Tuning "Color-aware two-branch DCNN for efficient plant disease classification" to embedded systems

1st Flávio Souza
CIn, UFPE
Recife, Brazil
fjms@cin.ufpe.br

2nd Gustavo Luna Filho
CIn, UFPE
Recife, Brazil
gjlf@cin.ufpe.br

***Abstract*—This document presents the model optimization process, a useful resource in the use of machine learning via devices with reduced computational resources. This project is based on the model obtained through the article *Color-aware two-branch DCNN for efficient plant disease classification*.**

***Index Terms*—DCNN, model optimization, quantization, TensorFlow Lite.**

I. Introduction

Highly accurate deep learning models can be cumbersome, requiring a lot of computational power and thus reducing inference time. Accelerating the inference time of these models by compressing them into smaller models is a widely practiced technique. By making parameters smaller, based on technique, models can be made to use less RAM. This can also simplify the model, reducing latency over the original model, and thus increasing inference speed.

This document presents the methodology and results of the reduction process of the plant disease classification model, obtained from a model derived from the paper *Color-aware two-branch DCNN for efficient plant disease classification* [1], with the aim of applying it to embedded systems, or that is, in devices with reduced computational resources, enabling the farmer or technician to use this diagnostic system in remote locations, without an Internet connection, at low power and low cost.

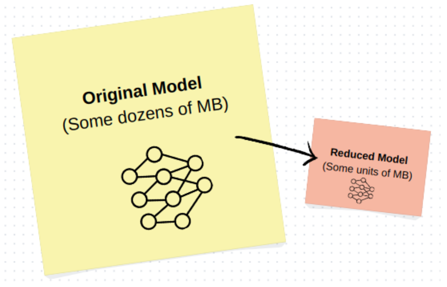


Fig. 1. The target.

II. Motivation

The motivation that led us to change the base article's model used is to reduce its size and computational effort, so that its use in embedded systems may be viable. From there, we looked for how it would be possible to make this change in the article's model without reducing the accuracy using resources from the TensorFlow framework.

III. Methodology

Recent works have shown interest in optimizing the integration of deep learning algorithms in microcontrollers where the approach is focused on the design of deep learning networks.

There are several approaches to design neural networks, but they are mainly applied in mobile devices or microcontrollers with high computational capacity. Even so, some works have proven that it is possible to optimize them even more, considering not only the total number of parameters and network operations but also the low computational capacity and the need for low latency of these devices.

The main technique used in this work was quantization. The fundamental idea behind quantization is that converting weights and inputs to integer types consumes less memory and, on specific hardware, calculations are faster.

However, there is a trade-off: with quantization, we can lose significant precision. But, We'll get into that later looking at how quantization works.

Here, the first aim is to replicate the example code proposed in the code's repository [2], running the best rate of L+AB discovered by the authors (20%L +80%AB). The model also provided in its repository was obtained under high computational resources, with images of the dataset in higher resolution. Despite being available for download, we decided to obtain our own model, using the minimum resources available on the Google Colaboratory platform, using the model obtained by us as the ground truth for the other stages of this study.

In the next step, we aim to reduce our model by converting it into a suitable format via the TensorFlow Lite Converter tool [3]. There are some methods to archive optimized models [4] [5], we plan to pass through some of them and choose the best result: minimum size and maximum accuracy.

Our project does not aim to deploy the reduced model in a real embedded system but to demonstrate a viable means to reach it without compromising the metrics of the original model.

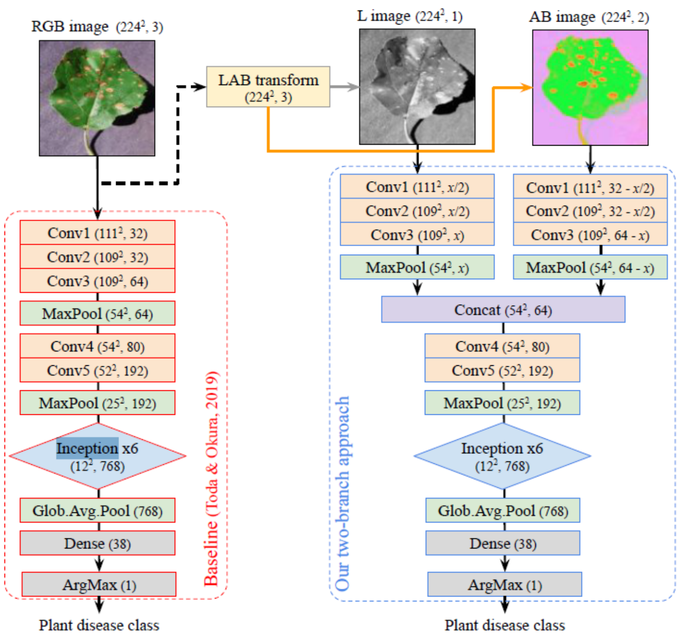


Fig. 2. Graphical representation of worked network architectures in the original paper.

IV. Data Pre-Processing

As the main objective of this work is about reducing the size of the article's model, so we will not delve into the pre-processing of data. For a detailed explanation it's recommended to consult the base paper [2].

V. Experiments

A. Logging a failed path: A subset

In the previous test the code [6] was changed to run in reduced resources of RAM, using appropriately the method provided by the authors *cai.datasets.load_images_from_folders()* we sliced the dataset to around 12% of the original. After the execution in verbose mode, the result is shown:

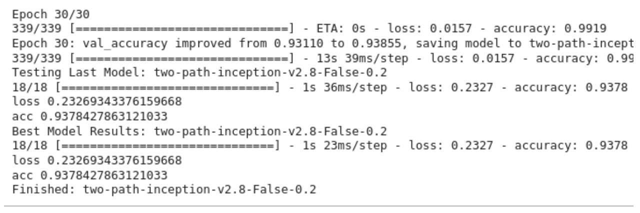
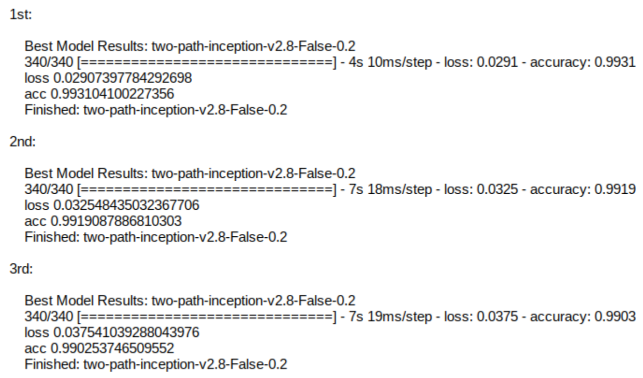

```
Epoch 30/30
339/339 [==============================] - ETA: 0s - loss: 0.0157 - accuracy: 0.9919
Epoch 30: val_accuracy improved from 0.93110 to 0.93855, saving model to two-path-incept
339/339 [==============================] - 13s 39ms/step - loss: 0.0157 - accuracy: 0.99
Testing Last Model: two-path-inception-v2.8-False-0.2
18/18 [==============================] - 1s 36ms/step - loss: 0.2327 - accuracy: 0.9378
loss 0.23269343376159668
acc 0.9378427863121033
Best Model Results: two-path-inception-v2.8-False-0.2
18/18 [==============================] - 1s 23ms/step - loss: 0.2327 - accuracy: 0.9378
loss 0.23269343376159668
acc 0.9378427863121033
Finished: two-path-inception-v2.8-False-0.2
```

Fig. 3. Result after 30 epochs.

In resume, to this primary experiment, loss: 0.2327 - accuracy: 0.9378, wherein first instance the raw dataset was reduced from 54305 to 6516 images, maintaining its 38 classes. But, its accuracy of 93.78%, a little bit different from 99.48% exposed in the original paper, motivating us to change our approach.

B. A promising path: The all dataset

The team migrated to an environment with greater resources (GPU Tesla T4 @ 27.3GB of RAM), which allowed us to reproduce the code available in Colab with the all dataset images but yet reduced(from 224x224x3 to 128x128x3), resulting after some attempts:


```
1st:

    Best Model Results: two-path-inception-v2.8-False-0.2
    340/340 [==============================] - 4s 10ms/step - loss: 0.0291 - accuracy: 0.9931
    loss 0.02907397784292698
    acc 0.993104100227356
    Finished: two-path-inception-v2.8-False-0.2

2nd:

    Best Model Results: two-path-inception-v2.8-False-0.2
    340/340 [==============================] - 7s 18ms/step - loss: 0.0325 - accuracy: 0.9919
    loss 0.032548435032367706
    acc 0.9919087886810303
    Finished: two-path-inception-v2.8-False-0.2

3rd:

    Best Model Results: two-path-inception-v2.8-False-0.2
    340/340 [==============================] - 7s 19ms/step - loss: 0.0375 - accuracy: 0.9903
    loss 0.037541039288043976
    acc 0.990253746509552
    Finished: two-path-inception-v2.8-False-0.2
```

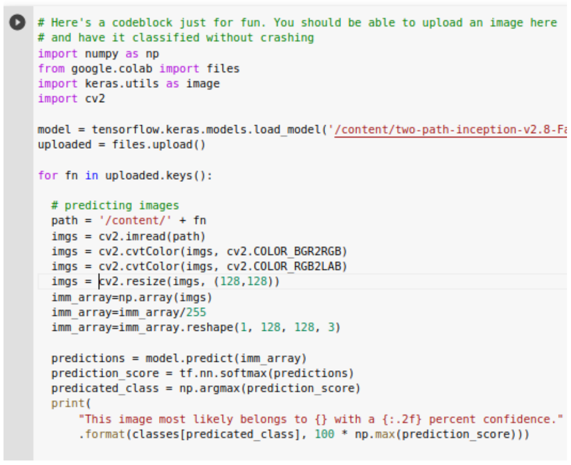
Fig. 4. Three attempts, one choice.

As we can see, despite being better than the previous model, these three attempts reached three models with an accuracy of 99.31%, 99.19%, and 99.03%, even smallest than 99.48% exposed in the original paper. It is important to enhance that every obtained model varies because the training data is chosen randomly in every run of the split section. Then in contact with the authors, we were recommended to resize the images to 224x224x3 to obtain closest results. However, our computational resources now available were still insufficient for such reproduction. Yes, for this task would be necessary a RAM greater than 28GB, which is impossible to guarantee. In possession of our best-saved model(two-path-inception-v6-False-0.2-best_result.hdf5), with 99.31% accuracy, we decided to go on the studies based on it.

C. Some predictions with obtained original model

Here, it was important to debug, searching for some abnormality.

Implemented a little code to suit the image and predict the diagnosis showing to the model a chosen picture randomly selected from the dataset. It is possible to see some tests run under images with a kind of disease.



```
# Here's a codeblock just for fun. You should be able to upload an image here
# and have it classified without crashing
import numpy as np
from google.colab import files
import keras.utils as image
import cv2

model = tensorflow.keras.models.load_model('/content/two-path-inception-v2.8-Fa
uploaded = files.upload()

for fn in uploaded.keys():

  # predicting images
  path = '/content/' + fn
  imgs = cv2.imread(path)
  imgs = cv2.cvtColor(imgs, cv2.COLOR_BGR2RGB)
  imgs = cv2.cvtColor(imgs, cv2.COLOR_RGB2LAB)
  imgs = cv2.resize(imgs, (128,128))
  imm_array=np.array(imgs)
  imm_array=imm_array/255
  imm_array=imm_array.reshape(1, 128, 128, 3)

  predictions = model.predict(imm_array)
  prediction_score = tf.nn.softmax(predictions)
  predicated_class = np.argmax(prediction_score)
  print(
      "This image most likely belongs to {} with a {:.2f} percent confidence."
      .format(classes[predicated_class], 100 * np.max(prediction_score)))
```

Fig. 5. A sample of code.

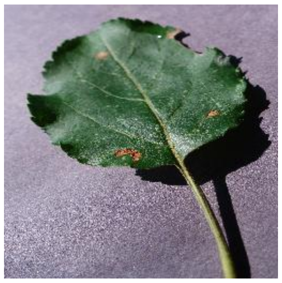
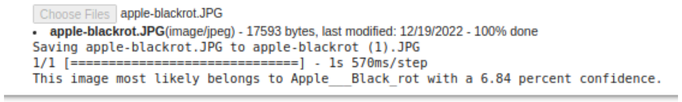


Fig. 6. Input: Apple(blackrot).



```
apple-blackrot.JPG
• apple-blackrot.JPG(image/jpeg) - 17593 bytes, last modified: 12/19/2022 - 100% done
Saving apple-blackrot.JPG to apple-blackrot (1).JPG
1/1 [==============================] - 1s 570ms/step
This image most likely belongs to Apple___Black_rot with a 6.84 percent confidence.
```

Fig. 7. Output: matches!

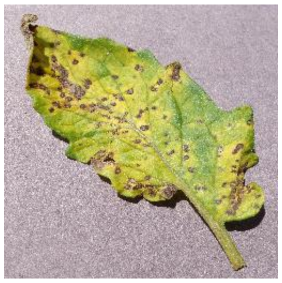
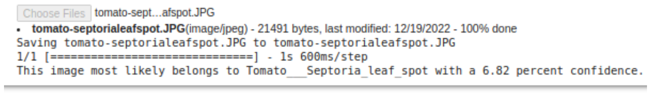


Fig. 8. Input: Tomato(septorial spot).



```
tomato-sept...afspot.JPG
• tomato-septorialeafspot.JPG(image/jpeg) - 21491 bytes, last modified: 12/19/2022 - 100% done
Saving tomato-septorialeafspot.JPG to tomato-septorialeafspot.JPG
1/1 [==============================] - 1s 600ms/step
This image most likely belongs to Tomato___Septoria_leaf_spot with a 6.82 percent confidence.
```

Fig. 9. Output: matches, again!

Notice that we choose the image without matter if belongs to the training set. All seems normal, and, the results show less confidence than 100%, then, confirmed, not from the training split.

With a total of 5,158,932 parameters, our "original" model is in a *.hdf5* file with 41.99MB of size, and ready to be crushed.

D. A Little Theory about Quantization

This technique reduces the number of bits used to represent each parameter, allowing the integration of a complete deep learning network even in a microcontroller with an architecture of only 8 bits. This technique changes the way deep learning network weights and activation are represented. In the beginning, each weight and activation is represented by a 32-bit real number. When the network is quantized, the 32-bit representation is reduced to a smaller number, such as 16 or 8 bits. In general, quantization is performed with eight bits, as some embedded systems are only capable of representing numbers in this format. However, this reduction can lead to a loss of precision in the performance of the deep learning network, although there are several techniques that try to reduce this loss of precision.

In quantization, there are two fundamental representations: integers and floating point numbers.

Integers are represented by their form in the base 2 numeral system. Depending on the number of digits used, an integer can take on several different sizes.

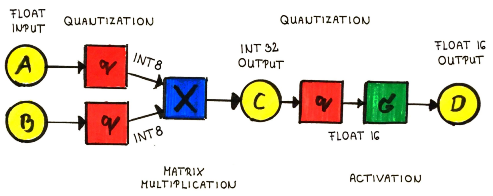


Fig. 10. Quantization scheme.

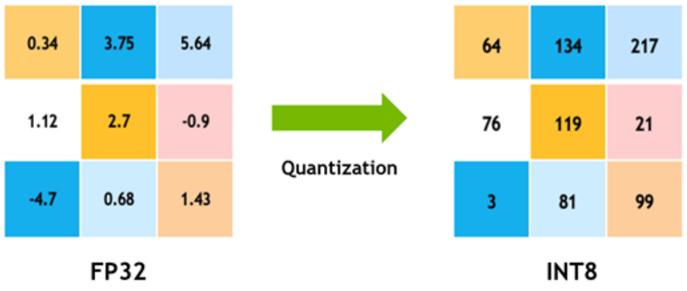
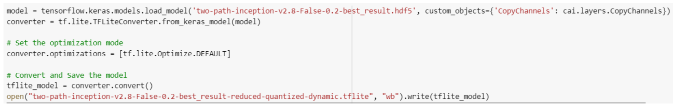


Fig. 11. In the end looks like this.

The TensorFlow framework stands us with a simple way to reduce our original model, with a distinct class called "lite", with or without quantization methods [4] [5].

Between the three approached reduction methods [6], No Quantization, FP16 Quantization, and Dynamic Range Quantization, is expected that this last is the most promising because it provides smaller memory usage and faster computation. In the figure below you can see a snippet of the code used in the conversion to TensorFlow Lite in the quantization technique [6].



```
model = tensorflow.keras.models.load_model('two-path-inception-v2.8-False-0.2-best_result.hdf5', custom_objects={'CopyChannels': cai.layers.CopyChannels})
converter = tf.lite.TFLiteConverter.from_keras_model(model)

# Set the optimization mode
converter.optimizations = [tf.lite.Optimize.DEFAULT]

# Convert and Save the model
tflite_model = converter.convert()
open("two-path-inception-v2.8-False-0.2-best_result-reduced-quantized-dynamic.tflite", "wb").write(tflite_model)
```

Fig. 12. Code snippet in the quantization process in colab.

VI. Results

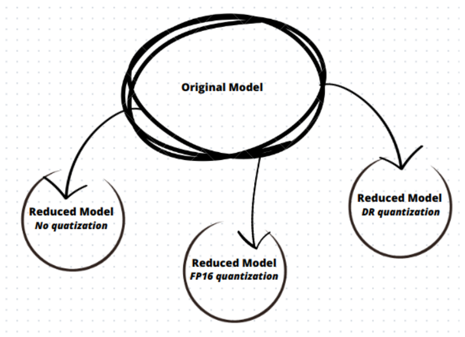


Fig. 13. Concluded task.

Finished the conversion of the original model to the 3 lite models, was implemented a routine to evaluate the respective models even without embedded systems to make some qualitative measurements. Remembering that the *evaluate* method doesn't exist yet in *lite* class. To do this *evaluate* process was previously created a folder to perform the predictions, a sample of 1071 images from the Plant Village dataset, again through author's method *cai.datasets.load_images_from_folders()*, enabling only to build a test folder.

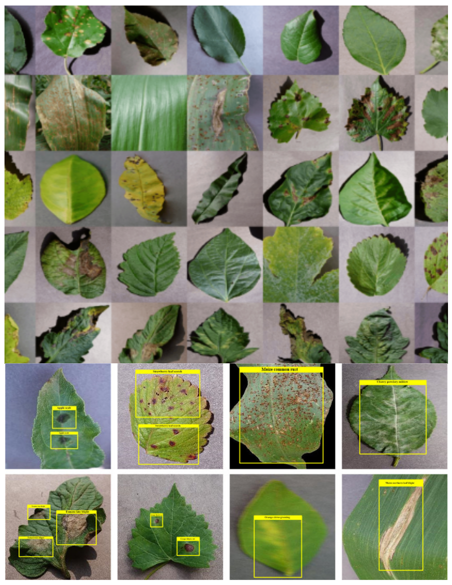


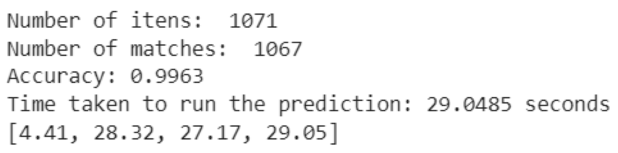
Fig. 14. A sample of Plant Village Dataset with healthy and diseased leaves highlighted.

After this step we could see the following results:

Technique	Size relation	Accuracy
No one	2x	99.63%
float16	4x	99.63%
Dynamic range	8x	99.63%

TABLE I
Resulting metrics

In all stages, the accuracy remained at 99.63%



```
Number of itens:  1071
Number of matches:  1067
Accuracy: 0.9963
Time taken to run the prediction: 29.0485 seconds
[4.41, 28.32, 27.17, 29.05]
```

Fig. 15. Accuracy and latency from the model reached by Dynamic Range Quantization.

Some others tests were performed to measure the feature *latency* of every reduced model, all three remained near 28s, the time needed to predict all 1071 images. Incomparable to the native method *evaluate*, with 4s to the same task through the original model.

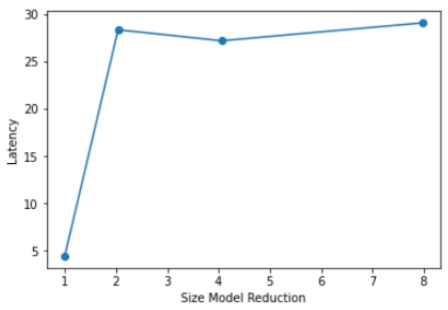


Fig. 16. Latencies.

Because of this disparity with the preview theory we are convinced that for this kind of test, it is necessary to use appropriate hardware 8-bit architecture to acquire suitable results or be, Colab platform couldn't be an appropriate environment to perform latency calculus.

VII. Conclusion

As seen in the last table, the smallest reduction was 8 times the original size, reaching some units of MB(falling from 40MB to almost 5MB), and the accuracy remains in the same value of 99,63%(only 3 wrong predictions between 1071), bigger than base paper.

VIII. Future Works

For future experiments, the prediction method can be applied to all dataset to achieve accurate results. Augmentation can be considered too.

As a next step, the *Full Integer Quantization* technique can be used to ensure the reduction of the original model, which will enable the integration into embedded systems, however, this method requires a sample of a dataset to calibrate the parameters.

As exposed before, optimizing deep learning algorithms is necessary to integrate them into embedded systems, as they often have limited resources. Then, to explore more this ecosystem deeply, another technique can be applied, focusing on the reduction of the size of deep learning networks, decreasing the total number of parameters, or be *Pruning* technique. The goal of this technique is to optimize the model by eliminating the values of the weight tensors. The aim is to get a computationally cost-efficient model that takes less amount of time in training. The necessity of pruning on one hand is that it saves time and resources while on the other hand is essential for the execution of the model in low-end devices such as mobile and other edge devices. Lastly, deploy the optimized model in a real embedded system to calculate the real latency.

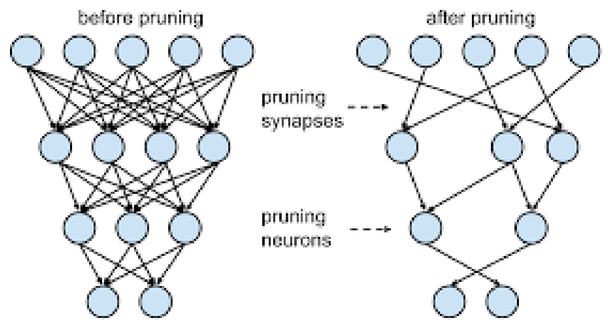


Fig. 17. Pruning.

Acknowledgment

We'd like to thank the paper's authors, mainly Joao Schuler, who supported us in this development, and to our professors, Adriano Oliveira and Cleber Zanchettin, and to our advisor, Stefan Blawid.

References

[1] J. P. Schwarz Schuler, S. Romani, M. Abdel-Nasser, H. Rashwan, and D. Puig, "Color-aware two-branch dcnn for efficient plant disease classification," *MENDEL*, vol. 28, pp. 55–62, Jun. 2022.

[2] "Color-aware two-branch dcnn for efficient plant disease classification." https://github.com/joaopauloschuler/two-branch-plant-disease. Accessed in: 2022-11-23.

[3] P. Warden and D. Situnayake, *TinyML*. O'Reilly Media, Incorporated, 2019.

[4] "Convert tensorflow models." https://www.tensorflow.org/lite/models/convert/convert_models. Accessed in: 2022-12-23.

[5] "Model optimization." https://www.tensorflow.org/lite/performance/model_optimization. Accessed in: 2022-12-23.

[6] "Project's repository." https://github.com/fjmsouza/DeepLearning. Accessed in: 2023-01-11.