👤 Tarea 1: El Documentador (Líder del Reporte)

Objetivo Principal: Obtener los **30 Puntos** de **"Claridad de Reporte y Código"** y los **25 Puntos** de **"Validación y Pruebas"** (en la sección del reporte). Eres el *integrador final*.

Tu criterio de "Excelente" (100 pts):

- *"Reporte bien estructurado, perspicaz, con lenguaje pulido. Código limpio, bien comentado y lógicamente organizado."*
- *"Pruebas extensas y relevantes con resultados documentados e interpretación significativa."*

Plan de Acción Paso a Paso:

1. **Día 1: Configuración**
    - Crea el proyecto en Overleaf (o tu editor de LaTeX local).
    - Importa FinalProject.tex y style.cls.
    - **Inmediatamente:** Llena la sección de \👥 Team Guidelines con los nombres, IDs y un *borrador* de las contribuciones de cada quien.
    - Crea los esqueletos de las secciones del reporte: "Introducción", "Diseño del Perceptrón", "Funciones de Activación", "Arquitectura Multi-Capa" y "Pruebas y Validación".
2. **Día 2-3: Recolección y Documentación del Diseño**
    - **Pide a Tarea 2:** El archivo Perceptron.hpp.
    - **Pide a Tarea 3:** El archivo ActivationFunctions.hpp.
    - **Pide a Tarea 4:** Los archivos Layer.hpp y Network.hpp.
    - **Entrevista** a tus compañeros:
        - Pregúntale a Tarea 4: "¿Por qué std::vector? ¿Cómo se comunica la clase Network con la clase Layer?"[1111].
        - Pregúntale a Tarea 2: "¿Cómo decidiste pasar la función de activación al perceptrón?"
    - Usa sus respuestas para escribir la sección de **"Decisiones de Diseño"**[2222]. Esto es lo que te dará los puntos de "perspicaz".
    - Crea un **diagrama de flujo** de la propagación hacia adelante (puedes usar TikZ como en el ejemplo [3] o una herramienta online). Esto es un requisito explícito.

3. **Día 4: Integración de Código y Pruebas**
    - **Pide a Tarea 5:** El main.cpp (o el archivo de pruebas) y los *resultados exactos* de la consola.
    - Pega el código *limpio y comentado* de las clases principales (Perceptron, Layer, Network) en el .tex usando el entorno \begin{codelisting}[4444].
    - Pega el código de las *pruebas* de Tarea 5 en la sección de "Validación".
    - Pega los *resultados* de la consola de Tarea 5 y añade la **"interpretación significativa"** (Ej: "Como se observa, al pasar la entrada [1, 1] a la red XOR, el resultado es 0.05, lo cual es cercano al valor esperado de 0, demostrando el éxito de la propagación").
4. **Día 5: Pulido Final**
    - Lee todo el reporte de principio a fin. Corrige gramática y asegúrate de que el lenguaje sea "pulido".
    - Compila el PDF final.
    - Prepara el .zip de entrega con: ReporteFinal.pdf, Perceptron.cpp, Perceptron.hpp, Layer.cpp, Layer.hpp, Network.cpp, Network.hpp, ActivationFunctions.hpp y main.cpp.

👤 Tarea 2: El Arquitecto (Clase Perceptron)

Objetivo Principal: Obtener los **15 Puntos** de **"Implementación del Perceptron"**.

Tu criterio de "Excelente" (100 pts):

- *"Totalmente implementado con lógica, estructura y comportamiento de entrada/salida correctos."*

Plan de Acción Paso a Paso:

1. **Día 1: Archivos y Header (Perceptron.hpp)**
    - Crea Perceptron.hpp y Perceptron.cpp.
    - En Perceptron.hpp, define tu clase. Necesitarás:
        - #include <vector>
        - #include "ActivationFunctions.hpp" (Coordina con Tarea 3 para el nombre del enum).

        - **Miembros Privados:** double bias;, std::vector<double> weights;.
        - **Miembros Públicos:**
        - Constructor: Perceptron(const std::vector<double>& initialWeights, double initialBias);[5555].
        - Setters/Getters: void setWeights(const std::vector<double>& newWeights);, std::vector<double> getWeights() const; (y lo mismo para bias)[6666].
        - Método de cómputo: double feedForward(const std::vector<double>& inputs, ActivationType actType);.
2. **Día 2: Implementación (Perceptron.cpp)**
    - Implementa el constructor (simplemente asigna los valores a los miembros).
    - Implementa los setters y getters.
    - **Implementa feedForward:** Esta es tu lógica principal.
        1. Calcula la suma ponderada: $sum = \sum(w_i * x_i) + b$[7777].
        2. *Importante:* Añade una verificación de seguridad: if (inputs.size() != weights.size()) { // lanzar error }.
        3. Llama a la función de activación (de Tarea 3): return ActivationFunctions::activate(sum, actType);.
3. **Día 3: Comentarios y Entrega**
    - Añade comentarios "Doxygen" a tu header (ej: /// @brief Calcula la salida del perceptrón...). Esto es clave para la rúbrica de "código bien comentado".
    - Envía Perceptron.hpp y Perceptron.cpp a **Tarea 1 (Documentador)**, **Tarea 4 (Ingeniero)** y **Tarea 5 (Validador)**.

👤 Tarea 3: El Especialista (Funciones de Activación)

Objetivo Principal: Obtener los **15 Puntos** de **"Múltiples Funciones de Activación"**.

Tu criterio de "Excelente" (100 pts):

- *"Tres o más funciones de activación implementadas correctamente y comparadas significativamente."*

Plan de Acción Paso a Paso:

1. **Día 1: Archivo y Definiciones (ActivationFunctions.hpp)**
    - Crea ActivationFunctions.hpp (probablemente no necesites un .cpp).
    - Incluye <cmath> y <algorithm>.
    - Define el enum: enum ActivationType { SIGMOID, RELU, TANH, STEP }; (Elige al menos 3, pero 4 es mejor ya que el PDF las menciona [8888]).
    - Define tu namespace o clase estática, por ejemplo namespace ActivationFunctions { ... }.
2. **Día 2: Implementación**
    - Implementa las funciones estáticas:
        - inline double sigmoid(double x) { return 1.0 / (1.0 + std::exp(-x)); }
        - inline double relu(double x) { return std::max(0.0, x); }
        - inline double tanh_act(double x) { return std::tanh(x); } // (std::tanh ya existe)
        - inline double step(double x) { return (x >= 0.0) ? 1.0 : 0.0; }
    - Crea la función "controladora" que Tarea 2 llamará:

C++

```
inline double activate(double x, ActivationType actType) {
    switch (actType) {
        case SIGMOID: return sigmoid(x);
        case RELU: return relu(x);
        case TANH: return tanh_act(x);
        case STEP: return step(x);
        default: return x; // O lanzar error
    }
}
```

3. **Día 3: Comparación y Entrega**

- Envía ActivationFunctions.hpp a **Tarea 1 (Documentador)** y **Tarea 2 (Arquitecto)**.
- **Para Tarea 1 (Documentador):** Escribe un párrafo breve comparando las funciones. (Ej: "Sigmoid y Tanh son no lineales y útiles para problemas de clasificación, pero Tanh está centrada en cero. ReLU es computacionalmente eficiente y evita el problema de 'vanishing gradient', siendo popular en capas ocultas."). Esto es el "comparadas significativamente".

👤 Tarea 4: El Ingeniero (Clases Layer y Network)

Objetivo Principal: Obtener los **15 Puntos** de **"Implementación Multi-capa"**.

Tu criterio de "Excelente" (100 pts):

- *"Propagación hacia adelante robusta a través de múltiples capas, con encapsulación y extensibilidad adecuadas."*

Plan de Acción Paso a Paso:

1. **Día 1: Archivos y Header (Layer.hpp)**
   - Necesitarás el Perceptron.hpp de Tarea 2.
   - Crea Layer.hpp y Layer.cpp.
   - En Layer.hpp, define la clase. Necesitarás:
     - #include "Perceptron.hpp"
     - **Miembros Privados:** std::vector<Perceptron> perceptrons; [9999] y ActivationType layerActType;.
     - **Miembros Públicos:**
     - Constructor: Layer(int numPerceptrons, int numInputsPerPerceptron, ActivationType actType); (Este constructor creará sus propios perceptrones).
     - Método de cómputo: std::vector<double> feedForward(const std::vector<double>& inputs);.
2. **Día 2: Implementación (Layer.cpp)**
   - Implementa el constructor: Haz un bucle numPerceptrons veces. En cada ciclo, crea los initialWeights (pueden ser aleatorios o todos 0.5) y un initialBias (ej. 0.1), crea un Perceptron y haz perceptrons.push_back(...).

- Implementa feedForward:
    1. Crea std::vector<double> layerOutputs;.
    2. Haz un bucle por cada Perceptron& p en perceptrons.
    3. Llama layerOutputs.push_back(p.feedForward(inputs, layerActType));.
    4. Retorna layerOutputs.

3. **Día 3: Nivel de Red (Network.hpp y Network.cpp)**
    - Crea Network.hpp y Network.cpp.
    - En Network.hpp:
        - #include "Layer.hpp"
        - **Miembros Privados:** std::vector<Layer> layers;[10101010].
        - **Miembros Públicos:** void addLayer(const Layer& layer); y std::vector<double> feedForward(std::vector<double> inputs);.
    - En Network.cpp:
        - Implementa addLayer (es solo un layers.push_back(layer);).
        - Implementa feedForward (esta es la propagación "robusta"):

C++

```
std::vector<double> currentOutputs = inputs;
for (Layer& layer : layers) {
   currentOutputs = layer.feedForward(currentOutputs);
}
return currentOutputs;
```

4. **Día 4: Comentarios y Entrega**
    - Comenta tus headers (.hpp) para la rúbrica.
    - Envía tus 4 archivos a **Tarea 1 (Documentador)** y **Tarea 5 (Validador)**.

👤 Tarea 5: El Validador (Pruebas e Integración)

Objetivo Principal: Obtener los **25 Puntos** de **"Validación y Pruebas"** (en la ejecución).

Tu criterio de "Excelente" (100 pts):

- *"Pruebas extensas y relevantes con resultados documentados e interpretación significativa."*

Plan de Acción Paso a Paso:

1. **Día 1: Recolección y Configuración**
    - Crea main.cpp.
    - Pide a Tarea 2, 3 y 4 sus archivos (.hpp y .cpp).
    - Configura tu compilador para enlazar todos los .cpp (o simplemente incluye todo en main.cpp si se complica).
2. **Día 2: Prueba Unitaria (Compuerta Lógica AND)**
    - **Prueba de Tarea 2/3:** Demuestra que un solo perceptrón funciona.
    - En main(), crea un perceptrón que simule una compuerta AND.
        - std::vector<double> weights = {1.0, 1.0};
        - double bias = -1.5;
        - Perceptron p_and(weights, bias);
    - Prueba todas las entradas usando la activación STEP:
        - std::cout << "AND(0,0): " << p_and.feedForward({0,0}, STEP) << std::endl; // Esperado: 0
        - std::cout << "AND(0,1): " << p_and.feedForward({0,1}, STEP) << std::endl; // Esperado: 0
        - std::cout << "AND(1,0): " << p_and.feedForward({1,0}, STEP) << std::endl; // Esperado: 0
        - std::cout << "AND(1,1): " << p_and.feedForward({1,1}, STEP) << std::endl; // Esperado: 1
    - Guarda esta salida de consola.
3. **Día 3: Prueba de Integración (Compuerta Lógica XOR)**
    - **Prueba de Tarea 4:** Demuestra que la red multi-capa funciona. XOR requiere una capa oculta.
    - Crea una Network: Network xor_net;
    - **Capa Oculta (con 2 neuronas, usando ReLU):**

        - Crea Layer hidden_layer(2, 2, RELU); // 2 neuronas, 2 entradas c/u
        - *Necesitarás* ajustar los pesos manualmente (¡ya que no hay training!) para que funcione.
        - hidden_layer.perceptrons[0].setWeights({-2, -2});
        - hidden_layer.perceptrons[0].setBias(3); // (Neurona NAND)
        - hidden_layer.perceptrons[1].setWeights({1, 1});
        - hidden_layer.perceptrons[1].setBias(-0.5); // (Neurona OR)
        - xor_net.addLayer(hidden_layer);
    - **Capa de Salida (con 1 neurona, usando SIGMOID):**
        - Crea Layer output_layer(1, 2, SIGMOID); // 1 neurona, 2 entradas (de la capa oculta)
        - output_layer.perceptrons[0].setWeights({1, 1});
        - output_layer.perceptrons[0].setBias(-1.5); // (Neurona AND)
        - xor_net.addLayer(output_layer);
4. **Día 4: Ejecución y Entrega**
    - Ejecuta las pruebas de la red XOR:
        - std::cout << "XOR(0,0): " << xor_net.feedForward({0,0})[0] << std::endl; // Esperado: ~0
        - std::cout << "XOR(0,1): " << xor_net.feedForward({0,1})[0] << std::endl; // Esperado: ~1
        - std::cout << "XOR(1,0): " << xor_net.feedForward({1,0})[0] << std::endl; // Esperado: ~1
        - std::cout << "XOR(1,1): " << xor_net.feedForward({1,1})[0] << std::endl; // Esperado: ~0
    - Envía tu main.cpp completo y *toda la salida de consola* (de la prueba AND y XOR) a **Tarea 1 (Documentador)**. Tu trabajo es la prueba viviente de que todo funciona.