Università di Bologna
Alma Mater Studiorum
Advanced Automotive Electronic Engineering

Deep Learning Project

Food-recognition

Michele Abbruzzese – michele.abbruzzese@studio.unibo.it
Simone Cattin – simone.cattin@studio,unibo.it
Matteo Totaro – matteo.totaro6@studio.unibo.it

Professor: **Andrea Asperti**

September 24, 2021

Abstract

Recognizing food from images is an extremely useful tool for a variety of use cases. In particular, It would allow people to track their food intake by simply taking a picture of what they consume. Food tracking can be of personal interest and can often be of medical relevance as well.

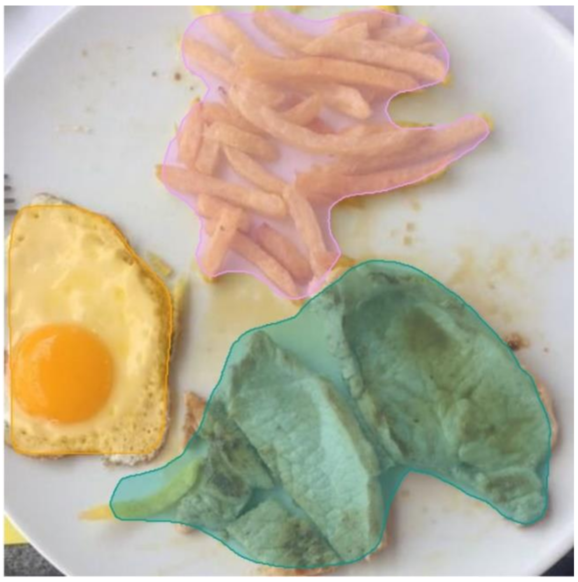
There are a lot of medical studies interested in food recognition because it allows to have important parameters into the diets of people.

Image-based food recognition has in the past few years made substantial progress thanks to advances in deep learning. But food recognition remains a difficult problem for a variety of reasons.

Contents

Problem Statement

The goal of this work is to train models which can look at images of food items and detect the individual food items present in them. We use a novel dataset of food images collected through the MyFoodRepo app where numerous volunteer Swiss users provide images of their daily food intake in the context of a digital cohort called "Food & You".



This growing data set has been annotated - or automatic annotations have been verified - with respect to segmentation, classification (mapping the individual food items onto an ontology of Swiss Food items), and weight/volume estimation.

Figure 1.1 Example of detection

Datasets

Finding annotated food images is difficult. There are some databases with some annotations, but they tend to be limited in important ways. Most food images on the internet are a lie.

Search for any dish, and you will find beautiful stock photography of that particular dish. Same on social media: we share photos of dishes with our friends when the image is exceptionally beautiful. But algorithms need to work on real-world images. In addition, annotations are generally missing - ideally, food images would be annotated with proper segmentation, classification, and volume/weight estimates.

Another big issue is linked to the dimension of a single image: big datasets need to have compressed images to gain some speed during evaluation, particularly during training.

The dataset for the AIcrowd Food Recognition Challenge provides thousands of training images, between 20 and 70 KB of storage each.

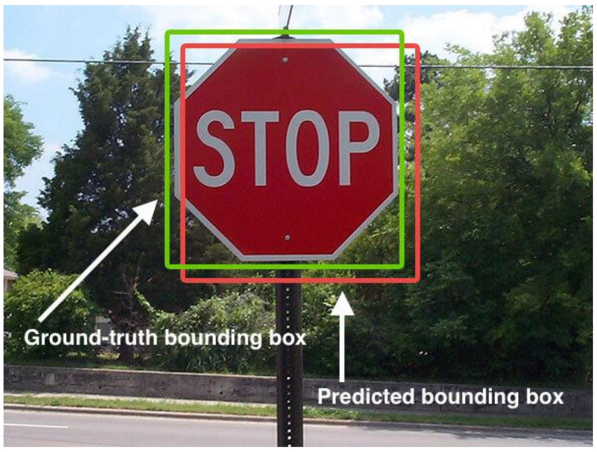
The dataset used contains:

- **Train-v0.4.tar.gz:** This is the Training Set of 24120 (as RGB images) food images, along with their corresponding 39328 annotations in MS-COCO format.
- **Val-v0.4.tar.gz:** This is the suggested Validation Set of 1269 (as RGB images) food images, along with their corresponding 2053 annotations in MS-COCO format.
- **Test-images-v0.4.tar.gz:** This is the debug Test Set for Round-4, where you are provided the same images as the validation set.

The dataset presents 273 categories. Through the Python code, we have applied a filter which was able to give us just the 16 most annotated categories. Since the bounding boxes and the masks of the model extrapolate the objects from the images, the data left on each picture will then be its background. For this reason, the training was performed on 16 categories + 1 BG, the background of the images.

Since we faced some problems during the unzipping of the datasets performed by Colab, same problem that other groups had, we uploaded the full uncompressed dataset on Google Drive.

Evaluation Criteria



In order to evaluate our trained model, we are going to use the Intersection over Union evaluation: for a known ground truth mask A, we propose a mask B, then we compute the ratio between the overlap over the union, IoU.

Intersection over Union is an evaluation metric used to measure the accuracy of an object detector on a particular dataset, it is best suited for our case.

IoU measures the overall overlap between the *true region* and the *proposed region*. Then we consider it a True detection, when there is at least half an overlap, or when **IoU > 0.5**

Then we can define the following parameters:

- Precision (IoU > 0.5)
- Recall (IoU > 0.5)

The final scoring parameters AP{IoU > 0.5} and AR{IoU > 0.5} are computed by averaging over all the precision and recall values for all known annotations in the ground truth.

Mask R-CNN for Object Detection and Segmentation

This project makes us of *Mask R-CNN* on Python 3, *Keras*, and *TensorFlow*. TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

Keras is an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library. Mask R-CNN is a deep learning model for computer vision developed by the Facebook AI group that achieves state-of-the-art results on semantic segmentation (object recognition and pixel labelling).

An implementation of the model is made available by Matterport on. The code in their repo works with MS Coco (a benchmark dataset for semantic segmentation) out of the box but provides for easy extensibility to any kind of dataset or image segmentation task.

Mask R-CNN (regional convolutional neural network) is a two-stage framework: the **first** stage scans the image and generates *proposals* (areas likely to contain an object). And the **second** stage classifies the proposals and generates bounding boxes and masks. Mask R-CNN is an extension of the popular Faster R-CNN object detection architecture. Mask R-CNN adds a branch to the already existing Faster R-CNN outputs. The Faster R-CNN method generates two things for each object in the image:

- Its class
- The bounding box coordinates

Mask R-CNN adds a third branch to these which outputs the object mask as well.

At a high level, Mask R-CNN consists of these modules:

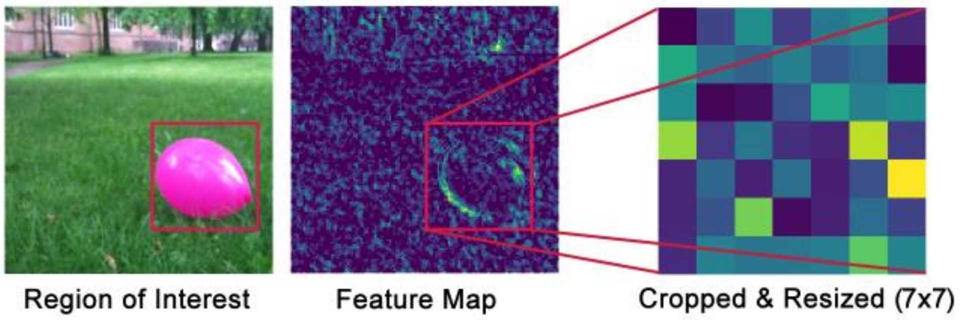
- **Backbone:** This is a standard convolutional neural network (typically, ResNet50 or ResNet101) that serves as a feature extractor. The early layers detect low level features (edges and corners), and later layers successively detect higher level features (car, person, sky). Passing through the backbone network, the image is converted from 1024x1024x3 (RGB) to a feature map of shape 32x32x2048. This feature map becomes the input for the following stages. This backbone can be improved using the Feature Pyramid Network (FPN) described below.

- **Region Proposal Network (RPN):** The RPN is a lightweight neural network that scans the image in a sliding-window fashion and finds areas that contain objects. The regions that the RPN scans over are called ***anchors***. Which are boxes distributed over the image area.

- **RoI Classifier & Bounding Box Regressor:** This stage runs on the *regions of interest* (ROIs) proposed by the RPN. And just like the RPN, it generates two outputs for each ROI:

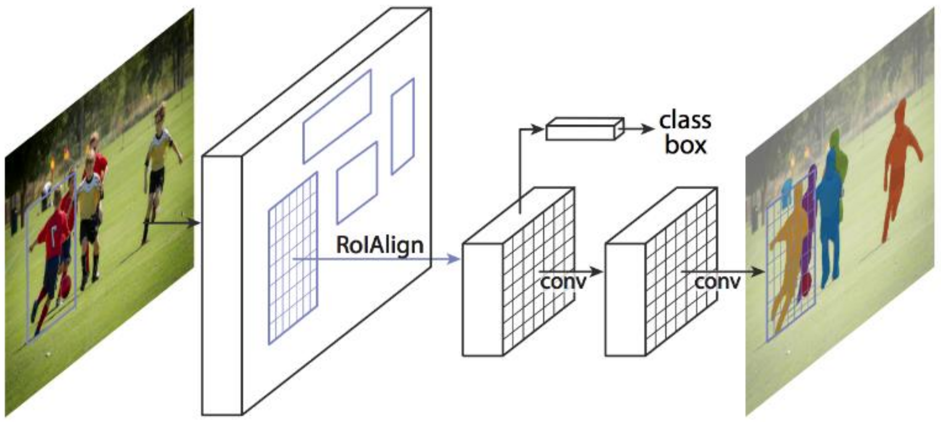
    - **Class:** The class of the object in the RoI. Unlike the RPN, which has two classes (FG/BG), this network is deeper and has the capacity to classify regions to specific classes (person, car, chair, …etc.). It can also generate a *background* class, which causes the RoI to be discarded.

    - **Bounding Box Refinement:** Very similar to how it's done in the RPN, and its purpose is to further refine the location and size of the bounding box to encapsulate the object.



We could understand better the Mask R-CNN through a generic example:

1. We take an image as input and pass it to the ConvNet, which returns the feature map for that image
2. Region proposal network (RPN) is applied on these feature maps. This returns the object proposals along with their objectness score
3. A RoI pooling layer is applied on these proposals to bring down all the proposals to the same size
4. Finally, the proposals are passed to a fully connected layer to classify and output the bounding boxes for objects. It also returns the mask for each proposal

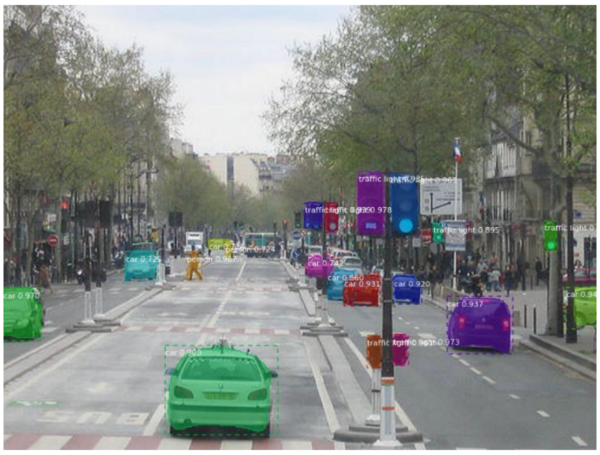


At the end we are able to say that the image segmentation made with the Mask R-CNN algorithm gives us three outputs for each object in the image: its class, bounding box coordinates and object mask.

It is simple, flexible and general approach and it is also the current state-of-art for image segmentation but requires high training time.

Visualizing the results

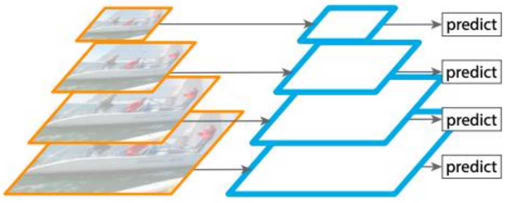
This fork of the matterport/mask_rcnn repo was set up to integrate with Weights and Biases (wandb).



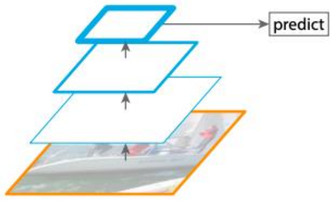
Wandb is a cloud interface for tracking model parameters and performance, allowing machine learning teams to coordinate work in a way similar to GitHub. ([Their github page](#).) The code in their repo works with MS Coco (a benchmark dataset for semantic segmentation) out of the box but provides for easy extensibility to any kind of dataset or image segmentation task.

The model generates bounding boxes and segmentation masks for each instance of an object in the image. It's based on Feature Pyramid Network (FPN) and a ResNet50 backbone.

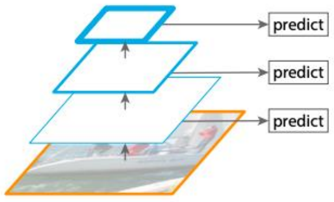
Detecting objects at different scales is challenging, even more so for small objects. We can use a pyramid of the same image at different scale to detect objects, as shown below. However, that is a compute costly process and hence is not used often. Alternatively, we create a pyramid of feature and use them for object detection.



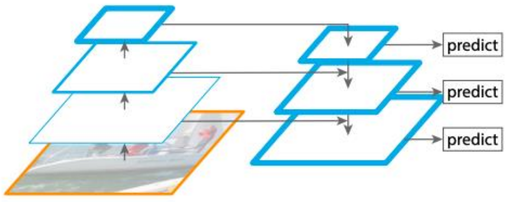
(a) Featurized image pyramid



(b) Single feature map



(c) Pyramidal feature hierarchy



(d) Feature Pyramid Network

For a Feature Pyramid Network (FPN) network, the anchors must be ordered in a way that makes it easy to match anchors to the output of the convolution layers that predict anchor scores and shifts.

The order of anchors is important. The same order has to be used in training and prediction phases. This must also match the order of the convolution execution. This is internally taken care of by the MaskRCNN library.

- Sort by pyramid level first. All anchors of the first level, then all of the second and so on. This makes it easier to separate anchors by level.
- Within each level, sort anchors by feature map processing sequence. Typically, a convolution layer processes a feature map starting from top-left and moving right row by row.
- For each feature map cell, pick any sorting order for the anchors of different ratios. Here we match the order of ratios passed to the function.

Anchor Stride: In the FPN architecture, feature maps at the first few layers are high resolution. For example, if the input image is 1024x1024 then the feature map of the first layer is 256x256, which generates about 200K anchors (256x256x3). These anchors are 32x32 pixels and their stride relative to image pixels is 4 pixels, so there is a lot of overlap. We can reduce the load significantly if we generate anchors for every other cell in the feature map. A stride of 2 will cut the number of anchors by 4, for example.

In this implementation MaskRCNN uses an anchor stride of 2, which is different from the paper.

Image Segmentation

Image segmentation is the process of partitioning a digital image into multiple segments (sets of pixels, also known as image objects). The goal of segmentation is to simplify and/or change the representation of an image into something that is more meaningful and easier to analyse. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images. More precisely, image segmentation is the process of assigning a label to every pixel in an image such that pixels with the same label share certain characteristics.

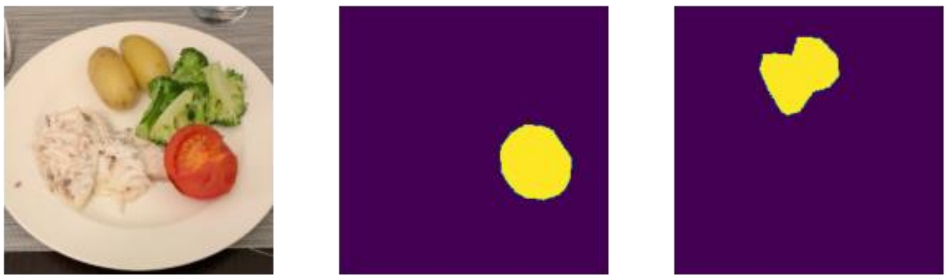
Semantic Segmentation detects all the objects present in an image at the pixel level. Outputs regions with different classes or objects.

It groups pixels in a semantically meaningful way. Pixels belonging to a person, road, building, fence, bicycle, cars or trees are grouped separately. Instance Segmentation is identifying each object instance for every known object within an image.

Instance segmentation, on the other hand, assigns a label to each pixel of the image. It is used for tasks such as counting the number of objects.

In our project, we are implementing an instance segmentation with object detection focused on foods.

Mini Masks



Instance binary masks can get large when training with high resolution images. For example, if training with 1024x1024 image then the mask of a single instance requires 1MB of memory (Numpy uses bytes for boolean values). If an image has 100 instances, then that's 100MB for the masks alone. To improve training speed, masks are optimized as:

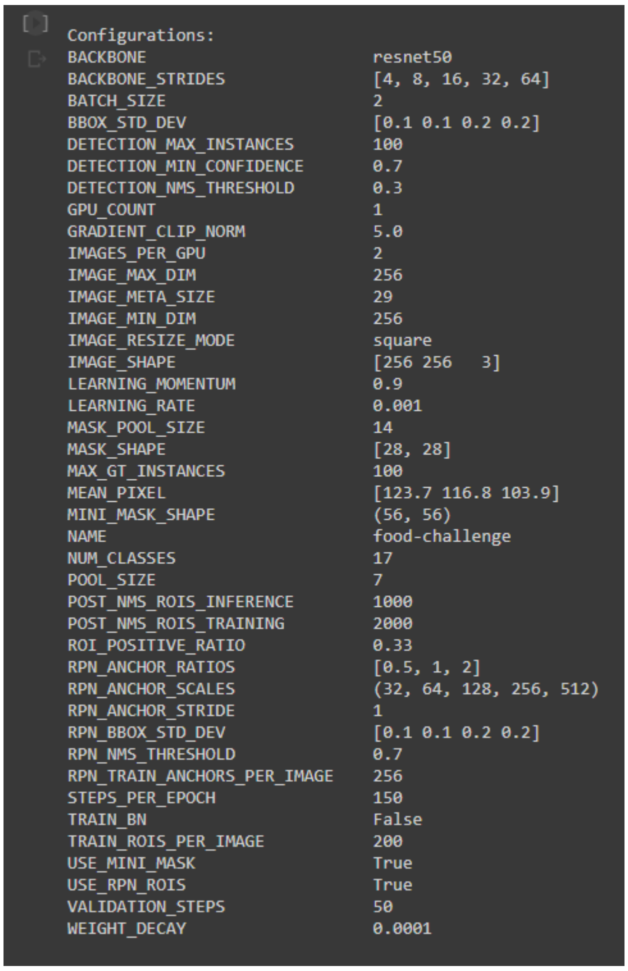
- Mask pixels that are inside the object bounding box are stored, rather than a mask of the full image. Most objects are small compared to the image size, so space is saved by not storing a lot of zeros around the object.
- The mask is resized to a smaller size (e.g. 56x56). For objects that are larger than the selected size we lose a bit of accuracy. However, this loss is negligible for most practical purposes. This size of the mini_mask can be set in the config class.

Data Augmentations

Image augmentation is a strategy that allows us to significantly increase the diversity of images available for training models, without collecting new images. Augmentations make the model more robust.

Results:

Training configuration:



```
Configurations:
BACKBONE                       resnet50
BACKBONE_STRIDES               [4, 8, 16, 32, 64]
BATCH_SIZE                     2
BBOX_STD_DEV                   [0.1 0.1 0.2 0.2]
DETECTION_MAX_INSTANCES        100
DETECTION_MIN_CONFIDENCE       0.7
DETECTION_NMS_THRESHOLD        0.3
GPU_COUNT                      1
GRADIENT_CLIP_NORM             5.0
IMAGES_PER_GPU                 2
IMAGE_MAX_DIM                  256
IMAGE_META_SIZE                29
IMAGE_MIN_DIM                  256
IMAGE_RESIZE_MODE              square
IMAGE_SHAPE                    [256 256   3]
LEARNING_MOMENTUM              0.9
LEARNING_RATE                  0.001
MASK_POOL_SIZE                 14
MASK_SHAPE                     [28, 28]
MAX_GT_INSTANCES               100
MEAN_PIXEL                     [123.7 116.8 103.9]
MINI_MASK_SHAPE                (56, 56)
NAME                           food-challenge
NUM_CLASSES                    17
POOL_SIZE                      7
POST_NMS_ROIS_INFERENCE        1000
POST_NMS_ROIS_TRAINING         2000
ROI_POSITIVE_RATIO             0.33
RPN_ANCHOR_RATIOS              [0.5, 1, 2]
RPN_ANCHOR_SCALES              (32, 64, 128, 256, 512)
RPN_ANCHOR_STRIDE              1
RPN_BBOX_STD_DEV               [0.1 0.1 0.2 0.2]
RPN_NMS_THRESHOLD              0.7
RPN_TRAIN_ANCHORS_PER_IMAGE    256
STEPS_PER_EPOCH                150
TRAIN_BN                       False
TRAIN_ROIS_PER_IMAGE           200
USE_MINI_MASK                  True
USE_RPN_ROIS                   True
VALIDATION_STEPS               50
WEIGHT_DECAY                   0.0001
```

The model has been trained in 3 different stages, with the following configurations.

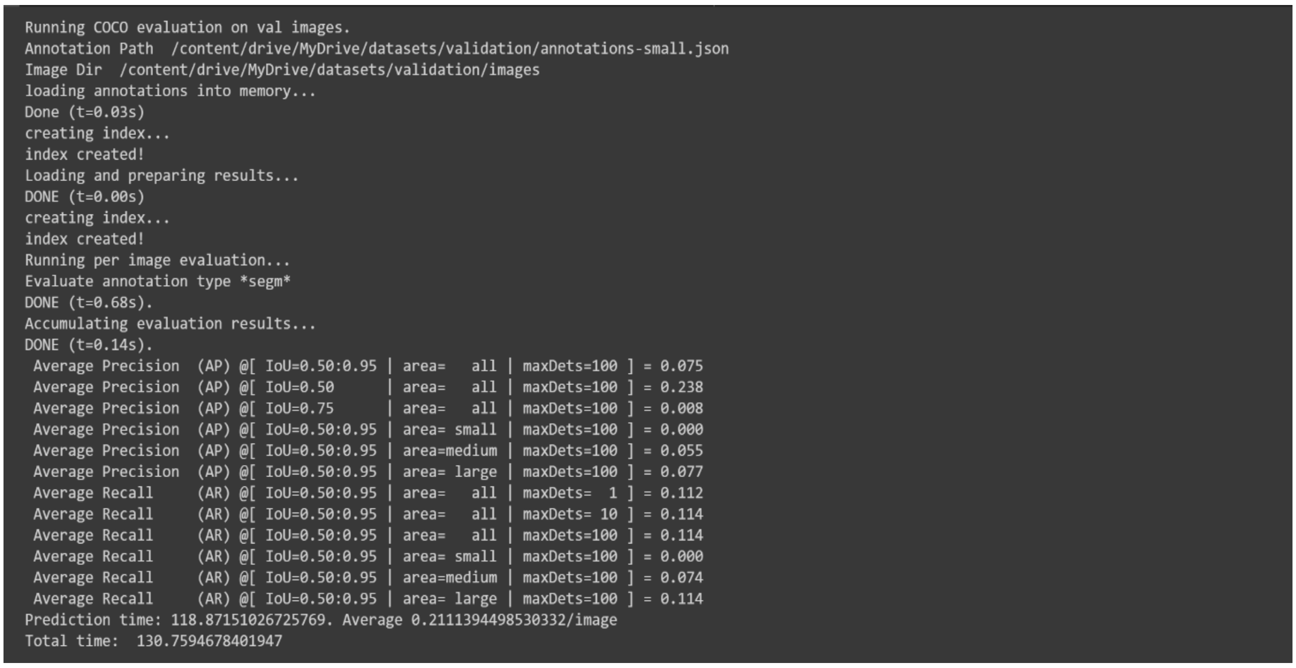
17 categories, up to 150 steps per epoch, up to 50 validation steps, a decreasing learning rate starting from 0.001 and two images per GPU.

We have chosen 70 steps per epochs, and 65 equally distributed epochs among the 3 stages. Each stage is made up of an increasing number of layers, starting from the first stage up to the third one which uses all the layers possible.

This configuration has been chosen also for the limits of the Google Colab's GPU, which did not permit us to train it for more time. We are given roughly 12GB of RAM and 69GB of ROM, but the amount of consecutive time of GPU usage are variable, so we could not stay connected for more than 3-4 hours.

Evaluation Results:

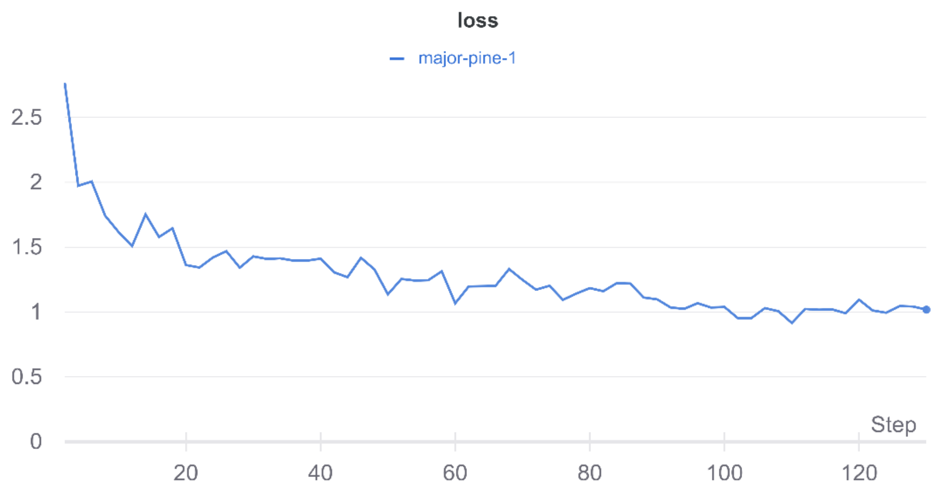
With this configuration on the given dataset, we have reached the following results in terms of loss, average recall, average precision and IoU:



```
Running COCO evaluation on val images.
Annotation Path  /content/drive/MyDrive/datasets/validation/annotations-small.json
Image Dir  /content/drive/MyDrive/datasets/validation/images
loading annotations into memory...
Done (t=0.03s)
creating index...
index created!
Loading and preparing results...
DONE (t=0.00s)
creating index...
index created!
Running per image evaluation...
Evaluate annotation type *segm*
DONE (t=0.68s).
Accumulating evaluation results...
DONE (t=0.14s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.075
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.238
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.008
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.055
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.077
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.112
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.114
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.114
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.074
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.114
Prediction time: 118.87151026725769. Average 0.2111394498530332/image
Total time:  130.7594678401947
```

In the above picture we can see different values of AP and AR, depending on the area and the given IoU. AP in particular increases obviously if the IoU decreases and increases the considered area, while AR increases if we consider a larger area.

Considering all the previous considerations, from a small interval of training time to the fact that we have a very big dataset that requires lots of computing, we were able to reach a 0.24 IoU.

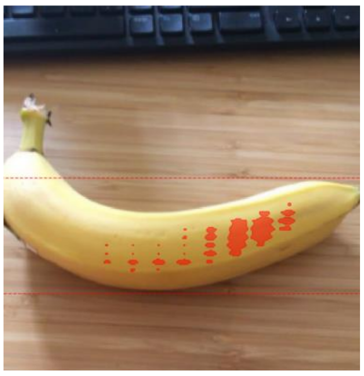


In this last graph we can see the behaviour of the losses, that due to the learning rate decreases as increases the number of steps and, more generally the total training of the net.

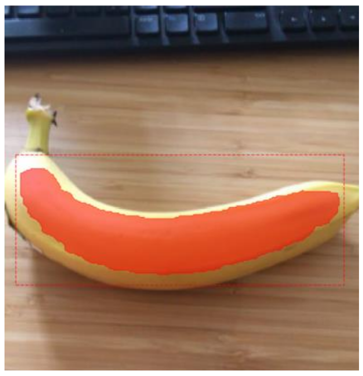
Being able to do lots of epochs is fundamental to obtain some decent loss. In fact, we see a great improvement with some continuous ripple just from 40 epochs and on.

Let's see now, practically, how the net has improved during the training stages:

Here's a picture taken after 15 steps:



And here's at the end of the training:



It's clear that the banana has been better recognised at the end of the training. Even though the IoU doesn't reach top notch results, the network was still able most of the times to recognize the content also due to the dataset reduction.

Conclusions

The dataset of pictures from the *"aicrowd challenge"* was very promising from the beginning: more than 24000 images of a Swiss canteen are treasure for these kinds of problems. As agreed with the other groups we had to shrink the dataset to just 16 categories, which cut our actual dataset to about 11000 images.

We are aware that ideally with an infinite amount of time we could reach amazing values of food recognition. Due to time limits of Google Colab, we ended up taking the most of their servers for some hours each day and in fact we obtain strong performance on the training and on the evaluation of the dataset.

The cohesion of anchors, mini masks, bounding boxes and ROIs were fundamental to obtain some decent recognitions with more than 0.7 of accuracy in just few hours of training. Still, this was not enough to reach a sufficient value of IoU; since most of the times even with a correct detection the actual mask was still far away from the ground truth, and this caused the AP to fall quickly.

This project has been very fun and interesting, and we were able to learn more about Deep Learning, Python and how to implement Neural Networks.

Bibliography

- https://www.aicrowd.com/challenges/food-recognition-challenge
- https://en.wikipedia.org/wiki/Image_segmentation
- https://towardsdatascience.com/computer-vision-instance-segmentation-with-mask-r-cnn-7983502fcad1
- https://engineering.matterport.com/splash-of-color-instance-segmentation-with-mask-r-cnn-and-tensorflow-7c761e238b46
- https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/