# UFOD: A Unified Framework for Object Detection

**Manuel García-Domínguez** and **César Domínguez** and **Jónathan Heras** and **Eloy Mata** and **Vico Pascual**[1]

**Abstract.** Object detection models based on deep learning techniques have been successfully applied in several contexts; however, non-expert users might find challenging the use of these techniques due to several reasons, including the necessity of trying different algorithms implemented in heterogeneous libraries, the lack of support of many state-of-the-art algorithms for training them on custom datasets, or the variety of metrics employed to evaluate detection algorithms. These challenges have been tackled in this work by the development of UFOD, an open-source framework that enables the training and comparison of object detection models on custom datasets using different underlying frameworks and libraries — currently, the most well-known object detection frameworks have been included in UFOD, and new tools can be easily incorporated thanks to UFOD's high-level API. Finally, to prove the suitability of our approach, the UFOD framework has been employed to construct models for several datasets, obtaining accurate results for them.

## 1 INTRODUCTION

Object detection has received a lot of attention in recent years due to its multiple applications in contexts such as, just to name but a few, agriculture [18], manufacturing [26], robotics [13] or medicine [20]. This interest is partially thanks to deep learning techniques, that provide accurate models for object detection [31, 39]. These techniques can be classified into two groups: *two-stage algorithms*, such as Faster R-CNN [36] or FPN [22], that include a step for generating object proposals, and another step for classifying such proposals; and, *one-stage algorithms*, such as RetinaNet [23], SSD [25] or YOLO [35], that conduct the detection process without the step that generates object proposals. In both kinds of algorithms, there is a trade-off between accuracy and speed; namely, one-stage detectors are faster but less accurate than two-stage detectors; and, therefore, they are applied in different contexts.

In spite of its success, the adoption of deep learning techniques by users outside the machine learning community is a slow process due to several factors [12]. First of all, the constant flow of new deep learning architectures for object detection makes it difficult to keep track of the best methods. In addition, even if there is a trend of publishing the source code of the algorithms associated with research papers, most algorithms are not prepared to train models with custom datasets, and it might be challenging to use such a code in new projects. This issue has been faced with the development of frameworks, such as the Tensorflow detection API [16] or MXNet [9], that provide several deep learning detection algorithms ready to be trained with the users' datasets.

The implementation of several algorithms in the same framework is an important feature since there is not a silver bullet solution to solve all the detection problems [29]; and, hence, it is necessary to search for the most suitable algorithm for each particular problem. However, there is not a single framework that provides all the existing detection algorithms, and each library uses its own format for encoding the datasets, training protocol, performance measures, and preprocessing steps. Therefore, it is difficult to train and compare different methods implemented in different frameworks, and conclude which one is the best for a particular problem. In this work, we tackle these challenges by developing *UFOD*, a framework that aims to facilitate the construction and usage of deep learning object detectors. Namely, the contributions of this work are the following:

- We conduct a survey of the frameworks that provide deep object detection algorithms, and analyse their particularities to train and use them, see Section 2.
- We design a pipeline to train and test detection algorithms independently of the framework or library where they were implemented. As a by-product, we have developed *UFOD*, an open-source and easily extensible framework that can be employed to train detection models using several deep learning libraries and algorithms. A detailed description of our methods and framework is provided in Section 3.
- We prove the suitability of *UFOD* by constructing several detection models using different underlying libraries and datasets, see Section 4.

This project brings to the table benefits from several perspectives. The UFOD framework can be easily extended to include state-of-the art object detection algorithms to be trained with custom datasets, facilitating in this way the dissemination of those algorithms. Moreover, users interesting in constructing a detection model can easily try different alternatives, and discover which is the one that adapts better to their task. The UFOD framework is available at `https://github.com/ManuGar/UFOD` where the interested reader can find all the necessary documentation to use this tool.

## 2 A SURVEY OF OBJECT DETECTION FRAMEWORKS

In the literature, we can find several recent surveys of deep learning techniques for object detection [2, 17, 24, 44]. These surveys are focused on aspects such as the detection algorithms, the employed datasets, or the evaluation metrics. However, up to the best of our knowledge, there is not a survey of the frameworks that provide those detection algorithms. In this section, we conduct such a study by analysing the available frameworks for object detection using deep learning, their features, and the algorithms that they offer, see Tables 1–3. Moreover, we present the drawbacks that are faced by users when employing these frameworks.

[1] Universidad de La Rioja, Spain, email: {manuel.garciad, cesar.dominguez, jonathan.heras, eloy.mata, vico.pascual}@unirioja.es

For our survey, we focus on object detection frameworks that provide several detection algorithms and can be trained on custom datasets. As can be seen in Table 1, this field is in constant evolution (all the tools have released new versions in the last year) and Python is the dominant programming language. On the contrary, there is not a single underlying deep learning library, but several libraries like Darknet [33], MXNet [9], PyTorch [30], and Tensorflow [15] are employed.

| Framework | Language | Year | Underlying library |
|---|---|---|---|
| Darknet detection [35] | C++ | 2018 | Darknet |
| Detectron [42] | Python | 2019 | PyTorch |
| MaskRCNN-benchmark [27] | Python | 2018 | PyTorch |
| MXNet Detection [10] | Python | 2019 | MXNet |
| MMDetection [8] | Python | 2019 | PyTorch |
| SimpleDet [11] | Python | 2019 | MXNet |
| Tensorflow Detection API [16] | Python | 2019 | Tensorflow |
| Tensorpack [41] | Python | 2019 | Tensorflow |

**Table 1.** General features of frameworks for object detection

As we have explained in the Introduction, there is not a single framework that supports all the object detection algorithms available in the literature. In Table 2, we include the algorithms that are provided by at least one of the tools; other algorithms, for instance MegDet [31] or SNIPER [39], are implemented using libraries like Pytorch or Tensorflow but they are not included as part of a framework, or provide an easy-to-follow procedure to construct detection models for custom datasets with them — these issues harm their usability and adoption. The most well-known and established deep object detection algorithms (that are Faster R-CNN [36], RetinaNet [23], SSD [25], and YOLO [35]) are available in several frameworks. On the contrary, more advanced or recent algorithms, such as Cascade R-CNN [7] or TridentNet [19], are only supported by a few systems. In this sense, the most complete framework is MMDetection [8], but even that system fails to include important algorithms like YOLO. Hence, in order to search for the best detection model, it is necessary to use several frameworks and libraries. Nevertheless, this presents several challenges due to the particularities of each system for training and evaluating detection algorithms, see Table 3.

In the pipeline to create a detection model, we can distinguish four stages that are common to any object detection algorithm: data and model preparation, training, evaluation, and usage. For the data and model preparation stage, one of the most important aspects is the annotation format taken as input by the frameworks — this is relevant since there is not a standard annotation format, and annotation tools, like LabelImg [40] or YOLO mark [5] produce different kinds of files. There are two common formats that are Pascal VOC [14] and COCO [21]; moreover, frameworks like Darknet or SimpleDet [11] provide transformers to convert from either Pascal or COCO formats to their own formats. Another aspect in the data preparation step is related to how the data is split into training and testing sets. Most frameworks require a specific folder structure, and in some cases it is also necessary to explicitly provide files with the lists of training and testing images. Finally, in this first stage, it remains the question of how to configure the model (that is, fixing hyperparameters such as the network architecture, the batch size, the learning rate and so on). Such a configuration is usually provided by means of either a configuration file or by modifying the source code that launches the training process — usually, the latter require some programming experience. Such a heterogeneity of annotation formats, file structures, and configuration files hinders the use of several frameworks and libraries.

For the training stage, the analysed frameworks provide similar features. Training from scratch object detectors that are based on deep learning algorithms requires large datasets (for instance, the Pascal VOC dataset contains 16K images, and the COCO dataset includes 80K images); however, most custom datasets are small. A successful approach to deal with this problem is fine-tuning [32], a transfer learning technique that re-uses a model trained in a source task, where a lot of data is available, in a new target task, with usually scarce data. This functionality is supported by all the analysed frameworks, that, in addition, provide a model zoo to apply fine-tuning from different source models. Another important aspect in this stage is the time required to train the detection models, this issue can be alleviated by the usage of multiple GPUs (a feature provided by all the analysed tools), and distributed training (a feature supported by all the systems but Darknet and MaskRCNN benchmark).

Finally, the last two stages of the pipeline, evaluation and usage, are closely related and pose similar challenges. Object detection algorithms are usually evaluated using a metric known as mAP [2]. However, there are multiple definitions of this metric and each framework has its own implementation. Therefore, it is not sound to directly compare the mAP results used with different systems. Similarly, the usage of the models greatly varies from tool to tool since each system loads models, performs predictions, and outputs the results in a particular way. This makes difficult to compare models generated with different systems. An approach to solve these problems is based on the definition of converters among libraries. Some of these solutions are collected in a project called *Deep learning model converter* [37]. Another similar project is the *Open Neural Network Exchange (ONNX)* [28] that aims to build models with the ONNX representation and then use the framework that better adapts to the users' goals to execute it. In the same line is the dnn module of OpenCV [4], where a representation close to Caffe's representation has been developed, and some frameworks and models have been converted to such a format. The main problem with the approach based on converters is that some frameworks do not implement all types of layers of deep learning models; so, certain models cannot be converted to all the frameworks.

In our work, we aim to deal with all the challenges explained in the different stages of the construction of deep learning object detectors by defining a high-level system that allows the integration of deep learning frameworks and libraries.

## 3 METHODS

UFOD (that stands for Unified Framework for Object Detection) is an open-source library that has been implemented in Python, and has been designed to simplify and automatise the process of training multiple object detection models using different libraries. The different parts of the framework, its dependencies, and several examples of the use of this framework are provided in the project webpage. In this section, we present the workflow of UFOD, and its extensible API.

The workflow of UFOD, depicted in Figure 1, captures all the necessary steps to train several object detection models and select the best one. Such a workflow can be summarised as follows. First of all, the user selects the dataset of images and some configuration parameters (mainly, the algorithms that will be trained and the frameworks or libraries that provide them). Subsequently, UFOD splits the dataset into a training set and a testing set. The training set is em-

|  | Cascade R-CNN | Fast R-CNN | Faster R-CNN | FCOS | Mask R-CNN | RetinaNet | R-FCN | SSD | TridentNet | YOLO |
|---|---|---|---|---|---|---|---|---|---|---|
| Darknet |  |  |  |  |  |  |  |  |  | ✓ |
| Detectron | * | ✓ | ✓ |  | ✓ | ✓ |  |  |  |  |
| MaskRCNN benchmark |  | ✓ | ✓ | * | ✓ | ✓ |  |  |  |  |
| MXNet Detection |  |  | ✓ |  |  |  |  | ✓ |  | ✓ |
| MMDetection | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  |  |
| SimpleDet | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |  |  | ✓ |  |
| Tensorflow Detection API |  |  | ✓ |  | ✓ |  | ✓ | ✓ |  |  |
| Tensorpack | ✓ |  | ✓ |  | ✓ |  |  |  |  |  |

**Table 2.** Object detection algorithms available in the frameworks. "✓" means officially supported, "*" means supported in a forked repository and blank means not supported

|  | Annotation format | Data split | Config | Multiple GPU | Distributed |
|---|---|---|---|---|---|
| Darknet | TXT | Folders, Files | File | * |  |
| Detectron | COCO | Registerec dataset | File | ✓ | ✓ |
| MaskRCNN benchmark | COCO | Folders | File | ✓ |  |
| MXNet Detection | PASCAL | Files | Code | ✓ | ✓ |
| MMDetection | COCO, PASCAL | Folders | File | ✓ | ✓ |
| SimpleDet | JSON | Folders | File | ✓ | ✓ |
| Tensorflow Detection API | PASCAL | Files | Code | ✓ | ✓ |
| Tensorpack | COCO | Folders | Code | ✓ | ✓ |

**Table 3.** Object detection features of the analysed frameworks. "✓" means officially supported, "*" means supported in a forked repository and blank means not supported

ployed to construct several object detection models, and the best of those models is selected based on their performance on the testing set. The output of the framework is the best model, and an application, in the form of a Jupyter notebook, to employ such a model. Apart from the first step — that is, the selection of the models to be trained — the rest of the process is conducted automatically by UFOD without any user intervention. We now provide a more detailed explanation of the main features of this framework.

We start by explaining how the execution of UFOD is launched. The UFOD framework is invoked as a Python program that takes as input a JSON file with three parameters: a dataset of annotated images, an execution mode, and the list of algorithms that will be trained. The first parameter is the path to the folder where the images and their annotations, in the Pascal VOC format, are stored. Here, the user might indicate explicitly a train/test dataset split by using a concrete folder structure, or just provide the images and their annotations. In the former case, UFOD will employ the provided split for training the models and testing them; and, in the latter, UFOD will divide the dataset by itself using, by default, a 75% for training and a 25% for testing — the percentage for the split division can be modified by the user.

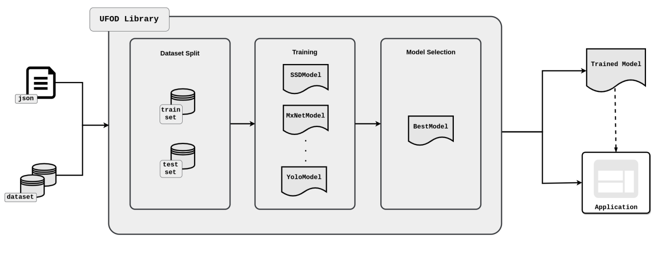

**Figure 1.** Workflow of the UFOD framework

The second parameter is the execution mode. *UFOD* can be run both locally, in a single computer with GPUs, or in a cluster of computers. The reason to include the cluster execution mode is the fact that training object detection algorithms is a time-consuming task; and, therefore, if several models must be trained, this task can take advantage of a distributed process to train them at the same time — this feature is implemented using the cluster manager SLURM [43]. Depending on the execution mode, some additional parameters must be configured; for example, the number of available GPUs if working locally, or the time that the tasks can be run in the cluster.

The third parameter is the list of algorithms that will be trained. The UFOD framework supports both algorithms implemented inside of deep learning frameworks, and also algorithms implemented as standalone libraries. Currently, the UFOD framework provides support for the algorithms provided in three widely employed object detection frameworks: Darknet [33] (that supports the TinyYOLO [34] and YOLO [35] algorithms), Tensorflow [15] (that provides, among others, the SSD and faster RCNN algorithms with several backbones), and MXNet [9] (that supports, among others, the SSD algorithm with different backbones). In addition, UFOD can also be employed to construct detection models using two Keras libraries that implement respectively the RetinaNet [23] and MaskRCNN [1] algorithms.

Using those three parameters, UFOD is able to train object detection models employing different algorithms and frameworks thanks to its extensible API. Independently of the deep learning framework and algorithm used, the procedure to train an object detection model can be summarised as follows: (1) organise the dataset using a particular folder structure; (2) transform the annotations of the dataset to the correct format; (3) create the necessary configuration files; and (4) launch the training process. Steps (1), (2) and (4) are framework operations, since all the models of a framework require the same folder structure, annotation format, and their training process is launched in the same way; whereas, Step (3) depends on the concrete model. These steps have been modelled using an abstract class called `IObjectDetection` (see Figure 2), that is particularised for each concrete framework and model. To be more concrete, for each framework that provides several detection algorithms, an abstract class providing the functionality for Steps (1), (2) and (4) is implemented (see, for instance, the class Darknet in Figure 2); in addition, such a class is extended with subclasses for each individual model implementing the functionality for Step (3) (see the classes Tiny YOLOV3 and YOLOV3 in Figure 2). In the case, of libraries that implement a single algorithm, a class with all the methods must be created (see, for instance, the class RetinaNet in Figure 2). Including a new framework or library in UFOD is as simple as defining a new class that extends `IObjectDetection` — the guidelines for such a process are provided in the project webpage.

After training the object detection models, it remains the question of deciding which model produces the best results. Each frame-

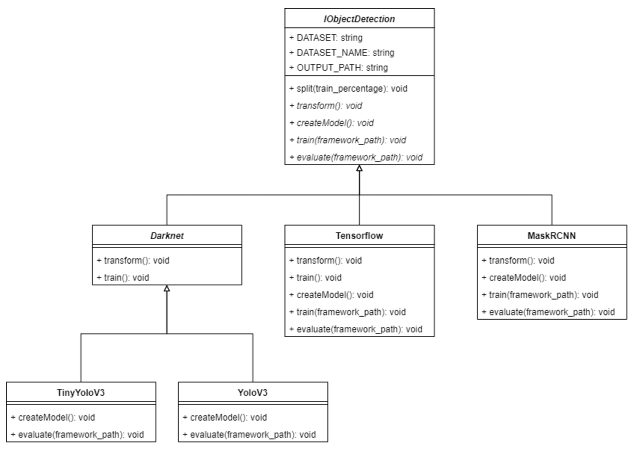

**Figure 2.** Part of the class diagram of the UFOD API for including object detection models and frameworks. To improve readability only some classes that extend the `IObjectDetection` class have been included in the diagram

work can compute the performance of their models; however, different frameworks employ different evaluation metrics; and, therefore, it is difficult to compare the models produced with different tools. To deal with this problem in UFOD, we have included a procedure to evaluate algorithms independently of the underlying framework. Such a procedure is a two-step process that can be summarised as follows. Given a folder with the images and annotations that will be employed for testing a model, (1) the model detects the objects in those images and stores the result in the Pascal VOC format inside the same folder; and (2) the original annotations and the generated detections are compared with a particular metric. This two-step process has been modelled in the UFOD API (see Figure 3) using the abstract class `IPredictor` to implement the first step — this class is particularised for each framework since all the models of the same framework perform predictions in the same way — and the abstract class `IEvaluator` to implement the second step — this class is particularised with different evaluation metric; and, currently, metrics such as the mAP, the IoU or the F1-score are supported. Using this approach, the models constructed using the UFOD framework can be compared even if they were constructed using different frameworks. Moreover, this part of the UFOD API can be employed to compare models constructed outside the framework by using a common metric.

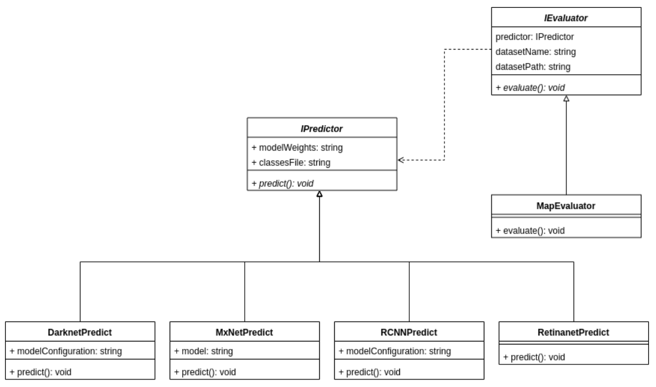

**Figure 3.** Class diagram of the *UFOD* API for evaluating object detection models

Finally, the output produced by the *UFOD* framework after comparing the constructed models is a model and the necessary code to use it. The generated code can be run using the command line, integrated in other Python programs, or executed using Jupyter notebooks. This last option is the simplest to use since Jupyter notebooks are documents for publishing code, results and explanations in a form that is both readable and executable [6].

In order to prove the feasibility of using the *UFOD* framework, we have tested it with several examples by training several different models coming from algorithms implemented in different frameworks.

## 4 RESULTS

In this section, we present the results obtained with the *UFOD* framework in two different datasets: the Optic dataset [3] and the Reconyx dataset [38]. The Optic dataset consists of ophthalmological images that are employed to detect discs where blood vessels join together inside the eyeball. This dataset contains 1250 images, was annotated with the YOLO format, and there is only one disc per image. The Reconyx dataset was created to detect different animals in the wild. This dataset consists of 946 images annotated using a custom format, and several animals might appear in the same image. Both dataset are small in comparison with the Pascal VOC or COCO datasets, but this is common when training a detection algorithm in a custom dataset. Since these two datasets were not annotated in the Pascal VOC format, we transformed those annotations using two Python scripts.

| | | Optic | | | Reconyx | |
|---|---|---|---|---|---|---|
| | F1 | IoU | mAP | F1 | IoU | mAP |
| tiny YOLO | 87.01 | 60.79 | 90.67 | 72.66 | 58.23 | 69.78 |
| YOLO | 91.89 | 75.56 | 90.91 | **80.42** | **61.76** | **81.89** |
| SSD 300 VGG16 | 98.99 | 84.66 | 90.91 | 71.50 | 58.06 | 60.96 |
| SSD 512 VGG16 | 97.44 | 83.35 | 91.91 | 64.40 | 47.28 | 65.28 |
| SSD 512 Mobilenet | **100** | **87.69** | **90.88** | 54.55 | 50.33 | 48.53 |
| SSD 512 Resnet | 99.00 | 86.77 | 90.91 | 68.41 | 54.12 | 66.21 |
| Mask RCNN | 94.91 | 72.65 | 90.91 | 14.93 | 9.16 | 10.58 |

**Table 4.** Results using different models for the Optic and Reconyx dataset. The best results for each dataset are shown in bold face.

For our experiments, we employed the following settings. Both datasets were randomly split into two different sets using a 75% for training, and a 25% for testing. The training set was employed to construct models using the YOLO v3 and Tiny YOLO v3 algorithms of the Darknet framework, the SSD algorithm (with 4 different backbones) of the MXNet framework, and the Mask RCNN from a Keras library. The performance of these models was evaluated on the test set using three different metrics: F1-score, IoU, and mAP, see Table 4. Moreover, for the Reconyx dataset, we computed the AP for each individual class, see Table 5. It is worth mentioning that all the processes for our experiments were conducted automatically by the UFOD framework just taking as input a JSON file with the configuration options.

The results for the Optic dataset are included in the three first columns of Table 4. For this dataset, *UFOD* obtains a mAP of 91% with all the models, but there are differences among them when using the F1-score and the IoU metrics. Namely, the SSD model using the Mobilenet backbone obtains the best results with a 100% F1-score and an 88% IoU. These results are similar to those reported in [3] where only a YOLO model was trained, and shows the benefits of trying different algorithms for different frameworks. For the Reconyx dataset, see the last three columns of Table 4, the best results

| | Class 1 | Class 2 | Class 3 | Class 4 | Class 5 | Class 6 | Class 7 | Class 8 | Class 9 | Class 10 | Class 11 | Class 12 | Class 13 | Class 14 | Class15 | Class 16 | Class 17 | Class 18 | Class 19 | Class 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tiny YOLO | 90.91 | 54.55 | 54.55 | 36.36 | 72.16 | 90.91 | 54.55 | 81.82 | 59.09 | 90.91 | 39.55 | **100** | 81.47 | **100** | 78.98 | 63.64 | 81.82 | **72.73** | 28.08 | 63.64 |
| YOLO | **100** | 81.82 | **81.82** | **54.55** | **79.90** | **100** | **81.82** | **90.91** | **63.64** | **100** | **66.54** | **100** | 90.26 | **100** | 70.30 | **81.82** | **90.15** | **72.73** | 67.96 | 63.64 |
| SSD 300 VGG16 | 54.55 | **92.42** | **81.82** | 9.09 | 54.55 | 61.89 | 54.55 | 54.55 | 54.55 | 76.85 | 35.23 | 97.73 | 81.48 | 72.73 | 22.40 | 72.73 | 62.50 | 27.27 | **73.18** | **79.16** |
| SSD 512 VGG16 | 81.82 | 74.55 | **81.82** | 27.27 | 44.45 | 76.01 | 78.18 | **90.91** | 50.65 | 90.91 | 27.48 | **100** | **90.91** | 27.27 | **81.21** | 59.31 | 72.22 | 63.64 | 32.31 | 54.55 |
| SSD 512 Mobilenet | 57.02 | 52.27 | 54.55 | 18.18 | 50.94 | 51.52 | 27.27 | 63.64 | 45.45 | 81.82 | 9.09 | **100** | 63.64 | 27.27 | 61.69 | 36.36 | 63.10 | 27.27 | 17.04 | 62.50 |
| SSD 512 Resnet | 90.91 | 74.55 | **81.82** | 27.27 | 63.64 | 54.55 | 72.73 | 80.86 | 27.27 | 90.91 | 20.45 | **100** | **90.91** | 72.73 | 78.79 | 59.09 | 72.25 | 54.55 | 38.30 | 72.73 |
| Mask RCNN | 14.94 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 18.79 | 0.00 | 45.45 | 0.00 | 6.06 | 9.09 | 0.00 | 45.77 | 0.00 | 9.09 | 9.09 | 16.88 | 36.36 |

**Table 5.** Average precision (AP) for each individual class of the Reconyx dataset using seven different models. The best result for each class is shown in bold face.

are obtained with the YOLO model. In addition, if we are focused on each individual class, see Table 5, we can also notice that YOLO obtains the best average precision for almost all the classes. If we compare our results with those presented in [38], the YOLO model trained with our framework obtains a better IoU (61%) than the one obtained in [38] (57%).

These two examples show the feasibility of using the UFOD framework to construct accurate models. In addition, our experiments also illustrate the benefits of easily training models with different underlying algorithms, since there is not always an algorithm that obtains the best results.

## 5 CONCLUSIONS AND FURTHER WORK

UFOD is a free and open-source framework that aims to facilitate the construction and use of the most suitable object detection model for the particular problems of each user. Namely, UFOD allows users to easily train and compare several object detection models on their own datasets using a common pipeline. The UFOD framework can be seen as a wrapper for the functionality provided by the multiple and heterogeneous existing frameworks and libraries that support object detection algorithms. This is achieved thanks to a high-level extensible API that abstracts all the steps that are required to train and evaluate a detection model. In this paper, we have presented the UFOD framework, and proven that without tweaking the underlying libraries, it is possible to construct accurate models.

As further work, there are several remaining tasks to tackle. First of all, we aim to include in UFOD state-of-the-art algorithms recently released such as MegDet [31] or SNIPER [39]. Moreover, the UFOD framework opens the door to facilitate the application of techniques that require the combination of several models like ensembles. Finally, we plan to extend the UFOD framework to deal with tasks like semantic or instance segmentation.

## ACKNOWLEDGEMENTS

This work was partially supported by Ministerio de Economía y Competitividad [MTM2017-88804-P], Agencia de Desarrollo Económico de La Rioja [2017-I-IDD-00018], a FPI grant from Community of La Rioja 2018 and the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF). CETA-CIEMAT belongs to CIEMAT and the Government of Spain.

## REFERENCES

[1] W. Abdulla. Mask r-cnn for object detection and instance segmentation on keras and tensorflow. `https://github.com/matterport/Mask_RCNN`, 2017.

[2] S. Agarwal, J. Ogier du Terrail, and F. Jurie, 'Recent Advances in Object Detection in the Age of Deep Convolutional Neural Networks', *CoRR*, **abs/1809.03193**, (2019).

[3] I. Ahmad, 'Automatic detection of diabetic retinopathy from fundus images using image processing and artificial neural network', *Department of computer Science and Engineering*, (2019).

[4] R. Alexander. Deep learning, now in opencv. `https://habr.com/company/intel/blog/333612/`, 2017.

[5] A. B. Alexey. YOLO mark. `https://github.com/AlexeyAB/Yolo_mark`, 2018.

[6] D. Avila et al. Project jupyter. `https://jupyter.org/`, 2018.

[7] Z. Cai and N. Vasconcelos, 'Cascade R-CNN: Delving into High Quality Object Detection', in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR'18, pp. 6154–6162, (2018).

[8] K. Chen et al., 'MMDetection: Open MMLab Detection Toolbox and Benchmark', *CoRR*, **abs/1906.07155**, (2019).

[9] T. Chen et al., 'MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems', *CoRR*, **abs/1512.01274**, (2015).

[10] T. Chen et al. Mxnet object detection. `https://gluon-cv.mxnet.io/build/examples_detection/index.html`, 2019.

[11] Y. Chen et al., 'Simpledet: A simple and versatile distributed framework for object detection and instance recognition', *CoRR*, **abs/1903.05831**, (2019).

[12] M. F. Dacrema, P. Cremonesi, and D. Jannach, 'Are we really making much progress? A worrying analysis of recent neural recommendation approaches', in *Proceedings of the ACM Conference on Recommender Systems*, RecSys'19, pp. 101–109. ACM, (2019).

[13] T-T. Do, A. Nguyen, and I. Reid, 'AffordanceNet: An End-to-End Deep Learning Approach for Object Affordance Detection', in *Proceedings of the IEEE International Conference on Robotics and Automation*, ICRA'18, pp. 5882–5889, (2018).

[14] M. Everingham et al., 'The Pascal Visual Object Classes (VOC) Challenge', *International Journal of Computer Vision*, **88**(2), 303–338, (2010).

[15] Google. TensorFlow: An open-source software library for Machine Intelligence. `https://www.tensorflow.org`, 2018.

[16] J. Huang et al., 'Speed/accuracy trade-offs for modern convolutional object detectors', in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR'17, pp. 3296–3305, (2017).

[17] L. Jiao et al., ' A Survey of Deep Learning-Based Object Detection', *IEEE Access*, **7**, 128837–128868, (2019).

[18] A. Kamilaris and F. X. Prenafeta-Bold, 'Deep learning in agriculture: A survey', *Computers and Electronics in Agriculture*, **147**, 70–90, (2018).

[19] Y. Li et al., 'Scale-Aware Trident Networks for Object Detection', *CoRR*, **abs/1901.01892**, (2019).

[20] Z. Li et al., 'CLU-CNNs: Object detection for medical images', *Neurocomputing*, **350**, 53–59, (2019).

[21] T-Y. Lin et al., 'Microsoft COCO: Common Objects in Context', in *Proceedings of the European Conference on Computer Vision (ECCV'14)*, volume 8693 of *Lecture Notes in Computer Science*, pp. 740–755, (2014).

[22] T-Y. Lin et al., 'Feature Pyramid Networks for Object Detection ', in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR'17, pp. 936–944, (2017).

[23] T-Y. Lin et al., 'Focal Loss for Dense Object Detection', in *Proceedings of the IEEE International Conference on Computer Vision*, ICCV'17, pp. 2999–3007, (2017).

[24] L. Liu et al., 'Deep Learning for Generic Object Detection: A Survey', *CoRR*, **abs/1809.02165**, (2018).

[25] W. Liu et al., 'SSD: Single Shot MultiBox Detectors', in *Proceedings of the European Conference on Computer Vision, ECCV'16*, volume 9905 of *Lecture Notes in Computer Science*, pp. 21–37. Springer, (2016).

[26] A. Luckow et al., 'Artificial Intelligence and Deep Learning Applications for Automotive Manufacturing', in *Proceedings of the IEEE Inter-*

*national Conference on Big Data*, Big Data'18, pp. 3144–3152, (2018).

[27] F. Massa and R. Girshick. maskrcnn-benchmark: Fast, modular reference implementation of Instance Segmentation and Object Detection algorithms in PyTorch. `https://github.com/facebookresearch/maskrcnn-benchmark`, 2018.

[28] Microsoft, Facebook open source & AWS. ONNX: Open Neural Network Exchange, 2018.

[29] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine learning*, MIT press, 2012.

[30] A. Paszke et al., 'Automatic differentiation in pytorch', in *Proceedings of the 31st Conference on Neural Information Processing Systems*, NIPS'17, pp. 1–4, (2017).

[31] C. Peng et al., 'MegDet: A Large Mini-Batch Object Detector ', in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR'18, pp. 6181–6189, (2018).

[32] A. S. Razavian et al., 'CNN features off-the-shelf: An astounding baseline for recognition', in *27th Conference on Computer Vision and Pattern Recognition Workshops*, CVPRW'14, pp. 512–519, (2014).

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

[34] J. Redmon and A. Farhadi, 'YOLO9000: Better, Faster, Stronger', *CoRR*, **abs/1612.08242**, (2016).

[35] J. Redmon and A. Farhadi, 'YOLOv3: An Incremental Improvement', *CoRR*, **abs/1804.02767**, (2018).

[36] S. Ren, K. He, R. Girshick, and J. Sun, 'Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks', in *Advances in Neural Information Processing Systems*, volume 28, 91–99, (2015).

[37] S. Yuan. Deep Learning Model Convertors. `https://github.com/ysh329/deep-learning-model-convertor`, 2018.

[38] Stefan Schneider, Graham W Taylor, and Stefan Kremer, 'Deep learning object detection methods for ecological camera trap data', in *2018 15th Conference on Computer and Robot Vision (CRV)*, pp. 321–328. IEEE, (2018).

[39] B. Singh, M. Najibi, and L. S. Davis, 'SNIPER: Efficient Multi-Scale Training', in *Proceedings of the 32nd Conference on Neural Information Processing Systems*, NeurIPS 2018, pp. 1–11, (2018).

[40] D. Tzutalin. LabelImg. `https://github.com/tzutalin/labelImg`, 2015.

[41] Y. Wu et al. Tensorpack. `https://github.com/tensorpack/`, 2016.

[42] Y. Wu et al. Detectron2. `https://github.com/facebookresearch/detectron2`, 2019.

[43] A. B. Yoo, M. A. Jette, and M. Grondona, 'SLURM: Simple Linux Utility for Resource Management', in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *JSSPP 2003*, pp. 44–60, (2003).

[44] Z. Q. Zhao et al., 'Object Detection With Deep Learning: A Review', *IEEE Transactions on Neural Networks and Learning Systems*, **30**(11), 3212–3232, (2019).