2 Redes Neuronales Unidireccionales: Ejercicios en Matlab y Python

OBJETIVOS

- Implementar una red unidireccional en Matlab
- Crear y entrenar una red unidireccional para aproximar funciones
- Utilizar redes unidireccionales como procedimiento de clasificación de datos
- Comprender el funcionamiento del método de validación cruzada en el entrenamiento de redes neuronales
- Desarrollar la primera red unidireccional para clasificar datos mediante Python

2.1 Aproximación de una función mediante una Red Unidireccional

El objetivo de este capítulo consiste en programar una aproximación de una función no lineal mediante una red neuronal unidireccional (FeedForwardNedwork) con objeto

de entender las capacidades de estas para aprender el comportamiento de funciones complejas a partir de un conjunto de datos que se utilizan para su aprendizaje. Para ello se hará uso de los comandos oportunos que nos proporciona Matlab para la creación, entrenamiento y funcionamiento de una red unidireccional.

En primer lugar vamos a considerar una función muy sencilla como la función seno. Obviamente no tiene sentido práctico aproximar una función tan sencilla como la función seno. Tomamos como ejemplo esta función (con una clara componente no lineal) con objeto de ver cómo podemos utilizar una red unidireccional para aproximar su comportamiento conociendo únicamente un conjunto de valores (entrada/salida). Para ello se tomarán como datos de entrenamiento de la red un conjunto de valores de la función seno dentro de un intervalo determinado. En nuestro caso utilizaremos como intervalo de los datos (-1, 1).

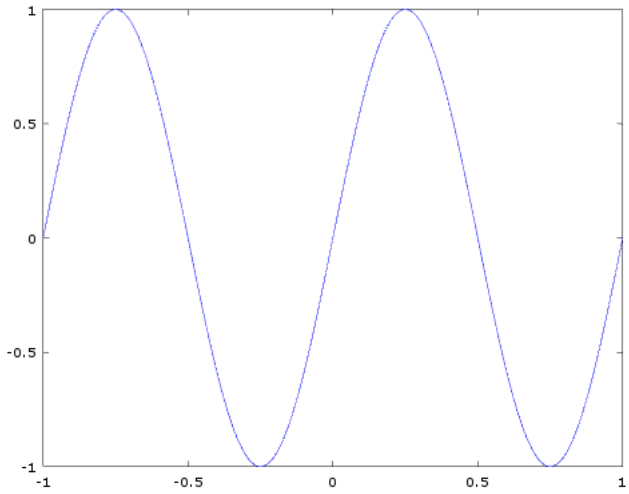

Figura 2.1: Función seno a aproximar

El objetivo pues consiste en implementar una red unidireccional de manera tal que aprenda, a partir de los datos que le suministremos a la red, cual es el comportamiento de esa función (completamente desconocida para la red).

Los pasos que se deberán realizar son los siguientes:

1. Cargar los valores tanto de entrada como de salida de esta red. Es decir cogeremos un conjunto de datos representativos de esta función que son los que utilizaremos con objeto de que la red aprenda su comportamiento.

```
P = -1:0.01:1;
T = sin(2*pi*P);
```

Training Function	Algorithm
trainlm	Levenberg-Marquardt
trainbr	Bayesian Regularization
trainbfg	BFGS Quasi-Newton
trainrp	Resilient Backpropagation
trainscg	Scaled Conjugate Gradient
traincgb	Conjugate Gradient with Powell/Beale Restarts
traincgp	Polak-Rubiere Conjugate Gradient
trainoss	One Step Secant
traingdx	Variable Learning Rate Gradient Descent
traingdm	Gradient Descent with Momentum
traingd	Gradient Descent

Tabla 2.1: Algoritmos de entrenamiento en Matlab

P se correspondería con los valores que le proporcionaríamos a la red como entrada, y T con los valores de salida para la red neuronal (supuesto que que queremos que la red aprenda el comportamiento de la función seno).

2. Se define el tipo de red neuronal. Obviamente, dado el problema, la red neuronal tendrá una neurona en la capa de entrada (los datos de entrada son valores reales), y una neurona en la capa de salida (la salida de la función es también un valor real), pero será necesario especificar cuántas neuronas queremos incluir en la capa oculta de la red unidireccional. Supongamos que diseñamos una red unidireccional (a la que vamos a llamar MLPnet) con 1 capa oculta de 5 neuronas:

```
1 MLPnet = feedforwardnet(5, 'trainlm');
```

Estamos creando una arquitectura de red neuronal con 5 neuronas en la capa oculta y además estamos indicando que la función de entrenamiento que vamos a utilizar para calcular los pesos y desviaciones de las neuronas en el método de optimización de Levenberg-Marquardt.

Matlab dispone de múltiples algoritmos de entrenamiento que se pueden elegir para entrenar una red unidireccional. En la tabla 2.1 se desglosan estos.

3. A continuación, una vez creada la red neuronal, procedemos al entrenamiento de la misma arquitectura hemos creado bajo el nombre *MLPnet*, con los datos de entrada y salida que tenemos:

```
1 net= train(MLPnet, P, T);
```

Así tenemos una red unidireccional con 5 neuronas en la capa oculta entrenada con un conjunto de datos que hemos extraído de la función seno. Es importante

observar la diferencia entre MLPnet y net. La primera es una arquitectura de red, la segunda es una red ya entrenada con un conjunto de datos. La función train ajusta los pesos y desviaciones de la red con estos datos de entrada hasta que finaliza el algoritmo de aproximación utilizado.

Si queremos observar cómo es la red que hemos creado, lo podemos realizar mediante el comando view.

```
1 view(net);
```

Como observamos en la figura 2.2 tenemos una red unidireccional con una capa oculta con 5 neuronas en esta capa oculta. La capa de entrada tiene una neurona, y la capa de salida tiene una única neurona también.

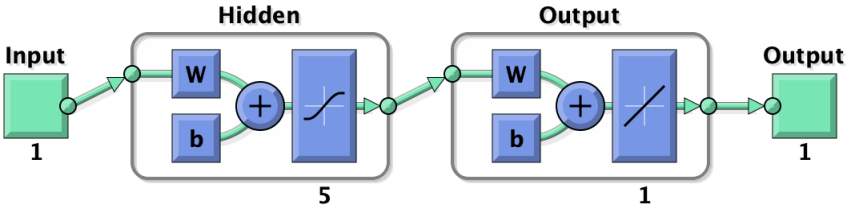

Figura 2.2: Arquitectura de la red unidireccional

4. Por último se comprueba el funcionamiento de la red ya entrenada

```
1 simOut = net(P);
```

Una vez entrenada la red, la salida de esta ante el mismo vector de entradas que hemos utilizado para entrenar dentro de este intervalo se observa en la figura 2.3. Obviamente, como sabemos esto no indica mucho, ya que lo único que podríamos extraer es que la red ha aprendido muy bien los datos que le hemos presentado a la entrada, pero no sabemos si ha aprendido el comportamiento de la función para otros valores distintos a los presentados y utilizados para el entrenamiento.

Si queremos evaluar y cuantificar la diferencia entre la salida de la red que nos está dando para estos datos, y la que nos debería dar (los resultados de evaluar la función seno a estos mismos datos), lo podemos hacer mediante el comando *perfm*:

```
1 perf = perform(net,simOut,T)
```

Como observamos la salida de la red entrenada en este mismo intervalo, y para los mismos datos de entrenamiento, es muy buena como es lógico. Nos podríamos preguntar qué ocurre si elegimos otros datos de entrada/salida a la red diferentes de los usados en entrenamiento. Si escogemos otros valores diferentes en dicho intervalo:

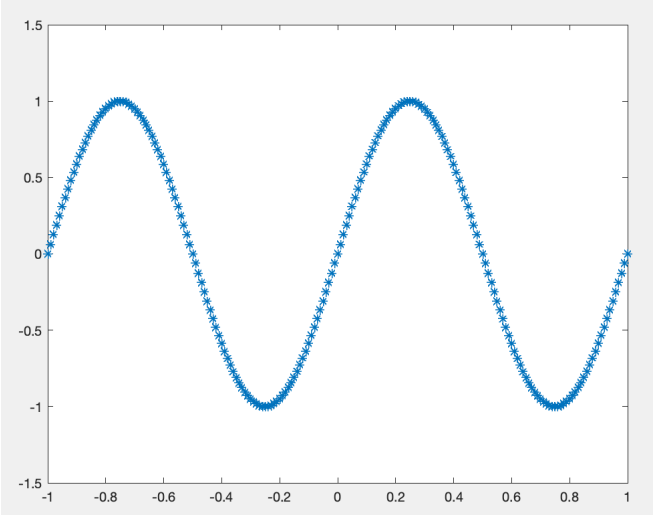

Figura 2.3: Resultados de la aproximación obtenida por la red para los datos de entrenamiento.

```
x=-0.9123:0.01:0.09123;
y=sin(2*pi*x);
simOut1=net(y);
perf1= perform(net,y,simOut1);
```

Como vemos, para estos datos que no han sido utilizados en el entrenamiento, la aproximación de la red también es muy buena. Es preciso observar que estos datos son distintos a los utilizados en el entrenamiento, pero se encuentran dentro del mismo intervalo. Pero qué ocurre si utilizamos datos fuera de dicho intervalo?

```
x2=2:0.01:4;
y2=sin(2*pi*x2);
simOut2=net(y2);
perf2= perform(net,y2,simOut2);
```

El comportamiento deja mucho que desear. Se recomienda dibujar los datos que debería haber dado la red, y los que realmente ha dado al utilizar otro intervalo completamente diferente al que se ha usado para entrenamiento.

Por último, ¿qué pasa si queremos diseñar una arquitectura de red unidireccional con más capas ocultas?. Para incluir nuevas capas ocultas lo podemos hacer al seleccionar la arquitectura de red. Por ejemplo, si queremos crear una red unidireccional con 2 capas ocultas, la primera con 5 neuronas y la segunda capa oculta con 4 neuronas, podemos hacerlo como:

```
red2CapasOcultas = feedforwardnet([5, 4], 'trainlm');
```

Como se puede observar en la figura 2.4, se ha creado dicha estructura de red unidireccional.

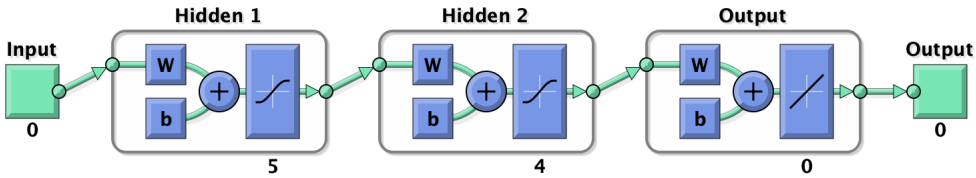

Figura 2.4: Red unidireccional con 2 capas ocultas

2.1.1 Ejercicio

Se propone a continuación realizar un programa en Matlab para aproximar una función más complicada (por ejemplo, que sea continua a trozos) con diferente número de capas ocultas y neuronas en cada una de estas capas. El objetivo consistiría en evaluar y analizar el comportamiento de la red ante una función bastante más complicada como la utilizada previamente.

2.2 Clasificación de Vinos

A continuación realizaremos una clasificación con datos más realistas. Para ello vamos a utilizar datos muy extendidos y conocidos como es las características de un conjunto de vinos. Estos datos se encuentran incorporados en Matlab y podemos hacer uso de los mismos de una forma directa y cómoda.

Consideremos que en una bodega se dispone de un total de 3 vinos diferentes. De cada uno de estos vinos de analizan un total de 13 características diferentes. Se han tomado un total de 178 muestras midiendo cada vez estas 13 características. Así se dispone de:

- 59 medidas del tipo de vino 1 (Clase 1)
- 71 medidas del tipo de vino 2 (Clase 2)
- 48 medidas del tipo de vino 3 (Clase 3)

El objetivo en esta ocasión consiste en diseñar una Red Unidireccional de forma que permita clasificar el tipo de vino a partir del conjunto de 13 características que se dispongan del mismo.

Los pasos a realizar serán los siguientes.

En primer lugar deberíamos leer los datos:

```
1 mData = load("wine.data.txt");
```

Los datos de entrada según los leemos del fichero presentan una configuración de 178 filas y 14 columnas. La primera columna se corresponde con el tipo de vino (Clase 1, 2 o 3) mientras que las restantes 13 columnas se corresponden con las características medidas de cada una de las 178 muestras que se disponen (una muestra para cada fila).

Algunas de las medidas que deberíamos realizar con los datos antes de emplearlos en la red neuronal que diseñemos, son:

- **Aleatorizar** los datos. No es bueno que los datos se encuentren agrupados
- **Normalizar** los datos de entrada/salida
- No utilizar los datos de entrenamiento como datos para comprobar el funcionamiento de la red una vez entrenada.

Por otro lado, para usar los datos, las matrices donde estos se encuentren deben venir expresadas según este formato:

- Entrada (filas: tantas como características, en este caso 13; columnas: tantas como patrones o muestras tengamos, en este caso 178)
- Salida (filas: tantas como neuronas de salida, en este caso 1; columnas: tantas como patrones o muestras de entrada, en este caso 178)

Por lo tanto, lo primero que tendremos que hacer es transponer los datos de entrada para ajustarlos a dicho formato.

```
1 mData=mData';
```

En ***Matlab*** todo este proceso de aleatorizar los datos y dividir estos en diferentes conjuntos para no usar los mismos datos como entrenamiento y validación, se encuentra más encapsulado. De esta manera podemos realizar lo siguiente:

```
1  % se leen los datos del fichero de texto almacenado
2  mData = load("./wine.data.txt");
3
4  % se transpone la matriz dado para adecuarlo al formato de Matlab.
5  % se separan los datos de entrada y salida
6  mData = mData';
7  mOutput = mData(1,:);
8  mInput = mData(2:end, :);
9
10 % se crea la red unidireccional con 5 neuronas en la capa oculta
11 % posteriormente se entrena
12 net = feedforwardnet(5);
13 net = train(net, mInput, mOutput);
14
15 % para la red entrenada, se calcula la salida que ofrece la red ante
```

```
% los datos de entrada
simOut = net(mInput);

% se representan los resultados
plot(mOutput, 'b*');
hold on;
plot(simOut,'ro');
```

Internamente Matlab realiza todos las operaciones que se han ido previamente comentando (normalizar los datos de entrada, aleatorizarlos, dividir los datos en conjuntos (validacion, entrenamiento, test,), ...). Para modificar estos parámetros y ver los valores por defecto, se puede hacer mediante un conjunto de comandos.

Por ejemplo, para ver los parámetros con los que se ha creado la red unidireccional, basta con:

```
net;
```

Nos proporciona todos y cada uno de estos parámetros que se han utilizado. Si queremos visualizar la proporción de datos utilizados en entrenamiento/validacion/test, basta con hacer:

```
net.divideParam
```

pudiendo modificar cualquiera de estos parámetros a voluntad:

```
net.divideParam.trainRatio = 0.6;
net.divideParam.valRatio = 0.25;
net.divideParam.testRatio = 0.1
```

Al realizar el entrenamiento, aparece la ventana representada en la figura 2.5.

Podemos observar al representar el rendimiento del entrenamiento, cómo este se ha producido. Observamos, pulsando el botón Performance el gráfico de la figura 2.6. Podemos observar cómo el entrenamiento ha finalizado en la decima iteración del algoritmo. Obsdervamo cómo hasta esa iteración, el error de validación y test disminuyen, pero a continuación o bien no disminuye o este aumenta. En cambio, como es lógico el error de entrenamiento (obtenido con los datos de entrenamiento), siempre disminuye.

Se observa, finalmente, en la figura 2.7 la clasificación de la red ante los datos de entrada (representado en circulos rojos), y la clasificación real (representada en la figura con '+' en color azul).

2.2.1 Ejercicio

Con objeto de mejorar el comportamiento de esta clasificación se propone modificar la estructura de la red con objeto de intentar mejorar el comportamiento de la red tras su aprendizaje. Algunas cuestiones a remarcar son las siguientes. En la figura se han representado los resultados de la red (una vez entrenada) para los datos que no se han utilizado para el entrenamiento. Cada vez que se entrena la red, los parámetros de la misma son diferentes y en consecuencia los resultados que se obtienen para la clasificación distintos.

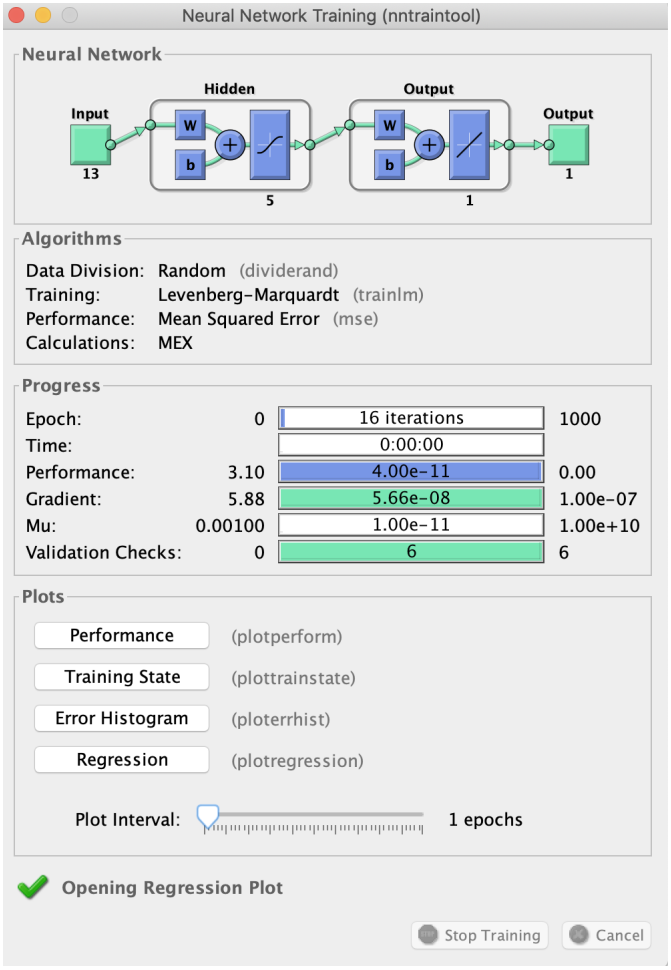

Figura 2.5: Entrenamiento de la red unidireccional con 5 capas ocultas

2.3 Clasificación mediante PYTHON

Vamos a realizar la misma clasificación haciendo uso de las capacidades del lenguaje Python. Para ello utilizaremos Anaconda.

A continuación se muestra el código que utilizaremos.

```
1 import numpy as np
2 from keras.models import Sequential
3 from keras.layers.core import Dense
```

Incorporamos la librería numpy, y de la librería de keras, haremos uso de la red neuronal unidireccional (modelo 'Sequential'). Lo primero que tendremos que hacer es incorporar

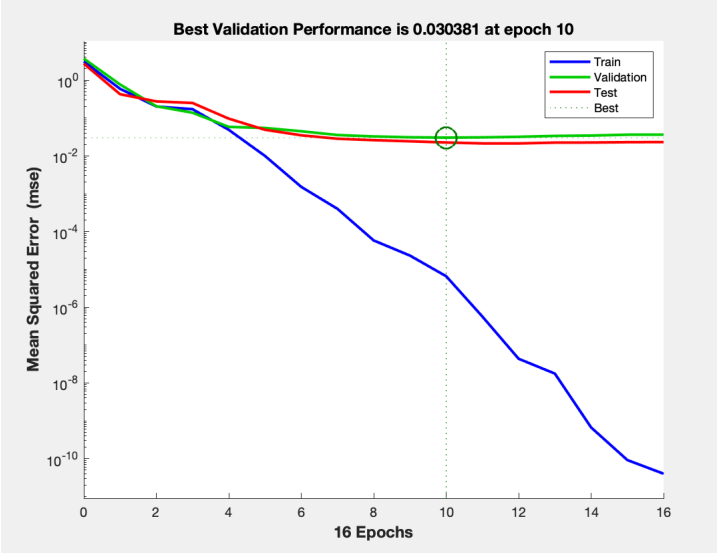

Figura 2.6: Proceso de entrenamiento de la red unidireccional con 5 capas ocultas

los datos. Existen múltiples posibilidades. Como los tenemos en un fichero de texto, los leemos a través del método 'loadtxt' de la librería 'numpy'. Es preciso observar que en nuestro fichero los datos se encuentran separados por una coma, por lo que usamos dicho delimitador:

```
# Se leen los datos del fichero
data=np.loadtxt('./wineData.txt', delimiter=",")
```

Los datos los tendríamos almacenados en una matriz, que hemos denominado *data*. Si observamos se corresponde con una matriz de 178 filas y 14 columnas. La primera columna se corresponde con el tipo de vino, mientras que las 13 columnas restantes se corresponden con características medidas sobre dichos vinos.

Con objeto de que no se presenten sesgos indeseados, en ocasiones conviene aleatorizar los datos. En caso de que veamos oportuno aleatorizar estos datos, lo podemos realizar con el código siguiente. Esto nos permite simplemente aleatorizar las filas de la matriz.

```
# Aleatorizamos las filas de la matriz
dataRandom=data
np.random.shuffle(dataRandom)
```

A continuación podemos extraer los datos de entrada y salida. Los datos de salida se corresponderán con la primera columna, mientras que los datos de entrada se corresponden con los datos almacenados en las últimas 13 columnas.

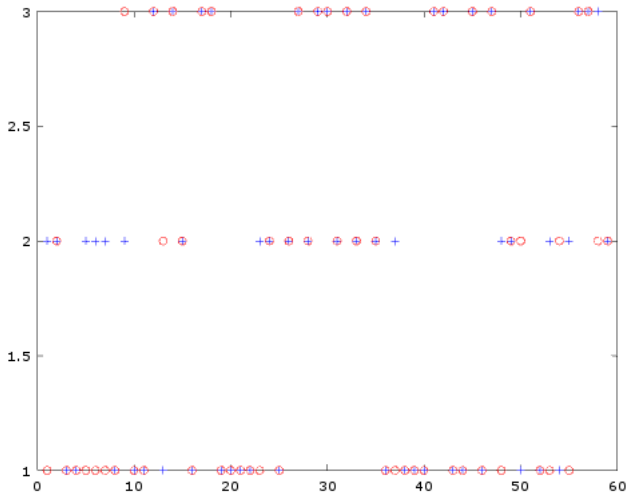

Figura 2.7: Resultados de la red. En circulo rojo se observa el resultado de la red y en cruces azul la clasificación real

```
# Se extraen los datos de entrada/salida
x = dataRandom[:, 1:].astype(float)
y = dataRandom[:, 0].astype(int)
```

Como se observa, los datos de entrada se corresponden con 13 características de tipo *float* para cada muestra. El dato de salida es un número entero que nos indica el tipo de vino (1, 2 o 3). Si miramos los datos de entrada, observamos que hay valores muy diversos en cuanto a magnitud de los mismos. Algunos se corresponden con valores muy pequeños (por ejemplo la característica 8), mientras que otros tienen valores muy grandes (la característica 12). Con objeto de independizar del funcionamiento de la red del valor (mayor o menor) de cada característica, es aconsejable normalizar los datos posibilitando así que todas las características que se utilizan tengan el mismo 'peso'. Esto lo podemos hacer rápidamente haciendo uso de la librería de Python *sklearn*.

```
# Se normalizan los datos de entrada
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x = sc.fit_transform(x)
```

Ahora tenemos los datos de entrada ya normalizados. A continuación observamos la variable de salida. Observamos que tenemos tres clases distintas, y se ha identificado cada clase con un valor numérico. Dado que estamos resolviendo un problema de clasificación (en este caso no binaria), no tiene sentido que una de las clases tenga un valor 3 veces superior a otra. Por este motivo resulta razonable **codificar** los valores

de cada una de las clases, de manera que todas se comporten de igual forma. Por ejemplo a la clase representada por un 1, la podemos identificar con (1 0 0), a la clase representada por un 2, la podemos codificar con (0 1 0), y así sucesivamente. Esto lo podemos hacer en Python mediante el método *OneHotEncoder* que se encuentra dentro de la librería *sklearn*.

```
# Se codifican las clases de salida: 1 (1 0 0), 2 (0 1 0), 3(0 0
    1)
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder()
#La siguiente funcion necesita un array bidimensional como entrada
y_reshaped = np.array(y).reshape(-1,1)
y = ohe.fit_transform(y_reshaped).toarray()
```

Una cuestión a tener en cuenta en el código previo es que la función *fit_ transform* que realiza la codificación, precisa un array bidimensional, y nuestros valores de salida (y) se encuentran en un array unidimensional. Para ello, previamente lo que hacemos es formatear esos valores a un array bidimensional (*reshape*). Como vemos tras ejecutar el código anterior ya tenemos codificados los valores de salida en valores homogéneos, cada uno de los cuales representa un conjunto diferente.

Ya estamos listos para crear la red neuronal y proceder al entrenamiento. Pero antes, en ocasiones es útil dividir este conjunto de datos, si son los únicos disponibles en un conjunto de datos que usaremos para entrenamiento, y otro conjunto de datos que utilizaremos para validación. Esto lo podemos hacer de múltiples formas. Directamente manejando los datos y extrayendo de los mismos una proporción determinada, o bien haciendo uso del método *train_ test_ split* que está incluido en la libreria *sklearn*. Vamos a escoger un 10% de los datos como valores para validación o test.

```
# Dividimos los datos en un conjunto de entrenamiento y otro de
    test
# En este caso dejamos un 10% como test
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size =
    0.1)
```

Ahora tenemos dos conjuntos de valores entrada/salida (x, y respectivamente). Los que usaremos como entrenamiento que hemos denotado como *train*, constituidos por un conjunto de 160 datos y los que usaremos para comprobar este entrenamiento y que reservaremos, constituidos por un conjunto de 18 datos.

A continuación vamos a crear una red neuronal constituida por una red unidireccional (*Sequential* en la librería *Keras*). Crearemos esta red con la capa de entrada (que obviamente tiene que tener 13 neuronas dado que cada dato de entrada tiene 13 características), una capa oculta con 10 neuronas, y una capa de salida (tal y como hemos representado cada conjunto de salida, la capa de salida tiene que tener 3 neuronas). Esto lo podemos realizar con el código siguiente:

```
# Se crea la red neuronal con una capa oculta de 10 neuronas
# capa de entrada de 13 neuronas, y de salida de 3 neuronas
# una vez que se ha modificado la salida
model = Sequential()
model.add(Dense(10, input_dim=13, activation='relu'))
model.add(Dense(3, activation='softmax'))

model.summary()
```

Si observamos al generar la red neuronal, además del número de neuronas en cada capa, hemos indicado el tipo de función de activación para las neuronas dentro de cada una de estas capas. Para la capa oculta hemos elegido la función de activación tiepo *relu*, mientras que para las neuronas de la capa de salida hemos optado por la función *softmax*. Otra cuestión a tener en cuenta es el tipo de capa que vamos incorporando a la red que estamos construyendo. Son capas de tipo *Dense*, en el que cada una de las neuronas se encuentra conectada a todas las neuronas de la capa precedente (que es el típico funcionamiento de la red unidireccional).

A continuación se establecen algunos parámetros en la utilización de la red neuronal. El principal de todos se corresponde con el método de entrenamiento. En el caso que se propone se ha elegido *adam*. Seleccionados estos parámetros se procede a realizar un entrenamiento de la red con un total de 100 iteraciones usando únicamente los datos de entrenamiento que previamente habíamos reservado

```
# Ajustes del modelo para entrenar
model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])
history=model.fit(x_train, y_train, epochs=100)
```

El código previo procede a realizar un entrenamiento de la red construida con los datos de entrenamiento. Finalizado este, tenemos una red entrenada con los datos que se le han proporcionado. Pudiera ser útil dibujar los resultados en cuanto a la precisión con estos datos de entrenamiento para poder constatar cómo ha evolucionado este entrenamiento.

```
# Se dibujan los resultados
import matplotlib.pyplot as plt
plt.plot(history.history['accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train'], loc='upper left')
plt.show()
```

Con los datos que nos hemos reservado previamente (test), podemos constatar el comportamiento de la red. Es preciso recordar que estos datos no han sido mostrados a la red en ningún momento del entrenamiento. Para ello basta con emplear el método *predict* y pasarle como entrada los valores de test. No obstante dado que tenemos codificados los conjuntos deberemos posteriormente a convertir los valores de salida que nos ofrece la red en función de las etiquetas de los conjuntos disponibles.

```
# Se evalua el modelo con los datos de test
y_pred=model.predict(x_test)

# Convertimos las predicciones a las etiquetas
pred = list()
for i in range(len(y_pred)):
    pred.append(np.argmax(y_pred[i]))

# Convertimos los valores deseados y_test
test = list()
for i in range(len(y_test)):
    test.append(np.argmax(y_test[i]))

from sklearn.metrics import accuracy_score
a = accuracy_score(pred,test)
print('Accuracy is:', a*100)
```

Otra alternativa de entrenamiento es utilizar el método de validación cruzada. Para ello podemos emplear el siguiente código. De esta manera podemos observar el sobreentronamiento que se puede producir sobre los datos.

```
# Otra alternativa es entrenar mediante el metodo de validacion
    cruzada
history2 = model.fit(x_train, y_train,validation_data = (x_test,
    y_test), epochs=100, batch_size=64)

# Se dibujan los resultados
import matplotlib.pyplot as plt
plt.plot(history2.history['accuracy'])
plt.plot(history2.history['val_accuracy'])
plt.title('Model accuracy Validacion Cruzada')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

plt.plot(history2.history['loss'])
plt.plot(history2.history['val_loss'])
plt.title('Model loss')
```

```
17 plt.ylabel('Loss')
18 plt.xlabel('Epoch')
19 plt.legend(['Train', 'Test'], loc='upper left')
20 plt.show()
```

2.4 Clasificar patrones

Se propone en este último ejercicio la realización de una red neuronal para clasificar patrones. Para ello se deberá elegir un conjunto de datos de entrada a clasificar y diseñar y entrenar una red neuronal que posibilite la consecución de esta clasificación.

En la siguiente dirección se disponen de diferentes datos a elegir:

https://archive.ics.uci.edu/ml/datasets.html