C5 OPTIMIZACIÓN DEL MODELO DE CONSUMO ENERGÉTICO DE UNA CPU

December 1, 2025

1 Objetivo del proyecto

Este proyecto se enfoca en predecir el consumo de energía de una CPU basándose en su frecuencia de reloj, un problema fundamental para el diseño de sistemas de refrigeración y al gestión de la eficiencia energética.

El objetivo es construir un **modelo de regresión lineal simple** que relacione la frecuencia de reloj (GHz) con el consmo de energía (Watt). Para ello, se implementarán y compararán tres métodos de optimización para encontrar los parámetros del modelo que minimicen el error cuadrático:

- Método de descenso de gradiente
- Método de Newton
- Método Quasi-Newton (BFGS)

El modelo a optimizar es la ecuación de la recta:

$$Y = \beta_0 + \beta_1 X$$

Donde:

- Y es el **consumo de energia (Watt)**.
- X es la **frecuencia de reloj (GHz)**.
- β_0 y β_1 son los parámetros del modelo (intercepto y pendiente) que debemos estimar.

2 Datos del problema

Se utilizarán los siguientes datos, que registran la frecuencia de reloj y el consumo de energía correspondiente de una CPU.

Primero, se realiza un diagrama de dispersión para observar la relación entre las variables. Como se muestra en la gráfica, existe una clara tendencia lineal positiva: a mayor frecuencia, mayor consumo de energía.

[1]:
```
# Cargar librerías necesarias
# ---------------------------
import numpy as np
```

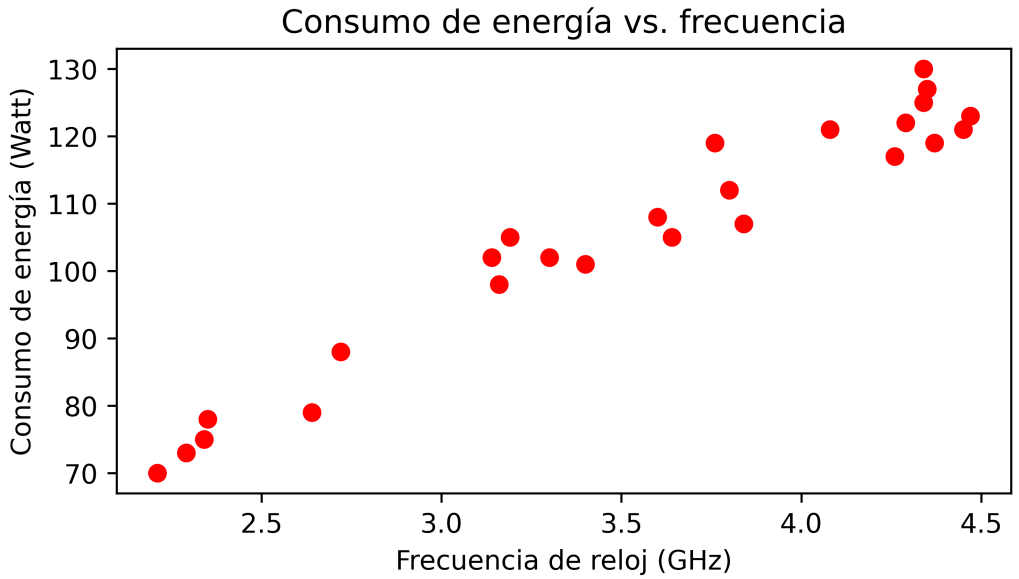
```
import matplotlib.pyplot as plt

# Configuraciones iniciales
# -------------------------
plt.rcParams["figure.dpi"] = 600
```

[2]:
```
# Datos
# -----
frecuencia_reloj = np.array([4.29, 4.34, 2.72, 4.08, 3.60,
                             3.30, 3.84, 2.34, 3.64, 3.76,
                             3.14, 3.80, 4.34, 2.64, 3.16,
                             4.35, 4.45, 2.29, 3.19, 3.40,
                             4.26, 2.35, 4.47, 4.37, 2.21])

consumo_energia = np.array([122, 130, 88, 121, 108,
                            102, 107, 75, 105, 119,
                            102, 112, 125, 79, 98,
                            127, 121, 73, 105, 101,
                            117, 78, 123, 119, 70])
```

[3]:
```
# Diagrama de dispersión
# ----------------------
fig, ax = plt.subplots(figsize = (6, 3))
ax.plot(frecuencia_reloj, consumo_energia, "ro")
ax.set_title("Consumo de energía vs. frecuencia")
ax.set_xlabel("Frecuencia de reloj (GHz)")
ax.set_ylabel("Consumo de energía (Watt)")
plt.show()
```



3 Formulación del problema de optimización

Para encontrar los mejores parámetros β_0 y β_1, se debe minimizar la **función objetivo**, que en este caso es la suma de los errores al cuadrado, la cual viene dada por:

$$f(\beta) = \sum_{i=1}^{n} (y_i - (\beta_0 + \beta_1 x_i))^2$$

Para aplicar los algoritmos, necesitamos definir su gradiente (primera derivada) y su matriz Hessiana (segunda derivada).

- **Vector gradiente** $\nabla f(\beta)$: Indica la dirección de máximo crecimiento de la función de objetivo. Lo usaremos para movernos en la dirección opuesta.

$$\nabla f(\beta) = \left[\frac{\partial f}{\partial \beta_0}, \frac{\partial f}{\partial \beta_1}\right]^\top = \left[-2\sum(y_i - (\beta_0 + \beta_1 x_i)), -2\sum x_i(y_i - (\beta_0 + \beta_1 x_i))\right]^\top$$

- **Matriz Hessiana** $\mathbf{H}_f(\beta)$: Describe la curvatura de la función. Es crucial para el método de Newton.

$$\mathbf{H}_f(\beta) = \begin{bmatrix} \dfrac{\partial^2 f}{\partial \beta_0^2} & \dfrac{\partial^2 f}{\partial \beta_0 \partial \beta_1} \\ \dfrac{\partial^2 f}{\partial \beta_1 \partial \beta_0} & \dfrac{\partial^2 f}{\partial \beta_1^2} \end{bmatrix} = \begin{bmatrix} 2n & 2\sum x_i \\ 2\sum x_i & 2\sum x_i^2 \end{bmatrix}$$

[4]:
```python
# Variables para el modelo
# ------------------------
x = frecuencia_reloj
y = consumo_energia

# Función objetivo a minimizar
# ----------------------------
def f_objetivo(beta, x, y):
    error = y - (beta[0] + beta[1] * x)
    return np.sum(error ** 2)

# Gradiente de la función objetivo
# --------------------------------
def g_objetivo(beta, x, y):
    error = y - (beta[0] + beta[1] * x)
    d_beta_0 = -2 * np.sum(error)
    d_beta_1 = -2 * np.sum(error * x)
    return np.array([d_beta_0, d_beta_1])
```

```python
# Hessiana de la función objetivo
# -------------------------------
def h_objetivo(beta, x, y):
    n = len(x)
    d00 = 2 * n
    d01 = 2 * np.sum(x)
    d11 = 2 * np.sum(x ** 2)
    return np.array([[d00, d01], [d01, d11]])
```

4 Implementación y resultados de los métodos de optimización

4.1 Método 1: Método de descenso de gradiente

La regla de actualización es:

$$\beta_{k+1} = \beta_k - \alpha \nabla f(\beta_k)$$

[5]:
```python
# Algoritmo de descenso de gradiente
# ----------------------------------
def dg_rls(beta_actual, x, y, alpha, max_iter = 10000, tol = 1e-6):
    beta_historial = [beta_actual]
    for _ in range(max_iter):
        beta_nuevo = beta_actual - alpha * g_objetivo(beta_actual, x, y)
        beta_historial.append(beta_nuevo)
        criterio_1 = np.linalg.norm(beta_nuevo - beta_actual)
        criterio_2 = np.linalg.norm(g_objetivo(beta_nuevo, x, y))
        if criterio_1 < tol or criterio_2 < tol:
            break
        beta_actual = beta_nuevo
    beta_historial = np.array(beta_historial)
    return beta_nuevo, beta_historial

# Parámetros y ejecución
alpha_dg = 0.001
beta_inicial = np.array([0.0, 1.0])
beta_dg, hist_dg = dg_rls(beta_inicial, x, y, alpha_dg)

print("--- Descenso de gradiente ---")
print("Beta estimado =", beta_dg)
print("Número de iteraciones =", len(hist_dg))
```

```
--- Descenso de gradiente ---
Beta estimado = [22.38189069 23.40600308]
Número de iteraciones = 5150
```

4.2 Método 2: Método de Newton

La regla de actualización es:

$$\beta_{k+1} = \beta_k - \mathbf{H}_k^{-1}(\beta_k)\nabla f(\beta_k)$$

[6]:
```
# Algoritmo de Newton
# -------------------
def newton_rls(beta_actual, x, y, max_iter = 10000, tol = 1e-6):
    beta_historial = [beta_actual]
    for _ in range(max_iter):
        beta_nuevo = beta_actual - np.linalg.inv(h_objetivo(beta_actual, x, y))␣
 ↪@ g_objetivo(beta_actual, x, y)
        beta_historial.append(beta_nuevo)
        criterio_1 = np.linalg.norm(beta_nuevo - beta_actual)
        criterio_2 = np.linalg.norm(g_objetivo(beta_nuevo, x, y))
        if criterio_1 < tol or criterio_2 < tol:
            break
        beta_actual = beta_nuevo
    beta_historial = np.array(beta_historial)
    return beta_nuevo, beta_historial

# Parámetros y ejecución
# ----------------------
beta_inicial = np.array([0.0, 1.0])
beta_newton, hist_newton = newton_rls(beta_inicial, x, y)

print("--- Método de Newton ---")
print("Beta estimado =", beta_newton)
print("Número de iteraciones =", len(hist_newton))
```

```
--- Método de Newton ---
Beta estimado = [22.3823692  23.40587309]
Número de iteraciones = 2
```

4.3 Método 3: Quasi-Newton (BFGS)

Este método aproxima la inversa de la Hessiana en lugar de calcularla directamente, logrando un equilibrio en la la velocidad del método de Newton y la simplicidad del descenso de gradiente.

[7]:
```
# Algoritmo Quasi-Newton
# ----------------------
def h_bfgs(h_actual, s_actual, y_actual):
    yts_actual = np.dot(y_actual, s_actual)
    if yts_actual <= 1e-10:
        return h_actual
    n = s_actual.shape[0]
    I = np.identity(n)
```

```
    aux1 = I - np.outer(s_actual, y_actual) / yts_actual
    aux2 = I - np.outer(y_actual, s_actual) / yts_actual
    aux3 = np.outer(s_actual, s_actual) / yts_actual
    h_nuevo = aux1 @ h_actual @ aux2 + aux3
    return h_nuevo

def backtracking(f, grad_f, x_actual, p_actual, alpha_inicial = 1.0, rho = 0.5,␣
 ↪c = 1e-4):
    alpha = alpha_inicial
    gradiente_actual = grad_f(x_actual, x, y)
    while f(x_actual + alpha * p_actual, x, y) > f(x_actual, x, y) + c * alpha␣
 ↪* np.dot(gradiente_actual, p_actual):
        alpha = alpha * rho
    return alpha

def qn_rls(f, grad_f, beta_actual, x, y, max_iter = 10000, tol = 1e-6):
    beta_historial = [beta_actual]
    h_actual = np.identity(len(beta_actual))
    for i in range(max_iter):
        grad_actual = grad_f(beta_actual, x, y)
        p_actual = -h_actual @ grad_actual
        alpha_actual = backtracking(f, grad_f, beta_actual, p_actual)
        beta_nuevo = beta_actual + alpha_actual * p_actual
        beta_historial.append(beta_nuevo)
        criterio_1 = np.linalg.norm(beta_nuevo - beta_actual)
        criterio_2 = np.linalg.norm(g_objetivo(beta_nuevo, x, y))
        if criterio_1 < tol or criterio_2 < tol:
            break
        grad_nuevo = grad_f(beta_nuevo, x, y)
        s_actual = beta_nuevo - beta_actual
        y_actual = grad_nuevo - grad_actual
        h_actual = h_bfgs(h_actual, s_actual, y_actual)
        beta_actual = beta_nuevo
    beta_historial = np.array(beta_historial)
    return beta_nuevo, beta_historial

# Ejecución
beta_inicial = np.array([0.0, 1.0])
beta_qn, hist_qn = qn_rls(f_objetivo, g_objetivo, beta_inicial, x, y)

print("--- Método de Quasi-Newton ---")
print("Beta estimado =", beta_qn)
print("Número de iteraciones =", len(hist_qn))
```

```
--- Método de Quasi-Newton ---
Beta estimado = [22.3823692  23.40587309]
Número de iteraciones = 7
```

5 Comparación de resultados y conclusión

Una vez ejecutados los tres algoritmos, podemos comparar su rendimiento y el modelo final obtenido. Todos los métodos convergen a la misma solución, validando la implementación. Sin embargo, su eficiencia computacional es drásticamente diferente:

- El **método de descenso de gradiente** es el más lento, requiriendo miles de iteraciones. Su simplicidad es su ventaja, pero su convergencia lenta puede ser un problema en funciones más complejas.
- El **método de Newton** es extremadamente rápido, convergiendo en solo 2 pasos. Esto se debe a que la función de costo para la regresión lineal es una función cuadrática, y Newton encuentra el mínimo de una cuadrática en una sola iteración (la segunda es para confirmar la convergencia). Su desventaja es el costo computacional de calcular e invertir la matriz Hessiana en problemas no lineales complejos.
- El **método Quasi-Newton (BFGS)** ofrece un excelente equilibrio. Es mucho más rápido que el descenso de gradiente y evita el cálculo explícito de la Hessiana, lo que lo hace ideal para problemas de optimización a gran escala donde el método de Newton sería inviable.

El modelo final de regresión lineal es:

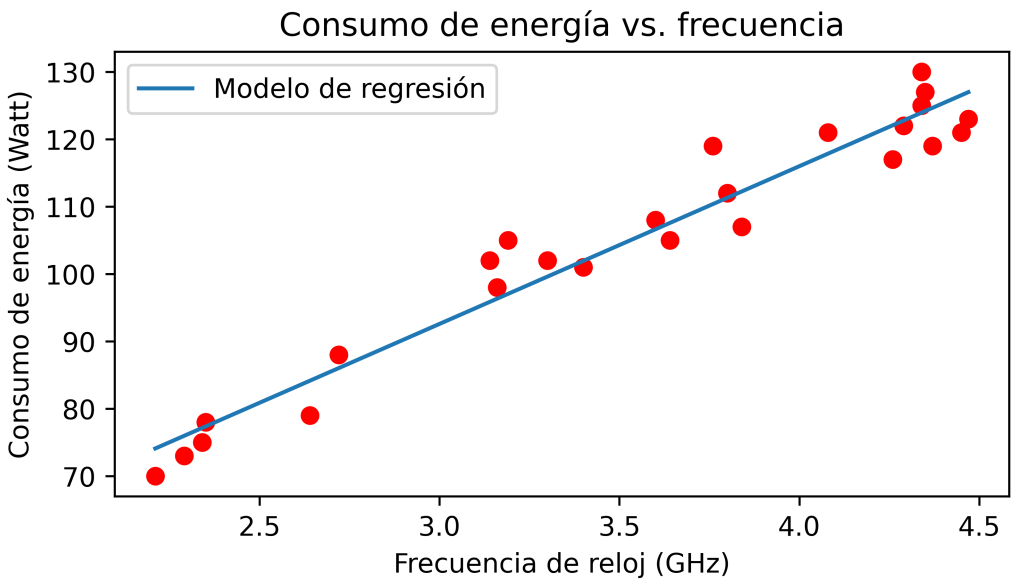
$$\text{Consumo de energía (Watt)} = 22.38 + 23.41 \times \text{Frecuencia de reloj (GHz)}$$

Este modelo permite al equipo de ingeniería predecir con precisión el consumo energético de la CPU, facilitando el diseño de sistema de disipación de calor y optimizando la gestión de la energia del sistema.

[8]:
```
# Gráfica de la recta de regresión
# --------------------------------
fig, ax = plt.subplots(figsize = (6, 3))
ax.plot(frecuencia_reloj, consumo_energia, "ro")

x_regresion = np.linspace(min(frecuencia_reloj), max(frecuencia_reloj), 100)
y_regresion = beta_newton[0] + beta_newton[1] * x_regresion

ax.plot(x_regresion, y_regresion, label = "Modelo de regresión")
ax.set_title("Consumo de energía vs. frecuencia")
ax.set_xlabel("Frecuencia de reloj (GHz)")
ax.set_ylabel("Consumo de energía (Watt)")
ax.legend()
plt.show()
```


Consumo de energía vs. frecuencia
Modelo de regresión
Consumo de energía (Watt)
Frecuencia de reloj (GHz)
130
120
110
100
90
80
70
2.5
3.0
3.5
4.0
4.5