Universidad de Alcalá de Henares

Escuela Politécnica Superior

Sistemas de Control Inteligente

PL1 - Identificación y control neuronal

Autores:

Elizabeth Adwoa Oppong Ansah

Adrián Retamosa Gómez

Fecha:

2 de noviembre de 2025

Puesto:

2

Índice

1. Preparación, desarrollo y entrenamiento de la red neuronal

1.1. Introducción y Fundamentos Teóricos

La Práctica 1 se centra en la **identificación y control neuronal** mediante el diseño, la implementación y la evaluación de Redes Neuronales Multicapa (MLP) para dos tareas fundamentales: la **aproximación de funciones (regresión)** y la **clasificación de patrones**.

Los objetivos de esta práctica son, por un lado, implementar una red neuronal desde cero en MATLAB, basándose en los principios de la propagación hacia adelante (forward propagation) y la retropropagación (backpropagation); y, por otro, utilizar las funciones de alto nivel de la Deep Learning Toolbox (como `fitnet` y `patternnet`) para comparar la eficiencia y el rendimiento entre ambos enfoques.

1.2. Arquitectura y Ecuaciones Base de la Red Neuronal

El modelo implementado en toda la práctica es una Red Neuronal Multicapa (Multi-Layer Perceptron, MLP) con una **única capa oculta** de H neuronas y una capa de salida de C neuronas. Además, la capa oculta tiene una función de activación **tansig** y la de salida una función **lineal**. Se aplica el algoritmo de **Descenso por el Gradiente con Momento** como método de aprendizaje. La topología de la red (Capas de W_1 y b_1) se define por las siguientes ecuaciones de propagación hacia adelante:

1.2.1. Ecuaciones de Propagación (Válidas para Regresión y Clasificación)

Las siguientes ecuaciones de propagación hacia adelante se mantienen constantes para las implementaciones de regresión y clasificación:

- **Entrada Ponderada Capa Oculta:**

$$Z_1 = W_1 X + b_1$$

- **Salida Capa Oculta (Activación `tansig`):**

$$A_1 = \text{tansig}(Z_1)$$

- **Entrada Ponderada Capa de Salida:**

$$Z_2 = W_2 A_1 + b_2$$

Donde X es la matriz de entradas, W_i son las matrices de pesos y b_i son los vectores de bias.

1.2.2. Ecuaciones Específicas de cada red neuronal

Como ya se ha mencionado, las ecuaciones de actualización de pesos se basan en el Descenso por Gradiente con Momento, y la fórmula genérica es:

$$W = W - lr \left(\frac{\partial L}{\partial W} + \lambda W \right)$$

Donde lr es la tasa de aprendizaje (learning rate) y λ es el término de regularización (momentum de inercia).

Componente	Regresión (Ejercicio 1)	Clasificación (Ejercicio 3)
Función de Salida	**Lineal** ($Y = Z_2$) $Y = W_2 A_1 + b_2$	**Softmax** ($\hat{Y} = \text{softmax}(Z_2)$) $\hat{Y} = \frac{e^{Z'_2}}{\sum_{j=1}^{C} e^{Z'_{2_j}}}$ con $Z_2 = W_2 A_1 + b_2$
Función de Pérdida (L) **Fórmula de L**	**Error Cuadrático Medio (MSE)** $L = \frac{1}{N}\sum_{i=1}^{N}(Y_i - T_i)^2$	**Entropía Cruzada (Cross-Entropy)** $L = -\frac{1}{N}\sum_{i=1}^{N} Y_i \log(\hat{Y}_i + \text{eps})$
Gradiente Base	**Error de Salida (dY)** $dY = \frac{2}{N}(Y - T)$	**Error de Logits (dZ_2)** $dZ_2 = \frac{1}{N}(\hat{Y} - Y)$
Gradiente de Pesos $\mathbf{W_2}$	$dW_2 = dY A_1^T + \lambda W_2$	$dW_2 = dZ_2 A_1^T + \lambda W_2$

Cuadro 1: Diferencias en las ecuaciones clave de salida, pérdida y gradientes base para la implementación MLP from scratch.

La diferencia crucial entre las tareas de regresión y clasificación reside en la función de activación final y la función de pérdida a minimizar, tal y como se presenta en el Cuadro 1.

Una vez definidas las ecuaciones de la red se procede con la implementación en MATLAB para la regresión.

1.3. Ejercicio 1: Aproximación de Funciones (II) - MLP from Scratch para Regresión

Este ejercicio se centra en la implementación completa de una Red Neuronal Multicapa (MLP) para la **regresión**, utilizando exclusivamente álgebra matricial fundamental en MATLAB, sin recurrir a las funciones de entrenamiento de alto nivel de la Toolbox. El objetivo es aproximar la función definida en el `simplefit_dataset` (dataset proporcionada por MATLAB).

1.3.1. Cargar los datos, definir los hiperparámetros y entrenar la red

El primer paso consiste en la carga del conjunto de datos y la definición de los hiperparámetros del modelo. El conjunto de datos se particiona en tres subconjuntos:

- **70 % para entrenamiento**, ajuste los pesos y sesgos de la red mediante el algoritmo de optimización (Descenso por el Gradiente con Momento).
- **15 % para validación** y monitorizar el rendimiento del modelo durante el entrenamiento en datos que no ha visto (se usa la técnica de parada temprana, o early stopping, si el error de validación comienza a aumentar mientras el error de entrenamiento sigue disminuyendo).
- **15 % para test** que se utiliza una sola vez al final del proceso, y proporciona una medida final, imparcial y no sesgada de la capacidad de generalización del modelo.

Los hiperparámetros definidos son los siguientes:

- **Neuronas Ocultas (H): 10.** Define la capacidad del modelo para aprender patrones complejos; un valor mayor aumenta la complejidad computacional (hay más mínimos locales y parámetros).
- **Épocas Máximas: 1000.** Número máximo de ciclos de entrenamiento sobre todo el conjunto de datos.
- **Tasa de Aprendizaje (lr):** $1e-2$**.** Controla el tamaño del paso dado en la dirección del gradiente durante la actualización de pesos.
- **Regularización** (λ)**:** 0,01 **(Penalización L2).** Coeficiente que penaliza los pesos grandes para evitar el sobreajuste (overfitting) del modelo a los datos de entrenamiento.

El entrenamiento se realiza mediante un bucle que ejecuta el ciclo *Forward* → *Loss* → *Backward* → *Actualizacion de Parametros* (Descenso por el Gradiente con Momento).

```
for e = 1:epochs
    % FORWARD
    [Ytr, cacheTr] = forward_regression(Xtr, P, actName);
    % LOSS
    mseTr(e) = mse_loss(Ytr, Ttr);

    % BACKWARD
    G = backward_regression(Xtr, Ttr, Ytr, cacheTr, P, actName, lambda);

    % ACTUALIZACION DE PESOS (Descenso por Gradiente con Momento)
    P.W1 = P.W1 - lr * G.dW1;
    P.b1 = P.b1 - lr * G.db1;
    P.W2 = P.W2 - lr * G.dW2;
    P.b2 = P.b2 - lr * G.db2;
end
```

Listing 1: Fragmento del bucle de entrenamiento

1.3.2. Implementar la función de inicialización de parámetros de la red

La función `init_params` inicializa las matrices de pesos (W) con valores aleatorios pequeños (distribución Gaussiana escalada) y los vectores de bias (b) a cero.

```
function P = init_params(D, H, C, scale)
    % W1: HxD, b1: Hx1 ; W2: CxH, b2: Cx1
    if nargin < 4, scale = 1e-2; end
    P.W1 = randn(H, D) * scale;
    P.b1 = zeros(H, 1);
    P.W2 = randn(C, H) * scale;
    P.b2 = zeros(C, 1);
end
```

Listing 2: Función init_params

1.3.3. Desarrollar el algoritmo de propagación hacia adelante (forward propagation)

La función `forward_regression` calcula la salida de la red (Y), aplicando la función de activación `tansig` en la capa oculta y una función **lineal** en la capa de salida.

- La **combinación de la capa oculta** es $tansig(Z_1) = tansig(W_1 X + b_1))$.
- La **salida final** es $purelin(Y) = purelin(W_2 A_1 + b_2)$.

```
function [Y, cache] = forward_regression(X, P, actName)
    % X: D x N  -> Z1=W1*X+b1; A1=act(Z1); Y=W2*A1 + b2 (lineal)
    Z1 = P.W1*X+P.b1; % Combinacion lineal

    switch lower(actName)
        case 'tansig', A1 = tansig(Z1);
        % ... otros casos ...
        case 'purelin',A1 = Z1;
    end

    Y = P.W2*A1+P.b2; % Salida final (lineal)
    cache = struct('Z1',Z1,'A1',A1);
end
```

Listing 3: Función forward_regression

1.3.4. Implementar el algoritmo de retropropagación (backpropagation)

La función `backward_regression` calcula los gradientes de la función de pérdida (MSE) con respecto a los pesos y bias de ambas capas, aplicando la regla de la cadena y el término de regularización λ.

- **Gradiente dY:** $dY = \frac{2}{N}(Y - T)$
- **Gradiente dZ_1:** Se calcula como $dZ_1 = dA_1(1 - A_1^2)$, donde $1 - A_1^2$ es la derivada de `tansig`.
- **Actualización de W_1 y W_2:** Incorpora el término de regularización L2 λW_i.

```
function G = backward_regression(X, T, Y, cache, P, actName, lambda)
    m  = size(X,2);
    A1 = cache.A1; Z1 = cache.Z1;

    dY  = (2/m) * (Y - T);

    % Gradientes Capa de Salida
    G.dW2 = dY * A1' + lambda*P.W2;
    G.db2 = sum(dY,2) + lambda*P.b2;

    % Retropropagacion
    dA1 = P.W2' * dY;
    dZ1 = dA1 .* (1 - A1.^2); % dZ1 para 'tansig'

    % Gradientes Capa Oculta
    G.dW1 = dZ1 * X' + lambda * P.W1;
    G.db1 = sum(dZ1,2) + lambda * P.b1;
end
```

Listing 4: Función backward_regression

1.3.5. Implementar la función de pérdida (MSE) para evaluar el error de la red

La función `mse_loss` calcula el **Error Cuadrático Medio** (MSE), que es el promedio del cuadrado del error sobre todas las muestras y salidas.

```
function m = mse_loss(Y, T)
    C = size(T,1);
    % MSE medio por muestra y salida
    m = mean(sum((Y-T).^2, 1)/C);
end
```

Listing 5: Función mse_loss

1.4. Comparación de modificaciones respecto a la arquitectura base

A continuación, se presenta la evaluación del rendimiento del modelo MLP From-Scratch ante diversas modificaciones de hiperparámetros y funciones de activación, en comparación con la **Arquitectura Base** (H=10, lr=1e-2, λ=0.01, tansig).

El rendimiento se mide por el MSE (Error Cuadrático Medio) alcanzado en el conjunto de Validación y la Época de convergencia.

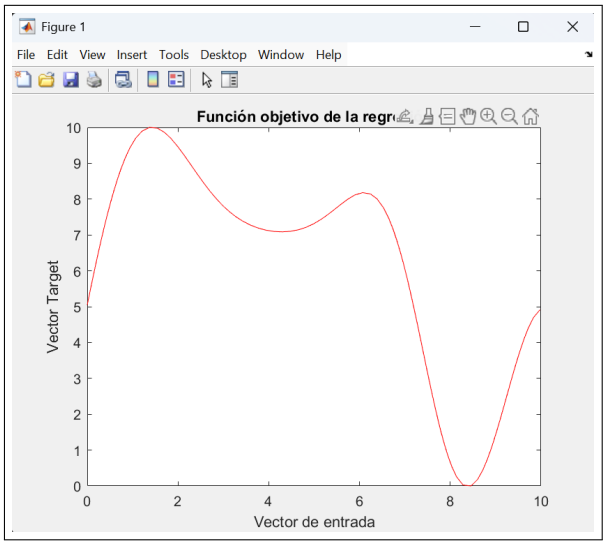


Figura 1: Función objetivo de la regresión (Target).

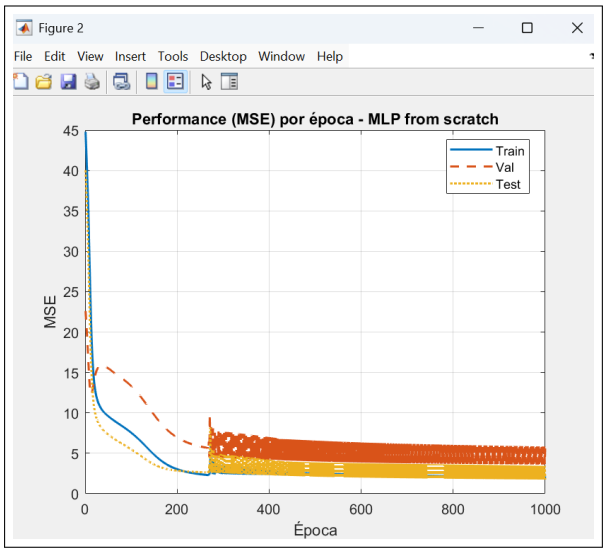


Figura 2: Performance (MSE) de la Arquitectura Base.

Análisis de la Performance de la Arquitectura Base:

- **Convergencia Rápida Inicial:** El error de entrenamiento (Training) y el error de validación (Validation) descienden de manera rápida y constante en las primeras ≈ 100 épocas. Esto indica que el lr $= 1e-2$ es apropiado para esta arquitectura y permite un rápido descenso hacia la zona de mínimo de la función de pérdida.
- **Estabilidad y Suavidad:** La curva muestra un progreso relativamente suave, lo que es característico del uso del **momento** en el Descenso por Gradiente. El momento ayuda a suavizar las oscilaciones y acelera la convergencia en zonas de pendiente baja.
- **Detección de Sobreajuste (Overfitting):** Alrededor de la época ≈ 300, se observa que la curva de error de Validación deja de descender o incluso puede empezar a ascender ligeramente, mientras

que el error de Entrenamiento continúa disminuyendo. Esto es el indicador clásico de **sobreajuste**: la red está empezando a memorizar el ruido específico del conjunto de entrenamiento en lugar de generalizar los patrones subyacentes.

Esta configuración logra una buena capacidad de modelado para la función objetivo sin caer en una inestabilidad excesiva, lo que la convierte en una base sólida para la comparación.

1.4.1. Tabla de Comparativa de Rendimiento

La siguiente tabla resume los resultados clave de las modificaciones de la arquitectura, comparando cada cambio con la Arquitectura Base (H=10, lr=$1e-2$, λ=0.01, tansig).

Modificación	lr	λ	MSE Final (Val)	Convergencia (Época)
Base (H = 10, tansig)	$1e-2$	0,01	**2,433485**	≈ 300
H=4 (Capacidad Reducida)	$1e-2$	0,01	**3,517787**	≈ 150
H=20 (Mayor Capacidad)	$1e-2$	0,01	**2,522202**	≈ 100
lr=1.0 (Muy Alto)	1,0	0,01	**NaN**	1
lr=1e-3 (Muy Bajo)	$1e-3$	0,01	**2,494351**	≈ 200
$\lambda = 0{,}2$ (Mayor Reg.)	$1e-2$	0,2	**2,855542**	≈ 80
$\lambda = 0{,}6$ (Gran Reg.)	$1e-2$	0,6	**3,509121**	≈ 100
logsig (Sigmoide)	$1e-2$	0,01	**2,507365**	≈ 400
ReLU	$1e-2$	0,01	**3,637743**	≈ 100

Cuadro 2: Resumen de rendimiento (MSE en Validación) de las modificaciones de hiperparámetros.

1.4.2. Análisis Detallado de las Modificaciones

Caso Base: La arquitectura inicial presenta una convergencia estable en las primeras épocas, deteniéndose en un MSE de Validación de **2,433485** en la época $\approx$ 300. Esta configuración demuestra un buen equilibrio entre la complejidad de la red y el learning rate. Pero no consigue ajustarse con precisión a la función original.

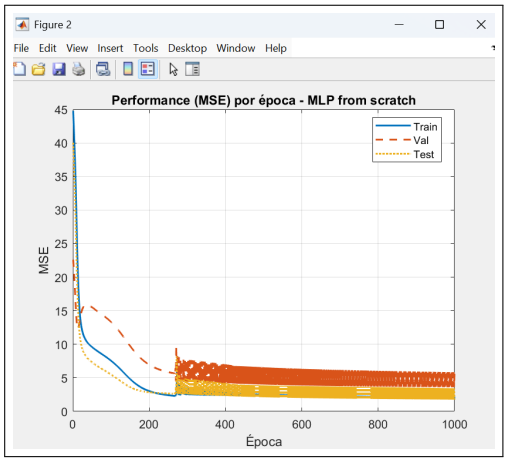


Figura 3: Performance MSE del Caso Base

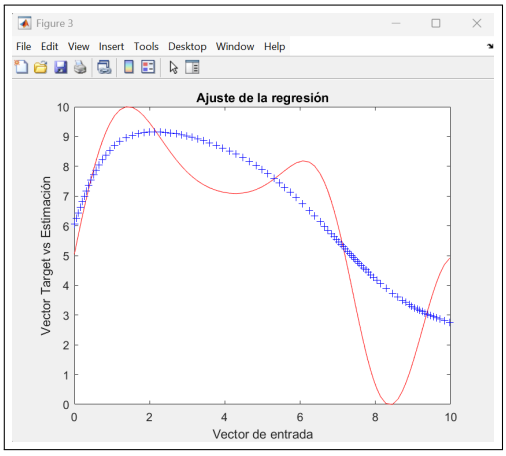


Figura 4: Ajuste de la Regresión del Caso Base

Caso 1: H=4 (Capacidad Reducida): Al reducir las neuronas a $H = 4$, el MSE de validación (**3,517787**) es **significativamente peor** que el de la Base (**2,433485**). Esto se debe a un **underfitting** (bajo ajuste), dado que la red carece de la capacidad necesaria para capturar las no linealidades de la función objetivo. La red converge rápidamente (≈ 150 épocas) a un error alto.

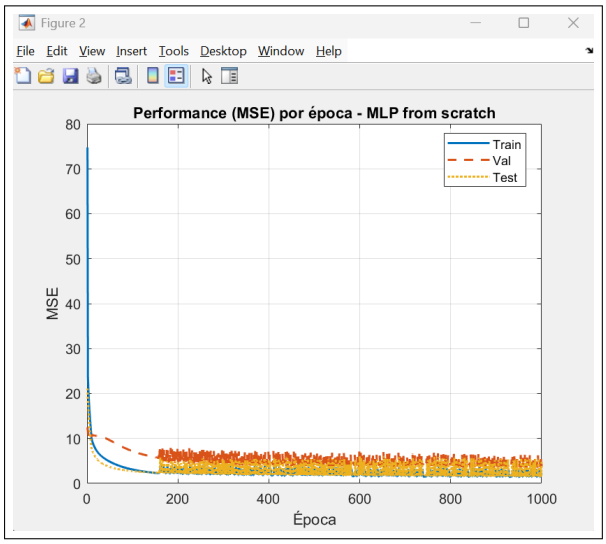


Figura 5: Performance MSE con $H = 4$.

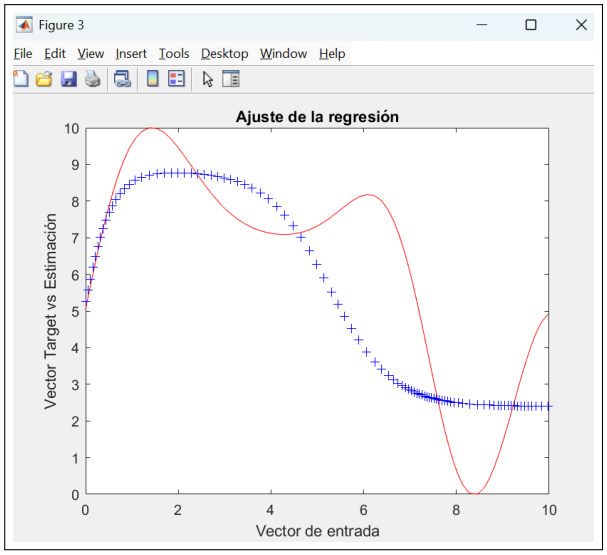


Figura 6: Ajuste de la Regresión con $H = 4$.

Caso 2: H=20 (Mayor Capacidad) El MSE de validación (**2,522202**) es **peor** que el de la Base (**2,433485**). Aunque una mayor capacidad puede ajustarse mejor a los datos de entrenamiento, el rendimiento en validación es inferior, sugiriendo que la red se detuvo antes de alcanzar el mínimo óptimo. La convergencia ocurre en la época ≈ 100.

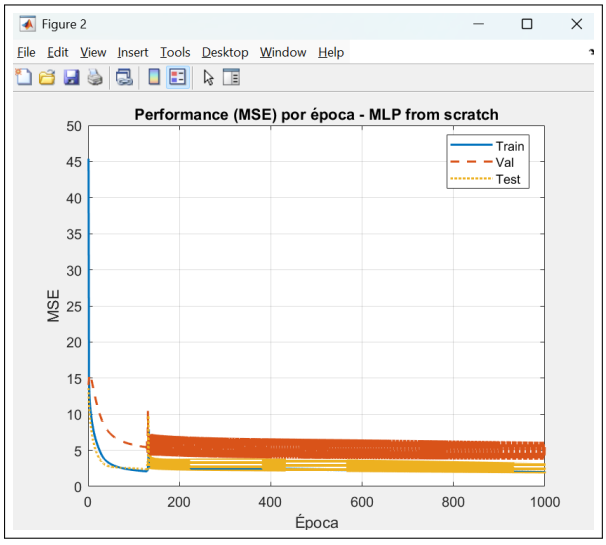


Figura 7: Performance MSE con $H = 20$.

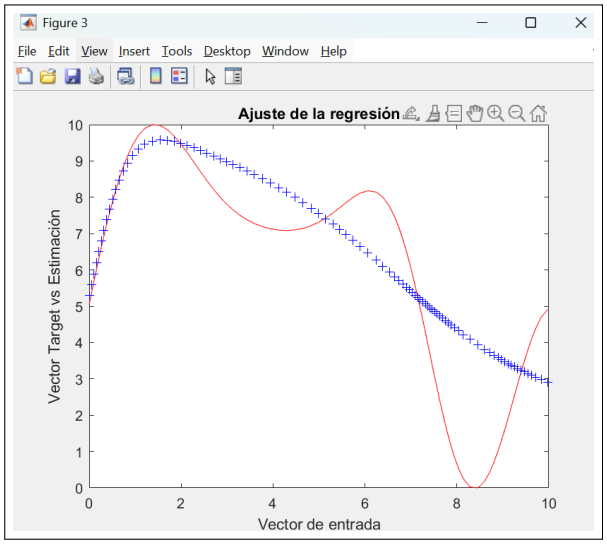


Figura 8: Ajuste de la Regresión con $H = 20$.

Caso 3: lr=1.0 (Tasa de Aprendizaje Muy Alta) Con lr = 1,0, el modelo **diverge** inmediatamente, resultando en **NaN** en la Época 1. Una tasa de aprendizaje tan grande causa inestabilidad y el algoritmo sobrepase constantemente el mínimo de la función de pérdida.

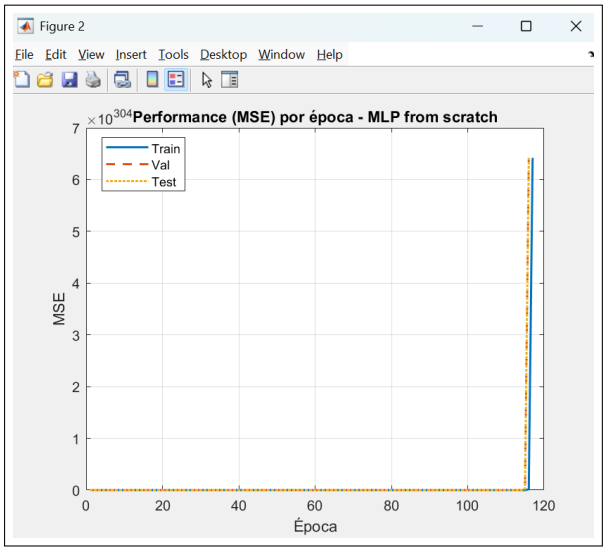


Figura 9: Performance MSE con $lr = 1{,}0$.

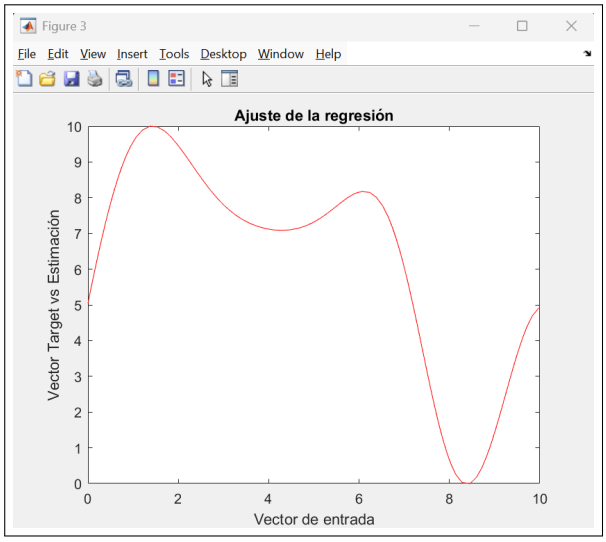


Figura 10: Ajuste de la Regresión con $lr = 1{,}0$.

Caso 4: lr=1e-3 (Tasa de Aprendizaje Muy Baja) El MSE final (**2,494351**) es **peor** que el de la Base. Un lr tan pequeño implica pasos muy cautelosos. Aunque la red aprende, no converge lo suficiente dentro de las 1000 épocas. Si se hubieran permitido más épocas, el rendimiento podría haber mejorado.

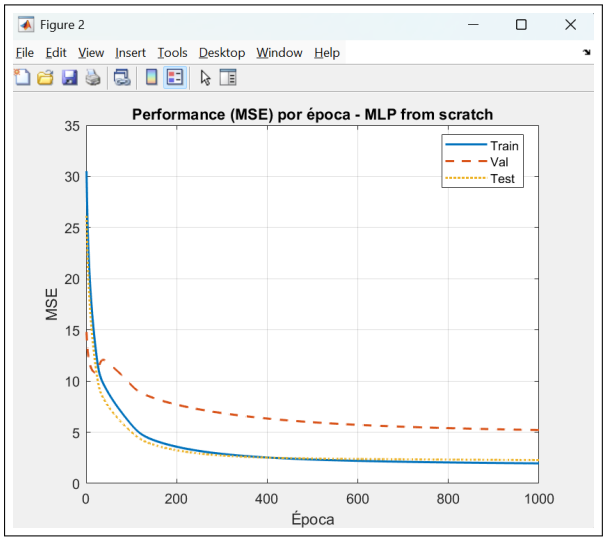


Figura 11: Performance MSE con $lr = 1e-3$.

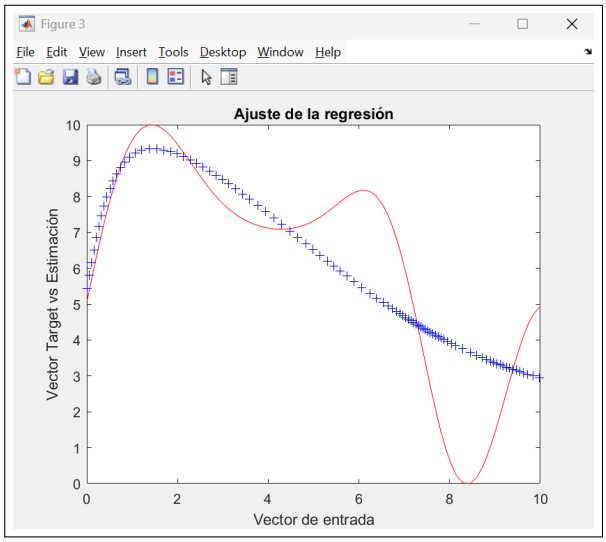


Figura 12: Ajuste de la Regresión con $lr = 1e-3$.

Caso 5: $\lambda = 0{,}2$ (Mayor Regularización) El MSE (**2,855542**) es **peor** que el de la Base. El aumento de λ causa **sobre-regularización**, forzando a la red a simplificar el modelo para evitar pesos grandes. Esto dificulta la captura de la no linealidad, resultando en un MSE más alto.

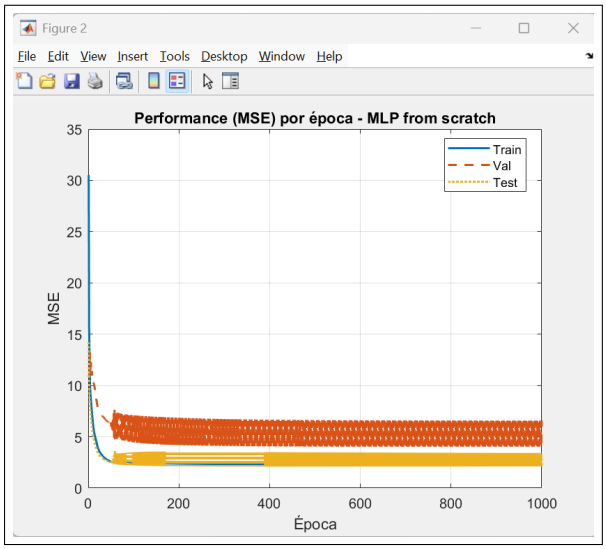


Figura 13: Performance MSE con $\lambda = 0{,}2$.

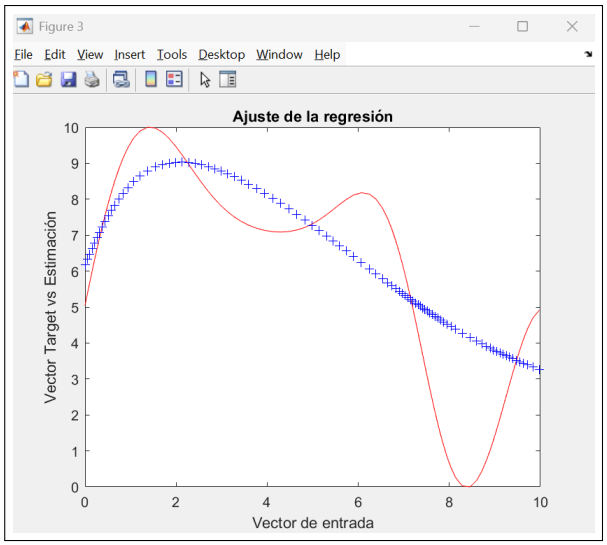


Figura 14: Ajuste de la Regresión con $\lambda = 0{,}2$.

Caso 6: $\lambda = 0{,}6$ (Gran Regularización) El MSE (**3,509121**) es **significativamente peor** que la Base. Una penalización tan alta simplifica drásticamente el modelo, resultando en un ajuste muy pobre, similar al caso de $H = 4$ (falta de capacidad).

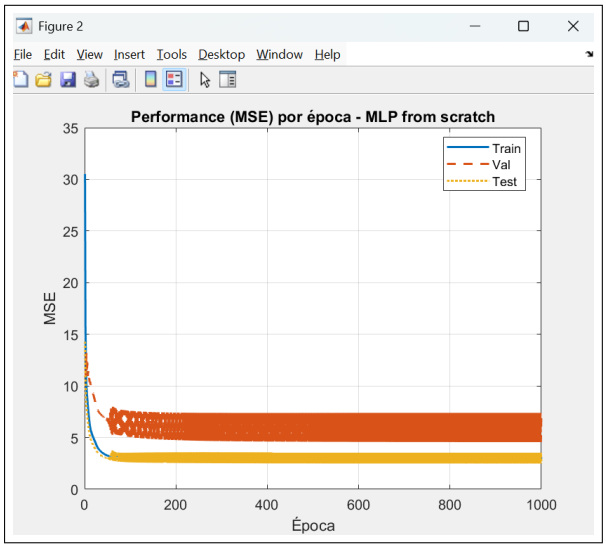


Figura 15: Performance MSE con $\lambda = 0{,}6$.

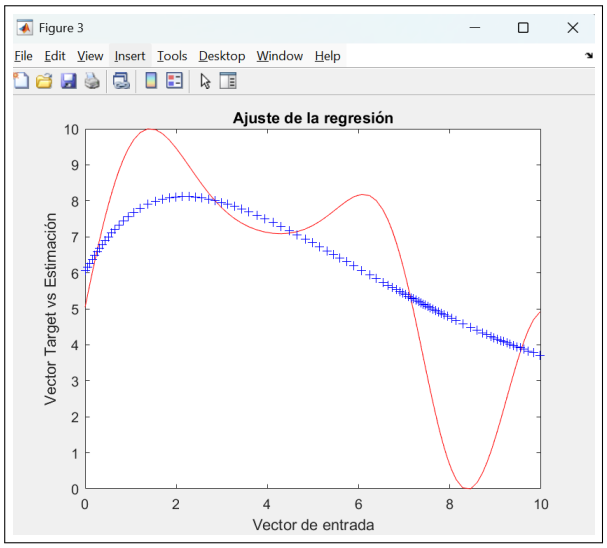


Figura 16: Ajuste de la Regresión con $\lambda = 0{,}6$.

Caso 7: Sigmoide (`logsig`) El MSE (**2,507365**) es **peor** que el de la Base. La función `logsig` es propensa al **Gradiente Evanescente**, lo que lleva a un aprendizaje inestable y a un rendimiento final inferior.

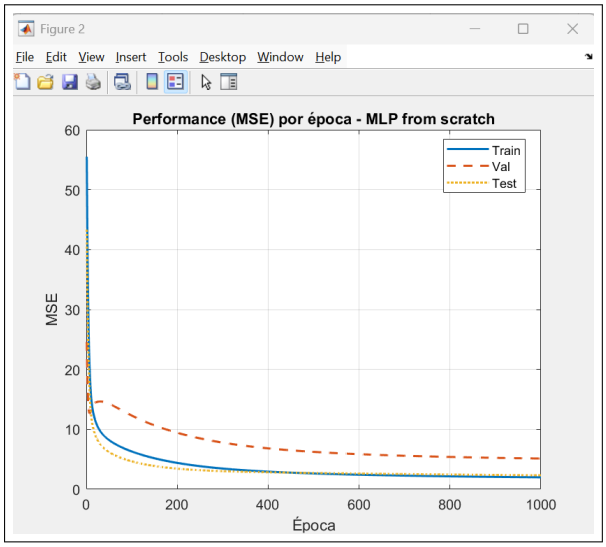


Figura 17: Performance MSE con `logsig`.

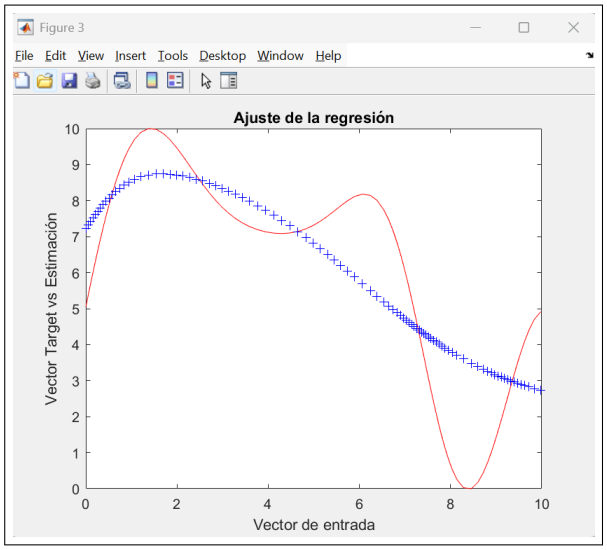


Figura 18: Ajuste de la Regresión con `logsig`.

Caso 8: ReLU (`relu`) El MSE (**3,637743**) es **peor** que el de la Base. La ReLU es menos adecuada para modelar funciones suaves y continuas de regresión, resultando en una aproximación por segmentos que no captura las curvaturas de la función objetivo.

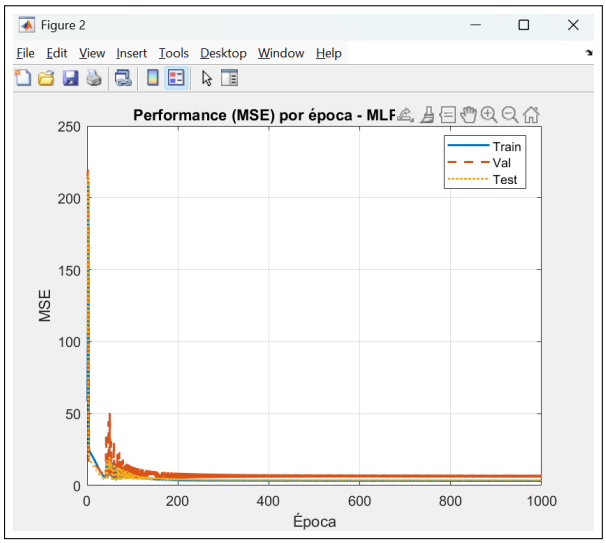


Figura 19: Performance MSE con `relu`.

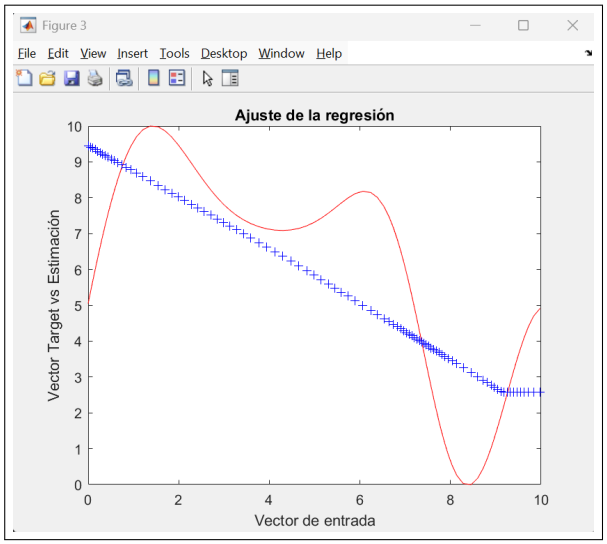


Figura 20: Ajuste de la Regresión con `relu`.

1.5. Ejercicio 2: Aproximación de Funciones (II) - Red de Alto Nivel (`fitnet`) Regresión

Este ejercicio aborda el mismo problema de regresión que el Ejercicio 1, pero empleando la función de alto nivel `fitnet`, que automatiza la gestión de la arquitectura, la división de datos y el bucle de entrenamiento. El objetivo es comparar la eficiencia de dos algoritmos de entrenamiento: `traingdm` (Descenso por Gradiente con Momento) y `trainlm` (Levenberg-Marquardt)

1.5.1. Configuración de la Red y Script Base

Se utiliza una red `fitnet` con $\mathbf{H = 10}$ neuronas en la capa oculta. Por defecto, `fitnet` incluye preprocesamiento (como la normalización) y utiliza la métrica **MSE** (Error Cuadrático Medio).

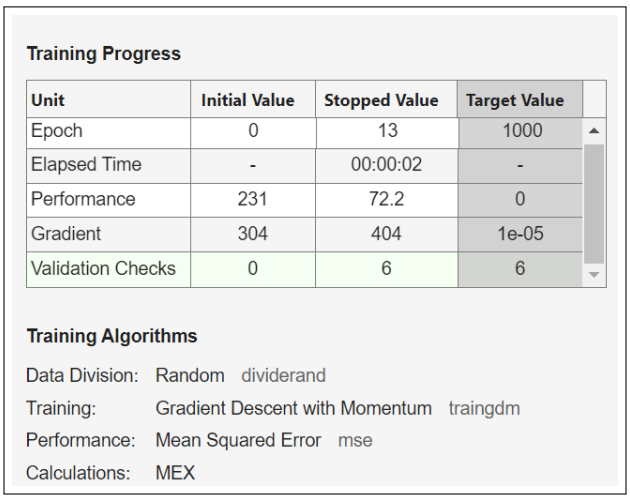
- **Arquitectura:** Red `fitnet` (`tansig` en oculta, lineal en salida).
- **División de Datos:** Estándar 70 %/15 %/15 % (Entrenamiento/Validación/Prueba).

```
%% Creacion de la red
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize);
% Division del conjunto de datos
net.divideFcn = 'dividerand';
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio   = 15/100;
net.divideParam.testRatio  = 15/100;
% Entrenamiento (algoritmo a elegir: traingdm o trainlm)
[net,tr] = train(net,inputs,targets);
% Inferencia y metricas
outputs = net(inputs);
performance = perform(net,targets,outputs) % MSE global
```

Listing 6: Script base utilizando fitnet en MATLAB

1.5.2. Comparativa de Algoritmos de Entrenamiento

Caso 1: Descenso por el Gradiente con Momento (`traingdm`): Se fuerza la red a utilizar el algoritmo traingdm `net.trainFcn = 'traingdm'` (por defecto en fitnet suele ser 'trainlm'). El entrenamiento se detuvo al alcanzar el criterio de validación (6 fallos consecutivos) en la Época 13. El rendimiento final fue MSE = 72,2.



Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	13	1000
Elapsed Time	-	00:00:02	-
Performance	231	72.2	0
Gradient	304	404	1e-05
Validation Checks	0	6	6

Training Algorithms

Data Division: Random dividerand
Training: Gradient Descent with Momentum traingdm
Performance: Mean Squared Error mse
Calculations: MEX

Figura 21: Resultados del entrenamiento con `traingdm`.

Análisis de las Gráficas (`traingdm`)

- `plotperform` **(Rendimiento MSE):** La Figura 22 muestra un MSE de validación inicial alto y una curva errática, característica de los algoritmos de primer orden. El mejor rendimiento se alcanzó en la Época 7 (MSE $\approx 35{,}75$).

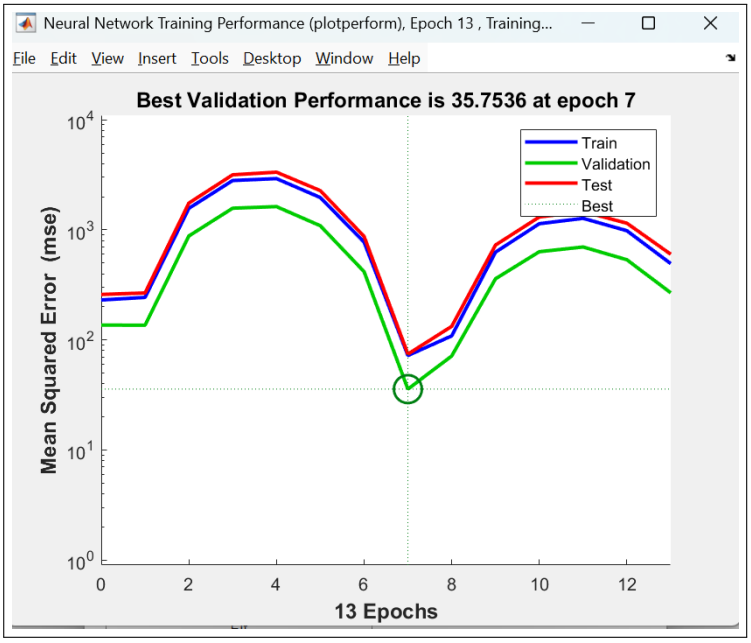


Figura 22: Performance (MSE) por época con `traingdm`.

- `plottrainstate` **(Estado de Entrenamiento):** La Figura 23 confirma la detención en la Época 13 por 6 fallos de validación. El Gradiente (≈ 404) es alto, lo que indica que el proceso se detuvo prematuramente, lejos de un mínimo óptimo.

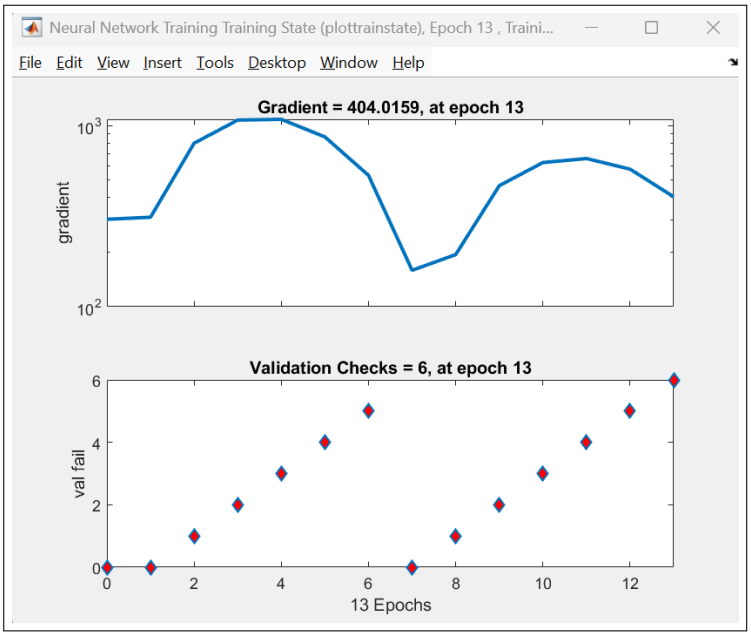


Figura 23: Gradiente y Validation Checks con `traingdm`.

- `ploterrhist` **(Histograma de Errores):** La Figura 24 muestra una **distribución ancha de errores** (Targets − Outputs), con una gran cantidad de errores en los extremos, confirmando el MSE elevado.

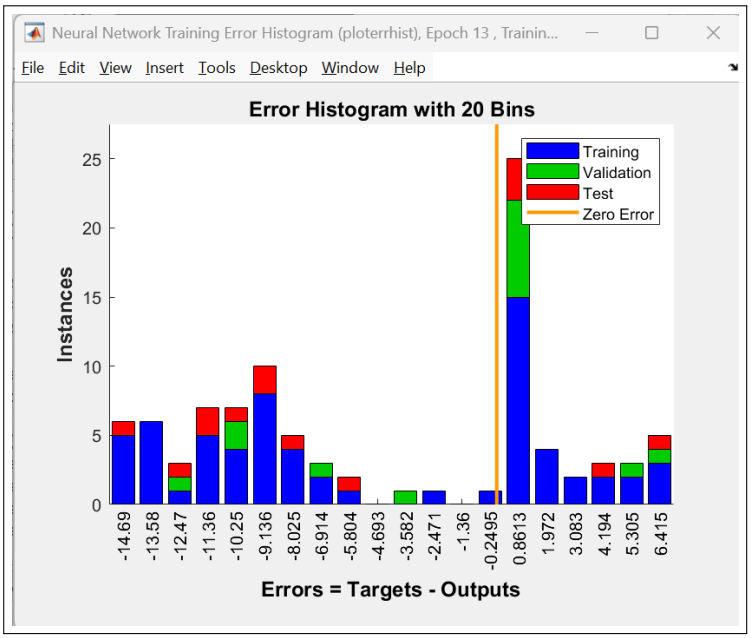


Figura 24: Histograma de errores.

- `plotregression` **y** `plotfit` **(Ajuste):** La correlación global es baja ($\mathbf{R \approx 0{,}89}$), y el ajuste de la función (Figura 26) muestra que el modelo **no logra capturar las oscilaciones** de la función objetivo.

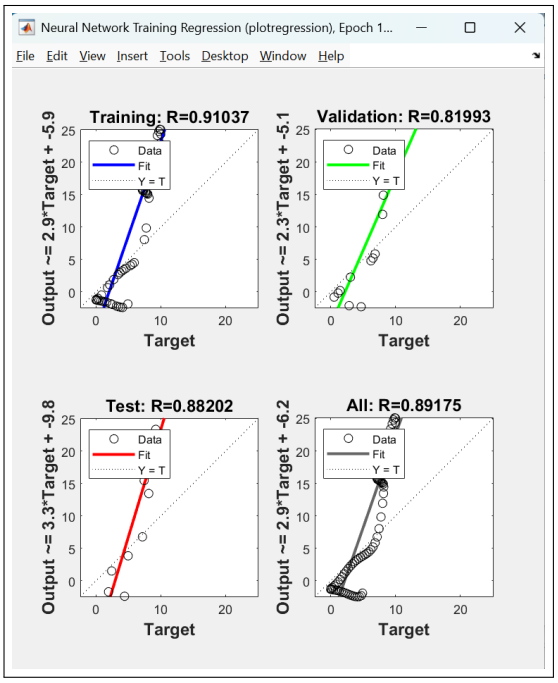


Figura 25: Gráfica de Regresión ($\mathbf{R = 0{,}89175}$) con `traingdm`.

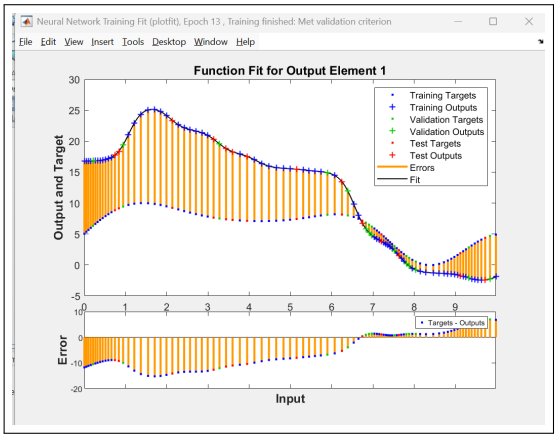


Figura 26: Ajuste de la Función con `traingdm`.

Caso 2: Levenberg-Marquardt (`trainlm`) Se configura la red para usar el algoritmo de segundo orden `net.trainFcn = 'trainlm'`. La detención por validación ocurre en la Época 375.

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	375	1000
Elapsed Time	-	00:00:21	-
Performance	231	6.69e-07	0
Gradient	304	4.77e-05	1e-07
Mu	0.001	1e-07	1e+10
Validation Checks	0	6	6

Training Algorithms

Data Division: Random dividerand
Training: Levenberg-Marquardt trainlm
Performance: Mean Squared Error mse
Calculations: MEX

Figura 27: Resultados del entrenamiento con `trainlm`.

Análisis de las Gráficas (`trainlm`)

- `plotperform` **(Rendimiento MSE):** El MSE final es de $\mathbf{6{,}69 \times 10^{-5}}$. El mejor rendimiento de validación ($\mathbf{5{,}6691 \times 10^{-5}}$) se alcanza en la Época 369. El descenso del error es rápido y constante, mostrando una gran eficiencia.

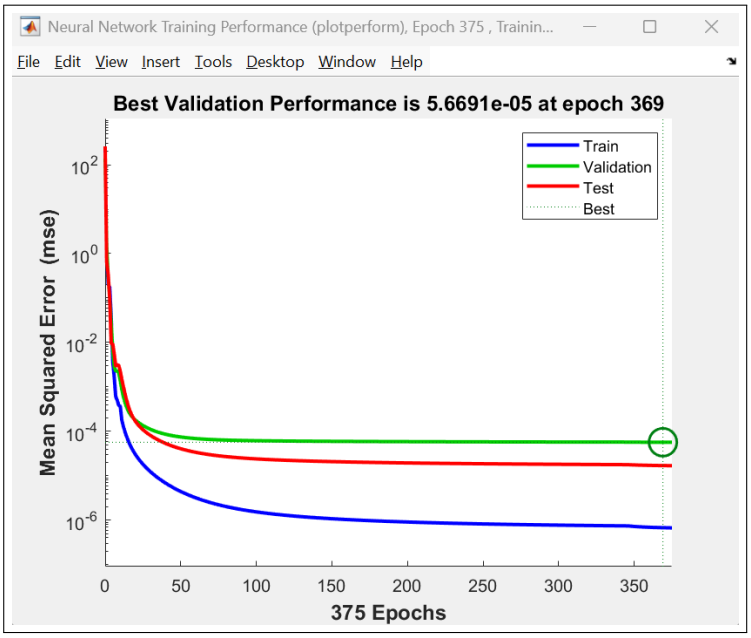


Figura 28: Performance (MSE) por época con `trainlm`.

- `plottrainstate` **(Estado de Entrenamiento):** La Figura 29 confirma la detención en la Época 375 tras 6 fallos, con un gradiente final extremadamente bajo ($\mathbf{4{,}77 \times 10^{-5}}$).

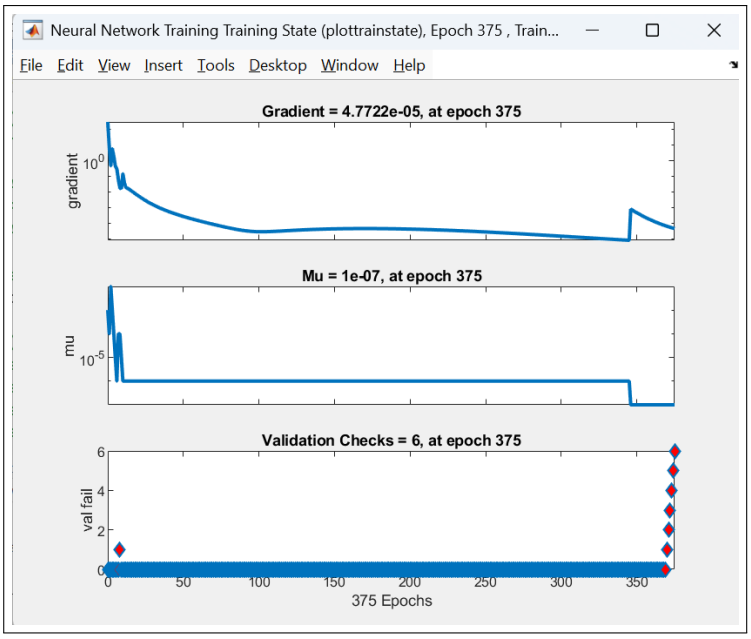


Figura 29: Estado de entrenamiento: Gradiente y Validation Checks con `trainlm`.

- `ploterrhist` **(Histograma de Errores):** La Figura 30 muestra que la inmensa mayoría de los errores se concentran en el centro (cercanos a cero), reflejando el ajuste de alta precisión.

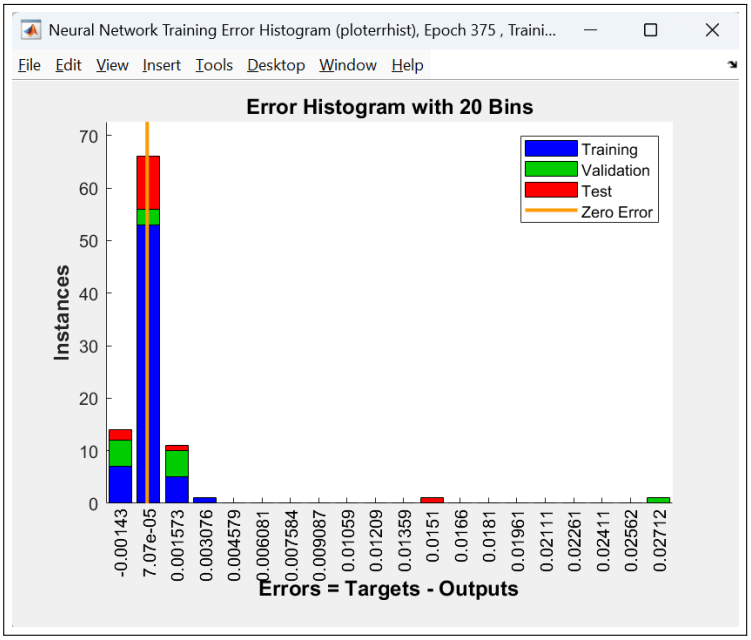


Figura 30: Histograma de errores con `trainlm`.

- `plotregression` y `plotfit` **(Ajuste):** La correlación es $\mathbf{R = 1}$ para todas las particiones. El ajuste de la función (Figura 32) es indistinguible del target.

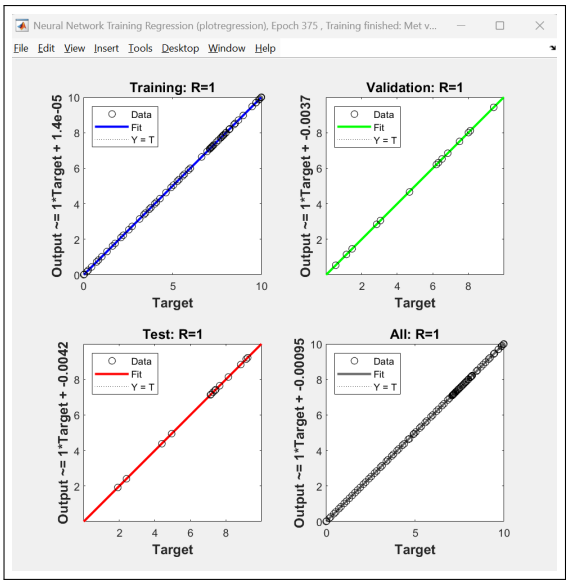


Figura 31: Gráfica de Regresión ($\mathbf{R = 1}$) con `trainlm`.

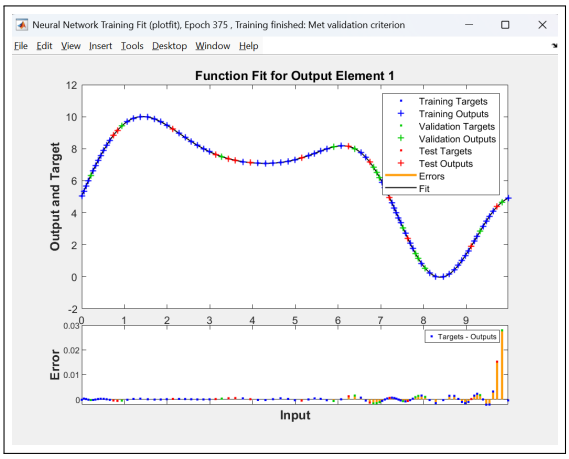


Figura 32: Ajuste de la Función con `trainlm`.

1.5.3. Conclusiones sobre la Aproximación de Funciones

A partir de la comparación entre la implementación desde cero (Ejercicio 1), `traingdm` y `trainlm`, se extraen las siguientes conclusiones:

1. **Eficiencia del Optimizador:** El algoritmo Levenberg-Marquardt (`trainlm`) es el más eficiente. Alcanza un MSE de $\mathbf{6{,}69 \times 10^{-5}}$ en 375 épocas, mientras que el `traingdm` se detiene prematuramente con un MSE de 72,2. La superioridad de `trainlm` se debe a su uso de información de segundo orden (curvatura de la pérdida) para dar pasos óptimos.

2. **Rendimiento vs. Implementación Propia:** Los resultados de `fitnet` con `trainlm` (MSE $\approx 10^{-5}$) son órdenes de magnitud superiores al MLP From-Scratch (MSE $\approx 10^{0}$), lo que destaca la robustez de la Toolbox de MATLAB, incluyendo la normalización de datos y la gestión del proceso de entrenamiento.

3. **Precisión y Generalización:** El MSE final de $\mathbf{6{,}69 \times 10^{-5}}$ y la correlación $\mathbf{R = 1}$ indican una **aproximación casi perfecta** de la función.

1.6. Ejercicio 3: Clasificación (I) - MLP from Scratch

Este ejercicio consiste en adaptar la implementación "desde cero" del Ejercicio 1 para resolver un problema de **clasificación binaria** utilizando el `cancer_dataset`. A diferencia de la regresión, el objetivo es asignar cada entrada a una de dos clases discretas (maligno o benigno). Para ello, se modifica la arquitectura de la red y el proceso de entrenamiento.

1.6.1. Cambios Clave Respecto a la Regresión

Las principales diferencias respecto al MLP de regresión son:

- **Capa de Salida:** Se sustituye la activación lineal por **Softmax**, adecuada para la clasificación multiclase. Esta función convierte los valores de salida de la red en un vector de probabilidades, donde cada elemento representa la probabilidad de pertenecer a una clase específica y la suma de todos los elementos es 1.
- **Función de Pérdida:** Se reemplaza el MSE por la **Entropía Cruzada** (Cross-Entropy). Esta es la función de pérdida estándar para problemas de clasificación, ya que mide la divergencia entre la distribución de probabilidad predicha y la distribución real.
- **Targets:** Se utiliza el `cancer_dataset`, donde los targets están en formato *one-hot*.
- **Métrica de Evaluación:** El rendimiento del modelo ya no se mide con el MSE, sino con el **Accuracy** (Precisión), que es el porcentaje de muestras clasificadas correctamente.

La implementación requiere modificar las funciones de *forward*, *backward* y *loss*:

- `forward_classification`**:** Implementa la salida Softmax.
$$\hat{Y} = \text{softmax}(Z_2)$$
- `crossentropy_loss`**:** Calcula la pérdida de entropía cruzada.
$$L = -\frac{1}{N}\sum_{i=1}^{N} Y_i \log(\hat{Y}_i + \text{eps})$$
- `backward_classification`**:** El gradiente clave en la salida se simplifica a:
$$dZ_2 = \frac{1}{N}(\hat{Y} - Y)$$
- `accuracy_cls`**:** Calcula el porcentaje de aciertos comparando el `argmax` de la salida y el target.

1.6.2. Implementación del Clasificador

Se ha adaptado el código base de regresión para incorporar las nuevas funciones de `forward_classif`, `cross_entropy_loss`, `backward_classif` y `accuracy_cls`. El gradiente inicial en la retropropagación (dZ_2) se simplifica a la diferencia entre la predicción y el valor real: $dZ_2 = (\hat{Y} - Y)/N$. El bucle de entrenamiento sigue la misma estructura, pero ahora optimiza la entropía cruzada.

```
function y_hat = softmax(z)
    % Se resta el maximo para estabilidad numerica y evitar overflow
    z_stable = z - max(z, [], 1);
    y_hat = exp(z_stable) ./ sum(exp(z_stable), 1);
end
```

Listing 7: Función de activación Softmax con estabilización numérica.

```
function acc = accuracy_cls(Y_hat, Y)
    % Accuracy: argmax(prob) vs argmax(target one-hot)
    [~, pred] = max(Y_hat, [], 1);
    [~, true_] = max(Y, [], 1);
    acc = mean(pred == true_);
end
```

Listing 8: Cálculo del Accuracy.

Se entrenó el modelo con los mismos hiperparámetros base de la regresión (H=10, lr=1e-2, λ=0.01). Los resultados muestran que el modelo aprende a clasificar los datos con una alta precisión. Como se observa en la Figura 33, el modelo presenta un estancamiento inicial durante aproximadamente 920 épocas, donde la precisión no mejora. Sin embargo, en el tramo final del entrenamiento, encuentra una dirección de mejora y la precisión aumenta de forma abrupta. Este comportamiento sugiere que la tasa de aprendizaje podría ser subóptima, provocando una convergencia muy lenta. El accuracy final en el conjunto de test fue de **76.4 %**.

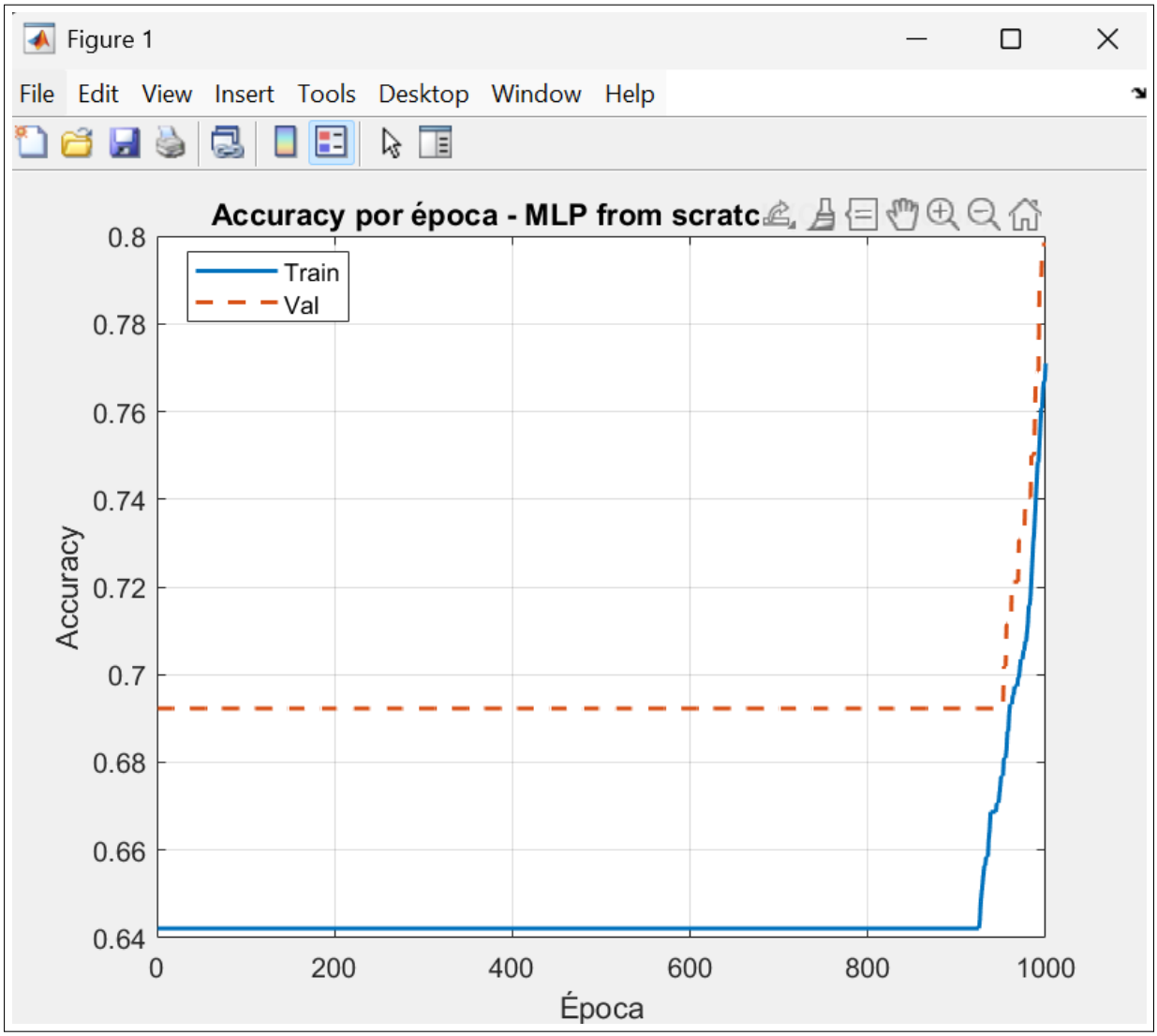


Figura 33: Curvas de Pérdida (Cross-Entropy) y Accuracy para el clasificador manual.

La matriz de confusión (Figura 34) revela que el rendimiento del modelo no es uniforme entre clases. Es capaz de identificar correctamente todas las muestras de la **Clase 2**, pero a costa de clasificar incorrectamente 25 muestras de la **Clase 1** como si fueran de la Clase 2 (falsos negativos). A pesar de este desequilibrio, el modelo demuestra una capacidad de generalización razonable.

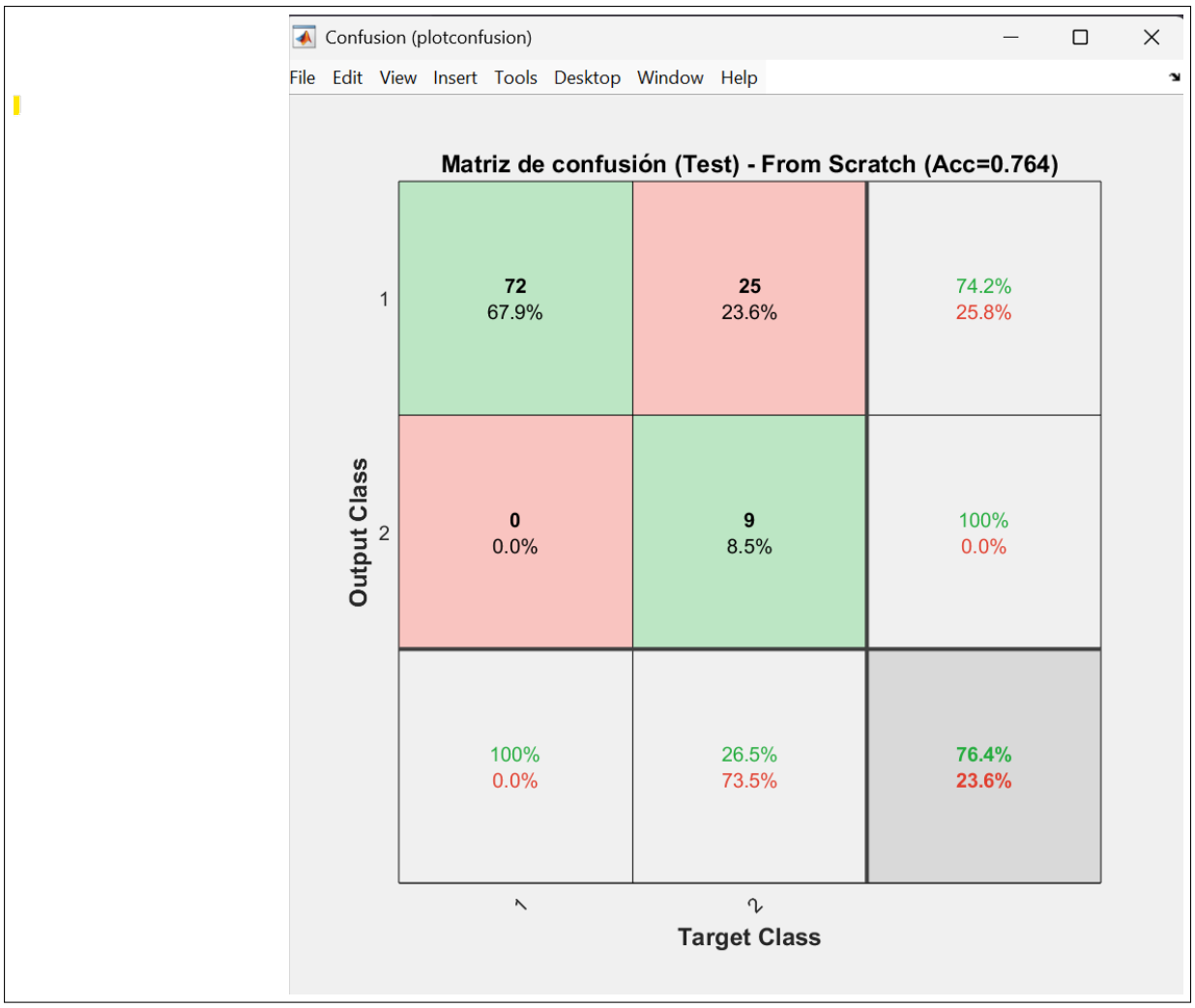


Figura 34: Matriz de confusión para el conjunto de test del clasificador manual.

1.7. Ejercicio 4: Clasificación (II) - Red de Alto Nivel (patternnet)

En este ejercicio se utiliza la función `patternnet` de la Deep Learning Toolbox, diseñada específicamente para tareas de clasificación. Esta función automatiza la creación de un MLP con una capa de salida con activación Softmax y la función de pérdida de Entropía Cruzada.

1.7.1. Configuración de la Red y Comparativa de Algoritmos

Se configura una red con 10 neuronas ocultas y se utiliza el `cancer_dataset`. Al igual que en el ejercicio de regresión, se compara el rendimiento de los algoritmos de entrenamiento **traingdm** (Descenso por Gradiente con Momento) y **trainlm** (Levenberg-Marquardt).

```
hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize);

% Algoritmo a elegir: 'traingdm' o 'trainlm'
net.trainFcn = 'traingdm';

% Plots integrados para analisis
net.plotFcns = {'plotperform','plottrainstate','plotconfusion','plotroc'
    };

[net, tr] = train(net, inputs, targets);
```

Listing 9: Script base para clasificación con `patternnet`.

1.7.2. Caso 1: Descenso por Gradiente con Momento (traingdm)

El entrenamiento con `traingdm` se configuró para un máximo de 1000 épocas. Los resultados muestran un rendimiento muy superior al de la implementación manual. La curva de rendimiento (Figura 35) muestra que el error de validación seguía descendiendo en la época 1000, lo que indica que el entrenamiento se detuvo por alcanzar el límite de épocas y no por el criterio de *early stopping*. A pesar de ello, la pérdida final es baja (0.0539). El rendimiento final en el conjunto de test alcanza un **Accuracy de 91.4 %**.

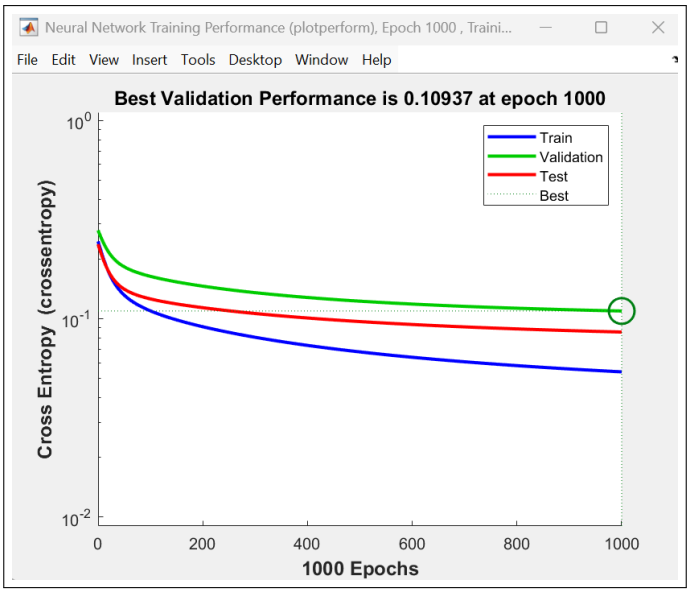


Figura 35: Performance (Cross-Entropy) con `traingdm`.

La matriz de confusión (Figura 36) confirma la alta precisión, con solo 9 errores de clasificación en el conjunto de test. La curva ROC (Figura 37) está muy próxima a la esquina superior izquierda, lo que es indicativo de un clasificador excelente.

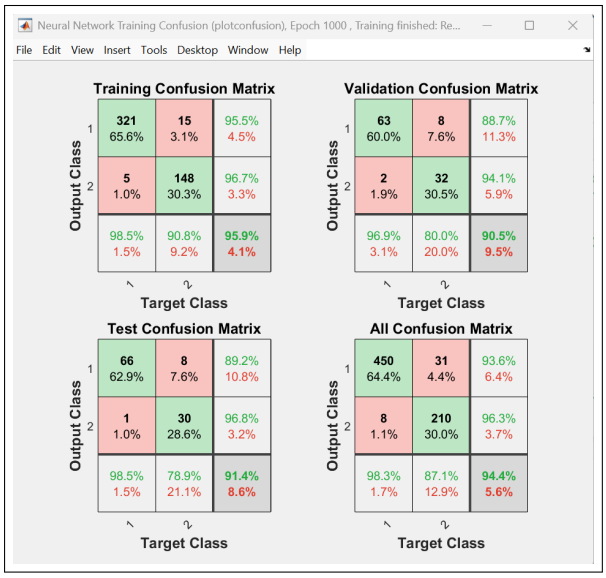


Figura 36: Matriz de Confusión (Test) con `traingdm`.

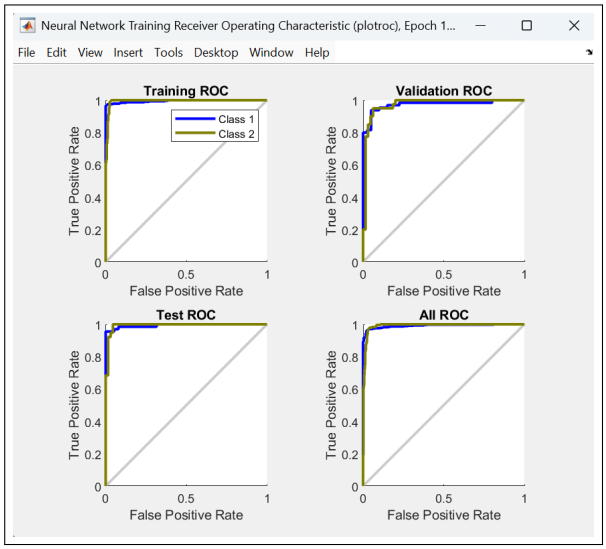


Figura 37: Curva ROC (Test) con `traingdm`.

1.7.3. Caso 2: Levenberg-Marquardt (trainlm)

Al cambiar el algoritmo a `trainlm`, se observa una gran mejora en la eficiencia y en la precisión final del modelo.

El algoritmo `trainlm` demuestra una velocidad de convergencia extraordinaria, deteniéndose en tan solo **8 épocas** gracias al mecanismo de *early stopping*. El mejor rendimiento de validación se alcanzó en la época 2 (Figura 38), que evita el sobreajuste y garantiza una gran capacidad de generalización. El resultado es un clasificador casi perfecto, con un **Accuracy en test del 98.1 %**.

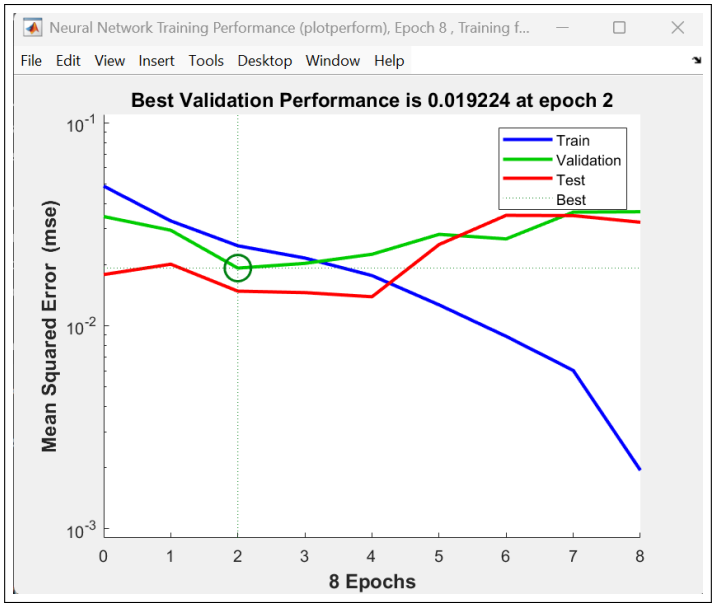


Figura 38: Performance con `trainlm`.

La matriz de confusión (Figura 39) muestra únicamente 2 errores en todo el conjunto de test. La curva ROC (Figura 40) es prácticamente ideal, confirmando la superioridad de este modelo.

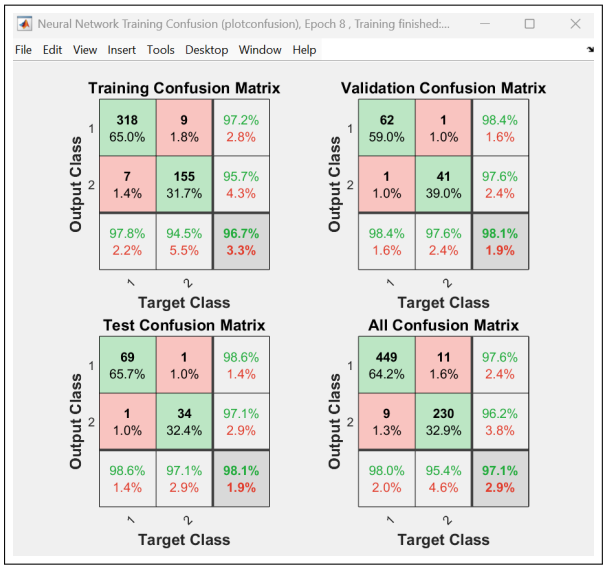


Figura 39: Matriz de Confusión (Test) con `trainlm`.

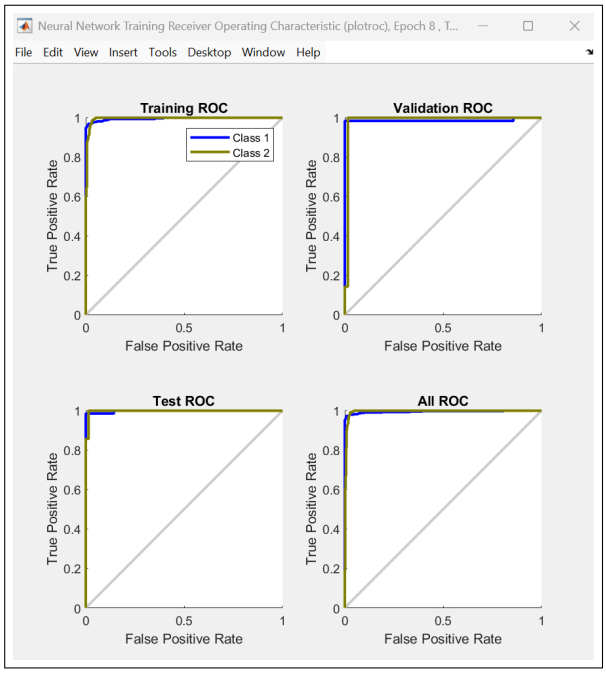


Figura 40: Curva ROC (Test) con `trainlm`.

1.7.4. Conclusiones sobre la Clasificación

1. **Eficiencia del Optimizador:** El algoritmo Levenberg-Marquardt (`trainlm`) es superior a `traingdm` para este problema. Logra un accuracy mayor (**98.1 %** vs. **91.4 %**) de forma muy rápida (converge en 8 épocas frente a las 1000 de `traingdm`). Esto se debe a su uso de información de segundo orden (curvatura de la pérdida) para dar pasos de optimización mucho más efectivos.

2. **Rendimiento Implementación Manual vs. Toolbox:** El clasificador de la toolbox con `trainlm` (**98.1 %** de accuracy) es también superior al implementado desde cero (from scratch) (**76.4 %**). Esta diferencia resalta cómo las optimizaciones internas de MATLAB, como la normalización automática de los datos de entrada y el uso de algoritmos de entrenamiento avanzados, mejoran significativamente el rendimiento final.

3. **Importancia de las Métricas:** Esta práctica demuestra la necesidad de utilizar métricas adecuadas para cada tarea. Mientras el MSE fue clave en la regresión, el **Accuracy**, la **Matriz de Confusión** y las **Curvas ROC** son herramientas indispensables para comprender el rendimiento de un clasificador, su comportamiento por clase y el equilibrio que exista entre sensibilidad y especificidad.

2. Parte 2: Emulación de Controlador de Posición con Red No Recursiva

En esta segunda parte de la práctica, el objetivo es diseñar un controlador neuronal capaz de **emular el comportamiento de un controlador de posición de tipo "caja negra"** proporcionado. Se utilizará el entorno de Simulink para modelar y simular un robot móvil, y MATLAB para generar los datos de entrenamiento y construir la red neuronal.

2.1. Descripción del Sistema

El sistema consiste en el control de posición de un robot móvil en un entorno de 10x10 metros. El objetivo es que una red neuronal aprenda a replicar las salidas (velocidad lineal V y angular W) del controlador 'controlblackbox.slx' a partir de sus entradas (error de distancia E_d y error de ángulo E_θ).

La visión general del sistema en Simulink es la siguiente:

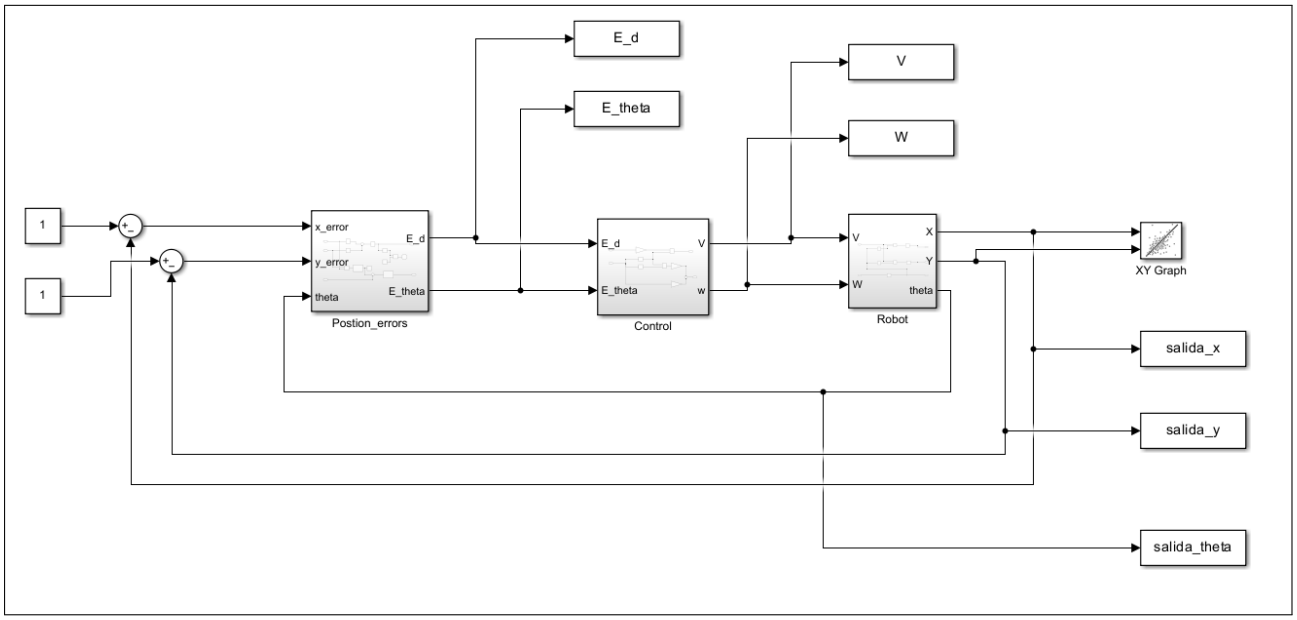


Figura 41: Esquema general del sistema de control de posición en Simulink.

El sistema se compone de tres subsistemas principales: Position_errors, Control y Robot.

2.1.1. Subsistema Robot

Este bloque, reutilizado de la práctica anterior, modela el comportamiento del robot. Recibe como entradas la velocidad lineal (V) y angular (W) y calcula la posición (x, y) y orientación (θ) del robot en cada instante de tiempo, basándose en el siguiente modelo dinámico:

$$x_k = x_{k-1} + V_{k-1} T_s \cos(\theta_{k-1})$$

$$y_k = y_{k-1} + V_{k-1} T_s \sin(\theta_{k-1})$$

$$\theta_k = \theta_{k-1} + W_{k-1} T_s$$

Donde T_s es el tiempo de muestreo.

2.1.2. Subsistema Position_errors

Este subsistema calcula los dos errores que tendrá el controlador:

- **Error de distancia (E_d):** La distancia euclidiana entre la posición actual del robot (x_k, y_k) y la posición objetivo $(refx, refy)$.

$$E_d = \sqrt{(refx - x_k)^2 + (refy - y_k)^2}$$

- **Error de ángulo (E_θ):** La diferencia entre el ángulo necesario para apuntar al objetivo y la orientación actual del robot. Se utiliza la función `atan2` para obtener el ángulo correcto en los cuatro cuadrantes.

$$E_\theta = \text{atan2}(refy - y_k, refx - x_k) - \theta_k$$

Adicionalmente, este bloque incluye una **condición de parada** que detiene la simulación si el error de distancia E_d es inferior a 0.01 metros.

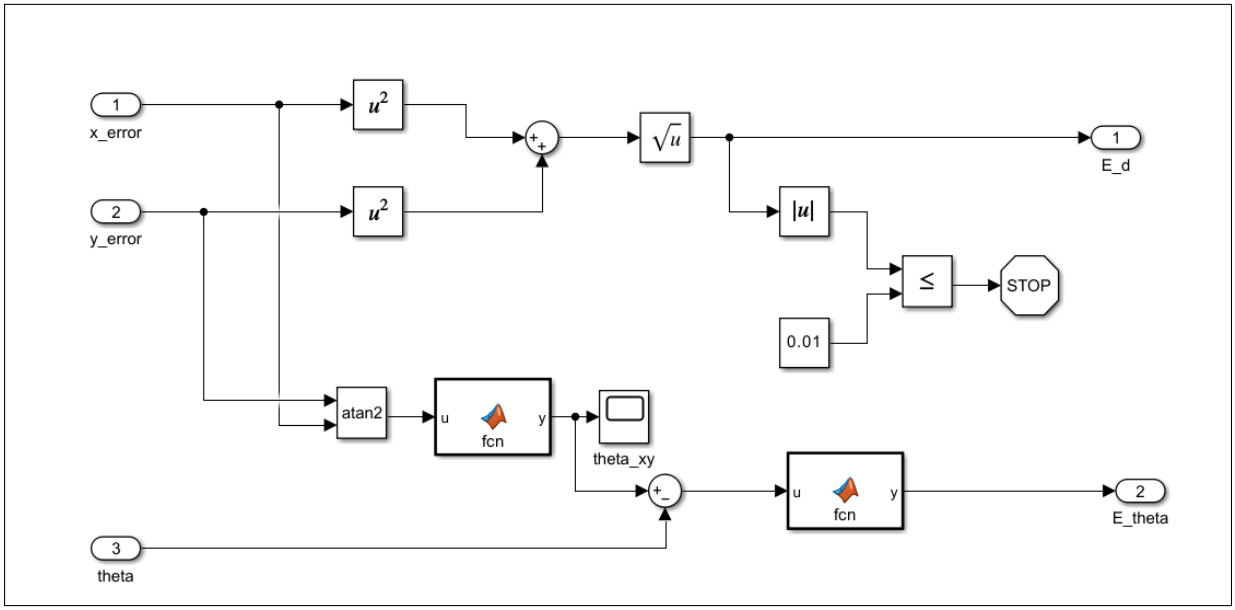


Figura 42: Implementación interna del subsistema de cálculo de errores.

2.1.3. Subsistema Control (Caja Negra)

Este es el bloque proporcionado `controlblackbox.slx`, cuyo funcionamiento interno es desconocido, actúa como el experto que se va a emular. Recibe los errores E_d y E_θ y genera las señales de control V y W que guían al robot hacia el objetivo.

2.2. Desarrollo de la Práctica

El desarrollo se divide en tres fases: la generación de datos a partir del controlador, el entrenamiento de la red neuronal con esos datos, y la posterior comparación entre el controlador original y el neuronal.

2.2.1. Apartado C

Tras la configuración previa del sistema de control (explicación introductoria de la práctica) y de los parámetros de simulación (apartado A), se puede hacer uso del código proporcionado en el apartado B para comprobar que se generan las variables de las entradas y salidas del controlador, y las salidas del robot en la simulación:

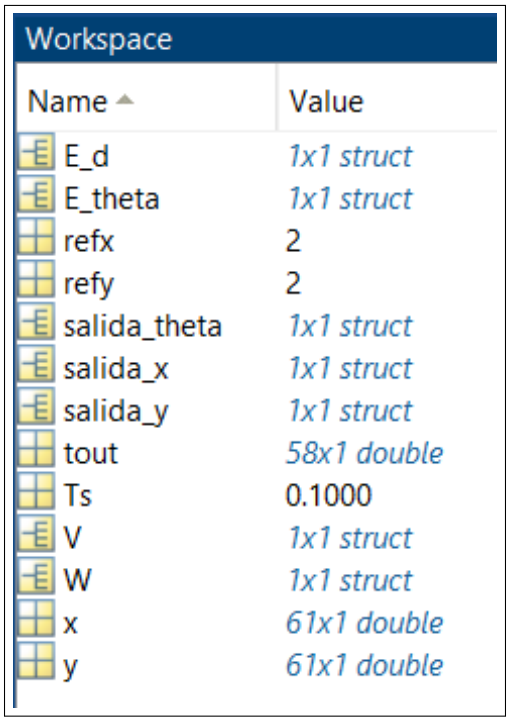


Figura 43: Apartado C

Efectivamente, se obtienen las variables deseadas. Al acceder a una de ellas, por ejemplo *E_theta*, se ve lo siguiente:

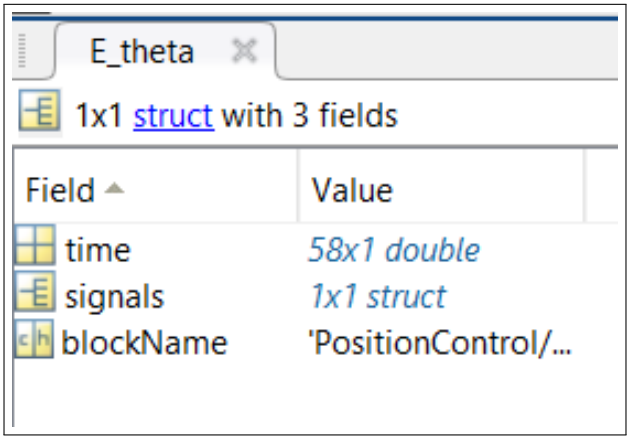


Figura 44: Estructura de la variable *E_theta*

La variable *E_theta* contiene tres campos. Esto se debe a la configuración de los bloques *To workspace*, del apartado 1.4 de la práctica.

2.2.2. Apartado D

El código proporcionado muestra la trayectoria del robot:

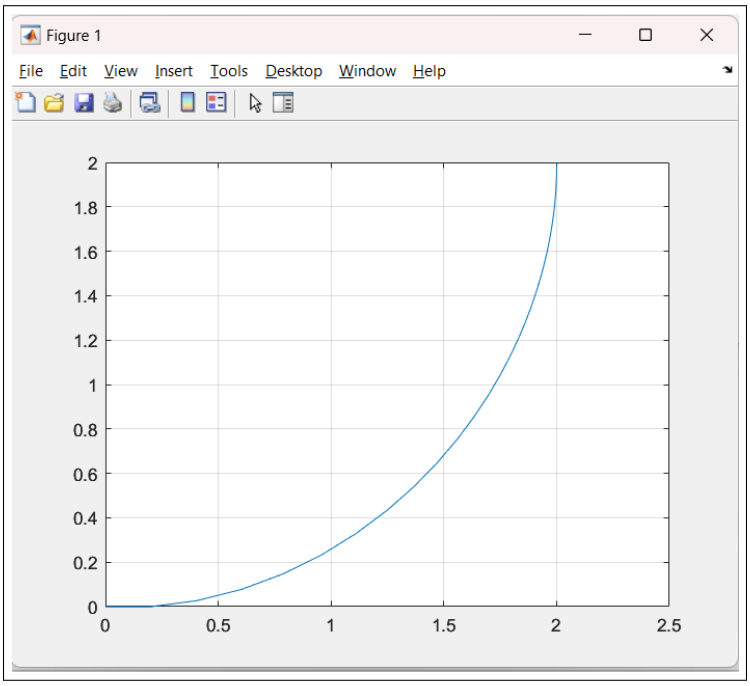


Figura 45: Apartado D

Se comprueba que, efectivamente, el robot comienza su trayectoria en el punto $(0, 0)$ y finaliza en el punto de referencia del apartado B: el punto $(2, 2)$.

2.2.3. Apartado E

Haciendo uso del código proporcionado en el apartado E, se realizan las 30 simulaciones con datos de entrada aleatorios y se obtienen las matrices deseadas:

inputs

2x1950 double

	1	2	3	4	5	6	7	8	9	10	11
1	5.1353	4.9487	4.7258	4.4752	4.2062	3.9273	3.6458	3.3679	3.0981	2.8397	2.5951
2	0.9111	0.8219	0.7421	0.6707	0.6067	0.5494	0.4979	0.4516	0.4099	0.3722	0.3382
3											

Figura 46: Matriz *inputs*

En la matriz *inputs*, de tamaño 2×1950, se encuentran los valores de *E_d* y *E_theta*.

outputs

2x1950 double

	1	2	3	4	5	6	7	8	9	10	11
1	3.1472	3.3694	3.4833	3.5059	3.4555	3.3493	3.2031	3.0302	2.8414	2.6452	2.4481
2	1.3954	1.3205	1.2402	1.1576	1.0751	0.9947	0.9175	0.8443	0.7754	0.7110	0.6512
3											

Figura 47: Matriz *outputs*

En la matriz *outputs*, del mismo tamaño, se almacenan los valores de V y W.

2.2.4. Apartado F

Para el apartado del **diseño de la red neuronal**, se debe establecer el **número de neuronas de la capa oculta**. Recordar de las sesiones de teoría que se permite un máximo del **15 %** de las entradas. Si se tienen **2 entradas** en **30 posiciones**, se tienen **60 datos**:

$$2 \times 30 = 60 \times 0{,}15 = 9 \text{ neuronas} \tag{1}$$

Por tanto, se va a establecer el número de neuronas en la capa oculta a **9 neuronas**.

Al ejecutar el código se obtiene la siguiente ventana:

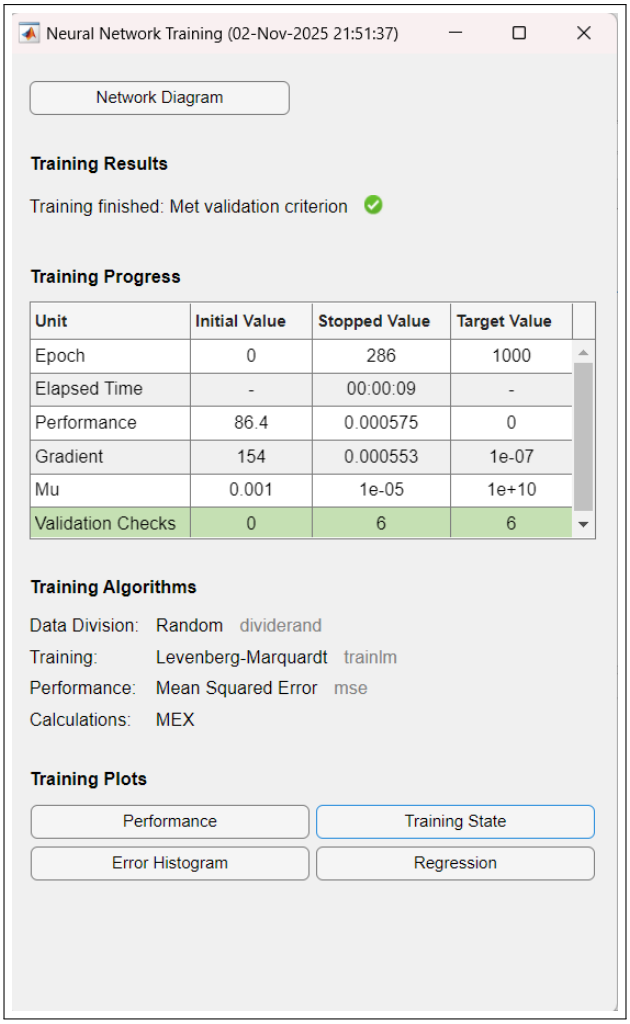


Figura 48: Resultados del entrenamiento de la red

Los resultados son variantes en cada ejecución de entrenamiento de la red, puesto que cabe la posibilidad de caer en un **mínimo local**. Aun así, las conclusiones generales son muy claras.

Respecto a las métricas:

- $MSE = 0{,}000575 \rightarrow$ un error cuadrático medio tan bajo indica un **ajuste muy bueno**
- $Gradiente = 0{,}000553 \rightarrow$ un gradiente muy cercano a cero muestra una **convergencia perfecta**
- $Validation\ checks = 6 \rightarrow$ se detuvo tras 6 fallos de validación consecutivos para **evitar sobreajuste**
- $Epochs = 286 \rightarrow$ número de interaciones necesarias para alcanzar el criterio de parada y **converger**

En la pestaña de **rendimiento** (*Performance*) se puede ver lo siguiente:

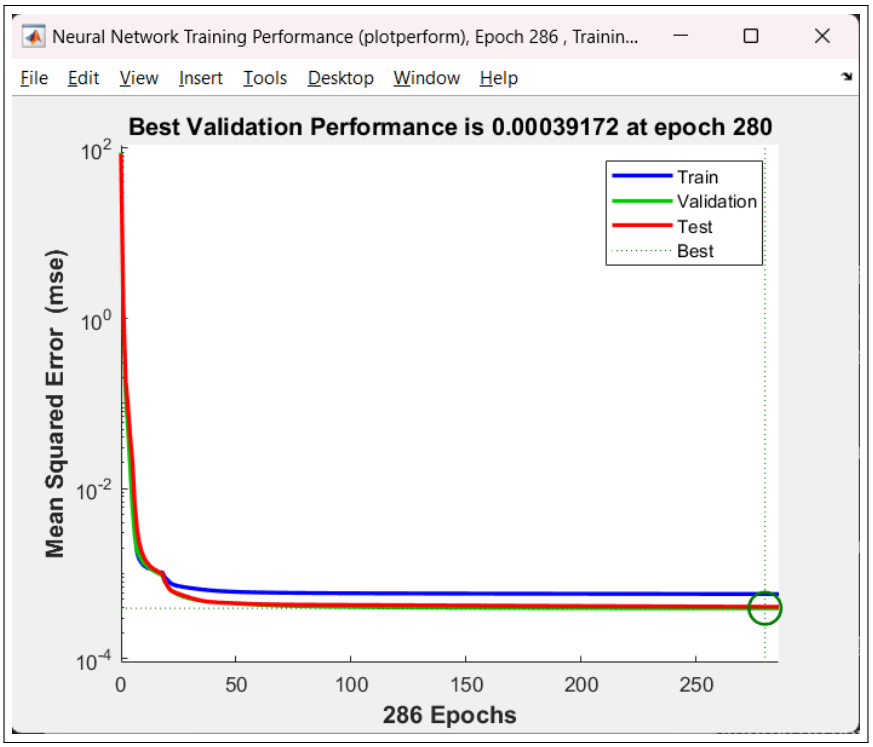


Figura 49: Pestaña *Performance*

En cuanto al comportamiento de los subconjuntos de entrenamiento, validación y test, se puede ver en la gráfica que **las tres curvas descienden de forma prácticamente idéntica**. Esto indica lo siguiente:

- **No hay sobreajuste**: si lo hubiera, las curvas de validación y test serían diferentes a la del entrenamiento
- Hay una **gran capacidad de generalización**: la red predice igual de bien con datos no vistos

Si se consulta la gráfica de **regresión**, se puede ver la siguiente imagen:

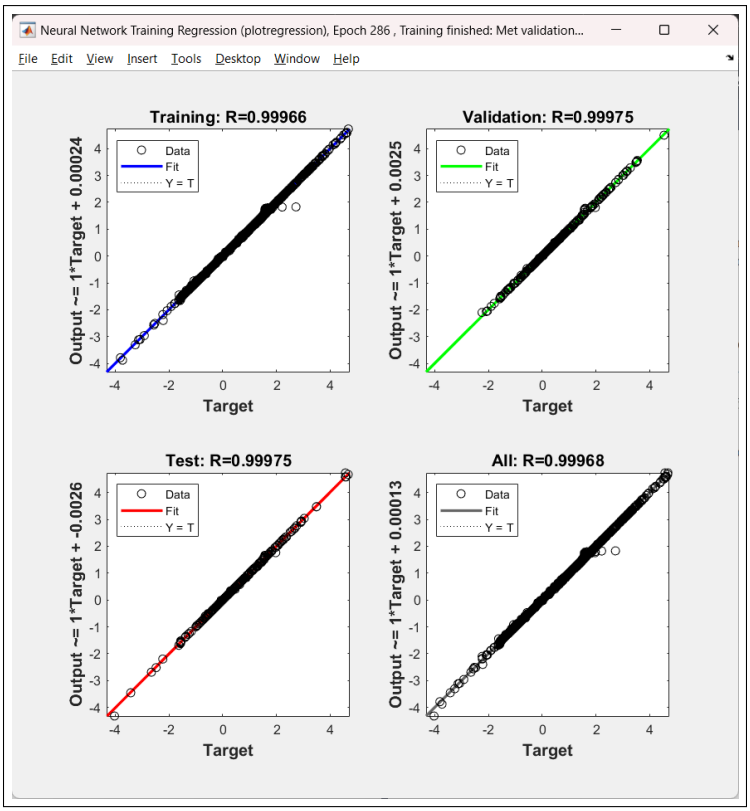


Figura 50: Pestaña *Regression*

Se puede ver como los **coeficientes de correlación** son, en todos los casos, muy próximos a 1, con valores:

- $Training \rightarrow R = 0{,}99966$
- $Validation \rightarrow R = 0{,}99975$
- $Test \rightarrow R = 0{,}99975$

Los puntos están alineados con la diagonal, por tanto **las salidas predichas coinciden con las salidas deseadas del controlador**.

2.2.5. Apartado G

Al ejecutar el comando `gensim(net,Ts)` en MATLAB, se obtiene el siguiente bloque en un fichero nuevo de Simulink:

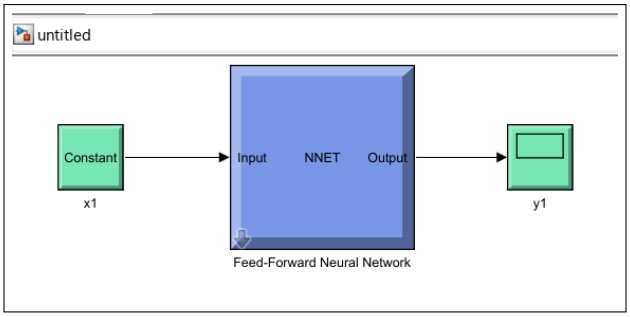


Figura 51: Sentencia `gensim`

2.2.6. Apartado H

Al copiar el sistema de `PositionControlNet.slx` e intoducir el bloque de la **red neuronal** en lugar del bloque de control, se queda el sistema de la siguiente manera:

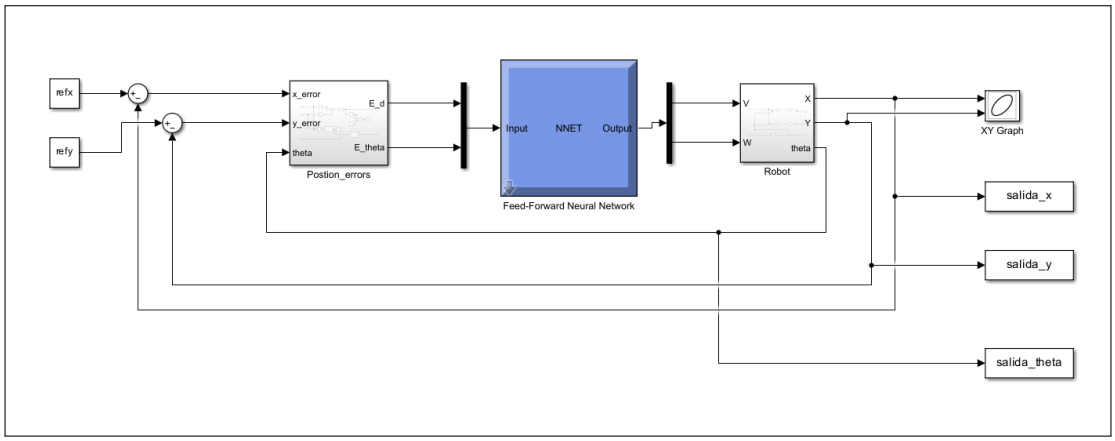


Figura 52: Sistema `PositionControlNet.slx`

2.2.7. Apartado I

Para el último apartado hay que comparar el comportamiento de ambas redes. Se puede hacer uso de un script como el siguiente:

```
%% Script de comparacion:

% Limpiar la consola:
clear;
clc;
close all;

%% Parametros:

% Tiempo de muestreo:
Ts = 100e-3;

% Posicion objetivo:
refx = 10.0;
refy = 10.0;

fprintf('=== COMPARACION DE CONTROLADORES ===\n');
fprintf('Objetivo: (%.2f, %.2f) m\n\n', refx, refy);

%% Simulacion del controlador original:

fprintf('Simulando controlador original...\n');
sim('PositionControl.slx');

% Extraer datos de la simulacion:
x_original = salida_x.signals.values;
y_original = salida_y.signals.values;
t_original = salida_x.time;

```

```
%% Simulacion del neurocontrolador:

fprintf('Simulando neurocontrolador...\n');
sim('PositionControlNet.slx');

% Extraer datos de la simulacion:
x_neuronal = salida_x.signals.values;
y_neuronal = salida_y.signals.values;
t_neuronal = salida_x.time;

%% Ajustar longitudes por si fueran diferentes:

% Tomar el minimo numero de muestras de ambas simulaciones:
n_min = min(length(t_original), length(t_neuronal));

% Recortar ambas trayectorias a la misma longitud:
x_original = x_original(1:n_min);
y_original = y_original(1:n_min);
x_neuronal = x_neuronal(1:n_min);
y_neuronal = y_neuronal(1:n_min);

%% Calcular error entre trayectorias:

% Calcular el error de posicion mediante la distancia euclidea:
error_posicion = sqrt((x_original - x_neuronal).^2 + (y_original -
    y_neuronal).^2);

%% Mostrar estadisticas por consola:

fprintf('\n========================================\n');
fprintf('ESTADISTICAS DEL ERROR:\n');
fprintf('========================================\n');
fprintf('Error de posicion (distancia euclidea):\n');
fprintf('  Error medio:  %.6f m\n', mean(error_posicion));
fprintf('  Error maximo: %.6f m\n', max(error_posicion));
fprintf('  Error final:  %.6f m\n', error_posicion(end));
fprintf('========================================\n\n');

%% Crear graficas:

figure('Name', 'Comparacion de controladores', 'Position', [100, 100,
    1200, 500]);

% ===== Grafica 1: trayectorias superpuestas =====
subplot(1, 2, 1);
% Trayectoria del controlador original:
plot(x_original, y_original, 'b-', 'LineWidth', 2, 'DisplayName', '
    Controlador original');
hold on;
% Trayectoria del neurocontrolador:
plot(x_neuronal, y_neuronal, 'r--', 'LineWidth', 2, 'DisplayName', '
    Neurocontrolador');
% Punto de inicio:
plot(0, 0, 'ko', 'MarkerSize', 7, 'MarkerFaceColor', 'k', 'DisplayName',
     'Inicio');
% Punto objetivo:
plot(refx, refy, 'rx', 'MarkerSize', 15, 'LineWidth', 3, 'DisplayName',
    'Objetivo');
```

```
grid on;
xlabel('Posicion X (m)', 'FontSize', 11);
ylabel('Posicion Y (m)', 'FontSize', 11);
title('Trayectorias', 'FontSize', 12, 'FontWeight', 'bold');
legend('Location', 'best', 'FontSize', 10);
axis equal;

% ===== Grafica 2: error de posicion =====
subplot(1, 2, 2);
% Grafica de barras del error:
bar(1:n_min, error_posicion, 'k', 'EdgeColor', 'none');
grid on;
xlabel('Muestra (tiempo discreto)', 'FontSize', 11);
ylabel('Error (m)', 'FontSize', 11);
title('Error de posicion', 'FontSize', 12, 'FontWeight', 'bold');
ylim([0, max(error_posicion)*1.1]);

% Mensaje de confirmacion:
fprintf('Graficas generadas correctamente.\n');
```

Se obtienen los siguientes resultados por consola:

```
Command Window
=== COMPARACIÓN DE CONTROLADORES ===
Objetivo: (10.00, 10.00) m

Simulando controlador original...
Simulando neurocontrolador...

========================================
ESTADÍSTICAS DEL ERROR:
========================================
Error de posición (distancia euclídea):
  Error medio:  0.033281 m
  Error máximo: 0.249663 m
  Error final:  0.001147 m
========================================

Gráficas generadas correctamente.
fx >>
```

Figura 53: Comparación de controladores - salida por consola

También se obtienen las siguientes gráficas:

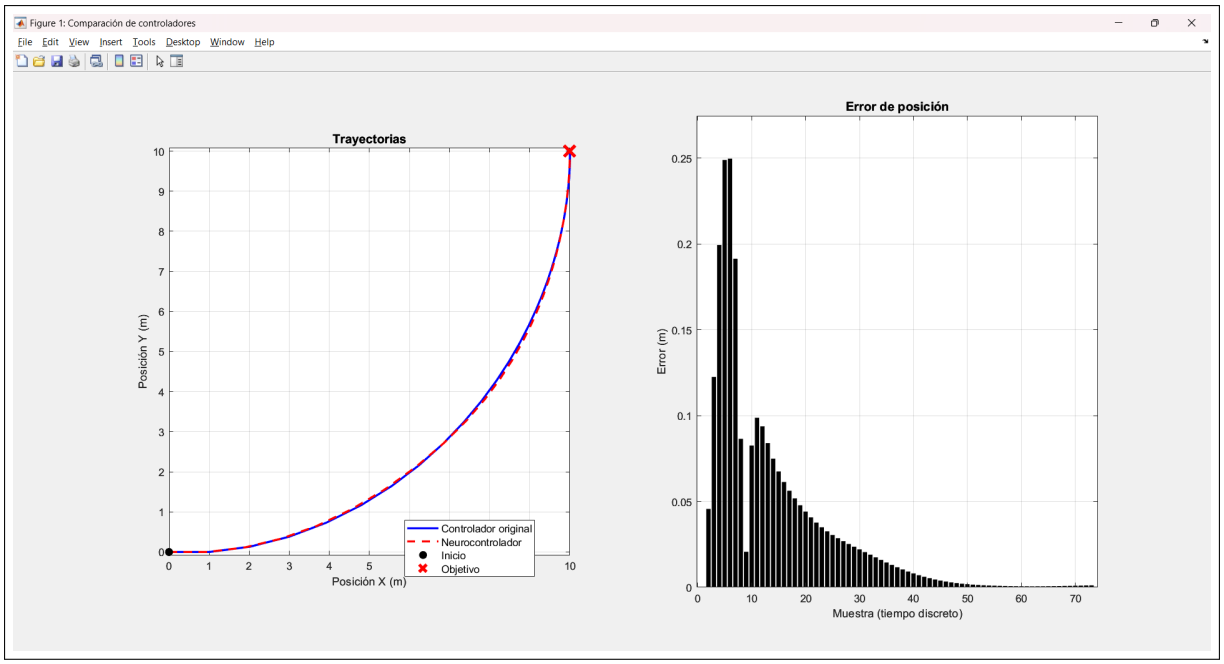


Figura 54: Comparación de controladores - gráficas

De los resultados por consola se puede concluir lo siguiente:

- El error medio es muy pequeño, unos 3.3cm → ambas trayectorias difieren en menos de 4 centímetros
- El error máximo, de unos 25cm, ocurre al principio del recorrido: en sistemas de control, es habitual que haya más diferencias al arrancar
- El error final es de unos 1.1mm: prácticamente perfecto

En cuanto a los resultados de la gráfica de error:

- Se muestran unos picos altos al comienzo, debido a las pequeñas diferencias en la respuesta inicial
- Posteriormente se produce un descenso gradual, indicando una sincronización de los controladores
- Al aproximarse al objetivo, ambos controladores convergen de manera casi idéntica en el punto

En general, y observando la gráfica de las trayectorias, se puede ver como los errores son prácticamente despreciables durante todo el recorrido, ya que ambas curvas están solapadas una sobre la otra.