ANEXO

9 de noviembre de 2021

Practica 1: Clasificación Binaria de Imágenes usando Redes Neuronales Convolucionales

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

Seaborn es una librería de visualización de daos para Python desarrollada sobre matplotlib. Ofrece una interfaz de alto nivel para dibujar gráficas estadísticas atractivas e informativas.

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.

Pytorch es un paquete de Python diseñado para realizar cálculos numéricos haciendo uso de la programación de tensores. Además permite su ejecución en GPU para acelerar los cálculos.

La librería scikit-learn, también llamada sklearn, es un conjunto de rutinas escritas en Python para hacer análisis predictivo, que incluyen clasificadores, algoritmos de clusterización, etc.

El dataset consiste en 3 carpetas con imágenes ".jpgz archivos ".xml": Aegypti (520), Albopictus (520) y Ambas (56), que contienen en total 1096 elementos, de los cuales 548 son imágenes y 548 son archivos xml.

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
  - Seaborn versión 0.11.2
  - Matplotlib versión 3.3.4
  - Pytorch versión 1.9.0

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 precisión, recall, puntuación f1 y matríces de confusión.

4.3. Iniciando la práctica.

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

```
import os
import path
import shutil
import zipfile

import numpy as np
import pandas as pd
import seaborn as sns
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt

import torch
import torchvision
import torch.nn as nn
```

```
import torch.optim as optim
import torch.nn.functional as F
from torchvision import transforms, utils, datasets
from torch.utils.data import Dataset, DataLoader

from sklearn.metrics import classification_report, confusion_matrix
```

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

Establecemos la semilla aleatoria de manera manual si usamos CPU o le encargamos a CUDA que establezca la misma semilla si se usa GPU

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu
    ")
print("We're using =>", device)

torch.manual_seed(1234)
if device =='cuda':
    torch.cuda.manual_seed_all(1234)


We're using => cuda
```

Aquí definimos la ruta base de nuestros datos.

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

with zipfile.ZipFile(Path+'DataSetAedes.zip', 'r') as archive:
   # Extrae todo el contenido del achivo ZIP en el directorio
    actual
   archive.extractall(Path)

os.listdir(Path+"DataSetAedes")

['Aegypti', 'Albopictus', 'Ambos', 'DataSetAedes']
```

4.4. Definimos las transformaciones de las imágenes.

Definamos un diccionario para contener las transformaciones de imagen para conjuntos de entrenamiento, prueba y validación

Cambiaremos el tamaño de todas las imágenes para que tengan tamaño (224, 224) y también convertiremos las imágenes a tensor.

La función ToTensor de Pytorch convierte todos los tensores para que se encuentren entre (0, 1)

Esta función convierte una imagen PIL o un array de numpy numpy.ndarray de forma (H x W x C) "Height x Width x Channel.[en] el rango [0, 255] a un tensor de flotantes de Pytorch torch.FloatTensor de la forma (C x H x W) en el rango [0.0, 1.0]

```
image_transforms = { "train": transforms.Compose([transforms.Resize
    ((224, 224)),transforms.ToTensor()]),
                     "test": transforms.Compose([transforms.Resize
    ((224, 224)),transforms.ToTensor()]),
                     "val": transforms.Compose([transforms.Resize
    ((224, 224)),transforms.ToTensor()])}
```

Usaremos la librería splitfolder para poder separar nuestros datos en train, test y validation de manera automática.

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

splitfolders.ratio(Path+"DataSetAedes", output= Path+"data", seed
    =1, ratio=(.8, 0.1,0.1), group_prefix=2)
```

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

['test', 'train', 'val']
```

Para esta actividad específicamente, no tomaremos en cuenta la carpeta de archivos que contiene los collages de ambas especies de mosquitos. Las usaremos en otra ocasión.

```
for folders in os.listdir(Path+"data"):
    shutil.rmtree(Path+"data/"+folders+"/"+ "Ambos")
    os.rmdir(Path+"data/"+folders+"/"+ "DataSetAedes")
```

4.5. Trabajando con los Datasets.

Aquí inicializamos los Datasets usando las carpetas correspondientes y aplicamos sus transformaciones.

```

mosquitos_dataset_train = datasets.ImageFolder(root = Path + "data/
    train/",
                                           transform =
    image_transforms["train"])

mosquitos_dataset_test = datasets.ImageFolder(root = Path + "data/
    test/",
                                           transform =
    image_transforms["test"])

mosquitos_dataset_val = datasets.ImageFolder(root = Path + "data/
    val/",
                                           transform =
    image_transforms["val"])

print(mosquitos_dataset_train)
print(mosquitos_dataset_test)
print(mosquitos_dataset_val)

Dataset ImageFolder
    Number of datapoints: 416
    Root location: ../Practica 1/data/train/
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=bilinear)
               ToTensor()
           )
Dataset ImageFolder
    Number of datapoints: 52
```

```
    Root location: ../Practica 1/data/test/
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=bilinear)
               ToTensor()
           )
Dataset ImageFolder
    Number of datapoints: 52
    Root location: ../Practica 1/data/val/
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=bilinear)
               ToTensor()
           )
```

La función class_to_idx está predefinida en PyTorch. Devuelve los IDs de clase presentes en el conjunto de datos.

```
mosquitos_dataset_train.class_to_idx

{'Aegypti': 0, 'Albopictus': 1}
```

Ahora haremos un mapeo de ID a clase.

```
idx2class = {v: k for k, v in mosquitos_dataset_train.class_to_idx.
    items()}
```

Ahora haremos una función que tomará un objeto de clase Dataset y regresará un diccionario con el conteo de muestras d clase. Usaremos este diccionario para construir unas gráficas y observar la distribución de clases de nuestros datos.

La función get_class_distribution() recibe un argumento llamado dataset_obj

- Primero inicializamos el diccionario count_dict donde el conteo de todas nuestras clases comienza en 0.
- Después, iteraremos a través de todo el dataset e iremos incrementando el contador por 1 por cada que nos encontremos con una nueva clase.

plot_from_dict() recibe 3 argumentos: un diccionario dict_obj, plot_title, y **kwargs que serán usados después para construir subgráficas usando Seaborn

- Primero convertimos el diccionario en un dataframe.
- Juntamos el dataframe y hacemos la gráfica.

```
def get_class_distribution(dataset_obj):
    count_dict = {k:0 for k,v in dataset_obj.class_to_idx.items()}
    for _, label_id in dataset_obj:
        label = idx2class[label_id]
        count_dict[label] += 1
    return count_dict
def plot_from_dict(dict_obj, plot_title, **kwargs):
    return sns.barplot(data = pd.DataFrame.from_dict([dict_obj]).
    melt(), x = "variable", y="value", hue="variable", **kwargs).
    set_title(plot_title)
```

```
plt.figure(figsize=(15,8))
plot_from_dict(get_class_distribution(mosquitos_dataset_train),
    plot_title="Dataset train elements)")
```

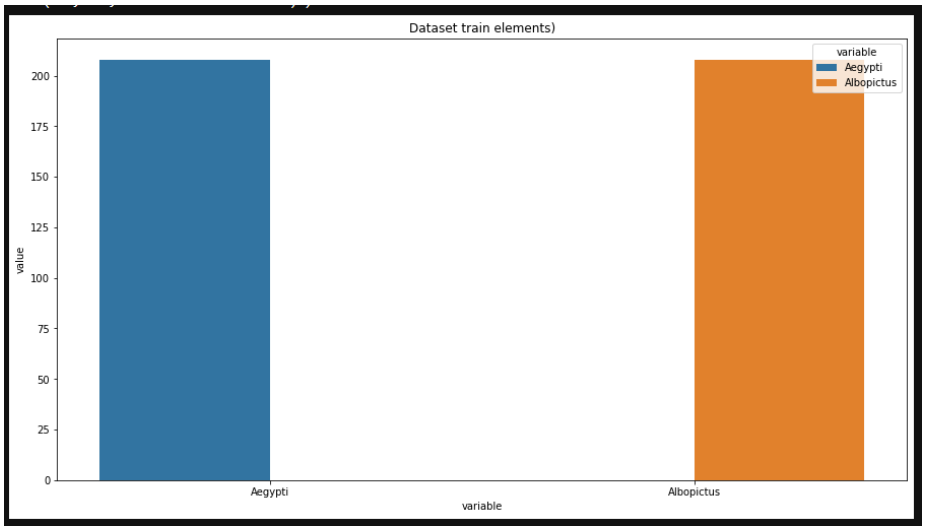


Figura 1: Tabla que muestra la separación de las dos clases de imágenes que tenemos.

Creamos los Dataloaders para cada uno de nuestros sets procurando poner la opción de shuffle=True para que nuestras imágenes no se encuentren en un orden específico

```
train_loader = DataLoader(dataset=mosquitos_dataset_train, shuffle=
    True, batch_size=8)

test_loader = DataLoader(dataset=mosquitos_dataset_test, shuffle=
    True, batch_size=1)

val_loader = DataLoader(dataset=mosquitos_dataset_val, shuffle=True
    , batch_size=1)
```

4.6. Explorando las imágenes.

Una vez cargados nuestros Datasets, podemos explorarlos un poco. Para ellos, creamos una función que tomará los dataloaders y regresará un diccionario con el conteo de clases, parecido al creado anteriormente.

- Inicializamos un diccionario count_dict con puros 0's.
- Si el batch_size de dataloader_obj es 1, entonces recorre el dataloader_obj y actualiza el contador.

- Si el batch_size de dataloader_obj NO es 1, entonces recorre el dataloader_obj para obtener el tamaño de los batches. Después recorre los batches para obtener los tensores individuales. Después, actualiza el contador.

```
def get_class_distribution_loaders(dataloader_obj, dataset_obj):
    count_dict = {k:0 for k,v in dataset_obj.class_to_idx.items()}
    if dataloader_obj.batch_size == 1:
        for _,label_id in dataloader_obj:
            y_idx = label_id.item()
            y_lbl = idx2class[y_idx]
            count_dict[str(y_lbl)] += 1
    else:
        for _,label_id in dataloader_obj:
            for idx in label_id:
                y_idx = idx.item()
                y_lbl = idx2class[y_idx]
                count_dict[str(y_lbl)] += 1
    return count_dict
```

Ahora mostraremos la distribución de clases usando la función plot_from_dict() que definimos anteriormente.

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(18,7))
plot_from_dict(get_class_distribution_loaders(train_loader,
    mosquitos_dataset_train), plot_title="Train Set", ax=axes[0])
plot_from_dict(get_class_distribution_loaders(val_loader,
    mosquitos_dataset_val), plot_title="Val Set", ax=axes[1])
```

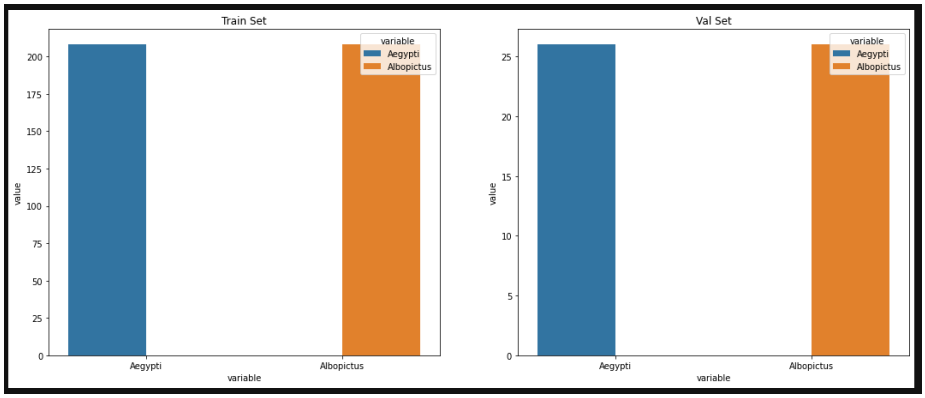


Figura 2: Gráficos de distribución de las imágenes.

Ahora que hemos comprobado la distribución de clases, veremos una sola imagen.

```
single_batch = next(iter(train_loader))
```

single_batch es una lista de 2 elementos. El primero es el contiene el tensor de la imagen mientras que el segundo elemento tiene las etiquetas.

La forma del primer elemento del tensor es (Batch, channels, height, width)

```
single_batch[0].shape

torch.Size([8, 3, 224, 224])
```

Y estas son las etiquetas de salida de los batches.

```
print("Output label tensors: ", single_batch[1])
print("\nOutput label tensor shape: ", single_batch[1].shape)

Output label tensors:  tensor([1, 1, 1, 1, 0, 0, 1, 0])
Output label tensor shape:  torch.Size([8])
```

Para mostrar la imagen, usaremos plt.imsow de matplotlib. La función espera que las dimensiones de la imagen sean (height, width, channels). Usaremos la función de .permute() en el tensor para poder "voltear"las dimensiones y poder mostrarla en una gráfica.

```
single_image = single_batch[0][0]
single_image.shape

torch.Size([3, 224, 224])
```

Definimos el estilo de Seaborn,

```
%matplotlib inline
sns.set_style('darkgrid')

plt.imshow(single_image.permute(1, 2, 0))
```

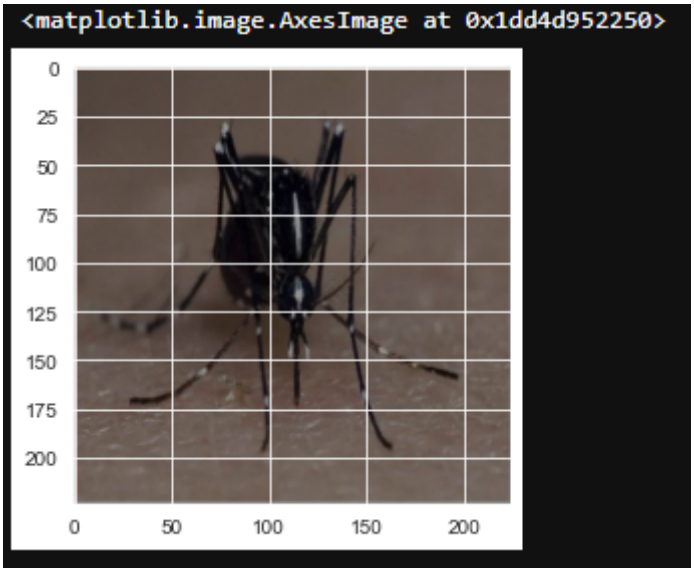


Figura 3: Imagen ploteada con Seaborn y separada en pequeños segmentos.

PyTorch nos ha facilitado trazar las imágenes en una cuadrícula directamente desde el batch.

Primero extraemos el tensor de imagen de la lista (devuelto por nuestro dataloader) y establecemos nrow. Luego usamos la función plt.imshow() para graficar nuestra cuadrícula. No olvidemos la función .permute() para trabajar con las dimensiones del tensor conrrectas.

```
single_batch_grid = utils.make_grid(single_batch[0], nrow=4)
plt.figure(figsize = (10,10))
plt.imshow(single_batch_grid.permute(1, 2, 0))
```

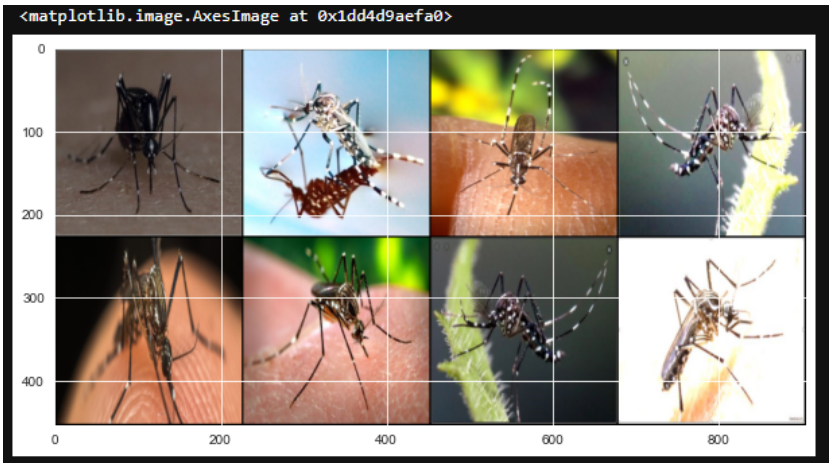


Figura 4: Batch separado en cuadrícula.

4.7. Definir la arquitectura de la CNN.

Nuestra arquitectura es simple. Usamos 4 bloques de capas Convolucionales. Cada bloque consiste en las siguientes capas;

Convolución + BatchNorm + ReLU + Dropout

```
class MosquitosClassifier(nn.Module):
  def __init__(self):
    super(MosquitosClassifier, self).__init__()
      self.block1 = self.conv_block(c_in=3, c_out=256, dropout=0.1,
                                    kernel_size=5, stride=1,
    padding=2)
      self.block2 = self.conv_block(c_in=256, c_out=128, dropout
    =0.1,
                                    kernel_size=3, stride=1,
    padding=1)
      self.block3 = self.conv_block(c_in=128, c_out=64, dropout
    =0.1,
                                    kernel_size=3, stride=1,
    padding=1)
      self.lastcnn = nn.Conv2d(in_channels=64, out_channels=2,
```

```
                                kernel_size=56, stride=1, padding=0)
      self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
  def forward(self, x):
    x = self.block1(x)
    x = self.maxpool(x)
    x = self.block2(x)
    x = self.block3(x)
    x = self.maxpool(x)
    x = self.lastcnn(x)
    return x
  def conv_block(self, c_in, c_out, dropout,  **kwargs):
    seq_block = nn.Sequential(
      nn.Conv2d(in_channels=c_in, out_channels=c_out, **kwargs),
      nn.BatchNorm2d(num_features=c_out),
      nn.ReLU(),
      nn.Dropout2d(p=dropout)
    )
    return seq_block
```

Ahora inicializaremos el modelo, el optimizador y la función de pérdida.

Luego, transferiremos el modelo a la GPU.

Usaremos la nn.CrossEntropyLoss aunque es un problema de clasificación binaria. Esto significa que, en lugar de devolver una única salida de 0/1, trataremos la devolución de 2 valores de 0 a 1. Más específicamente, las probabilidades de que la salida sea 0 o 1.

No tenemos que aplicar manualmente una capa log_softmax después de nuestra capa final porque la nn.CrossEntropyLosshace por nosotros.

Pero debemos aplicar la capa log_softmax para la valicadión y prueba final.

```
model = MosquitosClassifier()
model.to(device)
print(model)
optimizer = optim.SGD(model.parameters(), lr=0.008)
criterion = nn.CrossEntropyLoss()
```

Antes de comenzar nuestro entrenamiento, definamos una función para calcular la precisión por época.

Esta función toma y_pred y y_testcomo argumentos de entrada. Luego aplicamos softmax y_predy extraemos la clase que tiene una mayor probabilidad.

Después de eso, comparamos las clases predichas y las clases reales para calcular la precisión.

```
def binary_acc(y_pred, y_test):
    y_pred_tag = torch.log_softmax(y_pred, dim = 1)
    _, y_pred_tags = torch.max(y_pred_tag, dim = 1)
    correct_results_sum = (y_pred_tags == y_test).sum().float()
    acc = correct_results_sum/y_test.shape[0]
    acc = torch.round(acc * 100)
    return acc
```

También definiremos 2 diccionarios que almacenarán la precisión/época y la pérdida/época para los conjuntos de entrenamiento y validación.

```
accuracy_stats = {
    'train': [],
```

```
    "val": []
}
loss_stats = {
    'train': [],
    "val": []
}
```

Hemos puesto un model.train() antes del bucle. model.train() le dice a PyTorch que estás en modo de entrenamiento. Bueno, ¿por qué tenemos que hacer eso? Si está usando capas como Dropout o BatchNorm que se comportan de manera diferente durante el entrenamiento y la evaluación (por ejemplo; no usar dropoutdurante la evaluación), debe decirle a PyTorch que actúe en consecuencia. Si bien el modo predeterminado en PyTorch es el entremaniento, no es necesario que lo escriba explícitamente. Pero es buena práctica.

De manera similar, llamaremos model.eval() cuando probemos nuestro modelo. Lo veremos a continuación. Regresando con el entrenamiento; comenzamos un ciclo for. En la parte superior de este ciclo for, inicializamos nuestra loss y accuracy por época en 0. Después de cada época, imprimiremos la loss/accuracy y la restableceremos a 0.

Luego tenemos otro ciclo for. Este bucle for se utiliza para obtener nuestros datos batches de train_loader.

Aplicamos optimizer.zero_grad() antes de hacer predicciones. Dado que la función .backward() acumula gradientes, debemos establecerla en 0 manualmente por minibatches. A partir de nuestro modelo definido, obtenemos una predicción, obtenemos la loss (y accuracy) para ese minibatch, realizamos la propagación hacia atrás usando loss.backward() y optimizer.step().

Finalmente, agregamos todas las loss (y accuracy) de minibatches para obtener la pérdida promedio (y precisión) para esa época. Sumamos todas las loss/accuracy para cada minibatch y finalmente lo dividimos por el número de minibatches, es decir, longitud del trainloader para obtener la loss/accuracy promedio por época.

El procedimiento que seguimos para el entrenamiento es exactamente el mismo para la validación, excepto por el hecho de que lo concluimos con la función torch.no_grad y no realizamos ninguna propagación hacia atrás. torch.no_grad le dice a PyTorch que no queremos realizar una retropropagación, lo que reduce el uso de memoria y acelera el cálculo.

```

print("Begin training.")
for e in tqdm(range(1, 21)):
    # TRAINING
    train_epoch_loss = 0
    train_epoch_acc = 0
    model.train()
    for X_train_batch, y_train_batch in train_loader:
        X_train_batch, y_train_batch = X_train_batch.to(device),
    y_train_batch.to(device)
        optimizer.zero_grad()
        y_train_pred = model(X_train_batch).squeeze()
        train_loss = criterion(y_train_pred, y_train_batch)
        train_acc = binary_acc(y_train_pred, y_train_batch)
```

```
        train_loss.backward()
        optimizer.step()
        train_epoch_loss += train_loss.item()
        train_epoch_acc += train_acc.item()
    # VALIDATION
    with torch.no_grad():
        model.eval()
        val_epoch_loss = 0
        val_epoch_acc = 0
        for X_val_batch, y_val_batch in val_loader:
            X_val_batch, y_val_batch = X_val_batch.to(device),
    y_val_batch.to(device)
            y_val_pred = model(X_val_batch).squeeze()
            y_val_pred = torch.unsqueeze(y_val_pred, 0)
            val_loss = criterion(y_val_pred, y_val_batch)
            val_acc = binary_acc(y_val_pred, y_val_batch)
            val_epoch_loss += val_loss.item()
            val_epoch_acc += val_acc.item()
    loss_stats['train'].append(train_epoch_loss/len(train_loader))
    loss_stats['val'].append(val_epoch_loss/len(val_loader))
    accuracy_stats['train'].append(train_epoch_acc/len(train_loader
    ))
    accuracy_stats['val'].append(val_epoch_acc/len(val_loader))
    print(f'Epoch {e+0:02}: | Train Loss: {train_epoch_loss/len(
    train_loader):.5f} | Val Loss: {val_epoch_loss/len(val_loader)
    :.5f} | Train Acc: {train_epoch_acc/len(train_loader):.3f}| Val
     Acc: {val_epoch_acc/len(val_loader):.3f}')
```

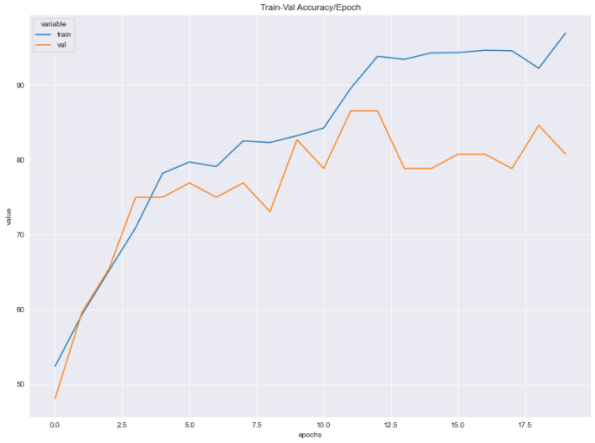
```
Begin training.
Error displaying widget: model not found
Epoch 01: | Train Loss: 32.82149 | Val Loss: 12.73058 | Train Acc: 52.404| Val Acc: 48.077
Epoch 02: | Train Loss: 3.67646 | Val Loss: 2.51428 | Train Acc: 59.308| Val Acc: 59.615
Epoch 03: | Train Loss: 1.31585 | Val Loss: 0.75446 | Train Acc: 65.154| Val Acc: 65.385
Epoch 04: | Train Loss: 0.78206 | Val Loss: 0.52377 | Train Acc: 70.942| Val Acc: 75.000
Epoch 05: | Train Loss: 0.46409 | Val Loss: 0.52571 | Train Acc: 78.212| Val Acc: 75.000
Epoch 06: | Train Loss: 0.46852 | Val Loss: 0.56520 | Train Acc: 79.712| Val Acc: 76.923
Epoch 07: | Train Loss: 0.54159 | Val Loss: 0.52865 | Train Acc: 79.115| Val Acc: 75.000
Epoch 08: | Train Loss: 0.42695 | Val Loss: 0.47568 | Train Acc: 82.538| Val Acc: 76.923
Epoch 09: | Train Loss: 0.38664 | Val Loss: 0.71303 | Train Acc: 82.327| Val Acc: 73.077
Epoch 10: | Train Loss: 0.39174 | Val Loss: 0.44839 | Train Acc: 83.231| Val Acc: 82.692
Epoch 11: | Train Loss: 0.37828 | Val Loss: 0.44125 | Train Acc: 84.269| Val Acc: 78.846
Epoch 12: | Train Loss: 0.27022 | Val Loss: 0.45166 | Train Acc: 89.558| Val Acc: 86.538
Epoch 13: | Train Loss: 0.21976 | Val Loss: 0.43640 | Train Acc: 93.846| Val Acc: 86.538
Epoch 14: | Train Loss: 0.18804 | Val Loss: 0.43476 | Train Acc: 93.442| Val Acc: 78.846
Epoch 15: | Train Loss: 0.19522 | Val Loss: 0.47034 | Train Acc: 94.308| Val Acc: 78.846
Epoch 16: | Train Loss: 0.15929 | Val Loss: 0.46225 | Train Acc: 94.346| Val Acc: 80.769
Epoch 17: | Train Loss: 0.16823 | Val Loss: 0.46349 | Train Acc: 94.654| Val Acc: 80.769
Epoch 18: | Train Loss: 0.15892 | Val Loss: 0.56042 | Train Acc: 94.596| Val Acc: 78.846
Epoch 19: | Train Loss: 0.26647 | Val Loss: 0.46294 | Train Acc: 92.250| Val Acc: 84.615
Epoch 20: | Train Loss: 0.10369 | Val Loss: 0.44206 | Train Acc: 96.962| Val Acc: 80.769
```

Figura 5: Progreso del entrenamiento de la red neuronal

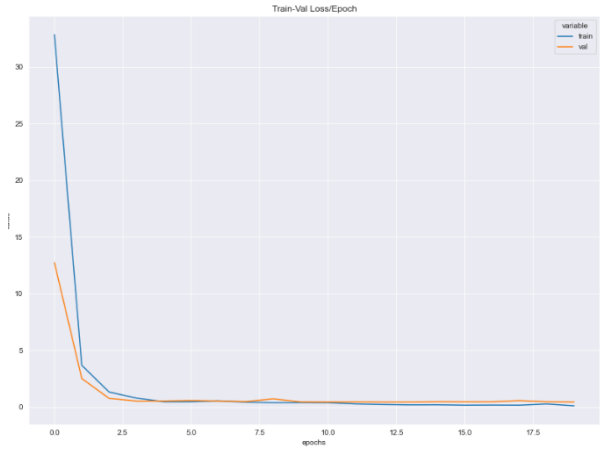
4.8. Análisis de errores.

Para trazar las gráficas de pérdida y precisión, nuevamente creamos un dataframe a partir de los diccionarios accuracy_stats y loss_stats.

```
train_val_acc_df = pd.DataFrame.from_dict(accuracy_stats).
    reset_index().melt(id_vars=['index']).rename(columns={"index":"
    epochs"})
train_val_loss_df = pd.DataFrame.from_dict(loss_stats).reset_index
    ().melt(id_vars=['index']).rename(columns={"index":"epochs"})
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(30,10))
sns.lineplot(data=train_val_acc_df, x = "epochs", y="value", hue="
    variable",  ax=axes[0]).set_title('Train-Val Accuracy/Epoch')
sns.lineplot(data=train_val_loss_df, x = "epochs", y="value", hue="
    variable", ax=axes[1]).set_title('Train-Val Loss/Epoch')
```



(a) Precisión



(b) Pérdida

Figura 6: Gráficos de pérdida y precisión de la red durante le entrenamiento.

4.9. Definir la arquitectura de la CNN.

Después del entrenamiento, debemos comprobar cómo se comporta nuestro modelo. Usaremos model.eval() antes de correr nuetro códido de prueba.

Para decirle a PyTorch que no queremos realizar una retropropagación durante la inferencia, usamos torch.no_grad(), tal como lo hicimos para el ciclo de validación anterior.

- Comenzamos una lista que va a tener nuestras predicciones Entonces iteramos a travéz de nuestros batches usando test_loader. Para cada batch:
  - Movemos nuestros mini bacthes de entrada la la GPU.
  - Hacemos nuestras predicciones en nuestro modelo entrenado.
  - Aplicamos la función de activación log_softmax a las predicciones y legimos el índice del que tenga la mayor probabilidad.
  - Movemos el batch de la GPU a la CPU.
  - Convertimos el tensor a un objeto de nunpy y lo agregamos a nuestra lista.

```
y_pred_list = []
y_true_list = []
with torch.no_grad():
    for x_batch, y_batch in tqdm(test_loader):
        x_batch, y_batch = x_batch.to(device), y_batch.to(device)
        y_test_pred = model(x_batch)
        _, y_pred_tag = torch.max(y_test_pred, dim = 1)
        y_pred_list.append(y_pred_tag.cpu().numpy())
        y_true_list.append(y_batch.cpu().numpy())
```

Aplanaremos la lista para que podamos usarla como entrada para confusion_matrix() y classification_report()

```
y_pred_list = [i[0][0][0] for i in y_pred_list]
y_true_list = [i[0] for i in y_true_list]
```

4.10. Reporte de clasificación.

Finalmente, mostramos nuestro reporte de clasificación que contiene la precisión, el recall y la puntuación F1.

```
print(classification_report(y_true_list, y_pred_list))


              precision    recall  f1-score   support

           0       0.86      0.73      0.79        26
           1       0.77      0.88      0.82        26

    accuracy                           0.81        52
   macro avg       0.82      0.81      0.81        52
weighted avg       0.82      0.81      0.81        52
```

4.11. Matriz de Confusión.

Usaremos la función confusion_matrix() para realizar nuestra matris de confusión.

```
print(confusion_matrix(y_true_list, y_pred_list))
[[19  7]
 [ 3 23]]
```

Creamos un dataframe de la matriz de confusión y lo graficamos en un mapa de calor usando la lirería de Seaborn.

```
confusion_matrix_df = pd.DataFrame(confusion_matrix(y_true_list,
    y_pred_list)).rename(columns=idx2class, index=idx2class)
fig, ax = plt.subplots(figsize=(7,5))
sns.heatmap(confusion_matrix_df, annot=True, ax=ax)
```

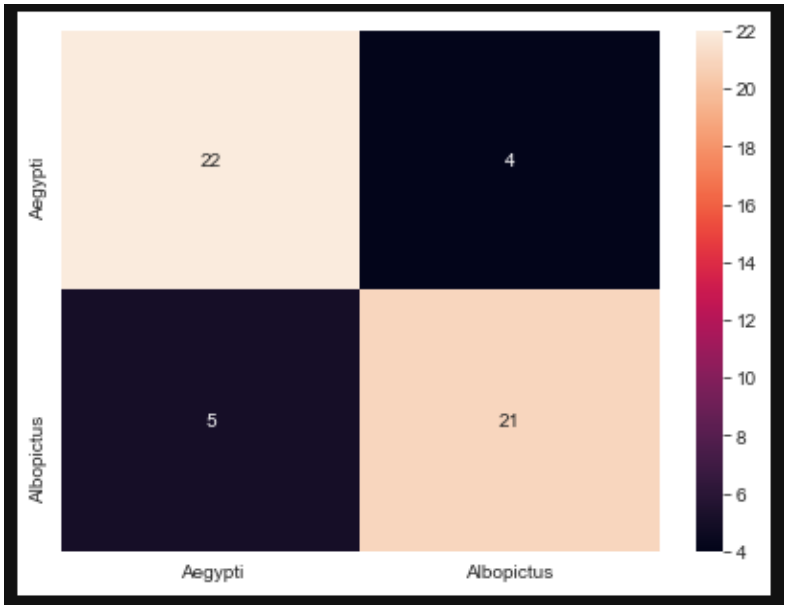


Figura 7: Matriz de confusión que muestra el resultado de la clasificación de las imágenes.

4.12. Implementación.

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