P2_CNN_HfaJrc

November 8, 2024

1 Práctica 1.2 - CNNs (2024-2025) - Aprendizaje Profundo (Grado en IA)

Integrantes	Correo electrónico
Hugo Fole Abellás	hugo.fole.abellas@udc.es
José Romero Conde	j.rconde@udc.es

1.1 ***REDES CONVOLUCIONALES***

En esta segunda parte de la práctica desarrollaremos una red convolucional (CNN) para resolver el mismo problema que en la primera parte, identificar el elemento o animal que aparece en una fotografía.

Como ya comentamos, volveremos a utilizar el dataset `CIFAR10` de la librería `keras`. Dicho dataset contiene 60.000 imágenes a color de tamaño 32×32 de las que 50.000 se usarán para el entrenamiento de la red y 10.000 para testearla.

Las imágenes pertenecen a las 10 posibles categorías (6.000 imágenes por categoría).

Inicialmente, importaremos las librería a utilizar.

[17]:
```
import tensorflow as tf
import keras
import numpy as np
import matplotlib.pyplot as plt
```

A continuación, importaremos el dataset que será usado para entrenar la red que crearemos más adelante.

[18]:
```
from keras.datasets import cifar10

(x_train, y_train) , (x_test, y_test) = cifar10.load_data()
```

1.2 NORMALIZACIÓN DE LOS DATOS

Para poder trabajar de manera correcta con este dataset, deberemos seguir ciertos procesos de normalización de datos, de manera que, o bien optimicemos su tiempo de ejecución o bien nos permita trabajar con el dataset. Estos procesos son :

1. `Realizar One-hot encoding en los targets.`
2. `Normalizar los valores de las imágenes a float, ya que vienen en valores de 0 a 255.`

Al utilizar redes convolucionales no es necesario aplanar las imágenes en vectores dado que estas realizan la dependencia entre capas utilizando más de un pixel. Es decir, es justamente la estructura tridimensional de la imagen lo que nos interesa a la hora de convolucionar, ahí es justamente donde se les saca partido a las *CNNs*.

1.2.1 1. One-hot encoding

Como ya comentamos, haremos one-hot en los targets de nuestro dataset ya que vienen divididos en 10 tipos de salidas categóricas representadas por números. Las diferentes salidas las podemos ver en la siguiente tabla. |Número|Categoría| |——|———| |0|airplane| |1|automobile| |2|bird| |3|cat| |4|deer| |5|dog| |6|frog| |7|horse| |8|ship| |9|truck|

[19]:
```
y_train = keras.utils.to_categorical(y_train, num_classes = 10)
y_test = keras.utils.to_categorical(y_test, num_classes = 10)
```

1.2.2 2.Normalización datos

Como ya dijimos, en este apartado realizaremos la parte de normalización de datos de **uint8** con valores entre [0-255] a **float32**, para que pasen a estar en el rango de valores [0,1] ya que esto nos permite trabajar con redes de neuronas, agiliza el entrenamiento de dichas redes y aumenta la precisión de estas.

Además, utilizaremos el tipo de dato **float32** ya que, los datos ocupan la mitad de espacio y utilizaremos mucha menos memoria.

[20]:
```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train / 255.0
x_test = x_test / 255.0
```

1.3 CREACIÓN REDES NEURONALES CONVOLUCIONALES

Las redes neuronales convolucionales son un tipo de red que usa datos tridimensionales para la clasificación de imágenes. Se distinguen de otras redes por su alto rendimiento superior con entradas de imagen, voz o señales de audio. Para ello, tienen una composición especial diferente a la de las redes neuronales convencionales. Se componen de:

- `Capa convolucional` : suelen tener más de una y está compuesta por diversos filtros que tienen parámetros entrenables.
- `Capa de agrupación (pooling)` : estas capas se encargan de reducir la dimensionalidad de las capas, esto lo pueden hacer de dos maneras, por medio del valor máximo o del valor medio.
- `Capa "fully connected"` : esta última capa tendrá dimensión K, siendo K el número de clases en las que se puede clasificar. Como su nombre indica es una capa densa, es decir está completamente conectada. Utiliza una función **softmax** para proporcionar el resultado.

Dado que la finalidad de esta práctica es la clasificación, es redundante la utilización de métricas categóricas y funciones de pérdida categóricas también. Se usarán **funciones de pérdida** categóricas como : - `Categorical Cross Entropy` - `Categorical Focal Cross Entropy` - `...`

Y como **métrica** utilizaremos `Categorical Accuracy` ya que es la que nos da una aproximación que queremos dado que tenemos un dataset en el que la salidas son vectores de 10 valores binarios basados en One Hot enconding.

Para representar las gráficas de `loss` y `accuracy` utilizaremos una función que se nos proporcionó en uno de los laboratorios y también utilizamos en la parte 1 de esta práctica.

[21]:
```
def plot(train, validation, title):
    plt.clf()
    epochs = range(1, len(train) + 1)

    plt.plot(epochs, train, 'b-o', label='Training ' + title)
    plt.plot(epochs, validation, 'r--o', label='Validation '+ title)

    plt.title('Training and validation ' + title)
    plt.xlabel('Epochs')
    plt.ylabel(title)
    plt.legend()
    plt.show()
```

Para poder crear redes tendremos que importar de `keras` la función **layers** que nos permite crear diferentes tipos de capas para introducirlas en la redes que crearemos.

[22]:
```
from keras import layers
```

Comenzaremos creando una red sencilla para ver el funcionamiento y composición de esta. En esta red utilizamos la razón de 2^x para seleccionar el número de filtros por cada capa empezando en 32 filtros. Como tamaño de kernels utilizaremos el 3x3 que es uno de los más utilizados en la actualidad en redes convolucionales y como capas pooling utilizaremos filtros de 2x2 que también son de los más utilizados para el pooling en redes convolucionales. Por último, crearemos una única capa completamente conectada con 10 neuronas (una por cada clase de nuestro problema).

[20]:
```
inputs = keras.Input(shape=(32, 32, 3))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=4, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

#Compile
model.compile(optimizer="adam",
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[keras.metrics.CategoricalAccuracy()])
#Callback
```

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="cnn_cats_dogs_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
#Fitting
history = model.fit(x_train, y_train,validation_split = 0.1 ,epochs=50,␣
↪batch_size=64,callbacks = callbacks)
```

```
Epoch 1/50
704/704             9s 8ms/step -
categorical_accuracy: 0.3485 - loss: 1.7770 - val_categorical_accuracy: 0.5452 -
val_loss: 1.2948
Epoch 2/50
704/704             3s 4ms/step -
categorical_accuracy: 0.5689 - loss: 1.2150 - val_categorical_accuracy: 0.6250 -
val_loss: 1.0704
Epoch 3/50
704/704             2s 3ms/step -
categorical_accuracy: 0.6405 - loss: 1.0275 - val_categorical_accuracy: 0.6592 -
val_loss: 0.9663
Epoch 4/50
704/704             3s 3ms/step -
categorical_accuracy: 0.6821 - loss: 0.9119 - val_categorical_accuracy: 0.6994 -
val_loss: 0.8895
Epoch 5/50
704/704             2s 3ms/step -
categorical_accuracy: 0.7154 - loss: 0.8229 - val_categorical_accuracy: 0.6894 -
val_loss: 0.9057
Epoch 6/50
704/704             3s 5ms/step -
categorical_accuracy: 0.7416 - loss: 0.7540 - val_categorical_accuracy: 0.7164 -
val_loss: 0.8467
Epoch 7/50
704/704             4s 4ms/step -
categorical_accuracy: 0.7630 - loss: 0.6907 - val_categorical_accuracy: 0.7112 -
val_loss: 0.8372
Epoch 8/50
704/704             5s 4ms/step -
categorical_accuracy: 0.7805 - loss: 0.6419 - val_categorical_accuracy: 0.7286 -
val_loss: 0.7970
Epoch 9/50
704/704             4s 5ms/step -
categorical_accuracy: 0.8019 - loss: 0.5774 - val_categorical_accuracy: 0.7302 -
val_loss: 0.8187
Epoch 10/50
```

```
704/704 4s 4ms/step -
categorical_accuracy: 0.8162 - loss: 0.5327 - val_categorical_accuracy: 0.7366 -
val_loss: 0.7983
Epoch 11/50
704/704 5s 4ms/step -
categorical_accuracy: 0.8256 - loss: 0.5015 - val_categorical_accuracy: 0.7274 -
val_loss: 0.8268
Epoch 12/50
704/704 2s 3ms/step -
categorical_accuracy: 0.8442 - loss: 0.4518 - val_categorical_accuracy: 0.7320 -
val_loss: 0.8477
Epoch 13/50
704/704 3s 4ms/step -
categorical_accuracy: 0.8494 - loss: 0.4256 - val_categorical_accuracy: 0.7208 -
val_loss: 0.8970
Epoch 14/50
704/704 2s 3ms/step -
categorical_accuracy: 0.8707 - loss: 0.3740 - val_categorical_accuracy: 0.7226 -
val_loss: 0.9292
Epoch 15/50
704/704 3s 3ms/step -
categorical_accuracy: 0.8748 - loss: 0.3589 - val_categorical_accuracy: 0.7420 -
val_loss: 0.9095
Epoch 16/50
704/704 2s 3ms/step -
categorical_accuracy: 0.8875 - loss: 0.3259 - val_categorical_accuracy: 0.7328 -
val_loss: 0.9289
Epoch 17/50
704/704 3s 3ms/step -
categorical_accuracy: 0.8956 - loss: 0.2943 - val_categorical_accuracy: 0.7254 -
val_loss: 1.0243
Epoch 18/50
704/704 3s 4ms/step -
categorical_accuracy: 0.9077 - loss: 0.2659 - val_categorical_accuracy: 0.7258 -
val_loss: 1.0511
Epoch 19/50
704/704 3s 4ms/step -
categorical_accuracy: 0.9141 - loss: 0.2406 - val_categorical_accuracy: 0.7150 -
val_loss: 1.1460
Epoch 20/50
704/704 3s 5ms/step -
categorical_accuracy: 0.9245 - loss: 0.2212 - val_categorical_accuracy: 0.7286 -
val_loss: 1.1770
Epoch 21/50
704/704 5s 4ms/step -
categorical_accuracy: 0.9278 - loss: 0.2022 - val_categorical_accuracy: 0.7220 -
val_loss: 1.1990
Epoch 22/50
```

```
704/704            6s 6ms/step -
categorical_accuracy: 0.9397 - loss: 0.1763 - val_categorical_accuracy: 0.7298 -
val_loss: 1.2162
Epoch 23/50
704/704            4s 4ms/step -
categorical_accuracy: 0.9363 - loss: 0.1811 - val_categorical_accuracy: 0.7224 -
val_loss: 1.2846
Epoch 24/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9449 - loss: 0.1599 - val_categorical_accuracy: 0.7166 -
val_loss: 1.3884
Epoch 25/50
704/704            5s 4ms/step -
categorical_accuracy: 0.9524 - loss: 0.1387 - val_categorical_accuracy: 0.7234 -
val_loss: 1.4143
Epoch 26/50
704/704            2s 3ms/step -
categorical_accuracy: 0.9569 - loss: 0.1270 - val_categorical_accuracy: 0.7150 -
val_loss: 1.5157
Epoch 27/50
704/704            2s 3ms/step -
categorical_accuracy: 0.9531 - loss: 0.1347 - val_categorical_accuracy: 0.7128 -
val_loss: 1.6610
Epoch 28/50
704/704            4s 5ms/step -
categorical_accuracy: 0.9502 - loss: 0.1402 - val_categorical_accuracy: 0.7148 -
val_loss: 1.5986
Epoch 29/50
704/704            4s 4ms/step -
categorical_accuracy: 0.9666 - loss: 0.0973 - val_categorical_accuracy: 0.7074 -
val_loss: 1.6922
Epoch 30/50
704/704            5s 4ms/step -
categorical_accuracy: 0.9657 - loss: 0.0980 - val_categorical_accuracy: 0.7062 -
val_loss: 1.8449
Epoch 31/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9627 - loss: 0.1095 - val_categorical_accuracy: 0.7206 -
val_loss: 1.7585
Epoch 32/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9628 - loss: 0.1022 - val_categorical_accuracy: 0.7064 -
val_loss: 1.9784
Epoch 33/50
704/704            2s 3ms/step -
categorical_accuracy: 0.9620 - loss: 0.1097 - val_categorical_accuracy: 0.7004 -
val_loss: 1.9732
Epoch 34/50
```

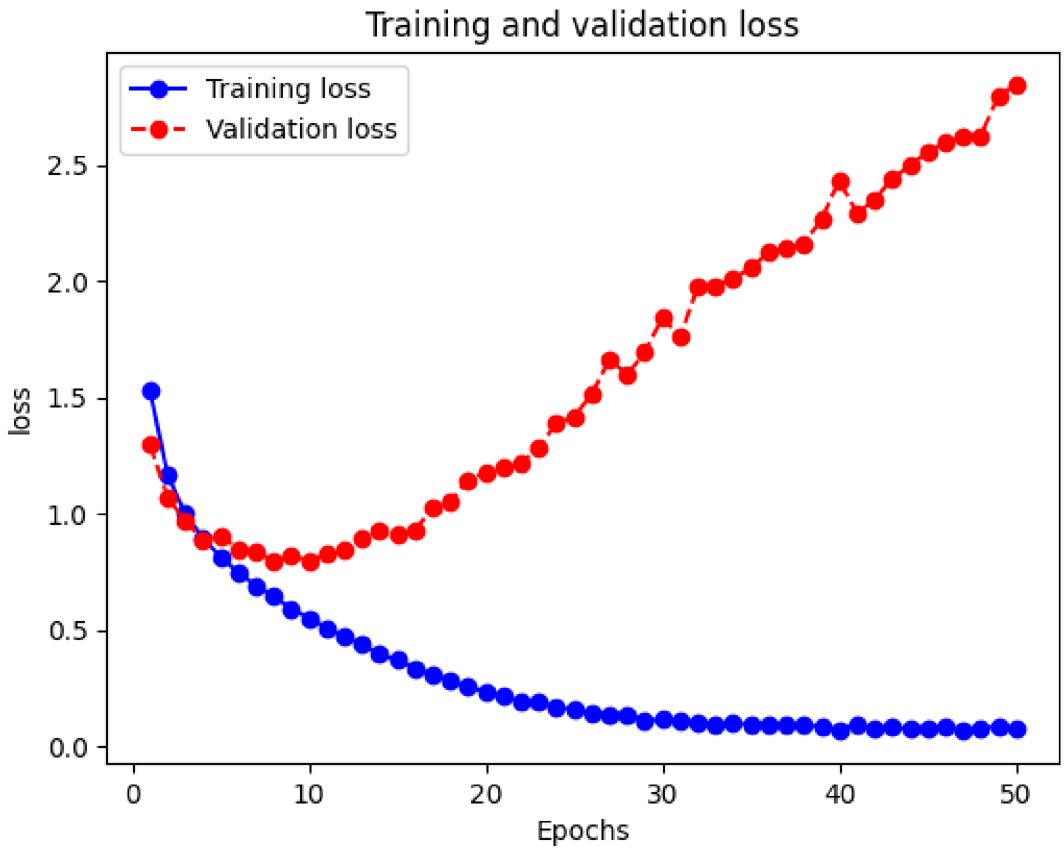
```
704/704            3s 3ms/step -
categorical_accuracy: 0.9699 - loss: 0.0849 - val_categorical_accuracy: 0.7078 -
val_loss: 2.0110
Epoch 35/50
704/704            2s 3ms/step -
categorical_accuracy: 0.9710 - loss: 0.0802 - val_categorical_accuracy: 0.7048 -
val_loss: 2.0546
Epoch 36/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9729 - loss: 0.0808 - val_categorical_accuracy: 0.7046 -
val_loss: 2.1219
Epoch 37/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9736 - loss: 0.0769 - val_categorical_accuracy: 0.7126 -
val_loss: 2.1375
Epoch 38/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9745 - loss: 0.0725 - val_categorical_accuracy: 0.7130 -
val_loss: 2.1593
Epoch 39/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9650 - loss: 0.1029 - val_categorical_accuracy: 0.7050 -
val_loss: 2.2656
Epoch 40/50
704/704            2s 3ms/step -
categorical_accuracy: 0.9785 - loss: 0.0629 - val_categorical_accuracy: 0.6964 -
val_loss: 2.4312
Epoch 41/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9673 - loss: 0.0961 - val_categorical_accuracy: 0.7064 -
val_loss: 2.2911
Epoch 42/50
704/704            5s 4ms/step -
categorical_accuracy: 0.9759 - loss: 0.0664 - val_categorical_accuracy: 0.7048 -
val_loss: 2.3479
Epoch 43/50
704/704            2s 3ms/step -
categorical_accuracy: 0.9730 - loss: 0.0755 - val_categorical_accuracy: 0.6952 -
val_loss: 2.4405
Epoch 44/50
704/704            2s 3ms/step -
categorical_accuracy: 0.9785 - loss: 0.0618 - val_categorical_accuracy: 0.7008 -
val_loss: 2.4926
Epoch 45/50
704/704            3s 4ms/step -
categorical_accuracy: 0.9752 - loss: 0.0699 - val_categorical_accuracy: 0.7102 -
val_loss: 2.5554
Epoch 46/50
```

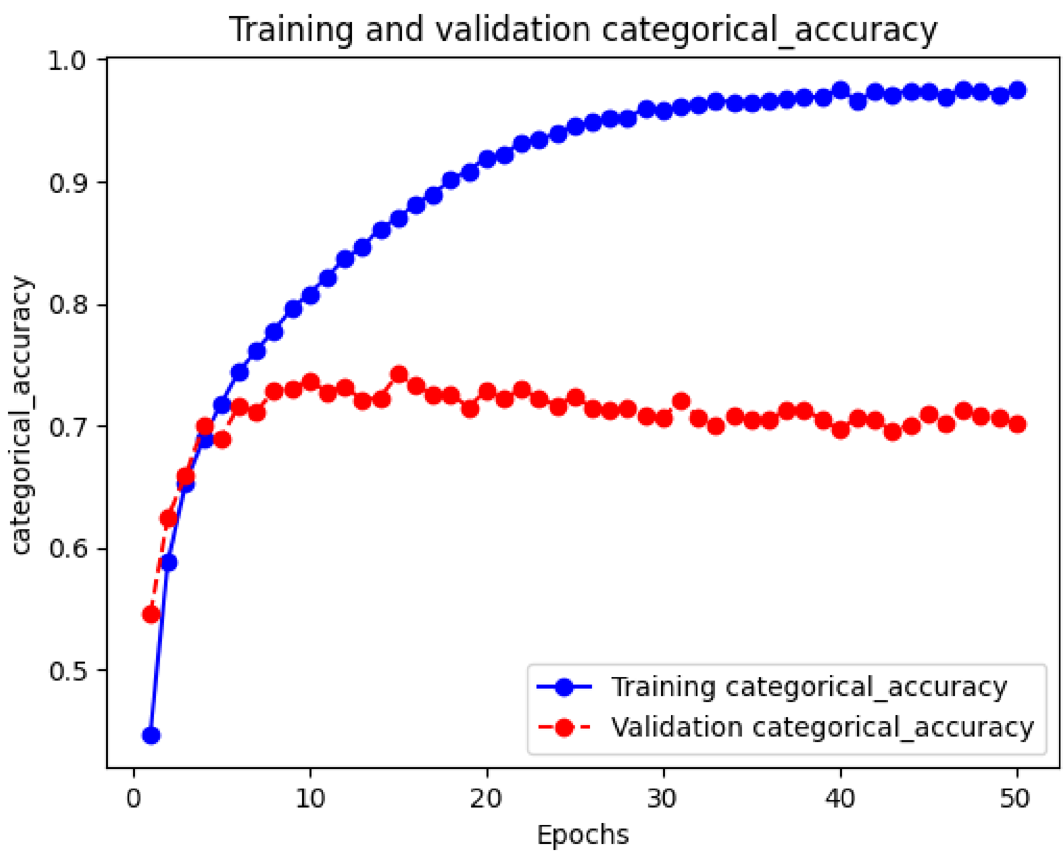
```
704/704             6s 5ms/step -
categorical_accuracy: 0.9671 - loss: 0.0916 - val_categorical_accuracy: 0.7024 -
val_loss: 2.5926
Epoch 47/50
704/704             4s 4ms/step -
categorical_accuracy: 0.9750 - loss: 0.0730 - val_categorical_accuracy: 0.7128 -
val_loss: 2.6185
Epoch 48/50
704/704             5s 4ms/step -
categorical_accuracy: 0.9775 - loss: 0.0621 - val_categorical_accuracy: 0.7080 -
val_loss: 2.6229
Epoch 49/50
704/704             5s 4ms/step -
categorical_accuracy: 0.9707 - loss: 0.0886 - val_categorical_accuracy: 0.7066 -
val_loss: 2.7932
Epoch 50/50
704/704             2s 3ms/step -
categorical_accuracy: 0.9728 - loss: 0.0797 - val_categorical_accuracy: 0.7014 -
val_loss: 2.8433
```

[21]:
```
loss, acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {acc:.3f}")
```

```
313/313             2s 5ms/step -
categorical_accuracy: 0.6865 - loss: 2.9846
Test accuracy: 0.685
```

[22]:
```
plot(history.history['loss'], history.history['val_loss'], 'loss')
plot(history.history['categorical_accuracy'], history.
 ↪history['val_categorical_accuracy'], 'categorical_accuracy')
```


Training and validation loss
Training loss
Validation loss
loss
Epochs
0.0
0.5
1.0
1.5
2.0
2.5
0
10
20
30
40
50



Como podemos observar, la mejoría respecto a la práctica anterior es muy notoria. Cuando utilizábamos redes neuronales comunes obteníamos un `accuracy` cercano a 0,5 como máximo, aquí, con la red convolucional más sencilla ya obtenemos un valor de `accuracy` de un 0,71. Esto evidencia la gran superioridad de las redes convolucionales al tratarse de clasificación de imágenes.

La razón de esto es por la operación de convolución, que permite crear una relación entre un píxel y su entorno, es decir, los píxeles más cercanos a este. Esto es debido a la propiedad de localidad de las imágenes, la cual dice que los píxeles cercanos dentro de una imagen tienden a estar fuertemente correlacionados. Esto, con redes neuronales convencionales sería imposible debido al formato de entrada que tienen, el cual es en vectores unidimensionales, en los que es imposible mapear y hacer relaciones entre píxeles.

En esta práctica no utilizaremos el hiperparámetro de `stride` dado que utilizamos capas de agrupamiento o pooling y no vemos necesario el uso de este hiperparámetro. No solo por esa razón, sino que también

---

En este próximo modelo utilizaremos otro optimizador, el `RMSProp`, el cual es un optimizador muy utilizado en Aprendizaje Profundo.

[ ]:
```python
inputs = keras.Input(shape=(32, 32, 3))

x = layers.Conv2D(filters=10, kernel_size=3,activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=50, kernel_size=3 ,padding = "same",␣
 ↪activation="relu")(x)
x = layers.MaxPooling2D(pool_size=3,padding="same")(x)
x = layers.Conv2D(filters=100, kernel_size=4, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2,padding ="same")(x)
x = layers.Conv2D(filters=300, kernel_size=2,padding="same",␣
 ↪activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2,padding ="same")(x)
x = layers.Conv2D(filters=600, kernel_size=3,padding="same",␣
 ↪activation="relu")(x)

x = layers.Flatten()(x)


outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

#Compile
model.compile(optimizer="RMSProp",
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[keras.metrics.CategoricalAccuracy()])
#Fitting
history = model.fit(x_train, y_train,validation_split = 0.1, epochs=20,␣
 ↪batch_size=64)
```

```
Epoch 1/20
704/704                37s 51ms/step -
categorical_accuracy: 0.2300 - loss: 2.0169 - val_categorical_accuracy: 0.3876 -
val_loss: 1.6728
Epoch 2/20
704/704                38s 53ms/step -
categorical_accuracy: 0.4914 - loss: 1.3875 - val_categorical_accuracy: 0.4124 -
val_loss: 1.8141
Epoch 3/20
704/704                38s 53ms/step -
categorical_accuracy: 0.5881 - loss: 1.1499 - val_categorical_accuracy: 0.5876 -
val_loss: 1.1736
Epoch 4/20
704/704                40s 56ms/step -
categorical_accuracy: 0.6469 - loss: 0.9849 - val_categorical_accuracy: 0.5708 -
val_loss: 1.3029
Epoch 5/20
704/704                41s 58ms/step -
```

```
categorical_accuracy: 0.6933 - loss: 0.8695 - val_categorical_accuracy: 0.5396 -
val_loss: 1.4262
Epoch 6/20
704/704            38s 54ms/step -
categorical_accuracy: 0.7246 - loss: 0.7781 - val_categorical_accuracy: 0.5910 -
val_loss: 1.2973
Epoch 7/20
704/704            38s 54ms/step -
categorical_accuracy: 0.7545 - loss: 0.7093 - val_categorical_accuracy: 0.6726 -
val_loss: 0.9714
Epoch 8/20
704/704            39s 56ms/step -
categorical_accuracy: 0.7783 - loss: 0.6406 - val_categorical_accuracy: 0.5948 -
val_loss: 1.3101
Epoch 9/20
704/704            39s 55ms/step -
categorical_accuracy: 0.7923 - loss: 0.5949 - val_categorical_accuracy: 0.6768 -
val_loss: 1.0347
Epoch 10/20
704/704            39s 56ms/step -
categorical_accuracy: 0.8128 - loss: 0.5380 - val_categorical_accuracy: 0.6828 -
val_loss: 1.0442
Epoch 11/20
704/704            39s 56ms/step -
categorical_accuracy: 0.8309 - loss: 0.4841 - val_categorical_accuracy: 0.6352 -
val_loss: 1.2197
Epoch 12/20
704/704            39s 56ms/step -
categorical_accuracy: 0.8419 - loss: 0.4438 - val_categorical_accuracy: 0.6894 -
val_loss: 1.0927
Epoch 13/20
704/704            40s 57ms/step -
categorical_accuracy: 0.8606 - loss: 0.4016 - val_categorical_accuracy: 0.6744 -
val_loss: 1.2083
Epoch 14/20
704/704            40s 57ms/step -
categorical_accuracy: 0.8684 - loss: 0.3745 - val_categorical_accuracy: 0.6714 -
val_loss: 1.3184
Epoch 15/20
704/704            39s 56ms/step -
categorical_accuracy: 0.8813 - loss: 0.3419 - val_categorical_accuracy: 0.6754 -
val_loss: 1.2812
Epoch 16/20
704/704            40s 57ms/step -
categorical_accuracy: 0.8921 - loss: 0.3070 - val_categorical_accuracy: 0.6702 -
val_loss: 1.4074
Epoch 17/20
704/704            40s 57ms/step -
```

```
categorical_accuracy: 0.8974 - loss: 0.2888 - val_categorical_accuracy: 0.6886 -
val_loss: 1.3504
Epoch 18/20
704/704            40s 56ms/step -
categorical_accuracy: 0.9071 - loss: 0.2653 - val_categorical_accuracy: 0.6750 -
val_loss: 1.6469
Epoch 19/20
704/704            40s 56ms/step -
categorical_accuracy: 0.9104 - loss: 0.2540 - val_categorical_accuracy: 0.6990 -
val_loss: 1.5167
Epoch 20/20
704/704            40s 57ms/step -
categorical_accuracy: 0.9205 - loss: 0.2305 - val_categorical_accuracy: 0.6622 -
val_loss: 1.9304
```

[ ]:
```
loss, acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {acc:.3f}")
```

```
313/313            1s 2ms/step -
categorical_accuracy: 0.6639 - loss: 1.8977
Test accuracy: 0.658
```

Podemos ver claramente que este optimizador no mejora los resultados respecto al optimizador `Adam`, por lo que de aquí en adelante usaremos `Adam` como optimizador por defecto, el cual es el más utilizado en Aprendizaje Profundo.

En este modelo empezaremos a utilizar capas convolucionales seguidas y mantendremos sus dimensiones utilizando padding para reducirlas solamente en las capas de agrupamiento o pooling. Además, utilizaremos una capa de normalización llamada `BatchNormalization`, más concretamente la usaremos durante la inferencia, para que utilice los valores globales de la media y la desviación del entrenamiento entero.

[ ]:
```
inputs = keras.Input(shape=(32, 32, 3))

x = layers.Conv2D(filters = 32, kernel_size = (3,3),padding = "same",activation␣
 ↪= "relu")(inputs)
x = layers.Conv2D(filters = 32, kernel_size = (3,3),padding = "same",activation␣
 ↪= "relu")(x)
x = layers.BatchNormalization()(x)

x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Conv2D(filters=64, kernel_size= (3,3) ,activation="relu",padding␣
 ↪="same")(x)
x = layers.Conv2D(filters=128, kernel_size= (3,3) ,activation="relu",padding␣
 ↪="same")(x)
x = layers.BatchNormalization()(x)
```

```
x = layers.MaxPooling2D(pool_size=2)(x)

x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu")(x)
x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu")(x)
x = layers.BatchNormalization()(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Flatten()(x)

outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

model.summary()
#Compile
model.compile(optimizer="Adam",
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[keras.metrics.CategoricalAccuracy()])
#Callback
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="cnn_cats_dogs_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
#Fitting
history = model.fit(x_train, y_train,validation_split = 0.1 ,epochs=75,␣
↪batch_size=64,callbacks = callbacks)
```

Model: "functional_2"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 32, 32, 3)	0
conv2d_12 (Conv2D)	(None, 32, 32, 32)	896
conv2d_13 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_6 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d_6 (MaxPooling2D)	(None, 16, 16, 32)	0
conv2d_14 (Conv2D)	(None, 16, 16, 64)	18,496

```
 conv2d_15 (Conv2D)                 (None, 16, 16, 128)              73,856

 batch_normalization_7              (None, 16, 16, 128)                 512
 (BatchNormalization)

 max_pooling2d_7 (MaxPooling2D)     (None, 8, 8, 128)                     0

 conv2d_16 (Conv2D)                 (None, 8, 8, 256)               295,168

 conv2d_17 (Conv2D)                 (None, 8, 8, 256)               590,080

 batch_normalization_8              (None, 8, 8, 256)                 1,024
 (BatchNormalization)

 max_pooling2d_8 (MaxPooling2D)     (None, 4, 4, 256)                     0

 flatten_2 (Flatten)                (None, 4096)                          0

 dense_2 (Dense)                    (None, 10)                       40,970


 Total params: 1,030,378 (3.93 MB)

 Trainable params: 1,029,546 (3.93 MB)

 Non-trainable params: 832 (3.25 KB)

Epoch 1/75
704/704              125s 174ms/step -
categorical_accuracy: 0.4316 - loss: 1.8813 - val_categorical_accuracy: 0.5484 -
val_loss: 1.3762
Epoch 2/75
704/704              126s 178ms/step -
categorical_accuracy: 0.6833 - loss: 0.9556 - val_categorical_accuracy: 0.6790 -
val_loss: 1.0453
Epoch 3/75
704/704              128s 181ms/step -
categorical_accuracy: 0.7630 - loss: 0.7063 - val_categorical_accuracy: 0.7400 -
val_loss: 0.7904
Epoch 4/75
704/704              127s 181ms/step -
categorical_accuracy: 0.8256 - loss: 0.5157 - val_categorical_accuracy: 0.7360 -
val_loss: 0.8594
Epoch 5/75
704/704              127s 181ms/step -
categorical_accuracy: 0.8654 - loss: 0.3937 - val_categorical_accuracy: 0.7352 -
```

```
val_loss: 0.9000
Epoch 6/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9091 - loss: 0.2638 - val_categorical_accuracy: 0.7682 -
val_loss: 0.8040
Epoch 7/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9280 - loss: 0.2094 - val_categorical_accuracy: 0.7762 -
val_loss: 0.9414
Epoch 8/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9570 - loss: 0.1270 - val_categorical_accuracy: 0.7534 -
val_loss: 1.0674
Epoch 9/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9499 - loss: 0.1434 - val_categorical_accuracy: 0.7858 -
val_loss: 0.9280
Epoch 10/75
704/704            128s 181ms/step -
categorical_accuracy: 0.9687 - loss: 0.0913 - val_categorical_accuracy: 0.7698 -
val_loss: 1.1695
Epoch 11/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9740 - loss: 0.0740 - val_categorical_accuracy: 0.7832 -
val_loss: 1.1236
Epoch 12/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9701 - loss: 0.0877 - val_categorical_accuracy: 0.7708 -
val_loss: 1.2053
Epoch 13/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9616 - loss: 0.1155 - val_categorical_accuracy: 0.7452 -
val_loss: 1.3373
Epoch 14/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9805 - loss: 0.0625 - val_categorical_accuracy: 0.7818 -
val_loss: 1.2298
Epoch 15/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9803 - loss: 0.0592 - val_categorical_accuracy: 0.7728 -
val_loss: 1.3660
Epoch 16/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9793 - loss: 0.0608 - val_categorical_accuracy: 0.7660 -
val_loss: 1.4462
Epoch 17/75
704/704            130s 184ms/step -
categorical_accuracy: 0.9795 - loss: 0.0624 - val_categorical_accuracy: 0.7678 -
```

```
val_loss: 1.4179
Epoch 18/75
704/704              130s 184ms/step -
categorical_accuracy: 0.9780 - loss: 0.0650 - val_categorical_accuracy: 0.8024 -
val_loss: 1.1011
Epoch 19/75
704/704              129s 183ms/step -
categorical_accuracy: 0.9864 - loss: 0.0387 - val_categorical_accuracy: 0.7824 -
val_loss: 1.3974
Epoch 20/75
704/704              129s 184ms/step -
categorical_accuracy: 0.9785 - loss: 0.0679 - val_categorical_accuracy: 0.7912 -
val_loss: 1.3021
Epoch 21/75
704/704              128s 182ms/step -
categorical_accuracy: 0.9879 - loss: 0.0372 - val_categorical_accuracy: 0.7974 -
val_loss: 1.3110
Epoch 22/75
704/704              129s 183ms/step -
categorical_accuracy: 0.9827 - loss: 0.0502 - val_categorical_accuracy: 0.7804 -
val_loss: 1.4498
Epoch 23/75
704/704              130s 185ms/step -
categorical_accuracy: 0.9844 - loss: 0.0484 - val_categorical_accuracy: 0.7962 -
val_loss: 1.4142
Epoch 24/75
704/704              131s 186ms/step -
categorical_accuracy: 0.9896 - loss: 0.0295 - val_categorical_accuracy: 0.7936 -
val_loss: 1.4153
Epoch 25/75
704/704              130s 184ms/step -
categorical_accuracy: 0.9767 - loss: 0.0767 - val_categorical_accuracy: 0.7986 -
val_loss: 1.2916
Epoch 26/75
704/704              130s 185ms/step -
categorical_accuracy: 0.9914 - loss: 0.0249 - val_categorical_accuracy: 0.7902 -
val_loss: 1.3212
Epoch 27/75
704/704              130s 185ms/step -
categorical_accuracy: 0.9924 - loss: 0.0240 - val_categorical_accuracy: 0.7776 -
val_loss: 1.4979
Epoch 28/75
704/704              129s 183ms/step -
categorical_accuracy: 0.9854 - loss: 0.0444 - val_categorical_accuracy: 0.7828 -
val_loss: 1.5059
Epoch 29/75
704/704              130s 184ms/step -
categorical_accuracy: 0.9904 - loss: 0.0274 - val_categorical_accuracy: 0.7904 -
```

```
val_loss: 1.6057
Epoch 30/75
704/704            129s 184ms/step -
categorical_accuracy: 0.9894 - loss: 0.0312 - val_categorical_accuracy: 0.8092 -
val_loss: 1.3767
Epoch 31/75
704/704            130s 184ms/step -
categorical_accuracy: 0.9905 - loss: 0.0303 - val_categorical_accuracy: 0.7830 -
val_loss: 1.6983
Epoch 32/75
704/704            130s 185ms/step -
categorical_accuracy: 0.9913 - loss: 0.0268 - val_categorical_accuracy: 0.7652 -
val_loss: 1.8909
Epoch 33/75
704/704            130s 184ms/step -
categorical_accuracy: 0.9889 - loss: 0.0364 - val_categorical_accuracy: 0.7890 -
val_loss: 1.6302
Epoch 34/75
704/704            129s 184ms/step -
categorical_accuracy: 0.9911 - loss: 0.0272 - val_categorical_accuracy: 0.7932 -
val_loss: 1.6851
Epoch 35/75
704/704            130s 184ms/step -
categorical_accuracy: 0.9906 - loss: 0.0265 - val_categorical_accuracy: 0.7850 -
val_loss: 1.5474
Epoch 36/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9906 - loss: 0.0287 - val_categorical_accuracy: 0.8068 -
val_loss: 1.3854
Epoch 37/75
704/704            130s 185ms/step -
categorical_accuracy: 0.9898 - loss: 0.0313 - val_categorical_accuracy: 0.8024 -
val_loss: 1.4354
Epoch 38/75
704/704            130s 184ms/step -
categorical_accuracy: 0.9932 - loss: 0.0200 - val_categorical_accuracy: 0.7864 -
val_loss: 1.6431
Epoch 39/75
704/704            130s 185ms/step -
categorical_accuracy: 0.9926 - loss: 0.0252 - val_categorical_accuracy: 0.8084 -
val_loss: 1.4143
Epoch 40/75
704/704            129s 184ms/step -
categorical_accuracy: 0.9913 - loss: 0.0272 - val_categorical_accuracy: 0.8040 -
val_loss: 1.5559
Epoch 41/75
704/704            130s 185ms/step -
categorical_accuracy: 0.9913 - loss: 0.0270 - val_categorical_accuracy: 0.7964 -
```

```
val_loss: 1.5156
Epoch 42/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9882 - loss: 0.0424 - val_categorical_accuracy: 0.7972 -
val_loss: 1.3653
Epoch 43/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9942 - loss: 0.0163 - val_categorical_accuracy: 0.8048 -
val_loss: 1.4282
Epoch 44/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9955 - loss: 0.0143 - val_categorical_accuracy: 0.8022 -
val_loss: 1.5556
Epoch 45/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9922 - loss: 0.0251 - val_categorical_accuracy: 0.7994 -
val_loss: 1.5141
Epoch 46/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9861 - loss: 0.0468 - val_categorical_accuracy: 0.8018 -
val_loss: 1.4373
Epoch 47/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9967 - loss: 0.0106 - val_categorical_accuracy: 0.7978 -
val_loss: 1.6339
Epoch 48/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9906 - loss: 0.0295 - val_categorical_accuracy: 0.8144 -
val_loss: 1.5844
Epoch 49/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9919 - loss: 0.0253 - val_categorical_accuracy: 0.8040 -
val_loss: 1.5532
Epoch 50/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9973 - loss: 0.0081 - val_categorical_accuracy: 0.8118 -
val_loss: 1.4998
Epoch 51/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9952 - loss: 0.0149 - val_categorical_accuracy: 0.8088 -
val_loss: 1.5018
Epoch 52/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9911 - loss: 0.0282 - val_categorical_accuracy: 0.7880 -
val_loss: 1.7499
Epoch 53/75
704/704            128s 182ms/step -
```

```
categorical_accuracy: 0.9950 - loss: 0.0160 - val_categorical_accuracy: 0.8158 -
val_loss: 1.5484
Epoch 54/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9951 - loss: 0.0144 - val_categorical_accuracy: 0.7946 -
val_loss: 1.7193
Epoch 55/75
704/704            133s 170ms/step -
categorical_accuracy: 0.9949 - loss: 0.0146 - val_categorical_accuracy: 0.7990 -
val_loss: 1.6040
Epoch 56/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9955 - loss: 0.0143 - val_categorical_accuracy: 0.8070 -
val_loss: 1.6678
Epoch 57/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9938 - loss: 0.0193 - val_categorical_accuracy: 0.7994 -
val_loss: 1.6171
Epoch 58/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9943 - loss: 0.0168 - val_categorical_accuracy: 0.7874 -
val_loss: 1.7202
Epoch 59/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9926 - loss: 0.0232 - val_categorical_accuracy: 0.8126 -
val_loss: 1.5957
Epoch 60/75
704/704            129s 184ms/step -
categorical_accuracy: 0.9949 - loss: 0.0164 - val_categorical_accuracy: 0.8180 -
val_loss: 1.5318
Epoch 61/75
704/704            129s 184ms/step -
categorical_accuracy: 0.9966 - loss: 0.0096 - val_categorical_accuracy: 0.7962 -
val_loss: 1.8141
Epoch 62/75
704/704            129s 183ms/step -
categorical_accuracy: 0.9954 - loss: 0.0127 - val_categorical_accuracy: 0.7980 -
val_loss: 1.7050
Epoch 63/75
704/704            128s 182ms/step -
categorical_accuracy: 0.9945 - loss: 0.0181 - val_categorical_accuracy: 0.7968 -
val_loss: 1.7307
Epoch 64/75
704/704            128s 181ms/step -
categorical_accuracy: 0.9951 - loss: 0.0168 - val_categorical_accuracy: 0.8034 -
val_loss: 1.5902
Epoch 65/75
704/704            128s 181ms/step -
```

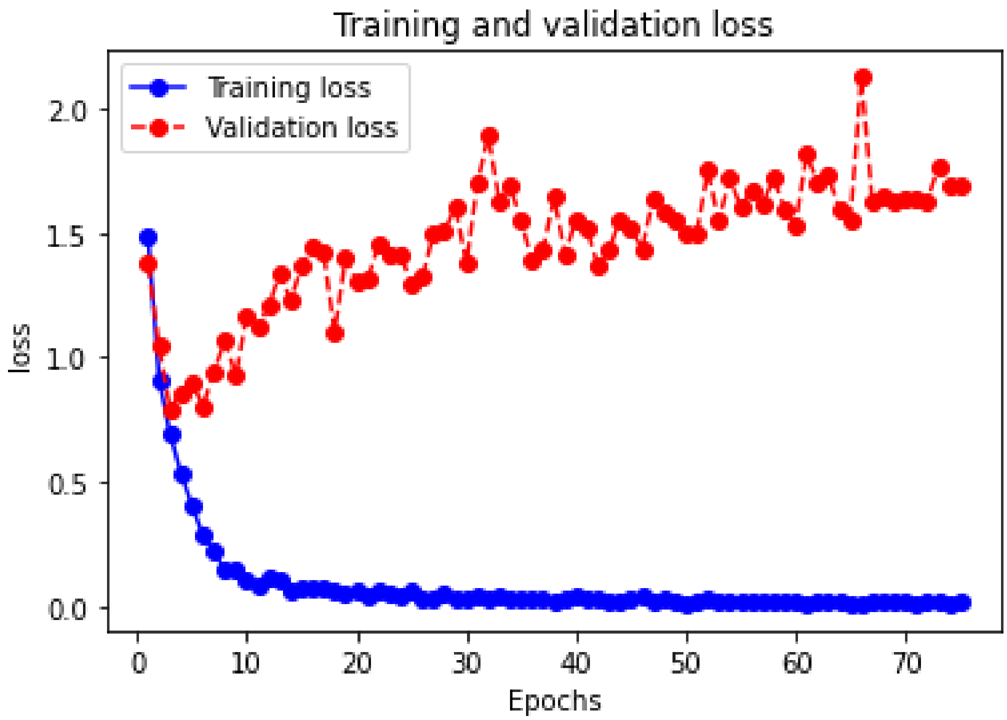
```
categorical_accuracy: 0.9969 - loss: 0.0097 - val_categorical_accuracy: 0.8120 -
val_loss: 1.5536
Epoch 66/75
704/704              128s 181ms/step -
categorical_accuracy: 0.9973 - loss: 0.0087 - val_categorical_accuracy: 0.7660 -
val_loss: 2.1273
Epoch 67/75
704/704              127s 181ms/step -
categorical_accuracy: 0.9943 - loss: 0.0178 - val_categorical_accuracy: 0.7992 -
val_loss: 1.6266
Epoch 68/75
704/704              128s 182ms/step -
categorical_accuracy: 0.9945 - loss: 0.0167 - val_categorical_accuracy: 0.8026 -
val_loss: 1.6524
Epoch 69/75
704/704              128s 181ms/step -
categorical_accuracy: 0.9957 - loss: 0.0127 - val_categorical_accuracy: 0.8056 -
val_loss: 1.6287
Epoch 70/75
704/704              128s 182ms/step -
categorical_accuracy: 0.9938 - loss: 0.0209 - val_categorical_accuracy: 0.8010 -
val_loss: 1.6372
Epoch 71/75
704/704              128s 182ms/step -
categorical_accuracy: 0.9964 - loss: 0.0102 - val_categorical_accuracy: 0.8110 -
val_loss: 1.6321
Epoch 72/75
704/704              128s 181ms/step -
categorical_accuracy: 0.9963 - loss: 0.0120 - val_categorical_accuracy: 0.8154 -
val_loss: 1.6298
Epoch 73/75
704/704              128s 182ms/step -
categorical_accuracy: 0.9919 - loss: 0.0261 - val_categorical_accuracy: 0.8108 -
val_loss: 1.7672
Epoch 74/75
704/704              128s 182ms/step -
categorical_accuracy: 0.9968 - loss: 0.0096 - val_categorical_accuracy: 0.8066 -
val_loss: 1.6903
Epoch 75/75
704/704              128s 181ms/step -
categorical_accuracy: 0.9959 - loss: 0.0134 - val_categorical_accuracy: 0.8096 -
val_loss: 1.6919
```

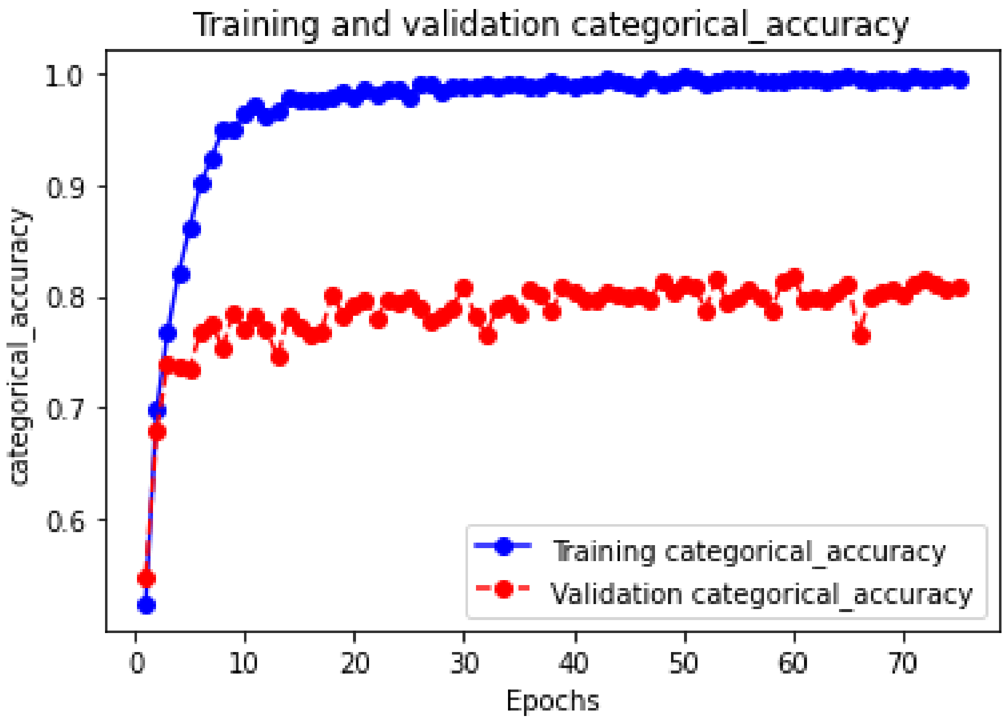
[ ]:
```
loss, acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {acc:.3f}")
```

```
313/313              8s 26ms/step -
categorical_accuracy: 0.7971 - loss: 1.7594
```

```
Test accuracy: 0.794
```

[ ]:
```
plot(history.history['loss'], history.history['val_loss'], 'loss')
plot(history.history['categorical_accuracy'], history.
 ↪history['val_categorical_accuracy'], 'categorical_accuracy')
```





Podemos ver claramente que en el modelo obtenido se produce **overfitting** con unos valores de epochs muy bajos. Esto se puede deber a la poca regularización aplicada, siendo esta una normalización de los valores cada dos capas convolucionales. Por esta razón, a partir de ahora haremos una regularización más fuerte y aplicaremos `Early Stopping` de manera que no se pierda tiempo una vez se sepa que el modelo está sobreentrenado.

A continuación, probaremos con un modelo como el anterior pero en el cual utilizaremos regularizaciones en algunas capas en vez de utilizar capas `BatchNormalization`. Además, veremos qué ocurre si añadimos otra capa densa con mayor número de neuronas antes de la capa densa final con 10 neuronas.

Para poder utilizar regularizadores como L1 o L2, tendremos que importar el módulo `regularizers` de la librería `keras`.

[23]:
```
from keras import regularizers
```

[23]:
```
inputs = keras.Input(shape=(32, 32, 3))

x = layers.Conv2D(filters = 32, kernel_size = (3,3),padding = "same",activation␣
↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(inputs)
x = layers.Conv2D(filters = 32, kernel_size = (3,3),padding = "same",activation␣
↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(x)

x = layers.MaxPooling2D(pool_size=2)(x)
```

```
x = layers.Conv2D(filters=64, kernel_size= (3,3) ,activation="relu",padding␣
↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.Conv2D(filters=128, kernel_size= (3,3) ,activation="relu",padding␣
↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)

x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Flatten()(x)
x = layers.Dense(100,activation = "relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

model.summary()
#Compile
model.compile(optimizer="Adam",
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[keras.metrics.CategoricalAccuracy()])
#Callback
callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_loss",
        patience=5,
        start_from_epoch=5)
]
#Fitting
history = model.fit(x_train, y_train,validation_split = 0.1 ,epochs=50,␣
↪batch_size=64,callbacks = callbacks)
```

Model: "functional_11"

```
 Layer (type)                          Output Shape                       ␣
↪Param #

 input_layer_11 (InputLayer)           (None, 32, 32, 3)                     ␣
↪   0

 conv2d_66 (Conv2D)                    (None, 32, 32, 32)                    ␣
↪896
```

```
 conv2d_67 (Conv2D)                    (None, 32, 32, 32)                  ␣
↪9,248

 max_pooling2d_38 (MaxPooling2D)       (None, 16, 16, 32)                    ␣
↪  0

 conv2d_68 (Conv2D)                    (None, 16, 16, 64)                 ␣
↪18,496

 conv2d_69 (Conv2D)                    (None, 16, 16, 128)                ␣
↪73,856

 max_pooling2d_39 (MaxPooling2D)       (None, 8, 8, 128)                     ␣
↪  0

 conv2d_70 (Conv2D)                    (None, 8, 8, 256)                 ␣
↪295,168

 conv2d_71 (Conv2D)                    (None, 8, 8, 256)                 ␣
↪590,080

 max_pooling2d_40 (MaxPooling2D)       (None, 4, 4, 256)                     ␣
↪  0

 flatten_10 (Flatten)                  (None, 4096)                          ␣
↪  0

 dense_28 (Dense)                      (None, 100)                       ␣
↪409,700

 dense_29 (Dense)                      (None, 10)                          ␣
↪1,010


 Total params: 1,398,454 (5.33 MB)

 Trainable params: 1,398,454 (5.33 MB)

 Non-trainable params: 0 (0.00 B)

Epoch 1/50
704/704             18s 18ms/step -
categorical_accuracy: 0.2781 - loss: 2.0509 - val_categorical_accuracy: 0.5060 -
val_loss: 1.4418
Epoch 2/50
```

```
704/704            12s 11ms/step -
categorical_accuracy: 0.5128 - loss: 1.4191 - val_categorical_accuracy: 0.6000 -
val_loss: 1.2189
Epoch 3/50
704/704            8s 11ms/step -
categorical_accuracy: 0.5971 - loss: 1.2245 - val_categorical_accuracy: 0.6378 -
val_loss: 1.1299
Epoch 4/50
704/704            10s 11ms/step -
categorical_accuracy: 0.6415 - loss: 1.1313 - val_categorical_accuracy: 0.6664 -
val_loss: 1.0745
Epoch 5/50
704/704            10s 11ms/step -
categorical_accuracy: 0.6780 - loss: 1.0406 - val_categorical_accuracy: 0.7050 -
val_loss: 0.9859
Epoch 6/50
704/704            8s 11ms/step -
categorical_accuracy: 0.7077 - loss: 0.9672 - val_categorical_accuracy: 0.7284 -
val_loss: 0.9333
Epoch 7/50
704/704            10s 11ms/step -
categorical_accuracy: 0.7347 - loss: 0.9092 - val_categorical_accuracy: 0.7230 -
val_loss: 0.9591
Epoch 8/50
704/704            9s 12ms/step -
categorical_accuracy: 0.7478 - loss: 0.8782 - val_categorical_accuracy: 0.7438 -
val_loss: 0.9065
Epoch 9/50
704/704            8s 12ms/step -
categorical_accuracy: 0.7627 - loss: 0.8340 - val_categorical_accuracy: 0.7612 -
val_loss: 0.8701
Epoch 10/50
704/704            13s 15ms/step -
categorical_accuracy: 0.7833 - loss: 0.7916 - val_categorical_accuracy: 0.7514 -
val_loss: 0.8929
Epoch 11/50
704/704            8s 11ms/step -
categorical_accuracy: 0.7950 - loss: 0.7564 - val_categorical_accuracy: 0.7644 -
val_loss: 0.8731
Epoch 12/50
704/704            7s 11ms/step -
categorical_accuracy: 0.8059 - loss: 0.7378 - val_categorical_accuracy: 0.7814 -
val_loss: 0.8359
Epoch 13/50
704/704            8s 11ms/step -
categorical_accuracy: 0.8160 - loss: 0.7125 - val_categorical_accuracy: 0.7698 -
val_loss: 0.8654
Epoch 14/50
```

```
704/704 8s 11ms/step -
categorical_accuracy: 0.8252 - loss: 0.6801 - val_categorical_accuracy: 0.7630 -
val_loss: 0.9037
Epoch 15/50
704/704 10s 11ms/step -
categorical_accuracy: 0.8403 - loss: 0.6480 - val_categorical_accuracy: 0.7654 -
val_loss: 0.9159
Epoch 16/50
704/704 9s 13ms/step -
categorical_accuracy: 0.8361 - loss: 0.6591 - val_categorical_accuracy: 0.7822 -
val_loss: 0.8552
Epoch 17/50
704/704 9s 11ms/step -
categorical_accuracy: 0.8526 - loss: 0.6195 - val_categorical_accuracy: 0.7554 -
val_loss: 0.9440
```

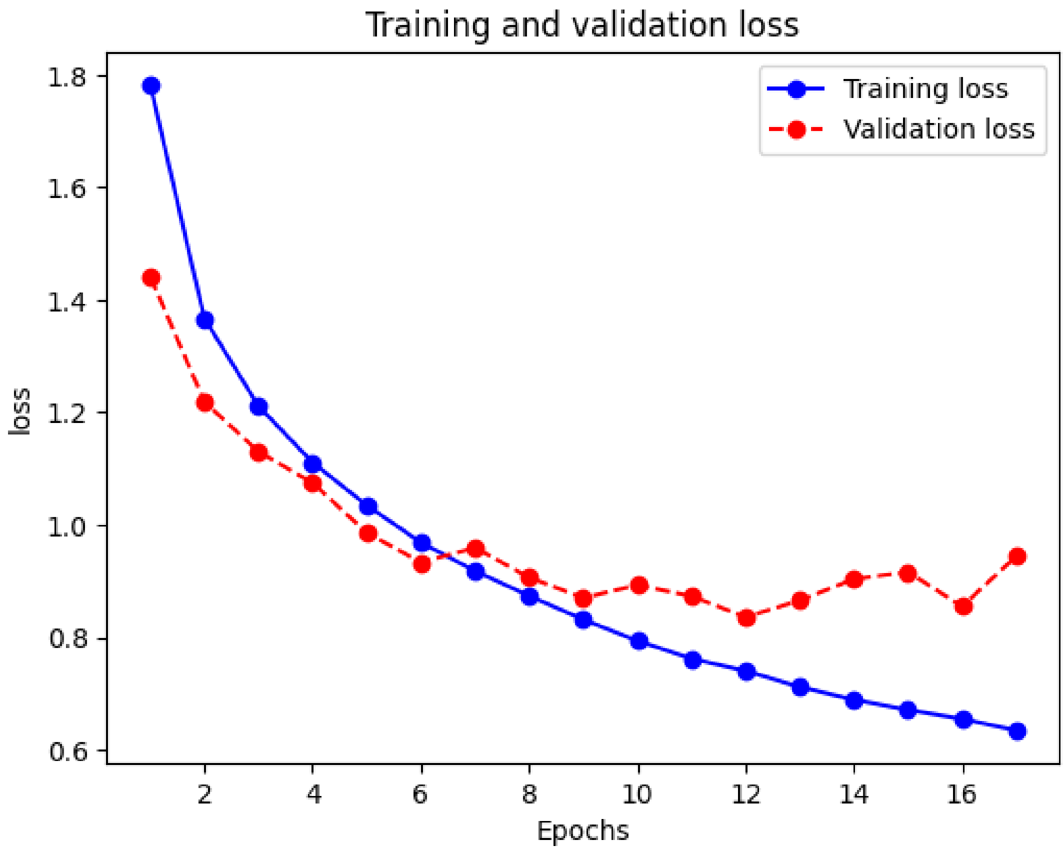
[24]:

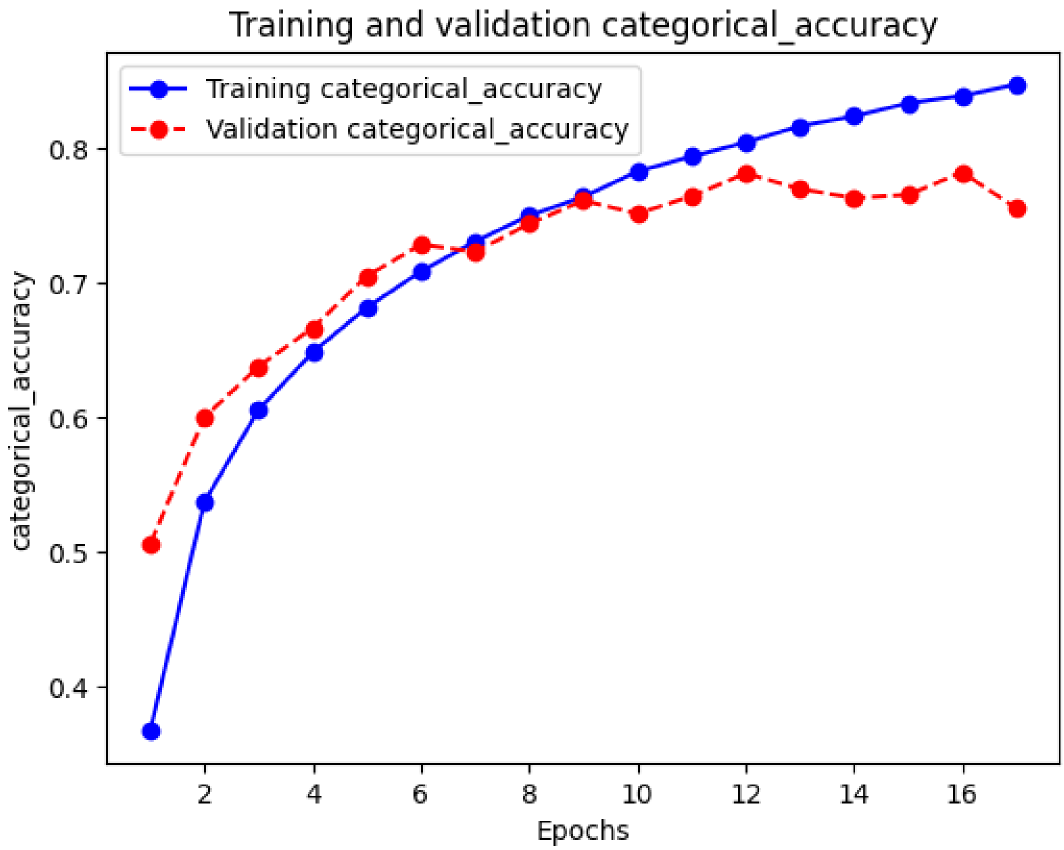
```
loss, acc = model.evaluate(x_test, y_test)
print(f"Test accuracy: {acc:.3f}")
```

```
313/313 3s 6ms/step -
categorical_accuracy: 0.7362 - loss: 0.9980
Test accuracy: 0.743
```

[25]:

```
plot(history.history['loss'], history.history['val_loss'], 'loss')
plot(history.history['categorical_accuracy'], history.
 ↪history['val_categorical_accuracy'], 'categorical_accuracy')
```


Training and validation loss
Training loss
Validation loss
loss
Epochs
1.8
1.6
1.4
1.2
1.0
0.8
0.6
2
4
6
8
10
12
14
16



Se puede notar cierta mejoría a la hora del overfitting pero tambíen se sabe que no es suficiente. Por lo tanto, aplicaremos otro método de regularización para ver si podemos solventar ete problema.

En esta añadiremos otro método de regularización, como es el `Dropout`, esto ayudará a reducir el overfitting que podemos ver en el modelo anterior. Además, para intentar aumentar la precisión, añadiremos una capa densa más, cambiaremos el número de filtros y cambiaremos alguna capa.

[ ]:
```
inputs = keras.Input(shape=(32, 32, 3))

x = layers.Conv2D(filters = 32, kernel_size = (3,3),padding = "same",activation␣
↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(inputs)
x = layers.Conv2D(filters = 32, kernel_size = (3,3),padding = "same",activation␣
↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(x)

x = layers.MaxPooling2D(pool_size=2)(x)

x = layers.Conv2D(filters=64, kernel_size= (3,3) ,activation="relu",padding␣
↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
```

```
x = layers.Conv2D(filters=128, kernel_size= (3,3) ,activation="relu",padding␣
↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.05)(x)

x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.05)(x)

x = layers.Flatten()(x)
x = layers.Dense(500,activation = "relu")(x)
x = layers.Dense(100,activation = "relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)
model_11 = keras.Model(inputs=inputs, outputs=outputs)

model_11.summary()
#Compile
model_11.compile(optimizer="Adam",
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[keras.metrics.CategoricalAccuracy()])
#Callback
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="cnn_cats_dogs_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
#Fitting
history_11 = model_11.fit(x_train, y_train,validation_split = 0.1 ,epochs=75,␣
↪batch_size=128,callbacks = callbacks)
```

Model: "functional_13"

Layer (type)	Output Shape	Param #
input_layer_14 (InputLayer)	(None, 32, 32, 3)	0
conv2d_72 (Conv2D)	(None, 32, 32, 32)	896
conv2d_73 (Conv2D)	(None, 32, 32, 32)	9,248
max_pooling2d_44 (MaxPooling2D)	(None, 16, 16, 32)	0

conv2d_74 (Conv2D)	(None, 16, 16, 64)	18,496
conv2d_75 (Conv2D)	(None, 16, 16, 128)	73,856
max_pooling2d_45 (MaxPooling2D)	(None, 8, 8, 128)	0
dropout_43 (Dropout)	(None, 8, 8, 128)	0
conv2d_76 (Conv2D)	(None, 8, 8, 256)	295,168
conv2d_77 (Conv2D)	(None, 8, 8, 256)	590,080
max_pooling2d_46 (MaxPooling2D)	(None, 4, 4, 256)	0
dropout_44 (Dropout)	(None, 4, 4, 256)	0
flatten_13 (Flatten)	(None, 4096)	0
dense_47 (Dense)	(None, 500)	2,048,500
dense_48 (Dense)	(None, 100)	50,100
dense_49 (Dense)	(None, 10)	1,010

Total params: 3,087,354 (11.78 MB)

Trainable params: 3,087,354 (11.78 MB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/75
704/704             118s 166ms/step -
categorical_accuracy: 0.2670 - loss: 2.0711 - val_categorical_accuracy: 0.4848 -
val_loss: 1.4467
Epoch 2/75
704/704             119s 168ms/step -
categorical_accuracy: 0.5128 - loss: 1.4008 - val_categorical_accuracy: 0.5460 -
val_loss: 1.3274
Epoch 3/75
704/704             119s 169ms/step -
categorical_accuracy: 0.5886 - loss: 1.2253 - val_categorical_accuracy: 0.6504 -
val_loss: 1.0600
Epoch 4/75
704/704             114s 162ms/step -
categorical_accuracy: 0.6498 - loss: 1.0854 - val_categorical_accuracy: 0.6808 -
```

```
val_loss: 1.0097
Epoch 5/75
704/704            119s 169ms/step -
categorical_accuracy: 0.6905 - loss: 0.9803 - val_categorical_accuracy: 0.6910 -
val_loss: 0.9940
Epoch 6/75
704/704            115s 163ms/step -
categorical_accuracy: 0.7209 - loss: 0.9075 - val_categorical_accuracy: 0.7234 -
val_loss: 0.9088
Epoch 7/75
704/704            113s 160ms/step -
categorical_accuracy: 0.7483 - loss: 0.8374 - val_categorical_accuracy: 0.7374 -
val_loss: 0.8781
Epoch 8/75
704/704            114s 161ms/step -
categorical_accuracy: 0.7644 - loss: 0.7934 - val_categorical_accuracy: 0.7364 -
val_loss: 0.9063
Epoch 9/75
704/704            113s 160ms/step -
categorical_accuracy: 0.7800 - loss: 0.7590 - val_categorical_accuracy: 0.7414 -
val_loss: 0.8818
Epoch 10/75
704/704            113s 160ms/step -
categorical_accuracy: 0.7968 - loss: 0.7096 - val_categorical_accuracy: 0.7470 -
val_loss: 0.8834
Epoch 11/75
704/704            114s 161ms/step -
categorical_accuracy: 0.8094 - loss: 0.6739 - val_categorical_accuracy: 0.7740 -
val_loss: 0.7937
Epoch 12/75
704/704            114s 162ms/step -
categorical_accuracy: 0.8254 - loss: 0.6291 - val_categorical_accuracy: 0.7512 -
val_loss: 0.8953
Epoch 13/75
704/704            115s 163ms/step -
categorical_accuracy: 0.8332 - loss: 0.6147 - val_categorical_accuracy: 0.7710 -
val_loss: 0.8607
Epoch 14/75
704/704            116s 164ms/step -
categorical_accuracy: 0.8433 - loss: 0.5932 - val_categorical_accuracy: 0.7842 -
val_loss: 0.8315
Epoch 15/75
704/704            119s 169ms/step -
categorical_accuracy: 0.8627 - loss: 0.5470 - val_categorical_accuracy: 0.7686 -
val_loss: 0.8920
Epoch 16/75
704/704            113s 161ms/step -
categorical_accuracy: 0.8634 - loss: 0.5405 - val_categorical_accuracy: 0.7688 -
```

```
val_loss: 0.8839
Epoch 17/75
704/704              113s 161ms/step -
categorical_accuracy: 0.8751 - loss: 0.5139 - val_categorical_accuracy: 0.7782 -
val_loss: 0.8629
Epoch 18/75
704/704              115s 163ms/step -
categorical_accuracy: 0.8802 - loss: 0.4936 - val_categorical_accuracy: 0.7842 -
val_loss: 0.8421
Epoch 19/75
704/704              112s 159ms/step -
categorical_accuracy: 0.8871 - loss: 0.4785 - val_categorical_accuracy: 0.7784 -
val_loss: 0.8849
Epoch 20/75
704/704              112s 159ms/step -
categorical_accuracy: 0.8943 - loss: 0.4628 - val_categorical_accuracy: 0.7862 -
val_loss: 0.8768
Epoch 21/75
704/704              126s 179ms/step -
categorical_accuracy: 0.9022 - loss: 0.4395 - val_categorical_accuracy: 0.7714 -
val_loss: 0.9247
Epoch 22/75
704/704              1446s 2s/step -
categorical_accuracy: 0.8997 - loss: 0.4424 - val_categorical_accuracy: 0.7658 -
val_loss: 0.9864
Epoch 23/75
704/704              161s 228ms/step -
categorical_accuracy: 0.9174 - loss: 0.4053 - val_categorical_accuracy: 0.7654 -
val_loss: 0.9878
Epoch 24/75
704/704              109s 155ms/step -
categorical_accuracy: 0.9153 - loss: 0.4037 - val_categorical_accuracy: 0.7762 -
val_loss: 0.9416
Epoch 25/75
704/704              109s 154ms/step -
categorical_accuracy: 0.9233 - loss: 0.3839 - val_categorical_accuracy: 0.7820 -
val_loss: 0.9789
Epoch 26/75
704/704              108s 153ms/step -
categorical_accuracy: 0.9245 - loss: 0.3810 - val_categorical_accuracy: 0.7710 -
val_loss: 1.0639
Epoch 27/75
704/704              105s 149ms/step -
categorical_accuracy: 0.9262 - loss: 0.3818 - val_categorical_accuracy: 0.7796 -
val_loss: 1.0406
Epoch 28/75
704/704              110s 156ms/step -
categorical_accuracy: 0.9345 - loss: 0.3555 - val_categorical_accuracy: 0.7730 -
```

```
val_loss: 1.0077
Epoch 29/75
704/704            107s 153ms/step -
categorical_accuracy: 0.9344 - loss: 0.3546 - val_categorical_accuracy: 0.7738 -
val_loss: 1.0237
Epoch 30/75
704/704            108s 153ms/step -
categorical_accuracy: 0.9358 - loss: 0.3497 - val_categorical_accuracy: 0.7726 -
val_loss: 1.0515
Epoch 31/75
704/704            107s 152ms/step -
categorical_accuracy: 0.9446 - loss: 0.3245 - val_categorical_accuracy: 0.7794 -
val_loss: 1.0565
Epoch 32/75
704/704            109s 155ms/step -
categorical_accuracy: 0.9411 - loss: 0.3355 - val_categorical_accuracy: 0.7772 -
val_loss: 1.1106
Epoch 33/75
704/704            108s 153ms/step -
categorical_accuracy: 0.9399 - loss: 0.3354 - val_categorical_accuracy: 0.7672 -
val_loss: 1.0724
Epoch 34/75
704/704            108s 154ms/step -
categorical_accuracy: 0.9461 - loss: 0.3213 - val_categorical_accuracy: 0.7734 -
val_loss: 1.0695
Epoch 35/75
704/704            110s 156ms/step -
categorical_accuracy: 0.9468 - loss: 0.3203 - val_categorical_accuracy: 0.7752 -
val_loss: 1.0932
Epoch 36/75
704/704            109s 154ms/step -
categorical_accuracy: 0.9502 - loss: 0.3092 - val_categorical_accuracy: 0.7764 -
val_loss: 1.1343
Epoch 37/75
704/704            110s 157ms/step -
categorical_accuracy: 0.9487 - loss: 0.3134 - val_categorical_accuracy: 0.7776 -
val_loss: 1.1152
Epoch 38/75
704/704            105s 149ms/step -
categorical_accuracy: 0.9396 - loss: 0.3395 - val_categorical_accuracy: 0.7640 -
val_loss: 1.1840
Epoch 39/75
704/704            105s 149ms/step -
categorical_accuracy: 0.9467 - loss: 0.3147 - val_categorical_accuracy: 0.7670 -
val_loss: 1.1404
Epoch 40/75
704/704            106s 150ms/step -
categorical_accuracy: 0.9441 - loss: 0.3302 - val_categorical_accuracy: 0.7766 -
```

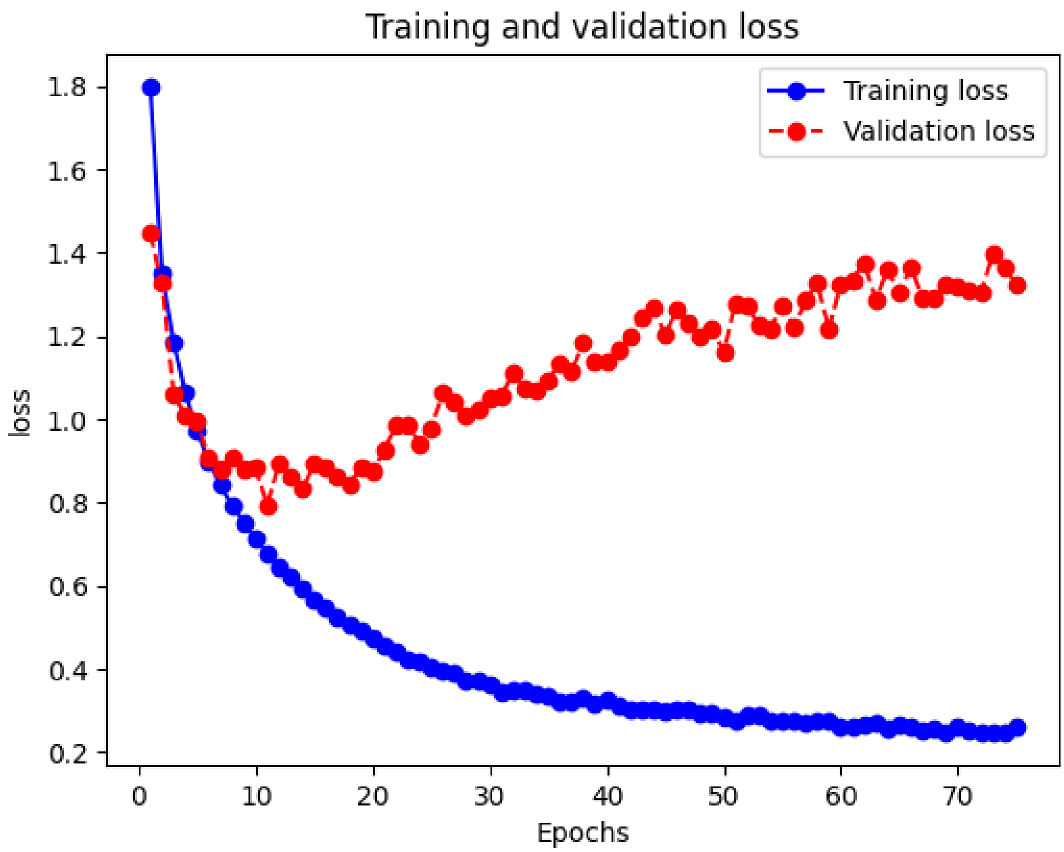
```
val_loss: 1.1392
Epoch 41/75
704/704             104s 148ms/step -
categorical_accuracy: 0.9560 - loss: 0.2957 - val_categorical_accuracy: 0.7682 -
val_loss: 1.1645
Epoch 42/75
704/704             105s 149ms/step -
categorical_accuracy: 0.9533 - loss: 0.2964 - val_categorical_accuracy: 0.7660 -
val_loss: 1.1986
Epoch 43/75
704/704             104s 147ms/step -
categorical_accuracy: 0.9508 - loss: 0.3011 - val_categorical_accuracy: 0.7590 -
val_loss: 1.2436
Epoch 44/75
704/704             105s 149ms/step -
categorical_accuracy: 0.9547 - loss: 0.2940 - val_categorical_accuracy: 0.7592 -
val_loss: 1.2679
Epoch 45/75
704/704             106s 150ms/step -
categorical_accuracy: 0.9568 - loss: 0.2883 - val_categorical_accuracy: 0.7698 -
val_loss: 1.2021
Epoch 46/75
704/704             113s 161ms/step -
categorical_accuracy: 0.9539 - loss: 0.2906 - val_categorical_accuracy: 0.7578 -
val_loss: 1.2640
Epoch 47/75
704/704             108s 153ms/step -
categorical_accuracy: 0.9522 - loss: 0.3000 - val_categorical_accuracy: 0.7622 -
val_loss: 1.2297
Epoch 48/75
704/704             105s 149ms/step -
categorical_accuracy: 0.9544 - loss: 0.2902 - val_categorical_accuracy: 0.7720 -
val_loss: 1.1970
Epoch 49/75
704/704             105s 149ms/step -
categorical_accuracy: 0.9572 - loss: 0.2796 - val_categorical_accuracy: 0.7600 -
val_loss: 1.2166
Epoch 50/75
704/704             106s 150ms/step -
categorical_accuracy: 0.9609 - loss: 0.2724 - val_categorical_accuracy: 0.7742 -
val_loss: 1.1603
Epoch 51/75
704/704             104s 148ms/step -
categorical_accuracy: 0.9654 - loss: 0.2597 - val_categorical_accuracy: 0.7648 -
val_loss: 1.2788
Epoch 52/75
704/704             104s 147ms/step -
categorical_accuracy: 0.9476 - loss: 0.3042 - val_categorical_accuracy: 0.7676 -
```

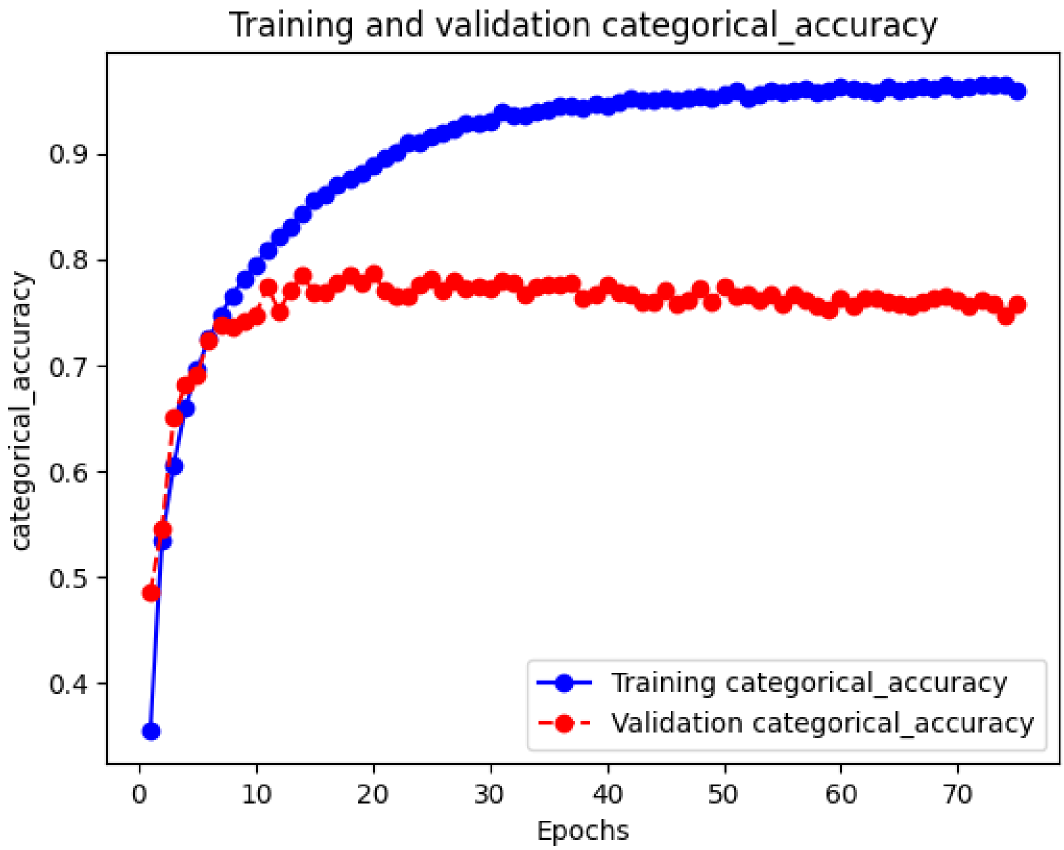
```
val_loss: 1.2717
Epoch 53/75
704/704            106s 151ms/step -
categorical_accuracy: 0.9606 - loss: 0.2722 - val_categorical_accuracy: 0.7620 -
val_loss: 1.2258
Epoch 54/75
704/704            105s 149ms/step -
categorical_accuracy: 0.9617 - loss: 0.2663 - val_categorical_accuracy: 0.7666 -
val_loss: 1.2180
Epoch 55/75
704/704            105s 149ms/step -
categorical_accuracy: 0.9625 - loss: 0.2650 - val_categorical_accuracy: 0.7576 -
val_loss: 1.2705
Epoch 56/75
704/704            106s 151ms/step -
categorical_accuracy: 0.9610 - loss: 0.2693 - val_categorical_accuracy: 0.7664 -
val_loss: 1.2219
Epoch 57/75
704/704            105s 149ms/step -
categorical_accuracy: 0.9649 - loss: 0.2572 - val_categorical_accuracy: 0.7614 -
val_loss: 1.2846
Epoch 58/75
704/704            107s 152ms/step -
categorical_accuracy: 0.9594 - loss: 0.2681 - val_categorical_accuracy: 0.7552 -
val_loss: 1.3258
Epoch 59/75
704/704            115s 164ms/step -
categorical_accuracy: 0.9609 - loss: 0.2684 - val_categorical_accuracy: 0.7532 -
val_loss: 1.2148
Epoch 60/75
704/704            110s 156ms/step -
categorical_accuracy: 0.9628 - loss: 0.2609 - val_categorical_accuracy: 0.7640 -
val_loss: 1.3250
Epoch 61/75
704/704            110s 156ms/step -
categorical_accuracy: 0.9629 - loss: 0.2557 - val_categorical_accuracy: 0.7564 -
val_loss: 1.3340
Epoch 62/75
704/704            110s 156ms/step -
categorical_accuracy: 0.9613 - loss: 0.2622 - val_categorical_accuracy: 0.7632 -
val_loss: 1.3753
Epoch 63/75
704/704            109s 155ms/step -
categorical_accuracy: 0.9581 - loss: 0.2727 - val_categorical_accuracy: 0.7636 -
val_loss: 1.2876
Epoch 64/75
704/704            106s 151ms/step -
categorical_accuracy: 0.9640 - loss: 0.2508 - val_categorical_accuracy: 0.7596 -
```

```
val_loss: 1.3590
Epoch 65/75
704/704            106s 151ms/step -
categorical_accuracy: 0.9658 - loss: 0.2503 - val_categorical_accuracy: 0.7582 -
val_loss: 1.3041
Epoch 66/75
704/704            118s 168ms/step -
categorical_accuracy: 0.9657 - loss: 0.2462 - val_categorical_accuracy: 0.7558 -
val_loss: 1.3651
Epoch 67/75
704/704            115s 163ms/step -
categorical_accuracy: 0.9691 - loss: 0.2354 - val_categorical_accuracy: 0.7604 -
val_loss: 1.2920
Epoch 68/75
704/704            109s 155ms/step -
categorical_accuracy: 0.9656 - loss: 0.2436 - val_categorical_accuracy: 0.7628 -
val_loss: 1.2920
Epoch 69/75
704/704            112s 159ms/step -
categorical_accuracy: 0.9674 - loss: 0.2419 - val_categorical_accuracy: 0.7654 -
val_loss: 1.3231
Epoch 70/75
704/704            108s 154ms/step -
categorical_accuracy: 0.9639 - loss: 0.2514 - val_categorical_accuracy: 0.7612 -
val_loss: 1.3185
Epoch 71/75
704/704            107s 152ms/step -
categorical_accuracy: 0.9674 - loss: 0.2396 - val_categorical_accuracy: 0.7558 -
val_loss: 1.3086
Epoch 72/75
704/704            107s 152ms/step -
categorical_accuracy: 0.9690 - loss: 0.2359 - val_categorical_accuracy: 0.7622 -
val_loss: 1.3068
Epoch 73/75
704/704            107s 152ms/step -
categorical_accuracy: 0.9677 - loss: 0.2383 - val_categorical_accuracy: 0.7572 -
val_loss: 1.3953
Epoch 74/75
704/704            108s 154ms/step -
categorical_accuracy: 0.9664 - loss: 0.2421 - val_categorical_accuracy: 0.7472 -
val_loss: 1.3651
Epoch 75/75
704/704            109s 155ms/step -
categorical_accuracy: 0.9566 - loss: 0.2718 - val_categorical_accuracy: 0.7570 -
val_loss: 1.3231
```

[ ]:
```
loss, acc = model_11.evaluate(x_test, y_test)
print(f"Test accuracy: {acc:.3f}")
plot(history_11.history['loss'], history_11.history['val_loss'], 'loss')
plot(history_11.history['categorical_accuracy'], history_11.
 ↪history['val_categorical_accuracy'], 'categorical_accuracy')
```

```
313/313              7s 24ms/step -
categorical_accuracy: 0.7572 - loss: 1.3486
Test accuracy: 0.755
```





En este próximo modelo, porbaremos otra configuración de los hiperparámetros, tanto de regularizadores como L2 y Dropout, como de las distintas capas de la red.

[ ]:
```
inputs = keras.Input(shape=(32, 32, 3))

x = layers.Conv2D(filters = 32, kernel_size = (3,3),padding = "same",activation␣
 ↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(inputs)
x = layers.Conv2D(filters = 32, kernel_size = (3,3),padding = "same",activation␣
 ↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.1)(x)

x = layers.Conv2D(filters=64, kernel_size= (3,3) ,activation="relu",padding␣
 ↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.Conv2D(filters=128, kernel_size= (3,3) ,activation="relu",padding␣
 ↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.1)(x)
```

```python
x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.1)(x)

x = layers.Flatten()(x)
x = layers.Dense(512,activation = "relu")(x)
x = layers.Dropout(0.2)(x)
x = layers.Dense(100,activation = "relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)
model_12 = keras.Model(inputs=inputs, outputs=outputs)

model_12.summary()
#Compile
model_12.compile(optimizer="Adam",
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[keras.metrics.CategoricalAccuracy()])
#Callback
callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_loss",

        patience=5,
        start_from_epoch=5)
]
#Fitting
history_12 = model_12.fit(x_train, y_train,validation_split = 0.1 ,epochs=75,␣
↪batch_size=256,callbacks = callbacks)
```

Model: "functional_2"

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 32, 32, 3)	0
conv2d_12 (Conv2D)	(None, 32, 32, 32)	896
conv2d_13 (Conv2D)	(None, 32, 32, 32)	9,248
max_pooling2d_6 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_8 (Dropout)	(None, 16, 16, 32)	0

conv2d_14 (Conv2D)	(None, 16, 16, 64)	18,496
conv2d_15 (Conv2D)	(None, 16, 16, 128)	73,856
max_pooling2d_7 (MaxPooling2D)	(None, 8, 8, 128)	0
dropout_9 (Dropout)	(None, 8, 8, 128)	0
conv2d_16 (Conv2D)	(None, 8, 8, 256)	295,168
conv2d_17 (Conv2D)	(None, 8, 8, 256)	590,080
max_pooling2d_8 (MaxPooling2D)	(None, 4, 4, 256)	0
dropout_10 (Dropout)	(None, 4, 4, 256)	0
flatten_2 (Flatten)	(None, 4096)	0
dense_6 (Dense)	(None, 512)	2,097,664
dropout_11 (Dropout)	(None, 512)	0
dense_7 (Dense)	(None, 100)	51,300
dense_8 (Dense)	(None, 10)	1,010

Total params: 3,137,718 (11.97 MB)

Trainable params: 3,137,718 (11.97 MB)

Non-trainable params: 0 (0.00 B)

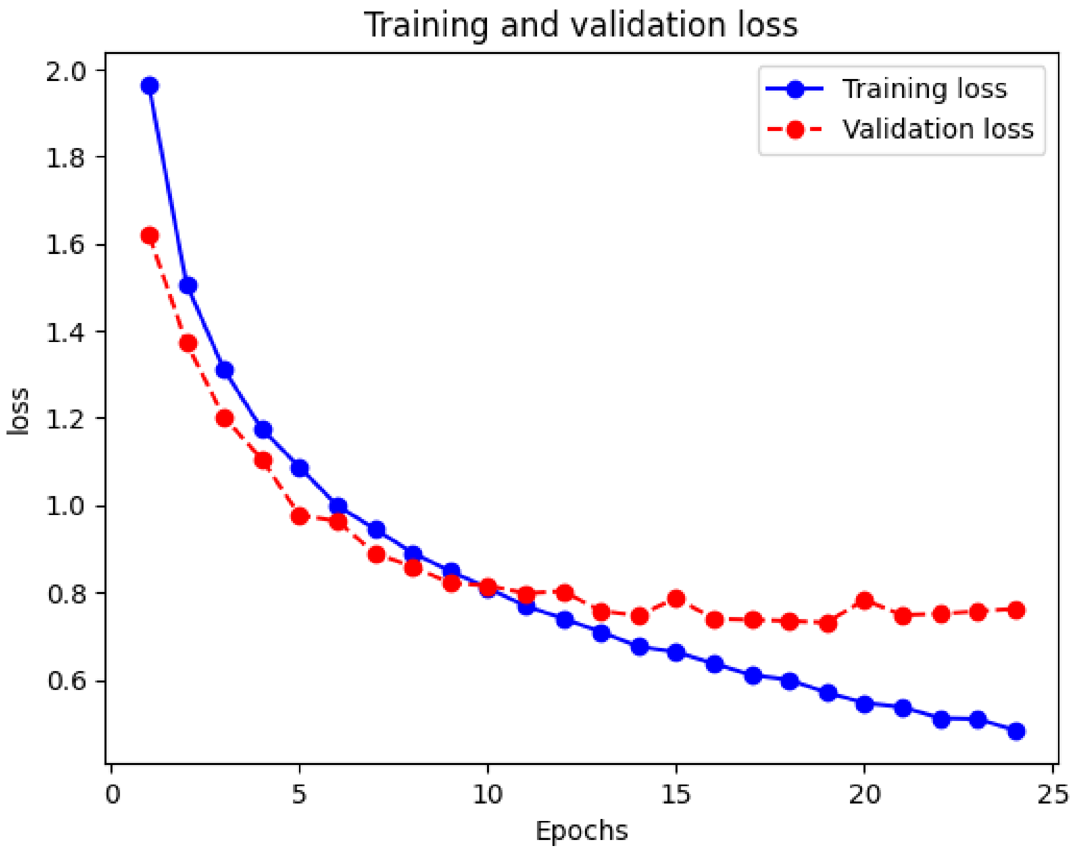
```
Epoch 1/75
176/176             105s 588ms/step -
categorical_accuracy: 0.2302 - loss: 2.2615 - val_categorical_accuracy: 0.4280 -
val_loss: 1.6209
Epoch 2/75
176/176             102s 578ms/step -
categorical_accuracy: 0.4521 - loss: 1.5588 - val_categorical_accuracy: 0.5260 -
val_loss: 1.3720
Epoch 3/75
176/176             101s 574ms/step -
categorical_accuracy: 0.5367 - loss: 1.3555 - val_categorical_accuracy: 0.6032 -
val_loss: 1.2011
Epoch 4/75
```

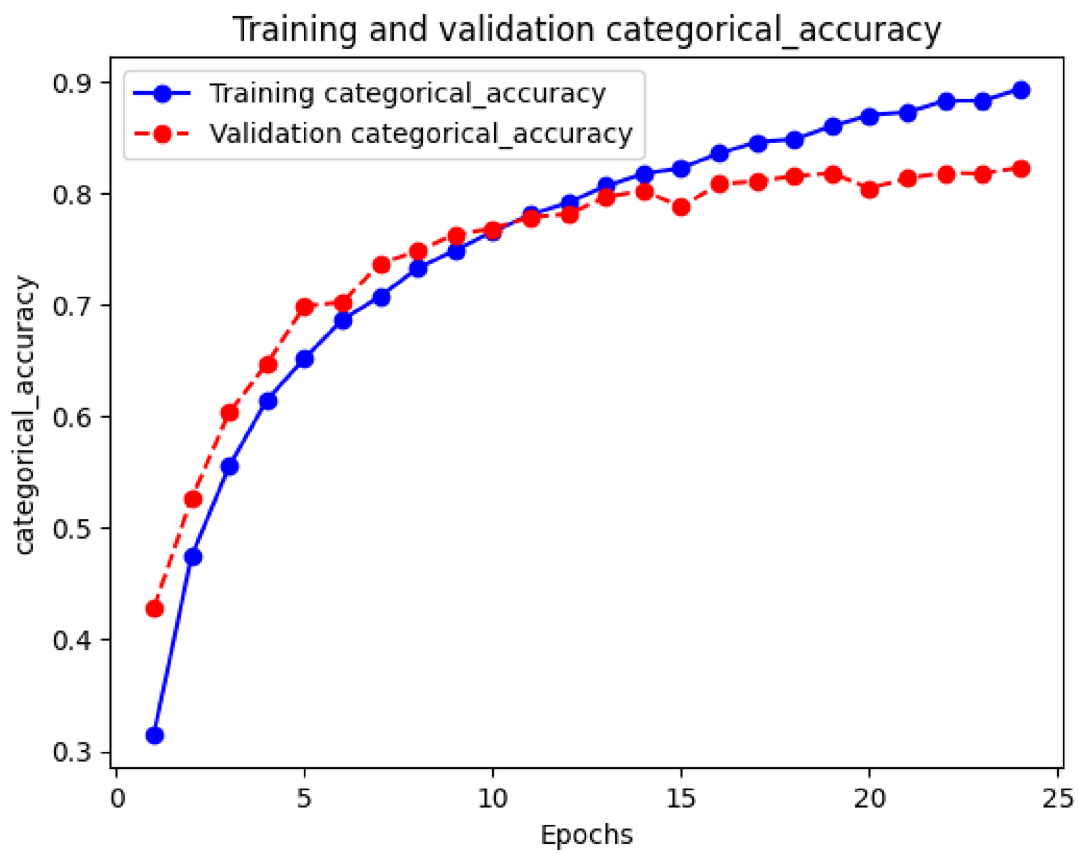
```
176/176              101s 572ms/step -
categorical_accuracy: 0.6062 - loss: 1.1949 - val_categorical_accuracy: 0.6466 -
val_loss: 1.1058
Epoch 5/75
176/176              104s 589ms/step -
categorical_accuracy: 0.6492 - loss: 1.0939 - val_categorical_accuracy: 0.6986 -
val_loss: 0.9768
Epoch 6/75
176/176              102s 582ms/step -
categorical_accuracy: 0.6843 - loss: 1.0079 - val_categorical_accuracy: 0.7020 -
val_loss: 0.9639
Epoch 7/75
176/176              100s 570ms/step -
categorical_accuracy: 0.7057 - loss: 0.9493 - val_categorical_accuracy: 0.7362 -
val_loss: 0.8898
Epoch 8/75
176/176              104s 589ms/step -
categorical_accuracy: 0.7287 - loss: 0.8976 - val_categorical_accuracy: 0.7480 -
val_loss: 0.8591
Epoch 9/75
176/176              668s 4s/step -
categorical_accuracy: 0.7521 - loss: 0.8415 - val_categorical_accuracy: 0.7628 -
val_loss: 0.8215
Epoch 10/75
176/176              102s 580ms/step -
categorical_accuracy: 0.7627 - loss: 0.8137 - val_categorical_accuracy: 0.7680 -
val_loss: 0.8161
Epoch 11/75
176/176              100s 567ms/step -
categorical_accuracy: 0.7788 - loss: 0.7692 - val_categorical_accuracy: 0.7782 -
val_loss: 0.7978
Epoch 12/75
176/176              100s 570ms/step -
categorical_accuracy: 0.7944 - loss: 0.7294 - val_categorical_accuracy: 0.7810 -
val_loss: 0.8028
Epoch 13/75
176/176              101s 577ms/step -
categorical_accuracy: 0.8121 - loss: 0.6953 - val_categorical_accuracy: 0.7962 -
val_loss: 0.7574
Epoch 14/75
176/176              100s 569ms/step -
categorical_accuracy: 0.8193 - loss: 0.6639 - val_categorical_accuracy: 0.8020 -
val_loss: 0.7470
Epoch 15/75
176/176              100s 571ms/step -
categorical_accuracy: 0.8272 - loss: 0.6473 - val_categorical_accuracy: 0.7874 -
val_loss: 0.7870
Epoch 16/75
```

```
176/176              103s 588ms/step -
categorical_accuracy: 0.8324 - loss: 0.6430 - val_categorical_accuracy: 0.8078 -
val_loss: 0.7396
Epoch 17/75
176/176              104s 589ms/step -
categorical_accuracy: 0.8494 - loss: 0.5973 - val_categorical_accuracy: 0.8106 -
val_loss: 0.7373
Epoch 18/75
176/176              117s 666ms/step -
categorical_accuracy: 0.8527 - loss: 0.5837 - val_categorical_accuracy: 0.8148 -
val_loss: 0.7349
Epoch 19/75
176/176              103s 583ms/step -
categorical_accuracy: 0.8615 - loss: 0.5660 - val_categorical_accuracy: 0.8176 -
val_loss: 0.7312
Epoch 20/75
176/176              101s 572ms/step -
categorical_accuracy: 0.8746 - loss: 0.5354 - val_categorical_accuracy: 0.8042 -
val_loss: 0.7828
Epoch 21/75
176/176              100s 569ms/step -
categorical_accuracy: 0.8784 - loss: 0.5197 - val_categorical_accuracy: 0.8130 -
val_loss: 0.7479
Epoch 22/75
176/176              101s 574ms/step -
categorical_accuracy: 0.8850 - loss: 0.5059 - val_categorical_accuracy: 0.8180 -
val_loss: 0.7510
Epoch 23/75
176/176              104s 593ms/step -
categorical_accuracy: 0.8830 - loss: 0.5045 - val_categorical_accuracy: 0.8172 -
val_loss: 0.7567
Epoch 24/75
176/176              100s 571ms/step -
categorical_accuracy: 0.8971 - loss: 0.4768 - val_categorical_accuracy: 0.8228 -
val_loss: 0.7629
```

[ ]:
```
loss, acc = model_12.evaluate(x_test, y_test)
print(f"Test accuracy: {acc:.3f}")
plot(history_12.history['loss'], history_12.history['val_loss'], 'loss')
plot(history_12.history['categorical_accuracy'], history_12.
 ↪history['val_categorical_accuracy'], 'categorical_accuracy')#
```

```
313/313              7s 22ms/step -
categorical_accuracy: 0.8079 - loss: 0.8052
Test accuracy: 0.812
```


Training and validation loss
Training loss
Validation loss
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
loss
0
5
10
15
20
25
Epochs



Repetimos el procedimiento anterior para generar otro modelo diferente y ver si nos ofrece un mejor resultado.

[10]:
```
inputs = keras.Input(shape=(32, 32, 3))

x = layers.Conv2D(filters = 64, kernel_size = (3,3),padding = "same",activation␣
↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(inputs)
x = layers.Conv2D(filters = 64, kernel_size = (3,3),padding = "same",activation␣
↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.1)(x)

x = layers.Conv2D(filters=128 , kernel_size= (3,3) ,activation="relu",padding␣
↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.Conv2D(filters=128, kernel_size= (3,3) ,activation="relu",padding␣
↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.15)(x)
```

```
x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.15)(x)

x = x = layers.Conv2D(filters = 512, kernel_size = (3,3),padding =␣
↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.1)(x)

x = layers.Flatten()(x)
x = layers.Dense(256,activation = "relu")(x)
x = layers.Dropout(0.15)(x)
x = layers.Dense(64,activation = "relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)
model_13 = keras.Model(inputs=inputs, outputs=outputs)

model_13.summary()
#Compile
model_13.compile(optimizer="Adam",
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[keras.metrics.CategoricalAccuracy()])
#Callback
callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_loss",
        patience=10,
        start_from_epoch=10)
]
#Fitting
history_13 = model_13.fit(x_train, y_train,validation_split = 0.1 ,epochs=75,␣
↪batch_size=1024,callbacks = callbacks)

loss, acc = model_13.evaluate(x_test, y_test)
print(f"Test accuracy: {acc:.3f}")
plot(history_13.history['loss'], history_13.history['val_loss'], 'loss')
plot(history_13.history['categorical_accuracy'], history_13.
↪history['val_categorical_accuracy'], 'categorical_accuracy')#
```

```
Model: "functional_1"


 Layer (type)                         Output Shape                          ␣
↪Param #
```

```
input_layer_1 (InputLayer)          (None, 32, 32, 3)               ␣
↪  0

conv2d_7 (Conv2D)                   (None, 32, 32, 64)            ␣
↪1,792

conv2d_8 (Conv2D)                   (None, 32, 32, 64)           ␣
↪36,928

max_pooling2d_4 (MaxPooling2D)      (None, 16, 16, 64)              ␣
↪  0

dropout_5 (Dropout)                 (None, 16, 16, 64)              ␣
↪  0

conv2d_9 (Conv2D)                   (None, 16, 16, 128)          ␣
↪73,856

conv2d_10 (Conv2D)                  (None, 16, 16, 128)         ␣
↪147,584

max_pooling2d_5 (MaxPooling2D)      (None, 8, 8, 128)               ␣
↪  0

dropout_6 (Dropout)                 (None, 8, 8, 128)               ␣
↪  0

conv2d_11 (Conv2D)                  (None, 8, 8, 256)           ␣
↪295,168

conv2d_12 (Conv2D)                  (None, 8, 8, 256)           ␣
↪590,080

max_pooling2d_6 (MaxPooling2D)      (None, 4, 4, 256)               ␣
↪  0

dropout_7 (Dropout)                 (None, 4, 4, 256)               ␣
↪  0

conv2d_13 (Conv2D)                  (None, 4, 4, 512)         ␣
↪1,180,160

max_pooling2d_7 (MaxPooling2D)      (None, 2, 2, 512)               ␣
↪  0
```

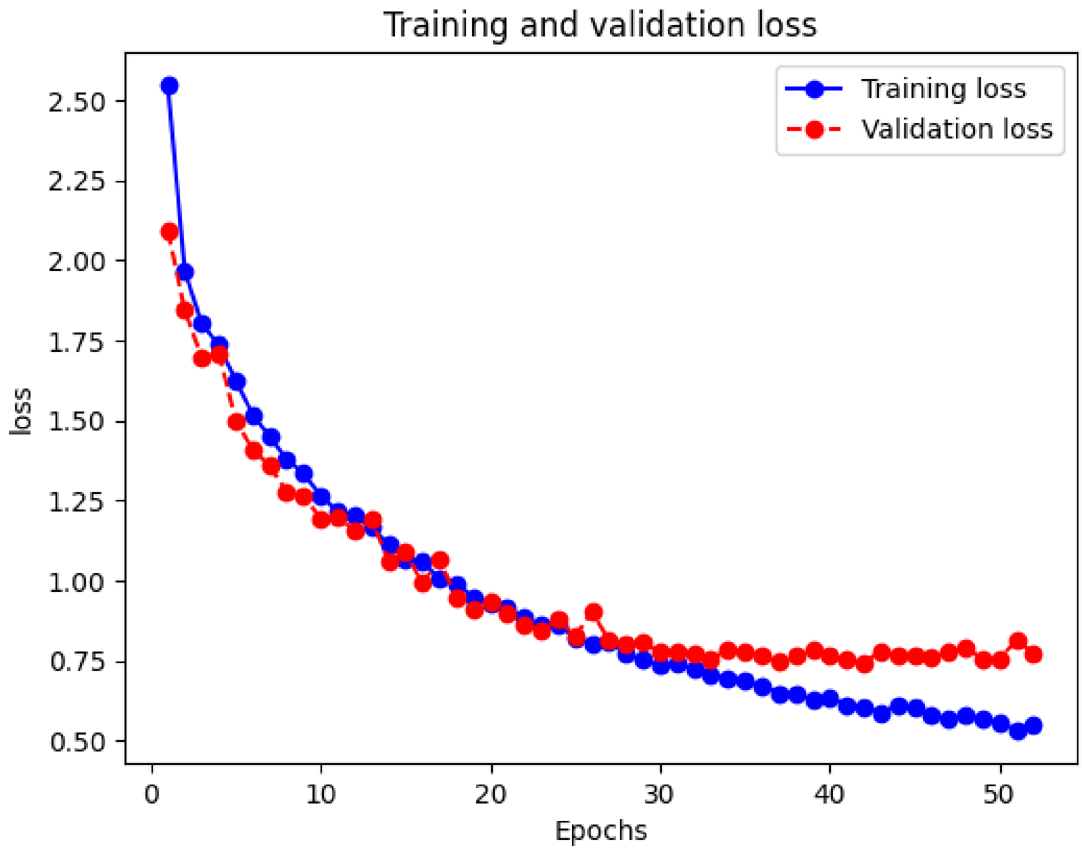
```
 dropout_8 (Dropout)                   (None, 2, 2, 512)                     ␣
 ↪  0

 flatten_1 (Flatten)                   (None, 2048)                          ␣
 ↪  0

 dense_3 (Dense)                       (None, 256)                         ␣
 ↪524,544

 dropout_9 (Dropout)                   (None, 256)                           ␣
 ↪  0

 dense_4 (Dense)                       (None, 64)                           ␣
 ↪16,448

 dense_5 (Dense)                       (None, 10)                            ␣
 ↪650


 Total params: 2,867,210 (10.94 MB)

 Trainable params: 2,867,210 (10.94 MB)

 Non-trainable params: 0 (0.00 B)

Epoch 1/75
44/44                 91s 1s/step -
categorical_accuracy: 0.1391 - loss: 2.8723 - val_categorical_accuracy: 0.2624 -
val_loss: 2.0934
Epoch 2/75
44/44                 63s 206ms/step -
categorical_accuracy: 0.2919 - loss: 2.0282 - val_categorical_accuracy: 0.3566 -
val_loss: 1.8434
Epoch 3/75
44/44                 10s 208ms/step -
categorical_accuracy: 0.3578 - loss: 1.8275 - val_categorical_accuracy: 0.4056 -
val_loss: 1.6960
Epoch 4/75
44/44                 10s 207ms/step -
categorical_accuracy: 0.3919 - loss: 1.7498 - val_categorical_accuracy: 0.4188 -
val_loss: 1.7108
Epoch 5/75
44/44                 10s 207ms/step -
categorical_accuracy: 0.4263 - loss: 1.6634 - val_categorical_accuracy: 0.4858 -
val_loss: 1.5007
```

```
Epoch 6/75
44/44            9s 208ms/step -
categorical_accuracy: 0.4736 - loss: 1.5398 - val_categorical_accuracy: 0.5346 -
val_loss: 1.4083
Epoch 7/75
44/44            10s 212ms/step -
categorical_accuracy: 0.5145 - loss: 1.4492 - val_categorical_accuracy: 0.5498 -
val_loss: 1.3605
Epoch 8/75
44/44            10s 214ms/step -
categorical_accuracy: 0.5369 - loss: 1.3887 - val_categorical_accuracy: 0.5926 -
val_loss: 1.2761
Epoch 9/75
44/44            10s 219ms/step -
categorical_accuracy: 0.5644 - loss: 1.3327 - val_categorical_accuracy: 0.5878 -
val_loss: 1.2667
Epoch 10/75
44/44            10s 217ms/step -
categorical_accuracy: 0.5805 - loss: 1.2805 - val_categorical_accuracy: 0.6206 -
val_loss: 1.1935
Epoch 11/75
44/44            10s 214ms/step -
categorical_accuracy: 0.6116 - loss: 1.2190 - val_categorical_accuracy: 0.6284 -
val_loss: 1.1975
Epoch 12/75
44/44            9s 213ms/step -
categorical_accuracy: 0.6169 - loss: 1.2080 - val_categorical_accuracy: 0.6392 -
val_loss: 1.1578
Epoch 13/75
44/44            10s 212ms/step -
categorical_accuracy: 0.6217 - loss: 1.1883 - val_categorical_accuracy: 0.6262 -
val_loss: 1.1892
Epoch 14/75
44/44            10s 211ms/step -
categorical_accuracy: 0.6465 - loss: 1.1233 - val_categorical_accuracy: 0.6770 -
val_loss: 1.0609
Epoch 15/75
44/44            10s 212ms/step -
categorical_accuracy: 0.6733 - loss: 1.0632 - val_categorical_accuracy: 0.6612 -
val_loss: 1.0875
Epoch 16/75
44/44            10s 213ms/step -
categorical_accuracy: 0.6724 - loss: 1.0695 - val_categorical_accuracy: 0.7114 -
val_loss: 0.9922
Epoch 17/75
44/44            10s 215ms/step -
categorical_accuracy: 0.6932 - loss: 1.0120 - val_categorical_accuracy: 0.6904 -
val_loss: 1.0648
```

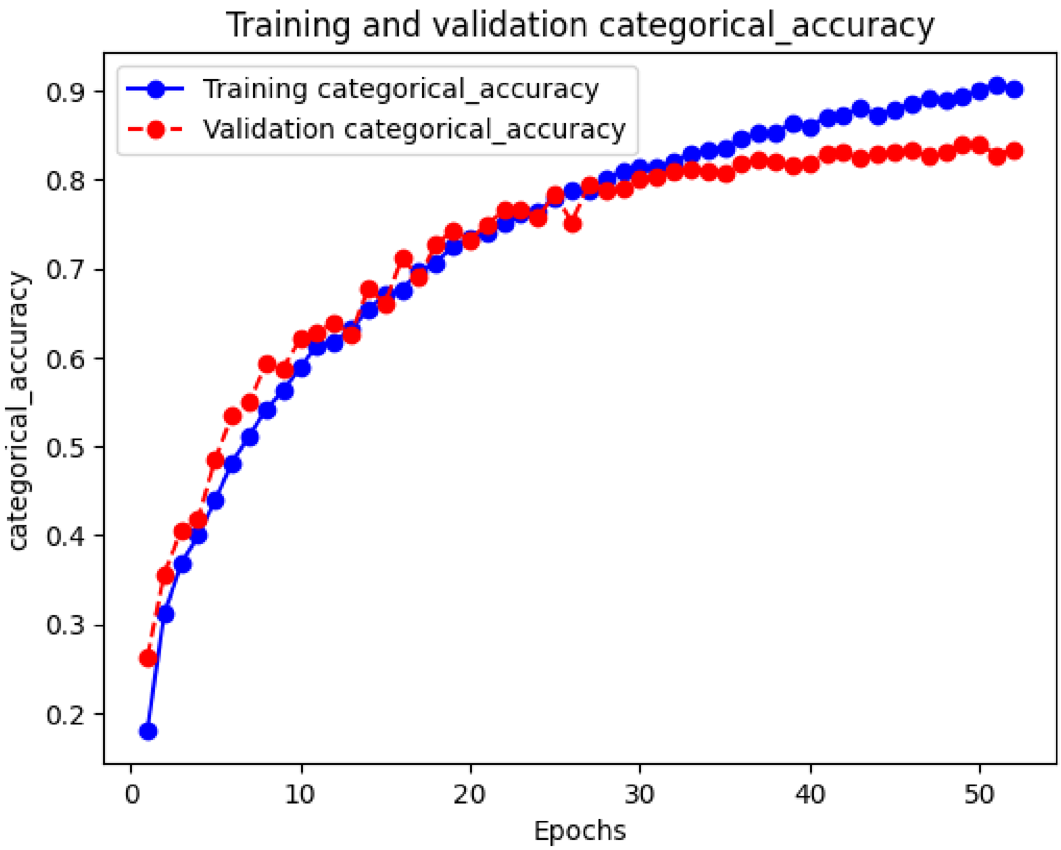
```
Epoch 18/75
44/44            10s 215ms/step -
categorical_accuracy: 0.6996 - loss: 1.0086 - val_categorical_accuracy: 0.7266 -
val_loss: 0.9463
Epoch 19/75
44/44            10s 215ms/step -
categorical_accuracy: 0.7283 - loss: 0.9438 - val_categorical_accuracy: 0.7432 -
val_loss: 0.9126
Epoch 20/75
44/44            9s 215ms/step -
categorical_accuracy: 0.7311 - loss: 0.9374 - val_categorical_accuracy: 0.7320 -
val_loss: 0.9361
Epoch 21/75
44/44            10s 215ms/step -
categorical_accuracy: 0.7374 - loss: 0.9223 - val_categorical_accuracy: 0.7498 -
val_loss: 0.9001
Epoch 22/75
44/44            9s 215ms/step -
categorical_accuracy: 0.7507 - loss: 0.8861 - val_categorical_accuracy: 0.7652 -
val_loss: 0.8644
Epoch 23/75
44/44            10s 213ms/step -
categorical_accuracy: 0.7655 - loss: 0.8487 - val_categorical_accuracy: 0.7664 -
val_loss: 0.8437
Epoch 24/75
44/44            10s 213ms/step -
categorical_accuracy: 0.7662 - loss: 0.8432 - val_categorical_accuracy: 0.7568 -
val_loss: 0.8779
Epoch 25/75
44/44            10s 215ms/step -
categorical_accuracy: 0.7787 - loss: 0.8247 - val_categorical_accuracy: 0.7844 -
val_loss: 0.8263
Epoch 26/75
44/44            10s 215ms/step -
categorical_accuracy: 0.7865 - loss: 0.7986 - val_categorical_accuracy: 0.7520 -
val_loss: 0.9033
Epoch 27/75
44/44            9s 215ms/step -
categorical_accuracy: 0.7813 - loss: 0.8199 - val_categorical_accuracy: 0.7944 -
val_loss: 0.8119
Epoch 28/75
44/44            9s 214ms/step -
categorical_accuracy: 0.7998 - loss: 0.7726 - val_categorical_accuracy: 0.7888 -
val_loss: 0.8007
Epoch 29/75
44/44            10s 214ms/step -
categorical_accuracy: 0.8126 - loss: 0.7447 - val_categorical_accuracy: 0.7906 -
val_loss: 0.8072
```

```
Epoch 30/75
44/44            9s 215ms/step -
categorical_accuracy: 0.8131 - loss: 0.7372 - val_categorical_accuracy: 0.8010 -
val_loss: 0.7760
Epoch 31/75
44/44            10s 214ms/step -
categorical_accuracy: 0.8203 - loss: 0.7176 - val_categorical_accuracy: 0.8038 -
val_loss: 0.7797
Epoch 32/75
44/44            10s 214ms/step -
categorical_accuracy: 0.8163 - loss: 0.7331 - val_categorical_accuracy: 0.8092 -
val_loss: 0.7744
Epoch 33/75
44/44            9s 214ms/step -
categorical_accuracy: 0.8274 - loss: 0.7107 - val_categorical_accuracy: 0.8124 -
val_loss: 0.7563
Epoch 34/75
44/44            10s 214ms/step -
categorical_accuracy: 0.8368 - loss: 0.6829 - val_categorical_accuracy: 0.8102 -
val_loss: 0.7825
Epoch 35/75
44/44            10s 213ms/step -
categorical_accuracy: 0.8379 - loss: 0.6884 - val_categorical_accuracy: 0.8072 -
val_loss: 0.7769
Epoch 36/75
44/44            10s 214ms/step -
categorical_accuracy: 0.8487 - loss: 0.6618 - val_categorical_accuracy: 0.8176 -
val_loss: 0.7666
Epoch 37/75
44/44            10s 214ms/step -
categorical_accuracy: 0.8536 - loss: 0.6438 - val_categorical_accuracy: 0.8230 -
val_loss: 0.7467
Epoch 38/75
44/44            10s 213ms/step -
categorical_accuracy: 0.8559 - loss: 0.6399 - val_categorical_accuracy: 0.8196 -
val_loss: 0.7677
Epoch 39/75
44/44            10s 215ms/step -
categorical_accuracy: 0.8653 - loss: 0.6217 - val_categorical_accuracy: 0.8150 -
val_loss: 0.7870
Epoch 40/75
44/44            10s 213ms/step -
categorical_accuracy: 0.8574 - loss: 0.6390 - val_categorical_accuracy: 0.8176 -
val_loss: 0.7663
Epoch 41/75
44/44            10s 214ms/step -
categorical_accuracy: 0.8736 - loss: 0.6009 - val_categorical_accuracy: 0.8288 -
val_loss: 0.7530
```

```
Epoch 42/75
44/44            9s 213ms/step -
categorical_accuracy: 0.8752 - loss: 0.5966 - val_categorical_accuracy: 0.8316 -
val_loss: 0.7434
Epoch 43/75
44/44            9s 213ms/step -
categorical_accuracy: 0.8806 - loss: 0.5856 - val_categorical_accuracy: 0.8240 -
val_loss: 0.7809
Epoch 44/75
44/44            9s 213ms/step -
categorical_accuracy: 0.8753 - loss: 0.6042 - val_categorical_accuracy: 0.8292 -
val_loss: 0.7646
Epoch 45/75
44/44            9s 213ms/step -
categorical_accuracy: 0.8840 - loss: 0.5848 - val_categorical_accuracy: 0.8300 -
val_loss: 0.7679
Epoch 46/75
44/44            10s 213ms/step -
categorical_accuracy: 0.8873 - loss: 0.5795 - val_categorical_accuracy: 0.8328 -
val_loss: 0.7606
Epoch 47/75
44/44            10s 213ms/step -
categorical_accuracy: 0.8952 - loss: 0.5564 - val_categorical_accuracy: 0.8262 -
val_loss: 0.7757
Epoch 48/75
44/44            10s 213ms/step -
categorical_accuracy: 0.8912 - loss: 0.5699 - val_categorical_accuracy: 0.8302 -
val_loss: 0.7928
Epoch 49/75
44/44            10s 213ms/step -
categorical_accuracy: 0.8960 - loss: 0.5631 - val_categorical_accuracy: 0.8404 -
val_loss: 0.7537
Epoch 50/75
44/44            10s 213ms/step -
categorical_accuracy: 0.9063 - loss: 0.5416 - val_categorical_accuracy: 0.8400 -
val_loss: 0.7567
Epoch 51/75
44/44            9s 214ms/step -
categorical_accuracy: 0.9131 - loss: 0.5184 - val_categorical_accuracy: 0.8278 -
val_loss: 0.8137
Epoch 52/75
44/44            10s 213ms/step -
categorical_accuracy: 0.9051 - loss: 0.5431 - val_categorical_accuracy: 0.8340 -
val_loss: 0.7742
313/313            4s 8ms/step -
categorical_accuracy: 0.8290 - loss: 0.8001
Test accuracy: 0.827
```



Training and validation loss
Training loss
Validation loss
loss
Epochs
2.50
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0
10
20
30
40
50



Podemos ver que este modelo es el que mejor resultados nos ofrece y claramente no está sobreentrenado. Esto es debido a la fuerte regularización que aplicamos al modelo, tanto con `Dropout` o `L2` como con `Early Stopping`.

2 Data augmentation

A continuación, utilizaremos el mismo modelo que en la ejecución anterior, pero le añadiremos la utilización de la técnica de `Data Augmentation`, de esta manera, dado que tenemos un buen valor de `accuracy`, comprobaremos el efecto que tendrá el aumento de instancias a la hora de entrenar el modelo.

Utilizaremos tres maneras para la creación de nuevas instancias:

- RandomFlip : invierte la imagen, en nuestro caso horizontalmente.
- RandomRotation : inclina la imagen en un rango de angulos especificado por nosotros. Rota tanto hacia la izq como hacia la drch

[24]:
```
data_augmentation = keras.Sequential(
[
    layers.RandomFlip('horizontal'),                # de que forma flipea
    layers.RandomRotation(0.1)                    # de que forma rota
```

```
])
```

```
[ ]: inputs = keras.Input(shape=(32, 32, 3))

x2 = data_augmentation(inputs)
x = layers.Conv2D(filters = 64, kernel_size = (3,3),padding = "same",activation␣
 ↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(x2)
x = layers.Conv2D(filters = 64, kernel_size = (3,3),padding = "same",activation␣
 ↪= "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.1)(x)

x = layers.Conv2D(filters=128 , kernel_size= (3,3) ,activation="relu",padding␣
 ↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.Conv2D(filters=128, kernel_size= (3,3) ,activation="relu",padding␣
 ↪="same",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.15)(x)

x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
 ↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = x = layers.Conv2D(filters = 256, kernel_size = (3,3),padding =␣
 ↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.15)(x)

x = x = layers.Conv2D(filters = 512, kernel_size = (3,3),padding =␣
 ↪"same",activation = "relu",kernel_regularizer = regularizers.l2(1e-3))(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Dropout(0.1)(x)

x = layers.Flatten()(x)
x = layers.Dense(256,activation = "relu")(x)
x = layers.Dropout(0.15)(x)
x = layers.Dense(64,activation = "relu")(x)
outputs = layers.Dense(10, activation="softmax")(x)
model2 = keras.Model(inputs=inputs, outputs=outputs)


model2.summary()
#Compile
model2.compile(optimizer="Adam",
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[keras.metrics.CategoricalAccuracy()])
#Callback
callbacks = [
```

```
    keras.callbacks.EarlyStopping(
        monitor="val_loss",
        patience=15,
        start_from_epoch=5,
        restore_best_weights=True
        )
]
#Fitting
history2 = model2.fit(x_train, y_train,validation_split = 0.3 ,epochs=200,␣
 ↪batch_size=1024,callbacks = callbacks)
```

```
Model: "functional_15"


 Layer (type)                    Output Shape                                ␣
 ↪Param #

 input_layer_15 (InputLayer)     (None, 32, 32, 3)                             ␣
 ↪   0

 sequential_2 (Sequential)       (None, 32, 32, 3)                             ␣
 ↪   0

 conv2d_84 (Conv2D)              (None, 32, 32, 64)                          ␣
 ↪1,792

 conv2d_85 (Conv2D)              (None, 32, 32, 64)                         ␣
 ↪36,928

 max_pooling2d_48 (MaxPooling2D) (None, 16, 16, 64)                            ␣
 ↪   0

 dropout_60 (Dropout)            (None, 16, 16, 64)                            ␣
 ↪   0

 conv2d_86 (Conv2D)              (None, 16, 16, 128)                        ␣
 ↪73,856

 conv2d_87 (Conv2D)              (None, 16, 16, 128)                       ␣
 ↪147,584

 max_pooling2d_49 (MaxPooling2D) (None, 8, 8, 128)                             ␣
 ↪   0

 dropout_61 (Dropout)            (None, 8, 8, 128)                             ␣
 ↪   0
```

```
conv2d_88 (Conv2D)                    (None, 8, 8, 256)                  ␣
↪295,168

conv2d_89 (Conv2D)                    (None, 8, 8, 256)                  ␣
↪590,080

max_pooling2d_50 (MaxPooling2D)       (None, 4, 4, 256)                      ␣
↪  0

dropout_62 (Dropout)                  (None, 4, 4, 256)                      ␣
↪  0

conv2d_90 (Conv2D)                    (None, 4, 4, 512)                ␣
↪1,180,160

max_pooling2d_51 (MaxPooling2D)       (None, 2, 2, 512)                      ␣
↪  0

dropout_63 (Dropout)                  (None, 2, 2, 512)                      ␣
↪  0

flatten_12 (Flatten)                  (None, 2048)                           ␣
↪  0

dense_36 (Dense)                      (None, 256)                        ␣
↪524,544

dropout_64 (Dropout)                  (None, 256)                            ␣
↪  0

dense_37 (Dense)                      (None, 64)                          ␣
↪16,448

dense_38 (Dense)                      (None, 10)                             ␣
↪650


 Total params: 2,867,210 (10.94 MB)

 Trainable params: 2,867,210 (10.94 MB)

 Non-trainable params: 0 (0.00 B)

Epoch 1/200
```

```
35/35 41s 298ms/step -
categorical_accuracy: 0.1285 - loss: 2.9143 - val_categorical_accuracy: 0.2321 -
val_loss: 2.1723
Epoch 2/200
35/35 20s 280ms/step -
categorical_accuracy: 0.2520 - loss: 2.1251 - val_categorical_accuracy: 0.3048 -
val_loss: 2.0078
Epoch 3/200
35/35 10s 280ms/step -
categorical_accuracy: 0.3202 - loss: 1.9510 - val_categorical_accuracy: 0.3794 -
val_loss: 1.7796
Epoch 4/200
35/35 10s 280ms/step -
categorical_accuracy: 0.3572 - loss: 1.8332 - val_categorical_accuracy: 0.3437 -
val_loss: 1.8353
Epoch 5/200
35/35 10s 277ms/step -
categorical_accuracy: 0.3778 - loss: 1.7693 - val_categorical_accuracy: 0.4289 -
val_loss: 1.6222
Epoch 6/200
35/35 10s 278ms/step -
categorical_accuracy: 0.4080 - loss: 1.7004 - val_categorical_accuracy: 0.4449 -
val_loss: 1.5833
Epoch 7/200
35/35 10s 275ms/step -
categorical_accuracy: 0.4287 - loss: 1.6427 - val_categorical_accuracy: 0.4655 -
val_loss: 1.5584
Epoch 8/200
35/35 10s 273ms/step -
categorical_accuracy: 0.4557 - loss: 1.5754 - val_categorical_accuracy: 0.4589 -
val_loss: 1.5439
Epoch 9/200
35/35 10s 276ms/step -
categorical_accuracy: 0.4604 - loss: 1.5518 - val_categorical_accuracy: 0.5113 -
val_loss: 1.4624
Epoch 10/200
35/35 10s 276ms/step -
categorical_accuracy: 0.4957 - loss: 1.4826 - val_categorical_accuracy: 0.4932 -
val_loss: 1.5328
Epoch 11/200
35/35 10s 277ms/step -
categorical_accuracy: 0.4975 - loss: 1.4741 - val_categorical_accuracy: 0.5440 -
val_loss: 1.3587
Epoch 12/200
35/35 10s 283ms/step -
categorical_accuracy: 0.5154 - loss: 1.4406 - val_categorical_accuracy: 0.5427 -
val_loss: 1.3734
Epoch 13/200
```

```
35/35              10s 277ms/step -
categorical_accuracy: 0.5297 - loss: 1.4060 - val_categorical_accuracy: 0.5567 -
val_loss: 1.3272
Epoch 14/200
35/35              10s 275ms/step -
categorical_accuracy: 0.5323 - loss: 1.3941 - val_categorical_accuracy: 0.5591 -
val_loss: 1.3427
Epoch 15/200
35/35              10s 277ms/step -
categorical_accuracy: 0.5335 - loss: 1.3974 - val_categorical_accuracy: 0.5816 -
val_loss: 1.2815
Epoch 16/200
35/35              10s 277ms/step -
categorical_accuracy: 0.5635 - loss: 1.3263 - val_categorical_accuracy: 0.5983 -
val_loss: 1.2259
Epoch 17/200
35/35              10s 282ms/step -
categorical_accuracy: 0.5593 - loss: 1.3282 - val_categorical_accuracy: 0.5253 -
val_loss: 1.3923
Epoch 18/200
35/35              10s 282ms/step -
categorical_accuracy: 0.5417 - loss: 1.3811 - val_categorical_accuracy: 0.5768 -
val_loss: 1.3522
Epoch 19/200
35/35              10s 275ms/step -
categorical_accuracy: 0.5515 - loss: 1.3626 - val_categorical_accuracy: 0.5971 -
val_loss: 1.2613
Epoch 20/200
35/35              10s 276ms/step -
categorical_accuracy: 0.5886 - loss: 1.2721 - val_categorical_accuracy: 0.6272 -
val_loss: 1.1717
Epoch 21/200
35/35              10s 276ms/step -
categorical_accuracy: 0.5982 - loss: 1.2416 - val_categorical_accuracy: 0.5799 -
val_loss: 1.2836
Epoch 22/200
35/35              10s 276ms/step -
categorical_accuracy: 0.5845 - loss: 1.2766 - val_categorical_accuracy: 0.6221 -
val_loss: 1.2163
Epoch 23/200
35/35              10s 276ms/step -
categorical_accuracy: 0.6080 - loss: 1.2250 - val_categorical_accuracy: 0.6158 -
val_loss: 1.2100
Epoch 24/200
35/35              10s 282ms/step -
categorical_accuracy: 0.6013 - loss: 1.2409 - val_categorical_accuracy: 0.6358 -
val_loss: 1.1588
Epoch 25/200
```

```
35/35              10s 277ms/step -
categorical_accuracy: 0.6245 - loss: 1.1883 - val_categorical_accuracy: 0.6330 -
val_loss: 1.1754
Epoch 26/200
35/35              10s 277ms/step -
categorical_accuracy: 0.6257 - loss: 1.1849 - val_categorical_accuracy: 0.6635 -
val_loss: 1.0827
Epoch 27/200
35/35              10s 278ms/step -
categorical_accuracy: 0.6342 - loss: 1.1722 - val_categorical_accuracy: 0.6455 -
val_loss: 1.1279
Epoch 28/200
35/35              10s 284ms/step -
categorical_accuracy: 0.6283 - loss: 1.1789 - val_categorical_accuracy: 0.6811 -
val_loss: 1.0473
Epoch 29/200
35/35              10s 278ms/step -
categorical_accuracy: 0.6508 - loss: 1.1220 - val_categorical_accuracy: 0.6771 -
val_loss: 1.0479
Epoch 30/200
35/35              10s 281ms/step -
categorical_accuracy: 0.6605 - loss: 1.1074 - val_categorical_accuracy: 0.6602 -
val_loss: 1.1065
Epoch 31/200
35/35              10s 276ms/step -
categorical_accuracy: 0.6469 - loss: 1.1449 - val_categorical_accuracy: 0.6745 -
val_loss: 1.0820
Epoch 32/200
35/35              10s 277ms/step -
categorical_accuracy: 0.6574 - loss: 1.1117 - val_categorical_accuracy: 0.6764 -
val_loss: 1.0606
Epoch 33/200
35/35              10s 277ms/step -
categorical_accuracy: 0.6615 - loss: 1.1089 - val_categorical_accuracy: 0.7070 -
val_loss: 0.9878
Epoch 34/200
35/35              10s 283ms/step -
categorical_accuracy: 0.6651 - loss: 1.0945 - val_categorical_accuracy: 0.6927 -
val_loss: 1.0176
Epoch 35/200
35/35              10s 283ms/step -
categorical_accuracy: 0.6721 - loss: 1.0775 - val_categorical_accuracy: 0.6757 -
val_loss: 1.0919
Epoch 36/200
35/35              10s 275ms/step -
categorical_accuracy: 0.6760 - loss: 1.0699 - val_categorical_accuracy: 0.6826 -
val_loss: 1.0794
Epoch 37/200
```

```
35/35 10s 276ms/step -
categorical_accuracy: 0.6844 - loss: 1.0541 - val_categorical_accuracy: 0.6802 -
val_loss: 1.0755
Epoch 38/200
35/35 10s 283ms/step -
categorical_accuracy: 0.6882 - loss: 1.0446 - val_categorical_accuracy: 0.6897 -
val_loss: 1.0580
Epoch 39/200
35/35 10s 279ms/step -
categorical_accuracy: 0.6898 - loss: 1.0426 - val_categorical_accuracy: 0.7254 -
val_loss: 0.9544
Epoch 40/200
35/35 10s 277ms/step -
categorical_accuracy: 0.6907 - loss: 1.0344 - val_categorical_accuracy: 0.7032 -
val_loss: 1.0087
Epoch 41/200
35/35 10s 277ms/step -
categorical_accuracy: 0.6974 - loss: 1.0338 - val_categorical_accuracy: 0.7264 -
val_loss: 0.9486
Epoch 42/200
35/35 10s 276ms/step -
categorical_accuracy: 0.7071 - loss: 1.0043 - val_categorical_accuracy: 0.7338 -
val_loss: 0.9158
Epoch 43/200
35/35 10s 277ms/step -
categorical_accuracy: 0.6988 - loss: 1.0264 - val_categorical_accuracy: 0.7160 -
val_loss: 0.9823
Epoch 44/200
35/35 10s 283ms/step -
categorical_accuracy: 0.7057 - loss: 1.0108 - val_categorical_accuracy: 0.7231 -
val_loss: 0.9565
Epoch 45/200
35/35 10s 277ms/step -
categorical_accuracy: 0.6990 - loss: 1.0280 - val_categorical_accuracy: 0.7193 -
val_loss: 0.9669
Epoch 46/200
35/35 10s 277ms/step -
categorical_accuracy: 0.7167 - loss: 0.9715 - val_categorical_accuracy: 0.7428 -
val_loss: 0.9055
Epoch 47/200
35/35 10s 276ms/step -
categorical_accuracy: 0.7150 - loss: 0.9959 - val_categorical_accuracy: 0.7486 -
val_loss: 0.9031
Epoch 48/200
35/35 10s 276ms/step -
categorical_accuracy: 0.7225 - loss: 0.9738 - val_categorical_accuracy: 0.6955 -
val_loss: 1.0736
Epoch 49/200
```

```
35/35               10s 277ms/step -
categorical_accuracy: 0.7239 - loss: 0.9675 - val_categorical_accuracy: 0.7573 -
val_loss: 0.8665
Epoch 50/200
35/35               10s 278ms/step -
categorical_accuracy: 0.7280 - loss: 0.9597 - val_categorical_accuracy: 0.7315 -
val_loss: 0.9492
Epoch 51/200
35/35               10s 277ms/step -
categorical_accuracy: 0.7275 - loss: 0.9562 - val_categorical_accuracy: 0.7239 -
val_loss: 0.9993
Epoch 52/200
35/35               10s 276ms/step -
categorical_accuracy: 0.7306 - loss: 0.9426 - val_categorical_accuracy: 0.7447 -
val_loss: 0.9358
Epoch 53/200
35/35               10s 275ms/step -
categorical_accuracy: 0.7408 - loss: 0.9297 - val_categorical_accuracy: 0.7270 -
val_loss: 0.9683
Epoch 54/200
35/35               10s 276ms/step -
categorical_accuracy: 0.7327 - loss: 0.9448 - val_categorical_accuracy: 0.7485 -
val_loss: 0.9147
Epoch 55/200
35/35               10s 276ms/step -
categorical_accuracy: 0.7440 - loss: 0.9212 - val_categorical_accuracy: 0.7154 -
val_loss: 1.0336
Epoch 56/200
35/35               10s 276ms/step -
categorical_accuracy: 0.7373 - loss: 0.9359 - val_categorical_accuracy: 0.7431 -
val_loss: 0.9425
Epoch 57/200
35/35               10s 276ms/step -
categorical_accuracy: 0.7492 - loss: 0.9045 - val_categorical_accuracy: 0.7469 -
val_loss: 0.9175
Epoch 58/200
35/35               10s 276ms/step -
categorical_accuracy: 0.7406 - loss: 0.9329 - val_categorical_accuracy: 0.7600 -
val_loss: 0.8858
Epoch 59/200
35/35               10s 275ms/step -
categorical_accuracy: 0.7445 - loss: 0.9204 - val_categorical_accuracy: 0.7618 -
val_loss: 0.8791
Epoch 60/200
35/35               10s 276ms/step -
categorical_accuracy: 0.7537 - loss: 0.8957 - val_categorical_accuracy: 0.7572 -
val_loss: 0.8911
Epoch 61/200
```

```
35/35              10s 282ms/step -
categorical_accuracy: 0.7526 - loss: 0.8917 - val_categorical_accuracy: 0.7405 -
val_loss: 0.9545
Epoch 62/200
35/35              10s 277ms/step -
categorical_accuracy: 0.7537 - loss: 0.8950 - val_categorical_accuracy: 0.7794 -
val_loss: 0.8305
Epoch 63/200
35/35              10s 278ms/step -
categorical_accuracy: 0.7583 - loss: 0.8950 - val_categorical_accuracy: 0.7721 -
val_loss: 0.8698
Epoch 64/200
35/35              10s 277ms/step -
categorical_accuracy: 0.7489 - loss: 0.9208 - val_categorical_accuracy: 0.7817 -
val_loss: 0.8228
Epoch 65/200
35/35              10s 276ms/step -
categorical_accuracy: 0.7658 - loss: 0.8681 - val_categorical_accuracy: 0.7810 -
val_loss: 0.8162
Epoch 66/200
35/35              10s 283ms/step -
categorical_accuracy: 0.7637 - loss: 0.8748 - val_categorical_accuracy: 0.7803 -
val_loss: 0.8284
Epoch 67/200
35/35              10s 279ms/step -
categorical_accuracy: 0.7672 - loss: 0.8721 - val_categorical_accuracy: 0.7649 -
val_loss: 0.8990
Epoch 68/200
35/35              10s 280ms/step -
categorical_accuracy: 0.7606 - loss: 0.8828 - val_categorical_accuracy: 0.7831 -
val_loss: 0.8261
Epoch 69/200
35/35              10s 277ms/step -
categorical_accuracy: 0.7715 - loss: 0.8531 - val_categorical_accuracy: 0.7797 -
val_loss: 0.8402
Epoch 70/200
35/35              10s 275ms/step -
categorical_accuracy: 0.7691 - loss: 0.8582 - val_categorical_accuracy: 0.7846 -
val_loss: 0.8255
Epoch 71/200
35/35              10s 276ms/step -
categorical_accuracy: 0.7554 - loss: 0.9015 - val_categorical_accuracy: 0.7361 -
val_loss: 0.9792
Epoch 72/200
35/35              10s 279ms/step -
categorical_accuracy: 0.7696 - loss: 0.8678 - val_categorical_accuracy: 0.7916 -
val_loss: 0.8070
Epoch 73/200
```

```
35/35 10s 277ms/step -
categorical_accuracy: 0.7810 - loss: 0.8397 - val_categorical_accuracy: 0.7863 -
val_loss: 0.8197
Epoch 74/200
35/35 10s 283ms/step -
categorical_accuracy: 0.7832 - loss: 0.8313 - val_categorical_accuracy: 0.7756 -
val_loss: 0.8529
Epoch 75/200
35/35 10s 277ms/step -
categorical_accuracy: 0.7761 - loss: 0.8400 - val_categorical_accuracy: 0.7653 -
val_loss: 0.8979
Epoch 76/200
35/35 10s 275ms/step -
categorical_accuracy: 0.7875 - loss: 0.8236 - val_categorical_accuracy: 0.7888 -
val_loss: 0.8205
Epoch 77/200
35/35 10s 283ms/step -
categorical_accuracy: 0.7870 - loss: 0.8213 - val_categorical_accuracy: 0.7633 -
val_loss: 0.9088
Epoch 78/200
35/35 10s 278ms/step -
categorical_accuracy: 0.7686 - loss: 0.8662 - val_categorical_accuracy: 0.7982 -
val_loss: 0.7967
Epoch 79/200
35/35 10s 277ms/step -
categorical_accuracy: 0.7876 - loss: 0.8192 - val_categorical_accuracy: 0.8027 -
val_loss: 0.7903
Epoch 80/200
35/35 10s 277ms/step -
categorical_accuracy: 0.7895 - loss: 0.8158 - val_categorical_accuracy: 0.7987 -
val_loss: 0.7992
Epoch 81/200
35/35 10s 282ms/step -
categorical_accuracy: 0.7851 - loss: 0.8251 - val_categorical_accuracy: 0.7747 -
val_loss: 0.8888
Epoch 82/200
35/35 10s 276ms/step -
categorical_accuracy: 0.7860 - loss: 0.8339 - val_categorical_accuracy: 0.8035 -
val_loss: 0.7851
Epoch 83/200
35/35 10s 278ms/step -
categorical_accuracy: 0.7864 - loss: 0.8206 - val_categorical_accuracy: 0.8079 -
val_loss: 0.7789
Epoch 84/200
35/35 10s 278ms/step -
categorical_accuracy: 0.7868 - loss: 0.8241 - val_categorical_accuracy: 0.7955 -
val_loss: 0.8236
Epoch 85/200
```

```
35/35            10s 278ms/step -
categorical_accuracy: 0.7976 - loss: 0.8011 - val_categorical_accuracy: 0.8087 -
val_loss: 0.7754
Epoch 86/200
35/35            10s 277ms/step -
categorical_accuracy: 0.7995 - loss: 0.7977 - val_categorical_accuracy: 0.7955 -
val_loss: 0.8017
Epoch 87/200
35/35            10s 283ms/step -
categorical_accuracy: 0.7864 - loss: 0.8298 - val_categorical_accuracy: 0.7912 -
val_loss: 0.8412
Epoch 88/200
35/35            10s 286ms/step -
categorical_accuracy: 0.7878 - loss: 0.8196 - val_categorical_accuracy: 0.8091 -
val_loss: 0.7830
Epoch 89/200
35/35            10s 277ms/step -
categorical_accuracy: 0.8016 - loss: 0.7869 - val_categorical_accuracy: 0.8038 -
val_loss: 0.7951
Epoch 90/200
35/35            10s 280ms/step -
categorical_accuracy: 0.8052 - loss: 0.7818 - val_categorical_accuracy: 0.8000 -
val_loss: 0.8103
Epoch 91/200
35/35            10s 280ms/step -
categorical_accuracy: 0.8034 - loss: 0.7873 - val_categorical_accuracy: 0.8000 -
val_loss: 0.8095
Epoch 92/200
35/35            10s 276ms/step -
categorical_accuracy: 0.8026 - loss: 0.7964 - val_categorical_accuracy: 0.7954 -
val_loss: 0.8324
Epoch 93/200
35/35            10s 277ms/step -
categorical_accuracy: 0.7996 - loss: 0.8053 - val_categorical_accuracy: 0.7967 -
val_loss: 0.8066
Epoch 94/200
35/35            11s 287ms/step -
categorical_accuracy: 0.8029 - loss: 0.7882 - val_categorical_accuracy: 0.8063 -
val_loss: 0.7879
Epoch 95/200
35/35            10s 295ms/step -
categorical_accuracy: 0.8046 - loss: 0.7952 - val_categorical_accuracy: 0.8123 -
val_loss: 0.7751
Epoch 96/200
```

Hemos tenido un problema con la plataforma de `Google Colab`, ya que mientras se realizaba esta ejecución, se agotó nuestro tiempo de ejecución con GPU y se perdieron los datos de los pesos y demás previamente a poder hacer la predicción con las entradas de test.

Pese a este problema podemos asegurar que el modelo mejoraba y no sobreentrenaba dado que aplicamos una fuerte regularización y los valores finales de validación no dejaban que se produjera el `Early Stopping`.

Con todo esto dicho podemos asegurar la eficacia del método de `Data Augmentation` que aumenta el tamaño de las entradas y permite que el modelo sea entrenado con un mayor número de instancias.

[ ]:

[ ]:

[ ]:

3 Conclusión

Modelo	*Acuracy (%)*	*Observacion*	*Porque no fue el definitivo*
1 - Naive (Adam)	69	Exibe un mejor comportamiento que las redes mas sofisticadas de la practica anterior	Aun no aplicamos ninguna tecnica de las estudiadas
2 - RMS Prop	66	Para lo que queda de practica usaremos Adam	No podemos quedarnos con un modelo peor que uno anterior
3 - paddingy Batch-Normalization	79	La mejor hasta el momento. Demostracion del efecto de usar las tecnicas desarrolladas por la investigacion. Un exito.	Aun quedan tecnicas por aplicar
4 - coste en la norma l2	74	Nos hemos pasado con la regularizacion. Aunque ahora sobreentrena mucho menos, tiene un peor comportamiento en test porque la regularizacion tan excesiva no le permitio aprender	Minimo deberiamos igualar el acuracy previo
5 - l2 y dropout	75	ligeramente mejor que solo con l2. De todas formas podria ser por azar. Estos fracasos demuestran la importancia de	Minimo deberiamos igualar el acuracy previo
6 - l2 y dropout v2	81	La mejor hasta el momento. Demostracion del efecto de ajustar propiamente los hiperparametros.	Ahora que ya hemos conseguido superarnos vamos a intentar modificarla para ver si conseguimos algo mejor
7 - l2 y dropout v3	83	La mejor hasta el momento. Ampliamos la red con mas capas convolucionales.	Deberiamos probar data augmentation antes de darnos por conluidos
8 - Data augmentation	81	Muy buen comportamiento. Claramente buena idea	-