Deep Learning - Cats & Dogs Classification

January 10, 2021

1 Práctica 4

1.1 Clasificación de imágenes de perros y gatos empleando técnicas de Deep Learning

1.1.1 Introducción

El objetivo de esta práctica es establecer un primer contacto con el mundo de las redes neuronales y el Deep Learning, en concreto en el ámbito de la Visión por Computador. Para tal fin se propone construir una red neuronal capaz de clasificar imágenes de perros y gatos.

Para la realización de la práctica se empleará la API Keras de TensorFlow, una herramienta muy utilizada actualmente. Además se va ha hacer uso de la plataforma Google Colaboratory, que permite, entre otras cosas, hacer uso gratuito de potentes GPU en servidores remotos. Así pues, esta práctica tambien sirve como una primera toma de contacto con estas conocidas y útiles herramientas.

1.1.2 Datos

En este práctica se hará uso de un dataset que contiene un total de 3000 imágenes de perros y gatos de diferentes tamaños. Este dataset ya se proporciona con la organización de directorios apropiada para este ejercicio:

- Cat_dog_filtered -> directorio padre
- Train -> conjunto de entrenamiento
  - Cats -> Clase de gatos
  - Dogs -> Clase de perros
- Valdiation -> Conjunto de validación
  - Cats -> Clase de gatos
  - Dogs -> Clase de perros

Data augmentation La técnica del *data augmentation* permite aumentar el conjunto de datos de datos original. En particular, permite crear nuevas imágenes aplicando transformaciones geométricas sobre las existente, generando así nuevas imágenes realistas para entrenar el modelo. Esta técnica nos permite evitar el *overfitting* del modelo cuando no se dispone de un conjunto muy grande de datos, lo cual puede derivar en el modelado del ruido presente en estos.

1.2 Comienzo de la práctica

Primero de todo se importan todas aquellas librerías necesarias para el desarrollo de la práctica. De Keras, nos interesa la librería *layers*, la cual permite instanciar objetos a modo de diferentes capas

de la red neuronal, y *Sequential*, que será el modelo a emplear en esta práctica. Este modelo es apropiado cuando se busca una red construida por una secuencia de capas, donde cada una recibe con entrada un solo tensor y tiene como salida otro tensor.

[ ]:
```
import tensorflow as tf
import matplotlib.pyplot as plt

from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
```

A continuación confirmamos que nos podemos conectar a una GPU con TensorFlow.

[ ]:
```
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
Found GPU at: /device:GPU:0
```

Bien, parece que existe una conexión.

Se definen algunos parámetros importantes para el desarrollo de la práctica:

- *Batch size* : tamaño del *batch* de imágenes que se leerá cada vez del disco.
- *img_height* y *img_width* : tamaño de la imagen leída. Si una imagen del conjunto no coincide con este tamaño, se transforma a dichas dimensiones.

[ ]:
```
batch_size = 25 #adecuado para el problema
img_height = 256
img_width = 256
```

Data augmentation A continuación se crean los objetos que van a permitir coger *batches* de imágenes desde disco y generar artificialmente nuevas en tiempo real. Este objeto es el *ImageDataGenerator* de Keras. En su constructor, permite definir una serie de parámetros a utilizar en la generación de imágenes, como el ángulo máximo de rotación (*rotation range*) o la fracción total de zoom (*zoom range*).

Se definen así los generadores de imágenes para el conjunto de entrenamiento (*train_datagen*) y validación (*val_datagen*).

[ ]:
```
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center = True,
    featurewise_std_normalization = True,
    rotation_range = 90,
    width_shift_range = 0.1,
    height_shift_range = 0.1,
    zoom_range = 0.2
)
```

```
val_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
    featurewise_center = True,
    featurewise_std_normalization = True,
    rotation_range = 90,
    width_shift_range = 0.1,
    height_shift_range = 0.1,
    zoom_range = 0.2
)
```

Se definen los directorios donde residen las imágenes buscadas. El argumento *class mode* permite etiquetar automáticamente de forma binaria las imágenes pertenecientes a las dos clases: perros y gatos divididos en directorios separados. Esta infromación se almacena en el objeto *train_generator* y *validation_generator*, que se emplearán posteriormente cuando se entrene el modelo. Serán los encargados de leer imágenes de disco y aplicar en tiempo real el *data augmentation*.

Una gran ventaja de emplear estos generadores en la práctica es que no es necesario que el conjunto entero de datos quepa en memoria RAM, lo cual es poco realista en la moyoría de problemas. Esto es así ya que durante el entrenamiento del modelo se van cargando *batches* de tamaño reducido desde disco y no se carga el dataset entero. Cada *batch* se emplea para realizar un paso en el descenso del gradiente. Cuando se ha completado, se carga otro *batch* y se realiza lo mismo.

[ ]:
```
train_generator = train_datagen.flow_from_directory('./drive/MyDrive/DP_Vision/
↪in/cat_and_dog_filtered/train',
                                                    target_size =␣
↪(img_height,img_width),
                                                    batch_size = batch_size,
                                                    class_mode = 'binary')

validation_generator = val_datagen.flow_from_directory('./drive/MyDrive/
↪DP_Vision/in/cat_and_dog_filtered/validation',
                                                    target_size =␣
↪(img_height,img_width),
                                                    batch_size = batch_size,
                                                    class_mode = 'binary')
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

Por salida estándar se nos muestra el número de imágenes de entrenamiento y validación encontradas y el número de clases total en ambos grupos. En este caso 2 clases, perros y gatos.

Modelo 1 Se define la arquitectura del primer modelo. Lo primero de todo es la capa de entrada, en la cual definimos el tamaño de las imágenes. En este caso, al ser imágenes RGB se debe poner un 3 en la tercera dimensión (número de canales). Posteriormente se alternan capas de convolución y capas de *max pooling*. Las capas de *max pooling* ayudan a reducir la dimensionalidad de las capas de convolución extrayendo el valor máximo de cada paso del filtro por la imagen. Además permiten cada vez más identificar carácterísticas de más alto nivel a medida que se hace más profunda la red. Finalmente se aplana a una sola dimensión el resultado de las capas de convolución, de tal

manera que se dispone la información en el formato adecuado para introducirlo en un perceptrón multicapa. Este estrá compuesto por una capa oculta de 128 neuronas y una capa de salida que emplea la función *softmax* para generar una salida con valores entre 0 y 1, tal que la suma de las salidas sea 1. Así se consigue una distribución de probabilidad de la pertenencia de una observación (imagen) a cada clase.

La diferencia entre este modelo y el siguiente es la función de activación y de coste, como se verá más adelante.

[ ]:
```
model = Sequential([
  layers.InputLayer(input_shape=(img_height,img_width,3)),
  layers.Conv2D(10, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(20, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(40, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(2,activation='softmax') # 2 classes
])
```

A continuación se defininen la función de pérdida a optimizar, así como la métrica a evaluar durante el entrenamiento. Como función de pérdida se escoge la *sparse categorical cross entropy*, que según la documentación oficial, es válida únicamente cuando se disponen de dos o más salidas en forma de distribución de probabilidad. La métrica escogida es el *accuracy*.

También se podría seleccionar el método de optimización de la función de coste. En este caso se escoge el de por defecto, el algoritmo RMSprop.

[ ]:
```
model.compile(#optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
```

La función *summary* nos permite ver de forma esquemática la arquitectura de la red creada.

[ ]:
```
model.summary()
```

```
Model: "sequential_10"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_30 (Conv2D)           (None, 256, 256, 10)      280
_________________________________________________________________
max_pooling2d_30 (MaxPooling (None, 128, 128, 10)      0
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 128, 128, 20)      1820
_________________________________________________________________
max_pooling2d_31 (MaxPooling (None, 64, 64, 20)        0
_________________________________________________________________
```

```
conv2d_32 (Conv2D)           (None, 64, 64, 40)        7240
_________________________________________________________________
max_pooling2d_32 (MaxPooling (None, 32, 32, 40)        0
_________________________________________________________________
flatten_10 (Flatten)         (None, 40960)             0
_________________________________________________________________
dense_20 (Dense)             (None, 128)               5243008
_________________________________________________________________
dense_21 (Dense)             (None, 2)                 258
=================================================================
Total params: 5,252,606
Trainable params: 5,252,606
Non-trainable params: 0
_________________________________________________________________
```

Finalmente se procede a entrenar el modelo. Se hace uso de la función *fit_generator*, que nos permite entrenar el modelo haciendo uso de los generadores creados tanto para el conjunto de entrenamiento como validación. El arguemento *steps_per_epoch* permite establecer el número de pasos de descenso del gradiente a realizar por *epoch*. Es decir, el número de *batches* por *epoch*. En este caso, para asegurar que no se excede el número de imágenes disponibles (lo cual ha sucedido), este argumento se inicializa al número de *batches* máximo que encaja con las imágenes que se tienen.

En un principio el número de *batches* se inicializó a 32 y los pasos por *epoch* a 100, lo cual supone un total de 3200 imágenes. Esto hacía saltar un error que daba entender que se paraba el entrenamiento por insuficiencia de *inputs*. Así que se modificó a los valores actuales.

[ ]:
```
history = model.fit_generator (
    train_generator,
    steps_per_epoch = 2000/batch_size,
    epochs = 100,
    validation_data = validation_generator,
    validation_steps=1000/batch_size
)
```

```
/usr/local/lib/python3.6/dist-
packages/tensorflow/python/keras/engine/training.py:1844: UserWarning:
`Model.fit_generator` is deprecated and will be removed in a future version.
Please use `Model.fit`, which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '
/usr/local/lib/python3.6/dist-
packages/keras_preprocessing/image/image_data_generator.py:720: UserWarning:
This ImageDataGenerator specifies `featurewise_center`, but it hasn't been fit
on any training data. Fit it first by calling `.fit(numpy_data)`.
  warnings.warn('This ImageDataGenerator specifies '
/usr/local/lib/python3.6/dist-
packages/keras_preprocessing/image/image_data_generator.py:728: UserWarning:
This ImageDataGenerator specifies `featurewise_std_normalization`, but it hasn't
been fit on any training data. Fit it first by calling `.fit(numpy_data)`.
  warnings.warn('This ImageDataGenerator specifies '
```

```
Epoch 1/100
80/80 [==============================] - 48s 595ms/step - loss: 212.7686 -
accuracy: 0.5028 - val_loss: 0.6928 - val_accuracy: 0.5810
Epoch 2/100
80/80 [==============================] - 46s 581ms/step - loss: 0.7317 -
accuracy: 0.5615 - val_loss: 0.6864 - val_accuracy: 0.5650
Epoch 3/100
80/80 [==============================] - 46s 578ms/step - loss: 0.7147 -
accuracy: 0.5540 - val_loss: 0.6876 - val_accuracy: 0.5700
Epoch 4/100
80/80 [==============================] - 46s 576ms/step - loss: 0.7021 -
accuracy: 0.5568 - val_loss: 0.7001 - val_accuracy: 0.5390
Epoch 5/100
80/80 [==============================] - 46s 577ms/step - loss: 0.7108 -
accuracy: 0.5722 - val_loss: 0.6642 - val_accuracy: 0.5850
Epoch 6/100
80/80 [==============================] - 46s 575ms/step - loss: 0.7028 -
accuracy: 0.5855 - val_loss: 0.6917 - val_accuracy: 0.5580
Epoch 7/100
80/80 [==============================] - 46s 581ms/step - loss: 0.7110 -
accuracy: 0.5877 - val_loss: 0.6974 - val_accuracy: 0.5460
Epoch 8/100
80/80 [==============================] - 46s 574ms/step - loss: 0.7199 -
accuracy: 0.5687 - val_loss: 0.6660 - val_accuracy: 0.6060
Epoch 9/100
80/80 [==============================] - 47s 587ms/step - loss: 0.6781 -
accuracy: 0.6226 - val_loss: 0.6496 - val_accuracy: 0.6220
Epoch 10/100
80/80 [==============================] - 46s 582ms/step - loss: 0.6647 -
accuracy: 0.6153 - val_loss: 0.7086 - val_accuracy: 0.5610
Epoch 11/100
80/80 [==============================] - 46s 579ms/step - loss: 0.6771 -
accuracy: 0.6277 - val_loss: 0.6916 - val_accuracy: 0.5810
Epoch 12/100
80/80 [==============================] - 46s 578ms/step - loss: 0.6851 -
accuracy: 0.6200 - val_loss: 0.7295 - val_accuracy: 0.6250
Epoch 13/100
80/80 [==============================] - 46s 578ms/step - loss: 0.6480 -
accuracy: 0.6394 - val_loss: 0.6433 - val_accuracy: 0.6200
Epoch 14/100
80/80 [==============================] - 46s 583ms/step - loss: 0.6657 -
accuracy: 0.6390 - val_loss: 0.7583 - val_accuracy: 0.5680
Epoch 15/100
80/80 [==============================] - 46s 575ms/step - loss: 0.6575 -
accuracy: 0.6286 - val_loss: 0.6312 - val_accuracy: 0.6630
Epoch 16/100
80/80 [==============================] - 46s 579ms/step - loss: 0.6471 -
accuracy: 0.6305 - val_loss: 0.6188 - val_accuracy: 0.6550
```

```
Epoch 17/100
80/80 [==============================] - 46s 574ms/step - loss: 0.6441 -
accuracy: 0.6587 - val_loss: 0.6045 - val_accuracy: 0.6880
Epoch 18/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6311 -
accuracy: 0.6675 - val_loss: 0.6624 - val_accuracy: 0.6210
Epoch 19/100
80/80 [==============================] - 46s 575ms/step - loss: 0.6329 -
accuracy: 0.6548 - val_loss: 0.6580 - val_accuracy: 0.6070
Epoch 20/100
80/80 [==============================] - 46s 574ms/step - loss: 0.6158 -
accuracy: 0.6593 - val_loss: 0.7263 - val_accuracy: 0.5780
Epoch 21/100
80/80 [==============================] - 46s 580ms/step - loss: 0.6024 -
accuracy: 0.6821 - val_loss: 0.6286 - val_accuracy: 0.6810
Epoch 22/100
80/80 [==============================] - 46s 573ms/step - loss: 0.6165 -
accuracy: 0.6454 - val_loss: 0.6598 - val_accuracy: 0.6590
Epoch 23/100
80/80 [==============================] - 46s 575ms/step - loss: 0.6232 -
accuracy: 0.6965 - val_loss: 0.5968 - val_accuracy: 0.6880
Epoch 24/100
80/80 [==============================] - 46s 574ms/step - loss: 0.6133 -
accuracy: 0.6736 - val_loss: 0.6160 - val_accuracy: 0.6680
Epoch 25/100
80/80 [==============================] - 46s 575ms/step - loss: 0.6033 -
accuracy: 0.6658 - val_loss: 0.6000 - val_accuracy: 0.6810
Epoch 26/100
80/80 [==============================] - 46s 575ms/step - loss: 0.6125 -
accuracy: 0.6785 - val_loss: 0.6253 - val_accuracy: 0.6520
Epoch 27/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6615 -
accuracy: 0.6739 - val_loss: 0.6326 - val_accuracy: 0.6870
Epoch 28/100
80/80 [==============================] - 46s 578ms/step - loss: 0.5937 -
accuracy: 0.6786 - val_loss: 0.6105 - val_accuracy: 0.6800
Epoch 29/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5786 -
accuracy: 0.6874 - val_loss: 0.6581 - val_accuracy: 0.6380
Epoch 30/100
80/80 [==============================] - 46s 577ms/step - loss: 0.5994 -
accuracy: 0.6855 - val_loss: 0.6932 - val_accuracy: 0.6580
Epoch 31/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5987 -
accuracy: 0.6905 - val_loss: 0.5946 - val_accuracy: 0.6800
Epoch 32/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5789 -
accuracy: 0.6969 - val_loss: 0.6360 - val_accuracy: 0.6250
```

```
Epoch 33/100
80/80 [==============================] - 46s 577ms/step - loss: 0.5846 -
accuracy: 0.6812 - val_loss: 0.5933 - val_accuracy: 0.6720
Epoch 34/100
80/80 [==============================] - 46s 582ms/step - loss: 0.5790 -
accuracy: 0.7020 - val_loss: 0.6007 - val_accuracy: 0.6650
Epoch 35/100
80/80 [==============================] - 46s 576ms/step - loss: 0.5920 -
accuracy: 0.6836 - val_loss: 0.8272 - val_accuracy: 0.5530
Epoch 36/100
80/80 [==============================] - 46s 576ms/step - loss: 0.6051 -
accuracy: 0.6960 - val_loss: 0.5966 - val_accuracy: 0.6870
Epoch 37/100
80/80 [==============================] - 46s 576ms/step - loss: 0.5757 -
accuracy: 0.7069 - val_loss: 0.5829 - val_accuracy: 0.6900
Epoch 38/100
80/80 [==============================] - 46s 574ms/step - loss: 0.5847 -
accuracy: 0.7053 - val_loss: 0.5818 - val_accuracy: 0.6830
Epoch 39/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5833 -
accuracy: 0.7043 - val_loss: 0.5777 - val_accuracy: 0.7060
Epoch 40/100
80/80 [==============================] - 46s 573ms/step - loss: 0.5772 -
accuracy: 0.6990 - val_loss: 0.5849 - val_accuracy: 0.7070
Epoch 41/100
80/80 [==============================] - 46s 580ms/step - loss: 0.5708 -
accuracy: 0.7136 - val_loss: 0.5790 - val_accuracy: 0.6880
Epoch 42/100
80/80 [==============================] - 47s 589ms/step - loss: 0.5661 -
accuracy: 0.7052 - val_loss: 0.5722 - val_accuracy: 0.7180
Epoch 43/100
80/80 [==============================] - 46s 581ms/step - loss: 0.5807 -
accuracy: 0.7128 - val_loss: 0.5681 - val_accuracy: 0.6930
Epoch 44/100
80/80 [==============================] - 46s 581ms/step - loss: 0.5768 -
accuracy: 0.6969 - val_loss: 0.7631 - val_accuracy: 0.5700
Epoch 45/100
80/80 [==============================] - 46s 577ms/step - loss: 0.5877 -
accuracy: 0.6909 - val_loss: 0.5805 - val_accuracy: 0.7070
Epoch 46/100
80/80 [==============================] - 46s 581ms/step - loss: 0.5761 -
accuracy: 0.6998 - val_loss: 0.5890 - val_accuracy: 0.6970
Epoch 47/100
80/80 [==============================] - 47s 584ms/step - loss: 0.5642 -
accuracy: 0.7085 - val_loss: 0.5810 - val_accuracy: 0.7000
Epoch 48/100
80/80 [==============================] - 47s 593ms/step - loss: 0.5683 -
accuracy: 0.6974 - val_loss: 0.6504 - val_accuracy: 0.6690
```

```
Epoch 49/100
80/80 [==============================] - 48s 602ms/step - loss: 0.5858 -
accuracy: 0.6870 - val_loss: 0.5809 - val_accuracy: 0.7090
Epoch 50/100
80/80 [==============================] - 48s 602ms/step - loss: 0.5795 -
accuracy: 0.6962 - val_loss: 0.6041 - val_accuracy: 0.7060
Epoch 51/100
80/80 [==============================] - 49s 609ms/step - loss: 0.5806 -
accuracy: 0.7147 - val_loss: 0.6226 - val_accuracy: 0.6780
Epoch 52/100
80/80 [==============================] - 49s 615ms/step - loss: 0.5725 -
accuracy: 0.7003 - val_loss: 0.5930 - val_accuracy: 0.6970
Epoch 53/100
80/80 [==============================] - 49s 616ms/step - loss: 0.5577 -
accuracy: 0.7159 - val_loss: 0.7138 - val_accuracy: 0.5860
Epoch 54/100
80/80 [==============================] - 50s 627ms/step - loss: 0.5827 -
accuracy: 0.7091 - val_loss: 0.7646 - val_accuracy: 0.6060
Epoch 55/100
80/80 [==============================] - 50s 622ms/step - loss: 0.5773 -
accuracy: 0.7033 - val_loss: 0.5700 - val_accuracy: 0.7100
Epoch 56/100
80/80 [==============================] - 50s 627ms/step - loss: 0.5567 -
accuracy: 0.7228 - val_loss: 0.5790 - val_accuracy: 0.6810
Epoch 57/100
80/80 [==============================] - 50s 624ms/step - loss: 0.5813 -
accuracy: 0.7011 - val_loss: 0.5698 - val_accuracy: 0.6940
Epoch 58/100
80/80 [==============================] - 50s 624ms/step - loss: 0.5895 -
accuracy: 0.7013 - val_loss: 0.5771 - val_accuracy: 0.6960
Epoch 59/100
80/80 [==============================] - 50s 629ms/step - loss: 0.5749 -
accuracy: 0.7025 - val_loss: 0.6038 - val_accuracy: 0.6710
Epoch 60/100
80/80 [==============================] - 50s 633ms/step - loss: 0.5568 -
accuracy: 0.7191 - val_loss: 0.5785 - val_accuracy: 0.7190
Epoch 61/100
80/80 [==============================] - 50s 626ms/step - loss: 0.5686 -
accuracy: 0.7074 - val_loss: 0.6030 - val_accuracy: 0.6830
Epoch 62/100
80/80 [==============================] - 50s 630ms/step - loss: 0.5587 -
accuracy: 0.7199 - val_loss: 0.6478 - val_accuracy: 0.6500
Epoch 63/100
80/80 [==============================] - 50s 630ms/step - loss: 0.5843 -
accuracy: 0.7016 - val_loss: 0.5635 - val_accuracy: 0.7100
Epoch 64/100
80/80 [==============================] - 51s 636ms/step - loss: 0.5606 -
accuracy: 0.7049 - val_loss: 0.5621 - val_accuracy: 0.7120
```

```
Epoch 65/100
80/80 [==============================] - 50s 631ms/step - loss: 0.5583 -
accuracy: 0.7001 - val_loss: 0.5682 - val_accuracy: 0.7060
Epoch 66/100
80/80 [==============================] - 51s 636ms/step - loss: 0.5689 -
accuracy: 0.6921 - val_loss: 0.5637 - val_accuracy: 0.7110
Epoch 67/100
80/80 [==============================] - 51s 638ms/step - loss: 0.5516 -
accuracy: 0.7334 - val_loss: 0.5625 - val_accuracy: 0.7130
Epoch 68/100
80/80 [==============================] - 51s 640ms/step - loss: 0.5647 -
accuracy: 0.7244 - val_loss: 0.5459 - val_accuracy: 0.7320
Epoch 69/100
80/80 [==============================] - 51s 639ms/step - loss: 0.5507 -
accuracy: 0.6967 - val_loss: 0.5361 - val_accuracy: 0.7370
Epoch 70/100
80/80 [==============================] - 51s 638ms/step - loss: 0.5515 -
accuracy: 0.7184 - val_loss: 0.5767 - val_accuracy: 0.6960
Epoch 71/100
80/80 [==============================] - 50s 631ms/step - loss: 0.5588 -
accuracy: 0.7203 - val_loss: 0.5800 - val_accuracy: 0.7010
Epoch 72/100
80/80 [==============================] - 49s 613ms/step - loss: 0.5582 -
accuracy: 0.7131 - val_loss: 0.5815 - val_accuracy: 0.7060
Epoch 73/100
80/80 [==============================] - 50s 622ms/step - loss: 0.5834 -
accuracy: 0.7244 - val_loss: 0.5840 - val_accuracy: 0.6880
Epoch 74/100
80/80 [==============================] - 50s 622ms/step - loss: 0.5854 -
accuracy: 0.6884 - val_loss: 0.5528 - val_accuracy: 0.7250
Epoch 75/100
80/80 [==============================] - 50s 623ms/step - loss: 0.5414 -
accuracy: 0.7393 - val_loss: 0.5675 - val_accuracy: 0.7100
Epoch 76/100
80/80 [==============================] - 50s 630ms/step - loss: 0.5541 -
accuracy: 0.7194 - val_loss: 0.5690 - val_accuracy: 0.7120
Epoch 77/100
80/80 [==============================] - 51s 637ms/step - loss: 0.5544 -
accuracy: 0.7109 - val_loss: 0.5641 - val_accuracy: 0.7150
Epoch 78/100
80/80 [==============================] - 51s 637ms/step - loss: 0.5475 -
accuracy: 0.7435 - val_loss: 0.5820 - val_accuracy: 0.6960
Epoch 79/100
80/80 [==============================] - 51s 641ms/step - loss: 0.5377 -
accuracy: 0.7314 - val_loss: 0.5893 - val_accuracy: 0.6670
Epoch 80/100
80/80 [==============================] - 51s 639ms/step - loss: 0.5418 -
accuracy: 0.7443 - val_loss: 0.5816 - val_accuracy: 0.7240
```

```
Epoch 81/100
80/80 [==============================] - 51s 639ms/step - loss: 0.5353 -
accuracy: 0.7301 - val_loss: 0.8561 - val_accuracy: 0.6640
Epoch 82/100
80/80 [==============================] - 50s 622ms/step - loss: 0.5869 -
accuracy: 0.7162 - val_loss: 0.5578 - val_accuracy: 0.7160
Epoch 83/100
80/80 [==============================] - 49s 609ms/step - loss: 0.5720 -
accuracy: 0.7067 - val_loss: 0.5620 - val_accuracy: 0.7220
Epoch 84/100
80/80 [==============================] - 47s 591ms/step - loss: 0.5410 -
accuracy: 0.7362 - val_loss: 0.6043 - val_accuracy: 0.7110
Epoch 85/100
80/80 [==============================] - 47s 594ms/step - loss: 0.5519 -
accuracy: 0.7284 - val_loss: 0.5598 - val_accuracy: 0.7280
Epoch 86/100
80/80 [==============================] - 47s 586ms/step - loss: 0.5383 -
accuracy: 0.7417 - val_loss: 0.5821 - val_accuracy: 0.7160
Epoch 87/100
80/80 [==============================] - 46s 577ms/step - loss: 0.5436 -
accuracy: 0.7276 - val_loss: 0.5458 - val_accuracy: 0.7290
Epoch 88/100
80/80 [==============================] - 46s 579ms/step - loss: 0.5384 -
accuracy: 0.7388 - val_loss: 0.5694 - val_accuracy: 0.7000
Epoch 89/100
80/80 [==============================] - 46s 574ms/step - loss: 0.5606 -
accuracy: 0.7096 - val_loss: 0.5786 - val_accuracy: 0.7300
Epoch 90/100
80/80 [==============================] - 46s 576ms/step - loss: 0.5501 -
accuracy: 0.7191 - val_loss: 0.5931 - val_accuracy: 0.6840
Epoch 91/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5403 -
accuracy: 0.7096 - val_loss: 0.5620 - val_accuracy: 0.7100
Epoch 92/100
80/80 [==============================] - 46s 581ms/step - loss: 0.5643 -
accuracy: 0.7081 - val_loss: 0.5706 - val_accuracy: 0.7020
Epoch 93/100
80/80 [==============================] - 46s 578ms/step - loss: 0.5531 -
accuracy: 0.7105 - val_loss: 0.6005 - val_accuracy: 0.7030
Epoch 94/100
80/80 [==============================] - 46s 573ms/step - loss: 0.5299 -
accuracy: 0.7401 - val_loss: 0.5523 - val_accuracy: 0.7340
Epoch 95/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5385 -
accuracy: 0.7439 - val_loss: 0.5754 - val_accuracy: 0.7100
Epoch 96/100
80/80 [==============================] - 46s 573ms/step - loss: 0.5509 -
accuracy: 0.7305 - val_loss: 0.5650 - val_accuracy: 0.7060
```

```
Epoch 97/100
80/80 [==============================] - 46s 575ms/step - loss: 0.6410 -
accuracy: 0.7368 - val_loss: 0.6151 - val_accuracy: 0.6800
Epoch 98/100
80/80 [==============================] - 46s 574ms/step - loss: 0.5579 -
accuracy: 0.7273 - val_loss: 0.6079 - val_accuracy: 0.6900
Epoch 99/100
80/80 [==============================] - 47s 584ms/step - loss: 0.5487 -
accuracy: 0.7199 - val_loss: 0.5760 - val_accuracy: 0.7180
Epoch 100/100
80/80 [==============================] - 46s 575ms/step - loss: 0.8669 -
accuracy: 0.7350 - val_loss: 0.6054 - val_accuracy: 0.6980
```

A continuación se realizan unas gráficas que permiten observas la evolución del *accuracy* y el valor de la función de coste a cada *epoch* del entrenamiento y por cada conjunto de imágenes (entrenamiento y validación).

[ ]:
```
epochs = 100

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

Lo primero a comentar es que el entrenamiento del modelo parece ir bien: a cuantas más *epochs*, menor error en la clasificación. Así que el proceso de aprendizaje está asegurado. Por otro lado, el *accuracy* del entrenamiento y el de validación son similares en cada *epoch*, por lo que no existen indicio de *overfitting*. Si que se observan algunas *epochs* en lo que existe una mayor diferencia entre entrenamiento y validación, pero son casos puntuales y no mayor a 0.2. Por el contrario, un ejemplo de *overfitting* sería aquel en el que el *accuracy* del entrenamiento crece a medida que se hacen más *epochs*, mientras que el de la validación se mantiene casi constante a partir de un cierto punto. Esto indicaría un aprendizaje demasiado exacto del conjunto de entrenamiento y, por lo tanto, mala generalización a nuevas observaciones.

En la segunda gráfica se contempla como el error a partir de la primera *epoch* desciende a valores muy cercano a 0 en el conjunto de entrenamiento. Dicho descenso no ocurre en la validación ya que en el momento de evaluar la función de coste con este grupo, los pesos ya estan actualizados y, por lo tanto, se producen mejores resultados que al principio con una inicialización aleatoria de los pesos.

Modelo 2 En este segundo modelo se toma como función de activación de la útima capa la función sigmoide. Esta función dará la probabilidad de que una imagen pertenezca a la clase 1 (perros, en este caso). Por lo tanto, en la última capa solo se dispondrá de 1 neurona, a diferencia de las dos neuronas del modelo 1 que ofrecían las probabilidaded de que una imagen fuese perro o gato, cuya suma era igual a 1. Si se pusiesen dos neuronas con una función de activación sigmoidal, darían dos probabilidades independientes, lo cual resulta difícil de interpretar y posiblemente no sea correcto para el entrenamiento de la red.

La arquitecutra de esta red es idéntica a la del modelo 1 con tal de poder establecer una comparación valida posteriormente.

[ ]:
```
model_2 = Sequential([
  layers.InputLayer(input_shape=(img_height,img_width,3)),
  layers.Conv2D(10, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(20, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Conv2D(40, 3, padding='same', activation='relu'),
  layers.MaxPooling2D(),
  layers.Flatten(),
  layers.Dense(128, activation='relu'),
  layers.Dense(1,activation='sigmoid')
])
```

En este caso, al no disponerse de dos o más salidas en forma de distribución de probabilidad, la función de coste *sparse categorical cross entropy* no es adecuada para el entrenamiento. Por ese motivo se hace uso de *binary cross entropy*.

Tanto la métrica como el método de optimización se mantienen iguales al modelo 1.

[ ]:
```
model_2.compile(loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=['accuracy'])
```

[ ]:
```
model_2.summary()
```

```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_33 (Conv2D)           (None, 256, 256, 10)      280
_________________________________________________________________
max_pooling2d_33 (MaxPooling (None, 128, 128, 10)      0
_________________________________________________________________
conv2d_34 (Conv2D)           (None, 128, 128, 20)      1820
_________________________________________________________________
max_pooling2d_34 (MaxPooling (None, 64, 64, 20)        0
_________________________________________________________________
conv2d_35 (Conv2D)           (None, 64, 64, 40)        7240
_________________________________________________________________
```

```
max_pooling2d_35 (MaxPooling (None, 32, 32, 40)        0
_________________________________________________________________
flatten_11 (Flatten)         (None, 40960)             0
_________________________________________________________________
dense_22 (Dense)             (None, 128)               5243008
_________________________________________________________________
dense_23 (Dense)             (None, 1)                 129
=================================================================
Total params: 5,252,477
Trainable params: 5,252,477
Non-trainable params: 0
_________________________________________________________________
```

Se entrena el modelo con las mismas *epochs* y *steps_per_epoch*.

[ ]:
```
history_2 = model_2.fit_generator (
    train_generator,
    steps_per_epoch = 2000/batch_size,
    epochs = 100,
    validation_data = validation_generator,
    validation_steps=1000/batch_size
)
```

```
/usr/local/lib/python3.6/dist-
packages/tensorflow/python/keras/engine/training.py:1844: UserWarning:
`Model.fit_generator` is deprecated and will be removed in a future version.
Please use `Model.fit`, which supports generators.
  warnings.warn('`Model.fit_generator` is deprecated and '
/usr/local/lib/python3.6/dist-
packages/keras_preprocessing/image/image_data_generator.py:720: UserWarning:
This ImageDataGenerator specifies `featurewise_center`, but it hasn't been fit
on any training data. Fit it first by calling `.fit(numpy_data)`.
  warnings.warn('This ImageDataGenerator specifies '
/usr/local/lib/python3.6/dist-
packages/keras_preprocessing/image/image_data_generator.py:728: UserWarning:
This ImageDataGenerator specifies `featurewise_std_normalization`, but it hasn't
been fit on any training data. Fit it first by calling `.fit(numpy_data)`.
  warnings.warn('This ImageDataGenerator specifies '
```

```
Epoch 1/100
80/80 [==============================] - 47s 582ms/step - loss: 452.0248 -
accuracy: 0.4988 - val_loss: 0.7108 - val_accuracy: 0.5210
Epoch 2/100
80/80 [==============================] - 46s 582ms/step - loss: 0.7463 -
accuracy: 0.5034 - val_loss: 0.8171 - val_accuracy: 0.5020
Epoch 3/100
80/80 [==============================] - 46s 576ms/step - loss: 1.0355 -
accuracy: 0.5462 - val_loss: 0.6808 - val_accuracy: 0.5650
Epoch 4/100
```

```
80/80 [==============================] - 46s 576ms/step - loss: 0.7667 -
accuracy: 0.5448 - val_loss: 0.6808 - val_accuracy: 0.5540
Epoch 5/100
80/80 [==============================] - 46s 574ms/step - loss: 0.7024 -
accuracy: 0.5571 - val_loss: 0.6876 - val_accuracy: 0.5400
Epoch 6/100
80/80 [==============================] - 46s 576ms/step - loss: 0.7433 -
accuracy: 0.5567 - val_loss: 0.6802 - val_accuracy: 0.5610
Epoch 7/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6741 -
accuracy: 0.5571 - val_loss: 0.7002 - val_accuracy: 0.5480
Epoch 8/100
80/80 [==============================] - 46s 579ms/step - loss: 0.6942 -
accuracy: 0.5685 - val_loss: 0.7249 - val_accuracy: 0.5090
Epoch 9/100
80/80 [==============================] - 47s 585ms/step - loss: 0.7398 -
accuracy: 0.5342 - val_loss: 0.8728 - val_accuracy: 0.5460
Epoch 10/100
80/80 [==============================] - 46s 578ms/step - loss: 0.7641 -
accuracy: 0.5597 - val_loss: 0.6619 - val_accuracy: 0.5590
Epoch 11/100
80/80 [==============================] - 46s 578ms/step - loss: 0.6839 -
accuracy: 0.5726 - val_loss: 0.7357 - val_accuracy: 0.5700
Epoch 12/100
80/80 [==============================] - 46s 577ms/step - loss: 0.8729 -
accuracy: 0.5449 - val_loss: 0.6780 - val_accuracy: 0.5630
Epoch 13/100
80/80 [==============================] - 46s 579ms/step - loss: 0.6907 -
accuracy: 0.5227 - val_loss: 0.7029 - val_accuracy: 0.5250
Epoch 14/100
80/80 [==============================] - 46s 576ms/step - loss: 0.7181 -
accuracy: 0.5830 - val_loss: 0.8179 - val_accuracy: 0.5150
Epoch 15/100
80/80 [==============================] - 46s 579ms/step - loss: 0.7022 -
accuracy: 0.5391 - val_loss: 0.6688 - val_accuracy: 0.5700
Epoch 16/100
80/80 [==============================] - 47s 584ms/step - loss: 0.8244 -
accuracy: 0.5972 - val_loss: 0.6686 - val_accuracy: 0.5770
Epoch 17/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6778 -
accuracy: 0.5787 - val_loss: 0.6795 - val_accuracy: 0.5490
Epoch 18/100
80/80 [==============================] - 46s 577ms/step - loss: 0.7474 -
accuracy: 0.5942 - val_loss: 0.8118 - val_accuracy: 0.5240
Epoch 19/100
80/80 [==============================] - 46s 576ms/step - loss: 0.6868 -
accuracy: 0.5732 - val_loss: 0.6680 - val_accuracy: 0.6140
Epoch 20/100
```

```
80/80 [==============================] - 46s 578ms/step - loss: 0.6585 -
accuracy: 0.5724 - val_loss: 0.6474 - val_accuracy: 0.6270
Epoch 21/100
80/80 [==============================] - 46s 576ms/step - loss: 0.6627 -
accuracy: 0.6235 - val_loss: 0.6879 - val_accuracy: 0.6340
Epoch 22/100
80/80 [==============================] - 46s 581ms/step - loss: 0.6764 -
accuracy: 0.6303 - val_loss: 0.6826 - val_accuracy: 0.5950
Epoch 23/100
80/80 [==============================] - 47s 584ms/step - loss: 0.6367 -
accuracy: 0.6335 - val_loss: 0.7181 - val_accuracy: 0.6050
Epoch 24/100
80/80 [==============================] - 46s 576ms/step - loss: 0.7197 -
accuracy: 0.6614 - val_loss: 0.7663 - val_accuracy: 0.5350
Epoch 25/100
80/80 [==============================] - 46s 578ms/step - loss: 0.7473 -
accuracy: 0.6413 - val_loss: 0.6329 - val_accuracy: 0.6670
Epoch 26/100
80/80 [==============================] - 46s 575ms/step - loss: 0.6240 -
accuracy: 0.6497 - val_loss: 0.6801 - val_accuracy: 0.6380
Epoch 27/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6231 -
accuracy: 0.6791 - val_loss: 0.6365 - val_accuracy: 0.6650
Epoch 28/100
80/80 [==============================] - 46s 576ms/step - loss: 0.6810 -
accuracy: 0.6810 - val_loss: 0.6036 - val_accuracy: 0.6460
Epoch 29/100
80/80 [==============================] - 47s 584ms/step - loss: 0.6153 -
accuracy: 0.6801 - val_loss: 0.6767 - val_accuracy: 0.6260
Epoch 30/100
80/80 [==============================] - 46s 579ms/step - loss: 0.6430 -
accuracy: 0.6542 - val_loss: 0.6198 - val_accuracy: 0.6450
Epoch 31/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6340 -
accuracy: 0.6557 - val_loss: 0.6086 - val_accuracy: 0.6860
Epoch 32/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6423 -
accuracy: 0.6708 - val_loss: 0.6240 - val_accuracy: 0.6910
Epoch 33/100
80/80 [==============================] - 46s 576ms/step - loss: 0.6384 -
accuracy: 0.6578 - val_loss: 0.6403 - val_accuracy: 0.6270
Epoch 34/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6123 -
accuracy: 0.6597 - val_loss: 0.6473 - val_accuracy: 0.6630
Epoch 35/100
80/80 [==============================] - 46s 578ms/step - loss: 0.6146 -
accuracy: 0.6690 - val_loss: 0.5867 - val_accuracy: 0.6940
Epoch 36/100
```

```
80/80 [==============================] - 47s 584ms/step - loss: 0.6277 -
accuracy: 0.6727 - val_loss: 0.6104 - val_accuracy: 0.6540
Epoch 37/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6059 -
accuracy: 0.6861 - val_loss: 0.6070 - val_accuracy: 0.6780
Epoch 38/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6016 -
accuracy: 0.6702 - val_loss: 0.6375 - val_accuracy: 0.6550
Epoch 39/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6231 -
accuracy: 0.6705 - val_loss: 0.6469 - val_accuracy: 0.6630
Epoch 40/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5935 -
accuracy: 0.7000 - val_loss: 0.5721 - val_accuracy: 0.7050
Epoch 41/100
80/80 [==============================] - 46s 578ms/step - loss: 0.5956 -
accuracy: 0.6752 - val_loss: 0.6351 - val_accuracy: 0.6400
Epoch 42/100
80/80 [==============================] - 46s 581ms/step - loss: 0.6170 -
accuracy: 0.6821 - val_loss: 0.6068 - val_accuracy: 0.6860
Epoch 43/100
80/80 [==============================] - 47s 584ms/step - loss: 0.5913 -
accuracy: 0.6766 - val_loss: 0.6088 - val_accuracy: 0.6680
Epoch 44/100
80/80 [==============================] - 46s 580ms/step - loss: 0.6070 -
accuracy: 0.6952 - val_loss: 0.6268 - val_accuracy: 0.6710
Epoch 45/100
80/80 [==============================] - 46s 576ms/step - loss: 0.5982 -
accuracy: 0.6850 - val_loss: 0.5902 - val_accuracy: 0.6760
Epoch 46/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6474 -
accuracy: 0.7020 - val_loss: 0.5839 - val_accuracy: 0.6860
Epoch 47/100
80/80 [==============================] - 46s 576ms/step - loss: 0.6383 -
accuracy: 0.6654 - val_loss: 0.5970 - val_accuracy: 0.6880
Epoch 48/100
80/80 [==============================] - 46s 579ms/step - loss: 0.5794 -
accuracy: 0.7069 - val_loss: 0.5976 - val_accuracy: 0.6970
Epoch 49/100
80/80 [==============================] - 46s 577ms/step - loss: 0.7776 -
accuracy: 0.7212 - val_loss: 0.5713 - val_accuracy: 0.7200
Epoch 50/100
80/80 [==============================] - 47s 584ms/step - loss: 0.5963 -
accuracy: 0.6937 - val_loss: 0.6139 - val_accuracy: 0.6950
Epoch 51/100
80/80 [==============================] - 46s 578ms/step - loss: 0.5979 -
accuracy: 0.6965 - val_loss: 0.6193 - val_accuracy: 0.6730
Epoch 52/100
```

```
80/80 [==============================] - 46s 576ms/step - loss: 0.5890 -
accuracy: 0.6988 - val_loss: 0.6376 - val_accuracy: 0.6670
Epoch 53/100
80/80 [==============================] - 46s 577ms/step - loss: 0.6156 -
accuracy: 0.7211 - val_loss: 0.6148 - val_accuracy: 0.6590
Epoch 54/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5837 -
accuracy: 0.7015 - val_loss: 0.6119 - val_accuracy: 0.6930
Epoch 55/100
80/80 [==============================] - 46s 580ms/step - loss: 0.5823 -
accuracy: 0.7048 - val_loss: 0.5890 - val_accuracy: 0.6930
Epoch 56/100
80/80 [==============================] - 46s 582ms/step - loss: 0.5701 -
accuracy: 0.7038 - val_loss: 0.7542 - val_accuracy: 0.6890
Epoch 57/100
80/80 [==============================] - 46s 578ms/step - loss: 0.5929 -
accuracy: 0.6976 - val_loss: 0.5952 - val_accuracy: 0.6900
Epoch 58/100
80/80 [==============================] - 46s 578ms/step - loss: 0.5787 -
accuracy: 0.7098 - val_loss: 0.5802 - val_accuracy: 0.6880
Epoch 59/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5567 -
accuracy: 0.7058 - val_loss: 0.6547 - val_accuracy: 0.7080
Epoch 60/100
80/80 [==============================] - 46s 578ms/step - loss: 0.5995 -
accuracy: 0.6856 - val_loss: 0.6567 - val_accuracy: 0.6450
Epoch 61/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5926 -
accuracy: 0.6983 - val_loss: 0.5867 - val_accuracy: 0.6880
Epoch 62/100
80/80 [==============================] - 46s 579ms/step - loss: 0.5931 -
accuracy: 0.6831 - val_loss: 0.5988 - val_accuracy: 0.7070
Epoch 63/100
80/80 [==============================] - 46s 582ms/step - loss: 0.5699 -
accuracy: 0.7119 - val_loss: 0.6355 - val_accuracy: 0.7160
Epoch 64/100
80/80 [==============================] - 46s 579ms/step - loss: 0.5919 -
accuracy: 0.6969 - val_loss: 0.5924 - val_accuracy: 0.7040
Epoch 65/100
80/80 [==============================] - 46s 579ms/step - loss: 0.6099 -
accuracy: 0.7098 - val_loss: 0.6838 - val_accuracy: 0.6970
Epoch 66/100
80/80 [==============================] - 46s 575ms/step - loss: 0.5727 -
accuracy: 0.7151 - val_loss: 0.6089 - val_accuracy: 0.7150
Epoch 67/100
80/80 [==============================] - 46s 579ms/step - loss: 0.5716 -
accuracy: 0.7150 - val_loss: 0.6239 - val_accuracy: 0.6930
Epoch 68/100
```

```
80/80 [==============================] - 46s 578ms/step - loss: 0.5974 -
accuracy: 0.6915 - val_loss: 0.5962 - val_accuracy: 0.6650
Epoch 69/100
80/80 [==============================] - 47s 585ms/step - loss: 0.5633 -
accuracy: 0.6954 - val_loss: 0.7219 - val_accuracy: 0.6390
Epoch 70/100
80/80 [==============================] - 49s 612ms/step - loss: 0.5751 -
accuracy: 0.6970 - val_loss: 0.5911 - val_accuracy: 0.6960
Epoch 71/100
80/80 [==============================] - 46s 583ms/step - loss: 0.5727 -
accuracy: 0.6985 - val_loss: 0.5928 - val_accuracy: 0.7000
Epoch 72/100
80/80 [==============================] - 47s 584ms/step - loss: 0.5764 -
accuracy: 0.7008 - val_loss: 0.7621 - val_accuracy: 0.6680
Epoch 73/100
80/80 [==============================] - 46s 582ms/step - loss: 0.5726 -
accuracy: 0.7260 - val_loss: 0.6485 - val_accuracy: 0.7140
Epoch 74/100
80/80 [==============================] - 47s 585ms/step - loss: 0.5597 -
accuracy: 0.7253 - val_loss: 0.5914 - val_accuracy: 0.7040
Epoch 75/100
80/80 [==============================] - 47s 584ms/step - loss: 0.5683 -
accuracy: 0.7255 - val_loss: 0.7264 - val_accuracy: 0.6910
Epoch 76/100
80/80 [==============================] - 47s 586ms/step - loss: 0.5532 -
accuracy: 0.7312 - val_loss: 0.6369 - val_accuracy: 0.6690
Epoch 77/100
80/80 [==============================] - 47s 585ms/step - loss: 0.5970 -
accuracy: 0.7082 - val_loss: 0.7573 - val_accuracy: 0.6370
Epoch 78/100
80/80 [==============================] - 46s 580ms/step - loss: 0.7131 -
accuracy: 0.6977 - val_loss: 0.5726 - val_accuracy: 0.7000
Epoch 79/100
80/80 [==============================] - 46s 580ms/step - loss: 0.5970 -
accuracy: 0.6941 - val_loss: 0.6310 - val_accuracy: 0.6620
Epoch 80/100
80/80 [==============================] - 46s 579ms/step - loss: 0.5678 -
accuracy: 0.7114 - val_loss: 0.6888 - val_accuracy: 0.6500
Epoch 81/100
80/80 [==============================] - 46s 581ms/step - loss: 0.5534 -
accuracy: 0.7194 - val_loss: 0.7622 - val_accuracy: 0.6970
Epoch 82/100
80/80 [==============================] - 46s 581ms/step - loss: 0.5895 -
accuracy: 0.6883 - val_loss: 0.7205 - val_accuracy: 0.6590
Epoch 83/100
80/80 [==============================] - 47s 587ms/step - loss: 0.5859 -
accuracy: 0.7160 - val_loss: 0.5703 - val_accuracy: 0.7180
Epoch 84/100
```

```
80/80 [==============================] - 46s 580ms/step - loss: 0.5481 -
accuracy: 0.7123 - val_loss: 0.5995 - val_accuracy: 0.6790
Epoch 85/100
80/80 [==============================] - 46s 579ms/step - loss: 0.5698 -
accuracy: 0.7143 - val_loss: 0.5758 - val_accuracy: 0.7060
Epoch 86/100
80/80 [==============================] - 46s 583ms/step - loss: 0.5575 -
accuracy: 0.7228 - val_loss: 0.6731 - val_accuracy: 0.7120
Epoch 87/100
80/80 [==============================] - 46s 576ms/step - loss: 0.6075 -
accuracy: 0.7004 - val_loss: 0.8407 - val_accuracy: 0.7060
Epoch 88/100
80/80 [==============================] - 46s 580ms/step - loss: 0.6004 -
accuracy: 0.7109 - val_loss: 0.6691 - val_accuracy: 0.6690
Epoch 89/100
80/80 [==============================] - 46s 572ms/step - loss: 0.6071 -
accuracy: 0.6968 - val_loss: 0.5620 - val_accuracy: 0.7140
Epoch 90/100
80/80 [==============================] - 47s 592ms/step - loss: 0.5793 -
accuracy: 0.7002 - val_loss: 0.5858 - val_accuracy: 0.6990
Epoch 91/100
80/80 [==============================] - 46s 576ms/step - loss: 0.5463 -
accuracy: 0.7186 - val_loss: 0.5794 - val_accuracy: 0.7100
Epoch 92/100
80/80 [==============================] - 46s 574ms/step - loss: 0.5758 -
accuracy: 0.7072 - val_loss: 0.6430 - val_accuracy: 0.6320
Epoch 93/100
80/80 [==============================] - 46s 581ms/step - loss: 0.5308 -
accuracy: 0.7401 - val_loss: 0.5824 - val_accuracy: 0.7100
Epoch 94/100
80/80 [==============================] - 46s 578ms/step - loss: 0.5627 -
accuracy: 0.7169 - val_loss: 0.5900 - val_accuracy: 0.7120
Epoch 95/100
80/80 [==============================] - 46s 580ms/step - loss: 0.6008 -
accuracy: 0.7214 - val_loss: 0.6248 - val_accuracy: 0.6970
Epoch 96/100
80/80 [==============================] - 46s 578ms/step - loss: 0.6017 -
accuracy: 0.6988 - val_loss: 0.5917 - val_accuracy: 0.7090
Epoch 97/100
80/80 [==============================] - 46s 581ms/step - loss: 0.5571 -
accuracy: 0.7367 - val_loss: 0.5872 - val_accuracy: 0.7090
Epoch 98/100
80/80 [==============================] - 46s 579ms/step - loss: 0.5807 -
accuracy: 0.7199 - val_loss: 0.6039 - val_accuracy: 0.7140
Epoch 99/100
80/80 [==============================] - 46s 576ms/step - loss: 0.5669 -
accuracy: 0.7160 - val_loss: 0.6230 - val_accuracy: 0.6990
Epoch 100/100
```

```
80/80 [==============================] - 46s 577ms/step - loss: 0.5577 -
accuracy: 0.7329 - val_loss: 0.5716 - val_accuracy: 0.7060
```

Se realizan las mismas gráficas de los valores del *accuracy* y la función de coste.

[ ]:
```
epochs_2 = 100

acc_2 = history_2.history['accuracy']
val_acc_2 = history_2.history['val_accuracy']

loss_2 = history_2.history['loss']
val_loss_2 = history_2.history['val_loss']

epochs_range_2 = range(epochs_2)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range_2, acc_2, label='Training Accuracy')
plt.plot(epochs_range_2, val_acc_2, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy Model 2')

plt.subplot(1, 2, 2)
plt.plot(epochs_range_2, loss_2, label='Training Loss')
plt.plot(epochs_range_2, val_loss_2, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss Model 2')
plt.show()
```

A simple vista cuesta decidir que modelo funciona mejor observando las gráficas de ambos modelos.

En cuanto al *accuracy*, este modelo 2 parece mantener una mayor armonía en los valores de entrenamiento y validación, no aparecen diferencias puntuales que destaquen a diferencia de lo que pasaba en el modelo 1.

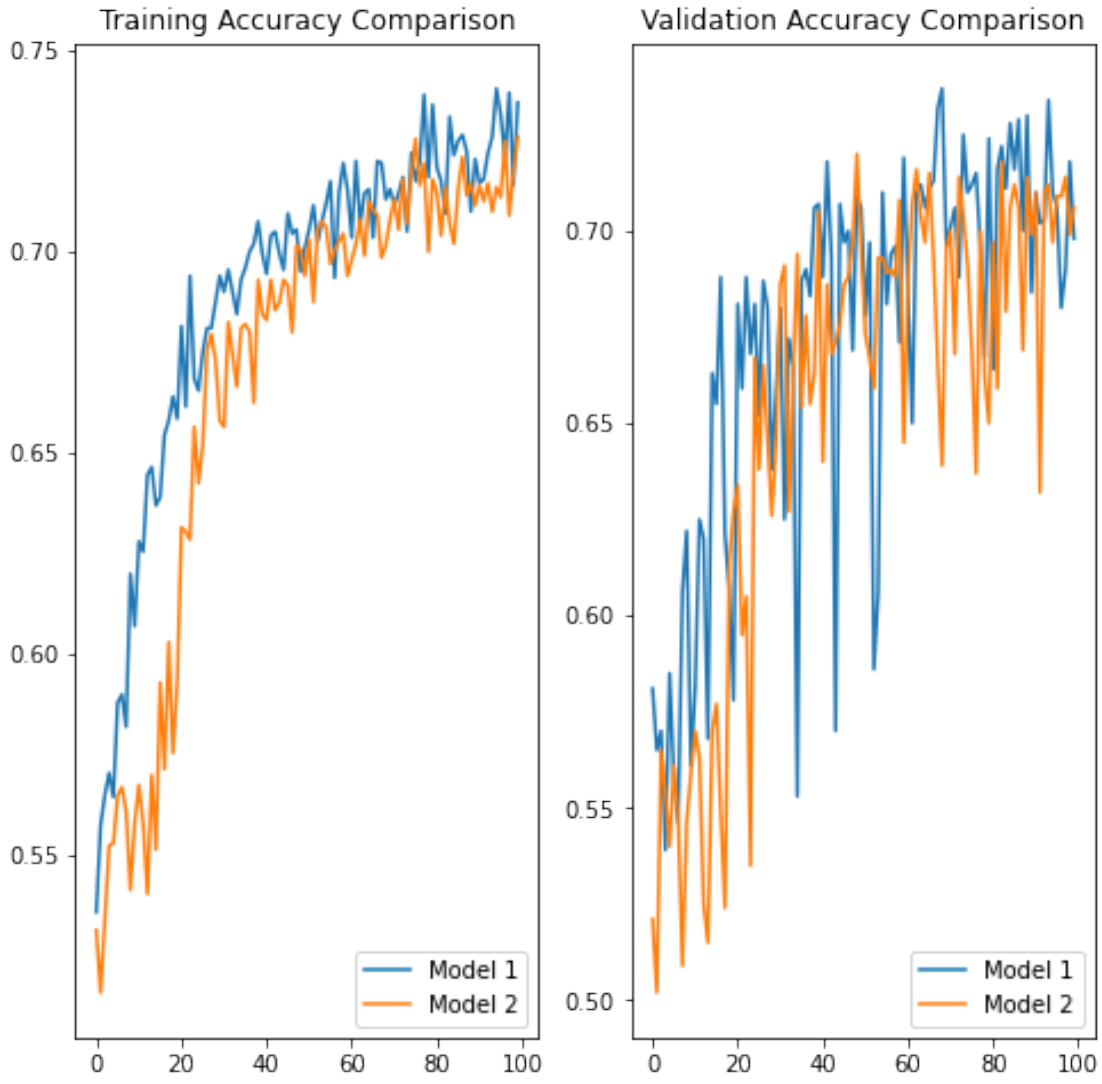
Por otro lado el valor del error, con las gráficas disponibles, parece bastante similar. No obstante, si se observan los valores numéricos que se ofrecen por la salida estándar durante el entrenamiento, se comprueba que el modelo 1 llega a valores un poco más bajos (en torno al 0.55-0.6) que el modelo 2 (en torno el 0.6-0.65).

A continuación se plantean una serie de gráficas que ayudan a realizar una mejor comparación y selección del mejor modelo.

[ ]:
```
epochs = 100
epoch_range = range(epochs)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Model 1')
plt.plot(epochs_range, acc_2, label='Model 2')
plt.legend(loc='lower right')
plt.title('Training Accuracy Comparison')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, val_acc, label='Model 1')
plt.plot(epochs_range, val_acc_2, label='Model 2')
plt.legend(loc='lower right')
plt.title('Validation Accuracy Comparison')
plt.show()
```



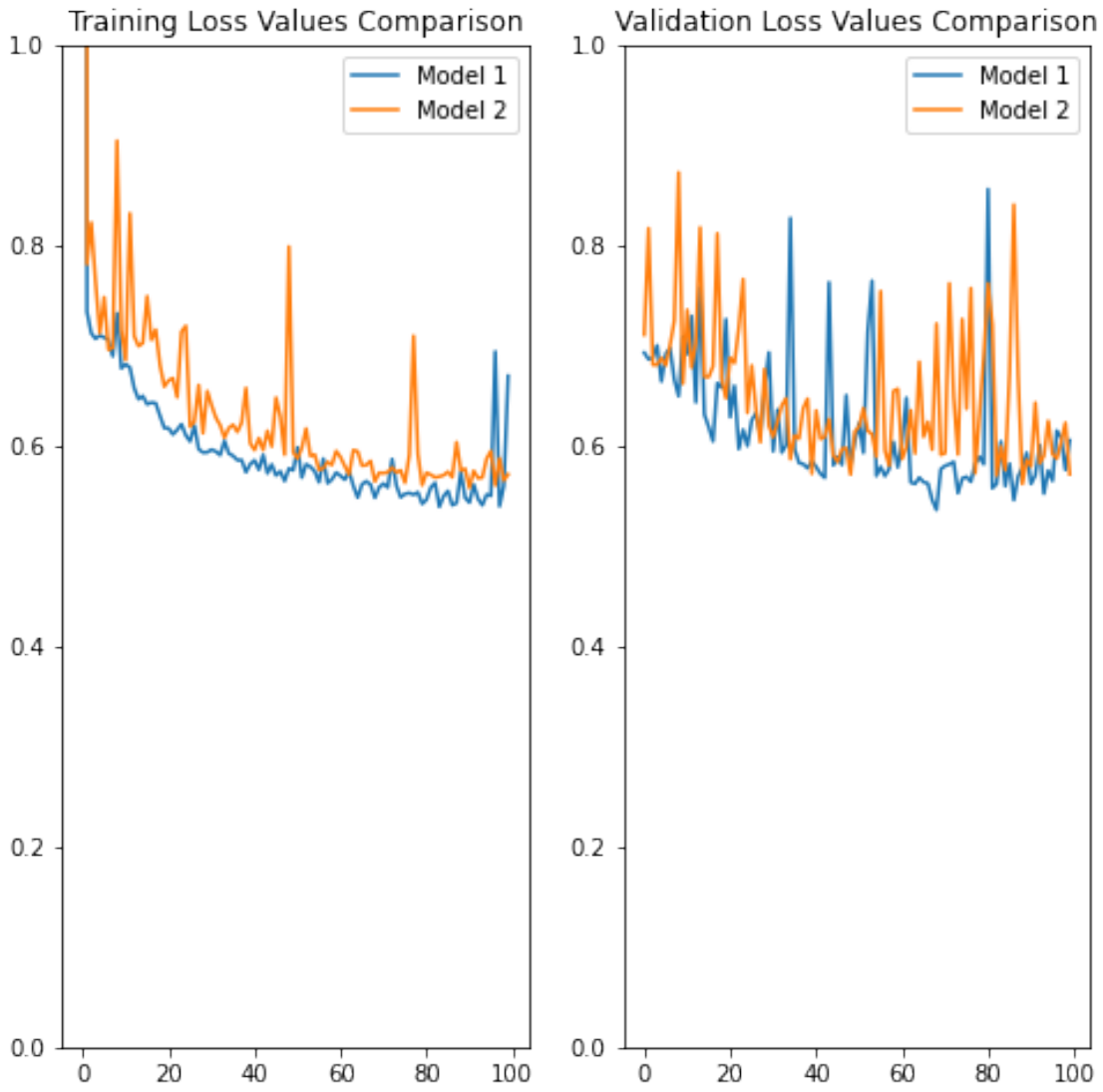
Como puede apreciarse, si solo se fija el punto de mira en el grupo de entrenamiento el modelo 1 obtiene mejor *accuracy* que el modelo 2 en todo momento, especialmente al principio del entrenamiento. Sin embargo este *accuracy* no es tan relevante como el de validación, donde si que queda expuesta la capacidad de generalización del modelo. En este segundo *accuracy*, ambos modelos presentan un progreso parecido. Sin embargo, si tuviese que elegir uno, sería el modelo 1, el cual se impone al modelo 2 tanto al principio como al final del modelo en términos de *accuracy*, a pesar de presentar algunos valles puntuales (ya se ha comentado que no son de gran importancia).

Finalmente se realizan unas gráficas del valor de la función de coste que permitan observar con mayor claridad los valores adquiridos por ambos modelos durante el entrenamiento.

[ ]:
```
plt.figure(figsize=(8,8))
plt.subplot(1,2,1)
plt.plot(epoch_range, loss, label = 'Model 1')
plt.plot(epoch_range, loss_2, label = 'Model 2')
plt.ylim([0,1])
plt.legend(loc='upper right')
plt.title('Training Loss Values Comparison')

plt.subplot(1,2,2)
plt.plot(epoch_range, val_loss, label = 'Model 1')
plt.plot(epoch_range, val_loss_2, label = 'Model 2')
plt.ylim([0,1])
plt.legend(loc='upper right')
plt.title('Validation Loss Values Comparison')
```

[ ]:
```
Text(0.5, 1.0, 'Validation Loss Values Comparison')
```



De nuevo, en cuanto al conjunto de entrenamiento, el modelo 1 parece tener en todo momento un valor de la función de coste menor al modelo 2. En cuanto al conjunto de validación, esto parece ser cierto también, aunque de una forma menos explícita. Son aspectos que también se han observado en las gráficas del accuracy, pero como es lógico, esta similitud tenía que existir.

Sin embargo, esta comparación de funciones de coste puede no ser válida ya que las funciones de coste son distintas, y por lo tanto el error puede estar en escalas distintas. Ambas funciones deben tener sus mínimos en 0, ya que sino sería imposible de partida obtener un error 0, y por lo tanto una clasificación perfecta (aunque eso suena idealista), pero para una misma diferencia entre la salida deseada y la obtenida, el error puede ser distinto depende de la función usada.