
PRÁCTICA 1: IDENTIFICACIÓN Y CONTROL NEURONAL

Pablo Acereda García

Department of Computer Science
University of Alcala
28805 - Alcalá de Henares, Madrid, Spain
pablo.acereda@edu.uah.es

Laura Pérez Medeiro

Department of Computer Science
University of Alcala
28805 - Alcalá de Henares, Madrid, Spain
l.perez@edu.uah.es

September 12, 2020

Introducción

El objetivo de esta práctica es aprender el funcionamiento de identificación y control neuronal dentro del entorno de MATLAB. Para ello, se estudian diferentes tipos de redes neuronales (perceptron, fitnet o patternnet); además de diferentes métodos de entrenamiento (trainlm, trainbr, trainbfg, trainrp). Cada uno de los ejercicios ha sido cumplimentados en ficheros separados.

En cada uno de los ficheros se han incluido comandos que permiten el limpiado de la información cacheada además de la salida por pantalla de ejecuciones anteriores.

Se debe tener en cuenta que los comentarios presentes en el código de los archivos adjuntos no se encuentran presentes en los diferentes *snippets* de este documento. Con este motivo, si se desea tener una mayor claridad del funcionamiento del código se insta al lector a acceder a dichos ficheros.

Parte 1

Ejercicio 1

El ejercicio busca el desarrollo de un clasificador neuronal mediante un perceptron simple que clasifique los datos expuestos. El Perceptron es una función de tipo lineal, que puede resultar indicada cuando solo hay 2 clases dentro del conjunto de datos. Por este motivo, este tipo de red neuronal no es perfectamente funcional cuando hay diferentes clases en este caso 4.

Los datos a los que se ha hecho referencia con anterioridad cuentan con 2 variables que aportan información: x_1 y x_0 ; y 4 clases diferentes: desde 0 hasta 3. La codificación en MATLAB es la siguiente - [1]:

La capa de entrada del clasificador contará con una neurona de entrada, al igual que la salida¹ El número de neuronas es el elegido por defecto al generar el clasificador neuronal - [2].

Tras la ejecución de la red neuronal, se procede a representar los datos. Para poder representar los datos con mayor comodidad se utiliza la función *gscatter()*, que representa los puntos solicitados por clases - [3].

Los autores son conscientes de la existencia de las funciones *plotpv* y *plotpc*, pero debido al formato de la configuración inicial del set de datos no se han podido utilizar estas funciones.

La clasificación vuelve a realizarse tras la inclusión de un nuevo dato: $[0.0 \ -1.5]$ y de clase 3.

¹Para ver la configuración completa se recomienda realizar la ejecución del archivo adjunto.

Listing 1 Conjunto de datos (2 variables, 4 clases).

```
X = {[ 0.1; 1.2] ...
      [ 0.7; 1.8] ...
      [ 0.8; 1.6] ...
      [ 0.8; 0.6] ...
      [ 1.0; 0.8] ...
      [ 0.3; 0.5] ...
      [ 0.0; 0.2] ...
      [-0.3; 0.8] ...
      [-0.5; -1.5] ...
      [-1.5; -1.3],...
      [ 0.0; -1.5]
    };

Y = {2 2 2 0 0 3 3 3 1 1 ...
      3
    };
```

Listing 2 Neural Network definition.

```
net = configure(perceptron, X, Y);

net.biasConnect = 0;

net.IW{1, 1} = [1 -0.8];

net = adapt(net, X, Y)
view(net);

w = net.iw{1, 1} % w = [-1.1 3.8]

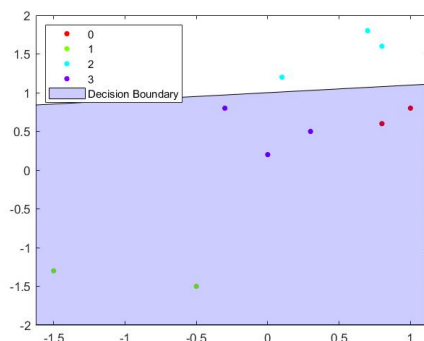
a = net(X)
err = cell2mat(a) - cell2mat(Y)
```

Listing 3 Perceptron classification representation.

```
figure(1)
gscatter(X(:, 1), X(:, 2), Y)
hold on

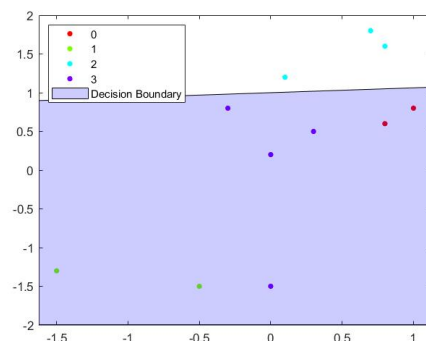
m = - w(1) / w(2);
x = -5 : 1 : 5;
y = m .* x + net.biasConnect;
a = area(x, y, -2, 'FaceColor', 'b');
a.FaceAlpha = 0.2;
hold off

legend('0', '1', '2', '3', 'Decision Boundary', 'Location', 'northwest')
```



(a) Datos Originales

2



(b) Datos Extendidos

Como cabía esperarse, ninguno de los clasificadores da la respuesta correcta al problema. Esto es principalmente debido a que existen un total de 4 clases. Al tratarse de un perceptron simple, este no puede realizar la división correcta de las mismas (problema XOR). Sin embargo, el clasificador consigue separar la clase 2 del resto de clases.

Puede observarse en las figuras 1a y 1b que la inclusión de un simple dato puede cambiar la pendiente del separador de clases.

Ejercicio 2

En este ejercicio, se usa la red *finet*, cuyo objetivo final es aproximar la función $f = \text{sinc}(t)$. El código que realiza este comportamiento ya ha sido realizado por el escritor de la práctica, por lo que no se incluirá el código original. Por el contrario, al solicitarse como objetivo final del ejercicio el aprender como afecta al resultado final el cambiar el método de entrenamiento (se solicita probar 4 diferentes) y el número de neuronas de la capa oculta, sí que se incluirá ese script como *snippet*.

En primer lugar, se procede a mostrar el resultado de la ejecución del script que varía el número de neuronas de la capa oculta. Como método de entrenamiento, se emplea *trainrp* - [2].

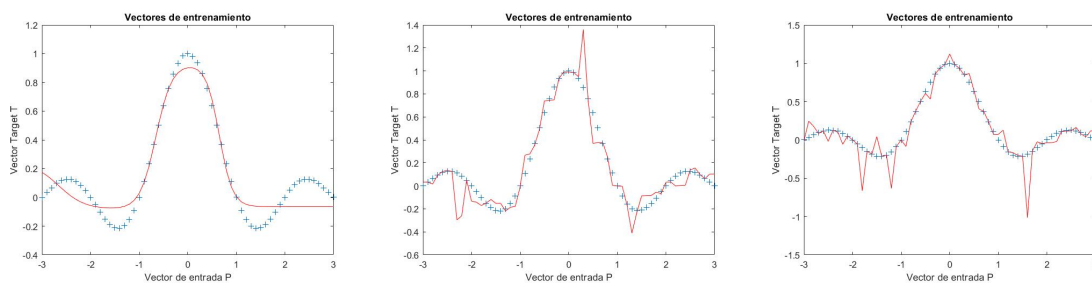


Figure 2: Entrenamiento cambiando número de neuronas de la capa oculta.

El emplear un número demasiado elevado de neuronas en la capa oculta puede suponer que la red entrenada se adapte en demasía al problema dado, y por lo tanto, su tasa de fallo aumente - en lugar de disminuir - al entrar nuevos datos.

Como se ha mencionado anteriormente, además de modificar el número de neuronas de la capa oculta, también se examinan diferentes métodos de entrenamiento para comprobar cual es el que más se adecua. Dentro de las diferentes opciones que permite MATLAB, se han elegido las siguientes:

trainbr Bayesian Regularization

trainrp Resilient Backpropagation

trainoss One Step Secant

traingd Gradient Descent

El código adaptado para mostrar los resultados de los cuatro métodos de entrenamiento en una sola figura es el siguiente - [4].

Estos métodos de entrenamiento han sido sometidos a diferentes tamaños de capa oculta, para comprobar cual es más adecuado para aproximarse a la función $f = \text{sinc}(t)$. Se ha utilizado un tamaño de capa oculta de 4 y de 200

Una vez más, al aumentar el número de neuronas de la capa oculta las funciones de entrenamiento quedan demasiado adaptadas al problema, por lo que al ser expuestas a nuevos datos (en este caso el set de pruebas) su número de fallos aumenta en lugar de verse reducido.

Puede observarse en las figuras 3a y 3b que las funciones que menos se han visto afectadas por este cambio son *trainbr* y *trainrp*. Observando resultados individuales, la función de entrenamientotrainbr es la más adecuada para este problema - [4].

Ejercicio 3

Acorde al enunciado, el objetivo de esta práctica es estudiar las herramientas que implementa MATLAB para el diseño y prueba de redes neuronales. Para ello, se debe ejecutar el código ya implementado por los escritores de la práctica, por lo que no se ve como algo necesario incluir un *snippet* del código.

Listing 4 Fitnet con diferentes métodos de entrenamiento y tamaños de la capa oculta.

```

t = -3:.1:3;
F = sinc(t) + .001 * randn(size(t));

title('Vectores de entrenamiento');

xlabel('Vector de entrada P');
ylabel('Vector Target T');

plot(t, F, '+');
hold on

hiddenLayerSize = 200;

trainfun    = ["trainbr", "trainrp", "trainoss", "traingd"];
trainfuncol = ['g', 'm', 'c', 'r'];

for i = 1 : 1 : length(trainfun)

    net = fitnet(hiddenLayerSize, trainfun(i));

    net.divideParam.trainRatio = 70 / 100;
    net.divideParam.valRatio   = 15 / 100;
    net.divideParam.testRatio  = 15 / 100;

    net = train(net, t, F);
    Y = net(t);

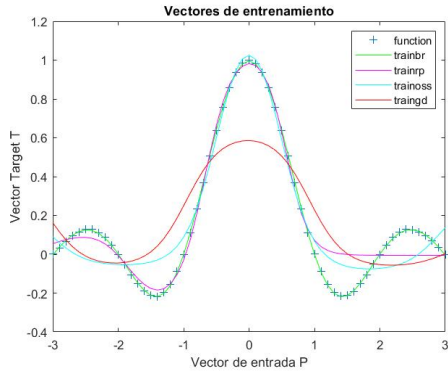
    plot(t, Y, strcat(trainfuncol(i), '-'));
    hold on;
end

hold off;
title('Vectores de entrenamiento');
xlabel('Vector de entrada P');
ylabel('Vector Target T');
legend(['function', trainfun]);

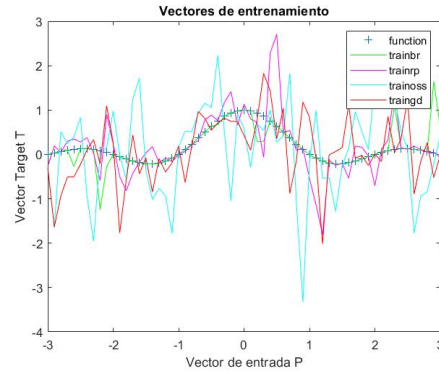
```

Por otro lado, el código ha sido editado para poder mostrar las gráficas *plotperform*, *plottrainstate*, *plotrrhist*, *plotregression* y *plotfit*. Estas gráficas pueden ser accedidas mediante la herramienta *nntraintool* - a la cual se accede cuando se entrena una red neuronal - o mediante las llamadas a las funciones correspondientes - [5].

El script resultante ha sido ejecutado con el conjunto de datos *bodyfat_dataset*. De nuevo, se solicita estudiar los resultados de entrenamiento al entrenarse con diferentes metodologías o al cambiar las proporciones de datos de entrenamiento-validación-test.



(a) Todas las funciones. Número de neuronas de la capa oculta = 4.



(b) Todas las funciones. Número de neuronas de la capa oculta = 200.

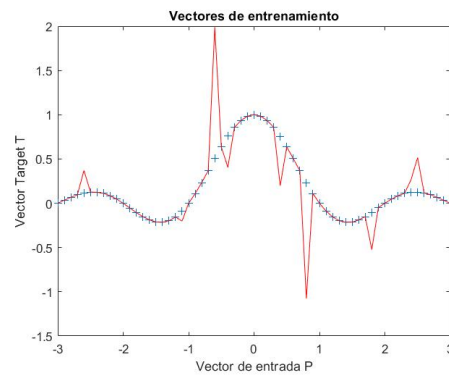
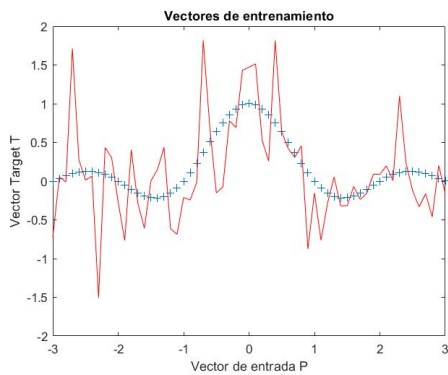
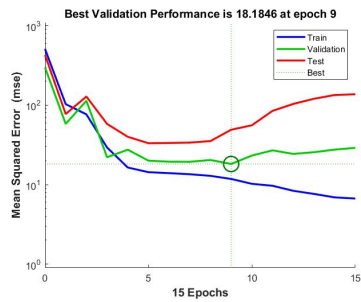
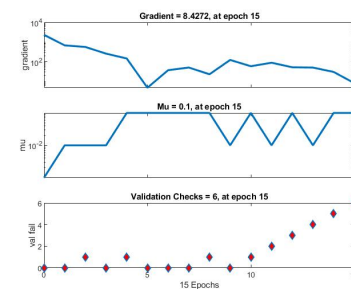


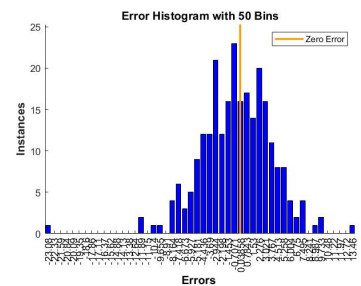
Figure 4: *trainrp* y *trainbr* respectivamente, con un tamaño de capa oculta de 200 neuronas.



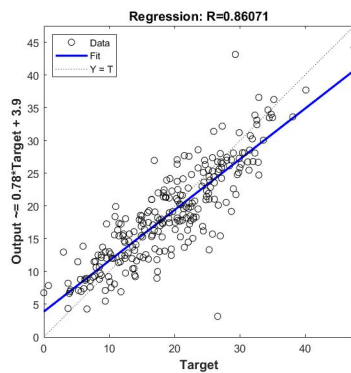
(a) Rendimiento



(b) Estado de Entrenamiento



(c) Histograma de Errores



(d) Regresión

Listing 5 Diseño y prueba de redes neuronales.

```
[inputs, targets] = bodyfat_dataset;

hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize, 'trainbr');

net.divideParam.trainRatio = 60/100;
net.divideParam.valRatio   = 20/100;
net.divideParam.testRatio  = 20/100;

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

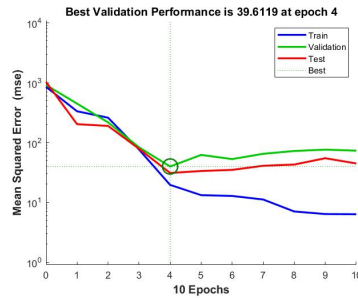
outputs    = net(inputs);
errors     = gsubtract(outputs, targets);
performance = perform(net, targets, outputs)

view(net)

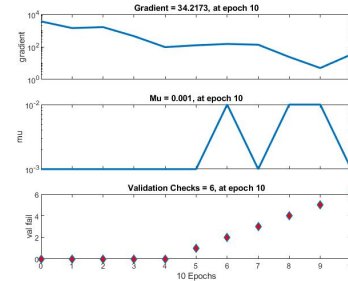
figure(1)
plotperform(tr)
figure(2)
plottrainstate(tr)
figure(3)
ploterrhist(errors, 'bins', 50)
figure(4)
plotregression(targets, outputs, 'Regression')
figure(5)
plotfit(net, inputs, targets)
```

Los resultados demuestran que el entrenamiento de la red neuronal aumenta en precisión conforme avanzan las épocas (*epoch*); la validación encuentra su error mínimo durante el periodo 9, momento a partir del cual solo aumenta; en el testing, el error aumenta tras durante el testeo, llegando a alcanzar más de 10^2 (un valor demasiado alto).

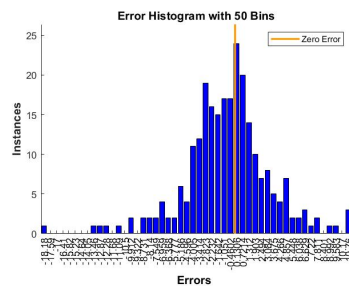
Para solucionar este problema, se puede probar una distribución de los datos diferente. En este caso se ha escogido 60/20/20 como valores para el entrenamiento (*train*), la validación (*validation*) y el testeo (*test*), respectivamente, de manera arbitraria.



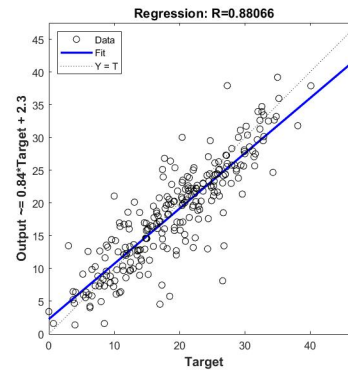
(a) Rendimiento



(b) Estado de Entrenamiento



(c) Histograma de Errores



(d) Regresión

Tras la nueva división de los datos, empleando el mismo algoritmo de entrenamiento, se consigue que los errores (teste, validación y entrenamiento) disminuyen, aunque la validación y el test siguen teniendo una cierta predisposición a incrementar levemente superada la cuarta época.

La función lineal mostrada en la regresión pasa a ser de $Output = 0.78 * Target + 3.9$ a valer $Output = 0.84 * Target + 2.3$. Siendo, aparentemente, la segunda una ecuación más fiel a los datos representados.

Para realizar un estudio más exhaustivo, se han empleado diferentes métodos de entrenamiento.

trainbr Bayesian Regularization

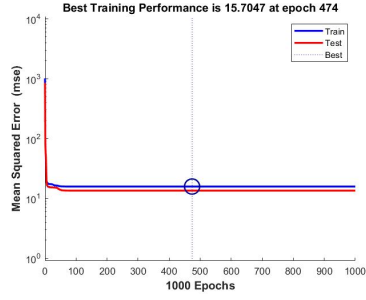
trainrp Resilient Backpropagation

trainoss One Step Secant

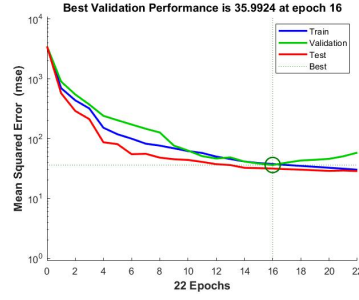
traingd Gradient Descent

A continuación se especificará y comparará cada uno de los grupos de gráficas obtenidos a lo largo de la ejecución del *script*

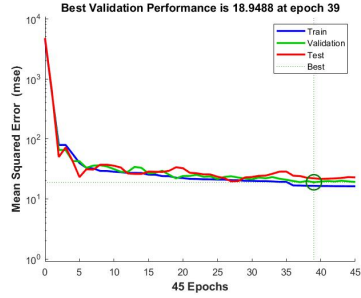
Referente al rendimiento, la función bayesiana ha conseguido los mejores resultados, consigue el mejor rendimiento de entre las cuatro gráficas. Por otro lado, la función *trainrp* consigue un rendimiento que empeora conforme se supera la época 16. *trainoss*, con un rendimiento entre las dos anteriores, posee un MSE muy similar entre los conjuntos de datos *train*, *validation* y *test*. Por último, el método de entrenamiento del gradiente, consigue el peor resultado, empeorando conforme recibe más entrenamiento.



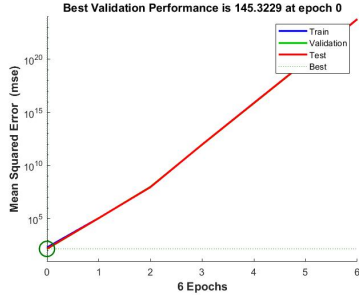
(a) Rendimiento *trainbr*



(b) Rendimiento *trainrp*

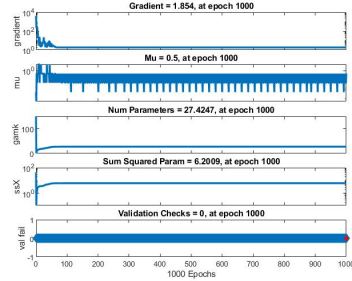


(c) Rendimiento *trainoss*

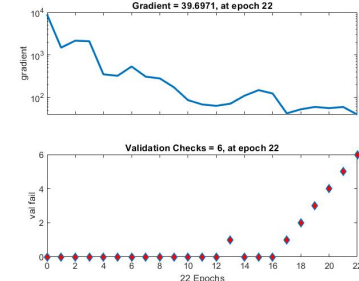


(d) Rendimiento *traingd*

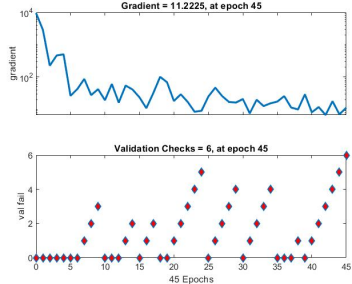
El valor del gradiente ² se muestra una vez más incremental para *traingd*³. En el caso de *trainbr*, el gradiente se mantiene constante a partir de cierto punto, lo que hace que el entrenamiento dure 1000 *epochs*; esto se debe a la manera en la que está codificado el algoritmo, la parada por validación se encuentra desactivada por defecto. El entrenamiento de *trainrp* y *trainoss* discurre de manera normal, alcanzando los mínimos locales en los valores 11.2225 y 39.6971, respectivamente.



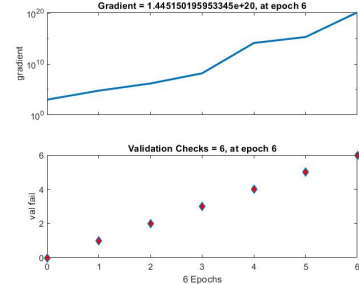
(a) Estado de Entrenamiento *trainbr*



(b) Estado de Entrenamiento *trainrp*



(c) Estado de Entrenamiento *trainoss*

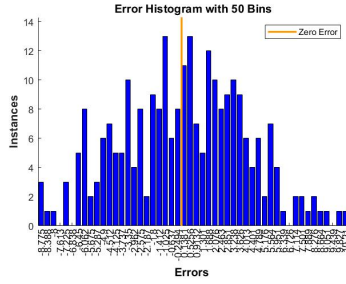


(d) Estado de Entrenamiento *traingd*

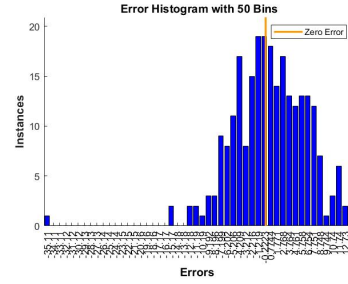
²El valor del gradiente por propagación inversa se encuentra en escala logarítmica. El valor en la parte superior de la gráfica es el valor para el cual se ha alcanzado el punto mínimo de un mínimo local.

³MATLAB detiene el entrenamiento del modelo cuando el valor de MSE - Mean Square Error - resulta incremental 6 veces de manera consecutiva, también conocido como *validation fails*.

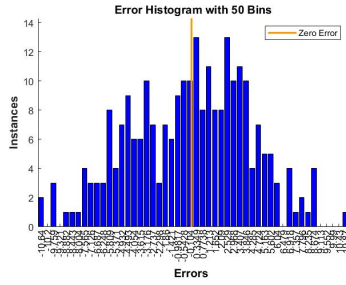
El histograma es una gráfica que representa el error cometido al estimar los valores empleados para el entrenamiento de la red neuronal, por tanto, cuanto más esté centrada la distribución de los errores de manera cercana al indicador *Zero Error*, mejor predicción dará el modelo. Atendiendo a este criterio, los extremos de *trainrp* están prácticamente libres, y los valores cercanos a $Error = 0$ están más poblados. Por esto mismo, se sabe que la función del gradiente es la que ha obtenido peores resultados, por tener su mayor concentración de errores más cercana a 1 que a 0. *trainoss* tiene, con alguna excepción, forma de pirámide escalonada, lo que también sugiere un entrenamiento exitoso. *trainbr* posee una forma bastante irregular, pero los errores que más se repiten se encuentran principalmente en la zona central, sugiriendo un resultado mejor que el resto de algoritmos de entrenamiento.



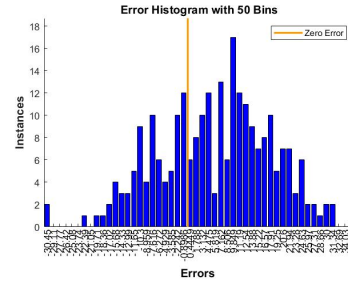
(a) Histograma de Errores *trainbr*



(b) Histograma de Errores *trainrp*

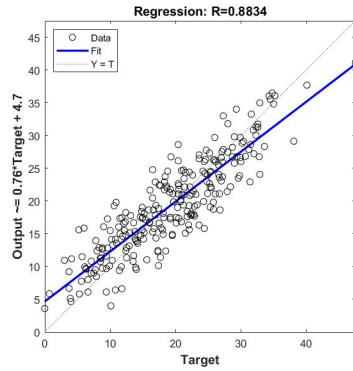


(c) Histograma de Errores *trainoss*

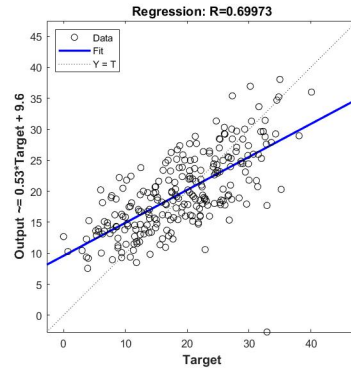


(d) Histograma de Errores *traingd*

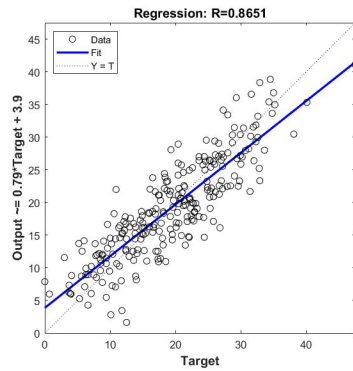
En la parte superior de cada gráfica aparece un valor, R , el cual indica lo óptima que es la regresión, siendo 1 el valor más óptimo. No hace falta observar mucho los resultados obtenidos para percatarse de que el método de entrenamiento más óptimo es en realidad la regularización bayesiana, tal y como se venía adelantando a lo largo del ejercicio.



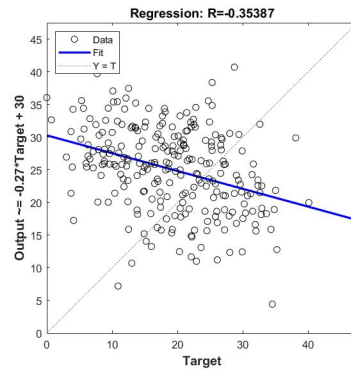
(a) Regresión *trainbr*



(b) Regresión *trainrp*



(c) Regresión *trainoss*



(d) Regresión *trainingd*

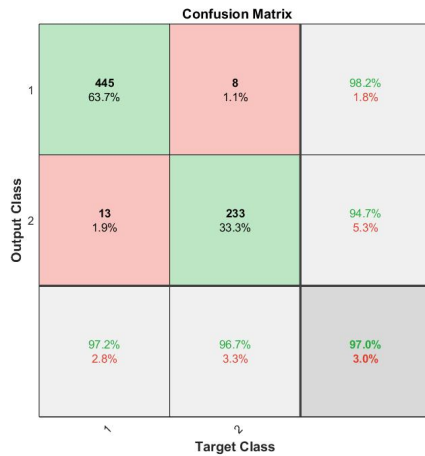
Ejercicio 4

Se busca aprender el funcionamiento de *patternnet*, una red neuronal optimizada para la clasificación de patrones. Al código otorgado por los profesores de la asignatura solamente se han añadido las funciones que permitan mostrar los resultados del entrenamiento - [6].

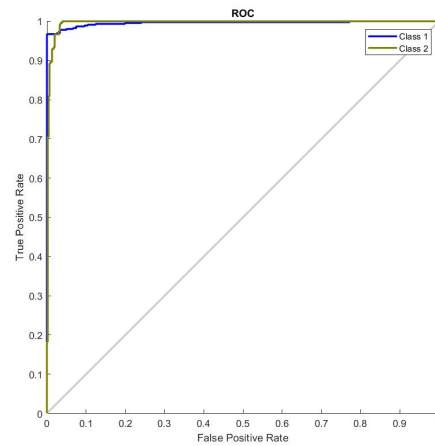
Listing 6 Clasificación de patrones.

```
figure(1)
plotconfusion(targets, outputs)
figure(2)
plotroc(targets, outputs)
```

Tras la ejecución del *script* adaptado para los datos de *cancer_dataset* se obtienen las siguientes gráficas.



(a) Matriz de confusión



(b) Receiver Operating Characteristic

En 11a, las filas indican el valor que se ha predicho; las columnas indican el valor real. Por tanto, la diagonal corresponde a los valores que han sido predichos de manera correcta. Interpretando el resultado obtenido, se sabe que el 97% de las estimaciones han sido correctas.

En la segunda gráfica (11b), es una métrica empleada para comprobar la calidad de los clasificadores. Cuanto más se acerquen las funciones a 1 en el eje de ordenadas. Por este mismo motivo, se conoce que los clasificadores empleados para este set de datos son una buena opción.

Para realizar un estudio más acertado de estas herramientas, se han empleado diferentes métodos de entrenamiento (los mismos empleados para los ejercicios 2 y 3 de este mismo apartado).

Confusion Matrix		
Output Class	1	2
	458 65.5%	3 0.4%
Output Class	1	2
	0 0.0%	238 34.0%
Target Class		
1	100% 0.0%	98.8% 1.2%
2	99.6% 0.4%	99.6% 0.4%

(a) Matriz de confusión *trainbr*

Confusion Matrix		
Output Class	1	2
	446 63.8%	41 5.9%
Output Class	1	2
	12 1.7%	200 28.6%
Target Class		
1	97.4% 2.6%	83.0% 17.0%
2	92.4% 7.6%	92.4% 7.6%

(b) Matriz de confusión *trainrp*

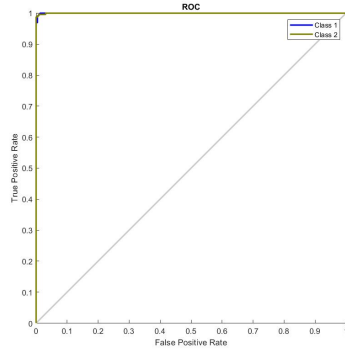
Confusion Matrix		
Output Class	1	2
	446 63.8%	11 1.6%
Output Class	1	2
	12 1.7%	230 32.9%
Target Class		
1	97.4% 2.6%	95.4% 4.6%
2	96.4% 3.3%	96.4% 3.3%

(c) Matriz de confusión *trainoss*

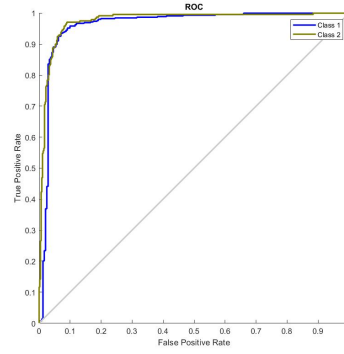
Confusion Matrix		
Output Class	1	2
	446 63.8%	13 1.9%
Output Class	1	2
	12 1.7%	228 32.6%
Target Class		
1	97.4% 2.6%	94.6% 5.4%
2	96.4% 3.6%	96.4% 3.6%

(d) Matriz de confusión *traingd*

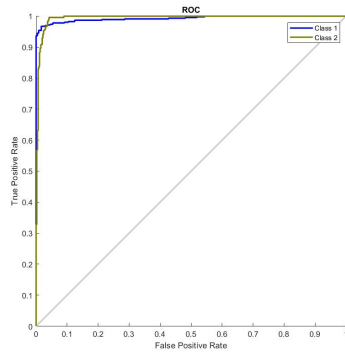
De entre todos los resultados obtenidos, el más acertado ha sido el de el algoritmo de entrenamiento *trainbr*, con un 99.6% de acierto sobre el set. El peor algoritmo que se podría emplear sobre este set es *trainrp*, que a pesar de poseer el peor resultado tiene un 92.4% de acierto.



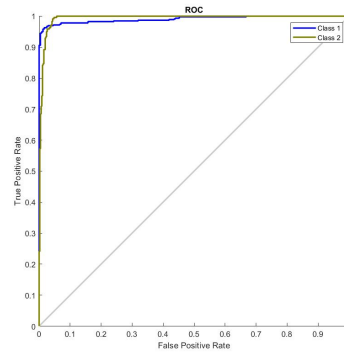
(a) ROC *trainbr*



(b) ROC *trainrp*



(c) ROC *trainoss*



(d) ROC *traingd*

Evaluando el resultado obtenido de entre las representaciones de *ROC* se corrobora que el mejor método de entrenamiento pertenece a *trainbr*, y el peor a *trainrp*; debido a la proximidad/distancia de los valores de *True Positive Rate* para *False Positive Rate* entre 0 y 0.2.

Parte 2

El objetivo de este apartado es comprender e implementar una red neuronal recursiva - con la ayuda de Simulink - que permita el control de la trayectoria de un robot.

Ejercicio 1

Como preámbulo, se estudia el entorno en el que el robot será controlado: entorno sin obstáculos de dimensiones de 10x10 metros. El origen de coordenadas de este entorno se encuentra en el centro geométrico del mismo.

Se busca generar el controlador neuronal capaz de emular el comportamiento de un controlador. El esquema a seguir ha sido proporcionado por los escritores de esta práctica.

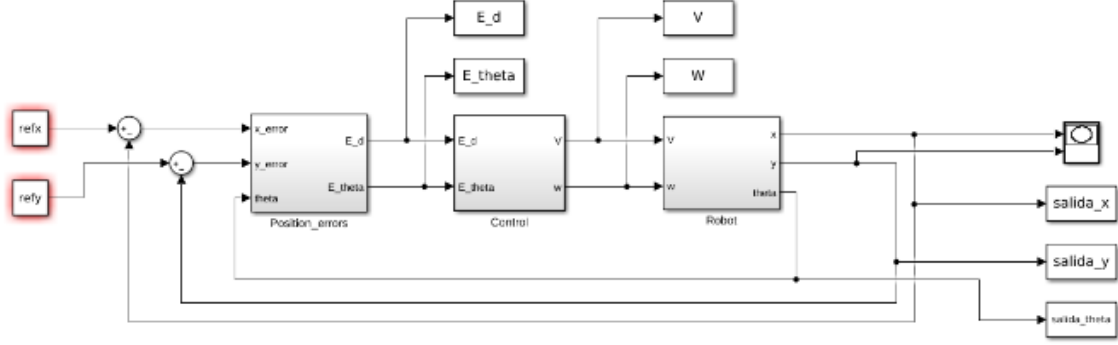


Figure 14: Esquema de control de posición.

El esquema anterior está compuesto por diferentes parte, o módulos. Uno de ellos, es el encargado de simular el funcionamiento del robot. Toda la información referente a este bloque ya ha sido referenciada en la práctica anterior, por lo que no parece necesario volver a recalcar lo que ya ha sido explicado.

Hay otro sistema existente que calcula el error en la posición del robot. Utiliza la posición del robot en respecto al objetivo, meta - se representa mediante las coordenadas x_r, y_r . En control del robot se lleva a cabo mediante el cómputo de dos errores: distancia al objetivo - [1]

$$E_d = \sqrt{(x_r - x_k)^2 + (y_r - y_k)^2} \quad (1)$$

Que permite controlar si el robot debe seguir moviéndose, la velocidad a la que debe proceder, etc. Y el ángulo respecto al objetivo - [2]

$$E_\theta = \text{atan2}(y_r - y_k, x_r - x_k) - \theta_k \quad (2)$$

De manera que se dirija hacia el mismo, no tome una ruta alternativa, se aleje del mismo, etc. Para la creación del bloque que se encarga de calcular los errores se emplea la codificación por bloques de las fórmulas expuestas anteriormente, teniendo en cuenta que E_θ debe estar comprendido dentro del intervalo $[-\pi, \pi]$ - [15].

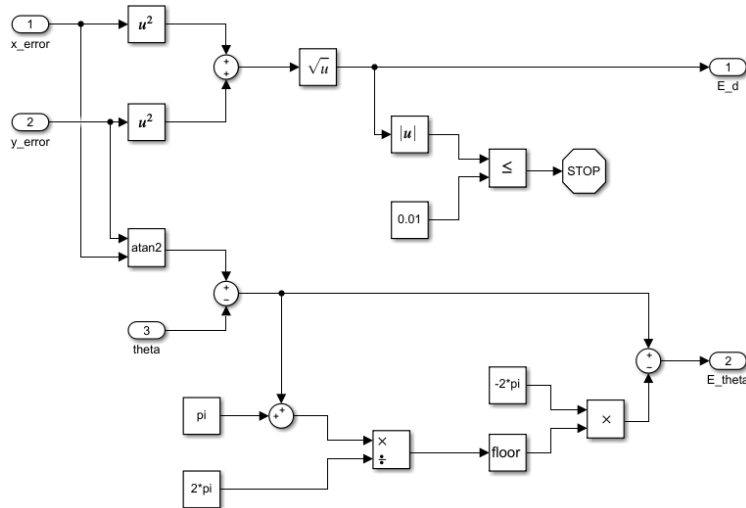


Figure 15: Cálculo de errores.

El bloque que controla los errores de distancia y ángulo necesita a su salida de un controlador de parada, el cual se encargue de que detener el movimiento del robot cuando la proximidad al objetivo sea inferior a 0.01 metros. Para ello se emplea el bloque recomendado por los escritores de la práctica - [16].

El último de los bloques, subsistema de control, ha sido generado por parte de los escritores de esta práctica, por lo que no se ve necesario realizar ningún tipo de análisis referente a este bloque.

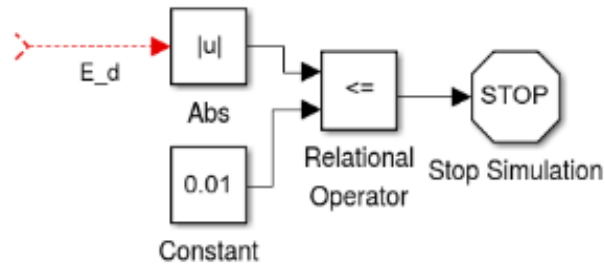


Figure 16: Condición de parada.

Ejercicio 2

Una vez montado el sistema anterior, es momento de desarrollar el control neuronal para el robot. Pero antes de proceder por esta ruta, es necesario saber si el sistema implementado funciona correctamente. Para ello, se ejecuta el script llamado *RunPositionControl.m*, facilitado por los desarrolladores del enunciado de esta práctica, y se ejecuta el esquema de control del robot. Se observan los siguientes resultados - 17:

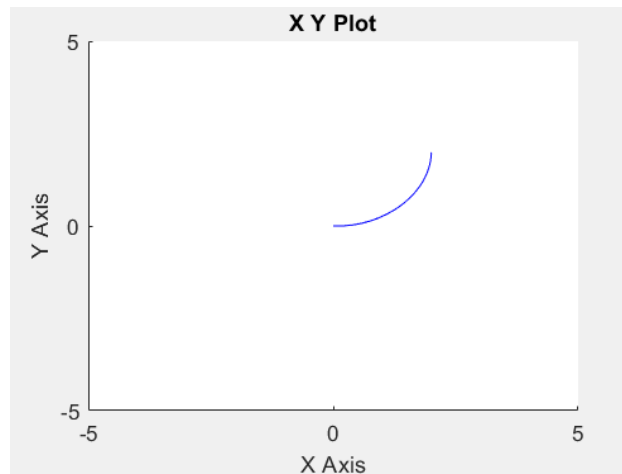


Figure 17: Trayectoria del robot con posición inicial (2.0, 2.0).

La trayectoria seguida por el robot cabe esperarse, el robot se dirige hacia el objetivo hasta alcanzar el punto del espacio (2.0, 2.0). Para comprobar que la ejecución en efecto produce valores, se observa que se generan dentro del *workspace* las variables con los datos correspondientes - [18a y 18b]:

Workspace	
Name ^	Value
ans	1x1 SimulationOutput
E_d	1x1 struct
E_theta	1x1 struct
logout	1x1 Dataset
out	1x1 SimulationOutput
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

(a) Variables generadas durante la ejecución.

Variables - salida_theta	
Field ^	Value
time	1001x1 double
signals	1x1 struct
blockName	'PositionControl/To Workspace'

Variables - salida_x	
Field ^	Value
time	1001x1 double
signals	1x1 struct
blockName	'PositionControl/To Workspace'

Variables - salida_theta	
Field ^	Value
time	1001x1 double
signals	1x1 struct
blockName	'PositionControl/To Workspace2'

(b) Ejemplos de la estructura de las variables.

Pero mediante la ejecución de la simulación no es la única manera de representar la trayectoria del robot. Debido a que la salida del sistema da la posición respecto al eje X e Y del robot en cada momento, es posible recuperar esos datos mediante el código proporcionado por los profesores de la asignatura e imprimir dicho datos mediante una función, dentro del entorno de MATLAB - [19].

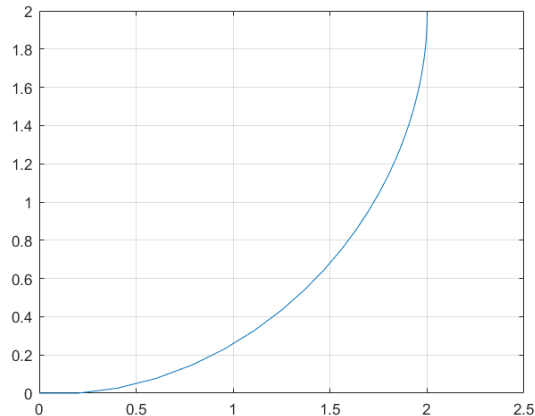
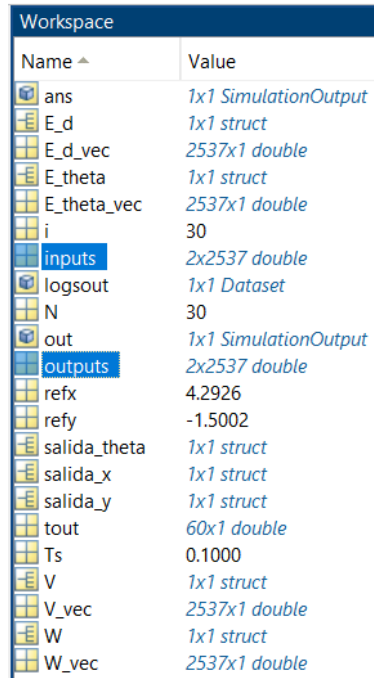


Figure 19: Trayectoria del robot con posición inicial (2.0, 2.0). Representación mediante comandos.

El resultado obtenido es el mismo conseguido mediante la simulación del robot. Aunque al realizarlo mediante la función *plot* se dispone de algunas opciones más propias del entorno (zoom, selección de datos, etc.). El resultado obtenido es el mismo conseguido mediante la simulación del robot. Aunque al realizarlo mediante la función *plot* se dispone de algunas opciones más propias del entorno (zoom, selección de datos, etc.).

Lo próximo que se pide es realizar una ejecución consecutiva de 30 simulaciones (para ello se usa el bucle proporcionado por los redactores de la práctica). De la salida se piden las matrices obtenidas, pero debido a las dimensiones de las mismas, se adjunta tan solo una prueba de que la ejecución ha sido exitosa y que se han generados las matrices en cuestión - [20].



Name	Value
ans	1x1 SimulationOutput
E_d	1x1 struct
E_d_vec	2537x1 double
E_theta	1x1 struct
E_theta_vec	2537x1 double
i	30
inputs	2x2537 double
logout	1x1 Dataset
N	30
out	1x1 SimulationOutput
outputs	2x2537 double
refx	4.2926
refy	-1.5002
salida_theta	1x1 struct
salida_x	1x1 struct
salida_y	1x1 struct
tout	60x1 double
Ts	0.1000
V	1x1 struct
V_vec	2537x1 double
W	1x1 struct
W_vec	2537x1 double

Figure 20: Variables generadas por la ejecución de las simulaciones.

Una vez realizadas estas comprobaciones para cerciorarse de que el modelo está bien formado, es momento de sustituir el controlador por una red neuronal. Se pide que solo tenga una capa oculta y que esta disponga del número más óptimo de neuronas posibles. Dicho valor ha de ser encontrado empíricamente.

Para los escritores de este documento, el mejor rendimiento ha sido obtenido al estimar un número de neuronas en capa oculta de 50. Para demostrarlo, se adjunta la comparativa de rendimiento con otros valores⁴ - [21].

⁴También se ha calculado con valores de $n = [40, 45, 55, 60]$, pero no se desea sobrecargar con imágenes este documento, por lo que se hace la comparativa con valores de n mucho menores/mayores.

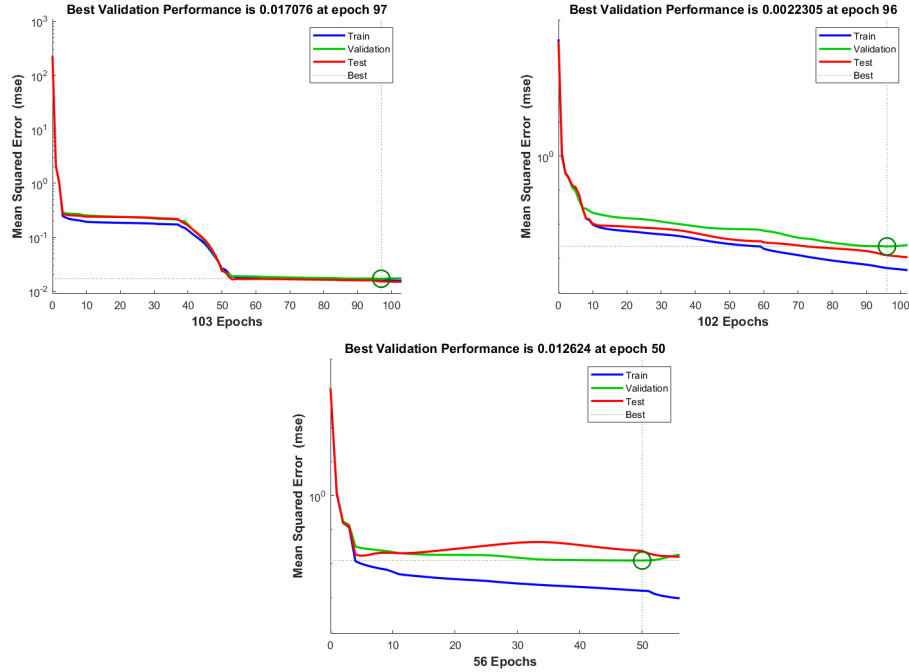


Figure 21: Rendimientos de diferentes tamaños de capa oculta.

Tras esta fase de generación y entrenamiento de la red neuronal, se pasa a la creación del bloque de *Simulink* correspondiente. Este bloque será posteriormente utilizado dentro del modelo de control del robot, como controlador, para ver el funcionamiento del mismo usando redes neuronales - [22].

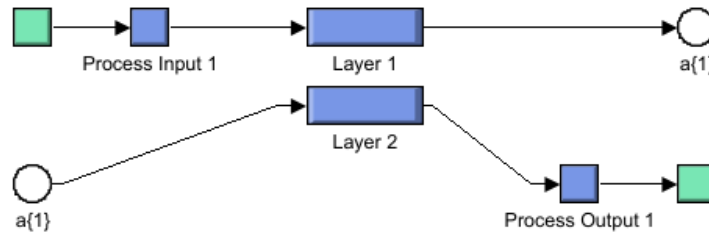


Figure 22: Red neuronal tras el entrenamiento con los datos generados por 30 diferentes trayectorias.

La red neuronal que se ha generado debe ahora insertarse dentro del controlador del robot, sustituyendo el bloque controlador proporcionado anteriormente por los profesores de la asignatura. Debido a que el número de entradas y salidas es diferente, es necesario incluir un multiplexor a la entrada y un demultiplexor a la salida del controlador neuronal - [23].

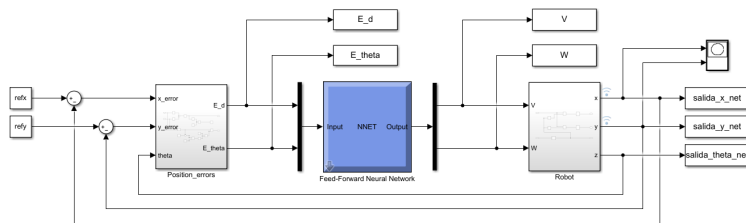


Figure 23: Sistema de control del robot mediante red neuronal.

Por último, se solicita la comparación de ambos sistemas, el lógico contra el neuronal. De los resultados se espera que el más fiable sea obviamente el sistema lógico, al ser el que ha proporcionado los datos de la red neuronal, y al ser el que contiene el comportamiento real que debería tener el robot en su estado más óptimo. Pero, del controlador neuronal (gracias a los múltiples datos de entrada y salida proporcionados por las numerosas ejecuciones del controlador lógico) se espera un funcionamiento bastante similar en la mayoría de las ejecuciones.

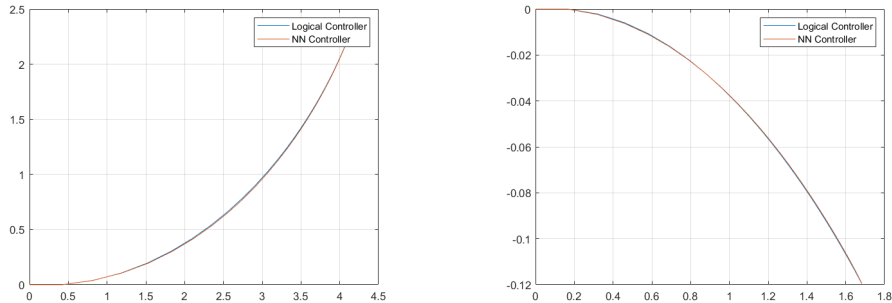


Figure 24: Comparación de controlador lógico contra controlador neuronal.

Como se demuestra por las anteriores capturas, el resultado obtenido es el esperado. La trayectoria del robot con el controlador neuronal se aproxima en gran medida con la del controlador lógico.

Parte 3

Ejercicio 1

El objetivo de este apartado es realizar el entrenamiento de una red neuronal de tipo NARX que sea capaz de emular la respuesta del sistema dinámico cuyas entradas, salidas y ecuaciones son desconocidas, pero del cual se conoce que posee un tiempo discreto de $T_s = 0.1s$.

El sistema desconocido del que habla es el bloque central del simulador encontrado en la figura 25. Dentro del modelo de la figura se busca almacenar la reacción del sistema ante entradas aleatorias.

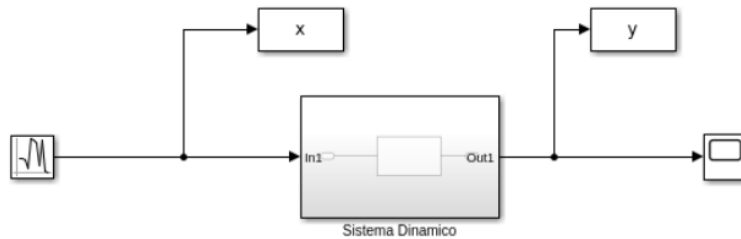


Figure 25: Sistema dinámico por identificar.

Para ello se utiliza el script proporcionado por el los desarrolladores de esta práctica. Además, se incluye el fragmento de código que permite el almacenamiento y recuperación de los datos sacados a partir de la simulación del sistema dinámico desconocido en ficheros *.mat* - [7].

Listing 7 Guardar datos de simulación.

```
save('inputs.mat', 'inputs');  
save('outputs.mat', 'outputs');  
  
load('inputs.mat', 'inputs');  
load('outputs.mat', 'outputs');
```

Tras el tratamiento de la información recomendado por la práctica, se realiza el entrenamiento de la red NARX (tipo *feedforward*). Tal y como se explica en el enunciado de la práctica, el entrenamiento de la red neuronal supone la conversión de la misma a una red neuronal con recursividad - [26].

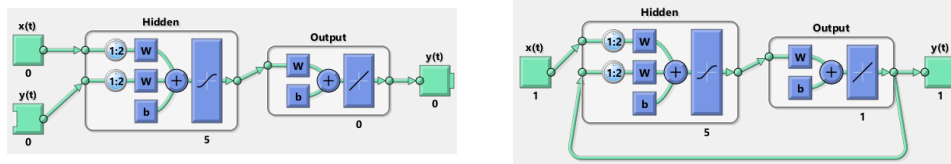


Figure 26: Transformación a red neuronal recursiva tras el entrenamiento.

Como último paso de este ejercicio, se pide generar el bloque correspondiente para la red neuronal e introducirlo al modelo anteriormente generado - mediante el comando *gensim*.

El resultado obtenido se puede observar en la figura 27.

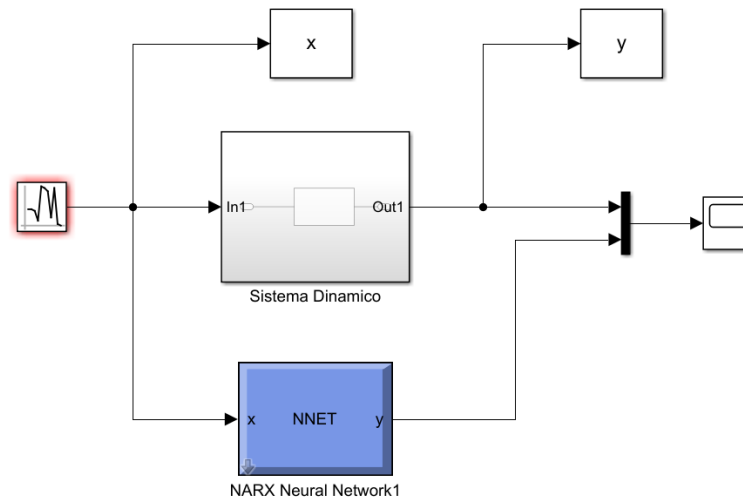


Figure 27: Modelo para la comparación del sistema dinámico contra NARX.

Desde este modelo (27) se puede observar la comparativa de resultados en la gráfica *scope*. La comparativa entre ambos sistemas puede observarse en la siguiente figura - [28]

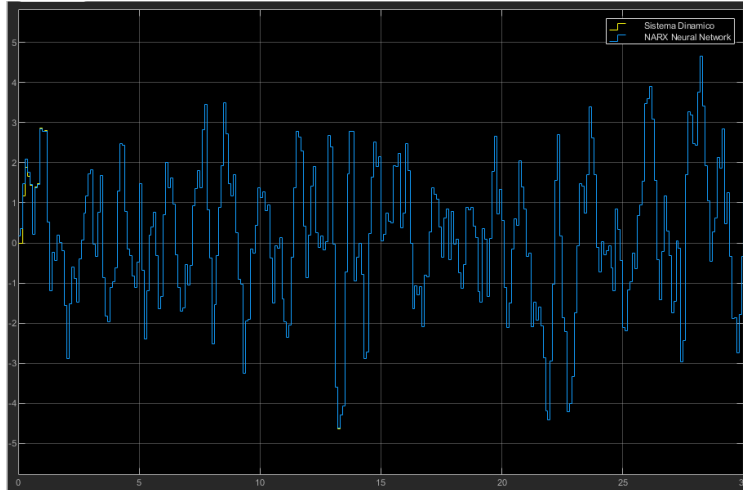


Figure 28: Gráfica comparativa de sistema dinámico contra NARX.

Al entrenarse contra un gran número de entradas, la red neuronal de tipo NARX es capaz de simular con gran precisión el trabajo del sistema dinámico desconocido.

Ejercicio 2

En este ejercicio nos encontramos con el sistema de control de robot ya utilizado en apartados anteriores.

Para ello, se incluye el bloque generador de trayectoria, el cual ha sido proporcionado por los desarrolladores de esta práctica. Este bloque depende de varios parámetros de trayectoria (x_0 , y_0 , θ_0) y del período de muestreo (T_s). Este bloque debe ser introducido en lugar de los parámetros de entrada que había en apartados anteriores.

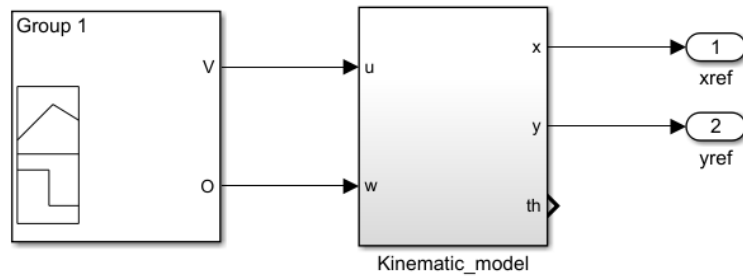


Figure 29: Generador de trayectoria.

El bloque controlador ha sido sustituido por uno especializado en el seguimiento de trayectorias (también proporcionado por los escritores de la práctica).

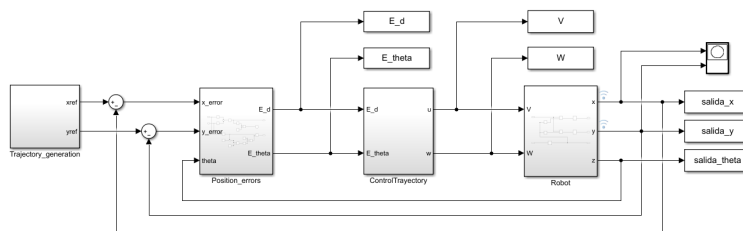


Figure 30: Modelo Robot para seguimiento de una trayectoria.

Para ejecutarlo, se ha generado un script el cual se encarga de iniciar los parámetros de trayectoria y período de muestro de los que se ha hablado con anterioridad; de simular el entorno; y de dibujar en una gráfica tanto la trayectoria generada como el recorrido realizado por el robot - [8].

Listing 8 Guardar datos de simulación.

```
Ts = 100e-3

x_0 = 0
y_0 = 0
th_0 = 0

sim('TrajectoryControl.slx')

x = x.signals.values;
y = y.signals.values;
x_out = salida_x.signals.values;
y_out = salida_y.signals.values;

figure;
plot (x, y);          hold on;
plot (x_out, y_out); hold off;
legend ('Trajectory Generated', 'Robot Trajectory')
grid on;
```

De este código, se obtienen las dos trayectorias mencionadas anteriormente. Como cabía esperarse, el robot es capaz de imitar la trayectoria del generador.

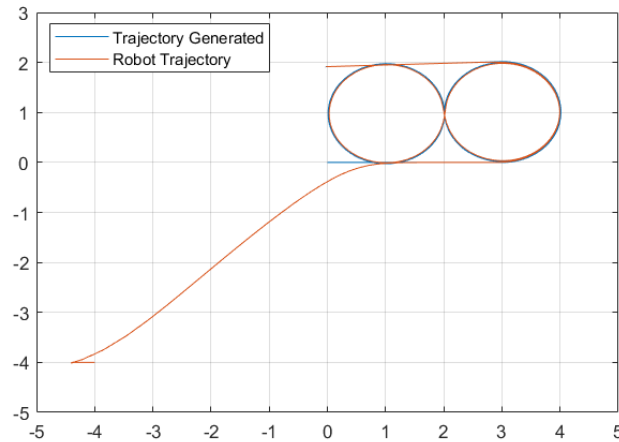


Figure 31: Robot siguiendo la trayectoria especificada.

Tras esto, lo que se pide es encontrar el número óptimo de neuronas para la capa oculta de la red neuronal NARX. Para ello se ha generado un script que permita numerosos experimentos y guardar los datos de la misma manera que se ha hecho en la figura 20.

De los diferentes experimentos realizados, se ha llegado a la conclusión de que el rendimiento más óptimo está en torno a las 15 neuronas en capa oculta.

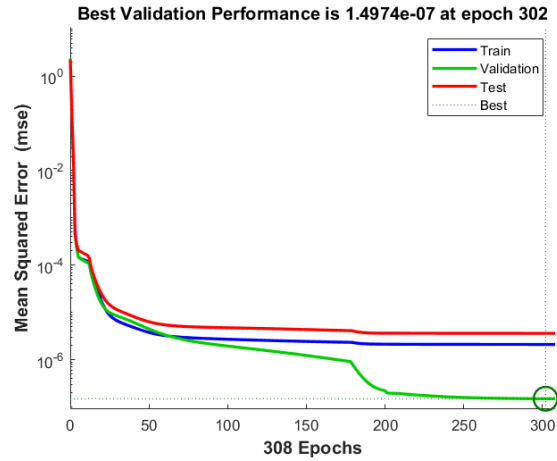


Figure 32: Rendimiento Óptimo con 15 neuronas en la capa oculta.

El proceso seguido ha sido el mismo que en apartado anterior, por lo que no se ve como necesario repetir información. Teniendo este conocimiento, se entrena un módulo de red neuronal tipo NARX recursivo, para poder utilizarse dentro del control de trayectoria del robot, para poder compararlo con respecto al controlador normal - 33.

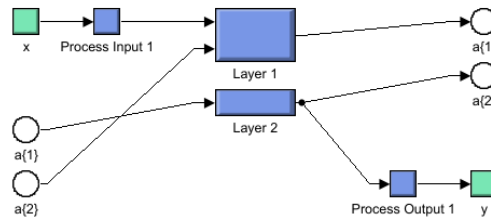


Figure 33: Red Neuronal de tipo NARX recursivo para control de trayectoria.

Se ha creado un script de MATLAB que busca comparar la eficacia del controlador lógico con respecto al controlador neuronal. Para ello es tan sencillo como realizar ambas simulaciones y contraponer los datos obtenidos.

De esto, se obtiene que la red neuronal ha sido capaz de aprender de la trayectoria seguida por el controlador lógico del robot y aproximar con bastante precisión cual debe ser su comportamiento - [34].

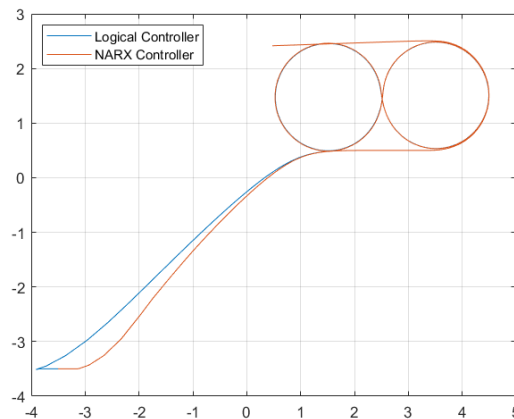


Figure 34: Comparativa de Controlador Lógico contra Controlador de Red Neuronal Narx.

Conclusión

De esta práctica, se ha aprendido la obtención de datos, entrenamiento y creación de redes neuronales capaces de imitar los comportamientos de los datos que han aprendido.

Es decir, con la cantidad suficiente de datos, una red neuronal es capaz de imitar con mucha precisión un comportamiento dado.

Aplicado al control de un robot, es posible seguir trayectorias dadas. Si bien es cierto, que para conseguir la precisión necesaria como para que el robot fuera capaz de imitar la trayectoria con un controlador neuronal, comenzando dicha trayectoria con cualquier ángulo y desde cualquier punto, sería necesaria una gran cantidad de datos y de experimentos, bastante superior a lo que los dispositivos de los escritores de este documento tienen acceso.