Data ScienceTech Institute

Makerere Fall Armyworm Crop Detection using Transfer Learning Model

Prepared by

Said Hamdi
Guillaume Nony
Deena Zamzam

2024

TABLE OF CONTENTS

1) Introduction

Fall armyworm is a devastating pest in Africa, posing a significant threat to maize crops, which are crucial for the food security and livelihoods of millions of people on the continent.

Machine learning models offer a promising approach to detecting and predicting fall armyworm infestations. It helps farmers to make informed decisions and implement preventive measures to safeguard their harvests.

The present study aims to classify if a plant has been affected by a fall armyworm (label 1) or not (label 0). This binary classification use case comes from the Zindi.africa various challenges providing a dataset of 2699 images equally distributed between classes 0 and 1.

In this notebook, we'll use TensorFlow and Keras libraries to implement the Transfer Learning / Fine-Tuning methodology on the VGG19 pre-trained model (more details to follow).

Also, before stepping into the fine-tuning part, the performance of the model will be studied for 3 class modes: 'binary', 'sparse' or 'categorical'. Finally, the fine-tuning step shall be implemented on the best performing class mode only.

2) Data Loading and Exploration

The project data includes a train.CSV file with 1,619 training images along with their labels, a test.CSV file with 1,080 image names only, and an Images folder containing a total of 2,699 images for both the training and testing datasets. Google Colab will be used as the integrated development environment (IDE) for this project.

The "Train" dataframe (from train CSV) file contains two columns: the Image id and label of values 0 or 1 (the Test dataframe is 1 column only). As an extra step to facilitate dealing with the images in the code, we are going to merge the image path with its id so the final result will look like this:

[7]:

	Image_id	Label
0	id_02amazy34fgh2.jpg	1
1	id_02mh3w48pmyc9.jpg	0
2	id_02rpb463h9d3w.jpg	0
3	id_02wc3jeeao8ol.jpg	1
4	id_03t2hapb8wz8p.jpg	1

[10]:

	Image_id	Label
0	/content/drive/MyDrive/crop-disease/images/id_...	1
1	/content/drive/MyDrive/crop-disease/images/id_...	0
2	/content/drive/MyDrive/crop-disease/images/id_...	0
3	/content/drive/MyDrive/crop-disease/images/id_...	1
4	/content/drive/MyDrive/crop-disease/images/id_...	1
...	...	...
1614	/content/drive/MyDrive/crop-disease/images/id_...	0
1615	/content/drive/MyDrive/crop-disease/images/id_...	1
1616	/content/drive/MyDrive/crop-disease/images/id_...	0
1617	/content/drive/MyDrive/crop-disease/images/id_...	0
1618	/content/drive/MyDrive/crop-disease/images/id_...	0

Note: same processing is done on the 'Test_df' dataframe

3) Image Data Generation

In this line of code, the `ImageDataGenerator` object is being initialized with various parameters for data preprocessing and augmentation before feeding the data into a neural network for training. Combined with the 'flow_from_dataframe()' method, it produces 'dataframe iterators' (datasets of image batches) which allow optimal data access, model's generalization, and robustness.

Hence, the function gen_ds_per_ClassMode is used to create three 'dataframe iterators' based on the specified class mode and whether it is for training or validation data.

```
#apply the function to generate train/validation datasets
train_ds_sparse = gen_ds_per_ClassMode('sparse', 'training')
train_ds_binary = gen_ds_per_ClassMode('binary', 'training')
train_ds_categorical = gen_ds_per_ClassMode('categorical', 'training')
validation_ds_sparse = gen_ds_per_ClassMode('sparse', 'validation')
validation_ds_binary = gen_ds_per_ClassMode('binary', 'validation')
validation_ds_categorical = gen_ds_per_ClassMode('categorical', 'validation')
```

The class mode in the context of image data generators in deep learning frameworks like Keras specifies how the labels or target values are represented: 'sparse' (labels are integers), 'binary' (label is 0 or 1), and 'categorical' (labels are one-hot encoded vectors).

The choice of class mode depends on the nature of the classification problem you are working on and how the labels are represented in your dataset.

4) Build Transfer Learning Model

Transfer learning is a technique in machine learning where a model created for one task is repurposed as the foundation for a model tackling a similar task. Transfer learning is advantageous in terms of computational efficiency and can yield superior outcomes with limited data. In this project, we are employing a pre-trained model to categorize plant images as infected or non-infected. For further information, please visit: https://keras.io/guides/transfer_learning/

The typical procedure for transfer learning in deep learning involves the following steps:

1. Utilize layers from a pretrained model.
2. Freeze these layers to preserve the learned information during subsequent training.
3. Introduce new trainable layers above the frozen ones to translate the existing features into predictions for a new dataset.
4. Train the new layers on your dataset.

Various pre-trained models are available for image classification, such as VGG16, VGG19, InceptionV3, ResNet50, ResNetV2, and others. These models have been trained on extensive image datasets, enhancing their performance on our specific data.
The VGG19 model is being imported from the Keras library's applications module, which provides a collection of pre-trained deep learning models for various tasks.

Then creating an instance of the VGG19 model with specific configurations:

```
model = VGG19(include_top=False, input_shape=(224, 224, 3))
```

Here's an explanation of what this line of code does:

- `include_top=False` : the model will not include the top (fully connected) layers,
- `input_shape=(224, 224, 3)` : the model expects input images to be 224 pixels in height and 224 pixels in width, with 3 channels representing RGB color channels.

By creating the VGG19 model with `include_top=False`, you are essentially using the convolutional base of the VGG19 model without the fully connected layers on top. This allows you to add your own custom layers for tasks like feature extraction, fine-tuning, or transfer learning.

```
for layer in model.layers:
    layer.trainable=False
pretrained_part = Flatten()(model.layers[-1].output)
```

The provided code snippet involves freezing the layers of a pre-trained VGG19 model for feature extraction. Here's an explanation of what each part of the code does:

1. **for layer in model.layers:**
   layer.trainable = False
   ⇨ It freezes the weights of the pre-trained VGG19 model, which means that these layers will not be updated during training.
2. **pretrained_part = Flatten()(model.layers[-1].output)**
   ⇨ It creates a new layer by applying the `Flatten()` operation to the output of the last layer in the pre-trained VGG19 model.

This setup allows you to use the pre-trained convolutional base of VGG19 to extract features from input images while keeping the weights fixed, and then add your own custom layers on top for a specific classification or regression task.

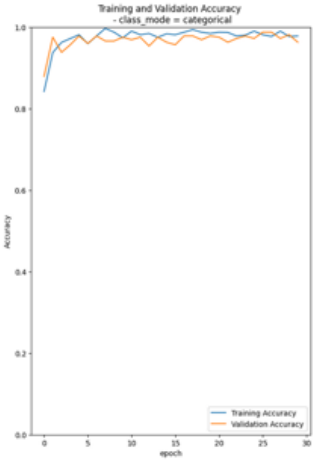
Then we define an output layer **for each of our class_modes: 'sparse', 'binary' and 'categorical'**

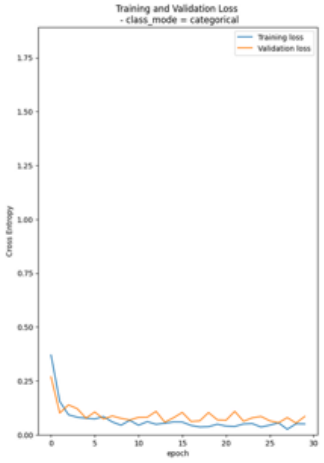
```
class_ = Dense(256, activation='relu')(pretrained_part)
out_ = Dense(1, activation='sigmoid')(class_) # for 'sparse'
model_sparse = Model(inputs = model.inputs, outputs = out_) # for 'sparse' or 'binary' (1D)
#out_binary = Dense(1, activation='sigmoid')(class_) # for 'binary'
model_binary = Model(inputs = model.inputs, outputs = out_) # for 'binary'
out_categ = Dense(2, activation='sigmoid')(class_) # for class_mode='categorical' (2D)
model_categ = Model(inputs = model.inputs, outputs = out_categ) # for class_mode='categorical' (2D)
```

5)Learning curves for each model variant

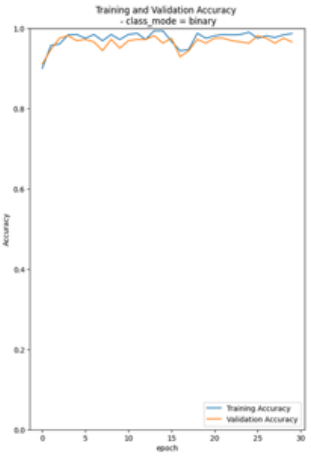
In this part we wrote the code to plot the training and validation accuracy and loss for multiple training histories. The resulting graph will display the training and validation metrics for each classification mode based on the provided training histories.

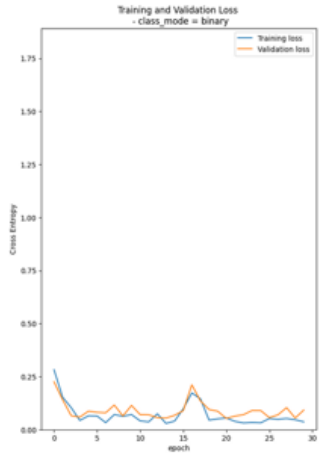
SPARSE MODE



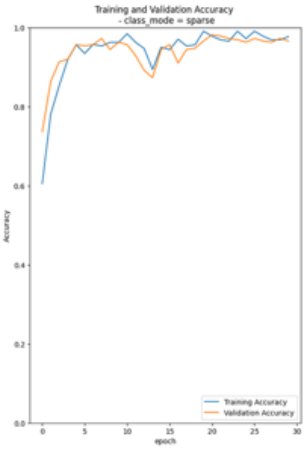


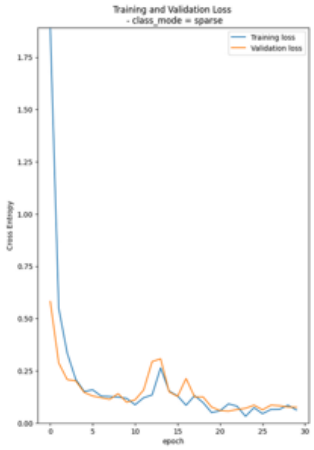
BINARY MODE





CATEGORICAL MODE





Overall, the resulting graph will provide a visual representation of how the model's training and validation accuracy and loss evolve over training epochs for different classification modes. It helps in analyzing the model's performance and understanding how well it generalizes to unseen data based on the validation metrics.

Three versions of the pre-trained VGG19 model for transfer learning were employed, each configured with a distinct class mode: sparse, binary, and categorical. While the 'sparse' and 'categorical' variants exhibit commendable performance in accuracy and cross-entropy metrics, our primary focus lies on the 'Binary class_mode' model due to its relevance to a binary classification task. We will proceed to execute the predict function using all model variations to classify the images in the test dataset.

6) Fine Tune Model

Fine-Tuning, akin to Transfer Learning, is a fundamental technique in deep learning that facilitates the transfer of acquired knowledge from solving one problem to a related one, thereby reducing the data and computational resources needed for training.

While transfer learning involves keeping the pre-trained model's weights fixed and solely training new layers, Fine-Tuning goes a step further by enabling updates to the pre-trained layers.

This step is discretionary and involves unfreezing the entire model (or a portion of it) to retrain on new data using a very low learning rate. This process has the potential to bring significant enhancements by gradually adjusting the pre-existing features to suit the characteristics of the new data.

```
#fine-tuned model fitting
history_binary_fullmod = FT_model_binary.fit(train_ds_binary,
            validation_data=validation_ds_binary,
            epochs=30,
            steps_per_epoch=10,
            verbose=2)
```

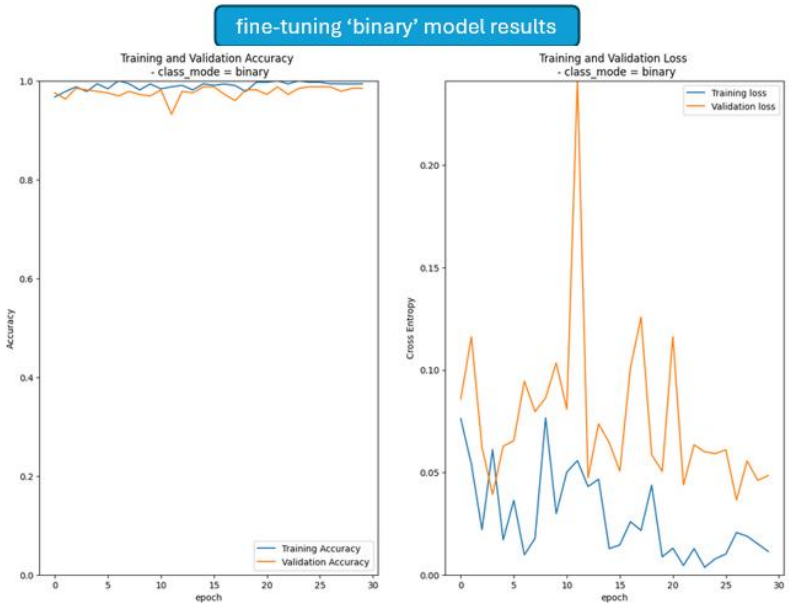
The provided code snippet is using the `fit` method on a model (`FT_model_binary`) to train it on a binary classification task. Here's an explanation in English:

The `history_binary_fullmod` variable stores the training history of the model `FT_model_binary` after fitting it to the binary training dataset (`train_ds_binary`) with validation data provided by `validation_ds_binary`. The training process is configured to run for 30 epochs, with each epoch consisting of 10 steps per epoch. The `verbose=2` parameter indicates that the training progress will be displayed in a detailed manner during the training process.

7) Results Evaluation

After fine-tuning the binary variant model, it exhibited improved performance compared to the results obtained through Transfer Learning:

1. Both training and validation accuracy values are now closer to 1, indicating a significant improvement in the model's predictive accuracy.
2. The loss function values for both training and validation sets are notably low, suggesting that the model is effectively minimizing errors during training. It's important to note that the loss curve may vary if the model is trained again.



CONCLUSION

Overall, the transfer learning / fine-tuning methods were used to classify maize crop images as infected (label 1) or not infected (label 0). The VGG19 was chosen to this end as a pre-trained model.

As a result, after a successful transfer learning, the fine-tuning process showed even better classification results based on accuracy and binary cross-entropy loss function values.

Also, as part of this "challenge", a prediction was made on the test.csv input file which refers to images with unknown labels. The output label predictions (stored in the "my_submission.csv" file) exhibited a very high accuracy score from Zindi's web platform "Get a score" tool (accuracy score of more than 99% on the test dataset).

Finally, such binary classification problem could have been tackled using other approaches.

In the general case (2 classes or more), here are 2 techniques that can be followed:

1. **Global average pooling:** Instead of adding fully connected layers on top of the convolutional base, we add a global average pooling layer and feed its output directly into the SoftMax activated layer. Lin et al. (2013) provides a detailed discussion on the advantages and disadvantages of this approach.
2. **Linear support vector machines:** According to Tang (2013), we can improve classification accuracy by training a linear SVM classifier on the features extracted by the convolutional base. Further details about the advantages and disadvantages of the SVM approach can be found in the paper.