Matemáticas para la Inteligencia Artificial: Implementación desde Cero en C

Prefacio: Forjando la Inteligencia Artificial desde sus Cimientos

- Por qué aprender IA a bajo nivel con C.
- De la ecuación a la función en C: construyendo sin dependencias.
- Requisitos: conocimientos básicos de C (punteros, `structs`, `malloc`/`free`).
- Estructura del libro: de la teoría matemática a la implementación práctica.

Parte I: Los Pilares Fundamentales en C

Capítulo 1: Cálculo Diferencial - Programando el Cambio y la Optimización

- **1.1. Representando Funciones en C:**
  - Uso de punteros a funciones para representar funciones matemáticas.
  - Creando un "paisaje de datos" con arrays.
- **1.2. La Derivada: Implementando el Corazón del Aprendizaje:**
  - **Implementación en C:** Cálculo de la derivada numérica (aproximación por diferencias finitas).
  - Interpretación del resultado de la función C en el contexto de la optimización.
- **1.3. La Regla de la Cadena: La Columna Vertebral del *Backpropagation* en C:**
  - Estructurando un grafo computacional simple con `structs` y punteros.
  - **Implementación en C:** Una función `chain_rule()` que demuestre la propagación hacia atrás.
- **1.4. Gradientes: Navegando en Múltiples Dimensiones con Punteros:**
  - **Implementación en C:** Una función que calcula el gradiente numérico para una función multivariable.
  - El Gradiente Negativo: El camino para minimizar el error, implementado con aritmética de punteros.
- **1.5. Aplicación Práctica: El Descenso del Gradiente en C:**
  - **Implementación en C:** El algoritmo completo de Descenso del Gradiente desde cero para encontrar el mínimo de una función.
  - Gestión de la tasa de aprendizaje y el número de iteraciones.

Capítulo 2: Álgebra Lineal - El Lenguaje de los Datos en Memoria

- **2.1. Los Ladrillos de la IA: Estructuras de Datos en C:**

    - **Implementación en C:** Creación de `struct Vector` y `struct Matrix` usando asignación dinámica de memoria (`malloc`).
    - Funciones de utilidad: `create_vector()`, `create_matrix()`, `free_vector()`, `free_matrix()`. La importancia de evitar fugas de memoria.
- **2.2. Operaciones Clave para Redes Neuronales en C:**
    - **Implementación en C:** Funciones `dot_product()`, `vector_add()`.
    - **Implementación en C:** La función `matrix_multiply()`. Consideraciones de rendimiento y eficiencia de caché en los bucles anidados.
- **2.3. Propiedades y Operaciones Matriciales Esenciales en C:**
    - **Implementación en C:** Funciones para crear una matriz identidad y para transponer una matriz (`transpose_matrix()`).
    - Discusión sobre la complejidad de implementar la inversión de matrices y algoritmos alternativos.
- **2.4. Descomposición de Matrices desde Cero:**
    - Explicación teórica de SVD y PCA.
    - **Implementación en C:** Un ejemplo de un algoritmo de descomposición más simple (ej. Descomposición LU) para ilustrar el proceso.

Parte II: Probabilidad y Estadística - Generando Incertidumbre Controlada

Capítulo 3: Fundamentos de Probabilidad en C

- **3.1. Generando Aleatoriedad:**
    - El uso de `rand()` y `srand()` para simular eventos probabilísticos.
    - Limitaciones y cómo crear generadores de números pseudoaleatorios de mejor calidad.
- **3.2. Implementando el Teorema de Bayes:**
    - **Implementación en C:** Un programa simple que actualice una creencia (probabilidad) basándose en nueva evidencia.
- **3.3. Generando Variables Aleatorias:**
    - **Implementación en C:** Funciones para generar muestras de distribuciones discretas (Bernoulli, Uniforme) y continuas (Normal, usando la transformación de Box-Muller).

Capítulo 4: Estadística para el Machine Learning en C

- **4.1. Calculando Estadísticas Descriptivas:**
    - **Implementación en C:** Funciones para calcular media, varianza y desviación estándar de un array de datos.
- **4.2. Implementando Estimadores:**

    - **Implementación en C:** Un ejemplo de Máxima Verosimilitud (MLE) para un modelo simple (ej. estimar el parámetro de una distribución de Bernoulli).
- **4.3. Métricas de Evaluación desde Cero:**
    - **Implementación en C:** Funciones `calculate_accuracy()`, `calculate_mse()` que tomen arrays de predicciones y valores reales.

Parte III: Optimización - La Búsqueda de la Excelencia en C

Capítulo 5: Más Allá del Descenso del Gradiente en C

- **5.1. Implementando Variantes del Descenso del Gradiente:**
    - **Implementación en C:** Descenso del Gradiente Estocástico (SGD), seleccionando muestras aleatorias de un dataset.
    - **Implementación en C:** Mini-batch Gradient Descent.
- **5.2. Algoritmos de Optimización Avanzados en C:**
    - **Implementación en C:** Añadiendo un término de "momento" a nuestra implementación de Descenso del Gradiente.
    - **Implementación en C:** Una versión simplificada de Adam, gestionando los promedios móviles de primer y segundo momento.

Parte IV: Construyendo una Red Neuronal en C

Capítulo 6: Juntando todas las Piezas

- **6.1. Estructuras de Datos para una Red Neuronal:**
    - **Implementación en C:** `struct Layer`, `struct Network` que unen nuestras matrices de pesos, vectores de sesgos y funciones de activación.
- **6.2. El Ciclo de Vida del Aprendizaje en C:**
    - **Implementación en C:** La función `feedforward()`.
    - **Implementación en C:** La función `backpropagate()`, que utiliza la regla de la cadena para calcular gradientes para cada peso y sesgo.
    - **Implementación en C:** La función `update_weights()`, que aplica un algoritmo de optimización.
- **6.3. Un Proyecto Completo: Red Neuronal para XOR:**
    - **Implementación en C:** Un programa completo que define, entrena y prueba una red neuronal para resolver el problema XOR, un "Hola Mundo" clásico de las redes neuronales.

Apéndices:

- **A. Notación Matemática.**
- **B. Gestión Avanzada de Memoria y Punteros en C.**
- **C. Guía de Estilo y Depuración para Código de IA en C.**
- **D. Soluciones a ejercicios seleccionados.**

Capítulo 1: Cálculo Diferencial - Programando el Cambio y la Optimización

1.1. Representando Funciones en C

¡Bienvenido al primer capítulo práctico! Aquí es donde las ideas matemáticas abstractas comienzan a tomar forma en el mundo lógico y estructurado del lenguaje C. Nuestro primer desafío es responder a una pregunta fundamental: si en la inteligencia artificial todo se trata de optimizar funciones (como minimizar una función de coste), ¿cómo representamos una "función" en nuestro código?

En matemáticas, una función como f(x)=x2 es una regla que toma una entrada, , y devuelve una salida. En C, tenemos una herramienta análoga y sorprendentemente elegante para manejar este concepto: los **punteros a funciones**.

Uso de punteros a funciones para representar funciones matemáticas

Un puntero, como ya sabrás, es una variable que almacena la dirección de memoria de otra variable. De la misma manera, un **puntero a función** es una variable que almacena la dirección de memoria donde reside el código de una función. Esto nos permite tratar a las funciones como si fueran datos: podemos pasarlas como argumentos a otras funciones, devolverlas desde funciones o almacenarlas en estructuras.

Esta capacidad es la piedra angular de nuestro enfoque. Nos permitirá escribir un único algoritmo de optimización (como el Descenso del Gradiente) y luego pasarle *cualquier* función que queramos optimizar.

La sintaxis para declarar un puntero a función puede parecer intimidante al principio, pero es lógica:

```
// return_type (*pointer_name)(parameter_types);
double (*funcion_matematica_ptr)(double);
```

Esta línea declara una variable llamada `funcion_matematica_ptr` que es un puntero (`*`) capaz de apuntar a cualquier función que devuelva un `double` y acepte un único `double` como argumento.

Veamos un ejemplo práctico. Definiremos dos funciones matemáticas simples y luego usaremos un puntero para referirnos a ellas.

```
#include <stdio.h>
#include <math.h>

// Definimos una función matemática: f(x) = x^2
double parabola(double x) {
    return x * x;
}

// Definimos otra función: g(x) = x^3 - 2x + 5
double cubica(double x) {
    return pow(x, 3) - 2 * x + 5;
}

int main() {
    // Declaramos nuestro puntero a función.
    // Puede apuntar a cualquier función que coincida con su firma: double mi_func(double);
    double (*funcion_actual_ptr)(double);

    // 1. Apuntamos a la función 'parabola'
    funcion_actual_ptr = &parabola;

    // La llamamos a través del puntero
    double resultado1 = funcion_actual_ptr(5.0);
    printf("Usando el puntero en parabola(5.0): %f\n", resultado1); // Salida: 25.000000

    // 2. Ahora, el mismo puntero apunta a la función 'cubica'
    funcion_actual_ptr = &cubica;

    // La llamamos de nuevo a través del puntero
    double resultado2 = funcion_actual_ptr(5.0);
    printf("Usando el puntero en cubica(5.0): %f\n", resultado2); // Salida: 120.000000

    return 0;
}
```

Como puedes ver, la variable `funcion_actual_ptr` actúa como un contenedor versátil para la lógica matemática que queramos ejecutar.

Creando un "paisaje de datos" con arrays

El objetivo de la optimización es encontrar el punto más bajo (o más alto) en el "paisaje" de una función. Para visualizar este paisaje, podemos evaluarla en muchos puntos diferentes dentro de un rango y ver qué valores produce. En C, la forma más natural de hacer esto es iterar sobre un rango de valores y almacenar o imprimir los resultados.

Vamos a crear una función de utilidad que haga exactamente eso. Esta función tomará un puntero a la función que queremos mapear, los límites de un rango (`x_min`, `x_max`) y el número de pasos que queremos evaluar.

```
#include <stdio.h>
#include <math.h>

// --- (Incluir las funciones parabola y cubica de arriba) ---

double parabola(double x) { return x * x; }
double cubica(double x) { return pow(x, 3) - 2 * x + 5; }


// Esta función toma un puntero a función y mapea su salida en un rango.
void mapear_paisaje(double (*func_ptr)(double), double x_min, double x_max, int pasos) {
    // Validamos que los pasos sean positivos para evitar un bucle infinito.
    if (pasos <= 0) {
        printf("El número de pasos debe ser positivo.\n");
        return;
    }

    // Calculamos el tamaño de cada incremento en el eje x.
    double incremento = (x_max - x_min) / (double)pasos;

    printf("--- Mapeando el paisaje de la función ---\n");
    printf("      x         |      f(x)\n");
    printf("----------------|-----------------\n");

    for (int i = 0; i <= pasos; i++) {
        // Calculamos el valor actual de x.
        double x_actual = x_min + i * incremento;
        // Usamos el puntero para llamar a la función que nos pasaron.
        double y_actual = func_ptr(x_actual);
        // Imprimimos los resultados en una tabla.
        printf("  %12.4f  |  %15.4f\n", x_actual, y_actual);
    }
    printf("----------------------------------------\n\n");
}
```

```
int main() {
    // Mapeamos la función 'parabola' en el rango de -10 a 10.
    mapear_paisaje(&parabola, -10.0, 10.0, 20);

    // Mapeamos la función 'cubica' en el mismo rango.
    mapear_paisaje(&cubica, -10.0, 10.0, 20);

    return 0;
}
```

Al ejecutar este código, obtendrás una tabla de valores para cada función. Si graficaras estos puntos, verías la forma de la parábola (con su mínimo en x=0) y la curva de la función cúbica. ¡Hemos creado una representación numérica del paisaje de la función!

Con estas dos herramientas, los punteros a funciones y el mapeo de paisajes, hemos sentado las bases. Ahora estamos listos para el siguiente paso: calcular la pendiente en cualquier punto de este paisaje, lo que nos llevará directamente al concepto de la derivada.

1.2. La Derivada: El Corazón del Aprendizaje

En la sección anterior, aprendimos a representar funciones y a visualizar sus "paisajes". Ahora, daremos el paso más crucial para la optimización: aprender a **medir la inclinación** en cualquier punto de ese paisaje. Esa medida de inclinación es la **derivada**.

Definición intuitiva y formal de la derivada

Intuitivamente, la derivada de una función en un punto específico nos dice dos cosas:

1. **La dirección del cambio:** ¿La función está creciendo (pendiente positiva), decreciendo (pendiente negativa) o está en un punto plano (pendiente cero)?
2. **La magnitud del cambio:** ¿Qué tan rápido está creciendo o decreciendo? Una pendiente pronunciada significa un cambio rápido.

En el contexto de la IA, si nuestra "función" es el error del modelo, la derivada nos dirá exactamente cómo ajustar los parámetros (pesos) para reducir ese error. Si la pendiente es positiva, necesitamos movernos hacia atrás; si es negativa, hacia adelante.

Formalmente, la derivada se define como el límite de la tasa de cambio promedio a medida que el intervalo se hace infinitesimalmente pequeño:

f′(x)=lim_hto0fracf(x+h)−f(x)h

Esta fórmula calcula la pendiente de la línea que pasa por dos puntos muy, muy cercanos en la curva de la función. A medida que la distancia h entre esos puntos tiende a cero, la pendiente

de esa línea se convierte en la pendiente de la línea tangente en el punto x, que es, por definición, la derivada.

Implementación en C: Cálculo de la derivada numérica

En un ordenador, no podemos hacer que h sea *infinitamente* pequeño, pero podemos hacerlo *extremadamente* pequeño. Si elegimos un valor muy pequeño para h (por ejemplo, 0.00001), podemos obtener una aproximación numérica muy precisa de la derivada. Este método se conoce como **diferenciación numérica por diferencias finitas**.

Vamos a crear una función en C que implemente esta idea. Tomará un puntero a la función que queremos derivar, el punto x en el que queremos calcular la pendiente y el pequeño valor h.

```
#include <stdio.h>
#include <math.h>

// Definimos la función que queremos analizar: f(x) = x^2
double parabola(double x) {
    return x * x;
}

/**
 * @brief Calcula la derivada numérica de una función en un punto dado.
 * * @param func_ptr Un puntero a la función a derivar (debe tomar un double y devolver un double).
 * @param x El punto en el que se calculará la derivada.
 * @param h Un valor muy pequeño para aproximar el límite.
 * @return La pendiente (derivada) aproximada de la función en el punto x.
 */
double calcular_derivada_numerica(double (*func_ptr)(double), double x, double h) {
    // Implementamos la definición de la derivada: (f(x+h) - f(x)) / h
    double fx_mas_h = func_ptr(x + h);
    double fx = func_ptr(x);

    return (fx_mas_h - fx) / h;
}

int main() {
    // Definimos nuestro "infinitesimal" h. Un valor pequeño pero no tan pequeño
    // como para causar problemas de precisión de punto flotante.
    const double h = 1e-5; // 0.00001

    printf("Analizando la función f(x) = x^2\n");
    printf("---------------------------------\n");
```

```
    // --- Caso 1: Un punto con pendiente positiva ---
    double punto1 = 5.0;
    double pendiente1 = calcular_derivada_numerica(&parabola, punto1, h);
    printf("La pendiente en x = %f es aproximadamente: %f\n", punto1, pendiente1);
    // Analíticamente, la derivada de x^2 es 2x. En x=5, f'(5) = 2*5 = 10.

    // --- Caso 2: Un punto con pendiente negativa ---
    double punto2 = -4.0;
    double pendiente2 = calcular_derivada_numerica(&parabola, punto2, h);
    printf("La pendiente en x = %f es aproximadamente: %f\n", punto2, pendiente2);
    // Analíticamente, f'(-4) = 2*(-4) = -8.

    // --- Caso 3: El mínimo de la función ---
    double punto3 = 0.0;
    double pendiente3 = calcular_derivada_numerica(&parabola, punto3, h);
    printf("La pendiente en x = %f es aproximadamente: %f\n", punto3, pendiente3);
    // Analíticamente, f'(0) = 2*0 = 0.

    printf("\n");
    printf("Una pendiente de ~0 indica un punto crítico (mínimo, máximo o de inflexión).\n");
    printf("¡Hemos encontrado el fondo del valle de la parábola!\n");

    return 0;
}
```

Interpretación de los resultados

Al ejecutar el código anterior, verás que los resultados numéricos son extremadamente cercanos a los resultados analíticos (que calculamos a mano usando la regla de que la derivada de x2 es 2x):

- En `x = 5.0`, la pendiente es `~10.0`. Es positiva y grande, lo que nos dice que la función está subiendo rápidamente.
- En `x = -4.0`, la pendiente es `~-8.0`. Es negativa, lo que nos dice que la función está bajando.
- En `x = 0.0`, la pendiente es `~0.00001`, un número muy cercano a cero. Esto es clave: **una pendiente de cero nos indica que hemos llegado a un punto plano**. En el caso de la parábola, es su punto más bajo, el **mínimo**.

¡Acabas de implementar una de las herramientas más poderosas del aprendizaje automático! Esta simple función `calcular_derivada_numerica` es nuestra brújula. Nos permite, en

cualquier punto de nuestro paisaje de error, preguntar: "¿Hacia dónde está la bajada?". La respuesta nos la da el signo de la derivada.

En el siguiente apartado, extenderemos esta idea a funciones con múltiples variables, lo que nos llevará al concepto del **gradiente**.

1.3. La Regla de la Cadena: La Columna Vertebral del *Backpropagation*

Hasta ahora, hemos tratado con funciones simples. Pero una red neuronal no es más que una función extremadamente compleja, compuesta por muchas funciones más pequeñas anidadas unas dentro de otras (una capa de la red procesa la salida de la capa anterior).

Si queremos saber cómo un pequeño ajuste en un peso al principio de la red afecta al error final, no podemos usar nuestra función `calcular_derivada_numerica` directamente. Sería computacionalmente desastroso. Necesitamos una forma analítica de "desenrollar" estas funciones anidadas. Esa forma es la **Regla de la Cadena**.

La Regla de la Cadena: Descomponiendo el Cambio

La Regla de la Cadena nos dice cómo calcular la derivada de una función compuesta. Si tenemos una variable final `z` que depende de una variable intermedia `y`, y `y` a su vez depende de una variable inicial `x`, la regla nos dice que el cambio de `z` con respecto a `x` es:

En español: "El impacto de `x` en `z` es igual al impacto de `y` en `z` multiplicado por el impacto de `x` en `y`".

Este principio es la base del algoritmo de **backpropagation** (retropropagación). En una red neuronal, calculamos el error al final de la cadena de funciones. Luego, usamos la regla de la cadena para propagar ese error "hacia atrás", capa por capa, calculando la contribución de cada peso y sesgo al error total.

Estructurando un Grafo Computacional en C

Para implementar la regla de la cadena de forma sistemática, primero necesitamos una manera de representar la secuencia de operaciones. Lo haremos con un **grafo computacional**. Cada nodo de nuestro grafo representará un valor (un número) y la operación que lo generó.

En C, una `struct` es perfecta para definir nuestros nodos. Cada nodo necesitará:

1. `valor`: El resultado numérico de la operación.
2. `grad`: El gradiente (la derivada) de la salida final del grafo con respecto a este nodo. Lo inicializaremos a `0.0`.
3. Punteros a sus "padres": los nodos que actuaron como entradas para su operación.
4. Una función para realizar la retropropagación.

```
#include <stdio.h>
#include <stdlib.h>

// Definimos la estructura para un nodo en nuestro grafo computacional.
typedef struct Nodo {
    double valor;
    double grad;
    void (*backprop)(struct Nodo*); // Puntero a su función de backprop
    struct Nodo* padre1;
    struct Nodo* padre2;
} Nodo;
```

Ahora, creemos funciones que realicen operaciones (como sumar o multiplicar) y que construyan este grafo. Estas funciones crearán un nuevo nodo, calcularán su `valor` y guardarán quiénes son sus padres.

Implementación en C: Forward y Backward Pass

Vamos a modelar una expresión simple: f=(x+y)*z. Esto se descompone en:

1. a=x+y
2. f=a*z

El "forward pass" es el cálculo del valor final de `f`. El "backward pass" es el cálculo de los gradientes.

Primero, las funciones de `backprop`. Si `f = a * z`, la regla de la cadena nos dice:

- El gradiente que fluye hacia `a` es `grad_de_f * z`.
- El gradiente que fluye hacia `z` es `grad_de_f * a`.

Y si `a = x + y`:

- El gradiente que fluye hacia `x` es `grad_de_a * 1`.
- El gradiente que fluye hacia `y` es `grad_de_a * 1`.

Implementémoslo:

```
// Declaraciones adelantadas para que las funciones se conozcan entre sí
void backprop_add(Nodo* nodo);
void backprop_multiply(Nodo* nodo);

// Función para crear un nodo que representa una entrada (una variable)
Nodo* crear_variable(double valor) {
```

```
    Nodo* n = (Nodo*)malloc(sizeof(Nodo));
    n->valor = valor;
    n->grad = 0.0;
    n->backprop = NULL; // Las variables no tienen función de backprop propia
    n->padre1 = NULL;
    n->padre2 = NULL;
    return n;
}

// Función para sumar dos nodos (FORWARD PASS)
Nodo* add(Nodo* n1, Nodo* n2) {
    Nodo* resultado = (Nodo*)malloc(sizeof(Nodo));
    resultado->valor = n1->valor + n2->valor;
    resultado->grad = 0.0;
    resultado->backprop = &backprop_add; // Asignamos la función de backprop correcta
    resultado->padre1 = n1;
    resultado->padre2 = n2;
    return resultado;
}

// Función para multiplicar dos nodos (FORWARD PASS)
Nodo* multiply(Nodo* n1, Nodo* n2) {
    Nodo* resultado = (Nodo*)malloc(sizeof(Nodo));
    resultado->valor = n1->valor * n2->valor;
    resultado->grad = 0.0;
    resultado->backprop = &backprop_multiply; // Asignamos la función de backprop correcta
    resultado->padre1 = n1;
    resultado->padre2 = n2;
    return resultado;
}

// BACKWARD PASS para la suma: a = x + y
void backprop_add(Nodo* nodo) {
    // El gradiente se distribuye con un factor de 1 a cada padre.
    nodo->padre1->grad += nodo->grad * 1.0;
    nodo->padre2->grad += nodo->grad * 1.0;
}

// BACKWARD PASS para la multiplicación: f = a * z
void backprop_multiply(Nodo* nodo) {
    // El gradiente se distribuye al padre1 multiplicado por el valor del padre2, y viceversa.
    nodo->padre1->grad += nodo->grad * nodo->padre2->valor;
    nodo->padre2->grad += nodo->grad * nodo->padre1->valor;
}
```

```
int main() {
    // --- FORWARD PASS ---
    // Construimos el grafo para f = (x + y) * z
    // con x=2, y=3, z=4
    Nodo* x = crear_variable(2.0);
    Nodo* y = crear_variable(3.0);
    Nodo* z = crear_variable(4.0);

    Nodo* a = add(x, y);      // a = 2 + 3 = 5
    Nodo* f = multiply(a, z); // f = 5 * 4 = 20

    printf("Forward Pass:\n");
    printf("x=%.2f, y=%.2f, z=%.2f\n", x->valor, y->valor, z->valor);
    printf("a = x + y = %.2f\n", a->valor);
    printf("f = a * z = %.2f\n\n", f->valor);

    // --- BACKWARD PASS ---
    // Iniciamos la retropropagación desde el nodo final.
    // La derivada de f con respecto a f es 1.
    f->grad = 1.0;

    printf("Backward Pass (calculando gradientes):\n");

    // Ejecutamos la retropropagación en orden topológico inverso.
    // 1. Desde f hacia a y z
    f->backprop(f);
    // 2. Desde a hacia x e y
    a->backprop(a);

    printf("df/df = %.2f (por definición)\n", f->grad);
    printf("df/dz = %.2f (debería ser a=5)\n", z->grad);
    printf("df/da = %.2f (debería ser z=4)\n", a->grad);
    printf("df/dx = %.2f (debería ser z=4)\n", x->grad);
    printf("df/dy = %.2f (debería ser z=4)\n", y->grad);

    // Liberar memoria
    free(x); free(y); free(z); free(a); free(f);

    return 0;
}
```

Interpretación de los Gradientes Finales

Al ejecutar este código, los gradientes finales nos dicen exactamente cuánto afectaría un pequeño cambio en cada variable de entrada al resultado final `f`:

- `df/dx = 4.0`: Si aumentas `x` en una cantidad muy pequeña `h`, `f` aumentará `4.0 * h`.
- `df/dy = 4.0`: Si aumentas `y` en una cantidad muy pequeña `h`, `f` aumentará `4.0 * h`.
- `df/dz = 5.0`: Si aumentas `z` en una cantidad muy pequeña `h`, `f` aumentará `5.0 * h`.

Hemos construido con éxito un sistema que no solo calcula un resultado, sino que también "entiende" la contribución de cada una de sus partes a ese resultado. Este es el mecanismo exacto que permite a una red neuronal saber si debe aumentar o disminuir cada uno de sus millones de pesos para mejorar su rendimiento.

Ahora que entendemos cómo derivar funciones de una y varias variables, estamos listos para el siguiente paso: el **gradiente**.

1.4. Gradientes: Navegando en Múltiples Dimensiones con Punteros

En las secciones anteriores, hemos construido una base sólida para entender cómo cambian las funciones. Sin embargo, los modelos de inteligencia artificial rara vez son funciones de una sola variable. Una red neuronal puede tener millones de parámetros (pesos y sesgos), y el "paisaje de error" es un hiperplano en millones de dimensiones.

Para navegar este paisaje complejo, necesitamos una generalización de la derivada: el **gradiente**.

Derivadas Parciales: Aislando el Impacto de una Variable

Cuando una función tiene múltiples entradas, como , no podemos hablar de "la" derivada. En su lugar, hablamos de **derivadas parciales**. Una derivada parcial mide cómo cambia la función cuando modificamos *una sola* de sus variables, manteniendo todas las demás constantes.

Se denota con el símbolo partial. Por ejemplo, para la función f(x,y)=x2+3y:

- La derivada parcial con respecto a x, denotada fracpartialfpartialx, nos dice cómo cambia f cuando solo movemos x. Tratamos `y` como si fuera una constante, por lo que fracpartialfpartialx=2x.
- La derivada parcial con respecto a y, denotada fracpartialfpartialy, nos dice cómo cambia f cuando solo movemos y. Tratamos `x` como si fuera una constante, por lo que fracpartialfpartialy=3.

El Vector Gradiente: La Brújula que Apunta al Máximo Crecimiento

El **gradiente** de una función, denotado como nablaf, no es más que un vector que agrupa todas las derivadas parciales de la función. Para nuestra función f(x,y), el gradiente es:

nablaf(x,y)=beginbmatrixfracpartialfpartialxfracpartialfpartialyendbmatrix=beginbmatrix2x3endbmatrix

Este vector tiene una propiedad mágica y fundamental: **en cualquier punto, el gradiente apunta en la dirección de máximo crecimiento de la función**. Es la brújula perfecta que nos dice hacia dónde está la subida más pronunciada.

Y si el gradiente nos muestra la subida más rápida, el **gradiente negativo** (−nablaf) nos muestra la bajada más rápida. Este es el secreto del Descenso del Gradiente: en cada paso, calculamos el gradiente y nos movemos en la dirección opuesta para minimizar el error.

Implementación en C: El Gradiente Numérico

Para implementar esto en C, representaremos un punto en un espacio multidimensional usando un array de `double` y su tamaño. La función que queremos analizar deberá aceptar este array como entrada.

Nuestra función para calcular el gradiente numérico, `calcular_gradiente_numerico`, hará lo siguiente:

1. Creará un vector (un array de `double`) para almacenar el gradiente.
2. Iterará sobre cada dimensión del punto de entrada.
3. Para cada dimensión `i`, calculará la derivada parcial fracpartialfpartialx_i usando el método de diferencias finitas que ya conocemos: perturbará solo la componente `i` del punto de entrada en una cantidad `h` y verá cómo cambia la salida de la función.
4. Almacenará este resultado en la componente `i` del vector gradiente.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Definimos una función de ejemplo con 2 variables: f(x, y) = x^2 + y^2
// Su mínimo está claramente en (0, 0).
double paraboloide(double* punto, int dimensiones) {
    // Asumimos que dimensiones es 2 para este ejemplo.
    double x = punto[0];
    double y = punto[1];
    return x * x + y * y;
}

/**
 * @brief Calcula el gradiente numérico de una función multivariable en un punto.
 * * @param func_ptr Puntero a la función a derivar. La función debe aceptar un array
 * de doubles (el punto) y un int (el número de dimensiones).
```

```
 * @param punto El punto (representado como un array) en el que calcular el gradiente.
 * @param dimensiones El número de variables de la función.
 * @param h Un valor muy pequeño para la diferenciación numérica.
 * @return Un nuevo array (asignado dinámicamente) que contiene el vector gradiente.
 * ¡El llamador es responsable de liberar esta memoria con free()!
 */
double* calcular_gradiente_numerico(double (*func_ptr)(double*, int), double* punto, int
dimensiones, double h) {
    // Asignamos memoria para el vector gradiente.
    double* gradiente = (double*)malloc(dimensiones * sizeof(double));
    if (gradiente == NULL) {
        perror("Fallo al asignar memoria para el gradiente");
        return NULL;
    }

    // Almacenamos el valor original de la función en el punto.
    double valor_original = func_ptr(punto, dimensiones);

    // Iteramos sobre cada dimensión para calcular la derivada parcial.
    for (int i = 0; i < dimensiones; i++) {
        // Guardamos el valor original de la componente actual.
        double temp_valor = punto[i];

        // Perturbamos la componente actual en una cantidad h.
        punto[i] = temp_valor + h;

        // Calculamos el valor de la función con la perturbación.
        double valor_perturbado = func_ptr(punto, dimensiones);

        // Calculamos la derivada parcial.
        double derivada_parcial = (valor_perturbado - valor_original) / h;
        gradiente[i] = derivada_parcial;

        // Restauramos el valor original de la componente para no afectar
        // el cálculo de la siguiente derivada parcial.
        punto[i] = temp_valor;
    }

    return gradiente;
}

void imprimir_vector(double* vector, int dimensiones, const char* nombre) {
    printf("%s: [ ", nombre);
    for (int i = 0; i < dimensiones; i++) {
```

```
        printf("%.4f ", vector[i]);
    }
    printf("]\n");
}

int main() {
    const int DIM = 2;
    const double h = 1e-5;

    // Punto de prueba: (3.0, 4.0)
    double punto_prueba[DIM] = {3.0, 4.0};

    printf("Analizando f(x,y) = x^2 + y^2 en el punto (3.0, 4.0)\n");

    // La derivada analítica de f es [2x, 2y].
    // En (3, 4), el gradiente debería ser [6, 8].

    double* gradiente = calcular_gradiente_numerico(&paraboloide, punto_prueba, DIM, h);

    if (gradiente) {
        imprimir_vector(punto_prueba, DIM, "Punto    ");
        imprimir_vector(gradiente, DIM, "Gradiente");

        printf("\nEl gradiente [6.0000 8.0000] nos dice que la subida más pronunciada está en esa
dirección.\n");
        printf("Para minimizar la función, debemos movernos en la dirección opuesta: [-6, -8].\n");
        printf("Esta dirección apunta directamente hacia el mínimo en (0,0).\n");

        // ¡No olvides liberar la memoria que asignamos!
        free(gradiente);
    }

    return 0;
}
```

Con esta función, `calcular_gradiente_numerico`, hemos creado una herramienta universal. Sin importar cuán compleja sea nuestra función de coste, siempre que podamos escribirla en C, ahora podemos encontrar la dirección para optimizarla.

Hemos completado la parte teórica del cálculo. Estamos listos para el gran final de este capítulo: usar el gradiente para implementar el algoritmo de Descenso del Gradiente.

1.5. Aplicación Práctica: El Descenso del Gradiente en C

Este es el momento de la verdad. Hemos aprendido a representar funciones, a calcular su pendiente (derivada) y a encontrar la dirección de máxima pendiente en múltiples dimensiones (gradiente). Ahora, vamos a utilizar esa brújula, el gradiente, para crear un algoritmo que pueda encontrar automáticamente el punto más bajo en cualquier paisaje de error: el **Descenso del Gradiente**.

La Función de Coste y el Algoritmo

En el aprendizaje automático, nuestro objetivo es minimizar una **función de coste** (o función de pérdida), que mide qué tan "equivocado" está nuestro modelo. El Descenso del Gradiente es un algoritmo iterativo que hace exactamente eso. Imagina que estás en una montaña en un día de niebla densa y quieres llegar al valle. ¿Qué harías? Darías un paso en la dirección en la que el suelo desciende más rápido, y repetirías el proceso.

El Descenso del Gradiente hace lo mismo matemáticamente. La regla de actualización en cada paso es muy simple:

```
nuevo_punto = punto_actual - tasa_de_aprendizaje *
gradiente_en_el_punto_actual
```

- **`punto_actual`**: Nuestra posición actual en el paisaje (los valores actuales de los pesos del modelo).
- **`gradiente_en_el_punto_actual`**: El vector que nos dice la dirección de la subida más pronunciada.
- **`tasa_de_aprendizaje` (learning rate)**: Un número pequeño (ej. 0.01) que controla el tamaño del paso que damos. Es crucial: si es muy grande, podríamos pasarnos del mínimo; si es muy pequeño, tardaremos demasiado en llegar.

El algoritmo completo es:

1. Empezar en un punto aleatorio (pesos iniciales aleatorios).
2. Repetir un número determinado de veces (iteraciones): a. Calcular el gradiente en el punto actual. b. Actualizar el punto dando un paso en la dirección opuesta al gradiente.
3. El punto final es nuestra mejor aproximación del mínimo.

Implementación en C: El Algoritmo Completo

Vamos a escribir una función en C que implemente este algoritmo. Reutilizará nuestra función `calcular_gradiente_numerico` de la sección anterior.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// --- Incluimos las funciones de la sección 1.4 ---
```

```
// Función de ejemplo: f(x, y) = x^2 + y^2
double paraboloide(double* punto, int dimensiones) {
    double x = punto[0];
    double y = punto[1];
    return x * x + y * y;
}

// Función para calcular el gradiente numérico
double* calcular_gradiente_numerico(double (*func_ptr)(double*, int), double* punto, int
dimensiones, double h) {
    double* gradiente = (double*)malloc(dimensiones * sizeof(double));
    if (gradiente == NULL) return NULL;

    double valor_original = func_ptr(punto, dimensiones);
    for (int i = 0; i < dimensiones; i++) {
        double temp_valor = punto[i];
        punto[i] = temp_valor + h;
        double valor_perturbado = func_ptr(punto, dimensiones);
        gradiente[i] = (valor_perturbado - valor_original) / h;
        punto[i] = temp_valor;
    }
    return gradiente;
}

void imprimir_vector(double* vector, int dimensiones, const char* nombre) {
    printf("%s: [ ", nombre);
    for (int i = 0; i < dimensiones; i++) {
        printf("%.4f ", vector[i]);
    }
    printf("]\n");
}

/**
 * @brief Ejecuta el algoritmo de Descenso del Gradiente para encontrar el mínimo de una
función.
 *
 * @param func_ptr Puntero a la función a minimizar.
 * @param punto_inicial El punto de partida (se modificará en el lugar).
 * @param dimensiones El número de variables de la función.
 * @param tasa_aprendizaje El tamaño del paso en cada iteración.
 * @param iteraciones El número de veces que se repetirá el proceso.
 */
```

```
void descenso_del_gradiente(double (*func_ptr)(double*, int), double* punto_actual, int
dimensiones, double tasa_aprendizaje, int iteraciones) {
    const double h = 1e-5;

    printf("--- Iniciando Descenso del Gradiente ---\n");
    printf("Tasa de aprendizaje: %.4f, Iteraciones: %d\n", tasa_aprendizaje, iteraciones);
    imprimir_vector(punto_actual, dimensiones, "Punto inicial");
    printf("----------------------------------------\n");

    for (int i = 0; i < iteraciones; i++) {
        // 1. Calcular el gradiente en el punto actual.
        double* gradiente = calcular_gradiente_numerico(func_ptr, punto_actual, dimensiones, h);
        if (gradiente == NULL) {
            printf("Error al calcular el gradiente. Abortando.\n");
            return;
        }

        // 2. Actualizar cada componente del punto.
        for (int j = 0; j < dimensiones; j++) {
            punto_actual[j] -= tasa_aprendizaje * gradiente[j];
        }

        // Imprimimos el progreso cada cierto número de iteraciones.
        if ((i + 1) % 10 == 0) {
            printf("Iteración %d -> ", i + 1);
            imprimir_vector(punto_actual, dimensiones, "Punto");
        }

        // ¡Importante! Liberamos la memoria del gradiente en cada paso.
        free(gradiente);
    }
    printf("----------------------------------------\n");
}

int main() {
    const int DIM = 2;

    // Punto de partida, podría ser aleatorio.
    double punto[DIM] = {10.0, -8.0};

    // Parámetros del algoritmo
    double tasa_aprendizaje = 0.1;
    int iteraciones = 50;
```

```
    // Ejecutamos el algoritmo
    descenso_del_gradiente(&paraboloide, punto, DIM, tasa_aprendizaje, iteraciones);

    printf("Proceso finalizado.\n");
    imprimir_vector(punto, DIM, "Mínimo encontrado");
    printf("El mínimo real es [0.0000 0.0000]\n");

    return 0;
}
```

Conclusión del Capítulo 1

¡Felicidades! Si has seguido todos los pasos, has logrado algo extraordinario. Has construido desde cero, en C, un algoritmo de optimización funcional capaz de encontrar el mínimo de una función matemática.

En este capítulo, hemos viajado desde la idea abstracta de una "función" hasta un programa concreto que la optimiza. Hemos aprendido que:

1. Los **punteros a función** nos dan la flexibilidad para tratar la lógica matemática como datos.
2. La **derivada** es la pendiente, nuestra guía para saber si subir o bajar.
3. El **gradiente** es la generalización de la derivada, nuestra brújula en paisajes de altas dimensiones.
4. El **Descenso del Gradiente** es el vehículo que, usando esa brújula, nos lleva hacia el valle más profundo.

Este proceso es, en esencia, el corazón del entrenamiento de la mayoría de los modelos de aprendizaje automático. Cuando un modelo "aprende", lo que realmente está haciendo es ejecutar una versión mucho más sofisticada de este algoritmo para minimizar su error ajustando sus millones de parámetros.

En el siguiente capítulo, cambiaremos de enfoque. Dejaremos el cálculo y nos adentraremos en el **Álgebra Lineal**, el lenguaje que nos permitirá manejar y transformar los datos en la escala masiva que la IA moderna requiere.

Capítulo 2: Álgebra Lineal - El Lenguaje de los Datos en Memoria

2.1. Los Ladrillos de la IA: Estructuras de Datos en C

En el capítulo anterior, dominamos el arte de la optimización. Ahora, cambiamos nuestro enfoque hacia el objeto de esa optimización: los **datos**. En la inteligencia artificial, los datos no

son solo números sueltos; son colecciones estructuradas que representan desde los píxeles de una imagen hasta los pesos de una red neuronal. El lenguaje para manipular estas colecciones de datos es el **Álgebra Lineal**.

Nuestro primer paso es forjar los "ladrillos" fundamentales con los que construiremos todo lo demás: estructuras en C para representar vectores y matrices.

¿Qué son y cómo representan datos del mundo real?

- **Escalares**: Un único número. En C, es simplemente una variable como `double` o `float`.
- **Vectores**: Una lista ordenada de números. Un vector puede representar muchas cosas: las características de un objeto (ej. [altura, peso, edad]), las coordenadas de un punto en el espacio, o una fila de píxeles en una imagen.
- **Matrices**: Una tabla o cuadrícula rectangular de números. Una matriz es una estructura increíblemente versátil: puede representar una imagen completa en escala de grises, los pesos que conectan dos capas de una red neuronal, o un lote de datos donde cada fila es un ejemplo y cada columna una característica.
- **Tensores**: Una generalización de los anteriores a más dimensiones. Un escalar es un tensor de rango 0, un vector es de rango 1, y una matriz es de rango 2. Una colección de imágenes en color (alto, ancho, canales de color) formaría un tensor de rango 3.

Para mantener nuestro código limpio y robusto, no usaremos simples arrays. Crearemos nuestras propias `struct` que contengan no solo los datos, sino también metadatos importantes como sus dimensiones.

Implementación en C: `struct Vector` y `struct Matrix`

Para un **vector**, necesitamos saber su tamaño y tener un puntero a sus datos.

```
typedef struct {
    int tamano;      // Número de elementos en el vector
    double* datos;   // Puntero a los datos (array dinámico)
} Vector;
```

Para una **matriz**, necesitamos saber su número de filas y columnas. Para mejorar el rendimiento y simplificar la gestión de memoria, almacenaremos todos sus datos en un único bloque de memoria contiguo, en lugar de un array de punteros. Accederemos a un elemento en la fila `i` y columna `j` con la fórmula `datos[i * columnas + j]`.

```
typedef struct {
    int filas;
    int columnas;
    double* datos;   // Puntero a un bloque de memoria contiguo
```

```
} Matriz;
```

Funciones de Utilidad: Creación y Liberación

Un principio fundamental al trabajar en C es que toda memoria que se solicita (`malloc`) debe ser devuelta (`free`). Para evitar fugas de memoria, crearemos funciones de utilidad que manejen la creación y destrucción de nuestras estructuras de forma segura.

```
#include <stdio.h>
#include <stdlib.h>

// Definición de las estructuras
typedef struct {
    int tamano;
    double* datos;
} Vector;

typedef struct {
    int filas;
    int columnas;
    double* datos;
} Matriz;

/**
 * @brief Crea un nuevo vector del tamaño especificado.
 * @param tamano El número de elementos del vector.
 * @return Un puntero al nuevo Vector, o NULL si falla la asignación.
 * @note Todos los elementos se inicializan a 0.0.
 */
Vector* crear_vector(int tamano) {
    Vector* v = (Vector*)malloc(sizeof(Vector));
    if (v == NULL) return NULL;

    v->tamano = tamano;
    v->datos = (double*)calloc(tamano, sizeof(double)); // calloc inicializa a cero
    if (v->datos == NULL) {
        free(v); // Liberamos la estructura si falla la asignación de datos
        return NULL;
    }
    return v;
}

/**
```

```
 * @brief Libera la memoria ocupada por un vector.
 */
void liberar_vector(Vector* v) {
    if (v != NULL) {
        free(v->datos); // Primero liberamos el array de datos
        free(v);      // Luego liberamos la estructura
    }
}

/**
 * @brief Crea una nueva matriz de las dimensiones especificadas.
 * @return Un puntero a la nueva Matriz, o NULL si falla la asignación.
 * @note Todos los elementos se inicializan a 0.0.
 */
Matriz* crear_matriz(int filas, int columnas) {
    Matriz* m = (Matriz*)malloc(sizeof(Matriz));
    if (m == NULL) return NULL;

    m->filas = filas;
    m->columnas = columnas;
    m->datos = (double*)calloc(filas * columnas, sizeof(double));
    if (m->datos == NULL) {
        free(m);
        return NULL;
    }
    return m;
}

/**
 * @brief Libera la memoria ocupada por una matriz.
 */
void liberar_matriz(Matriz* m) {
    if (m != NULL) {
        free(m->datos);
        free(m);
    }
}

// --- Funciones de utilidad para imprimir ---
void imprimir_vector(const Vector* v) {
    if (v == NULL) return;
    printf("Vector (tamaño %d): [ ", v->tamano);
    for (int i = 0; i < v->tamano; i++) {
        printf("%.2f ", v->datos[i]);
```

```
    }
    printf("]\n");
}

void imprimir_matriz(const Matriz* m) {
    if (m == NULL) return;
    printf("Matriz (%d x %d):\n", m->filas, m->columnas);
    for (int i = 0; i < m->filas; i++) {
        printf("[ ");
        for (int j = 0; j < m->columnas; j++) {
            printf("%6.2f ", m->datos[i * m->columnas + j]);
        }
        printf("]\n");
    }
}

int main() {
    // --- Demostración del Vector ---
    printf("--- Creando un vector ---\n");
    Vector* mi_vector = crear_vector(5);
    mi_vector->datos[1] = 10.5;
    mi_vector->datos[3] = -3.2;
    imprimir_vector(mi_vector);

    // --- Demostración de la Matriz ---
    printf("\n--- Creando una matriz ---\n");
    Matriz* mi_matriz = crear_matriz(3, 4);
    mi_matriz->datos[0 * 4 + 1] = 5.0;  // Fila 0, Col 1
    mi_matriz->datos[1 * 4 + 3] = -9.9; // Fila 1, Col 3
    mi_matriz->datos[2 * 4 + 0] = 1.1;  // Fila 2, Col 0
    imprimir_matriz(mi_matriz);

    // --- Liberando memoria ---
    printf("\n--- Liberando memoria ---\n");
    liberar_vector(mi_vector);
    liberar_matriz(mi_matriz);
    printf("Memoria liberada correctamente.\n");

    return 0;
}
```

Con estas robustas estructuras de datos y sus funciones de gestión, hemos sentado las bases del Álgebra Lineal en C. Hemos reemplazado los arrays básicos por "objetos" inteligentes que conocen sus propias dimensiones y cuya memoria podemos gestionar de forma segura.

En la siguiente sección, daremos vida a estos ladrillos implementando las operaciones fundamentales que nos permitirán transformar los datos, exactamente como lo hace una red neuronal.

2.2. Operaciones Clave para Redes Neuronales en C

Con nuestros "ladrillos" (`Vector` y `Matriz`) listos, es hora de construir. En esta sección, implementaremos las operaciones que forman el corazón computacional de casi todas las redes neuronales. Estas no son meras rutinas matemáticas; son las acciones que permiten a los datos fluir, transformarse y, en última instancia, generar predicciones.

El Producto Escalar (Dot Product): La Operación Más Importante

Si tuvieras que elegir una única operación matemática que defina las redes neuronales, sería el producto escalar (o producto punto). Es una operación simple entre dos vectores del mismo tamaño que produce un único número (un escalar).

Cálculo: Se calcula multiplicando los elementos correspondientes de los dos vectores y sumando todos los resultados. Para V=[v_1,v_2,...,v_n] y W=[w_1,w_2,...,w_n], el producto escalar es .

Significado en IA:

- **Cálculo de la salida de una neurona:** La operación principal que realiza una neurona es un producto escalar entre su vector de pesos y el vector de entradas, seguido de la adición de un sesgo y una función de activación.
- **Medida de similitud:** El producto escalar está relacionado con el ángulo entre dos vectores. Un producto escalar grande y positivo indica que los vectores apuntan en una dirección similar.

Multiplicación de Matrices: Transformando Datos Capa por Capa

Mientras que el producto escalar opera sobre vectores, la multiplicación de matrices nos permite realizar transformaciones complejas a conjuntos enteros de datos. Cuando los datos de entrada pasan de una capa de una red neuronal a la siguiente, lo que ocurre matemáticamente es una multiplicación de matrices: el vector de activaciones de la capa de entrada se multiplica por la matriz de pesos de la capa.

Cálculo: La multiplicación de dos matrices, A y B, para producir una matriz C (C=AtimesB) es más compleja. Para que sea posible, el **número de columnas de la matriz** A **debe ser igual**

al número de filas de la matriz B. El elemento en la fila `i` y columna `j` de la matriz resultante C se calcula haciendo el producto escalar del vector fila `i` de A con el vector columna `j` de B.

Significado en IA:

- **Transformación Lineal:** Es la forma de aplicar una transformación lineal (rotar, escalar, sesgar) a los datos.
- **Procesamiento en Lote:** Permite procesar múltiples ejemplos de datos (cada uno una fila en una matriz de entrada) simultáneamente.

Implementación en C

Ahora, traduciremos estas operaciones a funciones en C, utilizando las estructuras que definimos en la sección anterior.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h> // Usaremos assert para validaciones críticas

// --- Estructuras y funciones de utilidad de la sección 2.1 ---
typedef struct {
    int tamano;
    double* datos;
} Vector;

typedef struct {
    int filas;
    int columnas;
    double* datos;
} Matriz;

Vector* crear_vector(int tamano) {
    Vector* v = (Vector*)malloc(sizeof(Vector));
    if (v == NULL) return NULL;
    v->tamano = tamano;
    v->datos = (double*)calloc(tamano, sizeof(double));
    if (v->datos == NULL) { free(v); return NULL; }
    return v;
}

void liberar_vector(Vector* v) {
    if (v != NULL) { free(v->datos); free(v); }
}

Matriz* crear_matriz(int filas, int columnas) {
```

```
    Matriz* m = (Matriz*)malloc(sizeof(Matriz));
    if (m == NULL) return NULL;
    m->filas = filas;
    m->columnas = columnas;
    m->datos = (double*)calloc(filas * columnas, sizeof(double));
    if (m->datos == NULL) { free(m); return NULL; }
    return m;
}

void liberar_matriz(Matriz* m) {
    if (m != NULL) { free(m->datos); free(m); }
}

void imprimir_vector(const Vector* v) {
    if (v == NULL) return;
    printf("Vector (tamaño %d): [ ", v->tamano);
    for (int i = 0; i < v->tamano; i++) { printf("%.2f ", v->datos[i]); }
    printf("]\n");
}

void imprimir_matriz(const Matriz* m) {
    if (m == NULL) return;
    printf("Matriz (%d x %d):\n", m->filas, m->columnas);
    for (int i = 0; i < m->filas; i++) {
        printf("[ ");
        for (int j = 0; j < m->columnas; j++) {
            printf("%6.2f ", m->datos[i * m->columnas + j]);
        }
        printf("]\n");
    }
}
// --- Fin de las funciones de utilidad ---


/**
 * @brief Calcula el producto escalar de dos vectores.
 * @return El resultado escalar.
 */
double producto_escalar(const Vector* v1, const Vector* v2) {
    // La operación solo es válida si los vectores tienen el mismo tamaño.
    assert(v1->tamano == v2->tamano && "Los vectores deben tener el mismo tamaño para el
producto escalar.");

    double resultado = 0.0;
```

```
    for (int i = 0; i < v1->tamano; i++) {
        resultado += v1->datos[i] * v2->datos[i];
    }
    return resultado;
}

/**
 * @brief Multiplica dos matrices (A x B).
 * @return Un puntero a una nueva Matriz con el resultado, o NULL si la operación no es válida.
 */
Matriz* multiplicar_matrices(const Matriz* A, const Matriz* B) {
    // El número de columnas de A debe ser igual al número de filas de B.
    assert(A->columnas == B->filas && "Dimensiones inválidas para la multiplicación de
matrices.");

    // La matriz resultante tendrá las filas de A y las columnas de B.
    Matriz* C = crear_matriz(A->filas, B->columnas);
    if (C == NULL) return NULL;

    // Bucle principal para la multiplicación
    for (int i = 0; i < A->filas; i++) {
        for (int j = 0; j < B->columnas; j++) {
            double suma = 0.0;
            for (int k = 0; k < A->columnas; k++) {
                // El elemento C(i,j) es el producto escalar de la fila i de A y la columna j de B.
                suma += A->datos[i * A->columnas + k] * B->datos[k * B->columnas + j];
            }
            C->datos[i * C->columnas + j] = suma;
        }
    }
    return C;
}

int main() {
    // --- Demostración del Producto Escalar ---
    printf("--- Producto Escalar ---\n");
    Vector* v1 = crear_vector(3);
    v1->datos[0] = 1; v1->datos[1] = 2; v1->datos[2] = 3;

    Vector* v2 = crear_vector(3);
    v2->datos[0] = 4; v2->datos[1] = -5; v2->datos[2] = 6;

    imprimir_vector(v1);
    imprimir_vector(v2);
```

```
    double resultado_pe = producto_escalar(v1, v2);
    printf("Resultado del producto escalar: %.2f\n", resultado_pe); // 1*4 + 2*(-5) + 3*6 = 4 - 10 +
18 = 12

    // --- Demostración de la Multiplicación de Matrices ---
    printf("\n--- Multiplicación de Matrices ---\n");
    Matriz* A = crear_matriz(2, 3);
    A->datos[0*3+0]=1; A->datos[0*3+1]=2; A->datos[0*3+2]=3;
    A->datos[1*3+0]=4; A->datos[1*3+1]=5; A->datos[1*3+2]=6;

    Matriz* B = crear_matriz(3, 2);
    B->datos[0*2+0]=7; B->datos[0*2+1]=8;
    B->datos[1*2+0]=9; B->datos[1*2+1]=10;
    B->datos[2*2+0]=11; B->datos[2*2+1]=12;

    printf("Matriz A:\n");
    imprimir_matriz(A);
    printf("Matriz B:\n");
    imprimir_matriz(B);

    Matriz* C = multiplicar_matrices(A, B);
    printf("Resultado de A x B:\n");
    imprimir_matriz(C);

    // --- Liberando toda la memoria ---
    liberar_vector(v1);
    liberar_vector(v2);
    liberar_matriz(A);
    liberar_matriz(B);
    liberar_matriz(C);

    return 0;
}
```

Con estas dos funciones, `producto_escalar` y `multiplicar_matrices`, ya tenemos el motor computacional necesario para una red neuronal simple. Hemos dado vida a nuestras estructuras de datos, permitiéndoles interactuar y transformar la información.

En la siguiente sección, exploraremos otras operaciones matriciales importantes, como la transposición, que son cruciales para arquitecturas de IA más modernas como los *Transformers*.

2.3. Propiedades y Operaciones Matriciales Esenciales en C

Además de la multiplicación, existen otras operaciones y tipos de matrices que son fundamentales en el álgebra lineal y tienen aplicaciones directas en la IA. En esta sección, implementaremos algunas de estas operaciones clave y discutiremos conceptos importantes que, aunque no implementemos por completo, son cruciales para una comprensión profunda del campo.

La Matriz Identidad y la Matriz Inversa

- **La Matriz Identidad ()**: Es el equivalente matricial del número 1. Es una matriz cuadrada (mismo número de filas que de columnas) que tiene unos en su diagonal principal y ceros en todas las demás posiciones. Su propiedad fundamental es que para cualquier matriz A, se cumple que AtimesI=ItimesA=A. Es decir, multiplicar por la identidad no altera la matriz original.
- **La Matriz Inversa (**A−1**)**: Para una matriz cuadrada A, su inversa A−1 es aquella que, al multiplicarla por A, da como resultado la matriz identidad: AtimesA−1=A−1timesA=I. La inversión de matrices es crucial para resolver sistemas de ecuaciones lineales. Sin embargo, **no todas las matrices tienen una inversa**. Además, su cálculo es computacionalmente costoso y numéricamente sensible. Algoritmos como la eliminación de Gauss-Jordan o la descomposición LU se utilizan para este fin, pero su implementación robusta está fuera del alcance de este capítulo introductorio. Es importante conocer su existencia, pero en la práctica del *deep learning*, rara vez se calculan inversas explícitas, prefiriendo en su lugar métodos de optimización como el descenso del gradiente.

La Transposición de Matrices: Fundamental para los *Transformers*

Esta es una de las operaciones más importantes en la IA moderna. La **transposición** de una matriz A consiste en intercambiar sus filas por sus columnas, resultando en una nueva matriz denotada como AT. Si A tiene dimensiones mtimesn, su transpuesta AT tendrá dimensiones ntimesm.

Significado en IA:

- **Reorientación de datos:** Permite cambiar la orientación de los datos para que las dimensiones sean compatibles en operaciones como la multiplicación.
- **Mecanismo de Atención:** Es la piedra angular del mecanismo de *self-attention* en la arquitectura **Transformer** (la base de modelos como GPT). En este mecanismo, se calcula una puntuación de similitud entre diferentes partes de una secuencia de entrada mediante la operación QtimesKT, donde Q es la matriz de "consultas" (*queries*) y KT es la **transpuesta** de la matriz de "claves" (*keys*). Esta operación no sería posible sin la transposición para alinear las dimensiones correctamente.

Implementación en C

Añadamos las funciones para crear una matriz identidad y para transponer una matriz existente a nuestro conjunto de herramientas.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// --- Estructuras y funciones de utilidad de secciones anteriores ---
typedef struct {
    int tamano;
    double* datos;
} Vector;

typedef struct {
    int filas;
    int columnas;
    double* datos;
} Matriz;

Vector* crear_vector(int tamano) {
    Vector* v = (Vector*)malloc(sizeof(Vector));
    if (v == NULL) return NULL;
    v->tamano = tamano;
    v->datos = (double*)calloc(tamano, sizeof(double));
    if (v->datos == NULL) { free(v); return NULL; }
    return v;
}

void liberar_vector(Vector* v) {
    if (v != NULL) { free(v->datos); free(v); }
}

Matriz* crear_matriz(int filas, int columnas) {
    Matriz* m = (Matriz*)malloc(sizeof(Matriz));
    if (m == NULL) return NULL;
    m->filas = filas;
    m->columnas = columnas;
    m->datos = (double*)calloc(filas * columnas, sizeof(double));
    if (m->datos == NULL) { free(m); return NULL; }
    return m;
}

void liberar_matriz(Matriz* m) {
    if (m != NULL) { free(m->datos); free(m); }
```

```
}

void imprimir_matriz(const Matriz* m) {
    if (m == NULL) return;
    printf("Matriz (%d x %d):\n", m->filas, m->columnas);
    for (int i = 0; i < m->filas; i++) {
        printf("[ ");
        for (int j = 0; j < m->columnas; j++) {
            printf("%6.2f ", m->datos[i * m->columnas + j]);
        }
        printf("]\n");
    }
}
// --- Fin de las funciones de utilidad ---


/**
 * @brief Crea una matriz identidad de un tamaño dado.
 * @param tamano El número de filas y columnas de la matriz cuadrada.
 * @return Un puntero a la nueva Matriz identidad.
 */
Matriz* crear_matriz_identidad(int tamano) {
    Matriz* I = crear_matriz(tamano, tamano);
    if (I == NULL) return NULL;

    // Colocamos 1.0 en la diagonal principal
    for (int i = 0; i < tamano; i++) {
        I->datos[i * tamano + i] = 1.0;
    }
    return I;
}

/**
 * @brief Calcula la transpuesta de una matriz.
 * @param A La matriz original.
 * @return Un puntero a una nueva Matriz que es la transpuesta de A.
 */
Matriz* transponer_matriz(const Matriz* A) {
    // La matriz transpuesta tiene las dimensiones intercambiadas.
    Matriz* At = crear_matriz(A->columnas, A->filas);
    if (At == NULL) return NULL;

    for (int i = 0; i < A->filas; i++) {
        for (int j = 0; j < A->columnas; j++) {
```

```
            // El elemento (i, j) de A se convierte en el elemento (j, i) de At.
            At->datos[j * At->columnas + i] = A->datos[i * A->columnas + j];
        }
    }
    return At;
}


int main() {
    // --- Demostración de la Matriz Identidad ---
    printf("--- Matriz Identidad ---\n");
    Matriz* I = crear_matriz_identidad(4);
    imprimir_matriz(I);

    // --- Demostración de la Transposición ---
    printf("\n--- Transposición de Matriz ---\n");
    Matriz* A = crear_matriz(2, 3);
    // Llenamos la matriz A con algunos valores
    for (int i = 0; i < 2 * 3; i++) {
        A->datos[i] = i + 1;
    }

    printf("Matriz Original A (2x3):\n");
    imprimir_matriz(A);

    Matriz* At = transponer_matriz(A);
    printf("Matriz Transpuesta At (3x2):\n");
    imprimir_matriz(At);

    // --- Liberando toda la memoria ---
    liberar_matriz(I);
    liberar_matriz(A);
    liberar_matriz(At);

    return 0;
}
```

Con la adición de la transposición, nuestro conjunto de herramientas de álgebra lineal es ahora mucho más potente y está preparado para abordar arquitecturas más complejas. Hemos establecido una base sólida para manipular datos en casi cualquier forma que una red neuronal pueda requerir.

En la siguiente sección, concluiremos nuestra exploración del álgebra lineal discutiendo conceptos como los eigenvalores y la descomposición de matrices, que son fundamentales

para técnicas de reducción de dimensionalidad como el Análisis de Componentes Principales (PCA).

2.4. Descomposición de Matrices y Reducción de Dimensionalidad

En las secciones anteriores, construimos las herramientas para operar con matrices. Ahora, concluiremos nuestro viaje por el álgebra lineal explorando algunos de sus conceptos más profundos. Aunque la implementación completa de estos algoritmos en C es extremadamente compleja y propia de librerías numéricas especializadas, entender su propósito es crucial para comprender técnicas avanzadas de IA, especialmente en el campo del *unsupervised learning* (aprendizaje no supervisado) y el preprocesamiento de datos.

Espacios Vectoriales y Transformaciones Lineales

Imagina que todos los vectores posibles de un cierto tamaño (ej. todos los vectores de 2 dimensiones) viven en un "universo" llamado **espacio vectorial**. Cuando multiplicamos un vector por una matriz, estamos realizando una **transformación lineal**: estamos moviendo ese vector a una nueva posición dentro de su universo de una manera estructurada (las líneas rectas permanecen rectas y el origen no se mueve). La matriz de transformación codifica esta acción de "mover", "rotar", "escalar" o "sesgar" el espacio.

Eigenvalores y Eigenvectores: La Esencia de la Transformación

Dentro de todo este movimiento, a menudo existen vectores especiales. Para una matriz de transformación dada, sus **eigenvectores** son aquellos vectores que, después de la transformación, no cambian su dirección; solo son escalados. El factor por el cual son escalados se llama **eigenvalor**.

- **Eigenvector**: Un vector cuya dirección no es alterada por una transformación lineal.
- **Eigenvalor ()**: El factor de escala asociado a un eigenvector.

Matemáticamente, para una matriz A y su eigenvector v, se cumple que: Acdotv=lambdacdotv.

Significado en IA: Los eigenvectores de una matriz (como una matriz de covarianza de datos) nos revelan los "ejes de mayor importancia" o las direcciones de mayor varianza en los datos. Esta es la idea central detrás de una de las técnicas de reducción de dimensionalidad más famosas: el Análisis de Componentes Principales (PCA).

Descomposición de Matrices (SVD y PCA)

La **descomposición de matrices** es el proceso de "factorizar" una matriz en el producto de varias matrices más simples. Es análogo a factorizar el número 12 en 2times2times3.

- **Análisis de Componentes Principales (PCA)**: Es un algoritmo que utiliza los eigenvectores de la matriz de covarianza de los datos para encontrar un nuevo sistema de coordenadas. Los ejes de este nuevo sistema (llamados componentes principales)

están ordenados por la cantidad de varianza de los datos que explican. Al quedarnos solo con los primeros componentes principales, podemos reducir drásticamente la dimensionalidad de nuestros datos perdiendo la menor cantidad de información posible.
- **Descomposición en Valores Singulares (SVD)**: Es una técnica de descomposición aún más general y poderosa. Cualquier matriz A puede ser descompuesta en el producto de tres matrices: A=USigmaVT. Las matrices U y V son ortogonales (representan rotaciones) y Sigma es una matriz diagonal que contiene los "valores singulares". SVD es la base matemática sobre la que se construye PCA y tiene innumerables aplicaciones, desde la compresión de imágenes hasta los sistemas de recomendación.

La implementación de SVD o PCA desde cero es una tarea monumental. Sin embargo, podemos ilustrar el concepto de descomposición implementando una más sencilla: la **Descomposición LU**.

Implementación en C: Descomposición LU

La descomposición LU factoriza una matriz cuadrada A en el producto de una matriz triangular inferior (L, *Lower*) y una matriz triangular superior (U, *Upper*): A=LU. Es un paso fundamental para resolver sistemas de ecuaciones lineales de forma eficiente.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

// --- Estructuras y funciones de utilidad de secciones anteriores ---
typedef struct {
    int filas;
    int columnas;
    double* datos;
} Matriz;

Matriz* crear_matriz(int filas, int columnas) {
    Matriz* m = (Matriz*)malloc(sizeof(Matriz));
    if (m == NULL) return NULL;
    m->filas = filas;
    m->columnas = columnas;
    m->datos = (double*)calloc(filas * columnas, sizeof(double));
    if (m->datos == NULL) { free(m); return NULL; }
    return m;
}

void liberar_matriz(Matriz* m) {
    if (m != NULL) { free(m->datos); free(m); }
}
```

```
void imprimir_matriz(const Matriz* m) {
    if (m == NULL) return;
    printf("Matriz (%d x %d):\n", m->filas, m->columnas);
    for (int i = 0; i < m->filas; i++) {
        printf("[ ");
        for (int j = 0; j < m->columnas; j++) {
            printf("%8.4f ", m->datos[i * m->columnas + j]);
        }
        printf("]\n");
    }
     printf("\n");
}
// --- Fin de las funciones de utilidad ---

/**
 * @brief Realiza la descomposición LU de una matriz cuadrada A.
 * @param A La matriz a descomponer.
 * @param L Puntero a una Matriz donde se almacenará la matriz triangular inferior.
 * @param U Puntero a una Matriz donde se almacenará la matriz triangular superior.
 * @note Esta es una implementación simple del algoritmo de Doolittle.
 */
void descomposicion_lu(const Matriz* A, Matriz* L, Matriz* U) {
    assert(A->filas == A->columnas && "La matriz debe ser cuadrada para la descomposición
LU.");
    int n = A->filas;

    for (int i = 0; i < n; i++) {
        // Matriz U (Superior)
        for (int k = i; k < n; k++) {
            double suma = 0.0;
            for (int j = 0; j < i; j++) {
                suma += (L->datos[i * n + j] * U->datos[j * n + k]);
            }
            U->datos[i * n + k] = A->datos[i * n + k] - suma;
        }

        // Matriz L (Inferior)
        L->datos[i * n + i] = 1.0; // Diagonal de L es 1
        for (int k = i + 1; k < n; k++) {
            double suma = 0.0;
            for (int j = 0; j < i; j++) {
                suma += (L->datos[k * n + j] * U->datos[j * n + i]);
            }
```

```
            L->datos[k * n + i] = (A->datos[k * n + i] - suma) / U->datos[i * n + i];
        }
    }
}

int main() {
    printf("--- Descomposición LU ---\n");
    int n = 3;
    Matriz* A = crear_matriz(n, n);
    // Matriz de ejemplo
    A->datos[0*n+0] = 2; A->datos[0*n+1] = -1; A->datos[0*n+2] = -2;
    A->datos[1*n+0] = -4; A->datos[1*n+1] = 6; A->datos[1*n+2] = 3;
    A->datos[2*n+0] = -4; A->datos[2*n+1] = -2; A->datos[2*n+2] = 8;

    printf("Matriz Original A:\n");
    imprimir_matriz(A);

    Matriz* L = crear_matriz(n, n);
    Matriz* U = crear_matriz(n, n);

    descomposicion_lu(A, L, U);

    printf("Matriz Triangular Inferior L:\n");
    imprimir_matriz(L);
    printf("Matriz Triangular Superior U:\n");
    imprimir_matriz(U);

    // Liberando memoria
    liberar_matriz(A);
    liberar_matriz(L);
    liberar_matriz(U);

    return 0;
}
```

Conclusión del Capítulo 2

En este capítulo, hemos pasado de no tener nada a construir un completo (aunque básico) conjunto de herramientas de álgebra lineal en C. Hemos aprendido a:

1. Crear y gestionar de forma segura **estructuras de datos** para vectores y matrices.
2. Implementar las **operaciones fundamentales** como el producto escalar y la multiplicación de matrices.
3. Realizar operaciones clave como la **transposición**, vital para arquitecturas modernas.

4. Comprender la **intuición detrás de conceptos avanzados** como los eigenvalores y la descomposición de matrices, que son la base para el preprocesamiento de datos y la reducción de dimensionalidad.

Ahora posees el conocimiento para representar y manipular los datos. En los siguientes capítulos, combinaremos este conocimiento con las técnicas de optimización del Capítulo 1 para empezar a construir modelos de aprendizaje automático reales.

Parte II: Probabilidad y Estadística - El Arte de la Incertidumbre

Capítulo 3: Fundamentos de Probabilidad

3.1. Generando Aleatoriedad

Hasta ahora, nuestro universo ha sido determinista. Una operación siempre producía el mismo resultado. Sin embargo, el mundo real es inherentemente incierto y ruidoso. La inteligencia artificial no solo debe procesar datos, sino también cuantificar su confianza, manejar la ambigüedad y hacer predicciones en escenarios que nunca ha visto antes. Para todo esto, necesitamos el lenguaje de la **probabilidad**.

Nuestro primer paso en este nuevo mundo es aprender a generar lo impredecible, a simular el azar. En C, la librería estándar nos proporciona herramientas para generar números **pseudoaleatorios**. Se les llama "pseudoaleatorios" porque no son verdaderamente aleatorios (un ordenador es una máquina determinista), sino que son generados por un algoritmo que produce una secuencia de números que aparenta ser aleatoria y tiene buenas propiedades estadísticas.

El uso de `rand()` y `srand()`

Las dos funciones clave de la librería `<stdlib.h>` son:

- `int rand(void)`: Cada vez que se la llama, esta función devuelve un número entero pseudoaleatorio en el rango de `0` a `RAND_MAX` (una constante definida en `<stdlib.h>`, cuyo valor suele ser 32767 o mayor).
- `void srand(unsigned int seed)`: Esta función "siembra" o inicializa el generador de números pseudoaleatorios. El generador utiliza esta semilla (`seed`) como punto de partida para crear su secuencia.

Aquí está la regla de oro: para una misma semilla, la secuencia de números generada por `rand()` será **siempre la misma**. Esto es útil para la depuración (podemos reproducir

resultados), pero para una simulación real, queremos una secuencia diferente cada vez que ejecutamos el programa.

La solución común es sembrar el generador con un valor que sea diferente en cada ejecución. El candidato perfecto es la hora actual del sistema, que podemos obtener con la función `time(NULL)` de la librería `<time.h>`.

Simulando Eventos Probabilísticos

Para simular eventos del mundo real, como lanzar un dado (que tiene 6 caras) o una moneda (2 caras), podemos usar el operador módulo (`%`) para acotar el resultado de `rand()` al rango que necesitamos. Por ejemplo, `rand() % 6` nos dará un número entre 0 y 5. Si le sumamos 1, tendremos nuestro rango deseado de 1 a 6.

Veamos una implementación práctica.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>   // Necesaria para la función time()

/**
 * @brief Simula el lanzamiento de un dado de N caras.
 * @param caras El número de caras del dado.
 * @return Un número entero entre 1 y 'caras'.
 */
int lanzar_dado(int caras) {
    // rand() % caras -> nos da un número entre 0 y (caras - 1)
    // + 1            -> ajusta el rango a 1 hasta 'caras'
    return (rand() % caras) + 1;
}

/**
 * @brief Simula el lanzamiento de una moneda.
 * @return 0 para Cara (Heads), 1 para Cruz (Tails).
 */
int lanzar_moneda() {
    // rand() % 2 -> nos da 0 o 1
    return rand() % 2;
}

int main() {
    // Semillamos el generador de números aleatorios.
    // Usamos time(NULL) para asegurar que la semilla sea diferente
    // en cada ejecución del programa.
    // ¡Esto solo debe hacerse UNA VEZ al principio del programa!
```

```
    srand(time(NULL));

    printf("--- Simulación de Lanzamientos Aleatorios ---\n\n");

    // --- Simulación de Dados ---
    int num_lanzamientos_dado = 5;
    printf("Lanzando un dado de 6 caras %d veces:\n", num_lanzamientos_dado);
    for (int i = 0; i < num_lanzamientos_dado; i++) {
        printf("Lanzamiento %d: %d\n", i + 1, lanzar_dado(6));
    }

    printf("\n");

    // --- Simulación de Monedas ---
    int num_lanzamientos_moneda = 5;
    printf("Lanzando una moneda %d veces:\n", num_lanzamientos_moneda);
    for (int i = 0; i < num_lanzamientos_moneda; i++) {
        int resultado = lanzar_moneda();
        if (resultado == 0) {
            printf("Lanzamiento %d: Cara\n", i + 1);
        } else {
            printf("Lanzamiento %d: Cruz\n", i + 1);
        }
    }

    printf("\n--- Limitaciones ---\n");
    printf("El generador rand() es simple. Para aplicaciones científicas o criptográficas\n");
    printf("se requieren generadores de mayor calidad, pero para nuestros propósitos\n");
    printf("educativos, es perfectamente adecuado.\n");


    return 0;
}
```

Limitaciones y Siguientes Pasos

Es importante ser consciente de que `rand()` es un generador relativamente simple (a menudo, un Generador Lineal Congruencial o LCG). Aunque es suficiente para muchos propósitos, incluyendo los ejemplos de este libro, puede mostrar patrones o correlaciones no deseadas en simulaciones estadísticas a gran escala. Para aplicaciones criptográficas o científicas de alta precisión, se utilizan algoritmos más sofisticados como el Mersenne Twister.

Con esta capacidad para generar eventos aleatorios, hemos abierto la puerta a la simulación y al modelado de la incertidumbre. En las siguientes secciones, utilizaremos esta base para

explorar conceptos de probabilidad más profundos, como el Teorema de Bayes, y para generar muestras de distribuciones de probabilidad específicas, una tarea fundamental en el aprendizaje automático.

3.2. Probabilidad Condicional y Teorema de Bayes

En la sección anterior, aprendimos a simular el azar. Ahora, daremos un paso más allá: aprenderemos a **razonar sobre el azar** y, lo que es más importante, a **actualizar nuestras creencias cuando obtenemos nueva evidencia**. Esta es la esencia del aprendizaje. La herramienta matemática que nos permite hacer esto es el **Teorema de Bayes**.

Probabilidad Condicional: "¿Sabiendo esto, qué probabilidad hay de aquello?"

La probabilidad condicional nos da la probabilidad de que ocurra un evento A, **dado que ya sabemos que ha ocurrido** otro evento B. Se denota como y se lee como "la probabilidad de A dado B".

Intuitivamente, el conocimiento de que B ha ocurrido reduce nuestro "universo" de posibles resultados a solo aquellos en los que B es cierto. La probabilidad condicional es la fracción de ese nuevo universo en la que A también es cierto.

El Teorema de Bayes: La Base del Aprendizaje y la Inferencia

El Teorema de Bayes es una fórmula elegante que nos permite invertir la probabilidad condicional. Si conocemos la probabilidad de que ocurra una evidencia dado un evento (P(B|A)), el teorema nos permite calcular la probabilidad de que el evento haya ocurrido dada la evidencia (P(A|B)). Esta inversión es la base de la inferencia y el diagnóstico.

La fórmula es:

P(A|B)=fracP(B|A)cdotP(A)P(B)

Desglosemos sus componentes, que son la base del razonamiento bayesiano:

- P(A|B): **Probabilidad a Posteriori**. Es lo que queremos calcular. Es nuestra creencia actualizada sobre la hipótesis A después de haber observado la evidencia B.
- P(A): **Probabilidad a Priori**. Nuestra creencia inicial sobre la hipótesis A, antes de tener ninguna evidencia nueva.
- P(B|A): **Verosimilitud (Likelihood)**. La probabilidad de observar la evidencia B si nuestra hipótesis A es cierta. ¿Qué tan probable es que la evidencia que vimos sea generada por este evento?
- P(B): **Evidencia**. La probabilidad total de observar la evidencia B, bajo cualquier circunstancia. Se calcula sumando todas las formas en que B puede ocurrir: P(B)=P(B|A)cdotP(A)+P(B|negA)cdotP(negA).

Implementación en C: Un Ejemplo de Diagnóstico Médico

Imaginemos un escenario clásico para ilustrar el poder del Teorema de Bayes.

- Una enfermedad rara afecta al **1%** de la población.
- Existe una prueba para detectarla, pero no es perfecta:
    - Si una persona tiene la enfermedad, la prueba da positivo el **99%** de las veces (sensibilidad).
    - Si una persona está sana, la prueba da positivo el **5%** de las veces (tasa de falsos positivos).

Pregunta: Si una persona elegida al azar se hace la prueba y da **positivo**, ¿cuál es la probabilidad real de que tenga la enfermedad?

Vamos a usar C para calcular la respuesta.

```
#include <stdio.h>

/**
 * @brief Estructura para almacenar las probabilidades de nuestro problema.
 */
typedef struct {
    double prob_A;          // P(A): Probabilidad a priori de la hipótesis (tener la enfermedad)
    double prob_B_dado_A;   // P(B|A): Probabilidad de la evidencia si la hipótesis es cierta (test + | enfermo)
    double prob_B_dado_no_A; // P(B|~A): Probabilidad de la evidencia si la hipótesis es falsa (test + | sano)
} BayesProblema;

/**
 * @brief Calcula la probabilidad a posteriori usando el Teorema de Bayes.
 * @param problema La estructura que contiene todas las probabilidades necesarias.
 * @return La probabilidad a posteriori P(A|B).
 */
double teorema_bayes(const BayesProblema* problema) {
    // P(A)
    double p_A = problema->prob_A;
    // P(~A) = 1 - P(A)
    double p_no_A = 1.0 - p_A;

    // P(B|A)
    double p_B_dado_A = problema->prob_B_dado_A;
    // P(B|~A)
    double p_B_dado_no_A = problema->prob_B_dado_no_A;

    // Calculamos la evidencia total P(B)
    // P(B) = P(B|A)*P(A) + P(B|~A)*P(~A)
```

```
    double p_B = (p_B_dado_A * p_A) + (p_B_dado_no_A * p_no_A);

    // Aplicamos el Teorema de Bayes
    // P(A|B) = (P(B|A) * P(A)) / P(B)
    double p_A_dado_B = (p_B_dado_A * p_A) / p_B;

    return p_A_dado_B;
}

int main() {
    printf("--- Teorema de Bayes: Problema de Diagnóstico Médico ---\n\n");

    // Definimos las probabilidades de nuestro problema
    BayesProblema problema_medico = {
        .prob_A = 0.01,           // 1% de la población tiene la enfermedad
        .prob_B_dado_A = 0.99,    // 99% de los enfermos dan positivo (sensibilidad)
        .prob_B_dado_no_A = 0.05  // 5% de los sanos dan positivo (falsos positivos)
    };

    printf("Probabilidad a priori de tener la enfermedad P(Enfermo): %.2f%%\n",
problema_medico.prob_A * 100.0);
    printf("Probabilidad de un test positivo si se está enfermo P(Test+|Enfermo): %.2f%%\n",
problema_medico.prob_B_dado_A * 100.0);
    printf("Probabilidad de un test positivo si se está sano P(Test+|Sano): %.2f%%\n\n",
problema_medico.prob_B_dado_no_A * 100.0);

    // Calculamos el resultado
    double resultado_posterior = teorema_bayes(&problema_medico);

    printf("-----------------------------------------------------------------\n");
    printf("Pregunta: Si una persona da positivo, ¿cuál es la probabilidad de que esté
enferma?\n");
    printf("Resultado P(Enfermo|Test+): %.2f%%\n", resultado_posterior * 100.0);
    printf("-----------------------------------------------------------------\n\n");

    printf("Conclusión: A pesar de que la prueba parece muy precisa, la baja probabilidad\n");
    printf("inicial de la enfermedad hace que un resultado positivo solo nos dé un ~16.5%%
de\n");
    printf("certeza. ¡Nuestra intuición puede ser engañosa!\n");

    return 0;
}
```

El Poder del Razonamiento Bayesiano

El resultado es sorprendente y anti-intuitivo para muchos. Aunque la prueba es muy buena para detectar la enfermedad cuando está presente, la gran cantidad de personas sanas genera un número de falsos positivos que "ahoga" a los verdaderos positivos.

Este tipo de razonamiento es fundamental en IA. Un clasificador de spam, por ejemplo, utiliza el Teorema de Bayes para calcular la probabilidad de que un correo sea spam dada la presencia de ciertas palabras. Comienza con una probabilidad a priori (la tasa general de spam) y la actualiza basándose en la evidencia (las palabras en el correo).

Hemos aprendido a codificar una de las formas más fundamentales de aprendizaje. En la siguiente sección, exploraremos las distribuciones de probabilidad, que nos permitirán modelar diferentes tipos de datos y eventos aleatorios.

3.3. Variables Aleatorias y Distribuciones de Probabilidad

Hemos aprendido a simular eventos aleatorios individuales, pero en el mundo real, los datos a menudo siguen patrones predecibles. Las alturas de las personas, los errores de medición o los resultados de tirar un dado muchas veces no son completamente caóticos; se agrupan siguiendo una forma específica. Esta forma es descrita por una **distribución de probabilidad**.

Una **variable aleatoria** es una variable cuyo valor es un resultado numérico de un fenómeno aleatorio. Una **distribución de probabilidad** es una función matemática que describe la probabilidad de que una variable aleatoria tome cada uno de sus posibles valores.

Discretas vs. Continuas

Las distribuciones se dividen en dos grandes familias:

1. **Distribuciones Discretas**: La variable solo puede tomar un número finito o contable de valores distintos (ej. el resultado de un dado: {1, 2, 3, 4, 5, 6}).
    - **Bernoulli**: Representa un único experimento con dos posibles resultados (éxito/fracaso, cara/cruz, 1/0). Es la base de la clasificación binaria.
    - **Uniforme Discreta**: Todos los resultados posibles tienen exactamente la misma probabilidad (ej. un dado justo).
2. **Distribuciones Continuas**: La variable puede tomar cualquier valor dentro de un rango dado (ej. la altura de una persona).
    - **Uniforme Continua**: Cualquier valor dentro de un rango [a, b] tiene la misma probabilidad de ocurrir.
    - **Normal (o Gaussiana)**: La famosa "curva de campana". Es, con diferencia, la distribución más importante en estadística.

La Distribución Normal: ¿Por qué está en todas partes?

La distribución normal es omnipresente en la naturaleza y la estadística por el **Teorema del Límite Central**. Este teorema establece que si sumas un gran número de variables aleatorias independientes (sin importar su distribución original), la distribución de esa suma se aproximará a una distribución normal.

En IA, se utiliza para:

- Inicializar los pesos de las redes neuronales.
- Modelar el ruido en los datos.
- Como base para modelos probabilísticos más complejos.

Se define por dos parámetros: la **media** (), que es el centro de la campana, y la **desviación estándar** (sigma), que mide cuán ancha o estrecha es la campana.

Implementación en C: Generando Muestras

Ahora, vamos a escribir funciones en C para "muestrear" estas distribuciones, es decir, para generar números que sigan el patrón de cada una.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

/**
 * @brief Genera una muestra de una distribución uniforme discreta.
 * @param min El valor mínimo del rango (inclusivo).
 * @param max El valor máximo del rango (inclusivo).
 * @return Un entero aleatorio en el rango [min, max].
 */
int muestra_uniforme_discreta(int min, int max) {
    int rango = max - min + 1;
    return (rand() % rango) + min;
}

/**
 * @brief Genera una muestra de una distribución de Bernoulli.
 * @param p La probabilidad de éxito (un valor entre 0.0 y 1.0).
 * @return 1 (éxito) o 0 (fracaso).
 */
int muestra_bernoulli(double p) {
```

```
    // Generamos un número aleatorio entre 0.0 y 1.0
    double muestra_aleatoria = (double)rand() / RAND_MAX;
    if (muestra_aleatoria < p) {
        return 1; // Éxito
    } else {
        return 0; // Fracaso
    }
}

/**
 * @brief Genera una muestra de una distribución normal (gaussiana).
 * @param media (mu) El centro de la distribución.
 * @param desv_est (sigma) La desviación estándar de la distribución.
 * @return Un double aleatorio que sigue la distribución normal.
 * @note Utiliza la transformación de Box-Muller, un método estándar.
 */
double muestra_normal(double media, double desv_est) {
    // La transformación de Box-Muller requiere dos muestras uniformes.
    double u1 = (double)rand() / RAND_MAX;
    double u2 = (double)rand() / RAND_MAX;

    // Aplicamos la transformación.
    double z0 = sqrt(-2.0 * log(u1)) * cos(2.0 * M_PI * u2);

    // Ajustamos la muestra a la media y desviación estándar deseadas.
    return z0 * desv_est + media;
}

int main() {
    // Semillamos el generador una sola vez.
    srand(time(NULL));

    printf("--- Muestreo de Distribuciones de Probabilidad ---\n\n");

    // --- Uniforme Discreta (lanzar un dado) ---
    printf("10 muestras de una distribución Uniforme Discreta (1-6):\n");
    for (int i = 0; i < 10; i++) {
        printf("%d ", muestra_uniforme_discreta(1, 6));
    }
    printf("\n\n");

    // --- Bernoulli (moneda trucada, 70% de caras) ---
    printf("10 muestras de una distribución de Bernoulli (p=0.7):\n");
    int exitos = 0;
```

```
    for (int i = 0; i < 10; i++) {
        int resultado = muestra_bernoulli(0.7);
        if (resultado == 1) exitos++;
        printf("%d ", resultado);
    }
    printf("\n(Total de éxitos: %d)\n\n", exitos);

    // --- Normal (Gaussiana) ---
    printf("10 muestras de una distribución Normal (media=0, desv_est=1):\n");
    for (int i = 0; i < 10; i++) {
        printf("%.4f ", muestra_normal(0.0, 1.0));
    }
    printf("\n\n");

    printf("10 muestras de una distribución Normal (media=100, desv_est=15):\n");
    for (int i = 0; i < 10; i++) {
        printf("%.4f ", muestra_normal(100.0, 15.0));
    }
    printf("\n");

    return 0;
}
```

Valor Esperado y Varianza

Finalmente, dos conceptos clave para describir cualquier distribución:

- **Valor Esperado (o Media)**: Es el valor promedio que esperaríamos obtener si tomáramos un número infinito de muestras de la distribución. Es el centro de masa de la distribución.
- **Varianza**: Mide la dispersión de los datos alrededor de la media. Una varianza baja significa que los datos están muy agrupados; una varianza alta significa que están muy dispersos. La desviación estándar (sigma) es simplemente la raíz cuadrada de la varianza.

Conclusión del Capítulo 3

En este capítulo, hemos sentado las bases de la probabilidad. Hemos pasado de generar simple aleatoriedad a entender cómo actualizar nuestras creencias con el Teorema de Bayes y, finalmente, a modelar y generar datos que siguen patrones específicos a través de las distribuciones de probabilidad.

Estas herramientas son el puente hacia la **estadística**, que exploraremos en el siguiente capítulo. Usaremos la estadística para analizar los datos que generamos, para sacar conclusiones de ellos y para medir qué tan buenos son nuestros modelos de IA.

Capítulo 4: Estadística para el Machine Learning

4.1. Calculando Estadísticas Descriptivas

En el capítulo anterior, aprendimos a generar datos que siguen patrones específicos. Ahora, nos ponemos el sombrero de detective. La **estadística** es la ciencia de recolectar, analizar e interpretar datos. Es el puente entre los datos en bruto y el conocimiento útil.

La estadística se divide en dos grandes ramas:

1. **Estadística Descriptiva**: Se enfoca en resumir y describir las características principales de un conjunto de datos. Nos da una "fotografía" clara de los datos que tenemos.
2. **Estadística Inferencial**: Va un paso más allá. Utiliza los datos de una muestra para hacer inferencias, predicciones o conclusiones sobre una población más grande.

En esta sección, nos centraremos en la **estadística descriptiva**. Antes de entrenar cualquier modelo de IA, es fundamental entender nuestros datos. ¿Cuál es su valor típico? ¿Qué tan dispersos están? Estas preguntas se responden con las métricas descriptivas.

Describiendo Datos: Medidas Clave

Las métricas más comunes se agrupan en dos categorías:

- **Medidas de Tendencia Central** (¿dónde está el "centro" de los datos?):
    - **Media (Promedio)**: La suma de todos los valores dividida por el número de valores. Es sensible a valores atípicos.
    - **Mediana**: El valor que se encuentra justo en el medio de un conjunto de datos ordenado. Es robusta frente a valores atípicos.
    - **Moda**: El valor que aparece con más frecuencia.
- **Medidas de Dispersión** (¿qué tan "esparcidos" están los datos?):
    - **Varianza**: El promedio de las diferencias al cuadrado entre cada valor y la media. Mide la dispersión total.
    - **Desviación Estándar**: La raíz cuadrada de la varianza. Es más fácil de interpretar porque está en las mismas unidades que los datos originales.

Implementación en C: Funciones Estadísticas

Vamos a escribir funciones en C para calcular estas métricas para un conjunto de datos representado por un array de `double`.

#include <stdio.h>

```c
#include <stdlib.h>
#include <math.h>

/**
 * @brief Función de comparación para qsort (ordena de menor a mayor).
 */
int comparar_doubles(const void* a, const void* b) {
    double arg1 = *(const double*)a;
    double arg2 = *(const double*)b;
    if (arg1 < arg2) return -1;
    if (arg1 > arg2) return 1;
    return 0;
}

/**
 * @brief Calcula la media (promedio) de un array de datos.
 */
double calcular_media(const double* datos, int n) {
    double suma = 0.0;
    for (int i = 0; i < n; i++) {
        suma += datos[i];
    }
    return suma / n;
}

/**
 * @brief Calcula la mediana de un array de datos.
 * @note Esta función crea una copia de los datos para no modificar el array original.
 */
double calcular_mediana(const double* datos, int n) {
    // Creamos una copia para no alterar el array original al ordenarlo.
    double* datos_ordenados = (double*)malloc(n * sizeof(double));
    for (int i = 0; i < n; i++) {
        datos_ordenados[i] = datos[i];
    }

    // Ordenamos la copia. qsort es una función estándar de C para ordenar.
    qsort(datos_ordenados, n, sizeof(double), comparar_doubles);

    double mediana;
    if (n % 2 == 0) {
        // Si el número de datos es par, la mediana es el promedio de los dos del medio.
        mediana = (datos_ordenados[n / 2 - 1] + datos_ordenados[n / 2]) / 2.0;
    } else {
```

```
        // Si es impar, es justo el valor del medio.
        mediana = datos_ordenados[n / 2];
    }

    free(datos_ordenados);
    return mediana;
}

/**
 * @brief Calcula la varianza de un conjunto de datos.
 */
double calcular_varianza(const double* datos, int n) {
    double media = calcular_media(datos, n);
    double suma_diferencias_cuadradas = 0.0;
    for (int i = 0; i < n; i++) {
        suma_diferencias_cuadradas += pow(datos[i] - media, 2);
    }
    // Usamos la varianza muestral (dividiendo por n-1), que es común en estadística.
    return suma_diferencias_cuadradas / (n - 1);
}

/**
 * @brief Calcula la desviación estándar.
 */
double calcular_desv_est(double varianza) {
    return sqrt(varianza);
}

int main() {
    // Nuestro conjunto de datos de ejemplo.
    double dataset[] = {15.0, 20.0, 22.0, 25.0, 28.0, 30.0, 35.0, 40.0, 150.0};
    int n = sizeof(dataset) / sizeof(dataset[0]);

    printf("--- Análisis Estadístico Descriptivo ---\n\n");
    printf("Dataset: [ ");
    for(int i = 0; i < n; i++) {
        printf("%.1f ", dataset[i]);
    }
    printf("]\n\n");

    // Calculamos las métricas
    double media = calcular_media(dataset, n);
    double mediana = calcular_mediana(dataset, n);
    double varianza = calcular_varianza(dataset, n);
```

```
    double desv_est = calcular_desv_est(varianza);

    printf("--- Medidas de Tendencia Central ---\n");
    printf("Media: %.4f\n", media);
    printf("Mediana: %.4f\n", mediana);
    printf("Nota: La media es mucho mayor que la mediana debido al valor atípico (150.0).\n");
    printf("      La mediana es una medida más robusta en este caso.\n\n");

    printf("--- Medidas de Dispersión ---\n");
    printf("Varianza: %.4f\n", varianza);
    printf("Desviación Estándar: %.4f\n", desv_est);
    printf("Nota: La desviación estándar es alta, indicando que los datos están muy
dispersos.\n");

    return 0;
}
```

Conclusión de la Sección

Con estas simples funciones, hemos ganado una visión increíblemente poderosa de nuestro conjunto de datos. Antes de aplicar algoritmos complejos, ahora podemos responder preguntas básicas pero vitales: ¿Cuál es el valor "típico"? ¿Hay valores extraños o atípicos que podrían afectar a nuestro modelo? ¿Están los datos muy agrupados o muy dispersos?

La estadística descriptiva es el primer paso obligatorio en cualquier proyecto de ciencia de datos o IA. Es nuestro primer contacto con los datos, y nos permite formular hipótesis y tomar decisiones informadas sobre los siguientes pasos a seguir.

En la próxima sección, exploraremos cómo usar la estadística no solo para describir, sino para **hacer inferencias** y evaluar la confianza en nuestros resultados.

4.2. Estimadores y Métricas Clave

En la sección anterior, aprendimos a describir un conjunto de datos. Ahora, damos un paso hacia la inferencia y la evaluación. ¿Cómo podemos estimar los parámetros de un modelo a partir de los datos? Y una vez que tenemos un modelo que hace predicciones, ¿cómo medimos si es bueno o malo? Estas son dos de las preguntas más importantes en el aprendizaje automático.

Estimadores: Máxima Verosimilitude (MLE)

Un **estimador** es una regla para calcular una estimación de una cantidad de interés (un parámetro) basándose en los datos observados. Uno de los estimadores más importantes y

utilizados es el de **Máxima Verosimilitud** (MLE, por sus siglas en inglés, *Maximum Likelihood Estimation*).

La pregunta que MLE intenta responder es: **"Dados los datos que he observado, ¿cuáles son los valores de los parámetros del modelo que hacen que la observación de estos datos sea lo más probable posible?"**

Pensemos en un ejemplo simple: lanzamos una moneda 10 veces y obtenemos 7 caras. Queremos estimar la probabilidad p de que salga cara (el parámetro de nuestro modelo de Bernoulli). MLE nos diría que el valor de p que maximiza la probabilidad de haber observado 7 caras y 3 cruces es, intuitivamente, .

Correlación vs. Causalidad: Un Error Común

Antes de medir el rendimiento, es vital entender una distinción crucial.

- **Correlación**: Es una medida estadística que indica hasta qué punto dos variables se mueven juntas. Una correlación positiva alta significa que cuando una variable aumenta, la otra también tiende a hacerlo. Una correlación negativa significa que cuando una aumenta, la otra tiende a disminuir.
- **Causalidad**: Significa que un evento es el resultado directo de la ocurrencia de otro evento.

Correlación no implica causalidad. Este es quizás el principio más importante de la estadística. Por ejemplo, la venta de helados y los ataques de tiburones están altamente correlacionados. Pero uno no causa el otro. La causa subyacente (una "variable latente") es el buen tiempo: más gente va a la playa, nada y come helados. Un modelo de IA podría aprender esta correlación, pero no entendería la causalidad sin más información.

Una métrica común para medir la correlación lineal es el **coeficiente de correlación de Pearson**.

Métricas de Evaluación de Modelos

Una vez que nuestro modelo (sea de regresión o clasificación) hace predicciones, necesitamos métricas para cuantificar su rendimiento.

- **Para Regresión (predecir un valor numérico):**
    - **Error Cuadrático Medio (MSE - Mean Squared Error)**: Es el promedio de las diferencias al cuadrado entre los valores predichos y los valores reales. Penaliza más los errores grandes. Un MSE más bajo es mejor.
- **Para Clasificación (predecir una categoría):**
    - **Exactitud (Accuracy)**: La proporción de predicciones correctas. Es simple, pero puede ser muy engañosa si las clases están desbalanceadas (ej. 99% de los correos no son spam).

- **Precisión (Precision)**: De todos los que clasificamos como "positivos", ¿cuántos lo eran realmente? Es importante cuando el coste de un falso positivo es alto.
- **Sensibilidad (Recall)**: De todos los que eran realmente "positivos", ¿cuántos fuimos capaces de encontrar? Es importante cuando el coste de un falso negativo es alto (ej. diagnóstico médico).

Implementación en C

Vamos a implementar funciones para el coeficiente de correlación de Pearson y las métricas de evaluación.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

// Incluimos la función de la sección anterior
double calcular_media(const double* datos, int n) {
    double suma = 0.0;
    for (int i = 0; i < n; i++) {
        suma += datos[i];
    }
    return suma / n;
}

/**
 * @brief Calcula el coeficiente de correlación de Pearson entre dos conjuntos de datos.
 * @return Un valor entre -1 (correlación negativa perfecta) y 1 (correlación positiva perfecta).
 */
double calcular_correlacion(const double* x, const double* y, int n) {
    double media_x = calcular_media(x, n);
    double media_y = calcular_media(y, n);

    double numerador = 0.0;
    double den_x = 0.0;
    double den_y = 0.0;

    for (int i = 0; i < n; i++) {
        numerador += (x[i] - media_x) * (y[i] - media_y);
        den_x += pow(x[i] - media_x, 2);
        den_y += pow(y[i] - media_y, 2);
    }

    return numerador / (sqrt(den_x) * sqrt(den_y));
}
```

```
/**
 * @brief Calcula el Error Cuadrático Medio (MSE) para un modelo de regresión.
 */
double calcular_mse(const double* y_real, const double* y_predicho, int n) {
    double suma_errores_cuadrados = 0.0;
    for (int i = 0; i < n; i++) {
        suma_errores_cuadrados += pow(y_real[i] - y_predicho[i], 2);
    }
    return suma_errores_cuadrados / n;
}

/**
 * @brief Calcula las métricas de clasificación (Exactitud, Precisión, Sensibilidad).
 * @param y_real Array con las etiquetas verdaderas (0 o 1).
 * @param y_predicho Array con las etiquetas predichas (0 o 1).
 * @param n El número de muestras.
 * @param out_exactitud Puntero para devolver la exactitud.
 * @param out_precision Puntero para devolver la precisión.
 * @param out_sensibilidad Puntero para devolver la sensibilidad.
 */
void calcular_metricas_clasificacion(const int* y_real, const int* y_predicho, int n,
                                     double* out_exactitud, double* out_precision, double* out_sensibilidad) {
    int tp = 0, tn = 0, fp = 0, fn = 0; // Verdaderos Positivos, etc.

    for (int i = 0; i < n; i++) {
        if (y_real[i] == 1 && y_predicho[i] == 1) tp++;
        else if (y_real[i] == 0 && y_predicho[i] == 0) tn++;
        else if (y_real[i] == 0 && y_predicho[i] == 1) fp++;
        else if (y_real[i] == 1 && y_predicho[i] == 0) fn++;
    }

    *out_exactitud = (double)(tp + tn) / n;
    *out_precision = (tp + fp == 0) ? 0.0 : (double)tp / (tp + fp);
    *out_sensibilidad = (tp + fn == 0) ? 0.0 : (double)tp / (tp + fn);
}


int main() {
    printf("--- Coeficiente de Correlación ---\n");
    double temp_verano[] = {25, 30, 35, 28, 32};
    double ventas_helados[] = {200, 250, 300, 220, 280};
    int n_corr = 5;
    double corr = calcular_correlacion(temp_verano, ventas_helados, n_corr);
```

```
    printf("Correlación entre temperatura y venta de helados: %.4f (Muy alta)\n\n", corr);

    printf("--- Métricas de Regresión (MSE) ---\n");
    double y_real_reg[] = {2.5, 3.0, 4.2, 5.0};
    double y_pred_reg[] = {2.7, 3.2, 4.0, 4.9};
    int n_reg = 4;
    double mse = calcular_mse(y_real_reg, y_pred_reg, n_reg);
    printf("Error Cuadrático Medio (MSE): %.4f\n\n", mse);

    printf("--- Métricas de Clasificación ---\n");
    int y_real_clas[] =    {1, 0, 1, 1, 0, 0, 1, 1, 1, 0};
    int y_pred_clas[] =    {1, 1, 1, 0, 0, 0, 1, 1, 0, 0};
    int n_clas = 10;
    double exactitud, precision, sensibilidad;
    calcular_metricas_clasificacion(y_real_clas, y_pred_clas, n_clas, &exactitud, &precision,
&sensibilidad);
    printf("Exactitud (Accuracy): %.2f%%\n", exactitud * 100.0);
    printf("Precisión (Precision): %.2f%%\n", precision * 100.0);
    printf("Sensibilidad (Recall): %.2f%%\n", sensibilidad * 100.0);

    return 0;
}
```

Conclusión de la Sección

Con estas funciones, hemos adquirido la capacidad de evaluar cuantitativamente nuestros modelos. Ya no tenemos que "intuir" si un modelo es bueno; podemos medirlo. Estas métricas son la base para el proceso iterativo del machine learning: entrenar un modelo, medir su rendimiento, ajustar los hiperparámetros y repetir hasta que estemos satisfechos con el resultado.

En la próxima sección, cerraremos este capítulo explorando brevemente cómo la estadística nos permite ir más allá de la muestra y hacer inferencias sobre el mundo real, tocando conceptos como los tests de hipótesis y los intervalos de confianza.

Parte III: Optimización - La Búsqueda de la Excelencia

Capítulo 5: Más Allá del Descenso del Gradiente

5.1. Variantes del Descenso del Gradiente

En el Capítulo 1, implementamos el algoritmo de Descenso del Gradiente. Nuestra implementación, técnicamente conocida como **Descenso del Gradiente por Lotes (Batch Gradient Descent)**, tiene una característica clave: para dar un solo paso, necesita calcular el gradiente usando **todo el conjunto de datos**.

Esto es muy preciso y garantiza un camino suave hacia el mínimo. Sin embargo, en el mundo real, los conjuntos de datos pueden tener millones o miles de millones de ejemplos. Calcular el gradiente sobre todo el dataset para cada pequeño paso se vuelve computacionalmente prohibitivo. Sería como tener que preguntar la opinión de cada persona en un país antes de dar un solo paso.

Para solucionar esto, se desarrollaron variantes mucho más eficientes que son el estándar en el *deep learning* moderno.

Descenso del Gradiente Estocástico (SGD)

El **Descenso del Gradiente Estocástico (SGD)** lleva la idea al extremo opuesto. En lugar de usar todo el dataset, en cada paso hace lo siguiente:

1. Elige **un único ejemplo** de datos al azar.
2. Calcula el gradiente basándose *solo* en ese ejemplo.
3. Actualiza los parámetros del modelo.

- **Ventajas**: Es extremadamente rápido. Cada actualización es computacionalmente muy barata. La naturaleza ruidosa de las actualizaciones (ya que cada ejemplo tira en una dirección ligeramente diferente) puede ayudar al modelo a escapar de mínimos locales poco profundos.
- **Desventajas**: El camino hacia el mínimo es muy errático y ruidoso. En lugar de converger suavemente, el coste tiende a oscilar alrededor del mínimo sin llegar a asentarse perfectamente.

Descenso del Gradiente por Mini-Lotes (Mini-batch Gradient Descent)

Esta es la solución que domina la práctica. Es un compromiso perfecto entre la precisión del método por lotes y la velocidad del SGD. El algoritmo es el siguiente:

1. Baraja el conjunto de datos al principio de cada época (una pasada completa por los datos).
2. Divide el dataset en pequeños lotes (mini-batches) de un tamaño fijo (ej. 32, 64, 128 ejemplos).
3. Para cada mini-lote: a. Calcula el gradiente promediado sobre todos los ejemplos de ese mini-lote. b. Actualiza los parámetros del modelo.

- **Ventajas**:
    - Es mucho más rápido que el método por lotes.
    - Produce una convergencia mucho más estable y menos ruidosa que el SGD.

    - Permite aprovechar la computación paralela (vectorización) en CPUs y GPUs, ya que las operaciones sobre un mini-lote se pueden realizar de forma muy eficiente.
- **Desventajas**: Introduce un nuevo hiperparámetro a ajustar: el tamaño del lote (*batch size*).

Implementación en C: Regresión Lineal con Mini-Lotes

Vamos a implementar el Descenso del Gradiente por Mini-Lotes para un problema clásico: la regresión lineal. Nuestro objetivo será encontrar los parámetros `m` (pendiente) y `b` (intercepto) para la línea y=mx+b que mejor se ajuste a un conjunto de datos.

La función de coste que minimizaremos será el Error Cuadrático Medio (MSE).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

// Estructura para nuestro dataset de regresión lineal
typedef struct {
    double* x_datos;
    double* y_datos;
    int n_muestras;
} Dataset;

/**
 * @brief Baraja un array de índices en el lugar.
 * Algoritmo de Fisher-Yates.
 */
void barajar_indices(int* array, int n) {
    for (int i = n - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

/**
 * @brief Entrena un modelo de regresión lineal usando Descenso del Gradiente por Mini-Lotes.
 * @param datos El conjunto de datos.
 * @param m Puntero al parámetro de la pendiente (se modificará).
 * @param b Puntero al parámetro del intercepto (se modificará).
 * @param tasa_aprendizaje Controla el tamaño del paso.
```

```
 * @param epocas Número de pasadas completas por el dataset.
 * @param tamano_lote Número de muestras en cada mini-lote.
 */
void mini_batch_gd(const Dataset* datos, double* m, double* b, double tasa_aprendizaje, int
epocas, int tamano_lote) {
    int n = datos->n_muestras;
    int* indices = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        indices[i] = i;
    }

    printf("--- Iniciando entrenamiento con Mini-batch GD ---\n");
    printf("Épocas: %d, Tamaño de lote: %d, Tasa de aprendizaje: %.4f\n", epocas, tamano_lote,
tasa_aprendizaje);

    for (int i = 0; i < epocas; i++) {
        barajar_indices(indices, n); // Barajamos los datos al inicio de cada época

        // Iteramos sobre los mini-lotes
        for (int j = 0; j < n; j += tamano_lote) {
            double grad_m = 0.0;
            double grad_b = 0.0;
            int tamano_lote_actual = (j + tamano_lote <= n) ? tamano_lote : n - j;

            // Calculamos el gradiente para el mini-lote
            for (int k = 0; k < tamano_lote_actual; k++) {
                int indice_actual = indices[j + k];
                double x = datos->x_datos[indice_actual];
                double y = datos->y_datos[indice_actual];

                double y_pred = (*m) * x + (*b);

                // Derivada del MSE con respecto a m y b
                grad_m += -2.0 * x * (y - y_pred);
                grad_b += -2.0 * (y - y_pred);
            }

            // Actualizamos los parámetros usando el gradiente promedio del lote
            *m -= tasa_aprendizaje * (grad_m / tamano_lote_actual);
            *b -= tasa_aprendizaje * (grad_b / tamano_lote_actual);
        }

        if ((i + 1) % 10 == 0) {
            printf("Época %d -> m: %.4f, b: %.4f\n", i + 1, *m, *b);
```

```
        }
    }

    free(indices);
    printf("--- Entrenamiento finalizado ---\n");
}

int main() {
    srand(time(NULL));

    // Creamos un dataset sintético para y = 2x + 5 con algo de ruido
    Dataset datos;
    datos.n_muestras = 100;
    datos.x_datos = (double*)malloc(datos.n_muestras * sizeof(double));
    datos.y_datos = (double*)malloc(datos.n_muestras * sizeof(double));

    for (int i = 0; i < datos.n_muestras; i++) {
        datos.x_datos[i] = i;
        // Ruido normal con media 0 y desv_est 10
        double ruido = (double)rand() / RAND_MAX * 20.0 - 10.0;
        datos.y_datos[i] = 2.0 * i + 5.0 + ruido;
    }

    // Inicializamos los parámetros del modelo aleatoriamente
    double m = 0.0;
    double b = 0.0;

    // Entrenamos el modelo
    mini_batch_gd(&datos, &m, &b, 0.0001, 100, 10);

    printf("\nParámetros reales -> m: 2.0000, b: 5.0000\n");
    printf("Parámetros aprendidos -> m: %.4f, b: %.4f\n", m, b);

    // Liberamos memoria
    free(datos.x_datos);
    free(datos.y_datos);

    return 0;
}
```

Con esta implementación, hemos creado un optimizador mucho más realista y eficiente. Este enfoque de mini-lotes es el que encontrarás en el núcleo de todas las librerías modernas de *deep learning* como TensorFlow o PyTorch.

En la siguiente sección, exploraremos algoritmos de optimización aún más avanzados que no solo usan el gradiente, sino que también adaptan la tasa de aprendizaje sobre la marcha, como Adam.

5.2. Algoritmos de Optimización Avanzados

El Descenso del Gradiente por Mini-Lotes es un gran avance, pero el paisaje de la función de coste en problemas reales es a menudo traicionero. Puede estar lleno de "barrancos" (zonas donde la pendiente es muy pronunciada en una dirección pero muy suave en otra), "mesetas" (zonas casi planas donde el gradiente es minúsculo) y "mínimos locales" (pequeños valles que no son el punto más bajo de todo el paisaje).

Para navegar estos terrenos complejos de manera más eficiente, se han desarrollado optimizadores más inteligentes que el simple Descenso del Gradiente.

Momentum: Para no Atascarse en Mínimos Locales

La idea detrás del **Momentum** es simple y elegante. Imagina una pelota rodando por una colina. La pelota no solo se mueve en la dirección de la pendiente actual, sino que también acumula "inercia" o "momento" de sus movimientos pasados. Esta inercia le permite pasar por encima de pequeños baches (mínimos locales) y acelerar en las bajadas consistentes.

El método de Momentum implementa esta idea manteniendo un promedio móvil de los gradientes pasados. La actualización de los parámetros ya no depende solo del gradiente actual, sino de esta "velocidad" acumulada.

Adam: El Optimizador por Defecto

Adam (Adaptive Moment Estimation) es el optimizador de referencia en la mayoría de las aplicaciones de *deep learning* hoy en día. Combina dos ideas poderosas:

1. **Momentum (Primer Momento)**: Al igual que el método anterior, mantiene un promedio móvil exponencial de los gradientes para acelerar la convergencia.
2. **Tasa de Aprendizaje Adaptativa (Segundo Momento)**: Mantiene un segundo promedio móvil, pero esta vez de los *cuadrados* de los gradientes. Esta información se utiliza para adaptar la tasa de aprendizaje para cada parámetro individualmente. Si un parámetro recibe gradientes grandes y consistentes, su tasa de aprendizaje efectiva se reduce. Si recibe gradientes pequeños o esporádicos, su tasa de aprendizaje aumenta. Esto es increíblemente útil en problemas con datos dispersos o con parámetros que requieren ajustes de diferente magnitud.

Adam es robusto, computacionalmente eficiente y generalmente requiere poco ajuste de hiperparámetros, lo que lo convierte en una excelente opción por defecto.

Implementación en C: Optimizador Adam para Regresión Lineal

Vamos a reemplazar nuestra función de `mini_batch_gd` con una implementación completa del optimizador Adam. Necesitaremos una estructura para mantener el estado del optimizador (los promedios del primer y segundo momento para cada parámetro).

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

// Estructura para nuestro dataset
typedef struct {
    double* x_datos;
    double* y_datos;
    int n_muestras;
} Dataset;

// Estructura para mantener el estado del optimizador Adam
typedef struct {
    double m_m, m_b; // Primer momento (momentum) para m y b
    double v_m, v_b; // Segundo momento (tasa adaptativa) para m y b
} AdamState;

// Función para barajar índices (de la sección anterior)
void barajar_indices(int* array, int n) {
    for (int i = n - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

/**
 * @brief Entrena un modelo usando el optimizador Adam.
 * @param datos El conjunto de datos.
 * @param m Puntero al parámetro de la pendiente.
 * @param b Puntero al parámetro del intercepto.
 * @param tasa_aprendizaje Tasa de aprendizaje inicial.
 * @param epocas Número de pasadas completas por el dataset.
 * @param tamano_lote Número de muestras en cada mini-lote.
 */
void adam_optimizer(const Dataset* datos, double* m, double* b, double tasa_aprendizaje, int
epocas, int tamano_lote) {
    int n = datos->n_muestras;
```

```
    int* indices = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        indices[i] = i;
    }

    // Hiperparámetros de Adam
    const double beta1 = 0.9;
    const double beta2 = 0.999;
    const double epsilon = 1e-8;

    // Inicializamos el estado de Adam
    AdamState estado = {0};
    int t = 0; // Contador de pasos de tiempo

    printf("--- Iniciando entrenamiento con Adam Optimizer ---\n");
    printf("Épocas: %d, Tamaño de lote: %d, Tasa de aprendizaje: %.4f\n", epocas, tamano_lote,
tasa_aprendizaje);

    for (int i = 0; i < epocas; i++) {
        barajar_indices(indices, n);

        for (int j = 0; j < n; j += tamano_lote) {
            t++; // Incrementamos el paso de tiempo
            double grad_m = 0.0, grad_b = 0.0;
            int tamano_lote_actual = (j + tamano_lote <= n) ? tamano_lote : n - j;

            // Calculamos el gradiente para el mini-lote
            for (int k = 0; k < tamano_lote_actual; k++) {
                int idx = indices[j + k];
                double y_pred = (*m) * datos->x_datos[idx] + (*b);
                grad_m += -2.0 * datos->x_datos[idx] * (datos->y_datos[idx] - y_pred);
                grad_b += -2.0 * (datos->y_datos[idx] - y_pred);
            }
            grad_m /= tamano_lote_actual;
            grad_b /= tamano_lote_actual;

            // --- Actualización de Adam ---
            // 1. Actualizar promedios de momentos
            estado.m_m = beta1 * estado.m_m + (1 - beta1) * grad_m;
            estado.m_b = beta1 * estado.m_b + (1 - beta1) * grad_b;
            estado.v_m = beta2 * estado.v_m + (1 - beta2) * pow(grad_m, 2);
            estado.v_b = beta2 * estado.v_b + (1 - beta2) * pow(grad_b, 2);

            // 2. Corrección de sesgo (bias correction)
```

```
            double m_m_hat = estado.m_m / (1 - pow(beta1, t));
            double m_b_hat = estado.m_b / (1 - pow(beta1, t));
            double v_m_hat = estado.v_m / (1 - pow(beta2, t));
            double v_b_hat = estado.v_b / (1 - pow(beta2, t));

            // 3. Actualizar parámetros
            *m -= tasa_aprendizaje * m_m_hat / (sqrt(v_m_hat) + epsilon);
            *b -= tasa_aprendizaje * m_b_hat / (sqrt(v_b_hat) + epsilon);
        }

        if ((i + 1) % 10 == 0) {
            printf("Época %d -> m: %.4f, b: %.4f\n", i + 1, *m, *b);
        }
    }

    free(indices);
    printf("--- Entrenamiento finalizado ---\n");
}

int main() {
    srand(time(NULL));

    // Creamos el mismo dataset sintético que en la sección anterior
    Dataset datos;
    datos.n_muestras = 100;
    datos.x_datos = (double*)malloc(datos.n_muestras * sizeof(double));
    datos.y_datos = (double*)malloc(datos.n_muestras * sizeof(double));

    for (int i = 0; i < datos.n_muestras; i++) {
        datos.x_datos[i] = i;
        double ruido = (double)rand() / RAND_MAX * 20.0 - 10.0;
        datos.y_datos[i] = 2.0 * i + 5.0 + ruido;
    }

    // Inicializamos los parámetros
    double m = 0.0, b = 0.0;

    // Entrenamos con Adam. Notar que podemos usar una tasa de aprendizaje mayor.
    adam_optimizer(&datos, &m, &b, 0.01, 100, 10);

    printf("\nParámetros reales -> m: 2.0000, b: 5.0000\n");
    printf("Parámetros aprendidos -> m: %.4f, b: %.4f\n", m, b);

    free(datos.x_datos);
```

```
    free(datos.y_datos);

    return 0;
}
```

Conclusión del Capítulo 5

En este capítulo, hemos pasado de un optimizador simple a uno de los más sofisticados y utilizados en la práctica. Hemos visto que la optimización no se trata solo de seguir la pendiente, sino de hacerlo de manera inteligente, acumulando inercia y adaptándose a las características de cada parámetro.

Con un optimizador robusto como Adam, estamos mucho mejor equipados para entrenar las redes neuronales profundas que construiremos en la parte final de este libro. Hemos completado nuestra caja de herramientas de optimización. En el próximo capítulo, uniremos todo lo que hemos aprendido (cálculo, álgebra lineal, probabilidad y optimización) para construir nuestra primera red neuronal funcional desde cero.

5.3. Optimización Convexa

A lo largo de este capítulo, hemos desarrollado herramientas cada vez más sofisticadas para "descender" por el paisaje de una función de coste. Pero, ¿qué garantías tenemos de que el valle al que llegamos es realmente el punto más bajo de todo el mapa? La respuesta a esta pregunta reside en un concepto matemático clave: la **convexidad**.

¿Qué es una función convexa y por qué nos facilita la vida?

Intuitivamente, una función convexa es aquella que tiene la forma de un "cuenco" o un "valle" único. Si trazas una línea recta entre dos puntos cualesquiera de la gráfica de la función, toda la línea se mantendrá por encima o tocará la gráfica, nunca pasará por debajo.

[Imagen de una función convexa con una línea recta entre dos puntos]

Por el contrario, una función no convexa puede tener múltiples valles y picos, como una cadena montañosa.

[Imagen de una función no convexa con múltiples valles]

La belleza de las funciones convexas reside en una propiedad casi mágica para la optimización.

Mínimos Locales vs. Mínimo Global

- **Mínimo Local**: Es un punto en el paisaje que es más bajo que todos sus vecinos inmediatos. Es el fondo de un pequeño valle.

- **Mínimo Global**: Es el punto más bajo de todo el paisaje de la función, sin excepción.

En una función no convexa (con muchas montañas y valles), un algoritmo como el Descenso del Gradiente podría encontrar fácilmente el fondo de un valle cercano (un mínimo local) y atascarse allí, pensando que ha encontrado la mejor solución, sin saber que al otro lado de una montaña existe un valle mucho más profundo (el mínimo global).

[Imagen de una función no convexa mostrando un mínimo local y un mínimo global]

La Garantía de la Convexidad

Aquí es donde las funciones convexas brillan: **para una función convexa, cualquier mínimo local es también, por necesidad, el mínimo global.**

Si estás en un paisaje con forma de cuenco, no importa dónde empieces, si sigues descendiendo, inevitablemente llegarás al único y verdadero fondo. No hay otros valles que te puedan engañar.

Esta propiedad es tan poderosa que existe todo un subcampo de la optimización dedicado a ello. Si puedes formular tu problema de tal manera que la función de coste sea convexa (como ocurre en modelos clásicos como la Regresión Logística o las Support Vector Machines), entonces puedes tener la certeza de que el Descenso del Gradiente (o algoritmos similares) encontrará la mejor solución posible.

¿Y qué pasa con el Deep Learning?

Aquí viene la parte interesante y uno de los grandes misterios de la era moderna de la IA. Las funciones de coste de las redes neuronales profundas, con sus millones de parámetros, son **extremadamente no convexas**. Sus paisajes de error son increíblemente complejos, con una cantidad astronómica de mínimos locales, máximos y puntos de silla.

Teóricamente, nuestros optimizadores deberían atascarse en mínimos locales subóptimos todo el tiempo. Sin embargo, en la práctica, algoritmos como Adam funcionan sorprendentemente bien y son capaces de encontrar soluciones de muy alta calidad. La razón exacta de por qué esto ocurre es todavía un área de intensa investigación, pero algunas teorías sugieren que en estos espacios de altísima dimensionalidad, la mayoría de los mínimos locales son cualitativamente muy similares al mínimo global.

Conclusión del Capítulo 5

En este capítulo, hemos pasado de un optimizador simple a uno de los más sofisticados y utilizados en la práctica. Hemos visto que la optimización no se trata solo de seguir la pendiente, sino de hacerlo de manera inteligente, acumulando inercia y adaptándose a las características de cada parámetro. Finalmente, hemos entendido que la **convexidad** nos da una garantía de optimalidad, y aunque el *deep learning* opera en paisajes no convexos, las

herramientas que hemos construido son, sorprendentemente, las que nos permiten navegar estos terrenos complejos con un éxito notable.

Hemos completado nuestra caja de herramientas de optimización. En el próximo capítulo, el gran final, uniremos todo lo que hemos aprendido (cálculo, álgebra lineal, probabilidad y optimización) para construir nuestra primera red neuronal funcional desde cero.

Parte IV: Construyendo una Red Neuronal en C

Capítulo 6: Juntando todas las Piezas

6.1. Estructuras de Datos para una Red Neuronal

Este es el capítulo culminante. A lo largo de nuestro viaje, hemos forjado herramientas de cálculo para la optimización, de álgebra lineal para la manipulación de datos y de probabilidad para manejar la incertidumbre. Ahora, es el momento de ensamblar estas piezas para construir la estructura más icónica de la inteligencia artificial moderna: una **red neuronal**.

Nuestro primer paso es arquitectónico. Necesitamos definir en C las estructuras de datos que representarán los componentes de nuestra red: las **capas** y la **red** en su conjunto.

La Anatomía de una Red Neuronal

Una red neuronal está compuesta por capas de "neuronas". Cada capa recibe entradas (que son las salidas de la capa anterior) y produce salidas. La transformación que realiza cada capa se define por:

1. Una **matriz de pesos**: Cada conexión entre una neurona de la capa anterior y una de la capa actual tiene un peso asociado.
2. Un **vector de sesgos (biases)**: Cada neurona de la capa actual tiene un valor de sesgo que se suma al resultado ponderado.
3. Una **función de activación**: Una función no lineal (como Sigmoide o ReLU) que se aplica al resultado final de cada neurona para introducir complejidad y permitir al modelo aprender patrones no lineales.

Implementación en C: `struct Capa` y `struct Red`

Para representar una **capa**, nuestra `struct` necesitará almacenar sus pesos, sus sesgos y su función de activación.

```
// Estructuras de Álgebra Lineal (de capítulos anteriores)
typedef struct {
    int filas;
    int columnas;
```

```
    double* datos;
} Matriz;

// Un puntero a una función de activación.
// Toma una entrada (double) y devuelve una salida (double).
typedef double (*FuncionActivacion)(double);

// Estructura para una capa de la red neuronal
typedef struct {
    int num_entradas;
    int num_salidas; // (número de neuronas en esta capa)

    Matriz* pesos;   // Matriz de pesos (num_salidas x num_entradas)
    Matriz* sesgos;  // Vector de sesgos (num_salidas x 1)

    FuncionActivacion activacion; // Puntero a la función de activación
} Capa;
```

Ahora, una **red** no es más que una colección de estas capas.

```
// Estructura para la red neuronal completa
typedef struct {
    int num_capas;
    Capa** capas; // Un array de punteros a Capa
} Red;
```

Funciones de Creación e Inicialización

Necesitamos funciones para construir estas estructuras. Una parte crucial del proceso es la **inicialización de los pesos**. No podemos inicializarlos todos a cero, ya que esto impediría que las neuronas aprendieran de forma diferente. Una práctica común es inicializarlos con valores aleatorios pequeños, a menudo tomados de una distribución normal.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

// --- Incluir estructuras y funciones de Matriz de capítulos anteriores ---
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
```

```
// Prototipos de funciones de Matriz (suponemos que ya están implementadas)
Matriz* crear_matriz(int filas, int columnas);
void liberar_matriz(Matriz* m);
void imprimir_matriz(const Matriz* m);
double muestra_normal(double media, double desv_est) {
    double u1 = (double)rand() / RAND_MAX;
    double u2 = (double)rand() / RAND_MAX;
    double z0 = sqrt(-2.0 * log(u1)) * cos(2.0 * M_PI * u2);
    return z0 * desv_est + media;
}
// --- Fin de las dependencias ---

// Definición de las nuevas estructuras
typedef double (*FuncionActivacion)(double);

typedef struct {
    int num_entradas;
    int num_salidas;
    Matriz* pesos;
    Matriz* sesgos;
    FuncionActivacion activacion;
} Capa;

typedef struct {
    int num_capas;
    Capa** capas;
} Red;

// Función de activación de ejemplo: Sigmoide
double sigmoide(double x) {
    return 1.0 / (1.0 + exp(-x));
}

/**
 * @brief Crea e inicializa una nueva capa.
 */
Capa* crear_capa(int num_entradas, int num_salidas, FuncionActivacion func_act) {
    Capa* capa = (Capa*)malloc(sizeof(Capa));
    capa->num_entradas = num_entradas;
    capa->num_salidas = num_salidas;
    capa->activacion = func_act;

    // Creamos la matriz de pesos y la inicializamos aleatoriamente
    capa->pesos = crear_matriz(num_salidas, num_entradas);
```

```
    for (int i = 0; i < num_salidas * num_entradas; i++) {
        capa->pesos->datos[i] = muestra_normal(0.0, 1.0); // Inicialización normal
    }

    // Creamos el vector de sesgos (como una matriz de N x 1)
    capa->sesgos = crear_matriz(num_salidas, 1);
    // Los sesgos se pueden inicializar a cero

    return capa;
}

/**
 * @brief Libera la memoria de una capa.
 */
void liberar_capa(Capa* capa) {
    if (capa != NULL) {
        liberar_matriz(capa->pesos);
        liberar_matriz(capa->sesgos);
        free(capa);
    }
}

/**
 * @brief Crea una red neuronal vacía con una capacidad para N capas.
 */
Red* crear_red(int num_capas) {
    Red* red = (Red*)malloc(sizeof(Red));
    red->num_capas = num_capas;
    red->capas = (Capa**)malloc(num_capas * sizeof(Capa*));
    return red;
}

/**
 * @brief Libera la memoria de toda la red.
 */
void liberar_red(Red* red) {
    if (red != NULL) {
        for (int i = 0; i < red->num_capas; i++) {
            liberar_capa(red->capas[i]);
        }
        free(red->capas);
        free(red);
    }
}
```

```
int main() {
    srand(time(NULL)); // Semilla para la inicialización aleatoria

    printf("--- Construyendo la Arquitectura de una Red Neuronal ---\n\n");

    // Definimos la arquitectura para resolver el problema XOR
    // 2 neuronas de entrada, 1 capa oculta con 3 neuronas, 1 neurona de salida
    int num_capas_red = 2;
    Red* mi_red = crear_red(num_capas_red);

    // Creamos y añadimos las capas a la red
    mi_red->capas[0] = crear_capa(2, 3, &sigmoide); // Capa oculta
    mi_red->capas[1] = crear_capa(3, 1, &sigmoide); // Capa de salida

    printf("Red creada con %d capas.\n\n", mi_red->num_capas);

    printf("--- Capa 1 (Oculta) ---\n");
    printf("Entradas: %d, Salidas: %d\n", mi_red->capas[0]->num_entradas,
mi_red->capas[0]->num_salidas);
    //imprimir_matriz(mi_red->capas[0]->pesos); // Descomentar para ver pesos

    printf("\n--- Capa 2 (Salida) ---\n");
    printf("Entradas: %d, Salidas: %d\n", mi_red->capas[1]->num_entradas,
mi_red->capas[1]->num_salidas);
    //imprimir_matriz(mi_red->capas[1]->pesos); // Descomentar para ver pesos

    // Liberamos toda la memoria al final
    liberar_red(mi_red);
    printf("\nMemoria de la red liberada correctamente.\n");

    return 0;
}
```

Conclusión de la Sección

Hemos logrado un hito monumental. Ya no tenemos solo funciones y matrices aisladas; tenemos una **arquitectura**. Hemos definido la estructura de datos que encapsula la esencia de una red neuronal.

Con este esqueleto en su lugar, estamos listos para darle vida. En las siguientes secciones, implementaremos el "forward pass" (cómo la red hace una predicción) y el "backward pass" (cómo la red aprende de sus errores), utilizando las estructuras que acabamos de crear.

6.2. El Ciclo de Vida del Aprendizaje en C

Una red neuronal, una vez construida, tiene un ciclo de vida muy definido que se repite una y otra vez. Este ciclo consta de dos fases principales:

1. **La Propagación hacia Adelante (Feedforward Pass)**: La red toma una entrada y, capa por capa, la procesa para producir una salida o predicción. Es el modo "pensamiento" de la red.
2. **La Retropropagación (Backpropagation Pass)**: La red compara su predicción con el resultado esperado, calcula el error y propaga este error hacia atrás para determinar cómo debe ajustar cada uno de sus pesos y sesgos para mejorar. Es el modo "aprendizaje" de la red.

En esta sección, implementaremos ambas fases, creando las funciones que harán que nuestra red piense y aprenda.

Implementación en C: La Función `feedforward()`

El proceso de *feedforward* para una sola capa es el siguiente:

1. Multiplica la matriz de pesos de la capa por el vector de entrada.
2. Suma el vector de sesgos al resultado.
3. Aplica la función de activación a cada elemento del resultado.

La salida de este proceso se convierte en la entrada de la siguiente capa. Repetimos esto hasta la capa final.

Implementación en C: La Función `backpropagate()`

Esta es la parte más compleja. La retropropagación es una aplicación ingeniosa de la **Regla de la Cadena** del cálculo. El objetivo es calcular el gradiente de la función de coste (el error) con respecto a cada peso y sesgo de la red.

El proceso, a grandes rasgos, es:

1. Calcula el error en la capa de salida. Para un problema de regresión, esto podría ser (predicción - valor_real).
2. Calcula el gradiente del error con respecto a la salida de la última capa (antes de la activación). Esto implica multiplicar el error por la derivada de la función de activación.
3. Calcula el gradiente con respecto a los pesos y sesgos de la última capa.
4. **Propaga el error hacia atrás**: Calcula el error de la capa anterior basándose en el error de la capa actual y los pesos que las conectan.
5. Repite los pasos 2, 3 y 4 para cada capa, moviéndose desde la última hasta la primera.

Implementación en C: La Función `update_weights()`

Una vez que `backpropagate()` nos ha dado todos los gradientes (las direcciones para ajustar los parámetros), la función `update_weights()` simplemente aplica la regla de actualización de nuestro optimizador (como el Descenso del Gradiente simple):

```
nuevo_peso = peso_actual - tasa_de_aprendizaje * gradiente_del_peso
```

Implementación Completa del Ciclo

Ahora, unamos todo esto en código. Necesitaremos añadir algunas funciones de álgebra lineal (suma y multiplicación por escalar) y la derivada de nuestra función de activación.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <assert.h>

// --- Estructuras y funciones de capítulos anteriores ---
// (Incluimos Matriz, Capa, Red, crear_matriz, liberar_matriz, etc.)
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

typedef struct {
    int filas;
    int columnas;
    double* datos;
} Matriz;

typedef double (*FuncionActivacion)(double);

typedef struct {
    int num_entradas;
    int num_salidas;
    Matriz* pesos;
    Matriz* sesgos;
    FuncionActivacion activacion;
    // Para backprop, necesitamos guardar las salidas de cada capa
    Matriz* salidas;
} Capa;

typedef struct {
    int num_capas;
    Capa** capas;
```

```
} Red;

// --- Prototipos de funciones de álgebra lineal ---
Matriz* crear_matriz(int filas, int columnas);
void liberar_matriz(Matriz* m);
Matriz* multiplicar_matrices(const Matriz* A, const Matriz* B);
void sumar_matrices(Matriz* A, const Matriz* B); // A = A + B
Matriz* transponer_matriz(const Matriz* A);
void aplicar_funcion(Matriz* m, FuncionActivacion func);

// --- Funciones de Activación y sus Derivadas ---
double sigmoide(double x) { return 1.0 / (1.0 + exp(-x)); }
double derivada_sigmoide(double x) { return x * (1.0 - x); }

// --- Implementación de funciones de creación (ligeramente modificada) ---
Capa* crear_capa(int num_entradas, int num_salidas, FuncionActivacion func_act) {
    Capa* capa = (Capa*)malloc(sizeof(Capa));
    capa->num_entradas = num_entradas;
    capa->num_salidas = num_salidas;
    capa->activacion = func_act;
    capa->pesos = crear_matriz(num_salidas, num_entradas);
    capa->sesgos = crear_matriz(num_salidas, 1);
    // Inicialización aleatoria...
    for (int i = 0; i < num_salidas * num_entradas; i++) {
        capa->pesos->datos[i] = ((double)rand() / RAND_MAX) * 2.0 - 1.0;
    }
    return capa;
}
Red* crear_red(int num_capas) { /* ... igual que antes ... */ }
void liberar_capa(Capa* capa) {
    if (capa) {
        liberar_matriz(capa->pesos);
        liberar_matriz(capa->sesgos);
        liberar_matriz(capa->salidas); // Liberar las salidas guardadas
        free(capa);
    }
}
void liberar_red(Red* red) { /* ... igual que antes ... */ }

/**
 * @brief Realiza el feedforward a través de toda la red.
 * @param red La red neuronal.
 * @param entrada La matriz de entrada (debe ser de N x 1).
 * @return La matriz de salida de la red.
```

```
 */
Matriz* feedforward(Red* red, Matriz* entrada) {
    Matriz* salida_actual = entrada;
    for (int i = 0; i < red->num_capas; i++) {
        Capa* capa_actual = red->capas[i];

        // Calculamos la salida ponderada
        Matriz* ponderada = multiplicar_matrices(capa_actual->pesos, salida_actual);
        sumar_matrices(ponderada, capa_actual->sesgos);

        // Aplicamos la función de activación
        aplicar_funcion(ponderada, capa_actual->activacion);

        // Guardamos la salida para usarla en backprop
        liberar_matriz(capa_actual->salidas); // Liberamos la anterior si existe
        capa_actual->salidas = ponderada;

        salida_actual = ponderada;
    }
    return salida_actual;
}

/**
 * @brief Realiza la retropropagación y actualiza los pesos.
 */
void backpropagate(Red* red, Matriz* entrada, Matriz* objetivo, double tasa_aprendizaje) {
    // 1. Feedforward para obtener las salidas
    Matriz* salida_final = feedforward(red, entrada);

    // 2. Calcular el error inicial en la capa de salida
    Matriz* error = crear_matriz(objetivo->filas, objetivo->columnas);
    for(int i=0; i < objetivo->filas * objetivo->columnas; ++i) {
        error->datos[i] = objetivo->datos[i] - salida_final->datos[i];
    }

    // 3. Retropropagar el error
    for (int i = red->num_capas - 1; i >= 0; i--) {
        Capa* capa_actual = red->capas[i];
        Matriz* entrada_capa = (i == 0) ? entrada : red->capas[i - 1]->salidas;

        // Calcular el gradiente
        Matriz* gradiente = crear_matriz(capa_actual->salidas->filas,
capa_actual->salidas->columnas);
        for(int j=0; j < gradiente->filas * gradiente->columnas; ++j) {
```

```
            gradiente->datos[j] = derivada_sigmoide(capa_actual->salidas->datos[j]);
            gradiente->datos[j] *= error->datos[j] * tasa_aprendizaje;
        }

        // Calcular deltas de los pesos
        Matriz* entrada_transpuesta = transponer_matriz(entrada_capa);
        Matriz* delta_pesos = multiplicar_matrices(gradiente, entrada_transpuesta);

        // Actualizar pesos y sesgos
        sumar_matrices(capa_actual->pesos, delta_pesos);
        sumar_matrices(capa_actual->sesgos, gradiente);

        // Propagar el error a la capa anterior
        Matriz* pesos_transpuestos = transponer_matriz(capa_actual->pesos);
        Matriz* error_siguiente_capa = multiplicar_matrices(pesos_transpuestos, error);

        // Liberar memoria intermedia
        liberar_matriz(error);
        error = error_siguiente_capa; // El nuevo error para la siguiente iteración
        liberar_matriz(gradiente);
        liberar_matriz(entrada_transpuesta);
        liberar_matriz(delta_pesos);
        liberar_matriz(pesos_transpuestos);
    }
    liberar_matriz(error);
}


// --- Implementaciones de funciones de álgebra lineal ---
// (Aquí irían las implementaciones completas de crear_matriz, multiplicar_matrices, etc.)
// ...

int main() {
    srand(time(NULL));
    // Esta sección solo demuestra la estructura. El entrenamiento completo
    // se mostrará en la sección 6.3.
    printf("Este código define las funciones de feedforward y backpropagation.\n");
    printf("En la siguiente sección, las usaremos para resolver un problema real.\n");
    return 0;
}
```

Conclusión de la Sección

Hemos llegado al núcleo de la maquinaria de aprendizaje. Las funciones `feedforward` y `backpropagate` encapsulan el ciclo de "pensar" y "aprender" de nuestra red. Aunque el código es complejo, cada línea es una aplicación directa de los principios de álgebra lineal y cálculo que hemos estudiado.

- `feedforward` es una cascada de multiplicaciones de matrices y aplicaciones de funciones.
- `backpropagate` es una aplicación metódica de la regla de la cadena para distribuir la culpa del error.

Con este motor en su lugar, estamos listos para el desafío final: entrenar nuestra red para que resuelva un problema que no se puede resolver con una simple línea recta, el clásico problema XOR.

6.3. Un Proyecto Completo: Red Neuronal para XOR

Hemos llegado al final de nuestro viaje. Tenemos una arquitectura de red, un método para que haga predicciones (`feedforward`) y un mecanismo para que aprenda de sus errores (`backpropagate`). Ahora, vamos a darle un desafío: el problema **XOR (OR exclusivo)**.

¿Por qué el problema XOR?

El problema XOR es un "Hola, Mundo" clásico en el campo de las redes neuronales por una razón muy importante: **no es linealmente separable**. Esto significa que no puedes dibujar una única línea recta para separar los resultados "0" de los resultados "1".

La tabla de verdad de XOR es:

- 0 XOR 0 = **0**
- 0 XOR 1 = **1**
- 1 XOR 0 = **1**
- 1 XOR 1 = **0**

[Imagen de los puntos XOR en un gráfico 2D, mostrando que no se pueden separar con una sola línea]

Un modelo simple como la regresión logística fallaría. Se necesita una red neuronal con al menos una capa oculta para "doblar" el espacio y encontrar una solución. Si nuestra red puede resolver esto, significa que hemos construido con éxito un modelo capaz de aprender relaciones no lineales complejas.

Implementación en C: El Programa Completo

El siguiente código contiene todo lo que hemos construido. El `main` define el dataset XOR, crea nuestra red neuronal, la entrena durante miles de iteraciones y finalmente prueba su rendimiento.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <assert.h>

// --- Definición de Estructuras ---
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

typedef struct {
    int filas;
    int columnas;
    double* datos;
} Matriz;

typedef double (*FuncionActivacion)(double);

typedef struct {
    int num_entradas;
    int num_salidas;
    Matriz* pesos;
    Matriz* sesgos;
    FuncionActivacion activacion;
    Matriz* salidas; // Salida después de la activación (z)
    Matriz* z;       // Salida antes de la activación (ponderada)
} Capa;

typedef struct {
    int num_capas;
    Capa** capas;
} Red;

// --- Funciones de Activación y sus Derivadas ---
double sigmoide(double x) { return 1.0 / (1.0 + exp(-x)); }
double derivada_sigmoide(double x) { return x * (1.0 - x); } // Nota: toma la salida de la sigmoide como entrada

// --- Funciones de Utilidad de Matriz ---
```

```
Matriz* crear_matriz(int filas, int columnas) {
    Matriz* m = (Matriz*)malloc(sizeof(Matriz));
    assert(m != NULL);
    m->filas = filas;
    m->columnas = columnas;
    m->datos = (double*)calloc(filas * columnas, sizeof(double));
    assert(m->datos != NULL);
    return m;
}

void liberar_matriz(Matriz* m) {
    if (m != NULL) {
        free(m->datos);
        free(m);
    }
}

void imprimir_matriz(const Matriz* m, const char* nombre) {
    printf("%s (%d x %d):\n", nombre, m->filas, m->columnas);
    for (int i = 0; i < m->filas; i++) {
        printf("[ ");
        for (int j = 0; j < m->columnas; j++) {
            printf("%8.4f ", m->datos[i * m->columnas + j]);
        }
        printf("]\n");
    }
}

Matriz* matriz_desde_array(const double* arr, int filas, int columnas) {
    Matriz* m = crear_matriz(filas, columnas);
    for(int i=0; i < filas * columnas; i++) {
        m->datos[i] = arr[i];
    }
    return m;
}

void aplicar_funcion(Matriz* m, FuncionActivacion func) {
    for (int i = 0; i < m->filas * m->columnas; i++) {
        m->datos[i] = func(m->datos[i]);
    }
}

void sumar_matrices(Matriz* A, const Matriz* B) {
    assert(A->filas == B->filas && A->columnas == B->columnas);
```

```
    for (int i = 0; i < A->filas * A->columnas; i++) {
        A->datos[i] += B->datos[i];
    }
}

Matriz* multiplicar_matrices(const Matriz* A, const Matriz* B) {
    assert(A->columnas == B->filas);
    Matriz* C = crear_matriz(A->filas, B->columnas);
    for (int i = 0; i < C->filas; i++) {
        for (int j = 0; j < C->columnas; j++) {
            double suma = 0.0;
            for (int k = 0; k < A->columnas; k++) {
                suma += A->datos[i * A->columnas + k] * B->datos[k * B->columnas + j];
            }
            C->datos[i * C->columnas + j] = suma;
        }
    }
    return C;
}

Matriz* transponer_matriz(const Matriz* A) {
    Matriz* At = crear_matriz(A->columnas, A->filas);
    for (int i = 0; i < A->filas; i++) {
        for (int j = 0; j < A->columnas; j++) {
            At->datos[j * At->columnas + i] = A->datos[i * A->columnas + j];
        }
    }
    return At;
}

// --- Funciones de Red Neuronal ---
Capa* crear_capa(int num_entradas, int num_salidas, FuncionActivacion func_act) {
    Capa* capa = (Capa*)malloc(sizeof(Capa));
    capa->num_entradas = num_entradas;
    capa->num_salidas = num_salidas;
    capa->activacion = func_act;
    capa->pesos = crear_matriz(num_salidas, num_entradas);
    capa->sesgos = crear_matriz(num_salidas, 1);
    capa->salidas = NULL;
    capa->z = NULL;
    for (int i = 0; i < num_salidas * num_entradas; i++) {
        capa->pesos->datos[i] = ((double)rand() / RAND_MAX) * 2.0 - 1.0;
    }
    return capa;
```

```
}

void liberar_capa(Capa* capa) {
    if (capa) {
        liberar_matriz(capa->pesos);
        liberar_matriz(capa->sesgos);
        liberar_matriz(capa->salidas);
        liberar_matriz(capa->z);
        free(capa);
    }
}

Red* crear_red(int num_capas) {
    Red* red = (Red*)malloc(sizeof(Red));
    red->num_capas = num_capas;
    red->capas = (Capa**)malloc(num_capas * sizeof(Capa*));
    return red;
}

void liberar_red(Red* red) {
    if (red) {
        for (int i = 0; i < red->num_capas; i++) {
            liberar_capa(red->capas[i]);
        }
        free(red->capas);
        free(red);
    }
}

Matriz* feedforward(Red* red, Matriz* entrada) {
    Matriz* salida_actual = entrada;
    for (int i = 0; i < red->num_capas; i++) {
        Capa* capa_actual = red->capas[i];

        liberar_matriz(capa_actual->z);
        capa_actual->z = multiplicar_matrices(capa_actual->pesos, salida_actual);
        sumar_matrices(capa_actual->z, capa_actual->sesgos);

        liberar_matriz(capa_actual->salidas);
        capa_actual->salidas = crear_matriz(capa_actual->z->filas, capa_actual->z->columnas);
        for(int j=0; j < capa_actual->z->filas * capa_actual->z->columnas; j++) {
            capa_actual->salidas->datos[j] = capa_actual->z->datos[j];
        }
        aplicar_funcion(capa_actual->salidas, capa_actual->activacion);
```

```
        salida_actual = capa_actual->salidas;
    }
    return salida_actual;
}

void backpropagate(Red* red, Matriz* entrada, Matriz* objetivo, double tasa_aprendizaje) {
    Matriz* salida_final = red->capas[red->num_capas - 1]->salidas;

    Matriz* error = crear_matriz(objetivo->filas, 1);
    for(int i=0; i < objetivo->filas; i++) {
        error->datos[i] = objetivo->datos[i] - salida_final->datos[i];
    }

    for (int i = red->num_capas - 1; i >= 0; i--) {
        Capa* capa_actual = red->capas[i];
        Matriz* entrada_capa = (i == 0) ? entrada : red->capas[i - 1]->salidas;

        Matriz* salidas_deriv = crear_matriz(capa_actual->salidas->filas, 1);
        for(int j=0; j < salidas_deriv->filas; j++) {
            salidas_deriv->datos[j] = derivada_sigmoide(capa_actual->salidas->datos[j]);
        }

        for(int j=0; j < error->filas; j++) {
            salidas_deriv->datos[j] *= error->datos[j] * tasa_aprendizaje;
        }
        Matriz* gradiente = salidas_deriv;

        Matriz* entrada_transpuesta = transponer_matriz(entrada_capa);
        Matriz* delta_pesos = multiplicar_matrices(gradiente, entrada_transpuesta);

        sumar_matrices(capa_actual->pesos, delta_pesos);
        sumar_matrices(capa_actual->sesgos, gradiente);

        if (i > 0) {
            Matriz* pesos_transpuestos = transponer_matriz(capa_actual->pesos);
            Matriz* error_siguiente_capa = multiplicar_matrices(pesos_transpuestos, error);
            liberar_matriz(error);
            error = error_siguiente_capa;
            liberar_matriz(pesos_transpuestos);
        }

        liberar_matriz(gradiente);
        liberar_matriz(entrada_transpuesta);
```

```
        liberar_matriz(delta_pesos);
    }
    liberar_matriz(error);
}

// --- El Programa Principal ---
int main() {
    srand(time(NULL));

    // 1. Definir el dataset XOR
    double entradas_xor_arr[][2] = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};
    double salidas_xor_arr[][1] = {{0}, {1}, {1}, {0}};
    int num_muestras = 4;

    // 2. Crear la arquitectura de la red
    Red* red = crear_red(2);
    red->capas[0] = crear_capa(2, 3, &sigmoide); // Capa oculta: 2 entradas, 3 neuronas
    red->capas[1] = crear_capa(3, 1, &sigmoide); // Capa de salida: 3 entradas, 1 neurona

    // 3. El bucle de entrenamiento
    int epocas = 10000;
    double tasa_aprendizaje = 0.1;
    printf("--- Iniciando Entrenamiento para el problema XOR ---\n");
    for (int i = 0; i < epocas; i++) {
        double error_total = 0;
        for (int j = 0; j < num_muestras; j++) {
            Matriz* entrada = matriz_desde_array(entradas_xor_arr[j], 2, 1);
            Matriz* objetivo = matriz_desde_array(salidas_xor_arr[j], 1, 1);

            // Realizar feedforward y backpropagation
            feedforward(red, entrada);
            backpropagate(red, entrada, objetivo, tasa_aprendizaje);

            // Calcular error (opcional, para visualización)
            Matriz* salida_red = red->capas[red->num_capas - 1]->salidas;
            error_total += pow(objetivo->datos[0] - salida_red->datos[0], 2);

            liberar_matriz(entrada);
            liberar_matriz(objetivo);
        }
        if ((i + 1) % 1000 == 0) {
            printf("Época %d, Error: %f\n", i + 1, error_total / num_muestras);
        }
    }
```

```
    printf("--- Entrenamiento Finalizado ---\n\n");

    // 4. Probar la red entrenada
    printf("--- Resultados de la Red Entrenada ---\n");
    for (int i = 0; i < num_muestras; i++) {
        Matriz* entrada = matriz_desde_array(entradas_xor_arr[i], 2, 1);
        Matriz* salida = feedforward(red, entrada);
        printf("Entrada: [%.0f, %.0f] -> Salida: %.4f (Objetivo: %.0f)\n",
               entradas_xor_arr[i][0], entradas_xor_arr[i][1],
               salida->datos[0], salidas_xor_arr[i][0]);
        liberar_matriz(entrada);
    }

    // 5. Liberar toda la memoria
    liberar_red(red);
    printf("\nMemoria liberada. Fin del programa.\n");

    return 0;
}
```

Conclusión del Libro

Si has llegado hasta aquí, has completado un viaje extraordinario. Has pasado de los conceptos matemáticos más fundamentales a construir, desde cero y en un lenguaje de bajo nivel como C, una red neuronal capaz de aprender a resolver problemas complejos.

- En la **Parte I**, forjamos las herramientas del **Cálculo** para entender el cambio y del **Álgebra Lineal** para manipular datos.
- En la **Parte II**, adoptamos el lenguaje de la **Probabilidad y la Estadística** para manejar la incertidumbre y medir nuestros resultados.
- En la **Parte III**, perfeccionamos nuestros métodos de **Optimización** para navegar los complejos paisajes de error de forma eficiente.
- Y finalmente, en la **Parte IV**, hemos ensamblado todas estas piezas en una creación funcional.

No solo has aprendido las matemáticas *para* la IA; has aprendido las matemáticas *de* la IA. Entiendes lo que ocurre bajo el capó, desde la multiplicación de una matriz hasta la actualización de un peso mediante un gradiente. Este conocimiento profundo es la base más sólida que un ingeniero o científico puede tener para innovar y construir la próxima generación de sistemas inteligentes. ¡Felicidades!