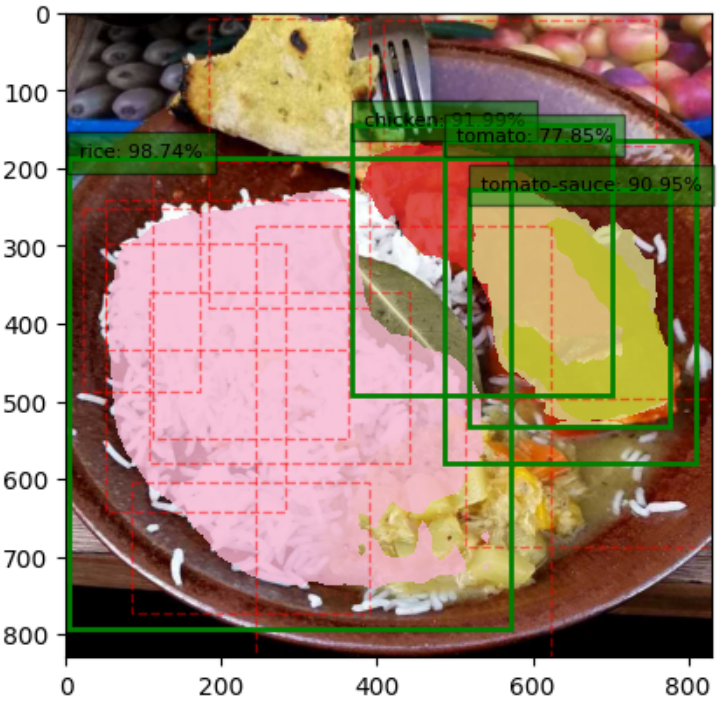
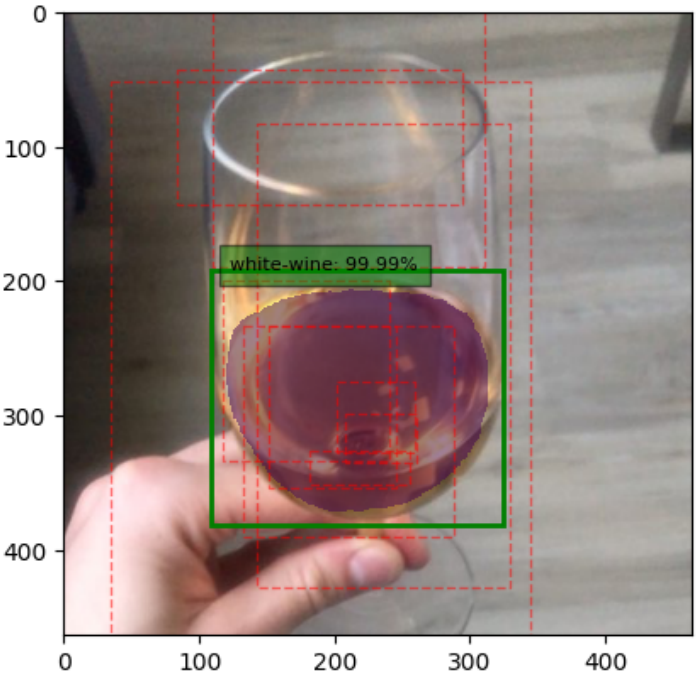
Food Recognition Project

Federico Cichetti - *federico.cichetti@studio.unibo.it*
Marcello Ceresini - *marcello.ceresini@studio.unibo.it*





GitHub repository:
https://github.com/MarcelloCeresini/ImageSegmentation.git

Introduction

Instance segmentation is one of the most studied and important tasks in the vision community, fastly developing and improving over time. Following the simpler tasks of object detection, classification and semantic segmentation, it is one of the most important tools to allow a computer to understand what is present in an image.

We explored the problem by developing a Deep Learning model able to recognize and localize different kinds of food on a dataset of pictures taken from smartphones, a problem that has been proposed in the **Food Recognition Challenge**, an Artificial Intelligence challenge held on AIcrowd.

To address this task, we first looked at various papers published in recent years containing Deep Learning solutions for Instance Segmentation: between the many architectures that have been proposed by researchers, we chose to follow the steps of **Mask R-CNN**[1], an architecture first presented in a paper from 2018 by researchers at Facebook AI Research (FAIR) that builds on the success reached in previous years by Convolutional Neural Networks, in particular **Faster R-CNN**[2].

We implemented a complex network composed of a backbone and multiple heads, that counts more than tens of millions of parameters. Given its complexity, a good amount of computational power for training and fine tuning would be required, together with a large and thorough dataset. We chose to implement the neural network and related code using Python and the Tensorflow 2.5 and Keras libraries.

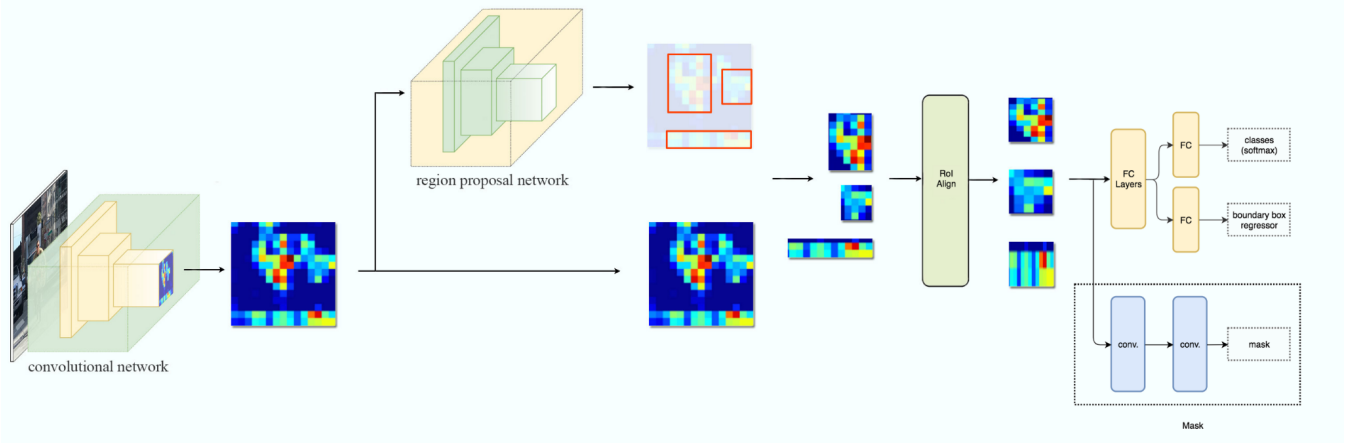
Since this challenge is oriented to developing solutions working in real-life scenarios, photos in the dataset do not reach a high level of resolution or contrast, being similar to the ones that could be captured in everyday life.
Moreover, since the dataset is oriented to supervised learning, it provides labeling of every single instance of food in every single picture, whose creation is a time-consuming effort that was brought on by simple volunteers and not by experts.
Another problem was related to the absence of a naming convention for classes of food, resulting in a lot of similar classes, each with a very small number of instances.
Putting the already difficult task of Image Segmentation together with low quality pictures and annotations brings another layer of difficulty to this project.

We tried to mitigate the mediocrity of the dataset and scarcity of training data by using a **pre-trained backbone** and employing **transfer learning**, with good results**.** Moreover, after a further exploration of the dataset we chose to group together or prune most of the classes, reducing the initial number of detectable classes from 273 to 28. We will explore this choice, as well as other implementation details in the next sections.

[1] https://arxiv.org/abs/1703.06870v3
[2] https://arxiv.org/abs/1506.01497

Network Architecture

The R in R-CNN stands for “Region (Based)”. Indeed, the key idea behind Mask R-CNN and its parents, R-CNN[3], Fast R-CNN[4] and Faster R-CNN is to extract a large number of **regions** or **proposals** from the image and apply further computations on the features extracted from these sub-regions rather than from the entirety of the image. This allows a finer granularity in the analysis of the images.



The full pipeline of Mask R-CNN

Backbone

The first step of Mask R-CNN is to generate **feature maps** by passing the image to the **backbone** network, in our case a **ResNet-50**, a well known and lightweight CNN from which we remove the last max pooling and classifier layers. We chose this CNN rather than others because of memory constraints due to executing the code on our local machines, but acknowledge that deeper and more complex networks would be able to generate better features.

Keras provides access to this network and allows us to automatically download good weights pre-trained on **ImageNet** using just a few lines of code. This trick greatly sped up training convergence, as we started training our network from weights which already encoded the knowledge needed for object classification. At training time, we mostly trained only the heads built on top of the backbone, or just some of the final layers of the networks to do some **fine-tuning**. The training experiments will be explained in detail in a later section.

Feature Pyramid Network

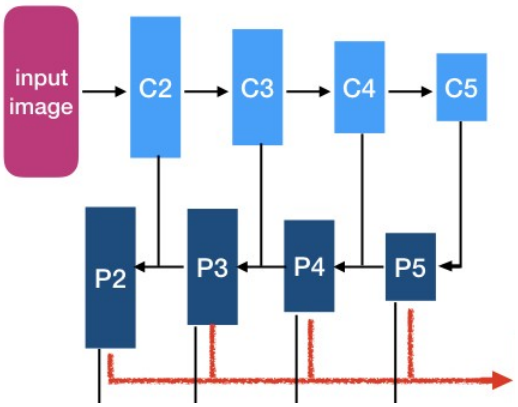
The feature maps are further processed with a **Feature Pyramid Network[5] (FPN),** which builds a **Feature Pyramid** in order to let the rest of the network deal with features at **different scales.** This greatly enhances the detection of smaller objects in the image.

An FPN is built with two pathways:

[3] https://arxiv.org/abs/1311.2524
[4] https://arxiv.org/abs/1504.08083
[5] https://arxiv.org/abs/1612.03144

- The **Bottom-Up pathway,** which in our case is represented by the backbone computation and the 5 feature maps produced at the end of each convolutional block of the ResNet. For simplicity, we call these feature maps C1, …, C5.
    - These feature maps decrease in size but become deeper and deeper: from 256x256x64 (C1) to 32x32x2048 (C5).

- The **Top-Down Pathway** gradually up-samples the feature maps from the previous layer and adds the feature maps from the backbones through **lateral connections.**
    - The feature maps grow in size, but are all kept at the same dimensionality (from P5 = 32x32x256 to P2 = 256x256x256)
    - A final 3x3 convolution is applied to each of the resulting feature maps
    - An additional feature map, P6 = 16x16x256 is obtained by subsampling from P5.



An FPN similar to the one that has been used in our implementation.

These feature maps are efficiently re-used when predicting the proposals, the bounding boxes and the masks. The model takes into account the **scale** of the predicted bounding box and connects the appropriate feature map to the heads.

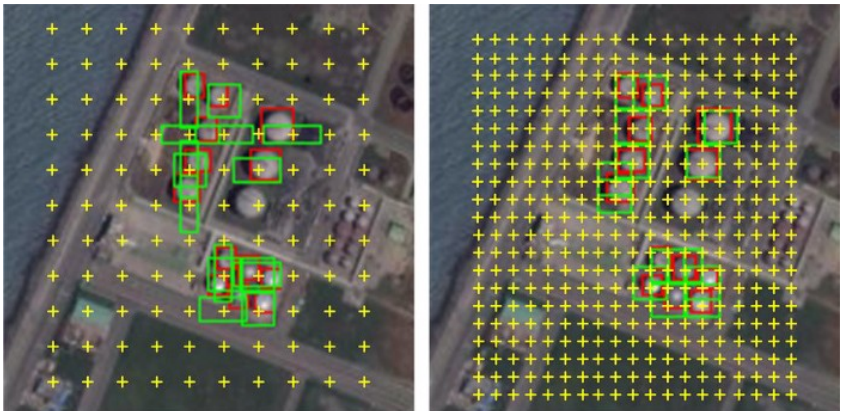
Region Proposal Network

While proposals are generated algorithmically in R-CNN and Fast R-CNN, Faster R-CNN introduces the **Region Proposal Network (RPN)** which generates proposals through a lightweight neural network. This means that the network can actually learn which proposals are good and which will be almost pointless for the rest of the network, thus generating potentially more interesting features.

The RPN is particularly efficient because it re-uses the same features extracted from the image by the backbone network. Also, it does not calculate region proposals from scratch, but uses some **precomputed anchors** from which it calculates some **deltas**, representing adjustments to these anchors. Furthermore, for each refined anchor, the RPN computes an **objectness score**, representing whether the region calculated from the RPN contains an object (positive class) or is a background region.

First of all, a set of **anchors** is pre-computed by applying some geometrical calculations over the sizes of the FPN feature maps. The traditional way of computing anchors would be to take a

feature map, use its pixels as **anchor points** (centers of the anchors) and apply 3 scales and 3 ratios on each of them in a sliding window fashion. Since we have multiple feature maps of different sizes, we instead calculate the anchors' bounding boxes using as anchors point each pixel of the 5 feature maps (P2, …, P6) while being sure that they are well aligned with the image's pixels. We then apply 3 ratios to each of them. As ratios (width/height) we use `[0.5, 1, 2]`, so each anchor point has one wide, one tall and one square anchor.

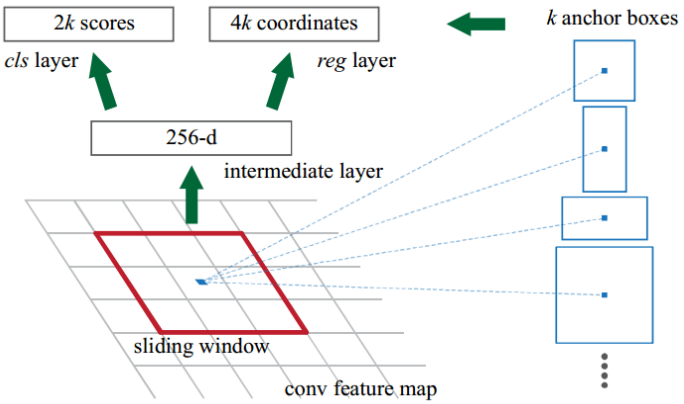


Anchor points from two feature maps visualized over the original image

This way, the model computes **multiscale anchors**, because anchors which are computed using smaller feature maps will be larger, while finer feature maps, often encoding for small details, will result in smaller anchors.

Anchors are passed as inputs to the model, which in the meanwhile does its own computations in terms of **1x1 convolutions** applied on each layer of the feature pyramid.

- The first convolution generates 6 values for each pixel of the corresponding feature map: the background and foreground classification logit scores for each of the 3 ratios of the anchors. These logits are softmaxed so that we produce actual **probability distributions**.
- The second convolution generates 12 values for each pixel: these are the 4 **deltas** to be applied on each of the 3 anchors.



The computation of the RPN submodule

Refinement Layer

The **Refinement Layer** does two things:

- It **applies the bounding box deltas** that have been calculated by the RPN on the precomputed anchors
- It uses these coordinates and the scores (also computed by the RPN) to **select a subset** of good proposals and reduce their number for computational efficiency. In our case we only select the best 6000 proposals.

The final coordinates of the proposals are computed this way:

$$
\begin{aligned}
tx &= xa + (x * wa) \\
ty &= ya + (y * ha) \\
tw &= ha * e^{w} \\
th &= wa * e^{h}
\end{aligned}
$$

where xa, ya, wa, ha are the anchors coordinates (center coordinates, width and height, normalized with respect to the image), x, y, w, h are the bounding box refinement outputs of the RPN (w and h are computed in log-space, while x and y must be normalized by the anchors' width and height) and tx, ty, tw, th are the computed coordinates of the proposal center, width and height.

The Refinement Layer then applies the so-called **Non-Max Suppression (NMS)**, which is a **greedy algorithm** that prunes out boxes which have a high **Intersection over Union (IoU)** with boxes that have a greater score and are therefore preferred. We use Tensorflow's implementation of this algorithm, which is available in the `tf.image` module. In particular, we used the `combined_non_maximum_suppression` version which is able to deal with batches automatically.

The **Intersection over Union (IoU)** is a simple metric that takes the area of the intersection of the two boxes, the area of their union, and calculates the ratio between them. If the two have no overlapping, the intersection area will be zero, so IoU will be zero. In the opposite case, when we have two completely overlapping boxes, the area of intersection will be identical to the area of the union, resulting in an IoU of 1. In our case, the threshold IoU over which a box is filtered out is fixed to 0.7.

After the deletion of all the boxes with IoU over 0.7 with other, better scored boxes, we only keep the top-k (with k = 2000 in training and 1000 during inference) proposals with respect to the scores, to ease up computation in the remaining part of the network.

Detection Target Layer

After the Refinement Layer, the model execution forks: if the model is in **training mode**, the computed proposals, as well as the (input) ground truth bounding boxes, classes and masks go through the Detection Target Layer, which is a layer that is used to append some useful groundtruth information to the proposals so that they can be used for training.

In particular, this layer takes care of:

- Determining whether the class of a computed proposal is **positive** or **negative**. By calculating the IoU with every groundtruth box and imposing a minimum threshold, we can determine if it matches with some groundtruth object or if it is just a background region. This is later used to compute the classification loss produced by the object classification head of the model.
- Applying a **smart subsampling** of the proposals. Since the number of negative proposals probably exceeds the positives by a great deal and we don't want the training to be biased towards background bounding boxes, we subsample them by keeping a **1:2 ratio** between positives and negatives. Actually, a 1:1 ratio would technically be preferable, but since our dataset is mostly composed of photos with just one annotation in the center of the image, we would risk having too few positive proposals and reducing by a lot the learning capability of the network.
- Calculating **groundtruth deltas**. These deltas are calculated as differences between the subsampled **positive** proposals from the RPN and their respective matched groundtruth bounding box. Later, they are used for calculating the loss with respect to the bounding box deltas produced by the respective head in the model. In particular:

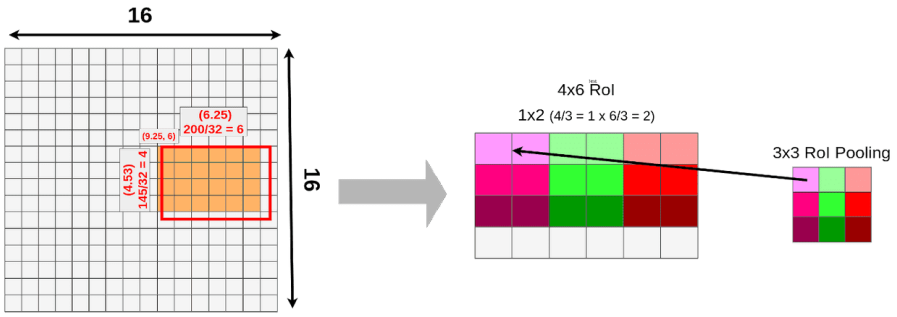
$$
\begin{aligned}
dy &= \frac{(y_{gt} - y)}{h} \\
dx &= \frac{(x_{gt} - x)}{w} \\
dh &= \log(h_{gt} - h) \\
dw &= \log(w_{gt} - w)
\end{aligned}
$$

  where dy, dx, dh, dw are the deltas from ground truth bounding boxes, the elements marked with gt are ground truth box coordinates and x, y, h, w are the proposal coordinates.
- **Associating groundtruth masks to proposals**, so that we can later compute the loss with respect to the masks produced by the model. In particular, training masks are represented as 28x28 binary images extracted through **ROIAlign** from the proposal and binarized using a 0.5 threshold.

ROIAlign

ROIAlign is a new kind of **feature map pooling** introduced in the Mask R-CNN paper.
ROI pooling in general is utilized to transform a ROI (Region of Interest) of any shape into a **fixed-size array**. This trick solves the problem of fixed image size requirements for the detection network, since we can further process these transformed fixed sized feature maps through dense layers and such.

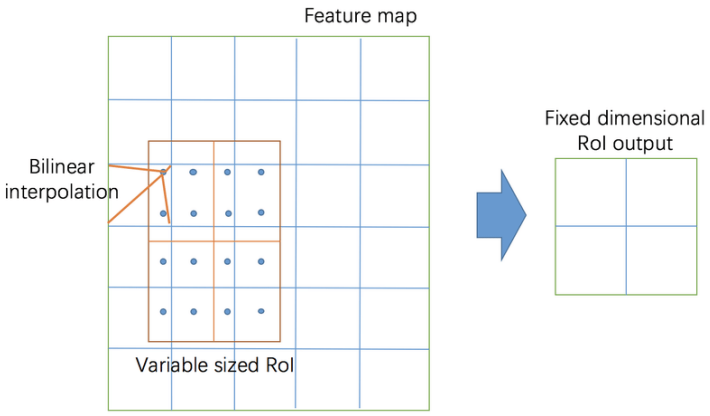
Given the floating point coordinates of the proposals, **ROIPool** (the main pooling algorithm also used in Faster-RCNN) would first quantize these coordinates with respect to the stride of the respective feature map, then subdivide these portion of the map into quantized bins, creating a grid as large as the desired output, and then it would aggregate results from the subregions of the grid by max pooling.



A demonstration of the coarse quantization of ROIPool

The authors of Mask R-CNN observed that the rough quantization of ROIPool was problematic for the computation of the mask, which instead requires **precise localization** of the boundaries of objects. Their **ROIAlign** pooling method removes this quantization by **properly aligning the extracted features with the input**.

- ROI bins are not quantized with respect to the feature maps, but are calculated in floating point coordinates.
- Within each bin, 4 points are sampled at regular locations. The value of the feature map on these points is computed by **bilinear interpolation** with respect to the surrounding pixels.
- With the above two steps we have removed the need for quantization.
- Then, the 4 points are aggregated by max or mean as is done for ROIPool, obtaining the final values of the feature map



A visualization of the ROIAlign method

In our code, rather than implementing ROIAlign from scratch, we use a “simplified” version provided with Tensorflow: the `crop_and_resize` function, which is basically the same as having just one point sampled through bilinear interpolation within each bin.

The ROIAlign layer is implemented at the beginning of the heads of the network, but with a spin: since the ROIs can have vastly different dimensions, we implemented an FPN to be able to extract features at different resolutions inside the image. We thus called this layer `PyramidROIAlign`. Within `PyramidROIAlign` we assign to each ROI the feature map that maximizes its expressiveness, choosing between P2, P3, P4 and P5 (the feature maps of the FPN). We use the following formula from the FPN paper[6]:

[6] https://arxiv.org/pdf/1612.03144

$$k = k_0 + log_2(\frac{\sqrt{wh}}{224})$$

Where k is the feature map level we assign to the ROI and k_0 is the level in which a square ROI with an area of 224^2 should be mapped into. It is usually set to 4, since Faster R-CNN systems implemented without the RPN always chose C_4 as their “single-scale” feature map. The “magic number” 224 is in reality the standard ImageNet training size.

Once every ROI is assigned to a level, the `crop_and_resize` function uses the right feature map for each single ROI, and extracts the correct features.

Detection Layer

As opposite to the Detection Target Layer, the Detection Layer is only executed when the model is in **evaluation mode**.

- It is used to **apply** to the selected proposals the corresponding **deltas** calculated in the bounding box regression branch using the formulas explained before.
- Also, it **filters out background proposals**, as well as low-confidence bounding boxes (simply applying a threshold).
- It applies **per-class NMS**, removing bounding boxes that have lower confidence and the same class as others they overlap with.
- As a final filtering step, it selects only the **top-k detections** with respect to their confidence, where k is set to 100. If there are less than k detections, the resulting tensor is padded with zeros.

Classification and Bounding Box Head

After all the work in the previous layers, finally the extracted features and proposals are fed into the heads of the model. The first head is responsible of both the **classification** and the **prediction** of the final **bounding boxes**: it starts by taking as input the feature maps and all the region of interest from the Detection Target Layer, and then applies the ROIAlign layer to receive from each ROI a 7x7 feature map.

Following this step, a 7x7 convolution and another 1x1 convolution are applied, without mixing the channels (the ROIs). To avoid the mixing we exploit the `keras.layers.TimeDistributed` layer, which simply applies the operation of the layer to each channel independently.

Finally, two dense layers complete the head:

- The first has the same shape as the number of total classes in the dataset, and is the **classification layer**. It is then followed by a softmax activation function that returns the final classification probabilities.
- The second is the **bounding box regressor layer**. It has four times the shape, because the network creates a bounding box (with four coordinates) for each class, for each ROI. In this case no activation is needed, since these are simply points in space.

Mask Head

As in the previous case, the ROIAlign layer is the first step inside the head. In this case, however, the pooling size is bigger, 14x14, to accomodate for the need of a higher resolution for the masks.

Then, we apply four identical blocks of 3x3 convolutions + ReLU, then a deconvolution with pool size 2 and stride 2, and finally a last 1x1 convolution with as many output channels as the number of classes: just as the bounding box regressor, the mask head predicts a mask for each class. As before, we don't need a final activation, because the values that come from this head still need to be processed before the loss function.

Starting from a 14x14 size after the ROIAlign layer, we end up with a 28x28 mask after the deconvolution. Many ground truth masks are way bigger than 28x28, but the DetectionTargetLayer already accounted for this difference by applying the ROIAlign on the groundtruth masks, shrinking them into a 28x28 feature map, to allow for a comparison inside the loss function.

The MaskRCNN object

The model is constructed in the `build()` method when creating the `MaskRCNN()` object, which contains the model itself as well as many other utility functions for training and compiling the model, dealing with save paths in the filesystem and running detection on images.

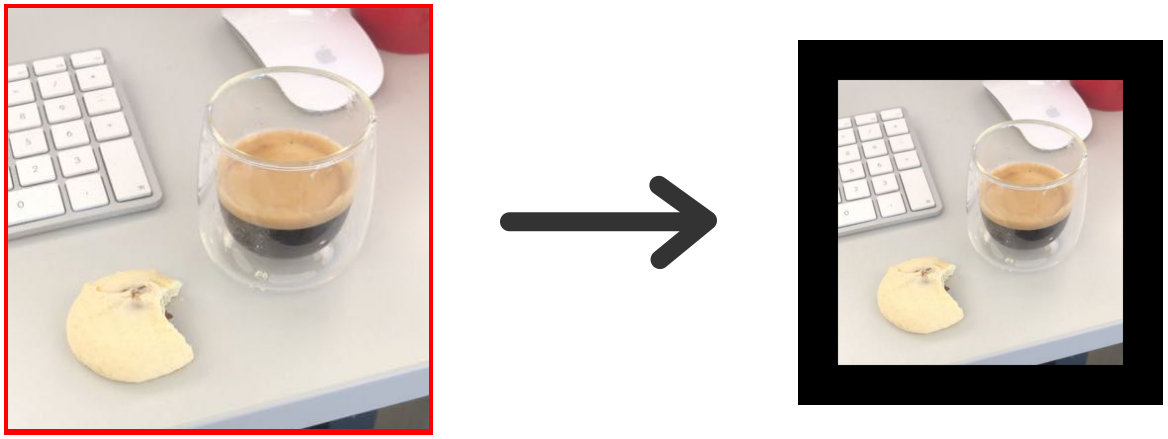
With the `MaskRCNN.train()` method, the group of layers to train can be specified: we can train either only the FPN, the RPN and the heads, we can start from the 3rd, 4th or 5th groups of layer of the backbone, or we could also train all the parameters of the network. We used this choice extensively in the training phase, so that we could maintain the stability of weights by applying fine tuning rather than training from scratch.

Another important method is the `MaskRCNN.detect()` method. It receives a batch of images, preprocesses them in the format preferred by the network, sends them as input to Mask R-CNN, intercepts its output and post-processes it.

Detection Pre-Processing

The main preprocessing step is the resizing of the image to a **square of fixed size** (we use 1024x1024) by **maintaining its original aspect ratio**. We use the `skimage.transform.resize` function, which uses bilinear interpolation for the resizing and automatically zero-pads the square to fill the parts that are not covered by the resized image.

We keep track of which part of the square contains the image and which is zero-padded in a `window` variable. We also save a `scale` variable for the resizing scale (greater than 1 means that the image has been enlarged, lower than one means that the image has been shrunk).
The resized image is then normalized by subtracting a pre-computed mean pixel (from the COCO dataset).



Preprocessing transformation. Original image on the left, preprocessed on the right. The actual preprocessed image would be a floating point image, but in this example we have de-normalized it.

As an additional preprocessing step, we **calculate the anchors** with the algorithm we have described before, since in evaluation mode the model expects the anchors to be part of the input rather than calculate them on the fly.

Detection Post-Processing

The outputs of the model get post-processed so that we can deal with the network's results in a more convenient way. Post-processing steps include:

- The **transformation of the coordinates** of both detection bounding boxes and RPN proposals from the network's format to the original image's format.
    - Indeed, the network uses normalized coordinates with respect to the square-resized and possibly zero-padded image created in pre-processing.
    - Thus, we first convert normalized coordinates with respect to the squared image in normalized coordinates with respect to the non-padded window containing the image.
    - Only then we denormalize the coordinates obtaining pixel coordinates in the original image.
- Filtering out boxes with **empty areas** (which we have found to happen a lot more than expected in early training).
- Similarly to the bounding boxes, the **masks**, which are expressed with respect to the proposals they were predicted into, **are transformed too**.
    - We use the same resizing with bilinear interpolation we had used for preprocessing.
    - Then, we **binarize** the mask using a 0.5 threshold and pad with 0s the area of the image that is not covered by the mask.
    - Therefore, the masks are transformed into a series of boolean masks as large as the image, which is a simple format we can use for further analysis and drawings.

The post-processed results are aggregated into a **Python dictionary**, which is the final output of the `detect()` method. We use a dictionary like the following:

```
{
   "rpn_boxes": [M, (y1, x1, y2, x2)],
   "rpn_classes": [M],
   "rois": [N, (y1, x1, y2, x2)],
   "class_ids": [N],
   "scores": [N],
   "masks": [H, W, N]
 }
```

Where `rpn_boxes` and `rpn_classes` are numpy vectors containing the proposals and their scores, `rois`, `class_ids` and `scores` contain the final detections, their predicted class ID and scores, while `masks` contains the binary masks, formatted as explained before.

A very simple example on the use of this output dictionary is provided in the `run_quick_test.py` script, which loads two images from the dataset, sends them to the network, parses the output dictionary and uses it to draw detections, classes and masks on the images.

Loading the dataset in memory

During training and evaluation we need to continuously feed the model with data from the dataset. Of course, loading the entirety of the dataset in memory is not a viable choice, so what is usually done is using some kind of **on-the-fly generator** that returns the next piece of data only when it is needed.
In Python, this concept is often implemented by using a Generator, which is a special function we can iterate on and at each step of the iteration computes the “next” datum.

Keras supports Python generators as inputs for the training phase, but there are even smarter kinds of generators provided by Keras itself: in our case, we decided to subclass the `Sequence` class, which is a **multiprocessing-safe generator**.

Our DataGenerator class implements a flexible data generator for our dataset: at each call, it generates **all the necessary inputs** for the model: preprocessed images, their metadata, groundtruth proposal deltas and RPN classes, groundtruth bounding box deltas, classes and masks.

The DataGenerator is designed to be **dataset-agnostic**, in the sense that it could be reused for datasets similar to this one (any dataset expressed in COCO format). This choice is reflected in the fact that it requires a “dataset” object at initialization time, which in our case must be of class `FoodDataset` (implemented in the file `food.py`). This class holds training or validation IDs, classes and image paths related to our dataset, as well as providing utility methods for loading masks and images in memory. We can see it as some sort of gateway to the dataset on disk.

The DataGenerator object does the following steps as soon as a request for the “next” batch is made:

1) The image, as well as the related groundtruth, is obtained from **querying the `FoodDataset` object**, asking for a specific (possibly random at training time) ID.

2) The image gets **preprocessed** similarly to what happens in detection and the masks follow the same treatment. Optionally, we could add some **data augmentations** (for example, for all of our training experiments, we have flipped the image 50% of the time, but we could add some more elaborate transformations as well).
3) The **RPN targets** are computed by matching the list of precomputed anchors (they are the same for all images) with groundtruth bounding boxes.
    a) For bounding box classes, we adopt the following algorithm:

        i) If an anchor overlaps a GT box with IoU $>= 0.7$ then it's positive (1).
        ii) Also, the anchor which has the highest IoU with a GT box is assigned to be of positive class even when its IoU is < 0.7. This means that there is always at least a positive match for each GT box.
        iii) If an anchor overlaps a GT box with IoU < 0.3 then it's negative (-1).
        iv) Anchors that don't match the conditions above are assigned a neutral class (0) and do not influence the loss function.

    b) Then, for positive anchors, the deltas are computed as differences with respect to the groundtruth boxes.
4) The DataGenerator finally populates a batch of data with all this information and returns it so that the model can use it.

Losses

Mask R-CNN is a complex model to train, because it's built using lots of interconnected submodules. We have the backbone, the FPN, the RPN and two output heads, and to obtain a model that performs well we need that each submodule does its own job well. Therefore, the model has **five loss functions**:

- `rpn_class_loss` is the loss resulting from the foreground/background classification of the proposals by the RPN.
- `rpn_bbox_loss` is the regression loss between the groundtruth proposals passed as input and the proposals computed by the RPN.
- `mrcnn_class_loss` is the classification loss between the groundtruth class of an object and the assigned class of the respective detection computed in the bounding box regression head of Mask R-CNN.
- `mrcnn_bbox_loss` is the regression loss between the groundtruth bounding box and the detection computed in the bounding box regression head.
- `mrcnn_mask_loss` is the loss between the groundtruth mask and the respective predicted mask.

rpn_class_loss

This loss is almost a standard **categorical crossentropy loss** for classification, with the only difference that we also take into account that groundtruth proposals can be classified as positive, negative or neutral, as explained in the section about the DataGenerator, and that **neutral proposals do not contribute to the loss**.
We filter them out and apply Keras's `SparseCategoricalCrossentropy` loss function (because we don't use one-hot encoded vectors). As a remainder, categorical cross entropy is defined as:

$$-\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

where $\hat{y}_i$ is the predicted probability for class i, while y_i is the groundtruth probability (1 or 0).

rpn_bbox_loss

This loss is a **smooth L1 Loss,** also known as **Huber Loss.** It's a kind of L1 loss that has been found to be more resistant to outliers. Basically, given a hyperparameter d, the Huber loss is defined as:

$$\begin{aligned} &\tfrac{1}{2}(y - \widehat{y})^2 && if\ |y - \widehat{y}| \leq d \\ &\tfrac{1}{2}d^2 + d(|(y - \widehat{y}| - d) && if\ |y - \widehat{y}| > d \end{aligned}$$

Before the computation, **only positive proposals are selected**, because it doesn't make much sense to penalize the model for applying wrong deltas to background boxes.

mrcnn_class_loss

The classification loss for the final bbox predictions is really simple, a straight up **categorical crossentropy loss**. In this case, in fact, there are no positive or neutral boxes to filter out, since the background is a class in itself (with id = 0). So we simply compare the ground truth classification of each box with its predicted counterpart, and take the mean between all the predictions.

mrcnn_bbox_loss

The loss for the bounding boxes, instead, needs to filter out all the background boxes. Moreover, since the bounding box head produces 4 predictions **for each class** for each ROI, we need to gather only the bounding boxes corresponding to the right class. Obviously, bounding boxes for other classes will not tell us anything interesting.

After the filtering, the loss consists simply in the application, for all the four elements of the bounding box, of the **Huber Loss** described above. In the end, as always, we take the mean of the results.

mrcnn_mask_loss

The final loss of the model is the mask loss. Similarly to the bounding box loss, masks do have to be filtered: we remove the background mask and for each of the predicted masks we only consider the mask for the correct groundtruth class.

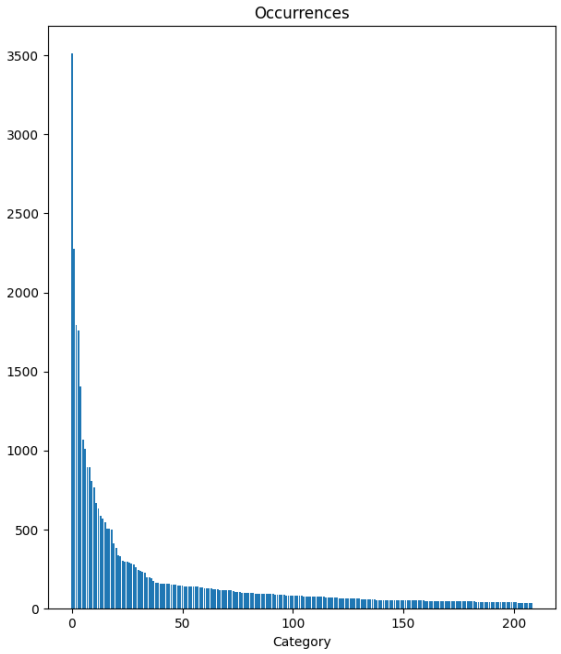
The remaining masks will be compared with the groundtruth masks that were shrunk and rounded by the DetectionTargetLayer. The loss function that we use here is the simple **binary cross entropy**, the binary version of the categorical cross entropy, since each "pixel" of the groundtruth

mask can be either zero or one (or in other words: every pixel is one of two classes). Also in this case, a mean between all the losses is taken.

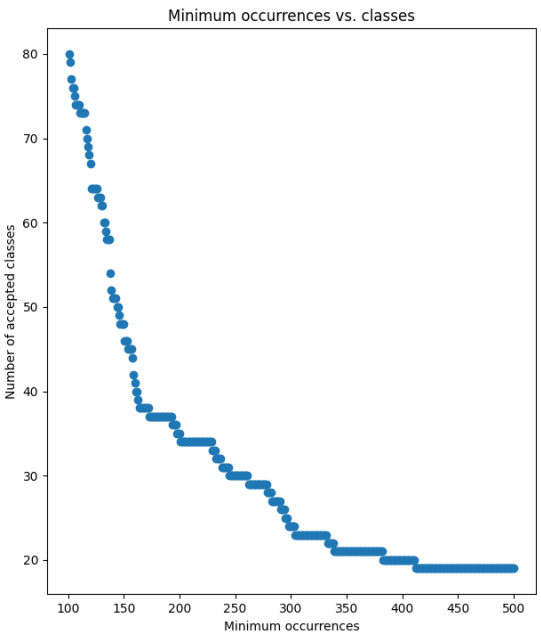
The final loss of the model is simply the sum of all the previous losses.

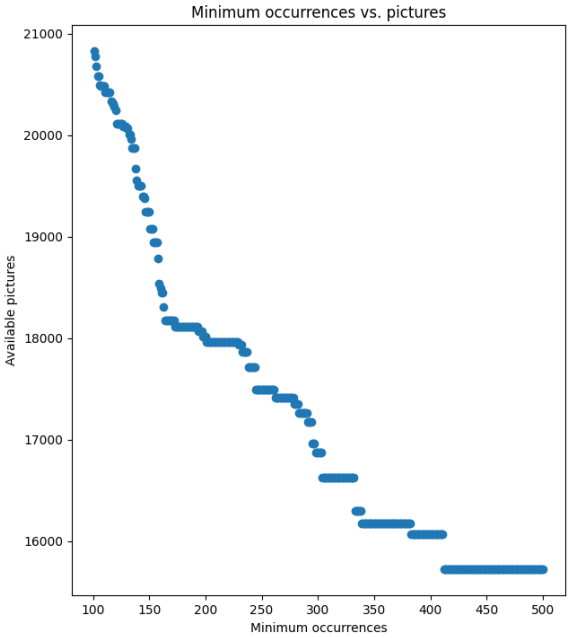
Dataset manipulation

To try and achieve good results, we thought that the best way to proceed would be to only use categories (foods) that appeared with a good frequency with respect to the whole dataset. Unfortunately in this way many of the images present in the dataset would have been discarded, not containing any of the accepted foods.
To improve the total number of images available for training, we decided to start by **grouping similar foods in a single class** (i.e. “bread-white”, “bread-wholemeal”, “bread-whole-wheat” etc… in a single “bread” category).
After that, we ended up with the following class distribution:



It is immediately visible that, even with the creation of super-categories, the dataset contains a vast majority of classes with a scarce frequency of occurrence.





So, we fixed a threshold and we discarded every class that didn't meet the requirement, along with all the images in the dataset that did not contain any of the remaining class instances.

We analyzed the number of accepted classes, accepting images in relation to the threshold, and decided to keep a ratio of minimum occurrences with respect to the total remaining images of about 1.5%. This allowed us to keep a good amount of the initial images - 17k out of 24k - while maintaining a high number of classes: 28.

Our final set of classes is: `{apple, avocado, banana, beef, bread, butter, carrot, cheese, chicken, chocolate, coffee, cucumber, egg, jam, mixed-vegetables, pasta, pizza, potatoes, rice, salad, salmon, tea, tomato, tomato-sauce, water, red-wine, white-wine, yogurt-sauce}`

Training

The model has been trained on both Windows 10 and Ubuntu 20.04 systems, on a computer equipped with 16 GB of RAM, an AMD Ryzen 7 3700x 8-core CPU and a NVIDIA RTX 3070 8 GB graphic card. Tensorflow's subsystem warned us that additional VRAM could have been beneficial for the training, but allowed it anyway.

Training experiments

The following is the sequence through which we have obtained our best weights. We are aware that a more exhaustive search and longer training could lead to even better results, but our time and resources were quite limited.

- We used **Resnet50** as backbone and we passed **2 images per minibatch** to the model. We decided to start with **1000 training steps** per epoch and **50 validation steps** at the end of each epoch, **LR of 0.002**, training only the heads of the model and saving only the "best" weights according to the "val_global_loss" metric, which is the sum of all other losses. The optimizer we used was **SGD**, with a **momentum of 0.9** (same as the paper). The **16th epoch** of training yielded the best results, thus was saved as "best model" temporarily.
- Starting from these first weights, another training has been done for **55 epochs**. **LR** was initially reduced to **0.001** and **after 40 epochs to 0.0001** (using Keras's LearningRateScheduler callback). Training and validation steps were increased to **2000 training steps per epoch** and **200 validation steps**, to obtain more truthful results.
- We then started fine-tuning the backbone as well, training both the heads and the **5th group** of layers of ResNet50. The training parameters were the same as the previous training experiment.
    - We also decided to investigate what could happen using a different optimizer. Since the training with SGD didn't seem to decrease our losses any further and one of the culprits could have been the fixed learning rate, we chose **Adadelta**, because it can automatically select a suitable learning rate for each weight during training. We trained the model up to global **epoch 111**.
- At this point, we found that our model was pretty good at classification and building masks, but **lacked in bounding boxes detection**. The reference implementation normalized deltas dividing them by their **precomputed average standard deviation**. Initially we chose not to replicate this behaviour, but as an additional test we switched our focus on training the heads only and starting from epoch 111 we arrived at **epoch 158** with this change. Despite the higher initial loss, the training quickly improved the bounding box losses.

Training oddities

The implementation of MaskRCNN that we used as a reference for our project made some unusual decisions regarding the way data is fed to the model during training. Indeed, the groundtruth boxes and masks are passed as inputs rather than being used for an external comparison with the model's outputs.
This quirk is reflected in the way our custom `DataGenerator` class is built: the `__getitem__` method returns:

```
inputs = [batch_images, batch_image_meta, batch_rpn_match, batch_rpn_bbox,
batch_gt_class_ids, batch_gt_boxes, batch_gt_masks]

outputs = []
```

so, no output, but only a long list of inputs.
This can be justified by the fact that the **groundtruth** is not only a passive part of the model that is simply used for the losses computation, but it's an **active part of the training process** itself, since training samples for both the RPN, the Mask and BBox heads are generated on the fly from the groundtruth samples and the RPN proposals within the model in the `DetectionTargetLayer`.

This unusual way of using the groundtruth is not just a hack, but it actually makes the `DataGenerator` **more lightweight**, transferring **more workload to the GPU** and allowing us to use

more **optimized Tensorflow functions** for processing data. It also gives us more freedom in the way we can build losses, since we are not forced to simply compare the output of the model with the ground truth, but we can also take into account other information created directly into the model.

The main problem of this approach is that Keras' `fit()` method is not able by default to understand what we're doing, since it expects an X (the input to the model) and a y (the groundtruth to compare with the output of the model on input X for calculating the losses).

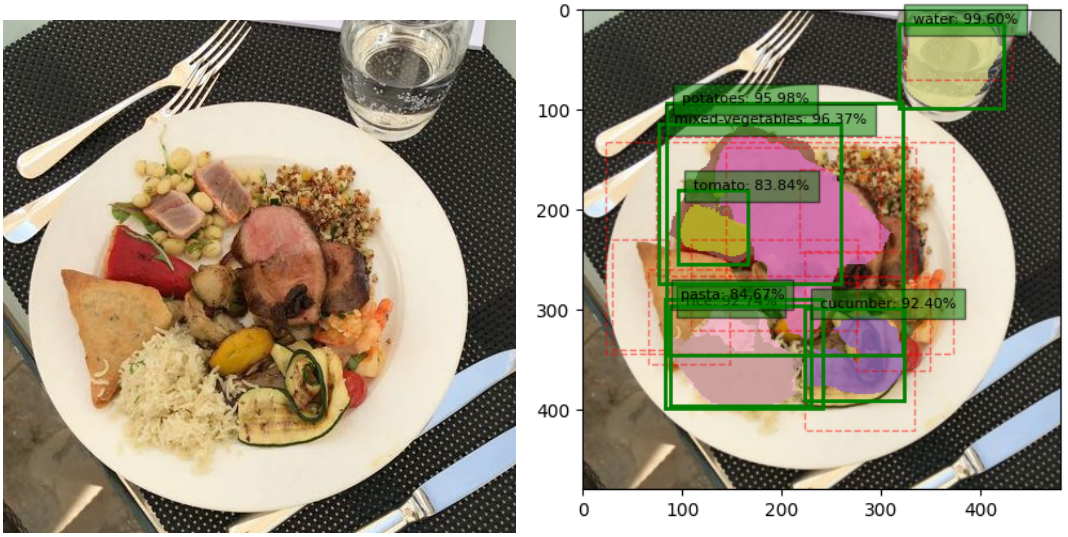
Since our y is an empty list, we decided to use a more advanced, but still well documented, functionality of Keras, that is **writing custom training/validation loops**. This way, we can instruct the training loop to ignore the "y" of our `DataGenerator`.

Another quirk in the training pipeline is that since we have to deal with multiple losses (two for the RPN, two for the bounding boxes and one for the mask head), and that these losses are built into the model as Keras Layers, rather than passing the loss functions to the `compile` method, we follow the reference implementation's idea and use the `add_loss()` API instead. This also allows us to easily retrieve the losses for the computation of the gradients in our custom training loop.

Training results

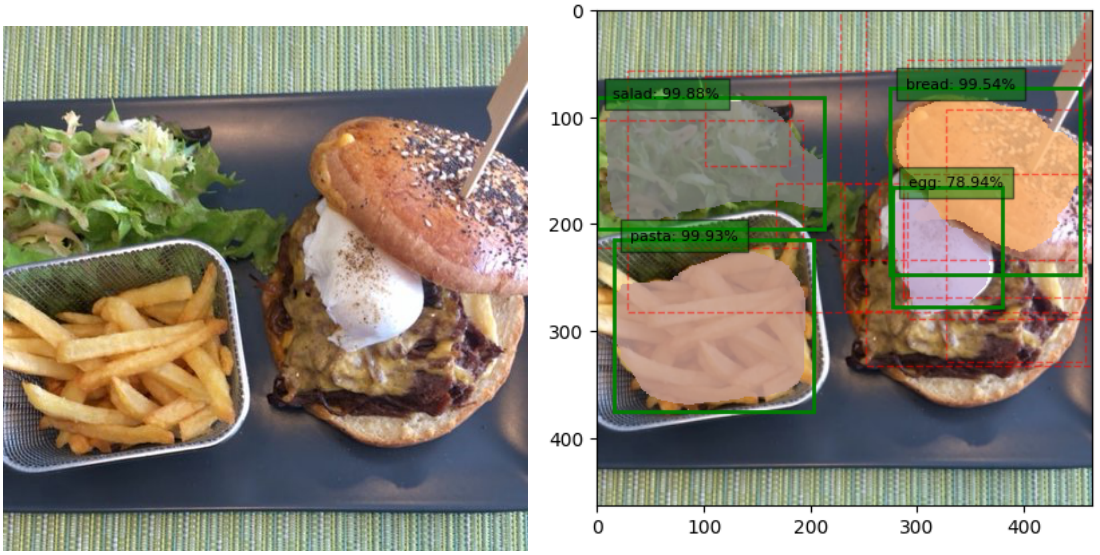
We noticed an overall qualitative improvement during the training epochs and also the quantitative results provided by the `COCOeval` tool (running the `food.py` script with the "`evaluate`" option) were nice. We reached a **0.498 Average Precision** and a **0.444 Average Recall**. These scores would be ranked in the top 10 for the Food Recognition challenge, following the official leaderboard. Of course, having worked with a modified dataset, we know that our results should be scaled down by some amount, but nonetheless this is a fine achievement. Our weights are available here.

We notice that the model has some problems dealing with very crowded photos, for example pictures of mixed salads or other complex dishes. Here's an example:



Also, our dataset manipulation removed some classes that are clearly present in some images, so the model often mistakens some objects as members of some similar (feature-wise) - but definitely

different classes. For instance, in the following case the model mistakes french fries for pasta, since “french fries” is not present in the dataset anymore. We understand the similarity between the two classes in this particular picture, but nonetheless the model has made a mistake.



In general, we recognize that the model could benefit from a better training schedule and that with more time and better experimental conditions we could have reached even better results. Furthermore, our training led the model to **slightly overfit** the training data. We observe this behaviour in the fact that - especially in the latest epochs - the training loss was slowly decreasing, but the validation loss remained stable.

Possible improvements and suggestions

Obviously there could be many ways of improving our results and our model, and here are some of our considerations:

- Since the capability of classification of the model is largely derived from the feature extraction part in the backbone, implementing a **larger** or **newer** model as **backbone** would have surely improved our results. We weren’t able to make these tests, as for many things in this list, because of our limited computational capability and time.
- The classification and mask heads play a big role in our network. The networks we used for the heads are quite simple and in our opinion **deeper networks** could lead to better results.
- One of the most complex parts of the network is surely the mask head, with a lot of pixels (28x28) per ROI to predict. The small size, however, does not help the accuracy of the masks, because ground truth masks have to be shrunk and rounded to be compared to the predicted ones. The bigger the size of the predicted mask, the less information we would lose by shrinking the groundtruth masks.
- Both the heads of the network start with a PyramidROIAlign layer: the classification head with a 7x7 layer, while the mask head with a 14x14 layer. Since their activity is similar, they could have been **merged into only one layer**, that would have been the start of the heads

instead. This could have reduced the number of parameters of the network and sped up training.
- As mentioned before, the training could have also benefited from improving and expanding the possibilities of **data augmentation** during the preprocessing step, which could reduce the chance of overfitting and create a more general model.
- Another possible change we could have explored is **pruning the dataset in a different way**: maybe changing the initial super categories could have brought better and more precise results. Moreover, if we had the possibility of implementing a bigger network, we could even revert to the initial categories and keep all of them.
- The losses are one of the most important parts of the program, and greatly influence the training of the network. We thought that instead of a simple sum between all the 5 losses, a **weighted sum** could bias the model towards the components in which it struggles the most. For instance, we observed that the bounding box regression and mask losses were more difficult to lower than the classification losses.
- One last suggestion would be, as always in ML, to **improve or expand the dataset**. Better labels and more pictures could greatly improve the training and could allow even bigger models with reduced risk of overfitting, and better results.

In general, all of the above points could be easily tested and evaluated with more computational power and time. The training of just our model took us more than a week, so to be able to try all different combinations we probably would have needed months. However, given our limited tools, we think we have achieved acceptable results, considering the size of the challenge.

Moreover, the whole structure of the model is devoted to **lightness** and **speed**, since many applications could reside on consumer level hardware or could need to process frames of a video. So, it is also important to state that our model, in inference mode, can process an image in **0.17 seconds**, or almost 6 images per second (on the machine described in the Training section). This would make it viable for real time applications.