Reto

November 21, 2022

1 Reto: Hotel-ID to Combat Human Trafficking 2022

Daniel Salvador Cázares García A01197517

Yoceline Aralí Mata Ledezma A01562116

Yolanda Elizondo Chapa A01137848

Izael Manuel Rascón Durán A01562240

Angel Corrales Sotelo A01562052

2 Contexto del reto

Las víctimas de la trata de personas suelen ser fotografiadas en habitaciones de hotel. La identificación de estos hoteles es vital para las investigaciones sobre la trata de personas, pero esto plantea retos particulares debido a la baja calidad de las imágenes y a los ángulos de cámara poco comunes.

Incluso sin víctimas en las imágenes, la identificación de hoteles en general es una tarea de reconocimiento visual difícil, con un gran número de clases y una variación intraclase potencialmente alta y baja entre clases. Para apoyar la investigación de esta difícil tarea y crear herramientas de búsqueda de imágenes para los investigadores de la trata de personas, creamos la aplicación móvil TraffickCam, que permite a los viajeros cotidianos enviar fotos de su habitación de hotel.

En este concurso, los competidores tienen la tarea de identificar el hotel que se ve en las imágenes de prueba del conjunto de datos de TraffickCam, que se basan en una gran galería de imágenes de entrenamiento con identificaciones de hoteles conocidas.

3 Imports

[17]:
```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from keras import models
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, Dropout,␣
 ↪BatchNormalization
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers
import os, random, shutil
```

```python
from statistics import mean
from random import sample
from shutil import move
```

4 Sobremuestreo y submuestreo

El problema que se enfrentó al dividir los datos en dos sets es que el número de datos por clase variaba mucho, mientras que algunas carpetas tenían más de 1500 imagenes otras tenían 1. Por lo tanto si se entrenaba el modelo de esta forma, este daría más importancia las clases con más datos. Es por esto que se decidió aplicar algunas técnicas como data oversampling y undersampling para regularizar y balancear las clases.

Primer aproximación: Si la clase tenía menos de 17 imagenes entonces se realizaba un aumento de datos añadiendo imagenes ligeramente modificadas creadas a partir de las imagenes ya existentes de la clase (Oversampling). Por el contrario si la clase tenía más de 17 imagenes entonces se aplicó undersampling, en el cual se disminuye las imagenes a un total de 17.

En resumen se buscó que todas las clases tuvieran 17 imagenes, por lo tanto se reducieron o aumentaron la cantidad necesaria para lograr esto en todas las clases y así lograr un balance entre estas.

Segunda aproximación (utilizada): Cuando la clase tenía menos de 12 imágenes esta se elimiaba del conjunto de datos, dejando así solo las clases con más de 11 imáges, a estas se les aplicó sobremuestreo para que el número mínimo de imagenes que pudiera tener una clase fuera 35.

En este caso no se aplicó submuestreo, sin embargo, se utlizaron diferentes pesos para las clases en el momento del entrenamiento, para que no afectara el hecho de que una clase tuviera una mayor cantidad de imágenes que otra.

[2]:
```python
train_datagen = ImageDataGenerator(rescale=1/255,
                                   rotation_range=40,
                                   width_shift_range=0.2,
                                   height_shift_range=0.2,
                                   shear_range=0.2,
                                   zoom_range=0.2,
                                   horizontal_flip=True,
                                   fill_mode='nearest')
```

[3]:
```python
def data_augmentation(target_number, train_dir, dirName):
  '''
  This function generate new images from the dirName folder in the train_dir
  path until it reach the total number of images equal to target_number.

  Args:
    target_number (int): Number of wished images to be total in the dirName
    train_dir (Path): Path of the dirName folder
    dirName (str): Name of the folder containing the sample images for generate
                   new images.

```

```
    Returns:
      None
    '''

    temp_dir = os.path.join(train_dir, dirName)

    img_names = [os.path.join(temp_dir, name) for name in os.listdir(temp_dir)]
    remainder = sample(img_names, k=target_number % len(img_names))
    target_per_img = target_number // len(img_names) - 1
    i, j, k = 0, 0, 0

    for img_path in img_names:
        img = tf.keras.preprocessing.image.load_img(img_path)
        img_array = tf.keras.preprocessing.image.img_to_array(img)
        img_array = img_array.reshape((1,) + img_array.shape)

        for batch in train_datagen.flow(img_array, batch_size=1):
            temp = tf.keras.preprocessing.image.array_to_img(batch[0])
            zeros = 9 - len(str(i + 1))
            name = zeros * '0' + str(i + 1)
            name = os.path.join(temp_dir, name + '.jpg')
            temp.save(name)
            i += 1
            if img_path in remainder:
                j += 1
                if j == (target_per_img + 1):
                    j = 0
                    break
            elif img_path not in remainder:
                k += 1
                if k == target_per_img:
                    k = 0
                    break
```

[4]:
```
images_per_folder = 35

def get_target_number(num_images):
    '''
      This function receives the number of images of a class and returns the␣
 ↪target number of images the class should have when doing data augmentation.

      Args:
        num_images (int): number of images the class currently has.

      Returns:
        None
    '''
```

```
  if num_images * 2 <= images_per_folder:
    target_number = images_per_folder
  else:
    target_number = num_images * 2

  return target_number
```

[5]:
```
import time

def oversampling(train_dir, sampleSize, num_images):
  '''
    This function traverse the dataset and applies data augmentation to every␣
  ↪class.
  '''
  count_timer = 0
  average = 0

  global count
  for dir in os.listdir(train_dir):

    if(num_images[dir] < sampleSize and num_images[dir] > 0):
      t1 = time.time() # empieza
      print(dir, num_images[dir])
      targetNumber = get_target_number(num_images[dir])
      data_augmentation(targetNumber, train_dir, dir)
      count -= 1
      t2 = time.time() # termina

      lasted = t2-t1
      count_timer += 1

      average = (average * (count_timer - 1) + lasted) / count_timer

      estimated_time = average * count


      print("Tiempo estimado: ", estimated_time/60)
      print("Faltan", count)
```

[6]:
```
def subsampling(path, newPath, sampleSize, num_images):
  '''
    Copy all the sub folders with sampleSize number of samples (images) to the
    newPath path. If the number of samples in a subfolder from path is equal to
    the sampleSize, just copy all the samples, if not, takes random samples.

    Args:
```

```
      path (Path): Path with the subfolder of the original samples.
      newPath (Path): Path where we want to copy the samples with the sampleSize
      sampleSize (int): Desire number of samples on the new subfolders.

    Returns:
      None
  '''
  global count2
  for dir in os.listdir(path):
    classPath = os.path.join(path, dir)
    imagesName = os.listdir(classPath)

    newClassPath = os.path.join(newPath, dir)
    num_images = num_images[dir]
    print(dir, num_images[dir])
    if not os.path.exists(newClassPath):
        os.mkdir(newClassPath)
        num = 0
    else:
      num = len(os.listdir(newClassPath))

    target = sampleSize - num

    if(num_images >= sampleSize and num < 17):
      sample = random.sample(imagesName, target)

      for fileN in sample:
        shutil.copy(os.path.join(classPath, fileN), newClassPath)
```

5 Creating training and validation folders

Para poder comprobar el modelo y descartar un posible overfitting se dividieron los datos proporcionados por Kaggle en dos carpetas. Una para entrenamiento y otra para validación.

[7]:
```
proportion = 0.3
base_dir = "dataset2"
train_dir = os.path.join(base_dir,'train_images_balanced')
validation_dir = os.path.join(base_dir, 'validation')

def split_dataset(proportion, train_dir, validation_dir):
  for folder in os.listdir(train_dir):
      path = os.path.join(train_dir, folder)
      new_path = os.path.join(validation_dir, folder)
      os.mkdir(new_path)
      images = [f for f in os.listdir(path)]
```

```
        sampled_images = sample(images, k=int(proportion*len(images)))
        if sampled_images:
            for sampled_image in sampled_images:
                move(os.path.join(path, sampled_image), new_path)
```

[8]:
```
def get_images_per_folder(base_path):

  folders = os.listdir(base_path)
  num_images = {}
  for x in folders:

    num_images[x] = len(os.listdir(os.path.join(base_path, x)))

  return num_images
```

Se puede observar que al crear la subdivisión de carpetas se mantiene el número de clases intactas y lo que varía es la cantidad de datos en cada uno, para el set de entrenamiento se dejo un 70% de los datos y para el de validación el otro 30%.

[25]:
```
images_validation = get_images_per_folder("dataset2/validation")
images_training = get_images_per_folder("dataset2/train_images_balanced")
```

Se asignó un peso distinto a cada clase dependiendo del número de imágenes de estas.

[12]:
```
class_weights = {}
folders = os.listdir(train_dir)
for count, folder in enumerate(folders):
    class_weights[count] = 1.0 / images_training[folder]
```

6 Selección del modelo

Para la selección de modelo se eligió entrenar redes neuronales convolucionales con distintas arquitecturas.

6.1 Transfer Learning

Para el entrenamiento del modelo se utilizó transfer learning con distintas arquitecturas, entre ellas VGG16, VGG19 y Resnet50, de la cual se eligió la que daba mejores resultados, Resnet50

6.2 Arquitectura

La arquitectura que se utilizó fue la siguiente:

6.2.1 Sobreajuste

Para reducir el sobreajuste se utilizó una capa de BatchNormalization

6.3 Entrenamiento

[13]:
```
train_datagen = ImageDataGenerator(rescale=1/255)

train_generator = train_datagen.flow_from_directory(train_dir,
                                                    target_size = (150, 150))

validation_datagen  = ImageDataGenerator(rescale=1/255)

validation_generator  = validation_datagen.flow_from_directory(validation_dir,
                                                    target_size = (150, 150))
```

```
Found 40259 images belonging to 1243 classes.
Found 16451 images belonging to 1243 classes.
```

[14]:
```
from keras.applications import ResNet50
```

[24]:
```
resnet50_conv_base = ResNet50(weights='imagenet',
                  include_top=False,
                  input_shape=(150, 150, 3))
```

[18]:
```
model = models.Sequential([
                          resnet50_conv_base,
                          Flatten(),
                          BatchNormalization(),
                          Dense(512, activation='relu',kernel_regularizer='l2'),
                          Dense(1243, activation='softmax')
                          ])
```

[19]:
```
config={
    "learning_rate": 1e-4,
    "epochs": 50,
    "batch_size": 128,
    "steps_per_epoch": 100,
    "validation_steps": 50,
    "validation_batch_size":128,
    "loss_function": "categorical_crossentropy"
}
```

6.3.1 Compilación

[20]:
```
model.compile(loss=config["loss_function"], optimizer=tf.keras.optimizers.
 ↪Adam(learning_rate=config["learning_rate"]),␣
 ↪metrics=['acc','top_k_categorical_accuracy'])
```

6.3.2 Fit

[21]:
```
history = model.fit(train_generator, steps_per_epoch=config["steps_per_epoch"],
↪, epochs=config["epochs"], validation_data=validation_generator,
↪validation_steps=config["validation_steps"])
```

```
Epoch 1/50
100/100 [==============================] - 100s 822ms/step - loss: 16.9665 -
acc: 0.0422 - top_k_categorical_accuracy: 0.0628 - val_loss: 17.5896 - val_acc:
0.0063 - val_top_k_categorical_accuracy: 0.0088
Epoch 2/50
100/100 [==============================] - 66s 665ms/step - loss: 15.6151 - acc:
0.1403 - top_k_categorical_accuracy: 0.1872 - val_loss: 18.5365 - val_acc:
0.0063 - val_top_k_categorical_accuracy: 0.0181
Epoch 3/50
100/100 [==============================] - 66s 658ms/step - loss: 14.7375 - acc:
0.1450 - top_k_categorical_accuracy: 0.2006 - val_loss: 22.4108 - val_acc:
6.2500e-04 - val_top_k_categorical_accuracy: 0.0069
Epoch 4/50
100/100 [==============================] - 68s 683ms/step - loss: 13.7367 - acc:
0.1955 - top_k_categorical_accuracy: 0.2599 - val_loss: 15.9310 - val_acc:
0.0025 - val_top_k_categorical_accuracy: 0.0162
Epoch 5/50
100/100 [==============================] - 69s 692ms/step - loss: 12.9237 - acc:
0.2275 - top_k_categorical_accuracy: 0.2953 - val_loss: 15.1823 - val_acc:
0.0037 - val_top_k_categorical_accuracy: 0.0137
Epoch 6/50
100/100 [==============================] - 65s 651ms/step - loss: 11.9694 - acc:
0.2841 - top_k_categorical_accuracy: 0.3706 - val_loss: 13.9812 - val_acc:
0.0012 - val_top_k_categorical_accuracy: 0.0075
Epoch 7/50
100/100 [==============================] - 66s 658ms/step - loss: 11.1925 - acc:
0.3250 - top_k_categorical_accuracy: 0.4147 - val_loss: 13.7931 - val_acc:
0.0081 - val_top_k_categorical_accuracy: 0.0194
Epoch 8/50
100/100 [==============================] - 66s 665ms/step - loss: 10.4946 - acc:
0.3616 - top_k_categorical_accuracy: 0.4538 - val_loss: 13.2607 - val_acc:
0.0225 - val_top_k_categorical_accuracy: 0.0500
Epoch 9/50
100/100 [==============================] - 67s 673ms/step - loss: 9.9410 - acc:
0.3873 - top_k_categorical_accuracy: 0.4822 - val_loss: 12.4584 - val_acc:
0.0706 - val_top_k_categorical_accuracy: 0.1056
Epoch 10/50
100/100 [==============================] - 68s 678ms/step - loss: 9.4503 - acc:
0.4072 - top_k_categorical_accuracy: 0.4963 - val_loss: 11.6239 - val_acc:
0.1063 - val_top_k_categorical_accuracy: 0.1575
Epoch 11/50
100/100 [==============================] - 68s 677ms/step - loss: 8.7603 - acc:
```

```
0.4550 - top_k_categorical_accuracy: 0.5531 - val_loss: 10.7443 - val_acc:
0.1488 - val_top_k_categorical_accuracy: 0.2169
Epoch 12/50
100/100 [==============================] - 68s 682ms/step - loss: 8.3617 - acc:
0.4844 - top_k_categorical_accuracy: 0.5738 - val_loss: 10.1324 - val_acc:
0.1912 - val_top_k_categorical_accuracy: 0.2688
Epoch 13/50
100/100 [==============================] - 64s 637ms/step - loss: 7.8459 - acc:
0.5228 - top_k_categorical_accuracy: 0.6091 - val_loss: 9.4617 - val_acc: 0.2313
- val_top_k_categorical_accuracy: 0.3275
Epoch 14/50
100/100 [==============================] - 66s 658ms/step - loss: 7.4047 - acc:
0.5484 - top_k_categorical_accuracy: 0.6397 - val_loss: 9.2638 - val_acc: 0.2256
- val_top_k_categorical_accuracy: 0.3156
Epoch 15/50
100/100 [==============================] - 69s 690ms/step - loss: 7.0140 - acc:
0.5809 - top_k_categorical_accuracy: 0.6666 - val_loss: 8.8844 - val_acc: 0.2525
- val_top_k_categorical_accuracy: 0.3475
Epoch 16/50
100/100 [==============================] - 66s 661ms/step - loss: 6.7401 - acc:
0.5819 - top_k_categorical_accuracy: 0.6800 - val_loss: 8.7402 - val_acc: 0.2481
- val_top_k_categorical_accuracy: 0.3375
Epoch 17/50
100/100 [==============================] - 63s 635ms/step - loss: 6.5160 - acc:
0.5894 - top_k_categorical_accuracy: 0.6922 - val_loss: 8.5039 - val_acc: 0.2438
- val_top_k_categorical_accuracy: 0.3575
Epoch 18/50
100/100 [==============================] - 64s 640ms/step - loss: 6.1183 - acc:
0.6237 - top_k_categorical_accuracy: 0.7163 - val_loss: 8.2504 - val_acc: 0.2544
- val_top_k_categorical_accuracy: 0.3700
Epoch 19/50
100/100 [==============================] - 65s 646ms/step - loss: 5.8839 - acc:
0.6406 - top_k_categorical_accuracy: 0.7400 - val_loss: 8.0600 - val_acc: 0.2562
- val_top_k_categorical_accuracy: 0.3756
Epoch 20/50
100/100 [==============================] - 66s 662ms/step - loss: 5.5739 - acc:
0.6678 - top_k_categorical_accuracy: 0.7616 - val_loss: 7.8097 - val_acc: 0.2700
- val_top_k_categorical_accuracy: 0.3925
Epoch 21/50
100/100 [==============================] - 66s 659ms/step - loss: 5.3370 - acc:
0.6869 - top_k_categorical_accuracy: 0.7819 - val_loss: 7.7403 - val_acc: 0.2625
- val_top_k_categorical_accuracy: 0.3787
Epoch 22/50
100/100 [==============================] - 67s 667ms/step - loss: 5.1863 - acc:
0.6825 - top_k_categorical_accuracy: 0.7809 - val_loss: 7.3473 - val_acc: 0.2969
- val_top_k_categorical_accuracy: 0.4194
Epoch 23/50
100/100 [==============================] - 68s 686ms/step - loss: 4.8959 - acc:
```

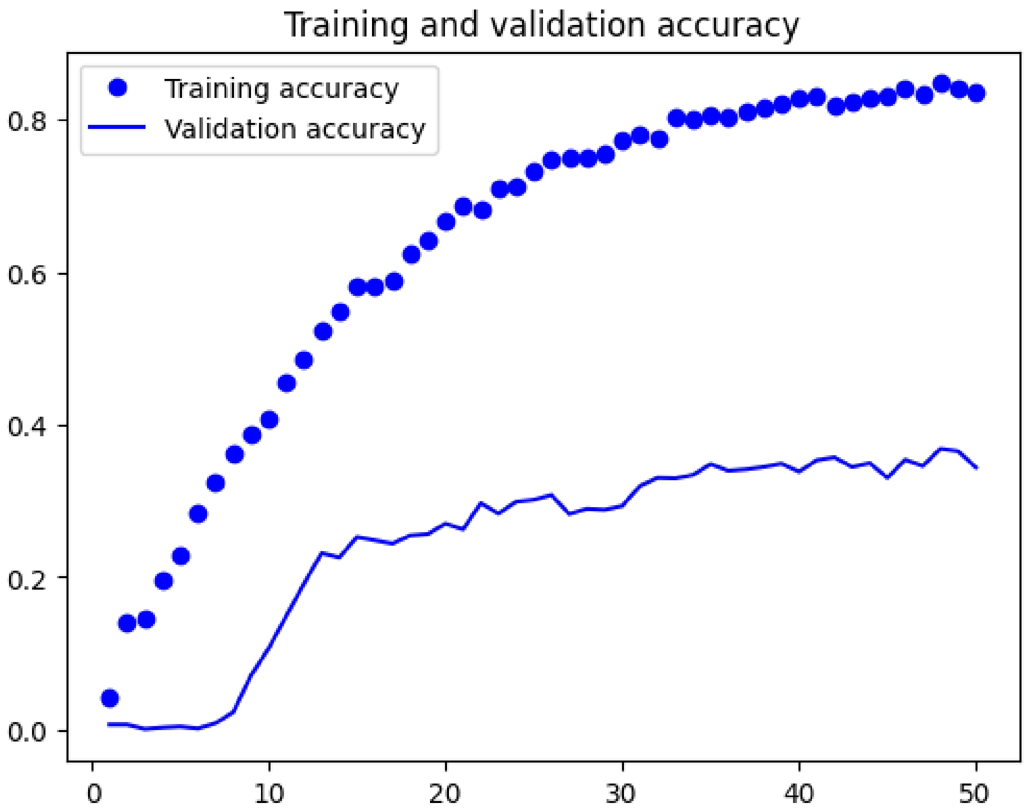
```
0.7106 - top_k_categorical_accuracy: 0.8022 - val_loss: 7.3775 - val_acc: 0.2831
- val_top_k_categorical_accuracy: 0.4100
Epoch 24/50
100/100 [==============================] - 68s 681ms/step - loss: 4.7494 - acc:
0.7125 - top_k_categorical_accuracy: 0.8078 - val_loss: 7.0903 - val_acc: 0.2988
- val_top_k_categorical_accuracy: 0.4406
Epoch 25/50
100/100 [==============================] - 66s 653ms/step - loss: 4.5681 - acc:
0.7334 - top_k_categorical_accuracy: 0.8225 - val_loss: 7.0321 - val_acc: 0.3013
- val_top_k_categorical_accuracy: 0.4369
Epoch 26/50
100/100 [==============================] - 66s 658ms/step - loss: 4.3984 - acc:
0.7472 - top_k_categorical_accuracy: 0.8363 - val_loss: 6.9490 - val_acc: 0.3075
- val_top_k_categorical_accuracy: 0.4306
Epoch 27/50
100/100 [==============================] - 66s 664ms/step - loss: 4.2956 - acc:
0.7494 - top_k_categorical_accuracy: 0.8378 - val_loss: 6.9607 - val_acc: 0.2825
- val_top_k_categorical_accuracy: 0.4275
Epoch 28/50
100/100 [==============================] - 68s 677ms/step - loss: 4.2197 - acc:
0.7506 - top_k_categorical_accuracy: 0.8414 - val_loss: 6.8689 - val_acc: 0.2894
- val_top_k_categorical_accuracy: 0.4300
Epoch 29/50
100/100 [==============================] - 69s 686ms/step - loss: 4.1206 - acc:
0.7550 - top_k_categorical_accuracy: 0.8537 - val_loss: 6.8137 - val_acc: 0.2881
- val_top_k_categorical_accuracy: 0.4331
Epoch 30/50
100/100 [==============================] - 69s 686ms/step - loss: 4.0168 - acc:
0.7719 - top_k_categorical_accuracy: 0.8572 - val_loss: 6.6294 - val_acc: 0.2931
- val_top_k_categorical_accuracy: 0.4594
Epoch 31/50
100/100 [==============================] - 66s 660ms/step - loss: 3.8559 - acc:
0.7816 - top_k_categorical_accuracy: 0.8681 - val_loss: 6.4875 - val_acc: 0.3194
- val_top_k_categorical_accuracy: 0.4669
Epoch 32/50
100/100 [==============================] - 67s 664ms/step - loss: 3.8376 - acc:
0.7744 - top_k_categorical_accuracy: 0.8653 - val_loss: 6.3406 - val_acc: 0.3300
- val_top_k_categorical_accuracy: 0.4719
Epoch 33/50
100/100 [==============================] - 68s 680ms/step - loss: 3.6755 - acc:
0.8037 - top_k_categorical_accuracy: 0.8816 - val_loss: 6.2653 - val_acc: 0.3294
- val_top_k_categorical_accuracy: 0.4744
Epoch 34/50
100/100 [==============================] - 70s 697ms/step - loss: 3.5867 - acc:
0.8012 - top_k_categorical_accuracy: 0.8878 - val_loss: 6.2329 - val_acc: 0.3338
- val_top_k_categorical_accuracy: 0.4825
Epoch 35/50
100/100 [==============================] - 67s 670ms/step - loss: 3.5569 - acc:
```

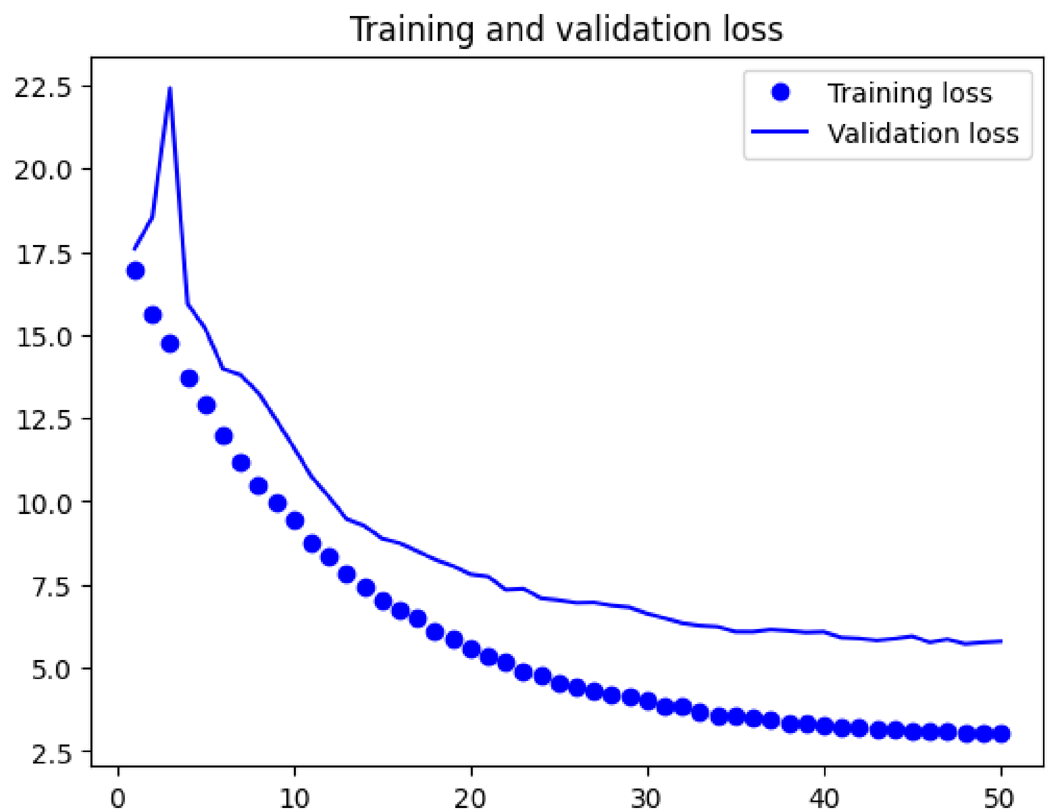
```
0.8047 - top_k_categorical_accuracy: 0.8816 - val_loss: 6.0809 - val_acc: 0.3481
- val_top_k_categorical_accuracy: 0.4969
Epoch 36/50
100/100 [==============================] - 65s 651ms/step - loss: 3.4991 - acc:
0.8034 - top_k_categorical_accuracy: 0.8813 - val_loss: 6.0789 - val_acc: 0.3394
- val_top_k_categorical_accuracy: 0.4875
Epoch 37/50
100/100 [==============================] - 74s 736ms/step - loss: 3.4438 - acc:
0.8094 - top_k_categorical_accuracy: 0.8909 - val_loss: 6.1466 - val_acc: 0.3413
- val_top_k_categorical_accuracy: 0.4900
Epoch 38/50
100/100 [==============================] - 73s 735ms/step - loss: 3.3574 - acc:
0.8169 - top_k_categorical_accuracy: 0.9013 - val_loss: 6.1096 - val_acc: 0.3444
- val_top_k_categorical_accuracy: 0.4881
Epoch 39/50
100/100 [==============================] - 69s 689ms/step - loss: 3.3242 - acc:
0.8197 - top_k_categorical_accuracy: 0.9013 - val_loss: 6.0597 - val_acc: 0.3487
- val_top_k_categorical_accuracy: 0.4900
Epoch 40/50
100/100 [==============================] - 69s 686ms/step - loss: 3.2462 - acc:
0.8291 - top_k_categorical_accuracy: 0.9041 - val_loss: 6.0771 - val_acc: 0.3381
- val_top_k_categorical_accuracy: 0.4825
Epoch 41/50
100/100 [==============================] - 68s 684ms/step - loss: 3.2281 - acc:
0.8316 - top_k_categorical_accuracy: 0.9072 - val_loss: 5.9049 - val_acc: 0.3531
- val_top_k_categorical_accuracy: 0.5056
Epoch 42/50
100/100 [==============================] - 68s 679ms/step - loss: 3.1988 - acc:
0.8194 - top_k_categorical_accuracy: 0.9094 - val_loss: 5.8774 - val_acc: 0.3569
- val_top_k_categorical_accuracy: 0.5000
Epoch 43/50
100/100 [==============================] - 69s 691ms/step - loss: 3.1834 - acc:
0.8244 - top_k_categorical_accuracy: 0.9078 - val_loss: 5.8193 - val_acc: 0.3444
- val_top_k_categorical_accuracy: 0.5125
Epoch 44/50
100/100 [==============================] - 68s 682ms/step - loss: 3.1482 - acc:
0.8275 - top_k_categorical_accuracy: 0.9147 - val_loss: 5.8705 - val_acc: 0.3494
- val_top_k_categorical_accuracy: 0.5006
Epoch 45/50
100/100 [==============================] - 69s 691ms/step - loss: 3.1136 - acc:
0.8309 - top_k_categorical_accuracy: 0.9156 - val_loss: 5.9468 - val_acc: 0.3300
- val_top_k_categorical_accuracy: 0.4900
Epoch 46/50
100/100 [==============================] - 66s 661ms/step - loss: 3.1080 - acc:
0.8409 - top_k_categorical_accuracy: 0.9166 - val_loss: 5.7624 - val_acc: 0.3537
- val_top_k_categorical_accuracy: 0.5225
Epoch 47/50
100/100 [==============================] - 68s 682ms/step - loss: 3.0756 - acc:
```

```
0.8322 - top_k_categorical_accuracy: 0.9200 - val_loss: 5.8541 - val_acc: 0.3456
- val_top_k_categorical_accuracy: 0.5013
Epoch 48/50
100/100 [==============================] - 66s 662ms/step - loss: 3.0380 - acc:
0.8472 - top_k_categorical_accuracy: 0.9181 - val_loss: 5.7151 - val_acc: 0.3681
- val_top_k_categorical_accuracy: 0.5181
Epoch 49/50
100/100 [==============================] - 71s 707ms/step - loss: 3.0125 - acc:
0.8416 - top_k_categorical_accuracy: 0.9259 - val_loss: 5.7582 - val_acc: 0.3650
- val_top_k_categorical_accuracy: 0.5094
Epoch 50/50
100/100 [==============================] - 77s 766ms/step - loss: 3.0234 - acc:
0.8350 - top_k_categorical_accuracy: 0.9172 - val_loss: 5.7901 - val_acc: 0.3438
- val_top_k_categorical_accuracy: 0.5131
```

6.3.3 Visualización de resultados

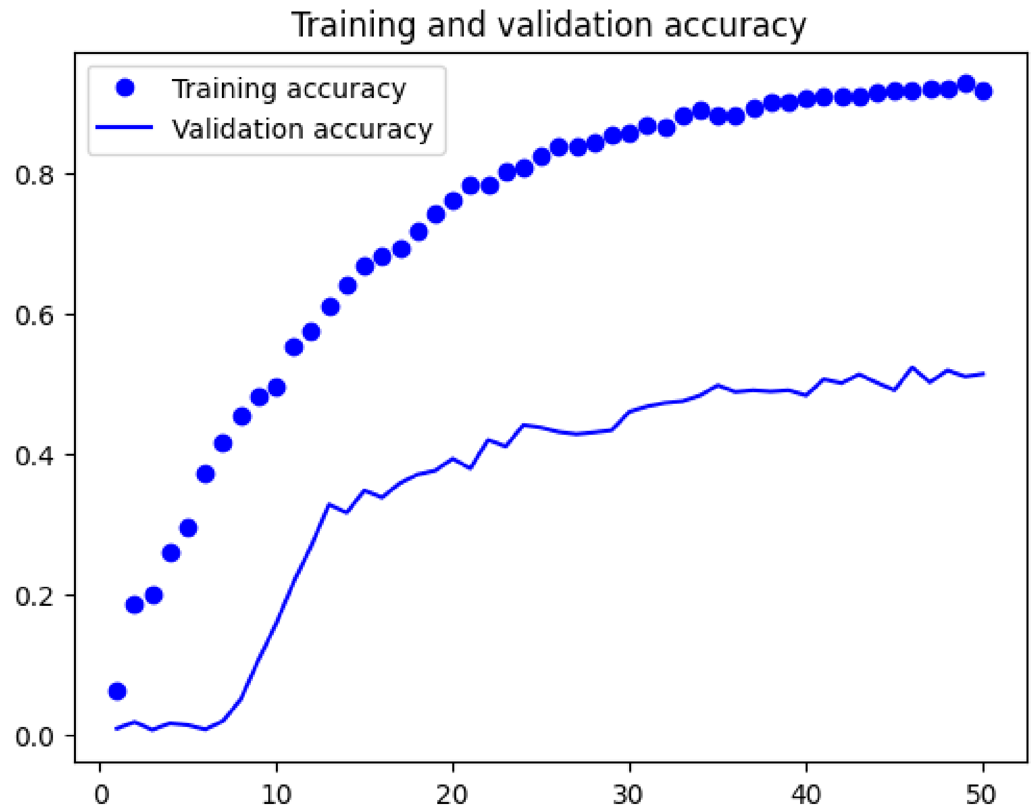
[22]:
```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, 'bo', label='Training accuracy')
plt.plot(epochs, val_acc, 'b', label='Validation accuracy')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

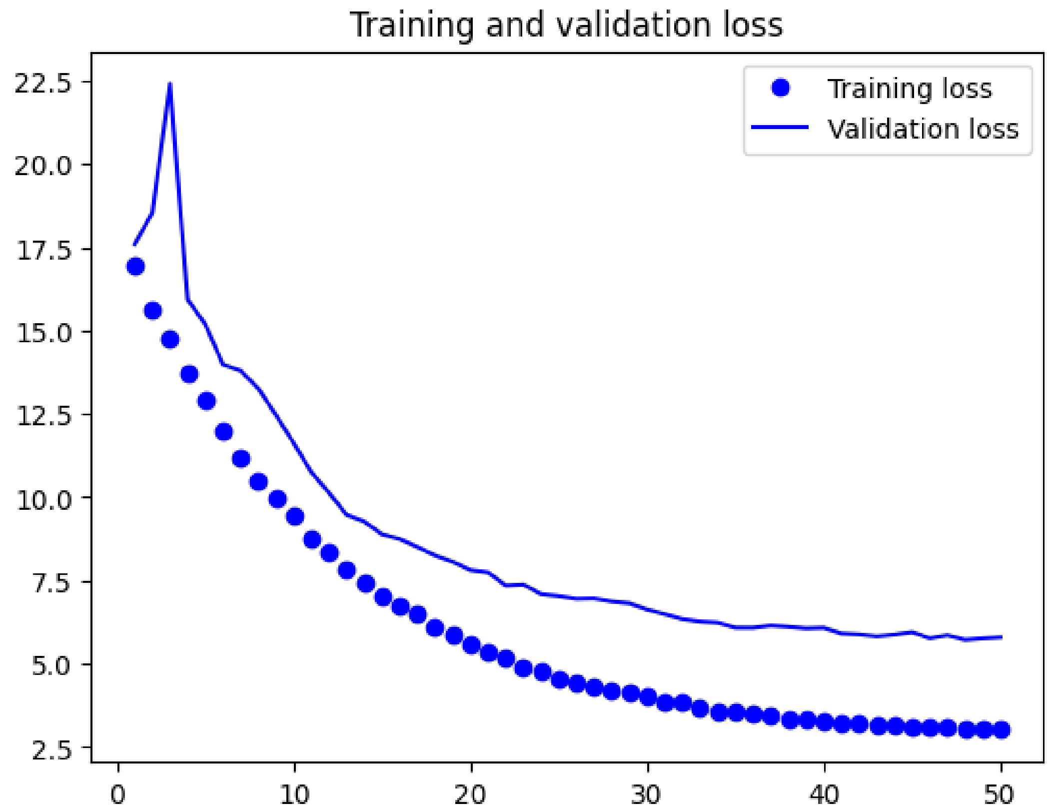

Training and validation accuracy
Training accuracy
Validation accuracy
0.0
0.2
0.4
0.6
0.8
0
10
20
30
40
50



[23]:

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


Training and validation accuracy
Training accuracy
Validation accuracy
0.0
0.2
0.4
0.6
0.8
0
10
20
30
40
50


Training and validation loss
Training loss
Validation loss
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0
10
20
30
40
50