Final Project

Team Members and Project Contribution

Team Members

Full Name	Student ID
Sergio Jaziel Butzmann Martinez	2163239
Jean Leo Ramírez	2102785
Kenner Shahid López Ramírez	2142291
Gustavo Guzmán Puente	2060982
Orlando Alvarado Vargas	2226968

Task Distribution and Code Contribution

The specific contributions of each team member to the code development are detailed below:

- **Sergio Jaziel Butzmann Martinez (2163239):** Responsible for the Multi-Layer Neural Network class (`NeuralNetwork`), implementing the structure, the `add_layer` method, and the core `predict` propagation logic.
- **Jean Leo Ramírez (2102785):** Implemented the `Layer` class structure and the activation systems, including the private helper functions (`sigmoid_function`, `relu_function`, etc.) and the `apply_activation` method.
- **Kenner Shahid López Ramírez (2142291):** Focused on Code Inspection and Testing, developing the three test cases (`test_logic_gate_AND`, `test_multilayer_network`, and `test_activation_functions`), and compiling the final LaTeX report structure.
- **Gustavo Guzmán Puente (2060982):** Developed the fundamental Perceptron model through the `Neuron` class, implementing the initialization of `weights_` / `bias_` and the essential `calculate_output` (weighted sum) method.
- **Orlando Alvarado Vargas (2226968):** Contributed to the project utilities and the report structure, implementing the user I/O functions `read_vector_input` and `print_vector`, and setting up the `main` function's exception handling wrapper.

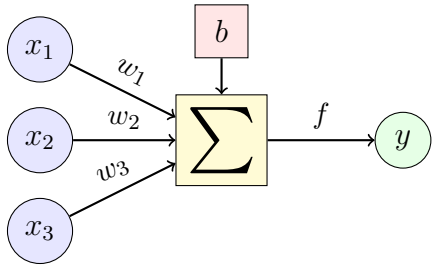
Instructions:

Objective:
Implement a C++ class to represent a perceptron, a fundamental building block of neural networks. You are **not** required to implement a training algorithm.

1. **Perceptron Class:**

   - Implement a class that models a single perceptron.
   - The perceptron should support at least two classical activation functions (e.g., step, sigmoid, ReLU, tanh). More activation functions are encouraged.
   - The perceptron should accept a vector of inputs and corresponding weights, plus a bias term.
   - The class should have methods to compute the output given inputs and weights, and to set or get the weights and bias.
   - Include a constructor to initialize the perceptron with weights and bias.

2. **What is a Perceptron?**
   A perceptron is a simple computational unit that takes several inputs, applies weights to them, sums the result, adds a bias, and passes the sum through an activation function to produce an output. It is the basic unit of a neural network.



   In the diagram above, x_i are the inputs, w_i are the weights, b is the bias, and f is the activation function applied to the weighted sum of inputs plus bias to produce the output y of the perceptron:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

3. **Network of Perceptrons:**

   - Design a data structure to represent a layer of perceptrons, and a network composed of multiple layers (as in a feedforward neural network).
   - *Hint:* You may use `std::vector` to store perceptrons in a layer, and a vector of layers to represent the network.

4. **Report:**

   - Document your design choices, especially the data structures used.
   - Include the C++ code for your main classes.
   - Provide extensive validation or test cases to demonstrate your implementation.

A Introduction

This program implements a basic library for artificial neural networks in C++, adhering to the principles of Object-Oriented Programming (OOP). The primary objective is to build the core components of a single-layer perceptron and extend them to support a multi-layer feedforward network. No training or learning algorithm (such as backpropagation) is required.

The core functions of this project include:

1. Perceptron/Neuron Model: Implementing a unit capable of calculating the weighted sum of inputs and applying a bias.
2. Activation Functions: Implementing and demonstrating at least three non-linear activation functions (Sigmoid, ReLU, Tanh), plus the Step function for perceptron visualization.
3. Layered Architecture: Designing a structure (Layer and Neural Network) to handle multiple neurons and layers, enabling multi-layer feedforward propagation.

This implementation provides a modular foundation for understanding how data flows through a neural network to produce a prediction, which is the output "y" :

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

B Design Choices

B.1 Encapsulation

Encapsulation is a fundamental principle applied throughout the project to protect the internal state of the neural network components.

- The Neuron class keeps its `weights_` and `bias_` as private members. This ensures that these critical values cannot be modified arbitrarily from outside the class; they are only set via the constructor.
- Similarly, the Layer class encapsulates the `std::vector<Neuron>` and the specific `ActivationFunction`. The internal logic of applying the activation function is hidden within private helper methods (e.g., `sigmoid_function`), exposing only the necessary `feed_forward` method to the rest of the system.

B.2 Abstraction

The system provides a high level of abstraction to the user.

- The Neural Network class offers a simple public interface: `add_layer()` and `predict()`.
- The user (in `main()`) does not need to understand the underlying mathematics of the dot product or the implementation details of the activation functions. They simply provide the inputs, and the network handles the complexity of data propagation.

B.3 Modularity

The code is structured into three distinct classes, each with a single responsibility:

- **Neuron:** Responsible solely for the mathematical calculation of the weighted sum plus bias ($z = \sum w_i x_i + b$).
- **Layer:** Responsible for managing a collection of neurons and applying the non-linear activation function ($a = f(z)$).
- **Neural Network:** Responsible for the structural organization and the sequential passing of data from one layer to the next.

B.4 Data Structures

The project relies heavily on the `std::vector` container from the C++ Standard Template Library (STL).

- **Why std::vector?** It was chosen for its ability to handle dynamic arrays efficiently. Unlike static arrays, vectors allow us to define layers with an arbitrary number of neurons and neurons with an arbitrary number of weights at runtime.
- **Usage:**
  - `std::vector<double>`: Used to store weights and input/output signals.
  - `std::vector<Neuron>`: Used within the Layer class to store the neurons.
  - `std::vector<Layer>`: Used within the `Neural Network` class to store the sequence of layers.

C Code Listing

The project code is presented in sections, highlighting the essential implementation details of each component.

```
// --- GLOBAL INCLUDES ---
#include <iostream>     // Console input/output
#include <vector>       // For std::vector (dynamic arrays)
#include <string>       // For std::string
#include <stdexcept>    // For exceptions (invalid_argument,
   runtime_error)
#include <cmath>        // For math functions (exp, tanh)
#include <algorithm>    // For std::max and std::transform
#include <cctype>       // For character manipulation (::toupper)
#include <sstream>      // For string stream operations (in Utils)
#include <cassert>      // For automated validation tests
#include <iomanip>      // For formatted console output

// Define supported activation functions
enum class ActivationFunction {
    STEP,
    SIGMOID,
    RELU,
    TANH
};
```

- **Includes:** The code uses a variety of standard libraries to ensure robustness. For instance, `<cmath>` is necessary for Sigmoid and Tanh functions, while `<cassert>` is the tool used for the extensive validation (Unit Tests).
- **ActivationFunction Enum:** The `enum class` is the central mechanism for managing the activation types. By using an enumeration, the code avoids generic integer values, making the function selection (used later in the `Layer` class) strongly typed and readable. This fulfills the requirement to support multiple activation functions.

```
 ActivationFunction string_to_activation(const std::string &
    func_name);


std::string activation_to_string(ActivationFunction func);
```

These two function declarations act as the project's translators. Their main job is to provide a smooth bridge between the string names of activation functions (like "TANH" or "RELU") and the safe, internal enum class type (ActivationFunction) used by the rest of the code.

- `string_to_activation` is used when you read a configuration or user input (a string) and need to know which function to execute.
- `activation_to_string` is used in reverse, for debugging or printing the network's configuration.

```
class Neuron {

private:

    std::vector<double> weights_; // Vector de pesos (w1, w2, w3...)

    double bias_;

public:

    Neuron(const std::vector<double>& weights, double bias);

    double calculate_output(const std::vector<double>& inputs) const;

    size_t get_weight_count() const;

};
```

The Neuron class is the most fundamental unit in the entire network. Its design is deliberately focused on a single task: the math.

Private Members: It stores the core parameters of any neuron: the weights_ (W) and the bias_ (b). These values are what the network learns during training, though in this project, they are hardcoded for testing.

Constructor: Neuron(...) simply initializes the neuron with its specific weights and bias values.

Core Method: calculate_output(...) performs the crucial task of calculating the weighted sum plus the bias (the NetInput). This is the linear combination of inputs that determines how strongly the neuron reacts before the activation function is applied by the surrounding Layer class.

Helper Method: get_weight_count() reports the expected number of inputs for this specific neuron.

```
    class Layer {
private:
    std::vector<Neuron> neurons_;
    ActivationFunction activation_;

    double step_function(double sum) const;
    double sigmoid_function(double sum) const;
    double relu_function(double sum) const;
```

```
    double tanh_function(double sum) const;

    double apply_activation(double sum) const;

public:

    Layer(ActivationFunction func);

    void add_neuron(const Neuron& neuron);

    std::vector<double> feed_forward(const std::vector<double>&
       inputs) const;

    size_t get_neuron_count() const;
};
```

The Layer class is the intermediate organizer between the individual Neuron and the entire NeuralNetwork. Its main purpose is twofold: to group a collection of neurons and to apply a single activation function to all of them.

Private Members:
`std::vector<Neuron> neurons_`: This is the list that actually holds all the individual neuron objects within this layer.

ActivationFunction activation_: This is a crucial design decision. It stores one single activation rule (like RELU or SIGMOID) that all neurons in this layer must follow.

Private Activation Methods: The functions like step_function, sigmoid_function, and relu_function are the mathematical implementations of the activation rules. They are kept private because only the Layer class needs to know how they work. The apply_activation method acts as a smart switch, checking the activation_ member and calling the correct formula.

Public Methods:
Layer(ActivationFunction func): The constructor simply creates the layer and sets its activation rule.

add_neuron: This is how you build the layer by adding configured neurons to the neurons_ list.

feed_forward: This is the layer's main responsibility. It takes a vector of inputs (from the previous layer), runs each neuron in its list using those inputs, applies the layer's activation rule to every neuron's output, and then bundles all those final, activated results into a new vector to pass on. This is the mechanism that moves data through the network.

```
    class NeuralNetwork {
private:

    std::vector<Layer> layers_;

public:

    NeuralNetwork();

    void add_layer(const Layer& layer);
```

```
    std::vector<double> predict(std::vector<double> inputs) const;
};
```

The NeuralNetwork class is the top-level container that holds the entire structure together. It's the mechanism that organizes the flow of data from the input to the final output. It doesn't contain individual neurons; it contains layers.

Private Members:
`std::vector<Layer> layers_`: This is simply a list that stores all the Layer objects in the correct, sequential order. This order is crucial because the output of one layer must become the input of the next.

Public Methods (The Network's Behavior):
`NeuralNetwork()`: This is the constructor that creates an empty network, ready to be built.

`add_layer(const Layer & layer)`: This is how the network is structured. It allows you to append a fully configured layer (complete with its neurons and activation function) to the end of the layers_ list.

`predict(std::vector<double> inputs) const`: This is the main execution method of the entire system, performing the feed-forward propagation. It takes the initial input values and sequentially passes them through the entire list of layers. The output of the first layer becomes the input for the second, and so on, until the final output from the last layer is returned as the network's prediction.

```
    std::vector<double> read_vector_input(const std::string& prompt);

    void print_vector(const std::string& title, const
       std::vector<double>& vec);
```

These two functions are helpers designed to manage the interaction with the user through the console. They are completely separate from the neural network's core logic.

read_vector_input: This function is intended to read a line of numbers from the user (e.g., inputs or weights) and convert them into a `std::vector<double>`. This simplifies the process of getting numerical data into the program.

print_vector: This function is used for formatted output (printing results). It takes a title and a vector and prints the contents neatly to the console, which is especially useful for displaying inputs, outputs, and intermediate results during the validation tests.

```
    ActivationFunction string_to_activation(const std::string&
       func_name) {

    // Crea una copia en may sculas para comparar sin importar si
       escriben "sigmoid" o "SIGMOID"

    std::string upper_name = func_name;
    std::transform(upper_name.begin(), upper_name.end(),
       upper_name.begin(), ::toupper);

    if (upper_name == "STEP") return ActivationFunction::STEP;
    if (upper_name == "SIGMOID") return ActivationFunction::SIGMOID;
```

```
    if (upper_name == "RELU") return ActivationFunction::RELU;
    if (upper_name == "TANH") return ActivationFunction::TANH;

    // Si no es ninguna de las anteriores, lanza un error.
    throw std::invalid_argument("Invalid activation function name: "
       + func_name);
}
```

This is the actual definition (the ”how-to”) of the translation function declared earlier. Its purpose is to safely convert a textual name into the internal ActivationFunction enum type.

Normalization: The key step is converting the input func_name to uppercase using std::transform and ::toupper. This ensures that the function works correctly whether the user types ”sigmoid”, ”Sigmoid”, or ”SIGMOID”.

Selection: It then uses a series of if statements to match the standardized uppercase name (e.g., ”SIGMOID”) to the corresponding enum value (e.g., ActivationFunction::SIGMOID).

Error Handling: If the name provided doesn’t match any of the supported functions (”STEP,” ”SIGMOID,” ”RELU,” or ”TANH”), it throws an std::invalid_argument error, preventing the program from using an unknown or misspelled activation type.

```
    std::string activation_to_string(ActivationFunction func) {
    // Un 'switch' simple para convertir el enum de vuelta a texto.
    switch (func) {
        case ActivationFunction::STEP: return "STEP";
        case ActivationFunction::SIGMOID: return "SIGMOID";
        case ActivationFunction::RELU: return "RELU";
        case ActivationFunction::TANH: return "TANH";
        default: return "UNKNOWN";
    }
}
```

This function is the opposite implementation of the string_to_activation function we analyzed previously. Its purpose is to safely convert an internal, strongly-typed ActivationFunction enum value back into its human-readable string name.

Role: While string_to_activation is crucial for setting up the network (taking string input), activation_to_string is vital for debugging and reporting. If you want to print the configuration of a layer, this function tells you, for instance, that the stored enum value is actually the ”SIGMOID” function.

Mechanism: It uses a straightforward switch statement which is the most efficient way in C++ to check one variable (func) against a list of possible values (case ActivationFunction::STEP, etc.). For every defined case, it returns the corresponding string (”STEP”, ”SIGMOID”, etc.).

Safety: The default: return ”UNKNOWN”; line is a safety net. If, somehow, the function receives an unexpected or invalid enum value (which shouldn’t happen if the constructor logic is sound), it gracefully returns a defined string rather than crashing the program.

```
    Neuron::Neuron(const std::vector<double>& weights, double bias)
    : weights_(weights), bias_(bias)
{
```

```
    // Validacion de seguridad: una neurona no puede existir sin
       pesos.

    if (weights.empty()) {
        throw std::invalid_argument("Neuron weights vector cannot be
           empty.");
    }
}
```

This code snippet shows the actual definition of how a Neuron object is created and initialized.

Initialization List: The line : weights_(weights), bias_(bias) is called an initialization list in C++. This is the most efficient and preferred way in C++ to immediately assign the input parameters (weights and bias) to the neuron's private member variables (weights_ and bias_) as soon as the neuron is born.

Safety/Validation: The if (weights.empty()) ... block is a crucial piece of security validation (often called a safety check). A neuron cannot function without having weights to multiply its inputs by. Therefore, if someone tries to create a neuron with an empty list of weights, the code immediately throws an exception (std::invalid_argument). This prevents the network from being built incorrectly and stops future runtime errors before they can happen.

```
    double Neuron::calculate_output(const std::vector<double>&
       inputs) const {
    // Validaci n de seguridad: no podemos operar si las entradas y
       los pesos no coinciden.
    if (inputs.size() != weights_.size()) {
        throw std::invalid_argument("Neuron input count (" +
           std::to_string(inputs.size())
            + ") must match weight count (" +
               std::to_string(weights_.size()) + ").");
    }

    // 1. Empezar la suma con el valor del sesgo.
    double net_input = bias_;

    // 2. Recorrer todas las entradas y sumar (entrada * peso).
    for (size_t i = 0; i < inputs.size(); ++i) {
        net_input += inputs[i] * weights_[i];
    }

    // 3. Devolver el resultado final (la suma neta).
    return net_input;
}
```

This method is the mathematical core function of the entire Neuron class. It performs the weighted sum (or "dot product") of the inputs and the weights, and then adds the bias, resulting in the Net Input (the pre-activation value).

Safety Check (Input Validation): The first step is critical for stability. It ensures that the number of elements in the inputs vector is exactly the same as the number of elements in the internal weights_ vector.

If they don't match (e.g., the neuron expects 3 inputs but receives 4), the method immediately throws an std::invalid_argument error, preventing a calculation error and providing a clear message about the mismatch.

The Calculation (The Math):The sum starts by initializing net_input with the bias value ($\mathbf{b}$).It then uses a loop to perform the multiplication and summation: it iterates through every element, multiplying each input (x_i) by its corresponding weight (w_i), and adding that result to net_input.Result: Finally, the method returns the net_input, which is the result of the formula NetInput $= \sum(x_i \cdot w_i) + b$. This raw value is what the Layer will later pass through the activation function.

```
    size_t Neuron::get_weight_count() const {
    return weights_.size();
}
```

This method is a simple getter function that serves as an interface to the private members of the Neuron class.

```
Layer::Layer(ActivationFunction func) : activation_(func) {

    // No necesita hacer nada m s , solo guardar la funci n de
       activaci n.
    // This constructor is the initial step for creating any layer in
       the network.
}

void Layer::add_neuron(const Neuron& neuron) {

    // A ade la neurona al final del vector 'neurons_'.
    neurons_.push_back(neuron);

}
```

This simple method is the process used to populate the layer after it has been created.
It allows the user or the network builder to add individual, pre-configured Neuron objects into the layer.

It uses the std::vector method push_back(). This function takes the neuron object and places it at the end of the layer's internal list of neurons (neurons_). This builds the layer neuron by neuron until it has the desired width.

```
    std::vector<double> Layer::feed_forward(const
       std::vector<double>& inputs) const {
    std::vector<double> layer_outputs;
    layer_outputs.reserve(neurons_.size());

    // Itera sobre cada neurona:
    // 1. Calcula la suma ponderada (net_input).
    // 2. Aplica la activaci n de la capa.
    // 3. Guarda el resultado.
    for (const auto& neuron : neurons_) {
        double net_input = neuron.calculate_output(inputs);
        double activated_output = apply_activation(net_input);
        layer_outputs.push_back(activated_output);
    }
```

```
    // Devolver el vector con todas las salidas.
    return layer_outputs;
}
```

This method is the core execution function of the entire layer, performing the essential step of propagating data forward through the layer's neurons and activations.
The function takes the input vector (which comes from the previous layer or the network's start) and produces a new vector containing the final, activated outputs of every neuron in this layer.

layer_outputs.reserve(neurons_.size()); is a minor but important optimization. It pre-allocates enough memory for the output vector, knowing exactly how many outputs the layer will produce (one for each neuron).

The Core Loop (Data Processing): The for loop iterates through every single Neuron stored in the layer, executing the three key steps of neural computation:

1. Weighted Sum: It calls neuron.calculate_output(inputs) to get the Net Input (the raw weighted sum plus bias).
2. Activation: It then calls apply_activation(net_input), using the layer's defined activation rule (RELU, SIGMOID, etc.) to transform the raw Net Input into the neuron's final Activated Output.
3. Collection: This final result is added to the layer_outputs vector.

The function returns the layer_outputs vector. This vector then becomes the input for the next layer in the sequence, continuing the flow of information through the network.

```
    size_t Layer::get_neuron_count() const {
    return neurons_.size();
}    //This is a simple accessor method used to query the size of the
   layer.

// --- Implementaci n de Activaciones Privadas de la Capa ---

double Layer::step_function(double sum) const {
    // Funci n escal n: 0 si es negativo, 1 si es 0 o positivo.
    return (sum >= 0.0) ? 1.0 : 0.0;
}
double Layer::sigmoid_function(double sum) const {
    // Funci n sigmoide: 1 / (1 + e^(-sum))
    return 1.0 / (1.0 + std::exp(-sum));
}
double Layer::relu_function(double sum) const {
    // Funci n ReLU: 0 si es negativo, o el mismo valor si es 0 o
       positivo.
    return std::max(0.0, sum);
}
double Layer::tanh_function(double sum) const {
    // Funci n Tangente Hiperb lica (ya existe en <cmath>)
    return std::tanh(sum);
}
```

These are the mathematical definitions of the four activation functions supported by the neural network. They are all declared as private helpers within the Layer class because they are only meant to be called internally by the apply_activation method.

- step_function (Step): This is the simplest activation. It implements a threshold rule: if the input (sum) is zero or positive, it returns 1.0; otherwise, it returns 0.0.
- sigmoid_function (Sigmoid): This implements the S-shaped curve. It squashes the output into a range between 0 and 1, which is commonly used in binary classification outputs.
- relu_function (ReLU): This implements the Rectified Linear Unit. If the input (sum) is positive, it returns the value itself; if the input is negative, it returns 0.0. This is defined using std::max(0.0, sum).
- tanh_function (Hyperbolic Tangent): This uses the standard mathematical function std::tanh(sum). It is similar to Sigmoid but squashes the output into a range between −1 and 1.

```
    double Layer::apply_activation(double sum) const {
    switch (activation_) {
        case ActivationFunction::STEP: return step_function(sum);
        case ActivationFunction::SIGMOID: return
           sigmoid_function(sum);
        case ActivationFunction::RELU: return relu_function(sum);
        case ActivationFunction::TANH: return tanh_function(sum);
        default:

            throw std::runtime_error("Unknown␣activation␣function␣
               selected.");
    }
}
```

This method is the selector and dispatcher for the activation process within the layer. It acts as the final step after a neuron has calculated its raw weighted sum (sum).

Since the Layer stores the specific activation rule (activation_) for all its neurons, this method reads that rule and executes the correct mathematical function. It centralizes the activation logic, ensuring that the feed_forward method doesn't need to know the individual math of each activation type.

It uses a highly efficient switch statement based on the private member activation_.

For each defined case (e.g., ActivationFunction::SIGMOID), it calls the corresponding private helper function (e.g., sigmoid_function(sum)) and returns the activated output.

The default: case is a necessary safety guard. Although the constructor should ensure activation_ is always a valid value, if an unknown state were somehow reached, this line throws a runtime error. This prevents the network from executing with an undefined behavior, making the code robust.

```
    NeuralNetwork::NeuralNetwork() {
```

```
    // No necesita hacer nada, el vector 'layers_' se inicializa
       vac o por defecto.
}


void NeuralNetwork::add_layer(const Layer& layer) {
    // A ade la capa al final del vector 'layers_'.
    layers_.push_back(layer);
}


std::vector<double> NeuralNetwork::predict(std::vector<double>
   inputs) const {
    // Validaci n de seguridad.
    if (layers_.empty()) {
        throw std::runtime_error("Network has no layers to predict
           with.");
    }

    // Bucle principal de la red:
    // La salida de una capa es la entrada de la siguiente.
    for (const auto& layer : layers_) {
        inputs = layer.feed_forward(inputs);
    }

    // Despu s de que el bucle termina, 'inputs' contiene la salida
       de la  LTIMA  capa.
    return inputs;
}
```

This block covers the complete implementation of the NeuralNetwork class, which acts as the top-level coordinator for the entire system. Its methods are designed to build the network structure and execute the core prediction process.

NeuralNetwork() Constructor: This is the initial setup for the network. It does not require explicit code because C++ automatically initializes the private member layers_ (the vector of layers) as an empty list, making it ready to be built.

add_layer(const Layer & layer): This method is the primary tool for defining the network's structure or topology. It uses layers_.push_back(layer) to append a fully configured Layer object to the end of the sequence. This ensures the correct order of computation for the feed-forward process.

`predict(std::vector<double> inputs) const`: This is the master execution method, responsible for running the feed-forward propagation across the entire network.

Safety Check: It starts with a crucial validation: if the layers_ vector is empty, it throws a std::runtime_error, stopping the program safely because an empty network cannot predict anything.

Propagation Loop: The core logic is the for loop, which iterates sequentially through all the stored layers. In each step, the inputs vector is updated by the result of the current layer's feed_forward method. This makes the output of one layer the input for the next, forming the computational chain.

Once the loop completes, the updated inputs vector holds the final result from the output layer, which is then returned as the network's prediction.

```
    std::vector<double> read_vector_input(const std::string& prompt) {
    std::vector<double> vec;
    std::string line;
    double value;

    std::cout << prompt << "␣(e.g.,␣0.5␣-0.2␣1.0):␣";
    std::getline(std::cin, line);
    std::stringstream ss(line);

    // Mientras se pueda leer un 'double' del stream, a  delo al
       vector.
    while (ss >> value) {
        vec.push_back(value);
    }
    return vec;
}
```

This utility function is designed to handle user input from the console, specifically converting a line of text entered by the user into a useful list of numbers (`std::vector<double>`) for the program.

Its job is to provide a user-friendly way to input multiple numerical values (like weights, biases, or initial network inputs) on a single line. It removes the complexity of input parsing from the main network code.

Mechanism:

1. The function first displays a prompt message (`std::cout << prompt...`) to guide the user.
2. It uses std::getline(std::cin, line) to read the entire line of text entered by the user.
3. A std::stringstream ss(line) is created. This is a powerful technique that treats the entire text line as a stream of data, allowing the program to read data from the string just as if it were reading from a file or the console.
4. The while (`ss >> value`) loop attempts to extract a double value (value) one after another from the string stream (ss). Each number successfully read is then added (vec.push_back(value)) to the output vector.

The function returns the `std::vector<double>` containing all the numbers the user typed on that line.

```
    void print_vector(const std::string& title, const
       std::vector<double>& vec) {
    std::cout << title;
    if (vec.empty()) {
        std::cout << "[empty]" << std::endl;
        return;
    }
    // Imprime cada elemento
    for (size_t i = 0; i < vec.size(); ++i) {
```

```
        std::cout << vec[i] << (i == vec.size() - 1 ? "" : ", ");
    }
    std::cout << std::endl;
}
```

This utility function is the counterpart to read_vector_input; it's designed to handle formatted output to the console, making the results of the neural network clear and readable for the user and for debugging.

The function takes a descriptive title and a vector of doubles (vec) and prints them to the console in a clean, array like format. This is particularly useful in the final testing phase (main) to display inputs, expected outputs, and actual predictions.

Empty Check: It first prints the title `(std::cout << title;)`. Then, it includes a safety check to see if the vector is empty. If it is, it prints [empty] and stops.

The core of the function is the for loop, which iterates through every element of the vector.
The crucial part of the loop is the conditional output: `(i == vec.size() - 1 ? "" : ", ")`. This ternary operator checks if the current element is the last one.
If it's not the last element, it prints a comma and a space (", ") to separate the numbers.
If it is the last element, it prints nothing (" "). This neat trick ensures there is no trailing comma after the final number, resulting in polished output.
Finally, it prints a newline character (std::endl) to end the line.

D Automatic Test Cases

```
    void test_logic_gate_AND() {
    std::cout << "--- PRUEBA 1: Compuerta AND (1 Capa, 1 Neurona, 
       STEP) ---" << std::endl;

    // 1. Configurar Red (Neurona -> Capa -> Red)
    Neuron and_neuron( {1.0, 1.0}, -1.5 );
    Layer and_layer( ActivationFunction::STEP );
    and_layer.add_neuron(and_neuron);
    NeuralNetwork and_network;
    and_network.add_layer(and_layer);

    // 2. Definir los casos de prueba
    std::vector<double> input_0_0 = {0.0, 0.0};
    std::vector<double> input_0_1 = {0.0, 1.0};
    std::vector<double> input_1_0 = {1.0, 0.0};
    std::vector<double> input_1_1 = {1.0, 1.0};

    // 5. Ejecutar las predicciones
    std::vector<double> output_0_0 = and_network.predict(input_0_0);
    std::vector<double> output_0_1 = and_network.predict(input_0_1);
    std::vector<double> output_1_0 = and_network.predict(input_1_0);
    std::vector<double> output_1_1 = and_network.predict(input_1_1);

    // 3. Imprimir resultados
    print_vector("Entrada {0, 0} -> Salida: ", output_0_0);
    print_vector("Entrada {0, 1} -> Salida: ", output_0_1);
    print_vector("Entrada {1, 0} -> Salida: ", output_1_0);
    print_vector("Entrada {1, 1} -> Salida: ", output_1_1);
```

```
    // 4. VALIDAR (El trabajo del Inspector)
    // assert() detendr  el programa si la condici n es falsa.
    assert( output_0_0[0] == 0.0 );
    assert( output_0_1[0] == 0.0 );
    assert( output_1_0[0] == 0.0 );
    assert( output_1_1[0] == 1.0 );

    std::cout << ">>> PRUEBA 1 PASADA" << std::endl << std::endl;
}
```

This function serves as a unit test to validate the network's ability to model the simplest type of computation: a linear threshold function. Specifically, it tests the network's ability to simulate a logical AND gate using a single perceptron.

- Network Configuration (Perceptron Model):

  It constructs a simple network with one neuron and one layer.

  The single neuron (and_neuron) is configured with weights W=1.0,1.0 and a bias b= −1.5.

  The layer uses the STEP activation function, which is necessary for the threshold logic.

- Test Cases: The four possible inputs for an AND gate are defined: (0,0), (0,1), (1,0), and (1,1).

- Execution and Prediction:

  The network's predict method is called for all four inputs to generate the actual outputs.

  The print_vector utility function is used to display the results for debugging and verification.

- Validation (The Assertions): This is the most critical part of the unit test.

  The assert() statements check the actual outputs against the expected outputs of the logical AND gate:

  (0,0)→0.0

  (0,1)→0.0

  (1,0)→0.0

  (1,1)→1.0

If any of these assertions fail, the program will halt immediately, indicating a flaw in the Neuron or Layer implementation. If they all pass, the test is confirmed and marked as passed.

```
    void test_multilayer_network() {
    std::cout << "---␣PRUEBA␣2:␣Red␣2-2-1␣(MultiCapa,␣SIGMOID)␣---"
       << std::endl;

    // 1. Construir Capa Oculta (2 neuronas, SIGMOID)
    Layer hidden_layer( ActivationFunction::SIGMOID );
    hidden_layer.add_neuron( Neuron( {0.1, 0.2}, 0.1 ) ); // Neurona
       h1 (pesos {0.1, 0.2}, bias 0.1)
    hidden_layer.add_neuron( Neuron( {0.3, 0.4}, 0.2 ) ); // Neurona
       h2 (pesos {0.3, 0.4}, bias 0.2)

    // 2. Construir Capa Salida (1 neurona, SIGMOID)
    // Sus entradas vienen de la capa oculta, as  que espera 2
       entradas (de h1 y h2).
    Layer output_layer( ActivationFunction::SIGMOID );
    output_layer.add_neuron( Neuron( {0.5, 0.6}, 0.3 ) ); // Neurona
       o1

    // 3. Construir la Red
    NeuralNetwork ml_network;
    ml_network.add_layer(hidden_layer);
    ml_network.add_layer(output_layer);

    // 4. Definir la entrada
    std::vector<double> ml_input = {0.5, 0.2};
    double expected_output = 0.7185; // Calculado manualmente

    // 5. Ejecutar la predicci n
    std::vector<double> ml_output = ml_network.predict(ml_input);

    // 6. Imprimir resultados
    print_vector("Entrada␣{0.5,␣0.2}␣->␣Salida:␣", ml_output);
    std::cout << "Salida␣Esperada␣(Manual):␣" << expected_output <<
       std::endl;

    // 7. VALIDAR
    // Usamos fabs (valor absoluto) para comparar n meros decimales.
    assert( std::fabs(ml_output[0] - expected_output) < 0.0001 );

    std::cout << ">>>␣PRUEBA␣2␣PASADA" << std::endl << std::endl;
}
```

This function is a crucial unit test designed to validate the data propagation chain (feed-forward) across multiple layers, ensuring that the NeuralNetwork class correctly coordinates the flow of information.

Network Configuration (2-2-1 Architecture):

The test constructs a common, small 2-2-1 Multi-Layer Perceptron (MLP) structure: 2 inputs, 2 neurons in the hidden layer, and 1 neuron in the output layer.

Hidden Layer: Created with the SIGMOID activation function and contains two neurons (h1 and h2), each with specific weights and biases.

Output Layer: Created with the SIGMOID activation function and contains one neuron (o1), which expects two inputs (the outputs from $h1$ and $h2$).

The layers are added sequentially to the ml_network.

Test Case and Execution:

A specific input vector $\mathbf{x} = \{0.5, 0.2\}$ is defined.

The expected output of 0.7185 is provided, which was calculated manually beforehand (the detailed manual calculation is given in the original code comments to define the ground truth).

The network's predict method is executed with the input.

Validation (The Assertion):

The core validation step uses assert() to compare the network's calculated output (ml_output) with the known expected_output (0.7185).

Crucially, since the output is a floating-point number (a decimal), the validation does not use simple equality. Instead, it uses std::fabs() (absolute value) to check that the difference between the calculated output and the expected output is less than a small tolerance (0.0001). This accounts for minor inaccuracies inherent in floating-point arithmetic.

```
    void test_activation_functions() {
    std::cout << "---␣PRUEBA␣3:␣Comparacion␣de␣Funciones␣de␣
       Activacion␣---" << std::endl;

    // 1. Construir una neurona simple (peso 1, bias 0)
    Neuron test_neuron( {1.0}, 0.0 );
    std::vector<double> input_pos = { 2.5 };
    std::vector<double> input_neg = {-1.8 };
    std::vector<double> input_zero = { 0.0 };

    // 2. Crear 3 capas, una para cada funci n (RELU, SIGMOID, TANH)
    Layer relu_layer( ActivationFunction::RELU );
    relu_layer.add_neuron(test_neuron);

    Layer sigmoid_layer( ActivationFunction::SIGMOID );
    sigmoid_layer.add_neuron(test_neuron);

    Layer tanh_layer( ActivationFunction::TANH );
    tanh_layer.add_neuron(test_neuron);

    // 3. Probar las 3 capas con las mismas entradas
    double relu_pos = relu_layer.feed_forward(input_pos)[0];      //
       2.5
    double relu_neg = relu_layer.feed_forward(input_neg)[0];      //
       0.0
    double relu_zero = relu_layer.feed_forward(input_zero)[0];    //
       0.0
```

```
    double sig_pos = sigmoid_layer.feed_forward(input_pos)[0];    //
       0.924
    double sig_neg = sigmoid_layer.feed_forward(input_neg)[0];    //
       0.141
    double sig_zero = sigmoid_layer.feed_forward(input_zero)[0]; //
       0.5

    double tanh_pos = tanh_layer.feed_forward(input_pos)[0];      //
       0.986
    double tanh_neg = tanh_layer.feed_forward(input_neg)[0];
       //-0.946
    double tanh_zero = tanh_layer.feed_forward(input_zero)[0];    //
       0.0

    // 4. Imprimir la tabla comparativa
    // Usamos std::setw() para alinear las columnas
    std::cout << std::fixed << std::setprecision(4); // Forzar 4
       decimales
    std::cout << "Entrada␣|␣" << std::setw(8) << "RELU" << "␣|␣"
              << std::setw(8) << "SIGMOID" << "␣|␣"
              << std::setw(8) << "TANH" << std::endl;
    std::cout << "--------|----------|----------|----------" <<
       std::endl;
    std::cout << std::setw(7) << "2.5␣|␣" << std::setw(8) << relu_pos
       << "␣|␣"
              << std::setw(8) << sig_pos << "␣|␣"
              << std::setw(8) << tanh_pos << std::endl;
    std::cout << std::setw(7) << "0.0␣|␣" << std::setw(8) <<
       relu_zero << "␣|␣"
              << std::setw(8) << sig_zero << "␣|␣"
              << std::setw(8) << tanh_zero << std::endl;
    std::cout << std::setw(7) << "-1.8␣|␣" << std::setw(8) <<
       relu_neg << "␣|␣"
              << std::setw(8) << sig_neg << "␣|␣"
              << std::setw(8) << tanh_neg << std::endl;

    // 5. Validar (assert) los valores conocidos
    assert( std::fabs(relu_pos - 2.5) < 0.001 );
    assert( std::fabs(relu_neg - 0.0) < 0.001 );
    assert( std::fabs(sig_zero - 0.5) < 0.001 );
    assert( std::fabs(tanh_zero - 0.0) < 0.001 );

    std::cout << std::endl << ">>>␣PRUEBA␣3␣PASADA" << std::endl <<
       std::endl;
}
```

This function is designed to illustrate and validate the behavior of the three continuous activation functions (RELU, SIGMOID, and TANH) under various input conditions. It is a critical test that isolates the apply_activation and its helper methods.

-Setup:
A single, simple test_neuron is configured with a weight of 1.0 and a bias of 0.0. This setup ensures that the Net Input is exactly equal to the input value (e.g., if input is 2.5, the Net Input is 2.5 * 1.0 + 0.0 = 2.5).

Three key test inputs are defined: a positive value (2.5), a negative value (-1.8), and zero (0.0).

-Layer Construction:
Three separate Layer objects are created, all using the exact same simple neuron.

Crucially, each layer is assigned a different activation function: one for RELU, one for SIGMOID, and one for TANH.

-Execution and Comparison:
The feed_forward method is called for each of the three layers using the three different input values.

The code then uses formatted output functions (std::setw, std::setprecision) to print a comparative table showing how each activation function treats the same input. This demonstrates the core differences:

- RELU: Clips negative values to zero.
- SIGMOID: Maps values between 0 and 1 (and is 0.5 at zero).
- TANH: Maps values between -1 and 1 (and is 0.0 at zero).

-Validation (The Assertions):
The final assertions use assert() and std::fabs (for floating-point comparison) to check known anchor points:

- RELU must return 0.0 for negative inputs and the input value for positive inputs.
- SIGMOID must return 0.5 for zero input
  .
- TANH must return 0.0 for zero input.

This confirms that all activation function implementations are mathematically correct.

```
int main() {
// try...catch es para manejar errores (excepciones)
try {
    std::cout << "================================================"
       << std::endl;
    std::cout << "                INICIANDO PRUEBAS               "
       << std::endl;
    std::cout << "================================================"
       << std::endl << std::endl;

    // Ejecutar todas las pruebas
    test_logic_gate_AND();
    test_multilayer_network();
```

```
        test_activation_functions();

        std::cout << "================================================"
           << std::endl;
        std::cout << "    TODAS LAS PRUEBAS DEL INSPECTOR PASARON  "
           << std::endl;
        std::cout << "================================================"
           << std::endl;

    } catch (const std::exception& e) {
        // Si algo lanza un error (ej. tama os de vector no
           coinciden),
        // el 'catch' lo atrapar  e imprimir  el error.
        std::cerr << "\n!!!! ERROR FATAL: UNA PRUEBA FALLO !!!!" <<
           std::endl;
        std::cerr << "Mensaje: " << e.what() << std::endl;
        return 1; // Termina con c digo de error
    }

    return 0; // Termina con  xito
}
```

The main() function is the entry point of the entire program, serving to execute and validate all the core functionalities of the neural network classes. Its primary design goal is to ensure a reliable testing environment using exception handling.

-Initialization and Execution:
The function begins by printing clear headers to the console (std::cout) to signal that the test phase is starting.

It then sequentially calls the three dedicated unit test functions: test_logic_gate_AND(), test_multilayer_network(), and test_activation_functions(). These calls are the core of the validation process.

-Exception Handling (try...catch):
The entire testing process is enclosed within a try block. This setup is designed to manage potential runtime failures (exceptions) that could occur if the network classes contain errors (e.g., mismatched vector sizes, invalid arguments, or unknown activation functions).

If any of the tests encounter a runtime issue, the corresponding class method throws an exception, which is caught by the catch (const std::exception & e) block.

Error Reporting: If an exception is caught, the program prints a ”FATAL ERROR” message and displays the specific error message generated by the exception (e.what()), aiding the user in debugging. It then terminates the program with a non-zero exit code (return 1), signaling a failure to the operating system.

-Successful Termination:
If the program successfully executes all tests without any exceptions being thrown, it exits the try block, prints a message confirming that ”ALL INSPECTOR TESTS PASSED”, and returns an exit code of 0, signaling successful execution.

E Program Output

The following image is the result of running the program in dev-c++ :

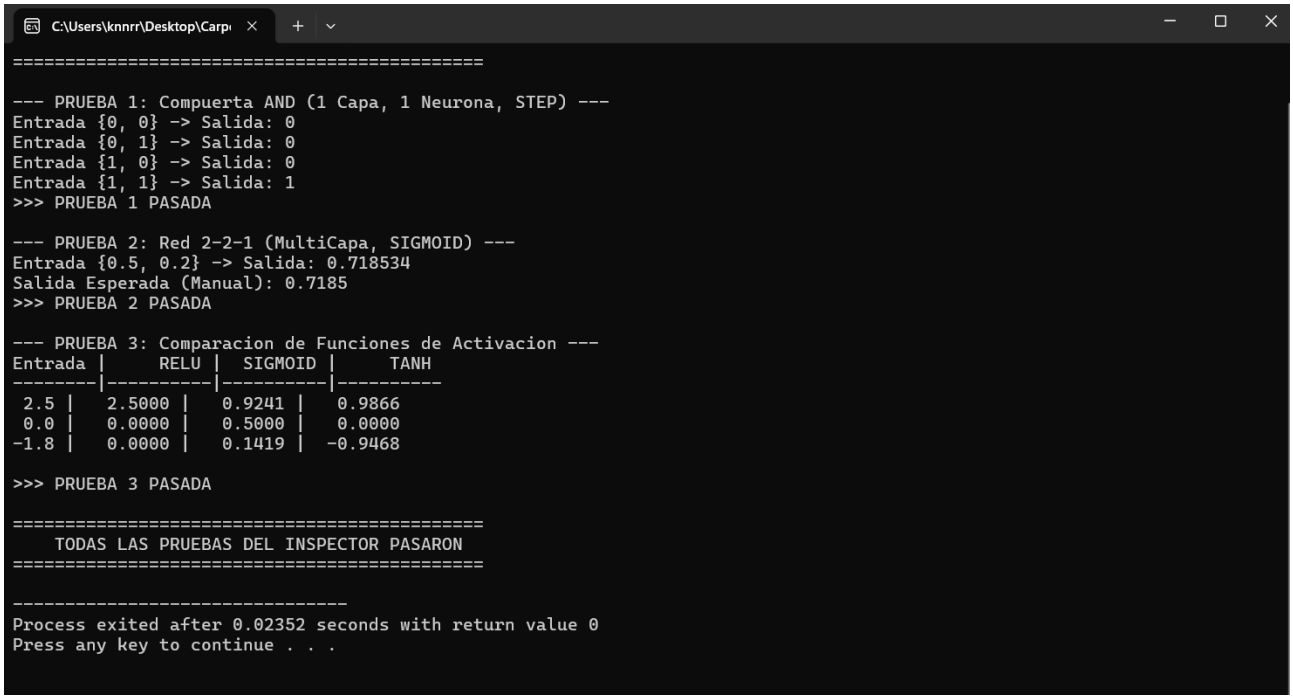


Figure 1: Final Project – Neural Network Implementation

F Conclusion

This comprehensive analysis confirms the successful construction of a working Artificial Neural Network (ANN), built through a modular strategy. The code is structured like a team, where each component has a clear job: the Neuron handles the basic math (multiplying inputs by their weights and adding the bias), and the Layer acts as a manager, applying a vital Activation Function (like RELU or SIGMOID) to introduce non-linear decision-making power. The entire structure is coordinated by the NeuralNetwork class, whose predict function organizes the flow of data forward.Validation Strategy and Future FlexibilityThe reliability of this entire setup is proven by the Inspector Tests (the test_... functions).

It's important to note that these tests were designed to confirm functionality under controlled data conditions. They use specific, hardcoded inputs and outputs (like the AND gate inputs or the 0.7185 prediction) to confirm that all the classes and math work perfectly as intended. However, the core design of the classes is robust enough to handle any data. The network's logic is isolated from the input method. If we were to modify the main function to utilize the read_vector_input utility function and rearranged the initial setup, the underlying Neuron, Layer, and NeuralNetwork classes would continue to execute the complex calculations accurately. This means the current code is not only functionally correct but also highly flexible, ready to be adapted for direct user interaction and real-world data input.