ANNDL – Challenge One

team_durian

Stefano Maxenti – 10526141 – 970133 Riccardo Mencucci – 10534455 – 970801

Introduction

The goal of our project is to design a neural network capable of classifying 14 types of leaves.
We are provided a dataset containing examples of each class. We use different computing systems: Google Colab, Kaggle and Gradient Paperspace. All relevant extra files not in the attached ZIP can be found here.

Data pre-processing

A very important part in machine learning algorithms concerns the pre-processing of data. This is fundamental in classifying images, because small variations of pixel are very common in image recognition task. In addition to that, the size of the dataset is small for the assigned task and its distribution unbalanced: for example, the ratio between the number of elements of the most numerous class and the least is 22 to 1.
We tackle these problems following this procedure:

1. we augment the dataset using the open-source library Augmentor (offline)
2. we use Keras pre-processing during training (online)

For step 1, we implement a custom function used to generate a background for the image.
We notice, in fact, that the provided dataset contains only segmented images with a solid background. Since we don't know how the test set is composed of, we decide to create a variable number of images with a different background. Our script finds the mask of black pixels using PIL functions and then modify all pixels with random values taken from a hand-crafted uniform distribution. In the final dataset, we generate 500 extra samples of each class.
We use the new dataset after the first submissions and notice a huge improvement. We also perform basic augmentation with flips, zoom, etc.
For step 2, we use the integrate Keras pre-processing functions to perform further augmentation at runtime on all images. However, as described later, in the final VGG19 model we do not perform the second augmentation.
Here is the snippet of code used to create the casual background: it is based to the fact that the original background is pure black with lighter values next to the border of the image.
We end up with a 75% training and 25% validation set.
Our final models are not retrained on the entire dataset with the found hyperparameters before being submitted.

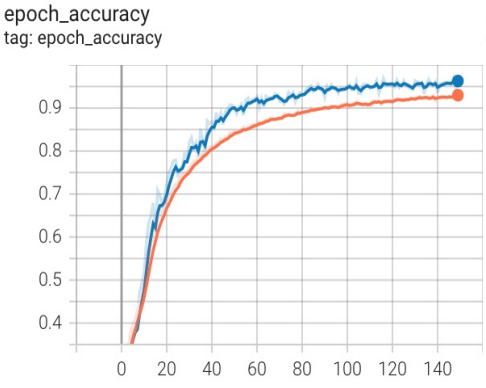
```
# Your class must implement the perform_operation method:
def perform_operation(self, images):
    augmented_images = []
    for image in images:
        im = image.convert('RGBA')
        data = np.array(im)
        # just use the rgb values for comparison
        rgb = data[:,:,:3]
        color = [0,0,0]   # Pure black
        color2 = [40,40,40] # Darker values next to edges
        mask = np.all((rgb >= color) & (rgb <= color2), axis = -1)
        mean = randint(0,255)
        data[mask] = np.random.uniform(mean-30,mean+30, data[mask].shape)
        new_im = Image.fromarray(data)
        new_im = new_im.convert('RGB')
        augmented_images.append(new_im)
    return augmented_images
    # Return the image so that it can further processed in the pipeline:
```

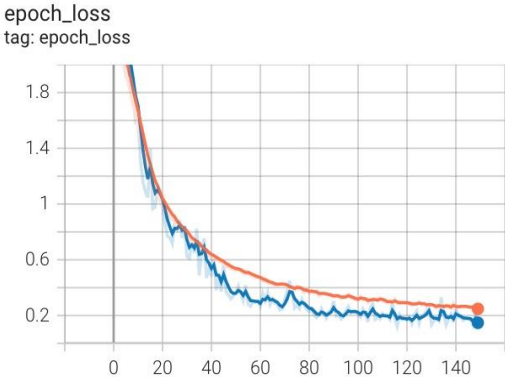
Custom model

Before testing a transfer learning model, we try our best to implement a custom model to train from scratch.
We investigate the possibility of using grayscale images instead of RGB. A quick data exploration allows to notice that leaves are all green and, whereas pigmentation appears, it is localized and could be identified also by edge detection. Grayscale convolutions are also faster since we need to perform one third of the operations (only one channel). After some trials (the same submission with the only difference of the colour levels leads to 6% advantage to the RGB one), nevertheless, we decide to keep going with RGB images, as we don't know whether the test set includes more vivid colours that could be key elements in classification.
Number of parameters varies a lot between submits, ranging from hundreds of thousands to tens of millions. The same structure (Convolutional layer(s), ReLu, MaxPooling) is kept and other hyperparameters (depth, number of filters, strides, etc) are changed between one trial and the other.
We also try to find difference between the *Flatten* layer and the *Global Average Pooling* Layer: eventually, we decide to use the latter because it reduces the number of parameters thus keeping the network simpler and lighter, without any apparent loss of generality.
The best (and smallest) custom model we submit in Phase1 scores 78% with only half a million of parameters after 79 epochs. We retrain it, with the same parameters but for a longer time (150 epochs) for Phase2 and score 82%. Before that, we submit the same model without the online augmentation but, differently for the VGG scenario described later, we obtain a much lower accuracy value.
Images are resized to 128x128 pixels: this dramatically decreases training time and allows to run for more epochs. We also try locally with 64x64 resizing, but results don't seem promising, so we believe that 128x128 might be an adequate compromise between speed and precision.
This model is similar, as architecture, to VGG: we use 3 macro layers, each one made up of two convolutional layers (*ReLu* activated) and a *MaxPooling* (2x2). After those, we have 5 *Dense* layers, with *Dropout* as a regularization technique (values ranging between 0.3 and 0.8).
Overall, there are **585,982** trainable parameters.
Fortunately the last epoch is the best one, both for accuracy (**97.16%**) and loss (0.1010). We include the Tensorboard graphs (training: orange, validation: blue).
We use the "DATASET_AUG" folder as training and validation set. In the Jupyter notebook we also display the activations of the first layer using a random image from the validation set and we show the confusion matrix on the validation set, with very good F1 scores (**0.97** overall).

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 128, 128, 16)	448
conv2d_1 (Conv2D)	(None, 128, 128, 16)	2320
max_pooling2d (MaxPooling2D)	(None, 64, 64, 16)	0
conv2d_2 (Conv2D)	(None, 64, 64, 32)	4640
conv2d_3 (Conv2D)	(None, 64, 64, 32)	9248
max_pooling2d_1 (MaxPooling2	(None, 32, 32, 32)	0
conv2d_4 (Conv2D)	(None, 32, 32, 64)	18496
conv2d_5 (Conv2D)	(None, 32, 32, 64)	36928
max_pooling2d_2 (MaxPooling2	(None, 16, 16, 64)	0
global_average_pooling2d (Gl	(None, 64)	0
dense (Dense)	(None, 768)	49920
dropout (Dropout)	(None, 768)	0
dense_1 (Dense)	(None, 512)	393728
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 128)	65664
dropout_2 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 32)	4128
dense_4 (Dense)	(None, 14)	462

Total params: 585,982
Trainable params: 585,982
Non-trainable params: 0



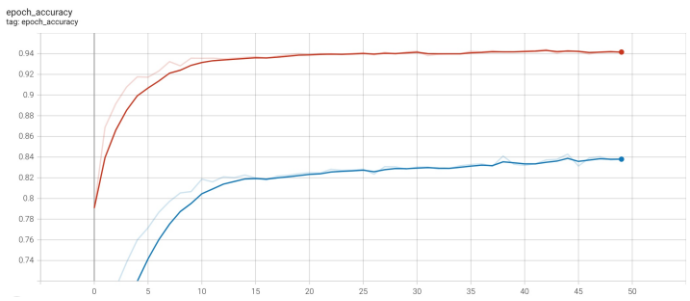


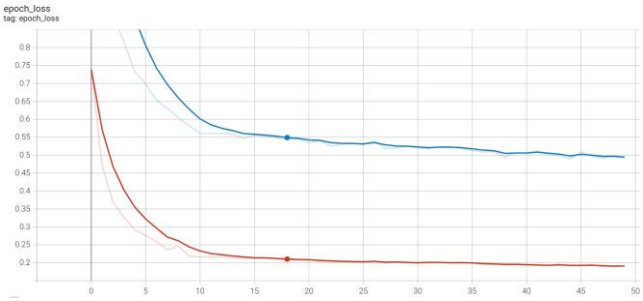
Transfer learning model

VGG19

We decide to use VGG19 loaded with the *Imagenet* weights. After 125 epochs we are able to provide 81% accuracy.
Our VGG model is trained in two steps: for 10 epochs we train just the classification dense layers, freezing the convolutional layers; for 115 epochs we finetune the last convolutional layers as well.
An interesting point is that using double data augmentation as in the custom model, we converge quickly on the validation set but not on the train set (stuck around 84%) and we score 75% on the online test set after about 50 epochs. Our hypothesis is that the network is struggling to learn because of the too many augmentations but performs well on the validation set because it is "easier" than the training one. Probably we would need much more time to converge, as we manage to do in the custom model.

An excessive amount of augmentation seems to be detrimental: the network loses the focus on common patterns because of the differences between the same image.
Here we show the accuracy score of this VGG model.

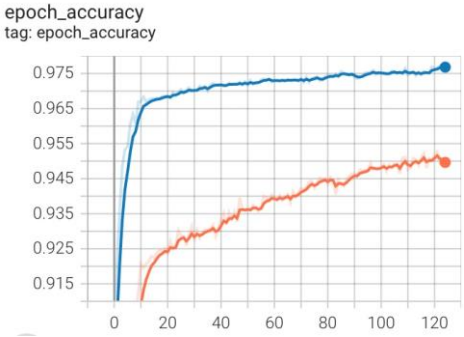


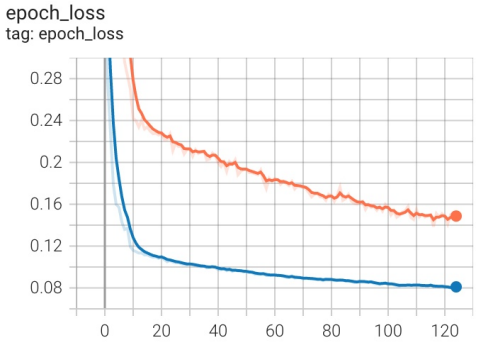


We also try VGG without the second step of data augmentation and we score 81% after 125 epochs. The same model score 84% on the Phase1 hidden test set.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 256, 256, 3)]	0
vgg19 (Functional)	(None, 8, 8, 512)	20024384
gpooling (GlobalAveragePooli	(None, 512)	0
dropout (Dropout)	(None, 512)	0
dense (Dense)	(None, 128)	65664
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 14)	1806

Total params: 20,091,854
Trainable params: 17,766,286
Non-trainable params: 2,325,568



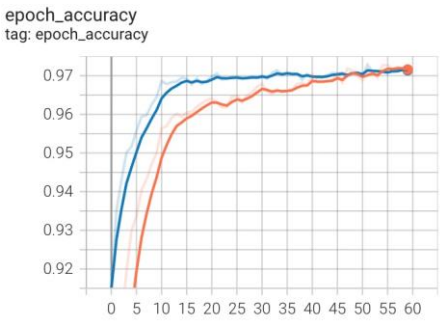


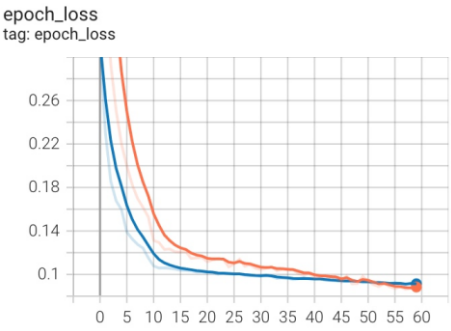
Xception

Transfer learning allows us to "plug in" different networks with minimal effort, not ever requiring a deep understanding of their models. We test Xception, which surprisingly scores 84% on the test set, even higher than VGG19. Probably we could reach an higher accuracy if we train for more time and we apply moderate data augmentation as described before.

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 256, 256, 3)]	0
xception (Functional)	(None, 8, 8, 2048)	20861480
gpooling (GlobalAveragePooli	(None, 2048)	0
dropout (Dropout)	(None, 2048)	0
dense (Dense)	(None, 128)	262272
dropout_1 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 14)	1806

Total params: 21,125,558
Trainable params: 8,666,438
Non-trainable params: 12,459,120





Conclusion and what we learn

We start from big models with millions of parameters and various convolutional layers. Against our first expectations, as in other things of life, bigger and more complex are not always synonymous of better: with our best custom model we are able to provide 82% accuracy with just as few as about 500.000 parameters.
This teaches us that first trials should always start with small models that may be able to provide adequate results in relatively small time, also on CPUs.
Of course, production models need to perform way better and pre-trained transfer learning models with fine tuning come in hand as a fast way to increase accuracy, but the excessive number of layers make the use of a GPU mandatory.
Another lesson we learn is the importance of data exploration, data pre-processing and data augmentation. Without those components, the resulting model is weak to overfitting and not able to generalize well enough.
Last thing but not least, we understand how it is difficult to handle all the hyperparameters and the effects of them changing. We learn that it is best to modify one at a time to see what happens instead of trying to modify too many at the same time.