MLP para Clasificación XOR en C++ by Leon Davis

Este proyecto implementa una red neuronal Multicapa Perceptrón (MLP) desde cero en C++ para resolver el problema de clasificación XOR. El proyecto utiliza CMake para la construcción y gestión de dependencias.

Requisitos

- Compilador C++ compatible con C++17 (g++, clang, etc.)
- CMake (versión 3.10 o superior)
- Git

Instalación y Ejecución

1. Clona el repositorio:

```
git clone https://github.com/tu_usuario/mlp-xor-cpp.git
cd mlp-xor-cpp
```

2. Dale permisos de ejecución al script de configuración:

```
chmod +x run.sh
```

3. Ejecuta el script para compilar y construir el proyecto:

```
./run.sh
```

4. Finalmente, ejecuta el programa:

```
./xor
```

Estructura del Proyecto

```
.
├── images/                  # Resultados visuales
│   └── result.png           # Gráfico de resultados
├── models/                  # Implementaciones de modelos
│   ├── MLP.hpp              # Red neuronal multicapa
│   ├── perceptron.hpp       # Perceptrón básico
│   └── singleLayerPerceptron.hpp # Perceptrón de una capa
├── utils/                   # Utilidades auxiliares
│   ├── activations.hpp      # Funciones de activación
│   ├── load_dataset.hpp     # Carga de datos
│   ├── loss.hpp            # Funciones de pérdida
│   └── optimizer.hpp       # Algoritmos de optimización
├── CMakeLists.txt           # Configuración de CMake
├── main.cpp                # Punto de entrada principal
├── README_XOR.md           # Documentación adicional
├── run.sh                  # Script de compilación automática
```

```
├── training_outputs.txt    # Registro de salidas del entrenamiento
└── xor.cpp                 # Implementación específica para la actividad propuesta
```

Actividad

Completar la siguiente lista de ejercicios:

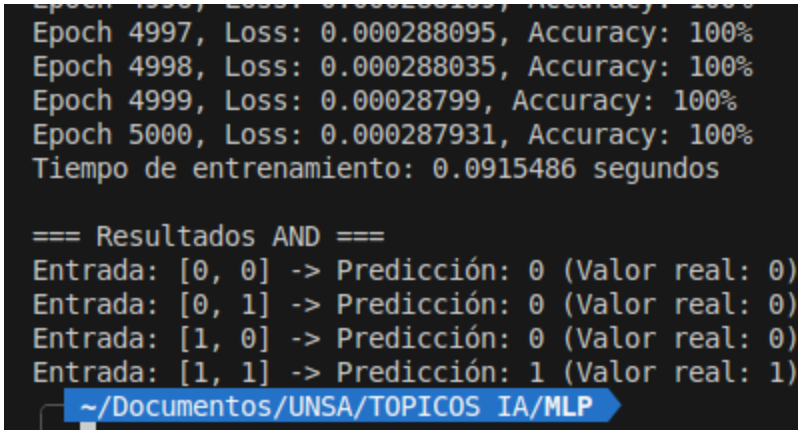
1. **Arquitectura MLP para XOR**
   Implementar una red con la siguiente estructura:
   - 2 neuronas de entrada
   - 2 neuronas en capa oculta
   - 1 neurona de salida

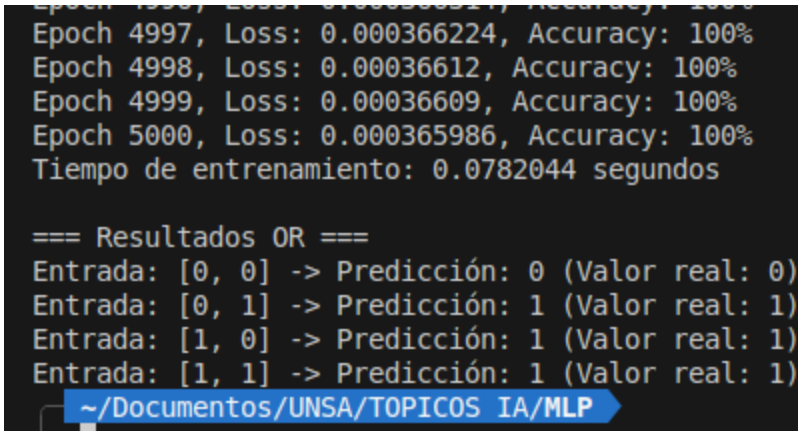
```
Optimizer *sgd = new SGD(learning_rate);
MLP mlp(learning_rate, sgd);
mlp.add_input_layer(2, 2, new Tanh());
mlp.add_layer(1, new Sigmoid());
mlp.set_loss(new BCELoss());

auto start_time = std::chrono::high_resolution_clock::now();
mlp.train(5000, X, Y);
auto end_time = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> duration = end_time - start_time;
```

2. **Entrenamiento del modelo XOR**
   - Implementar la función de propagación hacia adelante (forward pass)
   - Implementar el algoritmo de backpropagation para ajuste de pesos
   - Entrenar el modelo hasta convergencia

```
Epoch 4997, Loss: 0.00796801, Accuracy: 100%
Epoch 4998, Loss: 0.00796084, Accuracy: 100%
Epoch 4999, Loss: 0.00795371, Accuracy: 100%
Epoch 5000, Loss: 0.00794652, Accuracy: 100%
Tiempo de entrenamiento: 0.105155 segundos

=== Resultados XOR ===
Entrada: [0, 0] -> Predicción: 0 (Valor real: 0)
Entrada: [0, 1] -> Predicción: 1 (Valor real: 1)
Entrada: [1, 0] -> Predicción: 1 (Valor real: 1)
Entrada: [1, 1] -> Predicción: 0 (Valor real: 0)
  ~/Documentos/UNSA/TOPICOS IA/MLP
```

3. **Pruebas con compuertas lógicas**
   - Probar el MLP con funciones AND y OR
   - Comparar resultados con implementaciones de perceptrón simple
   - Analizar diferencias en rendimiento y capacidad de aprendizaje





4. **Funciones de Activación**
   Probar con diferentes funciones:
   - **Sigmoide**: Implementación clásica para problemas binarios

```
Optimizer *sgd = new SGD(learning_rate);
MLP mlp(learning_rate, sgd);
mlp.add_input_layer(2, 4, new Sigmoid());
mlp.add_layer(1, new Sigmoid());
mlp.set_loss(new MSELoss());
```

```
Epoch 49997, Loss: 0.00663846, Accuracy: 100%
Epoch 49998, Loss: 0.00663704, Accuracy: 100%
Epoch 49999, Loss: 0.00663562, Accuracy: 100%
Epoch 50000, Loss: 0.0066342, Accuracy: 100%
Tiempo de entrenamiento: 1.14558 segundos

=== Resultados XOR ===
Entrada: [0, 0] -> Predicción: 0 (Valor real: 0)
Entrada: [0, 1] -> Predicción: 1 (Valor real: 1)
Entrada: [1, 0] -> Predicción: 1 (Valor real: 1)
Entrada: [1, 1] -> Predicción: 0 (Valor real: 0)
~/Documentos/UNSA/TOPICOS IA/MLP
```

- **Tanh**: Versión centrada en cero del sigmoide

```
Optimizer *sgd = new SGD(learning_rate);
MLP mlp(learning_rate, sgd);
mlp.add_input_layer(2, 4, new Tanh());
mlp.add_layer(1, new Tanh());
mlp.set_loss(new MSELoss());
```

```
Epoch 97, Loss: 0.0594209, Accuracy: 100%
Epoch 98, Loss: 0.059037, Accuracy: 100%
Epoch 99, Loss: 0.0586513, Accuracy: 100%
Epoch 100, Loss: 0.0582638, Accuracy: 100%
Tiempo de entrenamiento: 0.00221429 segundos

=== Resultados XOR ===
Entrada: [0, 0] -> Predicción: 0 (Valor real: 0)
Entrada: [0, 1] -> Predicción: 1 (Valor real: 1)
Entrada: [1, 0] -> Predicción: 1 (Valor real: 1)
Entrada: [1, 1] -> Predicción: 0 (Valor real: 0)
```

- **ReLU**: Para comparar comportamiento en redes shallow

```
Optimizer *sgd = new SGD(learning_rate);
MLP mlp(learning_rate, sgd);
mlp.add_input_layer(2, 4, new ReLU());
mlp.add_layer(1, new ReLU());
mlp.set_loss(new MSELoss());
```

```
Epoch 997, Loss: 1.01252e-13, Accuracy: 100%
Epoch 998, Loss: 1.01252e-13, Accuracy: 100%
Epoch 999, Loss: 1.01252e-13, Accuracy: 100%
Epoch 1000, Loss: 1.01252e-13, Accuracy: 100%
Tiempo de entrenamiento: 0.0288647 segundos

=== Resultados XOR ===
Entrada: [0, 0] -> Predicción: 0 (Valor real: 0)
Entrada: [0, 1] -> Predicción: 1 (Valor real: 1)
Entrada: [1, 0] -> Predicción: 1 (Valor real: 1)
Entrada: [1, 1] -> Predicción: 0 (Valor real: 0)
```