eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Bachelor Degree in Computer Engineering

Computer Science

Thesis

Approaching deep learning based object detection in microscopy images to non-expert users

Author

Erlantz Calvo Carrillo

Advisors

Ignacio Arganda-Carreras (UPV/EHU)

Guillaume Jacquemet (Åbo Akademi University)

informatika fakultatea

facultad de informática

2021

Abstract

In this project, we have first carried out a study of the state of the art in object detection with Deep Learning, and then we have designed and implemented an approach that is oriented to be run in a cloud service by non-expert users. More specifically, due to its possible applications in microscopy image analysis, a web-based solution that uses the state-of-the-art RetinaNet model has been developed in the open-source ZeroCostDL4Mic environment [1]. Moreover, our implementation uses the TensorFlow 2 object detection API, that allows different backbone networks, and it has been accepted as part of the official ZeroCostDL4Mic platform. Finally, the evaluation of the proposed solution has been performed in a public dataset and compares positively with alternative state-of-the-art approaches.

All the code associated with this project is available and can be run in the following link: https://github.com/ErlantzCalvo/RetinaNet_ZeroCostDL4Mic

[1] https://github.com/HenriquesLab/ZeroCostDL4Mic/wiki

Contents

Abstract **i**

Contents **iii**

List of Figures **vii**

Table index **xiii**

List of algorithms **xv**

1 Introduction **1**

1.1 Object detection in microscopy images 1

1.2 Objectives 2

2 State of the art **5**

2.1 Introduction to object detection 5

2.2 Object detection progress in the past decades 7

2.3 Deep Learning for object detection 8

2.4 Single-shot detector and two-shot detectors 8

2.4.1 Two-shot detectors 9

2.4.2 Single-shot detector 10

2.5 Object detection metrics 13

3 Methodology **17**

3.1 RetinaNet 17

3.1.1 Focal Loss 18

3.1.2 Feature Pyramid Networks (FPN) 23

3.1.3 Bottom-Up Pathway 23

3.1.4 Top-Down Pathway and Lateral Connections 24

3.2 ZeroCostDL4Mic project 26

3.2.1 ZeroCostDL4Mic in Google Colab 30

3.3 Our implementation of RetinaNet in the ZeroCostDL4Mic project 31

3.3.1 Implementation library 31

3.3.2 Data pre-processing 32

3.3.3 Data augmentation 33

3.3.4 Creating a model from pre-trained model's weights 34

3.3.5 Training the model 37

3.3.6 Quality Control 41

3.4 User's interaction in the project 42

3.4.1 Initialising Colab session 42

3.4.2 Installing dependencies 43

3.4.3 Setting the training parameters 43

3.4.4 Data Augmentation 48

3.4.5 Loading weights from a pre-trained network 49

3.4.6 Train the network 49

3.4.7 Choosing model for evaluation 51

3.4.8 Inspection of the loss function 51

3.4.9 Quality metrics estimation 52

3.4.10 Use the model for actual predicting 55

4 Experimentation **57**

4.1 Familiarisation experiments 57

4.1.1 Pre-trained RetinaNet on the COCO dataset 57

4.1.2 Fine tuning for rubber ducky prediction 58

4.1.3 Extension to multi-class prediction 60

4.2 Dataset used 60

4.3 Experimental results of the RetinaNet model in the MDA-MB-231 cell dataset 61

5 Conclusions **63**

Bibliography **65**

List of Figures

1.1 Example of object detection from the PASCAL VOC dataset. From left to right: two pairs of input images and their corresponding detected objects (bounding boxes and class names), Source: https://www.jeremyjordan.me/object-detection-one-stage/ 2

2.1 Types of object detection methods. Examples of detection of specific object (first row) and detection of generic object categories (second row). Source: [1] 6

2.2 Milestones of object detection and recognition, including feature representations. The time period up to 2012 is dominated by handcrafted features, a transition took place in 2012 with the development of Deep Convolutional Neural Networks (DCNNs) [2] for image classification by Krizhevsky et al., with methods after 2012 dominated by related deep networks. Most of the listed methods are highly cited and won a major ICCV or CVPR prize. Source: [1] 7

2.3 Structure of a two-shot or two-stage detection model. A traditional CNN extract feature maps form the input image. Those feature maps are passed to the Region Proposal Network which detects the object instances in the feature maps and then the detected objects coordinates and the feature maps are passed to the classifier where the objects' classes are predicted. Source: https://medium.com/egen/region-proposal-network-rpn-backbone-of-faster-r-cnn-4a744a38d7f9 9

2.4 Basic operation of a RPN. From left to right : (a) Example of the two-shot detector's RPN processing. A sliding window of size 3×3 produces $k = 9$ number of proposals. The sliding window is composed by two sub-networks: a classification network (*cls*) to predict the objectness score, and a regression network (*reg*) to estimate the coordinates of the proposals; (b) example of the four values output by the regression sub-network. Source: https://medium.com/egen/region-proposal-network-rpn-backbone-of-faster-r-cnn-4a744a38d7f9 10

2.5 SSD traditional model architecture. SSD has a base VGG-16 network followed by multi-box convolutional layers. The VGG-16 base network for SSD is a standard CNN architecture for high quality image classification but without the final classification layers, so it is used for feature extraction. Additional convolutional layers are added next for detection and they decrease in size progressively. Source: https://towardsdatascience.com/review-ssd-single-shot-detector-object-detection-851a94607d11 11

2.6 Non-maximum suppression example removing the surplus predictions (left) and leaving the best one for the given object (right). Source: https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c 12

2.7 Example of how precision works. Here it can be seen that the model has predicted 1 True Positive and 0 False Positives, so applying the previous formula we get Precision = 1. Source: https://towardsdatascience.com/map-mean-average-precision-might-confuse-you-5956f1bfa9e2#:~:text=The%20mean%20Average%20Precision%20or,an%20IoU%20threshold%20of%200.5 14

2.8 Precision and Recall representation. Source: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173 15

3.1 RetinaNet detector's architecture. From left to right: (a) first part of the backbone network (a ResNet in the figure as well as in the original paper), (b) FPN where objects are detected and localised in each FPN's level by the class (c) and box (d) sub-networks. Source: https://developers.arcgis.com/python/guide/how-retinanet-works/ 18

3.2 Foreground-background class imbalance. Source: https://towardsdatascience.com/neural-networks-intuitions-3-focal-loss-for-dense-object-detection-paper-explanation-61bc0205114e 19

3.3 Example of Cross Entropy limitation. The ranges of loss values for both, the well classified examples $[0.6, 1.0]$ and the wrong classified examples $[0, 0.6)$ are indicated with arrows. The 100,000 easy examples receive a loss value 40 times lower than the 100 hard examples (approximately 0.1 vs 2.3 each). Source: [3] 21

3.4 The sigmoid function converts the value given as input into a real value contained in the range $(0, 1)$. 22

3.5 Feature Pyramid Network (FPN). FPN composes of a bottom-up and a top-down pathway. The bottom-up pathway is the usual convolutional network for feature extraction. As we go up, the spatial resolution decreases. With more high-level structures detected, the semantic value for each layer increases. FPN provides a top-down pathway to construct higher resolution layers from a semantic rich layer. Lateral connections are added in order to improve the upscaling process. Source: https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c 23

3.6 The *top-Down Pathway* process starts from the top of the feature pyramid and descends every level. The first layer is obtained from the last output of the Bottom-Up pathway. The 1×1 filter is applied to the given output, obtaining the *M5* layer and making a prediction on that layer. Then, the layer is upsampled, outputting the same feature map with twice the resolution. After the upsampling is done, an element-wise operation is done between the upsampled feature map and *C4* with the 1×1 convolution applied, obtaining the *M4* layer. After the element-wise merge operation, a convolutional filter of size 3×3 is applied to the *M4* layer. This process is repeated for the remaining levels of the feature pyramid. 24

3.7 Visual summary of how ZeroCostDL4Mic works. This picture is taken from the ZeroCostDL4Mic original paper [4]. (a) Paths to exploiting DL. Training on local servers and inference on local machines (or servers) (first row), cloud-based training and local inference (second row), cloud-based training and inference (third row) and pre-trained networks on standard machines (fourth row). (b) Overview of ZeroCostDL4Mic. The workflow of ZeroCostDL4Mic, featuring data transfer through Google Drive, training, quality control and prediction via Google Colab. (c) Overview of the bioimage analysis tasks currently implemented within the ZeroCostDL4Mic platform. 26

3.8 ZeroCostDL4Mic notebook's example. A first view of the ZeroCostDL4Mic Jupyter notebook structure in the Google Colaboratory virtual environment. This picture is taken from the ZeroCostDL4Mic original paper [4]. 28

3.9 Examples of different methods of data augmentation. Different transformations are applied to the original image in order to artificially create new samples. Source: `https://www.researchgate.net/figure/Data-augmentation-using-semantic-preserving-transformation-for-SBIR_fig2_319413978` 29

3.10 Representation of the variables used in the normalisation process. 33

3.11 Examples of data augmentation transformations permitted in our implementation. 34

3.12 Fine-tuning a pre-trained model. Using an already trained model (bottom), transfer learning is used so some of the layers of the model are frozen, i.e., their weights won't change during the training process and the remaining layers are fine tuned with new data (top). Source: [5] 36

3.13 Example of a graph defined in TensorFlow. Source: `https://www.tensorflow.org/guide/intro_to_graphs` 39

3.14 Example of how the GPU access checking is displayed. It can be appreciated that the Figure's GPU model is a Tesla T4, having 16GB of VRAM memory. 42

3.15 Example of what the user sees before and after he/she connects with a Google Drive account. 43

3.16 Fields of the parameters to set for training. 45

3.17 Output example of cell in Section 3.1 of the notebook. 47

3.18 Example of the data augmentation cell and its output. 48

3.19 Loading weights section cell. 49

3.20 Output example of the training process. 50

3.21 Model for evaluation choosing cell. 51

3.22 A display example of the cell from Section 5.1 of the notebook. In this example, in the first two plots, can be noted that both, the training loss (blue line) and the validation loss (orange line) still decrease in the last epoch (epoch number 100), so the model probably can improve if it continues training. It can also bee seen that the validation set's mAP metric continued improving slowly. 52

3.23 Qualitative evaluation: a prediction of an image from the test set established by the user. 53

3.24 Quantitative evaluation: mAP score for the test set and the AP metric of each class. 54

3.25 Display of three different views of a test image: the original image, its prediction and how it should have been predicted. 55

3.26 Cell from Section 6.1 of the notebook where the predictions will be made and saved. 56

3.27 Display example of the last cell in the notebook. 56

4.1 Prediction of two dogs in a picture by the model's prototype. 58

4.2 Manually annotated rubber ducks using the annotation tool from TF2 API. 59

4.3 Inference results using the model fine-tuned with rubber duckies. 59

4.4 Inference results using the model fine-tuned with rubber duckies, minions and platypuses. 60

4.5 Dataset's image example. From left to right: (a) original image, and (b) expert annotations on top of original image. 61

Table index

3.1 Comparison of boxes number between different object detection models. Note: the *k* letter stands for *thousand*. 19

3.2 Example of integer encoding. 37

3.3 One-hot encoding example. The table represents that the *Animal 1* is from class *Dog*, the *Animal 2* is from class *Cat*, the *Animal 3* is from class *Parrot* and *Animal 4* is from class *Cat*. 38

4.1 Comparison between RetinaNet and YOLOv2 performance on MDA-MB-231 cell dataset. mAP results correspond to the test set values. In bold, best time and evaluation metric. 62

List of Algorithms

1 Non-Maximum suppression algorithm 13

CHAPTER 1

Introduction

1.1 Object detection in microscopy images

In the Bioimage analysis[1] field, a great part of the financial and personal resources are destined to the manual annotation of images. For years, computer vision has provided methods to try to automate that process and alleviate that burden. Nowadays, thanks to the use of Deep Learning, great results are being achieved in that direction for many types of microscopy imaging and modalities.

Object detection is a computer technology related to computer vision and image processing that deals with detecting instances of semantic objects of a certain class (such as humans, buildings, or cars) in digital images and videos. Two different examples of object detection are shown in Figure 1.1.

[1]Bioimage analysis focuses on the use of computational techniques to analyse bioimages, especially cellular and molecular images, at large scale and high throughput.

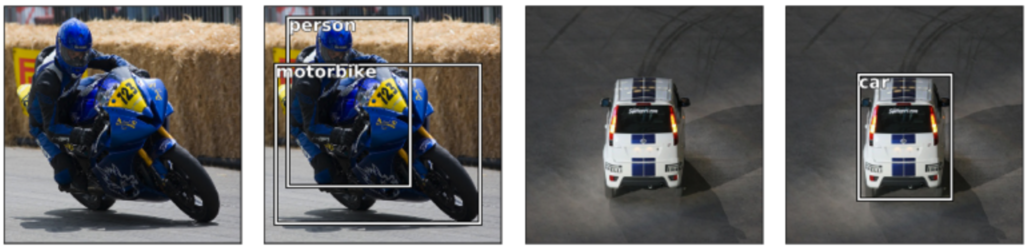


Figure 1.1: Example of object detection from the PASCAL VOC dataset. From left to right: two pairs of input images and their corresponding detected objects (bounding boxes and class names), Source: https://www.jeremyjordan.me/object-detection-one-stage/

Object detection is currently used in many everyday applications like face detection tools or medical applications such as cancer detection, improving our quality of life in recent years.

Our work in this project will be driven by the ZeroCostDL4Mic [4] project. This project aims to automatise the training and implementation of Deep Learning approaches for microscopy. Moreover, the idea is to build a solution where non-expert users are able to create, train and run Deep Learning models despite their possible lack of knowledge on this topic. To carry out this task, a user friendly executable environment which runs in a cloud service has been created. Indeed, the main purpose of this project is to build a state-of-the-art object detection with Deep Learning approach which can easily perform good quality object predictions on images.

1.2 Objectives

The main function of this project is to build a Deep Learning object detection approach that can be easily used by non-expert users on user-provided microscopy images. To achieve this objective, we have the following tasks to develop:

1. Bibliographic study of the state-of-art in object detection with Deep Learning to obtain the best results with the technology we have available to date.

2. Study of the Tensorflow Object Detection API[2] development platform to be able to create and test state-of-the-art object detection models.

[2]https://github.com/tensorflow/models/blob/master/research/object_detection/README.md

3. Study of the ZeroCostDL4Mic platform to implement Deep Learning based object detection models for non-expert users.
4. Selection and development of a solution based on both platforms, so users can achieve results with an acceptable minimum quality in their specific datasets.
5. Evaluation of the proposed solution using public datasets.
6. Comparison between this project's approach and alternative ones.

CHAPTER 2

State of the art

Object detection is a common problem in the domain of computer vision which deals with identifying and locating objects of certain classes in an image. Interpreting the object localisation can be done in various ways, including creating a bounding box around the object (see Figure 1.1) or marking every pixel in the image which contains the object (called segmentation) [1].

Although nowadays the most popular solutions for object detection tasks are based on Deep Learning, object detection has been a problem of interest in computer vision for a long time. Therefore, a brief introduction will be made in the following sections. Also, we will cover some theory behind object detection that will be needed to understand terms along the project.

2.1 Introduction to object detection

As a longstanding, fundamental and challenging problem in computer vision, object detection has been an active area of research for several decades. The goal of object detection is to determine whether there are any instances of objects from given categories (such as humans, cars, bicycles, dogs or cats) in an image and, if present, to return the spatial location and extent of each object instance.

Object detection methods can be grouped into one of two types: detection of specific instances versus detection of broad categories. The first type aims to detect instances of a particular object (such as Donald Trump's face, the Eiffel Tower, or a neighbour's dog), essentially a matching problem. The goal of the second type is to detect (usually previously unseen) instances of some predefined object categories (for example humans, cars, bicycles, and dogs).

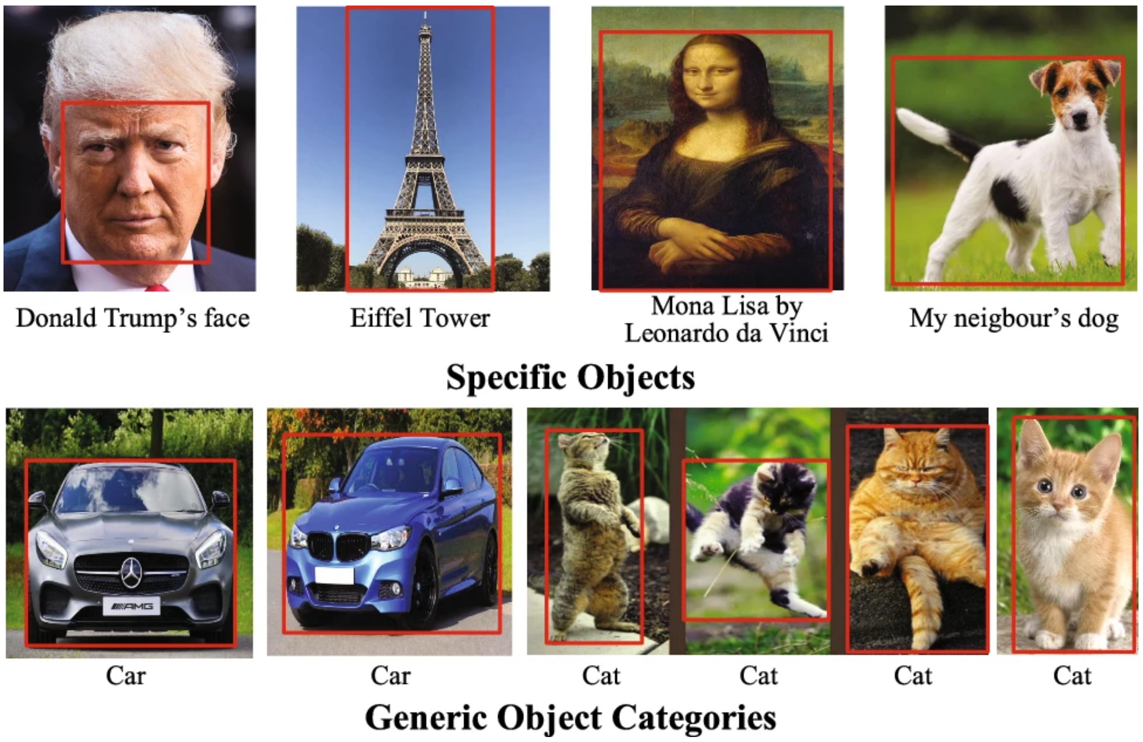


Figure 2.1: Types of object detection methods. Examples of detection of specific object (first row) and detection of generic object categories (second row). Source: [1]

2.2 Object detection progress in the past decades

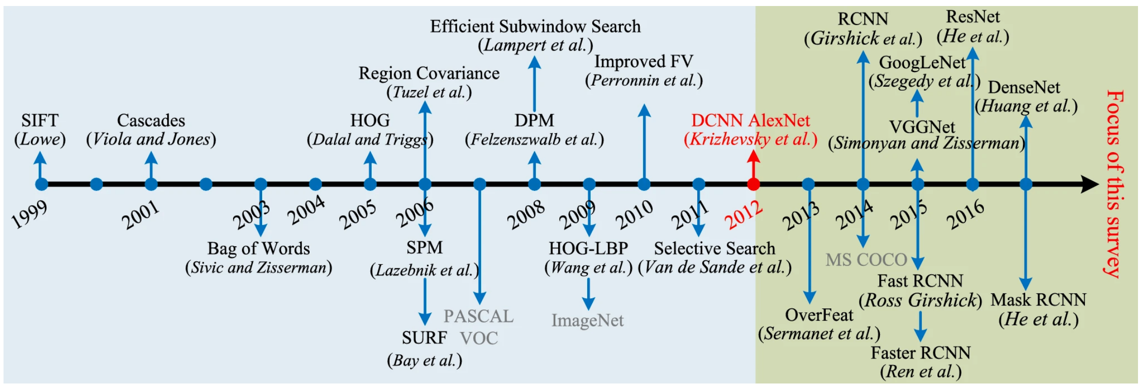

SIFT (Lowe)
Cascades (Viola and Jones)
HOG (Dalal and Triggs)
Region Covariance (Tuzel et al.)
Efficient Subwindow Search (Lampert et al.)
DPM (Felzenszwalb et al.)
Improved FV (Perronnin et al.)
DCNN AlexNet (Krizhevsky et al.)
RCNN (Girshick et al.)
ResNet (He et al.)
GoogLeNet (Szegedy et al.)
DenseNet (Huang et al.)
VGGNet (Simonyan and Zisserman)
Focus of this survey
1999
2001
2003
2004
2005
2006
2008
2009
2010
2011
2012
2013
2014
2015
2016
Bag of Words (Sivic and Zisserman)
SPM (Lazebnik et al.)
PASCAL VOC
SURF (Bay et al.)
HOG-LBP (Wang et al.)
ImageNet
Selective Search (Van de Sande et al.)
OverFeat (Sermanet et al.)
MS COCO
Fast RCNN (Ross Girshick)
Faster RCNN (Ren et al.)
Mask RCNN (He et al.)

Figure 2.2: Milestones of object detection and recognition, including feature representations. The time period up to 2012 is dominated by handcrafted features, a transition took place in 2012 with the development of Deep Convolutional Neural Networks (DCNNs) [2] for image classification by Krizhevsky et al., with methods after 2012 dominated by related deep networks. Most of the listed methods are highly cited and won a major ICCV or CVPR prize. Source: [1]

The first researches in the object recognition field were based on template matching techniques and simple part-based models [6], focusing on hardly variant shapes objects, such as faces. Before 1990, the leading paradigm of object recognition was based on geometric representations, with the focus later moving away from geometry and prior models towards the use of statistical classifiers, such as Neural Networks, based on appearance feature.

For years, the multistage hand tuned pipelines of handcrafted local descriptors and discriminative classifiers dominated a variety of domains in computer vision, including object detection, until the significant turning point in 2012 when Deep Convolutional Neural Networks (DCNNs) [7] achieved their record-breaking results in image classification with the AlexNet model, using GPUs for training a DCNN for first time, which makes feasible the computationally high costs. When AlexNet came out showing the availability of GPUs with very high computational capability, the object detection field as well as the Deep Learning field were revolutionised, giving a boost to the use of DCNN.

2.3 Deep Learning for object detection

As we have mentioned before, since AlexNet showed its potential, the research focus in most aspects of computer vision has been specifically on Deep Learning methods. The main challenge of object detection is to develop a model that can achieve high accuracy in the detection and localisation tasks with high efficiency. The high efficiency requires that the entire detection task runs in real time with a manageable memory and storage consumption, so all kind of devices can run it, such as smartphones or cars.

The milestone approaches appearing since Deep Learning entered the field are organised into two main categories:

- Two-stage (also known as two-shot) detection frameworks, which include a pre-processing step for generating object proposals.
- One-stage (also known as one-shot) detection frameworks, or region proposal free frameworks, which do all the process in the same step.

2.4 Single-shot detector and two-shot detectors

In object detection tasks, the model aims to sketch tight bounding boxes around desired classes in the image, alongside each object labelling.

2.4.1 Two-shot detectors

The two-shot detector approaches are based on two different steps: (1) the region proposals (candidate regions for the object of interest) and (2) the classification of those regions and refinement of the location prediction.

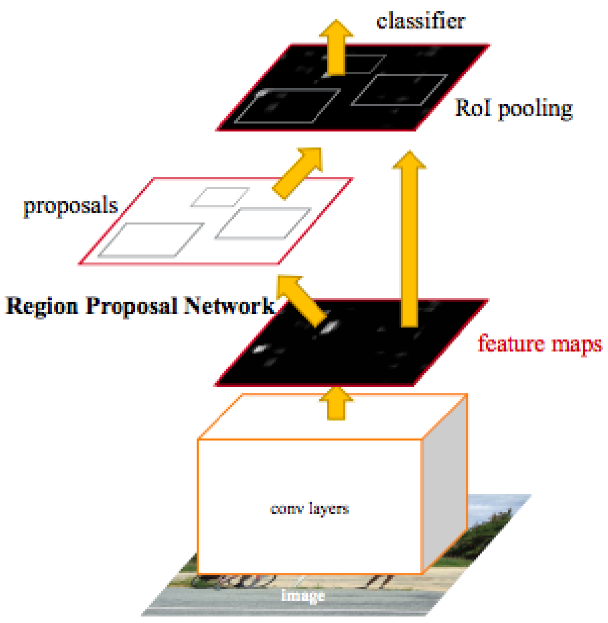
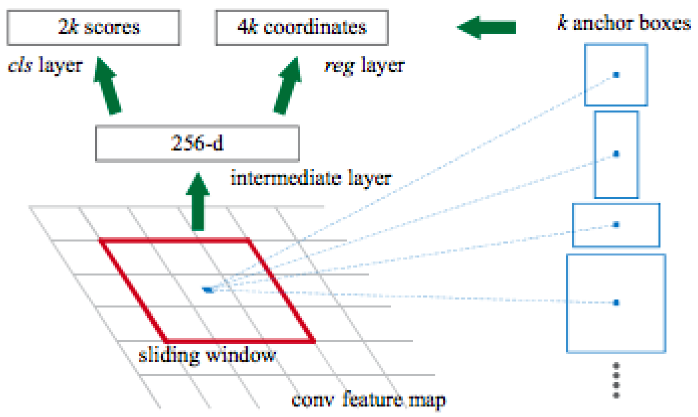


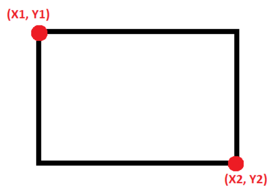
Figure 2.3: Structure of a two-shot or two-stage detection model. A traditional CNN extract feature maps form the input image. Those feature maps are passed to the Region Proposal Network which detects the object instances in the feature maps and then the detected objects coordinates and the feature maps are passed to the classifier where the objects' classes are predicted. Source: https://medium.com/egen/region-proposal-network-rpn-backbone-of-faster-r-cnn-4a744a38d7f9

As we can see in Figure 2.3, the two-shot detector uses a traditional CNN (usually pre-trained with the ImageNet dataset) as backbone, which is used to extract feature maps from the input image and then pass those feature maps to the Region Proposal Network and to the classifier.

When the feature maps are passed to the Region Proposal Network (RPN), this outputs rectangular object proposals. Each proposal has an *objectness* score, an indicator of belonging to an object class or background.



(a) Example of how a Region Proposal Network (RPN) works.



(b) The values that the *reg* network outputs

Figure 2.4: Basic operation of a RPN. From left to right : (a) Example of the two-shot detector's RPN processing. A sliding window of size 3×3 produces $k = 9$ number of proposals. The sliding window is composed by two sub-networks: a classification network (*cls*) to predict the objectness score, and a regression network (*reg*) to estimate the coordinates of the proposals; (b) example of the four values output by the regression sub-network. Source: `https://medium.com/egen/region-proposal-network-rpn-backbone-of-faster-r-cnn-4a744a38d7f9`

In Figure 2.4a, a RPN which uses a sliding window of 3×3 dimensions is represented. Each sliding window is mapped to a lower-dimensional vector which is fed into two sibling fully-connected layers: a box-regression layer (reg) and a box classification layer (cls). At each sliding window location, k region proposals are predicted, so the *reg* layer has $4k$ outputs, being these outputs the bounding boxes coordinates of the upper-left corner and the coordinates of the lower-right corner as is represented in Figure 2.4b, and the *cls* outputs $2k$ scores that estimate probability of object / not object for each proposal. The k proposals are parameterised relative to k reference boxes, called *anchors*. Once all the anchors have been predicted they are passed to the second step, the classification of the predicted regions where the layer in charge is a traditional CNN which work is to predict what class each object belongs to in the resulting region proposals.

2.4.2 Single-shot detector

The two-shot detection approaches have dominated the object detection field since RCNN [8] and Faster RCNN [9] because of their results on popular benchmark datasets. This happens because two-shot detectors achieve (generally) better accuracy than the one-shot detectors. Even though this feature is very important when we are looking for an object detection model, it does not mean that the two-stage models are better than the one-stage ones, because they are also computationally much more expensive and non-suitable for devices such as smartphones or wearables.

Unlike the two-shot detectors, the single-shot detector (SSD) skips the region proposal stage and yields final localisation and content prediction at once. This property makes SSD detectors way faster than the two-shot detectors in both training and inference, which is a very important factor for real-time applications.

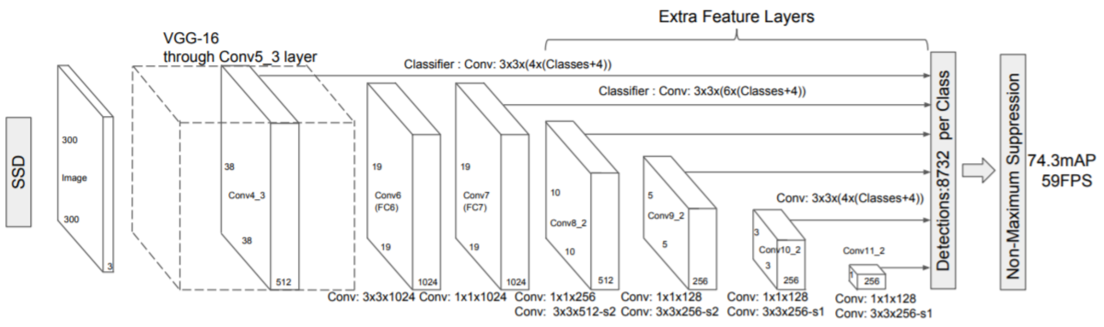


Figure 2.5: SSD traditional model architecture. SSD has a base VGG-16 network followed by multi-box convolutional layers. The VGG-16 base network for SSD is a standard CNN architecture for high quality image classification but without the final classification layers, so it is used for feature extraction. Additional convolutional layers are added next for detection and they decrease in size progressively. Source: https://towardsdatascience.com/review-ssd-single-shot-detector-object-detection-851a94607d11

As we can see in Figure 2.5, the SSD approach is based on a feed forward convolutional network that produces a fixed-size collection of bounding boxes, followed by a non-maximum suppression step to produce the final detections. The non-maximum suppression is the step in charge of cleaning the agglomeration of bounding boxes for the same prediction.

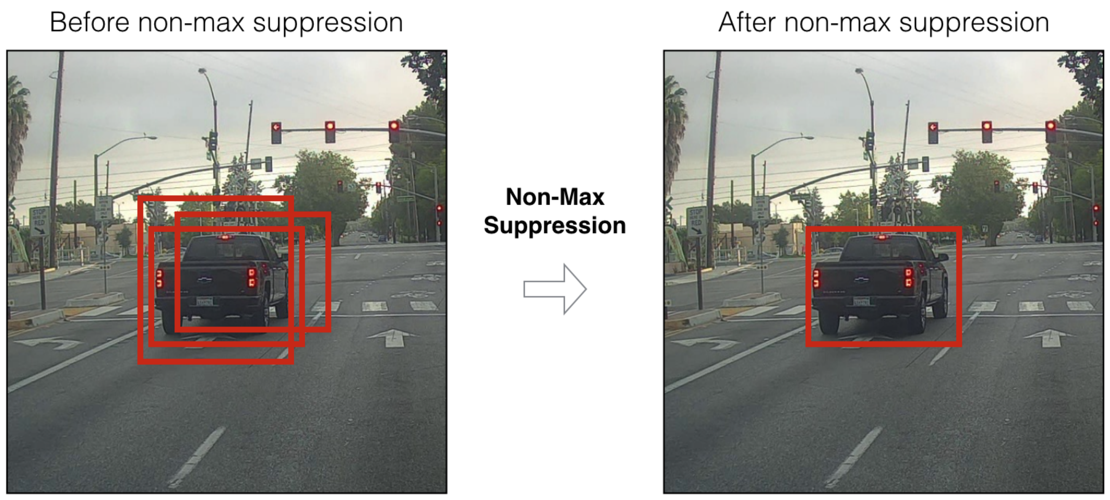


Figure 2.6: Non-maximum suppression example removing the surplus predictions (left) and leaving the best one for the given object (right). Source: https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c

In Figure 2.6, we show how the same objects contained in the image can be predicted with different bounding boxes before carrying out the non-maximum suppression process (left image) and how we get the best unique prediction for each object after the non-max suppression process (right image). Indeed, the algorithm filters the predicted proposals, leaving only the most accurate ones. More specifically, it takes a list of proposal boxes B, corresponding confidence scores S and overlap threshold N as input and works as follows:

Algorithm 1 Non-Maximum suppression algorithm

```
 1: INITIALIZE $Out = \emptyset$
 2: INITIALIZE $Overlap\_threshold = [0, 1]$
 3: while exist boxes in $B$ do
 4:     $Current\_Proposal$ = proposal with highest $S$ from $B$
 5:     $Out = \{Out, Current\_Proposal\}$
 6:     Remove $Current\_Proposal$ from $B$
 7:     for $P$ in $B$ do
 8:         $IOU = Calculate\_IOU(Current\_Proposal, Proposal)$
 9:         if $IOU > Overlap\_threshold$ then
10:             Remove $P$ from $B$
11:         end if
12:     end for
13: end while
14: return $Out$
```

The Intersection over Union (IoU) calculation is used to measure the overlap between two proposals, and it can be defined as follows:

$$IoU(b, b^g) = \frac{area(b \cap b^g)}{area(b \cup b^g)}$$

where b represents the first bounding box and b^g represents the second bounding box. In the numerator, we compute the area of overlap between the two bounding boxes.

The denominator is the area of union, or more simply, the area encompassed by the two boxes.

Finally, the non-maximum suppression algorithm returns Out, the list with the proposals that are classified as valid.

2.5 Object detection metrics

The mAP metric is a very popular evaluation metric used for object detection. Before getting into the explanation of said metric we have to understand two concepts: **Precision** and **Recall**. **Precision** measures how accurate a model's predictions are. It measures how many of the predictions that the model's made were actually correct.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

Where *True Positives* are the samples predicted as positive as was correct and *False Positives* are the samples predicted as positive but was incorrect.

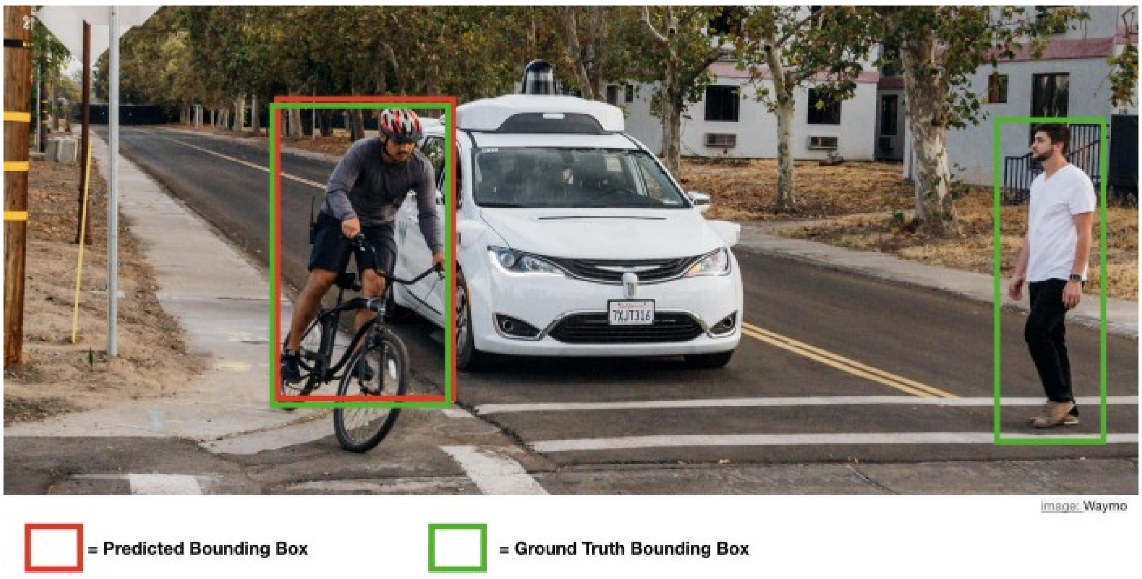


Figure 2.7: Example of how precision works. Here it can be seen that the model has predicted 1 True Positive and 0 False Positives, so applying the previous formula we get Precision = 1. Source: https://towardsdatascience.com/map-mean-average-precision-might-confuse-you-5956f1bfa9e2#:~:text=The%20mean%20Average%20Precision%20or,an%20IoU%20threshold%20of%200.5

Recall measures how well you find all the positives.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

The general definition for the Average Precision (AP) is finding the area under the precision-recall curve above.

$$AP = \int_0^1 Precision(Recall)\ dRecall$$

Precision and recall are always between 0 and 1. Therefore, AP falls within 0 and 1 also. Before calculating AP for the object detection, we often smooth out the zigzag pattern first.

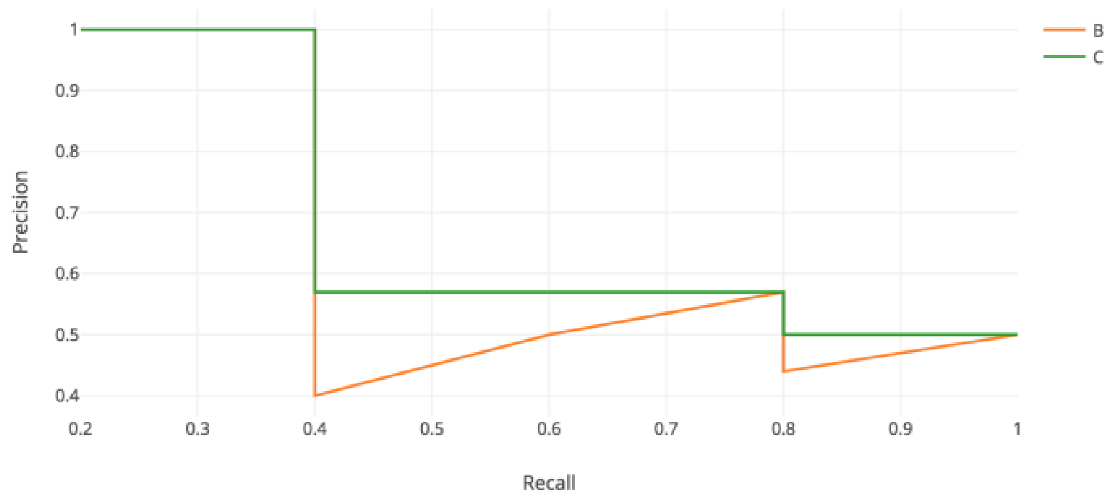


Figure 2.8: Precision and Recall representation. Source: https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173

In order to avoid the integration in the AP formula and the impact of its *wiggles*, a point interpolation is done over the Precision function. This interpolation is usually composed by 11 points, so after dividing the Recall value from 0 to 1.0 11 times we get the following formula:

$$AP = \frac{1}{11} \sum_{r=\{0,0.1,0.2,...,1.0\}} Precision(r)$$

Finally, we can calculate the mAP metric that is the average of the AP calculated for all the classes in the sample, this is,

$$mAP = \frac{1}{N} \sum_{n=1}^{N} AP_i$$

Where N is the number of classes in the sample.

CHAPTER 3

Methodology

In the year 2018, a group of researchers from the Facebook AI Research (FAIR) team published a paper called *Focal Loss for Dense Object Detection* [3] where they showed how the best object detectors to that date, when it comes to accuracy, were based on a two-stage approach. Then, they presented the single-shot detector called RetinaNet, which not only improved the one-stage models but also surpassed the metrics provided by the two-stage ones.

In this project, we have implemented the RetinaNet single-stage object detection model in the ZeroCostDL4Mic project, to facilitate its use to non-expert users. Thus, making available to the community this state-of-the-art object detection approach for microscopical image analysis.

3.1 RetinaNet

The RetinaNet model is a one-stage detector that, for the first time, matched the state-of-the-art results of more complex two-stage detectors in the public benchmark dataset COCO[1], such as the Feature Pyramid Network (FPN) [10], which is an architecture made by the same authors as RetinaNet, or Faster R-CNN variants. In order to achieve this

[1]Source: https://jonathan-hui.medium.com/object-detection-speed-and-accuracy-comparison-faster-r-cnn-r-fcn-ssd-and-yolo-5425656ae359

performance, the RetinaNet model added two improvements over existing single-stage object detection models: the Focal Loss and FPN.

The RetinaNet detector is a single network composed of a backbone network and two-task specific sub-networks. The backbone is an existing convolutional network model that extracts feature maps from the input images. The first sub-network's task is to classify the backbone's output into a class, like a common CNN does. The second sub-network's task is to make a bounding box regression. This architecture is well represented in Figure 3.1.

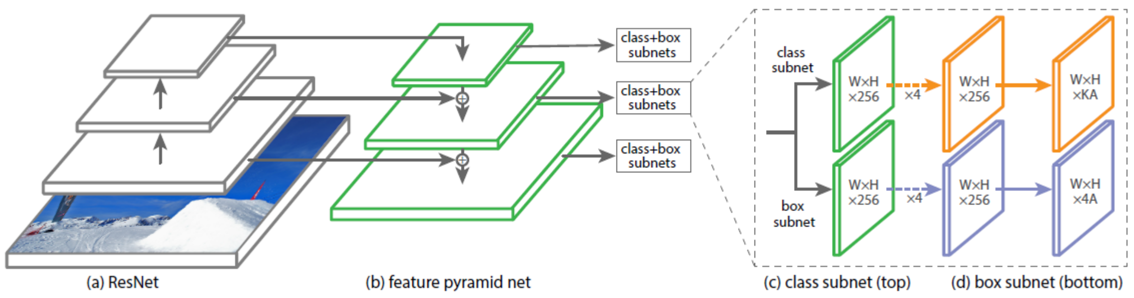


Figure 3.1: RetinaNet detector's architecture. From left to right: (a) first part of the backbone network (a ResNet in the figure as well as in the original paper), (b) FPN where objects are detected and localised in each FPN's level by the class (c) and box (d) sub-networks. Source: https://developers.arcgis.com/python/guide/how-retinanet-works/

In this section, we will explained the reasoning behind both concepts, the *Focal Loss* and the *FPN*, and how they work, so we can delve deeper into the model and understand why it has outperformed the past object detection models.

3.1.1 Focal Loss

In the original paper, they claim that the big difference in accuracy between the two-stage detectors and the one-stage detectors lies in a class imbalance problem during training. This happens because, in the one-stage object detection models, there is a dense sampling of anchor boxes, making the model suffer from a very large foreground-background class imbalance. This issue is natively resolved by the region proposal network of the two-stage detectors, where the number of candidate object locations are narrowed down to a small number, filtering out most background samples. In Table 3.1, we show an example of the number of boxes that the model generates compared to other approaches, so we can notice the class imbalance problem. This problem is caused by the tiny error of each negative example. Normally, the model will not be 100% certain that the box does not contain any objects. This cause that, if the model predicts that a box contains no objects with a security of 0.9, there's an error of 0.1. This is applied to all the boxes that don't contain any objects, adding a major error value.

Model	**Number of boxes**
Faster R-CNN [9]	1-2k
YOLOv2 [11]	1k
OverFeat [12]	1-2k
RetinaNet [3]	100k

Table 3.1: Comparison of boxes number between different object detection models. Note: the *k* letter stands for *thousand*.

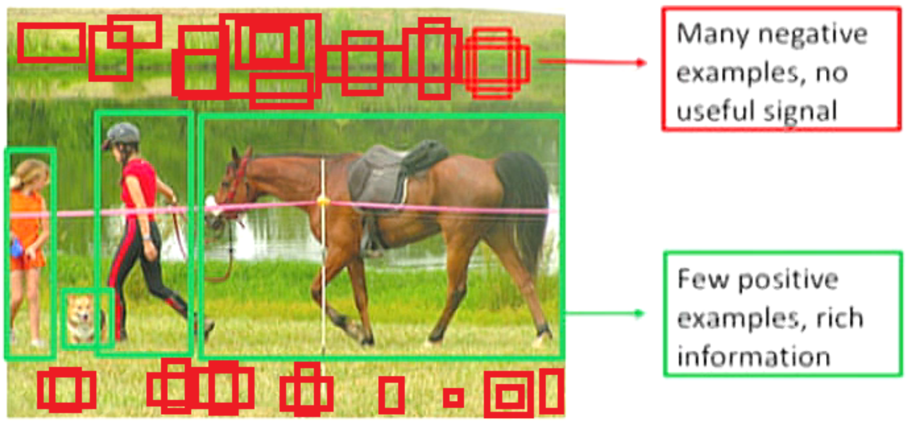

Many negative examples, no useful signal
Few positive examples, rich information

Figure 3.2: Foreground-background class imbalance. Source: `https://towardsdatascience.com/neural-networks-intuitions-3-focal-loss-for-dense-object-detection-paper-explanation-61bc0205114e`

In Figure 3.2, we can see an example of the problem about foreground-background mentioned in the paper. Note that, even though in the image there are some negative examples in the top-down borders (the ones with colour red), in the actual model there would be almost 100k boxes like those which would increase the loss function.

To solve this problem, a new loss function called Focal Loss is presented, eliminating this barrier. Indeed, the Focal Loss is designed to address the one-stage object detection scenario in which there is an extreme imbalance between foreground and background classes during training (i.e. 1:1000). The Focal Loss is based on the cross entropy loss for binary classification, which is defined as follows:

$$CE(p, y) = \begin{cases} -log(p) & if \quad y = 1 \\ -log(1-p) & otherwise \end{cases} \tag{3.1}$$

where $y \in \{\pm 1\}$, which specifies the ground-truth class (i.e., if there is an actual object in the given region or not), and $p \in [0,1]$ which is the model's estimated probability for the class with label $y = 1$. This way, when $y = 1$ and p is high ($p \approx 1$) the loss value for that prediction will be $-log(1) = 0$ and, when p tends to 0, the loss value is increased. Otherwise, when $y = -1$, what means that there is no object that can be detected in the region, the higher is p, the higher is the loss value.

For notational convenience, p_t is re-written as

$$p_t = \begin{cases} \text{p} & \text{if} \quad y = 1 \\ \text{1-p} & otherwise \end{cases} \tag{3.2}$$

and therefore $CE(p,y) = CE(p_t) = -log(p_t)$.

The reason why this loss function does not work properly for the one-stage object detection approaches is because, as it can be seen in Figure 3.3, even examples that are easily classified incur a high loss. When the loss of a large number of easily classified examples is summed, the resulting value can overwhelm the loss value of the rare class, even if each rare class has a significantly higher loss individually.

Let's put ourselves in the case where we have 100,000 easy examples and 100 hard examples. Looking at Figure 3.3, we can think that each easy example will output a 0.1 loss value (having the model a confidence value of 0.9 then $-log(0.9) \approx 0.1$), while the hard ones, assuming the prediction fails with 0.1 confidence, will output 2.3 ($-log(0.1) \approx 2.3$).

$$The\ loss\ from\ easy\ examples = 100,000 \times 0.1 = 10,000$$

$$The\ loss\ from\ hard\ examples = 100 \times 2.3 = 230$$

$$\frac{easy\ examples\ loss}{hard\ examples\ loss} = \frac{10,000}{230} = 43$$

In the previous equation, it can be seen that we obtain a loss value 43 times greater from the easy examples than from the hard ones, and that is the reason why the Cross Entropy loss is not the best loss for one-stage object detection models.

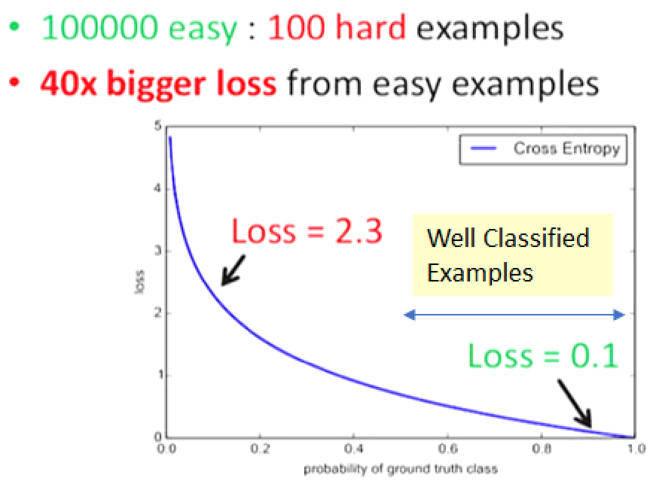


Figure 3.3: Example of Cross Entropy limitation. The ranges of loss values for both, the well classified examples $[0.6, 1.0]$ and the wrong classified examples $[0, 0.6)$ are indicated with arrows. The 100,000 easy examples receive a loss value 40 times lower than the 100 hard examples (approximately 0.1 vs 2.3 each). Source: [3]

One solution for this problem is the use of the **α-Balanced Cross Entropy**, which is a common method for addressing this class imbalance adding a weighting factor $\alpha \in [0, 1]$ for class 1 and 1 - α for class -1. In practice α is usually set to a value represented inversely by the class frequency, this is, the higher the number of a given class, the lower the value of α, being the lower the loss value. The α-Balanced Cross Entropy is defined as below:

$$CE(p_t) = -\alpha_t log(p_t)$$

This loss is considered as an experimental baseline for the proposed Focal Loss.

The α-Balanced Cross Entropy balances the importance of positive/negative examples, but it does not differentiate between easy/hard examples, and this is the reason why the **Focal Loss** is the turning point of the paper. In order to address this sample-difficulty issue, the paper proposes to reshape the loss function, so the easy examples have a lower weight and thus focus training on negative outputs of hard examples. To achieve this goal a modulating factor $(1 - p_t)^\gamma$ is added to the Cross Entropy formula:

$$Focal\ Loss(p_t) = (1 - p_t)^\gamma log(p_t)$$

In this formula, we have the tunable focusing parameter γ, where $\gamma \geq 0$ and $\gamma \in [0, 5]$. In this definition of Focal Loss can be noted that, when p_t tends to 1, the modulating factor approximates to 0 becoming the loss value lower for well classified examples. The tunable focusing parameter γ adjusts the rate at which the example's down-sampling is made. When $\gamma = 0$, the Focal Loss formula is converted to the Cross Entropy formula, and as the γ value is incremented, the modulating factor value increases.

Demonstration of Focal Loss conversion to CE when $\gamma = 0$:

$$Focal\ Loss(p_t) = (1 - p_t)^0 log(p_t) \rightarrow Focal\ Loss(p_t) = 1 \times log(p_t) \tag{3.3}$$

$$Cross\ Entropy(p_t) = 1 \times log(p_t) \tag{3.4}$$

The authors of the RetinaNet point out that the γ value with best results in their experiments is $\gamma = 2$.

In practice, an α-balanced Focal Loss variant is used since it shows slight accuracy improvement over the normal Focal Loss implementation:

$$Focal\ Loss(p_t) = \alpha(1 - p_t)^\gamma log(p_t)$$

Finally, the authors say that the loss layer is implemented with the sigmoid function in order to obtain the probability p and use it in the loss function.

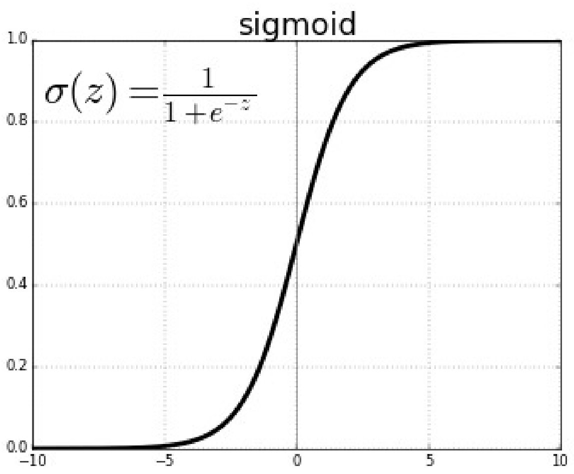


Figure 3.4: The sigmoid function converts the value given as input into a real value contained in the range $(0, 1)$.

3.1.2 Feature Pyramid Networks (FPN)

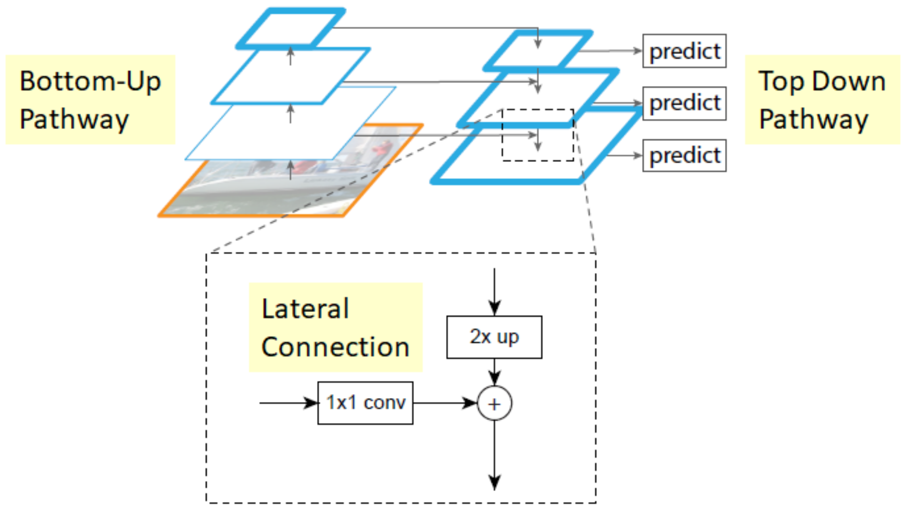


Figure 3.5: Feature Pyramid Network (FPN). FPN composes of a bottom-up and a top-down pathway. The bottom-up pathway is the usual convolutional network for feature extraction. As we go up, the spatial resolution decreases. With more high-level structures detected, the semantic value for each layer increases. FPN provides a top-down pathway to construct higher resolution layers from a semantic rich layer. Lateral connections are added in order to improve the upscaling process. Source: `https://jonathan-hui.medium.com/understanding-feature-pyramid-networks-for-object-detection-fpn-45b227b9106c`

The main purpose of the Feature Pyramid Networks (FPN) is to augment the standard convolutional network's output with a top-down pathway and lateral connections so the model gets a multi-scale feature pyramid from a single image (Figure 3.5). That way, each level of the pyramid can be used for detecting objects in a different scales.

3.1.3 Bottom-Up Pathway

The **Bottom-Up Pathway** is the feedforward computation of the CNN chosen as backbone, which computes a feature hierarchy composed by multiple feature maps, smaller and smaller in each step, specifically with a scaling step of 2. Usually, there are multiple layer producing outputs in the same hierarchy step so, in order to build our later feature pyramid, the last output of every step is taken. This choice is because the last layer of every step should have the strongest features.

3.1.4 Top-Down Pathway and Lateral Connections

The main objective of the **Top-Down Pathway** process is to get semantically stronger features using the outputs obtained in the Bottom-Up Pathway process. The key of this process is the use of different the different levels of the feature pyramid to get the most semantically rich features of the input image.

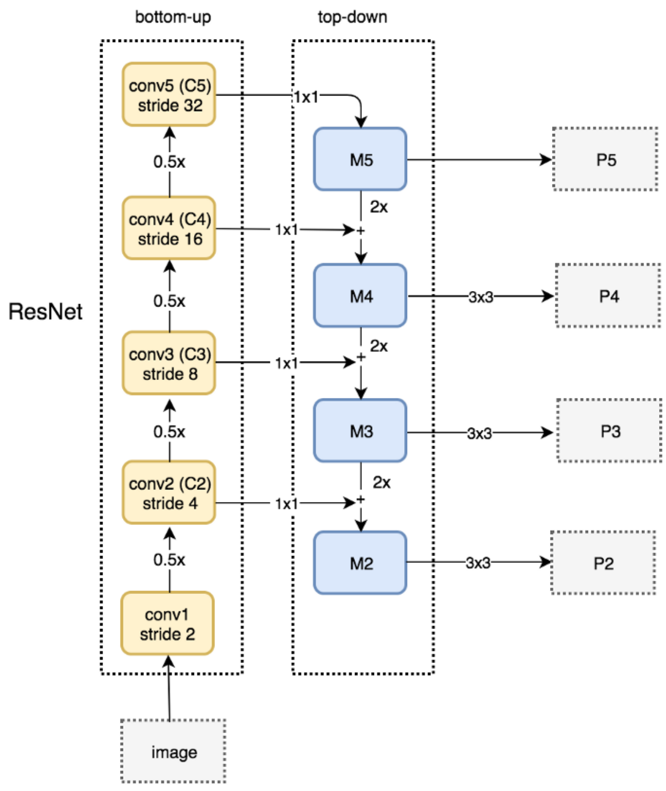


Figure 3.6: The *top-Down Pathway* process starts from the top of the feature pyramid and descends every level. The first layer is obtained from the last output of the Bottom-Up pathway. The 1×1 filter is applied to the given output, obtaining the *M5* layer and making a prediction on that layer. Then, the layer is upsampled, outputting the same feature map with twice the resolution. After the upsampling is done, an element-wise operation is done between the upsampled feature map and *C4* with the 1×1 convolution applied, obtaining the *M4* layer. After the element-wise merge operation, a convolutional filter of size 3×3 is applied to the *M4* layer. This process is repeated for the remaining levels of the feature pyramid.

The **Top-Down Pathway** process, as its name suggests, starts from the top of the feature pyramid and descends every level. The first layer, also known as the top of the pyramid, is obtained from the last output of the Bottom-Up pathway, this is, the output with the smallest resolution and semantically the richest one. The 1×1 filter is applied to the given output, obtaining the *M5* layer and making a prediction on that layer. After that prediction is done, the layer will be upsampled using the *nearest neighbours upsampling*[2] algorithm, giving as result the same feature layer as the previous one but with twice the resolution. Now that the upsampled feature map and the *C4* layer have the same dimensions, a element-wise operation is done between the upsampled feature map and *C4* with the 1×1 convolution applied which is passed from the bottom-up pathway through **lateral connections**, obtaining the *M4* layer. The element-wise operation enhances the features from the upsampled layer using the features from the bottom-up pathway. After the element-wise merge is done, a convolutional filter of size 3×3 is applied to the resulting *M4* layer, so the possible aliasing effect made by the merge operation is corrected, and a prediction is made with that filtered layer (*P4*). This process is repeated for the remaining levels of the feature pyramid (in Figure 3.6, M3 and M2 layers).

[2]The nearest neighbour algorithm has the effect of simply doubling rows and columns, doing an interpolation of the resulting point's neighbours.

3.2 ZeroCostDL4Mic project

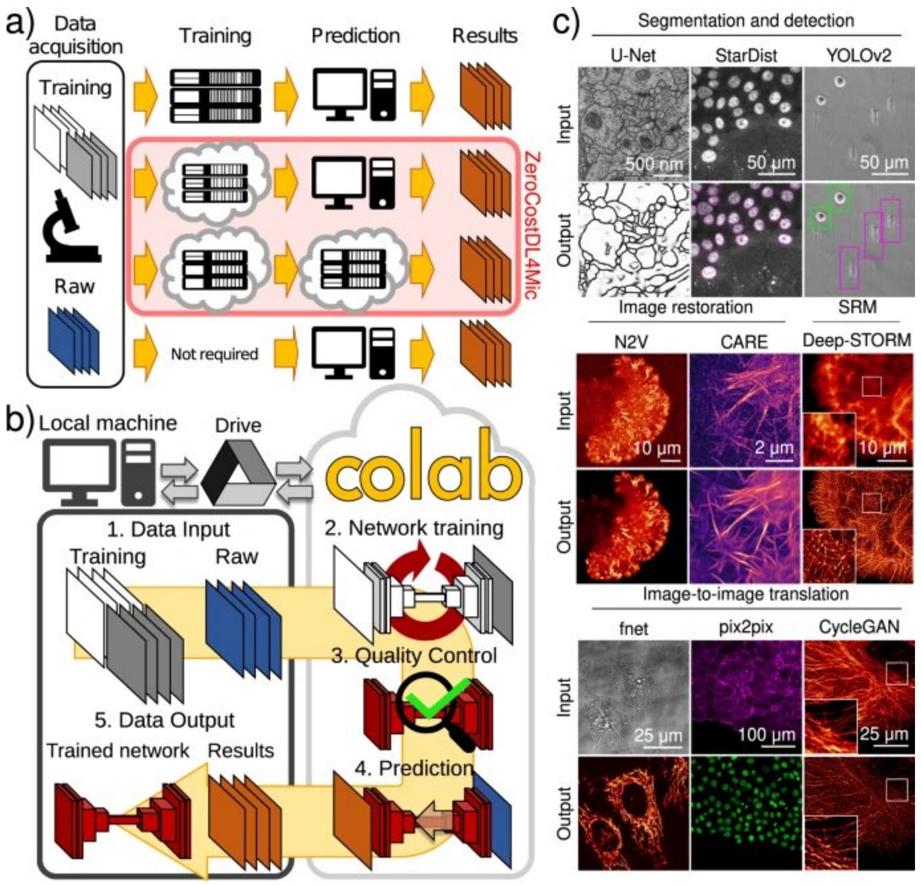


Figure 3.7: Visual summary of how ZeroCostDL4Mic works. This picture is taken from the ZeroCostDL4Mic original paper [4]. (a) Paths to exploiting DL. Training on local servers and inference on local machines (or servers) (first row), cloud-based training and local inference (second row), cloud-based training and inference (third row) and pre-trained networks on standard machines (fourth row). (b) Overview of ZeroCostDL4Mic. The workflow of ZeroCostDL4Mic, featuring data transfer through Google Drive, training, quality control and prediction via Google Colab. (c) Overview of the bioimage analysis tasks currently implemented within the ZeroCostDL4Mic platform.

The ZeroCost4Mic project is an entry-level, cloud-based Deep Learning-deployment platform (Figure 3.7-b) that simplifies Deep Learning use for microscopy. The project is a unified collection of self-explanatory Jupyter Notebooks, featuring an easy-to-use graphical user interface (GUI) that requires only a web browser and a Google account for a user to run any of their DL-based tasks (Figure 3.7-c). All calculations are performed in the cloud using Google Colaboratory (Colab for short), avoiding the need to purchase or install graphical processing units (GPUs) and associated software. Not needing GPUs nor computational capacity, together with being so simple to use for all kind of users (only a few mouse clicks are needed), make this project a great tool for non-experts users to use Deep Learning and automatise/improve plenty of visual tasks for microscopic images. Additionally, ZeroCostDL4Mic guides researchers on how to generate the training data necessary for Deep Learning, allowing them to get a deeper understanding of how the Deep Learning models work and giving them the capacity of running the models with different parameter configurations, so they can improve their results on specific datasets.

The ZeroCostDL4Mic project is composed of several models, each of them with the purpose of solving a specific task. Today those tasks include image segmentation and object detection (using U-Net [13], StarDist [14] and YOLOv2 [11]), image denoising and restoration (using CARE [15] and Noise2Void [16]), super-resolution microscopy (using Deep-STORM [17]) and image-to-image translations (using label-free prediction—fnet [18], pix2pix [19] and CycleGAN [20]).

Each of the ZeroCostDL4Mic notebooks follows the same workflow structure that covers all the crucial steps of every Deep Learning project, starting from the dependencies' importation, making the data processing, going through the model loading and training, and using that model, either for its validation or for inference over unseen data.

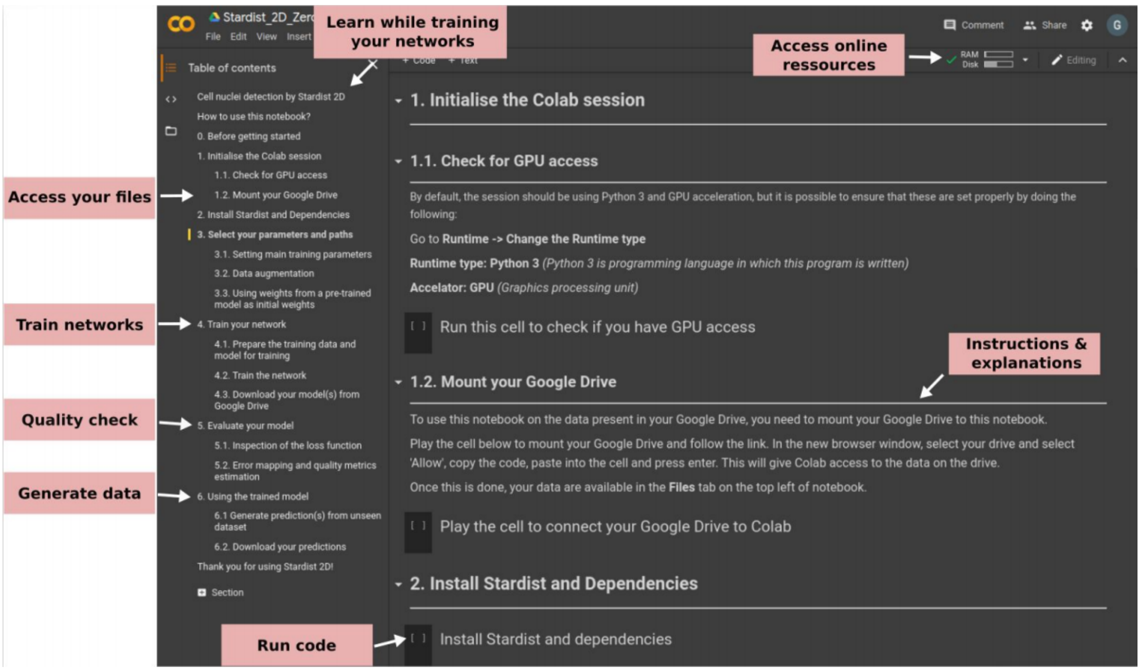


Table of contents

Cell nuclei detection by Stardist 2D
How to use this notebook?
0. Before getting started
1. Initialise the Colab session
1.1. Check for GPU access
1.2. Mount your Google Drive
2. Install Stardist and Dependencies
3. Select your parameters and paths
3.1. Setting main training parameters
3.2. Data augmentation
3.3. Using weights from a pre-trained model as initial weights
4. Train your network
4.1. Prepare the training data and model for training
4.2. Train the network
4.3. Download your model(s) from Google Drive
5. Evaluate your model
5.1. Inspection of the loss function
5.2. Error mapping and quality metrics estimation
6. Using the trained model
6.1 Generate prediction(s) from unseen dataset
6.2. Download your predictions
Thank you for using Stardist 2D!
Section

1. Initialise the Colab session

1.1. Check for GPU access

By default, the session should be using Python 3 and GPU acceleration, but it is possible to ensure that these are set properly by doing the following:

Go to **Runtime -> Change the Runtime type**

Runtime type: Python 3 *(Python 3 is programming language in which this program is written)*

Accelator: GPU *(Graphics processing unit)*

[ ] Run this cell to check if you have GPU access

1.2. Mount your Google Drive

To use this notebook on the data present in your Google Drive, you need to mount your Google Drive to this notebook.

Play the cell below to mount your Google Drive and follow the link. In the new browser window, select your drive and select 'Allow', copy the code, paste into the cell and press enter. This will give Colab access to the data on the drive.

Once this is done, your data are available in the **Files** tab on the top left of notebook.

[ ] Play the cell to connect your Google Drive to Colab

2. Install Stardist and Dependencies

[ ] Install Stardist and dependencies

Figure 3.8: ZeroCostDL4Mic notebook's example. A first view of the ZeroCostDL4Mic Jupyter notebook structure in the Google Colaboratory virtual environment. This picture is taken from the ZeroCostDL4Mic original paper [4].

In order to measure the model's quality, the ZeroCostDL4Mic notebooks contain the Quality Control (QC) section, where the user can test the model with new data (it should be unseen data for a fair measurement) and estimate quantitative metrics that indicate the model's prediction accuracy by comparing that prediction to ground-truth data. Those metrics vary for each model's task since their result can not be generalised and also their outputs aren't even related. For instance, object detection models output multiple real numbers indicating the objects contained in the given image and their associated classes, while denoising models output the input image without noise/stitches in it.

Another very common issue in Deep Learning is the lack of quality and/or quantity of data that will be used to build the model. Datasets are a crucial part of the model's generalisation capacity. The model will be built and it will learn according to the dataset's quality, since the dataset is the only learning source of the model. Taking this into account and knowing that these models will be built by users that might be out of the Deep Learning field, it is usually safe to think that those users may not have very dense datasets, limiting the model's improvement and generalisation, leading to a non-optimal performance. To address this issue, the ZeroCostDL4Mic has implemented one step where users can apply data augmentation techniques to their dataset. Data augmentation is a quite common solution that tries to lower the impact of having small datasets, enlarging the sample data by applying some transformations. In Figure 3.9, we can see multiple data augmentation techniques which, given the input image, generate 5 new images, this is, augments the dataset's size by 5 fold.

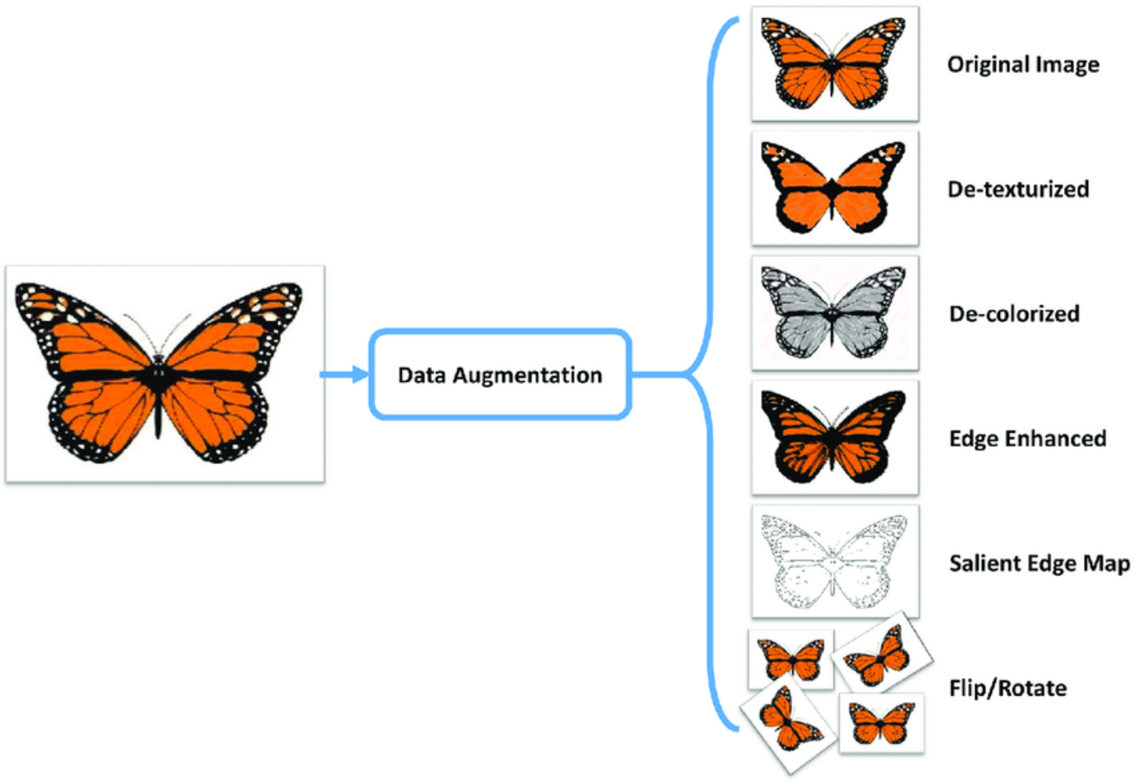


Figure 3.9: Examples of different methods of data augmentation. Different transformations are applied to the original image in order to artificially create new samples. Source: https://www.researchgate.net/figure/Data-augmentation-using-semantic-preserving-transformation-for-SBIR_fig2_319413978

Secondly, the ZeroCostDL4Mic project also includes the functionality to perform transfer learning via the loading of a pre-trained model as a starting model, rather than initialising the training with a blank model. This powerful approach allows the platform to benefit from the growing availability of pre-trained models from model Zoos[3] without compromising on the quality of the performance of a model on the specific data type provided by the user. This can also have several advantages in terms of shortening training times and reducing the amount of required training data. Moreover, it gives the user the chance of training again the same model several times in different epochs, having the advantage of accumulating new data over the time, so the model can be continuously improved.

[3]A model zoo is a machine learning model deployment platform where people upload already trained models, so any other person can reuse that model without the need of training it again from scratch

3.2.1 ZeroCostDL4Mic in Google Colab

As we have mentioned before, the ZeroCostDL4Mic projects are designed and implemented in the Google Colab environment, what means that the users have free access to high-performance computing resources needed to run the broad range of Deep Learning networks. However, to take advantage of these resources, users usually need to have previous programming knowledge, which limits the range of researchers that are able to exploit this environment. By establishing a user-friendly and efficient interface with Google Colab, the ZeroCost project aims to leverage this cloud-computing system to deploy state-of-the-art Deep Learning models for microscopy.

Google Colab provides access to remote virtual machines with free but finite resources which are made available for a specific runtime duration (maximum 12h). Even if this time limitation seems like a big inconvenience, one of the most important resources that Google Colab lends us, if not the most important, is high-end GPUs, a very expensive component that allows Deep Learning models to train and run between $5\times$ and $200\times$ faster than a normal PC's Central Processing Unit (CPU), making feasible for everyone with any kind of PC to use this technology.

Considering the resources available with Colab, ZeroCostDL4Mic is considered to be well suited for:

1. Prototyping image-analysis workflows and pipelines without financial investment.
2. Executing small-to-medium-size projects (a few 10's of GB of data) compared to large-scale projects often encountered in machine vision research.
3. Short-term projects not requiring a permanent investment in Deep Learning infrastructure.
4. As a resource for Deep Learning enthusiasts and students to learn about Deep Learning methods and state-of-the-art architectures, such as U-Net [13] or (generative adversarial networks) GANs [21].

In addition, the ZeroCostDL4Mic notebooks are designed to be run outside Google Colab in case the user wants it, downloading the notebooks in Jupyter notebook or Python file format.

3.3 Our implementation of RetinaNet in the ZeroCostDL4Mic project

To achieve the main purpose of this project, we decided to implement a Deep Learning based object detection solution as part of the ZeroCostDL4Mic platform. This way, users with no computer science background can apply this technology in their research. As aforementioned, ZeroCostDL4Mic already has some object detection models implemented, such as YOLOv2 [11]. However, that method is from 2016 and novel approaches have demonstrated to perform better in precision and speed. For that reason, we decided to implement RetinaNet, a one-stage solution with the current best benchmark results as described in Section 3.1.

3.3.1 Implementation library

Deep Learning is an emerging science whose core is based on highly complicated numerical computations that, making from scratch can take a very long time. Nowadays is very common to use already created libraries[4] that make it so much easier to build a Deep Learning model, making the numerical computations transparent to the developer, while performing the best as possible from both software and hardware points of view. Nowadays, two of the most popular Deep Learning libraries used by the community are TensorFlow [22] and PyTorch [23]. In this document, the difference between the two libraries will not be discussed but our choice for this project will be explained.

Although the original implementation of RetinaNet was presented in PyTorch by the Facebook AI Research developers, our implementation has been done in TensorFlow in order to learn it's technology, stateful graph structure and the object detection API[5] they made for the version 2.x of TensorFlow (TensorFlow 2), which is an open-source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models.

One of the most important reasons we decided to use the TensorFlow 2 (TF2) object detection API was the need of using pre-trained models, and the TF2 object detection API contains a Model Zoo of ready-to-use models. These models are pre-trained on various

[4]A library is a reusable chunk of code that can be included in your code.

[5]The acronym API stands for application programming interface, which is an interface that defines interactions between multiple software applications or mixed hardware-software intermediaries.

datasets such as COCO (Common Objects in Context) [24] dataset, the KITTI [25] dataset and the Open Images Dataset [26]. This need is related to the ZeroCostDL4Mic project's purpose of creating a Deep Learning model that, training in the less time possible, shows a good performance over new data, and this goal can only be achieved loading a knowledge base previously acquired with large datasets, so the model can learn new features way easier.

The TF2 object detection API libraries are available from their Github's page. Once the libraries are installed we can start preparing the environment.

3.3.2 Data pre-processing

First of all, the user needs to specify the path to the input images and their annotations. Next, we have to pre-process and save that data so lately can be used for training and validation purposes. Each image can contain n number of objects, being $n \in [0, \infty)$. All the data of every object of an image is saved in a table, including the bounding box location of every object in that image with its actual corresponding class. Then, all the bounding box coordinates are normalised to convert those numbers into the range $[0, 1]$ following this formula:

$$\begin{aligned}
\hat{Y}_{\text{min}} &= \frac{Y_{\text{min}}}{\textit{image height}} \\
\hat{X}_{\text{min}} &= \frac{X_{\text{min}}}{\textit{image width}} \\
\hat{Y}_{\text{max}} &= \frac{Y_{\text{max}}}{\textit{image height}} \\
\hat{X}_{\text{max}} &= \frac{X_{\text{max}}}{\textit{image width}}
\end{aligned}$$

where Y_{min} and X_{min} correspond to the origin (X, Y) of the bounding boxes, $\hat{Y}_{\text{min}}$ and $\hat{X}_{\text{min}}$ are those starting points normalised, Y_{max} and X_{max} correspond to the opposite corner coordinates (X, Y) of the bounding boxes, $\hat{Y}_{\text{max}}$ and $\hat{X}_{\text{max}}$ are those maximum points normalised and *image height and image width* are the dimensions of the image being processed ($\textit{width} \times \textit{height}$) (Figure 3.10).

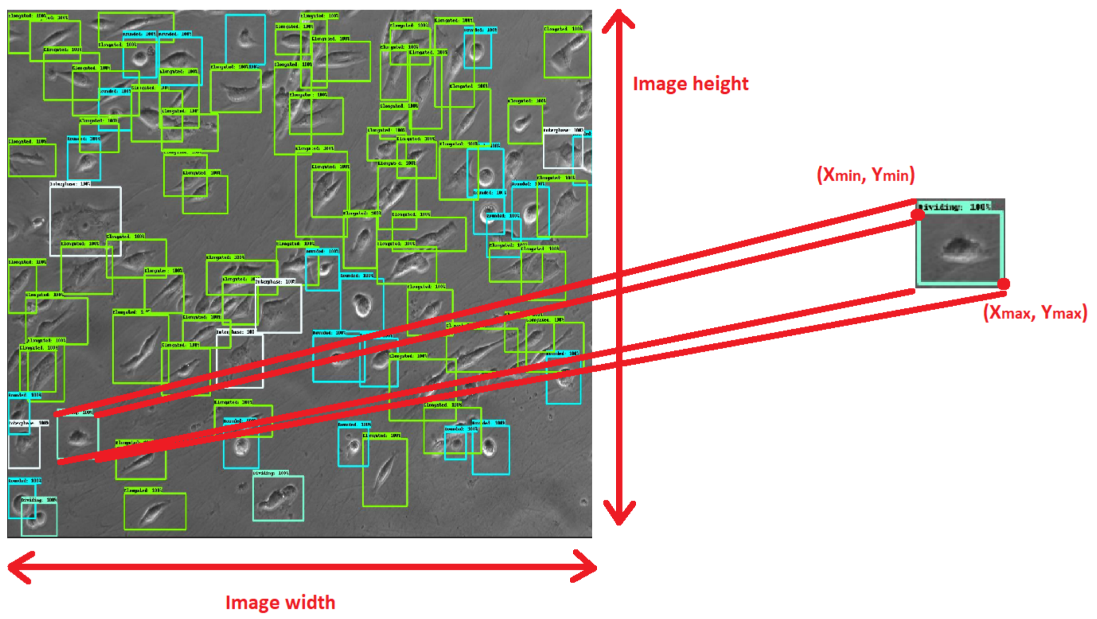


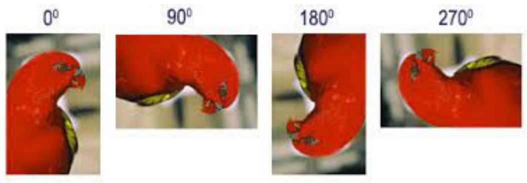
Figure 3.10: Representation of the variables used in the normalisation process.

The **validation set** is a a subset of the training set which provides an unbiased evaluation of a model fit while tuning the model's hyperparameters. The model does not tune its hyperparameters using this subset, so it provides a good estimate of the performance of the model on unseen data. The user choose what percentage of the training set will be used as validation set.

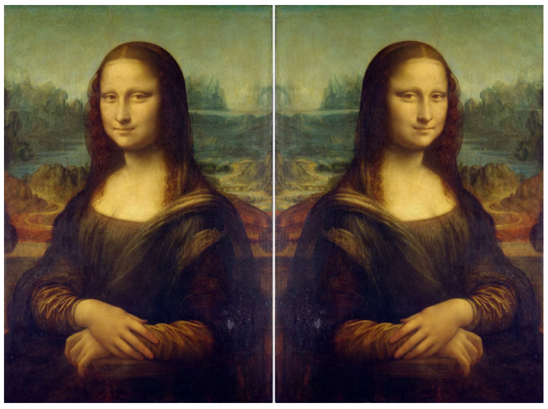
The classes names are saved for their possible display in the predictions and a folder to store the model is created, so it can be used afterwards.

3.3.3 Data augmentation

The next step in our implementation is to give the user the possibility to augment the training dataset using simple data augmentation techniques such as image flipping and rotation (Figure 3.11). This step can make the model to generalise better over the data, resulting in a better quality predictor. The data augmentation is only applied to the training set since it does not matter how big the validation set is because the model does not improve with it and is just oriented to measure how well the model is generalising. Also, it is important that data augmentation is done once the validation set and the training set are split. Otherwise augmented versions of the same image may end up in both the training and the validation sets, jeopardising the validation metrics. In our implementation, the training data can be augmented up to 8 times.



(a) Image rotation example.
Source: http://www.rmig.com:8080/erez3/html/index.html



(b) Image flip example.
Source: https://ipiccy.com/blog/357-2/mona-lisa-flipped-example

Figure 3.11: Examples of data augmentation transformations permitted in our implementation.

3.3.4 Creating a model from pre-trained model's weights

As we mentioned before, one of the most important reasons of using the TF2 object detection API is the ease of using a pre-trained model from a Zoo[6] that the library's authors point out, so we start training a new object detection model with already trained weights. The step after data augmentation is where the model is created with the configuration selected by the user. This is, the user choose what CNN to use as the model's backbone from a list, and the model's weights and configuration will be downloaded from the Zoo. The user can change some configuration parameters such as the *localisation weight* and the *classification weight*. These parameters are in charge of setting how the model's loss function will penalise both, the prediction of the objects classes and their localisation. These weight are real values contained in the range $[0, \infty)$ and they are set to 1.0 by default.

To better understand how these parameters work, let us see an example: If the user wants to prioritise that the model predicts the location well rather than the classification, the *localisation weight* can be set to 4.0 and the *classification weight* to 1.0. This decision makes the model's loss function be

$$Total\ loss = 1.0 \times classification\ loss + 4.0 \times localisation\ loss$$

[6]The model zoo can be found at https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

focusing more on improving the localisation prediction than the classification one.

In the current implementation, we have three possible backbone models to choose from:

- **ResNet50:** The classical Residual Network (ResNet) [27] that is 50-layers deep and takes as input images of size 640×640 pixels. This is the same CNN as the one used in the RetinaNet's original paper.

- **MobileNet v1:** MobileNet [28] v1 is an efficient model thought to be run in mobile and embedded vision applications. This network is worse than ResNet50 in terms of accuracy but better in terms of speed. This CNN takes as input images of size 640×640 pixels.

- **MobileNet v2:** Same network as the previous one, with some speed enhancements (almost twice faster than MobileNet v1) but worse accuracy results. This CNN takes as input images of size 320×320 pixels.

All the previous models use the FPN algorithm and are implemented as single-shot detectors.

When the weights are about to be loaded into the new model, two configuration parameters are always changed: the freezing of the batch normalisation and the trainable layers of the model.

Batch normalisation freeze

A **batch** is a cluster of n samples, being $n \in (0, training\ samples]$, and is the number of samples that are processed in the training process by the model before it tunes its weights. It can be said that the model "learns" taking into account each batch's loss value.

When we do not want a model to change the weights of one or more layers, we *freeze* those layers, avoiding them to tune their weights on training (Figure 3.12).

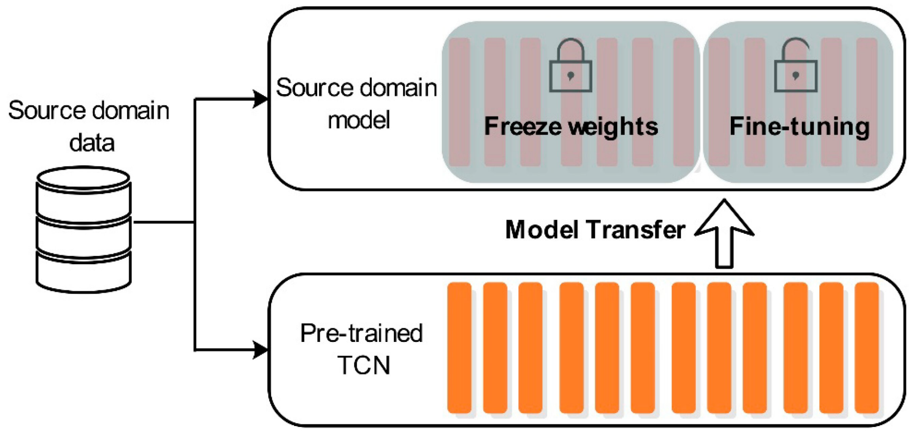


Figure 3.12: Fine-tuning a pre-trained model. Using an already trained model (bottom), transfer learning is used so some of the layers of the model are frozen, i.e., their weights won't change during the training process and the remaining layers are fine tuned with new data (top). Source: [5]

Batch normalisation [29] is a method used to make artificial neural networks faster and more stable through normalisation of the layers' inputs. The batch normalisation is based on converting every value in the batch to a value contained in the range $[0,1]$, being 1 the highest value of the actual batch. This method not only makes training faster, in some cases by halving the epochs[7] or better, and provides some regularisation, reducing generalisation error. The batch normalisation method uses two learnable parameters, β *and* γ [29], which are learned along with the original model parameters, and restore the representation power of the network. These two parameters are the reason of freezing the batch normalisation in the new model, because, as we are loading the parameters of a previously trained model, we are interested in keeping the original values that the previous trained model learned.

Another parameter that is changed is the model's training possibility, this is, if the model can tune its weights when training or they are frozen so the model can only be used for inference[8]. Since we want to train the model with new data, this parameter has to be set as *True* (training enabled).

Another option that the user has in this step is to avoid the creation of the new model and loading the weights from a previous model that was trained and saved in order to continue training it, with the same or a new dataset.

[7] An epoch is one training loop, this is, an iteration over all the batches of the dataset, when the model has seen all the training data.

[8] Inference is the process of running live data points into a machine learning algorithm to calculate an output.

3.3.5 Training the model

Now that we have an object detection model created and pre-trained weights are loaded, we will fine tune that model with the user-provided data. In order to save the actual model for a future use, the model's configuration and the dataset's classes are saved in the path that the user specified in Section 3.3.2. Including the model weights, these two blocks of information are the core necessary to load the current model in a future. Once teh two files with that information are created and saved, we start the training process.

Preparing data to train

Before the model starts training, we have to mould the data we have pre-processed before so that the model can take it as input. In order to train our model we need three different data subjects: the images' RGB arrays, the ground truth position of the bounding boxes and the actual class of each object. We extracted, processed and saved this data in Section 3.3.2, but the model needs the inputs to be in tensor form. A **tensor** is a generalisation of vectors and matrices and is easily understood as a multidimensional array.

The first thing we do is to take all the images in the training set path, convert them into arrays and then, if they are originally in the greyscale chromatic mode, we convert the image to RGB (Red Green Blue). Finally we convert that array in a tensor and that would be the final version of how an input image has to be for our model. The next step is to convert the bounding box coordinates to tensors. The last step to have all the data ready is moulding the classes of each object to be model's input. Since we have the objects' bounding boxes and their classes previously ordered, we only have to take care of converting the classes to tensors. But there is a little problem, the classes are strings (e.g., dog, cat, parrot, etc.) and can't be converted to tensors. To solve this issue, we convert all the classes to a numerical value associated to its categorical value, this method is called *integer encoding*.

	Class 1	**Class** 2	**Class** 3
String	Cat	Dog	Parrot
Numerical value	1	2	3

Table 3.2: Example of integer encoding.

Now that we have the ground truth classes as numerical data, we convert it to tensor, but it is not its final form because the model asks that said input be one-hot encoded. This is because, for categorical variables where no such ordinal relationship exists, the integer encoding is not enough. In fact, using this encoding and allowing the model to assume a natural ordering between categories may result in poor performance or unexpected results, this is, in Table 3.2, the model can assume that a Parrot is more important than a Cat because a Parrot integer value is 3 and the cat's one 1 ($1 < 3$).

One-hot encoding is a method that is used to avoid this mentioned issue. One-hot encoding is a process by which categorical variables are converted into a form that could be provided to ML algorithms to do a better job in prediction. In Table 3.3 is represented how the one-hot encoding method would work for the example seen in Table 3.2. The mechanic is to create a different field for each existing class and then set the object's actual class to 1 and the remaining classes to 0.

	Cat	**Dog**	**Parrot**
Animal 1	0	1	0
Animal 2	1	0	0
Animal 3	0	0	1
Animal 4	1	0	0

Table 3.3: One-hot encoding example. The table represents that the *Animal 1* is from class *Dog*, the *Animal 2* is from class *Cat*, the *Animal 3* is from class *Parrot* and *Animal 4* is from class *Cat*.

Finally, after applying the one-hot encoding conversion to the classes' tensor, we have the 3 necessary training input tensors and, with this, all the data moulded to start the training process.

Training process

Even though the TF2 object detection API gives the option to do both the training as well as the validation and inference process automatically, in this project we have decided to make those processes manually, using low level TensorFlow code and functions that manage the model's **graph**, updating the parameters and working directly with gradients. This decision was made looking for self-learning of both the bases of Deep Learning and the TensorFlow library and also more technical benefits for the project such as making it a lot more customisable.

Graphs are data structures that contain a set of TensorFlow operations, which represent units of computation; and tensors, which represent the units of data that flow between operations. Since these graphs are data structures, they can be saved, run, and restored all without the original Python code (Figure 3.13).

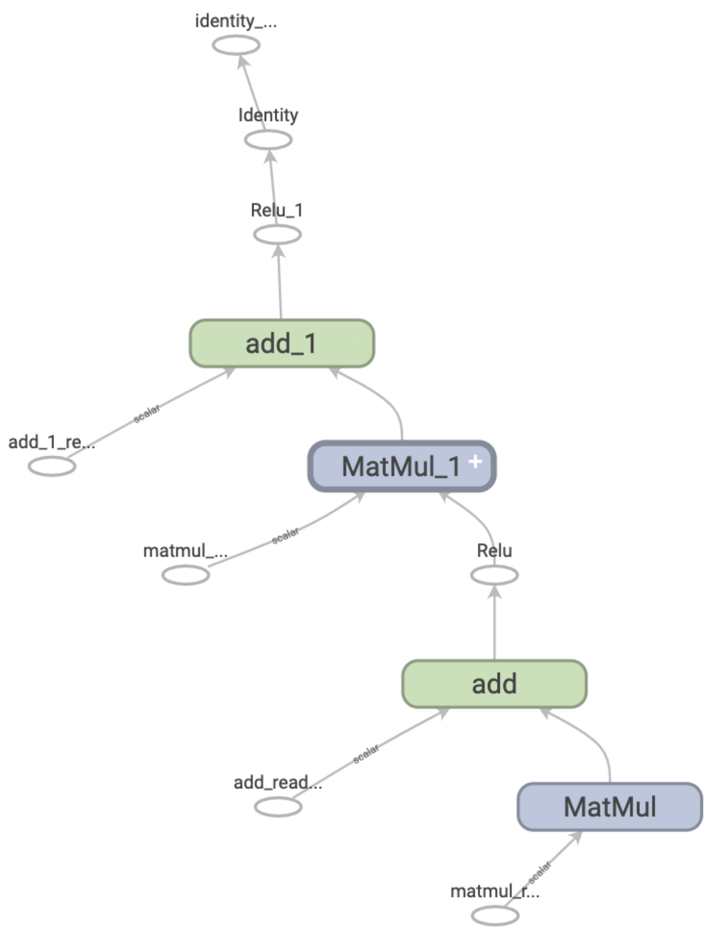


Figure 3.13: Example of a graph defined in TensorFlow. Source: https://www.tensorflow.org/guide/intro_to_graphs

The first step done in the training process is to calculate how many batches per epoch (*batches_per_epoch*) and batches for all the training process (*num_batches*) will be. This information will be used in the training loop to know when the model has to stop training (when the batch *num_batches* arrives) and for making some calculations when one epoch finishes (*batches_per_epoch*). These two values are calculated as follows:

$$batches\_per\_epoch = \frac{number\ of\ training\ images}{batch\ size}$$
$$num\_batches = number\ of\ epochs \times batches\_per\_epoch$$

where *batch size* and *number of epochs* are given by the user in Section 3.3.2. After those two values are calculated, we specify which weights has to fine tune, being these weights all those referring to object localisation and classification.

Now, we define the train and validation steps. These steps are defined using some low-level TensorFlow functions which access directly the model's graph. Each step takes a batch of the three different tensors created in the data preparation process (images tensors, groundtruth locations and groundtruth classes) and outputs their three corresponding loss values: Localisation loss, classification loss and total loss (the sum of the localisation and classification loss). The training step works as follows:

1. We create a variable that will be stored in the graph. This variable's dimensions have to be specified so TensorFlow can manage the environment's memory underneath. The dimensions that will be specified are $Batch\ size \times 640 \times 640 \times 3$ (image width, height and RGB channels for each image in the batch).
2. The objects' groundtruth locations and the objects' groundtruth classes are provided to the model.
3. Since the images from the user's dataset can be of any size, they have to be resized to match the model's backbone's input dimensions.
4. A prediction for the images in the batch passed as input is done, taking the predicted locations and classes for their corresponding objects.
5. Now, having the model's predictions and the images' groundtruth information, the loss value is calculated by the model itself, giving as output both, the localisation loss and the classification loss.
6. Taking into account the loss values, we calculate the total loss of the batch making the sum of the two previous losses.
7. The gradient is computed using the total loss value and the hyperparameters we want to fine tune, returning the loss function's derivative.
8. Backpropgation is carried out to compute the gradient of the loss function with respect to each weight of the network.

The validation step is analogous. It contains the same structure as the training step with only one difference, the model does not make the gradient descent process, avoiding it to improve with the given input. Knowing this, the validation step would be the same as the validation step removing the points 7 and 8.

With the training and validation steps defined we create the **optimiser**. Optimisers tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimisers aim at shaping and moulding your model into its most accurate possible form by adjusting its weights. The loss function is the guide to the terrain, telling the optimiser when it's moving in the right or wrong direction. There is a wide variety of optimisers, but in this project we opted to use Adam [30] since we found it to work generally better, even though in the first instance we experimented with the Stochastic gradient descent (SGD) [31] optimiser.

Next, the training loop is made. Here, the model will train for the number of epochs that the user specified previously, running a training step with each batch in a epoch and a validation step at the end of every epoch. This way, the user can see how the model is evolving during the training phase and its actual improvement. The training loop is build as follows:

1. A subset with a size corresponding to the batch size of images and their corresponding groundtruth information is taken from the training dataset.
2. A training step is computed using the previous randomly chosen batches.
3. Once an epoch is completed, a validation step is performed and evaluation metrics are displayed to the user. In our case, we use the Mean Average Precision (mAP) metric, a popular evaluation metric used for object detection (i.e. localisation and classification).
4. Repeat the same process for all the batches that defined one epoch.
5. Finally, model weights are saved and all the loss values calculated in each epoch are exported to a local file.

3.3.6 Quality Control

Once the user has a functional model, it can be evaluated for any input. In this section, the user can set the paths to test images and their annotations in order to test the model's quality, showing its predictions together with the mAP values. Later, a comparison between the original test image, the predicted image and the groundtruth image will be displayed to the user.

Finally, the model can be used for predicting any kind of input images (the same type of images that the model has been trained for), so in this final step the user hasn't got the need of using annotations.

3.4 User's interaction in the project

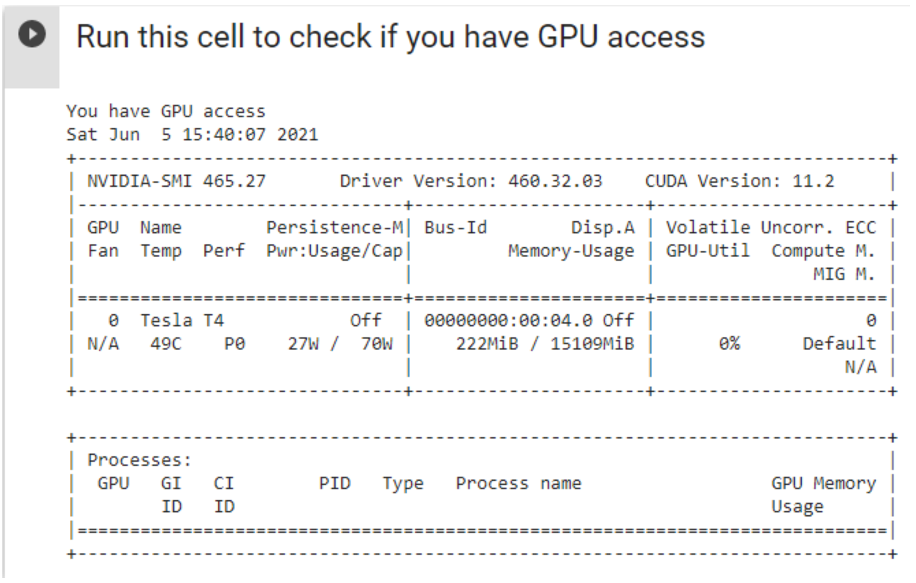
In this section we will show an example of execution from the user's perspective using the dataset described in Section 4.2.

The run that will be seen in this section is made using *the example dataset* which can be downloaded in the notebook's Section 3.

Note that only some cells in the notebook are mandatory to be executed, and we will mention them as we go.

3.4.1 Initialising Colab session

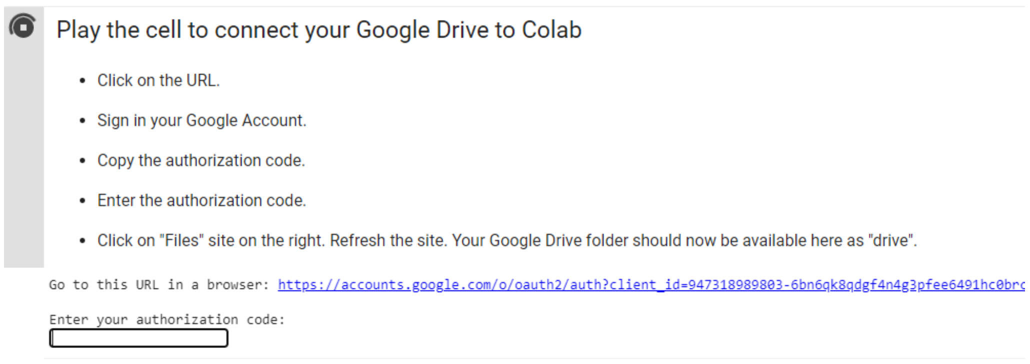
The first runnable cell of the notebook will connect the notebook to Colab and check if we are granted with access to a GPU (see Figure 3.14).



```
You have GPU access
Sat Jun  5 15:40:07 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 465.27       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   49C    P0    27W /  70W |    222MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Figure 3.14: Example of how the GPU access checking is displayed. It can be appreciated that the Figure's GPU model is a Tesla T4, having 16GB of VRAM memory.

The next step is to connect with a Google Drive account to which the user will have read/write rights, this is, the user will be able to create, delete and read files contained in that Google Drive account (see Figure 3.15a). The field under the *Enter your authorisation code* text needs to be filled with a code obtained in the link shown also in the output.

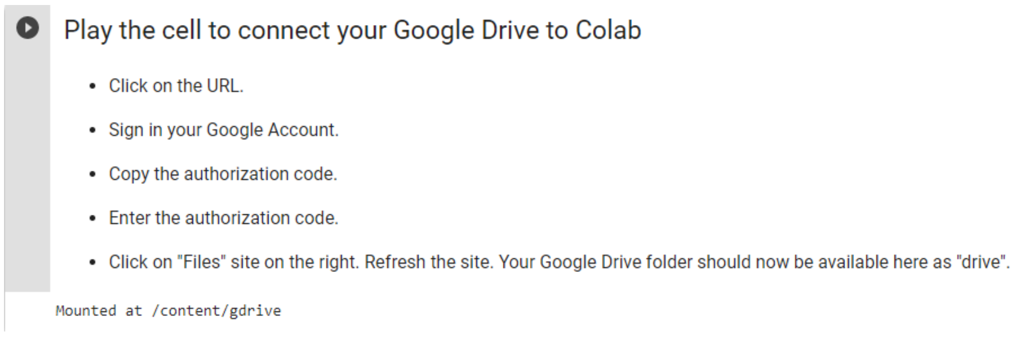


Play the cell to connect your Google Drive to Colab

- Click on the URL.
- Sign in your Google Account.
- Copy the authorization code.
- Enter the authorization code.
- Click on "Files" site on the right. Refresh the site. Your Google Drive folder should now be available here as "drive".

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc

Enter your authorization code:
```

(a) Before the user is connected to a Google Drive account.



Play the cell to connect your Google Drive to Colab

- Click on the URL.
- Sign in your Google Account.
- Copy the authorization code.
- Enter the authorization code.
- Click on "Files" site on the right. Refresh the site. Your Google Drive folder should now be available here as "drive".

```
Mounted at /content/gdrive
```

(b) After the user is connected to a Google Drive account.

Figure 3.15: Example of what the user sees before and after he/she connects with a Google Drive account.

3.4.2 Installing dependencies

Next, the model dependencies are installed and imported, together with our own code functions. After this cell is run, a message which says that everything has been done with no errors should be shown to the user.

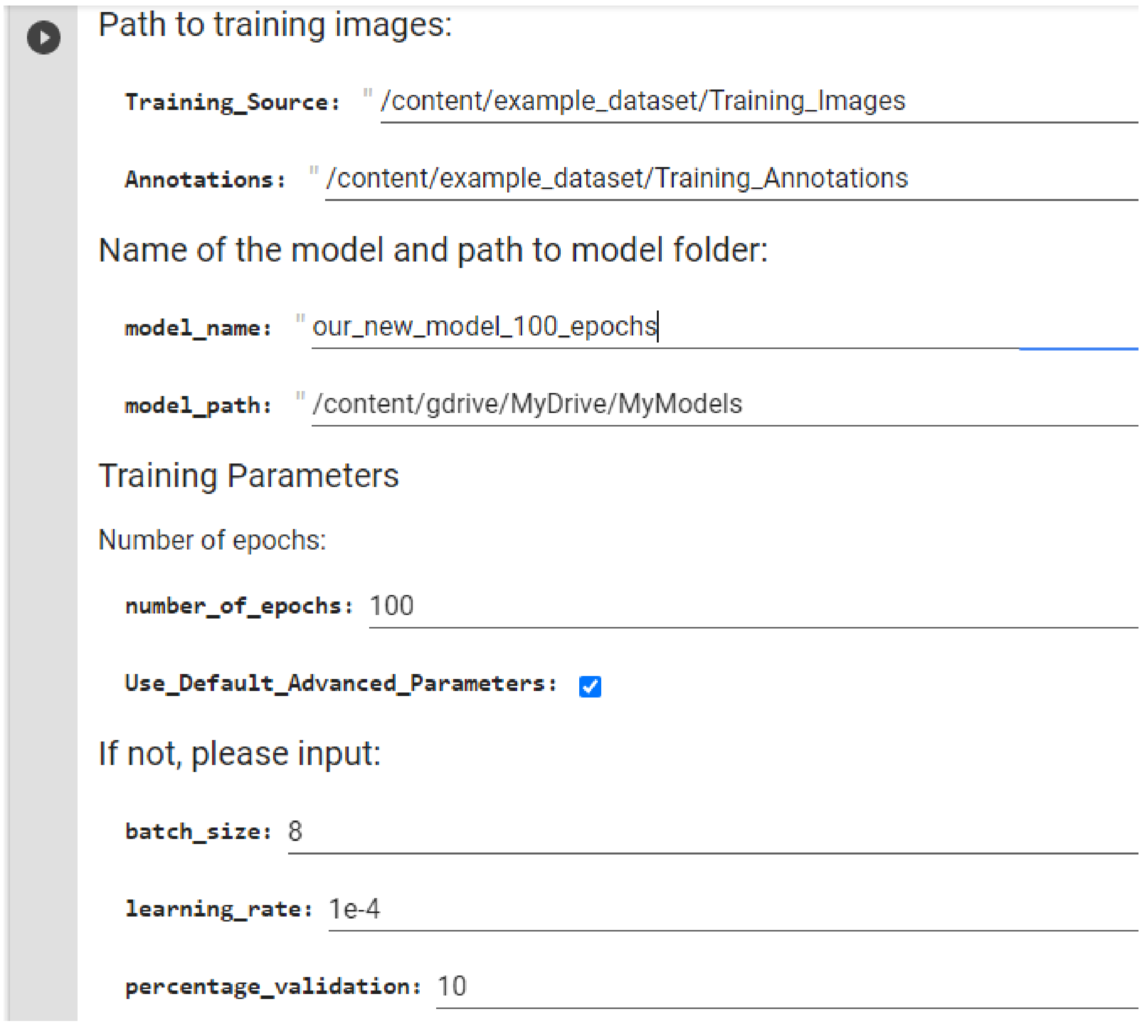
3.4.3 Setting the training parameters

Now that all the dependencies are installed the notebook is ready to be used.

When we were building the notebook we stop to think about the type of user who only wants to test the model. For this kind of user it would be probably annoying looking for an annotated dataset with the XML PASCAL VOC format required on the internet so we decided to create a optional cell at the beginning of Section 3 that can be used for downloading an already annotated dataset with the required format and placing it on the user's Colab's current session so, when the user finishes using the notebook, the dataset would be deleted automatically. We will refer to this dataset as *the example dataset*, which is saved in the */content/example_dataset* path.

Note: This dataset is defined later, in Chapter 4.

After this last optional cell comes the cell where the model's training parameters are set as well as the dataset that will be used to train the model. This cell only has to be executed if user wants to train a model. The parameters that are set in this cell are the ones displayed in Figure 3.16. Most of the fields are filled with default values that users can change as they like, but the *Training_Source, Annotations, model_name and model_path* fields are mandatory to be filled with their respective paths (The *Training_Source and Annotations* fields contains the example dataset's training images and annotation paths by default for instructional purposes).


Path to training images:

Training_Source: " /content/example_dataset/Training_Images

Annotations: " /content/example_dataset/Training_Annotations

Name of the model and path to model folder:

model_name: " our_new_model_100_epochs

model_path: " /content/gdrive/MyDrive/MyModels

Training Parameters

Number of epochs:

number_of_epochs: 100

Use_Default_Advanced_Parameters: ✓

If not, please input:

batch_size: 8

learning_rate: 1e-4

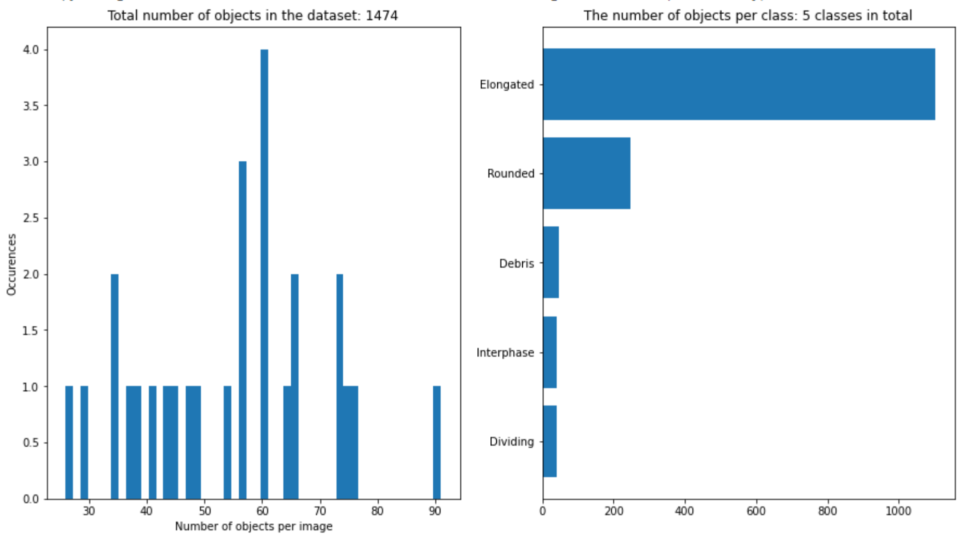
percentage_validation: 10

Figure 3.16: Fields of the parameters to set for training.

The *Use_Default_Advanced_Parameters* checkbox is used for using the training parameters defined by us if is checked. If it is not checked the parameters that will be used are the ones of the fields that come after the *If not, please input:* text. Those parameters have the same values as the following fields in Figure 3.16. Once the cell is run, a folder for this new model is created in the path *model_path* + *model_name* (I.e. */content/gdrive/MyDrive/MyModels/our_new_model_100_epochs* in the example of Figure 3.16). Also, two new folder called *Validation_Source* and *Validation_Source_annotations* will be created in the dataset's path where a random number *n* of images from training set's will be moved and used as validation set, being *n* the number corresponding to the training percentage taken as validation chosen by the user.

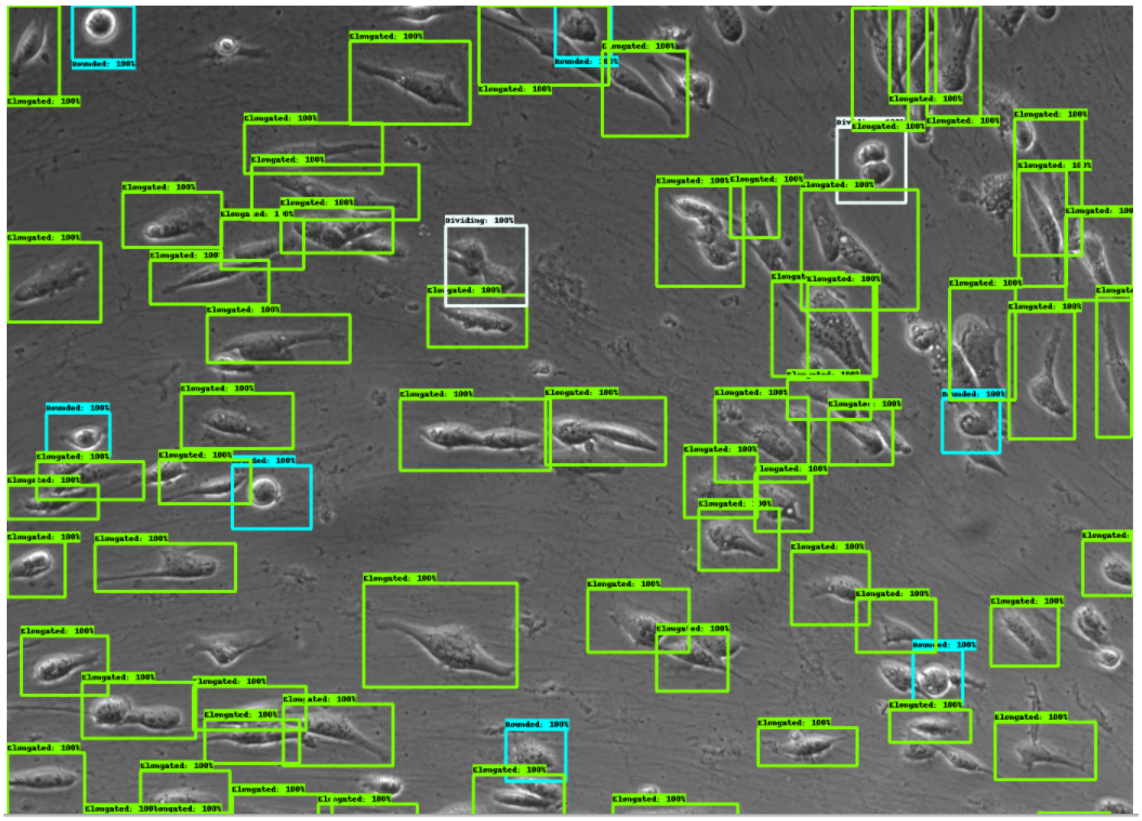
Once the cell has been run dataset's statistics will be displayed as well as the representation of a image from the training set with its groundtruth information over it as it can be seen in Figure 3.17b.

```
Default advanced parameters enabled
Counter({'Elongated': 1103, 'Rounded': 246, 'Debris': 45, 'Dividing': 40, 'Interphase': 40})
```



```
Now an image from the training set will be shown with its annotations in order to visualise that the annotations are properly matched.
```

(a) Statistics output for the example dataset. It is displayed how many objects per each class are in the dataset and two different plots that gives a general idea to the user of how the chosen dataset is.

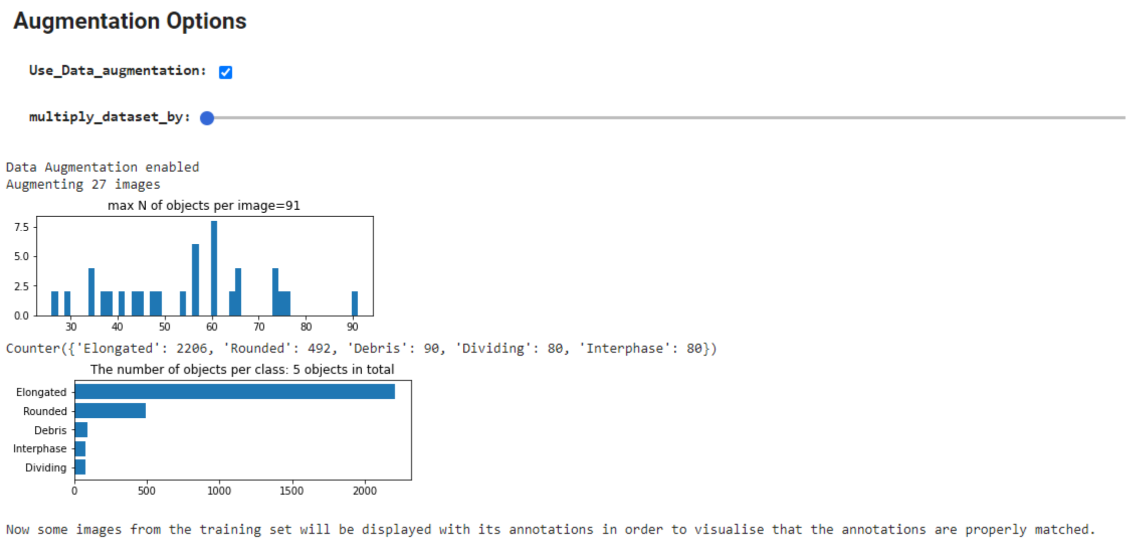


(b) An image from the training set is displayed with its corresponding annotations (the ground truth objects localisation and class) drawn over it.

Figure 3.17: Output example of cell in Section 3.1 of the notebook.

3.4.4 Data Augmentation

After having defined the training set path and having it split from the validation set now the user can perform the data augmentation technique in order to make his/her training set richer. Is important to know that doing data augmentation is not mandatory, but running this cell it is in order to train the model later. This happens because some variables are initialised inside this cell, so if the user doesn't want to make the data augmentation he/she has to uncheck the *Use_Data_Augmentation* checkbox and run the cell shown in Figure 3.18.



Augmentation Options

`Use_Data_augmentation:` ✓

`multiply_dataset_by:`

```
Data Augmentation enabled
Augmenting 27 images
```

```
Counter({'Elongated': 2206, 'Rounded': 492, 'Debris': 90, 'Dividing': 80, 'Interphase': 80})
```

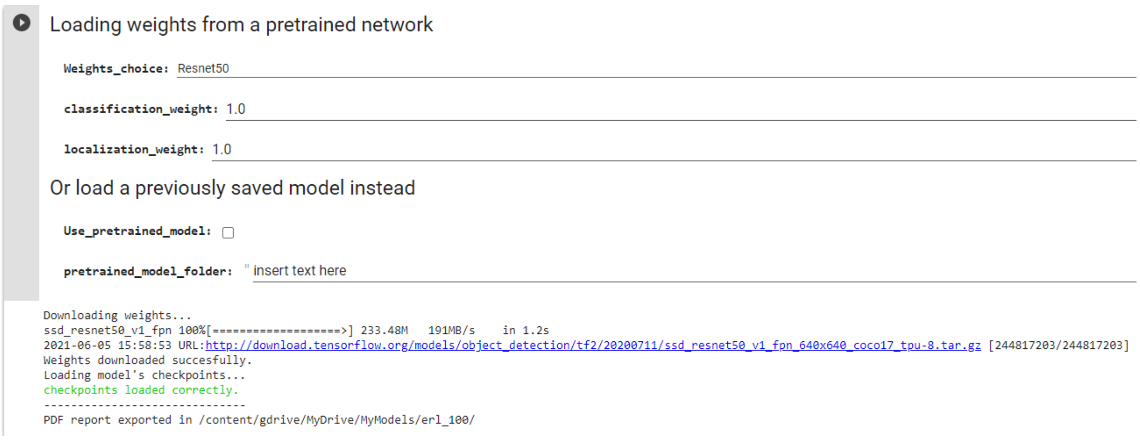
```
Now some images from the training set will be displayed with its annotations in order to visualise that the annotations are properly matched.
```

Figure 3.18: Example of the data augmentation cell and its output.

If the user unchecks the *Use_Data_Augmentation* checkbox then the data augmentation process will not be done and it will be represented in the output by a simple text that says "Data augmentation disabled". Otherwise, the data augmentation process will be carried out, multiplying the training set the same amount of times as established in the *multiply_dataset_by* slider, being 8 the maximum amount of times. After the data augmentation process is finished, creating two new folder where the augmented images and their annotations are saved are placed in the dataset's defined path, an output similar to the one of Figure 3.18 will be displayed, showing the same type of statistics as in the *Setting the training parameters* cell (section 3.1) and then a few augmented images with their augmented annotations drawn over them like in Figure 3.17b.

3.4.5 Loading weights from a pre-trained network

At this point the dataset that will be used should be ready. As we mentioned before, we don't want to create this model from scratch but we want to create it and load the weights from another pre-trained model, so the only thing we have to do is fine tune it with the dataset with which the model has to work.


Loading weights from a pretrained network

Weights_choice: Resnet50

classification_weight: 1.0

localization_weight: 1.0

Or load a previously saved model instead

Use_pretrained_model: ☐

pretrained_model_folder: " insert text here

```
Downloading weights...
ssd_resnet50_v1_fpn 100%[===================>] 233.48M   191MB/s    in 1.2s
2021-06-05 15:58:53 URL:http://download.tensorflow.org/models/object_detection/tf2/20200711/ssd_resnet50_v1_fpn_640x640_coco17_tpu-8.tar.gz [244817203/244817203]
Weights downloaded succesfully.
Loading model's checkpoints...
checkpoints loaded correctly.
------------------------------
PDF report exported in /content/gdrive/MyDrive/MyModels/erl_100/
```

Figure 3.19: Loading weights section cell.

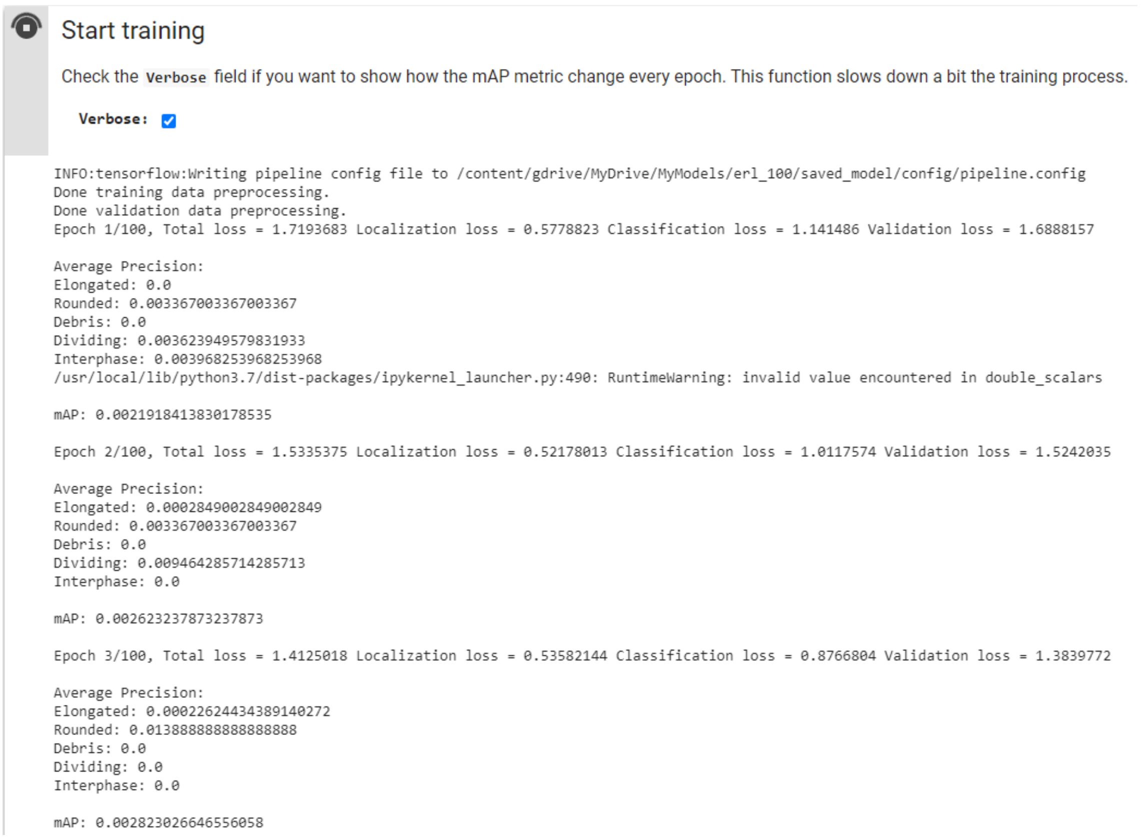
In Section 3.3 of the notebook, if the user wants to create a new model, he/she can choose which pre-trained backbone want to load to the new model from a list of backbones set by us. Once the backbone is chosen, the user can set the loss weights that the model will have for localisation and classification in the model's training phase (this functionality is previously explained in this document, in Section 3.3.4), by default these two values are set to 1.0. When the backbone and the values for the two losses are set the user can run the cell, which will display the backbone weights download and a text saying that everything went fine as it can be seen in Figure 3.19.

Another option for the user is to load a previous trained model that he/she saved in a previous use of the notebook, so he/she can continue training it with the same data or new dataset, improving the model's quality. To use this option the user just needs to check the *Use_pre-trained_model* checkbox and write the pre-trained model's path in the *pre-trained_model_folder* field.

3.4.6 Train the network

With all the scenario ready for the model usage, the user can start training the model.

The model training cell is an executable cell that the user cannot interact with much more than choosing if the validation's set mAP metric is going to be calculated and displayed (checking the *verbose* checkbox). If the mAP metric is calculated, the training process' duration is slowed down a bit. Once the user runs the training cell, the model starts training, showing the model losses (training total loss, training classification loss, training localisation loss and validation total loss) every epoch as well as the validation set's AP metric for each class and its mAP (Figure 3.20).


Start training

Check the `Verbose` field if you want to show how the mAP metric change every epoch. This function slows down a bit the training process.

Verbose: ✓

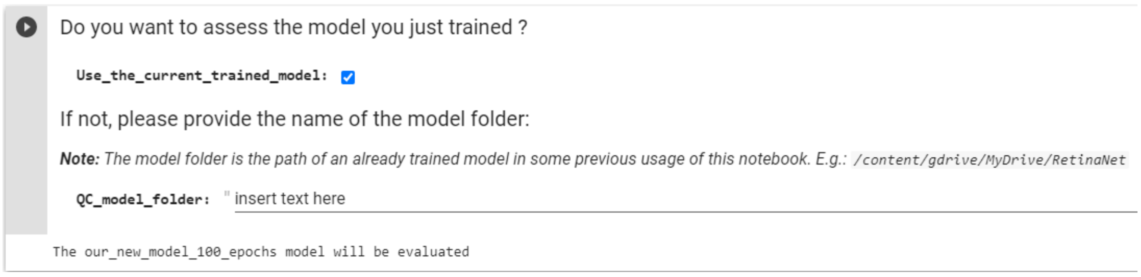
```
INFO:tensorflow:Writing pipeline config file to /content/gdrive/MyDrive/MyModels/erl_100/saved_model/config/pipeline.config
Done training data preprocessing.
Done validation data preprocessing.
Epoch 1/100, Total loss = 1.7193683 Localization loss = 0.5778823 Classification loss = 1.141486 Validation loss = 1.6888157

Average Precision:
Elongated: 0.0
Rounded: 0.003367003367003367
Debris: 0.0
Dividing: 0.003623949579831933
Interphase: 0.003968253968253968
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:490: RuntimeWarning: invalid value encountered in double_scalars

mAP: 0.0021918413830178535

Epoch 2/100, Total loss = 1.5335375 Localization loss = 0.52178013 Classification loss = 1.0117574 Validation loss = 1.5242035

Average Precision:
Elongated: 0.0002849002849002849
Rounded: 0.003367003367003367
Debris: 0.0
Dividing: 0.009464285714285713
Interphase: 0.0

mAP: 0.002623237873237873

Epoch 3/100, Total loss = 1.4125018 Localization loss = 0.53582144 Classification loss = 0.8766804 Validation loss = 1.3839772

Average Precision:
Elongated: 0.00022624434389140272
Rounded: 0.013888888888888888
Debris: 0.0
Dividing: 0.0
Interphase: 0.0

mAP: 0.002823026646556058
```

Figure 3.20: Output example of the training process.

Once the training process is finished, the model and its configuration is saved in the path specified by the user in the section 3.1 of the notebook as well as a PDF file with the model's training information. Finally, the user has a valid RetinaNet model that can be evaluated and used for object detection tasks.

3.4.7 Choosing model for evaluation

In order to evaluate a model, in notebook's Section 5 the user will find a cell like the one in Figure 3.21 where he/she has to specify what model is going to be evaluated, a previously saved one or one that has been trained in this notebook's run.


Do you want to assess the model you just trained ?

`Use_the_current_trained_model:` ☑

If not, please provide the name of the model folder:

Note: *The model folder is the path of an already trained model in some previous usage of this notebook. E.g.:* `/content/gdrive/MyDrive/RetinaNet`

`QC_model_folder:` " insert text here

```
The our_new_model_100_epochs model will be evaluated
```

Figure 3.21: Model for evaluation choosing cell.

If the user wants to use a model trained in this notebook's session, he/she has to check the *Use_the_current_trained_model* checkbox, otherwise this checkbox has to be unchecked and the previously saved model's path has to be specified in the *QC_model_folder* field. *Note: the QC abbreviations stands for 'Quality Control'*. After running the cell with an option set, a text with the chosen model's name will be displayed.

3.4.8 Inspection of the loss function

It is good practice to evaluate the training progress by comparing the training loss with the validation loss. The latter is a metric which shows how well the network performs on a subset of unseen data which is set aside from the training dataset.

During training, both values should decrease before reaching a minimal value which does not decrease further even if the model continues training. Comparing the development of the validation loss with the training loss can give insights into the model's performance, and here is why the validation set is a very important/honest key in the training process, because it is really usual to reach a point in training where the training loss value continues decreasing but the validation loss value has reached a minimum value and won't decrease anymore, this is, the model continues improving with the training data but not for unseen data. When this behaviour happens is called *overfitting*, which means that the model is not actually improving, it is only learning how the training data is so, learning that data, the model can improve the loss function. In order to have a good quality model we don't want the model to overfit, so the ideal scenario would be to stop the training when the validation loss stops decreasing.

In this cell it can also be seen the evolution of the validation's mAP metric over the epochs, which is also a honest metric that can be used to see if the model continued improving when the training process ended.

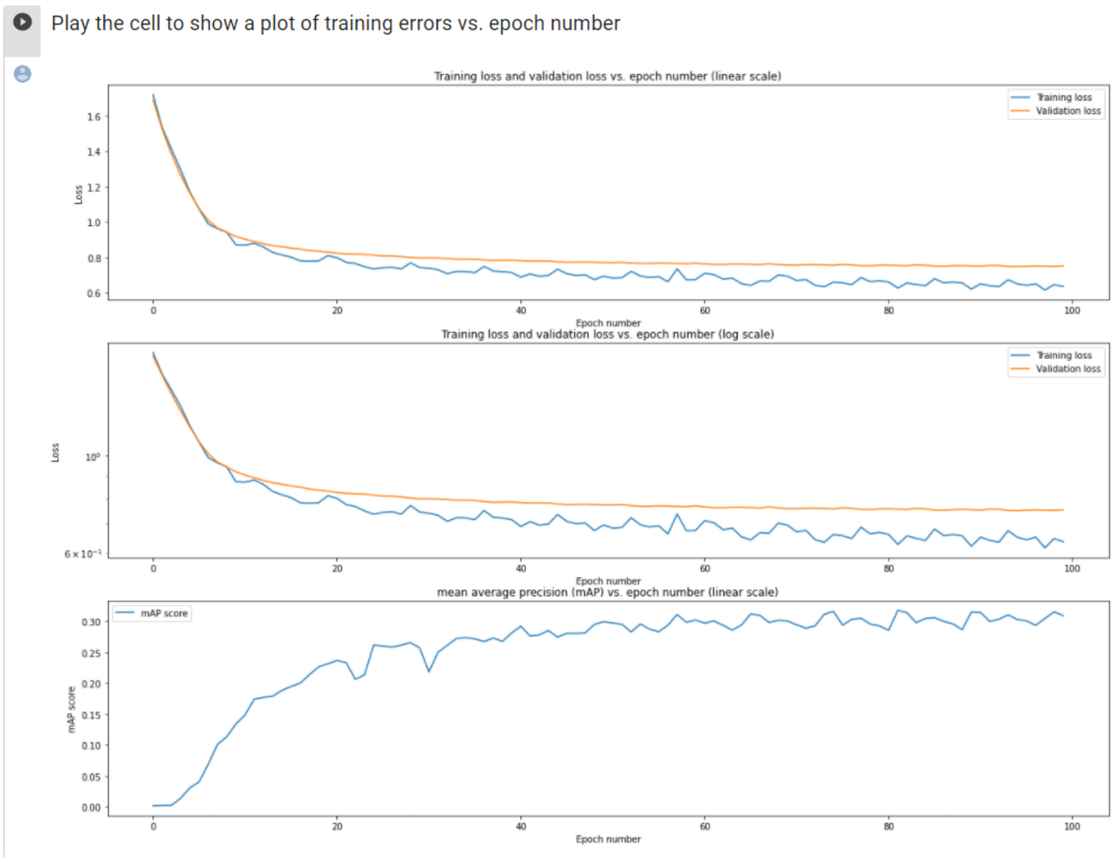
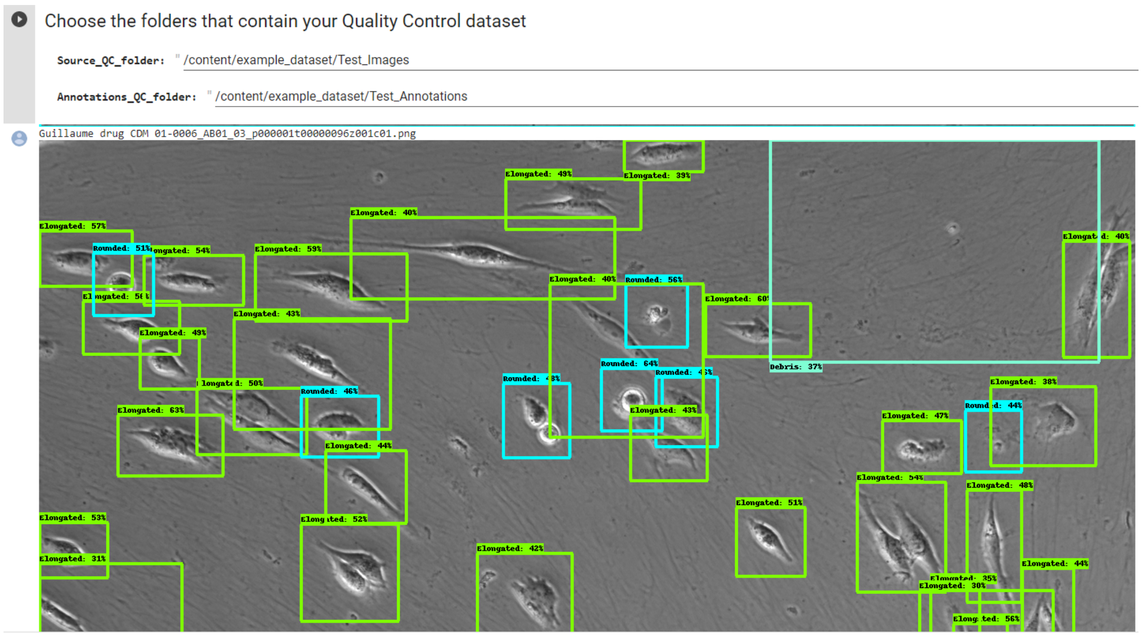

Play the cell to show a plot of training errors vs. epoch number

Figure 3.22: A display example of the cell from Section 5.1 of the notebook. In this example, in the first two plots, can be noted that both, the training loss (blue line) and the validation loss (orange line) still decrease in the last epoch (epoch number 100), so the model probably can improve if it continues training. It can also bee seen that the validation set's mAP metric continued improving slowly.

3.4.9 Quality metrics estimation

The Section 5.2 of the notebook is directed to check how a RetinaNet model can perform with new data both quantitatively and qualitatively.
The user has to set two paths: one where a set of images unseen by the model will be (a test set) and other one which the annotations of those images.

Qualitative evaluation is performed by the user after observing the displayed predictions to judge if they are good enough or not (see Figure 3.23).


Choose the folders that contain your Quality Control dataset

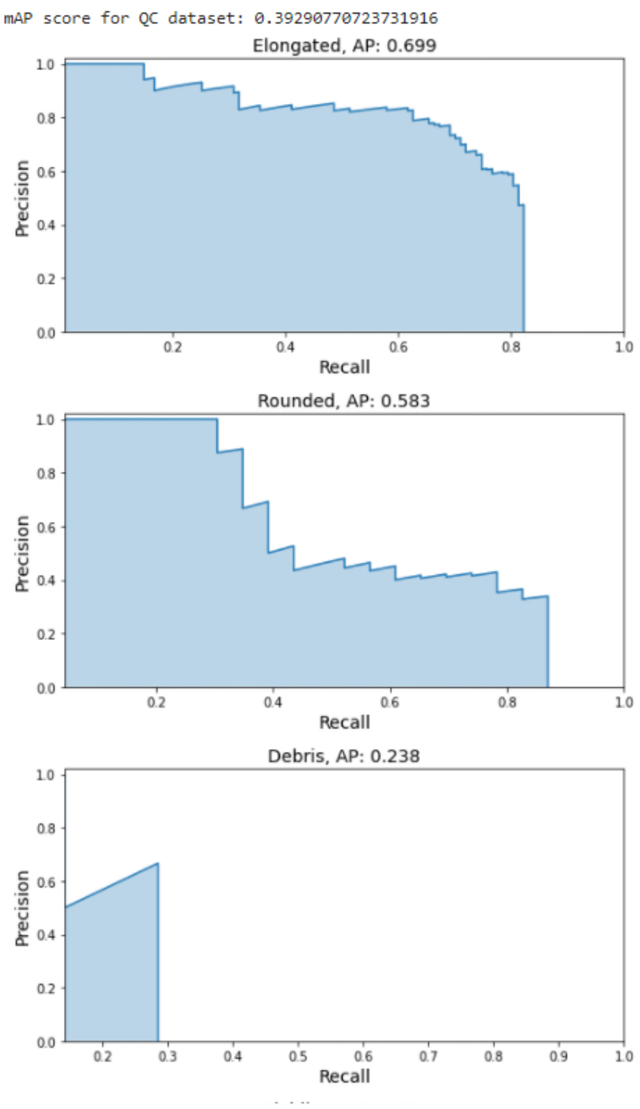
Source_QC_folder: " /content/example_dataset/Test_Images

Annotations_QC_folder: " /content/example_dataset/Test_Annotations

Guillaume drug CDM 01-0006_AB01_03_p000001t00000096z001c01.png

Figure 3.23: Qualitative evaluation: a prediction of an image from the test set established by the user.

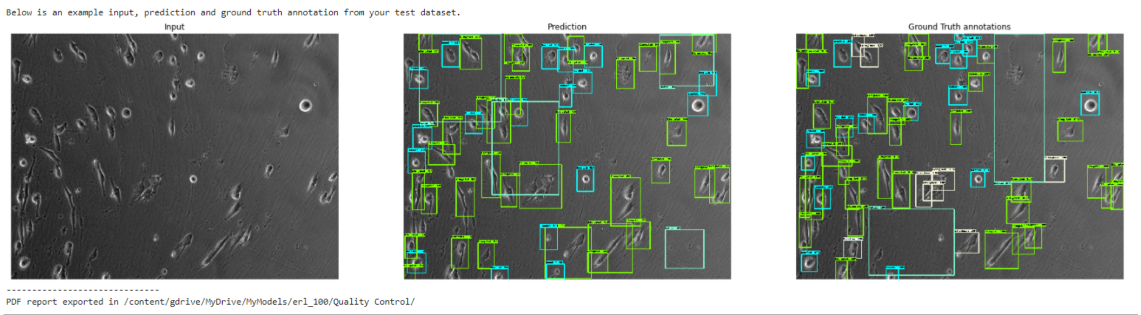
On the other hand, the quantitative evaluation is given by the mAP metric that will be calculated using the unseen annotated data set by the user, having numerical values for each class in the images and for the overall predictions (Figure 3.24).



```
mAP score for QC dataset: 0.39290770723731916
```

Figure 3.24: Quantitative evaluation: mAP score for the test set and the AP metric of each class.

Finally, a randomly selected original image from the test set, its prediction and its ground truth are displayed (Figure 3.25).



```
Below is an example input, prediction and ground truth annotation from your test dataset.
------------------------------
PDF report exported in /content/gdrive/MyDrive/MyModels/erl_100/Quality Control/
```

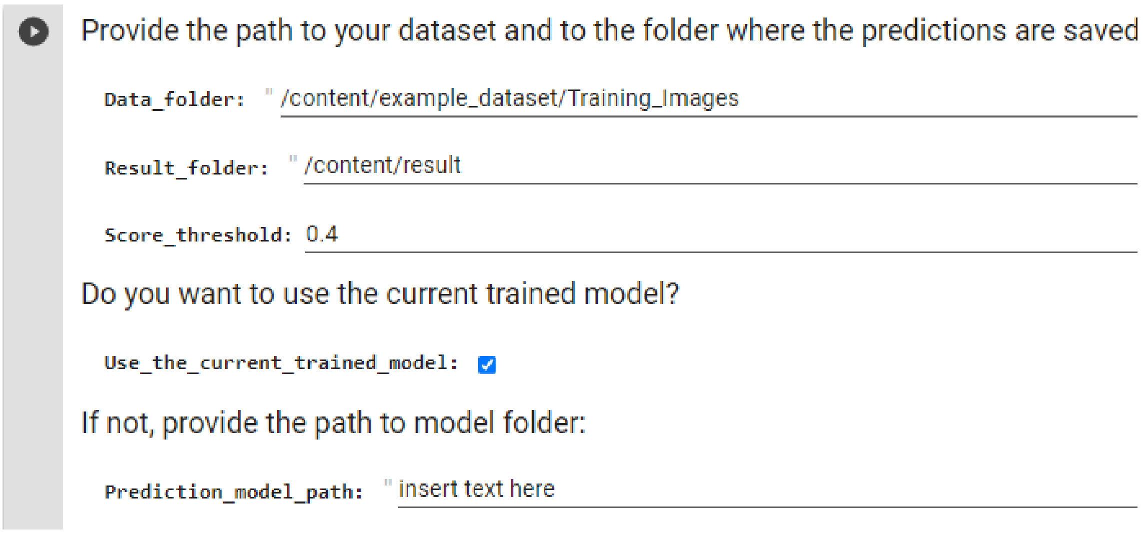
Figure 3.25: Display of three different views of a test image: the original image, its prediction and how it should have been predicted.

3.4.10 Use the model for actual predicting

This section allows the user to use the model to make predictions on new data.

To use this functionality it is the cell in Section 6.1 of the notebook where the user only needs to insert the following information:

- **Data_folder (path of the images)**: Path to the folder that contains the images to be predicted.
- **Result_folder**: Path where the predictions will be saved (if the path does not exist it will be automatically created).
- **Score_threshold**: A real value contained in the range [0, 1] which will limit the display of the predictions, this is, the minimum percentage of confidence of the model in the predictions. Set to 0.4 by default.
- **Use_the_current_trained_model**: Checkbox that set if the model that has been trained in the current notebook’s session (if is checked then it will be used the current trained model).
- **Prediction_model_path**: If the previous checkbox is not checked then the model that will be used for prediction is the one from the path inserted in this field.



Provide the path to your dataset and to the folder where the predictions are saved

`Data_folder:` " /content/example_dataset/Training_Images

`Result_folder:` " /content/result

`Score_threshold:` 0.4

Do you want to use the current trained model?

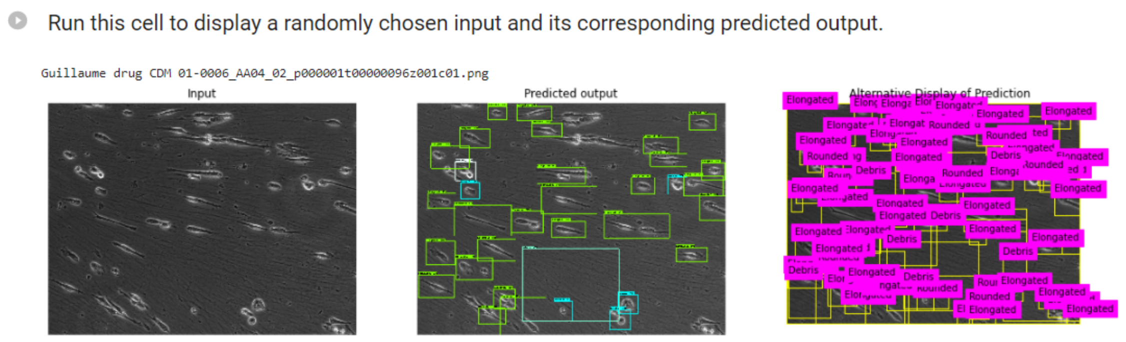
`Use_the_current_trained_model:` ✓

If not, provide the path to model folder:

`Prediction_model_path:` " insert text here

Figure 3.26: Cell from Section 6.1 of the notebook where the predictions will be made and saved.

The following cell and also the last executable cell that the notebook has (Section 6.2 of the notebook) is a cell which shows randomly one of the predictions of the previous cell with its corresponding original image and with a different display format (Figure 3.27).



Run this cell to display a randomly chosen input and its corresponding predicted output.

```
Guillaume drug CDM 01-0006_AA04_02_p000001t00000096z001c01.png
```

Figure 3.27: Display example of the last cell in the notebook.

CHAPTER 4

Experimentation

4.1 Familiarisation experiments

In the beginning of this project, in order to learn how RetinaNet works and how it could be implemented with the TF2 object detection API, a previous Jupyter notebook was done building the RetinaNet model from scratch to, later in that notebook, keep adding functionalities until, in the end, the notebook could be a complete and useful object detection model able to be quickly fine-tuned.

Note: this first notebook can be seen in the following link: `https://github.com/ErlantzCalvo/Object-detection-Tensorflow-2`

4.1.1 Pre-trained RetinaNet on the COCO dataset

To get familiarised with both RetinaNet and the TF2 object detection API, we created a first notebook that allows to predict objects from the COCO dataset.

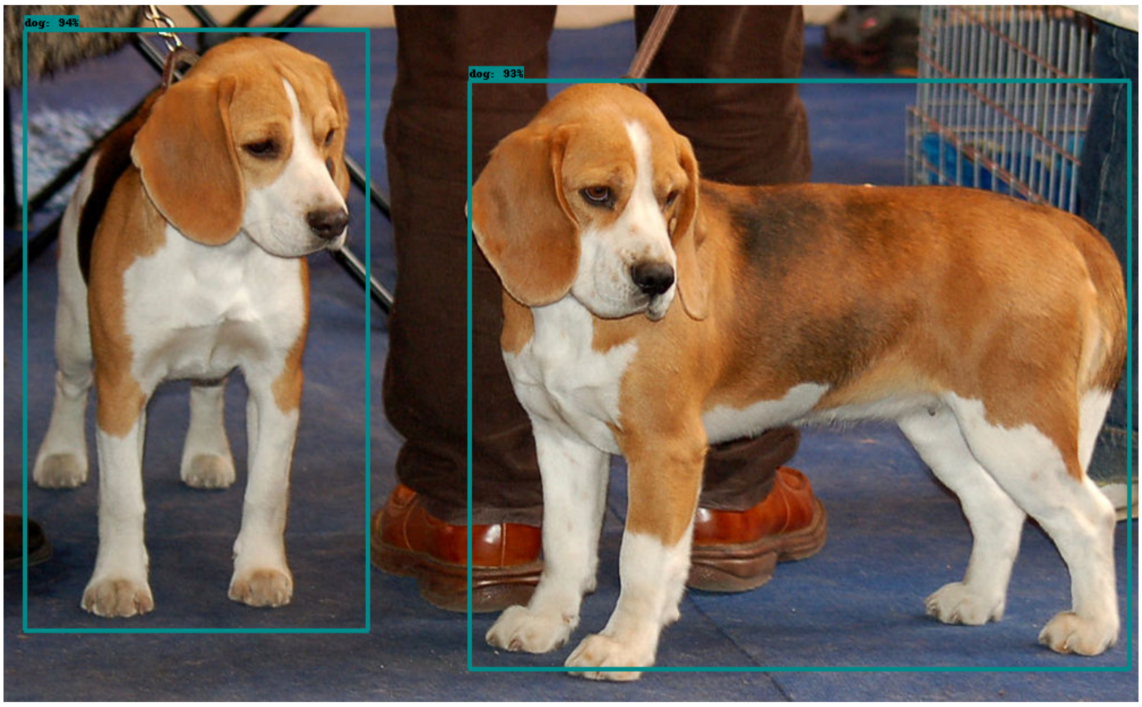


Figure 4.1: Prediction of two dogs in a picture by the model's prototype.

4.1.2 Fine tuning for rubber ducky prediction

The next step was to adapt the notebook to make fine-tuning and predict a new class.

Addressing this task required one of two, either get an already annotated dataset or take a set of images which contain one type of object and label it. Fortunately the TF2 object detection API contains a tool that makes easy the image annotation in the Jupyter notebook execution so we decided to download a set of rubber ducky images (5 images) and label them manually (Figure 4.2).

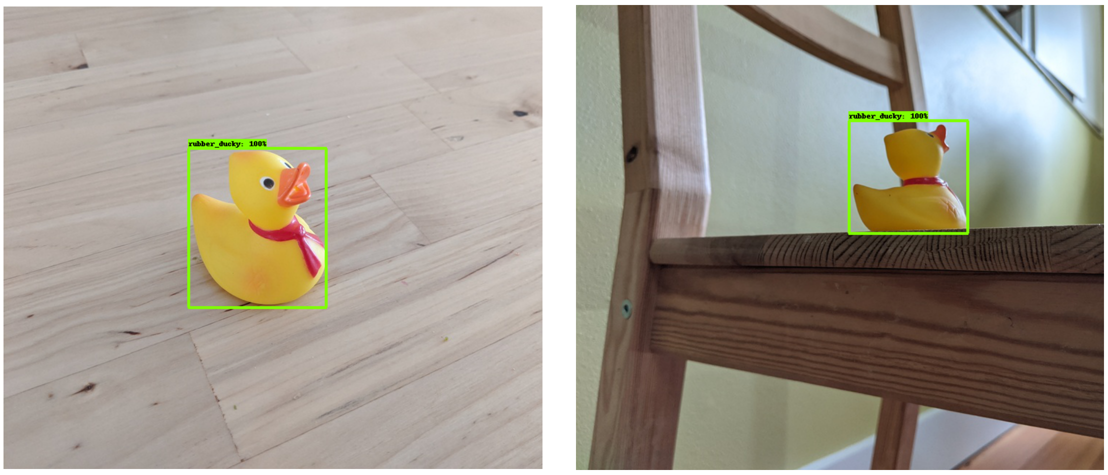


Figure 4.2: Manually annotated rubber ducks using the annotation tool from TF2 API.

After finishing the annotation part the pre-trained model's weights are downloaded and the model is configured for predicting only one class, the rubber duckies. Then the model is fine-tuned making a custom training process which works in the same way a the Zero-CostDL4Mic RetinaNet project's training process (Section 3.3.5 of this document).

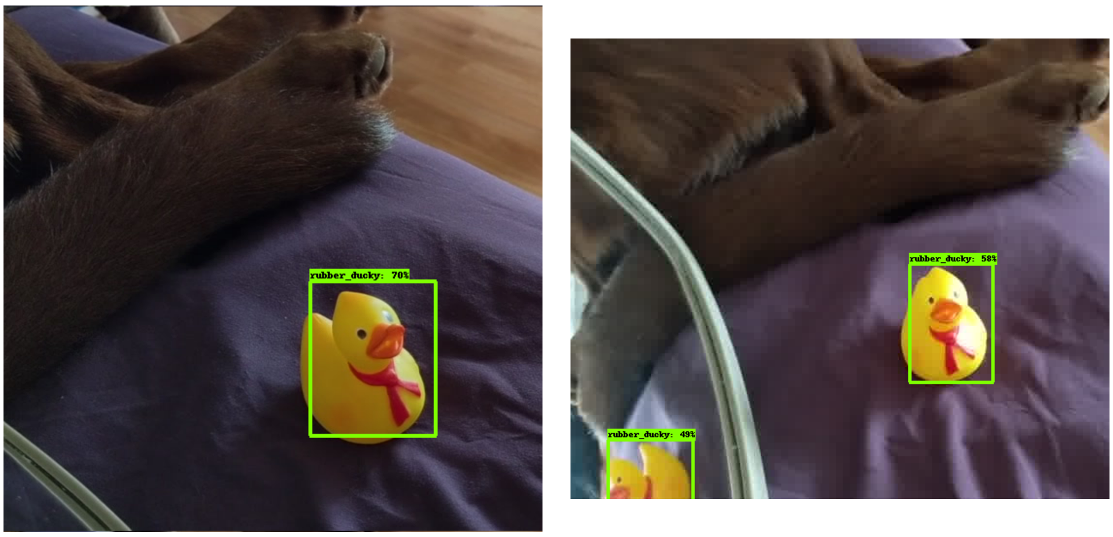


Figure 4.3: Inference results using the model fine-tuned with rubber duckies.

Note: in this notebook we tried making the fine-tune without loading the weights from a pre-trained model and the results and accuracy of the model's predictions were consistently worse than with the weights loaded.

4.1.3 Extension to multi-class prediction

Once we had a model that could easily be fine-tuned, a big part of the object detection task was achieved but there was still an important functionality of this project missing, the multi-class prediction. In the duckies notebook, the RetinaNet model only predicted one single class, the rubber duck class. Taking into account what the ZeroCostDL4Mic project is meant to be we can't have a model which only can predict objects of one class, so we had to figure that point out.

The following step was to get the model to predict several classes at the same time. This functionality was achieved modifying the model's configuration indicating the number of classes that the model has to predict and building some pre-processing methods for building up any kind of dataset passed. In order to test this new multi-class functionality we added two new classes (Platypus and Minion[1]), making the fine-tuning and inference for a test set.

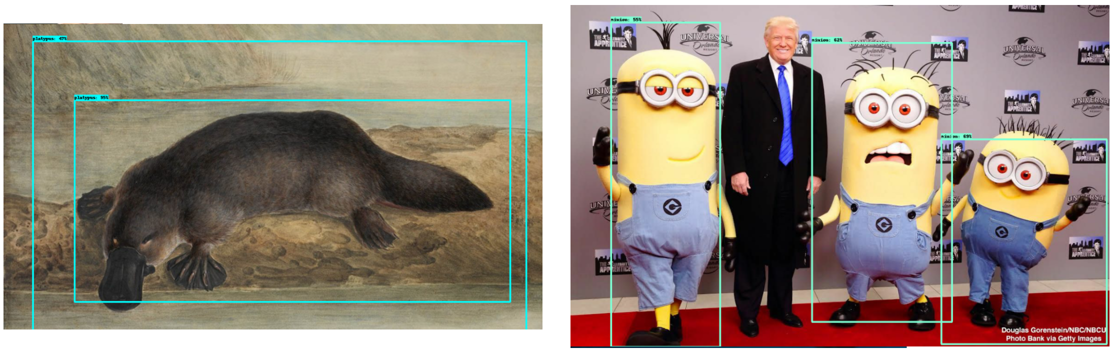


Figure 4.4: Inference results using the model fine-tuned with rubber duckies, minions and platypuses.

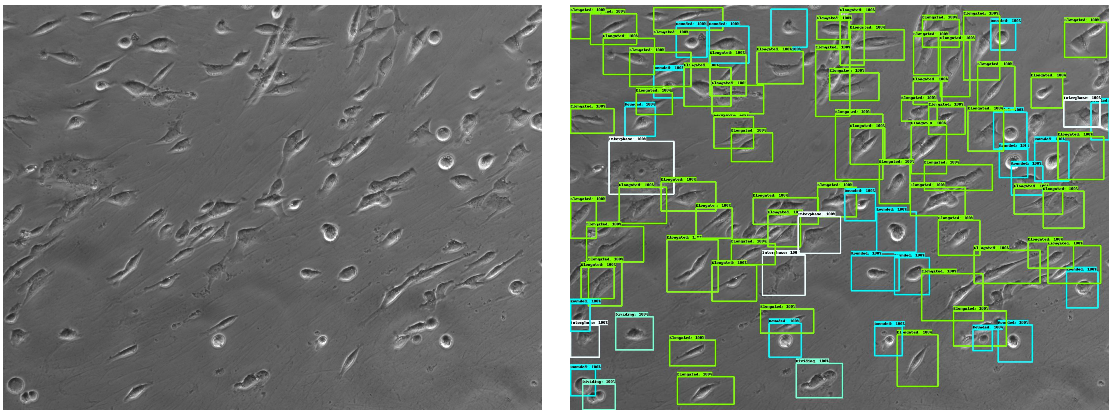
In this toy notebook we didn't measure the accuracy of the model with any metric, it was all subjectively valued.

4.2 Dataset used

Our final ZeroCostSL4Mic notebook has been developed and tested with a dataset of 33 images containing MDA-MB-231 cells migrating on cell-derived matrices generated by fibroblasts [32]. Every image has its corresponding XML PASCAL VOC format annotation file attached, where all the cell's bounding boxes location coordinates and class

[1]Minions are fictional yellow creatures that appear in the Despicable Me franchise.

are manually labelled. All the images are 2D grey scale images in *png* format and the dimensions of every image is 1388×1040 $(Width \times Height)$.



(a) (b)

Figure 4.5: Dataset's image example. From left to right: (a) original image, and (b) expert annotations on top of original image.

The given dataset contains 5 different types of cell: Elongated, Rounded, Dividing, Debris and Interphase. The figure 4.5b shows all the different types of cell, being each cell's bounding box represented by a different colour depending on it's class, this is, the green coloured boxes represent the *Elongated* cells, the blue coloured boxes represent the *Rounded* cells, the aquamarine coloured boxes represent the *Interphase* cells, the white coloured boxes represent a *Debris* zone and the beige coloured boxes represent the *Dividing* cells.

4.3 Experimental results of the RetinaNet model in the MDA-MB-231 cell dataset

Different experiments have been performed to test our implemented solution in the aforementioned MDA-MB-231 cell dataset. In order to compare it with other available solutions, we repeated the experiments with the YOLOv2 implementation available as a ZeroCostDL4Mic notebook (FOOTNOTE).

For these tests, we have decided to use the ResNet50 as backbone of our RetinaNet model instead of the two available MobileNet models, which usually perform faster but not as accurate.

	Epochs	**Batch size**	**Learning rate**	**Training time (mins)**	**mAP**
YOLOv2	50	16	0.0001	25	0.22
YOLOv2	50	4	0.0001	34	0.20
YOLOv2	100	8	0.0001	46	0.22
RetinaNet	100	16	0.0001	**9**	**0.39**
RetinaNet	100	4	0.0001	14	**0.40**
RetinaNet	200	8	0.0001	28	0.38

Table 4.1: Comparison between RetinaNet and YOLOv2 performance on MDA-MB-231 cell dataset. mAP results correspond to the test set values. In bold, best time and evaluation metric.

It is important to know that, due to the hardware limitations of the Colab environment, the maximum batch size that can be set in our notebook is ≈ 16. Otherwise, setting the batch size over 16 can cause the model to exceed memory capacity, causing the session close and losing the notebook's executed state until the moment of that memory exceed, which probably comes in the training phase. The election of the ResNet50 backbone is also related to this limitation because, if there was no memory limitation, choosing the ResNet101 or ResNet150 models would result in better accuracy scores, but they are very heavy in terms of hardware needs.

In Table 4.1, some of the tests with both models (RetinaNet and YOLOv2) can be seen. Each row of the table represents a different configuration for the model given in the first cell of the row, either a change on the batch size or in the number of epochs. Note that YOLOv2 needs less epochs to converge its loss function but it also requires a notably greater amount of time compared with the RetinaNet model.

We can also observe how, the lower is the batch size, the longer the training time. This happens because the model tune its weights after it has seen all the data from a batch and calculated the loss value relative to that batch. This means that the model tunes more times its weights when the batch size is lower, what requires more operations to finish an epoch.

In a general scope, the table shows that RetinaNet outperforms YOLOv2 in both measures, speed and precision (mAP). This means that the model has achieved the ambitious goal we set for ourselves at the beginning of the project, building an object detection with Deep Learning approach that could improve the existing ones.

CHAPTER 5

Conclusions

First of all, throughout this project we have gone around the state-of-the-art object detection with Deep Learning techniques in order to understand how the most popular approaches of this field work and try to learn the most suitable approach to meet the requirements of our initial objective. We have delved into the one-stage and two-stage methods to understand their strengths and weaknesses and we have come to the conclusion that, although the two-stage approaches have always performed better than the one-stage ones in terms of accuracy, their overall speed is quite worse and that feature does not fit in our needs. Later, we have seen the RetinaNet model and how it improved the object detection field, outperforming the accuracy of the most popular two-stage approaches while it preserves the one-stage models speed. It seemed that it was the ideal approach for our project.

Then we studied what tools to use to implement the RetinaNet model in our project and learnt about the TF2 object detection API and all the possibilities that it gives us like, in our particular case, the strong feature of having a great model Zoo.

After we studied the TF2 object detection API, we had to learn about the ZeroCostDL4Mic project's workflow and how we could mould a RetinaNet model build and usage to work within said project.

Next, we faced one of the most difficult parts of the whole project and its core, the RetinaNet implementation in the ZeroCostDL4Mic project. We went through many technical difficulties caused by our inexperience using the TF2 object detection API and many other external code libraries and, also, moulding existing ZeroCostDL4Mic notebooks to work with our implementation. Finally and after putting a lot of effort into it, we managed to build a ZeroCostDL4Mic notebook that makes easy the task of building a RetinaNet model for every kind of user without the need for prior knowledge of Deep Learning.

Once we had the final notebook, we tested it using an actual annotated dataset which is publicly available. In order to make easier this task, we implemented the functionality of downloading a public annotated dataset in the notebook for all the users that want to try the notebook without the requirement of having an annotated dataset. We trained a model and evaluated it using the said public dataset and checked that the predictions seemed not only viable, but also of quality.

Finally, we have compared the results of our RetinaNet model with the results of an already existing ZeroCostDL4Mic object detection model, the YOLOv2 project. This comparison was done using the public dataset that can be downloaded in the RetinaNet notebook. The results not only represent an improvement of our project compared to the YOLOv2 one, but also the performance of the RetinaNet model is almost twice as good as the YOLOv2 one, which is a great improvement. With all these data in mind, it can be said that we have successfully met all the objectives proposed for the project.

As future work we will make a performance comparison of our model with more architectures than YOLOv2. Also, we will add more backbone networks to choose when building the RetinaNet model in the notebook. Then, we will study in a deeper way the reasoning of the model's hyperparameters to get a good knowledge of what values are the ideal for each model.

Bibliography

[1] L. Liu, W. Ouyang, X. Wang, P. W. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, “Deep learning for generic object detection: A survey,” *CoRR*, vol. abs/1809.02165, 2018.

[2] T. N. Sainath, A.-r. Mohamed, B. Kingsbury, and B. Ramabhadran, “Deep convolutional neural networks for lvcsr,” in *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 8614–8618, IEEE, 2013.

[3] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal loss for dense object detection,” 2018.

[4] L. v. Chamier, R. F. Laine, J. Jukkala, C. Spahn, D. Krentzel, E. Nehme, M. Lerche, S. Hernández-Pérez, P. K. Mattila, E. Karinou, S. Holden, A. C. Solak, A. Krull, T.-O. Buchholz, M. L. Jones, L. A. Royer, C. Leterrier, Y. Shechtman, F. Jug, M. Heilemann, G. Jacquemet, and R. Henriques, “Zerocostdl4mic: an open platform to use deep-learning in microscopy,” *bioRxiv*, 2020.

[5] N. Zhai and X. Zhou, “Temperature prediction of heating furnace based on deep transfer learning,” *Sensors*, vol. 20, no. 17, 2020.

[6] M. A. Fischler and R. A. Elschlager, “The representation and matching of pictorial structures,” *IEEE Transactions on computers*, vol. 100, no. 1, pp. 67–92, 1973.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Commun. ACM*, vol. 60, p. 84–90, May 2017.

[8] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” *CoRR*, vol. abs/1311.2524, 2013.

[9] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: towards real-time object detection with region proposal networks,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 6, pp. 1137–1149, 2016.

[10] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, “Feature pyramid networks for object detection,” *CoRR*, vol. abs/1612.03144, 2016.

[11] J. Redmon and A. Farhadi, “YOLO9000: better, faster, stronger,” *CoRR*, vol. abs/1612.08242, 2016.

[12] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, “Overfeat: Integrated recognition, localization and detection using convolutional networks,” *arXiv preprint arXiv:1312.6229*, 2013.

[13] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” *CoRR*, vol. abs/1505.04597, 2015.

[14] U. Schmidt, M. Weigert, C. Broaddus, and G. Myers, “Cell detection with star-convex polygons,” *CoRR*, vol. abs/1806.03535, 2018.

[15] M. Weigert, U. Schmidt, T. Boothe, A. Müller, A. Dibrov, A. Jain, B. Wilhelm, D. Schmidt, C. Broaddus, S. Culley, *et al.*, “Content-aware image restoration: pushing the limits of fluorescence microscopy,” *Nature methods*, vol. 15, no. 12, pp. 1090–1097, 2018.

[16] A. Krull, T. Buchholz, and F. Jug, “Noise2void - learning denoising from single noisy images,” *CoRR*, vol. abs/1811.10980, 2018.

[17] E. Nehme, L. E. Weiss, T. Michaeli, and Y. Shechtman, “Deep-storm: super-resolution single-molecule microscopy by deep learning,” *Optica*, vol. 5, no. 4, pp. 458–464, 2018.

[18] C. Ounkomol, S. Seshamani, M. M. Maleckar, F. Collman, and G. R. Johnson, “Label-free prediction of three-dimensional fluorescence images from transmitted-light microscopy,” *Nature methods*, vol. 15, no. 11, pp. 917–920, 2018.

[19] P. Isola, J. Zhu, T. Zhou, and A. A. Efros, “Image-to-image translation with conditional adversarial networks,” *CoRR*, vol. abs/1611.07004, 2016.

[20] J. Zhu, T. Park, P. Isola, and A. A. Efros, “Unpaired image-to-image translation using cycle-consistent adversarial networks,” *CoRR*, vol. abs/1703.10593, 2017.

[21] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial networks,” 2014.

[22] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, “Tensorflow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” *CoRR*, vol. abs/1912.01703, 2019.

[24] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft COCO: common objects in context,” *CoRR*, vol. abs/1405.0312, 2014.

[25] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? the kitti vision benchmark suite,” in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3354–3361, 2012.

[26] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari, “The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale,” *IJCV*, 2020.

[27] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *CoRR*, vol. abs/1512.03385, 2015.

[28] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *CoRR*, vol. abs/1704.04861, 2017.

[29] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” *CoRR*, vol. abs/1502.03167, 2015.

[30] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[31] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT’2010*, pp. 177–186, Springer, 2010.

[32] G. Jacquemet and L. von Chamier, “ZeroCostDL4Mic - YoloV2 example training and test dataset,” July 2020.