



Universidad de Alcalá

Escuela Politécnica Superior

Universidad de Alcalá

Práctica 1

Identificación y control neuronal

Sistemas de control inteligente

Laboratorio Lunes 12:00 – 14:00

Grado en Ingeniería Informática – Curso 2018/2019

Eduardo Graván Serrano – 03212337L

Marcos Barranquero Fernández – 51129104N

CONTENIDO

Parte 1	3
Ejercicio 1	3
Ejercicio 2	6
Descenso por el gradiente	8
Retroceso resiliente propagado.....	8
Descenso por el gradiente con inercia	8
Regularización bayesiana	9
Ejercicio 3	9
Apartado 1	9
Algoritmo descenso por el gradiente	12
Algoritmo retroceso resiliente propagado	12
Algoritmo Descenso por el gradiente con inercia	13
Algoritmo regularización bayesiana.....	13
Ejercicio 4	14
Algoritmo descenso por el gradiente	15
Algoritmo descenso por el gradiente con inercia.....	15
Algoritmo resilient backpropagation.....	16
Algoritmo regularización bayesiana.....	17
Parte 2	18
Objetivo y Descripción del Sistema	18
Desarrollo de la Práctica	20
Parte 3	27
Ejercicio 1	27
Ejercicio 2	30

PARTE 1

EJERCICIO 1

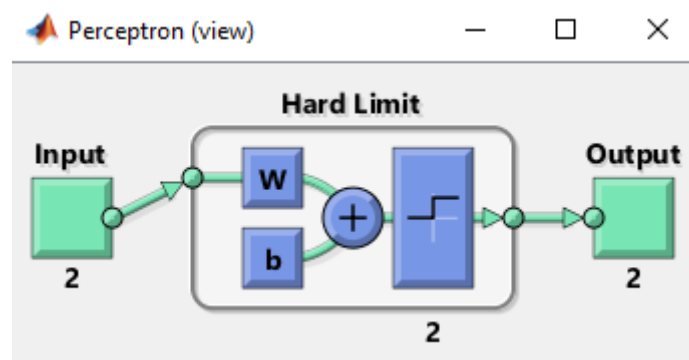
Inicialmente se parte de la estructura del ejemplo con los siguientes datos:

```
% Entradas
P=[0.1 0.7 0.8 0.8 1.0 0.3 0.0 -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];
% Targets
T=[1 1 1 0 0 1 1 1 0 0;
    0 0 0 0 0 1 1 1 1 1];
```

Para visualizar la estructura de la neurona podemos emplear el comando view:

```
% Red neuronal de tipo perceptron
net = perceptron;
view(net)
```

Generando el siguiente diagrama:



Vemos que tenemos una entrada de tamaño 2, una salida de tamaño 2 (2 dígitos binarios para codificar las 4 clases). Por tanto, se tiene 1 capa con 2 neuronas. Visualizamos también que emplea la función de transferencia de Hard Limit.

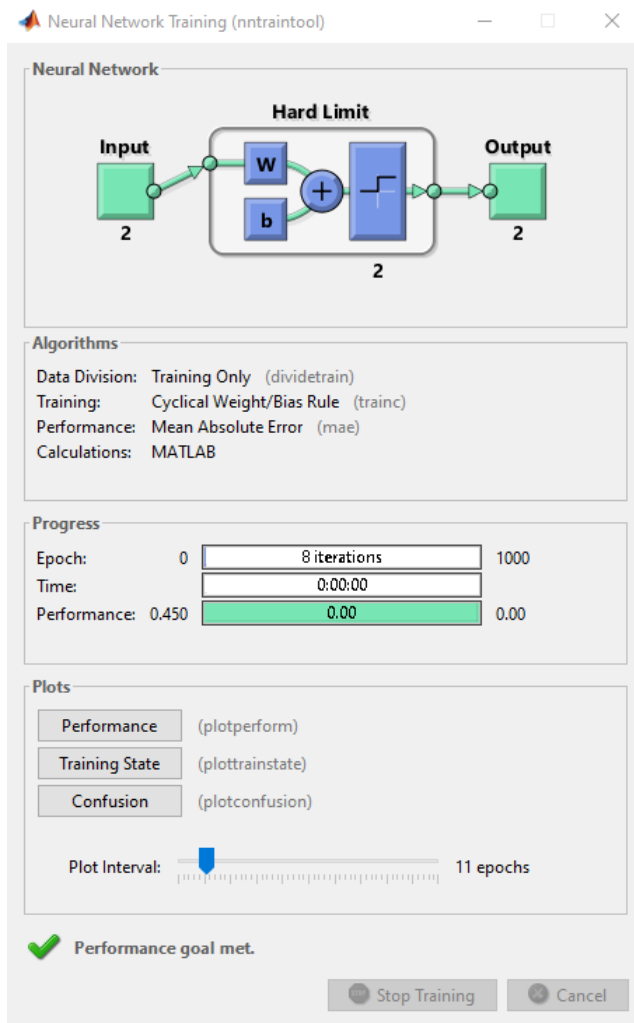
También podemos visualizarlo accediendo a la variable:

```
% Tenemos 1 capa y 2 neuronas
disp("Nº de capas: " + net.numLayers)
disp("Nº de neuronas: " + net.layers{1}.size)
```

Finalmente podemos dibujar la gráfica:

```
% Dibujo la gráfica de selección de tipo esta
% Esto dibuja datos
plotpv(P,T);

% Esto dibuja gráficas de las redes neuronales que
% clasifican los datos
plotpc(net.iw{1,1},net.b{1});
```

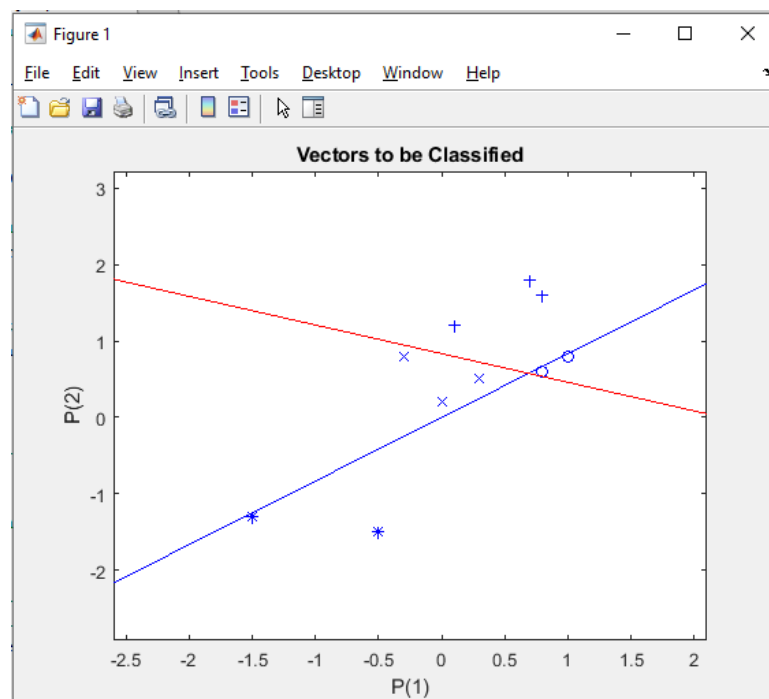


Vemos que toma 8 iteraciones del algoritmo hasta poder clasificar en 4 clases (4 cuadrantes) los diferentes elementos proporcionados.

Generando la gráfica, podemos ver que clasifica los datos en 4 cuadrantes, acordes a las 4 clases. Esto es acorde con la fórmula dada en teoría:

$$N^{\circ} \text{cuadrantes} = 2^{\text{tamaño entrada}}$$

Gráfica generada:



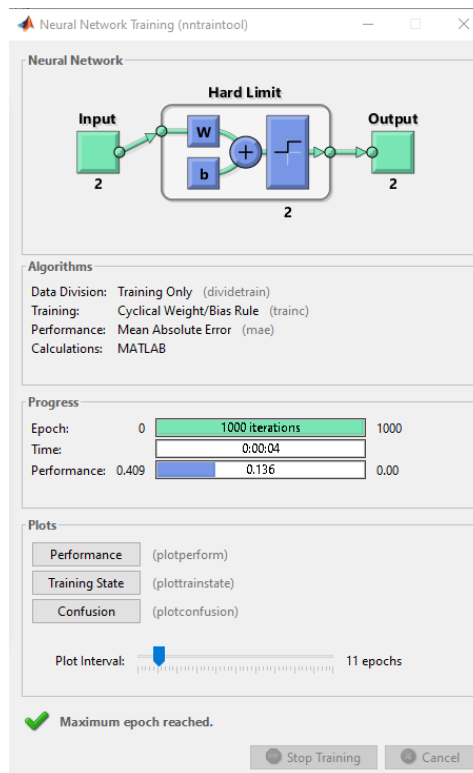
Si añadimos el dato sugerido, la entrada queda así:

```
% Entradas
P=[0.1 0.7 0.8 0.8 1.0 0.3 0.0 -0.3 -0.5 -1.5 0.0;
    1.2 1.8 1.6 0.6 0.8 0.5 0.2 0.8 -1.5 -1.3 -1.5];
```

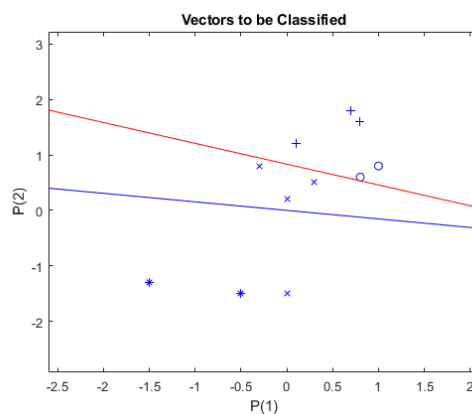
La matriz objetivo queda así:

```
% Targets nuevos
T=[1 1 1 0 0 1 1 1 0 0 1;
    0 0 0 0 0 1 1 1 1 1 1];
```

Y vemos que no consigue realizar la clasificación. Llega al límite preestablecido de 1000 iteraciones.



Si generamos la gráfica, observamos que no consigue clasificar en 4 cuadrantes distinguidos los elementos, sino que quedan cuadrantes heterogéneos. Esta es una de las principales limitaciones del perceptrón.



EJERCICIO 2

En este ejercicio se propone estudiar cómo afecta el algoritmo de entrenamiento seleccionado a la aproximación de funciones. Para ello, se propone aproximar la siguiente función durante el dominio del tiempo establecido:

```
% Definimos vectores de entrada y salida.
tiempo = -3:.1:3; % eje de tiempo
funcion_a_aproximar = sinc(tiempo) + 0.001 * randn(size(tiempo)); %
    funcion que se desea aproximar
```

Se tienen los siguientes algoritmos de entrenamiento:

- **Descenso por el gradiente:** se emplea el gradiente para encontrar el mínimo de la función.
- **Retroceso propagado resiliente:** algoritmo que pretende mitigar los efectos de las magnitudes sobre las derivadas parciales, y es eficiente en memoria y en tiempo.
- **Descenso por el gradiente con inercia:** actualiza peso y vías acorde al gradiente y la inercia.
- **Regularización bayesiana con propagación de retroceso:** actualiza el peso y las vías según la optimización de Levenberg-Marquardt.

Para cargar diferentes algoritmos, se ha diseñado la función de elección de algoritmo:

```
function eleccion = elegir_algoritmo()
    eleccion = "";

    while true
        disp('1) Descenso por el gradiente. ');
        disp('2) Resilient Backpropagation. ');
        disp('3) Descenso por el gradiente con inercia. ');
        disp('4) Regularizacion bayesiana con propagación de retroceso. ');
        numero_algoritmo = input('Selecciona algoritmo de entrenamiento: ');

        switch numero_algoritmo
            case 1
                eleccion = 'traingd';
                break;
            case 2
                eleccion = 'trainrp';
                break;
            case 3
                eleccion = 'traingdm';
                break;
            case 4
                eleccion = 'trainbr';
                break;
            otherwise
                disp('Error. Opci?n no valida. ');
                break;
        end
    end
end
```

Finalmente, con el siguiente script creamos la red neuronal e imprimimos la gráfica de entrenamiento:

```
% Cargo numero de neuronas de la capa intermedia.
hiddenLayerSize = input('¿Cuántas neuronas de la capa oculta se emplearán?:');

% Pido algoritmo de entrenamiento
algoritmo_entrenamiento = elegir_algoritmo();

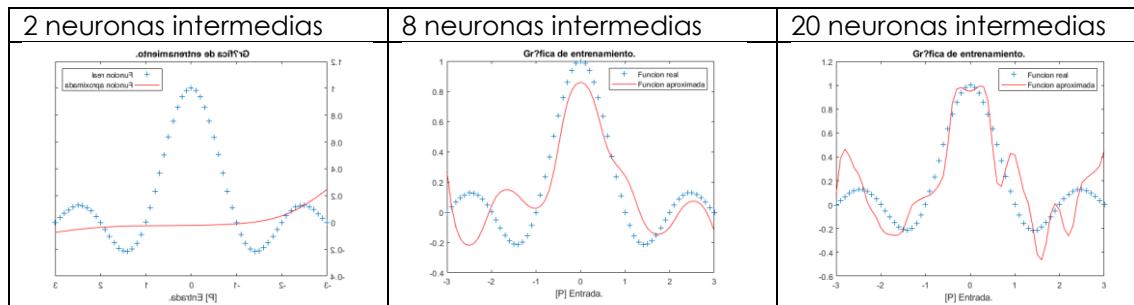
% Creo red neuronal
net = fitnet(hiddenLayerSize, algoritmo_entrenamiento);
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Y la entreno
net = train(net, tiempo, funcion_a_aproximar);
Y = net(tiempo);

% Realizamos representacion grafica
hold on;
plot(tiempo, funcion_a_aproximar, '+');
plot(tiempo, Y, '-r');
hold off;
title('Grafica de entrenamiento. ');
xlabel('[P] Entrada');
ylabel('[T] Target');
legend('Funcion real', 'Funcion aproximada')
```

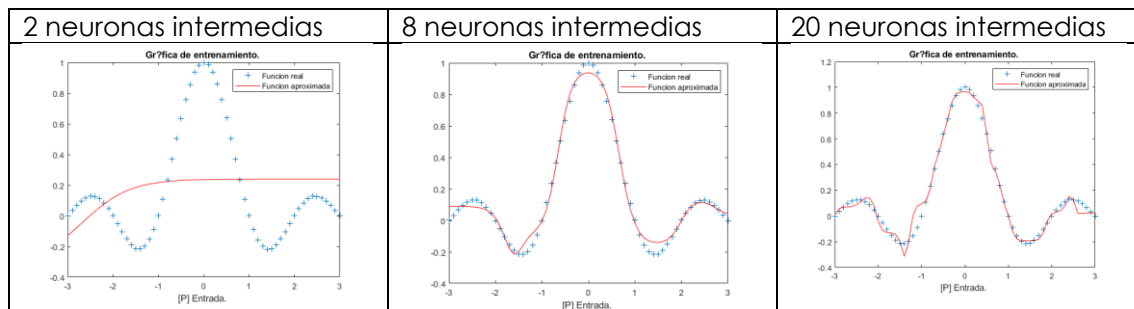
Para realizar el estudio, se han realizado varias simulaciones con tamaños distintos de capas intermedias para cada algoritmo.

DESCENSO POR EL GRADIENTE



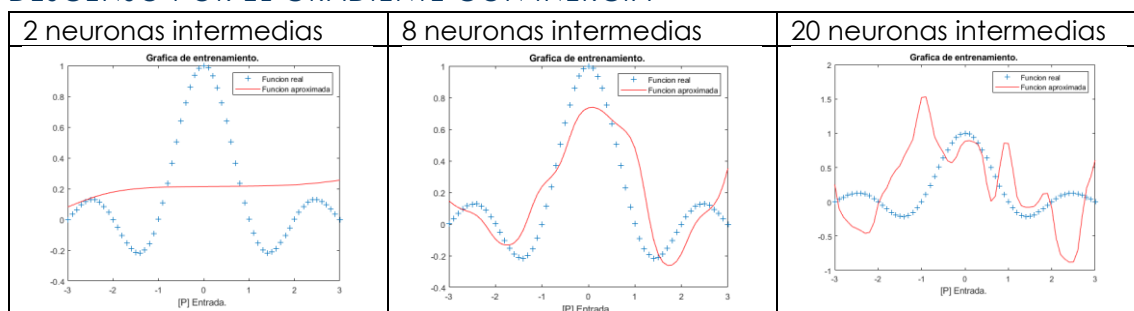
Observamos que a mayor número de capas consigue aproximarse más en puntos concretos, sin embargo, también incrementa el número de picos que tiene la función aproximada. No consigue ajustarse con precisión a la función original.

RETROCESO RESILIENTE PROPAGADO



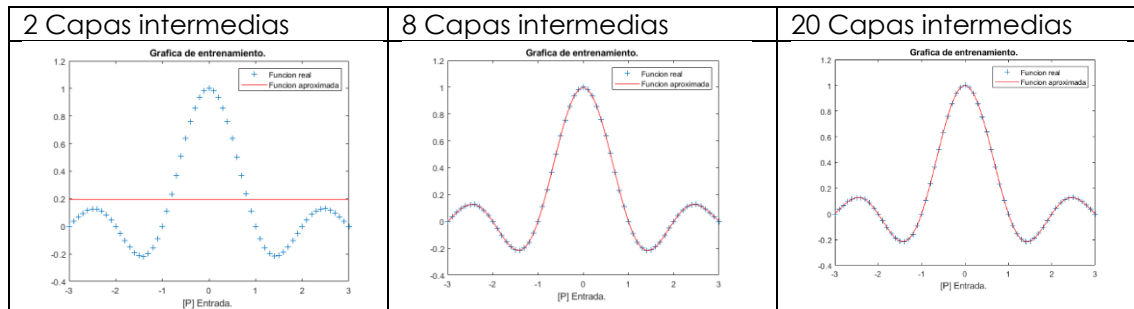
Observamos una vez más el mismo patrón que con el descenso por el gradiente: a mayor número de capas intermedias, obtenemos más precisión en puntos concretos pero mayor cantidad de picos.

DESCENSO POR EL GRADIENTE CON INERCIA



En este caso podemos contemplar que a mayor número de capas intermedias y una vez pasado cierto umbral, la red neuronal comenzará a aproximar de peor forma a la función objetivo.

REGULARIZACIÓN BAYESIANA



Finalmente, empleando este algoritmo concluimos que realiza una aproximación perfecta una vez se tienen las suficientes capas intermedias, lo cual es congruente ya que hace aproximaciones más precisas a cambio de consumir mayor cantidad de recursos y cálculos más complejos.

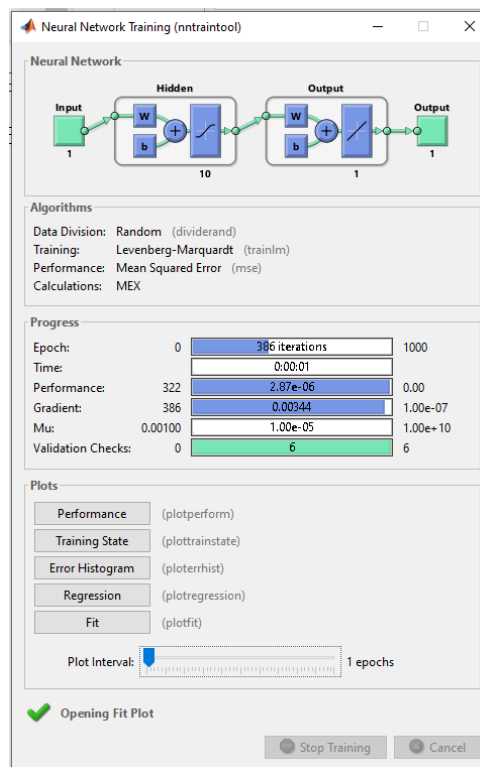
EJERCICIO 3

APARTADO 1

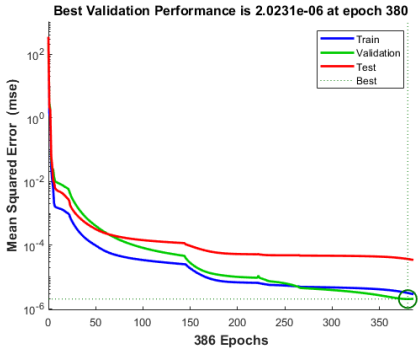
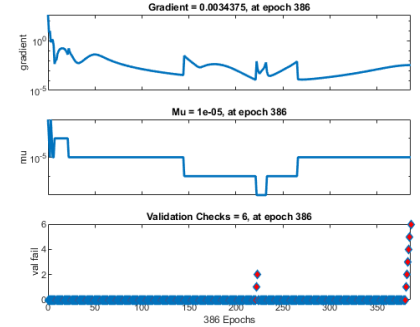
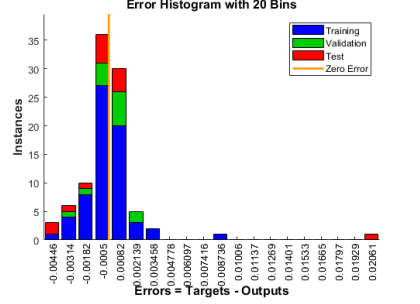
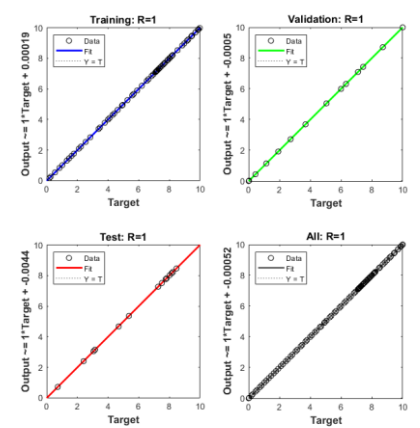
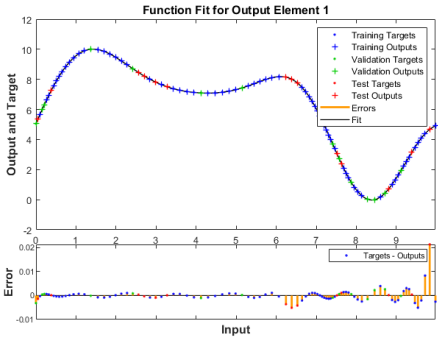
Ejecutando el script propuesto, podemos observar las siguientes gráficas:

- **PlotPerform:** contrasta el error contra las épocas al actualizar los pesos.
- **PlotTrainState:** contrasta el estado de entrenamiento contra las épocas, permitiendo ver su evolución.
- **Plotrrhist:** permite ver la evolución e histograma del error contra las épocas.
- **Plotregression y plotfit:** permiten ver como evolucionan los datos de entrenamiento, validación y test.

Al ejecutar el script obtenemos el siguiente resultado:



Observamos que le ha tomado 386 iteraciones. Si generamos los gráficos...

Plot	Descripción
 <p>Best Validation Performance is 2.0231e-06 at epoch 386</p>	<p>PlotPerform: observamos que el rendimiento de la validación aumenta conforme se realizan más iteraciones. La gráfica muestra el mejor rendimiento basándose en la iteración con menor error de validación.</p>
 <p>Gradient = 0.0034375, at epoch 386</p> <p>Mu = 1e-05, at epoch 386</p> <p>Validation Checks = 6, at epoch 386</p>	<p>PlotTrainState: observamos que la ganancia de entrenamiento μ fluctúa conforme el gradiente incrementa o decrementa. Matlab detiene el entrenamiento tras 6 fallos seguidos en la validación de datos. En el tercer gráfico, que muestra el número de aciertos y fallos en la validación, se ve que ha fallado 7 veces antes de detenerse.</p>
 <p>Error Histogram with 20 Bins</p>	<p>Plotrrhist: el histograma muestra la diferencia de los outputs de la salida de la aproximación en esa iteración con respecto a los targets de la red neuronal. Podemos ver que al principio se produce un incremento de errores, y que conforme se vuelve a avanzar en las iteraciones se disminuye la cantidad de error cometido. Cuando vuelve a incrementarse al finalizar en la fase de test, Matlab detiene el training por cometer más de 6 fallos en la validación y test.</p>
 <p>Training: R=1</p> <p>Validation: R=1</p> <p>Test: R=1</p> <p>All: R=1</p>	 <p>Function Fit for Output Element 1</p>
<p>Plotregression y plotfit: Finalmente, ambos histogramas de como evolucionan las fases de entrenamiento, validación y test.</p>	

Se prepara el siguiente script para estudiar el impacto de modificar la división de los datos de entrenamiento, validación y test sobre el conjunto de datos bodyfat_dataset:

```
clear all;
close all;
% Carga de datos de ejemplo disponibles en la toolbox
[inputs, targets] = bodyfat_dataset;

% Creación de la red
hiddenLayerSize = input('¿Cuántas neuronas de la capa oculta se emplearan?:');

% Pido algoritmo de entrenamiento
algoritmo_entrenamiento = elegir_algoritmo();

net = fitnet(hiddenLayerSize, algoritmo_entrenamiento);

% División del conjunto de datos para entrenamiento, validación y test
entrenamiento = input('Introduce ratio de entrenamiento [0-1]: ');
evaluacion = input('Introduce ratio de evaluacion [0-1]: ');
testeo = input('Introduce ratio de test [0-1]: ');
net.divideParam.trainRatio = entrenamiento;
net.divideParam.valRatio = evaluacion;
net.divideParam.testRatio = testeo;

% Entrenamiento de la red
[net, tr] = train(net, inputs, targets);

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

% Visualización de la red
view(net)

function eleccion = elegir_algoritmo()
    eleccion = "";

    while true
        disp('1) Descenso por el gradiente. ');
        disp('2) Resilient Backpropagation. ');
        disp('3) Descenso por el gradiente con inercia. ');
        disp('4) Regularización bayesiana con propagación de retroceso. ');
        numero_algoritmo = input('Selecciona algoritmo de entrenamiento: ');

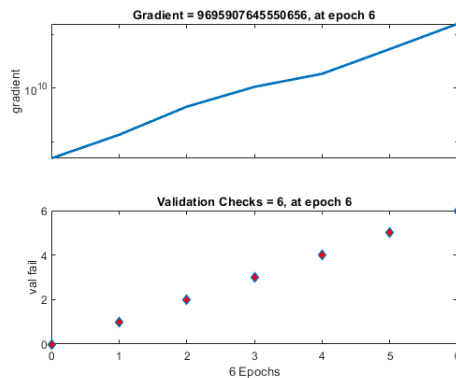
        switch numero_algoritmo
            case 1
                eleccion = 'traingd';
                break;
            case 2
                eleccion = 'trainrp';
                break;
            case 3
                eleccion = 'traingdm';
                break;
            case 4
                eleccion = 'trainbr';
                break;
            otherwise
                disp('Error. Opción no válida. ');
                break;
        end
    end

end

end
```

ALGORITMO DESCENSO POR EL GRADIENTE

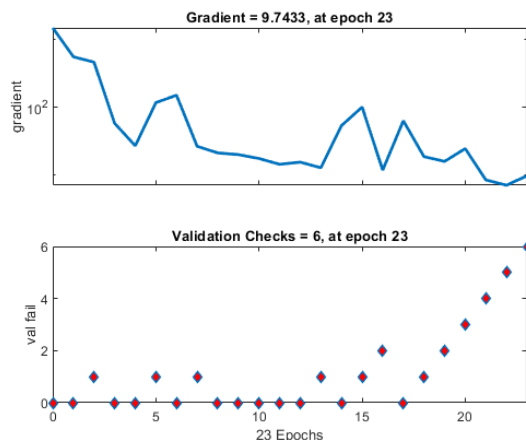
Observamos que para el algoritmo del descenso por el gradiente, independientemente del número de neuronas de la capa intermedia y la distribución entre validación, test y entrenamiento, siempre se detiene a las 6 iteraciones ya que comete 6 errores de validación seguidos:



Esto es debido a que, por la forma y tipo de datos que conforman el conjunto de bodyfat_dataset, es imposible realizar una clasificación con este algoritmo.

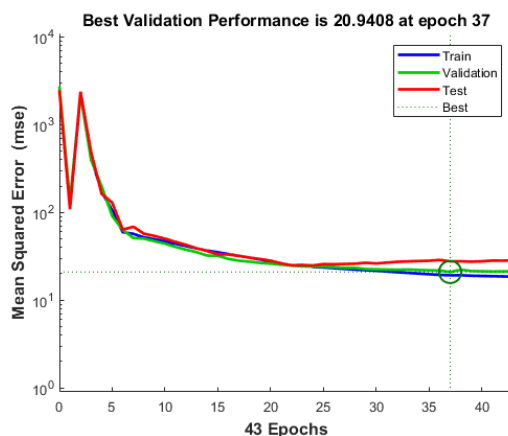
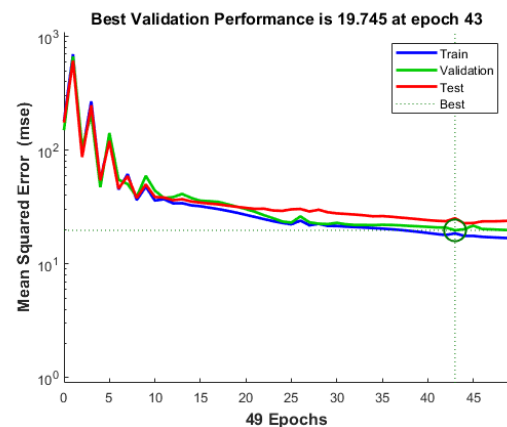
ALGORITMO RETROCESO RESILIENTE PROPAGADO

Con este algoritmo y utilizando 5 neuronas en la capa oculta, y distribución de 0.7-0.15-0.15 en entrenamiento, validación y test, consigue aguantar más iteraciones y clasificar de mejor forma:



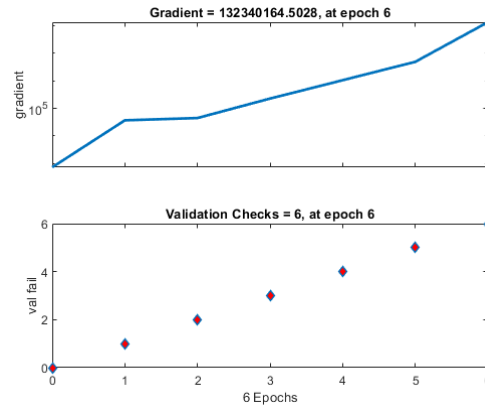
Observamos que si aumentamos el número de neuronas, obtenemos un mejor rendimiento y aguanta más iteraciones consiguiendo clasificar de mejor forma:

Finalmente, configurando un diferente ratio de distribución a 0.5-0.25-0.25 en entrenamiento, validación y test, obtenemos resultados similares:

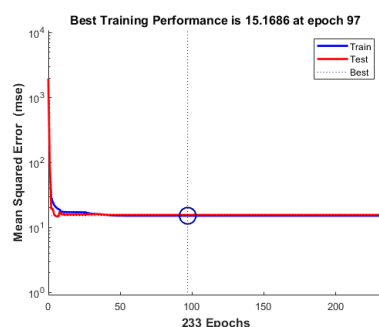
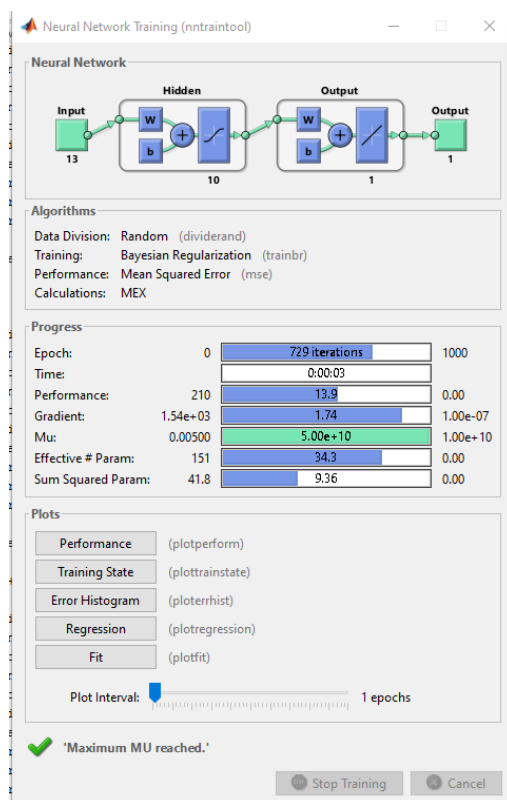


ALGORITMO DESCENSO POR EL GRADIENTE CON INERCIA

Con este algoritmo tenemos un caso similar al uso de descenso por el gradiente: nada más comenzar falla 6 veces seguidas en la validación, por lo que no se puede entrenar una red neuronal con este algoritmo para este set de datos:

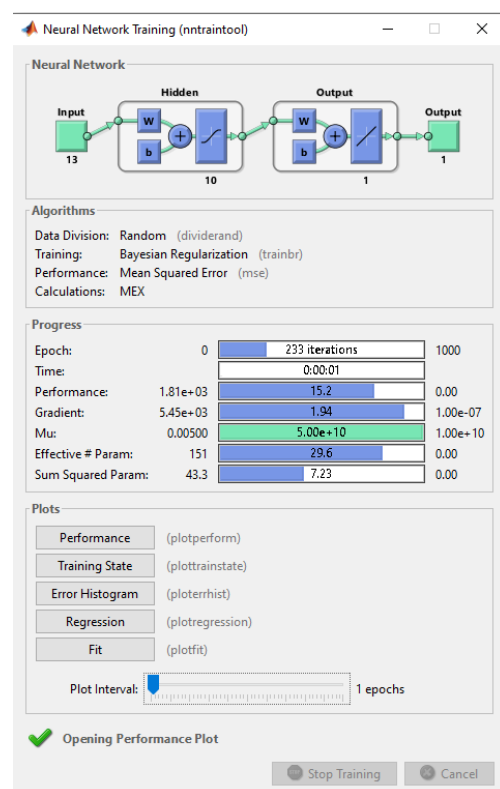


ALGORITMO REGULARIZACIÓN BAYESIANA



Para este algoritmo, vemos que toma una gran cantidad de iteraciones, pero consigue clasificar considerablemente el set de datos.

Si redistribuimos los ratios de entrenamiento, validación y test, manteniendo el mismo número de neuronas en la capa intermedia, obtenemos un mejor rendimiento y menos iteraciones para llegar al mismo resultado.



EJERCICIO 4

En este ejercicio se propone realizar un estudio similar al anterior, cambiando el set de datos por *cancer_dataset*, la red neuronal por una *patternet*, y utilizar las diferentes gráficas de evaluación asociadas a esta red neuronal. Se tiene el siguiente script para la evaluación:

```
clear all;
close all;
% Carga de datos de ejemplo disponibles en la toolbox
[inputs, targets] = simpleclass_dataset;

% CreaciOn de la red
hiddenLayerSize = input('¿Cuántas neuronas de la capa oculta se emplearan?:');

% Pido algoritmo de entrenamiento
algoritmo_entrenamiento = elegir_algoritmo();

net = patternnet(hiddenLayerSize, algoritmo_entrenamiento);

% DivisiOn del conjunto de datos para entrenamiento, validaciOn y test
entrenamiento = input('Introduce ratio de entrenamiento [0-1]: ');
evaluacion = input('Introduce ratio de evaluacion [0-1]: ');
testeo = input('Introduce ratio de test [0-1]: ');
net.divideParam.trainRatio = entrenamiento;
net.divideParam.valRatio = evaluacion;
net.divideParam.testRatio = testeo;

% Entrenamiento de la red
[net, tr] = train(net, inputs, targets);

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

% VisualizaciOn de la red
view(net)

function eleccion = elegir_algoritmo()
    eleccion = "";

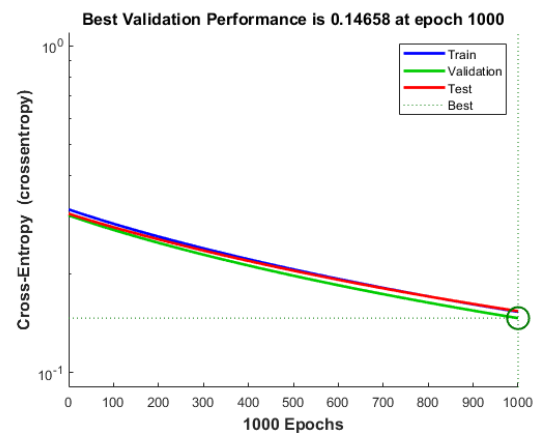
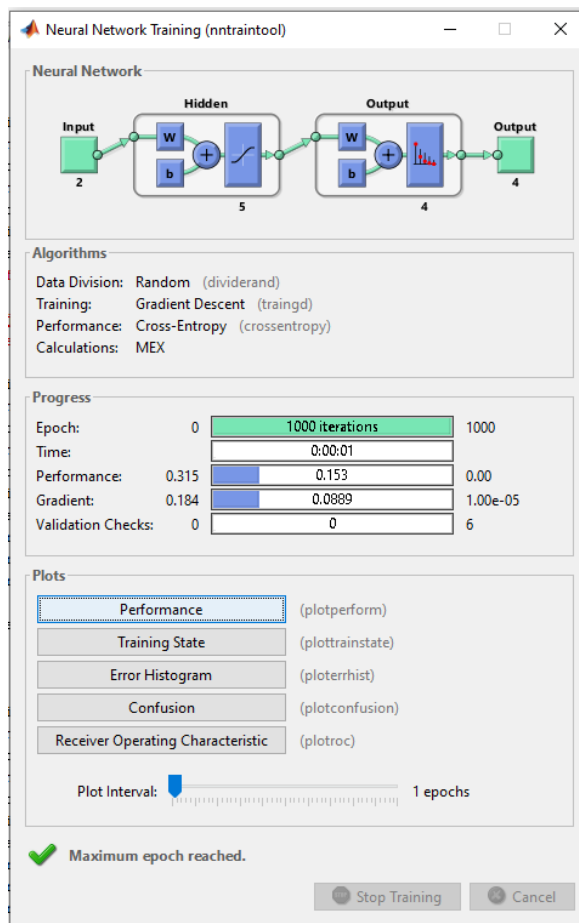
    while true
        disp('1) Descenso por el gradiente. ');
        disp('2) Resilient Backpropagation. ');
        disp('3) Descenso por el gradiente con inercia. ');
        disp('4) Regularizacion bayesiana con propagaci?n de retroceso. ');
        numero_algoritmo = input('Selecciona algoritmo de entrenamiento: ');

        switch numero_algoritmo
            case 1
                eleccion = 'traingd';
                break;
            case 2
                eleccion = 'trainrp';
                break;
            case 3
                eleccion = 'traingdm';
                break;
            case 4
                eleccion = 'trainbr';
                break;
            otherwise
                disp('Error. Opci?n no valida. ');
                break;
        end
    end

end
```

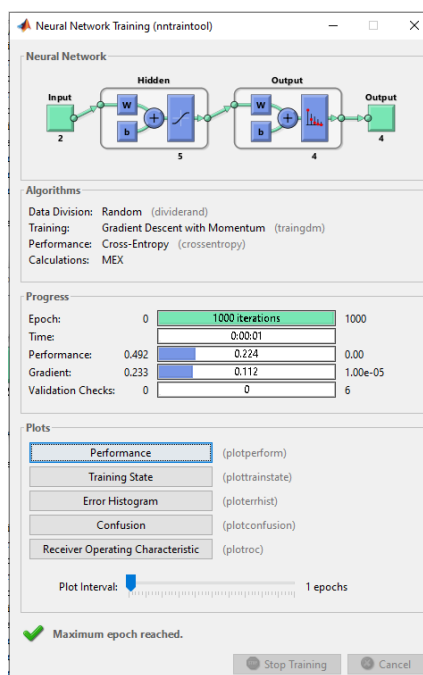
ALGORITMO DESCENSO POR EL GRADIENTE

Realizando la distribución típica 0.7-0.15-0.15, vemos que el rendimiento desciende progresivamente pero se alcanzan las 1000 iteraciones y por tanto se detiene:



Tanto cambiando los ratios como el número de neuronas de la capa intermedia se obtienen resultados similares, por lo que podemos concluir que este algoritmo no es el ideal para esta distribución de datos.

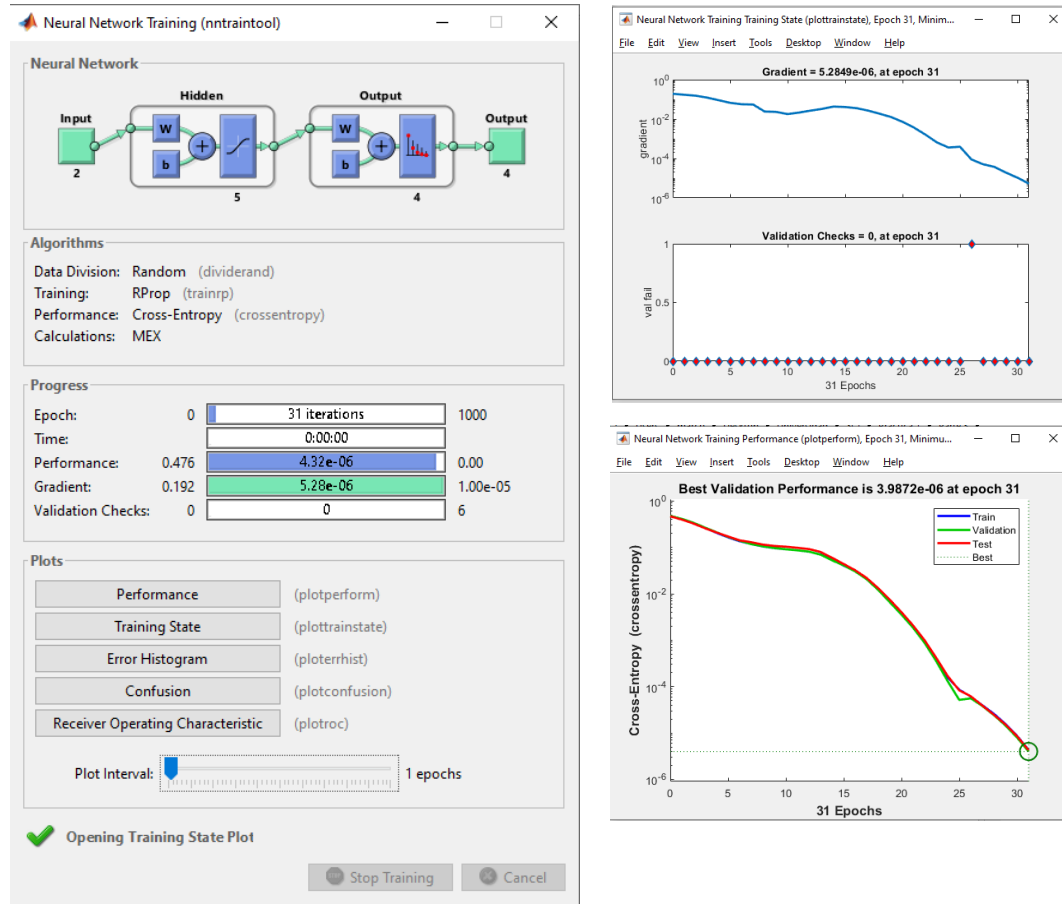
ALGORITMO DESCENSO POR EL GRADIENTE CON INERCIA



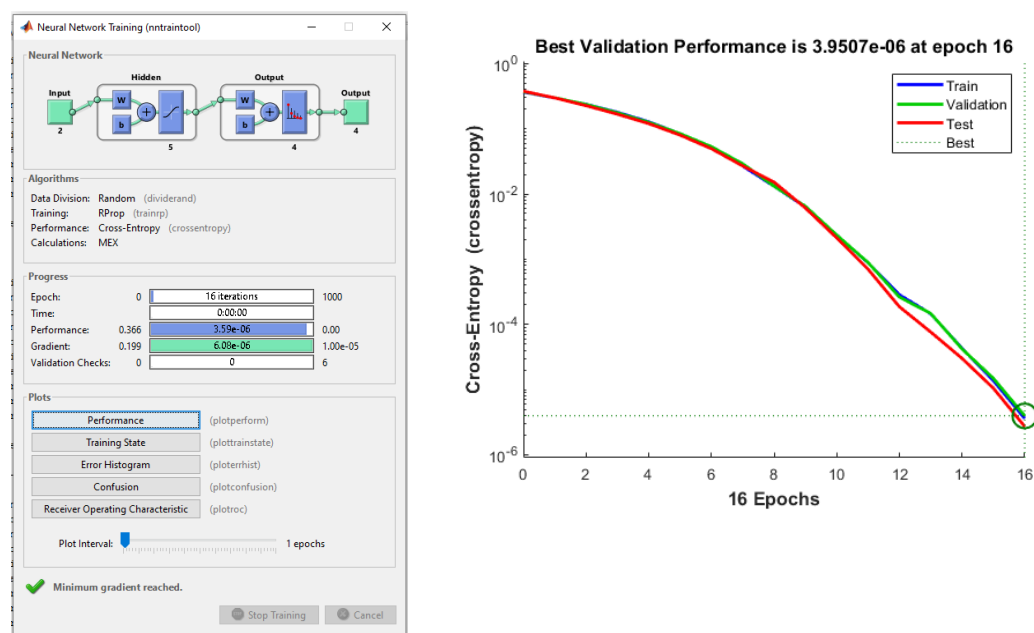
Como era de esperar, en este algoritmo tampoco conseguimos detener el entrenamiento antes de las 1000 iteraciones establecidas como límite, aunque obtenemos mayor rendimiento.

ALGORITMO RESILIENT BACKPROPAGATION

Vemos que empleando este algoritmo, consigue realizar la clasificación en pocas iteraciones, lo que ya es una mejora sobre los dos anteriores.

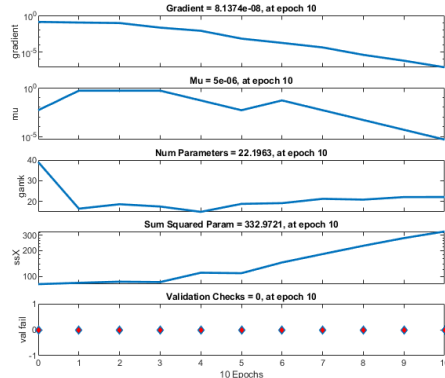
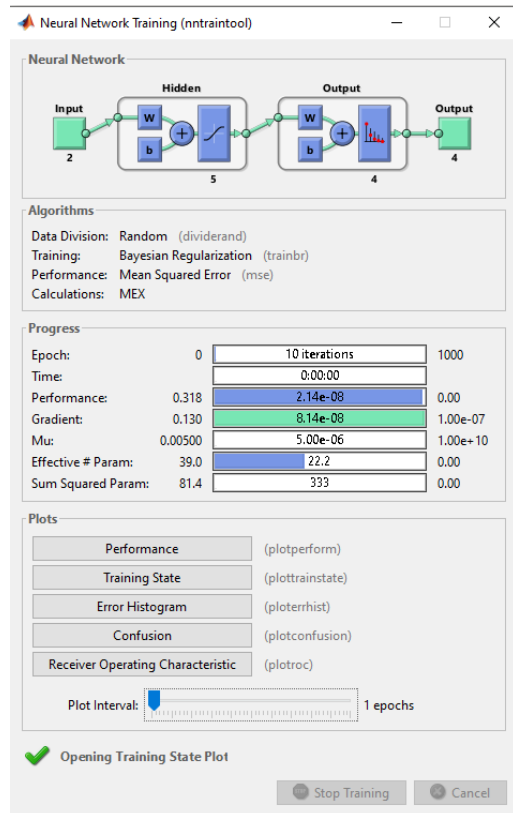


Empleando una distribución de 0.6-0.2-0.2, conseguimos acabar incluso en menos iteraciones, obteniendo el mismo resultado:

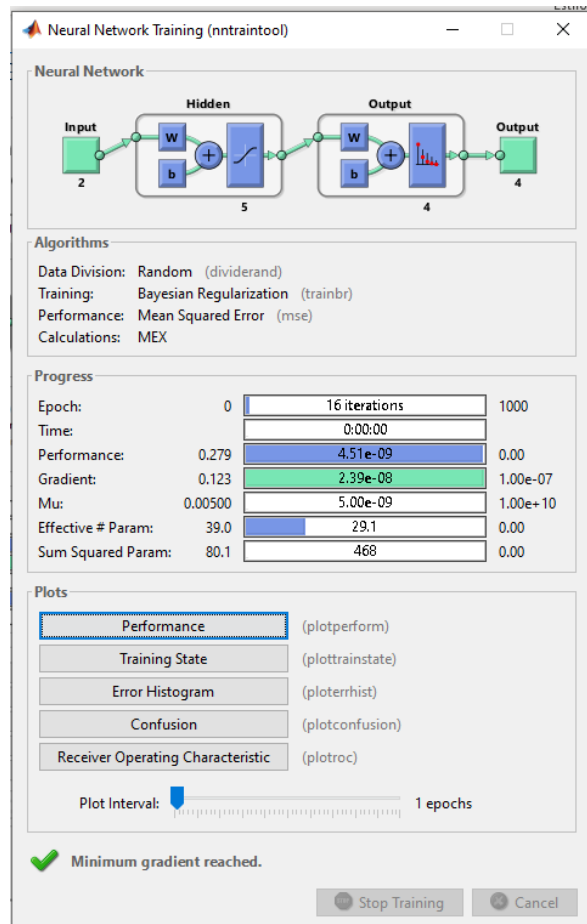


ALGORITMO REGULARIZACIÓN BAYESIANA

Finalmente, obtenemos el mejor rendimiento con el algoritmo de regularización bayesiana, tomando solo 10 iteraciones para llegar al mínimo gradiente, con la distribución típica de 0.7-0.15-0.15:



Observamos que si empleamos distribuciones distintas como 0.6-0.2-0.2 y 0.8-0.1-0.1, obtenemos un incremento de iteraciones, por lo que el ratio de 0.7-0.15-0.15 resulta ser el más óptimo. Aún así, obtenemos el mismo resultado:

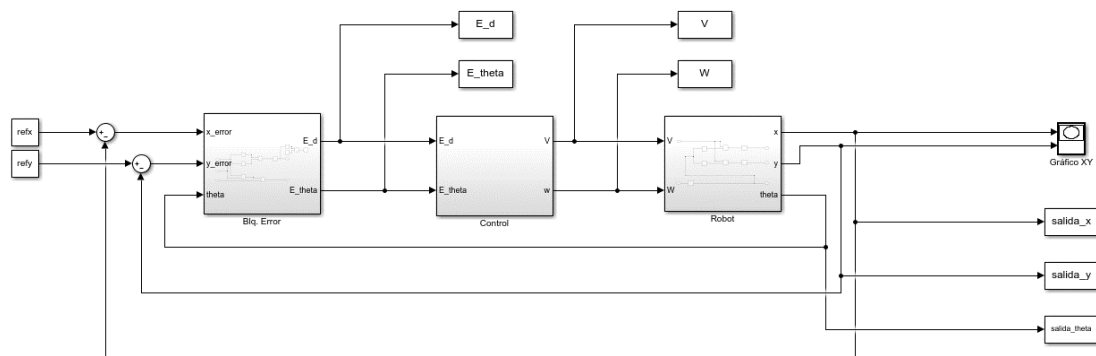


PARTE 2

OBJETIVO Y DESCRIPCIÓN DEL SISTEMA

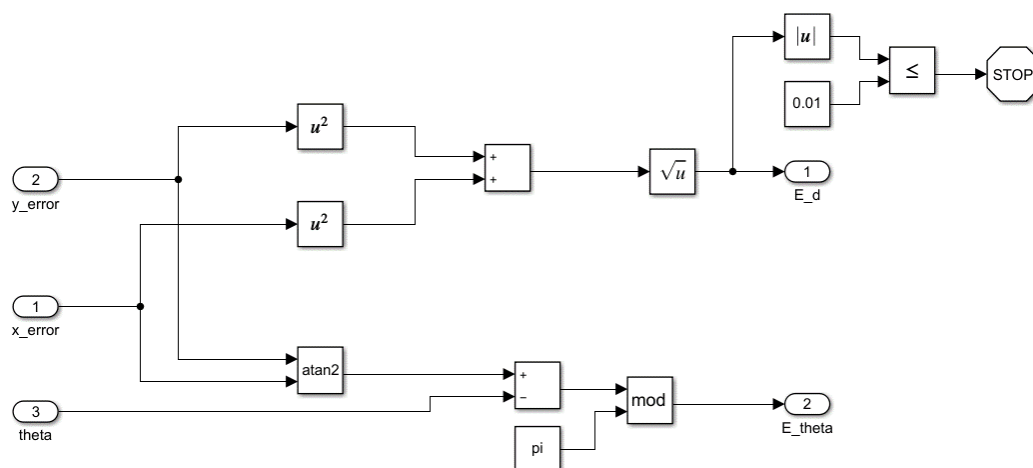
En este primer apartado de la práctica, se nos pide montar el circuito en simulink siguiendo los pasos que indica el enunciado.

La visión general del sistema creado es la siguiente:



Las salidas van al workspace de Matlab para poder trabajar con ellas. Las salidas x e y también se llevan a un bloque de función encargado de pintar la gráfica que generan estas dos señales. Como podemos apreciar, tenemos tres subsistemas.

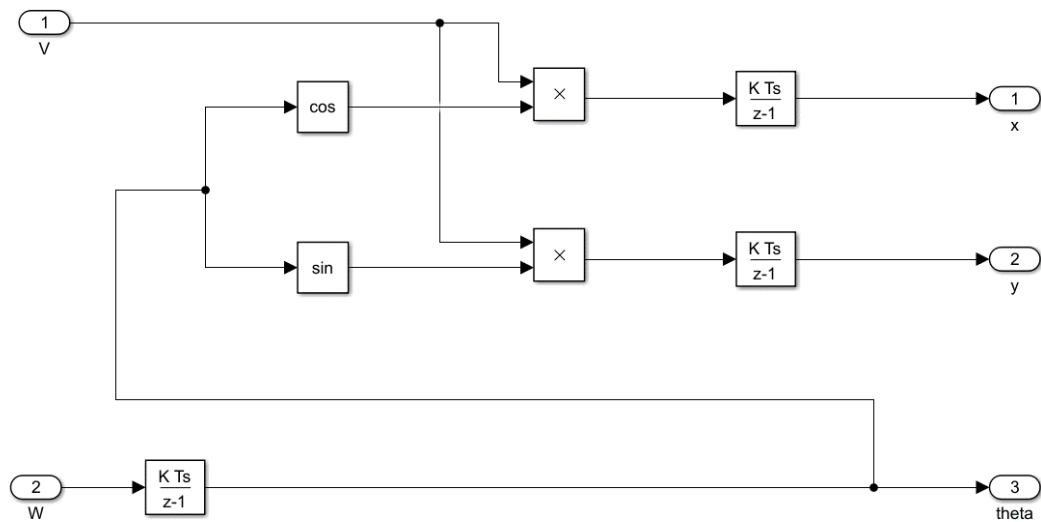
El bloque de error tiene el siguiente circuito en su interior:



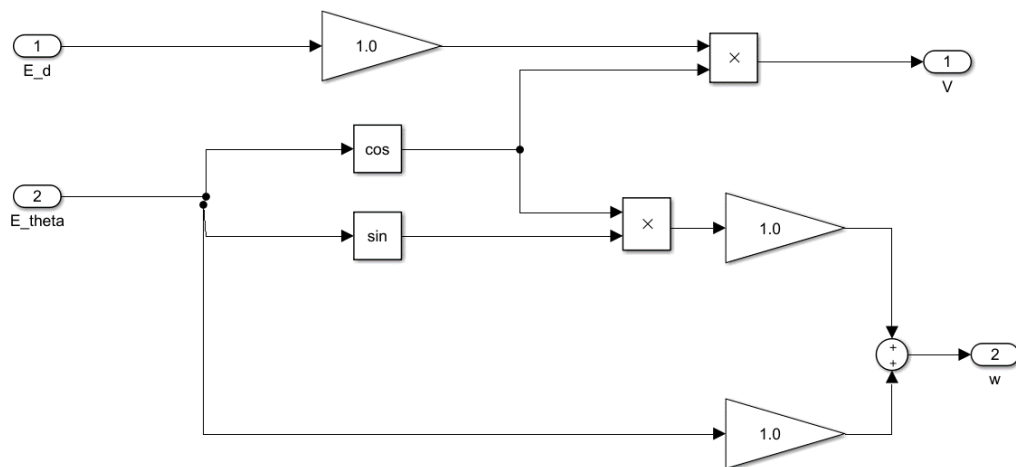
En esta imagen se puede observar la condición de parada del sistema que nos pide el enunciado. Si E_d es menor que 0.01 en cualquier momento, la ejecución parará.

El enunciado también nos pide que el valor de E_theta siempre tenga un valor comprendido en el intervalo $[-\pi, \pi]$. La función atan2 ya trabaja en módulo π por lo que se entiende que con eso valdría para que el valor no saliese de este intervalo, pero por si acaso se ha limitado el valor de la salida E_theta a este intervalo.

El subsistema "Robot" es el que usamos en la práctica 0 y tiene el siguiente circuito interno:



Por último, tenemos el subsistema "Control". Este subsistema nos lo proporciona el enunciado y tiene el siguiente circuito interno:



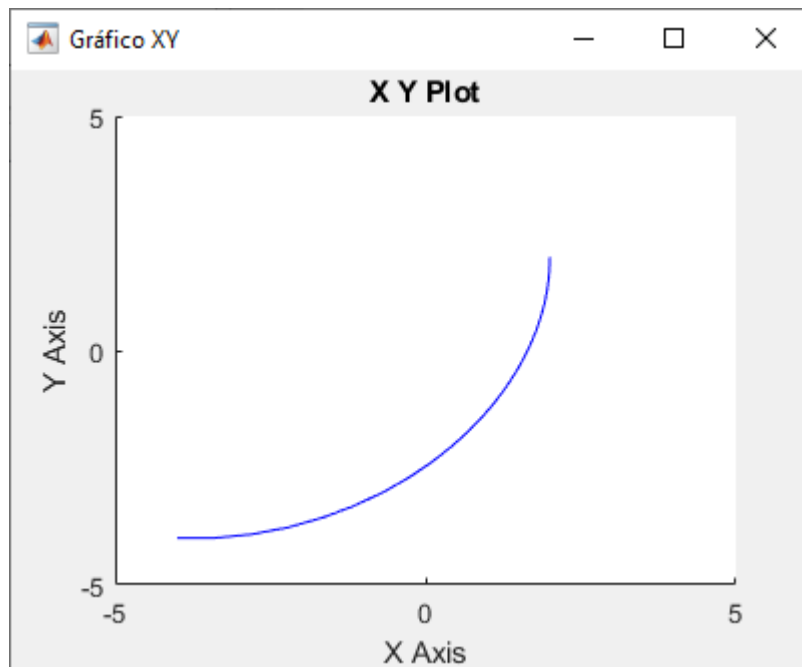
Lo único que queda es configurar las variables de salida hacia el workspace para que funcionen como Structures with Time. Por último, limitamos el tiempo de simulación a 100 segundos, y definimos el tiempo de salto a una variable "Ts" que definiremos en el archivo de script de Matlab.

DESARROLLO DE LA PRÁCTICA

Se nos pide crear un script de Matlab que inicialice las variables Ts, refx y refy que funcionan como parámetros y entradas al sistema que hemos creado en simulink. El código es el siguiente:

```
%Tiempo de muestreo
Ts=100e-3
% Referencia x-y de posicion
refx=2.0;
refy=2.0;
% Ejecutar Simulacion
sim('PositionControl.slx')
```

El bloque que tiene el sistema para pintar gráficos con dos variables nos enseña la siguiente figura una vez ejecutamos el script y el robot se pone en funcionamiento:



Workspace	
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	69x1 double
Ts	0.1000
V	1x1 struct
W	1x1 struct

Como podemos apreciar en el apartado de variables del workspace de Matlab, las salidas a workspace que teníamos en el sistema de simulink consiguen satisfactoriamente sacar las variables al workspace de Matlab.

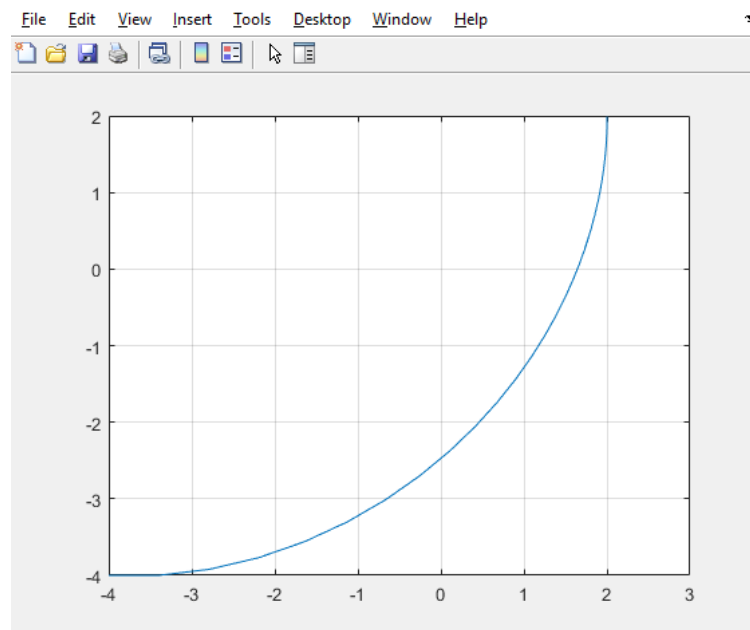
La salida E_d tiene los siguientes valores entre otros:

E_d.signals.values	
	1
1	8.4853
2	8.0722
3	7.6179
4	7.1383

Se nos pide que añadamos las siguientes líneas al script que teníamos:

```
% Mostrar
x=salida_x.signals.values;
y=salida_y.signals.values;
figure;
plot(x,y);
grid on;
```

Esto lo único que hace es sacar un plot con las salidas x e y del sistema, es decir, exactamente lo mismo que hacía el bloque que habíamos implementado en el sistema. La gráfica es la siguiente:



El siguiente paso es realizar 30 simulaciones del sistema, almacenando las salidas y las entradas del bloque de control en arrays. El código es el siguiente:

```
% Generar N posiciones aleatorias, simular y guardar en variables
N=30
E_d_vec=[];
E_theta_vec=[];
V_vec=[];
W_vec=[];
for i=1:N
    refx=10*rand-5;
    refy=10*rand-5;
    sim('PositionControl.slx')
    E_d_vec=[E_d_vec;E_d.signals.values];
    E_theta_vec=[E_theta_vec;E_theta.signals.values];
    V_vec=[V_vec; V.signals.values];
    W_vec=[W_vec; W.signals.values];
end
inputs=[E_d_vec'; E_theta_vec'];
outputs=[V_vec'; W_vec'];
```

Si ahora ejecutamos, podemos ver como el sistema se ejecuta 31 veces (teniendo en cuenta la llamada inicial), cada una de estas ejecuciones se ve reflejada en el gráfico XY generado por el bloque de simulink del sistema.

El objetivo de obtener estos resultados es el de tener una serie de entradas y targets para crear una red neuronal que simule el comportamiento del bloque de control. La red neuronal se genera y se entrena con el siguiente código:

```
% Entrenar red neuronal con x neuronas en la capa oculta
net = feedforwardnet(x);
net = configure(net,inputs,outputs);
net = train(net,inputs,outputs);
```

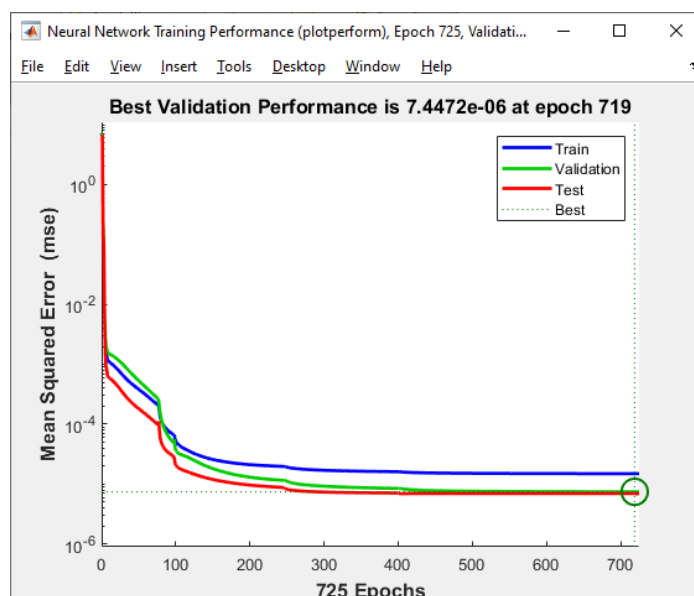
Donde x es el número de neuronas que queremos que tenga la capa oculta de la red. El valor de x hay que hallarlo experimentalmente, para lo que hemos empezado probando en 1 neurona y hemos ido incrementando y comparando los resultados de rendimiento de la red.

El mejor rendimiento de validación para distintos valores de x (en el orden en el que se probaron) es:

1:	0.39 en la época 307.
2:	0.22 en la época 18.
5:	7.4472e-06 en la época 719
10:	0.00035585 en la época 180
7:	0.0054313 en la época 17
6:	0.00060833 en la época 154
4:	0.00016405 en la época 103

De entre todos los valores probados el menor con diferencia es cuando usamos 5 neuronas. También se debe remarcar que los valores de entrada que se le pasan al sistema son valores aleatorios, por lo que los valores pueden variar bastante entre cada ejecución.

La gráfica que modela los comportamientos de entrenamiento, prueba y validación para x=5 es la siguiente:

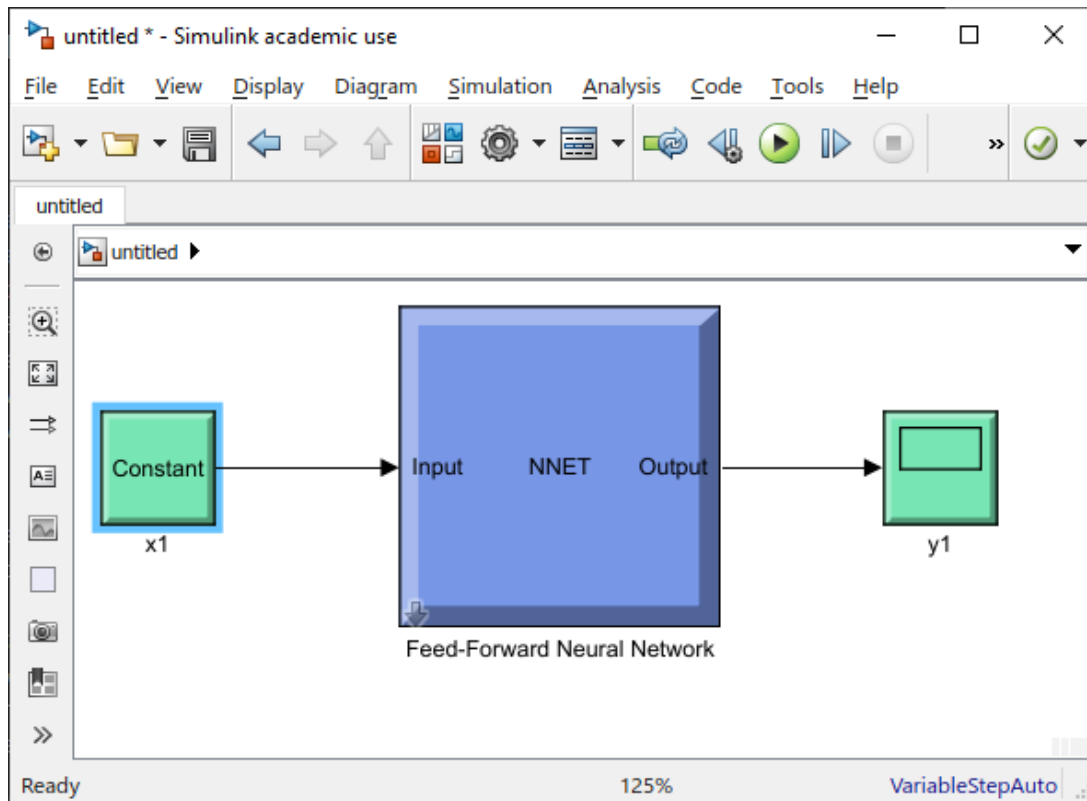


El valor del MSE es lo suficientemente bajo como para considerarse que la red neuronal emula el comportamiento de la caja de control satisfactoriamente, por lo que el entrenamiento ha sido bueno.

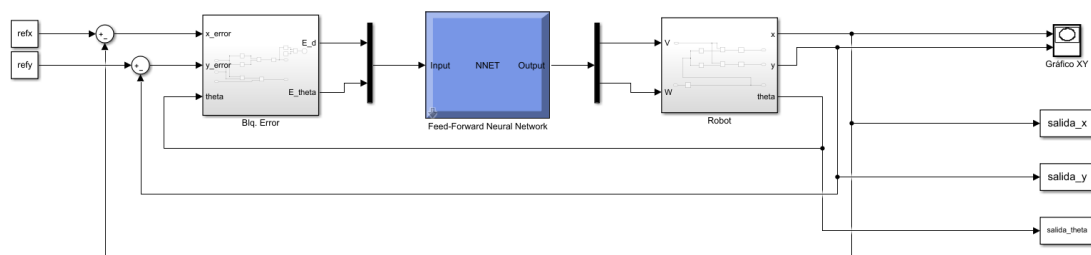
Ahora se nos pide generar la red neuronal con la línea de código:

```
% Generar bloque de Simulink con el controlador neuronal  
gensim(net,Ts)
```

Esto nos abre una nueva ventana de simulink donde tenemos nuestra red neuronal que simula el comportamiento de la caja negra de control:



Creamos un nuevo esquema en simulink sustituyendo el bloque de control por la red neuronal y usando multiplexores y demultiplexores en las entradas y salidas de la red. La visión general del sistema es la siguiente:



Para hacer la comparativa entre ambos sistemas, se crea un nuevo script de Matlab con el siguiente código:

```
clear all; close all;  
%Tiempo de muestreo  
Ts=100e-3  
  
% N valores diferentes de refx/refy
```

```

% Warning: se van a generar tantas graficas comparativas de trayectoria
% como el valor de N (N = 5 -> 5 graficas)

N = 5

arrayErrores_x = [];
arrayErrores_y = [];

for i=1:N
    % Generamos los valores aleatorios de refx/refy
    refx=10*rand-5;
    refy=10*rand-5;
    sim('PositionControl.slx') % Simulamos con el sistema original
    x = salida_x.signals.values; % Almacenamos x
    y = salida_y.signals.values; % Almacenamos y

    sim('PositionControlNet.slx') % Simulamos con el sistema con red neuronal
    x_net = salida_x.signals.values; % Almacenamos x
    y_net = salida_y.signals.values; % Almacenamos y

    % Pintamos la grafica resultante comparando ambas trayectorias
    figure(i);
    hold on;
    original = plot(x,y);
    red = plot(x_net, y_net);
    hold off;
    grid on;
    legend([original red], {'Control original', 'Control red neuronal'});

    % Calculamos el error entre ambos valores y lo almacenamos en un array
    % Cogemos la longitud del vector de trayectoria mas pequeno para no
    % calcular el error con elementos que no tengan companero
    errorParcial_x = [];
    errorParcial_y = [];
    for j=1:min(length(x),length(x_net))
        errorParcial_x = [errorParcial_x ; abs(x(j) - x_net(j))];
        errorParcial_y = [errorParcial_y ; abs(y(j) - y_net(j))];
    end
    arrayErrores_x = [arrayErrores_x ; errorParcial_x];
    arrayErrores_y = [arrayErrores_y ; errorParcial_y];
end

figure(i+1);
plotErrores = plot(arrayErrores_x, arrayErrores_y, 'o');
grid on;
title('Valores del error entre ambos controles');
legend(plotErrores, {'Valor del error'});
xlabel('Error en x');
ylabel('Error en y');

```


Lo que hace este código es entrar en un bucle de longitud N. Dentro de este bucle se calculan unos valores aleatorios de $refx$ y $refy$ como ya hacíamos en los scripts anteriores.

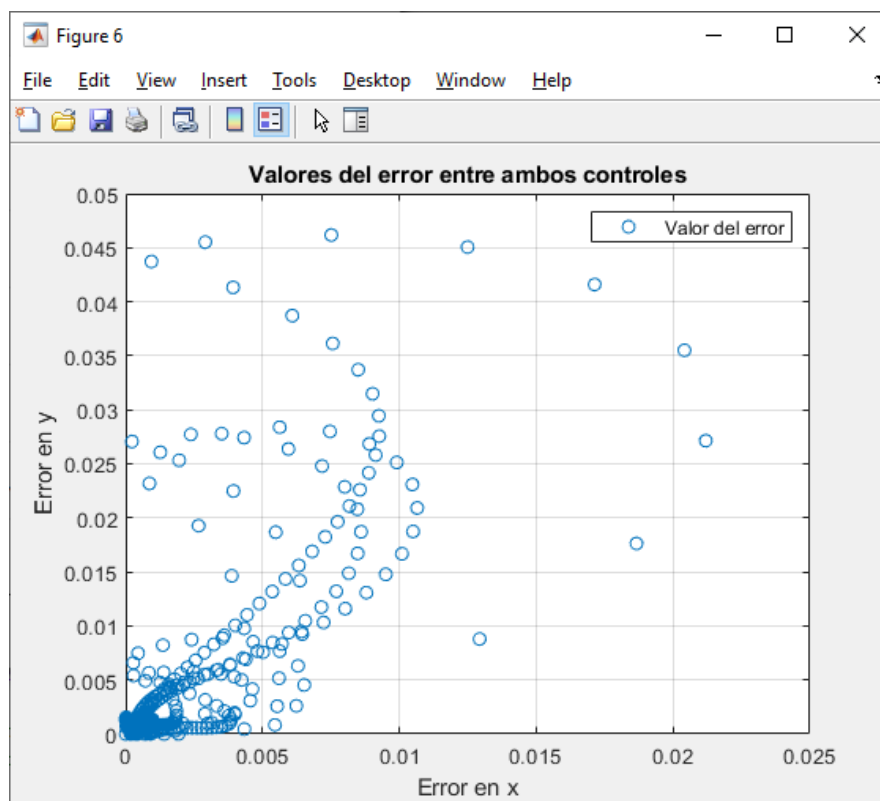
Con estos valores aleatorios, llamamos a ambas simulaciones y almacenamos en variables las salidas x e y para ambos sistemas.

Una vez hecho esto pintamos estas salidas x e y para que se puedan comparar ambas trayectorias. Con estos arrays de posiciones entramos en un segundo bucle, en el cual se van a calcular los errores para ambos vectores de posición (x e y). Debido a que ambas simulaciones pueden dar lugar a diferente número de salidas para estas posiciones, se coge la longitud del array de posiciones más pequeño y se compara hasta ese valor, desestimando el resto de los valores de posición que devuelve la simulación.

Estos errores se van añadiendo a un array de errores parciales donde se almacena el valor absoluto de la resta entre las posiciones de ambas simulaciones. Una vez hemos calculado todos los errores parciales para una iteración, se concatenan al array de errores totales para cada una de las coordenadas.

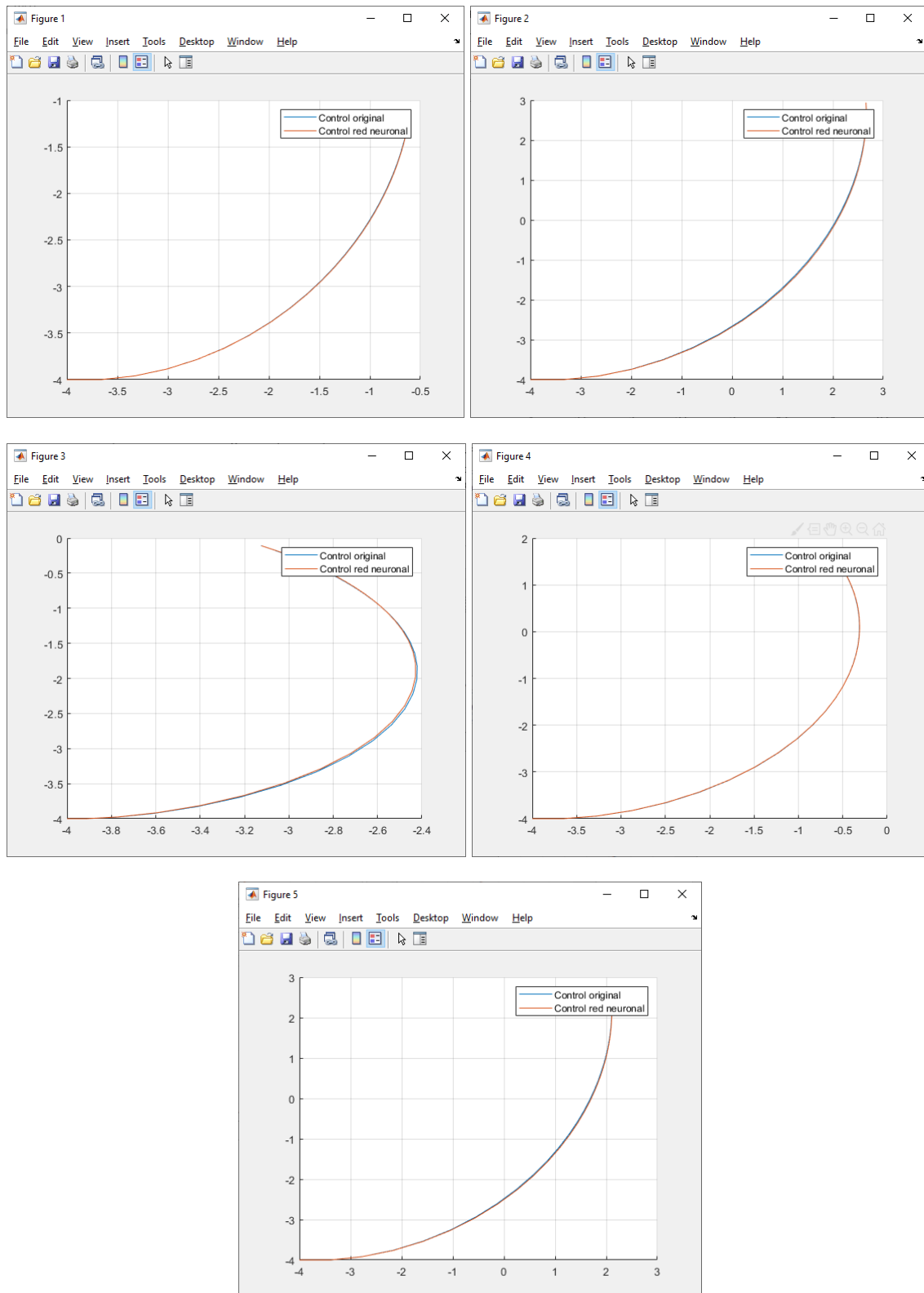
Cuando terminan todas las iteraciones, se pinta un plot en el cual se ven reflejados los valores de los errores para todas las posiciones (X , Y) de cada una de las iteraciones.

Probamos a ejecutar realizando 5 iteraciones el programa nos genera la siguiente gráfica de errores:



Como podemos observar, la gran mayoría de puntos del error se encuentran muy próximos a 0. Los puntos más alejados no llegan a una diferencia en valor absoluto de 0.05, por lo que el error es bastante bajo entre estos sistemas. Esto se puede corroborar si echamos un vistazo a las gráficas comparativas de las trayectorias de ambos sistemas.

Para la gráfica de error anterior, tenemos estas gráficas comparativas de trayectorias:

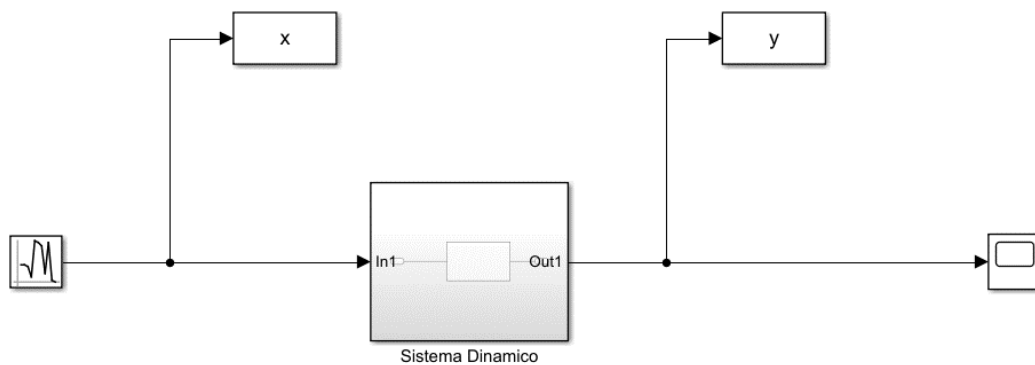


Lo cual corrobora lo que ya indicaba los datos de los errores. Hay algunos puntos donde se distancian un poco más, que son los puntos que darían estos errores más elevados en la gráfica de errores; pero, por lo general, ambas trayectorias son muy parecidas.

PARTE 3

EJERCICIO 1

El primer ejercicio de esta tercera parte de la práctica nos pide entrenar una red neuronal en paralelo a un sistema dinámico para que la red se comporte como el sistema. Para ello se monta el sistema que se pide en el enunciado con el bloque proporcionado, quedando de la siguiente manera:



Se ejecuta el siguiente código:

```
clear all; close all;
% Generacion de datos de simulacion
Ts = 0.1;
sim('test_bench.slx')
inputs=x.signals.values';
outputs=y.signals.values';
```

Y como se puede comprobar, simulink es capaz de enviar los datos al workspace correctamente:

Workspace	
Name ▲	Value
inputs	1x301 double
outputs	1x301 double
tout	301x1 double
Ts	0.1000
x	1x1 struct
y	1x1 struct

Se genera una red neuronal NARX con 5 neuronas, se transforman los arrays para que la red pueda trabajar con ellos, se entrena la red neuronal y se convierte en una red recursiva con el siguiente código:

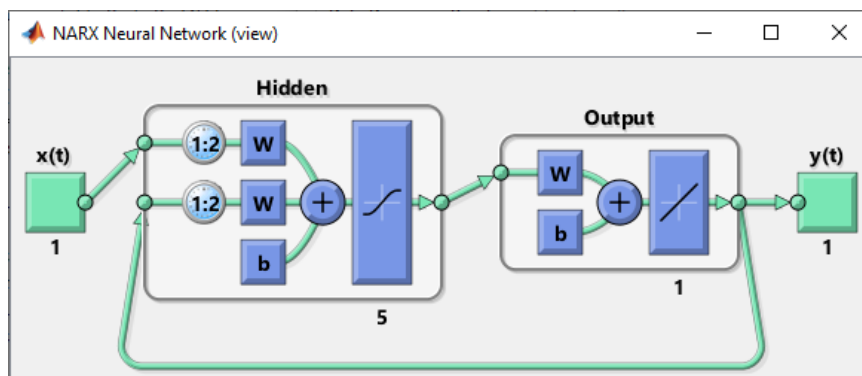
```
% Definición del modelo NARX
N=5;
net = narxnet(1:2,1:2,[N]);

% Se preparan los arrays
nT=size(inputs,2);
inputsc=mat2cell(inputs,1,ones(nT,1));
outputsc=mat2cell(outputs,1,ones(nT,1));

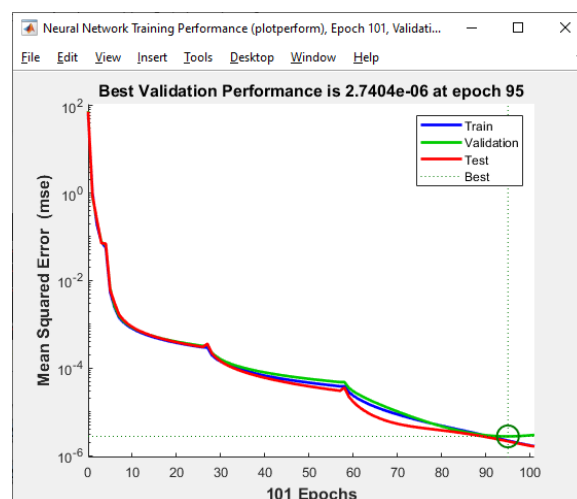
[x,xi,ai,t] = preparets(net,inputsc,{},outputsc);

% Se entrena la red NARX
net = train(net,x,t,xi,ai);
net = closeLoop(net);
view(net)
```

Al ejecutar este script, Matlab nos presenta con la siguiente figura que representa la red neuronal creada:



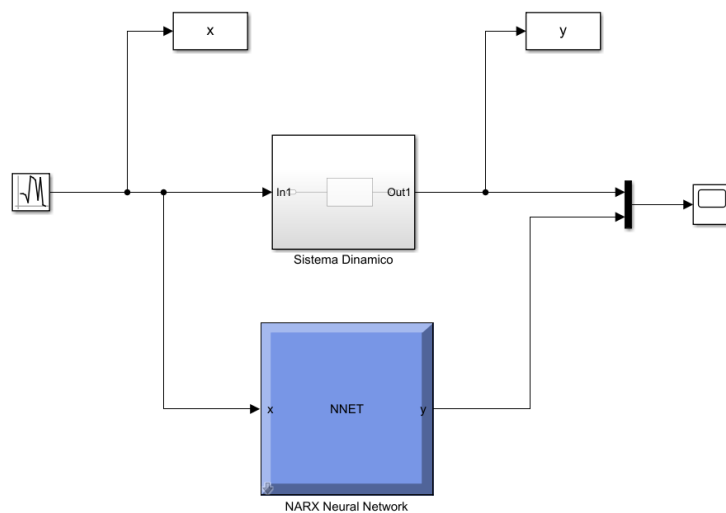
Esta red tiene una gráfica de error de entrenamiento, validación y prueba de la siguiente forma:



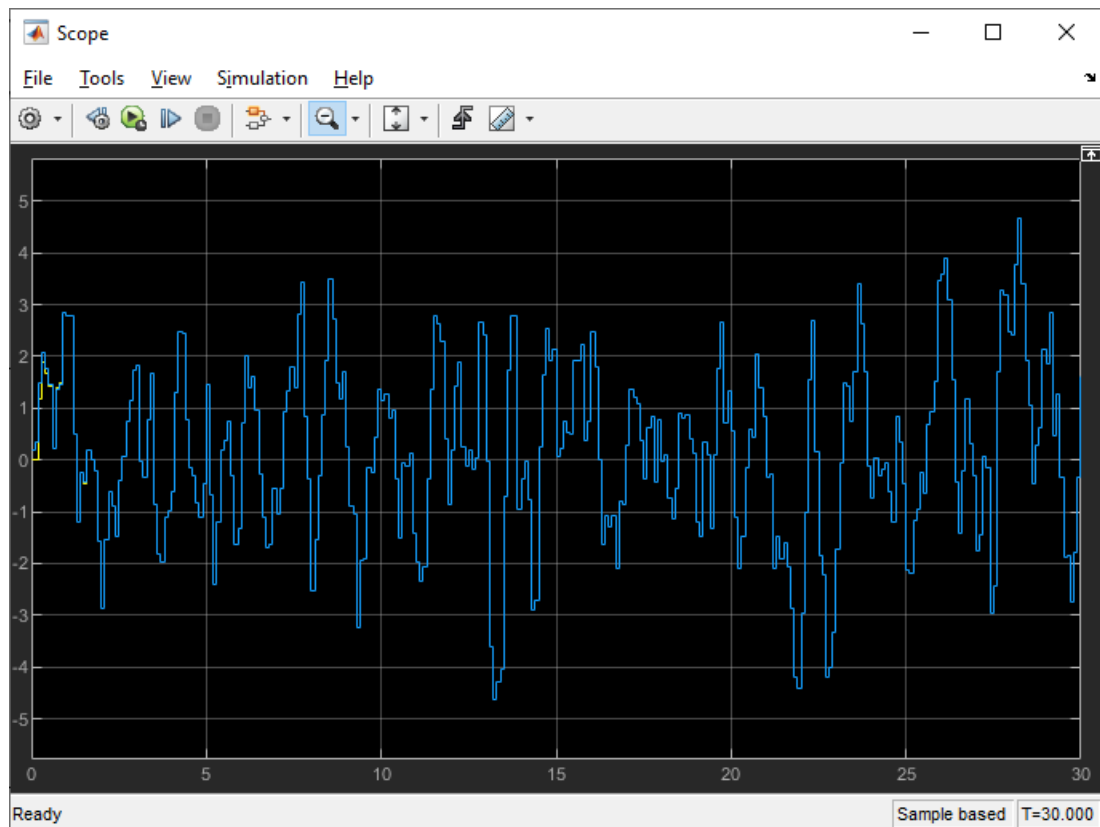
Son buenos niveles de error y la red neuronal podría sustituir perfectamente al sistema dinámico que nos da el enunciado.

Por último, se nos pide generar esta red neuronal que hemos creado y añadirla en paralelo a un nuevo archivo simulink para comparar el error entre el sistema original y la red neuronal.

Para ello se construye el siguiente modelo en simulink:



Y el error entre la red neuronal y el sistema dinámico nos lo proporciona el bloque de scope con la entrada multiplexada. La señal que recibe este scope es la siguiente:

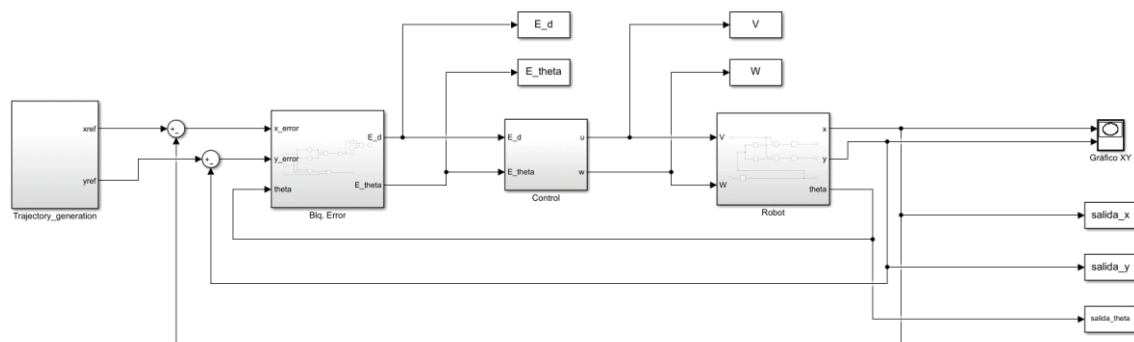


Como se puede apreciar, en el inicio las señales generadas por el sistema y la red neuronal son algo diferentes ya que la red neuronal no tiene entrenamiento suficiente como para emular el comportamiento del sistema a la perfección, pero pasados los primeros dos segundos las diferencias entre el sistema original y la red neuronal son mínimas o incluso nulas, generándose señales muy similares.

EJERCICIO 2

Para este ejercicio se toma como base el fichero PositionControl.slx creado para la parte 2 de esta práctica. De este fichero vamos a cambiar las señales refx y refy por un generador de trayectoria que nos proporciona el enunciado y el bloque de control por otro especialmente creado para controlar trayectorias también proporcionado por el enunciado.

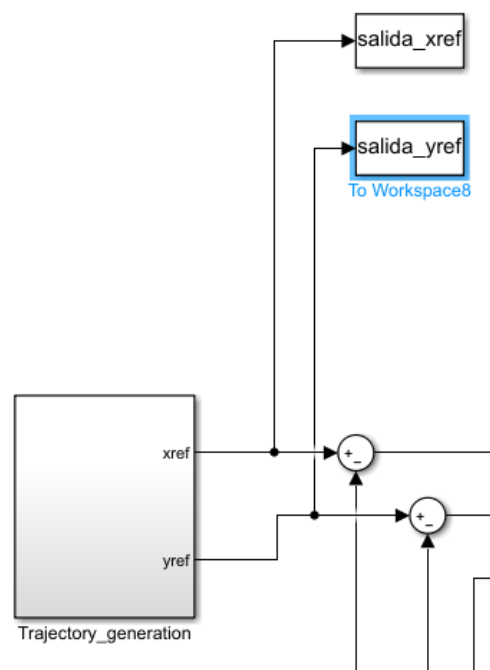
La visión general del sistema es la siguiente:



Con esto llegamos a la parte 2 del ejercicio, en el cual se nos pide el desarrollo de un sistema tomando esto como base.

El primer apartado nos pide comparar la trayectoria que toma el robot con la trayectoria que genera el bloque generador. Para ello conectamos dos bloques que envían datos al workspace que recogen los valores de refx y refy respectivamente y los tratamos desde el workspace.

Los bloques son los siguientes:



El código para que esto funcione y se pueda visualizar la diferencia en trayectoria es el siguiente:

```
clear all; close all;

% Inicializamos las variables necesarias para el sistema
Ts = 0.1;

x_0 = 0;
y_0 = 0;
th_0 = 0;

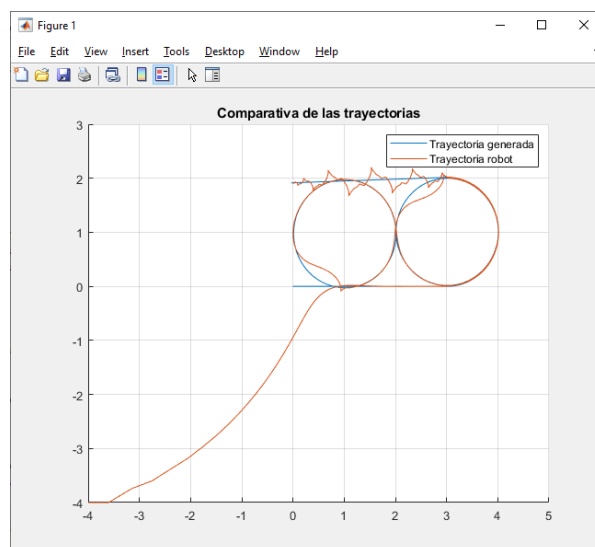
% Hacemos la simulacion
sim('TrajectoryControl.slx');

% Recogemos los datos de trayectoria que envia el modelo desde simulink
trayectoria_x = salida_xref.signals.values';
trayectoria_y = salida_yref.signals.values';

x = salida_x.signals.values';
y = salida_y.signals.values';

% Pintamos ambas trayectorias en un figure
figure(1);
hold on;
tray_original = plot(trayectoria_x, trayectoria_y);
tray_robot = plot(x, y);
hold off;
grid on;
legend([tray_original tray_robot], {'Trayectoria generada', 'Trayectoria robot'});
title('Comparativa de las trayectorias');
```

Y la gráfica comparativa entre ambas trayectorias es la siguiente:



Tras esto, para el apartado c, se prepara el siguiente script con la red neuronal:

```
clear all; close all;
% Generacion de datos de simulacion
Ts = 0.1;
x_0 = 0;
y_0 = 0;
th_0 = 0;
sim('TrajectoryControl.slx')
inputs=[E_d.signals.values';
        E_theta.signals.values'];
outputs=[V.signals.values';
         W.signals.values'];

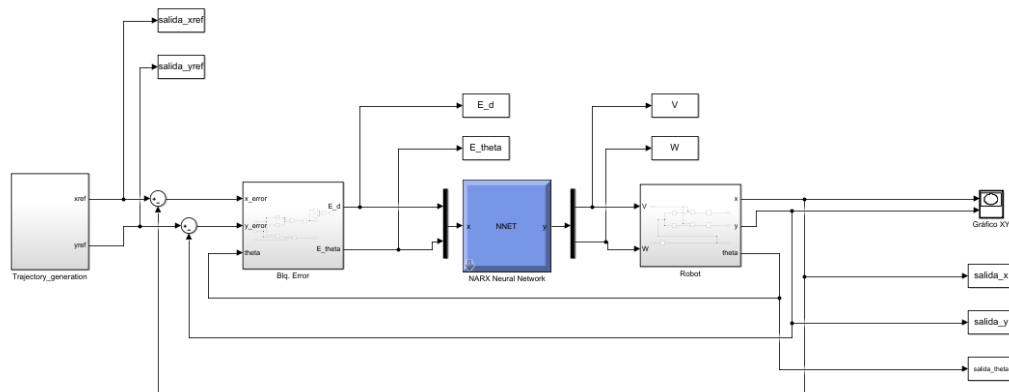
% Definicion del modelo NARX
N=15;
% 2 retardos en la entrada y uno en la salida
net = narxnet(0:1,1,5);

% Se preparan los arrays
nT=size(inputs,2);
inputsc=mat2cell(inputs,2,ones(nT,1));
%inputsc = num2cell(inputs,1);
outputsc=mat2cell(outputs,2,ones(nT, 1));
%outputsc = num2cell(outputs,1);
[x,xi,ai,t] = preparets(net,inputsc,{},outputsc);

% Se entrena la red NARX
net = train(net,x,t,xi,ai);
net = closeloop(net);
view(net)

% Se genera un bloque simulink con la red neuronal entrenada
gensim(net, Ts);
```


Generamos la red neuronal entrenada con gensim, y la añadimos en el TrajectoryControlNet sustituyendo el control sobre el robot:



Para evaluar cuántas neuronas tendrá la capa oculta, se hace mediante experimentación, con el siguiente script:

```
clear all; close all;

% Inicializamos las variables necesarias para el sistema
Ts = 0.1;

x_0 = 0;
y_0 = 0;
th_0 = 0;

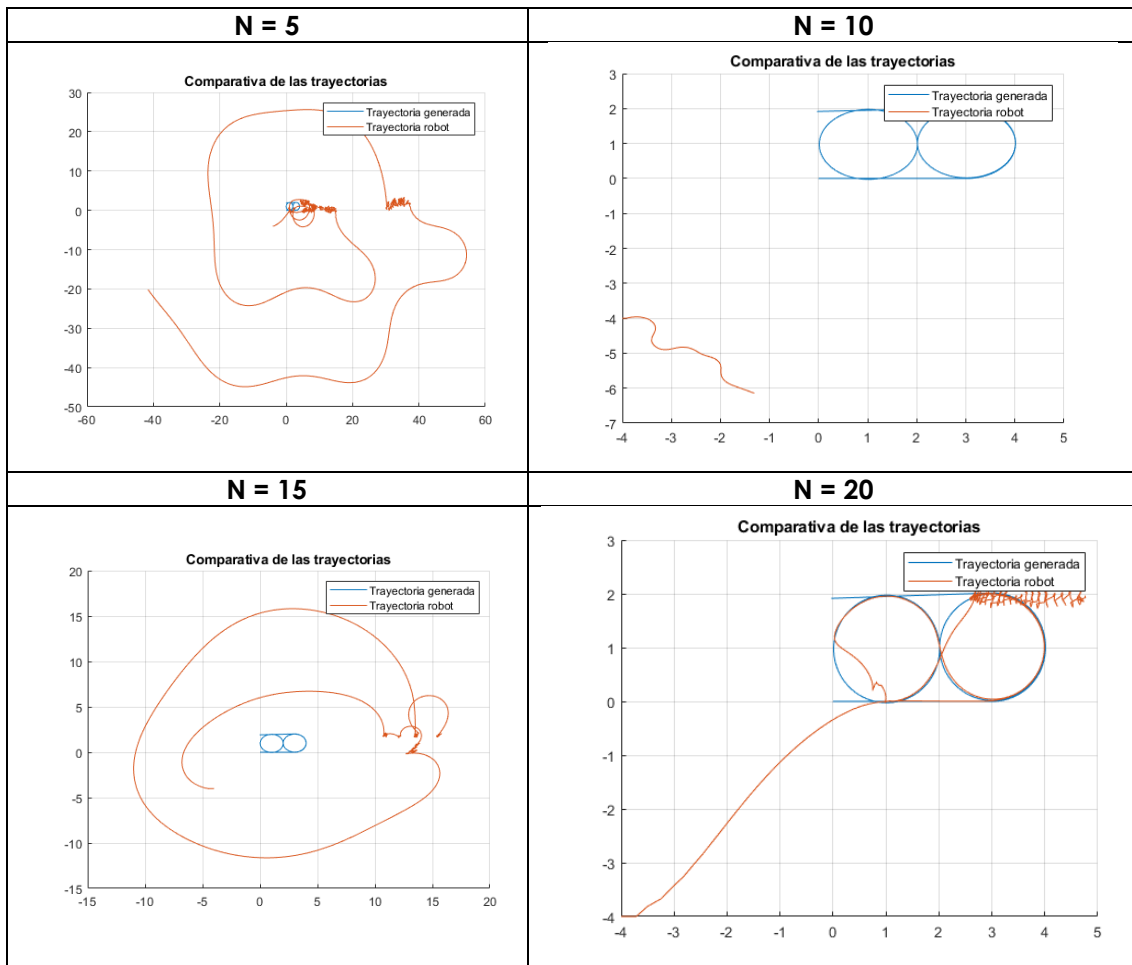
% Hacemos la simulacion
sim('TrajectoryControlNet.slx');

% Recogemos los datos de trayectoria que envia el modelo desde simulink
trayectoria_x = salida_xref.signals.values';
trayectoria_y = salida_yref.signals.values';

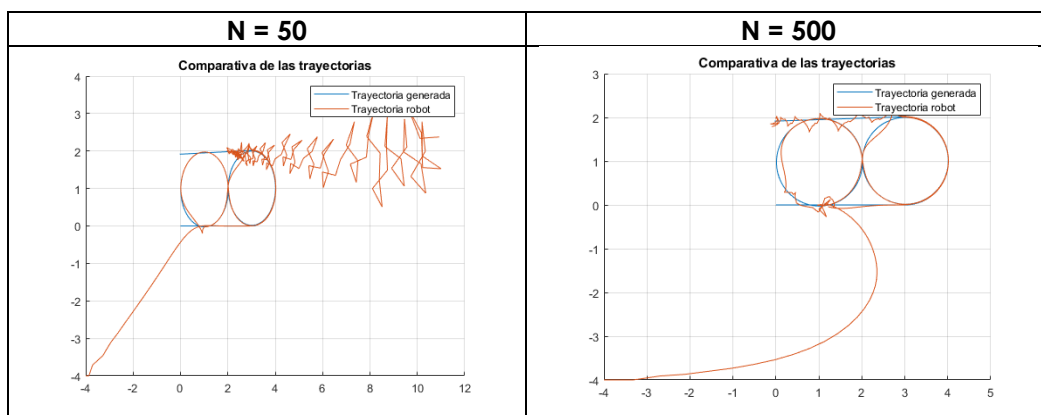
x = salida_x.signals.values';
y = salida_y.signals.values';

% Pintamos ambas trayectorias en un figure
figure(1);
hold on;
tray_original = plot(trayectoria_x, trayectoria_y);
tray_robot = plot(x, y);
hold off;
grid on;
legend([tray_original tray_robot], {'Trayectoria generada', 'Trayectoria robot'});
title('Comparativa de las trayectorias');
```

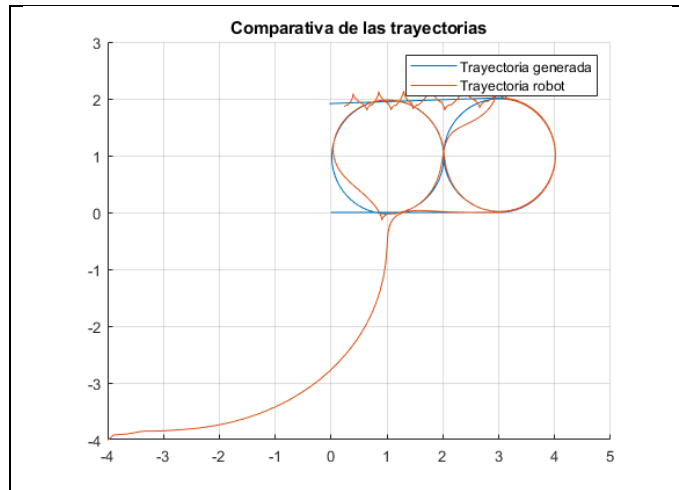
Se ha probado con N's de 5, 10, 15 y 20, obteniendo los siguientes resultados:



Observamos que conforme introducimos más neuronas en la capa intermedia, más se consigue aproximar la trayectoria controlada por la red neuronal a la trayectoria original. Seguimos incrementando:



Finalmente, con un $N = 50000$, conseguimos aproximar mediante la red neuronal de forma bastante cercana la trayectoria del control con el error inducido:



Podemos concluir qué a mayor fuerza bruta a base de cantidad de neuronas en la capa intermedia, se realizará una aproximación más cercana a la ruta seguida por el control con el error inducido.

Se prepara el siguiente script para comparar las 3 trayectorias: la original, la del robot, y la de la red neuronal:

```
clear all; close all;

% Inicializamos las variables necesarias para el sistema
Ts = 0.1;

x_0 = 0; y_0 = 0; th_0 = 0;

% Simulamos con el robot
sim('TrajectoryControl.slx');

% Recogemos los datos de trayectoria que envia el modelo desde simulink
trayectoria_x = salida_xref.signals.values';
trayectoria_y = salida_yref.signals.values';

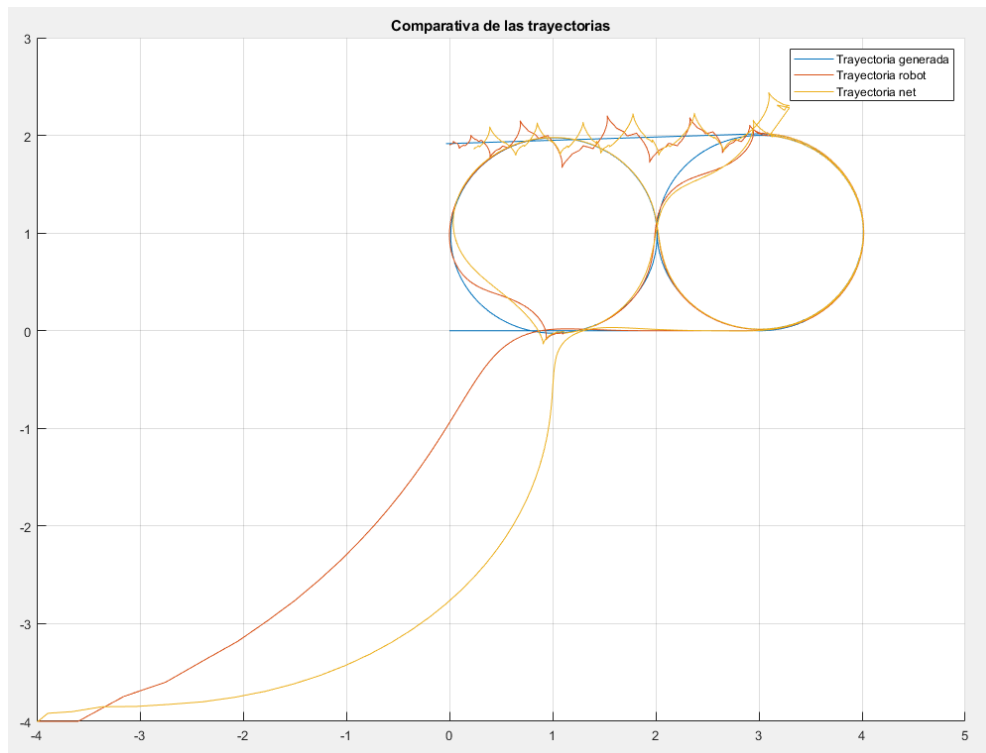
x_robot = salida_x.signals.values';
y_robot = salida_y.signals.values';

% Simulamos con la red neuronal
sim('TrajectoryControlNet.slx');

x_net = salida_x.signals.values';
y_net = salida_y.signals.values';

% Pintamos ambas trayectorias en un figure
figure(1);
hold on;
tray_original = plot(trayectoria_x, trayectoria_y);
tray_robot = plot(x_robot, y_robot);
tray_net = plot(x_net, y_net);
hold off;
grid on;
legend([tray_original tray_robot tray_net], {'Trayectoria generada', 'Trayectoria robot', 'Trayectoria net'});
title('Comparativa de las trayectorias');
```

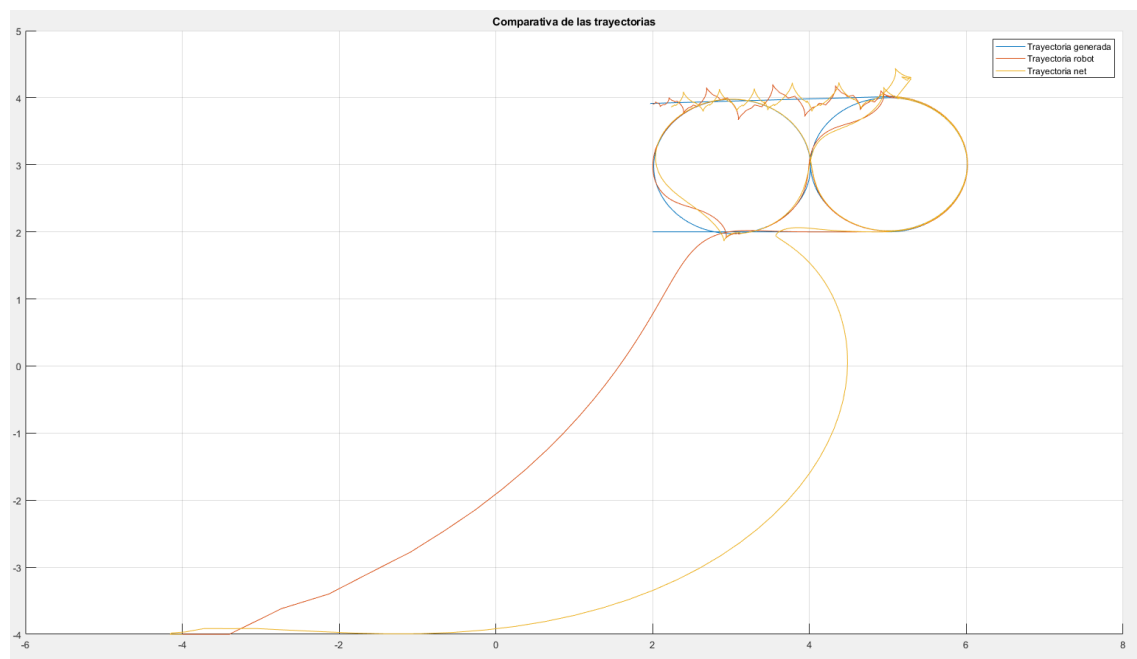
Al ejecutarlo con las variables $x_0 = 0$, $y_0 = 0$, se genera el siguiente gráfico:



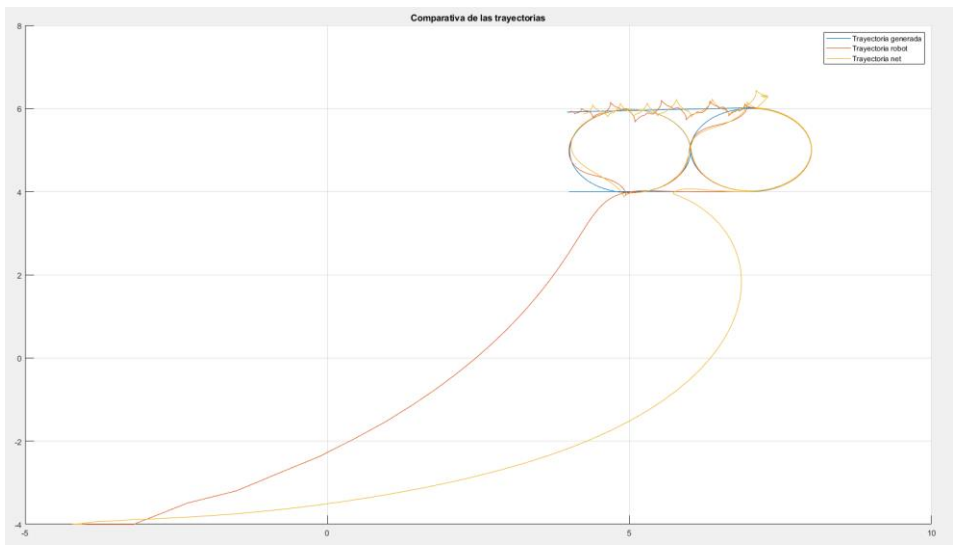
Observamos que la gráfica de la red neuronal se aproxima bastante a la trayectoria del robot – de la cual se ha entrenado –, y pese a ciertos picos y sobresaltos, consigue aproximarse adecuadamente a la trayectoria original.

Probamos a modificar estas variables para estudiar como afecta a la trayectoria:

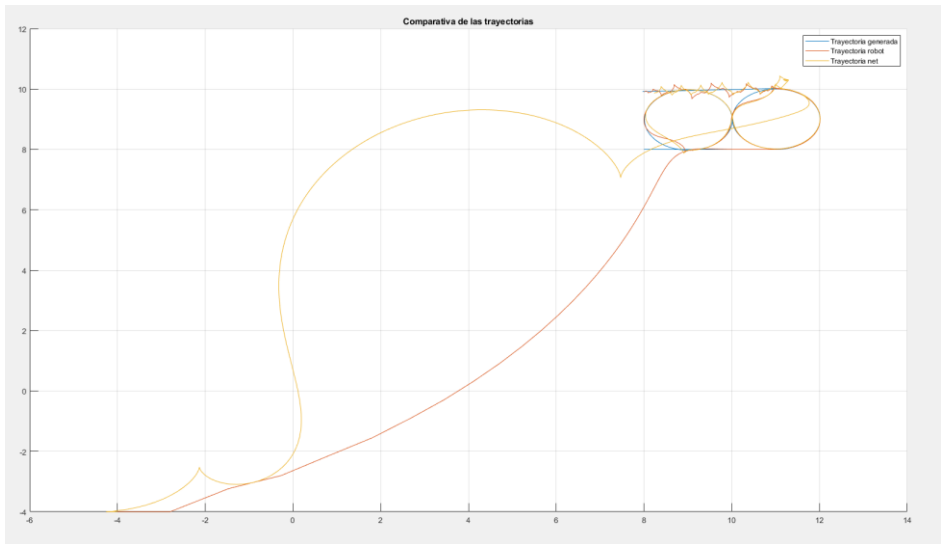
Para $x_0 = 2$, $y_0 = 2$:



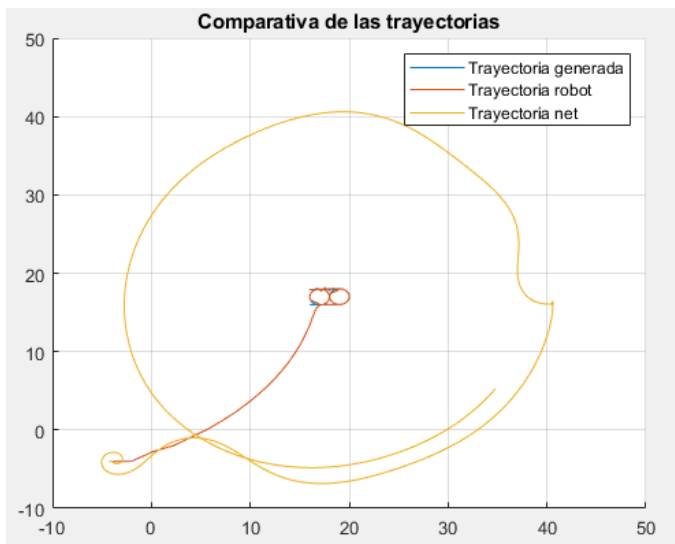
Para $x_0 = 4, y_0 = 4$:



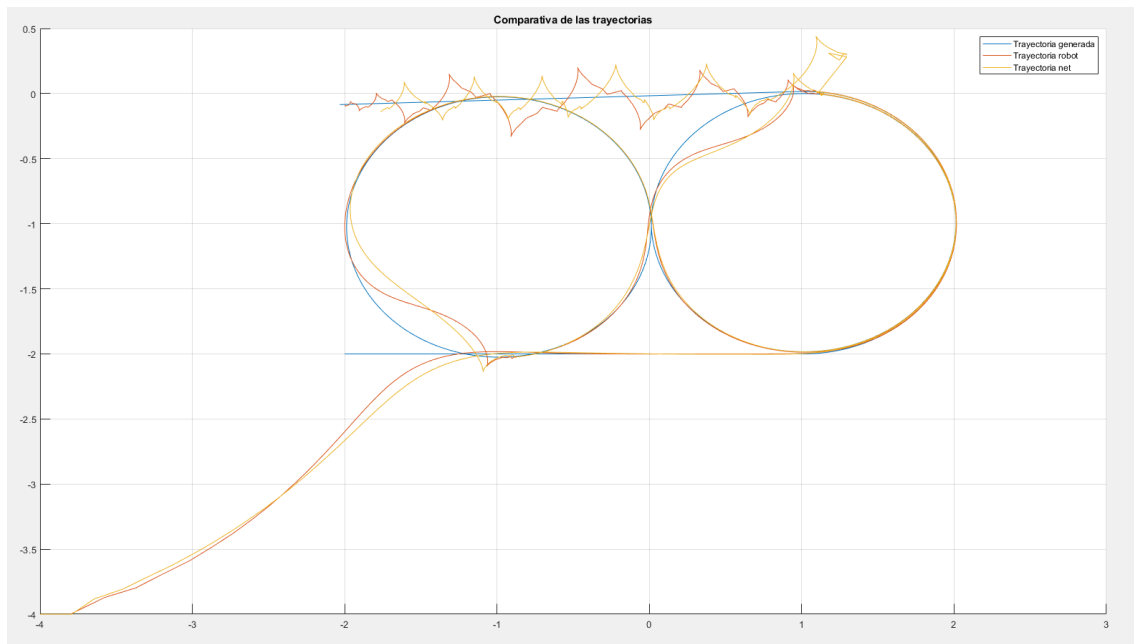
Para $x_0 = 8, y_0 = 8$:



Para $x_0 = 16, y_0 = 16$:



Para $x_0 = -2$, $y_0 = -2$:



Para valores inferiores a -3, el robot para su ejecución por lo que no se generan trayectorias.

Como podemos observar, según se van aumentando los valores de x_0 e y_0 la trayectoria del robot con el controlador original no varía, pero la trayectoria del robot con la red neuronal se va alejando más y más de la trayectoria original. Por el contrario, tomando valores inferiores a 0 como son -1, -2 o -3, la trayectoria mejora y se acerca más a la trayectoria del robot original.

El código presentado tiene las variables x_0 e y_0 inicializadas a 0, generando la trayectoria que corresponde a la primera de estas gráficas.

Vemos que es muy útil entrenar dicha red para un control de caja negra cuyos detalles de implementación son desconocidos. Observamos que existe cierto carácter no determinista a la hora de generar una red y entrenarla con los mismos argumentos, por lo que sería interesante crear varios modelos entrenados de la misma forma a la hora de realizar un estudio real, y comparar resultados entre estas redes antes de implementarlas.