deep-learning-i-esp

November 13, 2024

1 Introducción al Deep Learning y su Relación con la Ciencia de Datos

El **Deep Learning** es un campo avanzado dentro de la inteligencia artificial (IA) que se basa en el uso de redes neuronales profundas para modelar y resolver problemas complejos a partir de grandes volúmenes de datos. A través de estructuras en capas, el deep learning extrae patrones de datos sin necesidad de que los humanos definan manualmente las características o atributos clave, haciendo de esta una herramienta poderosa en la **ciencia de datos**.

En el contexto de la ciencia de datos, el deep learning permite abordar problemas en los que los datos son extensos, variados y a menudo no estructurados, como en el reconocimiento de imágenes, procesamiento de lenguaje natural y la predicción en series temporales. Los modelos de deep learning aprenden jerárquicamente, de modo que las primeras capas pueden detectar características simples (por ejemplo, bordes en una imagen), mientras que las capas posteriores construyen sobre estas características para identificar patrones de mayor nivel (por ejemplo, rostros o escenas completas).

A diferencia de los métodos tradicionales de aprendizaje automático, que requieren una ingeniería de características explícita, el deep learning permite que los datos brutos se transformen automáticamente en representaciones útiles para el modelo. Este enfoque facilita que los científicos de datos construyan modelos de alta precisión que identifican patrones complejos, mejorando la capacidad predictiva y adaptativa en una amplia variedad de aplicaciones.

Dado el papel central que el deep learning tiene en el análisis y extracción de valor a partir de datos, su éxito depende en gran medida de algoritmos de **optimización** eficientes, los cuales guían el proceso de aprendizaje ajustando los parámetros del modelo para minimizar los errores en las predicciones. A continuación, exploramos algunos de los conceptos fundamentales en deep learning, incluyendo el **descenso de gradiente** y variantes de optimización, que son clave en el aprendizaje de los modelos.

2 Deep Learning I: Conceptos Fundamentales

2.1 1. Descenso de Gradiente

El **descenso de gradiente** es uno de los métodos de optimización más importantes en deep learning, y su propósito es ajustar los parámetros de una red neuronal para reducir al mínimo una función de pérdida, es decir, la diferencia entre las predicciones del modelo y los valores reales. A continuación, revisamos algunas funciones de pérdida y variantes del descenso de gradiente que permiten a los modelos aprender de los datos.

2.1.1 1.1. ¿Cómo Aprender de los Datos?

Para que una red neuronal mejore su precisión, necesita una función de pérdida que mida el error de sus predicciones frente a los valores reales. Entre las funciones de pérdida más comunes están:

- **Square / Euclidean Loss**: mide el error cuadrático medio, útil en problemas de regresión.
- **Hinge / Margin Loss**: utilizada en máquinas de vectores soporte (SVM) para clasificación binaria.
- **Pérdida Logística**: usada en regresión logística para clasificación.
- **Sigmoid Cross-Entropy Loss**: común en clasificadores softmax para clasificación multinomial.
- **Descenso de Gradiente por Lotes** (Batch Gradient Descent): entrena el modelo usando el conjunto completo de datos en cada iteración.
- **Stochastic Gradient Descent (SGD)**: procesa cada muestra individualmente, permitiendo actualizaciones rápidas aunque ruidosas.
- **Mini-batch Gradient Descent**: mezcla los enfoques anteriores para lograr un balance entre velocidad y estabilidad en las actualizaciones.

2.1.2 1.2. Mini-batch Gradient Descent (Descenso del Gradiente por Mini-lotes)

El **mini-batch gradient descent** es una variación que divide el conjunto de datos en pequeños lotes, actualizando los parámetros de la red después de procesar cada lote. Este enfoque permite un entrenamiento eficiente y una convergencia estable.

2.2 2. Diferenciación Automática

La **diferenciación automática** permite calcular derivadas de manera precisa y eficiente en deep learning, facilitando el ajuste iterativo de los parámetros del modelo.

2.3 Fundamentos de Optimización

La optimización es una parte central del deep learning, ya que permite encontrar los mejores parámetros del modelo que minimicen la función de pérdida. Aunque los métodos basados en gradiente son los más utilizados, es útil conocer enfoques previos en optimización para entender su evolución y limitaciones.

2.3.1 Optimización Básica

Una estrategia inicial para minimizar una función $f(x)$ es **muestrear puntos cercanos** y dar pasos en dirección contraria al valor máximo. Sin embargo, este enfoque tiene limitaciones, ya que no puede acercarse al mínimo verdadero más allá del tamaño de paso seleccionado.

2.3.2 El Método Nelder-Mead

El método **Nelder-Mead** es una técnica de optimización que ajusta el tamaño de paso en función de la pérdida del nuevo punto calculado. Si el nuevo punto es mejor que los anteriores, el método expande el paso para acelerar el descenso; si es peor, reduce el paso para acercarse al mínimo. Usualmente, el paso se ajusta a la mitad al contraer y se duplica al expandir.

Este método es útil en problemas de optimización no diferenciables o de alta dimensionalidad, y se extiende a espacios de múltiples dimensiones añadiendo un punto adicional por cada dimensión.

En estos casos, el peor punto se reemplaza por un reflejo a través del centroide de los demás puntos. Si el nuevo punto es mejor que el anterior, se intenta una expansión exponencial en esa línea; si no, se reduce el paso hacia un punto más favorable.

Ver "Un Tutorial Interactivo sobre Optimización Numérica"

2.3.3 Respuesta a las Preguntas:

1. **¿Cuáles son las limitaciones de este método desde un punto de vista computacional?**

   El método Nelder-Mead es ineficiente en espacios de alta dimensionalidad porque el número de puntos requeridos crece con las dimensiones del problema. Esto incrementa el costo computacional y puede hacer que la convergencia sea lenta y menos estable, especialmente en funciones no convexas o con numerosos mínimos locales. Además, este método no utiliza derivadas, por lo que en problemas donde es posible calcular gradientes de manera eficiente, los métodos de gradiente suelen ser más rápidos y precisos.

2. **¿En qué casos es una alternativa real?**

   Nelder-Mead es útil cuando la función objetivo es no diferenciable, discontinua o cuando calcular derivadas es computacionalmente costoso. Es también una alternativa viable en problemas de baja a mediana dimensionalidad o cuando el espacio de búsqueda es ruidoso y presenta múltiples mínimos locales, ya que el método es menos susceptible a las fluctuaciones en la superficie de búsqueda.

2.4 3. Descenso de Gradiente

Para profundizar en el descenso de gradiente, consideremos una función $f(w) : \mathbf{R} \to \mathbf{R}$ y el objetivo de encontrar el valor w que minimiza la función. La derivada de f respecto a w, denotada como $f'(w)$ o $\frac{\delta f}{\delta w}$, es fundamental para determinar la dirección de descenso en el espacio de parámetros.

La derivada en el punto w se define como:

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

2.4.1 3.1. Primer Enfoque para Minimizar la Función

1. Iniciar con un valor aleatorio w^0.
2. Calcular la derivada $f'(w) = \lim_{h \to 0} \frac{f(w+h) - f(w)}{h}$.
3. Avanzar en pequeños pasos en la dirección opuesta de la derivada: $w^{i+1} = w^i - hf'(w^i)$.

La búsqueda del mínimo termina cuando la derivada se aproxima a cero, señal de que no hay más dirección descendente. En este punto, w se convierte en un **punto crítico**, que puede ser un máximo, mínimo o punto de silla.

Si f es una **función convexa**, la derivada cero indica un mínimo global. En otros casos, puede ser un mínimo/máximo local o un punto de silla.

[1]:
```
# numerical derivative at a point x by using finite differences
```

```
def f(x):
    return x**2

def fin_dif(x,
            f,
            h = 0.00001):
    '''
    This method returns the derivative of f at x
    by using the finite difference method
    '''
    return (f(x+h) - f(x))/h

x = 2.0
print("{:2.4f}".format(fin_dif(x,f)))
```

```
4.0000
```

NOTA: Se puede demostrar que la "fórmula de diferencia centrada" es mejor al calcular derivadas numéricas:

$$\lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

El error en la aproximación de "diferencia finita" se puede derivar del teorema de Taylor y, asumiendo que f es diferenciable, es $O(h)$. En el caso de la "diferencia centrada", el error es $O(h^2)$.

Hay dos problemas con las derivadas numéricas:

- Es aproximado.
- Es muy lento de evaluar (dos evaluaciones de función: $f(x+h), f(x-h)$).

¡Nuestros conocimientos de Cálculo podrían ayudar!

2.5 1.2. Segundo enfoque

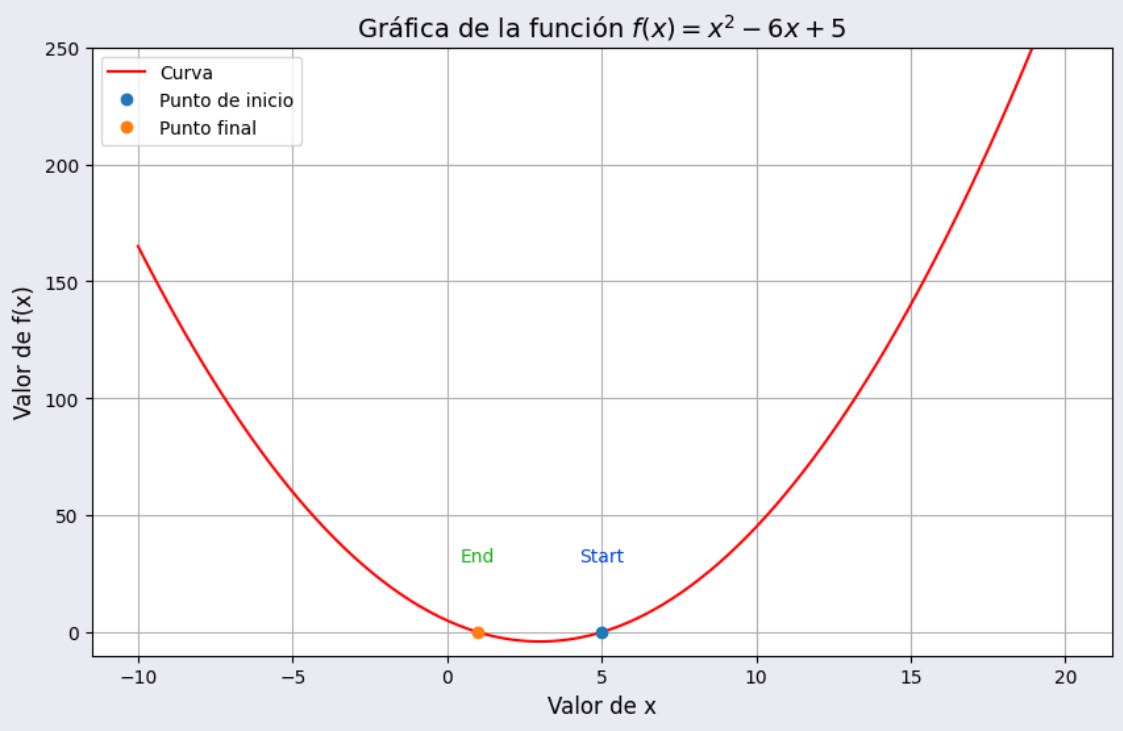
Para encontrar el mínimo local usando gradient descend y en el caso de conocer una expresión analítica de la derivada de la **función** que queremos minimizar, puedes comenzar en un punto aleatorio y moverte en la dirección de descenso más pronunciado en relación con la derivada:

- Comenzar desde un valor x aleatorio.
- Calcular la derivada $f'(x)$ analíticamente.
- Dar un pequeño paso en la dirección opuesta de la derivada.

[2]:
```
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
```

```
from scipy import stats

%matplotlib inline

# Datos
x = np.linspace(-10, 20, 100)
y = x**2 - 6*x + 5
start = 5

# Crear una única figura y ejes
fig, ax = plt.subplots(1, 1, figsize=(10, 6))  # Ajuste de tamaño de la figura
fig.set_facecolor('#EAEAF2')

# Graficar la curva
ax.plot(x, y, 'r-', label='Curva')  # Curva de la función

# Graficar el punto de inicio
ax.plot([start], [start**2 - 6*start + 5], 'o', label='Punto de inicio')  #␣
 ↪Punto de inicio
ax.text(start, start**2 - 6*start + 35, 'Start', ha='center', color=sns.
 ↪xkcd_rgb['blue'])

# Cálculo para el punto final
d = 2 * start - 6
end = start - d

# Graficar el punto final
ax.plot([end], [end**2 - 6*end + 5], 'o', label='Punto final')  # Punto final
ax.text(end, start**2 - 6*start + 35, 'End', ha='center', color=sns.
 ↪xkcd_rgb['green'])

# Ajustar el límite del eje Y
ax.set_ylim([-10, 250])

# Añadir la cuadrícula
ax.grid(True)

# Añadir los nombres de los ejes
ax.set_xlabel('Valor de x', fontsize=12)
ax.set_ylabel('Valor de f(x)', fontsize=12)

# Título de la gráfica
ax.set_title('Gráfica de la función $f(x) = x^2 - 6x + 5$', fontsize=14)

# Añadir la leyenda
ax.legend()
```

```
# Mostrar la figura
plt.show()
```



[3]:
```
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider, IntSlider
%matplotlib inline

# Función de costo
def cost_function(x):
    return x**2 - 6*x + 5

# Derivada numérica usando diferencias finitas
def numerical_derivative(x, func, h=0.00001):
    return (func(x + h) - func(x)) / h

# Método del gradiente descendente
def gradient_descent(start, func, derivative_func, learning_rate=0.1,␣
 ↪n_iterations=10):
    x = start
```

```
    trajectory = [x]
    for _ in range(n_iterations):
        gradient = derivative_func(x, func)
        x = x - learning_rate * gradient
        trajectory.append(x)
    return trajectory

# Función para actualizar la gráfica
def update_plot(start, learning_rate, n_iterations):
    x = np.linspace(-10, 20, 100)
    y = cost_function(x)

    # Crear la figura y los ejes
    fig, ax = plt.subplots(1, 1, figsize=(10, 6))
    fig.set_facecolor('#EAEAF2')

    # Graficar la función de costo
    ax.plot(x, y, 'r-', label='Función de costo')

    # Obtener la trayectoria del gradiente descendente
    trajectory = gradient_descent(start, cost_function, numerical_derivative,␣
 ↪learning_rate, int(n_iterations))

    # Graficar los puntos y las líneas de la trayectoria
    for i in range(len(trajectory) - 1):
        ax.plot([trajectory[i]], [cost_function(trajectory[i])], 'o',␣
 ↪color='blue')  # Puntos de la trayectoria
        ax.plot([trajectory[i], trajectory[i+1]],␣
 ↪[cost_function(trajectory[i]), cost_function(trajectory[i+1])], 'b-')  #␣
 ↪Líneas de trayectoria

    ax.plot([trajectory[-1]], [cost_function(trajectory[-1])], 'o',␣
 ↪color='blue')  # Dibuja el último punto

    # Configurar los límites y la cuadrícula
    ax.set_ylim([-10, 250])
    ax.grid(True)

    # Etiquetas de los ejes
    ax.set_xlabel('Valor de x', fontsize=12)
    ax.set_ylabel('Valor de la función de costo', fontsize=12)

    # Título de la gráfica
    ax.set_title('Gradiente Descendente: Evolución del Costo', fontsize=14)

    # Mostrar la gráfica
    plt.show()
```

```
# Crear los sliders interactivos
interact(update_plot,
         start=FloatSlider(value=15, min=-10, max=20, step=0.1),
         learning_rate=FloatSlider(value=0.1, min=0.01, max=1, step=0.01),
         n_iterations=IntSlider(value=10, min=1, max=50, step=1))
```

```
interactive(children=(FloatSlider(value=15.0, description='start', max=20.0,␣
↪min=-10.0), FloatSlider(value=0.1…
```

[3]:
```
<function __main__.update_plot(start, learning_rate, n_iterations)>
```

¡Hay un problema! ¿Cuál?

Necesitamos definir un *paso* adecuado para modular el valor del gradiente.

[4]:
```
old_min = 0
temp_min = 15
step_size = 0.01
precision = 0.01

def f(x):
    return x**2 - 6*x + 5

def f_derivative(x):
    import math
    return 2*x -6

mins = []
cost = []

while abs(temp_min - old_min) > precision:
    old_min = temp_min
    gradient = f_derivative(old_min)
    move = gradient * step_size
    temp_min = old_min - move
    cost.append((3-temp_min)**2)
    mins.append(temp_min)

# Redondeando el resultado a 2 dígitos debido al tamaño del paso
print("El mínimo local ocurre en {:3.2f}.".format(round(temp_min,2)))
```

```
El mínimo local ocurre en 3.48.
```

[5]:
```
import warnings
warnings.filterwarnings('ignore')

import numpy as np
```

```
import seaborn as sns
import matplotlib.pyplot as plt
from ipywidgets import interact, FloatSlider
%matplotlib inline

# Función de costo
def cost_function(x):
    return x**2 - 6*x + 5

# Derivada de la función
def cost_derivative(x):
    return 2*x - 6

# Método del gradiente descendente con trayectoria
def gradient_descent_with_trajectory(start, step_size, precision):
    old_min = 0
    temp_min = start
    mins = [temp_min]
    cost = [cost_function(temp_min)]

    while abs(temp_min - old_min) > precision:
        old_min = temp_min
        gradient = cost_derivative(old_min)
        move = gradient * step_size
        temp_min = old_min - move
        mins.append(temp_min)
        cost.append(cost_function(temp_min))

    # Imprimir el resultado
    print("El mínimo local ocurre en {:3.6f}.".format(round(temp_min, 2)))

    return mins, cost

# Función para actualizar la gráfica
def update_plot(start, step_size, precision):
    x = np.linspace(-10, 20, 100)
    y = cost_function(x)

    # Crear la figura y los ejes
    fig, ax = plt.subplots(1, 1, figsize=(10, 6))
    fig.set_facecolor('#EAEAF2')

    # Graficar la función de costo
    ax.plot(x, y, 'r-', label='Función de costo')

    # Obtener la trayectoria del gradiente descendente
```

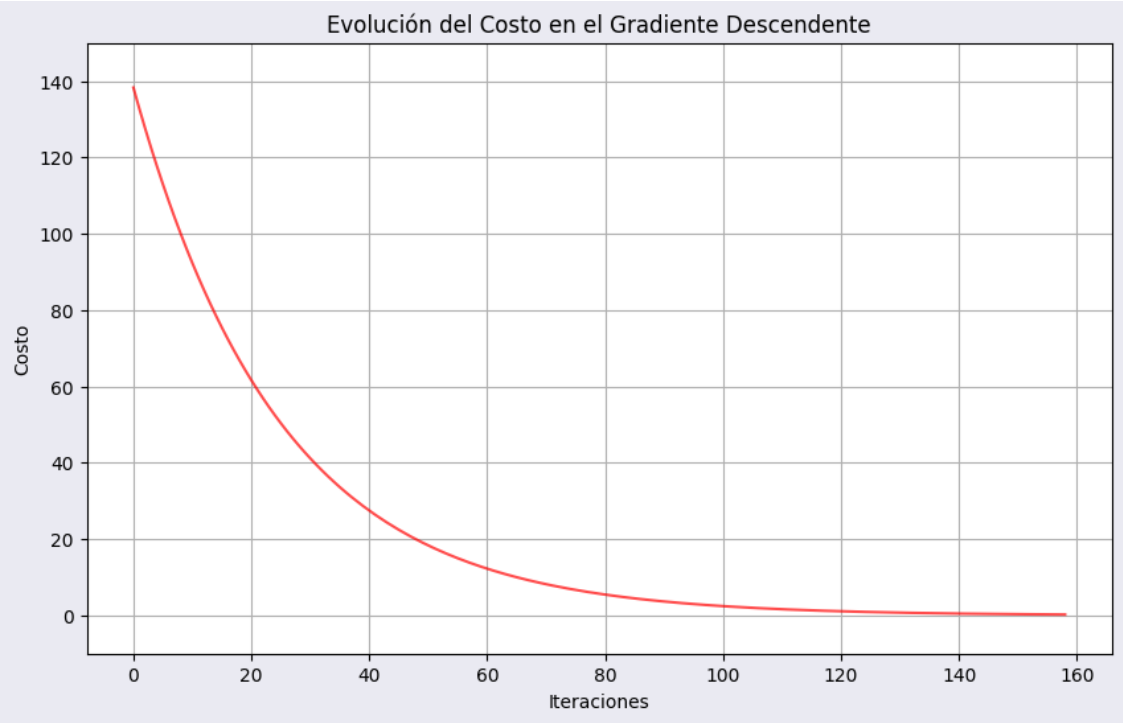
```
    trajectory, costs = gradient_descent_with_trajectory(start, step_size,␣
 ↪precision)

    # Graficar los puntos y las líneas de la trayectoria
    for i in range(len(trajectory) - 1):
        ax.plot([trajectory[i]], [cost_function(trajectory[i])], 'o',␣
 ↪color='blue')  # Puntos de la trayectoria
        ax.plot([trajectory[i], trajectory[i+1]],␣
 ↪[cost_function(trajectory[i]), cost_function(trajectory[i+1])], 'b-')  #␣
 ↪Líneas de trayectoria

    ax.plot([trajectory[-1]], [cost_function(trajectory[-1])], 'o',␣
 ↪color='blue')  # Dibuja el último punto

    # Configurar los límites y la cuadrícula
    ax.set_ylim([-10, 250])
    ax.grid(True)

    # Añadir título y etiquetas a los ejes
    ax.set_title('Evolución del Gradiente Descendente', fontsize=14)
    ax.set_xlabel('Valor de x', fontsize=12)
    ax.set_ylabel('Costo', fontsize=12)

    # Mostrar la gráfica
    plt.show()

# Crear los sliders interactivos
interact(update_plot,
         start=FloatSlider(value=15, min=-10, max=20, step=0.1),
         step_size=FloatSlider(value=0.1, min=0.01, max=1, step=0.01),
         precision=FloatSlider(value=0.001, min=0.0001, max=0.5, step=0.01))
```

```
interactive(children=(FloatSlider(value=15.0, description='start', max=20.0,␣
 ↪min=-10.0), FloatSlider(value=0.1…
```

[5]:
```
<function __main__.update_plot(start, step_size, precision)>
```

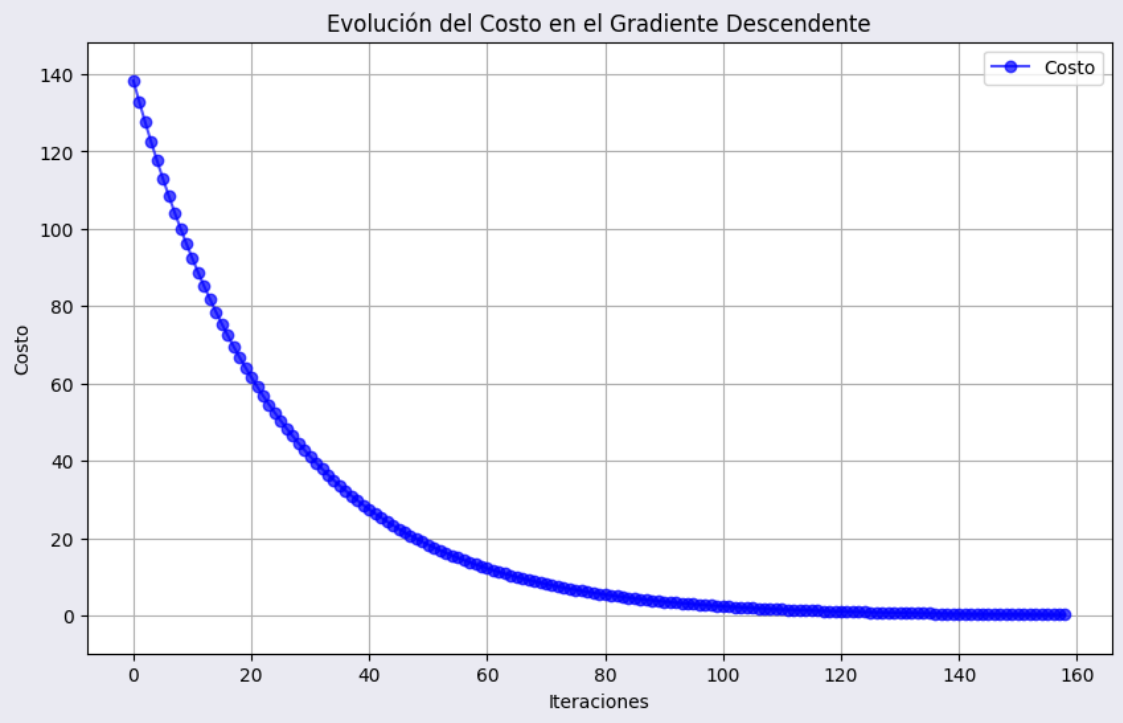
[6]:
```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 20, 100)

# Suponiendo que `cost` es una lista o arreglo de costos
x, y = (zip(*enumerate(cost)))

# Crear la figura y los ejes, estableciendo el tamaño de la figura
fig, ax = plt.subplots(figsize=(10, 6))  # Ajustamos el tamaño aquí directamente
```

```
fig.set_facecolor('#EAEAF2')

# Graficar los datos
ax.plot(x, y, 'r-', alpha=0.7)

# Configuración de límites y estética
ax.set_ylim([-10, 150])
ax.grid(True)

# Añadir título y etiquetas a los ejes
ax.set_xlabel('Iteraciones')
ax.set_ylabel('Costo')
ax.set_title('Evolución del Costo en el Gradiente Descendente')

# Mostrar la gráfica
plt.show()
```



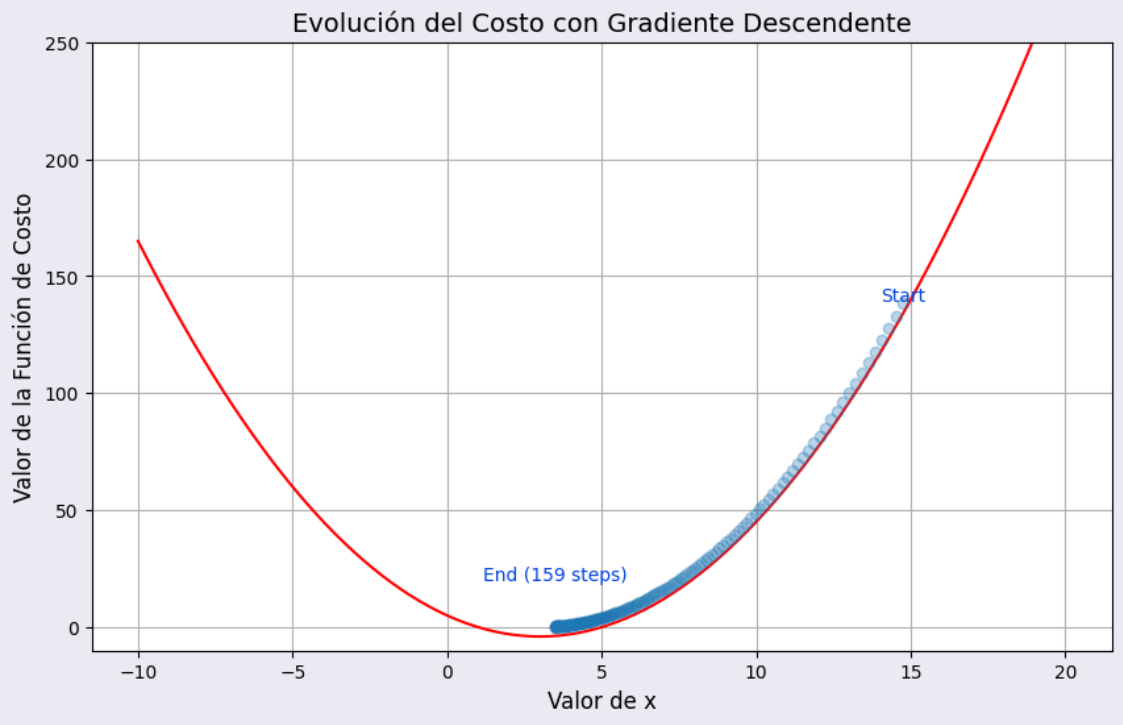
[7]:
```
import matplotlib.pyplot as plt
import numpy as np

def plot_cost(costs):
    # Crear un rango de iteraciones basado en la longitud de costs
    iterations = np.arange(len(costs))
```

```
    # Configurar la figura y los ejes, estableciendo el tamaño de la figura
    fig, ax = plt.subplots(figsize=(10, 6))  # Se ajusta el tamaño aquí␣
↪directamente
    fig.set_facecolor('#EAEAF2')

    # Graficar el costo a lo largo de las iteraciones
    ax.plot(iterations, costs, 'b-', marker='o', alpha=0.7, label='Costo')

    # Etiquetas y título
    ax.set_xlabel('Iteraciones')
    ax.set_ylabel('Costo')
    ax.set_title('Evolución del Costo en el Gradiente Descendente')
    ax.legend()

    # Configurar los límites y la estética
    ax.set_ylim([min(costs) - 10, max(costs) + 10])
    ax.grid(True)

    # Mostrar la gráfica
    plt.show()

# Supongamos que `costs` es la variable con el historial de costos obtenidos
plot_cost(cost)  # Asegúrate de tener la variable `cost` definida
```



[8]:
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Datos
x = np.linspace(-10, 20, 100)
y = x**2 - 6*x + 5

# Crear una figura y un conjunto de ejes
fig, ax = plt.subplots(figsize=(10, 6))
fig.set_facecolor('#EAEAF2')

# Graficar la función de costo
ax.plot(x, y, 'r-', label='Función de costo')

# Configurar los límites y la cuadrícula
ax.set_ylim([-10, 250])
ax.grid(True)

# Graficar los puntos de la trayectoria de gradiente (suponiendo que ya tienes␣
 ↪los valores de `mins` y `cost`)
ax.plot(mins, cost, 'o', alpha=0.3, label='Trayectoria')

# Desplazamiento de las etiquetas en Y
start_label_offset = 5  # Desplazamiento para la etiqueta "Start"
end_label_offset = 20   # Desplazamiento para la etiqueta "End"

# Etiqueta de "Start" en el primer valor de la trayectoria
ax.text(mins[0],
        cost_function(mins[0]) + start_label_offset,  # Desplazamos la etiqueta␣
 ↪en Y
        'Start',
        ha='center',
        color=sns.xkcd_rgb['blue'])

# Etiqueta de "End" en el último valor de la trayectoria
ax.text(mins[-1],
        cost[-1] + end_label_offset,  # Desplazamos la etiqueta en Y
        'End (%s steps)' % len(mins),
        ha='center',
        color=sns.xkcd_rgb['blue'])

# Añadir título y etiquetas a los ejes
ax.set_title('Evolución del Costo con Gradiente Descendente', fontsize=14)
ax.set_xlabel('Valor de x', fontsize=12)
ax.set_ylabel('Valor de la Función de Costo', fontsize=12)
```

```
# Mostrar la gráfica
plt.show()
```



Alpha El tamaño del paso, **alfa**, es un concepto delicado: ya que si es demasiado pequeño convergeremos lentamente a la solución, pero si es demasiado grande podemos divergir de la solución.

Hay varias políticas a seguir a la hora de seleccionar el tamaño del paso:

- Pasos de tamaño constante. En este caso, el tamaño de paso determina la precisión de la solución.
- Pasos de tamaño decreciente.
- En cada paso, seleccionar el paso óptimo.

La última política es buena, pero demasiado cara.

2.6 1.3. De las derivadas al gradiente: minimización de funciones n-dimensionales.

Consideremos una función n-dimensional $f : \mathbf{R}^n \to \mathbf{R}$. Por ejemplo:

$$f(\mathbf{x}) = \sum_n x_n^2$$

Nuestro objetivo es encontrar el argumento $\mathbf{x}$ que minimice esta función.

El gradiente de f es el vector cuyas componentes son las n derivadas parciales de f. Se trata pues de una función vectorial.

El gradiente apunta en la dirección de la mayor tasa de incremento de la función.

$$\nabla f = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})$$

[9]:
```
def f(x):
    return sum(x_i**2 for x_i in x)

def fin_dif_partial_centered(x,
                             f,
                             i,
                             h=1e-6):
    '''
    This method returns the partial derivative of the i-th
    component of f at x
    by using the centered finite difference method
    '''
    w1 = [x_j + (h if j==i else 0) for j, x_j in enumerate(x)]
    w2 = [x_j - (h if j==i else 0) for j, x_j in enumerate(x)]
    return (f(w1) - f(w2))/(2*h)

def gradient_centered(x,
                      f,
                      h=1e-6):
    '''
    This method returns the gradient vector of f at x
    by using the centered finite difference method
    '''
    return[round(fin_dif_partial_centered(x,f,i,h), 10) for i,_ in enumerate(x)]


x = [1.0,1.0,1.0]

print('{:.6f}'.format(f(x)), gradient_centered(x,f))
```

```
3.000000 [2.0000000001, 2.0000000001, 2.0000000001]
```

La función que hemos evaluado, $f(\mathbf{x}) = x_1^2 + x_2^2 + x_3^2$, es 3 en $(1, 1, 1)$ y el vector gradiente en este punto es $(2, 2, 2)$.

Entonces, podemos seguir estos pasos para maximizar (o minimizar) la función:

- Partir de un vector aleatorio $\mathbf{x}$.
- Calcular el vector gradiente.
- Dar un pequeño paso en la dirección opuesta al vector gradiente.

Es importante tener en cuenta que este cálculo del gradiente es muy costoso: si $\mathbf{x}$ tiene dimensión n, tenemos que evaluar f en $2 * n$ puntos.

3 2. Aprender de los Datos

El **Aprender de los Datos** es una disciplina científica que se ocupa del diseño y desarrollo de algoritmos que permiten a las computadoras inferir, a partir de los datos, un modelo que represente de forma compacta los datos crudos y/o que permita buenas capacidades de generalización. Existen dos enfoques principales en el aprendizaje de datos: **aprendizaje supervisado** y **aprendizaje no supervisado**.

En el aprendizaje supervisado, el objetivo es aprender un modelo que prediga un valor o clase a partir de un conjunto de datos etiquetados. En este caso, el modelo se ajusta para minimizar una función de costo que mide la discrepancia entre las predicciones del modelo y los valores reales. Este proceso se ve desde un punto de vista de **optimización**. El escenario común en el aprendizaje supervisado está compuesto por los siguientes elementos:

- Un conjunto de datos $(\mathbf{x}, y)$ de n ejemplos. Por ejemplo:
  - $\mathbf{x}$: características de un jugador de videojuegos; y: pagos mensuales.
  - $\mathbf{x}$: datos de sensores de un motor de automóvil; y: probabilidad de fallo del motor.
  - $\mathbf{x}$: datos financieros de un cliente de banco; y: calificación crediticia.

  Si y es un valor real, el problema es de **regresión**. Si y es binario o categórico, el problema es de **clasificación**.
- Una **función objetivo** $f_{(\mathbf{x},y)}(\mathbf{w})$, que se desea minimizar, representando la discrepancia entre el modelo y los datos observados.
- Un **modelo** M representado por un conjunto de parámetros $\mathbf{w}$.
- El **gradiente** de la función objetivo, denotado como $\nabla f_{(\mathbf{x},y)}(\mathbf{w})$, con respecto a los parámetros del modelo.

3.0.1 2.1. Función Objetivo en Regresión

Para problemas de **regresión**, la función objetivo busca minimizar la discrepancia entre el valor predicho por el modelo y el valor real. En este caso, la función de pérdida se define como el error cuadrático medio (ECM):

$$f_{(\mathbf{x},y)}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - M(\mathbf{x}_i, \mathbf{w}))^2$$

Donde: - n es el número total de ejemplos en el conjunto de datos. - y_i es el valor real de la i-ésima muestra. - $M(\mathbf{x}_i, \mathbf{w})$ es la predicción del modelo para la entrada $\mathbf{x}_i$ utilizando los parámetros $\mathbf{w}$.

El objetivo es encontrar los parámetros $\mathbf{w}$ que minimicen esta función.

3.0.2 2.2. Square / Euclidean Loss (Pérdida Cuadrática)

La **Square Loss** o **Euclidean Loss** es la función de pérdida más comúnmente utilizada en problemas de regresión. Se define como:

$$L(y, f(\mathbf{x})) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i))^2$$

Donde: - y_i es el valor real de la i-ésima muestra. - $f(\mathbf{x}_i)$ es la predicción del modelo para la entrada $\mathbf{x}_i$.

Esta fórmula mide la diferencia cuadrática entre el valor real y la predicción del modelo, lo cual penaliza más los errores grandes.

La **Square Loss** también puede ser utilizada en problemas de clasificación mediante una pequeña modificación:

$$L(y, f(\mathbf{x})) = \frac{1}{n} \sum_{i=1}^{n} (1 - y_i f(\mathbf{x}_i))^2$$

Aquí, y_i es un valor binario (en el caso de clasificación binaria) y $f(\mathbf{x}_i)$ es la salida del modelo.

3.0.3 2.3. Hinge / Margin Loss (Pérdida Hinge)

La **Hinge Loss** es comúnmente utilizada en **Máquinas de Vectores de Soporte (SVM)** para clasificación binaria. Se define como:

$$L(y, f(\mathbf{x})) = \frac{1}{n} \sum_{i=1}^{n} \max(0, 1 - y_i f(\mathbf{x}_i))$$

Donde: - y_i es la etiqueta binaria de la i-ésima muestra ($y_i \in \{-1, 1\}$). - $f(\mathbf{x}_i)$ es la predicción del modelo para la entrada $\mathbf{x}_i$.

La **Hinge Loss** penaliza las predicciones incorrectas que están cerca del margen de separación. La idea es que si el producto entre la etiqueta y_i y la predicción $f(\mathbf{x}_i)$ es menor que 1, entonces la función de pérdida aumenta.

3.0.4 2.4. Pérdida Logística (Regresión Logística)

La **pérdida logística** es utilizada en **regresión logística** y también se utiliza ampliamente en clasificación binaria. Se define como:

$$L(y, f(\mathbf{x})) = \frac{1}{n} \sum_{i=1}^{n} \log(1 + \exp(-y_i f(\mathbf{x}_i)))$$

Donde: - y_i es la etiqueta binaria de la i-ésima muestra. - $f(\mathbf{x}_i)$ es la salida del modelo para la entrada $\mathbf{x}_i$.

La **pérdida logística** se utiliza para problemas de clasificación binaria, ya que es continua y diferenciable, lo que permite la aplicación de métodos de optimización como el **descenso de gradiente**.

3.0.5 2.5. Sigmoid Cross-Entropy Loss (Pérdida de Entropía Cruzada Sigmoide)

La **entropía cruzada** es una función de pérdida utilizada en problemas de clasificación multiclase, especialmente en combinación con el clasificador **Softmax**. La fórmula general para la entropía cruzada es:

$$H(y, \hat{y}) = -\sum_j y_j \log \hat{y}_j$$

Donde: - y_j es la probabilidad real de la clase j. - $\hat{y}_j$ es la probabilidad predicha para la clase j por el modelo.

En el contexto de clasificación multiclase, el objetivo es minimizar la entropía cruzada para que la distribución de probabilidad predicha por el modelo se aproxime lo más posible a la distribución real de las clases.

La **entropía cruzada** se utiliza en combinación con la función **Softmax**, que convierte las salidas del modelo en una distribución de probabilidad sobre las clases posibles. La fórmula del **Softmax** es:

$$P(\mathbf{y}_i = j \mid \mathbf{x_i}) = \frac{e^{f_j(\mathbf{x_i})}}{\sum_k e^{f_k(\mathbf{x_i})}}$$

Donde: - $f_j(\mathbf{x_i})$ es la predicción para la clase j para la entrada $\mathbf{x_i}$. - La función Softmax normaliza las predicciones para que sumen 1, lo que las convierte en probabilidades.

Minimizar la entropía cruzada junto con Softmax resulta en un modelo que se ajusta de manera óptima para predecir las probabilidades correctas de las clases.

3.1 2.5. Descenso de gradiente por lotes (Batch Gradient Descend)

Podemos implementar **Gradient descend** de la siguiente manera (*descenso de gradiente por lotes*):

[10]:
```
import numpy as np
import random

# f = 2x
x = np.arange(10)
y = np.array([2*i for i in x])

# f_target = 1/n Sum (y - wx)**2
def target_f(x,y,w):
    return np.sum((y - x * w)**2.0) / x.size

# gradient_f = 2/n Sum 2wx**2 - 2xy
def gradient_f(x,y,w):
    return 2 * np.sum(2*w*(x**2) - 2*x*y) / x.size

def step(w,grad,alpha):
    return w - alpha * grad
```

```
def BGD(target_f,
        gradient_f,
        x,
        y,
        toler = 1e-6,
        alpha=0.01):
    '''
    Batch gradient descend by using a given step
    '''
    w = random.random()
    val = target_f(x,y,w)
    i = 0
    while True:
        i += 1
        gradient = gradient_f(x,y,w)
        next_w = step(w, gradient, alpha)
        next_val = target_f(x,y,next_w)
        if (abs(val - next_val) < toler):
            return w
        else:
            w, val = next_w, next_val

print('{:.6f}'.format(BGD(target_f, gradient_f, x, y)))
```

```
2.000091
```

3.2 2.6. Descenso de Gradiente Estocástico (SGD)

En el entrenamiento de modelos de deep learning, los parámetros de la red se ajustan para minimizar una **función de coste** que mide el error entre las predicciones del modelo y los valores reales en el conjunto de datos. En muchos casos, este ajuste se realiza con métodos de **descenso de gradiente** convencionales, que calculan el gradiente de la función de coste evaluando todas las muestras del conjunto de datos $(\mathbf{x}_i, y_i)$ en cada paso de actualización de parámetros.

Si el conjunto de datos es muy grande, esta estrategia se vuelve computacionalmente costosa, ya que implica recorrer todo el dataset para calcular cada paso de gradiente. En este contexto, utilizamos una estrategia llamada **Descenso de Gradiente Estocástico** o **SGD** (*Stochastic Gradient Descent*), que permite realizar actualizaciones de los parámetros basadas en una muestra aleatoria de los datos, reduciendo el costo computacional por iteración.

3.2.1 Funcionamiento de SGD

En el descenso de gradiente estocástico, la **función de coste** $J(\theta)$ es aditiva, ya que se calcula sumando los errores de reconstrucción de cada muestra del conjunto de datos. Para ilustrarlo, si usamos una función de coste cuadrática, se expresa como:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f(\mathbf{x}_i; \theta) \right)^2$$

donde N es el número total de muestras, y_i es el valor real de la i-ésima muestra, $\mathbf{x}_i$ representa sus características, θ son los parámetros del modelo, y $f(\mathbf{x}_i; \theta)$ es la predicción del modelo para esa muestra.

En SGD, en lugar de calcular el gradiente sobre el conjunto completo de datos, aproximamos el gradiente utilizando solo **una muestra aleatoria** (o en algunos casos, un pequeño subconjunto de muestras, denominado **mini-lote**):

$$\hat{\nabla} J(\theta) \approx \nabla J_i(\theta) = \frac{\partial}{\partial \theta} \left(y_i - f(\mathbf{x}_i; \theta) \right)^2$$

Donde $\nabla J_i(\theta)$ es el gradiente calculado para la i-ésima muestra. Luego, actualizamos los parámetros de la siguiente manera:

$$\theta := \theta - \eta \cdot \nabla J_i(\theta)$$

donde η es la tasa de aprendizaje, que controla el tamaño del paso en la dirección opuesta al gradiente de la función de pérdida.

3.2.2 Iteraciones y Épocas

Dado que en cada iteración de SGD se utiliza solo una muestra, el algoritmo necesita más iteraciones para ver todas las muestras en el conjunto de datos. Una **época** representa una iteración completa sobre todas las muestras del dataset, y en cada época las muestras se seleccionan en un orden aleatorio para evitar patrones en la actualización de los parámetros. En general, se repiten varias épocas para optimizar el modelo.

3.2.3 Ventajas y Convergencia de SGD

SGD presenta ventajas importantes, aunque también algunas compensaciones:

1. **Eficiencia Computacional**: Como solo requiere una muestra (o un mini-lote) en cada actualización, reduce considerablemente el costo computacional en comparación con el descenso de gradiente de lotes completos.
2. **Convergencia Estocástica**: Aunque cada paso de SGD sigue un gradiente aproximado (y, por lo tanto, es inexacto), el ruido introducido por la aleatoriedad permite que el algoritmo pueda "saltar" fuera de mínimos locales y explorar mejor el espacio de parámetros, lo que es particularmente útil en problemas no convexos.
3. **Teoría de Convergencia**: La convergencia de SGD ha sido analizada utilizando teorías de minimización convexa y de aproximación estocástica:
    - **En funciones convexas o pseudoconvexas** (donde cualquier mínimo local es también un mínimo global), SGD converge casi con seguridad a un mínimo global.
    - **En funciones no convexas** (que tienen múltiples mínimos locales), SGD converge casi con seguridad a un mínimo local.

4. **Velocidad de Convergencia**: Debido a que se basa en gradientes aproximados, la tasa de convergencia de SGD es más lenta en comparación con el descenso de gradiente de lotes completos. A medida que el algoritmo se aproxima al mínimo, los pasos pueden volverse ruidosos, lo que implica que las tasas de aprendizaje deben reducirse progresivamente para asegurar una convergencia más estable.

3.2.4 Desventajas y Estrategias de Ajuste

Algunas de las desventajas de SGD incluyen:

- **Convergencia más lenta** en comparación con métodos de gradiente en lotes completos.
- **Inestabilidad en la convergencia**, especialmente cerca del mínimo, debido a la variabilidad en el gradiente estocástico.

Para mejorar la eficiencia y estabilidad de SGD, se utilizan técnicas como: - **Reducción de la tasa de aprendizaje**: bajar gradualmente η a medida que avanza el entrenamiento. - **Uso de mini-lotes**: emplear pequeños subconjuntos de datos en lugar de una única muestra en cada iteración. - **Métodos avanzados**: como SGD con momentum, RMSprop o Adam, que adaptan la tasa de aprendizaje o incluyen un término de memoria para mejorar la estabilidad y acelerar la convergencia.

En resumen, el descenso de gradiente estocástico es una técnica de optimización poderosa y ampliamente utilizada en deep learning, particularmente en problemas de gran escala donde el descenso de gradiente convencional sería prohibitivo en términos computacionales.

[11]:
```python
import numpy as np
x = np.arange(10)
y = np.array([2*i for i in x])
data = list(zip(x,y))

for (x_i,y_i) in data:
    print('{:3d} {:3d}'.format(x_i,y_i))
print("")

def in_random_order(data):
    '''
    Random data generator
    '''
    import random
    indexes = [i for i,_ in enumerate(data)]
    random.shuffle(indexes)
    for i in indexes:
        yield data[i]

import numpy as np
import random

def SGD(target_f,
        gradient_f,
```

```
        x,
        y,
        toler = 1e-6,
        epochs=100,
        alpha_0=0.01):
    '''
    Stochastic gradient descend with automatic step adaptation (by
    reducing the step to its 95% when there are iterations with no increase)
    '''
    data = list(zip(x,y))
    w = random.random()
    alpha = alpha_0
    min_w, min_val = float('inf'), float('inf')
    epoch = 0
    iteration_no_increase = 0
    while epoch < epochs and iteration_no_increase < 100:
        val = target_f(x, y, w)
        if min_val - val > toler:
            min_w, min_val = w, val
            alpha = alpha_0
            iteration_no_increase = 0
        else:
            iteration_no_increase += 1
            alpha *= 0.95
        for x_i, y_i in list(in_random_order(data)):
            gradient_i = gradient_f(x_i, y_i, w)
            w = w - (alpha *  gradient_i)
        epoch += 1
    return min_w

print('w: {:.6f}'.format(SGD(target_f, gradient_f, x, y)))
```

```
 0   0
 1   2
 2   4
 3   6
 4   8
 5  10
 6  12
 7  14
 8  16
 9  18

w: 2.000000
```

4 3. Mini-batch Gradient Descend (Descenso del gradiente por mini-lotes)

En código, el descenso del gradiente por lotes general se ve algo así:

```
nb_epochs = 100
for i in range(nb_epochs):
    grad = evaluate_gradient(target_f, data, w)
    w = w - learning_rate * grad
```

Para un número predefinido de epochs, primero calculamos el vector gradiente de la función objetivo para todo el conjunto de datos con respecto a nuestro vector de parámetros.

Stochastic gradient descent (SGD) en cambio, realiza una actualización de los parámetros para cada ejemplo de entrenamiento y etiqueta:

```
nb_epochs = 100
for i in range(nb_epochs):
    np.random.shuffle(data)
    for sample in data:
        grad = evaluate_gradient(target_f, sample, w)
        w = w - learning_rate * grad
```

El **Mini-batch gradient descent** finalmente toma lo mejor de ambos mundos y realiza una actualización para cada minilote de n ejemplos de entrenamiento:

```
nb_epochs = 100
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    grad = evaluate_gradient(target_f, batch, w)
    w = w - learning_rate * grad
```

El Mini-batch SGD tiene la ventaja de que trabaja con una estimación del gradiente ligeramente menos ruidosa. Sin embargo, a medida que aumenta el tamaño del minilote, disminuye el número de actualizaciones realizadas por cada cálculo efectuado (al final se vuelve muy ineficiente, como el Batch Gradient Descend).

Existe un compromiso óptimo (en términos de eficiencia computacional) que puede variar en función de la distribución de los datos y de las particularidades de la clase de función considerada, así como de la forma en que se implementen los cálculos.

5 4. Diferenciación Automática

> El algoritmo de **backpropagation** (retropropagación) fue introducido originalmente en la década de 1970, pero su importancia no fue completamente apreciada hasta un famoso artículo de 1986 por David Rumelhart, Geoffrey Hinton y Ronald Williams. (Michael Nielsen en "Neural Networks and Deep Learning", http://neuralnetworksanddeeplearning.com/chap2.html).
>
> **backpropagation** es el algoritmo clave que hace que el entrenamiento de deep models sea computacionalmente viable. Para las redes neuronales modernas, puede hacer que

> el entrenamiento con gradient descend sea hasta diez millones de veces más rápido, en comparación con una naive implementation. Esa es la diferencia entre un modelo que tarda una semana en entrenarse y otro que tardaría 200,000 años. (Christopher Olah, 2016)

Hemos visto que, para optimizar nuestros modelos, necesitamos calcular la derivada de la función de pérdida respecto a todos los parámetros del modelo.

El cálculo de derivadas en modelos computacionales se aborda mediante cuatro métodos principales:

- trabajando manualmente las derivadas y codificando el resultado (como en el artículo original que describe la retropropagación);
- diferenciación numérica (usando aproximaciones de diferencia finita);
- diferenciación simbólica (usando manipulación de expresiones en software, como Sympy);
- y diferenciación automática (AD).

La **diferenciación automática** (AD) funciona aplicando sistemáticamente la **regla de la cadena** del cálculo diferencial a nivel del operador elemental.

Supongamos $ y = f(g(x)) $ nuestra función objetivo. En su forma básica, la regla de la cadena establece:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial g}\frac{\partial g}{\partial x}$$

o, si hay más de una variable g_i entre y y x (por ejemplo, si f es una función bidimensional como $f(g_1(x), g_2(x))$), entonces:

$$\frac{\partial y}{\partial x} = \sum_i \frac{\partial y}{\partial g_i}\frac{\partial g_i}{\partial x}$$

> Ver https://www.math.hmc.edu/calculus/tutorials/multichainrule/

Ahora, veamos cómo la AD permite la evaluación precisa de derivadas con machine precision, con solo un pequeño factor constante de sobrecarga.

En su descripción más básica, AD se basa en el hecho de que todos los cálculos numéricos son en última instancia composiciones de un conjunto finito de operaciones elementales para las cuales se conocen las derivadas.

Por ejemplo, consideremos el cálculo de la derivada de esta función, que representa un modelo de red neuronal de 1 capa:

$$f(x) = \frac{1}{1 + e^{-(w^T \cdot x + b)}}$$

Primero, escribamos cómo evaluar $f(x)$ a través de una secuencia de operaciones primitivas:

```
x = ?
f1 = w * x
f2 = f1 + b
f3 = -f2
```

```
f4 = 2.718281828459 ** f3
f5 = 1.0 + f4
f = 1.0/f5
```

El signo de interrogación indica que x es un valor que debe proporcionarse.

Este *programa* puede calcular el valor de x y también **poblar variables del programa**.

Podemos evaluar $\frac{\partial f}{\partial x}$ en algún x utilizando la regla de la cadena. Esto se llama *forward-mode differentiation*.

En nuestro caso:

[12]:
```
def f(x,w,b):
    f1 = w * x
    f2 = f1 + b
    f3 = -f2
    f4 = 2.718281828459 ** f3
    f5 = 1.0 + f4
    return 1.0/f5

def dfdx_forward(x, w, b):
    f1 = w * x
    p1 = w                              # p1 = df1/dx
    f2 = f1 + b
    p2 = p1 * 1.0                       # p2 = p1 * df2/df1
    f3 = -f2
    p3 = p2 * -1.0                      # p3 = p2 * df3/df2
    f4 = 2.718281828459 ** f3
    p4 = p3 * 2.718281828459 ** f3      # p4 = p3 * df4/df3
    f5 = 1.0 + f4
    p5 = p4 * 1.0                       # p5 = p4 * df5/df4
    f = 1.0/f5
    df = p5 * -1.0 / f5 ** 2.0          # df/dx = p5 * df/df5
    return f, df

der = (f(3+0.00001, 2, 1) - f(3, 2, 1))/0.00001

print("Value of the function at (3, 2, 1): ",f(3, 2, 1))
print("df/dx Derivative (fin diff) at (3, 2, 1): ",der)
print("df/dx Derivative (aut diff) at (3, 2, 1): ",dfdx_forward(3, 2, 1)[1])
```

```
Value of the function at (3, 2, 1):  0.9990889488055992
df/dx Derivative (fin diff) at (3, 2, 1):  0.0018204242002717306
df/dx Derivative (aut diff) at (3, 2, 1):  0.0018204423602438651
```

Es interesante observar que este *programa* puede derivarse automáticamente si tenemos acceso a **subrutinas que implementan las derivadas de funciones primitivas** (como $\exp(x)$ o $1/x$) y todas las variables intermedias se calculan en el orden correcto.

También es interesante señalar que AD permite la evaluación exacta de las derivadas a **machine**

precision, con sólo un pequeño factor constante de sobrecarga.

Forward differentiation es eficiente para funciones $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ con $n << m$ (sólo se necesitan $O(n)$ barridos).

Para los casos $n >> m$ se necesita una técnica diferente. Para ello, reescribiremos la regla de la cadena como:

$$\frac{parcial f}{parcial x} = \frac{parcial g}{parcial x}\frac{parcial f}{parcial g}$$

para propagar derivadas hacia atrás desde una salida dada. Esto se denomina *diferenciación en modo inverso*. El paso inverso comienza en el final (es decir, $\frac{\partial f}{\partial f} = 1$) y se propaga hacia atrás a todas las dependencias.

[13]:
```
def dfdx_backward(x, w, b):
    import numpy as np
    f1 = w * x
    f2 = f1 + b
    f3 = -f2
    f4 = 2.718281828459 ** f3
    f5 = 1.0 + f4
    f = 1.0/f5

    pf = 1.0                              # pf = df/df
    p5 = 1.0 * -1.0 / (f5 * f5) * pf      # p5 = pf * df/df5
    p4 = p5 * 1.0                         # p4 = p5 * df5/df4
    p3 = p4 * np.log(2.718281828459) \
          * 2.718281828459 ** f3          # p3 = p4 * df4/df3
    p2 = p3 * -1.0                        # p2 = p3 * df3/df2
    p1 = p2 * 1.0                         # p1 = p2 * df2/df1
    dfx = p1 * w                          # dfx = p1 * df1/dx
    return f, dfx

print("df/dx Derivative (aut diff) at (3, 2, 1): ",
      dfdx_backward(3, 2, 1)[1])
```

```
df/dx Derivative (aut diff) at (3, 2, 1):  0.0018204423602438348
```

Cualquier función compleja que pueda descomponerse en un conjunto de funciones elementales puede derivarse de forma automática, con machine precision, mediante este algoritmo.

¡Ya no necesitamos codificar derivadas complejas para aplicar SGD!