Actividad 2

Jorge Augusto Balsells Orellana
Erick Wilfredo Díaz Saborio

May 24, 2021

1 Actividad 2: Reconocimiento de imagenes más complejas utilizando redes neuronales convolucionales.

En esta actividad se llevará el reconocimiento de imagenes al siguiente nivel, reconociendo imágenes reales de Gatos y Perros para clasificar una imagen entrante como una u otra. En particular, algunos dataset como el reconocimiento de la escritura a mano agilizará el proceso al hacer que todas las imágenes tengan el mismo tamaño y forma, y todas fueran monocromo. Las imágenes del mundo real no son así... tienen diferentes formas, proporciones de aspecto, etc., ¡y normalmente son en color!

Así que, como parte de la tarea necesaria para el proceso de los datos... no es menos importante redimensionarlos para que tengan una forma uniforme.

Para completar esta actividad, se deben realizar los siguientes pasos:

1. Explorar los datos de ejemplo de gatos y perros
2. Preprocesar los datos
3. Construir y entrenar una red neuronal para reconocer la diferencia entre los dos
4. Evaluar la precisión del entrenamiento y la validación

2 Exploración de los datos

Comenzar descargando los datos de ejemplo, un .zip de 2.000 fotos JPG de gatos y perros, y extrayéndolo localmente en /tmp.

[90]:
```
import matplotlib.pyplot as plt
import os
import zipfile
import tensorflow as tf
from tensorflow import keras
from PIL import Image
import numpy as np
```

[ ]:
```
import urllib.request
url = ' https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip'
filename = '/tmp/cats_and_dogs_filtered.zip'
urllib.request.urlretrieve(url, filename)
```

El siguiente código en Python utiliza la biblioteca del OS para usar las bibliotecas del sistema operativo, proporcionando acceso al sistema de archivos, y la biblioteca de archivos zip, que permite descomprimir los datos.

[ ]:
```
local_zip = 'cats_and_dogs_filtered.zip'

zip_ref = zipfile.ZipFile(local_zip, 'r')

zip_ref.extractall('.')
zip_ref.close()
```

A continuación se definen los directorios a utilizar para entrenar esta red neuronal

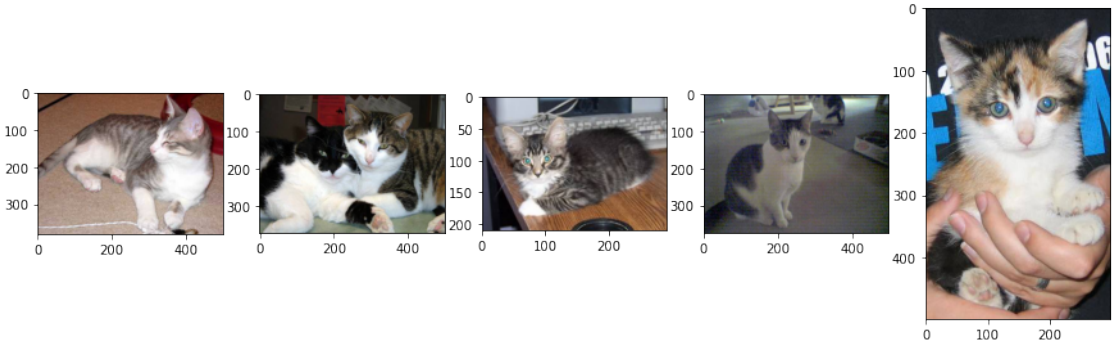
[2]:
```
base_dir = 'cats_and_dogs_filtered'

train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')

# Directorio para la entrenamiento de las imagenes de gatos
train_cats_dir = os.path.join(train_dir, 'cats')

# Directorio para la validación de las imagenes de perros
train_dogs_dir = os.path.join(train_dir, 'dogs')

# Directorio para la validación de las imagenes de perros y gatos
validation_cats_dir = os.path.join(validation_dir, 'cats')
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
```

Se observan los nombres de los archivos en los directorios de los entrenamientos de perros y gatos (las convenciones de nombres de archivos son las mismas en el directorio de validación):

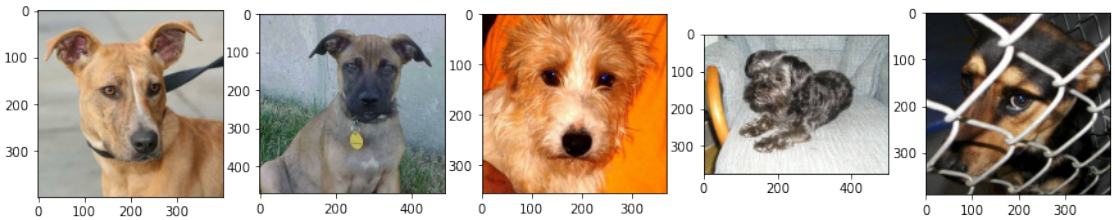
[3]:
```
train_cat_fnames = os.listdir( train_cats_dir )
train_dog_fnames = os.listdir( train_dogs_dir )

print(train_cat_fnames[:10])
print(train_dog_fnames[:10])
```

```
['cat.66.jpg', 'cat.976.jpg', 'cat.611.jpg', 'cat.607.jpg', 'cat.918.jpg',
'cat.845.jpg', 'cat.273.jpg', 'cat.95.jpg', 'cat.552.jpg', 'cat.109.jpg']
['dog.750.jpg', 'dog.367.jpg', 'dog.320.jpg', 'dog.941.jpg', 'dog.562.jpg',
'dog.211.jpg', 'dog.812.jpg', 'dog.49.jpg', 'dog.285.jpg', 'dog.572.jpg']
```

[72]:
```
fix, axs = plt.subplots(nrows=1, ncols=5, figsize=(15,15))
for i in range(5):
    img_path=os.path.join(train_cats_dir,train_cat_fnames[i])
    axs[i].imshow(Image.open(img_path))
```



[73]:
```
fix, axs = plt.subplots(nrows=1, ncols=5, figsize=(15,15))
for i in range(5):
    img_path=os.path.join(train_dogs_dir,train_dog_fnames[i])
    axs[i].imshow(Image.open(img_path))
```



Las imágenes tienen diferentes tamaños por lo que hay que estandarizar el tamaño en el preprocesamiento.

2.1 Dataset Size

Se descubre el número total de imágenes de gatos y perros en los directorios de training y validación:

[4]:
```
print('total training cat images :', len(os.listdir(      train_cats_dir ) ))
print('total training dog images :', len(os.listdir(      train_dogs_dir ) ))

print('total validation cat images :', len(os.listdir( validation_cats_dir ) ))
print('total validation dog images :', len(os.listdir( validation_dogs_dir ) ))
```

```
total training cat images : 1000
total training dog images : 1000
total validation cat images : 500
total validation dog images : 500
```

Se diseñará ahora una CNN para la clasificación de un problema binario (perros y gatos):

Ejercicio 1 (5 puntos): Diseñar una red neuroanl convolucional para clasificar las imagenes de perros y gatos. Evaluar los requisitos de la red neuronal que se pide y construirla completando

los parametros necesarios para que las capas de la red neuronal sean optimas para la tarea de clasificar los perros y gatos.

[5]:
```
# tu código para la red neuronal del ejercicio 1 aquí
modelo = keras.models.Sequential([
    keras.layers.Conv2D(filters=64, kernel_size=5, activation="relu",
                        padding="same", input_shape=[150, 150, 3]),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Conv2D(filters=128, kernel_size=3, activation="relu",␣
 ↪padding="same"),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Conv2D(filters=256, kernel_size=3, activation="relu",␣
 ↪padding="same"),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Flatten(),
    keras.layers.Dense(32, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")

])
```

Se comprueba el modelo:

[6]:
```
modelo.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 150, 150, 64)      4864
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 75, 75, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 75, 75, 128)       73856
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 37, 37, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 37, 37, 256)       295168
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 18, 18, 256)       0
_________________________________________________________________
flatten (Flatten)            (None, 82944)             0
_________________________________________________________________
dense (Dense)                (None, 32)                2654240
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 3,028,161
Trainable params: 3,028,161
```

```
Non-trainable params: 0
_________________________________________________________________
```

Ejercicio 2 (1 punto): A continuación habrá que compilar el modelo, para ello usar el optimizador RMSprop, para el loss. Se debe tener en cuenta los elementos a clasificar, en este caso perros y gatos (un clasificador binario). Como métrica mostrar la precisión.

[7]:
```
# tu código para la compilación del ejercicio 2 aquí
optimizer = tf.keras.optimizers.RMSprop(
    learning_rate=0.01,
    epsilon=0.01
)

modelo.compile(loss="binary_crossentropy",
              optimizer=optimizer, metrics=["accuracy"])
```

3 Preprocesamiento de los datos

Una buena practica cuando se trate de clasificar imagenes, es pretratar los datos (imagenes) para mejorar la precisión de la CNN

[42]:
```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Todas las imagenes tienen que ser reescaldas a 1./255.
train_datagen = ImageDataGenerator( rescale = 1.0/255. )
test_datagen  = ImageDataGenerator( rescale = 1.0/255. )

# --------------------
# Flujo de imagenes de entrenamiento en batches de 20 usando el "train_datagen␣
 ↪generator"
# --------------------
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='binary',
                                                    target_size=(150, 150))
# --------------------
# Flujo de imagenes de entrenamiento en batches de 20 usando el "test_datagen␣
 ↪generator"
# --------------------
validation_generator =  test_datagen.flow_from_directory(validation_dir,
                                                         batch_size=20,
                                                         class_mode  = 'binary',
                                                         target_size = (150,␣
 ↪150))
```

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

4 Construcción del modelo, entreno y validación.

Ejercicio 3 (1 puntos): Para concluir la creación de la CNN para la clasificación de perros y gatos, falta entrenar la red neuronal. Para ello escribir, crear la variable "history" y guardar en esa variable el modelo entrenado de la CNN. Para el entrenamiento, se deberán definir los "step_per_epoch", el número de "epochs", el número de "validation_steps" y usar "verbose=2".

[9]:
```
# tu código para la variable history que guarda el entrenamiento de la CNN del␣
↪ejercicio 3 aquí
step_size_train = train_generator.n//train_generator.batch_size
step_size_val = validation_generator.n//validation_generator.batch_size
history = modelo.fit(
    train_generator,
    epochs=50,
    steps_per_epoch=step_size_train,
    validation_data=validation_generator,
    validation_steps=step_size_val,
    verbose=2
)
```

```
Epoch 1/50
100/100 - 26s - loss: 0.9120 - accuracy: 0.4880 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 2/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4910 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 3/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4980 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 4/50
100/100 - 5s - loss: 0.6937 - accuracy: 0.4900 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 5/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4630 - val_loss: 0.6931 -
val_accuracy: 0.5000
Epoch 6/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.5000 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 7/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4930 - val_loss: 0.6931 -
val_accuracy: 0.5000
Epoch 8/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4990 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 9/50
100/100 - 5s - loss: 0.6931 - accuracy: 0.5140 - val_loss: 0.6934 -
val_accuracy: 0.5000
Epoch 10/50
```

```
100/100 - 5s - loss: 0.6934 - accuracy: 0.5030 - val_loss: 0.6933 -
val_accuracy: 0.5000
Epoch 11/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4950 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 12/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4870 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 13/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4910 - val_loss: 0.6933 -
val_accuracy: 0.5000
Epoch 14/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.5030 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 15/50
100/100 - 5s - loss: 0.6934 - accuracy: 0.5000 - val_loss: 0.6934 -
val_accuracy: 0.5000
Epoch 16/50
100/100 - 5s - loss: 0.6937 - accuracy: 0.4950 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 17/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4950 - val_loss: 0.6933 -
val_accuracy: 0.5000
Epoch 18/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4970 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 19/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4600 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 20/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4980 - val_loss: 0.6933 -
val_accuracy: 0.5000
Epoch 21/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4940 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 22/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4940 - val_loss: 0.6933 -
val_accuracy: 0.5000
Epoch 23/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4900 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 24/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4820 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 25/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4830 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 26/50
```

```
100/100 - 5s - loss: 0.6936 - accuracy: 0.4960 - val_loss: 0.6931 -
val_accuracy: 0.5000
Epoch 27/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4990 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 28/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4860 - val_loss: 0.6933 -
val_accuracy: 0.5000
Epoch 29/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.5020 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 30/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4980 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 31/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4970 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 32/50
100/100 - 5s - loss: 0.6934 - accuracy: 0.4980 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 33/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4910 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 34/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4890 - val_loss: 0.6931 -
val_accuracy: 0.5000
Epoch 35/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4910 - val_loss: 0.6931 -
val_accuracy: 0.5000
Epoch 36/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4960 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 37/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4920 - val_loss: 0.6931 -
val_accuracy: 0.5000
Epoch 38/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4900 - val_loss: 0.6931 -
val_accuracy: 0.5000
Epoch 39/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4790 - val_loss: 0.6933 -
val_accuracy: 0.5000
Epoch 40/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4910 - val_loss: 0.6933 -
val_accuracy: 0.5000
Epoch 41/50
100/100 - 5s - loss: 0.6937 - accuracy: 0.4810 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 42/50
```

```
100/100 - 5s - loss: 0.6935 - accuracy: 0.5010 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 43/50
100/100 - 5s - loss: 0.6934 - accuracy: 0.5010 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 44/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4940 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 45/50
100/100 - 5s - loss: 0.6935 - accuracy: 0.4950 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 46/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4770 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 47/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4930 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 48/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4930 - val_loss: 0.6931 -
val_accuracy: 0.5000
Epoch 49/50
100/100 - 5s - loss: 0.6936 - accuracy: 0.4790 - val_loss: 0.6932 -
val_accuracy: 0.5000
Epoch 50/50
100/100 - 5s - loss: 0.6933 - accuracy: 0.5100 - val_loss: 0.6933 -
val_accuracy: 0.5000
```

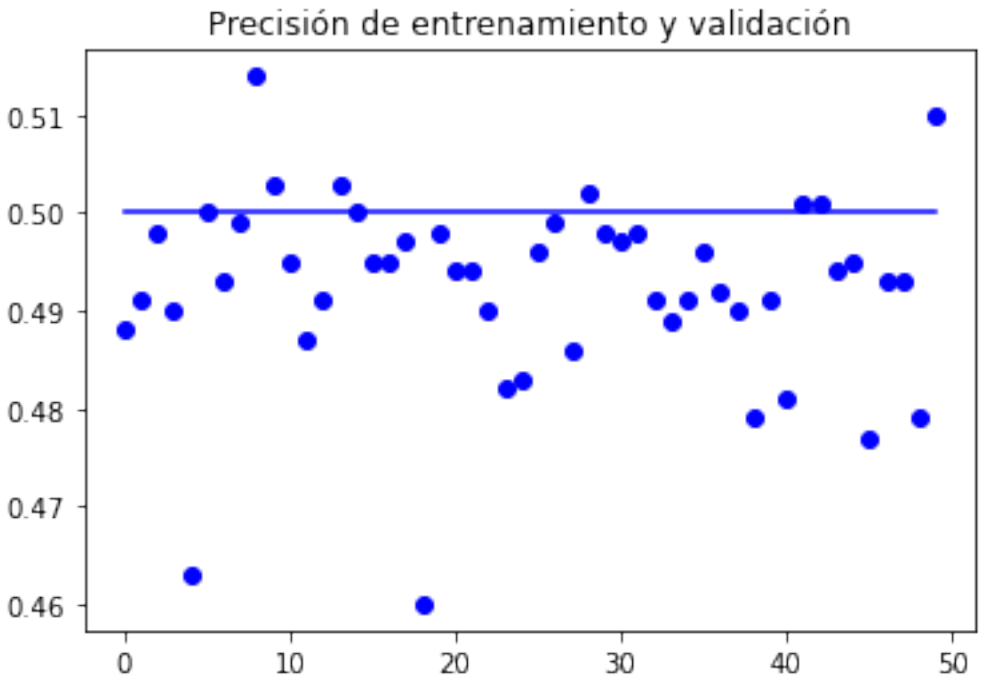
[10]:
```
labels = (train_generator.class_indices)
labels = dict((v,k) for k,v in labels.items())
print(labels)
```

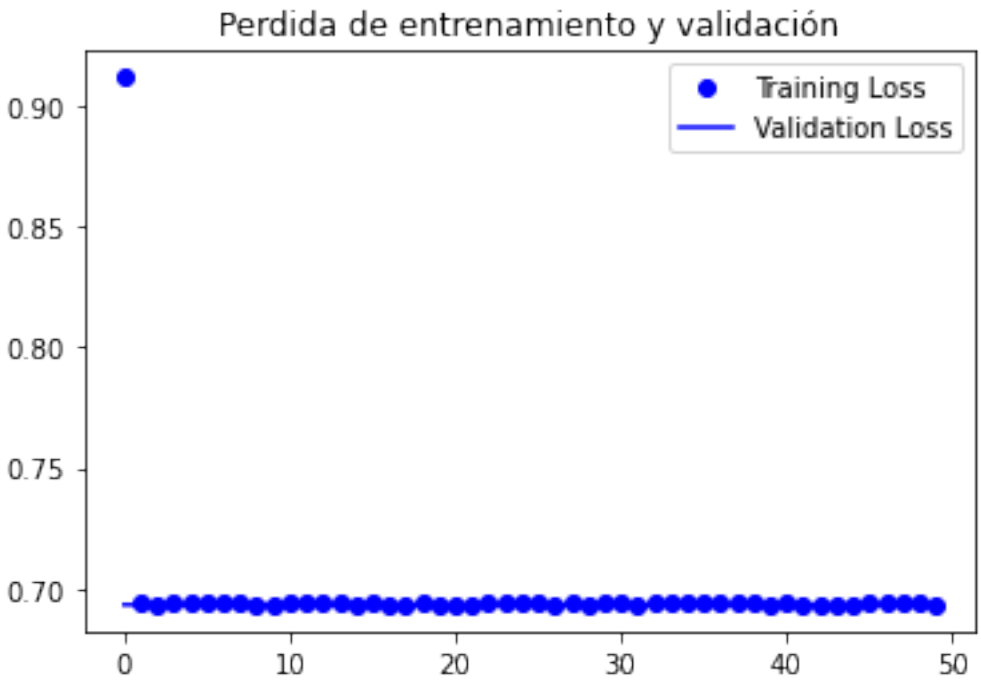
```
{0: 'cats', 1: 'dogs'}
```

5 Evaluando la precisión y la perdida del modelo

[11]:
```
#-----------------------------------------------------------
# Recuperar una lista de resultados de la lista de datos de los conjuntos de␣
 ↪entrenamiento y pruebas para cada epoch de entrenamiento
#-----------------------------------------------------------
import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']
```

```
epochs = range(len(acc))

#------------------------------------------------
# Imprimir la precisión del entrenamiento y validación por epoch
#------------------------------------------------
plt.plot(epochs, acc, 'bo', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Precisión de entrenamiento y validación')

plt.figure()

#------------------------------------------------
# Imprimir la perdida de entrenamiento y validación por epoch
#------------------------------------------------
plt.plot(epochs, loss, 'bo', label='Training Loss')
plt.plot(epochs, val_loss, 'b', label='Validation Loss')
plt.title('Perdida de entrenamiento y validación')
plt.legend()

plt.show()
```





Como se puede observar, este es un claro ejemplo de overfitting, algo que naturalmente se quiere evitar para que las redes neuronales sean lo más precisas posibles.

Ejercicio 4 (3 puntos): Para solucionar este problema, se debe rehacer algunas partes del código anterior utilizando las técnicas de regularización aprendidas en clase, como el data augmentation.

Consejo: Volver a escribir otra vez todas las secciones y cambiar el código necesario aplicando las técnicas de regularización elegidas. Se debe recordar que este tipo de técnicas se suele usar en el preprocesamiento de los datos.

Aclaración: Se espera que al menos se use una de las técnicas de regularización aprendidas en clase. Se valorará positivamente razonar el motivo del uso de la técnica escogida (o escogidas en el caso de ser varias) y no las otras.

6 Técnicas de Regularización

6.1 Dropout

Se aplico dropout es una técnica de regularización destructiva donde se elimina de forma aleatoria un porcentaje de los neuronas durante el entrenamiento. Se aplicaron 2 dropout de keras antes de las capas fully connected y entre las capas fully connected eliminando el 30% de las neuronas. Aplicando dropout a diferencia de el primer modelo se logro reducir la perdida de validación y aumentar la precisión.

6.2 Early Stopping

En aprendizaje automático, *"Early Stopping"* o *"detección temprana"*, es una técnica de regularización para evitar un *"overfitting"* o *"sobreajuste"* al entrenar una CNN. En Keras es un callback de una función que monitorea el resultado del "Loss" con el conjunto de datos de validación. En este caso, si no mejora despues de 3 epochs, se detiene el entreenamiento. Se llego a un epoch 31 de 50, se logro detener el entrenamiento del modelo antes de iniciar a hacer un "overfitting".

[24]:
```
### Tu código aquí para la reducción del overfitting del ejercicio 4 aquí ###
# tu código para la red neuronal del ejercicio 1 aquí
modelo = keras.models.Sequential([
    keras.layers.Conv2D(filters=64, kernel_size=5, activation="relu",
                        padding="same", input_shape=[150, 150, 3]),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Conv2D(filters=128, kernel_size=3, activation="relu",␣
 ↪padding="same"),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Conv2D(filters=256, kernel_size=3, activation="relu",␣
 ↪padding="same"),
    keras.layers.MaxPooling2D(pool_size=2),
    keras.layers.Dropout(0.3),
    keras.layers.Flatten(),
    keras.layers.Dense(32, activation="relu"),
    keras.layers.Dropout(0.3),
    keras.layers.Dense(1, activation="sigmoid")

])
```

[25]:
```
optimizer = tf.keras.optimizers.RMSprop(
    learning_rate=0.0001,
    epsilon=0.01
)

modelo.compile(loss="binary_crossentropy",
               optimizer=optimizer, metrics=["accuracy"])
```

[26]:
```
step_size_train = train_generator.n//train_generator.batch_size
step_size_val = validation_generator.n//validation_generator.batch_size

callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)

history = modelo.fit(
    train_generator,
    epochs=50,
    steps_per_epoch=step_size_train,
    validation_data=validation_generator,
    validation_steps=step_size_val,
    verbose=2,
```

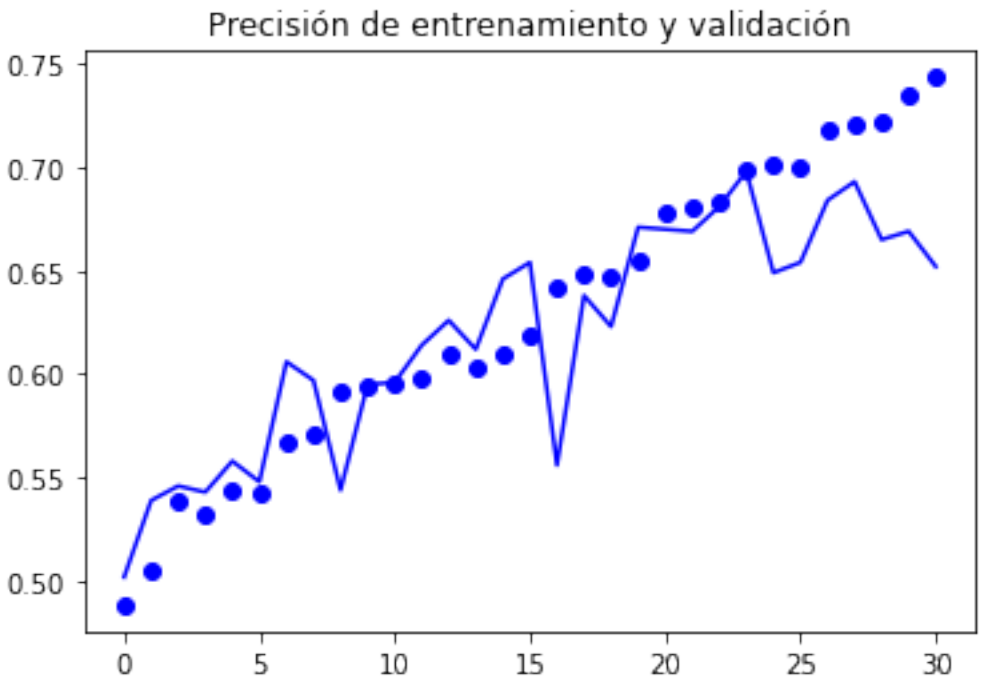
```
    callbacks=[callback]
)
```

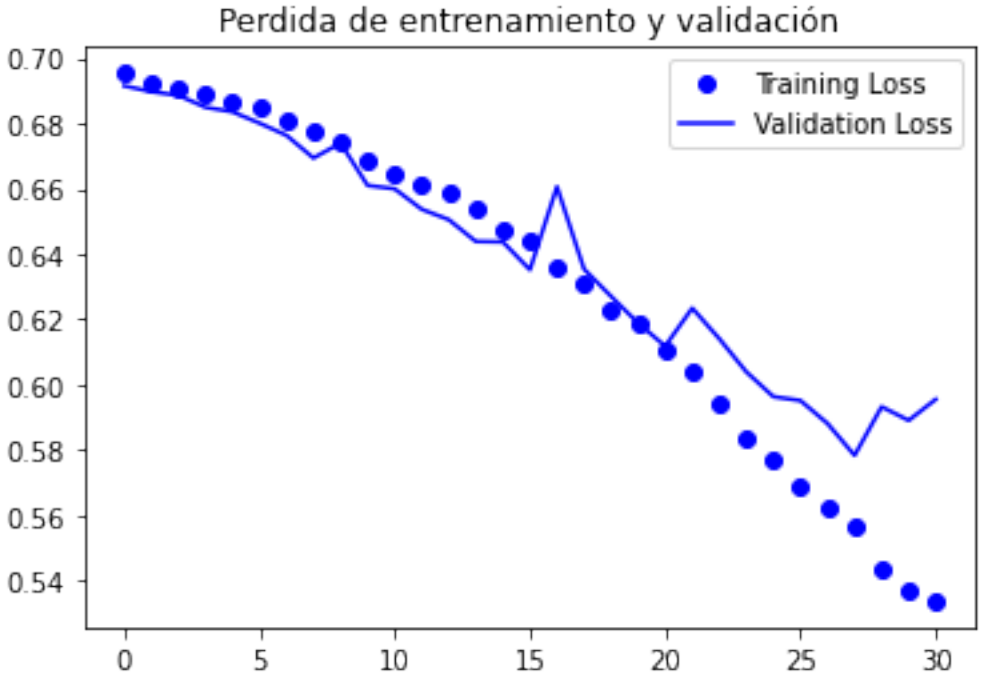
```
Epoch 1/50
100/100 - 6s - loss: 0.6954 - accuracy: 0.4885 - val_loss: 0.6915 -
val_accuracy: 0.5020
Epoch 2/50
100/100 - 5s - loss: 0.6925 - accuracy: 0.5055 - val_loss: 0.6897 -
val_accuracy: 0.5390
Epoch 3/50
100/100 - 5s - loss: 0.6909 - accuracy: 0.5385 - val_loss: 0.6886 -
val_accuracy: 0.5460
Epoch 4/50
100/100 - 5s - loss: 0.6893 - accuracy: 0.5320 - val_loss: 0.6849 -
val_accuracy: 0.5430
Epoch 5/50
100/100 - 5s - loss: 0.6865 - accuracy: 0.5445 - val_loss: 0.6835 -
val_accuracy: 0.5580
Epoch 6/50
100/100 - 5s - loss: 0.6847 - accuracy: 0.5425 - val_loss: 0.6801 -
val_accuracy: 0.5480
Epoch 7/50
100/100 - 5s - loss: 0.6806 - accuracy: 0.5665 - val_loss: 0.6763 -
val_accuracy: 0.6060
Epoch 8/50
100/100 - 5s - loss: 0.6779 - accuracy: 0.5710 - val_loss: 0.6694 -
val_accuracy: 0.5970
Epoch 9/50
100/100 - 5s - loss: 0.6745 - accuracy: 0.5920 - val_loss: 0.6740 -
val_accuracy: 0.5440
Epoch 10/50
100/100 - 5s - loss: 0.6683 - accuracy: 0.5940 - val_loss: 0.6610 -
val_accuracy: 0.5950
Epoch 11/50
100/100 - 5s - loss: 0.6647 - accuracy: 0.5950 - val_loss: 0.6600 -
val_accuracy: 0.5960
Epoch 12/50
100/100 - 5s - loss: 0.6612 - accuracy: 0.5980 - val_loss: 0.6537 -
val_accuracy: 0.6140
Epoch 13/50
100/100 - 5s - loss: 0.6587 - accuracy: 0.6095 - val_loss: 0.6505 -
val_accuracy: 0.6260
Epoch 14/50
100/100 - 5s - loss: 0.6542 - accuracy: 0.6030 - val_loss: 0.6439 -
val_accuracy: 0.6120
Epoch 15/50
100/100 - 5s - loss: 0.6472 - accuracy: 0.6095 - val_loss: 0.6437 -
```

```
val_accuracy: 0.6460
Epoch 16/50
100/100 - 5s - loss: 0.6441 - accuracy: 0.6190 - val_loss: 0.6352 -
val_accuracy: 0.6540
Epoch 17/50
100/100 - 5s - loss: 0.6359 - accuracy: 0.6420 - val_loss: 0.6606 -
val_accuracy: 0.5560
Epoch 18/50
100/100 - 5s - loss: 0.6312 - accuracy: 0.6480 - val_loss: 0.6354 -
val_accuracy: 0.6380
Epoch 19/50
100/100 - 5s - loss: 0.6233 - accuracy: 0.6470 - val_loss: 0.6271 -
val_accuracy: 0.6230
Epoch 20/50
100/100 - 5s - loss: 0.6189 - accuracy: 0.6550 - val_loss: 0.6189 -
val_accuracy: 0.6710
Epoch 21/50
100/100 - 5s - loss: 0.6104 - accuracy: 0.6780 - val_loss: 0.6118 -
val_accuracy: 0.6700
Epoch 22/50
100/100 - 5s - loss: 0.6037 - accuracy: 0.6800 - val_loss: 0.6235 -
val_accuracy: 0.6690
Epoch 23/50
100/100 - 5s - loss: 0.5939 - accuracy: 0.6830 - val_loss: 0.6141 -
val_accuracy: 0.6810
Epoch 24/50
100/100 - 5s - loss: 0.5835 - accuracy: 0.6985 - val_loss: 0.6039 -
val_accuracy: 0.6980
Epoch 25/50
100/100 - 5s - loss: 0.5770 - accuracy: 0.7010 - val_loss: 0.5964 -
val_accuracy: 0.6490
Epoch 26/50
100/100 - 5s - loss: 0.5689 - accuracy: 0.7000 - val_loss: 0.5952 -
val_accuracy: 0.6540
Epoch 27/50
100/100 - 5s - loss: 0.5628 - accuracy: 0.7185 - val_loss: 0.5882 -
val_accuracy: 0.6840
Epoch 28/50
100/100 - 5s - loss: 0.5565 - accuracy: 0.7205 - val_loss: 0.5784 -
val_accuracy: 0.6930
Epoch 29/50
100/100 - 5s - loss: 0.5437 - accuracy: 0.7225 - val_loss: 0.5933 -
val_accuracy: 0.6650
Epoch 30/50
100/100 - 5s - loss: 0.5373 - accuracy: 0.7345 - val_loss: 0.5891 -
val_accuracy: 0.6690
Epoch 31/50
100/100 - 5s - loss: 0.5339 - accuracy: 0.7435 - val_loss: 0.5956 -
```

```
val_accuracy: 0.6520
```

[27]:
```
#-----------------------------------------------------------
# Recuperar una lista de resultados de la lista de datos de los conjuntos de␣
 ↪entrenamiento y pruebas para cada epoch de entrenamiento
#-----------------------------------------------------------
import matplotlib.pyplot as plt

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

epochs = range(len(acc))

#------------------------------------------------
# Imprimir la precisión del entrenamiento y validación por epoch
#------------------------------------------------
plt.plot(epochs, acc, 'bo', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Precisión de entrenamiento y validación')

plt.figure()

#------------------------------------------------
# Imprimir la perdida de entrenamiento y validación por epoch
#------------------------------------------------
plt.plot(epochs, loss, 'bo', label='Training Loss')
plt.plot(epochs, val_loss, 'b', label='Validation Loss')
plt.title('Perdida de entrenamiento y validación')
plt.legend()

plt.show()
```





[16]:
```
labels = (train_generator.class_indices)
labels = dict((v,k) for k,v in labels.items())
```

```
print(labels)
```

```
{0: 'cats', 1: 'dogs'}
```

[30]:
```
modelo.save('actividad2_model.h5')
```

[5]:
```
modelo = keras.models.load_model('actividad2_model.h5')
```

7 Predicciones

[46]:
```
validation_generator.batch_index
```

[46]:
```
0
```

[45]:
```
test_images = []
batch_index = 0
test_label = []
while batch_index <= validation_generator.batch_index:
    data = validation_generator.next()
    test_images.extend(data[0])
    test_label.extend(data[1])
    batch_index = batch_index + 1
```

[78]:
```
predicciones_raw = modelo.predict(validation_generator)
```

[85]:
```
predicciones = np.where(predicciones_raw<0.5,0,1)
```

[87]:
```
con_mat = tf.math.confusion_matrix(labels=test_label, predictions=predicciones).
↪numpy()
```

7.1 Matriz de Confución

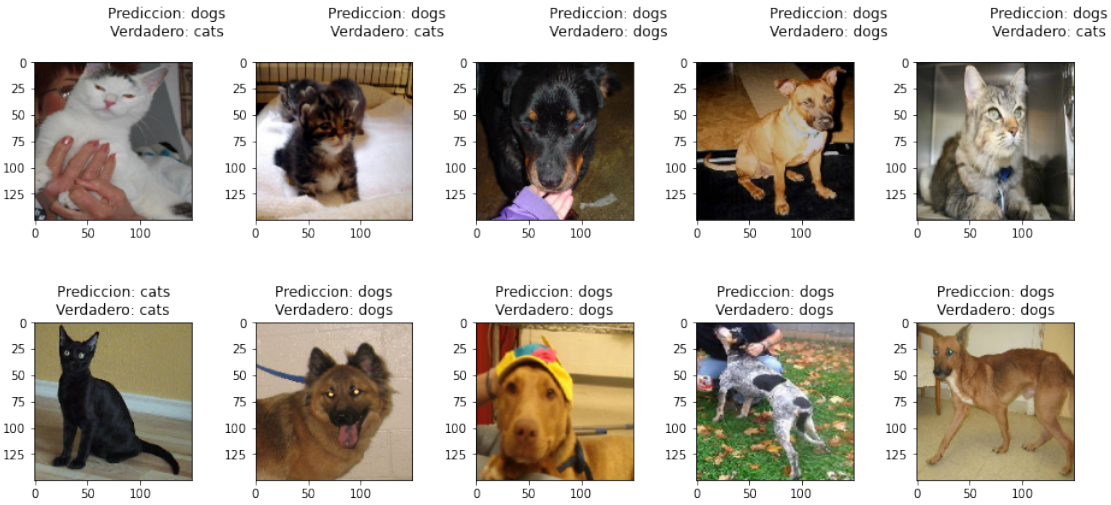
[92]:
```
con_mat
```

[92]:
```
array([[169, 331],
       [149, 351]], dtype=int32)
```

[61]:
```
fix, axs = plt.subplots(nrows=2, ncols=5, figsize=(15,15))
idx = 0


for i in range(5):
    y_hat = 0 if predicciones[idx] < 0.5 else 1
    prediccion_label = labels[y_hat]
    img=test_images[idx]
    axs[0,i].set_title(f"""
```

```
                        Prediccion: {prediccion_label}
                        Verdadero: {labels[test_label[idx]]}
                        """)
    axs[0,i].imshow(img)
    idx+=1

    y_hat = 0 if predicciones[idx] < 0.5 else 1
    prediccion_label = labels[y_hat]
    img=test_images[idx]
    axs[1,i].set_title(f"""Prediccion: {prediccion_label}\nVerdadero:␣
 ↪{labels[test_label[idx]]}""")
    axs[1,i].imshow(img)
    idx+=1

plt.subplots_adjust(left=0.1,
                    bottom=0.1,
                    right=0.9,
                    top=0.5,
                    wspace=0.4,
                    hspace=0.0)
plt.show()
```

7.2 Ejemplos de predicciones y su valor real



[ ]: