Práctica de redes neuronales con Keras

Sistemas Inteligentes

Grado en Ingeniería de la salud

Universidad de Sevilla

Keras es una biblioteca de Python que proporciona una interfaz amigable, modular y extensible para experimentar con redes neuronales. Actualmente Keras se desarrolla como parte de TensorFlow, la plataforma para aprendizaje automático de Google. Esto significa que para poder usar Keras se debe instalar el paquete `tensorflow` de Python. En esta práctica también se hará uso de los paquetes `numpy`, `pandas` y `sklearn`, que se deberán tener instalados.

In [1]:
```
# pip install tensorflow
```

Trabajar con redes neuronales implica el manejo de números pseudoaleatorios. Para que este documento sea reproducible es necesario, por tanto, establecer una semilla inicial para el generador de esos números.

In [2]:
```
from tensorflow import random as tensorflow_random

tensorflow_random.set_seed(394867)
```

```
2022-12-12 09:59:44.360138: I tensorflow/core/platform/cpu_feature_guard.cc:193] T
his TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDN
N) to use the following CPU instructions in performance-critical operations:  AVX2
FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compil
er flags.
2022-12-12 09:59:44.511980: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerro
r: libcudart.so.11.0: cannot open shared object file: No such file or directory
2022-12-12 09:59:44.512001: I tensorflow/compiler/xla/stream_executor/cuda/cudart_
stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your ma
chine.
2022-12-12 09:59:45.323035: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libnvinfer.so.7'; dlerror:
libnvinfer.so.7: cannot open shared object file: No such file or directory
2022-12-12 09:59:45.323146: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libnvinfer_plugin.so.7'; dl
error: libnvinfer_plugin.so.7: cannot open shared object file: No such file or dir
ectory
2022-12-12 09:59:45.323159: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:3
8] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you would like to use
Nvidia GPU with TensorRT, please make sure the missing libraries mentioned above a
re installed properly.
```

Importamos en primer lugar los paquetes que nos permitirán preprocesar los datos y dividirlos en subconjuntos de entrenamiento y prueba. También establecemos una semilla

inicial para que estas operaciones sean reproducibles y fijamos el tamaño máximo que debe tener un array para que se muestre completo.

In [3]:
```
import numpy
import pandas
from sklearn import model_selection

numpy.random.seed(43958734)
numpy.set_printoptions(threshold=10)
```

Finalmente importamos el paquete `keras`.

In [4]:
```
from tensorflow import keras
```

Redes neuronales para tareas de regresión

El conjunto de datos Gas Turbine CO and NOx Emission del repositorio UCI contiene las siguientes medidas, agregadas por hora y tomadas desde el 1 de enero de 2011 hasta el 31 de diciembre de 2015, de una turbina de gas localizada en la región noroccidental de Turquía:

- AT: temperatura ambiental, en grados centígrados.
- AP: presión ambiental, en milibares.
- AH: humedad ambiental, en porcentaje.
- AFDP: diferencia de presión en el filtro de aire, en milibares.
- GTEP: presión de escape de la turbina de gas, en milibares.
- TIT: temperatura de entrada de la turbina, en grados centígrados.
- TAT: temperatura de salida de la turbina, en grados centígrados.
- CDP: presión de descarga del compresor, en milibares.
- TEY: rendimiento energético de la turbina, en megavatios por hora.
- CO: emisiones de monóxido de carbono, en miligramos por metro cúbico.
- NOX: emisiones de óxidos de nitrógeno, en miligramos por metro cúbico.

Los datos se encuentran en el fichero `gas_turbine.csv`.

In [5]:
```
gas_turbine = pandas.read_csv("gas_turbine.csv")
gas_turbine.head()
```

Out[5]:

	AT	AP	AH	AFDP	GTEP	TIT	TAT	CDP	TEY	CO	NOX
0	4.5878	1018.7	83.675	3.5758	23.979	1086.2	549.83	11.898	134.67	0.32663	81.952
1	4.2932	1018.3	84.235	3.5709	23.951	1086.1	550.05	11.892	134.67	0.44784	82.377
2	3.9045	1018.4	84.858	3.5828	23.990	1086.5	550.19	12.042	135.10	0.45144	83.776
3	3.7436	1018.3	85.434	3.5808	23.911	1086.5	550.17	11.990	135.03	0.23107	82.505
4	3.7516	1017.8	85.182	3.5781	23.917	1085.9	550.00	11.910	134.67	0.26747	82.028

Se plantea el problema de predecir el rendimiento energético de la turbina a partir de las medidas ambientales (AT, AP y AH) y de las medidas de los sensores de la turbina (AFDP,

GTEP, TIT, TAT, CDP). Obsérvese que se trata de un problema de regresión, puesto que la variable objetivo es una variable continua.

En primer lugar, seleccionamos por un lado las variables predictoras y por otro lado la variable respuesta. En ambos casos transformamos los resultados a un array de NumPy, ya que ese tipo de dato es compatible con Keras mientras que los marcos de datos de Pandas no lo son.

In [6]:
```
atributos = gas_turbine.loc[:, 'AT':'CDP']
atributos = atributos.to_numpy()
print(atributos)
```

```
[[   4.5878 1018.7      83.675  ... 1086.2     549.83     11.898 ]
 [   4.2932 1018.3      84.235  ... 1086.1     550.05     11.892 ]
 [   3.9045 1018.4      84.858  ... 1086.5     550.19     12.042 ]
 ...
 [   5.482  1028.5      95.219  ... 1038.      543.48     10.462 ]
 [   5.8837 1028.7      94.2    ... 1076.9     550.11     11.771 ]
 [   6.0392 1028.8      94.547  ... 1067.9     548.23     11.462 ]]
```

In [7]:
```
objetivo = gas_turbine['TEY']
objetivo = objetivo.to_numpy()
print(objetivo)
```

```
[134.67 134.67 135.1  ... 107.81 131.41 125.41]
```

A continuación construimos los subconjuntos de entrenamiento y prueba para la construcción y evaluación de redes neuronales mediante aprendizaje supervisado.

In [8]:
```
(atributos_entrenamiento, atributos_prueba,
 objetivo_entrenamiento, objetivo_prueba) = model_selection.train_test_split(
    atributos, objetivo, test_size=.33)
```

Estamos ya en condiciones de construir una red neuronal que nos permita abordar el problema.

Keras admite dos aproximaciones a la hora de construir una red neuronal: el modelo funcional, más flexible, y el modelo secuencial, más simple y el cual será el que usaremos en esta práctica.

En el modelo secuencial se construye una red neuronal capa a capa, comenzando por la capa de entrada y continuando con el resto de capas hasta la última, que será la capa de salida.

La capa de entrada se construye como instancia de la clase `Input`, indicando la forma de la entrada, que puede ser un array con cualquier número de dimensiones. En nuestro caso la entrada será un array unidimensional con los valores de los atributos, pero podría ser por ejemplo una imagen bidimensional, etcétera.

Para construir una red neuronal con alimentación hacia adelante, en el que las capas están totalmente conectadas, hay que añadir instancias de la clase `layers.Dense`, indicando la cantidad de neuronas y la función de activación (por defecto, la identidad).

Por ejemplo, podemos construir una red neuronal con ocho neuronas de entrada, una por cada atributo, y una neurona de salida, que proporcione el rendimiento energético

predicho, de la siguiente manera:

In [9]:
```
red_turbina = keras.Sequential()
red_turbina.add(keras.Input(shape=(8,)))
red_turbina.add(keras.layers.Dense(1))
```

```
2022-12-12 09:59:47.184537: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerro
r: libcudart.so.11.0: cannot open shared object file: No such file or directory
2022-12-12 09:59:47.184646: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcublas.so.11'; dlerror:
libcublas.so.11: cannot open shared object file: No such file or directory
2022-12-12 09:59:47.184724: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcublasLt.so.11'; dlerro
r: libcublasLt.so.11: cannot open shared object file: No such file or directory
2022-12-12 09:59:47.184798: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcufft.so.10'; dlerror: l
ibcufft.so.10: cannot open shared object file: No such file or directory
2022-12-12 09:59:47.184870: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcurand.so.10'; dlerror:
libcurand.so.10: cannot open shared object file: No such file or directory
2022-12-12 09:59:47.184939: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcusolver.so.11'; dlerro
r: libcusolver.so.11: cannot open shared object file: No such file or directory
2022-12-12 09:59:47.185011: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcusparse.so.11'; dlerro
r: libcusparse.so.11: cannot open shared object file: No such file or directory
2022-12-12 09:59:47.185082: W tensorflow/compiler/xla/stream_executor/platform/def
ault/dso_loader.cc:64] Could not load dynamic library 'libcudnn.so.8'; dlerror: li
bcudnn.so.8: cannot open shared object file: No such file or directory
2022-12-12 09:59:47.185098: W tensorflow/core/common_runtime/gpu/gpu_device.cc:193
4] Cannot dlopen some GPU libraries. Please make sure the missing libraries mentio
ned above are installed properly if you would like to use GPU. Follow the guide at
https://www.tensorflow.org/install/gpu for how to download and setup the required
libraries for your platform.
Skipping registering GPU devices...
2022-12-12 09:59:47.185518: I tensorflow/core/platform/cpu_feature_guard.cc:193] T
his TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDN
N) to use the following CPU instructions in performance-critical operations:  AVX2
FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compil
er flags.
```

El método `summary` nos muestra la estructura de la red, indicando para cada capa la forma de su salida y cuántos parámetros posee. La capa de entrada no se muestra, ya que no posee parámetros. La primera dimensión en la forma de la salida de cada capa indica el tamaño de los lotes (de los que hablaremos posteriormente) y si su valor es `None` quiere decir que se determinará posteriormente.

In [10]:
```
red_turbina.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 1)                 9

=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

Se observa cómo la red neuronal posee una única capa (aparte de la de entrada) que proporciona como salida un array bidimensional None x 1 (es decir, para cada ejemplo proporciona un valor) y 9 parámetros (el peso de la conexión de cada entrada con la neurona de la capa más el sesgo de la neurona) entrenables (la red los aprenderá mediante el algoritmo de entrenamiento).

Los pesos y sesgos de la red se guardan en el atributo `weights` (son los arrays asociados al argumento numpy en la estructura de datos guardada en ese atributo).

In [11]:
```
red_turbina.weights
```

Out[11]:
```
[<tf.Variable 'dense/kernel:0' shape=(8, 1) dtype=float32, numpy=
 array([[ 0.48478258],
        [ 0.23702323],
        [ 0.34562147],
        [-0.7070208 ],
        [ 0.23453188],
        [-0.65631664],
        [-0.5645554 ],
        [ 0.6873833 ]], dtype=float32)>,
 <tf.Variable 'dense/bias:0' shape=(1,) dtype=float32, numpy=array([0.], dtype=flo
at32)>]
```

Si le pedimos a la red que prediga el rendimiento energético de los cinco primeros ejemplos del conjunto de datos, se obtendrán valores muy distintos de los correctos mostrados anteriormente, ya que todavía no se ha entrenado la red.

In [12]:
```
red_turbina.predict(gas_turbine.iloc[0:5, 0:8].to_numpy())
```

Out[12]:
```
1/1 [==============================] - 0s 64ms/step
array([[-739.42694],
       [-739.5368 ],
       [-739.7239 ],
       [-739.66815],
       [-739.4318 ]], dtype=float32)
```

Para entrenar una red neuronal hay que compilarla primero, estableciendo el algoritmo de aprendizaje (*optimizer*) y la función de pérdida (*loss*) a minimizar. Lo más cercano a lo explicado en clase es usar el error cuadrático medio (Keras no tiene implementada la suma de los errores cuadráticos, pero la minimización de ambas funciones es equivalente) y el algoritmo del descenso estocástico por el gradiente (*stochastic gradient descent*, SGD).

El descenso estocástico por el gradiente es el algoritmo de retropropagación, pero en lugar de actualizar los pesos tras procesar todos los ejemplos de entrenamiento lo hace tras procesar un subconjunto de ellos cada vez. Cada uno de estos subconjuntos se llama un lote (*batch*). Los lotes se construyen repartiendo aleatoriamente los ejemplos de

entrenamiento y cuando se han considerado todos los lotes (y, en consecuencia, todos los ejemplos) se dice que ha transcurrido una época (*epoch*).

In [13]:
```
red_turbina.compile(optimizer='SGD', loss='mean_squared_error')
```

Por defecto se usa un factor de aprendizaje igual a 0.01. Solo falta, entonces, proporcionar los ejemplos de entrenamiento junto con la salida esperada para cada uno de ellos, el tamaño de los lotes y el número de épocas a entrenar.

In [14]:
```
red_turbina.fit(atributos_entrenamiento, objetivo_entrenamiento,
                batch_size=256, epochs=10)
```

```
Epoch 1/10
97/97 [==============================] - 0s 934us/step - loss: nan
Epoch 2/10
97/97 [==============================] - 0s 847us/step - loss: nan
Epoch 3/10
97/97 [==============================] - 0s 848us/step - loss: nan
Epoch 4/10
97/97 [==============================] - 0s 886us/step - loss: nan
Epoch 5/10
97/97 [==============================] - 0s 864us/step - loss: nan
Epoch 6/10
97/97 [==============================] - 0s 852us/step - loss: nan
Epoch 7/10
97/97 [==============================] - 0s 886us/step - loss: nan
Epoch 8/10
97/97 [==============================] - 0s 832us/step - loss: nan
Epoch 9/10
97/97 [==============================] - 0s 822us/step - loss: nan
Epoch 10/10
97/97 [==============================] - 0s 830us/step - loss: nan
```

Out[14]:
```
<keras.callbacks.History at 0x7fdf803ba7c0>
```

El entrenamiento ha fallado, ya que se han producido desbordamientos numéricos al calcular la función de pérdida (nan quiere decir *not a number*). Los motivos de que se produzca esta circunstancia pueden ser múltiples y, a veces, difíciles de determinar. En este caso el problema se encuentra en que proporcionamos los valores en bruto de los atributos, cuando lo recomendable es que los valores que reciban como entrada las neuronas sean cercanos a cero.

Es necesario, pues, introducir tras la capa de entrada una capa de normalización como por ejemplo la disponible en Keras para tipificar las variables (a cada variable se le resta su media y se divide por su desviación típica). Los parámetros de esta capa de normalización deben ajustarse a partir únicamente de los datos de entrenamiento, lo que se hace mediante el método `adapt`.

In [15]:
```
normalizador = keras.layers.experimental.preprocessing.Normalization()
normalizador.adapt(atributos_entrenamiento)
```

Podemos comprobar como todos los atributos de entrenamiento tienen media y varianza aproximadamente 0 y 1, respectivamente, al ser normalizados.

In [16]:
```
print(numpy.mean(normalizador(atributos_entrenamiento), axis=0))
print(numpy.var(normalizador(atributos_entrenamiento), axis=0))
```

```
[ 1.6957632e-06 -2.6569342e-05 -1.0380697e-05  5.7639295e-07
  2.5071442e-06  7.2057173e-06  8.9993846e-05 -3.1143845e-06]
[1.0000004  1.0000004  1.0000049  1.000001   0.9999952  0.99995697
 0.99999523 1.0000012 ]
```

Al introducir en la red esa capa de normalización justo tras la capa de entrada, todos los datos que se proporcionen como entrada a la red se tipificarán usando las medias y desviaciones típicas calculadas.

In [17]:
```
red_turbina = keras.Sequential()
red_turbina.add(keras.Input(shape=(8,)))
red_turbina.add(normalizador)
red_turbina.add(keras.layers.Dense(1))
```

Se puede comprobar que los parámetros de esa capa de normalización no son entrenables, es decir, no se modificarán al aplicar a la red el algoritmo de descenso estocástico por el gradiente (la capa de normalización tiene 17 parámetros en total porque guarda la media y la desviación típica de cada uno de los ocho atributos y, además, la cantidad total de ejemplos a partir de los cuales se han calculado).

In [18]:
```
red_turbina.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization (Normalizatio  (None, 8)                17
 n)

 dense_1 (Dense)             (None, 1)                 9

=================================================================
Total params: 26
Trainable params: 9
Non-trainable params: 17
_________________________________________________________________
```

Como se ha vuelto a construir la red desde cero, también hay que volverla a compilar.

In [19]:
```
red_turbina.compile(optimizer='SGD', loss='mean_squared_error')
```

Ahora la red ya entrena sin problema.

In [20]:
```
red_turbina.fit(atributos_entrenamiento, objetivo_entrenamiento,
                batch_size=256, epochs=10)
```

```
Epoch 1/10
97/97 [==============================] - 0s 974us/step - loss: 4611.4688
Epoch 2/10
97/97 [==============================] - 0s 968us/step - loss: 93.6941
Epoch 3/10
97/97 [==============================] - 0s 873us/step - loss: 3.4680
Epoch 4/10
97/97 [==============================] - 0s 890us/step - loss: 1.5351
Epoch 5/10
97/97 [==============================] - 0s 924us/step - loss: 1.4602
Epoch 6/10
97/97 [==============================] - 0s 894us/step - loss: 1.4407
Epoch 7/10
97/97 [==============================] - 0s 905us/step - loss: 1.4256
Epoch 8/10
97/97 [==============================] - 0s 864us/step - loss: 1.4121
Epoch 9/10
97/97 [==============================] - 0s 876us/step - loss: 1.3991
Epoch 10/10
97/97 [==============================] - 0s 878us/step - loss: 1.3863
```

Out[20]:
```
<keras.callbacks.History at 0x7fdf8018c880>
```

Para comprobar el comportamiento de la red sobre los datos de prueba basta usar el método `evaluate`.

In [21]:
```
red_turbina.evaluate(atributos_prueba, objetivo_prueba)
```

```
379/379 [==============================] - 0s 815us/step - loss: 1.3453
```

Out[21]:
```
1.3452847003936768
```

Una forma de tratar de mejorar el rendimiento de la red neuronal es aumentar el número de épocas de entrenamiento. Esto, aparte del obvio coste en tiempo que supone, no garantiza que el algoritmo de entrenamiento no se quede atascado en un mínimo local de la función de pérdida, sin llegar a aproximarse nunca al mínimo global.

Otra vía para conseguir una red neuronal con mejor rendimiento es modificar la estructura de la misma. Por ejemplo, en el caso que nos ocupa, podemos probar a incluir una capa oculta de la que se ha elegido, de forma arbitraria, que tenga 10 neuronas.

In [22]:
```
red_turbina = keras.Sequential()
red_turbina.add(keras.Input(shape=(8,)))
red_turbina.add(normalizador)
red_turbina.add(keras.layers.Dense(10))
red_turbina.add(keras.layers.Dense(1))
```

Esta red tiene 101 parámetros entrenables, que provienen de los 8x10 pesos más 10 sesgos de las neuronas de la capa oculta más los 10 pesos y el sesgo de la neurona de la capa de salida. También tiene 17 parámetros no entrenables, que provienen de la capa de normalización.

In [23]:
```
red_turbina.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization (Normalizatio  (None, 8)                17
 n)

 dense_2 (Dense)             (None, 10)                90

 dense_3 (Dense)             (None, 1)                 11

=================================================================
Total params: 118
Trainable params: 101
Non-trainable params: 17
_________________________________________________________________
```

Al compilar la red se puede indicar también que, aparte de la función de pérdida, se calculan otras métricas adecuadas. Por ejemplo, el error absoluto medio (es decir, la media de las diferencias en valor absoluto entre los valores predichos y los valores correctos) es una métrica adecuada para una tarea de regresión.

In [24]:
```
red_turbina.compile(optimizer='SGD', loss='mean_squared_error',
                    metrics=['mean_absolute_error'])
```

In [25]:
```
red_turbina.fit(atributos_entrenamiento, objetivo_entrenamiento,
                batch_size=256, epochs=10)
```

```
Epoch 1/10
97/97 [==============================] - 0s 959us/step - loss: nan - mean_absolute
_error: nan
Epoch 2/10
97/97 [==============================] - 0s 917us/step - loss: nan - mean_absolute
_error: nan
Epoch 3/10
97/97 [==============================] - 0s 1ms/step - loss: nan - mean_absolute_e
rror: nan
Epoch 4/10
97/97 [==============================] - 0s 1ms/step - loss: nan - mean_absolute_e
rror: nan
Epoch 5/10
97/97 [==============================] - 0s 1ms/step - loss: nan - mean_absolute_e
rror: nan
Epoch 6/10
97/97 [==============================] - 0s 1ms/step - loss: nan - mean_absolute_e
rror: nan
Epoch 7/10
97/97 [==============================] - 0s 1ms/step - loss: nan - mean_absolute_e
rror: nan
Epoch 8/10
97/97 [==============================] - 0s 1ms/step - loss: nan - mean_absolute_e
rror: nan
Epoch 9/10
97/97 [==============================] - 0s 1ms/step - loss: nan - mean_absolute_e
rror: nan
Epoch 10/10
97/97 [==============================] - 0s 1ms/step - loss: nan - mean_absolute_e
rror: nan
```

Out[25]:
```
<keras.callbacks.History at 0x7fdf800639d0>
```

De nuevo se han producido desbordamientos en el cálculo de la función de pérdida. En este caso es debido a que las neuronas de la capa oculta tienen como función de activación a la función identidad (la función de activación por defecto). Por lo tanto, aunque ellas sí reciben las entradas normalizadas, los valores que devuelven, y que son las que recibe la neurona de la capa de salida, pueden alejarse bastante del valor cero. Si establecemos como función de activación de las neuronas de la capa oculta a la función sigmoide, esta normalizará al intervalo (0, 1) los valores que devuelven esas neuronas y la red se podrá entrenar sin problema.

In [26]:
```
red_turbina = keras.Sequential()
red_turbina.add(keras.Input(shape=(8,)))
red_turbina.add(normalizador)
red_turbina.add(keras.layers.Dense(10, activation='sigmoid'))
red_turbina.add(keras.layers.Dense(1))
```

Al compilar la red se pueden cambiar los parámetros de los distintos argumentos proporcionando, en lugar del nombre, una instancia de la clase que lo implementa. Por ejemplo, para establecer el factor de aprendizaje, en lugar de proporcionar el nombre `'SGD'` para indicar descenso estocástico por el gradiente como optimizador hay que proporcionar una instancia de la clase `SGD` que lo implementa.

In [27]:
```
red_turbina.compile(optimizer=keras.optimizers.SGD(learning_rate=0.02), loss='mean_
                    metrics=['mean_absolute_error'])
```

In [28]:
```
red_turbina.fit(atributos_entrenamiento, objetivo_entrenamiento,
                batch_size=256, epochs=10)
```

```
Epoch 1/10
97/97 [==============================] - 0s 1ms/step - loss: 473.1885 - mean_absol
ute_error: 7.7159
Epoch 2/10
97/97 [==============================] - 0s 1ms/step - loss: 3.7997 - mean_absolut
e_error: 1.2718
Epoch 3/10
97/97 [==============================] - 0s 1ms/step - loss: 1.9800 - mean_absolut
e_error: 0.9848
Epoch 4/10
97/97 [==============================] - 0s 1ms/step - loss: 1.4902 - mean_absolut
e_error: 0.8836
Epoch 5/10
97/97 [==============================] - 0s 1ms/step - loss: 1.2683 - mean_absolut
e_error: 0.8317
Epoch 6/10
97/97 [==============================] - 0s 1ms/step - loss: 1.1194 - mean_absolut
e_error: 0.7918
Epoch 7/10
97/97 [==============================] - 0s 1ms/step - loss: 1.0243 - mean_absolut
e_error: 0.7644
Epoch 8/10
97/97 [==============================] - 0s 1ms/step - loss: 1.0262 - mean_absolut
e_error: 0.7690
Epoch 9/10
97/97 [==============================] - 0s 1ms/step - loss: 0.9161 - mean_absolut
e_error: 0.7286
Epoch 10/10
97/97 [==============================] - 0s 1ms/step - loss: 0.9327 - mean_absolut
e_error: 0.7360
```

Out[28]:
```
<keras.callbacks.History at 0x7fdf800b88b0>
```

In [29]:
```
red_turbina.evaluate(atributos_prueba, objetivo_prueba)
```

```
379/379 [==============================] - 0s 861us/step - loss: 0.7964 - mean_abs
olute_error: 0.6850
```

Out[29]:
```
[0.7964341640472412, 0.6850011348724365]
```

Se obtiene un error sobre el conjunto de prueba menor que el obtenido con la red sin capa oculta.

Ejercicio 1

Se plantea el problema de predecir **a la vez** el rendimiento energético de la turbina (TEY), las emisiones de monóxido de carbono (CO) y las emisiones de óxidos de nitrógeno (NOX) a partir de las medidas ambientales (AT, AP y AH) y de las medidas de los sensores de la turbina (AFDP, GTEP, TIT, TAT, CDP).

Experimentar con diferentes arquitecturas de redes neuronales, que necesariamente deberán tener 3 neuronas en la capa de salida, y con diferentes configuraciones de entrenamiento para abordar el problema.

El objetivo último es construir una red neuronal con un error absoluto medio sobre el conjunto de prueba de alrededor de 1.5.

In [ ]:
```
```

Redes neuronales para tareas de clasificación

Iris es un conjunto de datos multivariante que se ha estudiado exhaustivamente y se ha convertido en un conjunto de datos de referencia a la hora de analizar el comportamiento de los distintos algoritmos de aprendizaje automático.

Iris recopila cuatro medidas (longitud y anchura de sépalo y pétalo) de 50 flores de cada una de las siguientes tres especies de lirios: *Iris setosa*, *Iris virginica* e *Iris versicolor*.

Los datos se encuentran en el fichero `iris.csv`.

In [30]:
```
iris = pandas.read_csv('iris.csv', header=None,
                       names=['Longitud sépalo', 'Anchura sépalo',
                              'Longitud pétalo', 'Anchura pétalo',
                              'Especie'])
iris.head()
```

Out[30]:

	Longitud sépalo	Anchura sépalo	Longitud pétalo	Anchura pétalo	Especie
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Se plantea el problema de predecir si una flor es o no de la especie *Iris setosa* a partir de las medidas de sus sépalos y pétalos. Obsérvese que se trata de un problema de clasificación, puesto que la variable objetivo es una variable categórica binaria.

En primer lugar, seleccionamos por un lado las variables predictoras y por otro lado la variable respuesta. En ambos casos transformamos los resultados a un array de NumPy. La variable respuesta la transformamos además a una variable numérica.

In [31]:
```
atributos = iris.loc[:, 'Longitud sépalo':'Anchura pétalo']
atributos = atributos.to_numpy()
print(atributos)
```

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 ...
 [6.5 3.  5.2 2. ]
 [6.2 3.4 5.4 2.3]
 [5.9 3.  5.1 1.8]]
```

In [32]:
```
objetivo = iris['Especie'] == 'Iris-setosa'
objetivo = objetivo.to_numpy().astype(float)
print(objetivo)
```

```
[1. 1. 1. ... 0. 0. 0.]
```

A continuación construimos los subconjuntos de entrenamiento y prueba.

In [33]:
```
(atributos_entrenamiento, atributos_prueba,
 objetivo_entrenamiento, objetivo_prueba) = model_selection.train_test_split(
    atributos, objetivo, test_size=.33)
```

Estamos ya en condiciones de construir una red neuronal que nos permita abordar el problema. Al tratarse este de un problema de clasificación binaria es habitual considerar una única neurona de salida que devuelva un valor entre 0 y 1 representando la probabilidad de que el ejemplo de entrada pertenezca a la clase positiva.

In [34]:
```
normalizador = keras.layers.experimental.preprocessing.Normalization()
normalizador.adapt(atributos_entrenamiento)
```

In [35]:
```
red_lirios = keras.Sequential()
red_lirios.add(keras.Input(shape=(4,)))
red_lirios.add(normalizador)
red_lirios.add(keras.layers.Dense(1, activation='sigmoid'))
```

In [36]:
```
red_lirios.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_1 (Normalizat  (None, 4)                9
 ion)

 dense_6 (Dense)             (None, 1)                 5

=================================================================
Total params: 14
Trainable params: 5
Non-trainable params: 9
_________________________________________________________________
```

La función de pérdida a minimizar adecuada en este caso es la entropía cruzada binaria:

$$C(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

donde y es la respuesta correcta e $\hat{y}$ es la respuesta proporcionada por el modelo.

También le vamos a pedir a la red que calcule su exactitud, es decir, la fracción de ejemplos clasificados correctamente. Una flor se clasificará como de la especie *Iris setosa* si la probabilidad de que sea así devuelta por la red es mayor que 0.5.

In [37]:
```
red_lirios.compile(optimizer='SGD', loss='binary_crossentropy',
                   metrics=['accuracy'])
```

In [38]:
```
red_lirios.fit(atributos_entrenamiento, objetivo_entrenamiento,
               batch_size=10, epochs=10)
```

```
Epoch 1/10
10/10 [==============================] - 0s 1ms/step - loss: 1.3124 - accuracy: 0.
1800
Epoch 2/10
10/10 [==============================] - 0s 1ms/step - loss: 1.1888 - accuracy: 0.
2200
Epoch 3/10
10/10 [==============================] - 0s 1ms/step - loss: 1.0773 - accuracy: 0.
2500
Epoch 4/10
10/10 [==============================] - 0s 1ms/step - loss: 0.9770 - accuracy: 0.
3100
Epoch 5/10
10/10 [==============================] - 0s 1ms/step - loss: 0.8871 - accuracy: 0.
3900
Epoch 6/10
10/10 [==============================] - 0s 1ms/step - loss: 0.8077 - accuracy: 0.
4400
Epoch 7/10
10/10 [==============================] - 0s 1ms/step - loss: 0.7378 - accuracy: 0.
4900
Epoch 8/10
10/10 [==============================] - 0s 1ms/step - loss: 0.6763 - accuracy: 0.
5900
Epoch 9/10
10/10 [==============================] - 0s 1ms/step - loss: 0.6223 - accuracy: 0.
6900
Epoch 10/10
10/10 [==============================] - 0s 1ms/step - loss: 0.5747 - accuracy: 0.
7600
```

Out[38]:
```
<keras.callbacks.History at 0x7fdf80174160>
```

In [39]:
```
red_lirios.evaluate(atributos_prueba, objetivo_prueba)
```

```
2/2 [==============================] - 0s 3ms/step - loss: 0.5258 - accuracy: 0.8000
[0.5258042216300964, 0.800000011920929]
```

Out[39]:

En este caso la red construida es capaz de identificar correctamente todas las flores del conjunto de prueba que son de la especie *Iris setosa*. Esto no es sorprendente, ya que es perfectamente conocido que las medidas de los sépalos y pétalos permiten separar esta especie de lirio de las otras dos especies. Separar estas dos últimas entre sí ya entraña un poco de mayor dificultad.

Ejercicio 2

Se plantea el problema de determinar la especie de lirio a la que pertenece una flor a partir de las medidas de sus sépalos y pétalos.

Experimentar con diferentes arquitecturas de redes neuronales y con diferentes configuraciones de entrenamiento para abordar el problema.

Puesto que se trata de un problema de clasificación multiclase, es conveniente que la capa de salida tenga tantas neuronas como clases y que la función de activación para estas neuronas sea la función softmax, que dada la tupla de entradas de las neuronas las normaliza a valores en el intervalo (0, 1) representando la probabilidad de pertenencia a a cada clase:

$$softmax(z_1, \ldots, z_n) = \left(\frac{\mathrm{e}^{z_1}}{\sum_{i=1}^{n} \mathrm{e}^{z_i}}, \ldots, \frac{\mathrm{e}^{z_n}}{\sum_{i=1}^{n} \mathrm{e}^{z_i}} \right)$$

De esta forma, cada ejemplo se clasificará en la clase más probable.

La función de pérdida a minimizar adecuada en este caso es la entropía cruzada categórica (`'categorical_crossentropy'`)

$$C(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} \mathbf{y}_i \log \hat{\mathbf{y}}_i$$

donde $\mathbf{y}$ es la codificación *one-hot* de la clase correcta e $\hat{\mathbf{y}}$ es el vector de probabilidades de pertenencia a cada clase proporcionado por el modelo.

Para codificar la variable objetivo mediante *one-hot* es útil la función `get_dummies` de la biblioteca Pandas.

In [40]:
```
objetivo = iris['Especie']
objetivo = pandas.get_dummies(objetivo)
print(objetivo)
objetivo = objetivo.to_numpy()
print(objetivo)
```

```
     Iris-setosa  Iris-versicolor  Iris-virginica
0              1                0               0
1              1                0               0
2              1                0               0
3              1                0               0
4              1                0               0
..           ...              ...             ...
145            0                0               1
146            0                0               1
147            0                0               1
148            0                0               1
149            0                0               1

[150 rows x 3 columns]
[[1 0 0]
 [1 0 0]
 [1 0 0]
 ...
 [0 0 1]
 [0 0 1]
 [0 0 1]]
```

El objetivo último es construir una red neuronal con una exactitud sobre el conjunto de prueba superior a 0.9.

In [ ]: