



Universidad
de Alcalá

SISTEMAS DE CONTROL INTELIGENTE

Práctica 1.
Identificación y control neuronal

Jorge Revenga Martín de Vidales
Ángel Salgado Aldao

Grado en Ingeniería Informática
Universidad de Alcalá

31 de octubre de 2023

Índice

I	2
1. Ejercicio 1. Perceptrón.	2
1.1. Código	2
1.2. Preguntas	2
1.3. Ejecución	3
2. Ejercicio 2. Aproximación de funciones.	4
2.1. Solución	4
2.2. Ejecución	5
2.3. Entrenamiento con Descenso de Gradiente estándar - 'traingd'	5
2.4. Entrenamiento con Descenso de Gradiente con Momento - 'traingdm'	6
2.5. Entrenamiento con Levenberg-Marquardt - 'trainlm'	6
2.6. Broyden-Fletcher-Goldfarb-Shanno - 'trainbr'	7
3. Ejercicio 3. Aproximación de funciones (II).	8
3.1. Backpropagation - train	9
3.2. Backpropagation con división de datos 60/20/20 - train	10
3.3. Descenso gradiente con división de datos 70/15/15 - traingd	11
3.4. Descenso gradiente con división de datos 60/20/20 - traingd	12
3.5. Levenberg-Marquardt con división de datos 70/15/15 - trainlm	13
3.6. Levenberg-Marquardt con división de datos 60/20/20 - train	14
4. Ejercicio 4. Clasificación.	15
4.1. Backpropagation	16
4.2. Backpropagation con división de datos 60/20/20	17
4.3. Descenso gradiente	18
4.4. Descenso gradiente con división de datos 60/20/20	19
4.5. Levenberg-Marquardt	20
4.6. Gráficas con Levenberg-Marquardt con división 60/20/20	21
II Diseño de un control de posición mediante una red neuronal no recursiva.	22
5. Desarrollo	22
5.1. Esquema de Simulink	22
5.2. Creaación script para simular el diagrama.	22
5.3. Ejecución script y comprobación de variables.	23
5.4. Trayectoria del robot.	24
5.5. Generar N posiciones aleatorias, simular y guardar en variables.	25
5.6. Entrenamiento de red neuronal con 10 neuronas en la capa oculta.	25
5.7. Generación de bloque de Simulink con el controlador neuronal.	26
5.8. Esquema de Simulink con la red neuronal en lugar del bloque controlador.	26
5.9. Comparación.	27

Parte I

1. Ejercicio 1. Perceptrón.

Se desea clasificar un conjunto de datos pertenecientes a cuatro clases diferentes. Los datos y las clases a las que pertenecen con los que se muestra a continuación:

x_1	x_0	Clase
0.1	1.2	2
0.7	1.8	2
0.8	1.6	2
0.8	0.6	0
1.0	0.8	0
0.3	0.5	3
0.0	0.2	3
-0.3	0.8	3
-0.5	-1.5	1
-1.5	-1.3	1

Se desea diseñar un clasificador neuronal mediante un perceptrón simple que clasifique estos datos. Diseñe el clasificador, visualice los parámetros de la red y dibuje los datos junto con las superficies que los separan.

1.1. Código

```

1  P = [0.1 0.7 0.8 0.8 1.0 0.3 0 -0.3 -0.5 -1.5; 1.2 1.8 1.6 0.6 0.8 0.5 0.2 0.8 -1.5 -1.3];
2  T = [1 1 1 0 0 1 1 1 0 0; 0 0 0 0 0 1 1 1 1 1];
3
4  net = perceptron;
5  net = train(net, P, T);
6  plotpv(P, T);
7  plotpc(net.iw{1,1}, net.b{1});

```

1.2. Preguntas

¿Consigue la red separar los datos?

Sí, los clasifica en las 4 clases correctamente

¿Cuántas neuronas tiene la capa de salida?, ¿por qué?

Tiene 4 neuronas, cada una genera una salida que representa la probabilidad de pertenencia a una clase particular, como tenemos 4 clases, tenemos 4 neuronas

¿Qué ocurre si se incorpora al conjunto un nuevo dato: [0.0 -1.5] de la clase 3?

Los datos dejan de ser linealmente separables, por lo que las líneas que dividen los datos en las clases dejan de aplicarse a todos los datos, su pendiente ha cambiado pero el nuevo dato no está separado en el mismo espacio que el resto de datos de clase 3. Está con los datos de clase 1

1.3. Ejecución

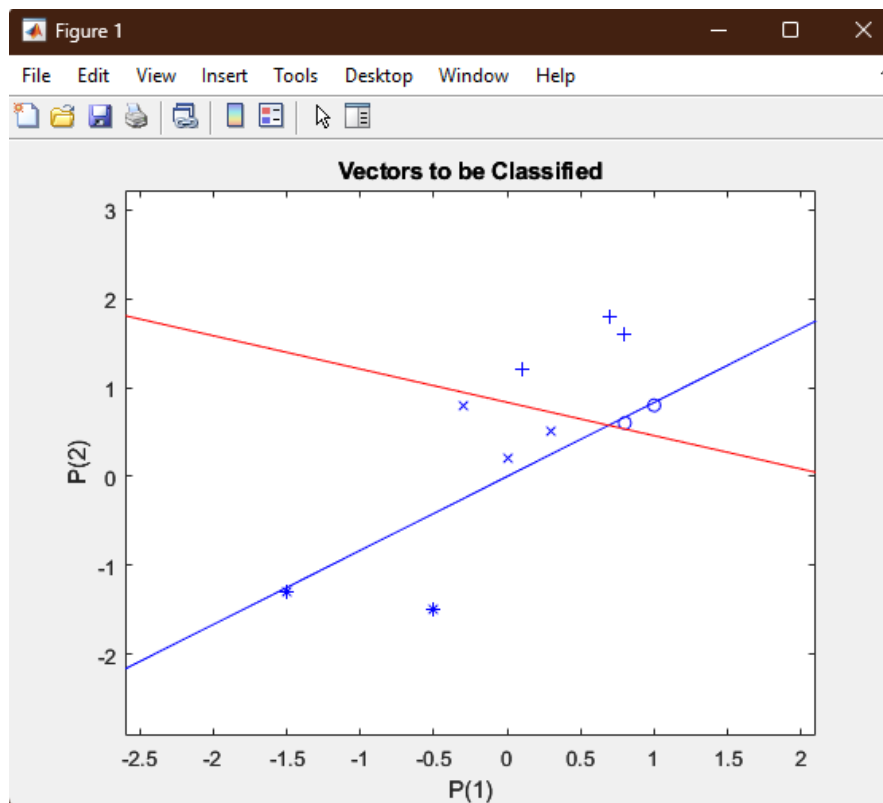


Figura 1: Ejecución ejercicio 1.

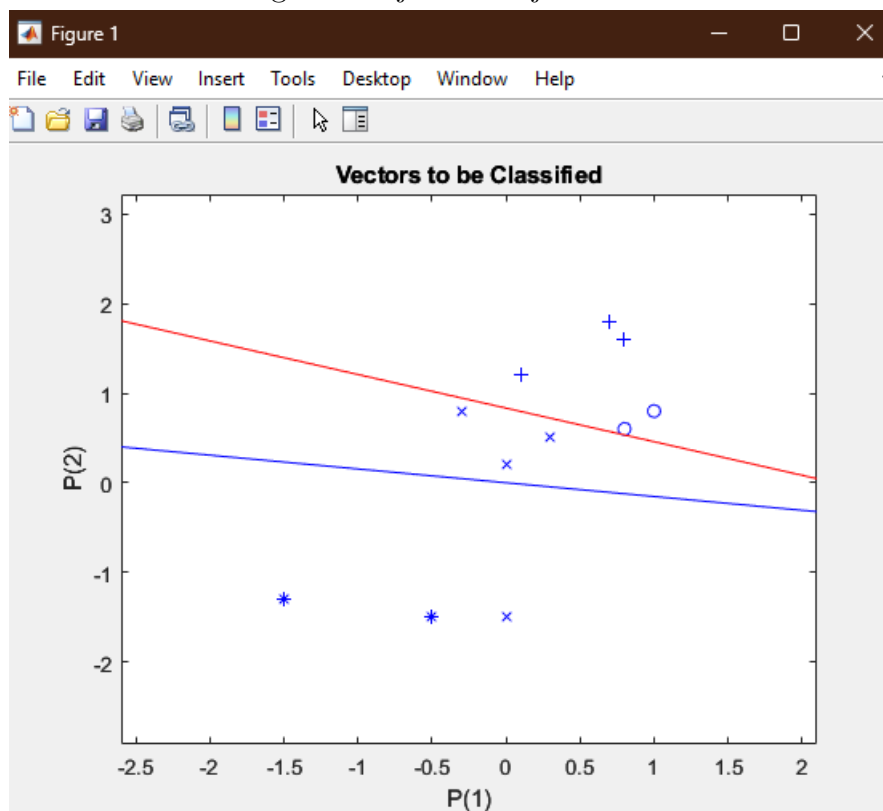


Figura 2: Ejecución ejercicio 1 con el nuevo dato.

2. Ejercicio 2. Aproximación de funciones.

Una de las aplicaciones inmediatas de las redes neuronales es la aproximación de funciones. Para ello, Matlab dispone de una red optimizada, `fitnet`, con la que se trabajará en este ejercicio. El objetivo en este caso es aproximar la función $f = \text{sinc}(t)$ tal y como se muestra a continuación:

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  % APROXIMACIÓN DE FUNCIONES
3  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4  clear all; close all;
5  % DEFINICIÓN DE LOS VECTORES DE ENTRADA-SALIDA
6  % =====
7  t = -3:.1:3; % eje de tiempo
8  F=sinc(t)+.001*randn(size(t)); % función que se desea aproximar
9  plot(t,F,'+');
10 title('Vectores de entrenamiento');
11 xlabel('Vector de entrada P');
12 ylabel('Vector Target T');
13 % DISEÑO DE LA RED
14 % =====
15 hiddenLayerSize = 4;
16 net = fitnet(hiddenLayerSize,'trainrp');
17 net.divideParam.trainRatio = 70/100;
18 net.divideParam.valRatio = 15/100;
19 net.divideParam.testRatio = 15/100;
20 net = train(net,t,F);
21 Y=net(t);
22 plot(t,F,'+'); hold on;
23 plot(t,Y,'-r'); hold off;
24 title('Vectores de entrenamiento');
25 xlabel('Vector de entrada P');
26 ylabel('Vector Target T');

```

Estudie los efectos sobre la solución final de modificar el método de entrenamiento (consulte la ayuda de Matlab y pruebe 4 métodos diferentes) y el número de neuronas de la capa oculta.

2.1. Solución

Para poder visualizar el código en funcionamiento primero debemos definir la función $\text{sinc}(t)$. Lo haremos en un archivo llamado `sinc.m`. En nuestro caso hemos definido la siguiente función $\text{sinc}(t)$:

```

1  function y = sinc(t)
2      y = sin(pi * t) ./ (pi * t);
3  end

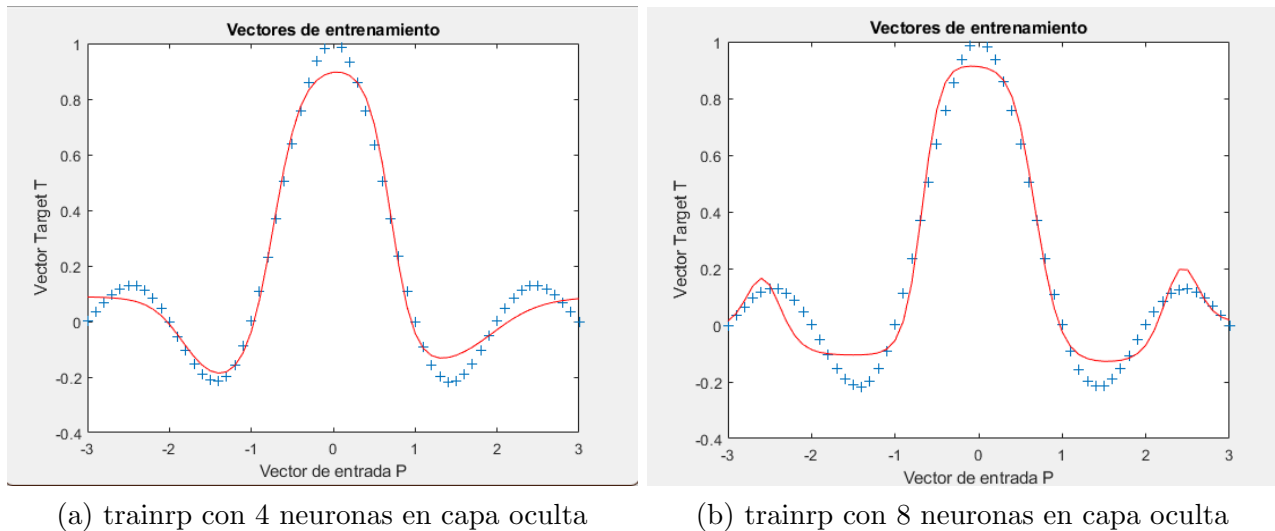
```

Para modificar el método de entrenamiento basta con cambiar el segundo parámetro de la función `fitnet()` por otra función de entrenamiento, para obtener más información también vamos a cambiar el número de neuronas de la capa oculta de 4 a 8 y comparar el desempeño de cada método de entrenamiento.

2.2. Ejecución

En el código del enunciado se usa la función 'trainrp', que se refiere a 'Resilient Backpropagation'. Este es un método de entrenamiento para redes neuronales que se enfoca en la convergencia rápida mediante la adaptación dinámica de las tasas de aprendizaje para cada peso de la red. Este es su desempeño:

Figura 3: Resilient Backpropagation



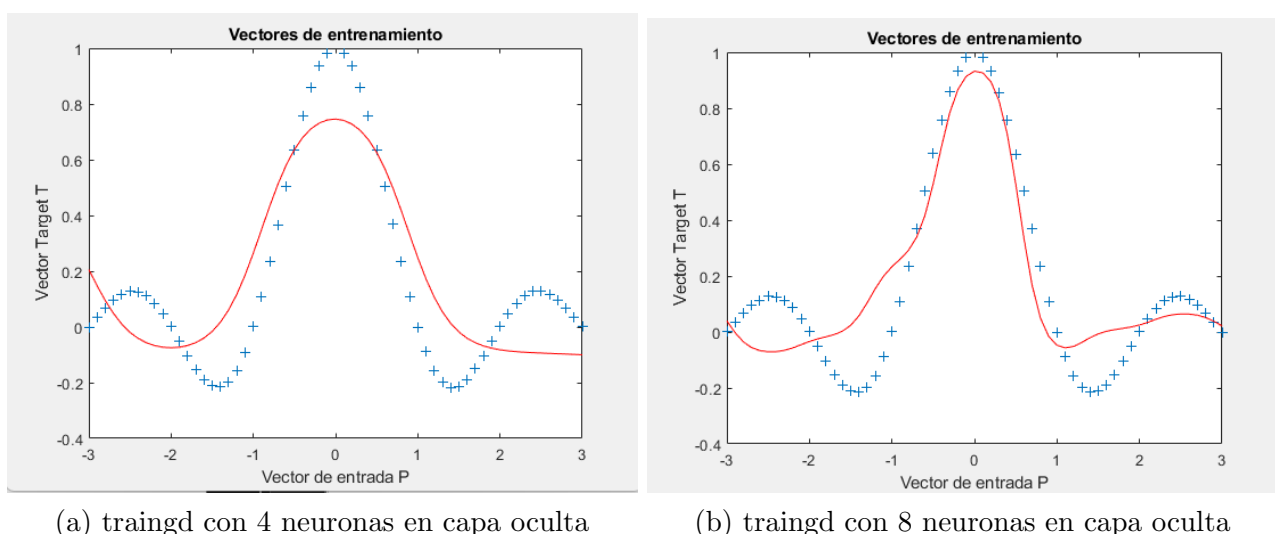
(a) trainrp con 4 neuronas en capa oculta

(b) trainrp con 8 neuronas en capa oculta

2.3. Entrenamiento con Descenso de Gradiente estándar - 'traingd'

Esta función de entrenamiento utiliza el método de descenso de gradiente estándar. Ajusta los pesos de la red en la dirección opuesta al gradiente de la función de costo con respecto a los pesos. Se considera simple y efectivo, pero que puede converger lentamente en algunos casos. En nuestro caso los resultados con 4 y 8 neuronas en la capa oculta no difieren mucho, por lo que se podría decir (con poco rigor) que converge lentamente.

Figura 4: Descenso gradiente estándar



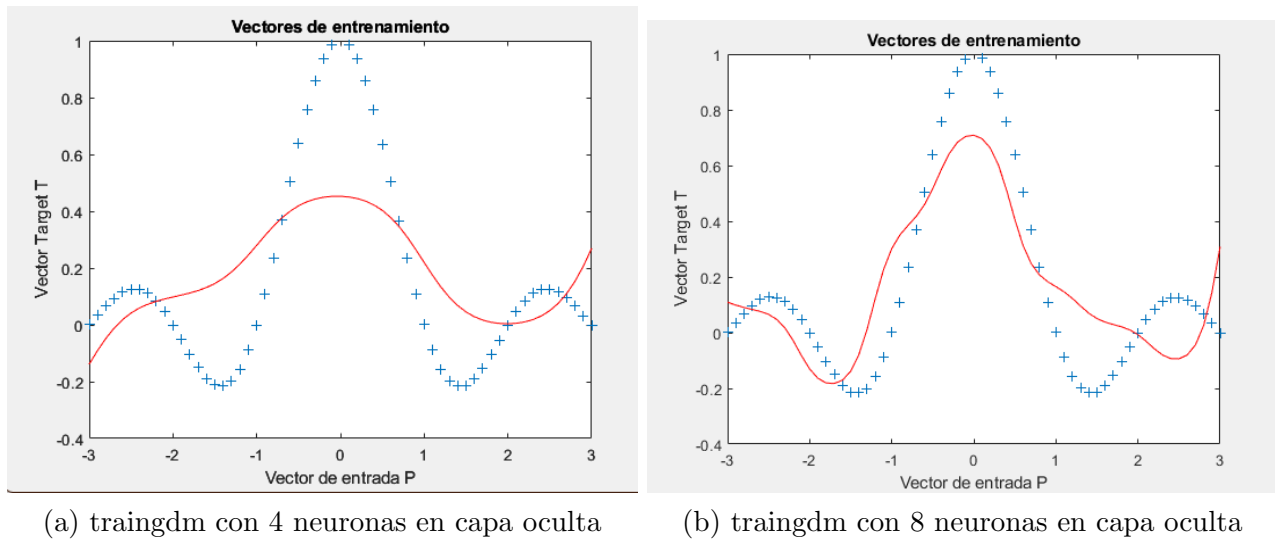
(a) traingd con 4 neuronas en capa oculta

(b) traingd con 8 neuronas en capa oculta

2.4. Entrenamiento con Descenso de Gradiente con Momento - 'traingdm'

Similar al descenso de gradiente, pero con la adición de "momentum". El momento ayuda a acelerar el entrenamiento al mantener una memoria de las actualizaciones de los pesos anteriores. Teóricamente esto puede ayudar a superar óptimos locales y acelerar la convergencia. Al aumentar el número de neuronas el resultado no muestra mucha mejoría, al igual que el descenso gradiente estándar

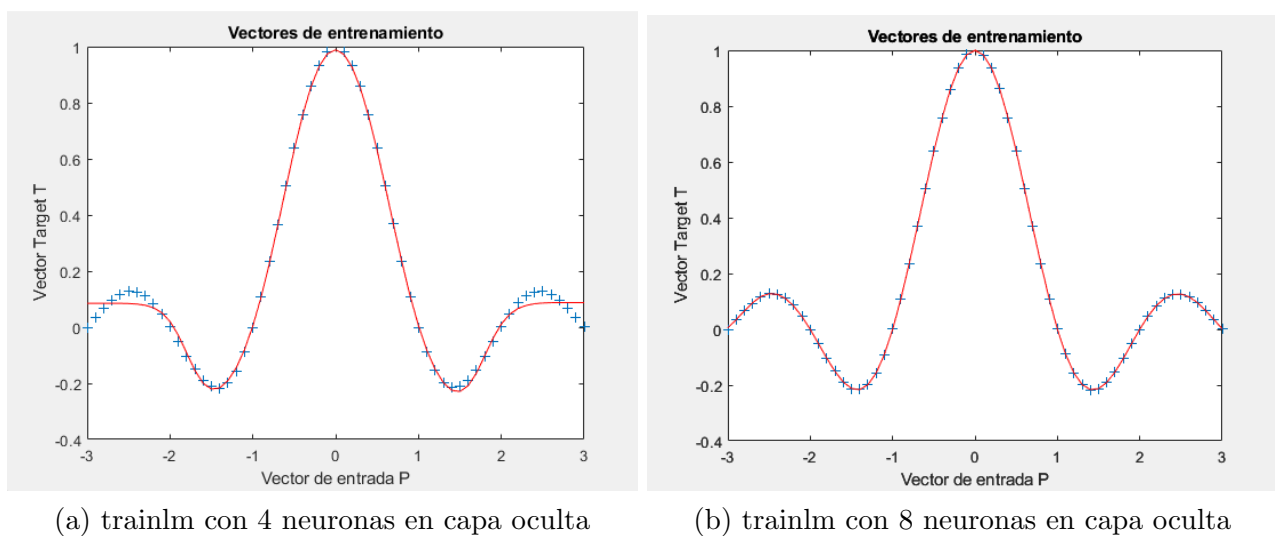
Figura 5: Descenso gradiente con momento



2.5. Entrenamiento con Levenberg-Marquardt - 'trainlm'

Utiliza el algoritmo de Levenberg-Marquardt, que es una técnica de optimización no lineal. Se observa muy buen desempeño con 4 neuronas en la capa oculta y notable mejora al subir a 8 neuronas.

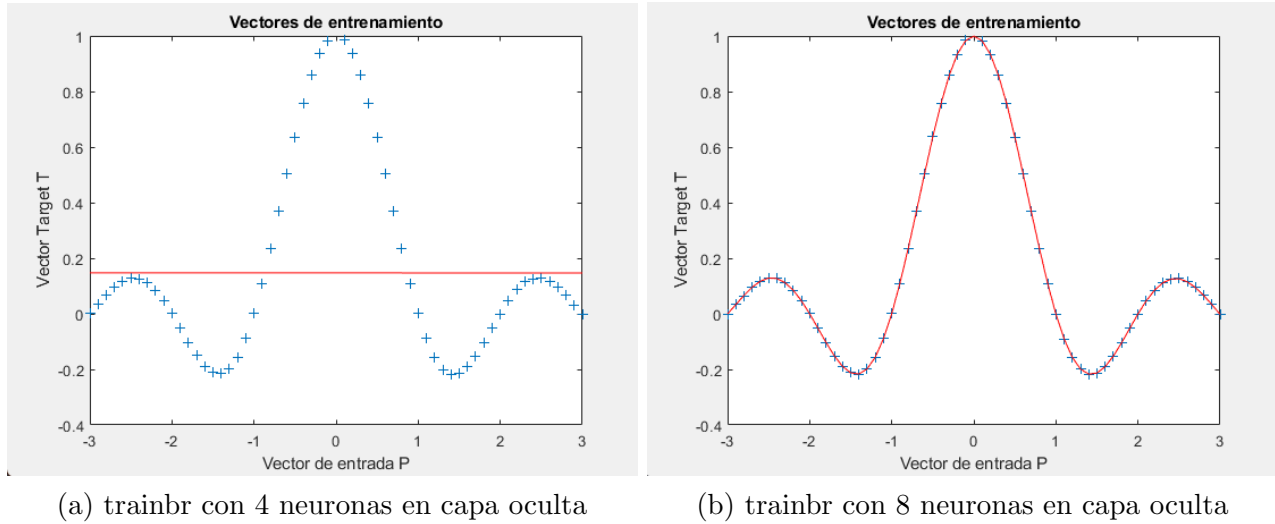
Figura 6: Levenberg-Marquardt



2.6. Broyden-Fletcher-Goldfarb-Shanno - 'trainbr'

Utiliza el algoritmo Broyden-Fletcher-Goldfarb-Shanno (BFGS), una técnica de optimización quasi-Newton. Podemos ver cómo el resultado del entrenamiento con 4 neuronas y con 8 es completamente diferente, pues la primera red no se acerca a la forma de la función y la segunda la aproxima con gran precisión:

Figura 7: Levenberg-Marquardt



3. Ejercicio 3. Aproximación de funciones (II).

En este ejercicio, se estudiarán en detalle las herramientas que facilita Matlab para el diseño y prueba de redes neuronales ejecutando el siguiente código de ejemplo:

```
1  % Carga de datos de ejemplo disponibles en la toolbox
2  [inputs,targets] = simplefit_dataset;
3  % Creación de la red
4  hiddenLayerSize = 10;
5  net = fitnet(hiddenLayerSize);
6  % División del conjunto de datos para entrenamiento, validación y test
7  net.divideParam.trainRatio = 70/100;
8  net.divideParam.valRatio = 15/100;
9  net.divideParam.testRatio = 15/100;
10 % Entrenamiento de la red
11 [net,tr] = train(net,inputs,targets);
12 % Prueba
13 outputs = net(inputs);
14 errors = gsubtract(outputs,targets);
15 performance = perform(net,targets,outputs)
16 % Visualización de la red
17 view(net)
```

Explore las gráficas disponibles:

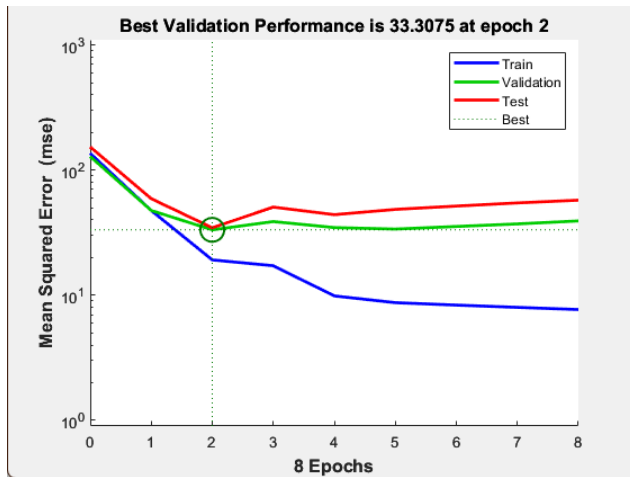
1. Performance: gráfica que representa el error en función del número de épocas para los datos de entrenamiento, validación y test.
2. Traininig State: evolución del entrenamiento.
3. Error Histogram: histograma del error.
4. Regression y Fit: ajuste de los datos de entrenamiento, validación y test.

Pruebe este mismo script con el conjunto de datos `bodyfat_dataset`, y evalúe sus resultados. Estudie la mejora que supone utilizar distintos métodos de entrenamiento y una división diferente de los datos (entrenamiento, validación y test).

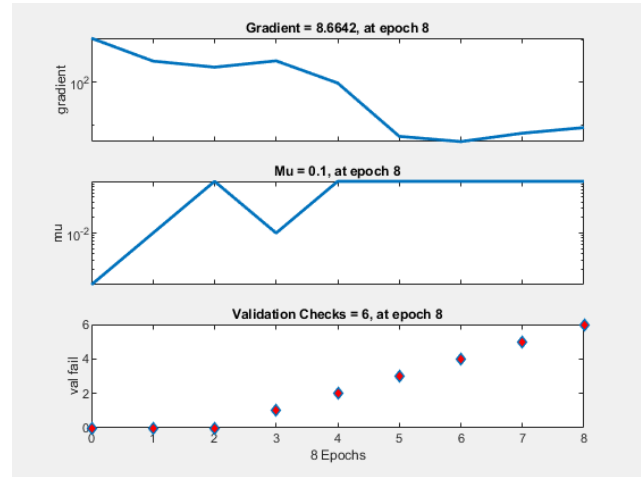
3.1. Backpropagation - train

Al ejecutar el código con el conjunto de datos `bodyfat_dataset`, obtenemos las gráficas de la figura 8. Estas nos proporcionan información útil sobre el entrenamiento. En la figura (a) podemos observar cómo cambia el error de la red con cada grupo de datos en función de las épocas de entrenamiento. Se observa que a partir de la segunda época el error disminuye para los datos del conjunto de entrenamiento, pero aumenta para el de validación, lo cual probablemente sea un ejemplo de sobreajuste de la red a los datos de entrenamiento.

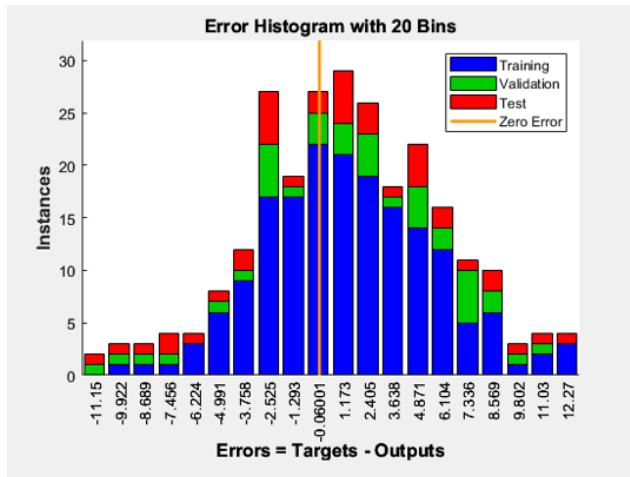
Figura 8: Gráficas con Backpropagation



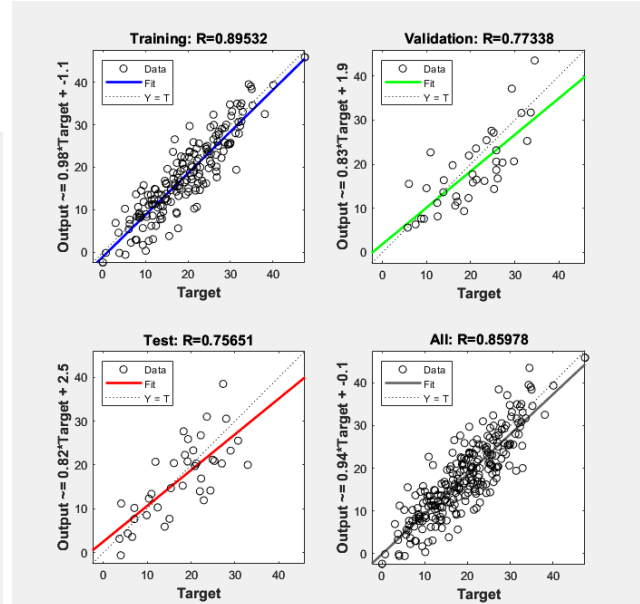
(a) Performance con train



(b) Training state con train



(c) Error Histogram con train

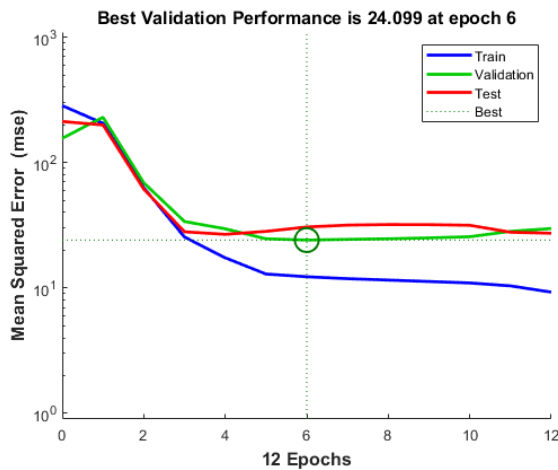


(d) Regression con train

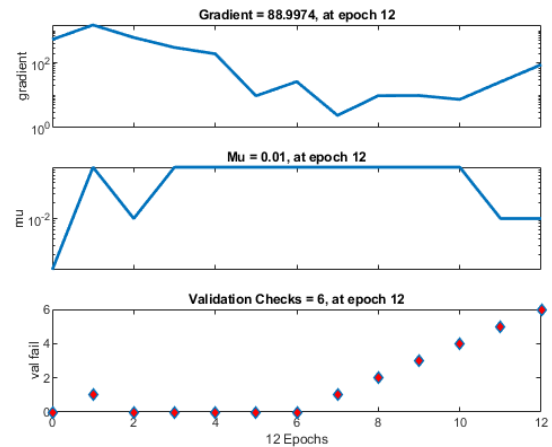
3.2. Backpropagation con división de datos 60/20/20 - train

Se ha repetido la ejecución anterior pero con una división diferente de los datos, en este caso hemos pasado de 70/15/15 a 60/20/20. Se puede ver que el punto en el que el mejor rendimiento respecto a los datos de validación ha pasado de ser en la época 6 en lugar de la 2 y una distribución más cercana a cero del error en el histograma. Además de esto no se observan cambios significativos

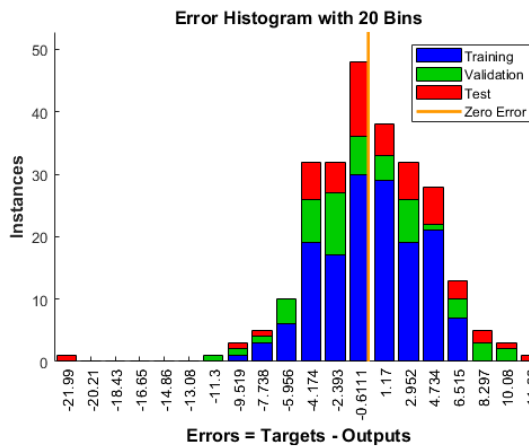
Figura 9: Gráficas con Backpropagation con división 60/20/20



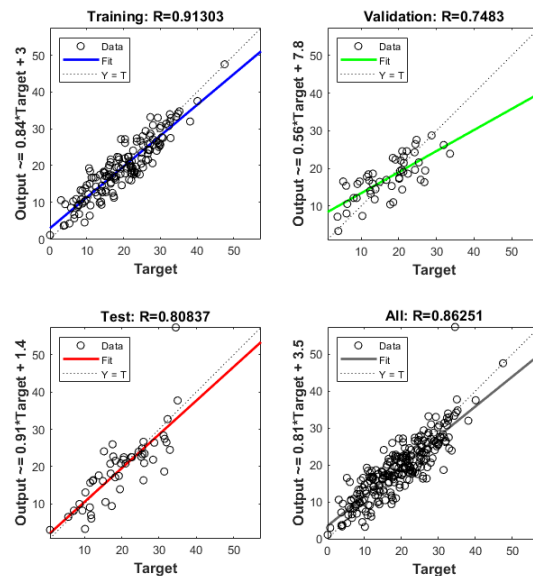
(a) Performance con train



(b) Training state con train



(c) Error Histogram con train

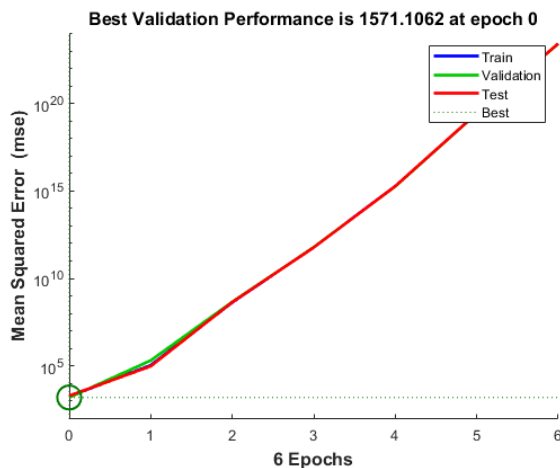


(d) Regression con train

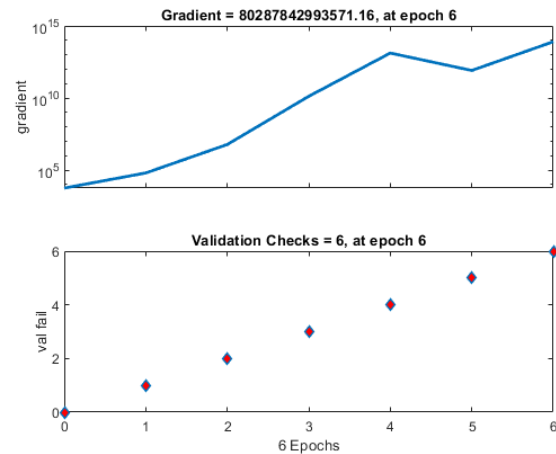
3.3. Descenso gradiente con división de datos 70/15/15 - traingd

Al repetir el proceso con el método de entrenamiento con descenso gradiente podemos observar que el desempeño empeora con el incremento de las épocas en todos los grupos de datos. También vemos como el error está tan alejado de cero que la línea vertical de “Zero Error” ni siquiera aparece en el histograma. Esto sumado al desplazamiento vertical de la línea de regresión respecto a la diagonal ($x=y$) en la gráfica de regresión nos muestra que la red tiende a realizar una sobreestimación de las predicciones.

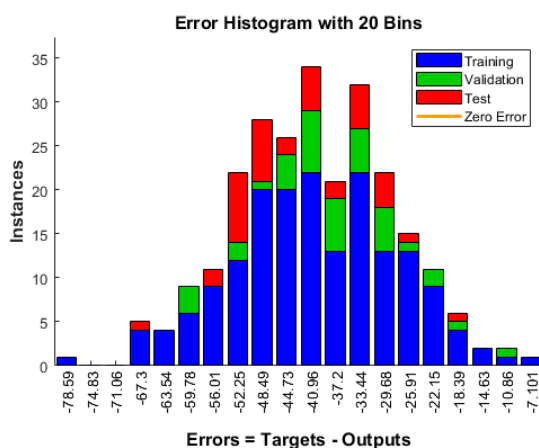
Figura 10: Gráficas con Descenso gradiente con división 70/15/15



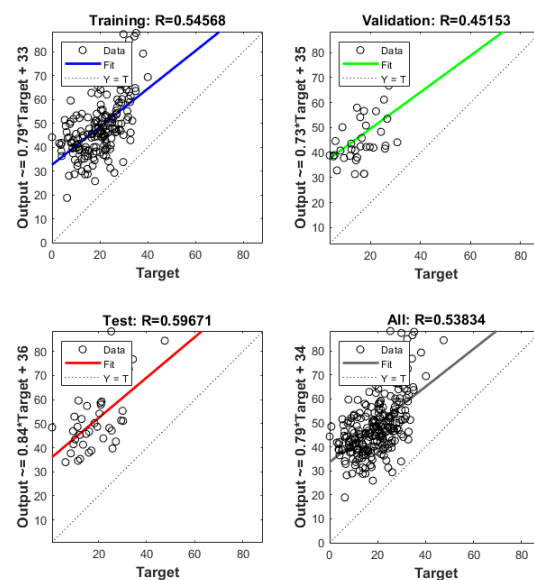
(a) Performance con traingd



(b) Training state con traingd



(c) Error Histogram con traingd

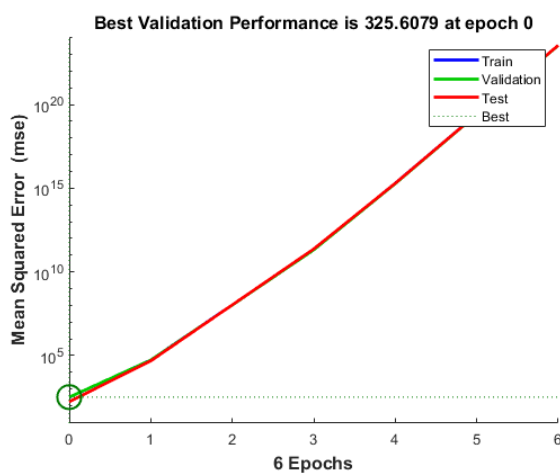


(d) Regression con traingd

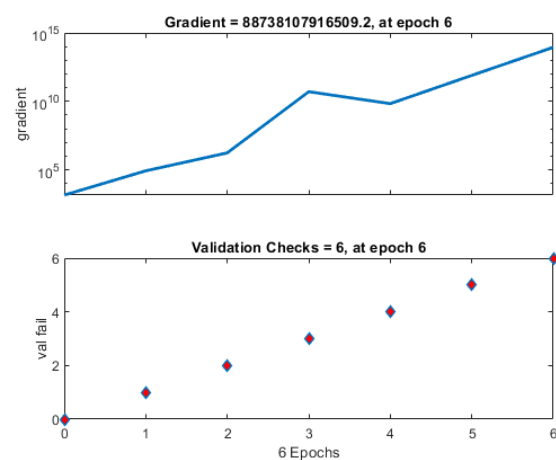
3.4. Descenso gradiente con división de datos 60/20/20 - traingd

Al comprobar los resultados de las gráficas con esta división de los datos destaca el cambio abrupto de los valores de los errores en el histograma y de las líneas de regresión anteriormente mencionados. Los valores medios pasan de estar alrededor de -40 a estar entre el 0 y el 20. Las líneas de regresión teniendo una pendiente inferior a 1 también son un indicativo de una subestimación de las predicciones, en contraposición a lo observado con la división 70/15/15. Esta traslación en los valores del error podría indicar que se están realizando divisiones de datos de manera incorrecta o sesgada. Por ejemplo, si los datos de prueba se recopilan de manera que se favorezcan ciertas clases o ejemplos particulares. También se puede observar una aparente dispersión mayor de los datos, aunque esta podría ser un efecto derivado de la escala utilizada en las gráficas variando

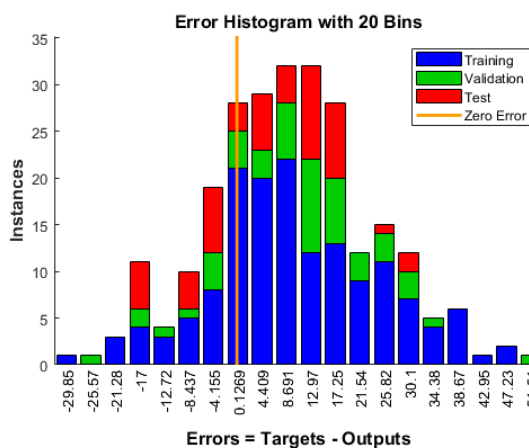
Figura 11: Gráficas con Descenso gradiente con división 60/20/20



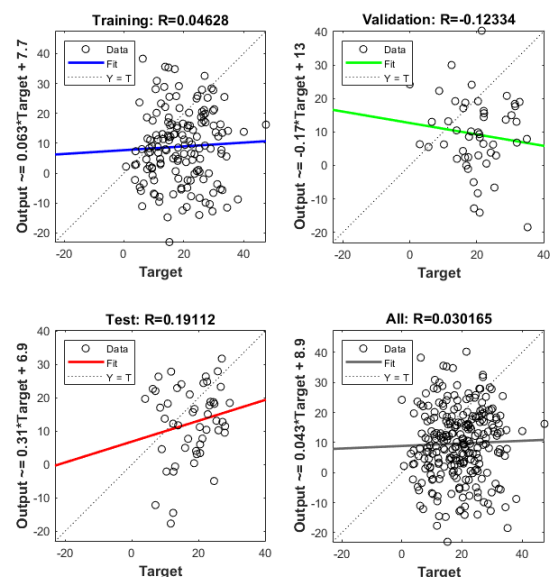
(a) Performance con traingd



(b) Training state con traingd



(c) Error Histogram con traingd

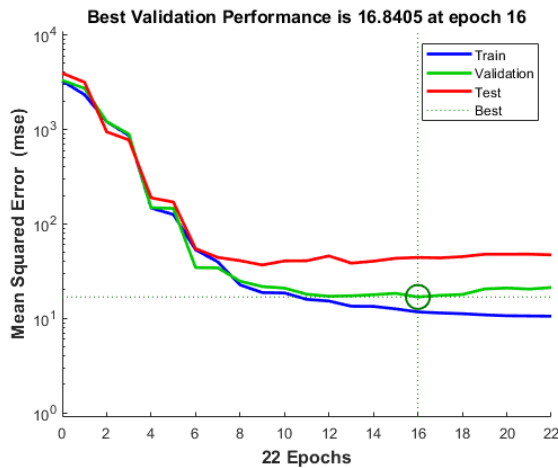


(d) Regression con traingd

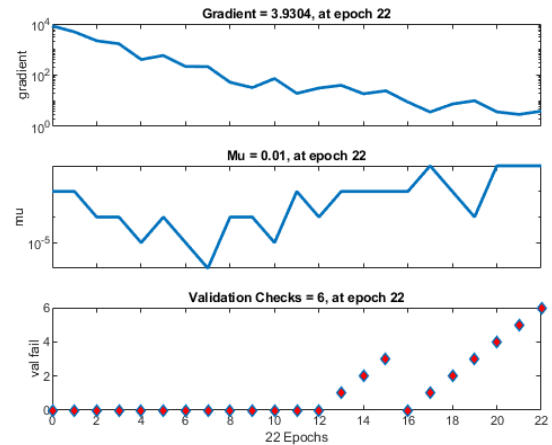
3.5. Levenberg-Marquardt con división de datos 70/15/15 - trainlm

Al observar las gráficas obtenidas de aplicar este método de entrenamiento, destacan el gran incremento en las épocas de entrenamiento (10 más que el mayor realizado con los otros métodos) y un descenso abrupto en los "validation checks" después de un número específico de épocas, en este caso, después de 16 épocas. Esto indica una mejora significativa en el rendimiento del modelo de aprendizaje durante ese período, además, coincide con la época de mejor rendimiento en el conjunto de validación

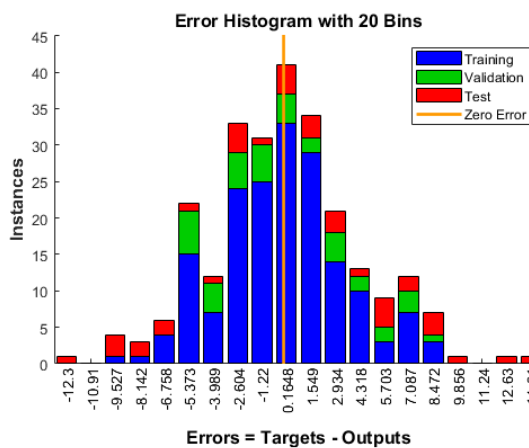
Figura 12: Gráficas con Levenberg-Marquardt con división 70/15/15



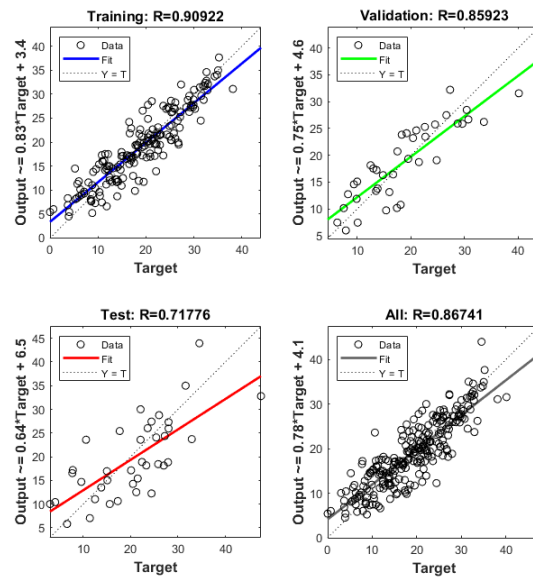
(a) Performance con trainlm



(b) Training state con trainlm



(c) Error Histogram con trainlm

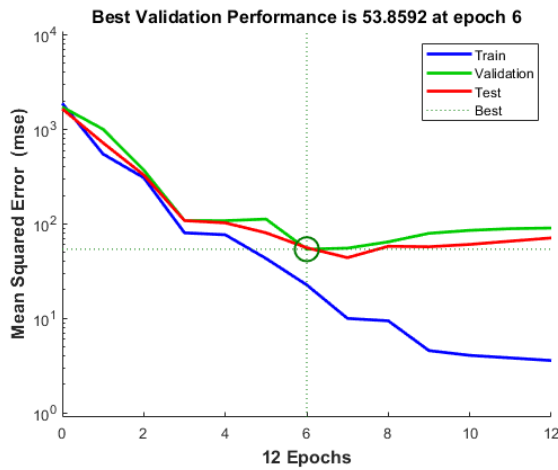


(d) Regression con trainlm

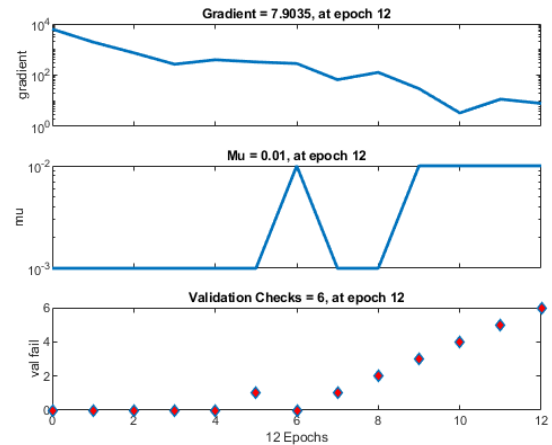
3.6. Levenberg-Marquardt con división de datos 60/20/20 - train

Al repetirse la ejecución anterior pero con una división diferente de los datos, el resultado es aparentemente peor pero más rápido, ya que el mejor rendimiento aparece después de 6 épocas, siendo mayor el grado de error comparado a la ejecución anterior. Podemos observar que a partir de esta época la red muestra signos de sobreajuste, por lo que podría intuirse que un grupo de datos de entrenamiento de mayor tamaño es más apropiado para redes entrenadas con este algoritmo.

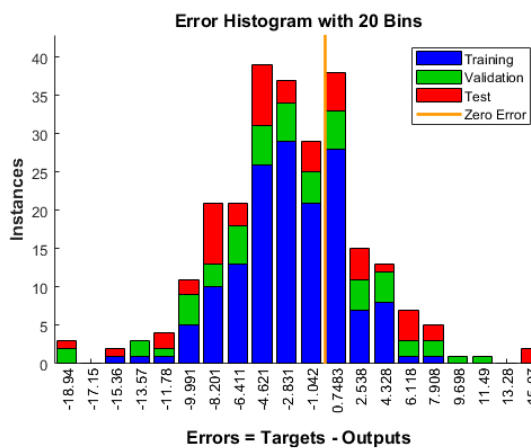
Figura 13: Gráficas con Levenberg-Marquardt con división 60/20/20



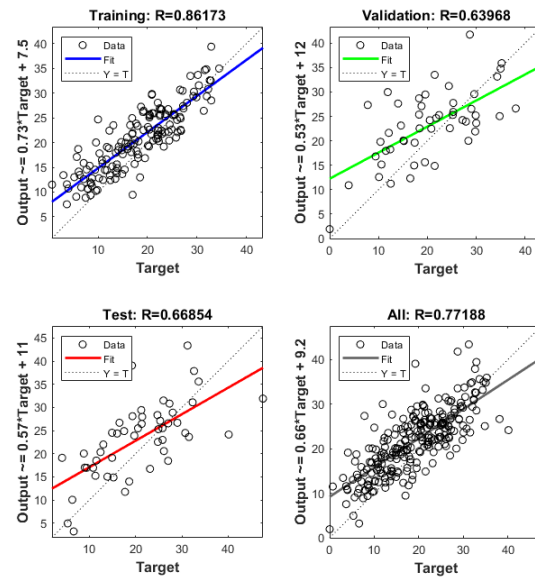
(a) Performance con trainlm



(b) Training state con trainlm



(c) Error Histogram con trainlm



(d) Regression con trainlm

4. Ejercicio 4. Clasificación.

La clasificación de patrones es una de las aplicaciones que dieron origen a las redes neuronales artificiales. Como en el caso anterior, la toolbox de redes neuronales de Matlab dispone de una red optimizada para la clasificación, `patternnet`, que analizaremos en este ejemplo. De estos resultados cabe destacar que no ha habido ningún falso negativo en las predicciones de validación.

```
1 % Carga de datos de ejemplo disponibles en la toolbox
2 [inputs,targets] = cancer_dataset;
3 % Creación de una red neuronal para el reconocimiento de patrones
4 hiddenLayerSize = 10;
5 net = patternnet(hiddenLayerSize);
6 % División del conjunto de datos para entrenamiento, validación y test
7 net.divideParam.trainRatio = 70/100;
8 net.divideParam.valRatio = 15/100;
9 net.divideParam.testRatio = 15/100;
10 % Entrenamiento de la red
11 [net,tr] = trainlm(net,inputs,targets);
12 % Prueba
13 outputs = net(inputs);
14 errors = gsubtract(targets,outputs);
15 performance = perform(net,targets,outputs)
16 % Visualización
17 view(net)
```

En lugar de las gráficas específicamente relacionadas con la aproximación de una función, en el caso de una tarea de clasificación, se ofrecen:

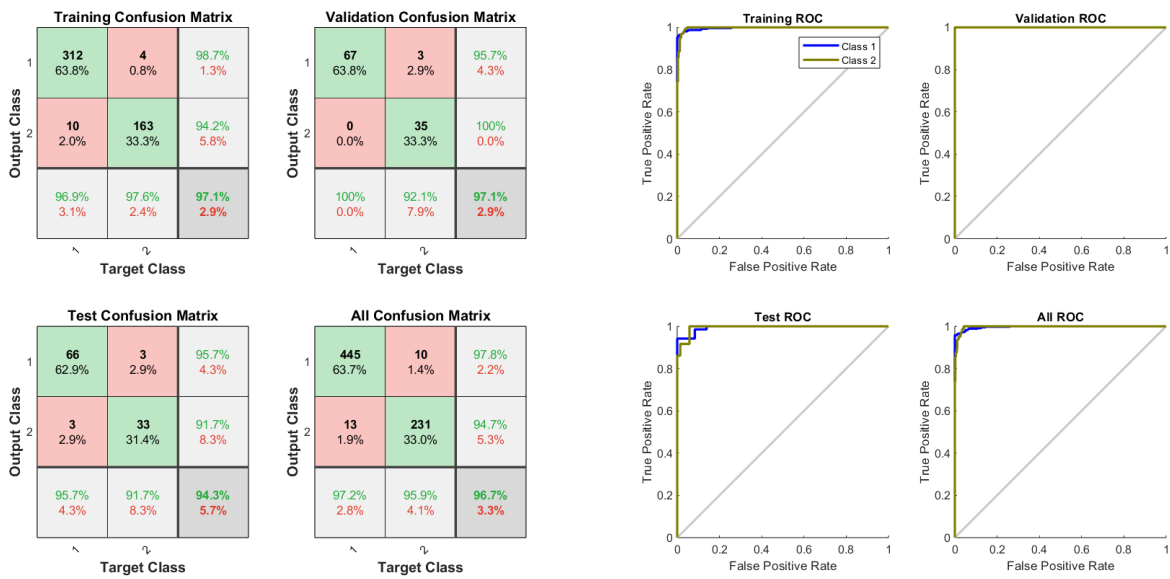
1. Confusion: matrices de confusión de los resultados.
2. Receiver Operating Characteristic: curvas ROC (característica operativa del receptor).

Pruebe este mismo script con el conjunto de datos `cancer_dataset`, y evalúe sus resultados. Estudie de nuevo la mejora que supone utilizar distintos métodos de entrenamiento y una división diferente de los datos (entrenamiento, validación y test).

4.1. Backpropagation

La matriz de confusión y gráfica de ROC obtenidas al entrenar la red con el algoritmo de backpropagation en el conjunto de datos cancer_dataset muestran la proporción de verdaderos positivos, verdaderos negativos, falsos positivos y falsos negativos. Los resultados obtenidos son relativamente buenos, ya que la precisión más baja es de un 95.7% y el área bajo la curva en la gráfica Receiver Operating Characteristic supone una gran parte (si no toda) del área total. Los resultados positivos se repiten con todos los métodos de entrenamiento probados, con ligeras variaciones en función de la división de los datos.

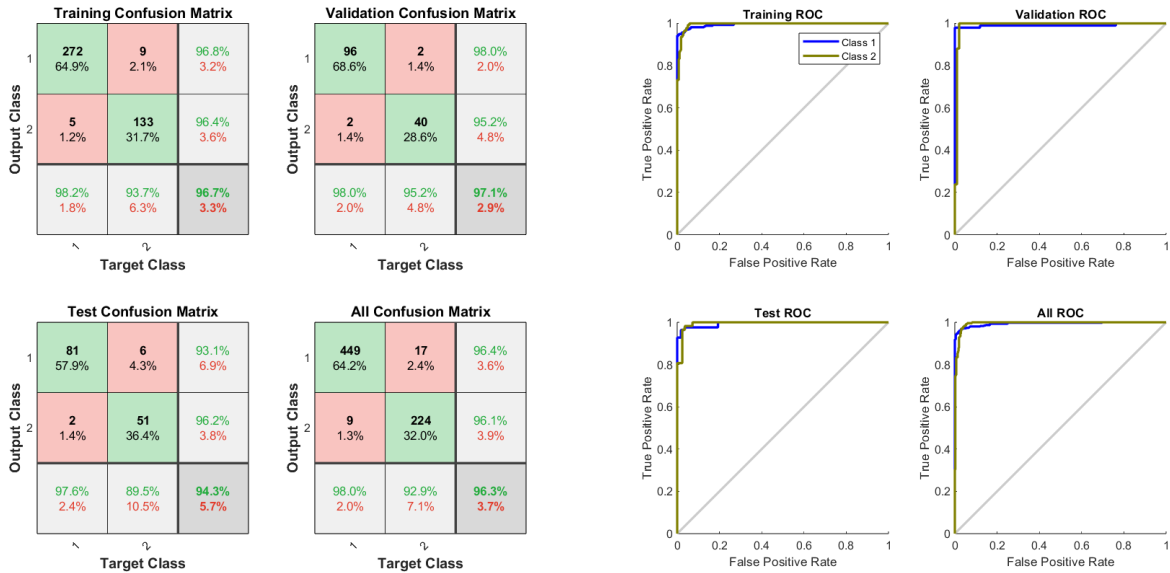
Figura 14: Gráficas con Backpropagation con división 70/15/15



4.2. Backpropagation con división de datos 60/20/20

Al cambiar la división de los datos se observa un resultado ligeramente peor en la matriz “All Confusion Matrix“, al igual que en la de validación. Ahora sí que hay falsos negativos en las predicciones de validación. Por lo demás los resultados son muy similares.

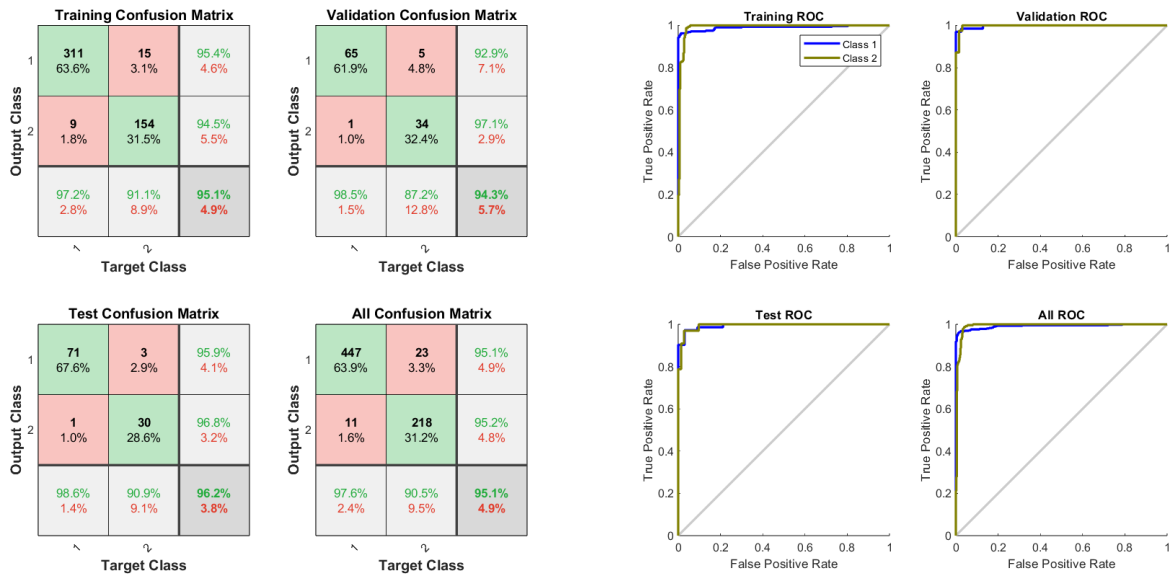
Figura 15: Gráficas con Backpropagation con división 60/20/20



4.3. Descenso gradiente

Las gráficas de la red entrenada con descenso gradiente muestran un resultado ligeramente peor a las entrenadas con backpropagation

Figura 16: Gráficas con Descenso gradiente con división 70/15/15



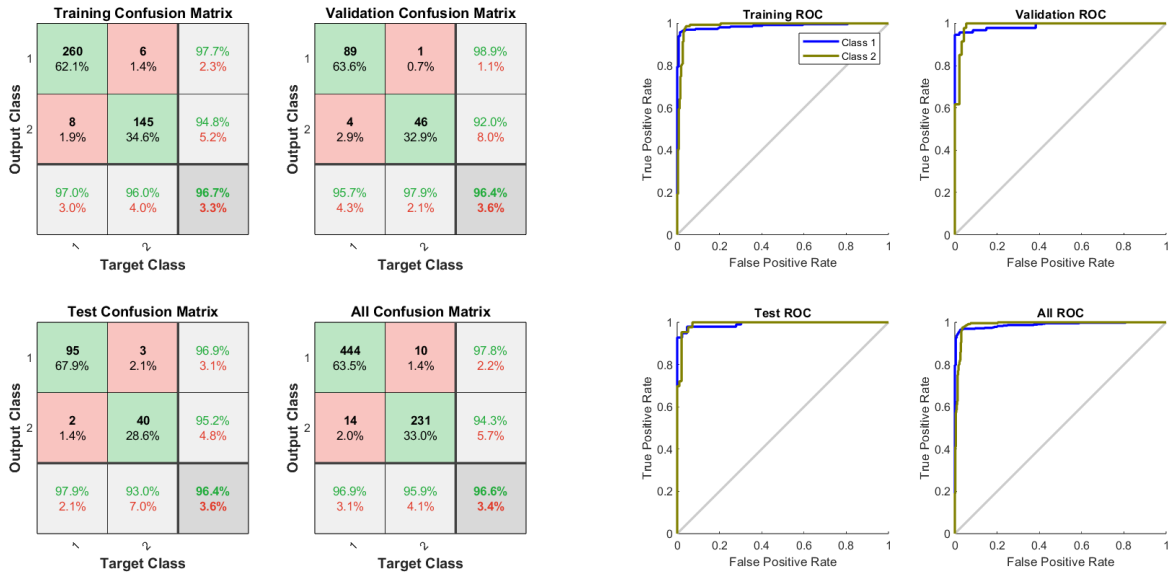
(a) Confusion con traingd

(b) Receiver Operating Charactersitic con traingd

4.4. Descenso gradiente con división de datos 60/20/20

Se ha repetido la ejecución anterior pero con una división diferente de los datos. El resultado en las predicciones de validación mejora en cuanto a precisión pero empeora en cuanto a sensibilidad. En conjunto resulta muy similar.

Figura 17: Gráficas con Descenso gradiente con división 60/20/20



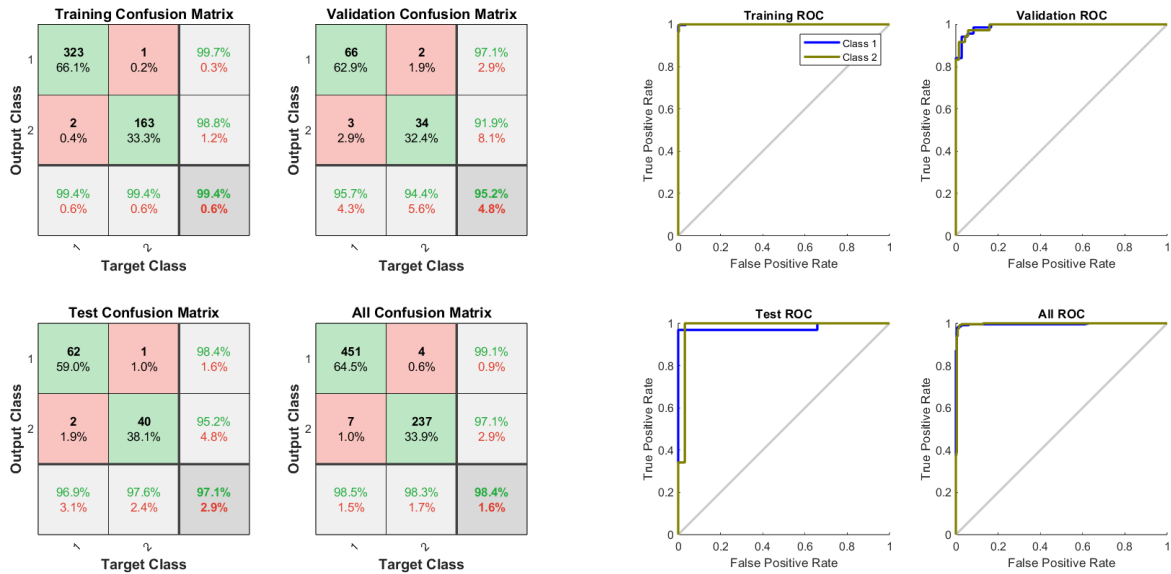
(a) Confusion con traingd

(b) Receiver Operating Charactersitic con traingd

4.5. Levenberg-Marquardt

La Matriz de confusión y las gráficas ROC muestran resultados muy similares con este algoritmo también. Aparentemente todos los métodos probados son bastante buenos en cuanto a clasificación.

Figura 18: Gráficas con Levenberg-Marquardt con división 70/15/15



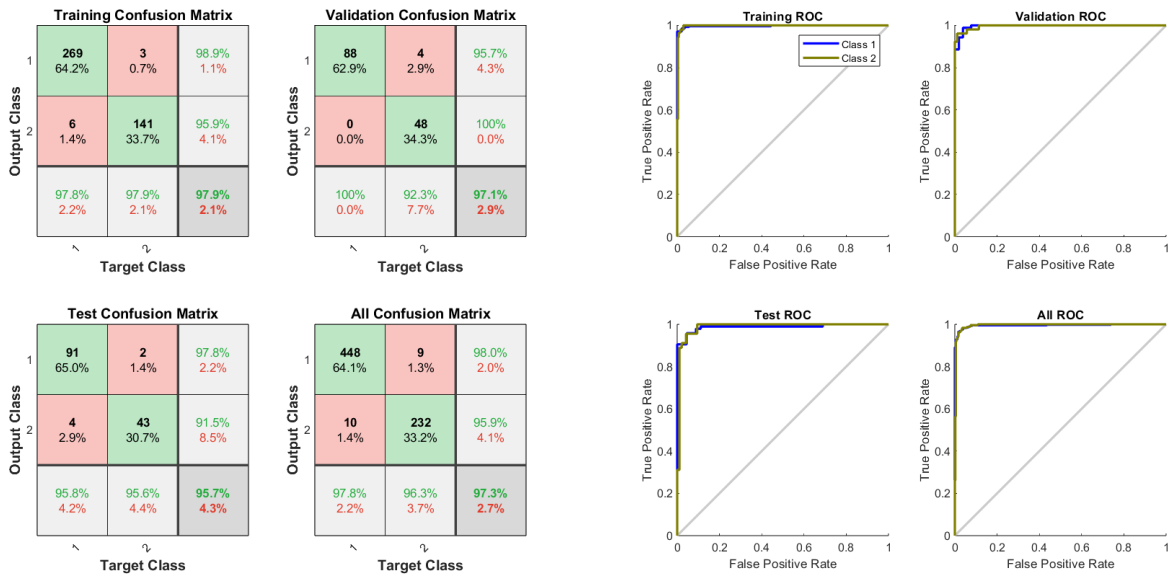
(a) Confusion con train

(b) Receiver Operating Characteristic con trainlm

4.6. Gráficas con Levenberg-Marquardt con división 60/20/20

Esta última matriz de confusión no presenta ningún falso negativo (al igual que backpropagation con 70/15/15) en las predicciones de validación a pesar del aumento en la cantidad de datos en este grupo respecto a la prueba anterior. Esto no representa un cambio significativo respecto al resto de pruebas (las cuales presentaban entre 0 y 4 falsos negativos en este área)

Figura 19: Gráficas con Levenberg-Marquardt con división 60/20/20



(a) Confusion con trainlm

(b) Receiver Operating Characteristic con trainlm

Parte II

Diseño de un control de posición mediante una red neuronal no recursiva.

5. Desarrollo

5.1. Esquema de Simulink

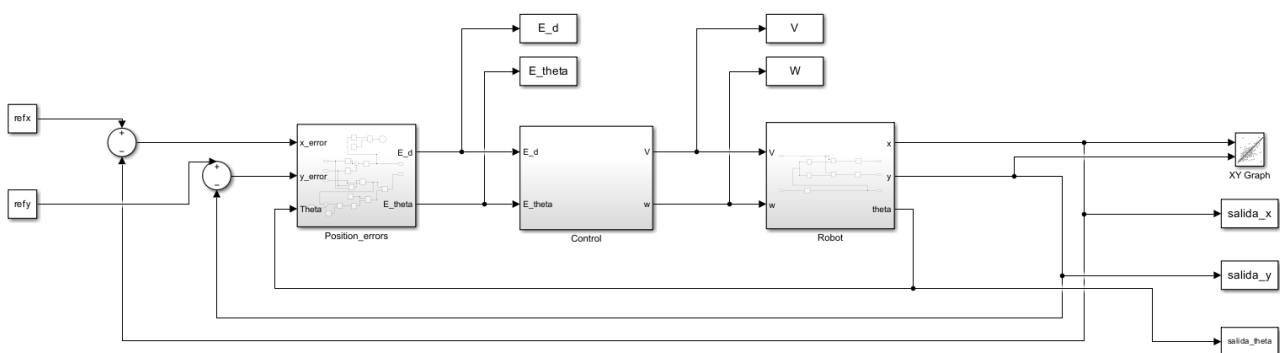


Figura 20: Esquema general Simulink.

5.2. Creación script para simular el diagrama.

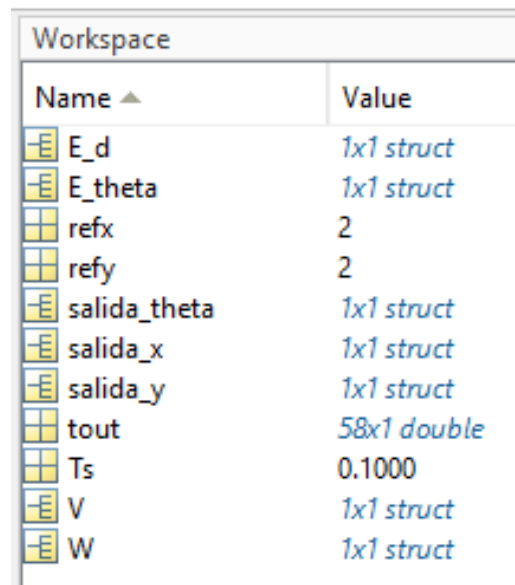
```

1 %Tiempo de muestreo
2 Ts=100e-3
3 % Referencia x-y de posicion
4 refx=2.0;
5 refy=2.0;
6 % Ejecutar Simulacion
7 sim('PositionControl.slx')

```

5.3. Ejecución script y comprobación de variables.

Podemos observar que se generan las variables que contienen las salidas y entradas del controlador y las salidas del robot durante la simulación correctamente.



Name ▲	Value
E_d	1x1 struct
E_theta	1x1 struct
refx	2
refy	2
salida_theta	1x1 struct
salida_x	1x1 struct
salida_y	1x1 struct
tout	58x1 double
Ts	0.1000
V	1x1 struct
W	1x1 struct

Figura 21: Variables.

5.4. Trayectoria del robot.

```
1 % Mostrar
2 x=salida_x.signals.values;
3 y=salida_y.signals.values;
4 figure;
5 plot(x,y);
6 grid on;
7 hold on;
```

Podemos ver la trayectoria que sigue el robot en la siguiente gráfica:

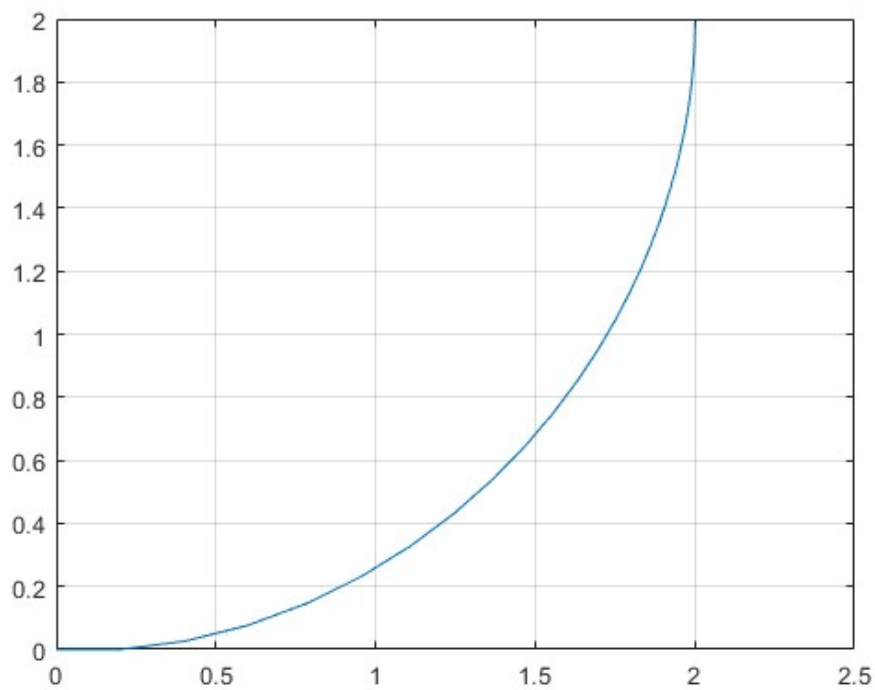


Figura 22: Trayectoria del robot.

5.5. Generar N posiciones aleatorias, simular y guardar en variables.

```

1  % Generar N posiciones aleatorias, simular y guardar en variables
2  N=30
3  E_d_vec=[];
4  E_theta_vec=[];
5  V_vec=[];
6  W_vec=[];
7  for i=1:N
8      refx=10*rand-5;
9      refy=10*rand-5;
10     sim('PositionControl.slx')
11     E_d_vec=[E_d_vec;E_d.signals.values];
12     E_theta_vec=[E_theta_vec;E_theta.signals.values];
13     V_vec=[V_vec; V.signals.values];
14     W_vec=[W_vec; W.signals.values];
15 end
16 inputs=[E_d_vec'; E_theta_vec'];
17 outputs=[V_vec'; W_vec'];

```

5.6. Entrenamiento de red neuronal con 10 neuronas en la capa oculta.

```

1  % Entrenar red neuronal con 10 neuronas en la capa oculta
2  net = feedforwardnet([10]);
3  net = configure(net,inputs,outputs);
4  net = train(net,inputs,outputs);

```

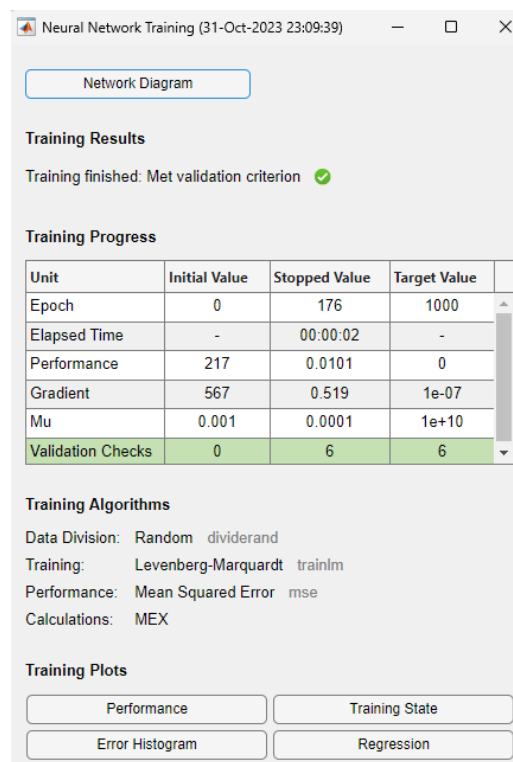


Figura 23: Resultados del entrenamiento.

5.9. Comparación.

Se ha hecho el siguiente código para estudiar las diferencias entre los comportamientos y el error que conlleva.

```

1  % Tiempo de muestreo
2  Ts = 100e-3;
3
4  % Número de simulaciones a realizar
5  num_simulations = 4;
6
7  % Inicializa vectores para almacenar los errores
8  errors = zeros(num_simulations, 1);
9
10 for i = 1:num_simulations
11     % Genera valores aleatorios para refx y refy
12     refx = 10 * rand - 5;
13     refy = 10 * rand - 5;
14
15     % Ejecuta Simulación 1 (PositionControl.slx)
16     sim('PositionControl.slx');
17
18     % Obtiene las trayectorias
19     x = salida_x.signals.values;
20     y = salida_y.signals.values;
21
22     % Ejecuta Simulación 2 (PositionControlNet.slx)
23     sim('PositionControlNet.slx');
24
25     % Obtiene las trayectorias de la red
26     x_net = salida_x_net.signals.values;
27     y_net = salida_y_net.signals.values;
28
29     % Asegura que las trayectorias tengan la misma longitud
30     min_length = min(length(x), length(x_net));
31     x = x(1:min_length);
32     y = y(1:min_length);
33     x_net = x_net(1:min_length);
34     y_net = y_net(1:min_length);
35
36     % Calcula el error entre las trayectorias
37     error = sqrt((x - x_net).^2 + (y - y_net).^2);
38
39     % Almacena el error en el vector de errores
40     errors(i) = mean(error);
41
42     % Muestra las trayectorias
43     figure;
44     plot(x, y);
45     hold on;
46     plot(x_net, y_net);
47     hold off;
48     grid on;
49     title(sprintf('Simulación %d: refx=%.2f, refy=%.2f', i, refx, refy));
50     legend('PositionControl', 'PositionControlNet');
51 end
52
53 % Calcula el error promedio de todas las simulaciones
54 average_error = mean(errors);
55
56 % Muestra el error promedio
57 fprintf('Error promedio entre las trayectorias: %.4f\n', average_error);
58
59
60
61

```

Podemos observar que las trayectorias son bastante parecidas y los errores no son demasiado grandes en las siguientes gráficas

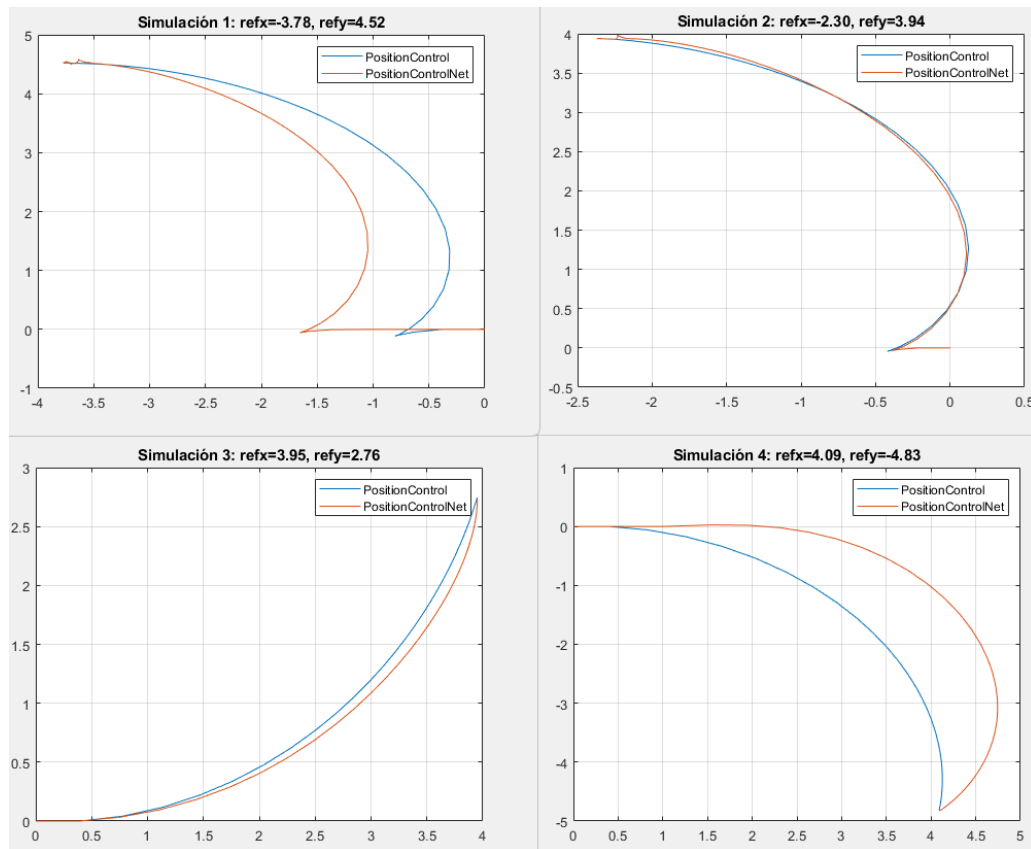


Figura 26: Gráficos para comparar trayectorias entre los dos esquemas.

Tras varias pruebas el error medio varía entre los siguientes valores:

```
Error promedio entre las trayectorias: 0.8672
>> compararComportamiento
Error promedio entre las trayectorias: 0.1258
>> compararComportamiento
Error promedio entre las trayectorias: 0.1403
>> compararComportamiento
Error promedio entre las trayectorias: 0.1500
>> compararComportamiento
Error promedio entre las trayectorias: 0.0995
>> compararComportamiento
Error promedio entre las trayectorias: 0.1655
>> compararComportamiento
Error promedio entre las trayectorias: 0.1338
>> compararComportamiento
Error promedio entre las trayectorias: 0.0616
>> compararComportamiento
Error promedio entre las trayectorias: 0.0736
>> compararComportamiento
Error promedio entre las trayectorias: 0.0298
```

Figura 27: Errores medios de varias ejecuciones.

Podemos concluir que los errores no son demasiado significativos y que están en un rango esperado.