G03_Project2B

April 29, 2021

Pràctica: Project 2B
Autores: Bo Miquel Nordfeldt, Joan Muntaner, Helena Antich
Fecha: Abril 2021

1 Descripción de la práctica

El objetivo de esta práctica es diseñar una Red Neuronal Convolucional, o 'Convolution Neural Network' (CNN), capaz de clasificar imágenes, con buena precisión, en dos tipos: imágenes que contengan perros e imágenes que contengan gatos.

Para ello se ha partido de la arquitectura proporcionada por el profesor, la cual tenía un 'accuracy' relativamente bajo y tenía 'overfitting' (es decir que el modelo sobre-entrenaba y se aprendía los datos de 'memoria'), y mediante la implementación de más capas en la red y el uso de diferentes técnicas vistas en clase se ha conseguido aumentar el 'accuracy' del modelo y reducir su 'overfitting'.

Este jupyter notebook contiene:

- Una breve descripción teorica de que es una Red Neuronal Convolucional
- La arquitectura inicial de la red proporcionada por el profesor
- Una batería de pruebas donde se explican brevemente que mejoras se han implementado y que resultados se han obtenido
- La arquitectura final explicada detalladamente

2 ¿Qué es una una Red Neuronal Convolucional?

Una Red Neuronal Convolucional es una red neuronal artificial especializada en trabajar con imágenes. Para ello esta red aplica la operación de convolución con la cual se obtienen las características, o 'features', de la imagen. Para realizar este cálculo en la red se utilizan capas de convolución. Estas capas contienen unas neuronas especiales que únicamente realizan esta operación. Además estas neuronas tienen la particularidad de que no estan totalmente conectadas con las capas posteriores y que los únicos pesos que aprenden son los valores de sus filtros de convolución.

Después de implementar la convolución se aplica la operación de 'pooling' con la cual se consigue reducir el tamaño de la imagen, reduciendo así la cantidad de parámetros que se deben aprender y por tanto evitando el 'overfitting' y reduciendo el coste computacional. Hay dos tipos de 'poolings': 'max-pooling' y 'average-pooling'. En el 'max-pooling' la imagen se divide en trozos pequeños y de cada trozo se utiliza, para montar la siguiente imagen, únicamente el píxel convolucionado

con mayor valor. En el 'average-pooling' también se realiza la división de la imagen en trozos pequeños, pero en vez de utilizar el valor más alto del trozo se usa la media de todos sus valores. Hoy en día hay una fuerte tendencia en usar la 'max-pooling' ya que con esta capa se guardan las características principales pero no su localización exacta, solo una localización aproximada, y eso hace más robusta la red frente a movimientos geométricos y a cambios de escalas. Estas capas no tienen ningún parámetro que aprender, por eso frecuentemente no se les considera una capa de una red neuronal sino el último paso de la capa de convolución.

Una vez acabada la fase de 'convolución-pooling', la red debe transformar la imagen, la cual es una matriz, en un vector mediante una capa de 'flatten'. Seguidamente se pueden utilizar los datos de estos vectores para alimentar una (o varias) capas de neuronas 'fully connected' (neuronas comunes) las cuales son las que se encargan de procesar o interpretar las características de la imagen. Finalmente, tenemos la capa de salida donde, dependiendo de lo que queramos hacer con la red, podemos tener una neurona de salida o varias. Por ejemplo, si queremos clasificar las imágenes en dos tipos podemos tener una única neurona de salida y que clasifique un tipo de imágenes con un 0 y el otro tipo de imágenes con un 1. Si se tienen más de dos tipos de imágenes se deben poner más neuronas de salida (uno por cada tipo).

Resumiendo, la estructura básica de una Red Neuronal Convolucional es la siguiente: CONV1-POOL1, CONV2-POOL2,… CONVn-POOLn, FLATTEN, FC1, FC2,… FCm, FCoutput

3 Librerías

[2]:
```
from keras.preprocessing import image
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import EarlyStopping
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import random
```

```
Using TensorFlow backend.
```

4 Código inicial

El código inicial propuesto por el profesor es muy intuitivo y esta formado por las partes descritas al inicio de este documento, es decir, presenta la estructura básica de una CNN formada por un conjunto de dos niveles de neuronas CONV-POOL, una capa de FLATTEN y un conjunto de neuronas FULLY CONNECTED formado por una capa de 128 neuronas y una capa de salida de tan sólo una neurona. Además, para compilar la CNN se usa un 'optimizer' tipo 'adam'. Sin embargo, para obtener una mayor información sobre el modelo propuesto se ha decidido visualizar la evolución del error del modelo en cada época. Además, es necesario mencionar que se han modificado los pasos por época a consecuencia del tiempo de computación y porque el ordenador

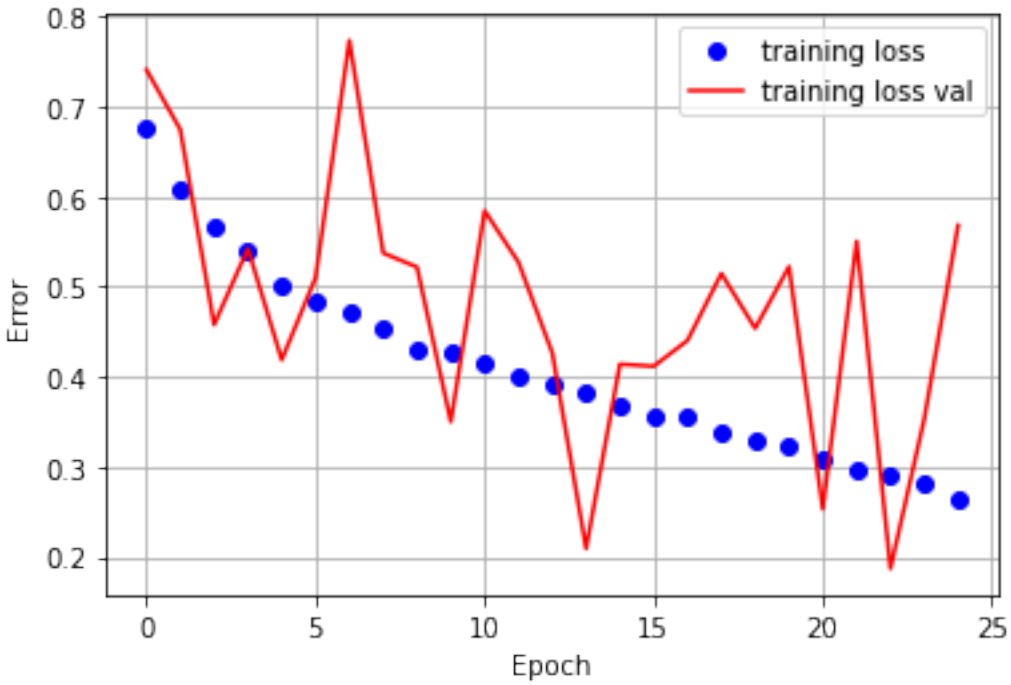
de alguno de los compañeros no soportaba el coste de computación. Así pues, se ha reducido los pasos por época de 8000 a 250.

[17]:
```
tf.random.set_seed(42)
np.random.seed(42)
random.seed(42)

# Initialising the CNN
classifier = Sequential()

# Step 1 - Convolution
classifier.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation =␣
↪'relu'))

# Step 2 - Pooling
classifier.add(MaxPooling2D(pool_size = (2, 2)))

# Adding a second convolutional layer
classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))

# Step 3 - Flattening
classifier.add(Flatten())

# Step 4 - Full connection
classifier.add(Dense(units = 128, activation = 'relu'))
classifier.add(Dense(units = 1, activation = 'sigmoid'))

# Compiling the CNN
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics =␣
↪['accuracy'])

# Part 2 - Fitting the CNN to the images

train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)

test_datagen = ImageDataGenerator(rescale = 1./255)

training_set = train_datagen.flow_from_directory(r'C:
↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
↪- Convolutional Neural Networks (CNN)\dataset\training_set',
                                                 target_size = (64, 64),
                                                 batch_size = 32,
```

```
                                                 class_mode = 'binary')

test_set = test_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\test_set',
                                            target_size = (64, 64),
                                            batch_size = 32,
                                            class_mode = 'binary')

history = classifier.fit_generator(training_set,
                                   steps_per_epoch = 250,
                                   epochs = 25,
                                   validation_data = test_set,
                                   validation_steps = 62)

history_dict=history.history
loss_values = history_dict['loss']
val_loss_values=history_dict['val_loss']
plt.plot(loss_values,'bo',label='training loss')
plt.plot(val_loss_values,'r',label='training loss val')
plt.legend()
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()



# Part 3 - Making new predictions
test_image = image.load_img(r'C:\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\single_prediction\cat_or_dog_1.
 ↪jpg', target_size = (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
training_set.class_indices
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'


print("El modelo considera que en la imagen hay un ", prediction)
print("La imagen contiene de verdad un perro")
```

```
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Epoch 1/25
250/250 [==============================] - 25s 98ms/step - loss: 0.6761 -
accuracy: 0.5745 - val_loss: 0.7410 - val_accuracy: 0.5670
Epoch 2/25
250/250 [==============================] - 23s 91ms/step - loss: 0.6089 -
accuracy: 0.6786 - val_loss: 0.6746 - val_accuracy: 0.6092
Epoch 3/25
250/250 [==============================] - 23s 92ms/step - loss: 0.5669 -
accuracy: 0.7034 - val_loss: 0.4579 - val_accuracy: 0.7099
Epoch 4/25
250/250 [==============================] - 25s 99ms/step - loss: 0.5392 -
accuracy: 0.7319 - val_loss: 0.5424 - val_accuracy: 0.7515
Epoch 5/25
250/250 [==============================] - 25s 99ms/step - loss: 0.5011 -
accuracy: 0.7531 - val_loss: 0.4188 - val_accuracy: 0.7795
Epoch 6/25
250/250 [==============================] - 24s 97ms/step - loss: 0.4829 -
accuracy: 0.7656 - val_loss: 0.5101 - val_accuracy: 0.7779
Epoch 7/25
250/250 [==============================] - 25s 99ms/step - loss: 0.4702 -
accuracy: 0.7740 - val_loss: 0.7735 - val_accuracy: 0.7820
Epoch 8/25
250/250 [==============================] - 23s 92ms/step - loss: 0.4535 -
accuracy: 0.7821 - val_loss: 0.5374 - val_accuracy: 0.7693
Epoch 9/25
250/250 [==============================] - 23s 92ms/step - loss: 0.4302 -
accuracy: 0.7993 - val_loss: 0.5219 - val_accuracy: 0.7840
Epoch 10/25
250/250 [==============================] - 23s 91ms/step - loss: 0.4280 -
accuracy: 0.7993 - val_loss: 0.3503 - val_accuracy: 0.7774
Epoch 11/25
250/250 [==============================] - 23s 92ms/step - loss: 0.4166 -
accuracy: 0.8090 - val_loss: 0.5844 - val_accuracy: 0.7800
Epoch 12/25
250/250 [==============================] - 23s 92ms/step - loss: 0.4006 -
accuracy: 0.8124 - val_loss: 0.5273 - val_accuracy: 0.7810
Epoch 13/25
250/250 [==============================] - 23s 91ms/step - loss: 0.3902 -
accuracy: 0.8185 - val_loss: 0.4267 - val_accuracy: 0.7388
Epoch 14/25
250/250 [==============================] - 23s 93ms/step - loss: 0.3818 -
accuracy: 0.8246 - val_loss: 0.2094 - val_accuracy: 0.8054
Epoch 15/25
250/250 [==============================] - 23s 91ms/step - loss: 0.3667 -
accuracy: 0.8336 - val_loss: 0.4140 - val_accuracy: 0.7942
Epoch 16/25
```

```
250/250 [==============================] - 23s 93ms/step - loss: 0.3573 -
accuracy: 0.8406 - val_loss: 0.4117 - val_accuracy: 0.7942
Epoch 17/25
250/250 [==============================] - 24s 96ms/step - loss: 0.3545 -
accuracy: 0.8424 - val_loss: 0.4404 - val_accuracy: 0.8171
Epoch 18/25
250/250 [==============================] - 24s 96ms/step - loss: 0.3380 -
accuracy: 0.8493 - val_loss: 0.5147 - val_accuracy: 0.7886
Epoch 19/25
250/250 [==============================] - 24s 97ms/step - loss: 0.3304 -
accuracy: 0.8566 - val_loss: 0.4538 - val_accuracy: 0.7967
Epoch 20/25
250/250 [==============================] - 24s 94ms/step - loss: 0.3231 -
accuracy: 0.8587 - val_loss: 0.5223 - val_accuracy: 0.8115
Epoch 21/25
250/250 [==============================] - 23s 94ms/step - loss: 0.3084 -
accuracy: 0.8635 - val_loss: 0.2532 - val_accuracy: 0.7805
Epoch 22/25
250/250 [==============================] - 22s 87ms/step - loss: 0.2972 -
accuracy: 0.8660 - val_loss: 0.5505 - val_accuracy: 0.8191
Epoch 23/25
250/250 [==============================] - 22s 88ms/step - loss: 0.2905 -
accuracy: 0.8737 - val_loss: 0.1871 - val_accuracy: 0.8120
Epoch 24/25
250/250 [==============================] - 22s 88ms/step - loss: 0.2822 -
accuracy: 0.8826 - val_loss: 0.3533 - val_accuracy: 0.8023
Epoch 25/25
250/250 [==============================] - 22s 88ms/step - loss: 0.2648 -
accuracy: 0.8891 - val_loss: 0.5682 - val_accuracy: 0.7957
```




```
El modelo considera que en la imagen hay un  dog
La imagen contiene de verdad un perro
```

Una vez ejecutado el modelo podemos observar que no aporta unos resultados nada satisfactorios. Presenta bastante 'overfitting' (tiene una diferencia de casi el 10% entre el 'accuracy' de 'train' y el 'accuracy' de 'test') y tan sólo es capaz de interpretar correctamente el 79% de las imágenes que nunca ha visto, es decir, que no ha entrenado con ellas.

A pesar de no ejecutar el código inicial sin modificar en este documento, se adjunta una imagen del código ejecutado donde los resultados obtenidos son aun peores porque el modelo presenta un 'overfitting' totalmente desproporcionado:

De esta manera, observando tanto el resultado obtenido del código ejecutado como el código inicial, los objetivos para mejorar el modelo propuestos son claros: * Reducir el overfitting * Mejorar en la medida de lo posible el rendimiento del modelo

Así pues, a continuación se presentan varias pruebas realizadas que han sido de ayuda para obtener un mejor modelo para el reconocimiento de imágenes que contienen perros y gatos.

5 Pruebas realizadas

Se han realizado diversas pruebas, se pueden encontrar algunas más en la carpeta adjunta de 'pruebas'. De entre todas ellas, las más destacables son las tres presentadas a continuación. Cabe añadir que se verá una mejora en cada una de ellas respecto a la anterior.

5.1 Prueba 1

Esta prueba fue una de las primeras en realizar donde se logró algo de mejora. Básicamente, se decidió añadir dos capas intermedias en el nivel de full conection y varios 'Dropouts'. Más contretamente, los cambios realizados fueron: * Capa 'Fully Connected' intermedia de 64 neuronas con una función de activación de tipo 'relu' * Capa 'Fully Connected' intermedia de 32 neuronas con una función de activación de tipo 'relu' * Se hace uso de la función 'Droput' para eliminar el 25% de conexiones entre capas * Se añade un 'Earlystopper' para evitar que el modelo siga entrenando si no obtiene unos resultados con una diferenica notable * Se utiliza el 'optimizer' 'RMSprop' en vez del 'adam'

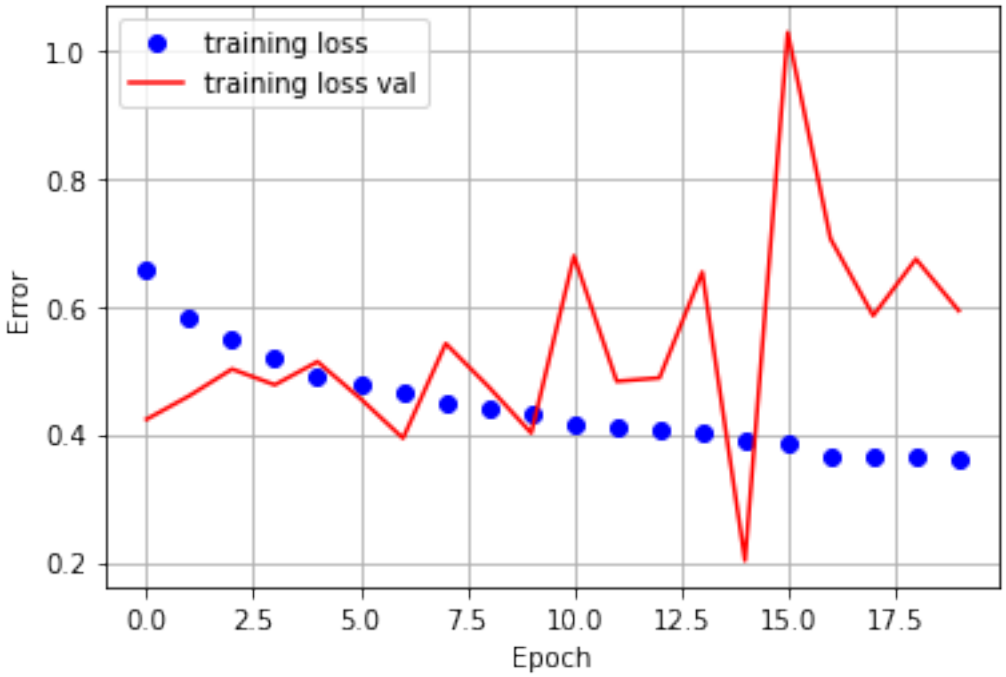
[19]:
```
# Importar las librerías y paquetes

tf.random.set_seed(42)
np.random.seed(42)
random.seed(42)

# Parte 1 - Construir el modelo de CNN

# Inicializar la CNN
classifier = Sequential()

# Paso 1 - Convolución
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3),
                      input_shape = (64, 64, 3), activation = "relu"))

# Paso 2 - Max Pooling
classifier.add(MaxPooling2D(pool_size = (2,2)))

# Una segunda capa de convolución y max pooling
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3), activation = "relu"))

classifier.add(MaxPooling2D(pool_size = (2,2)))

# Este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25)) # NUEVO

# Paso 3 - Flattening
classifier.add(Flatten())

# Paso 4 - Full Connection

classifier.add(Dense(units = 128, activation = "relu"))
classifier.add(Dense(units = 64, activation = "relu")) #NUEVO
classifier.add(Dense(units = 32, activation = "relu")) #NUEVO
```

```
#este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25)) # NUEVO
classifier.add(Dense(units = 1, activation = "sigmoid"))

# Compilar la CNN
classifier.compile(optimizer = "RMSprop", loss = "binary_crossentropy", metrics␣
 ↪= ["accuracy"]) #NUEVO
#classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics =␣
 ↪["accuracy"])

# Parte 2 - Ajustar la CNN a las imágenes para entrenar

train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

training_dataset = train_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\training_set',
                                                 target_size=(64, 64),
                                                 batch_size=32,
                                                 class_mode='binary')

testing_dataset = test_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\test_set',
                                             target_size=(64, 64),
                                             batch_size=32,
                                             class_mode='binary')

#NUEVO
earlystopper = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=100,␣
 ↪verbose=1, mode='auto')

# Ajusta el modelo 20 iteraciones con el 'earlystopper' y lo asigna al historial
history = classifier.fit_generator(training_dataset,
                         steps_per_epoch=350,
                         epochs=20,
                         validation_data=testing_dataset,
```

```
                         validation_steps=200,
                         callbacks = [earlystopper])

# Plots 'history'
history_dict=history.history
loss_values = history_dict['loss']
val_loss_values=history_dict['val_loss']
plt.plot(loss_values,'bo',label='training loss')
plt.plot(val_loss_values,'r',label='training loss val')
plt.legend()
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()


# Parte 3 - Cómo hacer nuevas predicciones
test_image = image.load_img(r'C:\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\single_prediction\cat_or_dog_1.
 ↪jpg', target_size = (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
training_dataset.class_indices
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'

print("El modelo considera que en la imagen hay un ", prediction)
print("La imagen contiene de verdad un perro")
```

```
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Epoch 1/20
350/350 [==============================] - 41s 118ms/step - loss: 0.6556 -
accuracy: 0.6089 - val_loss: 0.4233 - val_accuracy: 0.7168
Epoch 2/20
350/350 [==============================] - 40s 116ms/step - loss: 0.5835 -
accuracy: 0.6981 - val_loss: 0.4600 - val_accuracy: 0.7465
Epoch 3/20
350/350 [==============================] - 37s 105ms/step - loss: 0.5471 -
accuracy: 0.7229 - val_loss: 0.5017 - val_accuracy: 0.7618
Epoch 4/20
350/350 [==============================] - 38s 109ms/step - loss: 0.5198 -
accuracy: 0.7466 - val_loss: 0.4771 - val_accuracy: 0.7646
```

```
Epoch 5/20
350/350 [==============================] - 39s 111ms/step - loss: 0.4894 -
accuracy: 0.7645 - val_loss: 0.5133 - val_accuracy: 0.7511
Epoch 6/20
350/350 [==============================] - 37s 107ms/step - loss: 0.4795 -
accuracy: 0.7767 - val_loss: 0.4559 - val_accuracy: 0.7726
Epoch 7/20
350/350 [==============================] - 37s 106ms/step - loss: 0.4668 -
accuracy: 0.7846 - val_loss: 0.3937 - val_accuracy: 0.7629
Epoch 8/20
350/350 [==============================] - 37s 107ms/step - loss: 0.4495 -
accuracy: 0.7889 - val_loss: 0.5416 - val_accuracy: 0.7901
Epoch 9/20
350/350 [==============================] - 37s 105ms/step - loss: 0.4407 -
accuracy: 0.8014 - val_loss: 0.4743 - val_accuracy: 0.7968
Epoch 10/20
350/350 [==============================] - 37s 105ms/step - loss: 0.4305 -
accuracy: 0.8065 - val_loss: 0.4019 - val_accuracy: 0.7914
Epoch 11/20
350/350 [==============================] - 37s 106ms/step - loss: 0.4167 -
accuracy: 0.8143 - val_loss: 0.6783 - val_accuracy: 0.7487
Epoch 12/20
350/350 [==============================] - 37s 106ms/step - loss: 0.4113 -
accuracy: 0.8111 - val_loss: 0.4827 - val_accuracy: 0.7693
Epoch 13/20
350/350 [==============================] - 36s 104ms/step - loss: 0.4077 -
accuracy: 0.8178 - val_loss: 0.4880 - val_accuracy: 0.7979
Epoch 14/20
350/350 [==============================] - 37s 105ms/step - loss: 0.4020 -
accuracy: 0.8238 - val_loss: 0.6532 - val_accuracy: 0.7993
Epoch 15/20
350/350 [==============================] - 37s 106ms/step - loss: 0.3919 -
accuracy: 0.8268 - val_loss: 0.2028 - val_accuracy: 0.8026
Epoch 16/20
350/350 [==============================] - 40s 113ms/step - loss: 0.3846 -
accuracy: 0.8314 - val_loss: 1.0274 - val_accuracy: 0.6541
Epoch 17/20
350/350 [==============================] - 37s 107ms/step - loss: 0.3647 -
accuracy: 0.8402 - val_loss: 0.7053 - val_accuracy: 0.8147
Epoch 18/20
350/350 [==============================] - 37s 106ms/step - loss: 0.3670 -
accuracy: 0.8402 - val_loss: 0.5852 - val_accuracy: 0.7983
Epoch 19/20
350/350 [==============================] - 37s 106ms/step - loss: 0.3669 -
accuracy: 0.8406 - val_loss: 0.6736 - val_accuracy: 0.8215
Epoch 20/20
350/350 [==============================] - 38s 108ms/step - loss: 0.3622 -
accuracy: 0.8490 - val_loss: 0.5930 - val_accuracy: 0.7780
```




```
El modelo considera que en la imagen hay un  dog
La imagen contiene de verdad un perro
```

Como se puede observar, los resultados obtenidos son ligeramente mejores a los obtenidos en el código inicial. Gracias a los 'Dropout' se reduce de forma considerable el overfitting del modelo (aunque todavía es notable). Sin embargo, se reduce ligeramente la clasificación del modelo gracias a las capas intermedias añadidas. Además, cabe añadir que el 'Earlystopper' no se activa por la gran diferencia entre los resultados de cada 'epoch'.

5.2 Prueba 2

En esta prueba tan sólo ha habido un cambio respeto a la anterior: * Se ha eliminado la capa intermedia de 64 neruonas

De esta manera, permanece la capa intermedia de 32 neuronas.

[18]:
```
# Importar las librerías y paquetes

tf.random.set_seed(42)
np.random.seed(42)
random.seed(42)

# Parte 1 - Construir el modelo de CNN

# Inicializar la CNN
```

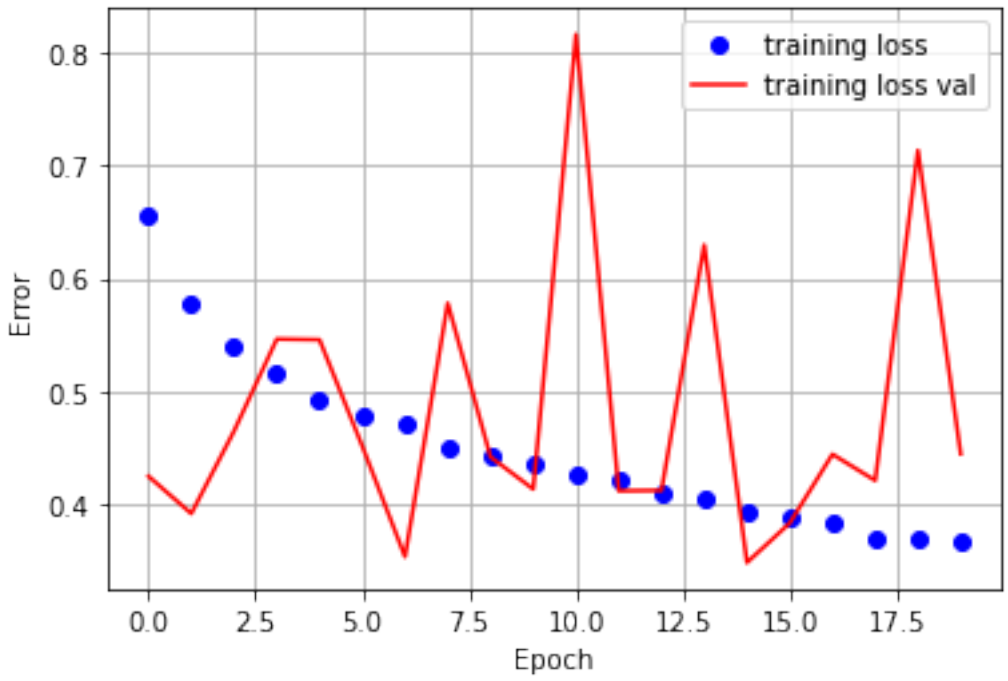
```
classifier = Sequential()

# Paso 1 - Convolución
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3),
                      input_shape = (64, 64, 3), activation = "relu"))

# Paso 2 - Max Pooling
classifier.add(MaxPooling2D(pool_size = (2,2)))

# Una segunda capa de convolución y max pooling
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3), activation = "relu"))

classifier.add(MaxPooling2D(pool_size = (2,2)))


#este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25)) # NUEVO

# Paso 3 - Flattening
classifier.add(Flatten())

# Paso 4 - Full Connection

classifier.add(Dense(units = 128, activation = "relu"))
#classifier.add(Dense(units = 64, activation = "relu")) #NUEVO
classifier.add(Dense(units = 32, activation = "relu")) #NUEVO

# Este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25)) # NUEVO
classifier.add(Dense(units = 1, activation = "sigmoid"))

# Compilar la CNN
classifier.compile(optimizer = "RMSprop", loss = "binary_crossentropy", metrics␣
 ↪= ["accuracy"]) #NUEVO
#classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics =␣
 ↪["accuracy"])

# Parte 2 - Ajustar la CNN a las imágenes para entrenar

train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)
```

```
test_datagen = ImageDataGenerator(rescale=1./255)

training_dataset = train_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\training_set',
                                                 target_size=(64, 64),
                                                 batch_size=32,
                                                 class_mode='binary')

testing_dataset = test_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\test_set',
                                             target_size=(64, 64),
                                             batch_size=32,
                                             class_mode='binary')

#NUEVO
earlystopper = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=100,␣
 ↪verbose=1, mode='auto')

# Ajusta el modelo a 20 iteraciones con el 'earlystopper' y lo asigna al␣
 ↪historial
history = classifier.fit_generator(training_dataset,
                         steps_per_epoch=350,
                         epochs=20,
                         validation_data=testing_dataset,
                         validation_steps=200,
                         callbacks = [earlystopper])

# Plots 'history'
history_dict=history.history
loss_values = history_dict['loss']
val_loss_values=history_dict['val_loss']
plt.plot(loss_values,'bo',label='training loss')
plt.plot(val_loss_values,'r',label='training loss val')
plt.legend()
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()


# Parte 3 - Cómo hacer nuevas predicciones
```

```
test_image = image.load_img(r'C:\Users\munta\OneDrive\Escritorio\Aprendizaje␣
↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
↪- Convolutional Neural Networks (CNN)\dataset\single_prediction\cat_or_dog_1.
↪jpg', target_size = (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
training_dataset.class_indices
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'

print("El modelo considera que en la imagen hay un ", prediction)
print("La imagen contiene de verdad un perro")
```

```
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Epoch 1/20
350/350 [==============================] - 41s 117ms/step - loss: 0.6560 -
accuracy: 0.6156 - val_loss: 0.4244 - val_accuracy: 0.7177
Epoch 2/20
350/350 [==============================] - 40s 114ms/step - loss: 0.5786 -
accuracy: 0.6980 - val_loss: 0.3914 - val_accuracy: 0.7631
Epoch 3/20
350/350 [==============================] - 40s 115ms/step - loss: 0.5387 -
accuracy: 0.7291 - val_loss: 0.4646 - val_accuracy: 0.7561
Epoch 4/20
350/350 [==============================] - 39s 112ms/step - loss: 0.5156 -
accuracy: 0.7475 - val_loss: 0.5463 - val_accuracy: 0.7774
Epoch 5/20
350/350 [==============================] - 39s 111ms/step - loss: 0.4919 -
accuracy: 0.7640 - val_loss: 0.5457 - val_accuracy: 0.7393
Epoch 6/20
350/350 [==============================] - 41s 116ms/step - loss: 0.4792 -
accuracy: 0.7713 - val_loss: 0.4513 - val_accuracy: 0.7718
Epoch 7/20
350/350 [==============================] - 46s 130ms/step - loss: 0.4702 -
accuracy: 0.7809 - val_loss: 0.3531 - val_accuracy: 0.7744
Epoch 8/20
350/350 [==============================] - 40s 115ms/step - loss: 0.4499 -
accuracy: 0.7928 - val_loss: 0.5781 - val_accuracy: 0.7516
Epoch 9/20
350/350 [==============================] - 39s 113ms/step - loss: 0.4436 -
accuracy: 0.7940 - val_loss: 0.4417 - val_accuracy: 0.7742
Epoch 10/20
350/350 [==============================] - 39s 112ms/step - loss: 0.4350 -
```

```
accuracy: 0.8041 - val_loss: 0.4131 - val_accuracy: 0.7791
Epoch 11/20
350/350 [==============================] - 39s 113ms/step - loss: 0.4253 -
accuracy: 0.8034 - val_loss: 0.8168 - val_accuracy: 0.8018
Epoch 12/20
350/350 [==============================] - 39s 112ms/step - loss: 0.4215 -
accuracy: 0.8075 - val_loss: 0.4116 - val_accuracy: 0.8051
Epoch 13/20
350/350 [==============================] - 38s 110ms/step - loss: 0.4085 -
accuracy: 0.8163 - val_loss: 0.4119 - val_accuracy: 0.7974
Epoch 14/20
350/350 [==============================] - 38s 108ms/step - loss: 0.4047 -
accuracy: 0.8191 - val_loss: 0.6298 - val_accuracy: 0.8152
Epoch 15/20
350/350 [==============================] - 39s 111ms/step - loss: 0.3919 -
accuracy: 0.8253 - val_loss: 0.3479 - val_accuracy: 0.8166
Epoch 16/20
350/350 [==============================] - 39s 110ms/step - loss: 0.3876 -
accuracy: 0.8329 - val_loss: 0.3829 - val_accuracy: 0.8179
Epoch 17/20
350/350 [==============================] - 38s 109ms/step - loss: 0.3836 -
accuracy: 0.8316 - val_loss: 0.4440 - val_accuracy: 0.8059
Epoch 18/20
350/350 [==============================] - 40s 114ms/step - loss: 0.3684 -
accuracy: 0.8419 - val_loss: 0.4207 - val_accuracy: 0.8193
Epoch 19/20
350/350 [==============================] - 39s 111ms/step - loss: 0.3689 -
accuracy: 0.8394 - val_loss: 0.7139 - val_accuracy: 0.8156
Epoch 20/20
350/350 [==============================] - 39s 110ms/step - loss: 0.3673 -
accuracy: 0.8413 - val_loss: 0.4444 - val_accuracy: 0.8179
```




```
El modelo considera que en la imagen hay un  dog
La imagen contiene de verdad un perro
```

Una vez ejecutado el modelo, podemos observar que está vez el modelo si mejora la clasificación de imágenes de testeo y, por ende, se reduce el 'overfitting' del modelo. Además, gracias al plot obtenido podemos observar que, a pesar de los sobresaltos del error en el testeo, éste parece que se va reduciendo a medida que augmentan las épocas.

5.3 Prueba 3

Esta prueba respeto a la anterior presenta dos cambios muy significativos: * Se decide doblar el numero de numero de neuronas de convolución y de polling. De dos niveles de CONV-POOL pasamos a tener 4 niveles * Gracias al plot anterior, se decide aumentar considerablemente el número de épocas. De 20 épocas a 60

[3]:
```
tf.random.set_seed(42)
np.random.seed(42)
random.seed(42)


# Parte 1 - Construir el modelo de CNN

# Inicializar la CNN
classifier = Sequential()
```

```
# Paso 1 - Convolución
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3),
                      input_shape = (64, 64, 3), activation = "relu"))

# Paso 2 - Max Pooling
classifier.add(MaxPooling2D(pool_size = (2,2)))

# Una segunda capa de convolución y max pooling
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3), activation = "relu"))

classifier.add(MaxPooling2D(pool_size = (2,2)))


#este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25 )) # NUEVO

# Paso 3 - Flattening
#classifier.add(Flatten())

#########################################################################

classifier.add(Conv2D(filters = 32,kernel_size = (3, 3),
                      input_shape = (64, 64, 3), activation = "relu"))

# Paso 2 - Max Pooling
classifier.add(MaxPooling2D(pool_size = (2,2)))

# Una segunda capa de convolución y max pooling
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3), activation = "relu"))

classifier.add(MaxPooling2D(pool_size = (2,2)))


# Este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25 )) # NUEVO

# Paso 3 - Flattening
classifier.add(Flatten())
#############################################################################################

# Paso 4 - Full Connection

classifier.add(Dense(units = 128, activation = "relu"))
#classifier.add(Dense(units = 64, activation = "relu")) #NUEVO
classifier.add(Dense(units = 32, activation = "relu")) #NUEVO
```

```
#este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25)) # NUEVO
classifier.add(Dense(units = 1, activation = "sigmoid"))

# Compilar la CNN
classifier.compile(optimizer = "RMSprop", loss = "binary_crossentropy", metrics␣
 ↪= ["accuracy"]) #NUEVO
#classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics =␣
 ↪["accuracy"])

# Parte 2 - Ajustar la CNN a las imágenes para entrenar
train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

training_dataset = train_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\training_set',
                                                 target_size=(64, 64),
                                                 batch_size=32,
                                                 class_mode='binary')

testing_dataset = test_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\test_set',
                                            target_size=(64, 64),
                                            batch_size=32,
                                            class_mode='binary')

#NUEVO
earlystopper = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=100,␣
 ↪verbose=1, mode='auto')

# Ajusta el modelo a 60 iteraciones con el 'earlystopper' y lo asigna al␣
 ↪historial
history = classifier.fit_generator(training_dataset,
                         steps_per_epoch=350,
                         epochs=60,
                         validation_data=testing_dataset,
```

```
                         validation_steps=200,
                         callbacks = [earlystopper])

# Plots 'history'
history_dict=history.history
loss_values = history_dict['loss']
val_loss_values=history_dict['val_loss']
plt.plot(loss_values,'bo',label='training loss')
plt.plot(val_loss_values,'r',label='training loss val')
plt.legend()
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()


# Parte 3 - Cómo hacer nuevas predicciones
test_image = image.load_img(r'C:\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\single_prediction\cat_or_dog_1.
 ↪jpg', target_size = (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
training_dataset.class_indices
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'

print("El modelo considera que en la imagen hay un ", prediction)
print("La imagen contiene de verdad un perro")
```

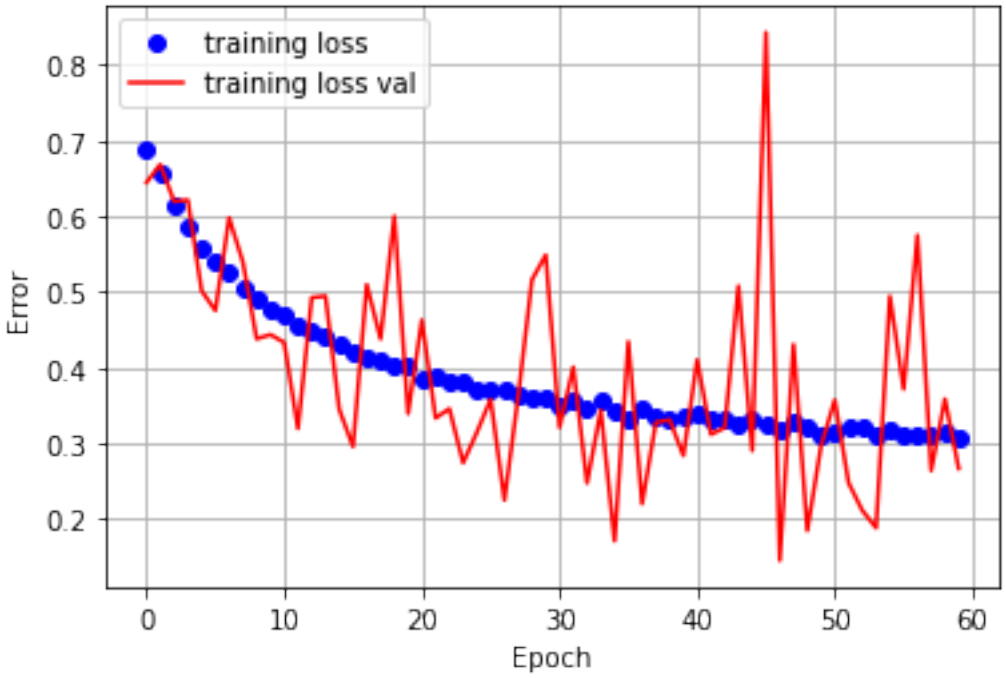
```
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Epoch 1/60
350/350 [==============================] - 73s 209ms/step - loss: 0.6889 -
accuracy: 0.5408 - val_loss: 0.6446 - val_accuracy: 0.6064
Epoch 2/60
350/350 [==============================] - 34s 98ms/step - loss: 0.6552 -
accuracy: 0.6143 - val_loss: 0.6683 - val_accuracy: 0.6126
Epoch 3/60
350/350 [==============================] - 35s 99ms/step - loss: 0.6154 -
accuracy: 0.6645 - val_loss: 0.6195 - val_accuracy: 0.6023
Epoch 4/60
350/350 [==============================] - 34s 98ms/step - loss: 0.5867 -
accuracy: 0.6941 - val_loss: 0.6214 - val_accuracy: 0.7294
```

```
Epoch 5/60
350/350 [==============================] - 34s 98ms/step - loss: 0.5581 -
accuracy: 0.7140 - val_loss: 0.5016 - val_accuracy: 0.7042
Epoch 6/60
350/350 [==============================] - 35s 99ms/step - loss: 0.5391 -
accuracy: 0.7337 - val_loss: 0.4755 - val_accuracy: 0.7663
Epoch 7/60
350/350 [==============================] - 35s 101ms/step - loss: 0.5257 -
accuracy: 0.7425 - val_loss: 0.5979 - val_accuracy: 0.6694
Epoch 8/60
350/350 [==============================] - 35s 99ms/step - loss: 0.5041 -
accuracy: 0.7553 - val_loss: 0.5395 - val_accuracy: 0.7541
Epoch 9/60
350/350 [==============================] - 35s 100ms/step - loss: 0.4916 -
accuracy: 0.7667 - val_loss: 0.4381 - val_accuracy: 0.7846
Epoch 10/60
350/350 [==============================] - 35s 100ms/step - loss: 0.4767 -
accuracy: 0.7799 - val_loss: 0.4437 - val_accuracy: 0.7639
Epoch 11/60
350/350 [==============================] - 35s 101ms/step - loss: 0.4714 -
accuracy: 0.7835 - val_loss: 0.4333 - val_accuracy: 0.8051
Epoch 12/60
350/350 [==============================] - 37s 105ms/step - loss: 0.4559 -
accuracy: 0.7941 - val_loss: 0.3197 - val_accuracy: 0.8164
Epoch 13/60
350/350 [==============================] - 35s 101ms/step - loss: 0.4496 -
accuracy: 0.7877 - val_loss: 0.4926 - val_accuracy: 0.8023
Epoch 14/60
350/350 [==============================] - 35s 101ms/step - loss: 0.4404 -
accuracy: 0.7960 - val_loss: 0.4946 - val_accuracy: 0.8391
Epoch 15/60
350/350 [==============================] - 35s 100ms/step - loss: 0.4315 -
accuracy: 0.8062 - val_loss: 0.3450 - val_accuracy: 0.8330
Epoch 16/60
350/350 [==============================] - 35s 100ms/step - loss: 0.4207 -
accuracy: 0.8065 - val_loss: 0.2959 - val_accuracy: 0.8424
Epoch 17/60
350/350 [==============================] - 35s 100ms/step - loss: 0.4143 -
accuracy: 0.8125 - val_loss: 0.5098 - val_accuracy: 0.8334
Epoch 18/60
350/350 [==============================] - 36s 102ms/step - loss: 0.4114 -
accuracy: 0.8151 - val_loss: 0.4380 - val_accuracy: 0.8104
Epoch 19/60
350/350 [==============================] - 35s 99ms/step - loss: 0.4016 -
accuracy: 0.8244 - val_loss: 0.6001 - val_accuracy: 0.8218
Epoch 20/60
350/350 [==============================] - 35s 101ms/step - loss: 0.4026 -
accuracy: 0.8212 - val_loss: 0.3395 - val_accuracy: 0.8441
```

```
Epoch 21/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3851 -
accuracy: 0.8251 - val_loss: 0.4630 - val_accuracy: 0.8498
Epoch 22/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3878 -
accuracy: 0.8260 - val_loss: 0.3335 - val_accuracy: 0.8531
Epoch 23/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3811 -
accuracy: 0.8309 - val_loss: 0.3459 - val_accuracy: 0.8486
Epoch 24/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3802 -
accuracy: 0.8304 - val_loss: 0.2745 - val_accuracy: 0.8219
Epoch 25/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3709 -
accuracy: 0.8365 - val_loss: 0.3149 - val_accuracy: 0.8185
Epoch 26/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3718 -
accuracy: 0.8391 - val_loss: 0.3585 - val_accuracy: 0.8547
Epoch 27/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3699 -
accuracy: 0.8357 - val_loss: 0.2252 - val_accuracy: 0.8537- los
Epoch 28/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3652 -
accuracy: 0.8371 - val_loss: 0.3623 - val_accuracy: 0.8553
Epoch 29/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3599 -
accuracy: 0.8430 - val_loss: 0.5162 - val_accuracy: 0.8408
Epoch 30/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3610 -
accuracy: 0.8418 - val_loss: 0.5488 - val_accuracy: 0.8434
Epoch 31/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3503 -
accuracy: 0.8496 - val_loss: 0.3212 - val_accuracy: 0.8517
Epoch 32/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3569 -
accuracy: 0.8471 - val_loss: 0.4009 - val_accuracy: 0.8474
Epoch 33/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3463 -
accuracy: 0.8476 - val_loss: 0.2478 - val_accuracy: 0.8698
Epoch 34/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3579 -
accuracy: 0.8434 - val_loss: 0.3442 - val_accuracy: 0.8155
Epoch 35/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3417 -
accuracy: 0.8513 - val_loss: 0.1713 - val_accuracy: 0.8640
Epoch 36/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3322 -
accuracy: 0.8571 - val_loss: 0.4345 - val_accuracy: 0.8293
```

```
Epoch 37/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3468 -
accuracy: 0.8479 - val_loss: 0.2204 - val_accuracy: 0.8700
Epoch 38/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3348 -
accuracy: 0.8537 - val_loss: 0.3279 - val_accuracy: 0.8481
Epoch 39/60
350/350 [==============================] - 36s 102ms/step - loss: 0.3326 -
accuracy: 0.8570 - val_loss: 0.3312 - val_accuracy: 0.8434
Epoch 40/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3339 -
accuracy: 0.8523 - val_loss: 0.2841 - val_accuracy: 0.8627
Epoch 41/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3375 -
accuracy: 0.8579 - val_loss: 0.4104 - val_accuracy: 0.8545
Epoch 42/60
350/350 [==============================] - 34s 98ms/step - loss: 0.3328 -
accuracy: 0.8621 - val_loss: 0.3120 - val_accuracy: 0.8704
Epoch 43/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3304 -
accuracy: 0.8579 - val_loss: 0.3208 - val_accuracy: 0.8626
Epoch 44/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3255 -
accuracy: 0.8566 - val_loss: 0.5079 - val_accuracy: 0.8709
Epoch 45/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3305 -
accuracy: 0.8587 - val_loss: 0.2905 - val_accuracy: 0.8487
Epoch 46/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3260 -
accuracy: 0.8598 - val_loss: 0.8427 - val_accuracy: 0.8469
Epoch 47/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3193 -
accuracy: 0.8671 - val_loss: 0.1455 - val_accuracy: 0.8703
Epoch 48/60
350/350 [==============================] - 34s 98ms/step - loss: 0.3290 -
accuracy: 0.8609 - val_loss: 0.4308 - val_accuracy: 0.8616
Epoch 49/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3206 -
accuracy: 0.8637 - val_loss: 0.1849 - val_accuracy: 0.8670
Epoch 50/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3116 -
accuracy: 0.8702 - val_loss: 0.2970 - val_accuracy: 0.8627
Epoch 51/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3158 -
accuracy: 0.8642 - val_loss: 0.3575 - val_accuracy: 0.8736
Epoch 52/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3216 -
accuracy: 0.8649 - val_loss: 0.2480 - val_accuracy: 0.8690
```

```
Epoch 53/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3227 -
accuracy: 0.8636 - val_loss: 0.2115 - val_accuracy: 0.8730
Epoch 54/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3119 -
accuracy: 0.8680 - val_loss: 0.1886 - val_accuracy: 0.8430
Epoch 55/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3182 -
accuracy: 0.8649 - val_loss: 0.4944 - val_accuracy: 0.8301
Epoch 56/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3110 -
accuracy: 0.8646 - val_loss: 0.3718 - val_accuracy: 0.8456
Epoch 57/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3103 -
accuracy: 0.8691 - val_loss: 0.5749 - val_accuracy: 0.8405
Epoch 58/60
350/350 [==============================] - 35s 100ms/step - loss: 0.3111 -
accuracy: 0.8692 - val_loss: 0.2642 - val_accuracy: 0.8621
Epoch 59/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3141 -
accuracy: 0.8711 - val_loss: 0.3588 - val_accuracy: 0.8602
Epoch 60/60
350/350 [==============================] - 35s 99ms/step - loss: 0.3089 -
accuracy: 0.8696 - val_loss: 0.2670 - val_accuracy: 0.8840
```



```
El modelo considera que en la imagen hay un  dog
La imagen contiene de verdad un perro
```

Una vez ejecutado el nuevo modelo, observamos que los resultados obtenidos són mucho mejores. Se ha reducido por completo el 'overfitting' y se ha mejorado considerablemente la clasificación. En este punto, el modelo es capaz de clasificar correctamente el 88,40% de las imágenes. Sin embargo, observando el plot de la evolución de los errores parece que es posible mejorar aún más el modelo. Con esto en mente se llega a la implementación de la CNN final.

6 CNN final

Después de las diversas pruebas realizadas se ha logrado obtener un mejor modelo para la clasificación de imágenes de perros y gatos. Como se ha mencionado anteriormente, con las pruebas realizadas se ha logrado obtener un modelo con neuronas convolucionales que ya no presentase 'overfitting'. De esta manera, el objetivo principal ha sido mejorar el modelo.

En las siguientes celdas se describirá el mejor modelo que se ha obtenido teniendo en cuenta la 'accuracy' obtenida y la evolución del error del modelo en cada época. Además, como se podrá observar en la siguientes celdas, se describirá de manera más detallada los cambios realizados respecto al código inicial presentado en el curso Udemy, cambios que han sido mencionados en las pruebas anteriores.

6.1 Implementación de las capas convolucionales

[5]:
```
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

# Inicializar la CNN
classifier = Sequential()

# Paso 1 - Convolución
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3),
                      input_shape = (64, 64, 3), activation = "relu"))

# Paso 2 - Max Pooling
classifier.add(MaxPooling2D(pool_size = (2,2)))

# Una segunda capa de convolución y max pooling
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3), activation = "relu"))

classifier.add(MaxPooling2D(pool_size = (2,2)))


#este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
```

```
classifier.add(Dropout(0.25 )) # NUEVO

#################### Duplicación de las capas convolucionales␣
 ↪####################

# Paso 1 - Convolución
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3),
                      input_shape = (64, 64, 3), activation = "relu"))

# Paso 2 - Max Pooling
classifier.add(MaxPooling2D(pool_size = (2,2)))

# Una segunda capa de convolución y max pooling
classifier.add(Conv2D(filters = 32,kernel_size = (3, 3), activation = "relu"))

classifier.add(MaxPooling2D(pool_size = (2,2)))

#############################################################################

#este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25 )) # NUEVO

# Paso 3 - Flattening
classifier.add(Flatten())
```

Esta parte del código es donde se encuentran la mayoría de los cambios, ya que en esta celda se lleva a cabo la implementación de las capas convolucionales del modelo. Como se ha descrito anteriormente, toda red con neuronas convolucionales presentan una estructura inicial muy parecida, la cual podemos identificar con los comentarios 'Paso 1', 'Paso 2' y 'Paso 3'. De esta manera, los cambios realizados son:

- Para reducir el overfitting se ha usado la función 'Dropout', que elimina el % de conexiones entre capas. En nuestro caso, se eliminan el 25% de conexiones entre varias capas convolucionales.
- Para mejorar el modelo, se ha decidido añadir más capas convolucionales, es decir, se han añadido nuevas capas de convolución y de polling (siguiendo el consejo realizado por el doctor ***Andrew Ng***). En nuestro caso, se ha decidido doblar el número de capas del código inicial (4 capas convolucionales y 4 capas de polling).

6.2 Full connection

[6]:
```
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

# Paso 4 - Full Connection
```

```
classifier.add(Dense(units = 128, activation = "relu"))
#classifier.add(Dense(units = 64, activation = "relu")) #NUEVO
classifier.add(Dense(units = 32, activation = "relu")) #NUEVO

#este dropout desactiva el 25% de las conexiones entre las neuronas, lo cual␣
 ↪mejora los resultados
classifier.add(Dropout(0.25)) # NUEVO

classifier.add(Dense(units = 1, activation = "sigmoid"))
```

Una vez establecidas las capas convolucionales del modelo, se lleva a cabo el 'Paso 4' llamado 'Full Connection'. En este momento, se implementan las capas encargadas de interpretar las características de las imágenes para así poder clasificarlas. Para ello se implementan capas de neuronas comunes que son alimentadas por los datos de las neuronas convolucionales.

En un inicio, el código inicial tan sólo presentaba dos capas. Una capa de 128 neuronas y una capa de salida de 1 neurona. Gracias a las pruebas realizadas (como la linia comentada de 64 neuronas), se ha comprobado que es necesario añadir más capas intermedias para que el modelo pueda mejorar la clasificación. Despues de varias pruebas, la mejor combinación de capas es la siguiente:

- Una primera capa de 128 neuronas con una función de activación tipo 'relu'
- Una segunda capa de 32 neuronas con una función de activación tipo 'relu'
- Una capa de salida de 1 neurona con una función de activación tipo 'sigmoid'. La función de activación sigmoid es ideal para casos binarios, es decir, perro o gato.

Además, es necesario mencionar que se ha usado una vez más la función 'Dropout' de un 25% entre las dos últimas capas para reducir aún más el overfitting que puede sufrir el modelo.

6.3 Compilar CNN

[7]:
```
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

# Compilar la CNN
classifier.compile(optimizer = "RMSprop", loss = "binary_crossentropy", metrics␣
 ↪= ["accuracy"]) #NUEVO
#classifier.compile(optimizer = "adam", loss = "binary_crossentropy", metrics =␣
 ↪["accuracy"])
```

Una vez implementado la arquitectura de la red neuronal de nuestro modelo, llega el momento de compilar la red. Para ello se ha decido usar un 'optimizer' distinto. En el código inicial, se recurre al optimizador 'adam' (el más habitual en las redes neuronales). Sin embargo, después de una búsqueda se ha decidido usar el optimizer 'RMSprop' obteniendo mejores resultados. Es necesario mencionar que durante la búsqueda de alternativas para la mejora del código se observó el uso del optimizer 'RMSprop' para la identificación de imágenes que representaban dígitos. Esta mejora se debe a que RMSprop es un algoritmo de optimización que aplica el descenso de gradiente

disminuyendo radicalmente el "learning rate" en el proceso. En el caso de del "adam", según nuestras pruebas, este proceso requiere de más épocas para conseguir la convergencia, lo que introduce "overfitting". Con un "learning rate" que decae más rápidamente es más fácil conseguir que el error cometido disminuya hasta oscilar muy poco sin que se haya introducido "overfitting".

6.4 Ajuste de la CNN para el entrenamiento

[8]:
```
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

# Parte 2 - Ajustar la CNN a las imágenes para entrenar

train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale=1./255)

training_dataset = train_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\training_set',
                                                 target_size=(64, 64),
                                                 batch_size=32,
                                                 class_mode='binary')

testing_dataset = test_datagen.flow_from_directory(r'C:
 ↪\Users\munta\OneDrive\Escritorio\Aprendizaje␣
 ↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
 ↪- Convolutional Neural Networks (CNN)\dataset\test_set',
                                             target_size=(64, 64),
                                             batch_size=32,
                                             class_mode='binary')

#NUEVO
earlystopper = EarlyStopping(monitor='val_loss', min_delta=0.01, patience=100,␣
 ↪verbose=1, mode='auto')

# Ajusta el modelo a más de 1000 iteraciones con el 'earlystopper' y lo asigna␣
 ↪al historial
history = classifier.fit_generator(training_dataset,
                         steps_per_epoch=350, #NUEVO
                         epochs=150, #NUEVO
```

```
                        validation_data=testing_dataset,
                        validation_steps=200,
                        callbacks = [earlystopper])
```

```
Found 8000 images belonging to 2 classes.
Found 2000 images belonging to 2 classes.
Epoch 1/150
350/350 [==============================] - 36s 103ms/step - loss: 0.6908 -
accuracy: 0.5290 - val_loss: 0.7028 - val_accuracy: 0.5874
Epoch 2/150
350/350 [==============================] - 35s 99ms/step - loss: 0.6475 -
accuracy: 0.6288 - val_loss: 0.6967 - val_accuracy: 0.6844
Epoch 3/150
350/350 [==============================] - 34s 98ms/step - loss: 0.6029 -
accuracy: 0.6745 - val_loss: 0.5479 - val_accuracy: 0.7117
Epoch 4/150
350/350 [==============================] - 34s 98ms/step - loss: 0.5758 -
accuracy: 0.7055 - val_loss: 0.5460 - val_accuracy: 0.7431
Epoch 5/150
350/350 [==============================] - 35s 99ms/step - loss: 0.5557 -
accuracy: 0.7169 - val_loss: 0.4563 - val_accuracy: 0.7300
Epoch 6/150
350/350 [==============================] - 34s 98ms/step - loss: 0.5216 -
accuracy: 0.7436 - val_loss: 0.6727 - val_accuracy: 0.7404
Epoch 7/150
350/350 [==============================] - 35s 100ms/step - loss: 0.5249 -
accuracy: 0.7401 - val_loss: 0.3278 - val_accuracy: 0.7761
Epoch 8/150
350/350 [==============================] - 36s 103ms/step - loss: 0.5076 -
accuracy: 0.7553 - val_loss: 0.3582 - val_accuracy: 0.7228
Epoch 9/150
350/350 [==============================] - 35s 101ms/step - loss: 0.4903 -
accuracy: 0.7686 - val_loss: 0.4118 - val_accuracy: 0.7990
Epoch 10/150
350/350 [==============================] - 35s 101ms/step - loss: 0.4771 -
accuracy: 0.7763 - val_loss: 0.6165 - val_accuracy: 0.7966
Epoch 11/150
350/350 [==============================] - 35s 100ms/step - loss: 0.4676 -
accuracy: 0.7820 - val_loss: 0.7272 - val_accuracy: 0.8235
Epoch 12/150
350/350 [==============================] - 36s 102ms/step - loss: 0.4594 -
accuracy: 0.7890 - val_loss: 0.3925 - val_accuracy: 0.8074
Epoch 13/150
350/350 [==============================] - 35s 99ms/step - loss: 0.4453 -
accuracy: 0.7912 - val_loss: 0.4158 - val_accuracy: 0.8186
Epoch 14/150
350/350 [==============================] - 35s 99ms/step - loss: 0.4371 -
```

```
accuracy: 0.7981 - val_loss: 0.4996 - val_accuracy: 0.8245
Epoch 15/150
350/350 [==============================] - 35s 100ms/step - loss: 0.4236 -
accuracy: 0.8058 - val_loss: 0.1907 - val_accuracy: 0.8382
Epoch 16/150
350/350 [==============================] - 35s 100ms/step - loss: 0.4188 -
accuracy: 0.8104 - val_loss: 0.5699 - val_accuracy: 0.8076
Epoch 17/150
350/350 [==============================] - 35s 99ms/step - loss: 0.4162 -
accuracy: 0.8112 - val_loss: 0.4386 - val_accuracy: 0.8438
Epoch 18/150
350/350 [==============================] - 35s 100ms/step - loss: 0.4055 -
accuracy: 0.8206 - val_loss: 0.4227 - val_accuracy: 0.8254
Epoch 19/150
350/350 [==============================] - 35s 99ms/step - loss: 0.4054 -
accuracy: 0.8156 - val_loss: 0.5362 - val_accuracy: 0.8468
Epoch 20/150
350/350 [==============================] - 35s 99ms/step - loss: 0.4009 -
accuracy: 0.8243 - val_loss: 0.4386 - val_accuracy: 0.8497
Epoch 21/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3874 -
accuracy: 0.8262 - val_loss: 0.3788 - val_accuracy: 0.8541
Epoch 22/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3841 -
accuracy: 0.8270 - val_loss: 0.2709 - val_accuracy: 0.8602
Epoch 23/150
350/350 [==============================] - 35s 101ms/step - loss: 0.3737 -
accuracy: 0.8358 - val_loss: 0.4053 - val_accuracy: 0.8562
Epoch 24/150
350/350 [==============================] - 35s 101ms/step - loss: 0.3777 -
accuracy: 0.8313 - val_loss: 0.3996 - val_accuracy: 0.8057
Epoch 25/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3774 -
accuracy: 0.8341 - val_loss: 0.2597 - val_accuracy: 0.8451
Epoch 26/150
350/350 [==============================] - 35s 101ms/step - loss: 0.3641 -
accuracy: 0.8414 - val_loss: 0.3611 - val_accuracy: 0.8471
Epoch 27/150
350/350 [==============================] - 36s 102ms/step - loss: 0.3628 -
accuracy: 0.8426 - val_loss: 0.2101 - val_accuracy: 0.8594
Epoch 28/150
350/350 [==============================] - 35s 101ms/step - loss: 0.3602 -
accuracy: 0.8428 - val_loss: 0.4501 - val_accuracy: 0.8624
Epoch 29/150
350/350 [==============================] - 36s 102ms/step - loss: 0.3605 -
accuracy: 0.8447 - val_loss: 0.3572 - val_accuracy: 0.8581
Epoch 30/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3619 -
```

```
accuracy: 0.8407 - val_loss: 0.3383 - val_accuracy: 0.8248
Epoch 31/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3511 -
accuracy: 0.8439 - val_loss: 0.3090 - val_accuracy: 0.8522
Epoch 32/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3509 -
accuracy: 0.8476 - val_loss: 0.5228 - val_accuracy: 0.8700
Epoch 33/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3490 -
accuracy: 0.8444 - val_loss: 0.2515 - val_accuracy: 0.8456
Epoch 34/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3417 -
accuracy: 0.8529 - val_loss: 0.3320 - val_accuracy: 0.8791
Epoch 35/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3469 -
accuracy: 0.8499 - val_loss: 0.2711 - val_accuracy: 0.8535
Epoch 36/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3428 -
accuracy: 0.8528 - val_loss: 0.1442 - val_accuracy: 0.8667
Epoch 37/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3403 -
accuracy: 0.8540 - val_loss: 0.4219 - val_accuracy: 0.8490
Epoch 38/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3357 -
accuracy: 0.8533 - val_loss: 0.3826 - val_accuracy: 0.8456
Epoch 39/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3318 -
accuracy: 0.8541 - val_loss: 0.2687 - val_accuracy: 0.8597 0.3307 - ac
Epoch 40/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3254 -
accuracy: 0.8604 - val_loss: 0.4886 - val_accuracy: 0.8194
Epoch 41/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3330 -
accuracy: 0.8506 - val_loss: 0.5607 - val_accuracy: 0.8603
Epoch 42/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3323 -
accuracy: 0.8560 - val_loss: 0.3418 - val_accuracy: 0.8438
Epoch 43/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3350 -
accuracy: 0.8532 - val_loss: 0.3241 - val_accuracy: 0.8530
Epoch 44/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3235 -
accuracy: 0.8620 - val_loss: 0.4293 - val_accuracy: 0.8767
Epoch 45/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3318 -
accuracy: 0.8595 - val_loss: 0.2386 - val_accuracy: 0.8438
Epoch 46/150
350/350 [==============================] - 35s 101ms/step - loss: 0.3211 -
```

```
accuracy: 0.8668 - val_loss: 0.3775 - val_accuracy: 0.8668
Epoch 47/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3286 -
accuracy: 0.8607 - val_loss: 0.1100 - val_accuracy: 0.8470
Epoch 48/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3200 -
accuracy: 0.8677 - val_loss: 0.3141 - val_accuracy: 0.8678
Epoch 49/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3238 -
accuracy: 0.8598 - val_loss: 0.1988 - val_accuracy: 0.8616
Epoch 50/150
350/350 [==============================] - 35s 101ms/step - loss: 0.3144 -
accuracy: 0.8659 - val_loss: 0.2176 - val_accuracy: 0.8604
Epoch 51/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3199 -
accuracy: 0.8671 - val_loss: 0.3961 - val_accuracy: 0.8777
Epoch 52/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3148 -
accuracy: 0.8684 - val_loss: 0.2953 - val_accuracy: 0.8728
Epoch 53/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3198 -
accuracy: 0.8600 - val_loss: 0.3363 - val_accuracy: 0.8564
Epoch 54/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3137 -
accuracy: 0.8685 - val_loss: 0.4321 - val_accuracy: 0.8733
Epoch 55/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3134 -
accuracy: 0.8642 - val_loss: 0.6081 - val_accuracy: 0.8168
Epoch 56/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3125 -
accuracy: 0.8646 - val_loss: 0.2369 - val_accuracy: 0.8767
Epoch 57/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3124 -
accuracy: 0.8693 - val_loss: 0.3609 - val_accuracy: 0.8712
Epoch 58/150
350/350 [==============================] - 35s 101ms/step - loss: 0.3163 -
accuracy: 0.8692 - val_loss: 0.3380 - val_accuracy: 0.8540
Epoch 59/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3063 -
accuracy: 0.8705 - val_loss: 0.2250 - val_accuracy: 0.8783
Epoch 60/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3153 -
accuracy: 0.8666 - val_loss: 0.2365 - val_accuracy: 0.8662
Epoch 61/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3040 -
accuracy: 0.8738 - val_loss: 0.1912 - val_accuracy: 0.8341
Epoch 62/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3064 -
```

```
accuracy: 0.8739 - val_loss: 0.2075 - val_accuracy: 0.8774
Epoch 63/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3054 -
accuracy: 0.8729 - val_loss: 0.1252 - val_accuracy: 0.8578
Epoch 64/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3070 -
accuracy: 0.8711 - val_loss: 0.3071 - val_accuracy: 0.8561
Epoch 65/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3047 -
accuracy: 0.8734 - val_loss: 0.3115 - val_accuracy: 0.8478
Epoch 66/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3065 -
accuracy: 0.8713 - val_loss: 0.6022 - val_accuracy: 0.8448
Epoch 67/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3116 -
accuracy: 0.8681 - val_loss: 0.2169 - val_accuracy: 0.8468
Epoch 68/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3079 -
accuracy: 0.8719 - val_loss: 0.4120 - val_accuracy: 0.8034
Epoch 69/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3110 -
accuracy: 0.8693 - val_loss: 0.2281 - val_accuracy: 0.8610
Epoch 70/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3057 -
accuracy: 0.8721 - val_loss: 0.3389 - val_accuracy: 0.8731
Epoch 71/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3030 -
accuracy: 0.8706 - val_loss: 0.2701 - val_accuracy: 0.8763
Epoch 72/150
350/350 [==============================] - 35s 101ms/step - loss: 0.2998 -
accuracy: 0.8759 - val_loss: 0.1905 - val_accuracy: 0.8586
Epoch 73/150
350/350 [==============================] - 35s 101ms/step - loss: 0.3007 -
accuracy: 0.8759 - val_loss: 0.2623 - val_accuracy: 0.8819
Epoch 74/150
350/350 [==============================] - 36s 102ms/step - loss: 0.3084 -
accuracy: 0.8696 - val_loss: 0.3366 - val_accuracy: 0.8407
Epoch 75/150
350/350 [==============================] - 35s 100ms/step - loss: 0.2985 -
accuracy: 0.8736 - val_loss: 0.3056 - val_accuracy: 0.8160
Epoch 76/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2984 -
accuracy: 0.8768 - val_loss: 0.2303 - val_accuracy: 0.8723
Epoch 77/150
350/350 [==============================] - 35s 100ms/step - loss: 0.2959 -
accuracy: 0.8783 - val_loss: 0.3106 - val_accuracy: 0.8240
Epoch 78/150
350/350 [==============================] - 35s 100ms/step - loss: 0.2937 -
```

```
accuracy: 0.8793 - val_loss: 0.2580 - val_accuracy: 0.8541
Epoch 79/150
350/350 [==============================] - 36s 101ms/step - loss: 0.2936 -
accuracy: 0.8756 - val_loss: 0.3774 - val_accuracy: 0.8593
Epoch 80/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2950 -
accuracy: 0.8796 - val_loss: 0.4704 - val_accuracy: 0.8668
Epoch 81/150
350/350 [==============================] - 35s 100ms/step - loss: 0.3016 -
accuracy: 0.8762 - val_loss: 0.3498 - val_accuracy: 0.8461
Epoch 82/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3015 -
accuracy: 0.8741 - val_loss: 0.2923 - val_accuracy: 0.8553
Epoch 83/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2896 -
accuracy: 0.8770 - val_loss: 0.2152 - val_accuracy: 0.8794
Epoch 84/150
350/350 [==============================] - 35s 100ms/step - loss: 0.2961 -
accuracy: 0.8813 - val_loss: 0.5509 - val_accuracy: 0.8737
Epoch 85/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2873 -
accuracy: 0.8814 - val_loss: 0.4905 - val_accuracy: 0.8811
Epoch 86/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2921 -
accuracy: 0.8764 - val_loss: 0.2420 - val_accuracy: 0.8660
Epoch 87/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2919 -
accuracy: 0.8799 - val_loss: 0.4299 - val_accuracy: 0.8684
Epoch 88/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2952 -
accuracy: 0.8750 - val_loss: 0.2358 - val_accuracy: 0.8508
Epoch 89/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2913 -
accuracy: 0.8799 - val_loss: 0.3670 - val_accuracy: 0.8720
Epoch 90/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2993 -
accuracy: 0.8764 - val_loss: 0.9480 - val_accuracy: 0.8456
Epoch 91/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2912 -
accuracy: 0.8793 - val_loss: 0.4154 - val_accuracy: 0.8821
Epoch 92/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2851 -
accuracy: 0.8813 - val_loss: 0.7382 - val_accuracy: 0.8146
Epoch 93/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2941 -
accuracy: 0.8820 - val_loss: 0.2562 - val_accuracy: 0.8769
Epoch 94/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2884 -
```

```
accuracy: 0.8839 - val_loss: 0.1637 - val_accuracy: 0.8451
Epoch 95/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2937 -
accuracy: 0.8793 - val_loss: 0.8104 - val_accuracy: 0.8742
Epoch 96/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2948 -
accuracy: 0.8790 - val_loss: 0.2290 - val_accuracy: 0.8676
Epoch 97/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2900 -
accuracy: 0.8826 - val_loss: 0.1930 - val_accuracy: 0.8279
Epoch 98/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2864 -
accuracy: 0.8819 - val_loss: 0.3357 - val_accuracy: 0.8737
Epoch 99/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2900 -
accuracy: 0.8829 - val_loss: 0.2774 - val_accuracy: 0.8630
Epoch 100/150
350/350 [==============================] - 35s 101ms/step - loss: 0.2841 -
accuracy: 0.8819 - val_loss: 0.3596 - val_accuracy: 0.8717
Epoch 101/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2871 -
accuracy: 0.8802 - val_loss: 0.4878 - val_accuracy: 0.8520
Epoch 102/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2826 -
accuracy: 0.8838 - val_loss: 0.3233 - val_accuracy: 0.8841
Epoch 103/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2778 -
accuracy: 0.8792 - val_loss: 0.1539 - val_accuracy: 0.8876
Epoch 104/150
350/350 [==============================] - 35s 100ms/step - loss: 0.2865 -
accuracy: 0.8855 - val_loss: 0.3894 - val_accuracy: 0.8750
Epoch 105/150
350/350 [==============================] - 36s 101ms/step - loss: 0.2856 -
accuracy: 0.8842 - val_loss: 0.2908 - val_accuracy: 0.8741
Epoch 106/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2936 -
accuracy: 0.8820 - val_loss: 0.3698 - val_accuracy: 0.8465
Epoch 107/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2833 -
accuracy: 0.8848 - val_loss: 0.2178 - val_accuracy: 0.8599
Epoch 108/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2887 -
accuracy: 0.8821 - val_loss: 0.2428 - val_accuracy: 0.8722
Epoch 109/150
350/350 [==============================] - 35s 101ms/step - loss: 0.2868 -
accuracy: 0.8839 - val_loss: 0.2557 - val_accuracy: 0.8722
Epoch 110/150
350/350 [==============================] - 35s 101ms/step - loss: 0.2945 -
```

```
accuracy: 0.8809 - val_loss: 0.2510 - val_accuracy: 0.8846
Epoch 111/150
350/350 [==============================] - 35s 100ms/step - loss: 0.2918 -
accuracy: 0.8815 - val_loss: 0.3562 - val_accuracy: 0.8772
Epoch 112/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2873 -
accuracy: 0.8832 - val_loss: 0.2791 - val_accuracy: 0.8764
Epoch 113/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2909 -
accuracy: 0.8839 - val_loss: 0.2995 - val_accuracy: 0.8391
Epoch 114/150
350/350 [==============================] - 34s 97ms/step - loss: 0.3010 -
accuracy: 0.8786 - val_loss: 0.3053 - val_accuracy: 0.8758
Epoch 115/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2903 -
accuracy: 0.8843 - val_loss: 0.4853 - val_accuracy: 0.8521
Epoch 116/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2975 -
accuracy: 0.8823 - val_loss: 0.3734 - val_accuracy: 0.8681
Epoch 117/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2977 -
accuracy: 0.8805 - val_loss: 0.4148 - val_accuracy: 0.8334
Epoch 118/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2914 -
accuracy: 0.8845 - val_loss: 0.1920 - val_accuracy: 0.8767
Epoch 119/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2966 -
accuracy: 0.8836 - val_loss: 0.2654 - val_accuracy: 0.8781
Epoch 120/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2952 -
accuracy: 0.8838 - val_loss: 0.3689 - val_accuracy: 0.8811
Epoch 121/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3002 -
accuracy: 0.8812 - val_loss: 0.2961 - val_accuracy: 0.8545
Epoch 122/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2897 -
accuracy: 0.8839 - val_loss: 0.3476 - val_accuracy: 0.8813
Epoch 123/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3011 -
accuracy: 0.8768 - val_loss: 0.2367 - val_accuracy: 0.8690
Epoch 124/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2833 -
accuracy: 0.8840 - val_loss: 0.3167 - val_accuracy: 0.8977
Epoch 125/150
350/350 [==============================] - 34s 98ms/step - loss: 0.2972 -
accuracy: 0.8825 - val_loss: 0.1848 - val_accuracy: 0.8852
Epoch 126/150
350/350 [==============================] - 35s 99ms/step - loss: 0.2927 -
```

```
accuracy: 0.8821 - val_loss: 0.2102 - val_accuracy: 0.8876
Epoch 127/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3021 -
accuracy: 0.8782 - val_loss: 0.2556 - val_accuracy: 0.8956
Epoch 128/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3012 -
accuracy: 0.8763 - val_loss: 0.7348 - val_accuracy: 0.8871
Epoch 129/150
350/350 [==============================] - 34s 97ms/step - loss: 0.3003 -
accuracy: 0.8776 - val_loss: 0.2041 - val_accuracy: 0.8975
Epoch 130/150
350/350 [==============================] - 34s 97ms/step - loss: 0.3024 -
accuracy: 0.8778 - val_loss: 0.2093 - val_accuracy: 0.8542
Epoch 131/150
350/350 [==============================] - 34s 97ms/step - loss: 0.2997 -
accuracy: 0.8794 - val_loss: 0.2462 - val_accuracy: 0.8885
Epoch 132/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3105 -
accuracy: 0.8819 - val_loss: 0.2624 - val_accuracy: 0.8433
Epoch 133/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3084 -
accuracy: 0.8763 - val_loss: 0.4226 - val_accuracy: 0.8742
Epoch 134/150
350/350 [==============================] - 34s 97ms/step - loss: 0.3104 -
accuracy: 0.8771 - val_loss: 0.2428 - val_accuracy: 0.8833
Epoch 135/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3008 -
accuracy: 0.8772 - val_loss: 0.2622 - val_accuracy: 0.8846
Epoch 136/150
350/350 [==============================] - 34s 97ms/step - loss: 0.3125 -
accuracy: 0.8737 - val_loss: 0.3937 - val_accuracy: 0.8796
Epoch 137/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3229 -
accuracy: 0.8712 - val_loss: 0.1808 - val_accuracy: 0.8561
Epoch 138/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3126 -
accuracy: 0.8768 - val_loss: 0.2777 - val_accuracy: 0.8696
Epoch 139/150
350/350 [==============================] - 34s 96ms/step - loss: 0.3272 -
accuracy: 0.8696 - val_loss: 0.1998 - val_accuracy: 0.8459
Epoch 140/150
350/350 [==============================] - 34s 97ms/step - loss: 0.3239 -
accuracy: 0.8681 - val_loss: 0.2113 - val_accuracy: 0.8347
Epoch 141/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3258 -
accuracy: 0.8688 - val_loss: 0.3206 - val_accuracy: 0.8550
Epoch 142/150
350/350 [==============================] - 34s 97ms/step - loss: 0.3419 -
```

```
accuracy: 0.8720 - val_loss: 0.2467 - val_accuracy: 0.8833
Epoch 143/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3220 -
accuracy: 0.8755 - val_loss: 0.3077 - val_accuracy: 0.8824
Epoch 144/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3281 -
accuracy: 0.8718 - val_loss: 0.3474 - val_accuracy: 0.8824
Epoch 145/150
350/350 [==============================] - 34s 98ms/step - loss: 0.3328 -
accuracy: 0.8671 - val_loss: 0.2527 - val_accuracy: 0.8750
Epoch 146/150
350/350 [==============================] - 34s 97ms/step - loss: 0.3458 -
accuracy: 0.8642 - val_loss: 0.2864 - val_accuracy: 0.8728
Epoch 147/150
350/350 [==============================] - 35s 99ms/step - loss: 0.3342 -
accuracy: 0.8713 - val_loss: 0.1567 - val_accuracy: 0.8892
Epoch 00147: early stopping
```

En esta celda se lleva a cabo el ajuste de la red para su entrenamiento, es decir, se inicia el entrenamiento del modelo a través de un conjunto de imágenes de perros y gatos. Para el entrenamiento se han usado un total de 8000 imágenes de perros y gatos y un total de 2000 imágenes per el testeo del modelo. Así pues, el proceso de aprendizaje del modelo tiene en cuenta un total de 10000 imágenes, hecho que conlleva un mayor tiempo de computación.

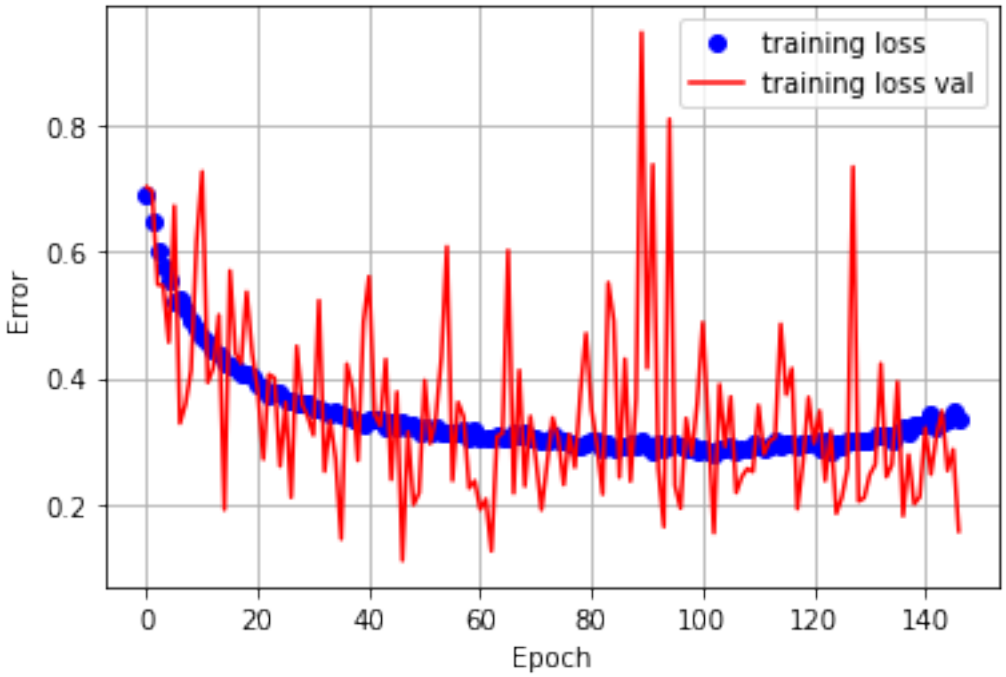
Con el código inicial el tiempo de computación era excesivamente alto, sin exagerar el modelo entrenaba durante horas. Para reducir este tiempo se han realizado dos cambios muy significativos: * Se han reducido los pasos por cada época. En un inicio eran 8000 pasos por época y ahora tan sólo 350, es decir, ahora mismo el modelo entrena con 350 'batches' de imágenes por cada 'epoch'. Como cada 'batch' es de 32 imágenes el modelo entrena con 11200 imágenes en cada 'epoch'. Antes, con los 8000 pasos por época, se entrenaba con 256000 imágenes en cada 'epoch' cosa que hacía que el tiempo de computo se disparase y que el modelo sobre-entrenase. * Se ha añadido un 'Earlystopper' al modelo. Básicamennte, es una función de Keras que obliga al modelo a dejar de entrenar si la métrica elegida no mejora significamente. En nuestro caso, se ha decido que el modelo deje de entrenar si la diferencia entre los errores es de 0.01. En otros modelos vistos durante la búsqueda, esta diferencia era aún menor.

Gracias a la visualización de la evolución del error del entrenamiento y testeo durante las pruebas se ha decidido aumentar el número de épocas a 150. En comparación a las pruebas realizadas, es un aumento considerable. Sin embargo, gracias al 'Earlystopper' definido, si el modelo no necesita tantas épocas para su entrenamiento el modelo se detendrá.

De esta manera, una vez ejecutada esta celda podemos observar la accuracy que presenta el modelo. Se ha logrado una ligera mejora respeto a las pruebas anteriores. El modelo es capaz de clasificar correctamente el 88,92% de las imagenes. Un resultado satisfactorio. Además, cabe destacar la activación del 'Earlystopper' en la época 147.

6.5 Plot de la evolución de la pérdida

[9]:
```
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

# Plots 'history'
history_dict=history.history
loss_values = history_dict['loss']
val_loss_values=history_dict['val_loss']
plt.plot(loss_values,'bo',label='training loss')
plt.plot(val_loss_values,'r',label='training loss val')
plt.legend()
plt.grid()
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()
```



Toda esta celda es un añadido al codigo inicial. Se ha decidido que sería de gran ayuda poder observar la evolución del error durante el aprendizaje y así poder observar si el modelo llega a estabilizarse. De esta manera, el plot obtenido ha sido de gran ayuda para la decisión del aumento de épocas durante el entrenamiento, como se ha mencionado anteriormente.

Como se puede observar en el plot obtenido, a medida que avanza el modelo, que aumentan las épocas, el modelo logra establecerse.

6.6 Comprobación de la funcionalidad de la CNN

Por último, en la siguiente celda no hay cambios a destacar. Tan sólo se han añadido dos 'prints' para observar si el modelo identifica de forma positiva una imagen. En esta caso, se ha decidido cargar la imagen de un perro.

[10]:
```
random.seed(42)
np.random.seed(42)
tf.random.set_seed(42)

# Parte 3 - Cómo hacer nuevas predicciones
test_image = image.load_img(r'C:\Users\munta\OneDrive\Escritorio\Aprendizaje␣
↪Profundo\G03_Project2B\G03_Project2B\deeplearning-az-master\datasets\Part 2␣
↪- Convolutional Neural Networks (CNN)\dataset\single_prediction\cat_or_dog_1.
↪jpg', target_size = (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
training_dataset.class_indices
if result[0][0] == 1:
    prediction = 'dog'
else:
    prediction = 'cat'

#NUEVO
print("El modelo considera que en la imagen hay un ", prediction)
print("La imagen contiene de verdad un perro")
```

```
El modelo considera que en la imagen hay un  dog
La imagen contiene de verdad un perro
```

Como se puede observar, el modelo logra identificar de forma correcta la imagen.

7 Conclusiones y trabajo futuro

Para finalizar, con este tabajo podemos concluir que el objetivo principal de la práctica (aumentar el 'accuracy' del modelo mientras se evita que sobre-entrene) ha sido alcanzado satisfactoriamente. Hemos partido de un modelo con un 'accuracy' en las imágenes de testeo del 80-79% y con un 'overfitting' considerable y hemos llegado a tener un 'accuracy' de casi el 89% y un 'overfitting' nulo.

Como trabajo futuro queda probar nuestro modelo con un dataset nuevo y verificar que funciona y seguir testeando nuevas arquitecturas para intentar aumentar aún más el 'accuracy'.