Detection Studio: A tool for evaluation of object detection neural networks

Vinay Sharma, *Member, IEEE,* Francisco Rivas, *Fellow, OSA,* Sergio Paniego, *Member, IEEE,* and José M. Cañas, *Fellow, OSA*

Abstract

This paper presents Detection Studio, an open source tool for evaluation of deep learning neural network models for object detection on images. The evaluation provides the common objective performance metrics like mean average precision and mean inference time. Some of the most famous object detection datasets are supported and additionally the most used deep learning frameworks are also supported. Models with different architectures or programmed using different middlewares can be compared.

A set of tools is provided to work with and manipulate datasets and the different models. These tools include the visualization of the dataset or its conversion between the different formats available. Additionally, new detection datasets can be created using Detection Studio from videos and webcams. The application can be accessed via command line or via graphical interface.

***Index Terms*—Object detection, open source, software tools, model evaluation.**

I. Introduction

THERE is an increase in application areas for deep learning networks. One of those areas is the detection of objects on images using computer vision, which thanks to the great variety of free access quality datasets focused on this topic presented in the past years, has experimented a great development. This has also make it possible to develop deep learning networks models with an outstanding performance that have became famous overtime. These object detection datasets and deep neural networks models are improved constantly, reaching better performances each time and showing the current importance of this problem in research and industry.

The typical workflow of developing a deep neural network involves some trial and error, changing some parameters in the network and retraining it again and again to improve its performance. This situation brings the problem of how to compare objectively the performance of those trained neural networks models so a developer can quickly understand which model is better. The tool presented in this paper, Detection Studio, tries to help developers testing object detection neural networks using objective performance metrics so the user can easily compare different networks and see which one performs better. A set of different tools are presented inside Detection Studio, giving each one of them some important feature to help comparing objectively different neural networks models for object detection. Accessible via: https://github.com/JdeRobot/DetectionStudio

II. State of the art

In this section, a review of the state of the art in object detection with deep learning is conducted.

A. Datasets

The emergence of free access image datasets along with the competitions that many of them are related to have boosted the fast pace development in object detection and computer vision in general. Some of them, supported in Detection Studio, are the following:

- **ImageNet** [1]**:** ImageNet is the largest public collection of images, containing 14,197,122 images, where 1,034,908 images have been annotated with bounding boxes, ideal for training and evaluating object detection models.
- **Pascal VOC** [2] Pascal VOC's 2012 release contains 11,530 images in training and validation datasets, spanning accross 20 classes. These have contain a total of 27,450 bounding box annotated objects.
- **Princeton RGB dataset** [3] This dataset contains 100 RGB-D videos of high diversity meant to design and compare various tracking algorithms. It is similar to Spinello and also uses a Depth Sensor to capture images.
- **Spinello dataset** [4] Spinello is a dataset containing 3000+ RGB-D images captured using a Microsoft Kinect, containing people and is meant for person detection and tracking in 3D space.
- **COCO** [5] COCO (Common Objects in Context) is designed for both object detection and segmentation. It contains around 330,000 images of which 200,000 are labelled containing 1.5 million object instances in total.

B. Frameworks

The object detection models are implemented using different deep learning frameworks instead of building everything from scratch. They have common deep learning pre-built components, ready to use and optimized for performance, making it easier for a programmer to built the model. Since there is a limited range of deep learning frameworks, this situation makes it easier for other researchers or programmers to understand the models, considering that they use a common and known framework. Some of the most common deep learning frameworks are the following, which all are supported in Detection Studio:

- TensorFlow [6].
- Caffe [7].
- Darknet [8].
- Keras [9].
- Pytorch [10].

C. Performance metrics

Evaluating the performance of the different approaches in this field is key. Differences between models are nowadays small, so a set of performance metrics is needed to determine which model outperforms. The performance of a model also depends on its purpose, so a model with a slightly worse performance but faster than the rest, could be more useful in a real time scenario. These are the metrics that Detection Studio considers when evaluating a model:

- Average Precision (AP). Fraction of the total amount of detections retrieves that are correct. Range from 0 to 1.
- Average Recall (AR). Fraction of the total amount of detections that are actually detected. Range from 0 to 1.
- Mean average precision (mAP). A range of Intersection over Union (IoU) values is consider, usually from 0.5 to 0.95. The IoU metric compares the ground truth bounding box with the detected ground truth and retrieves a value between 0 and 1 indicating how close the detected BB is to the ground truth. The higher the value of IoU the closes to the ground truth BB. Here the mean AP for IoU values from 0.5 to 0.95 is calculated (IoU=0.5:0.95).
- Mean inference time. In milliseconds, the importance of this metrics depends on the scenario where the model is applied. If the scenario implies giving fast and precise answers, this metric is key.

D. Network models

There are two main groups in the classification of object detection network models, Region Proposal-Based Framework and Regression/Classification-Based Framework, as divided in [11]. In the first one, a chain of correlated steps is conducted. These differentiated steps usually lead to a bottleneck in real-time. In the second group, the techniques are based on regression and only involve one global step, improving the computation time.

A brief explanation of three of the most successful network models is introduced, having Faster Regional-CNN (Faster R-CNN) in the Region Proposal-Based Framework group and Single Shot MultiBox Detector (SSD) and You Only Look Once (YOLO) in the Regression/Classification-Based Framework group.

1) Faster Regional-CNN [12]: Faster R-CNN is a multi-component detector comprising of a Region Proposal Network (RPN) which generates highly probable regions and are later fed into further layers which classify this region and a bounding box regressor which reduces the localization error in predicting bounding boxes. Faster R-CNN is an improvement upon R-CNN and Fast R-CNN to make it more real time and robust. Faster R-CNN also generates much less region proposals as compared to R-CNN and Fast-RCNN leading to reduced detection time while simultaneously maintaining detection accuracy.

Stages in Faster R-CNN:

a) Anchor generation: Anchors are basically regions which may contain a class. So, anchor generation must be as through as possible, because if a particular region is mixed then there is no way that it would be detected in the succeeding layers. These anchors are later refined using bounding box regressor to reduce the localization error, in order to better localize errors. Anchor Generation uses sophisticated algorithms to cover all of the image, like selective search which is later fed into the Region Proposal Network (RPN).

b) Region Proposal Network: The job of this component of the network is to output regions with high probability of containing objects. It takes anchors as input and outputs highly probable regions. Again, if a region containing an object isn't proposed then there is no way that it would be detected in the succeeding layers. Also, number of regions should be as low as possible so as to reduce detection time and as thorough as possible so as to reduce false negatives.

c) Classifier and bounding box regressor: The final component of the network classifies proposals from RPN into an object class or background i.e negative or no object present. Classification occurs first and then it's results are better localized to reduce the localization error or to accurately place the bounding box on the object being classified.
This regressor basically regresses 4 parameters, namely x, y, w and h, where x and y are the top left coordinates and w and h are the width and height of the bounding box.

2) Single Shot MultiBox Detector [13]: SSD proposes a more unified approach towards object detection as compared to Faster R-CNN in which detections can be generated in a single forward propagation of a unified network. It uses different techniques to propose regions and the whole working including Region Proposal, classification, bounding box regressor (for reducing localization loss) is part of a single unified network, which significantly increases it's detection speed.

The given image is divided in grids which can be 8x8 or 4x4. For each box present in the grid, SSD predicts the offsets for the bounding box present in that grid, and the confidence score (probability of each class in the particular region) for all object categories. Then each box is matched to a single class, and the final results are used to compute loss, including both classification and localization loss.

3) You Only Look Once (YOLO) [14]: This network was first proposed by Joseph Redmon and team, a PhD student at University of Washington, and was inspired by the idea of SSD's Unified Detection. It also uses a similar system as SSD to propose regions for further classification and regression.

Figure 2 describes in detail how regions are proposed for further classification and regression.

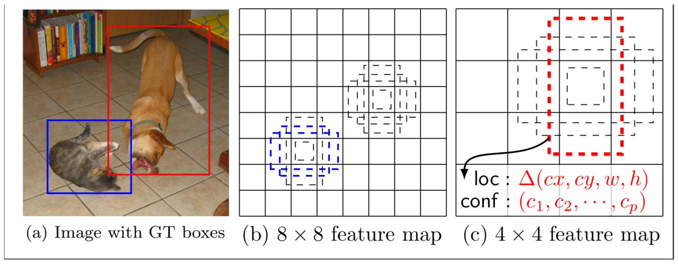

loc : $\Delta(cx, cy, w, h)$
conf : $(c_1, c_2, \cdots, c_p)$
(a) Image with GT boxes
(b) 8×8 feature map
(c) 4×4 feature map

Fig. 1. SSD uses 8x8 and 4x4 feature maps, in which each grid is a possible region and is used by the classifier, to classify probabilities of all the possible classes.

Similar to SSD, YOLO also divides the input image in a grid, and the grid size is variable i.e it depends on the dataset and the type of problem it's being used for. Let's assume a grid size of SxS, and for each grid cell B bounding boxes are generated, where B is also a variable and depends on the dataset and the type of problem it is being used for. For instance, $B = 2$ for Pascal VOC dataset. After generation, these bounding boxes are sent for classification and regression, to output final bounding boxes. Additionally, YOLO also has a very creative loss function, which takes care of both the classification and localization error, i.e a single combined loss function is used to minimize both classification and localization error. The prime selling point of YOLO was real time detection which was made possible by using a unified network. The accuracy for YOLO is great, in certain cases lower than other network models but the speed is high, making it possible for real time use.

From this initial release, several improvements have been applied to YOLO until the last version available, YOLOv4 [15]. For example in YOLOv3 [16], bounding boxes using dimension clusters as anchor boxes are included , proposed in [17] or an hybrid feature extractor approach between YOLOv2 and residual networks. YOLOv4 follows the same idea of incremental improvements taking different ideas that have been proven to work and making YOLO more efficient.

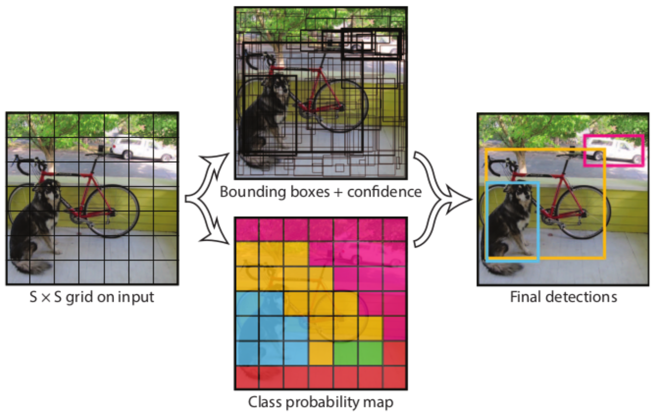
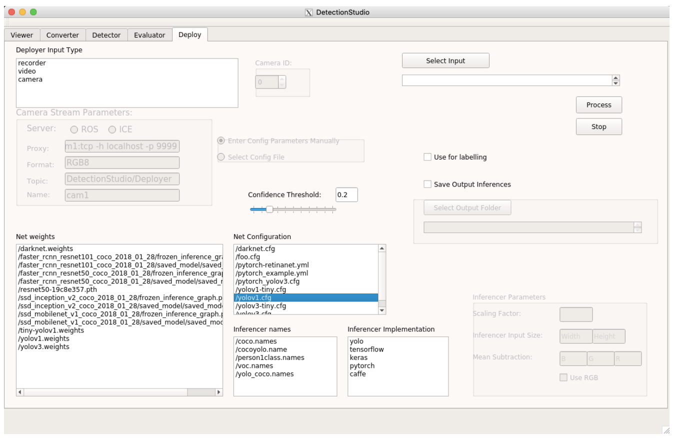


Fig. 2. Yolo uses a SxS grid in which B bounding boxes are predicted for each grid cell.

III. Detection Studio tool

Detection Studio is a multi-platform graphical user interface (GUI) and command line application that provides several tools to compare deep learning neural network models for object detection images. Its GUI is based on Qt (see Figure 3) and written in C++. The application is both supported in Linux and macOS. The tool accepts neural networks models trained in different frameworks. These supported frameworks are TensorFlow, Keras, Caffe, Pytorch and Darknet, which need to be installed in order to be used in the platfor. This feature provides a broader spectre of applicability for the application. Additionally, Detection Studio provides support for a wide variety of dataset formats, having YOLO, COCO, ImageNet and Pascal VOC supported.

The general application architecture, where all the different functionality is offered, is divided in two separated groups (see Figure 4). In the first groups, the general workflow tools are provided. In the second group, additional tool that can be used in parallel to the main ones are found. In the figure, this two groups are shown. The main workflow group in the connected part drawn on the diagram on the left hand side. The additional tools are unconnected from the main core of the application, displayed on the right hand side, providing another functionality that is not used in the main workflow.



Deployer Input Type
recorder
video
camera

Camera ID: 0

Camera Stream Parameters:
Server: ○ ROS ○ ICE
Proxy: m1:tcp -h localhost -p 9999
Format: RGB8
Topic: DetectionStudio/Deployer
Name: cam1

[✓ Enter Config Parameters Manually]
Select Config File

☐ Use for labelling
☐ Save Output Inferences
☐ Use RGB

Confidence Threshold: 0.2

Net weights
/darknet.weights
/faster_rcnn_resnet101_coco_2018_01_28/frozen_inference_gra
/faster_rcnn_resnet101_coco_2018_01_28/saved_model/saved_
/faster_rcnn_resnet50_coco_2018_01_28/frozen_inference_grap
/faster_rcnn_resnet50_coco_2018_01_28/saved_model/saved_r
/resnet50-19c8e357.pth
/ssd_inception_v2_coco_2018_01_28/frozen_inference_graph.p
/ssd_inception_v2_coco_2018_01_28/saved_model/saved_mode
/ssd_mobilenet_v1_coco_2018_01_28/frozen_inference_graph.p
/ssd_mobilenet_v1_coco_2018_01_28/saved_model/saved_mod
/tiny-yolov1.weights
/yolov1.weights
/yolov3.weights

Net Configuration
/darknet.cfg
/foo.cfg
/pytorch-retinanet.yml
/pytorch_example.yml
/pytorch_yolov3.cfg
/yolov1-tiny.cfg
/yolov1.cfg
/yolov3-tiny.cfg

Inferencer names
/coco.names
/cocoyolo.name
/person1class.names
/voc.names
/yolo_coco.names

Inferencer Implementation
yolo
tensorflow
keras
pytorch
caffe

Fig. 3. Detection Studio GUI.

The core Detection Studio workflow contains four different tools: Viewer, Detector, Evaluator and Deployer. These tools shape the connected part of the architecture because they can be used combined by a user comparing different neural networks models for object detection and are part of what could be a common workflow when using the application. This workflow would include as an example, running Detector over different networks to generate the detection datasets and then running Evaluator to evaluate the different generated detection datasets. The different tools presented below are provided in the GUI as separated tabs and are completely independent when used, giving flexibility to the different users possible use cases. This philosophy is shared in the whole project, providing each functionality as a different and independent tool.

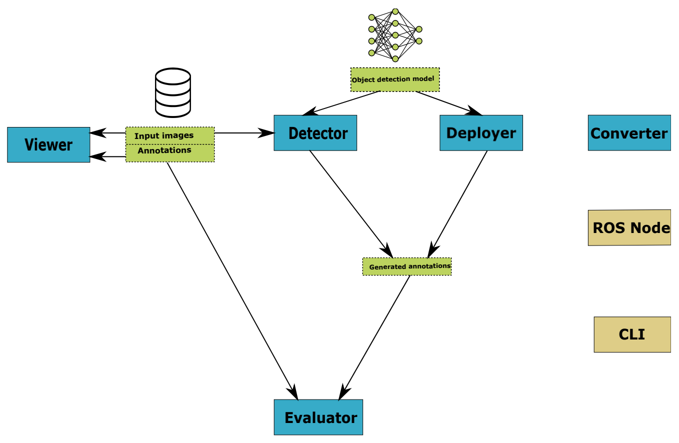


Fig. 4. General Detection Studio architecture.

In the first group, where the connected part is found, the tools are the following:

A. Viewer

Viewer is used to view annotated datasets. It reads the input images and the annotations files and it displays the images with the objects highlighted with their class name on it. This tool supports several dataset implementations like COCO, Imagenet, Pascal VOC, Princeton or Spinello. It also supports displaying and labelling depth images (for the datasets that give support to this feature) by converting them into a human readable depth map.

Images are displayed one by one, showing the image with its corresponding detected objects with a bounding box and a tag label naming the class group it belongs to. This bounding box and label has a different color depending on its detection. The final annotated images that Viewer displays can be further filtered based on some specific classes i.e. only particular classes will be labelled and only images containing those specific classes will be displayed. This option can be interesting when looking for images that contains objects belonging to a specific class.

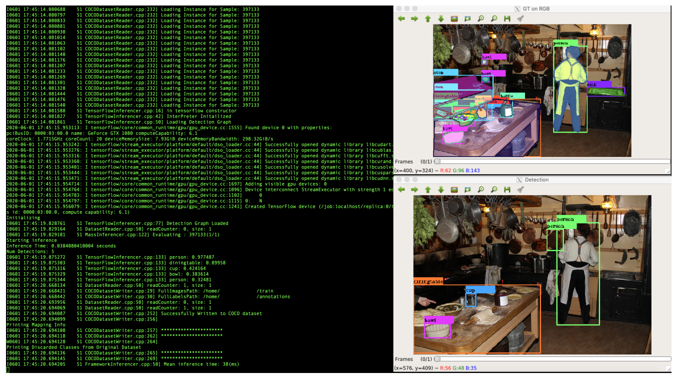
This functionality requires an input dataset, the object's class names supported and the dataset implementation type as input. It can be observed in the Figure 4 that this functionality is connected to a dataset (input images and annotations).

This input information is captured by the tool and then a dataset reader is created depending on the specific dataset provided. It reads the whole images dataset and objects annotations and displays each image with the annotations on it. The user can then navigate the whole dataset using the keyboard.

B. Detector

This tool provides the ability to create a new annotated dataset with generated labels using different neural networks models. It takes a dataset of images from a specific dataset implementation and a specific neural network as input and then generates a annotated detection dataset with the generated object labels for every image in the dataset provided. This generated annotated dataset contains the images with the detected objects, their position in the image and probabilities for those predictions. Detector takes as input parameters the given neural network weights, the inferencer implementation (TensorFlow, Keras, Darknet, Caffe or PyTorch), the network configuration in case it is needed by the specific framework and the object class names in order to generate the detection. Support for detection over depth images is also provided for the datasets supporting this feature.

The detection is run over the full input dataset and the output is written into a specific selected folder, provided before starting running the tool, where the annotations are written in a JSON format file. Two windows are displayed while running this tool (see Figure 6), one with the image and groud truth annotations and and another one with the same image and the objects detected inside bounding boxes with their class written over it.



```
I0601 17:45:14.080688    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.080797    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.080833    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.080881    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.080930    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081014    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081063    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081102    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081140    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081176    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081207    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081233    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081269    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081301    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081328    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081444    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081476    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081540    51 COCODatasetReader.cpp:232] Loading Instance for Sample: 397133
I0601 17:45:14.081588    51 TensorFlowInferencer.cpp:16] in tensorflow constructor
I0601 17:45:14.081827    51 TensorFlowInferencer.cpp:42] InterPreter Initailized
I0601 17:45:14.081861    51 TensorFlowInferencer.cpp:50] Loading Detection Graph
2020-06-01 17:45:15.953113: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1555] Found device 0 with properties:
pciBusID: 0000:03:00.0 name: GeForce GTX 1080 computeCapability: 6.1
coreClock: 1.7715GHz coreCount: 20 deviceMemorySize: 7.93GiB deviceMemoryBandwidth: 298.32GiB/s
2020-06-01 17:45:15.953242: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcudart.
2020-06-01 17:45:15.953276: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcublas.
2020-06-01 17:45:15.953316: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcufft.s
2020-06-01 17:45:15.953360: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcurand.
2020-06-01 17:45:15.953401: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcusolve
2020-06-01 17:45:15.953444: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcuspars
2020-06-01 17:45:15.953471: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library libcudnn.s
2020-06-01 17:45:15.954714: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1697] Adding visible gpu devices: 0
2020-06-01 17:45:15.954764: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1096] Device interconnect StreamExecutor with strength 1 ed
2020-06-01 17:45:15.954783: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1102]      0
2020-06-01 17:45:15.954797: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] 0:   N
2020-06-01 17:45:15.956079: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1241] Created TensorFlow device (/job:localhost/replica:0/t
s id: 0000:03:00.0, compute capability: 6.1)
Initializing
I0601 17:45:19.828761    51 TensorFlowInferencer.cpp:77] Detection Graph Loaded
I0601 17:45:19.829164    51 DatasetReader.cpp:50] readCounter: 0, size: 1
I0601 17:45:19.829181    51 MassInferencer.cpp:122] Evaluating : 397133(1/1)
Starting inference
Inference Time: 0.0384080410004 seconds
Num Detections: 5
I0601 17:45:19.875272    51 TensorFlowInferencer.cpp:133] person: 0.977487
I0601 17:45:19.875303    51 TensorFlowInferencer.cpp:133] diningtable: 0.89958
I0601 17:45:19.875316    51 TensorFlowInferencer.cpp:133] cup: 0.424164
I0601 17:45:19.875329    51 TensorFlowInferencer.cpp:133] bowl: 0.383614
I0601 17:45:19.875344    51 TensorFlowInferencer.cpp:133] person: 0.32481
I0601 17:45:20.668134    51 DatasetReader.cpp:50] readCounter: 1, size: 1
I0601 17:45:20.668421    51 COCODatasetWriter.cpp:29] FullImagesPath: /home/            /train
I0601 17:45:20.668442    51 COCODatasetWriter.cpp:30] FullLabelsPath: /home/            /annotations
I0601 17:45:20.693956    51 DatasetReader.cpp:50] readCounter: 0, size: 1
I0601 17:45:20.694069    51 DatasetReader.cpp:50] readCounter: 1, size: 1
I0601 17:45:20.694087    51 COCODatasetWriter.cpp:252] Successfully Written to COCO dataset
I0601 17:45:20.694099    51 COCODatasetWriter.cpp:256]
Printing Mapping Info
I0601 17:45:20.694108    51 COCODatasetWriter.cpp:257] ********************
I0601 17:45:20.694118    51 COCODatasetWriter.cpp:262] ********************
W0601 17:45:20.694128    51 COCODatasetWriter.cpp:264]
Printing Discarded Classes from Original Dataset
I0601 17:45:20.694136    51 COCODatasetWriter.cpp:265] ********************
I0601 17:45:20.694145    51 COCODatasetWriter.cpp:269] ********************
I0601 17:45:20.694205    51 FrameworkInferencer.cpp:50] Mean inference time: 38(ms)
```

Fig. 5. Detector output example.

The console also provides log information about the execution. This information shows for each image the detected objects classes with their probabilities sorted. Additionally, information about the mean amount of time spent inferencing the images is shown, one of the most interesting and used metrics in object detection field.

One of the most interesting features provided by Detection studio tool set is the combination of this tool and Evaluator.

C. Evaluator

As the name suggests, Evaluator can evaluate two annotated datasets with the same dataset format considering one as the ground truth and other as the generated detections dataset (it could be generated by Detector tool). Evaluator supports mAP and mAR as described in the state of the art section of this paper or in more detail in COCO dataset paper [5]. It outputs mAP and mAR performance metric for each class and for different IoU thresholds.

Every object detection in an image will be evaluated, comparing the detection in both datasets. Since the evaluation procedure in the application is written in C++, it provides a faster performance than the original COCO toolbox which is written in Python.

The evaluation can be further filtered by a specific object class from the detected dataset, so only the classes selected will be considered during the evaluation. Several other options can be selected to give the user more control of the evaluation process. There are two types of IoU available in Evaluator: bounding boxes and masks. Additionally, the different person classes available in some of the dataset class names can be merged into just one person class that contains all the different ones.

In addition to the already introduced Evaluator output information, it also generates a *csv* format report with that information and performance information about each object class.

This tool and Detector can be used combined to generate different dataset of generated annotations using some object detection neural networks and then compare them with the Evaluator functionality. Detector is run over a dataset using different neural networks models with different parameters. It creates several outputs with different characteristics and then Evaluator can be used to compare then objectively, creating a series of metrics that provide information about the performance of each neural network model over a certain dataset. With this information about the performance a user can select which neural network is better for a specific problem.

This workflow is discusses further in the fourth section, where two experiments that use Detection Studio are explained.

D. Deployer

Deployer tool functionality is similar to Detector. It takes as input images that can be fetch from different sources: a video camera (webcam), a video or a ROS/ICE stream and runs inferences in real time over those images using a given object detection neural network, with its configuration file if needed, class names and inferencer implementation (information needed by the application). The main difference between Detector and Deployer is that Detector works with datasets of image files and Deployer accepts a broader variety of input sources.

Once the desired network model and framework for inferencing have been chosen, the tool displays a video player that plays the input while displaying the objects detected with their class names in real time. If the input is a video file, two video players are displayed, one of then with the raw video and the other one with the video and detected objects, in a similar manner as Detector. This video player offers flexibility to pause and play it again or go back and forward in the playback frame by frame. Another feature provided by Deployer is the confidence threshold (minimum value to consider a detection) that can be adjusted to different values to show the differences in the inferences in real time. This will affect the real time detections in the video, since if the threshold is set to a high value the number of objects that will be found in a frame will probably be lower and the other way around. While the video plays, the console outputs the detected objects for each frame of the video with the percentage of confidence and the time spent on inferencing.

The predicted labels can be saved to an output file if needed, setting an output folder in the GUI, which for example can be used to create new images datasets with annotations from a video record or webcam output.

In the group of tools that are used in parallel to the main workflow, three tools are found: AutoEvaluator (command line interface), Converter and Labelling. Each one is also provided in a separated tab. They provide additional functionality to the core of the application.

E. Command line interface (AutoEvaluator and Splitter)

In addition to the GUI offered by Detection Studio, it also supports some command line based applications giving the user flexibility when using the functionalities. To a user with experience with the platform or used to work with the command line it can be easier using the command line applications directly instead of inserting the parameters in the GUI. To use this functionality, a configuration file is provided to the command line application instead of providing the information directly via the GUI. In this configuration file, the selected tool parameters are described. The core of the different tools provided by the GUI application or the command line application is the same.

The available applications are the same offered in the GUI application except Deployer (Viewer, Detector, Evaluator and Converter and available) and additionally AutoEvaluator and Splitter.

AutoEvaluator is a tool based on Detector and Evaluator functionality but it gives more power to the user since it can evaluate multiple networks on a single or multiple datasets in a single run, accelerating the experiments development time. In the configuration file, the datasets and the networks models to evaluate are defined. The results are then written in an output file in *csv* format with the evaluation metrics generated (same behaviour as the normal Evaluator tool). The difference with the GUI provided functionality here is that this functionality can evaluate multiple networks on multiple datasets at the same time, only providing one file describing the configuration. In the GUI, this operation is completed in two different steps, which are separated.

Splitter is another tool offered using command line. It is actually part of Converter functionality. It can split an input dataset in two separated parts, train and test. To do so, Splitter needs an input dataset, its implementation format, its object class names, a ratio of separation and a directory to write the new separated parts. The ratio sets the percentage of the dataset that goes to the train set, leaving the remaining part to the test set.

F. Converter

The datasets formats can differ from one specific implementation to another, so the purpose of this tool is to convert a certain dataset format to another one. This tool receives as input a dataset with the objects class names that are supported by it and the type of dataset format it implements. It needs the type of dataset as input to create a reader, a tool that understands the format for a specific dataset. The format implementation of the wanted converted dataset is also needed, so Detection Studio creates a writer, another tool that knows how to write on a specific dataset format. Converter also gives the opportunity of filtering by object classes if its provided with a set of class names, so a user can select which object classes to consider in the output dataset or even map the object classes to writer classes in the output dataset. This means that the in the case that the object class names in the input and output dataset are different, the application tries to map from the input class names to the output ones, considering the common class name connection between the common datasets and also considering synonyms.

The converted dataset can be splitted into test and train parts. To do so, a train ratio is provided to the tool and it divides the dataset in two separated parts. This option can be useful to create divisions of the converted dataset.

After the conversion is completed, Viewer functionality can be used to display the converted dataset and make sure the process completed successfully or it also can be used with the different tools provided by Detection Studio.

G. *Labelling*

Detection Studio provides the user with different tools related with labelling a dataset. This functionality is offered within Deployer tool, complementing it. This means that this functionality is provided in the video player created when using Deployer.

- The first feature is the possibility of adjust the bounding boxes generated. The user can adjust the size and position of a certain detection bounding box stopping the video when the error is found and then adjusting the distribution of the box to the object.
- The second feature is changing the class name for every detected object. This means that a user can select a detected bounding box in the video image and change the class name in real time to one of the class names provided or to a completely different one, also having the chance of adjusting the probability of the selection.
- The third feature is related with the previous ones and is adding new detections. The user can draw a new bounding box in a stopped frame and then give this a class name and probability.

H. *Deployer as a ROS Node*

Detection Studio Deployer tool functionality is also provided as a ROS Node, an extension application that complements the ROS core. It can be used in a decoupled ROS application easily. This node also needs a configuration file with the same detailed information as the Deployer tool offered in Detection Studio. Having that file, the node takes the input stream and outputs the objects detected in each frame using the object detection model specified in the configuration file. This output is then written on real time, while live images arrive. For each image, each object detected is specified with its position in the image (bounding box) and the probabilities for each prediction.

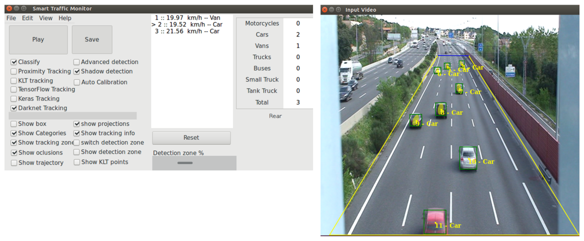
The Deployer as a ROS Node functionality makes it easier for software developers to add the functionality offered in Deployer to other projects where the core is ROS. An example would be a project with ROS as core that uses live video stream, detects the objects on the video and does something based on the detected objects. The first experiment presented in section IV could be an example of use for this functionality in the future.

IV. Objective comparison

This section is dedicated to present real experiments conducted using Detection Studio, showing its value in the comparison of neural networks models and datasets. Two experiments are described as examples. On the first one, a traffic monitoring application (*Smart-Traffic-Sensor*), uses it to compare networks and in the second one a comparison of famous networks is conducted.

A. *Smart-Traffic-Sensor evaluation*

Smart-Traffic-Sensor is an application that monitors road traffic using computer vision. This application receives an image as input and generates as output an image with the objects detected on it (see Figure 6) and some statistics. On a first version called *Traffic Monitor* [18], traditional computer vision techniques were used to evaluate the real camera video and track the different vehicles on the image. After this version, a new development approach was followed in [19], using state of the art deep learning techniques in order to compare how the performance improves from the previous approach to the new one. In this new version, several deep neural networks were trained using different frameworks (TensorFlow, Darknet and Keras)in the task of detection and classification of vehicles with different conditions, and then they were compared using Detection Studio. In Table 1, the comparison of the different networks used for the project is showed. The metrics used to compare the performance of the networks were mAP, mAR and the mean inference time in milliseconds. A value of 0.5 was considered as IoU threshold to consider a prediction as having enough quality to be considered. In each row, a different neural network is considered and in the columns the output statistics are displayed. The experiment was conducted on a computer without GPU. An other important remark to consider is that all the experiments were conducted using the same dataset created for this concrete application from traffic images.



```
 1 :: 19.97  km/h -- Van
> 2 :: 19.52  km/h -- Car
 3 :: 21.56  km/h -- Car
```

Motorcycles	0
Cars	2
Vans	1
Trucks	0
Buses	0
Small Truck	0
Tank Truck	0
Total	3

✓ Classify	☐ Advanced detection
☐ Proximity Tracking	✓ Shadow detection
☐ KLT tracking	☐ Auto Calibration
☐ TensorFlow Tracking	
☐ Keras Tracking	
✓ Darknet Tracking	
☐ Show box	✓ show projections
✓ Show Categories	✓ Show tracking info
✓ Show tracking zone	☐ switch detection zone
✓ Show oclusions	☐ Show detection zone
☐ Show trajectory	☐ Show KLT points

Fig. 6. Smart-Traffic-Sensor output.

Six neural networks are considered in the experiment, divided in pairs that have been trained using a different framework. For each pair of networks, one of then is a pretained

Network	mAP (Overall) (IoU= 0.5:0.95)	mAR (Overall) (IoU= 0.5:0.95)	Mean inference time (ms)
Keras SSD VVG-16	0	0	0
Yolov3	0	0	14162
TensorFlow SSD MobileNet v2	0.0035	0.0373	142
Keras SSD VVG-16 Pretrained	0.6709	0.7082	3194
Yolov3 Pretraine	0.8641	0.9385	16894
TensorFlow SSD MobileNet v2 Pretrained	0.3283	0.4231	76

Table 1. Comparison of networks using Detection Studio. Table extracted from [19].

network and the second one is the same pretrained network but trained again on the dataset created for the experiment.

In the case of TensorFlow framework, the pretrained network is a SSD MobileNet v2 [20] trained with COCO dataset . The Keras trained architecture is a SSD VGG-16 [21] and a pretrained YOLO v3 [22] with Darknet network is also considered in this comparison.

This experiment results show that the best performing neural network (NN) is the YOLO network trained in the application dataset, which is considered the top performing when using the tool and is used as reference. Its performance is higher, but the mean inference time is also higher, which could mean that its more complex or deeper. The results also show that the pretained networks performance is completely incorrect, but a retrain on the collected dataset results on a great rise in performance, resulting on networks which performance is quite high, having the YOLOv3 network as the best performing one. The high mean inference time could be associated with the use of a CPU instead of a GPU.

In this project, Detection Studio was used for a second type of experiment. In the second experiment, Detection Studio toolbox was used to compare the performance of the previous version of the project (*Traffic-Monitor*) against the new version (*Smart-Traffic-Sensor*). The dataset is divided in three different slots for this experiment: the first one contains the high-quality videos, the second one the videos where the weather is bad and the third one the low quality videos. These slots are the input for the best version of *Smart-Traffic-Sensor*, *Traffic-Monitor* and the best performing network (YOLO, as shown in the previous experiment), and the performance results obtained by each system using Detection Studio are compared. In Table 2 this experiment is displayed, with the measures provided by application. The best performance is obtained by *Smart-Traffic-Sensor*, which is quite better than the previous version of the application and some points better than the best performing NN alone.

System type	mAP	mAR
Smart-Traffic-Sensor	0.8926	0.9009
Traffic-Monitor	0.4374	0.5940
Best perfoming NN (YOLO)	0.8316	*0.8966*

Table 2. Comparison of systems in hight-quality videos using Detection Studio. Table extracted from [19].

B. Comparison of well-known networks results using Detection Studio

In this second experiment, four different pretrained object detection networks are evaluated using Detection Studio. The network selection consider different popular object detection methods [11]: SSD, Faster RCNN and YOLOv3. In the process, *Detector* and *Evaluator* are the tools involved. The results obtained are compared between them and with the ones provided by the developers.

In this experiment, the dataset evaluated is COCO minival. The experiments are run on a Nvidia GeForce GTX 1080 GPU. The expected results are a mean inference time close to the one provided by the developers and a mAP approximately equal to that reported by developers.

The selected networks are an implementation of SSD Inception v2, Faster RCNN Resnet 101, YOLOv3 and Faster RCNN Resnet 50 FPN. The first and second are downloaded from the TensorFlow detection model zoo [23]. It offers a broad variety of pretrained networks with metrics. For YOLOv3 the configuration and weights are downloaded from the official documentation and the fourth is included in PyTorch vision model zoo [24]. With this set of different networks, the wide variety of frameworks supported is shown in a real world example. Involving in this experiment TensorFlow, PyTorch and Yolo-OpenCV module.

In table 3, the results obtained are displayed. The mean inference time is slightly higher for the experiments conducted with the tool. This is probably due to the different GPU used. TensorFlow's pretrained networks and YoloV3 official results were obtained using a Nvidia GeForce GTX TITAN X card and PyTorch's pretrained network using V100 GPUs.

Detection Studio considers both AP and AR in the evaluation, providing these metrics to the user from a IOU of 0.5 to 0.95 and the mean of each metric for that range. The mAP results are close in every experiment, having YOLOv3 as the best performing network, as expected.

With this experiment, the use of Detection Studio for validation of official neural network results is shown and the broad variety of application for the tool box.

V. Conclusions

Detection Studio, an open source software application for evaluation of object detection models has been presented in this paper. The different tools included in the application for working with object detection networks and datasets have been described in detail and both the possibility of using its GUI or command line application. With the two experiments

Network	mAP (Overall) (IoU= 0.5:0.95)	mAR (Overall) (IoU= 0.5:0.95)	Mean inference time (ms)
SSD inceptionv2 pretrained using DetectionStudio	0.27	0.31	44
SSD inceptionv2 pretrained	0.24	x	42
Yolov3 using DetectionStudio	0.47 (IoU=0.5)	0.5 (IoU=0.5)	31
Yolov3	0.55 (IoU=0.5)	x	29
Faster RCNN resnet101 pretrained using DetectionStudio	0.37	0.43	122
Faster RCNN resnet101 pretrained	0.32	x	106
Faster RCNN resnet50 FPN pretrained using DetectionStudio	0.35	0.46	102
Faster RCNN resnet50 FPN pretrained	37.0	x	59

Table 3. Comparison of networks using Detection Studio. The result *x* is used when the official results do not give that information.

conducted, its applicability in real world scenarios has been demonstrated. The application and source code is available at: https://github.com/JdeRobot/DetectionStudio

Acknowledgments

The authors thank Google for funding the project in their program *Google Summer of Code* 2018 and 2019.

References

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[2] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes Challenge: A Retrospective," *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136, Jan 2015.

[3] S. Song and J. Xiao, "Tracking Revisited Using RGBD Camera: Unified Benchmark and Baselines," in *2013 IEEE International Conference on Computer Vision*, Dec 2013, pp. 233–240.

[4] L. Spinello and K. O. Arras, "People detection in RGB-D data," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2011, pp. 3838–3843.

[5] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[7] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[8] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[9] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8026–8037.

[11] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, "Object detection with deep learning: A review," *arXiv preprint arXiv:1807.05511*, 2018.

[12] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *ArXiv e-prints*, Jun. 2015.

[13] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: single shot multibox detector," *CoRR*, vol. abs/1512.02325, 2015.

[14] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[15] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[16] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018.

[17] ——, "Yolo9000: better, faster, stronger," *arXiv preprint*, 2017.

[18] R. Kachach and J. M. Caas, "Hybrid three-dimensional and support vector machine approach for automatic vehicle tracking and classification using a single camera," *Journal of Electronic Imaging*, vol. 25, no. 3, pp. 1 – 24, 2016.

[19] J. F. Martnez, "Monitorizacin Visual de Trfico Rodado usando Deep Learning," Master's thesis, Universidad Rey Juan Carlos, Madrid, Spain, 2019.

[20] "SSD Mobilenet V2 COCO Config," https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssd_mobilenet_v2_coco.config, Accessed: 2019-11-20.

[21] "SSD: Single-Shot Multibox Detector Keras implementation," https://github.com/pierluigiferrari/ssd_keras, Accessed: 2020-05-20.

[22] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015.

[23] "Tensorflow detection model zoo," https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md, Accessed: 2020-05-20.

[24] "Pytorch torchvision models," https://pytorch.org/docs/stable/torchvision/models.html, Accessed: 2019-11-20.

[25] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen, "Deep learning for generic object detection: A survey," *arXiv preprint arXiv:1809.02165*, 2018.

[26] M. Papadomanolaki, M. Vakalopoulou, S. Zagoruyko, and K. Karantzalos, "Benchmarking deep learning frameworks for the classification of very high resolution satellite multispectral data." *ISPRS Annals of Photogrammetry, Remote Sensing & Spatial Information Sciences*, vol. 3, no. 7, 2016.

[27] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *arXiv preprint arXiv:1511.06435*, 2015.

[28] L. Deng and D. Yu, "Deep learning: Methods and applications," *Foundations and Trends in Signal Processing*, vol. 7, no. 34, pp. 197–387, 2014.

[29] V. Kovalev, A. Kalinovsky, and S. Kovalev, "Deep Learning with Theano, Torch, Caffe, TensorFlow, and Deeplearning4J: Which One Is the Best in Speed and Accuracy?" 10 2016.

[30] J. Lawrence, J. Malmsten, A. Rybka, D. A. Sabol, and K. Triplin, "Comparing TensorFlow Deep Learning Performance Using CPUs, GPUs, Local PCs and Cloud," 2017.

[31] K. Pogorelov, M. Riegler, P. Halvorsen, C. Griwodz, T. de Lange, K. Randel, S. Eskeland, D. Nguyen, D. Tien, O. Ostroukhova *et al.*, "A comparison of deep learning with global features for gastrointestinal disease detection," 2017.

[32] A. M. Nadig, "Deep learning framework analysis," *International Research Journal of Engineering and Technology (IRJET)*, vol. 05, no. 08, 2018.

[33] A. Shatnawi, G. Al-Bdour, R. Al-Qurran, and M. Al-Ayyoub, "A comparative study of open source deep learning frameworks," in *Information and Communication Systems (ICICS), 2018 9th International Conference on*. IEEE, 2018, pp. 72–77.

Vinay Sharma He graduated in Computer Engineering in 2009, and a Pd.D. in Computer Vision in 2017, both from the Rey Juan Carlos University. His research interests include computer vision, robotics and deep learning.

Francisco Rivas Biography text here.

Sergio Paniego received the Double degree in Informatics and Software Engineering in 2018 from the Universidad Rey Juan Carlos.

José María Cañas received the Telecommunication Engineer degree in 1995, and a Ph.D. in Computer Science in 2003, both from the Universidad Politcnica de Madrid. Currently he is associate professor at Universidad Rey Juan Carlos, leading the Robotics Group. His research interests include robotics and artificial vision.