Boat Detection Report

Computer Vision

Fabio Vaccaro

University of Padua
Department of Information Engineering
17 July 2021

Contents

1 Introduction

1.1 Challenge

The goal of this project was simple: to find boats and any floating means of transport for humans, categorize and draw a bonding box around it. However there were some limitations with the strategy we were allowed to follow: the use of deep learning was permitted but it should not be an end-to-end learning approach. Also the use of C++ was a must, although Python was allowed for the machine learning/deep learning part.

1.2 Strategy

I started thinking to use a Cascade of Classifiers. I implemented it but, even after some post-processing the results were not adequate to the performance requirements I was imposing myself. At a later time I realized that, maybe, the poor results were caused by a wrong ratio between positive and negative samples.

I then thought to use some types of segmentation techniques. At that time I was experimenting with the Watershed algorithm. I soon realized that the datasets provided contained very difficult samples. In my imagination the boat was in the middle of the sea, without any visual distraction a part from the sea. That was not the case.

Finally I considered to turn my attention towards CNNs. To achieve better results and reduce training time, I decided to use transfer learning in my project. My neural network would be a fine-tuned version of the popular Faster R-CNN. But my idea was not stopping here, I thought to introduce k-means clustering as a segmentation technique into the pre-processing data augmentation phase (more on this later in Section 2).

1.3 Purpose

Considering that CNNs for object detection are a technique already widely experimented the aim of this project is to show how more traditional computer vision techniques could be truly important used in pre-processing as a data augmentation technique. So in the results we will focus on the comparison of the system trained on the standard dataset compared to the performance obtained with the model trained with the dataset pre-processed.

1.4 Pipeline

One of the most challenging goals of this project was the integration of the deep learning part with the more traditional computer vision one. The pipeline was designed to maximize performance while keeping convenience both for the training and inference part.

The training has been divided into two smaller segments: the first one is the pre-processing, made with C++, of the dataset used later for training and the second one is Python code deployed as a notebook. This way we could exploit the power of our multithreaded CPU for the pre-processing thanks to the optimized and fast C++ implementation of OpenCV, while taking advantage of the advanced frameworks for deep learning available in Python and run the training code on the cloud on Colab Pro or on Google Cloud Platform, if we need more power.

The inference, on the contrary, has been studied to be managed directly from the C++ code which, under the hood, uses Python code to make the prediction.

The usage of the two programming languages for both training and inference phase is better explained in Figure 1.

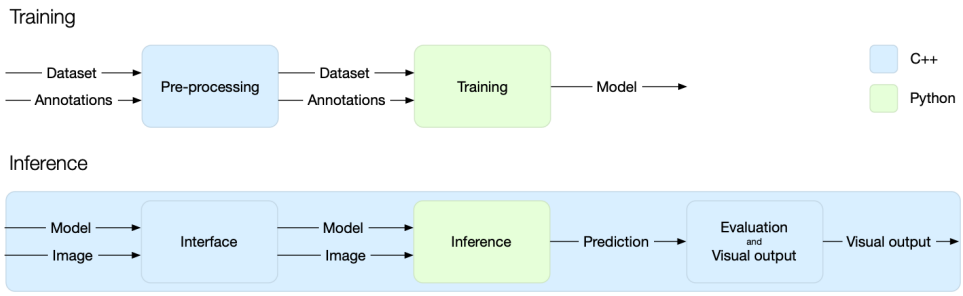


Figure 1: Pipeline: C++ and Python usage

2 Pre-processing

To improve performance of neural networks it is common to make some data augmentation before or during the training step. The reason why this is done is because NNs need lots of data. So if we have a small dataset we use this trick to artificially enlarge it. The keypoint is to add some randomness to data to make the neural network to learn more features. We are basically giving them more samples even if we don't actually have them. Standard techniques used combined with CNNs are cropping, rotating, flipping. I have implemented into the system two kinds of data augmentation.

2.1 Flipping

The first one is implemented in the *Dataset* module and it is applied live at every image request. This should be a light operation to be made during learning and with the lower performance of Python. I have chosen an horizontal flip (vertical flip would have no sense as boats are never upside down like for example airplanes). This is a quick operation and it is also very easy and fast to re-calculate the bounding boxes' positions.

2.2 Segmentation

The second one has been implemented during the C++ pre-processing as it is much more demanding from a computational point of view. The basic idea was to apply some kind of alterations only to particular image segments. I have opted for a clustering-based segmentation approach implemented with the k-means algorithm provided by OpenCV.

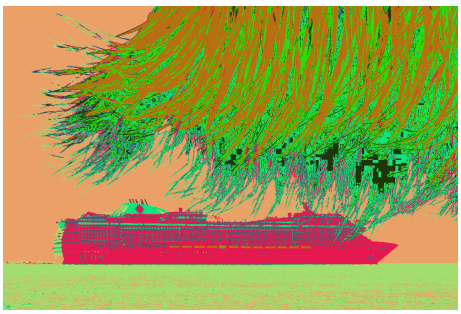
2.2.1 Clustering

k-means algorithm is run on top of a HSV version of the image as we are trying to cluster different segments with the same color. Various values for the number of cluster to be extracted were tested. Good results can be obtained with k = 8. In Figure 2 we can see the original image and the clusters obtained from k-means algorithm here represented as a colored image.
Obtained the clusters, I scan them to count the candidates pixels and the total ones. These are pixels with a HSV value in a predefined color area which represents blue (for sea water), light blue (for sky) and a typical "dirty green" (for rivers and for Venice's canals water). For each cluster a score is calculated as the percentage of candidate pixels found in the cluster.

(a) Original image

(b) Clustered image

(c) Original image



(d) Clustered image

Figure 2: Example of clustering

For image in Figure 2b and Figure 2d scores of their clusters are reported in Table 1.

Cluster	1	2	3	4	5	6	7	8
Figure 2b	74.25	12.11	34.28	58.46	38.70	9.46	63.52	0.15
Figure 2d	0.0	35.50	18.98	0.29	98.20	0.16	3.17	93.58

Table 1: Cluster's score of 2b and Figure 2d

2.2.2 Masks

After the score's calculation, only the top clusters are kept. This is determined with a minimum threshold (here equal to 30%) which can be tuned to obtain different results. The retained clusters are used to create the positive part of a binary mask. Results of previous images are in Figure 9.

(a) Original image

(b) Binary mask

(c) Original image

(d) Binary mask

Figure 3: Example of binary masks

Masks obtained in some cases could be used directly to find the bounding box, but not all images are without distracting elements and not all of them are segmented so well. More examples in Appendix A.1.

2.2.3 Selective blur

At this stage I apply Gaussian Blur only to white areas, maintaining sharp and clear black ones. Results can be analyzed in Figure 4. The aim of this process is to highlight only the part of the image which are important during training, removing distracting elements. These areas, in this particular case, are the sky and the sea which are easily recognizable with a combination of clustering on the color space and thresholding.

(a) Original image

(b) Binary mask

(c) Original image

(d) Binary mask

Figure 4: Example of blurred images

3 Training

3.1 Object detection techniques

In this section I will introduce the most important object detection techniques that have been the state-of-the-art for the last years.

3.1.1 Traditional object detection

Traditional object detection techniques are composed of three stages:

1. **Region proposal**: with the use of algorithms like Selective Search and EdgeBoxes the system extracts regions which could contain an object. These become candidates, region proposals.

2. **Feature extraction**: from each region a fixed-length feature vector is extracted using some image descriptor, for example HOG (histogram of oriented gradients).

3. **Classification**: using the features obtained in the previous step regions are classified using a classifier (distinguishing the probable class). It is often used SVM (support vector machine).

3.1.2 R-CNN

The R-CNN object detector paper was first published in 2013 by Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik [**?**]. It has stunning performances for the time and became the state of the art for object detection. R-CNN consists of 3 main steps:

1. **Region proposal**: in this first stage the system, using Selective Search algorithm, extracts 2000 proposal regions.

2. **Feature extraction**: this step is a clear departure from the traditional approach presented in Section 3.1.1. From each region a 4096 long feature vector is computed using a CNN instead of using an image descriptor (like HOG).

3. **Classification**: as before, a classifier, for example a SVM, is used on top of the feature extracted in the second step.

3.1.3 Fast R-CNN

Fast R-CNN first published in 2015 by Ross Girshick [**?**] overcomes several issues of R-CNN and its best advantage was speed. This model changes approach completely compared to the classic R-CNN as it is based on a single stage with respect to three stages of the previous model.

1. **Convolutional layers**: this step takes as input the image and RoIs (regions of interest) and returns a features map.

2. **RoI Pooling layer**: the feature map is fed into a ROI pooling layer where each proposal region is split into a grid of cells. Max pooling is applied to each grid to be represented by only one value. The values of all grip cells altogether represent the feature vector.

3. **FC layers**: the feature vector is fed to fully connected layers. The last one is divided into:

    - **Softmax layer**: to predict the class scores
    - **FC layer**: to predict the bounding boxes

3.1.4 Faster R-CNN

Faster R-CNN is a better version of the Fast R-CNN. It was published in 2015 by Shaoqing Ren, Kaiming He, Ross Girshick and Jian Sun [**?**]. The model consists of two modules:

1. **RPN**: it stands for region proposal network and it is a fully convolutional network that generates proposals with various scales and aspect ratios. It is important as it introduces attention into the system.

2. **Fast R-CNN**: without the use of a region proposal algorithm (as Selective Search) to calculate the likelihood and the class of each region. The clever idea was to exploit the features learnt during the RPN phase also in the second module sharing the same convolutional layers.

The advantage with respect to the use Selective Search is that also the region proposal search can be customized upon the specific dataset without the need for generic algorithm like Selective Search and EdgeBoxes. The second big advantage is that RPN shares the convolutional layers with R-CNN so it does not take extra time to produce the proposals

3.2 Model choice

For this project I decided to use the most advanced of the models presented in Section 3.1, Faster R-CNN model with a ResNet-50-FPN backbone which means that the convolutional part is taken care by a ResNet network. The model is pre-trained on the MS COCO dataset[1]. So I needed to fine tune the model on my dataset. This was done replacing the classifier with a new and blank module, *FastRCNNPredictor*.

3.3 Training environment choice

I have chosen Google Colab Pro as a development environment for Python as it gives very powerful GPUs at a very reasonable price. This improves the performance 60 times in my test using my Intel 8700K CPU with 32GB of RAM. Tesla P100 and V100 with 12 to 24GB of RAM on Colab was much faster. Running 10 epochs took 40 hours in my PC and 40 minutes on the Cloud Platform. This was the reason why I decided not to integrate the training phase inside of the C++ program.

[1] More details on the Microsoft COCO dataset in the references

3.4 Dataset and Annotations

3.4.1 Dataset

How a dataset is managed in a deep learning project is crucial. Some little oversight could lead to data leakage and so to wrong results. First of all the images requested as test images have been removed from the original dataset constituting the test dataset (the one on which to perform the last test). Once the results have been obtained, no changes should be made to the model, based on results obtained. Then the filtered dataset is split by the Python code into two subset: training dataset and validation dataset. The first one is actually used to train the system feeding it to the neural network, while the second one is used to calculate accuracy during training (for example to choose when to stop training) and later to fine tune the hyper-parameters of the system, trying to improve performance.

For this project two datasets have been used: the Kaggle dataset[2] (1461 images) and the MAR dataset[3] (4775 images). For the first one no test data is present, for the second one it is present a quite big test dataset.

3.4.2 Annotations

The bounding boxes for each sample for the training and testing dataset have been collected in group. I decided to use the powerful online tool called SuperAnnotate[4] and with 3-4 hours per person we managed it. I exported them as a COCO json file[5]. The annotation file is needed during the pre-processing phase as the program should alter it when it creates new image during data-augmentation.

3.5 Training code

I have chosen to code the model and training of it with PyTorch framework. I think it is a good choice if you want to go deep in the customization of the system, opposed to Keras which is too high level to achieve a specific result. Tensorflow, for me, is too complex, giving no advantages over PyTorch.

I have chosen to use SGD as an optimizer and I have used *StepLR* to decrease learning rate. Two classes were set for the last layer, one corresponding to background and one to the boat class. I trained the model for 10 epochs.

[2]https://www.kaggle.com/clorichel/boat-types-recognition/version/1
[3]http://www.diag.uniroma1.it// labrococo/MAR/classification.htm
[4]https://superannotate.com/
[5]More informations here: https://cocodataset.org/

Some additional modules[6] where taken from the official Github page of PyTorch Vision to speed up the training phase. Maybe I could have used PyTorch Lightning instead, it would have resulted in a cleaner code.

The model is finally exported and saved locally as a .pth file. This simplify the inference phase.

3.6 Folder structure

To use the provided notebook and run it as it is, you should respect the same directory structure as designed. The structure to use is the one provided in Figure 5.

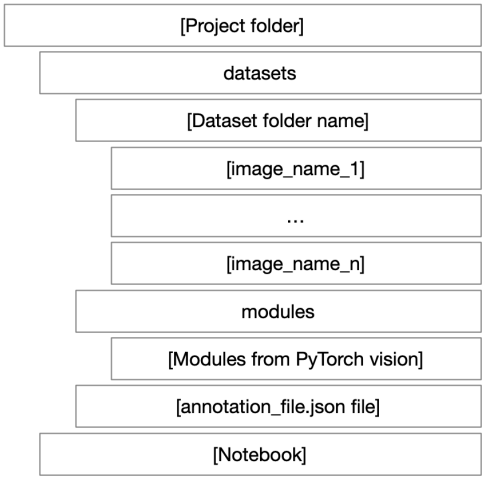


Figure 5: Directory tree for Google Drive

The only variables in the notebook to be changes are:

- *project_path*: The path of your Google Drive folder where you have the files for the training (beginning with "/content/drive/My Drive/"). As already clear from the Figure 5 the project_path should contain the datasets folder with inside the folder names dataset_name and the modules folder with inside some helper functions.
- *dataset_name*: The dataset folder name.
- *annotation_name*: The annotation json file name

[6] Modules available from here: https://github.com/pytorch/vision/tree/master/references/detection

4 Inference

4.1 Python integration

There were two paths to follow to use a model trained with Python inside of a C++ program:

- **ONNX export format**: stands for Open Neural Network Exchange[7] and it is an open format for Machine Learning models which allows for the model exchange between different frameworks. PyTorch has a specific method to export the trained model in this format and OpenCV has the correspondent method to import it into its neural networks module.
- **Python embedding**: in this case we use Python as a service[8]: the C++ program calls some methods from a Python file and asks the Python Interpreter to execute them. I have chosen this way as I thought it would have left me more freedom.

4.2 Evaluation

4.2.1 Intersection over union

Intersection over union (IoU) is used to compute similarity between two arbitrary shapes (volumes) A, B

$$IoU = \frac{|A \cap B|}{|A \cup B|}$$

4.2.2 Evaluation of the test dataset

To compute the IoU of an image I have iterated through all the bounding boxes and at each of them I have searched the from the ground-truth boxes and I have kept the one with the best IoU score. Obtained the best ones I have computed the average.

To compute the IoU of all the test set I have simply calculated the average of the image specific IoU.

[7] https://onnx.ai/

[8] https://docs.python.org/3/extending/embedding.html

4.3 Folder structure

To allow the C++ program to work some directories hierarchy should be kept or some parameters should be modified in the code. Some folder's position is compulsory, others are only reported as a suggestion. You can see the whole tree in Figure 6.

The folders and files not marked with the circle can be freely chosen with the command line parameters (more on this later).

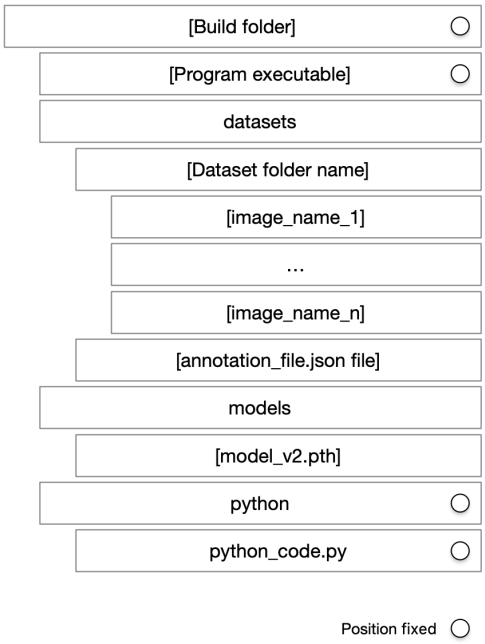


Figure 6: Directory tree for C++ program

5 How to use

To run the inference phase you have to install with pip some packages in the global default installation of Python. To check which version is seen by the program, run it. One of the first debugging output is the Python version used. Packages to install are: *numpy, torch, torchvision, Pillow, pycocotools, libmagic*. In the *CMakeLists* file should be added *include_directories(/usr/.../Versions/3.9/Headers/)* where the directory to include is the folder inside which Python.h is included. This header file is typically included in Python-dev versions.

5.1 Command line arguments

- **mode**: the mode to decide which program to run. This is the only positional argument. It can be:
  - prepro: runs the pre-processing of the dataset
  - eval: runs the inference for only one image
  - evalAll: runs the inference for the whole test dataset
- **image**: image on which to run eval mode
- **folder**: folder corresponding to the test dataset. It can also be a path relative to the executable folder
- **output**: output folder where images pre-processed by the prepro mode are put. It can also be a path relative to the executable folder
- **annotation**: annotation json file name or path. It can also be a path relative to the executable folder. The default value is datasets/kaggle_annotations.json
- **annotationoutput**: annotation json file name or path where to store the new annotation file after pre-processing. It can also be a path relative to the executable folder. The default value is datasets/kaggle_annotations_prepro.json
- **model**: model file name or path. It can also be a path relative to the executable folder. The default value is models/model_v2.pth
- **dataset**: folder where the images for pre-processing are located. It can also be a path relative to the executable folder. The default value is datasets/kaggle_dataset

5.2 Examples

5.2.1 Pre-processing

```
prepro
-folder=datasets/kaggle_dataset
-output=datasets/kaggle_dataset_prepro
-annotation=datasets/kaggle_annotations.json
-annotationoutput=datasets/kaggle_annotations_prepro.json
```

5.2.2 Single inference

```
eval
-image=datasets/kaggle_dataset/aida-ship-...-sea-144796.jpg
-annotation=datasets/kaggle_annotations.json
-model=models/model_v2.pth
```

5.2.3 Test dataset inference

```
evalAll
-folder=datasets/kaggle_dataset_test
-annotation=datasets/kaggle_annotations.json
-model=models/model_v2.pth
```

6 Results

6.1 Numerical results

The most important result obtained is the proof of this project's aim: pre-processing with some computer vision standard technique improves the training capability of the network. I have tested the model on the kaggle_dataset without any pre-processing and then with my pre-processing. In Table 2 you can see the comparison. The second dataset is MAR. The model was trained indipendently one from the other as I consider the two types of datasets too different. Anyway the Kaggle model can run pretty well even on the MAR dataset without having seen any image similar to those received for inference.

Dataset	IoU
Kaggle dataset without pre-processing	0.296879
Kaggle dataset with pre-processing	0.848984
MAR dataset without pre-processing	0.416525
MAR dataset with pre-processing	0.595419

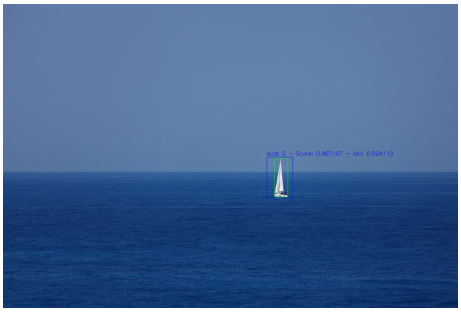
Table 2: Datasets results comparison

6.2 Graphical results

In Figure 7 it is possible to see some results of the entire process. On the left there are the ouput images from the model without any pre-processing. On the right the model with the pre-processing described in Section 2. Green bounding boxes are the ground truth, taken from the annotations file, the blue ones are those predicted by the neural network. On top of the box there are two important values. IoU and score which is the confidence of the network for that box. More results can be seen in the Appendix A.2.

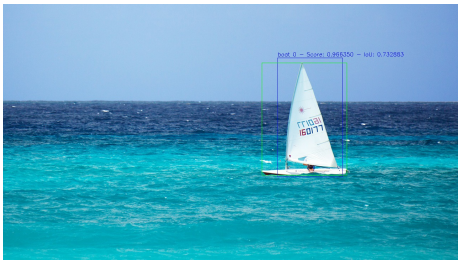
(a) Model without pre-processing

(b) Model with pre-processing



(c) Model without pre-processing

(d) Model with pre-processing



(e) Model without pre-processing

(f) Model with pre-processing

Figure 7: Results from the Kaggle test dataset

7 Conclusion and Improvements

As stated in Section 6, the use of standard and already well consolidated computer vision algorithm is a great idea because, as seen in Section 6, it improves performance by a pretty big percentage.

I have identified a few possible future improvements to this project:

- During the pre-processing after having found the clusters with k-means I could add another step in which to group pixels inside each cluster into more segmented clusters thanks to proximity. Maybe using the *connectedComponents* OpenCV method. This will solve one issue of my segmentation method: parts of the image which are clustered together with water or sky is considered water and then blurred. With this improvement this would be avoided. I have implemented a prototype of this pipeline but I realized that it was too complex from a computational point of view considering I have to compute it on thousands of images.

- Improve the selection of the color area that is considered to be water or sky. This could be better tuned manually, as done for this project or maybe could be implemented with a simple neural network or some other machine learning technique, or simply with an automated version of the manual one.

- For neural networks data is the key and here we have not enough samples to train. MAR dataset is quite big but is not suitable to train a model for the Kaggle dataset, images are very different. To increase performance we should increase the dataset using images similar to those the neural network is expected to receive in input during inference.

- Increase the number of parameters customizable from the command line. It was left with the minimal number of them to avoid complex syntax but it could be easily made.

- Introduce a library to fine tune the hyper-parameters of the NN. These were adjusted manually and not that much (that is not this project's purpose). Something like Optuna would make a big difference.

- Instead of fine-tuning the last part of the network with a bigger effort from the computational point of view I could unlock all (of a subset)

NN's weights and let the system be trained entirely. This could be very demanding but it could help increase the final results.

- Improve the pre-processing trying with different types of alterations instead of the gaussian blur. It could also be implemented some data-augmentation technique base on cropping. The drawback is that the annotation file should be edited as well.

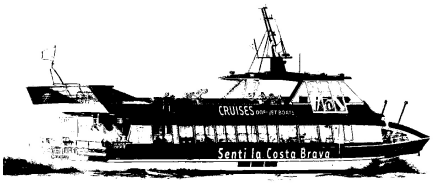
A Appendix

A.1 More masks

(a) Original image

(b) Binary mask

(c) Original image



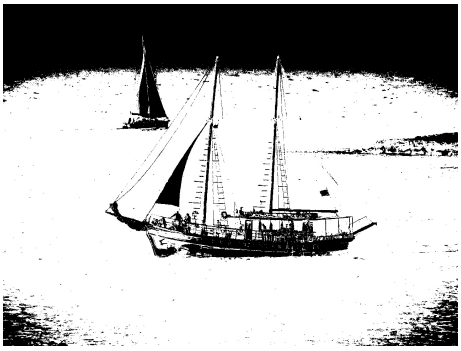
(d) Binary mask

(e) Original image

(f) Binary mask

Figure 8: More binary masks

(a) Original image



(b) Binary mask

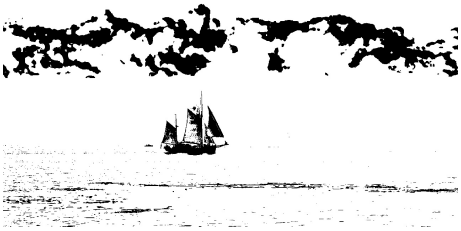
(c) Original image

(d) Binary mask

(e) Original image

(f) Binary mask

(g) Original image



(h) Binary mask

Figure 9: More binary masks

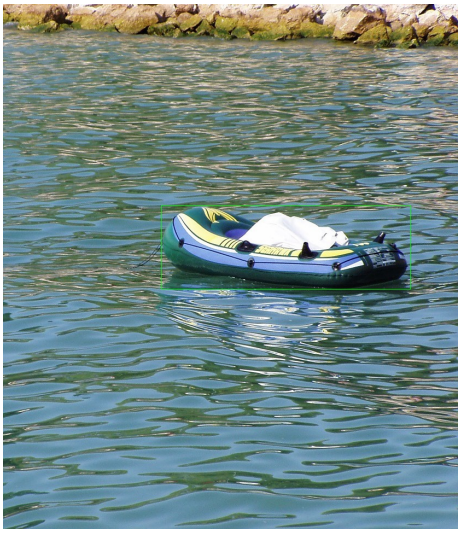
A.2 More results

(a) Model without pre-processing

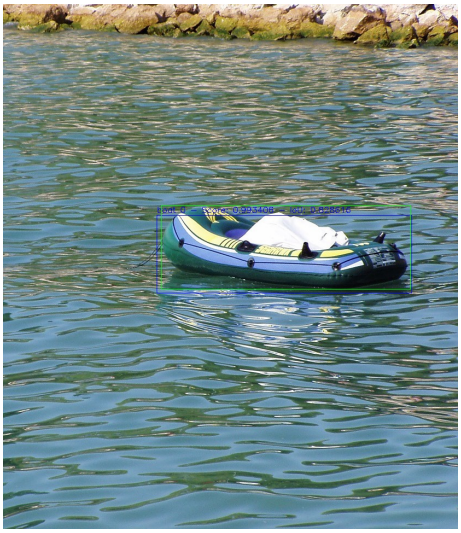
(b) Model with pre-processing

(c) Model without pre-processing

(d) Model with pre-processing

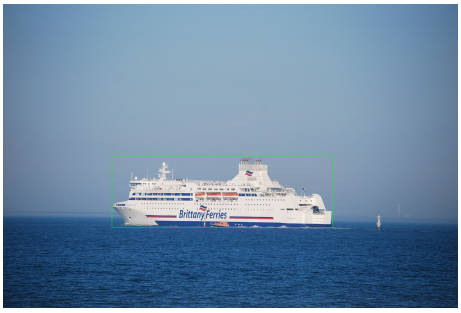


(e) Model without pre-processing

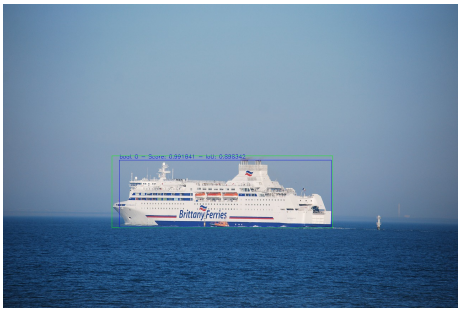


(f) Model with pre-processing

Figure 10: More results from the Kaggle test dataset



(a) Model without pre-processing



(b) Model with pre-processing

(c) Model without pre-processing

(d) Model with pre-processing

(e) Model without pre-processing

(f) Model with pre-processing

(g) Model without pre-processing

(h) Model with pre-processing

Figure 11: More results from the Kaggle test dataset