### **1. Implementación de Tensor SAMIR MENA**

El Tensor actúa como reemplazo a las estructuras como Eigen::Matrix, permitiendo mantener independencia total de bibliotecas externas y comprensión profunda de las operaciones lineales implicadas en el entrenamiento y propagación en la red neuronal.

El componente Tensor<T, 2> define una estructura genérica de datos multidimensionales para representar matrices numéricas, esenciales para operaciones de redes neuronales. Esta clase, implementada en tensor.h, sirve como base para representar entradas, pesos, salidas, gradientes y más dentro de la red.

#### **Inicialización**

cpp

CopiarEditar

Tensor<T, 2> X(std::array<size\_t, 2>{4, 2});

Esta línea crea un tensor de tipo float, con dimensiones 4×2. Internamente, la clase Tensor usa un std::vector<T> para almacenar los datos en orden lineal, mientras que la forma (shape) se guarda en un std::array<size\_t, N>.

#### **Acceso y modificación**

cpp

CopiarEditar

X(0, 0) = 0;

X(0, 1) = 0;

El operador () ha sido sobrecargado para acceder a elementos en N dimensiones, permitiendo que el acceso sea similar al de matrices convencionales. Esto facilita operaciones tipo:

cpp

CopiarEditar

for (size\_t i = 0; i < X.shape()[0]; ++i) {

for (size\_t j = 0; j < X.shape()[1]; ++j) {

std::cout << X(i, j) << " ";

}

}

#### **Integración con CSV**

La lectura de datos desde archivos externos también se realiza cargando directamente los valores dentro del Tensor. Por ejemplo:

cpp

CopiarEditar

Tensor<T, 2> X = load\_csv\_X("xor\_data.csv", 4, 2);

Esta función llena un tensor X a partir del archivo, donde cada fila representa una muestra y cada columna una característica (en este caso, x1 y x2).

### **2. Dense, NeuralNetwork y Activations MARCO APOLINARIO**

#### **Clase Dense**

Ubicada en dense.h, representa una **capa completamente conectada**. Su constructor recibe dimensiones de entrada y salida, junto con funciones para inicializar pesos y sesgos:

cpp

CopiarEditar

net.add\_layer(std::make\_unique<Dense<T>>(2, 4, w\_init, b\_init));

Esta línea crea una capa densa con:

* 2 neuronas de entrada (por los 2 features del XOR: x1 y x2),
* 4 neuronas ocultas,
* inicialización personalizada: pesos en 0.1 y bias en 0.0.

Internamente, Dense calcula la salida como:

ini

CopiarEditar

output = X \* W + B

Y guarda gradientes para retropropagación.

#### **Clase ReLU**

Ubicada en activation.h, es una función de activación que introduce **no linealidad**. Sin esto, la red solo podría modelar funciones lineales.

cpp

CopiarEditar

net.add\_layer(std::make\_unique<ReLU<T>>());

ReLU transforma cada elemento de la matriz de entrada según:

lua

CopiarEditar

ReLU(x) = max(0, x)

Esto permite que la red aprenda patrones más complejos como el XOR.

#### **Clase NeuralNetwork**

Ubicada en neural\_network.h, es una clase contenedora que organiza las capas y maneja el entrenamiento:

cpp

CopiarEditar

NeuralNetwork<T> net;

net.add\_layer(...);

...

net.train<MSELoss>(X, Y, 1000, 4, 0.1f);

* add\_layer() almacena secuencialmente capas Dense y ReLU.
* train<MSELoss>() realiza el entrenamiento con:  
  + 1000 épocas,
  + batch size 4 (todo el dataset),
  + learning rate de 0.1.

Durante train(), se llama internamente a:

1. forward() → Para obtener predicciones.
2. compute\_loss() → Calcula el error.
3. backward() → Propaga el error hacia atrás.
4. update() → Ajusta pesos y bias con gradientes.

#### **Estructura final usada en tu proyecto**

cpp

CopiarEditar

net.add\_layer(Dense(2, 4)); // Entrada a capa oculta

net.add\_layer(ReLU()); // Activación

net.add\_layer(Dense(4, 1)); // Capa de salida (una neurona)

Esto modela una red con arquitectura **2-4-1**, ideal para resolver el problema XOR.

### **3. Agente y entorno (EnvGym y PongAgent) ANA CORDERO**

Esta parte del código permite probar si el modelo puede usarse en un **entorno simulado**, como si se tratara de un videojuego simple. Aunque se entrena con datos XOR, el código integra la red con un **agente de Pong** y un entorno llamado EnvGym.

#### **Clase EnvGym**

Ubicada en EnvGym.h, representa una versión básica del entorno del juego Pong.

* env.reset() reinicia el entorno y devuelve el **estado inicial** del juego (posición de pelota y paleta).
* env.step(action, reward, done) aplica una acción (mover arriba, quedarse, o mover abajo), y devuelve:  
  + un nuevo estado,
  + una recompensa reward (ej. si devolvió la pelota),
  + y una bandera done para indicar si terminó el episodio.

Ejemplo:

cpp

CopiarEditar

EnvGym env;

float reward;

bool done = false;

auto state = env.reset();

#### **Clase PongAgent**

Ubicada en PongAgent.h, representa al agente que **decide acciones** basadas en una red neuronal. Su constructor recibe el modelo que usará para tomar decisiones:

cpp

CopiarEditar

auto dummy\_model = std::make\_unique<Dense<T>>(3, 3, w\_init, b\_init);

PongAgent<T> agent(std::move(dummy\_model));

**Importante:** En este caso estás pasando una red dummy, no la red entrenada con XOR, por eso el agente solo actúa con una red sin aprendizaje real. Aun así, el flujo simula cómo se **usa una red para decidir acciones**.

#### **Bucle de ejecución del juego**

cpp

CopiarEditar

for (int t = 0; t < 5; ++t) {

int action = agent.act(state); // Toma decisión

state = env.step(action, reward, done); // Ejecuta acción en entorno

std::cout << "Step " << t << " -> reward: " << reward << ", done: " << done << "\n";

if (done) break;

}

Aquí se simulan 5 pasos del juego:

* El agente elige una acción.
* Se aplica en el entorno.
* Se observa la recompensa.
* Se imprime el resultado.

Esto demuestra cómo una red puede integrarse a un **entorno dinámico y autónomo**.

### **4. Ejecución y procesamiento del archivo CSV DIEGO GIL**

Una parte clave de este proyecto fue demostrar que la red neuronal puede **leer datos desde un archivo externo (xor\_data.csv)**, entrenarse con ellos y luego **predecir** resultados. Este flujo representa una simulación real de entrada/salida de datos en aplicaciones de inteligencia artificial.

#### **Archivo xor\_data.csv**

Contiene las combinaciones de entrada y salida del problema XOR, por ejemplo:

CopiarEditar

0,0,0

0,1,1

1,0,1

1,1,0

Donde:

* Las dos primeras columnas son las entradas x1 y x2.
* La tercera columna es la salida esperada y.

#### **Lectura de los datos desde el archivo**

En el código, se utilizan dos funciones especializadas para cargar los datos:

cpp

CopiarEditar

Tensor<T, 2> load\_csv\_X("xor\_data.csv", 4, 2);

Tensor<T, 2> load\_csv\_Y("xor\_data.csv", 4, 2); // Columna 2 es la etiqueta

Ambas funciones procesan línea por línea el archivo:

cpp

CopiarEditar

std::ifstream file(filename);

std::string line;

while (std::getline(file, line) && row < rows) {

std::stringstream ss(line);

std::string value;

// Procesamiento por columnas

}

* load\_csv\_X(...) carga solo las primeras 2 columnas.
* load\_csv\_Y(...) extrae la columna **índice 2**, que es la salida y.

Ambas devuelven tensores listos para entrenar.

#### **Confirmación de la carga exitosa**

Se imprime la forma (shape) de las matrices cargadas:

cpp

CopiarEditar

std::cout << "Shape de X: " << X.shape()[0] << " x " << X.shape()[1] << "\n";

std::cout << "Shape de Y: " << Y.shape()[0] << " x " << Y.shape()[1] << "\n";

La salida debe confirmar:

yaml

CopiarEditar

Shape de X: 4 x 2

Shape de Y: 4 x 1

#### **Predicción final tras entrenamiento**

Después de entrenar el modelo con estos datos, se imprime el resultado:

cpp

CopiarEditar

Tensor<T, 2> predicciones = net.predict(X);

for (size\_t i = 0; i < predicciones.shape()[0]; ++i) {

std::cout << "Input: (" << X(i, 0) << ", " << X(i, 1)

<< ") -> Predicho: " << predicciones(i, 0)

<< " | Esperado: " << Y(i, 0) << "\n";

}

Esto permite **verificar visualmente** si el modelo aprendió correctamente el comportamiento XOR.

### **Ejecución de datos**

El proyecto utiliza un archivo xor\_data.csv como entrada, el cual contiene combinaciones del problema lógico XOR. Este archivo se procesa con funciones personalizadas que leen y convierten los datos en tensores: las dos primeras columnas se cargan como entradas (X) y la tercera como salidas esperadas (Y).

Una vez cargados los datos, la red neuronal se entrena usando el método train() con el algoritmo de optimización SGD y la función de pérdida MSE. Tras el entrenamiento, se ejecuta una predicción con net.predict(X), y se muestra en consola la comparación entre lo predicho y lo esperado. Este paso valida si la red aprendió correctamente el patrón lógico, imprimiendo líneas como:

yaml

CopiarEditar

Input: (0, 1) -> Predicho: 0.49 | Esperado: 1

Esto demuestra que el modelo procesa los datos del CSV, aprende del patrón XOR y es capaz de generar salidas coherentes con el entrenamiento.