Spacecraft detection and classification using Detectron2

Project Report - Alessandro Giffra

30.06.2023

1 Introduction

Computer vision is an interdisciplinary field of artificial intelligence (AI) in computer science, that deals with methods allowing computer to gain high-level understanding from digital images or videos. It seeks to replicate human intelligence and instincts to a computer (i.e., allow it to "see" and understand what it is seeing). Therefore, computer vision tasks include methods for acquiring, processing, analysing and understanding digital images, that can be used in a variety of different contexts and can be applied to several domains, from robot navigation to medical image analysis [1].

One of the fields in which computer vision and image analysis have become increasingly useful is aerospace. In the context of the SPARK 2022 challenge [2] organised by the Computer Vision Imaging & Machine Intelligence Research Group (CVI^2) at the University of Luxembourg, the goal of this project is to use a deep learning model in order to detect and classify spacecrafts. This task is very useful in aerospace in order to perform Space Situational Awareness (SSA), which consist in getting information and knowledge about space objects orbiting around the earth.

For this project, we will need to use object detection which is the computer vision technology that detects instances of objects from one or more classes in an image [3]. More specifically, it includes both object localisation (i.e. finding the object in the image), and object classification (i.e. determine to which category each object belongs to)[4]. Hence in the context of our project, we will locate where the spacecraft is in the image and identify its category among the eleven spacecraft types present in the provided dataset.

2 Related Work

There exists a lot of different methods for performing object detection, and as we saw in class, convolutional neural networks (CNNs) are very well suited for this task as they aim to mimic the natural visual perception mechanisms of living creatures, using convolution operations. Most of the object detection algorithms can be divided in two categories: (1) one-stage algorithms which predict bounding boxes and class labels in only one step (e.g., the YOLO (You Only Look Once) family first defined by Redmon et al. [5]), and (2) two-stage algorithms which localise and classify objects in a sequential manner: the region of interest (RoI) is first extracted, and then classified (e.g., the Region-based Convolutional Neural Network (R-CNN) family and in particular Fast R-CNN [6] and Faster R-CNN [7]).

The R-CNN approach usually consist of [8]:

- A region proposal algorithm that generates bounding boxes where objects are possibly located in the image. The accuracy of the model mainly depends on this algorithm.
- A feature generation step that gives features of the located objects.
- A classification layer, predicting to which class the objects belong to (i.e. what the objects are).
- A regression layer to improve the bounding boxes' precision.

Single-stage object detection algorithms are much faster than two-stage ones, as everything is done in only one step. However, they typically reach lower accuracy rates [9]. As for the challenge the priority is the accuracy and not the speed, we will use the most used two-stage algorithm of the R-CNN family: Faster R-CNN. The real latest member of the R-CNN family is Mask R-CNN [10] that extends Faster R-CNN for image segmentation purposes which is outside the scope of the project.

3 Materials and Methods

3.1 Dataset

The SPARK 2022 challenge provide a space annotated dataset [11] containing 110000 images (66000 for training, 22000 for validation and 22000 for testing) of eleven different types of space objects (ten classes of spacecrafts and one class of debris). The data have been generated under a realistic simulated space environment that accurately replicates various sensing conditions like extreme and challenging orbital scenarios, background noise and high image contrast. The different classes of objects are presented in Figure 1 below:

Class	Name	Index
Proba 2	`proba_2`	0
Cheops	`cheops`	1
Debris	`debris`	2
Double star	`double_star`	3
Earth Observation Sat 1	`earth_observation_sat_1`	4
Lisa Pathfinder	`lisa_pathfinder`	5
Proba 3 CSC	`proba_3_csc`	6
Proba 3 OCS	`proba_3_ocs`	7
Smart 1	`smart_1`	8
Soho	`soho`	9
Xmm Newton	`xmm_newton`	10

Figure 1: Classes of the objects contained in the dataset, together with their corresponding index [12].

Beside the 1024x1024 images, in the dataset we find the ground truth classes and bounding boxes as csv files (3 in total: for the training, validation and testing). In Figure 2 below you can find some example of the images that are included in the dataset (note that the bounding boxes are present only for illustration, the images used for the training won't include them).

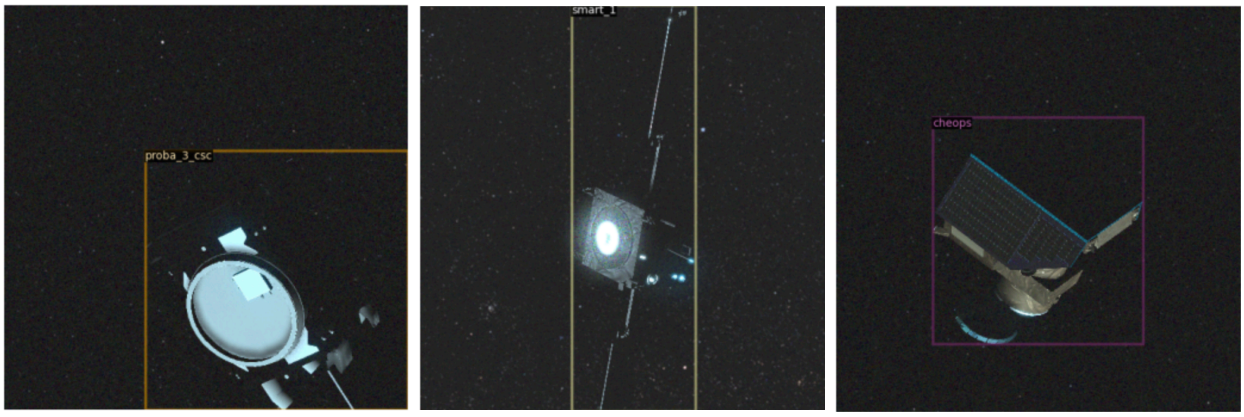


Figure 2: Examples of images in the dataset

3.2 Model and Framework

As stated in the *Related Work* section, for this project we will use the Faster R-CNN object detection model. Originally introduced in 2015, it extends Fast R-CNN (which makes use of a CPU-based region proposal algorithm) by adding an extra convolutional layer, the Region Proposal Network (RPN), that generates region proposals, before the actual Fast R-CNN object detector network is applied on those regions. These constitute the two modules on which Faster R-CNN's architecture relies on (see Figure 3 below).

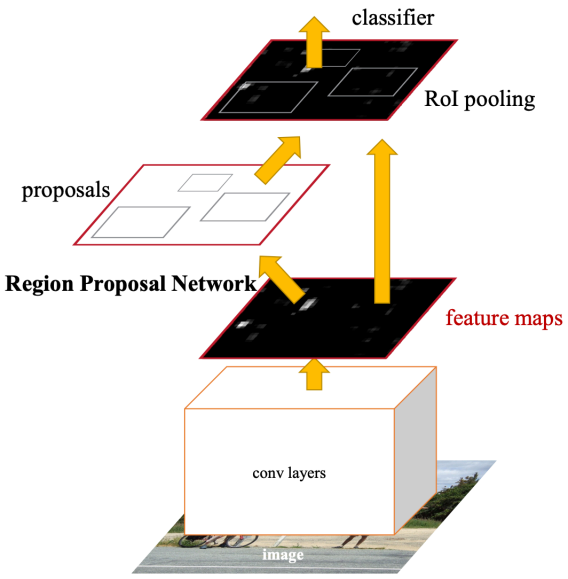


Figure 3: Architecture of Faster R-CNN [7].

Adding this additional convolutional layer compared to R-CNN and Fast R-CNN has the effect of improving computational efficiency, test time and performance.

In Figure 4 below we have the Faster R-CNN architecture.

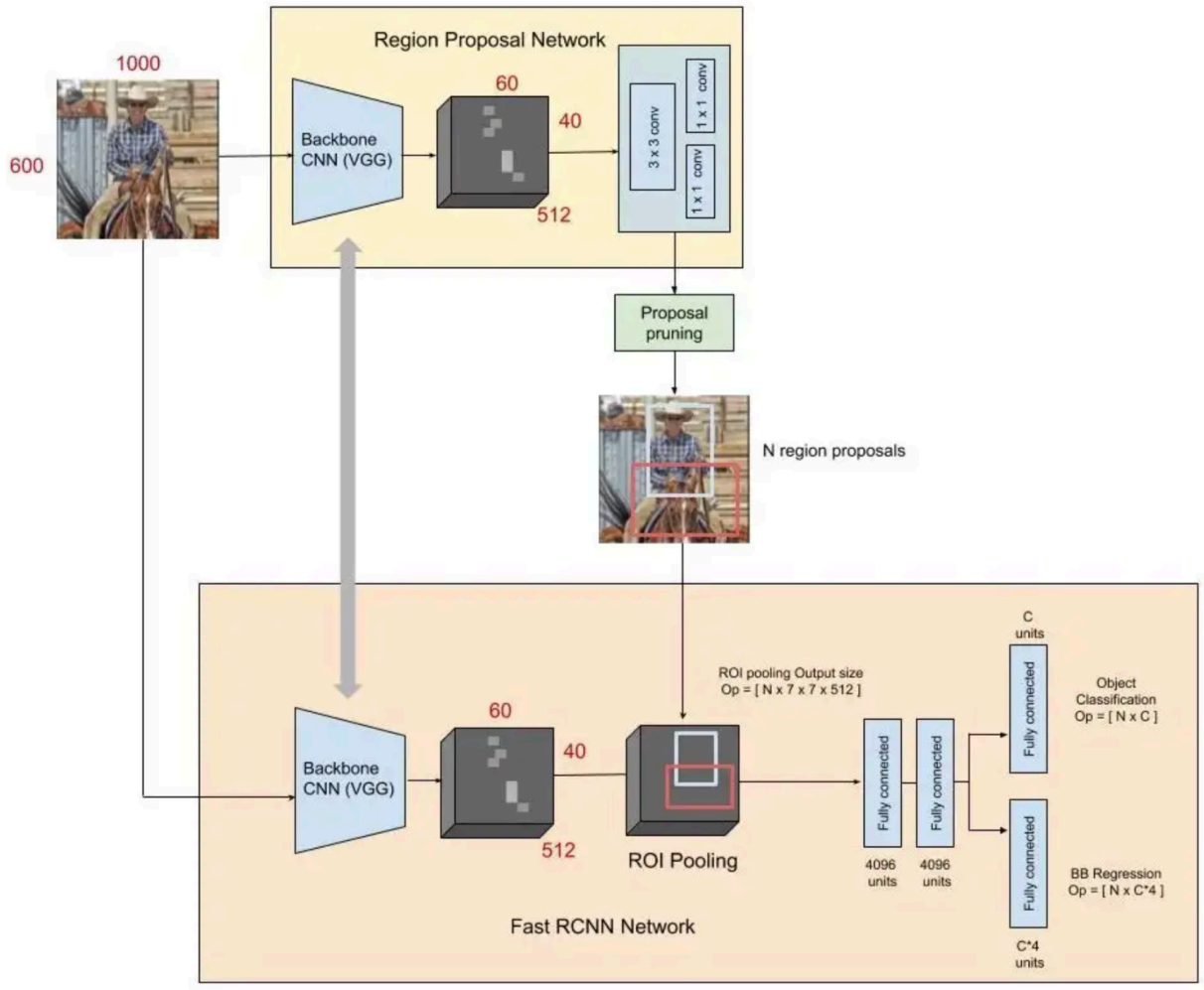


Figure 4: Faster R-CNN architecture, composed of the RPN for region proposals and Fast R-CNN as a detector [8].

The RPN takes an image as input, and outputs "region proposals" (i.e., rectangular bounding boxes that may contain objects) together with a score for each of them. This process is carried out using a fully convolutional network, sharing a common set of convolutional layers with the Fast R-CNN object detection network.

Given the previously generated set of region proposals, the Fast R-CNN model is then used as an object detector, consisting in a CNN backbone (like the RPN), a ROI pooling layer and a set of four fully connected layers, where two of them are split in two branches: one used for classification (returning the probability of an object belonging to each class) and the other for bounding box regression (to improve the accuracy of the bounding boxes) [8].

By looking for available Python frameworks using an implementation of Faster R-CNN, we ended up choosing the *Detectron2* framework. A Facebook AI Research (FAIR) group's open source object detection platform, implemented in PyTorch, which offer simple and flexible design [13]. It includes implementations of loads of object detection algorithms, including

the models from the R-CNN family and in particular our targeted Faster R-CNN.

To obtain our working implementation of Faster R-CNN using the Detectron2 framework, we based ourselves on the Google Colaboratory tutorial notebook provided by Facebook (available at `https://github.com/facebookresearch/detectron2`) to get started with the library. This notebook allowed us to have a first overview of the framework. After that we adapted the notebook for our need (i.e. perform spacecraft classification on the SPARK 2022 challenge dataset); but because of its big size (around 188GB), we made the choice to first use a smaller "dummy" dataset in the same format (i.e., file organisation, csv files, etc.) of the actual one. The goal was to have a working code and model that would only necessitate minor changes to be applied to the actual dataset later on.

3.3 Hardware

To download the dataset, run our code and perform the experiments in a reasonable time, a lot of computing power was needed. Hence, we used the University of Luxembourg's High Performance Computing (UL HPC) facility. Therefore, we created a Jupyter notebook to be run directly on the HPC and accessible from our local machine web browser. All the necessary libraries were installed inside a conda virtual environment to avoid conflicts between packages. To perform our experiments, we have used one node with a seven core CPU and a single NVIDIA Tesla V100 SXM2 16G [14] GPU on an Iris cluster.

3.4 Pre-porcessing

Since the official Detectron2 tutorial uses simple json COCO-formatted files as input, we transformed the csv files provided with the dataset, into COCO json files. The implemented Python functions can be found in Appendix 1.

Another thing we had to care about was to change the whole bounding box format (see Figure 5 below). It originally was under the form $[R_{\min}, C_{\min}, R_{\max}, C_{\max}]$ (with R referring to *row* and C to *column*) and we changed it to the $[x_{\min}, y_{\min}, x_{\max}, y_{\max}]$ format with $(0, 0)$ at the bottom left of the image. The implemented Python functions can be found in Appendix 2.

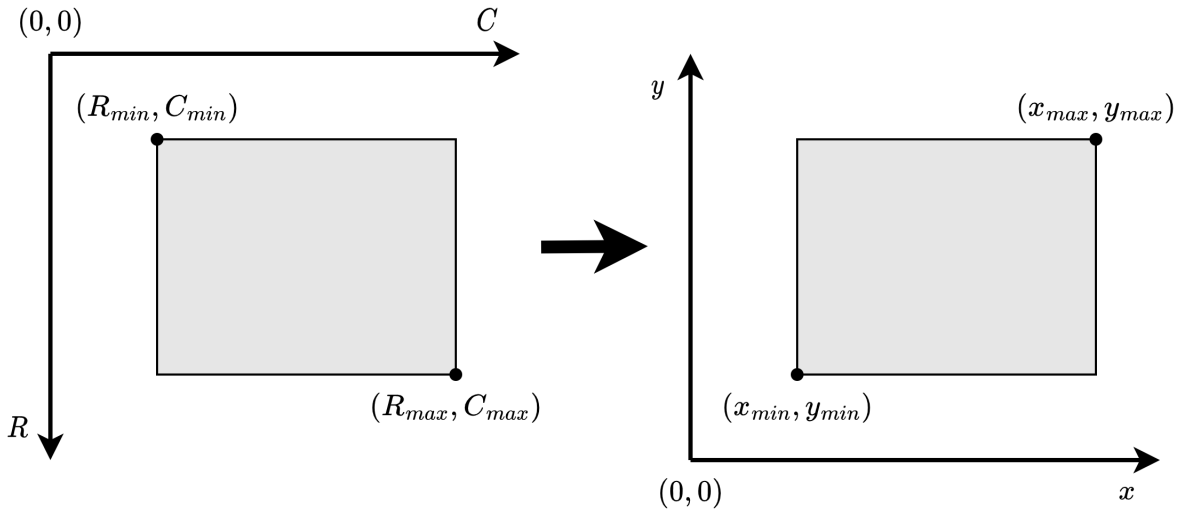


Figure 5: Transformation of bounding box coordinates, from original (left) to standard (right) format.

3.5 Hyperparameters and Metric

By default, Detectron2 does not evaluate the model's performance on the validation set during the training. To cope with that, we created a custom trainer class (`CocoTrainer()`, see Appendix 3), which extends the original `DefaultTrainer()` class. Moreover, since we are interested in monitoring how the training loss is evolving compared to the validation loss, we had to create an additional class (`LossEvalHook()`, see Appendix 4) where we compute the validation loss each time an evaluation is done (using `CocoTrainer()`).

With this done, we first loaded a pre-trained model included in Detectron2; we chose a Faster R-CNN model, trained on ImageNet using a ResNet network having 101 layer of depth, for its excellent speed-accuracy tradeoff). Then we fine-tuned the loaded model using the challenge dataset. The code snippet for this can be found in Appendix 5. The final hyperparameters values we used for our model are shown in the following Table 1.

Hyperparameter	Value
Batch size	8
Learning rate	0.001
Number of iterations	60000
Batch size per image	128
Evaluation period	20000

Table 1: Final hyperparameters values.

Note that the *Batch* size corresponds to the number of images that are propagated

through the network to train a single forward and backward pass. While, the *Batch size per image* is used to sample a subset of proposals coming from the RPN to compute the loss during the training.

The metric that we are considering to evaluate the performance of our model is the mean average precision (mAP or simply AP in Detectron2). It corresponds, for a set of detections, to the average over all classes of the AP for each class [15]. The AP takes into consideration three metrics: precision, recall and intersection over union (IoU). With those the precision-recall curve (PR curve) can be constructed by plotting precision values against recall values for different confidence thresholds (determined thanks to the IoU)[16]. Then, the AP corresponds to the area under this curve.

4 Results and Discussion

4.1 Results

To optimise the model and improve our results (i.e., to find the optimal hyperparameters values), we varied each hyperparameter one at a time. The first parameter that we modified was the learning rate (because it is one of the parameters that has the biggest influence on the AP), while the latest one was the number of iterations (as tuning this, followed by the other parameters would take too much time).

After many experiments, we figured out that the AP w.r.t. the validation set was still improving with the number of iterations (e.g., $\simeq$ 77.1 for 13,500 iterations, $\simeq$ 80.8 for 30,000 iterations and $\simeq$ 82.5 for 60,000 iterations). However, as the training performed using the values reported in Table 1 (section 3.5) took more or less 12 hours on the HPC (10 hours for the training and 2 hours for the evaluations on the validation set), we decided to stop there (due to time constraint).

In the Table 2 below you have a summary of our search for the optimal hyperparameters values (the first line representing the initial values).

BS	LR	Iter.	BS/image	Eval. period	AP
4	0.001	1,500	64	500	**51.901**
4	**0.01**	1500	64	500	**25.948**
4	**0.0001**	1500	64	500	**33.391**
2	0.001	1500	64	500	**36.877**
8	0.001	1500	64	500	**57.621**
8	0.001	1500	**32**	500	**57.088**
8	0.001	1500	**128**	500	**58.964**
8	0.001	1500	**256**	500	**56.602**
8	0.001	**4500**	128	1500	**63.369**
8	0.001	**13500**	128	4500	**77.144**
8	0.001	**30000**	128	10000	**80.822**
8	0.001	**60000**	128	20000	**82.499**

Table 2: Obtained results for some experimental settings. A green value indicates that the AP w.r.t. the validation set is improving (the higher, the better) and becomes the new reference. A red value indicates that the new AP is worse than before.

In the following Figure 6 we can have a more detailed view of the final results (in particular the AP for each class).

```
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.877
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.100
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.767
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.890
[12/23 09:53:09 d2.evaluation.coco_evaluation]: Evaluation results for bbox:
|   AP   |  AP50  |  AP75  |  APs  |  APm   |  APl   |
|:------:|:------:|:------:|:-----:|:------:|:------:|
| 82.499 | 98.465 | 93.085 | 5.050 | 68.505 | 84.092 |
[12/23 09:53:09 d2.evaluation.coco_evaluation]: Per-category bbox AP:
| category    | AP     | category                | AP     | category         | AP     |
|:------------|:-------|:------------------------|:-------|:-----------------|:-------|
| None        | nan    | cheops                  | 87.689 | debris           | 84.953 |
| double_star | 82.020 | earth_observation_sat_1 | 72.778 | lisa_pathfinder  | 87.936 |
| proba_2     | 87.987 | proba_3_csc             | 81.934 | proba_3_ocs      | 80.269 |
| smart_1     | 84.822 | soho                    | 82.595 | xmm_newton       | 74.510 |
[12/23 09:53:09 d2.engine.defaults]: Evaluation results for spark1_val in csv format:
[12/23 09:53:09 d2.evaluation.testing]: copypaste: Task: bbox
[12/23 09:53:09 d2.evaluation.testing]: copypaste: AP,AP50,AP75,APs,APm,APl
[12/23 09:53:09 d2.evaluation.testing]: copypaste: 82.4993,98.4651,93.0854,5.0495,68.5054,84.0921
```

Figure 6: Detailed results. Here we have an AP (that for Detectron2 corresponds to mAP) of 82.499.

And in the following Figure 7 we have the corresponding loss curve showing that we can still increase the number of iterations.

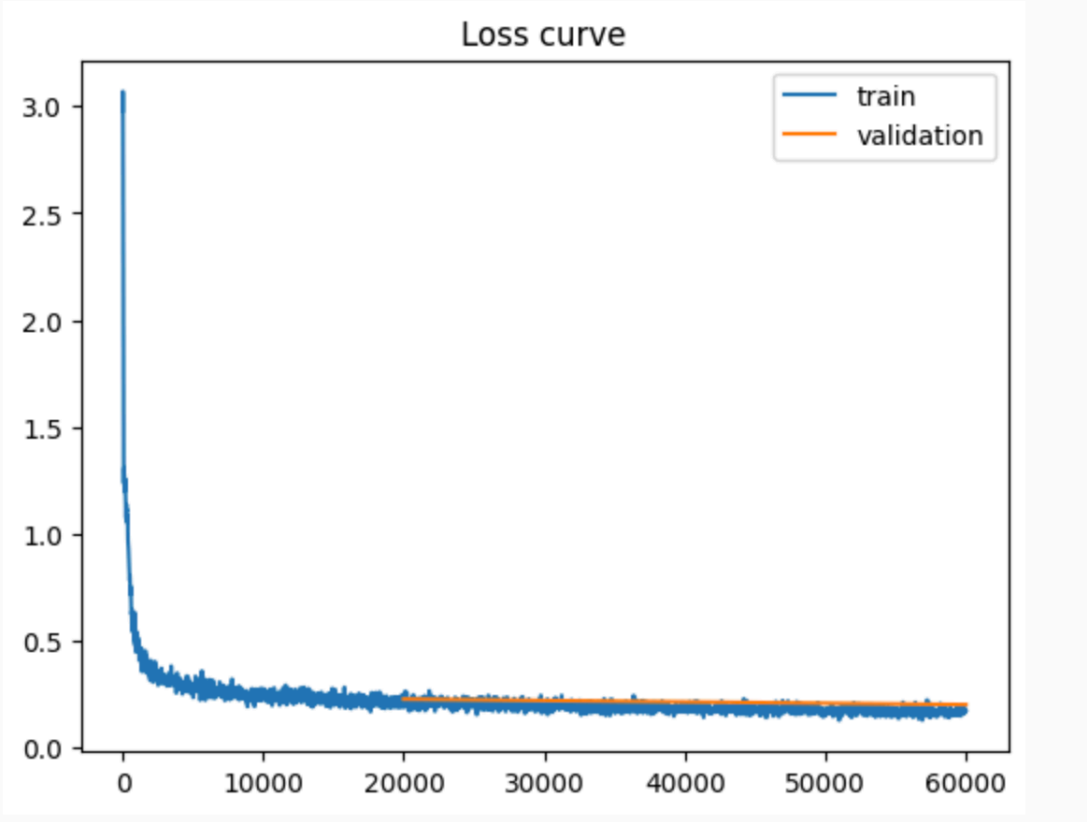
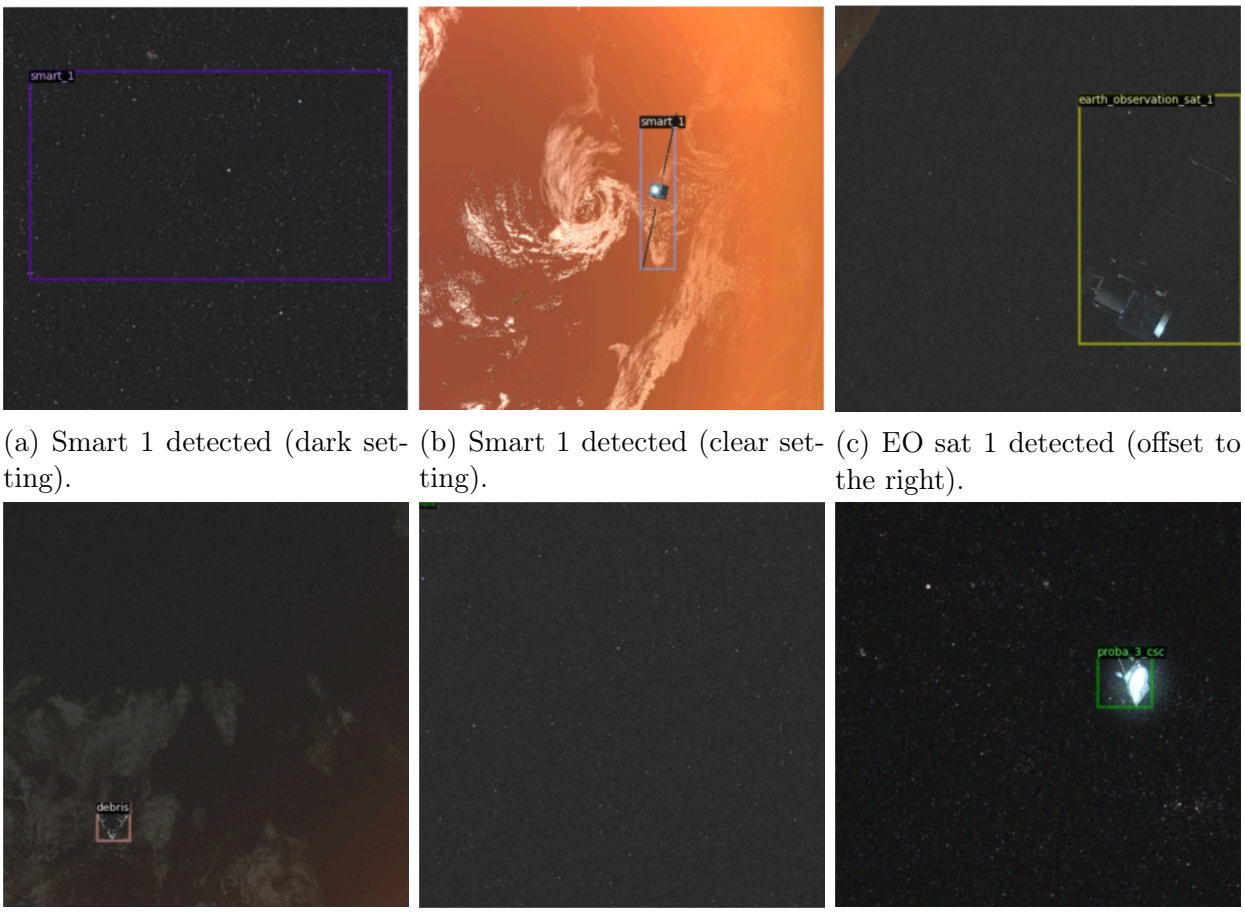


Figure 7: Loss curve.

By visualising some predictions of our model on the test set, we observed interesting insights. First, we saw that our model returned, for the most part, pretty good and overall accurate predictions (even for really challenging situations). Those include particularly dark environments (e.g., Figure 8a), small objects (e.g., Figure 8d), contrasting lighting, or objects that seem to visually be undistinguishable, or at least very difficult to distinguish from the background. It is also worth noting that the model could detect objects in situations where our eyes could not; and only 59 images (over 22000) had no predictions at all. Figure 8 provides some detected objects over the testing image set (where we can also see an image on which no prediction has been made, Figure 8e).



(a) Smart 1 detected (dark setting).

(b) Smart 1 detected (clear setting).

(c) EO sat 1 detected (offset to the right).

(d) Small debris detected.

(e) No object detected.

(f) Proba 3 CSC detected.

Figure 8: Examples of detected objects on the testing set of images.

5 Conclusion

In conclusion, using the Detectron2 framework which include an implementation of the Faster R-CNN object detection model, we successfully detected, localised and classified spacecraft objects in space images showing different scenarios. The model had good results and a satisfying accuracy even for very challenging situations, including where the object spills out of the image, is really small, blent into the background, or barely visible for a human eye. However, even if we are satisfied with what we have done, the task was far from being easy. Indeed, we encountered some issues with the install of Detectron2, the other needed libraries and for making a Jupyter notebook work on the HPC. In addition, Detectron2 has been quite complicated to adapt for our purpose, which makes at the end it less flexible than what we intended it to be. Lastly, even though we obtained a satisfactory accuracy, this is by no means perfect and there is still room for improvement. Therefore, we can assume that it is possible to obtain a better accuracy by training the model further and hence continuing to increase the number of iterations (not done as training the model can be really time consuming), or by doing data augmentation which is what could be done as a future work.

References

[1] D. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*, 2nd ed. Boston: Pearson, 2012.

[2] "SPARK2022 – CVI^2," https://cvi2.uni.lu/spark2022/.

[3] Y. Amit, P. Felzenszwalb, and R. Girshick, "Object Detection," in *Computer Vision: A Reference Guide*. Cham: Springer International Publishing, 2020, pp. 1–9.

[4] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, "Object Detection with Deep Learning: A Review," Apr. 2019.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," May 2016.

[6] R. Girshick, "Fast R-CNN," Sep. 2015.

[7] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," Jan. 2016.

[8] S. Ananth, "Faster R-CNN for object detection," https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46, Aug. 2019.

[9] P. Soviany and R. T. Ionescu, "Optimizing the Trade-off between Single-Stage and Two-Stage Object Detectors using Image Difficulty Prediction," Aug. 2018.

[10] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask R-CNN," Jan. 2018.

[11] "SPARK 2022 Dataset – CVI^2," https://cvi2.uni.lu/spark-2022-dataset/.

[12] "SPARK Challenge / SPARK 2022 Utils · GitLab," https://gitlab.uni.lu/spark-challenge/2022-utils.

[13] "Detectron2: A PyTorch-based modular object detection library," https://ai.facebook.com/blog/-detectron2-a-pytorch-based-modular-object-detection-library-/, Mar. 2019.

[14] "The Iris Cluster - HPC @ Uni.lu," https://hpc.uni.lu/old/systems/iris/.

[15] P. Henderson and V. Ferrari, "End-to-end training of object class detectors for mean average precision," Mar. 2017, comment: This version has minor additions to results (ablation study) and discussion.

[16] A. Anwar, "What is Average Precision in Object Detection & Localization Algorithms and how to calculate it?" https://towardsdatascience.com/what-is-average-precision-in-object-detection-localization-algorithms-and-how-to-calculate-it-3f330efe697b, May 2022.

Appendix

1.

```python
def convert_csv_to_coco(set_type):
  path = set_type + '_reformatted.csv'
  save_json_path = set_type + '_coco.json'

  data = pd.read_csv(path)

  images = []
  categories = []
  annotations = []

  category = {}
  category["supercategory"] = 'none'
  category["id"] = 0
  category["name"] = 'None'
  categories.append(category)

  data['fileid'] = data['filename'].astype('category').cat.codes
  data['categoryid']= pd.Categorical(data['class'],ordered= True).codes
  data['categoryid'] = data['categoryid']+1
  data['annid'] = data.index

  def image(row):
    image = {}
    image["height"] = row.height
    image["width"] = row.width
    image["id"] = row.fileid
    image["file_name"] = row.filename
    return image

  def category(row):
    category = {}
    category["supercategory"] = 'None'
    category["id"] = row.categoryid
    category["name"] = row[3]
    return category

  def annotation(row):
    annotation = {}
    area = (row.xmax-row.xmin)*(row.ymax-row.ymin)
    annotation["segmentation"] = []
    annotation["iscrowd"] = 0
    annotation["area"] = area
    annotation["image_id"] = row.fileid

    annotation["bbox"] = [row.xmin, row.ymin, row.xmax-row.xmin, row.ymax-row.ymin]

    annotation["category_id"] = row.categoryid
    annotation["id"] = row.annid
    return annotation

  for row in data.itertuples():
    annotations.append(annotation(row))

  imagedf = data.drop_duplicates(subset=['fileid']).sort_values(by='fileid')
  for row in imagedf.itertuples():
    images.append(image(row))

  catdf = data.drop_duplicates(subset=['categoryid']).sort_values(by='categoryid')
  for row in catdf.itertuples():
    categories.append(category(row))

  data_coco = {}
  data_coco["images"] = images
  data_coco["categories"] = categories
  data_coco["annotations"] = annotations

  json.dump(data_coco, open(save_json_path, "w"), indent=4)
```

2.

```python
def reformat_csv(set_type):
  IMG_PATH = set_type + '/'

  HEIGHT = 1024
  WIDTH = 1024

  # divide bbox column into 4 columns [xmin, ymin, xmax, ymax]
  df = pd.read_csv(set_type + '.csv')

  for index, row in tqdm(df.iterrows(), total=len(df)):
    splitted_row = row['bbox'].split(', ')
    df.loc[index, 'width'] = WIDTH
    df.loc[index, 'height'] = HEIGHT

    row_min = splitted_row[0][1:]
    column_min = splitted_row[1]
    row_max = splitted_row[2]
    column_max = splitted_row[3][:-1]

    df.loc[index, 'xmin'] = column_min
    df.loc[index, 'ymin'] = row_min
    df.loc[index, 'xmax'] = column_max
    df.loc[index, 'ymax'] = row_max

  df.pop('bbox')

  df.to_csv(set_type + '_reformatted.csv')
```

3.

```python
class CocoTrainer(DefaultTrainer):

  @classmethod
  def build_evaluator(cls, cfg, dataset_name, output_folder=None):
    if output_folder is None:
      output_folder = os.path.join(cfg.OUTPUT_DIR, "inference")

    return COCOEvaluator(dataset_name, cfg, True, output_folder)

  def build_hooks(self):
    hooks = super().build_hooks()
    hooks.insert(-1,LossEvalHook(
        cfg.TEST.EVAL_PERIOD,
        self.model,
        build_detection_test_loader(
            self.cfg,
            self.cfg.DATASETS.TEST[0],
            DatasetMapper(self.cfg,True)
        )
    ))
    # swap the order of PeriodicWriter and ValidationLoss
    # code hangs with #GPUs > 1 if this line is removed
    #hooks = hooks[:-2] + hooks[-2:][::-1]
    return hooks
```

4.

```python
class LossEvalHook(HookBase):

  def __init__(self, eval_period, model, data_loader):
    self._model = model
    self._period = eval_period
    self._data_loader = data_loader

  def _do_loss_eval(self):
    # Copying inference_on_dataset from evaluator.py
    total = len(self._data_loader)
    num_warmup = min(5, total - 1)

    start_time = time.perf_counter()
    total_compute_time = 0
    losses = []

    for idx, inputs in enumerate(self._data_loader):
      if idx == num_warmup:
        start_time = time.perf_counter()
        total_compute_time = 0

      start_compute_time = time.perf_counter()

      if torch.cuda.is_available():
        torch.cuda.synchronize()

      total_compute_time += time.perf_counter() - start_compute_time
      iters_after_start = idx + 1 - num_warmup * int(idx >= num_warmup)
      seconds_per_img = total_compute_time / iters_after_start

      if idx >= num_warmup * 2 or seconds_per_img > 5:
        total_seconds_per_img = (time.perf_counter() - start_time) / iters_after_start
        eta = datetime.timedelta(seconds=int(total_seconds_per_img * (total - idx - 1)))
        log_every_n_seconds(
            logging.INFO,
            "Loss on Validation  done {}/{}. {:.4f} s / img. ETA={}".format(
                idx + 1, total, seconds_per_img, str(eta)
            ),
            n=5,
        )

      loss_batch = self._get_loss(inputs)
      losses.append(loss_batch)

    mean_loss = np.mean(losses)
    self.trainer.storage.put_scalar('validation_loss', mean_loss)
    comm.synchronize()

    return losses

  def _get_loss(self, data):
    # How loss is calculated on train_loop
    metrics_dict = self._model(data)
    metrics_dict = {
        k: v.detach().cpu().item() if isinstance(v, torch.Tensor) else float(v)
        for k, v in metrics_dict.items()
    }
    total_losses_reduced = sum(loss for loss in metrics_dict.values())

    return total_losses_reduced

  def after_step(self):
    next_iter = self.trainer.iter + 1
    is_final = next_iter == self.trainer.max_iter

    if is_final or (self._period > 0 and next_iter % self._period == 0):
      self._do_loss_eval()

    self.trainer.storage.put_scalars(timetest=12)
```

5.

```
cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml"))

cfg.DATASETS.TRAIN = ("spark1_train",)
cfg.DATASETS.TEST = ("spark1_val",)
cfg.DATALOADER.NUM_WORKERS = 8 #2
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml")  # Let training initialize from model zoo

# each GPU "sees": (cfg.SOLVER.IMS_PER_BATCH / # of GPUs)
cfg.SOLVER.IMS_PER_BATCH = 8  #2 # This is the real "batch size" commonly known to deep learning people
cfg.SOLVER.BASE_LR = 0.001  # 0.00025 # pick a good LR
#cfg.SOLVER.WARMUP_ITERS = 200 #500
cfg.SOLVER.MAX_ITER = 60000 #1500 # adjust up if val mAP is still rising, adjust down if overfit
# The iteration number to decrease learning rate by GAMMA.
#cfg.SOLVER.STEPS = (400,) #(1000,) # optional
#cfg.SOLVER.GAMMA = 0.05 # optional

cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 128 #128  # The "RoIHead batch size". (default: 512)
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 12 #4 (see https://detectron2.readthedocs.io/tutorials/datasets.html#update-the-config-for-new-datasets)
cfg.TEST.EVAL_PERIOD = 20000 #100 #500

os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = CocoTrainer(cfg)
#trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```