Practica 1: Clasificación Binaria de Imágenes usando Redes Neuronales Convolucionales

9 de noviembre de 2021

Dataset: Aedes Aegipty y Aedes Albopictus

1. Objetivo de la práctica.

Clasificar e identificar dos especies diferentes de mosquitos que pertenecen a la misma familia.

2. Conceptos.

NumPy es una librería de Python especializada en el cálculo numérico y el análisis de datos, usado para trabajar con un gran volumen de datos.

Pandas es una librería de Python cuyo fin es la manipulación y análisis de datos, mediante el uso de Dataframes

Matplotlib es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

TensorFlow es una biblioteca de código abierto para aprendizaje automático desarrollado por Google y que brinda herramientas para crear sistemas capaces de construir y entrenar redes neuronales para detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos.

Keras es una librería de alto nivel de redes neuronales que corre sobre TensorFlow.

El dataset consiste en 2 carpetas con imágenes ".jpg": Aegypti (260) y Albopictus (260) y Ambas (260), que contienen en total 520 elementos.

3. Herramientas a usar.

- Computadora con acceso a internet.
- Los siguientes programas y librerías:

- Python versión 3.8.8
- Anaconda versión 4.10.1
- Jupyter-lab 3.014
- NumPy versión 1.21.4
- Pandas versión 1.2.4
- Matplotlib versión 3.3.4
- Zipfile versión 3.2.0
- Split-folders 0.5.1

4. Desarrollo.

4.1. Entender el problema.

Se tiene un dataset donde se encuentran imágenes de dos especies diferentes de mosquito que pertenecen a la misma familia, pero físicamente tienen diferentes características. Usaremos el poder de cómputo para poder hacer análisis de las imágenes y clasificarlas. La implementación de una red de clasificación binaria es lo más recomendable, puesto que solo son 2 especies a identificar.

4.2. Criterio de evaluación.

Usaremos la exactitud (accuracy)

4.3. Preparar los datos.

Debemos importar todas las librerías con las que estaremos trabajando durante la práctica.

```
import os
import random
import shutil
import zipfile
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing import image
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Estaremos trabajando con Tensorflow, por lo que es necesario comprobar que se esté usando nuestra GPU.

```
tf.config.list_physical_devices('GPU')


[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Aquí definimos la ruta base de nuestros datos. Además, extraeremos las carpetas de los archivos y comprobaremos que se encuentren ahí.

```
Path = "../Practica 1/" #Depende de la ruta en tu m quina

with zipfile.ZipFile(Path+'DataSetAedes.zip', 'r') as archive:
   # Extract all the contents of zip file in current directory
   archive.extractall(Path)

os.listdir(Path+"DataSetAedes")

['Aegypti', 'Albopictus']
```

Ahora dividiremos los datos en 2 carpetas; train y validation para usarlas con el modelo de red neuronal más adelante. Crearemos una carpeta "data"que tendrá las dos carpetas "trainz "valid". Dentro de cada carpeta se encuentran las carpetas .ªegyptiz .ªlbopictus". La cantidad de imágenes dentro de cada carpeta está en una proporción de 80-20.

```
import tqdm
import splitfolders
os.makedirs(Path+"data", exist_ok=True)

splitfolders.ratio(Path+"DataSetAedes", output= Path+"data",
    seed=1337, ratio=(.8, .2), group_prefix=None, move=True)
```

```
os.listdir(Path+"data")

['train', 'val']

os.listdir(Path+"data/train")

['Aegypti', 'Albopictus']
```

Queremos ver cómo son las imágenes por lo que vamos a asignar las carpetas a cariables para poder iterar dentro de ellas y sacar una muestra de imágenes aleatorias de la carpeta de "train". Haremos uso de Matplotlib que, aunque su principal uso es la creación de gráficas, puede mostrar imágenes .jpg en subplots.

```
train_Aegypti = Path+"data/train/Aegypti/"
train_Albopictus = Path+"data/train/Albopictus/"

train_dir_Aegypti = os.listdir(train_Aegypti)
train_dir_Albopictus = os.listdir(train_Albopictus)
```

```
plt.figure(figsize=(15,10))
for i in range(12):
    ran = random.choice(range(0,200))
    rand_Aegypti = [os.path.join(train_Aegypti, f) for f in
    train_dir_Aegypti[ran:ran+1]]
    rand_Albopictus = [os.path.join(train_Albopictus, g) for g in
    train_dir_Albopictus[ran:ran+1]]
    rand = random.choice(rand_Aegypti + rand_Albopictus)
    name = rand.split('/')
    name = 'Moquito ' + name[-2]
    plt.subplot(3, 4, i+1)
    img = plt.imread(rand)
```

```
    plt.imshow(img,cmap = 'gray')
    plt.axis(False)
    plt.title(name)
plt.show()
```

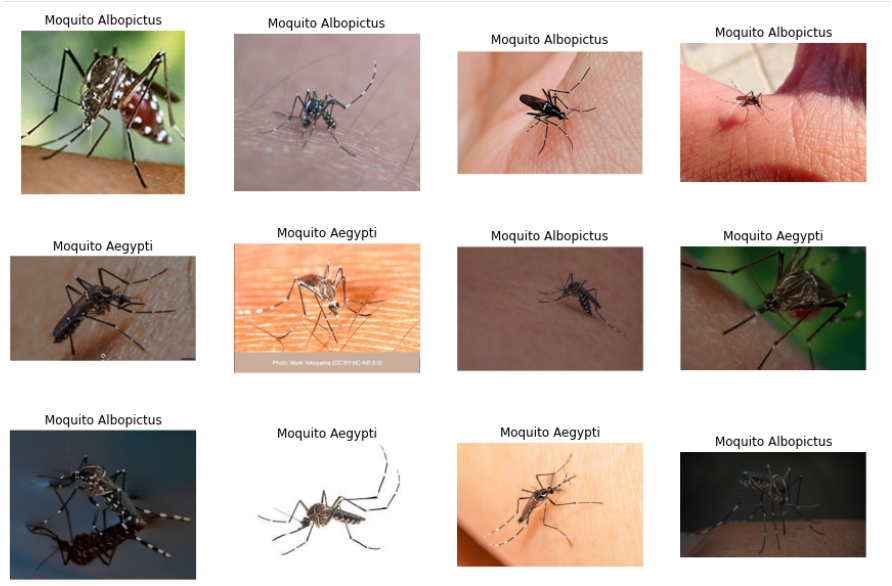

Figura 1: Gráfica con subgráficas presentando una muestra de las imágenes de las carpetas.

Podemos ver que tenemos una variedad de imágenes bastante amplia, algo que podemos ver es que los mosquitos están casi ciempre en el centro de la imagen. Podemos mandar llamar una imagen individual para poder verla con más claridad y ver las dimensiones en píxeles.

```
sample = random.choice(train_dir_Aegypti)
sample = plt.imread(train_Aegypti + sample)
samp = sample.shape
plt.imshow(sample)
print('Sample_Image_size(H*W):', samp)
```

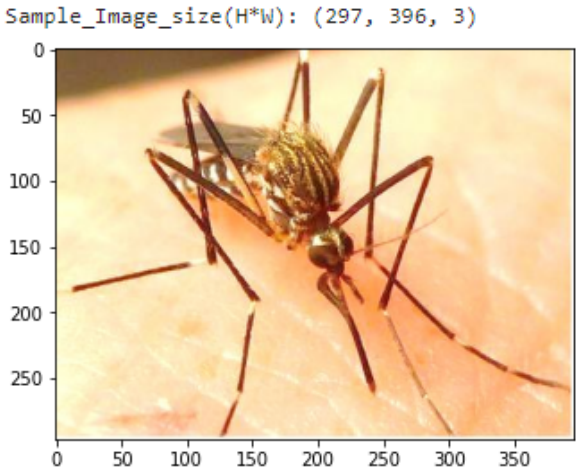

Figura 2: Gráfica con un mosquito y las medidas en pixeles en la parte superior.

Podríamos estar analizando todas las imágenes una por una para comprobar sus dimensiones, pero recurriremos a crear un DataFrame de pandas para poder tener una vista general de los datos de todas las imágenes. Esto con la finalidad de ver cuál es el tamaño promedio de las imágenes.

```
x, y = [], []
for i in train_dir_Aegypti:
    img_array = plt.imread(train_Aegypti + i)
    shape = img_array.shape
    x.append(shape[0])
    y.append(shape[1])
df_shape = pd.DataFrame({'height': x, 'width': y, 'index': range(0,
     len(train_dir_Aegypti))})
```

```
df_shape.head()

    height  width index
0    256    256     0
1    256    256     1
2    256    257     2
3    257    257     3
4    256    256     4
```

```
df_shape.describe()

        height      width      index
count  208.000000  208.000000  208.000000
mean   428.625000  630.552885  103.500000
std      252.104042  411.087749   60.188592
min      176.000000  237.000000    0.000000
25%      295.750000  421.250000   51.750000
50%      364.000000  590.000000  103.500000
75%      476.250000  677.000000  155.250000
max      2177.000000 3794.000000 207.000000
```

Viendo esto ya tenemos un promedio de las dimensiones que podemos manejar en la construcción del modelo.

4.4. Construir el Modelo.

Comenzaremos normalizando los valores de las imágenes para que sean binarios; solo 0 y 1. Esto porque es lo que queremos en realidad; un modelo de clasificación binaria.

```
#Normalizan los valores entre 0 y 1
train = ImageDataGenerator(rescale=1/255)
test = ImageDataGenerator(rescale=1/255)
```

Ahora vamos a generar los dataset de entrenamiento y prueba. Estas vamos a cargarlas haciendo las imágenes de un tamaño estándar cuadrado de 250 p0r 250 píxeles. Esto porque, aunque vimos que hay un tamaño estándar más grande, las imágenes más grandes causan que el entrenamiento tarde más si es que nuestro equipo de computo no es muy potente. El tamaño de batch es la cantidad de imágenes que pasará por paso de época durante el entrenamiento y el modo de clasificación es puesto en binario.

```
train_dataset = train.flow_from_directory(Path+"data/train/",
                                          target_size=(250,250),
                                          batch_size = 8,
                                          class_mode = 'binary')

Found 416 images belonging to 2 classes.

test_dataset = test.flow_from_directory(Path+"data/val/",
                                          target_size=(250,250),
                                          batch_size = 8,
                                          class_mode = 'binary')
Found 104 images belonging to 2 classes.

test_dataset.class_indices
{'Aegypti': 0, 'Albopictus': 1}
```

Ahora para poder hacer nuestro modelo lo qe debemos hacer es buscar características importantes de las imágenes. Podemos extraer estas características usando una red con capas de convolución y maxpooling. Teniendo eso en cuenta, creamos la estructura de la red neuronal así:

1. La primera capa será la que va a recibir las imágenes, por lo que usaremos una capa Convolucional. Las capas Convoluiconales se encargan de resaltar las patrones y caraterísticas importantes de las imágenes y reducirlas"pasando solo esta características al siguiente nivel

2. La capa de Maxpooling se encargará de pasar esas características más importantes para seguirlas reduciendo

3. Este proceso se repetirá por las siguientes 2 capas hasta llegar a la capa de Flatten donde reducirá las capas anteriores, que están en "forma de matriz", y las pasará a un vector.

4. Este vector se pasará a una capa Densa completamente conectada donde se dará a un neutron de salida con una función de activación sigmoide que clasificará en una de las dos clases la imagen alimentada a la red.

```
model = tf.keras.models.Sequential([
    # since Conv2D is the first layer of the neural network, we
    should also specify the size of the input
    tf.keras.layers.Conv2D(16, (3,3), activation='relu',
    input_shape=(250, 250, 3)),
    # apply pooling
    tf.keras.layers.MaxPooling2D(2,2),
    # and repeat the process
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # flatten the result to feed it to the dense layer
    tf.keras.layers.Flatten(),
    # and define 512 neurons for processing the output coming by
    the previous layers
    tf.keras.layers.Dense(512, activation='relu'),
    # a single output neuron. The result will be 0 if the image is
    an Aegypti, 1 if it is an Albopictus
    tf.keras.layers.Dense(1, activation='sigmoid')
])

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 248, 248, 16)      448

 max_pooling2d_3 (MaxPooling  (None, 124, 124, 16)     0
 2D)

 conv2d_4 (Conv2D)           (None, 122, 122, 32)      4640

 max_pooling2d_4 (MaxPooling  (None, 61, 61, 32)       0
 2D)

 conv2d_5 (Conv2D)           (None, 59, 59, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 29, 29, 64)       0
 2D)

 flatten_1 (Flatten)         (None, 53824)             0

 dense_2 (Dense)             (None, 512)               27558400

 dense_3 (Dense)             (None, 1)                 513

=================================================================
Total params: 27,582,497
Trainable params: 27,582,497
Non-trainable params: 0
_________________________________________________________________
```

Figura 3: Esta es la estructura de la red neuronal con la que se van a entrenar con las imágenes.

Ya con la estrucura de la red creada solo nos queda mandar a llamar el entrenamiento. Este lo realizaremos haciendo uso de la función de model.fit donde se le dirá a la red con qué imágenes se le alimentará, cuántas veces tendrá que hacer el entrenamiento y qué imágenes usará para corrobar si su clasificación es correcta.

```
model.compile(optimizer="adam",
              loss='binary_crossentropy',
              metrics = ['accuracy'])

history = model.fit(
            train_dataset, # pass in the training generator
            steps_per_epoch=50,
            epochs=15,
            validation_data=test_dataset, # pass in the validation
    generator
            validation_steps=13,
            verbose=2
            )
```

```
Epoch 1/15
50/50 - 5s - loss: 0.7722 - accuracy: 0.6050 - val_loss: 0.5843 - val_accuracy: 0.7885 - 5s/epoch - 95ms/step
Epoch 2/15
50/50 - 2s - loss: 0.5216 - accuracy: 0.7250 - val_loss: 0.4173 - val_accuracy: 0.7692 - 2s/epoch - 45ms/step
Epoch 3/15
50/50 - 2s - loss: 0.3259 - accuracy: 0.8625 - val_loss: 0.2845 - val_accuracy: 0.8654 - 2s/epoch - 46ms/step
Epoch 4/15
50/50 - 2s - loss: 0.3123 - accuracy: 0.8875 - val_loss: 0.2713 - val_accuracy: 0.8942 - 2s/epoch - 46ms/step
Epoch 5/15
50/50 - 2s - loss: 0.1813 - accuracy: 0.9375 - val_loss: 0.2337 - val_accuracy: 0.9135 - 2s/epoch - 46ms/step
Epoch 6/15
50/50 - 2s - loss: 0.0755 - accuracy: 0.9700 - val_loss: 0.1430 - val_accuracy: 0.9423 - 2s/epoch - 47ms/step
Epoch 7/15
50/50 - 2s - loss: 0.0517 - accuracy: 0.9900 - val_loss: 0.1298 - val_accuracy: 0.9519 - 2s/epoch - 46ms/step
Epoch 8/15
50/50 - 3s - loss: 0.0114 - accuracy: 1.0000 - val_loss: 0.1141 - val_accuracy: 0.9519 - 3s/epoch - 50ms/step
Epoch 9/15
50/50 - 3s - loss: 0.0082 - accuracy: 1.0000 - val_loss: 0.2604 - val_accuracy: 0.9615 - 3s/epoch - 53ms/step
Epoch 10/15
50/50 - 2s - loss: 0.0018 - accuracy: 1.0000 - val_loss: 0.1783 - val_accuracy: 0.9615 - 2s/epoch - 45ms/step
Epoch 11/15
50/50 - 2s - loss: 5.5160e-04 - accuracy: 1.0000 - val_loss: 0.1808 - val_accuracy: 0.9615 - 2s/epoch - 46ms/step
Epoch 12/15
50/50 - 2s - loss: 3.1428e-04 - accuracy: 1.0000 - val_loss: 0.1872 - val_accuracy: 0.9615 - 2s/epoch - 48ms/step
Epoch 13/15
50/50 - 2s - loss: 2.2375e-04 - accuracy: 1.0000 - val_loss: 0.1888 - val_accuracy: 0.9615 - 2s/epoch - 47ms/step
Epoch 14/15
50/50 - 2s - loss: 1.7200e-04 - accuracy: 1.0000 - val_loss: 0.1922 - val_accuracy: 0.9615 - 2s/epoch - 46ms/step
Epoch 15/15
50/50 - 2s - loss: 1.3581e-04 - accuracy: 1.0000 - val_loss: 0.1963 - val_accuracy: 0.9615 - 2s/epoch - 45ms/step
```

Figura 4: Aquí se ve el progreso del entrenamiento durante las 15 épocas. Así como la evolución de la exactitud y la perdida en el entrenamiento y la validación

Una vez entrenado el modelo podemos graficar el entrenamiento para poder analizarlo de una manera más precisa.

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

epochs = range(len(acc))

# plot accuracy with matplotlib
plt.plot(epochs, acc)
plt.plot(epochs, val_acc)
plt.title('Accuracy in training and validation')
plt.figure()

# plot loss with matplotlib
plt.plot(epochs, loss)
plt.plot(epochs, val_loss)
plt.title('Loss in training and validation')
```

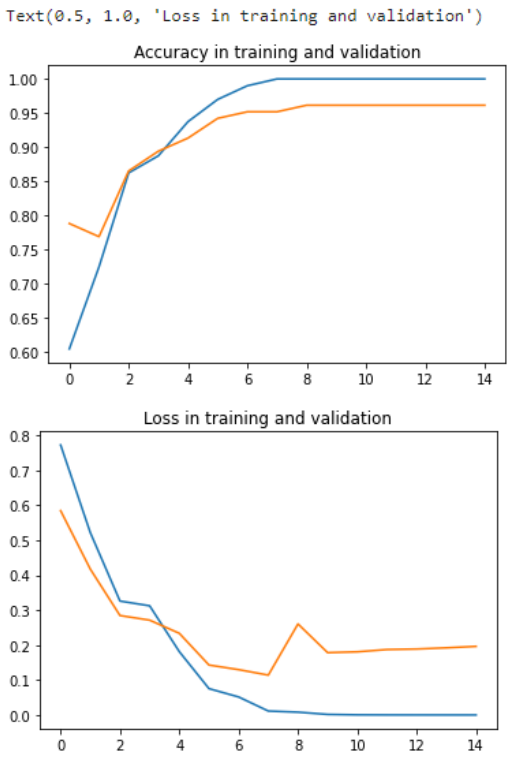

Figura 5: Las gráficas que muestran la exactitud y pérdida del entrenamiento del modelo en el conjunto de entrenamiento y validación.

4.5. Análisis de errores.

Como se puede ver, la clasificación del modelo es muy buena. La exactitud resulta ser bastante acertada, por lo que ahora debemos de corroborar el entrenamiento de la red neuronal. Esto podemos hacerlo cargándole una imagen de las que tengamos guardadas para que la clasifique y veamos en cuál de los dos grupos.

```
def predictImage(filename):
    img1 = image.load_img(filename,target_size=(250,250))

    plt.imshow(img1)

    Y = image.img_to_array(img1)

    X = np.expand_dims(Y,axis=0)
    val = model.predict(X)
    print(val)
    if val == 1:

        plt.xlabel("Albopictus",fontsize=30)


    elif val == 0:

        plt.xlabel("Aegypti",fontsize=30)

predictImage(Path + "Aedes3.jpg")
```

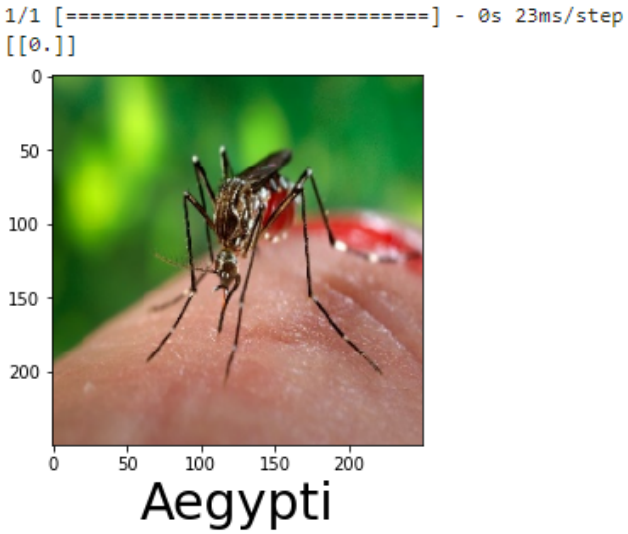

Figura 6: La imagen fue cargada al modelo y la predicción resultante logró ser acertada.

Aunque los resultados de la predicción y el entrenamiento del modelo resultaron ser satisfactorios, es puntual señalar que la cantidad de imágenes con las que se trabajo es reducida, por lo que nos estamos frente a un posible caso de sobreajuste. Esto se da por sobreentrenar la red y es causado por la cantidad limitada de información para trabajar.

Es necesario buscar una fuente de imágenes más grande o encontrar un método para aumentar la cantidad de datos para presentarlos a la red.

4.6. Implementación.

Consulta el Notebook de Jupyter en la carpeta de **Practica 1** llamado ***TF-Mosquitos.ipynb***