IT Academy by KIBERNUM

Métodos de Optimización

En el aprendizaje de máquina, ajustar los parámetros de un modelo para minimizar (o maximizar) una función de costo es fundamental. Esta tarea se aborda mediante métodos de optimización, que permiten encontrar la mejor configuración de parámetros para que el modelo tenga el mejor desempeño.

R **por Kibernum Capacitación S.A.**

Introducción a los Métodos de Optimización en Machine Learning

En el aprendizaje de máquina, ajustar los parámetros de un modelo para minimizar (o maximizar) una función de costo es fundamental. Esta tarea se aborda mediante métodos de optimización, que permiten encontrar la mejor configuración de parámetros para que el modelo tenga el mejor desempeño. Uno de los métodos más utilizados es el descenso de gradiente, un algoritmo iterativo que se basa en el cálculo del gradiente de la función de costo.

Conceptos Clave:

- Función de Costo: Representa el error o la diferencia entre la salida predicha y la salida real. Por ejemplo, en un problema de regresión lineal, la función de costo más común es el Error Cuadrático Medio (MSE).
- Gradiente: Es el vector de derivadas parciales de la función de costo respecto a cada parámetro. Indica la dirección en la que la función aumenta más rápidamente. Actualizando los parámetros en la dirección opuesta al gradiente, se consigue la disminución del error.
- Tasa de Aprendizaje (Learning Rate): Es un factor que determina el tamaño de los pasos dados en cada iteración del algoritmo. Una tasa muy alta puede provocar oscilaciones o divergencia, mientras que una tasa muy baja puede ralentizar la convergencia.
- Métodos de Primer y Segundo Orden

Métodos de primer Orden

Son algoritmos de optimización que utilizan únicamente información de la primera derivada (el gradiente) para actualizar los parámetros del modelo. Algunos de los métodos de primer orden más comunes son:

Descenso de Gradiente (Gradient Descent)

Calcula el gradiente de la función de costo usando todo el conjunto de datos y actualiza los parámetros en la dirección opuesta al gradiente.

Descenso de Gradiente Estocástico (Stochastic Gradient Descent, SGD)

Actualiza los parámetros utilizando un solo ejemplo (o un lote pequeño) en cada iteración, lo que mejora la eficiencia en grandes conjuntos de datos.

Descenso de Gradiente por Mini-batches

Es una combinación entre el descenso de gradiente clásico y el estocástico, actualizando los parámetros utilizando pequeños subconjuntos de datos, lo que permite un buen balance entre precisión y eficiencia computacional.

Estos métodos son preferidos en problemas de Machine Learning a gran escala, donde la eficiencia computacional y la capacidad para manejar grandes volúmenes de datos son cruciales.

Más Métodos de Primer Orden

Métodos con Momentum

Incorporan una "memoria" del gradiente previo para suavizar y acelerar la convergencia, ayudando a evitar oscilaciones.

Nesterov Accelerated Gradient (NAG)

Una mejora del Momentum que calcula el gradiente en una posición anticipada, lo que puede resultar en una convergencia más rápida.

Métodos Adaptativos

Algoritmos como Adagrad, RMSprop y Adam ajustan la tasa de aprendizaje de manera automática para cada parámetro, basándose en el historial de gradientes. Aunque utilizan ajustes adicionales, se consideran métodos de primer orden porque dependen únicamente de la información del gradiente.

Estos métodos son preferidos en problemas de Machine Learning a gran escala, donde la eficiencia computacional y la capacidad para manejar grandes volúmenes de datos son cruciales.

Ejemplos de Métodos de Primer Orden

A continuación, se presentan ejemplos sencillos en Python para cada uno de los métodos de primer orden en optimización, utilizando una función simple.

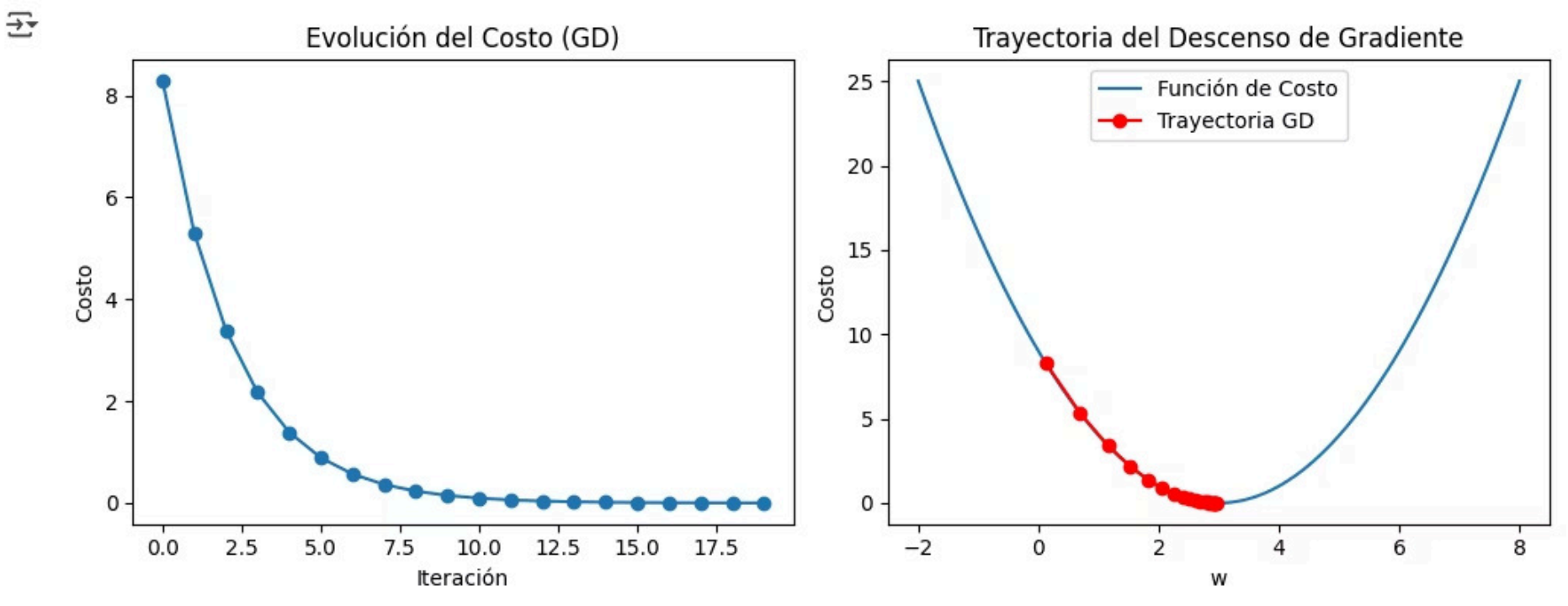
Se usa la función $f(w) = (w - 3)^2$ (que tiene mínimo en $w=3$) para ilustrar el proceso de optimización.

1. Descenso de Gradiente Básico (Gradient Descent)

En este método se calcula el gradiente de la función de costo utilizando el conjunto completo de datos (o, en este caso, la función analítica) y se actualiza el parámetro w en la dirección opuesta al gradiente.

Código de Descenso de Gradiente Básico

```python
import numpy as np
import matplotlib.pyplot as plt

def f(w):
    return (w - 3)**2

def grad_f(w):
    return 2 * (w - 3)

# Inicialización
w = np.random.randn()  # Valor inicial aleatorio
learning_rate = 0.1
num_iter = 20
w_hist = []
costo_hist = []

for i in range(num_iter):
    grad = grad_f(w)
    w = w - learning_rate * grad  # Actualización: se resta el gradiente
    w_hist.append(w)
    costo_hist.append(f(w))

# Visualización de la evolución del costo
plt.figure(figsize=(10, 4))
plt.subplot(1,2,1)
plt.plot(range(num_iter), costo_hist, marker='o')
plt.xlabel('Iteración')
plt.ylabel('Costo')
plt.title('Evolución del Costo (GD)')
plt.subplot(1,2,2)
w_range = np.linspace(-2, 8, 100)
plt.plot(w_range, (w_range - 3)**2, label='Función de Costo')
plt.plot(w_hist, costo_hist, 'ro-', label='Trayectoria GD')
plt.xlabel('w')
plt.ylabel('Costo')
plt.title('Trayectoria del Descenso de Gradiente')
plt.legend()
plt.tight_layout()
plt.show()
print("Valor final de w (GD):", w)
```




```
Valor final de w (GD): 2.958522755900109
```

Caso de uso: Optimizar el parámetro de un modelo de regresión simple minimizando el error cuadrático.

Descenso de Gradiente Estocástico (SGD)

En SGD se actualizan los parámetros utilizando un solo ejemplo (o muestra) a la vez. Imagina que tenemos un conjunto de datos donde la función de costo es la media de errores individuales.

```python
import numpy as np

# Supongamos que nuestro "dataset" es un conjunto de valores objetivo
data = np.array([2.5, 3.0, 3.5, 4.0])
w = np.random.randn()  # Inicialización
learning_rate = 0.1
num_epochs = 100

for epoch in range(num_epochs):
    for sample in data:
        grad = 2 * (w - sample)  # Gradiente para cada ejemplo: (w - sample)^2 derivada
        w = w - learning_rate * grad

print("Valor final de w (SGD):", w)
```

```
Valor final de w (SGD): 3.387533875338753
```

Caso de uso: Ajustar los parámetros de un modelo de regresión cuando se trabaja con grandes volúmenes de datos, actualizando el modelo tras cada muestra.

Descenso de Gradiente por Mini-batches

```python
import numpy as np

data = np.array([2.5, 3.0, 3.5, 4.0])
w = np.random.randn()  # Inicialización
learning_rate = 0.1
num_epochs = 100
batch_size = 2

for epoch in range(num_epochs):
    np.random.shuffle(data)
    for i in range(0, len(data), batch_size):
        batch = data[i:i+batch_size]
        grad = np.mean(2 * (w - batch))  # Promedio del gradiente en el lote
        w = w - learning_rate * grad

print("Valor final de w (Mini-batch GD):", w)
```

```
Valor final de w (Mini-batch GD): 3.2542883593983194
```

Caso de uso: Reducir la varianza en la actualización de parámetros en datasets moderados, combinando estabilidad y eficiencia computacional.

Descenso de Gradiente con Momentum

El método de Momentum añade una "memoria" del gradiente previo, lo que ayuda a suavizar y acelerar la convergencia.

```python
import numpy as np

def f(w):
    return (w - 3)**2

def grad_f(w):
    return 2 * (w - 3)

w = np.random.randn()
learning_rate = 0.1
num_iter = 20
momentum = 0.9
v = 0  # Inicialización de la velocidad
w_hist = []

for i in range(num_iter):
    grad = grad_f(w)
    v = momentum * v + learning_rate * grad  # Actualización del "momentum"
    w = w - v
    w_hist.append(w)

print("Valor final de w (Momentum):", w)
```

```
Valor final de w (Momentum): 4.272399707652392
```

Caso de uso: Mejorar la convergencia en modelos con superficies de error "irregulares", como en redes neuronales, evitando oscilaciones.

Nesterov Accelerated Gradient (NAG)

NAG es una variante del método con Momentum que calcula el gradiente en una posición anticipada (lookahead), lo que puede proporcionar actualizaciones más precisas.

```python
import numpy as np

def f(w):
    return (w - 3)**2

def grad_f(w):
    return 2 * (w - 3)

w = np.random.randn()
learning_rate = 0.1
num_iter = 20
momentum = 0.9
v = 0
w_hist = []

for i in range(num_iter):
    # Posición anticipada
    w_lookahead = w - momentum * v
    grad = grad_f(w_lookahead)
    v = momentum * v + learning_rate * grad
    w = w - v
    w_hist.append(w)

print("Valor final de w (NAG):", w)
```

```
Valor final de w (NAG): 3.1347532709782477
```

Caso de uso: Acelerar el proceso de optimización en entornos con gradientes ruidosos o cuando se requiere una convergencia más rápida.

Métodos Adaptativos: Adagrad

Adagrad adapta la tasa de aprendizaje para cada parámetro en función del historial de gradientes.

Métodos Adaptativos: RMSprop

RMSprop mejora Adagrad al introducir un factor de decaimiento en la acumulación de gradientes.

```python
import numpy as np

def f(w):
    return (w - 3)**2

def grad_f(w):
    return 2 * (w - 3)

w = np.random.randn()
learning_rate = 0.1
num_iter = 50
decay_rate = 0.9
epsilon = 1e-8
v = 0  # Promedio móvil de gradientes al cuadrado

for i in range(num_iter):
    grad = grad_f(w)
    v = decay_rate * v + (1 - decay_rate) * (grad**2)
    w = w - learning_rate * grad / (np.sqrt(v) + epsilon)

print("Valor final de w (RMSprop):", w)
```

Métodos Adaptativos: Adam

Adam combina las ventajas de Momentum y RMSprop, ajustando de forma adaptativa la tasa de aprendizaje para cada parámetro.

```python
import numpy as np

def f(w):
    return (w - 3)**2

def grad_f(w):
    return 2 * (w - 3)

w = np.random.randn()
learning_rate = 0.1
num_iter = 50
beta1 = 0.9
beta2 = 0.999
epsilon = 1e-8
m = 0  # Primer momento (promedio de gradientes)
v = 0  # Segundo momento (promedio de gradientes al cuadrado)

for t in range(1, num_iter+1):
    grad = grad_f(w)
    m = beta1 * m + (1 - beta1) * grad
    v = beta2 * v + (1 - beta2) * (grad**2)
    m_corr = m / (1 - beta1**t)  # Corrección de sesgo
    v_corr = v / (1 - beta2**t)  # Corrección de sesgo
    w = w - learning_rate * m_corr / (np.sqrt(v_corr) + epsilon)

print("Valor final de w (Adam):", w)
```

```
Valor final de w (Adam): 3.1972222401861825
```

Caso de uso para métodos adaptativos: Estos métodos se utilizan ampliamente en el entrenamiento de redes neuronales profundas, donde la alta dimensionalidad y la variabilidad de los gradientes requieren una adaptación constante de la tasa de aprendizaje para garantizar una convergencia estable y rápida.

Métodos de Segundo Orden

Los métodos de segundo orden son algoritmos de optimización que aprovechan la información de las segundas derivadas (a través de la matriz Hessiana) para mejorar la convergencia al ajustar los parámetros del modelo. Entre los métodos de segundo orden más comunes se encuentran:

Método de Newton

Utiliza tanto el gradiente como la Hessiana para determinar la dirección y el tamaño óptimo del paso. Su actualización se basa en la fórmula:

$$\mathbf{w}_{\text{new}} = \mathbf{w} - H^{-1}\nabla J(\mathbf{w}),$$

donde H es la matriz Hessiana y ∇J(w) es el gradiente de la función de costo. Es muy preciso, pero puede ser costoso computacionalmente debido al cálculo y la inversión de la Hessiana.

Métodos Quasi-Newton

Estos métodos buscan aproximar la Hessiana en lugar de calcularla explícitamente, reduciendo el costo computacional.

- BFGS (Broyden-Fletcher-Goldfarb-Shanno): Es uno de los métodos quasi-Newton más populares y actualiza iterativamente una aproximación de la Hessiana inversa.
- L-BFGS (Limited-memory BFGS): Es una variante de BFGS diseñada para problemas de alta dimensión, ya que utiliza menos memoria al almacenar solo una parte de la información histórica.

Método de Levenberg-Marquardt

Específicamente utilizado en problemas de ajuste de curvas y en la optimización de modelos no lineales, combina aspectos del método de Newton y del descenso de gradiente, ajustando dinámicamente la actualización de parámetros para mejorar la estabilidad.

Estos métodos son particularmente útiles cuando el costo de calcular la Hessiana completa es manejable y se requiere una convergencia rápida y precisa, aunque en problemas de alta dimensionalidad suelen preferirse métodos de primer orden o versiones limitadas de métodos de segundo orden (como L-BFGS) por cuestiones de escalabilidad y eficiencia computacional.

Ejemplos

1. Método de Newton (para una función unidimensional)

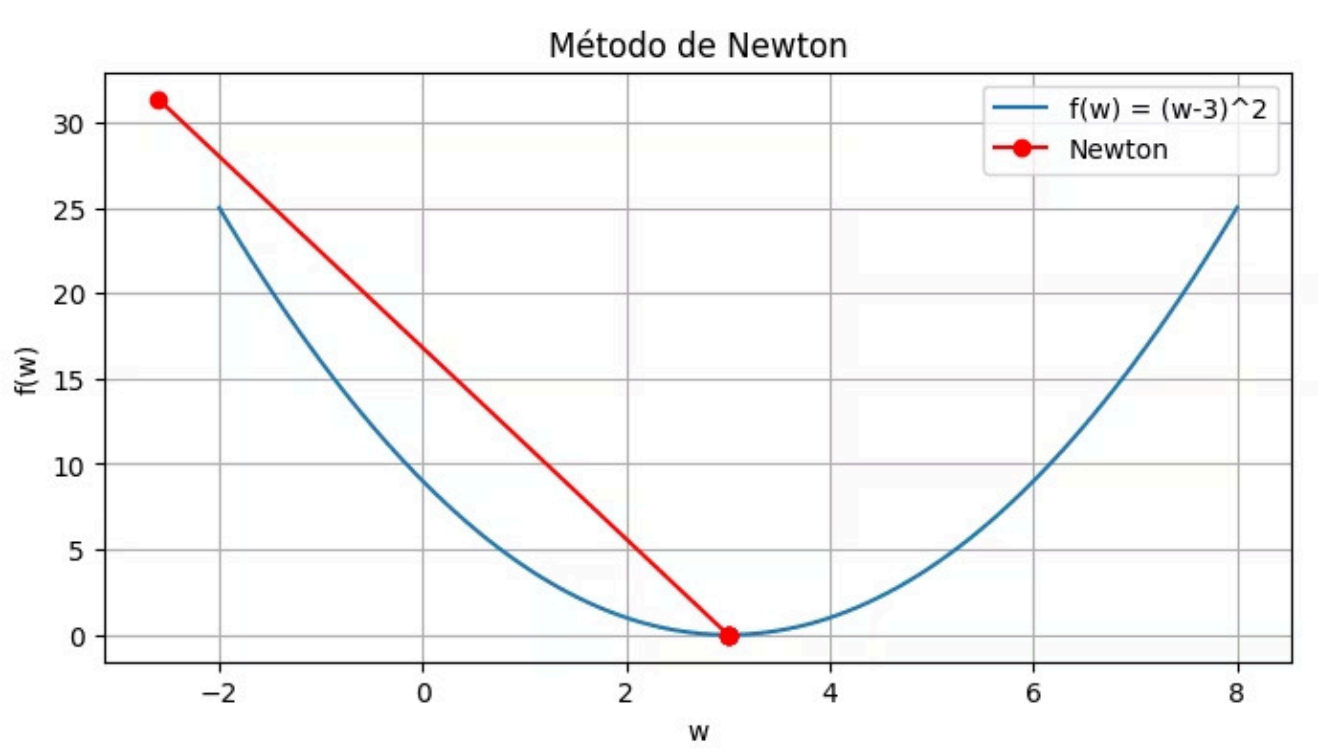
Considera la función $f(w)=(w-3)^2$. Su mínimo se encuentra en $w=3$. El método de Newton actualiza el parámetro según:

$$w_{\text{new}} = w - \frac{f'(w)}{f''(w)}$$

Donde

$$f'(w) = 2(w-3) \text{ y } f''(w) = 2.$$

```python
import numpy as np
import matplotlib.pyplot as plt

# Definir la función y sus derivadas
def f(w):
    return (w - 3)**2

def grad_f(w):
    return 2 * (w - 3)

def hess_f(w):
    return 2  # Derivada segunda constante

# Método de Newton
w = np.random.randn()  # Valor inicial aleatorio
w_hist = [w]
num_iter = 5

for i in range(num_iter):
    grad = grad_f(w)
    hess = hess_f(w)
    w = w - grad / hess
    w_hist.append(w)

# Visualización de la convergencia
w_range = np.linspace(-2, 8, 100)
f_vals = f(w_range)
plt.figure(figsize=(8,4))
plt.plot(w_range, f_vals, label='f(w) = (w-3)^2')
plt.plot(w_hist, [f(val) for val in w_hist], 'ro-', label='Newton')
plt.xlabel('w')
plt.ylabel('f(w)')
plt.title('Método de Newton')
plt.legend()
plt.grid(True)
plt.show()

print("Valor final de w (Newton):", w)
```




```
Valor final de w (Newton): 3.0
```

Interpretación: El método de Newton converge muy rápidamente (en este caso, en una o dos iteraciones) ya que la función es cuadrática. Este método es muy preciso cuando el costo de calcular la Hessiana es bajo.

Método Quasi-Newton (BFGS)

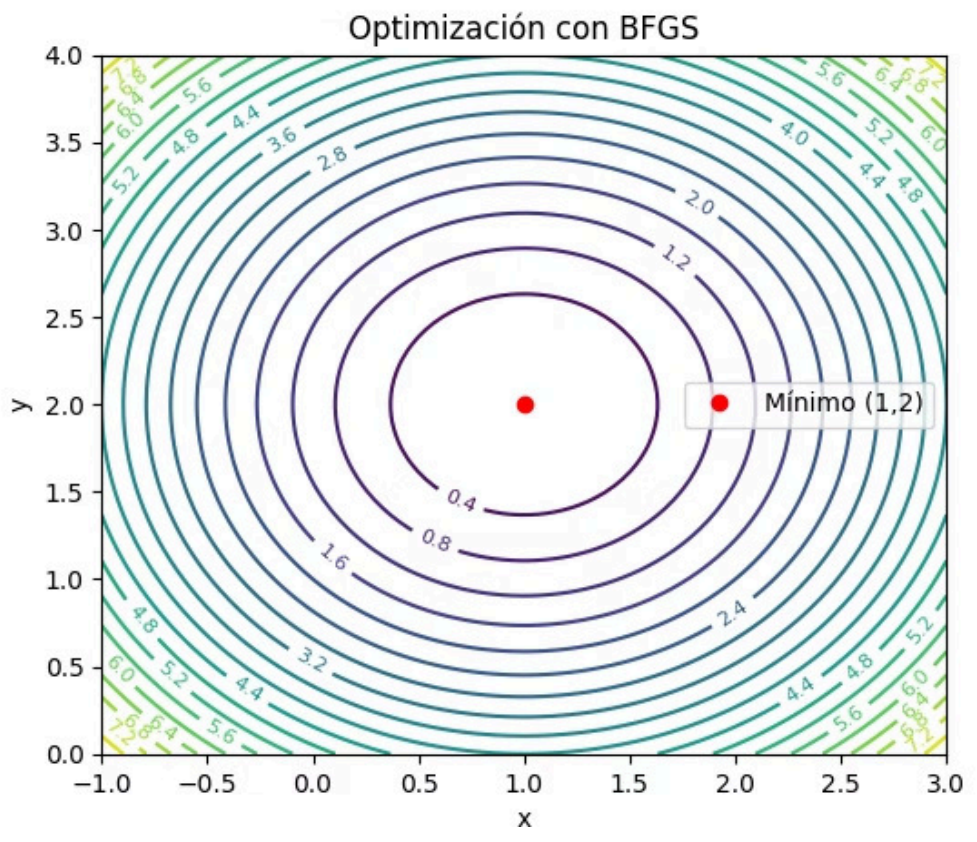
Utilizaremos el método BFGS de la biblioteca scipy.optimize.minimize para optimizar una función en dos variables. Considera la función:

$$f(x, y) = (x - 1)^2 + (y - 2)^2,$$

que tiene un mínimo en $(x, y) = (1, 2)$.

```python
import numpy as np
from scipy.optimize import minimize
import matplotlib.pyplot as plt

# Definir la función de costo en dos variables
def f_xy(params):
    x, y = params
    return (x - 1)**2 + (y - 2)**2

# Valor inicial aleatorio
initial_params = np.random.randn(2)

# Optimización con el método BFGS
result = minimize(f_xy, initial_params, method='BFGS')
print("Parámetros óptimos (BFGS):", result.x)
print("Valor mínimo de la función:", result.fun)

# Visualización 2D: contorno de la función y trayectoria del optimizador
x_vals = np.linspace(-1, 3, 100)
y_vals = np.linspace(0, 4, 100)
X, Y = np.meshgrid(x_vals, y_vals)
Z = (X - 1)**2 + (Y - 2)**2

plt.figure(figsize=(6,5))
contours = plt.contour(X, Y, Z, levels=20, cmap='viridis')
plt.clabel(contours, inline=True, fontsize=8)
plt.plot(result.x[0], result.x[1], 'ro', label='Mínimo (1,2)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Optimización con BFGS')
plt.legend()
plt.show()
```

```
Parámetros óptimos (BFGS): [0.99999999 1.99999999]
Valor mínimo de la función: 1.1472307348878504e-16
```



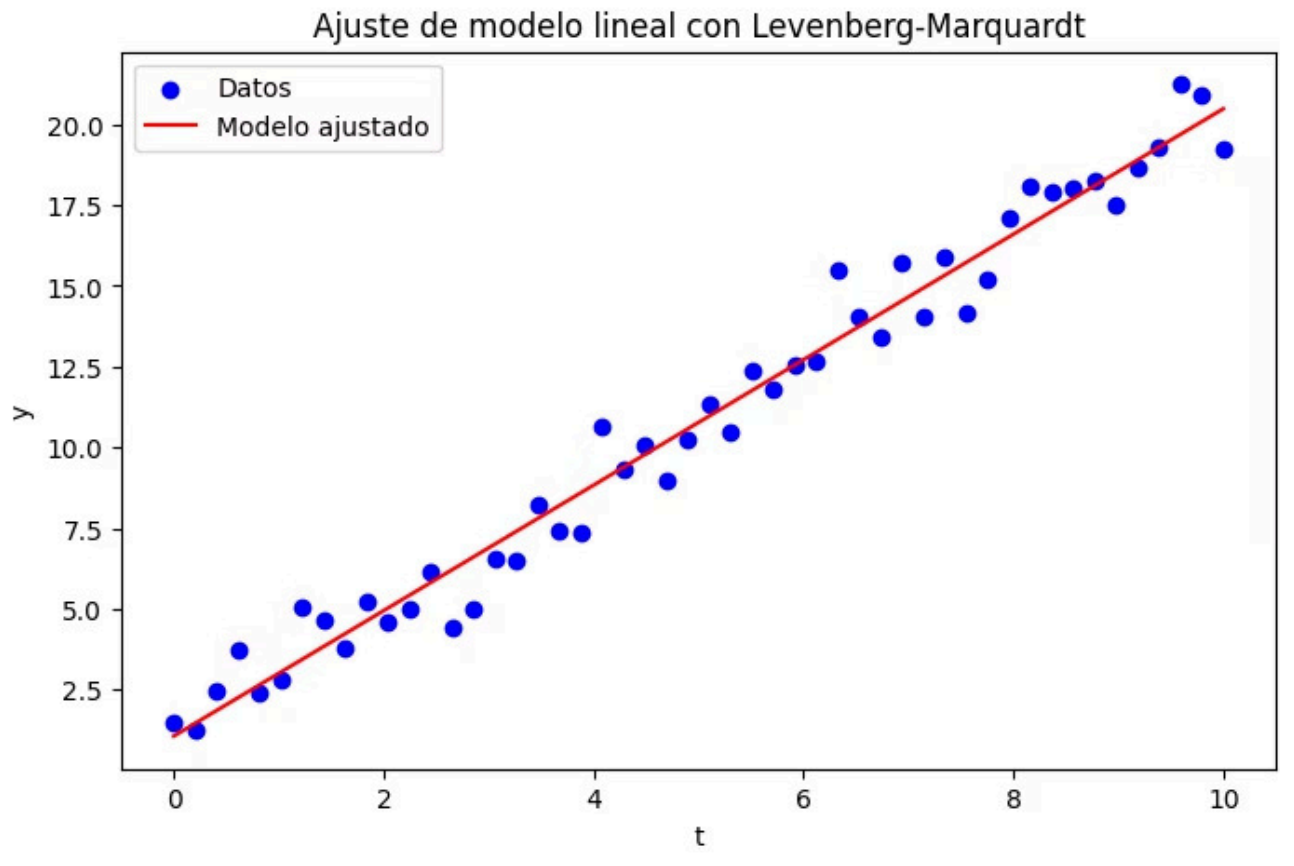
Interpretación: BFGS aproxima la Hessiana y suele ser efectivo para problemas de dimensión moderada. En este ejemplo, encuentra el mínimo de la función de forma precisa.

Método de Levenberg-Marquardt

El Método de Levenberg-Marquardt es muy utilizado en problemas de ajuste de curvas. En este ejemplo, ajustaremos un modelo lineal y = at + b a datos sintéticos. Se usará scipy.optimize.least_squares con el método 'lm' para emplear Levenberg-Marquardt.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import least_squares

# Generar datos sintéticos
np.random.seed(42)
t = np.linspace(0, 10, 50)
true_a, true_b = 2.0, 1.0
y_true = true_a * t + true_b
noise = np.random.normal(0, 1, t.shape)
y_data = y_true + noise

# Modelo: y = a*t + b
def model(params, t):
    a, b = params
    return a * t + b

# Función residual: diferencia entre datos observados y modelo
def residuals(params, t, y):
    return model(params, t) - y

# Valor inicial para los parámetros
initial_params = np.array([0.0, 0.0])

# Optimización con Levenberg-Marquardt
result = least_squares(residuals, initial_params, args=(t, y_data), method='lm')
a_opt, b_opt = result.x
print("Parámetros óptimos (LM): a =", a_opt, ", b =", b_opt)

# Visualización del ajuste
plt.figure(figsize=(8,5))
plt.scatter(t, y_data, label='Datos', color='blue')
plt.plot(t, model(result.x, t), label='Modelo ajustado', color='red')
plt.xlabel('t')
plt.ylabel('y')
plt.title('Ajuste de modelo lineal con Levenberg-Marquardt')
plt.legend()
plt.show()
```

```
Parámetros óptimos (LM): a = 1.9420165984712758 , b = 1.064443102989749
```



Interpretación: Levenberg-Marquardt es especialmente adecuado para problemas de ajuste de curvas y se utiliza para minimizar la suma de los cuadrados de los residuos. En este ejemplo, se ajusta un modelo lineal a datos ruidosos y se obtienen los parámetros óptimos.

Métodos de Segundo Orden vs. Métodos de Primer Orden

Cuándo preferir métodos de segundo orden:

Curvatura y Precisión

Cuando la función de costo presenta una curvatura pronunciada o compleja, los métodos de segundo orden son muy útiles. Estos métodos, como el algoritmo de Newton o BFGS, utilizan la matriz Hessiana (la cual contiene las segundas derivadas) para ajustar de forma adaptativa la magnitud y dirección de los pasos de actualización. Esto permite una convergencia más rápida y precisa, ya que tienen en cuenta cómo varía la pendiente a lo largo del espacio de parámetros.

Problemas con Parámetros Moderados

En modelos donde el número de parámetros no es excesivamente grande, el costo computacional de calcular e invertir la Hessiana es manejable. En estos casos, obtener actualizaciones más precisas compensa el esfuerzo adicional, haciendo que los métodos de segundo orden sean preferibles.

Por qué preferir métodos de segundo orden:

Convergencia Más Rápida

Al incorporar información de la curvatura de la función de costo, estos métodos pueden dar pasos de actualización que se ajustan mejor a la forma del paisaje de la función. Esto significa que, en teoría, requieren menos iteraciones para alcanzar la convergencia comparado con los métodos de primer orden.

Robustez Frente a Problemas de Condicionamiento

En problemas donde el gradiente puede no ser suficiente para determinar una dirección óptima debido a pendientes muy desiguales (condicionamiento pobre), la Hessiana ayuda a "normalizar" la actualización, haciendo el proceso de optimización más robusto.

Limitaciones de los Métodos de Segundo Orden

Costo Computacional

Calcular la Hessiana y su inversa puede ser muy costoso en términos de memoria y tiempo, especialmente cuando se trabaja con modelos de alta dimensión (muchos parámetros). En estos casos, los métodos de primer orden, como el descenso de gradiente o sus variantes estocásticas, son preferibles porque escalan mejor y requieren menos recursos.

Escalabilidad

Para grandes conjuntos de datos o modelos complejos (como las redes neuronales profundas), los métodos de primer orden son más prácticos, ya que pueden procesar actualizaciones en lotes pequeños (minibatch SGD) sin la necesidad de computar la matriz Hessiana completa.

Resumen:

Métodos de Segundo Orden

Se prefieren cuando se necesita una convergencia rápida y precisa en problemas con una cantidad moderada de parámetros y cuando la función de costo tiene una curvatura significativa que se puede explotar.

Métodos de Primer Orden

Son la opción más adecuada para problemas de gran escala, donde la simplicidad computacional y la escalabilidad son críticas, a pesar de que puedan requerir más iteraciones para converger.

En el contexto de Machine Learning, elegir entre métodos de primer o segundo orden depende del balance entre precisión y recursos computacionales disponibles, así como de la naturaleza del problema a resolver.

Relevancia en Machine Learning

En Aprendizaje de Máquina, los algoritmos se entrenan ajustando sus parámetros para minimizar una función de costo (o pérdida), que cuantifica el error entre las predicciones del modelo y los datos reales. La optimización consiste en encontrar el conjunto de parámetros que produzca el error mínimo, lo que se logra utilizando técnicas de optimización.

Métodos Utilizados:

Descenso de Gradiente (GD)

Es la técnica más común, en la que se calcula el gradiente (vector de derivadas parciales) de la función de costo y se actualizan los parámetros en la dirección opuesta a este gradiente.

Variantes:

Descenso de Gradiente Estocástico (SGD)

Actualiza los parámetros utilizando solo uno o un pequeño lote de ejemplos, lo que resulta en una convergencia más rápida en grandes conjuntos de datos.

Mini-Batch GD

Combina lo mejor del GD y SGD, utilizando pequeños lotes de datos para actualizar los parámetros, reduciendo la varianza en la estimación del gradiente.

Optimizadores Adaptativos (Adam, RMSprop, Adagrad)

Ajustan la tasa de aprendizaje de forma individual para cada parámetro, haciendo el proceso más robusto y eficiente en escenarios de alta dimensionalidad.

Beneficios en la Optimización de Algoritmos

- Permiten entrenar modelos de regresión, clasificación y otros algoritmos de Machine Learning de forma iterativa.
- Facilitan la adaptación a funciones de costo complejas que pueden tener múltiples mínimos locales, puntos de silla o regiones planas.
- Mejoran la velocidad y estabilidad de la convergencia del entrenamiento, especialmente cuando se usan variantes adaptativas.

Ejemplo

El siguiente código ilustra el uso del Descenso de Gradiente para minimizar una función cuadrática $f(w)=(w-3)^2$, que simboliza un caso sencillo de optimización:

```python
import numpy as np
import matplotlib.pyplot as plt

def f(w):
    return (w - 3)**2

def grad_f(w):
    return 2 * (w - 3)

# Inicialización
w = np.random.randn()  # Valor inicial aleatorio
learning_rate = 0.1
num_iter = 20
w_hist = []
cost_hist = []

for i in range(num_iter):
    grad = grad_f(w)
    w = w - learning_rate * grad  # Actualización de parámetros
    w_hist.append(w)
    cost_hist.append(f(w))

# Visualización
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(range(num_iter), cost_hist, marker='o')
plt.xlabel('Iteración')
plt.ylabel('Costo')
plt.title('Evolución del Costo')
plt.subplot(1,2,2)
w_range = np.linspace(-2, 8, 100)
plt.plot(w_range, (w_range - 3)**2, label='f(w)')
plt.plot(w_hist, cost_hist, 'ro-', label='Trayectoria GD')
plt.xlabel('w')
plt.ylabel('Costo')
plt.title('Trayectoria de Descenso de Gradiente')
plt.legend()
plt.tight_layout()
plt.show()

print("Valor final de w:", w)
```

Aplicación en Redes Neuronales

En las redes neuronales, el proceso de entrenamiento consiste en ajustar los pesos y sesgos de cada capa para minimizar la función de costo. Este ajuste se realiza mediante algoritmos de optimización que utilizan la información del gradiente, calculado a través del proceso de backpropagation.

Backpropagation

Utiliza la regla de la cadena para propagar el error desde la capa de salida hasta las capas anteriores, calculando los gradientes de la función de costo respecto a cada parámetro.

Desafíos:

- Las redes profundas tienen un gran número de parámetros, lo que hace que el cálculo y almacenamiento del gradiente sea intensivo.
- La complejidad del paisaje de la función de costo puede causar problemas de convergencia, como quedarse atrapado en mínimos locales o sufrir de gradientes que se desvanecen o explotan.

Optimizadores en Redes Neuronales:

- SGD y Mini-Batch SGD: Se utilizan comúnmente en redes neuronales debido a su eficiencia en el manejo de grandes conjuntos de datos.
- Optimizadores Adaptativos (Adam, RMSprop, etc.): Son ampliamente preferidos en la práctica, ya que ajustan la tasa de aprendizaje de forma individual para cada parámetro, mejorando la estabilidad y acelerando la convergencia del entrenamiento.

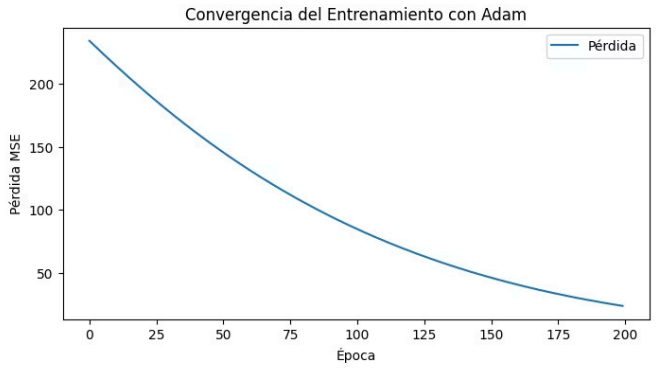
Ejemplo: Uso de Adam en PyTorch

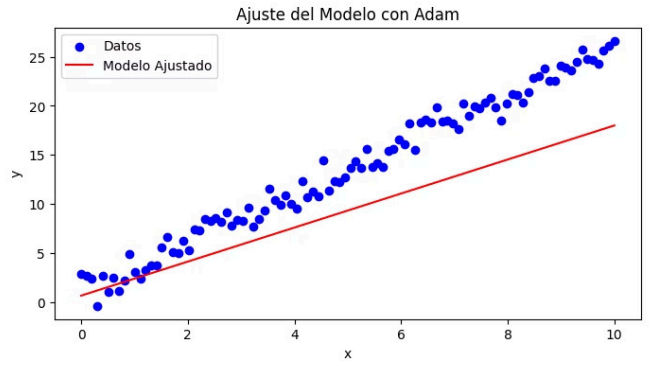
El siguiente código ejemplifica el uso de Adam para entrenar una red neuronal simple en PyTorch, aplicada a un problema de clasificación o regresión básico:

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Definir un modelo simple: una red neuronal de una capa
class SimpleNN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SimpleNN, self).__init__()
        self.fc = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        return self.fc(x)

# Datos sintéticos: ejemplo de regresión
torch.manual_seed(42)
# 100 muestras, 1 característica
x_train = torch.linspace(0, 10, 100).unsqueeze(1)
# Modelo lineal con ruido
y_train = 2.5 * x_train + 1.0 + torch.randn(x_train.size()) * 1.0

# Instanciar el modelo
model = SimpleNN(input_dim=1, output_dim=1)

# Definir la función de pérdida y el optimizador (Adam)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Entrenamiento del modelo
num_epochs = 200
loss_history = []

for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(x_train)
    loss = criterion(outputs, y_train)
    loss.backward()  # Backpropagation
    optimizer.step()  # Actualización de parámetros
    loss_history.append(loss.item())
    if (epoch+1) % 20 == 0:
        print(f"Epoch [{epoch+1}/{num_epochs}], Pérdida: {loss.item():.4f}")
```

```python
# Visualización del proceso de entrenamiento
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 4))
plt.plot(loss_history, label='Pérdida')
plt.xlabel('Época')
plt.ylabel('Pérdida MSE')
plt.title('Convergencia del Entrenamiento con Adam')
plt.legend()
plt.show()

# Visualización del ajuste del modelo
model.eval()

with torch.no_grad():
    predicted = model(x_train).detach().numpy()

plt.figure(figsize=(8, 4))
plt.scatter(x_train.numpy(), y_train.numpy(), label='Datos', color='blue')
plt.plot(x_train.numpy(), predicted, label='Modelo Ajustado', color='red')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Ajuste del Modelo con Adam')
plt.legend()
plt.show()
```

```
Epoch [20/200], Pérdida: 197.0507
Epoch [40/200], Pérdida: 162.5380
Epoch [60/200], Pérdida: 132.6837
Epoch [80/200], Pérdida: 107.2368
Epoch [100/200], Pérdida: 85.7952
Epoch [120/200], Pérdida: 67.9301
Epoch [140/200], Pérdida: 53.2174
Epoch [160/200], Pérdida: 41.2488
Epoch [180/200], Pérdida: 31.6370
Epoch [200/200], Pérdida: 24.0207
```





Interpretación:

- Backpropagation y Adam: Se utiliza el optimizador Adam para actualizar los parámetros de la red neuronal.
- Visualización: Se muestran tanto la evolución de la pérdida durante el entrenamiento como el ajuste final del modelo sobre los datos sintéticos.

Actividad Práctica Guiada: Implementación del descenso de gradiante en Python

Objetivo:

Implementar y comparar el Descenso de Gradiente básico y el Descenso de Gradiente Estocástico (SGD) en un problema sencillo de regresión lineal con datos simulados.

Paso a Paso (Sugerencia):

Generar Datos Sintéticos

- Crear un conjunto de puntos (xi, yi) con algo de ruido.
- Suponer un modelo y ≈ wx + b.

Definir la Función de Costo (MSE)

$$J(w,b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (wx_i + b))^2.$$

Implementar Descenso de Gradiente Básico

- Calcular el gradiente de J(w,b) con respecto a w y b.
- Actualizar los parámetros de forma iterativa.

Implementar SGD

Dividir los datos en lotes (batches) pequeños y actualizar w y b tras cada lote.

Comparar Rendimiento

- Observar la convergencia y el número de iteraciones requeridas para cada método.
- Graficar la evolución del costo y la trayectoria de los parámetros.

A continuación, se muestra un ejemplo completo en Python que implementa la actividad práctica guiada para comparar el Descenso de Gradiente Básico y el Descenso de Gradiente Estocástico (SGD) en un problema sencillo de regresión lineal utilizando datos simulados.

```python
import numpy as np
import matplotlib.pyplot as plt

# --------------------------------
# Paso 1: Generar Datos Sintéticos
# --------------------------------
# Fijar semilla para reproducibilidad
np.random.seed(42)

# Generar 100 puntos de datos para x entre 0 y 10
N = 100
x = np.linspace(0, 10, N)

# Definir el modelo real: y = w*x + b con w = 2.5, b = 1.0, y agregar ruido normal
true_w, true_b = 2.5, 1.0
noise = np.random.normal(0, 1, N)
y = true_w * x + true_b + noise

# --------------------------------
# Paso 2: Definir la Función de Costo (MSE) y sus Gradientes
# --------------------------------
def cost_function(w, b, x, y):
    """Calcula el Error Cuadrático Medio (MSE)"""
    N = len(x)
    return np.sum((y - (w*x + b))**2) / N

def gradients(w, b, x, y):
    """Calcula las derivadas parciales de la función de costo con respecto a w y b"""
    N = len(x)
    error = y - (w*x + b)
    grad_w = -2 * np.sum(x * error) / N
    grad_b = -2 * np.sum(error) / N
    return grad_w, grad_b
```

```python
# --------------------------------
# Paso 3: Implementar Descenso de Gradiente Básico (GD)
# --------------------------------
def gradient_descent(x, y, learning_rate=0.01, num_iters=200):
    w = 0.0  # Inicializar w
    b = 0.0  # Inicializar b
    cost_history = []
    w_history = []
    b_history = []

    for i in range(num_iters):
        grad_w, grad_b = gradients(w, b, x, y)
        w = w - learning_rate * grad_w
        b = b - learning_rate * grad_b
        current_cost = cost_function(w, b, x, y)
        cost_history.append(current_cost)
        w_history.append(w)
        b_history.append(b)

        # (Opcional) Imprimir progreso cada 50 iteraciones
        if (i+1) % 50 == 0:
            print(f"GD Iteración {i+1}: Costo = {current_cost:.4f}, w = {w:.4f}, b = {b:.4f}")

    return w, b, cost_history, w_history, b_history

# Ejecutar GD
w_gd, b_gd, cost_hist_gd, w_hist_gd, b_hist_gd = gradient_descent(x, y)
```

```python
# --------------------------------
# Paso 5: Comparar Rendimiento
# --------------------------------
plt.figure(figsize=(12, 5))

# Evolución del costo para GD y SGD
plt.subplot(1,2,1)
plt.plot(cost_hist_gd, label='GD')
plt.plot(cost_hist_sgd, label='SGD')
plt.xlabel('Iteración / Época')
plt.ylabel('Costo MSE')
plt.title('Evolución del Costo')
plt.legend()
plt.grid(True)

# Trayectoria de los parámetros en el espacio (w, b)
plt.subplot(1,2,2)
plt.plot(w_hist_gd, label='w (GD)', marker='o')
plt.plot(b_hist_gd, label='b (GD)', marker='o')
plt.plot(w_hist_sgd, label='w (SGD)', marker='x')
plt.plot(b_hist_sgd, label='b (SGD)', marker='x')
plt.xlabel('Iteración / Época')
plt.ylabel('Valor del Parámetro')
plt.title('Trayectoria de los Parámetros')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

# Mostrar parámetros finales
print("Parámetros finales con GD: w =", w_gd, ", b =", b_gd)
print("Parámetros finales con SGD: w =", w_sgd, ", b =", b_sgd)
```

```
GD Iteración 50: Costo = 0.8447, w = 2.5653, b = 0.4847
GD Iteración 100: Costo = 0.8330, w = 2.5539, b = 0.5601
GD Iteración 150: Costo = 0.8259, w = 2.5451, b = 0.6189
GD Iteración 200: Costo = 0.8216, w = 2.5382, b = 0.6648
SGD Época 10: Costo = 0.8685, w = 2.5639, b = 0.7559
SGD Época 20: Costo = 2.4630, w = 2.2863, b = 0.8656
SGD Época 30: Costo = 1.8403, w = 2.6895, b = 0.8219
SGD Época 40: Costo = 0.8171, w = 2.5203, b = 0.7518
SGD Época 50: Costo = 1.0605, w = 2.5937, b = 0.8652
```


```
Parámetros finales con GD: w = 2.538207099748425 , b = 0.664818146793009
Parámetros finales con SGD: w = 2.5936604063100246 , b = 0.8652473163377001
```

Explicación del Código

1 Generación de Datos Sintéticos:

- Se crean 100 puntos para x entre 0 y 10.
- Se genera y siguiendo un modelo lineal $y=2.5x+1.0$ y se añade ruido.

2 Definición de la Función de Costo (MSE) y Gradientes:

- La función cost_function calcula el Error Cuadrático Medio (MSE).
- La función gradients calcula las derivadas parciales de la función de costo con respecto a w y b.

3 Implementación de Descenso de Gradiente Básico (GD):

- Se actualizan w y b en cada iteración utilizando todo el conjunto de datos.
- Se guarda la evolución del costo y de los parámetros para visualización.

4 Implementación de SGD:

- Se mezcla el conjunto de datos y se procesan los ejemplos individualmente (o en pequeños lotes).
- Se actualizan w y b tras procesar cada mini-batch.
- Se guarda la evolución del costo por cada época.

5 Comparación de Rendimiento:

- Se grafican la evolución del costo y la trayectoria de los parámetros para comparar la convergencia de GD y SGD.
- Se imprimen los parámetros finales obtenidos con cada método.

Conclusiones sobre Métodos de Optimización

Importancia Fundamental

Los métodos de optimización son el corazón del aprendizaje automático, permitiendo que los modelos aprendan de los datos al ajustar sus parámetros para minimizar el error.

Equilibrio entre Precisión y Eficiencia

La elección entre métodos de primer orden (como SGD) y segundo orden (como Newton) depende del equilibrio entre precisión de convergencia y eficiencia computacional requerida para cada problema específico.

Adaptabilidad a Diferentes Escenarios

Los optimizadores adaptativos como Adam combinan las ventajas de varios métodos, ajustando automáticamente las tasas de aprendizaje y proporcionando soluciones robustas para una amplia gama de problemas.

Evolución Continua

El campo de la optimización en machine learning sigue evolucionando, con nuevos algoritmos que buscan mejorar la velocidad de convergencia, la estabilidad y la capacidad para manejar funciones de costo complejas.

Dominar estos métodos de optimización es esencial para cualquier profesional en el campo del aprendizaje automático, ya que proporcionan las herramientas necesarias para entrenar modelos efectivos y eficientes en una amplia variedad de aplicaciones.