Model comparisson for Pool Detection in aerial images

Guilherme Ribeiro
Faculdade de Ciências da Universidade de Lisboa
fc53699@alunos.fc.ul.pt

Rómulo Nogueira
Faculdade de Ciências da Universidade de Lisboa
fc56935@alunos.fc.ul.pt

Abstract

Standard object detection methods are widely used for various tasks. Meanwhile, with the appearance of transformers, their use for object detection has become more and more popular. We study this new aproach, by experimenting and comparing the different methods. We use pre-implemented versions of the various algorithms and train them for this specific task. We experiment on a custom dataset which results from merging two different datasets publicly available on the Kaggle platform.

1. Introduction

Pool detection has some real life applications. From water collection in fighting fire, correct tax assesment, health reasons and mosquitos control. To achieve such a task, Deep Neural Networks have played a crucial role completing the task at ease. Previous experiments have been done using the most "traditional" object detetion technology such as YOLO, Faster R-CNN, Masck R-CNN etc [5]. In this experiment we compare the use of the two most popular convolutional models, Faster R-CNN and YOLO to a transformer vision model, DETR.
To achieve this task we'll use aerial images. With this, our goal is to prove that the use of transformer vision models is equal in terms of performance to convolutional models.

1.1. Problem Statement

As said before, to perform such task we take use of aerial imagery combined from two different datasets. The first one being related to the Algarve landscape [2] and the other to general satalite images [1]. Both of these datasets can be found on the platform Kaggle. The use of two different datasets allows us to give a broader view to the model, making it learn the task for efficiently and in overal more robust. These datasets are composed of images, and XML files with the correpondent labels and bounding boxes. The images where pools were abscent, had no corresponding XML file.

1.2. Main contributions

- Provide a comprehensive pipeline to train three different models for this task.
- Comparison of traditional models with transformers based models.
- Scripts that can be used to extract principal features from a raw dataset *(image, label)* attending specific features of each model.

1.3. Evaluation

In order to evaluate the models, we decided on using the most common evaluation metrics:

- **Precision**
$$Precision = \frac{TP}{TP + FP}$$
- **F1-Score**
$$F1 - score = \frac{2TP}{2TP + FP + FN}$$
- **Recall**
$$Recall = \frac{TP}{TP + FN}$$

We have chosen to refrain from relying on accuracy as a primary metric, given its limited ability to provide a comprehensive assessment of object detection performance. Instead, we prioritize the metrics bellow such as precision, recall, and F1-score, which offer a more nuanced understanding of model performance in object detection tasks.

1.4. Model Selection

For each model, we'll use the following pre-implemented versions:

- Faster R-CNN from PyTorch [3]
- YOLO v8 from Ultralytics [6]
- DETR from Facebook [4].

All the above models have pre-trained versions but we'll train them ourselves from scratch.

2. Technical Approach

2.1. Data preparation

First, we developed a script that is able to extract all the required information from all the datasets and merge it into one. This script creates one new folder with two folders init.

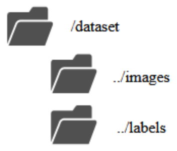

Figure 1. Data structure

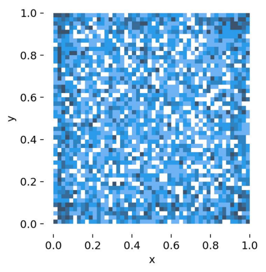

Figure 2. Pool distribution in the final dataset

However, this data structure only works with Faster R_CNN. The other models, YOLO and DETR, work with different data structures and so it was necessary to develop a script for each of them that transformed the original data structure into another structure that followed the requirements of each of the models.

For the YOLOv8(nano) model, first, we had to reshape the bounding boxes values into the required shape(box-center-x, box-center-y, box-height,box-width). Afterwards, these labels are put in the corresponding .txt files. The dataset was split with the same ratios as before.
The YOLO dataset folder has the following structure:

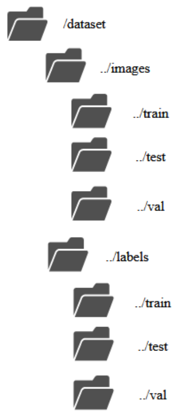

Figure 3. YOLO data structure

For the DETR model we created a script that extracts 70% of the dataset for training, 20% for validation and 10% for test. For each folder it was necessary to create an *annotations.json* file that mapped each image with it's labels and bounding boxes. The model's final data structure was:

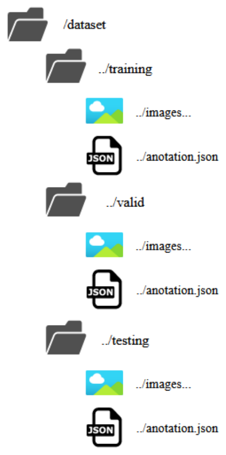

Figure 4. DETR data structure

Also, to speed up the training, we decided to resize all the images to 224x224. The necessary adjustments were made to the bounding boxes.

2.1.1 Dataset Class

Once again, the implementation of the Dataset Class which contains all the information about the images and their labels are different in the 3 models.

For the **Faster R_CNN**, it was implemented a Pytorch dataset class, *PoolDataset*. This class is crucial to the rest of the experiment. From this class, we're able to loop through our data retriveing the images and the correponding bounding boxes.

YOLO has no need for a dataset class, due to the data being already sepparated and the data is automatically fetched from the folders.

For the **DETR**, it was implementend a COCO dataset class, *CocoDetection*, that is also resposible to retrieve correctly the images and it's labels (respecting the COCO structure).

2.1.2 Training and Evaluation

Before getting into detail about each model, there was one technique that we used in all of them to see if the model was training correctly. We first created a small sample of data and trainend a model with it. Aftewards, we we validating with the same sample. This allowed us to assert whether the model's hyperparameters we correct and the model was capable of learning the task.

Also, every model was trained on 100 epochs.

Faster R_CNN

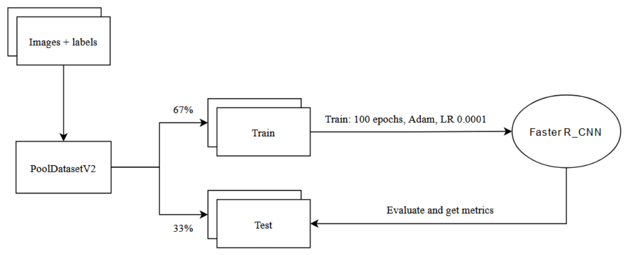

Figure 5. Training and evaluating process of Faster R_CNN

Before starting the training process itself, it was necessary to setup all the environment first. This setup consisted in:

1. Instantiate the dataset class (*PoolDatasetV2*)
2. Split into data for training (0.67%) and for testing (0.33%)
3. Create the data loaders (*train_loader* and *test_loader*)

Then we load the *Faster R_CNN* model from *PyTorch* using *ResNet-50* as backbone and we remove the "head" of the predefined model. Next we add a new predictor tailored to classify two specific classes (pool or not pool).

Finally, we define the *Adam* optimizer with a *learning rate* of 0.0001. Therefore, and due to the limitations of Kaggle Kernels and Google Colab, we trained the model for 80 epochs and retrained again for 20 more epochs.

Approximately 15 hours later we had our model trained and ready to be evaluated.

For the evaluation phase, we defined a couple of auxiliary functions which helped in the comparison of the predicted boxes and the ground truth boxes. This functions made comparison using only the predicted boxes above a mininum threshold (0.7) and uses the IoU (Intersection over Union) which is a method that measures the overlap between the two bounding boxes.

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}} \tag{1}$$

Finally, we loop trough the *test_loader* and the function continiously updates the TP, FP and FN giving as final ouput all the metrics.

YOLOv8 (nano)

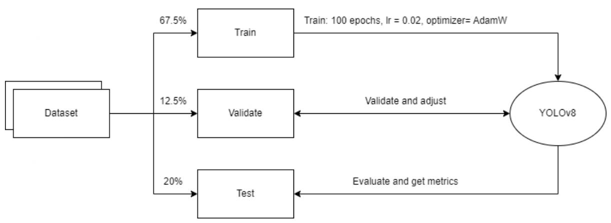

Figure 6. Training and evaluating process of YOLO

Before training the YOLO model, a config.yaml file was created. This file contained information about the path to the dataset, the train, validation and test folder, and lastly, all the classes existing in the dataset. In our case, only one class needed to be inputed, *1: Pool*, because YOLO predifines *0: Background*. We opted to import the *yolov8n*, so the nano version. This allowed us to train a YOLO model faster while having restricted computation power. Following this we started the training process. To train the YOLO, after many trials, we agreed on a learning rate of 0.02 and the final learning rate as 0.0002. As an optimizer, we choose the AdamW.

The Ultralytics API allows us to save the last and the best model after each epoch. Dus, in the end, the results obtained were through the best model and not the last one. Also, this feature allowed us to train the model in sepparate days.

During training the loss decreased in a good rate, with no need for learning rates adjustments.

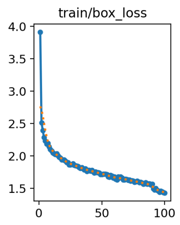

Figure 7. YOLO train/box loss

The validation and test is automatically implemented by the Ultralytics API. In the end, the API creates a folder with all the information and metrics.

DETR

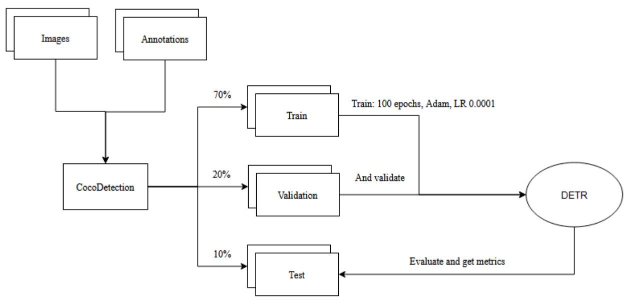

Figure 8. Training and evaluating process of DETR

To star with, a *CocoDetection* class is created. This class extends the torchvision.datasets.CocoDetection class with the purpose of integrating an image processor for preprocessing images and annotations.

Afterwards, an image processor is implemented by taking use of the DetrImageProcessor. This is where we load the *facebook/detr-resnet-50*.

Next, we load the different datasets (train, validate and test), with the *CocoDetection* class. These datasets will be later loaded into Pytorch Dataloaders. For each dataloader, we established batches of 16 images, along side with a collate function and in the case of training, the shuffle flag is turned on. Next, the *DE:TR* is loaded through the *Hugging-Face* API. With this, a Detr class is created with the definition of the various training steps, such as the foward pass and the optimizer configuration.

The following hyperparameters were defined:

- Learning Rate: 1e-4
- Learning Rate backbone: 1e-5
- Weight Decay: 1e-4
- Optimizer: AdamW

In the end, the model was saved so that it could be used later for testing.

For testing, a small script was designed where we incorporate previous implemented functions.

3. Results

The metrics from each model, can be found in the following table:

Model	Precision	Recall	F1-Score
Faster R-CNN	0.9950	0.9736	0.9842
YOLOv8(nano)	0.9289	0.7140	0.8074
DE:TR	0.7721	0.9009	0.8316

3.1. Model prediction Examples

These are some examples of the predictions produced by the models:

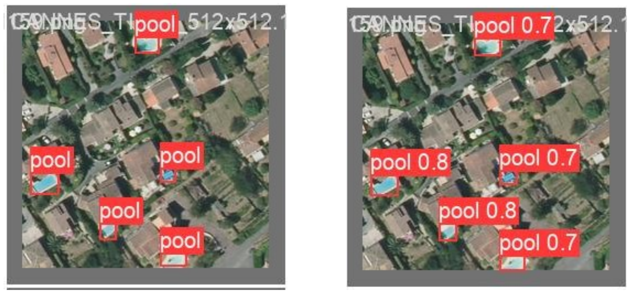

Figure 9. YOLO label and prediction example

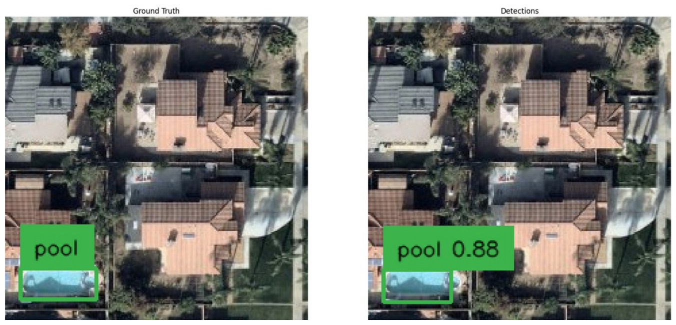

Figure 10. DETR label and prediction example

It is worth noting that our labels had a flag for difficulty, but we decided to ignore it and treat each image as the same. This led to some pools not being detected either because they were in a shadow or had some vegestation over it. This can be seen in Figure 12. Yet, our results still prove to be more than decent!

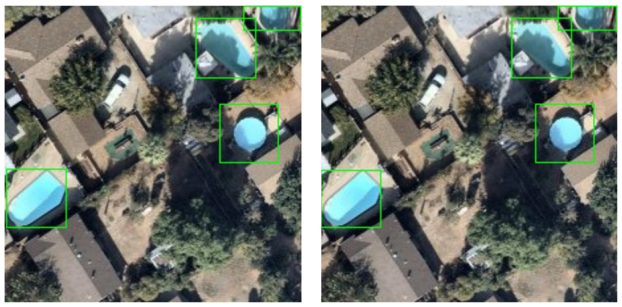

Figure 11. Faster R-CNN label and prediction example

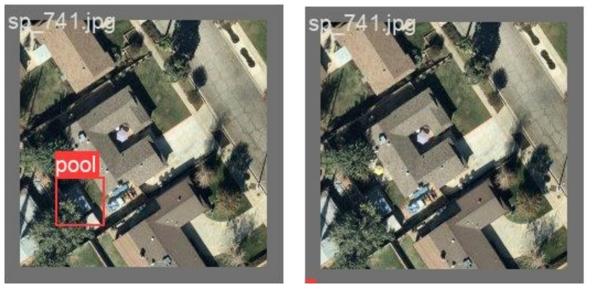

Figure 12. Example of Failed prediction - YOLO

3.2. Model Comparisson

As it can be seen from the table, the first two models were able to reach a precision above the 0.9. With Faster R-CNN reaching the highest score of 0.9950. In all honesty, DETR performance might have been affected by the fact that we were running out of time, so we didn't manage to tune it as well as the others.

4. Conclusion and Future Work

In conclusion, we trained various three well known object detection models. DETR ended up having a poor performance compared with the other two for this specific task.

Futher work would involve better tunning the models, adapt the model to the so called "difficult" situations and fine-tunning the models to other tasks and asses their perfomance.

5. Acknowledgment

We thank the authors of the two datasets for making them publicly available. We would also like to thank the professor from our Deep Learning class, where this project was designed, for all the support during and off classes.

References

[1] Cecília Coelho. Swimming pool detection in satellite images, 2024. https://www.kaggle.com/datasets/cici118/swimming-pool-detection-in-satellite-images. 1

[2] Cecília Coelho. Swimming pool detection - algarve's landscape, 2024. https://www.kaggle.com/datasets/cici118/swimming-pool-detection-algarves-landscape. 1

[3] PyTorch. Faster r-cnn, 2024. https://pytorch.org/vision/main/models/faster$_r cnn.html$. 2

[4] Facebook Research. Detr: End-to-end object detection with transformers, 2024. https://github.com/facebookresearch/detr. 2

[5] Anthony Taing. Swimming pool detection from aerial imagery. 2022. https://cs231n.stanford.edu/reports/2022/pdfs/16.pdf. 1

[6] Ultralytics. Yolo v8, 2024. https://github.com/ultralytics/ultralytics. 2