Capítulo 5
Teoría de la Aproximación

En numerosas situaciones prácticas dentro de la ciencia y la ingeniería, nos encontramos con conjuntos de datos discretos, representados como pares ordenados (x_i, y_i), que provienen de mediciones experimentales o simulaciones. Estos datos a menudo presentan cierta dispersión o ruido inherente al proceso de obtención. El objetivo fundamental de la aproximación por mínimos cuadrados es encontrar una función $f(x)$, perteneciente a una familia de funciones predefinida (como líneas rectas, polinomios, exponenciales, etc.) que mejor se ajuste a estos datos observados. La noción de "mejor se ajuste" se formaliza mediante la minimización de la suma de los cuadrados de las diferencias verticales entre los valores observados y_i y los valores correspondientes predichos por la función $f(x_i)$.

Formalmente, para cada punto de dato (x_i, y_i), definimos el error o residuo e_i como la diferencia entre el valor observado y_i y el valor estimado $f(x_i)$, es decir, $e_i = y_i - f(x_i)$. La técnica de mínimos cuadrados busca minimizar la función objetivo S, que se define como la suma de los cuadrados de estos errores para todos los puntos de datos considerados:

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

Al minimizar esta suma de cuadrados, buscamos la función $f(x)$ cuyos valores se encuentren, en promedio y en términos cuadráticos, lo más cerca posible de los valores observados y_i. El proceso para encontrar esta función óptima implica determinar los parámetros que definen la forma específica de $f(x)$ de tal manera que la función objetivo S alcance su valor mínimo.

Ejemplo

Supongamos que tenemos los siguientes puntos de datos:

x_i	y_i
1	2
2	3
3	3
4	4
5	5

Nuestro objetivo es encontrar una función lineal de la forma

$$f(x) = a_0 + a_1 x$$

que mejor se ajuste a estos puntos utilizando el método de mínimos cuadrados.

Las ecuaciones normales para para regresión lineal son las siguientes

$$n \cdot a_0 + a_1 \left(\sum x_i \right) = \sum y_i$$

$$a_0 \left(\sum x_i \right) + a_1 \left(\sum x_i^2 \right) = \sum x_i y_i$$

Primero calculamos las sumatorias necesarias a partir de nuestros datos (donde $n = 5$ es el número de puntos):

$$\sum x_i = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum y_i = 2 + 3 + 3 + 4 + 5 = 17$$

$$\begin{aligned} \sum x_i^2 &= 1^2 + 2^2 + 3^2 + 4^2 + 5^2 \\ &= 1 + 4 + 9 + 16 + 25 = 55 \end{aligned}$$

$$\begin{aligned} \sum x_i y_i &= (1 \times 2) + (2 \times 3) + (3 \times 3) + (4 \times 4) + (5 \times 5) \\ &= 2 + 6 + 9 + 16 + 25 = 58 \end{aligned}$$

Ahora, sustituimos estos valores en las ecuaciones normales:

1. $5a_0 + 15a_1 = 17$
2. $15a_0 + 55a_1 = 58$

Tenemos un sistema de dos ecuaciones lineales con dos incógnitas (a_0 y a_1). Podemos resolverlo utilizando métodos como sustitución o eliminación. Usemos el método de eliminación:

Multiplicamos la primera ecuación por -3.

$$-15a_0 - 45a_1 = -51$$

Ahora sumamos esta ecuación a la segunda ecuación:

$$\begin{aligned}(15a_0 + 55a_1) + (-15a_0 - 45a_1) &= 58 + (-51)\\ 10a_1 &= 7\\ a_1 &= \frac{7}{10} = 0.7\end{aligned}$$

Ahora sustituimos el valor de a_1 en la primera ecuación original para encontrar a_0 :

$$\begin{aligned}5a_0 + 15\,(0.7) &= 17\\ 5a_0 + 10.5 &= 17\\ 5a_0 &= 17 - 10.5\\ 5a_0 &= 6.5\\ a_0 &= \frac{6.5}{5} = 1.3\end{aligned}$$

Por lo tanto la función lineal que mejor se ajusta a los datos por mínimos cuadrados es:

$$f\,(x) = 1.3 + 0.7x$$

5.1 Implementación en Python

A continuación se presenta el código mediante el cual se resuelve el problema anterior

```
1 # Datos del ejemplo
2 x = np.array([1, 2, 3, 4, 5])
3 y = np.array([2, 3, 3, 4, 5])
4
```

```
# Calcular las sumatorias necesarias
n = len(x)
sum_x = np.sum(x)
sum_y = np.sum(y)
sum_x_cuadrado = np.sum(x**2)
sum_xy = np.sum(x * y)

# Formar el sistema de ecuaciones
A = np.array([[n, sum_x],
              [sum_x, sum_x_cuadrado]])

b = np.array([sum_y, sum_xy])

# Resolver el sistema de ecuaciones lineales
coeficientes = np.linalg.solve(A, b)
a_0 = coeficientes[0]
a_1 = coeficientes[1]

print(f"Coeficiente a_0 (Intercepto): {a_0:.3f}")
print(f"Coeficiente a_1 (Pendiente): {a_1:.3f}")

# Generar los puntos para graficar el ajuste lineal
x_dom = np.linspace(min(x), max(x), 100)
y_eval = a_0 + a_1 * x_dom

# Graficar los datos y la línea de ajuste
plt.plot(x_dom, y_eval, label=f"Ajuste lineal: y = {a_0
  :.3f} + {a_1:.3f}x")
plt.scatter(x, y, c='red', zorder=3, label="Datos del
  Problema")
plt.xlabel("Valores de $x$")
plt.ylabel("Valores de $y$")
plt.title("Aproximación por Mínimos Cuadrados (Regresión
   Lineal)")
plt.legend()
plt.grid(linestyle='--')

```

```
39 # Guardar la gráfica y mostrar
40 plt.savefig('../Teoria-Aproximacion/Imagenes/Regresion-
     Lineal.pdf')
41 plt.show()
```

La gráfica de la regresión lineal y los datos del problema es la siguiente:

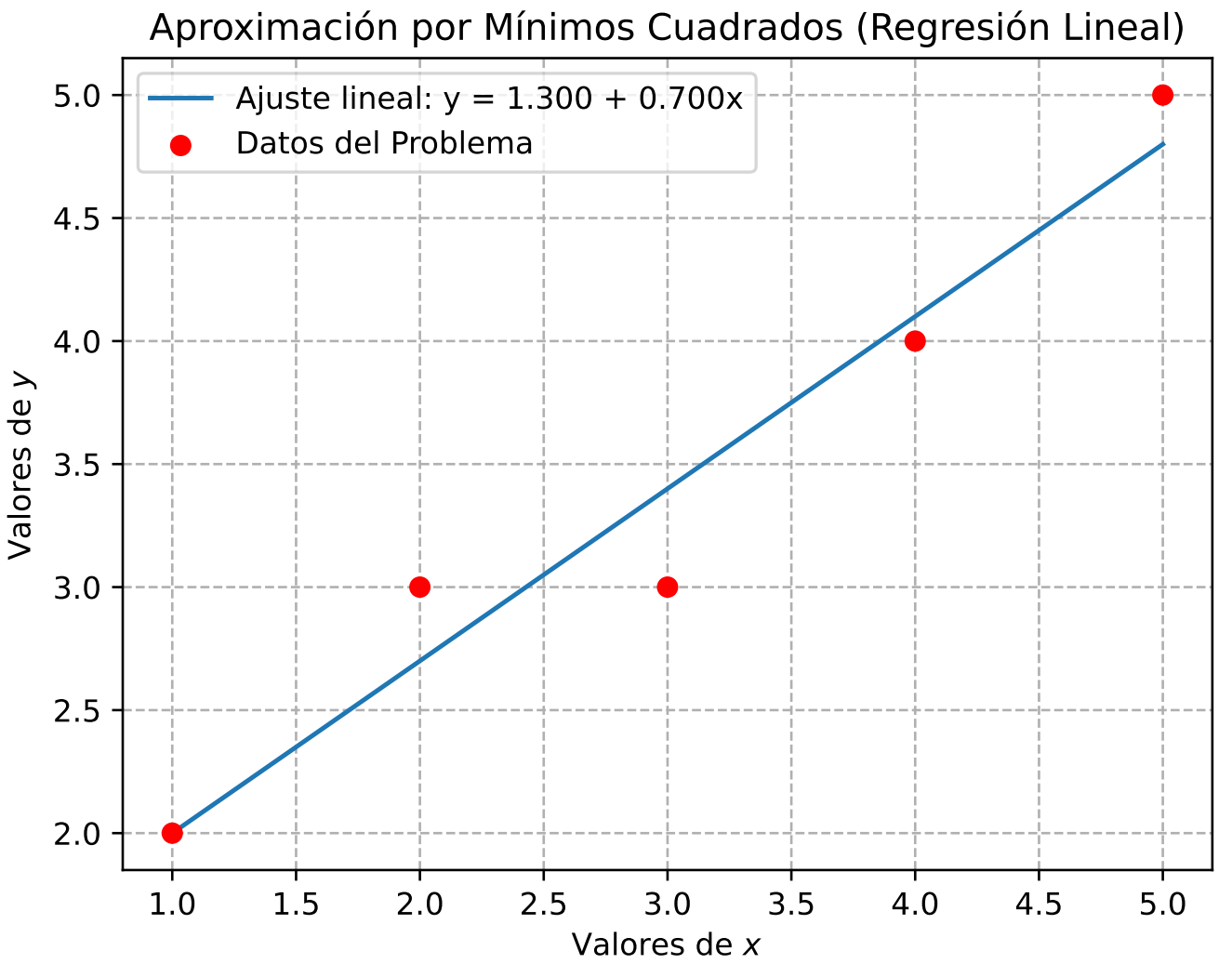

Figura 5.1: Regresión Lineal por Mínimos Cuadrados

5.2 Ajuste Lineal Mediante un Perceptron

En esta sección, exploraremos cómo un perceptrón simple puede ser utilizado para realizar un ajuste lineal a un conjunto de datos, similar al mpetodo de mínimos cuadrados que discutimos previamente. Si bien la motivación y enfoque difieren, entender esta conexión proporciona una base fundamental para comprender redes neuronales más complejas y sus capacidades de aprendizaje. Utilizaremos el mismo conjunto de datos del ejemplo anterior para comparar los resultados.

Un perceptrón es la forma más básica de procesamiento de una red neuronal que toma una o más entradas, las pondera, suma estas entradas ponderadas, les añade un sesgo (bias), y luego aplica una función de activación para producir una salida. Para realizar una regresión lineal, podemos considerar un perceptrón sin una función de activación no lineal en su salida. En este caso, la salida del perceptrón será una combinación lineal de sus entradas más el sesgo.

Considerando un solo atributo de entrada (x) y una salida deseada (y), nuestro perceptron para regresión lineal tendrá la siguiente forma

$$\hat{y} = w \cdot x + b$$

Donde

- $\hat{y}$ es la salida predicha del perceptrón (nuestra aproximación a y)
- x es la entrada
- w es el peso asociado a la entrada x
- b es el sesgo (bias), que actúa como el intercepto en nuestra función lineal.

Nuestro objetivo es encontrar los valores óptimos para el peso w y el sesgo b de manera que la salida del perceptrón $\hat{y}$ se aproxime lo más posible a los valores reales de y para nuestro conjunto de datos.

5.2.1 Aprendizaje del Perceptrón

El perceptrón aprende ajustando sus pesos y sesgos iterativamente basándose en el error entre sus predicciones y los valores reales. Una técnica común para este ajuste es el descenso del gradiente. Definimos una función de costo (similar a la suma de los cuadrados de los errores en mínimos cuadrados) que queremos minimizar. Para la regresión lineal, una función de costo común es el error cuadrático medio (MSE):

$$J\left(w, b\right) = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(w \cdot x_i + b - y_i\right)^2$$

El algoritmo de descenso del gradiente actualiza los pesos y el sesgo en dirección opuesta al gradiente de la función de costo con respecto

a esos parámetros:

$$w = w - \alpha \frac{\partial J}{\partial w}$$
$$b = b - \alpha \frac{\partial J}{\partial b}$$

Donde α es la tasa de aprendizaje, un hiperparámetro que controla el tamaño del paso en cada iteración.

Calculando las derivadas parciales de la función de costo se tiene:

$$\frac{\partial J}{\partial w} = \frac{2}{n} \sum_{i=1}^{n} (w \cdot x_i + b - y_i)\, x_i$$
$$\frac{\partial J}{\partial b} = \frac{2}{n} \sum_{i=1}^{n} (w \cdot x_i + b - y_i)$$

Ejemplo 1:

Supongamos que tenemos los siguientes puntos de datos:

x_i	y_i
1	2
2	3
3	3
4	4
5	5

Nuestro objetivo es encontrar una función lineal de la forma

$$f(x) = w \cdot x + b$$

que mejor se ajuste a estos puntos utilizando el perceptrón simple.

A continuación, implementaremos un perceptrón simple con descenso del gradiente en Python.

```
# Datos del ejemplo
x = np.array([1, 2, 3, 4, 5])
y = np.array([2, 3, 3, 4, 5])
n = len(x)

# Inicialización del peso y el sesgo
w = 0.0
b = 0.0
```

```
tasa_aprendizaje = 0.01

# Definir el número de iteraciones del aprendizaje
epocas = 1000

# Historial de la función de costo
historial = []

# Entrenamiento mediante el descenso del gradiente
for epoca in range(epocas):

    # Calcular las predicciones del perceptron
    y_ = w*x+b

    # Calcular el costo (MSE)
    costo = np.mean((y_ - y)**2)
    historial.append(costo)

    # Calcular los gradientes
    dw = (2/n) * np.sum((y_ - y) * x)
    db = (2/n) * np.sum(y_ - y)

    # Actualizar el peso y el sesgo
    w = w - tasa_aprendizaje * dw
    b = b - tasa_aprendizaje * db

# Imprimir el peso y el sesgo
print(f"Peso w aprendido: {w:.3f}")
print(f"Sesgo b aprendido: {b:.3f}")
print(f"La función aproximada por el perceptron es: f(x)
    = {b:.3f} + {w:.3f}x")

# Crear el dominio y rango de graficación
x_dom = np.linspace(min(x), max(x), 100)
y_eval = a_0 + a_1 * x_dom

# Graficar los datos y la línea de ajuste
```

```
45 plt.plot(x_dom, y_eval, label=f"Ajuste lineal: y = {a_0
     :.3f} + {a_1:.3f}x")
46 plt.scatter(x, y, c='red', zorder=3, label="Datos del
     Problema")
47 plt.xlabel("Valores de $x$")
48 plt.ylabel("Valores de $y$")
49 plt.title("Aproximación Aprendizaje del Perceptrón")
50 plt.legend()
51 plt.grid(linestyle='--')
52
53 # Guardar la gráfica y mostrar
54 plt.savefig('../Teoria-Aproximacion/Imagenes/Ajuste-
     Lineal-Perceptron.pdf')
55 plt.show()
56
57 # Gráfica de la función de costo durante el
     entrenamiento
58 plt.figure()
59 plt.plot(range(epocas), historial)
60 plt.xlabel('Época')
61 plt.ylabel('Costo (MSE)')
62 plt.title("Evolución del Costo durante el Entrenamiento
     del Perceptrón")
63 plt.grid(linestyle='--')
64
65 # Guardar y mostrar la gráfica
66 plt.savefig('../Teoria-Aproximacion/Imagenes/Evolucion-
     Costo.pdf')
```

El resultado es igual que cuando resolvimos usando el método de mínimos cuadrados. A continuación se muestra la gráfica de la evolución del aprendizaje del perceptrón.

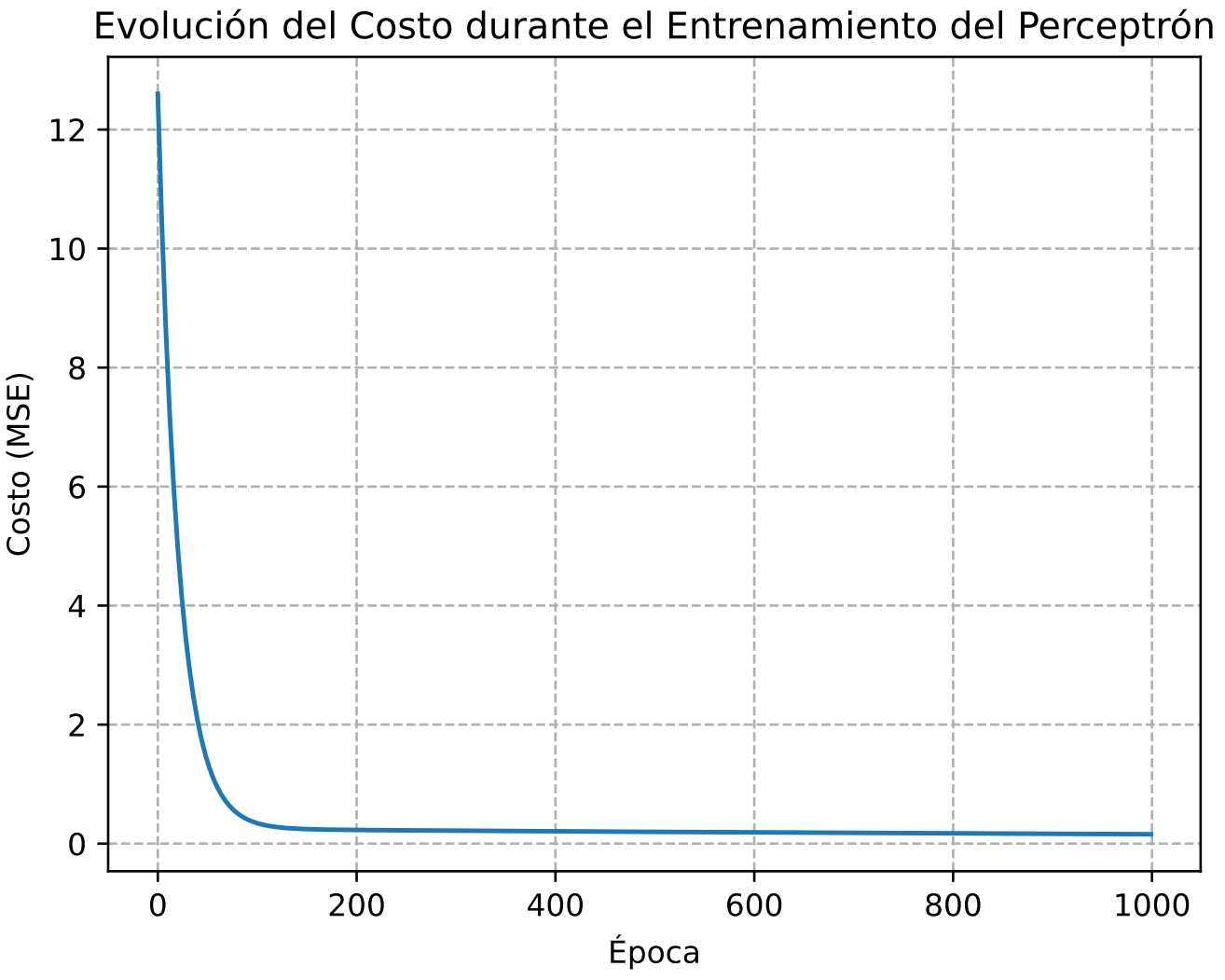

Figura 5.2: Entrenamiento de un perceptrón para aproximar una función lineal

5.3 Ajuste Cuadrático Mediante un Perceptron con Expansión de Características

En esta sección, extenderemos el concepto de perceptrón para aproximar una función cuadrática. Para lograr esto con perceptrón lineal, emplearemos la técnica de expansión de características. Transformaremos nuestros datos de entrada originales para que el perceptrón pueda aprender una relación no lineal en el espacio original, pero que sea lineal en el espacio de características expandido.

5.3.1 Generación de datos Cuadráticos con Perturbaciones

Comenzaremos generando un conjunto de datos que siga aproximadamente una función cuadrática.

Utilizaremos como ejemplo la función

$$f(x) = -x^2 + 4.75x - 5.08$$

y añadiremos pequeñas perturbaciones aleatorias para simular datos del mundo real con cierto nivel de ruido. Generaremos 12 puntos distribuidos uniformemente en el intervalo $[0, 5]$, tal como veremos en el siguiente código de Python.

```python
# Función cuadrática generadora
def funcion_cuadratica(x):
    return -x**2 + 4.75*x - 5.08

# Generar 12 puntos en el intervalo [0, 5]
x = np.linspace(0, 5, 12)
y_real = funcion_cuadratica(x)

# Lista de las perturbaciones aleatorias
ruido = 0.5 * np.random.rand(len(x))
y = y_real + ruido

# Visualizar los datos generados
plt.plot(x, y_real, label='Función Cuadrática Generadora')
plt.scatter(x, y, c='red', zorder=3, label='Datos Generados')
plt.xlabel('Valores de $x$')
plt.ylabel('Valores de $y$')
plt.title('Datos Generados para la Aproximación Cuadrática')
plt.legend()
plt.grid(linestyle='--')

# Guardar y mostrar la gráfica
plt.savefig('../Teoria-Aproximacion/Imagenes/Datos-Generados-Funcion-Cuadratica.pdf')
plt.show()
```

La gráfica generada es la siguiente

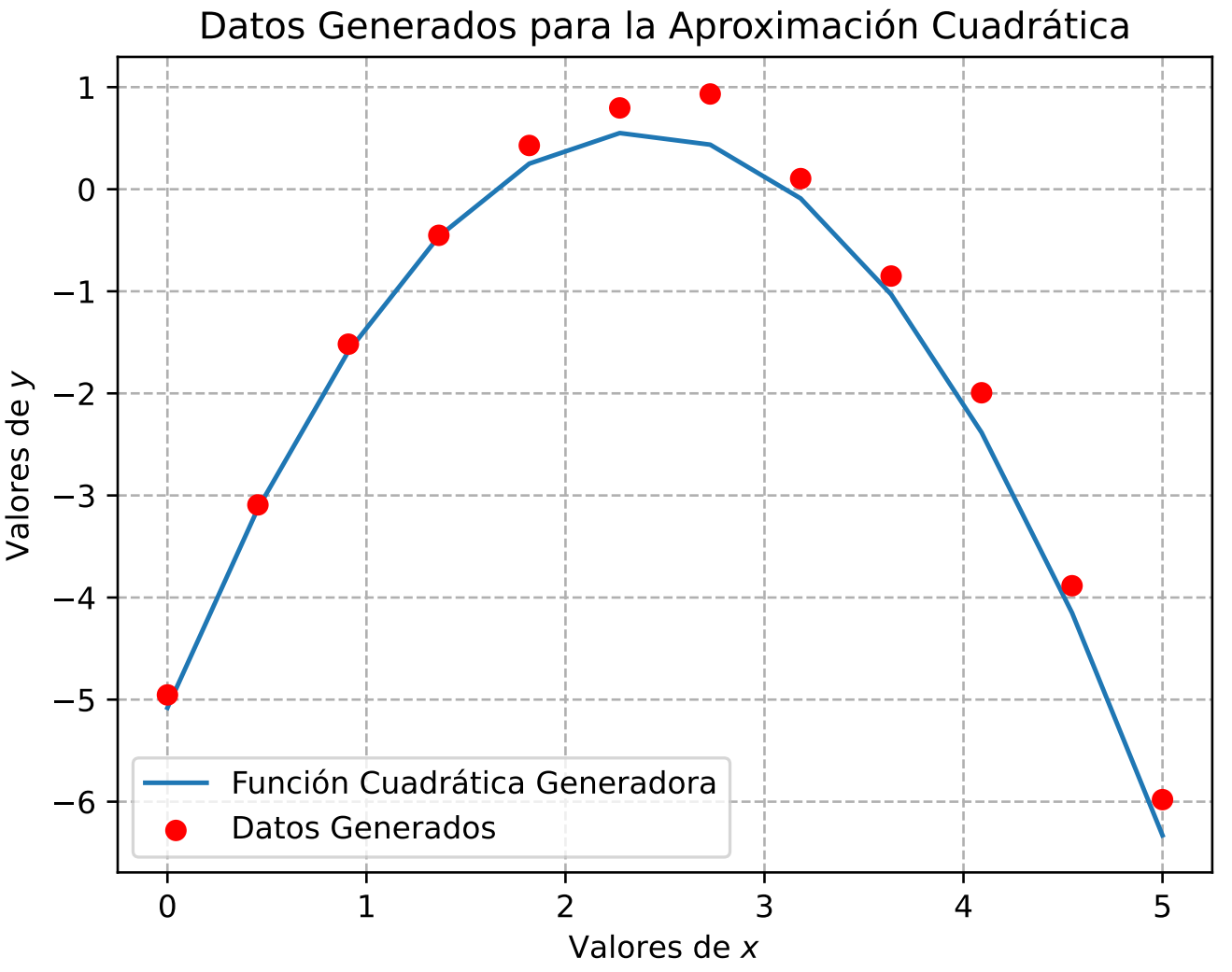

Figura 5.3: Gráfico de los Datos Generados

5.3.2 Aproximación Cuadrática con un Perceptron y Expansión de Características

Un perceptrón estándar con una sola entrada solo puede aprender relaciones lineales. Para que pueda aproximar una función cuadrática de la forma

$$\hat{y} = w_2 \cdot x^2 + w_1 \cdot x + b$$

necesitamos expandir nuestras características de entrada. En lugar de alimentar directamente el valor de x al perceptrón, crearemos un nuevo vector de características para cada punto de datos (x_i):

$$\mathbf{x}'_i = \begin{bmatrix} x_i^2 & x_i & 1 \end{bmatrix}$$

Aquí, hemos creados dos nuevas "entradas" para nuestro perceptrón: x_i^2 y x_i y el término constante "1" se multiplicará por el sesgo. Ahora, nuestro perceptrón lineal operará en este espacio de características expandido. La salida del perceptrón será:

$$\hat{y}_i = w_2 \left(x_i^2\right) + w_1 \left(x_i\right) + b \left(1\right)$$

Donde:

- $\hat{y}_i$ es la salida predicha para el punto x_i
- w_2 es el peso asociado a la característica x_i^2
- w_1 es el peso asociado a la característica x_i
- b es el sesgo (peso asociado a la característica constante 1)

Ahora, el problema se convierte en encontrar los valores óptimos para los pesos w_2, w_1 y el sesgo b que minimicen una función de costo, como el Error Cuadrático Medio (MSE), entre las predicciones $\hat{y}_i$ y los valores reales y_i.

5.3.2.1 La función de Costo (MSE)

La función de costo que utilizaremos es el Error Cuadrático Medio, definida como:

$$J\left(w_2, w_1, b\right) = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(w_2 \cdot x_i^2 + w_1 \cdot x_i + b - y_i\right)^2$$

Nuestro objetivo es encontrar los valores de w_2, w_1, y b que minimicen esta función de costo.

5.3.2.2 Aprendizaje Mediante Descenso del Gradiente

Al igual que en el caso del ajuste lineal, utilizaremos el algoritmo de descenso del gradiente para encontrar los pesos y el sesgo óptimos. Necesitamos calcular las derivadas parciales de la función de costo con respecto a cada parámatro:

$$\frac{\partial J}{\partial w_2} = \frac{2}{n}\sum_{i=1}^{n}\left(w_2 \cdot x_i^2 + w_1 \cdot x_i + b - y_i\right) \cdot x_i^2$$

$$\frac{\partial J}{\partial w_1} = \frac{2}{n}\sum_{i=1}^{n}\left(w_2 \cdot x_i^2 + w_1 \cdot x_i + b - y_i\right) \cdot x_i$$

$$\frac{\partial J}{\partial b} = \frac{2}{n}\sum_{i=1}^{n}\left(w_2 \cdot x_i^2 + w_1 \cdot x_i + b - y_i\right)$$

Luego actualizaremos los pesos y el sesgo iterativamente:

$$w_2 = w_2 - \alpha \frac{\partial J}{\partial w_2}$$
$$w_1 = w_1 - \alpha \frac{\partial J}{\partial w_1}$$
$$b = b - \alpha \frac{\partial J}{\partial b}$$

donde α es la tasa de aprendizaje.

5.3.2.3 Implementación en Python

A continuación, implementaremos el perceptrón con expansión de características y el algoritmo de descenso del gradiente en Python para aproximar la función cuadrática.

```
# Función cuadrática generadora
def funcion_cuadratica(x):
    return -x**2 + 4.75*x - 5.08

# Generar 12 puntos en el intervalo [0, 5]
x = np.linspace(0, 5, 12)
y_real = funcion_cuadratica(x)

# Lista de las perturbaciones aleatorias
ruido = 0.5 * np.random.rand(len(x))
y = y_real + ruido

# Inializamos los pesos y el sesgo
w2 = 0.0
w1 = 0.0
b = 0.0
alfa = 0.0001
epocas = 200000
historial_costo = []

# Entrenamos el perceptron mediante descenso del
  gradienet
for epoca in range(epocas):
```

```
23
24      # Calcular las predicciones del perceptrón con las
          características expandidas
25      y_ = w2 * x**2 + w1 * x + b
26
27      # Calcular el costo (MSE)
28      costo = np.mean((y_ - y)**2)
29      historial_costo.append(costo)
30
31      if (costo < 0.01):
32          break
33
34      # Calcular los gradientes
35      dw2 = (2/n) * np.sum((y_ - y) * x**2)
36      dw1 = (2/n) * np.sum((y_ - y) * x)
37      db = (2/n) * np.sum(y_ - y)
38
39      # Actualizamos los pesos y el sesgo
40      w2 = w2 - alfa * dw2
41      w1 = w1 - alfa * dw1
42      b  = b - alfa * db
43
44      # Imprimir el costo cada 10000 épocas para
          monitorear el progreso (opcional)
45      if (epoca + 1) %20000 == 0:
46          print(f'Época {epoca + 1}, Costo: {costo:.6f}')
47
48  # Imprimir los resultados
49  print(f"Peso w2 aprendido: {w2:.3f}")
50  print(f"Peso w1 aprendido: {w1:.3f}")
51  print(f"Sesgo aprendido: {b:.3f}")
52  print(f"Función Cuadrática Aproximada: f(x) = {w2:.3f}x
      ^2 + {w1:.3f}x + {b:.3f}")
53
54  # Generar los puntos para la curva de ajuste
55  x_dom = np.linspace(min(x), max(x), 100)
56  y_eval = w2 * x_dom**2 + w1 * x_dom + b
```

```
57
58 # Graficar los datos de aprendizaje y la aproximación
59 plt.plot(x_dom, y_eval, label='Función Aproximada')
60 plt.scatter(x, y, c='red', zorder=3, label='Datos de
     Entrenamiento')
61 plt.xlabel('Valores de $x$')
62 plt.ylabel('Valores de $y$')
63 plt.title('Aproximación Cuadrática Mediante un
     Perceptrón')
64 plt.legend()
65 plt.grid(linestyle='--')
66
67 # Guardar la gráfica y mostrar
68 plt.savefig('../Teoria-Aproximacion/Imagenes/
     Aproximacion-Funcion-Cuadratica.pdf')
69 plt.show()
70
71 # Graficar la función de costo durante el entrenamiento
72 plt.figure()
73 plt.plot(range(epocas), historial_costo)
74 plt.xlabel('Época')
75 plt.ylabel('Costo (MSE)')
76 plt.title(f'Evolución del Costo (MSE), LR={alfa}')
77 plt.grid(linestyle='--')
78
79 # Guardar la gráfica y mostrar
80 plt.savefig('../Teoria-Aproximacion/Imagenes/Evolucion-
     Aprendizaje-Aproximacion-Funcion-Cuadratica.pdf')
81 plt.show()
```

La gráfica de la evolución del aprendizaje se muestra en la siguiente figura:

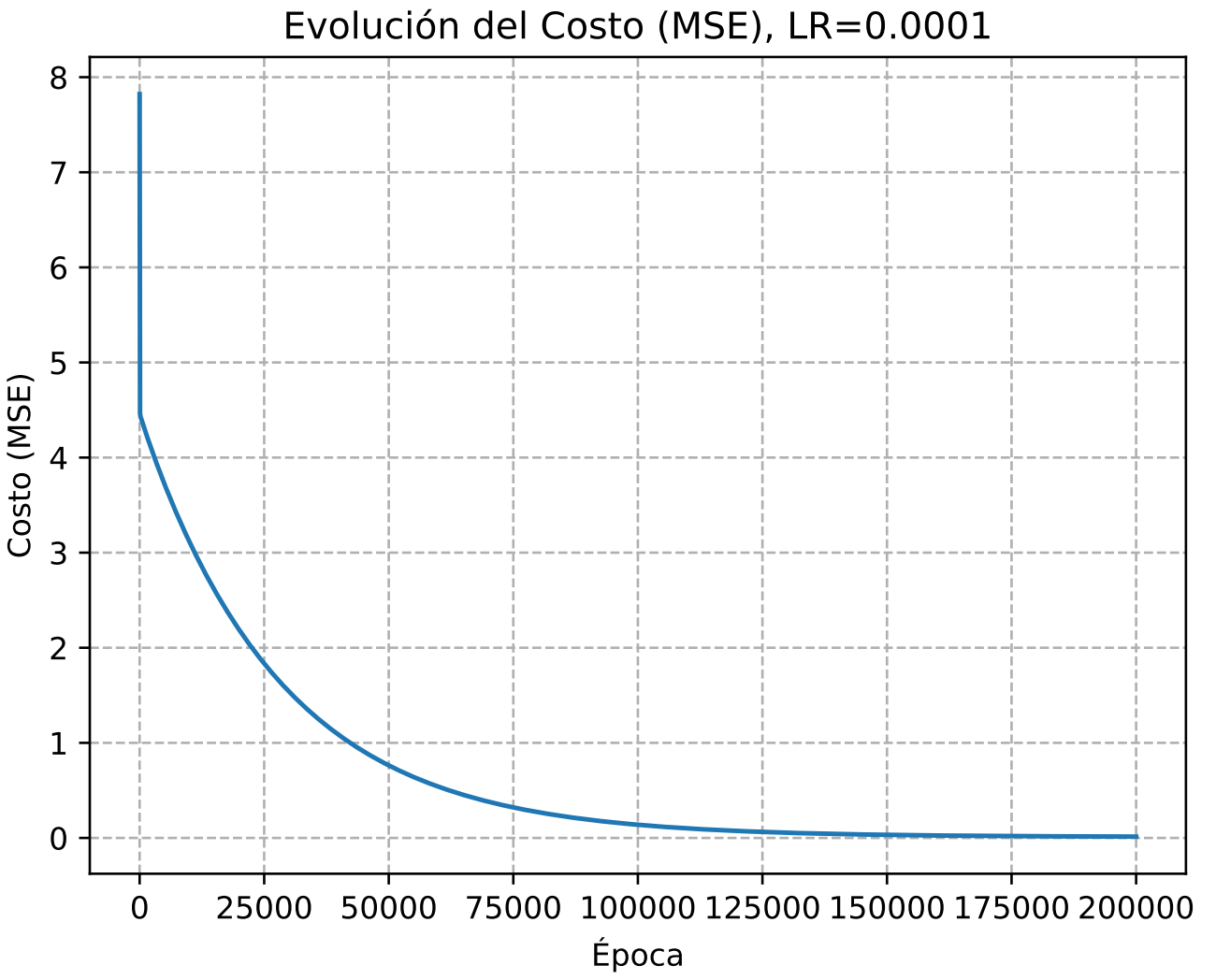

Figura 5.4: Evolución del Aprendizaje de la Aproximación de la Función Cuadrática

y el ajuste se presenta en la siguiente figura:

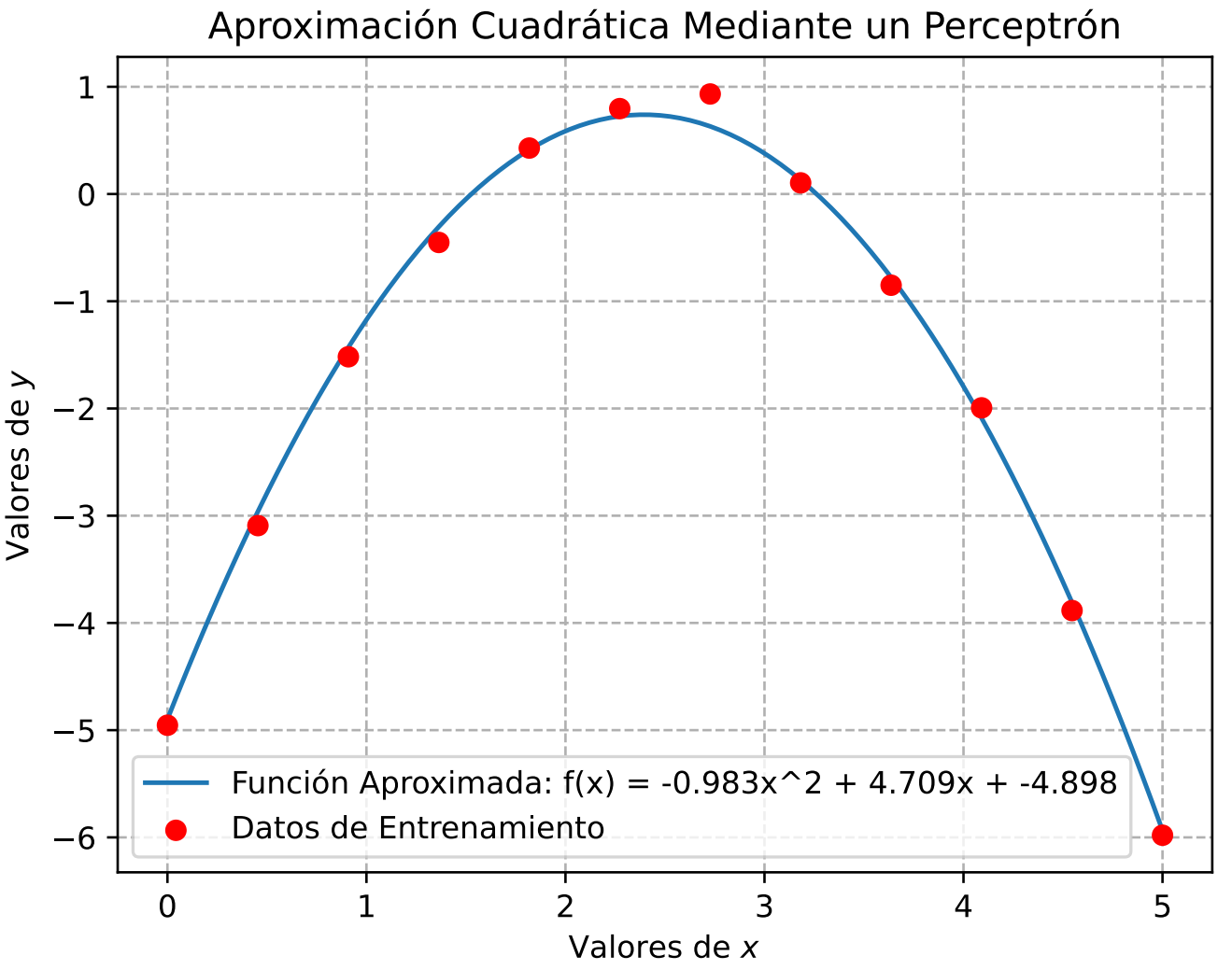

Figura 5.5: Ajuste de la función cuadrática

En particular este método se vuelve más costoso, porque la tasa de aprendizaje es muy pequeña y la cantidad de épocas es muy grande.

5.4 Aproximación Polinomial

Cómo vimos en las secciones anteriores, podemos aproximar un conjunto de datos a una función lineal o cuadrática. Queremos extender el método a las funciones polinómicas.

Sea una función polinómica definida por

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

que también se puede representar mediante la serie

$$P_n(x) = \sum_{k=0}^{n} a_k x^k$$

El objetivo del método es encontrar el conjunto de coeficientes del polinomio

$$a_0, a_1, \ldots a_n$$

a través del entrenamiento de una red (perceptrón) con el conjunto de puntos (nodos).

Por conveniencia y trabajando con la nomenclatura de pesos vamos a considerar el polinomio como sigue

$$P_n(x) = w_0 + w_1 x + \ldots + w_n \cdot x^n$$

y la evaluación de cualquier x_i esta dada por

$$\hat{y}_i = P_n(x_i) = w_0 + w_1 x_i + \ldots + w_n \cdot x_i^n$$

Luego la función de costo esta dada por

$$\begin{aligned} J(w_0, \ldots, w_n) &= \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(\sum_{k=0}^{r}\left(w_k \cdot x_i^k\right) - y_i\right)^2 \end{aligned}$$

entonces para calcular el gradiente de la función derivamos la función de costo para cada peso, así tenemos

$$\frac{\partial J}{\partial w_k} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)\, x_i^k$$

con esta derivada parcial actualizamos cada uno de los pesos w_k de la siguiente manera

$$w_k = w_k - \alpha\frac{\partial J}{\partial w_k}$$

para $k = 0, 1, \ldots, r$ donde r es el grado del polinomio y α es la tasa de aprendizaje.

5.4.1 Implementación del Código en Python

Lo primero que necesitamos es una colección de puntos (nodos) para realizar el entrenamiento, para ello vamos a usar la interpolación para crear un polinomio de grado cuatro que me interpole los siguientes puntos:

x	0	$\frac{\pi}{2}$	π	$\frac{3\pi}{2}$	2π
y	0	1	0	-1	0

Con el fin de tener una función parecida a una función cúbica, tal como se muestra en la siguiente figura

Figura 5.6: Gráfica del Polinomio Interpolador

El código para generar el polinomio interpolador es el siguiente

```
# Definir una función generadora de valores
nodos_x = np.array([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.
  pi])
nodos_y = np.array([0, 1, 0, -1, 0])
nodos = (nodos_x, nodos_y)
poly = polyInterpoLagrange(nodos)

x = np.linspace(min(nodos_x), max(nodos_x), 100)
y = evalPoly(poly, x)

# Graficar la función
plt.plot(x, y, label='Gráfica de la Función Polinómica')
```

```
plt.scatter(nodos_x, nodos_y, c='red', zorder=3, label='
  Puntos de Interpolación')
plt.xlabel('Valores de $x$')
plt.ylabel('Valores de $y$')
plt.title('Gráfica de Función Polimica Interpoladora')
plt.grid(linestyle='--')

# Mostrar y guardar la gráfica
plt.savefig('../Teoria-Aproximacion/Imagenes/Polinomio-
  Generador.pdf')
plt.show()
```

Con este polinomio, vamos a generar 24 datos tal como se muestra en la siguiente figura

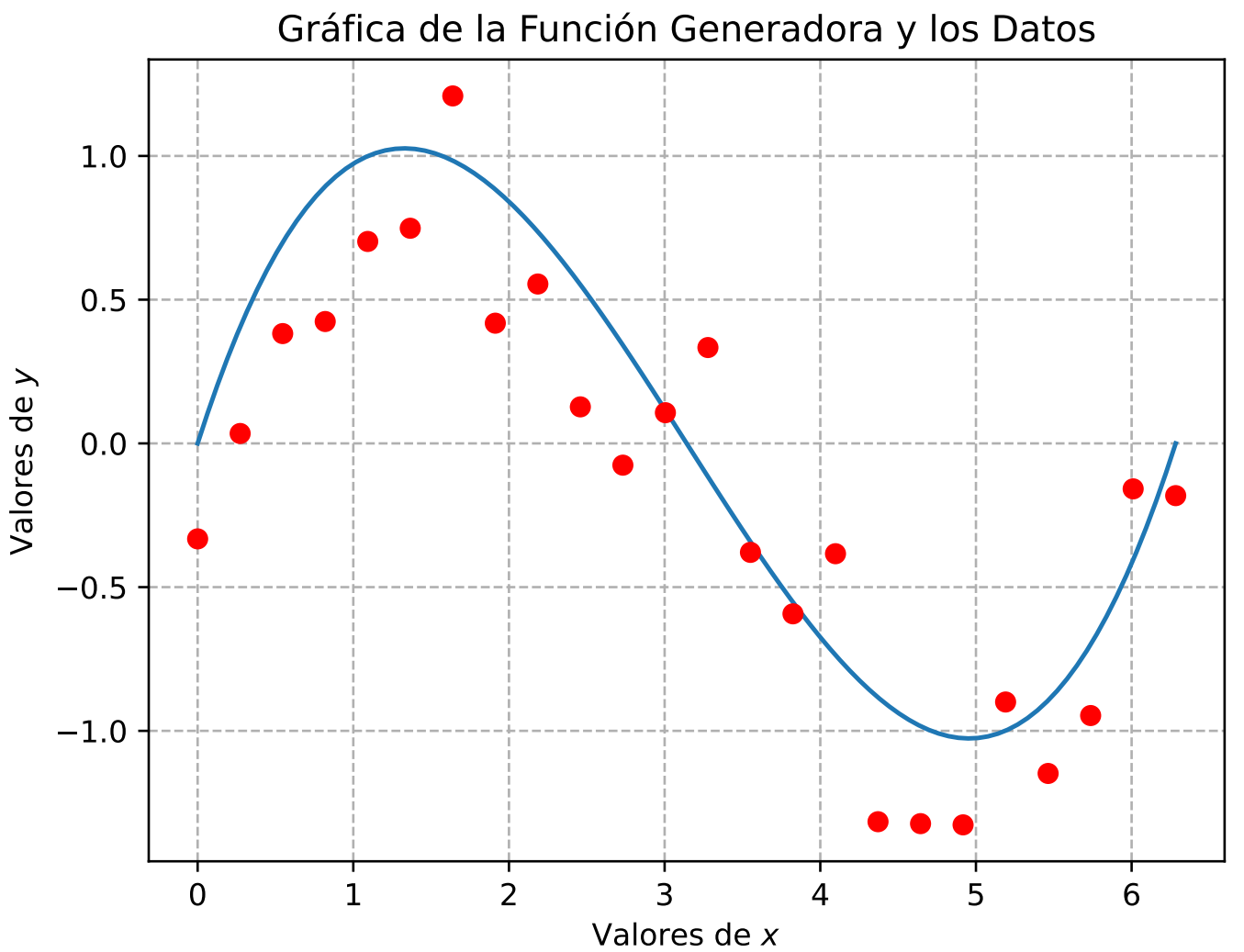

Figura 5.7: Gráfica de los Datos para el entrenamiento

El código para generar los datos de entrenamiento es el siguiente

```
# Generar los datos de entrenamiento
x_train = np.linspace(min(nodos_x), max(nodos_x), 24)
y_train = evalPoly(poly, x_train)
```

```
4
5 # Generar un ruido sobre los valores de y
6 ruido = np.random.rand(len(x_train)) - 0.5
7 y_train = y_train + ruido
8
9 # Graficar la función generadora y los puntos generados
    con ruido
10 plt.plot(x, y, label='Función Generadora')
11 plt.scatter(x_train, y_train, c='red', zorder=3, label='
    Datos')
12 plt.xlabel('Valores de $x$')
13 plt.ylabel('Valores de $y$')
14 plt.title('Gráfica de la Función Generadora y los Datos'
    )
15 plt.grid(linestyle='--')
16
17 # Mostrar y guardar la gráfica
18 plt.savefig('../Teoria-Aproximacion/Imagenes/Datos-
    Generados-Funcion-Polinomica.pdf')
19 plt.show()
```

El siguiente código nos permite entrenar al perceptrón para encontrar el ajuste polinómico

```
1 # Inializar los pesos, la tasa de aprendizaje y la
    cantidad de épocas
2 W = np.array([0.0, 0.0, 0.0, 0.0])
3 alfa = 0.00001
4 epocas = 1000000
5 historial_costo = []
6 n = len(x_train)
7 grado = 3
8
9 # Entrenar al perceptrón
10 for epoca in range(epocas):
11
12     # Calcular el valor aproximado del polinomio
13     y_ = evalPoly(W, x_train)
14
```

```
15      # Calcular el costo
16      costo = np.mean((y_ - y_train)**2)
17      historial_costo.append(costo)
18
19      # Calcular los gradientes y actualizar los pesos
20      for k in range(len(W)):
21          dW = (2/n) * np.sum((y_ - y_train) * x_train**(
              grado-k))
22          W[k] = W[k] - alfa*dW
23
24      # Mostrar el costo cada 1000 épocas
25      if (epoca + 1) %10000 == 0:
26          print("Costo (Época: "+str(epoca + 1)+"): ",
              costo, end='\r')
27
28  # Imprimir los resultados
29  print("Costo: ", costo)
30  print("Pesos entrenados: ", W)
31
32  # Graficar el historial del costo
33  plt.plot(range(epocas), historial_costo)
34  plt.xlabel('Épocas')
35  plt.ylabel('Costo (ECM)')
36  plt.title('Evolución del Costo del Algoritmo')
37  plt.grid(linestyle='--')
38
39  # Mostrar y guardar la gráfica
40  plt.savefig('../Teoria-Aproximacion/Imagenes/Evolucion-
      Costo-Aproximacion-Polinomica.pdf')
41  plt.show()
42
43  # Graficar los puntos de entrenamiento y la función
      encontrada
44  x_dom = np.linspace(min(x_train), max(x_train), 100)
45  y_eval = evalPoly(W, x_dom)
46  plt.plot(x_dom, y_eval, label='Polinomio Aproximado')
```

```
47 plt.scatter(x_train, y_train, c='red', zorder=3, label='
     Datos de Entrenamiento')
48 plt.xlabel('Valores de $x$')
49 plt.ylabel('Valores de $y$')
50 plt.title('Gráfica de la Función Aproximada y los Datos
     de Entrenamiento')
51 plt.grid(linestyle='--')
52
53 # Mostrar y guardar la gráfica
54 plt.savefig('../Teoria-Aproximacion/Imagenes/Funcion-
     Polinomica-Aproximada.pdf')
55 plt.show()
```

La gráfica de la evolución del costo se muestra en la siguiente figura:

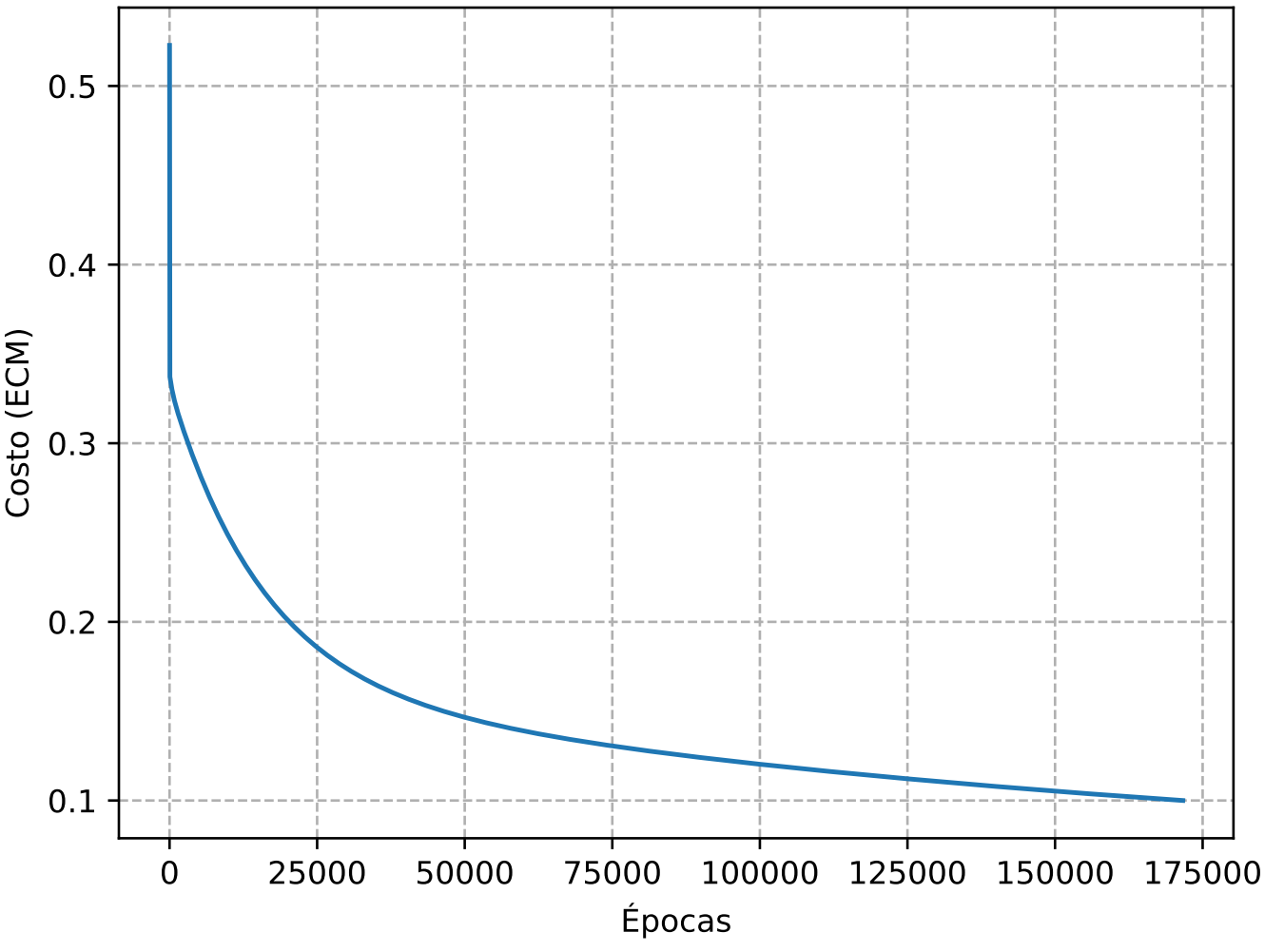

Figura 5.8: Evolución del Costo

y la gráfica del polinomio aproximado y los datos de entrenamiento se muestra en la siguiente figura

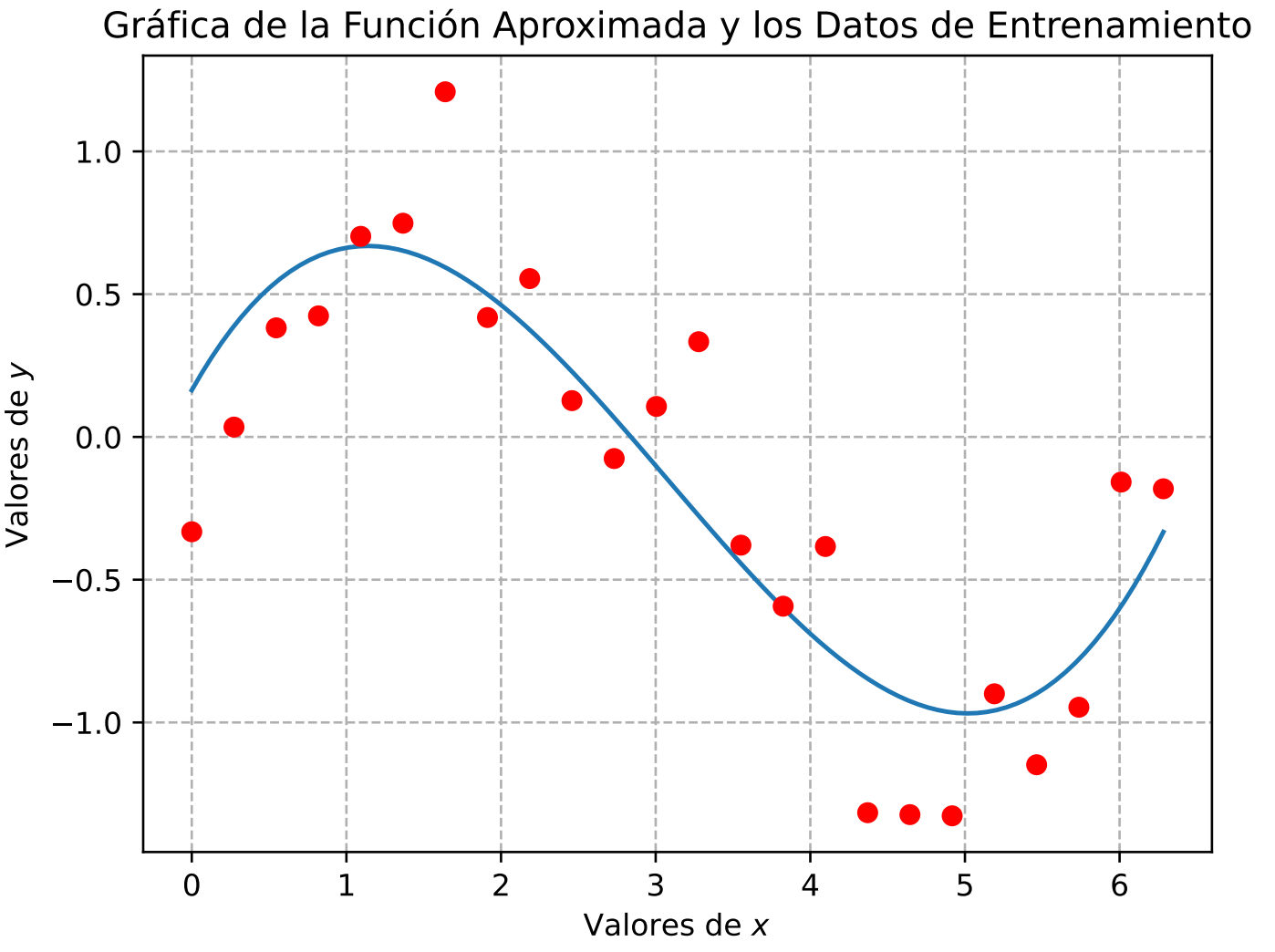

Figura 5.9: Gráfica del polinomio aproximado y los datos de entrenamiento

Una primera conclusión es que este algoritmo tarda mucho en realizar la aproximación (2 minutos y 43 segundos) dado que tiene una tasa de aprendizaje muy baja $\alpha = 0.00001$ y una cantidad muy grande de épocas $1.000.000$.

El polinomio encontrado es el siguiente

$$P_3(x) = 0.05604159x^3 - 0.51750655x^2 + 0.96011259x + 0.16373819$$

5.4.2 Cambio de Parada

En el algoritmo anterior, se dan una cantidad de épocas para abordar el problema, lo que podemos hacer ahora es establecer un valor de tolerancia para decirle al algoritmo cuando detenerse, tal como se muestra en el siguiente código.

```
1 # Inializar los pesos, la tasa de aprendizaje y la
    cantidad de épocas
2 W = np.array([0.0, 0.0, 0.0, 0.0])
```

```
alfa = 0.00005
historial_costo = []
n = len(x_train)
grado = len(W) - 1
tol = 0.1
epoca = 1
costo = 1

# Entrenar al perceptrón
while (costo > tol):

    # Calcular el valor aproximado del polinomio
    y_ = evalPoly(W, x_train)

    # Calcular el costo
    costo = np.mean((y_ - y_train)**2)
    historial_costo.append(costo)

    # Calcular los gradientes y actualizar los pesos
    for k in range(len(W)):
        dW = (2/n) * np.sum((y_ - y_train) * x_train**(
            grado-k))
        W[k] = W[k] - alfa*dW

    # Mostrar el costo cada 1000 épocas
    if (epoca) %100 == 0:
        print(f"Época: {epoca:.0f}; Costo: {costo:.7f}",
            end='\r')


    # Incrementar el valor de la época
    epoca = epoca + 1

# Imprimir los resultados
print("Costo: ", costo)
print("Pesos entrenados: ", W)
print("Epocas necesarias para el entrenamiento: ",epoca)
```

```
38
39 # Graficar el historial del costo
40 plt.plot(range(len(historial_costo)), historial_costo)
41 plt.xlabel('Épocas')
42 plt.ylabel('Costo (ECM)')
43 plt.title('Evolución del Costo del Algoritmo')
44 plt.grid(linestyle='--')
45
46 # Mostrar y guardar la gráfica
47 plt.savefig('../Teoria-Aproximacion/Imagenes/Evolucion-
     Costo-Aproximacion-Polinomica.pdf')
48 plt.show()
49
50 # Graficar los puntos de entrenamiento y la función
     encontrada
51 x_dom = np.linspace(min(x_train), max(x_train), 100)
52 y_eval = evalPoly(W, x_dom)
53 plt.plot(x_dom, y_eval, label='Polinomio Aproximado')
54 plt.scatter(x_train, y_train, c='red', zorder=3, label='
     Datos de Entrenamiento')
55 plt.xlabel('Valores de $x$')
56 plt.ylabel('Valores de $y$')
57 plt.title('Gráfica de la Función Aproximada y los Datos
     de Entrenamiento')
58 plt.grid(linestyle='--')
59
60 # Mostrar y guardar la gráfica
61 plt.savefig('../Teoria-Aproximacion/Imagenes/Funcion-
     Polinomica-Aproximada-Metodo-2.pdf')
62 plt.show()
```

En éste algoritmo el valor de tolerancia se fija en 0.1 y un costo inicial de 1 para poder que el ciclo while se pueda iniciar. La gráfica de la aproximación se presenta en la siguiente figura, comparando el resultado con el realizado en el método anterior.

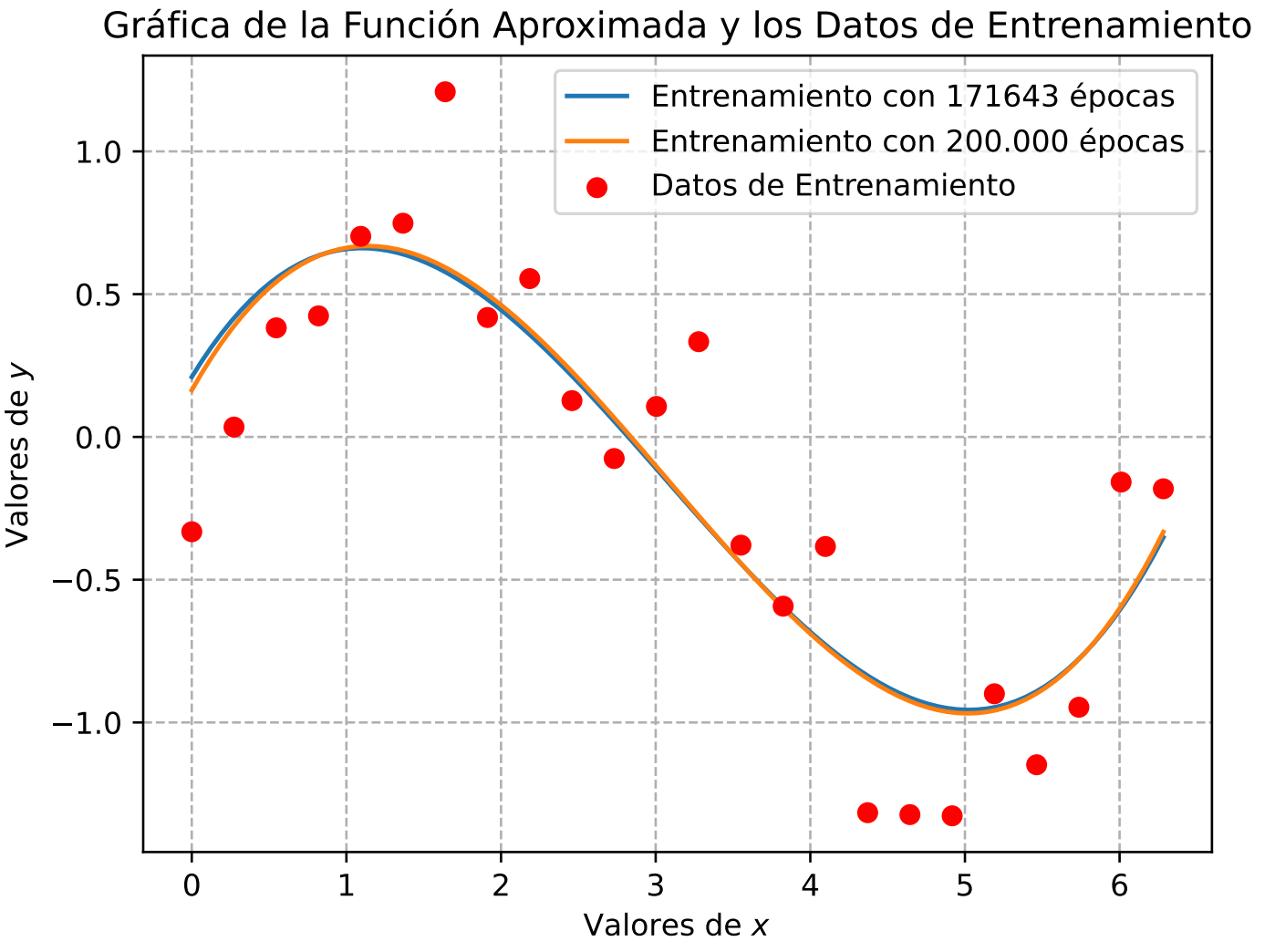

Figura 5.10: Comparación de los dos tipos de entrenamiento

5.5 Método Alternativo

Los dos últimos métodos que vimos consideran realizar el entrenamiento con una cantidad grande de épocas. A continuación presentamos un algoritmo para evitar el uso de las épocas.

5.5.1 Origen del Algoritmo

Consideremos la siguiente situación problema. Existe una vía que comunica dos ciudades A y B. Una persona se dispone viajar entre las dos ciudades de noche, la vía tiene malas condiciones, no están delimitados sus carriles y por la condición de poca luz, los conductores del carril contrario por lo general están conduciendo con “luces altas” esto hace que algunos conductores con poca experiencia tengan dificultades para encontrar el carril correcto que no se encuentra bien delimitado. A través de esta experiencia he encontrado una buena

solución para poder ir bien centrado en el carril correspondiente y es a través de una "caravana" de automóviles. El primer automóvil de la "caravana" esta conduciendo con ciertas dificultades, pero también permite que el segundo conductor tenga menos errores en la conducción y el movimiento del volante no tenga tantas amplitudes, a su vez esta información para al tercer conductor y así hasta el último conductor de la "caravana" reduciendo el error al mínimo.

Con esto en mente, podemos considerar a la vía como una función polinómica de orden $\mathcal{C}^{[1,\infty]}\left([a,b]\right)$ sobre la cual transitan n neuronas, cada neurona esta asociada a un punto de observación experimental. Procesa la información del primer "nodo observado" genera las actualizaciones y cálculos correspondientes, pasando esta información a la neurona siguiente. De esta forma cada neurona recorrerá los n nodos. Por ahora el algoritmo no es óptimo, ya que se deben hacer a lo máximo n^2 cálculos para encontrar la aproximación a la función polinómica, hecho sobre el cual se reflexionará más adelante encontrando una solución de optimización.

5.5.2 Definición del Algoritmo

Iniciemos con analizar que sucede en una (1) neurona.

Tenemos un conjunto de datos Ω, definidos como $\Omega = (x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$ con la condición de que $x_i \neq x_j$ para $i \neq j$, es decir todos los x_i para $i = 1, 2, \ldots, n$ son diferentes. Consideremos también que queremos aproximar el conjunto de nodos "observaciones" Ω a una función polinómica de grado n, tal como sigue

$$P_r(x) = a_0x^0 + a_1x^1 + a_2x^2 + \cdots + a_rx^r$$

Con esto en mente, consideremos estos coeficientes sobre un vector que tiene la forma

$$C = \begin{bmatrix} a_0 & a_1 & \cdots & a_n \end{bmatrix} \tag{5.1}$$

la idea es asignarle a la primera neurona N_1 los coeficientes del polinomio como elementos aleatorios, es decir

$$N_1 = \begin{bmatrix} \widetilde{a_0} & \widetilde{a_1} & \cdots & \widetilde{a_r} \end{bmatrix}$$

y con estos coeficientes, evaluar la función polinómica en x_0, para ello recordemos que un polinomio de grado r se puede escribir de la

siguiente forma:

$$P_r(x) = \sum_{k=0}^{r} a_k x^k$$

entonces al evaluar el polinomio tenemos que

$$\hat{y} = P_r(x_0)$$

donde $\hat{y}$ indica el valor calculado del polinomio con unos coeficientes aleatorios, luego podemos calcular el error así

$$\epsilon = y_i - \hat{y}$$

Esto también lo podemos ver de la siguiente forma

$$\epsilon = y_0 - \sum_{k=1}^{r} a_k (x_i)^k$$

es decir que ϵ es una función que depende de los valores de los coeficientes del polinomio a_k para $k = 1, 2, \ldots, r$ (en un caso particular).

La pregunta es cómo se puede minimizar el error cometido.

Para ello podemos proponer el Error Cuadrático Medio (ECM) que tendría la siguiente forma

$$\epsilon = \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{k=1}^{r} a_k (x_u)^k - y_i \right)^2$$

entonces al derivar tenemos,

$$\frac{\partial \epsilon}{\partial a_k|_{k=1}^{r}} = \frac{2}{n} \sum_{i=1}^{n} \left(\sum_{k=1}^{r} a_k (x_i)^k - y_i \right) x^k$$

Con estos valores podemos actualizar los valores de los coeficientes iniciales (obtenidos mediante procesos aleatorios) de la siguiente forma en 5.1:

$$a_k = a_k - \alpha \frac{\partial \epsilon}{\partial a_k|_{k=1}^{r}}$$

donde α es un factor de aprendizaje (debe considerarse un valor pequeño).

Como estos valores (los coeficientes) ya se actualizaron, la segunda neurona N_2 ya no recibirá coeficientes aleatorios, recibirá los calculados por la neurona N_1. Luego de que la neurona N_2 realiza el proceso anterior, pasa los coeficientes actualizados a la neurona N_3, así sucesivamente hasta la neurona N_n

5.6 Implementación

A continuación se presenta un código en Python implementando estas ideas.