As an initial approach, we started by looking at the problem: an image classification problem, with the class divided into subfolders. Thus, the first thing we did was try to apply the code seen during the exercise sessions. Hence we created a notebook on Colab and we started using the "*ImageDataGenerator*" and "*flow_from_directory*" functions for the image augmentation and we applied transfer learning using the pre-trained **Vgg16** network.
Now the problem was to get the data with "*flow_from_directory*", indeed in the exercise session we used this function with the data already divided in folders for the training, validation and test. So we made a custom function to get the folder containing all the data and sub-classes and create two new folders, one for training and one for validation, splitting the data in them. We decided to use 80% of images for training and 20% for validation (see "*build_train_valid_dirs*" function in "*utils.py*"). We didn't use any data for testing because we already had the test set on CodaLab for evaluation and we wanted to exploit all the available data we had.
Then we started to train the model described before: we applied data augmentation, a normalization of the input images (using "*rescale=1/255*") and finally we built the model using the Vgg16 with its layer not trainable and with a final dense layer with 32 neurons. As said before we used the same model we saw in the exercise session, with intent to have a baseline to start with. With this model we get our first evaluation on test set getting an accuracy of 0.51. (The model notebook is in the folder "*models/unbalanced_model*").
After this we tried experimenting with different models, changing the value of the parameters for the augmentation and adding new one, modifying the number of neurons of the dense layer or the resizing of the image when fed to the model. We tried different things to understand what could improve the performance but all these experiments led to results worse than the first one. So we understood that the problem was not only in the model but there was something more.
Then we restart by analyzing the dataset we had and we find out that it was heavily unbalanced, with the tomato class having over 5600 images and raspberry only 264. So we understood why despite the fact that we had an high accuracy when training the model, it was much lower during the test. To solve this problem we have come up with 3 ideas:

- Downsampling
- Upsampling
- Use some weights to balance the unbalanced classes

The idea behind the **downsampling** was easy: in a way to restore a balance in classes, we took the class with the lower number of images and we created a new dataset, with all the other classes having the same number of images as the class with the lower number. In order to accomplish this we made a function that, given a number, for each class create a vector of random sampled images and divide them in training and validation (with the same division 80/20 as before) and store them in the two different folders. The one (and big) problem of this approach was that we were discarding a lot of images and consequently information. Applying this we went from a dataset of 17728 images to one of just 3696 images. We tried it anyway but it

led to very bad results on the test set. We were expecting it and it was a confirmation, the model didn't learn enough. (The model notebook is in the folder "*models/balanced_model_downsampling*"). As we'll describe later, probably the bad results were caused not only by the reduced dataset, but also by the preprocessing made on images. Indeed as we found out later on, all the models we preprocessed with the "*rescale=1/255*" function, didn't bring any good results. We discovered only in the last days that using the vgg16 preprocessing led to way better results, but we didn't have time to retrain all the previous models.
After that, we had some problems in the model training because we were using Colab and it randomly stopped after some time of training. This brings us to lose 2 days of training, and then another 2 trying to make tensorflow work on the new apple silicon processor we started to use.
Solved these issues, we still had to solve the class imbalance problem. The second option was **upsampling**: we wanted to exploit the "imageDataGenerator" function to generate new images to be stored in the dataset. In such a way we could have new and more images for the classes with less images and rebalance the dataset. Even so we were concerned about the overfitting we could face as a result of using the same image multiple times. Thus we decided to discard this option and try the third one. After some research we found the *sklearn* function "*compute_class_weight*" that we used to compute the **class weight**. These can be used during the model fit and are used for weighting the loss function and it's useful to tell the model to pay more attention to samples from an under-represented class. The model obtained was still not good enough, indeed we got a 0.37 as result on the test set (The notebook is in the folder "*models/balanced_model_class_weights*" with name "*sklearn_weights_cat_crossentropy_loss_model.ipynb*").
We were not satisfied by the pre-created function that computed the class weights, so we decided to create our own function to compute them. The function is made in such a way that, taking the number of images for each class, return a weight for each class (inversely proportional to the number of images of the class) and the sum of all the weights is equal to 1. Despite this we were quite sure that the problem was not only in the weights but there was something more. After some code analysis and research we find out that the pre-trained model we used for the feature extraction (Vgg16) has its own preprocessing. We were preprocessing images in the "*imageDatagenerator*" function with "*rescale=1/255*", and we replace it with the "***preprocess_input***" function of "*tf.keras.applications.vgg16.preprocess_input*". This function takes the images and converts them from RGB to BGR, and zero-center each color channel with respect to the ImageNet dataset (and this was quite important because we use the vgg16 with the initial weights pre-trained on ImageNet). Using this new preprocessing and our custom function for compute the weights we finally had an improvement of performance to 0.67 on the test set.
At this time we were happy about the improvement but still not satisfied. We saw a big difference between the accuracy on our validation set (around 0.8-0.9) and the test set one (0.67). With this in mind we decide to improve the complexity of our model, increasing the number of neurons of the fully connected layer to 128, in order

to make it able to generalize more. Moreover we suspect that the test set images were a bit different from the dataset we got, so we decided to apply the augmentation also to the validation set. As a result of that, we got a final performance of 0.75 on the test set.
Finally, we decided to train five different models, tuning some parameters and changing the metric and the loss:

- We decided to apply a fine tuning technique to the model with 0.75 accuracy, so we loaded again the model and we set to trainable the last 4 layers of the vgg16 network.
- As a second model we take the last model (the one with 0.75 accuracy without fine tuning) and change the metric for the early stopping to the **F1 score**, implementing our own function to compute it (see "*custom_metrics.py*"). This is because after some research we find out that this metric was useful with imbalanced class problems. After that we fine tuned the model.
- In the third model we tried to increase the complexity again, so we decided to use 2 layers in the fully connected one, the first one with 256 neurons and the second with 128, and subsequently fine tuned it.
- In the fourth model we wanted to change the loss function, from the categorical cross-entropy to the ***categorical focal loss***. This loss function is useful to address the issue of the class imbalance problem, so we decided to remove the class weights for this model. This loss function is not defined in core TensorFlow (just in TensorFlow Addons repository) so we had to manually define it (see "*custom_metrics.py*").
- As last model we wanted to change the pre-trained model vgg16 but still apply transfer learning. Thus, after some research we decided to use **ResNet50**, another network trained on the ImageNet dataset. We find out that this neural network is widely used for image feature extraction and we saw that it is also faster than Vgg16 to train.

Unfortunately we were unable to test all these models on the first test set and on the final one. If we rely solely on the best score obtained in the final test set, the best model is the one trained with vgg16 and two final layers (The notebook is in the folder "*models/balanced_model_class_weights*" with name "*custom_weights_cat_crossentropy_loss_model.ipynb*") . Despite that, the model with 1 final layer with 128 neurons, and the one trained with the focal loss performed very well (The notebook for the latter model is in the folder "*models/balanced_model_class_weights*" with name "*custom_weights_cat_focal_loss_model.ipynb*").

Ps: In the folder we sent, there are multiple files: "custom_metrics.py" is the file where are defined all the metric we used during the training, in "utils.py" are defined the functions needed to run the notebook (the one to split dataset into training and validation, one for the callbacks, one that create the vgg *network with a dynamic number of final layer, and one to plot the data). Finally in the "models" folder there are the notebooks of the most important models we developed, that are described in this report.*