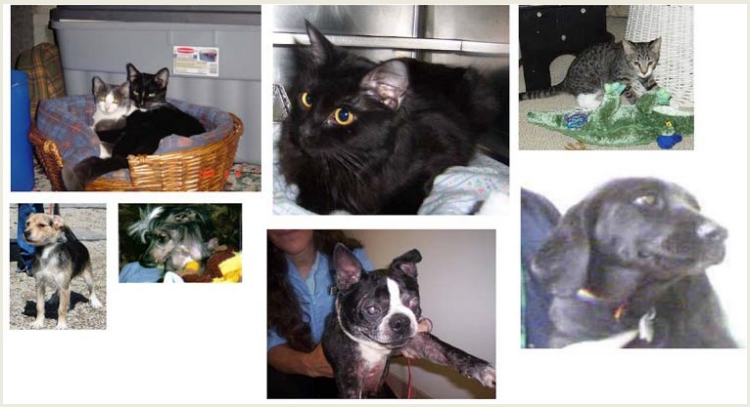
| ML_CLASIFICACION_CNN_02 | Red Neuronal Convolucional para Clasificación de Gatos | ML |
|---|---|---|

El objetivo de esta práctica es construir una red neuronal basada en convolución para la clasificación de gatos. Utilizamos una base de datos que contiene 25.000 imágenes (150x150 pixeles) de perros y gatos (12.500 perros y 12.500 gatos). Se utilizan 2.000 imágenes para entrenamiento (1.000 de cada clase), 1.000 para validación (500 de cada clase) y 1.000 para testeo y prueba (500 de cada clase).



Ejemplos de imágenes de la base de datos

En este caso, la base de datos no se integra dentro de la librería Keras, por lo que debemos descargarla. En concreto se encuentra disponible en Kaggle (www.kaggle.com/c/dogs-vs-cats/data). Es necesario crear previamente una cuenta para su descarga. Las imágenes son de media resolución en formato JPEG.

**SOLUCIÓN**

Creamos el código que organiza toda la información.

```
# Librerías
import os, sutil

# Directorio donde el dataset original fue descomprimido
original_dataset_dir = '/data'
# Directorio donde se construye la base de datos pequeña
base_dir = '/cats_and_dogs_small'
# Creamos los directorios (base, entrenamiento, validación y test para perros y gatos)
os.mkdir(base_dir)
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)
# Directorio con imágenes de gatos para entrenamiento
```

```
train_cats_dir = os.path.join(train_dir, 'cats')
os.mkdir(train_cats_dir)


# Directorio con imágenes de perros para entrenamiento
train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)
# Directorio con imágenes de gatos para validación
validation_cats_dir = os.path.join(validation_dir, 'cats')
os.mkdir(validation_cats_dir)

# Directorio con imágenes de perros para validación
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
os.mkdir(validation_dogs_dir)
# Directorio con imágenes de gatos para test
test_cats_dir = os.path.join(test_dir, 'cats')
os.mkdir(test_cats_dir)
# Directorio con imágenes de perros para test
test_dogs_dir = os.path.join(test_dir, 'dogs')
os.mkdir(test_dogs_dir)

# Copiamos las primeras 1.000 imágenes de gatos al directorio de entrenamiento
fnames = ['cat.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(train_cats_dir, fname)
        shutil.copyfile(src, dst)

# Copiamos las siguientes 500 imágenes de gatos al directorio de validación
fnames = ['cat.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(validation_cats_dir, fname)
        shutil.copyfile(src, dst)

# Copiamos las siguientes 500 imágenes de gatos al directorio de test
fnames = ['cat.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(test_cats_dir, fname)
        shutil.copyfile(src, dst)

# Copiamos las primeras 1.000 imágenes de perros al directorio de entrenamiento
fnames = ['dog.{}.jpg'.format(i) for i in range(1000)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(train_dogs_dir, fname)
        shutil.copyfile(src, dst)

# Copiamos las siguientes 500 imágenes de perros al directorio de validación
fnames = ['dog.{}.jpg'.format(i) for i in range(1000, 1500)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(validation_dogs_dir, fname)
        shutil.copyfile(src, dst)

# Copiamos las siguientes 500 imágenes de perros al directorio de test
fnames = ['dog.{}.jpg'.format(i) for i in range(1500, 2000)]
for fname in fnames:
        src = os.path.join(original_dataset_dir, fname)
        dst = os.path.join(test_dogs_dir, fname)
        shutil.copyfile(src, dst)
```

Comprobamos que el número de imágenes es correcto.

```
print('total training cat images:', len(os.listdir(train_cats_dir)))
total training cat images: 1000
print('total training dog images:', len(os.listdir(train_dogs_dir)))
total training dog images: 1000
print('total validation cat images:', len(os.listdir(validation_cats_dir)))
total validation cat images: 500
print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
total validation dog images: 500
print('total test cat images:', len(os.listdir(test_cats_dir)))
total test cat images: 500
print('total test dog images:', len(os.listdir(test_dogs_dir)))
total test dog images: 500
```

Construimos ahora la red neuronal basada en convolución. Se comienza con imágenes de entrada de 150x150, y termina en un mapa de características de 7x7, justo antes de la capa Flatten.

La profundidad de los mapas de características se incrementa desde 32 hasta 128, y su tamaño decrece de 148x148 hasta 7x7.

Como se trata de una clasificación binaria, la red termina con una sola capa (Dense Layer de tamaño 1) y una función sigmoide para la activación. Esta unidad devolverá la probabilidad asociada a una u otra clase (perro o gato). El código es el siguiente:

```
from keras import layers
from keras import models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

La arquitectura de la red es:

```
model.summary()

Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
_________________________________________________________________
maxpooling2d_1 (MaxPooling2D) (None, 74, 74, 32)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
_________________________________________________________________
maxpooling2d_2 (MaxPooling2D) (None, 36, 36, 64)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
_________________________________________________________________
maxpooling2d_3 (MaxPooling2D) (None, 17, 17, 128)      0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
```

```
_________________________________________________________________
maxpooling2d_4 (MaxPooling2D)  (None, 7, 7, 128)     0
_________________________________________________________________
flatten_1 (Flatten)            (None, 6272)          0
_________________________________________________________________
dense_1 (Dense)                (None, 512)           3211776
_________________________________________________________________
dense_2 (Dense)                (None, 1)             513
=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
```

Definimos la pérdida, el optimizador y la métrica:

```
from keras import optimizers
model.compile(loss='binary_crossentropy',
optimizer=optimizers.RMSprop(lr=1e-4),
metrics=['acc'])
```

Preprocesamos los datos. Leemos las imágenes (RGB), las convertimos en tensores (0 a 255) y los reescalamos a valores entre 0 y 1.

```
# Importamos librerías
from keras.preprocessing.image import ImageDataGenerator
# Reescalamos las imágenes
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
# Reescalamos todas las imágenes a 150 x 150
train_generator = train_datagen.flow_from_directory(
        train_dir,
        target_size=(150, 150)
        batch_size=20,
        class_mode='binary')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
```

Ajustamos el modelo utilizando un generador:

```
history = model.fit_generator(
        train_generator,
        steps_per_epoch=100,
        epochs=30,
        validation_data=validation_generator,
        validation_steps=50)

Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Epoch 1/30
100/100 [==============================] - 198s 2s/step - loss: 0.6864 - acc: 0.5510 - val_loss:
0.6806 - val_acc: 0.5090
Epoch 2/30
100/100 [==============================] - 198s 2s/step - loss: 0.6574 - acc: 0.6100 - val_loss:
0.6409 - val_acc: 0.6460
```

```
Epoch 3/30
100/100 [==============================] - 204s 2s/step - loss: 0.6165 - acc: 0.6670 - val_loss:
0.6400 - val_acc: 0.6490
Epoch 4/30
100/100 [==============================] - 211s 2s/step - loss: 0.5710 - acc: 0.7095 - val_loss:
0.5937 - val_acc: 0.6690
Epoch 5/30
100/100 [==============================] - 196s 2s/step - loss: 0.5343 - acc: 0.7375 - val_loss:
0.5862 - val_acc: 0.6890
Epoch 6/30
100/100 [==============================] - 194s 2s/step - loss: 0.4970 - acc: 0.7530 - val_loss:
0.6070 - val_acc: 0.6800
Epoch 7/30
100/100 [==============================] - 199s 2s/step - loss: 0.4680 - acc: 0.7735 - val_loss:
0.6020 - val_acc: 0.6810
Epoch 8/30
100/100 [==============================] - 198s 2s/step - loss: 0.4410 - acc: 0.7975 - val_loss:
0.5464 - val_acc: 0.7210
Epoch 9/30
100/100 [==============================] - 302s 3s/step - loss: 0.4059 - acc: 0.8115 - val_loss:
0.5839 - val_acc: 0.7030
Epoch 10/30
100/100 [==============================] - 205s 2s/step - loss: 0.3868 - acc: 0.8285 - val_loss:
0.5782 - val_acc: 0.7170
Epoch 11/30
100/100 [==============================] - 196s 2s/step - loss: 0.3579 - acc: 0.8465 - val_loss:
0.5779 - val_acc: 0.7060
Epoch 12/30
100/100 [==============================] - 199s 2s/step - loss: 0.3317 - acc: 0.8520 - val_loss:
0.5630 - val_acc: 0.7370
Epoch 13/30
100/100 [==============================] - 200s 2s/step - loss: 0.3004 - acc: 0.8725 - val_loss:
0.5746 - val_acc: 0.7400
Epoch 14/30
100/100 [==============================] - 194s 2s/step - loss: 0.2815 - acc: 0.8860 - val_loss:
0.6046 - val_acc: 0.7370
Epoch 15/30
100/100 [==============================] - 198s 2s/step - loss: 0.2591 - acc: 0.8980 - val_loss:
0.6804 - val_acc: 0.6970
Epoch 16/30
100/100 [==============================] - 200s 2s/step - loss: 0.2307 - acc: 0.9120 - val_loss:
0.6586 - val_acc: 0.6980
Epoch 17/30
100/100 [==============================] - 198s 2s/step - loss: 0.2150 - acc: 0.9190 - val_loss:
0.6267 - val_acc: 0.7240
Epoch 18/30
100/100 [==============================] - 198s 2s/step - loss: 0.1949 - acc: 0.9220 - val_loss:
0.6216 - val_acc: 0.7480
Epoch 19/30
100/100 [==============================] - 199s 2s/step - loss: 0.1692 - acc: 0.9430 - val_loss:
0.6927 - val_acc: 0.7250
Epoch 20/30
100/100 [==============================] - 200s 2s/step - loss: 0.1508 - acc: 0.9465 - val_loss:
0.7042 - val_acc: 0.7290
Epoch 21/30
100/100 [==============================] - 198s 2s/step - loss: 0.1330 - acc: 0.9550 - val_loss:
0.8791 - val_acc: 0.6810
Epoch 22/30
100/100 [==============================] - 201s 2s/step - loss: 0.1088 - acc: 0.9645 - val_loss:
0.7753 - val_acc: 0.7340
Epoch 23/30
100/100 [==============================] - 198s 2s/step - loss: 0.1031 - acc: 0.9665 - val_loss:
```

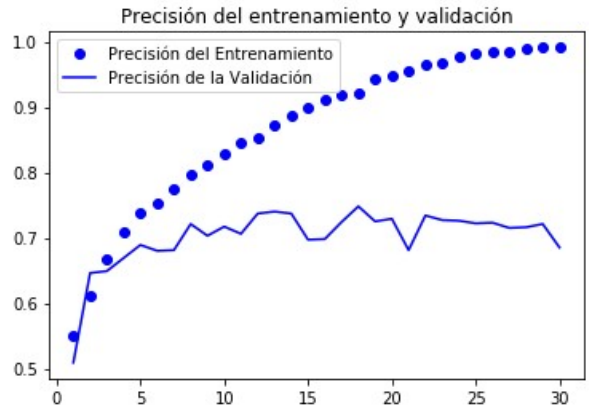
```
0.7813 - val_acc: 0.7270
Epoch 24/30
100/100 [==============================] - 196s 2s/step - loss: 0.0835 - acc: 0.9765 - val_loss:
0.8911 - val_acc: 0.7260
Epoch 25/30
100/100 [==============================] - 193s 2s/step - loss: 0.0695 - acc: 0.9810 - val_loss:
0.8778 - val_acc: 0.7220
Epoch 26/30
100/100 [==============================] - 192s 2s/step - loss: 0.0620 - acc: 0.9835 - val_loss:
0.8874 - val_acc: 0.7230
Epoch 27/30
100/100 [==============================] - 192s 2s/step - loss: 0.0541 - acc: 0.9835 - val_loss:
0.9895 - val_acc: 0.7150
Epoch 28/30
100/100 [==============================] - 195s 2s/step - loss: 0.0435 - acc: 0.9895 - val_loss:
1.0163 - val_acc: 0.7160
Epoch 29/30
100/100 [==============================] - 193s 2s/step - loss: 0.0399 - acc: 0.9905 - val_loss:
1.1198 - val_acc: 0.7210
Epoch 30/30

100/100 [==============================] - 190s 2s/step - loss: 0.0351 - acc: 0.9910 - val_loss:
1.3100 - val_acc: 0.6850
```

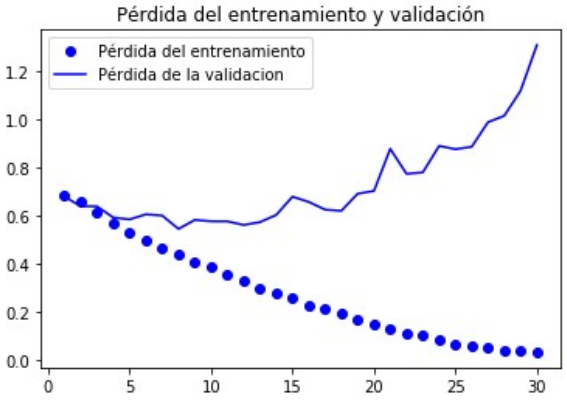
Guardamos el modelo:

```
model.save('cats_and_dogs_small_1.h5')
```

Visualizamos ahora la pérdida y precisión del ajuste del modelo después del proceso de entrenamiento y validación.

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```




Estas curvas demuestran que estamos 'sobreajustando' (overfitting) con los datos de entrenamiento. La precisión aumenta progresivamente para los datos de entrenamiento hasta casi el 100%, mientras que para la validación se queda entorno al 70 %. La pérdida decrece en el entrenamiento mientras que en la validación aumenta.

Este problema se produce debido a que la muestra utilizada para el entrenamiento (2.000 imágenes) es reducida. Este problema se puede reducir mediante regularización, o más concretamente con "Data Augmentation" para el caso de Deep Learning y Computer Visión.