Machine Learning

Final Project - Supervised Learning solution - 1st Delivery

Greening Citrus Leave Identification

Members:

-Erubiel Tun Moo.
-Christian Adriel Rodríguez Narvaez.
-Esau Alberto May Ceh.
-Edwin Antonio Can Pinto.

16/11/2023

Contenido

INTRODUCTION 2
DEVELOPMENT 3
IMPORTING LIBRERIES AND MODULES 3
PATH VARIABLE 3
DEFINE VARIABLES 3
LOAD TRAINING IMAGES 4
LOAD VALIDATION IMAGES 4
TARGET CLASS NAMES 5
FEATURE VECTOR MODEL 5
MODEL TRAINING: 8
MODEL TRAINING: 9
DEVIATIONS BETWEEN PLANNED ACTIONS AND THOSE EXECUTED: 11
CONCLUSION: 11

INTRODUCTION

Citrus greening, also known as Huanglongbing (HLB), is a major hazard to citrus farming around the world. The bacteria Candidatus Liberibacter asiaticus causes this deadly illness, which impairs the health and productivity of citrus trees, resulting in huge economic losses in the agricultural sector. One of the key visible signs of this illness is unusual greening and yellowing patterns in citrus leaves.

Traditional techniques of recognizing and diagnosing citrus greening rely on visual inspection by trained specialists, which is time-consuming, labor-intensive, and prone to human error. Machine learning advances provide a viable path for automating and improving the accuracy of citrus greening detection.

This technical paper delves into the design and implementation of a supervised machine learning model for accurately identifying and classifying greening citrus leaves. Our goal is to train a strong classification model capable of distinguishing between healthy foliage and leaves exhibiting indications of citrus greening using a collection of high-resolution photos of healthy and diseased citrus leaves.

Our goal is to develop a dependable tool that aids in the early diagnosis and mitigation of citrus greening by utilizing cutting-edge machine learning algorithms and image processing techniques. The successful adoption of such a methodology has the potential to change citrus disease management techniques by providing farmers and agricultural specialists with an efficient and proactive approach to monitoring and addressing the development of the disease.

DEVELOPMENT

IMPORTING LIBRERIES AND MODULES

As in any software project, we start the code by importing the libraries and modules that will be used, so in this case we import various Python libraries and TensorFlow-related modules, some of them are NumPy for numerical operations, PLI for image processing, Matplotlib for plotting, TQDM for progress bars during training, Scickit-Image for image transformation, and TensorFlow for machine learning.

In [1]:
```
# standard libraries
import numpy as np
import time
import PIL.Image as Image
import matplotlib.pylab as plt
import matplotlib.image as mpimg
%matplotlib inline
import datetime
from tqdm.keras import TqdmCallback
from skimage import transform
import requests

# tensorflow libraries
import tensorflow as tf
import tensorflow_hub as hub
```

PATH VARIABLE

Here it is defined the path to the directory containing training images. In this case, it points to a directory located in Kaggle.

In [2]:
```
# path variables
train_path = '/kaggle/input/citrus-greening/field'
```

DEFINE VARIABLES

Now very important variables are defined such as batch size, height, with, seed train validation, shuffle, and validation split.

Batch size: determines the number of samples used in each iteration during the model training. A larger batch size can speed up training but requires more memory.

Img_height and img_width: specify the dimensions to which all images will be resized. This ensure uniform dimensions in all images.

Seed_train_validation: sets a seed for random number generation during the data split into training and validation sets.

Shuffle_value: indicates whether to shuffle samples in the dataset. It prevents the model from learning patterns based on the order of sample presentation.

Validation_split: determines the fraction of data to be used as the validation set during training.

```
# define some variables
batch_size = 32
img_height = 300  # reduced from 600 to mitigate the memory issue
img_width = 300   # reduced from 600 to mitigate the memory issue
seed_train_validation = 1
shuffle_value = True
validation_split = 0.4
```

LOAD TRAINING IMAGES

This block uses TensowFlow's utility function to create a labeled dataset from images in the specified directory. The dataset is split into training and validation sets based on the validation_split parameter. Images are resized, organized into batches, and shuffling is applied.

```
# load training images
train_ds = tf.keras.utils.image_dataset_from_directory(
  train_path,
  validation_split=validation_split,
  subset="training",
  image_size=(img_height, img_width),
  batch_size=batch_size,
  seed = seed_train_validation,
  shuffle = shuffle_value )
```

LOAD VALIDATION IMAGES

Similar to the training set, this section loads a validation set using the same utility function, but with subset = "validation".

```
# load validation images
val_ds = tf.keras.utils.image_dataset_from_directory(
    train_path,
    validation_split=validation_split,
    subset="validation",
    image_size=(img_height, img_width),
    batch_size=batch_size,
    seed = seed_train_validation,
    shuffle = shuffle_value )
```

Basically, this whole section prepares datasets from training and validation by loading and organizing images from a specified directory using different libraries and TensorFlow utility function to simplify the process of creating labeled datasets and handling various parameters for training and validation.

TARGET CLASS NAMES

In the following section the code wants to show the classes previous assigned the in dataset. In this case we can notice the dataset are divided into 2 folders labeled. That means in the healthy folder there are only images of healthy and the greening folder just have non-healthy criticus leaves.

So, first we have the ***target class names:***

```
# target class names
class_names = train_ds.class_names
```

The class_names function is to show the class division of the code. And finally, just print. This facilitates the idea of chose the desired classes for utilization.

To the data preparation before the training section, it is necessary to normalize the image layer. The layer is the responsible of pixel values of images. The argument indicates that the layer will scale the pixel values by dividing them by 255.

The next step for the normalization data. This line applies the normalization to every image in the dataset (train_ds and for val_ds)

```
normalization_layer = tf.keras.layers.Rescaling(1./255)
train_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
val_ds = val_ds.map(lambda x, y: (normalization_layer(x), y)) # W
```

What does it consist of caching datasets and prefetching datasets?

- ***Caching datasets:*** Caching datasets storing dataset elements in memory after preprocessing. When you cache a dataset, it saves the processed element to they can be quickly accessed without re-computation.
- ***Prefetching***: Prefetching is a method in which model training and data loading happen at the same time. Prefetching enables the loading of following batches of data while the current batch is being processed by the model, eliminating the need to wait for one batch of data to be processed completely before loading the next batch.

FEATURE VECTOR MODEL

The variable *efficentnet_b7_fv* holds the URL pointing to a pre-trained model available on Kaggle. This refers to EfficientNet B7 Models used for extracting features from images.

```
# feature vector model
efficientnet_b7_fv = 'https://kaggle.com/models/tensorflow/efficientnet/frameworks/
feature_extractor_model = efficientnet_b7_fv

feature_extractor_layer = hub.KerasLayer(
    feature_extractor_model,
    input_shape=(img_width, img_height, 3),
    trainable=False)
```

Hub.KerasLayer, this function creates a Keras layer using TensorFlow. It provides a way to use pre-trained models and modules. So, the KerasLayer allows you to use a model from the URL.

Input_shapes, specifies the input shape expected by the models, define the width and the height of the input images.

Trainable = False, configuring ***trainable = false*** When a layer contains false, its weights become frozen and cannot be modified during training. This is frequently employed in transfer learning to add new layers for a given job while maintaining previously acquired features.

```
# add a classification layer
num_classes = len(class_names)

model = tf.keras.Sequential([
  feature_extractor_layer,
  tf.keras.layers.Dense(num_classes)
])

# model summary
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)    (None, 2560)              64097680

 dense (Dense)               (None, 2)                 5122

=================================================================
Total params: 64,102,802
Trainable params: 5,122
Non-trainable params: 64,097,680
_________________________________________________________________
```

Num_classes: This line calculates the number of classes in the classification problem; That is, it is responsible for showing us a list of the names of the classes (target classes) that were discovered during the loading of the training images.

After this, a sequential model is created using Tensorflow's Keras API. It starts with a pre-trained feature extraction layer obtained from TensorFlow Hub, that is, it takes the input images and extracts those relevant features from them to later add the classification layer on top of the feature extraction and has **num_classes** neurons, one for each class. Once this process is done, the output of the feature extractor is connected to the dense layer and will learn to assign the extracted features to the correct class.

Finally, the **model.summary()** line simply prints us a sample of the model architecture which includes information about the layers, output forms and trainable parameters.

```
[ ]:
# compile the model
model.compile(
  optimizer=tf.keras.optimizers.Adam(),
  loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
  metrics=['acc'])

# early stopping
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)

# Define Epochs
NUM_EPOCHS = 4
```

What happens here is that the model is configured for training, as we can see in the code we use an optimization algorithm called Adam that is responsible for adjusting the weights of the network to minimize the loss function, said loss function measures the difference between the predicted and true labels, finally, metrics are used to monitor during training and at the end print the accuracy of the model.

After this, early stopping is configured as a regularization technique, this means that training will stop if the loss metric stops remembering after a certain number of epochs **patience=3**. In the same way, the number of training epochs is defined, that is, the times that the model will see the entire training data set.

MODEL TRAINING:

In [7]:
```
# train the model
history = model.fit(train_ds,
                    validation_data=val_ds,
                    epochs=NUM_EPOCHS,
                    callbacks=[early_stopping, TqdmCallback(verbose=0)],verbose=0)

# view model accuracy
model_acc = '{:.2%}'.format(history.history['acc'][-1])
print(f"\n Model Accuracy Reached: {model_acc}")
```

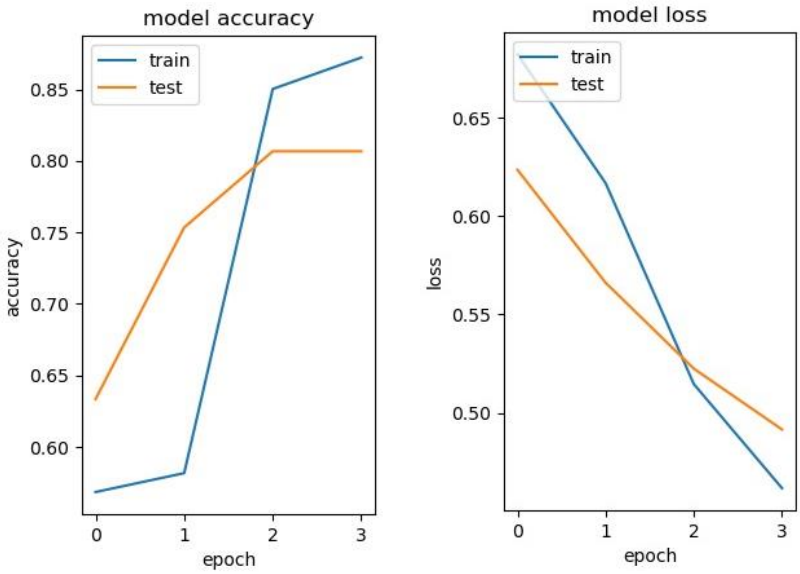
```
100% 4/4 [01:26<00:00, 14.59s/epoch, loss=0.462, acc=0.872, val_loss=0.492, val_acc=0.807]

 Model Accuracy Reached: 87.22%
```

It is during this line of code where the model training is actually done; During this process the **fit** method is used providing the training and validation data sets, the number of epochs, and the stop early callback. Finally, the final precision of the model is calculated after training and is intended to show us said precision in percentage format.

In [8]:
```
# summarize history for accuracy
plt.subplot(1,2,1)
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.subplot(1,2,2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

In these lines of code, matplotlib is used to plot the training history (weight and loss) over the epochs, with the aim of being able to visualize how the model learns the training data set.



MODEL TRAINING:

The final section of the code allows us to use a previously trained model to download an image from the Internet, process it, and determine its class. The image URL that has to be downloaded is entered first. Following that, the binary content of the image is retrieved from the specified URL using the requests library. The picture content is then written to the local file system when a file is opened in binary write mode.

The Pillow (picture) library is used to load the picture using the path of the recently saved file. The picture is then transformed into a NumPy array of type float32, and the pixel values are divided by 255 to normalize them. Furthermore, a new dimension is added to match the predicted input shape, and the image is downsized to the dimensions required by the model.

Next, a prediction is applied to the preprocessed image using the previously trained model. Finding the index of the highest value in the prediction array yields the predicted class. At last, the title indicating the expected class is shown alongside the test image. To generate a figure, read the test image, and display it with extra details like the title and disabled axes, utilize the matplotlib module.

```
img_url = 'https://flower.htgetrid.com/wp-content/uploads/2019/08/kartinka-3.-zelenye-prozhilki.jpg'
img_data = requests.get(img_url).content

with open('/kaggle/working/random_image_from_internet.jpg', 'wb') as handler:
    handler.write(img_data)

test_img_path = '/kaggle/working/random_image_from_internet.jpg'

test_image = Image.open(test_img_path)
test_image = np.array(test_image).astype('float32')/255
test_image = transform.resize(test_image, (img_width, img_height, 3))
test_image = np.expand_dims(test_image, axis=0)

# make predictions
prediction = model.predict(test_image)
pred_class = prediction.argmax()
print(f"The Predicted Class: {class_names[pred_class]}\n")

# view the test-image
plt.figure(figsize=(8,8))
test_img = mpimg.imread(test_img_path)
plt.imshow(test_img)
plt.title("predicted class: "+class_names[pred_class])
plt.axis('off')
plt.show()
```

predicted class: greening

```
1/1 [==============================] - 5s 5s/step
The Predicted Class: healthy
```

predicted class: healthy

DEVIATIONS BETWEEN PLANNED ACTIONS AND THOSE EXECUTED:

There were some deviations from our initial approach. We intended to use a data set covering various species of cocci with diseases detected and labeled. However, we face the difficulty of finding a complete data set that allows us to carry out effective training. This would have facilitated the classification of coconut species according to the presence of diseases and, if so, identifying the nature of these diseases; In the absence of a suitable data set, we adjusted our strategy and chose to work with citrus trees. In this new approach, we achieved a robust data set that allowed us to determine whether trees were healthy or affected by greening based on the evaluation of their leaves.

CONCLUSION:

For the realization of this training, we had to change our dataset from coconut palms to citrus plants, this change was made because the datasets that were found of coconuts images were very unfavorable for training which made an overfitting to our training and at the time of testing it gave us an incorrect classification. When we changed the dataset to citrus plants, it contained well-defined images with noticeable differences, so the training did give a correct classification. In the end the training achieved an accuracy of 87.22% using the citrus plants dataset. The objective of this code was to make a classification of the condition of the plant depending on its visual characteristics, although we could not use the coconut palms, we were able to make a classification with citrus plants, the program can classify an image of a citrus plant if it is healthy or diseased so we will have achieved our goal of classification.