada. Cada entrada del jacobiano es la derivada parcial de una salida con respecto a una entrada. Teorema de Taylor: Permite aproximar una función en un punto utilizando su derivada. Esta aproximación de primer orden se puede visualizar como la recta tangente a la función en ese punto. Resumen de la sección 2.3 - Optimización numérica y descenso de gradiente (págs. 27-28) Esta sección describe cómo se pueden usar los gradientes para optimizar funciones, específicamente a través del método de descenso de gradiente. Direcciones de descenso: En el contexto de la optimización, una dirección de descenso es cualquier dirección en la que la función disminuya su valor. Para funciones diferenciables, se pueden cuantificar todas las direcciones de descenso utilizando la derivada direccional. Dirección de descenso más pronunciado: Entre todas las direcciones de descenso, la que produce el mayor descenso en la función es la dirección opuesta al gradiente. Descenso de gradiente: Es un algoritmo iterativo que busca el mínimo de una función moviéndose en la dirección de descenso más pronunciado. En cada iteración, se calcula el gradiente de la función en el punto actual y se actualiza la posición en la dirección opuesta al gradiente. Momento: Es una técnica para mejorar la convergencia del descenso de gradiente. Introduce un término de ïnercia"que conserva parte de la dirección del gradiente de la iteración anterior, suavizando el movimiento del algoritmo.

1. **Derivadas de funciones escalares** La derivada de una función $y = f(x)$ es definida como:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Las propiedades principales incluyen linealidad, regla del producto, y la regla de la cadena, aplicables a funciones escalares y vectores.

2. **Gradientes y derivadas direccionales** Para una función $y = f(x)$ con un vector x como entrada, el gradiente se define como el vector de todas las derivadas parciales:

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \cdots & \frac{\partial f}{\partial x_d} \end{bmatrix}^T$$

Esto permite calcular derivadas direccionales, útiles en optimización.

3. **Jacobianos** Para una función vectorial $y = f(x)$, el Jacobiano es una matriz que contiene todas las derivadas parciales:

$$J = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_d} \end{bmatrix}$$

4. **Optimización Numérica y Descenso de Gradiente** En el descenso de gradiente, se actualiza una estimación x_t en cada iteración en la dirección opuesta al gradiente, con un tamaño de paso η:

$$x_{t+1} = x_t - \eta \nabla f(x_t)$$

Figura 1: Formulas

1.2. CÓDIGO PYTHON

```
import streamlit as st
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class GradientDescent:
    def __init__(self, learning_rate=0.01, momentum=0.9, max_iterations=1000,
    tolerance=1e-6):
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.max_iterations = max_iterations
        self.tolerance = tolerance

    def optimize(self, func, gradient_func, initial_point):
        current_point = np.array(initial_point, dtype=float)
        velocity = np.zeros_like(current_point)

        history_values = [func(current_point)]
```

```
        history_points = [current_point.copy()]

        for i in range(self.max_iterations):
            gradient = gradient_func(current_point)
            velocity = self.momentum * velocity - self.learning_rate *
            gradient
            new_point = current_point + velocity

            history_points.append(new_point.copy())
            history_values.append(func(new_point))

            if np.linalg.norm(new_point - current_point) < self.tolerance:
                break

            current_point = new_point

        return current_point, history_values, history_points

def example_function(x):
    """Función cuadrática: f(x,y) = x^2 + 2y^2"""
    return x[0]**2 + 2*x[1]**2

def example_gradient(x):
    """Gradiente de la función cuadrática"""
    return np.array([2*x[0], 4*x[1]])

def create_contour_data(x_range, y_range, func):
    x = np.linspace(*x_range, 100)
    y = np.linspace(*y_range, 100)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros_like(X)

    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Z[i,j] = func(np.array([X[i,j], Y[i,j]]))

    return X, Y, Z

st.title("Visualización de Descenso de Gradiente")
st.write("""
Esta aplicación demuestra el algoritmo de descenso de gradiente con momento
en una función cuadrática simple.
Ajusta los parámetros y observa cómo afectan a la convergencia del algoritmo.
""")

# Sidebar para parámetros
```

```
st.sidebar.header("Parámetros del Algoritmo")
learning_rate = st.sidebar.slider("Tasa de Aprendizaje", 0.01, 1.0, 0.1, 0.01)
momentum = st.sidebar.slider("Momento", 0.0, 0.99, 0.9, 0.01)
max_iterations = st.sidebar.slider("Máximo de Iteraciones", 100, 2000, 1000,
100)
initial_x = st.sidebar.slider("Punto Inicial X", -3.0, 3.0, 2.0, 0.1)
initial_y = st.sidebar.slider("Punto Inicial Y", -3.0, 3.0, 1.0, 0.1)

# Crear el optimizador con los parámetros seleccionados
optimizer = GradientDescent(
    learning_rate=learning_rate,
    momentum=momentum,
    max_iterations=max_iterations
)

# Optimizar
initial_point = np.array([initial_x, initial_y])
optimal_point, values_history, points_history = optimizer.optimize(
    example_function,
    example_gradient,
    initial_point
)

# Crear las visualizaciones
points_history = np.array(points_history)

# Crear dos columnas para las gráficas
col1, col2 = st.columns(2)

with col1:
    st.subheader("Convergencia de la Función Objetivo")
    fig1, ax1 = plt.subplots()
    ax1.plot(values_history)
    ax1.set_xlabel('Iteraciones')
    ax1.set_ylabel('Valor de la Función')
    ax1.grid(True)
    st.pyplot(fig1)

with col2:
    st.subheader("Trayectoria de Optimización")
    fig2, ax2 = plt.subplots()

    # Crear datos para el contour plot
    X, Y, Z = create_contour_data((-3, 3), (-3, 3), example_function)
    ax2.contour(X, Y, Z, levels=20)
    ax2.plot(points_history[:, 0], points_history[:, 1], 'b.-')
```

```
    ax2.plot(points_history[0, 0], points_history[0, 1], 'go', label='Inicio')
    ax2.plot(points_history[-1, 0], points_history[-1, 1], 'ro',
    label='Final')
    ax2.set_xlabel('x')
    ax2.set_ylabel('y')
    ax2.legend()
    ax2.grid(True)
    st.pyplot(fig2)

# Mostrar resultados
st.subheader("Resultados")
col3, col4 = st.columns(2)
with col3:
    st.write(f"Punto óptimo encontrado: [{optimal_point[0]:.4f},
    {optimal_point[1]:.4f}]")
with col4:
    st.write(f"Valor mínimo de la función:
    {example_function(optimal_point):.4f}")

# Información adicional
st.write("""
### Explicación de los Parámetros:
- **Tasa de Aprendizaje**: Controla el tamaño de los pasos en cada iteración.
- **Momento**: Determina cuánto influye la dirección anterior en el siguiente
paso.
- **Máximo de Iteraciones**: Límite de iteraciones para el algoritmo.
- **Punto Inicial**: Coordenadas (x,y) desde donde comienza la optimización.
""")

# Métricas adicionales
st.subheader("Métricas de Convergencia")
num_iterations = len(values_history) - 1
improvement = values_history[0] - values_history[-1]

col5, col6 = st.columns(2)
with col5:
    st.metric("Número de Iteraciones", num_iterations)
with col6:
    st.metric("Mejora Total", f"{improvement:.4f}")
```