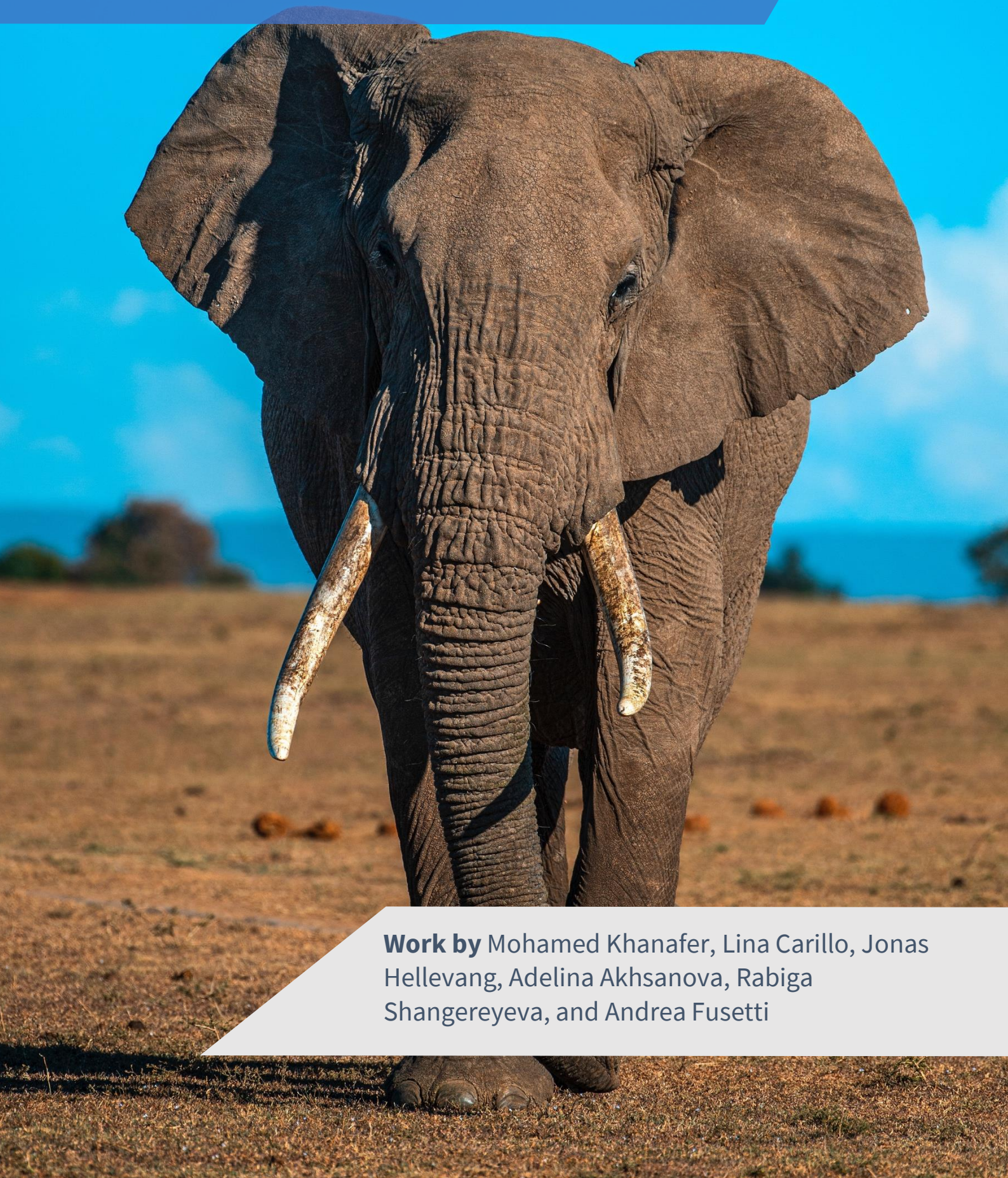
Computer Vision
Protecting the Endangered



Work by Mohamed Khanafer, Lina Carillo, Jonas Hellevang, Adelina Akhsanova, Rabiga Shangereyeva, and Andrea Fusetti

Table of Content

Introduction

What this report is

We have decided to combine our notebook with most of our report and thus we believe that the Colab notebook shared is more detailed in terms of explanations and development of our work. We nevertheless in this document provide a more general overview of our work. But we would advise to dedicate more time on the notebook.

Use Case: Saving Elephants

Lately, the large mammal populations have been declining dramatically each year. Especially concerned are the elephants, the largest land animals are slowly falling into the extinct list of animals. They are especially being poached for their ivory tusks which are used for jewelry and luxurious furniture elements. Ivory tusks trades are illegal but there is still a growing market for it in Asia, highlighting the continuous threat to the animal.

As a result, the estimated population of elephants decreased from 12 million to 400 000 in the last century alone according to the WWF. Moreover, an estimated 20,000 elephants are poached for their tusks every year.

National parks like Zakouma National Park in Chad, Akagera National Park in Rwanda and many other parks in different countries have been trying to prevent poaching for elephants but they traditionally rely on expensive options such as vehicle and foot patrols which put a financial strain on the reserves and parks that are home to elephants. (African Parks.org, 2020)

This is where our proposed solution using computer vision comes in. Our idea is to use real-time object detection coupled with readily available drones in order to partly solve problem of poaching and animal extinction in general. Indeed, with advancement in technology, drones are becoming cheaper and more powerful each year. And with the development of machine learning model in computer vision, we believe that we could develop performant models at affordable prices.

As a result the drone would be sending real-time streams to a control room where our custom-trained object detector could help with:

1. Precise control of population of endangered species;
2. Poaching control;
3. Control of movement of animals in order to save them from especially dangerous zones.

Our solution will be focusing on points 1 and 2 mainly as we will train our various models to recognize animals (especially elephants) from aerial shots.

Work overview

To reach a proper model for our use case, we have followed the following development steps:

1. We build a pipeline with images taken from normal angles and score it;
2. We build a second pipeline with images taken from above, aerial views and again assess the performance of the model and compare it to the first pipeline;
3. We build a 3rd pipeline for aerial view that we have taken from our drone and assess how the model is performing.

Our goal in creating various pipelines and models trained on different datasets is to be able to propose an efficient solution in the end.

Indeed, in pipeline 1, we focus on animal images taken from normal camera angles, to be able to assess the accuracy of recognizing animals from low angles. We focused on 3 animal classes: sheep, elephants, and horses.

We would then compare the model with the 2nd pipeline that we would train with images shot from above. We expect the images shot from above to be harder to identify for the model and thus wanted to check this hypothesis. In here, because images were harder to find, we limited the model to 2 classes: sheep and elephants.

We finally run a 3rd pipeline model with images taken from one environment and with 1 object class only that are sheep. We focus here on sheep because we build a model with our own data. Indeed, as will be explained, we have shot sheep images from a drone and built our dataset around this for this final model.

As stated, we will compare those 3 pipelines and conclude on some parameters and factors that should be taken into consideration for the development of a better solution for our use case.

Pipeline I

Preparation of the Dataset for Pipeline I

Data Sources

Because we want to train our first pipeline on "normal angle" images, we tried finding rich animal datasets that were of quality. We did find a very good dataset on Kaggle that contains around 28,000 medium quality animal images divided into 10 categories: dog, cat, horse, spider, butterfly, chicken, sheep, cow, squirrel, elephant.

We decided to go ahead with this dataset as it had been checked by humans already and thus errors or bad images would have been filtered out.

We also found another dataset containing sheep images of quality.

We were interested in including sheep because for our live demo and presentation, we had as a goal to fly our drone over a farm with sheep and see how our model would perform. This could give us a good benchmark as of how the model would perform over a national park to trace other animals of interest like elephants in our use case.

So, the first pipeline's goal is mainly to see how animal images taken from "normal angles" would score. We thus decided to run a multiclass pipeline with images of elephants, horses, and sheep.

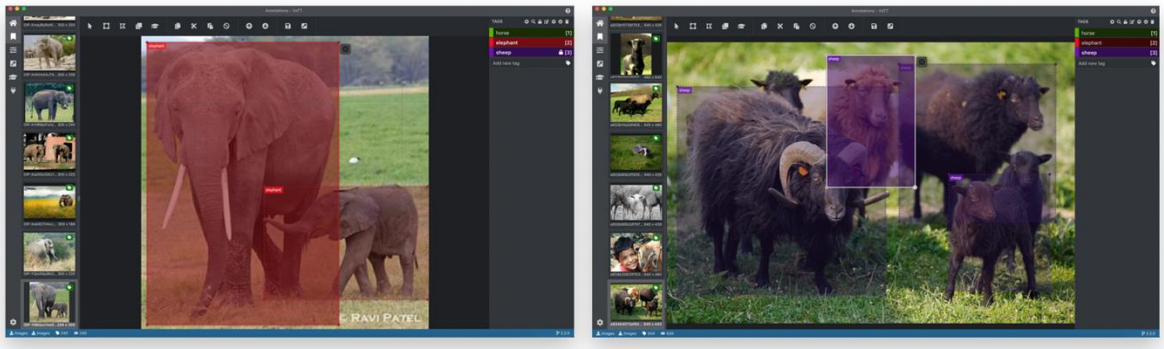
Data Annotation and labelling

We have greatly used and followed the code of Anton Mu. We thus, as he suggests in his implementation, used Microsoft's Visual Object Tagging Tool (VoTT) to annotate our images. We created 3 label classes and anotated the following:

- 250 images for elephants;
- 210 images for sheeps;
- 260 images for horses

We tried our best to be as accurate as possible in order to have performant results.
Because we had ample images for each category, we judged non necessary to perform any data augmentation on this dataset. No pre-processing or feature engineering were carried neither because we judged that we had many different environments represented and wanted to see the effect this would have had on the predictive ability of the model.

The interface when we annotated looked like the following:



Once the annotation was done, we had for each image in the data a generated JSON file as well as a VoTT csv file. We run a code to transform those bounding box annotations into a YOLO format. This generated two output files: data_train.txt (located in the folder

TrainYourOwnYOLO/Data/Source_Images/Training_Images/vott-csv-export) and data_classes.txt (located in the folder TrainYourOwnYOLO/Data/Model_Weights).

Setting up the Repo and Google Colab

For our pipeline, as mentioned, we greatly used and were inspired by the code of AntonMu in which he explains a simple process to apply transfer learning to our custom dataset.

To be able to adapt it to our data and develop on his code, we forked it to our github, cloned it locally on our machine and carried out the necessary edits. After the edits needed were done (changing the model parameters, the train and test images, etcs.), we pushed the edits onto the Github repository. Then, we cloned this repository in our notebook to run the files we edited. Along the process, we pointed out to which files we edited and the motives of our decisions.

After each pipeline is run, we save the model weights as well as the train, test, and predictions into folders (Pipeline 1, Pipeline 2, and Pipeline 3 respectively). We then upload those folders into the repository to be able to access them later on if needed because we reuse the code for a next pipeline.

Training our Own YOLO using Transfer Learning

Next, we train the model on the data that we have annotated with VoTT.
To be able to do this, we have to download the pre-trained YOLOv3 weights and convert them to the keras format. The weights used here are pre-trained on the ImageNet 1000 dataset. The dataset contains images of animals and thus we expect the performance to be quite good for this first pipeline.
However, it was not trained on aerial pictures of animals, this is where our second pipeline will be interesting to test.

Training Performance Evaluation

The loading and converting of the YOLO weights took us around 20 minutes. And then for training the actual model, it took us around 110 minutes given the high number of images we are training on.
We set the training parameter at 51 epochs for training and the same for testing. And as can be seen from the graph outputed by the Tensorboard in our notebook, there was an early stopping after the 73th epochs, point after which the model was not learning anymore. Indeed, early stopping is a built-in feature that will automatically stop training the model if the validation loss is no longer getting better.

Running the Detector on unseen images

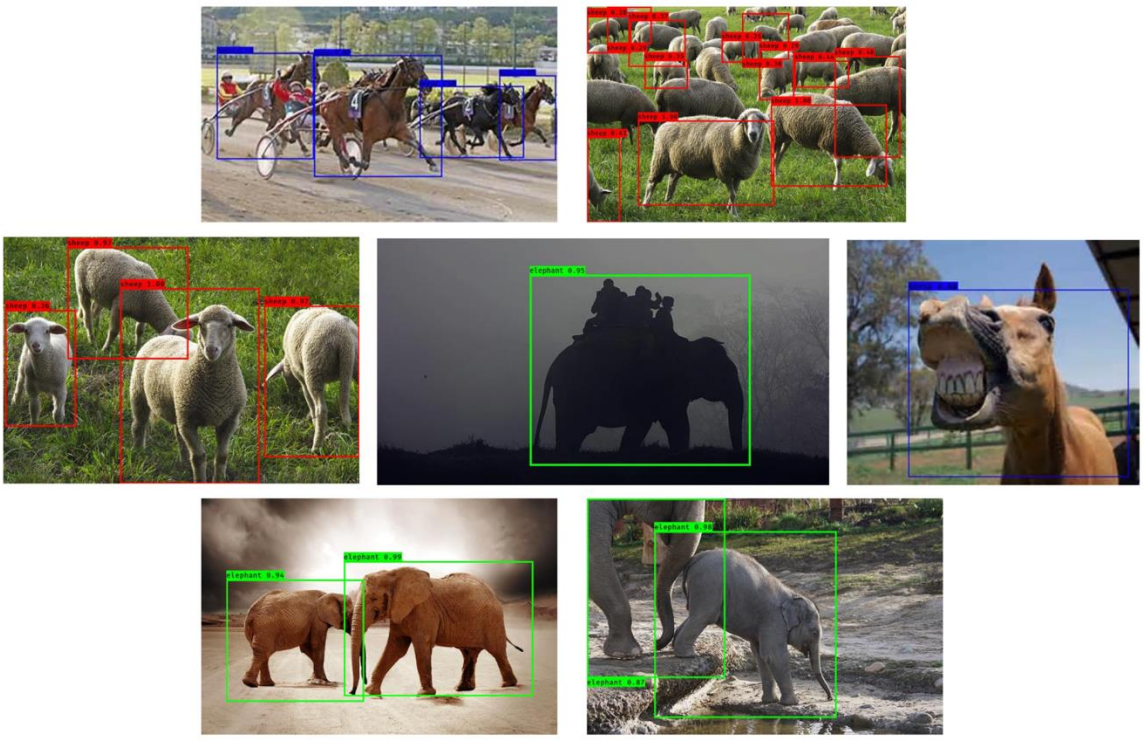
Finally, we were able to run our trained model with custom data on unseen test images. We have left 69 pictures to be tested on.
The model was then able to detect the animals and give us 5 information: the class, the confidence and the coordinates of the estimated location of the animal.

Calculating the mAP on Tested Images

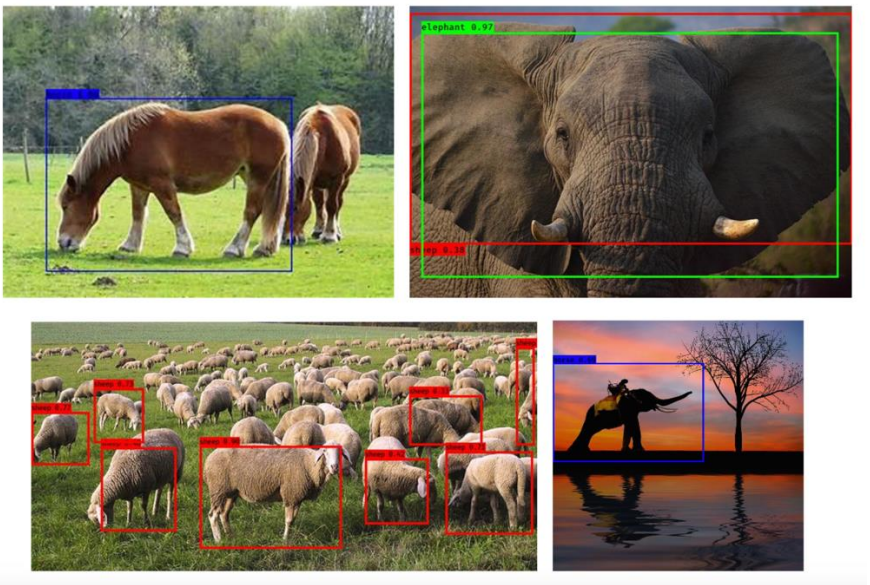
Downloading and Saving the Pipeline results

To be able to save our model's results and develop on them, we downloaded the results and saved them into the file Pipeline 1 because of the limited 12 hours runtime allowed on Google Colab.

Most of the tested images are well classified as can be seen in these selected good classifications:



And we also have some misclassifications like in here:



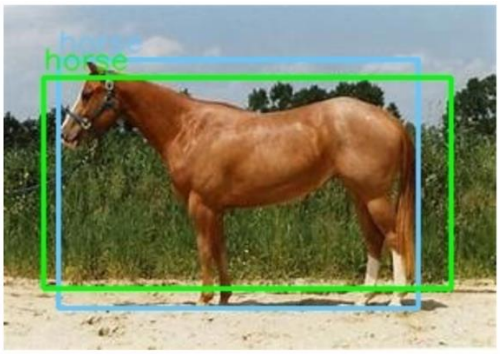
Metric used: mAP

For evaluating our models, we chose to calculate the mean Average Precision (mAP) on our tested images. We used Cartucho's implementation here.

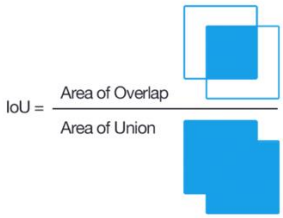
Explanation of the metric

This is how we understood the metric:

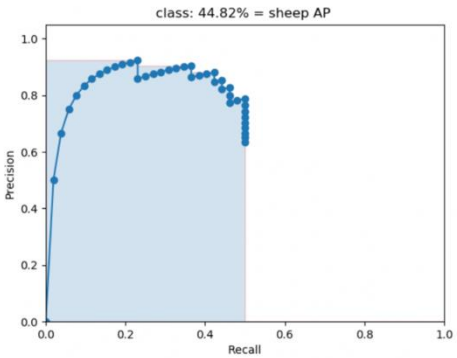
1. First for each image we find two boundaries:
   a. The ground-truth bounding boxes (in our case it is hand-labeled bounding boxes) and
   b. The predicted bounding boxes from our model.



2. Then we find the Intersection over Union (IoU) between these two boundaries. IoU is a ratio between the area of overlap and area of the union of boundaries defined in the previous step. When IoU is greater than 50% it is considered as positive true:



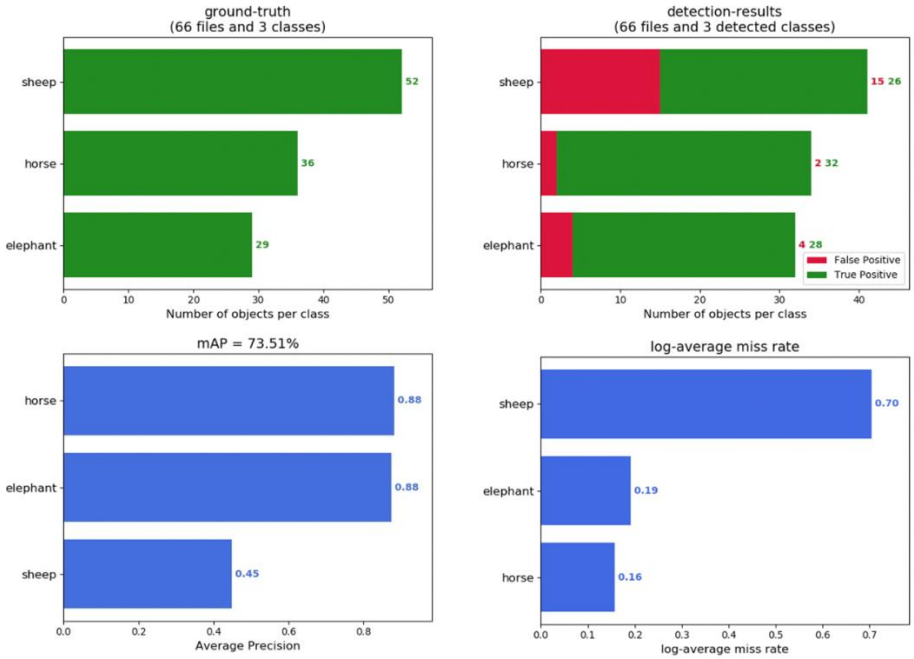
3. After that, we calculate the precision/recall curve. Average Precision is the area under the curve calculated by numerical integration.

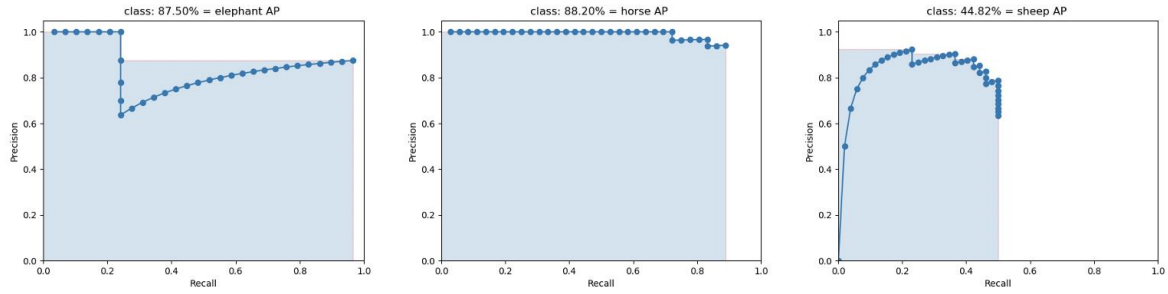


4. Finally, the Mean Average precision is just a mean of all Average Precisions calculated before. Its value can vary from 0 to 100%. The better performance of the model the higher its mAP.

Implementation of the mAP

As mentioned above, we runned the implementation from a github repository. We have to run it locally and upload the results here. Here is how we did it:
We first we created ground-truth text file for each image. Each text file contains lines in the following format: which are the names of the class (eg: “horse”), confidence value and the four coordinates for the bounding box boundaries.
Those text files were used to compare the output of our model and here are the full metrics we get:





Observations from the metrics

Based on these metrics, we can observe:

- Most of the objects were recognized properly as is highlighted by the metrics.

- We can also compare False Positive and True Positive for the 3 different classes. Class sheep show the highest rate of False Positive meaning many objects of the other two classes were recognized as sheep. This could be explained by the asymmetry of the number of objects. Indeed, the number of objects in class "sheep" were higher. That could be why the objects that were difficult to recognize could have been assigned to the “most probable” class – “sheeps”.

- mAP shows high results for classes of horses and elephants (88%) meaning the model performs well in detecting the pictures with those classes. For class "sheep" the performance is only 45%. We assume that the model performs better for images with one object or where objects do not overlap each other. Most of the pictures with sheep have herds, so it is difficult for a model to recognize every object. For the first pipeline, it could mean that the model performs better for solitary animals or animals in small groups.

- We also get the Log-average miss rate which is similar to mAP but refers to the objects that are not depicted. The Log-average miss rate is calculated by averaging miss rate at nine FPPI (False Positive Per Image) rates evenly spaced in log-space in the range. This gives a single number that summarizes the whole miss-rate vs. FPPI curve for easy comparison. High value at class “sheep” means that objects of class “sheep” were defined as “non-sheeps” and assigned to other classes. The reason for that can be the variety of sheep compared to other classes. For example, some sheeps had a darker color of fur that is why the model could define sheep as a horse or elephant.

Conclusion on Pipeline I

So as has been highlighted, with a proper dataset, the model is able to perform fairly well on classifying different animals. Although the sheeps get worse results than the other 2 animals, we believe that this could be fixed by increasing input data of sheeps in training. Also, further preprocessing could be done to solve it too.
But for our use case, what is more important, is being able to identify animals from an aerial point of view as our use case uses drones for object detection. This is the purpose of the 2nd pipeline that we dive in next.

Pipeline II

So as expected, the previous model's performance was almost perfect. And as mentioned, this is due in part to the fact that the weights used for transfer learning already were trained on animal images. Another reason is that the dataset we built was quite large and this pushed the performance higher.
Now, the second pipeline tries to tackle one of the main concerns for our use case: aerial shots. We would like to see how and if taking shots from an above perspective will affect the model.

To have a better focus, we reduce the number of classes to 2 and keep only elephants and sheeps.

Preparation of the Dataset for Pip. II

Data Sources

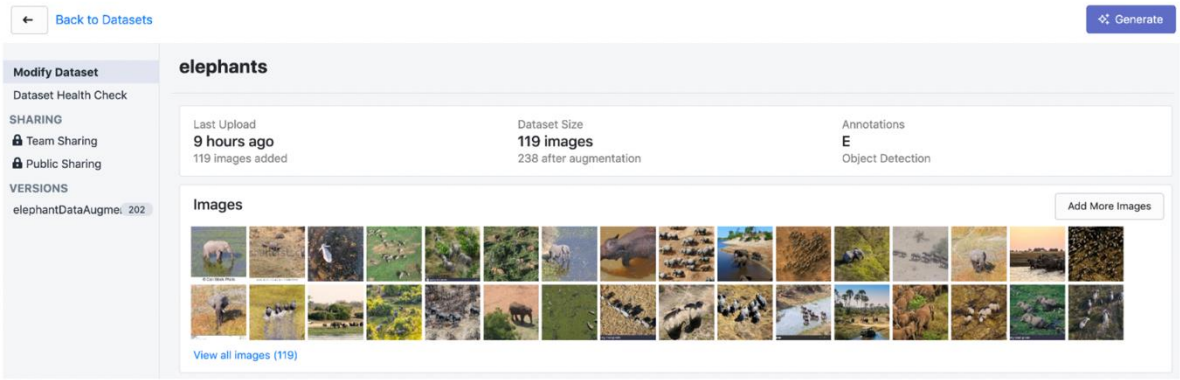
It was not as easy to find data of aerial shots for animals and thus we had to build the dataset from scratch. To do so, we took all the images of the Google search "Areal shots of sheeps" as well as most of the shots for the search "Areal shots of elephants". Those images amounted to around 100.
Then, we noticed that there were many videos of drone shots, and thus watched videos and captured images from them to constitute the second half of the dataset for each animal.
Finally, to increase the number of images and make the model more able to generalize, we applied Data Augmentation.

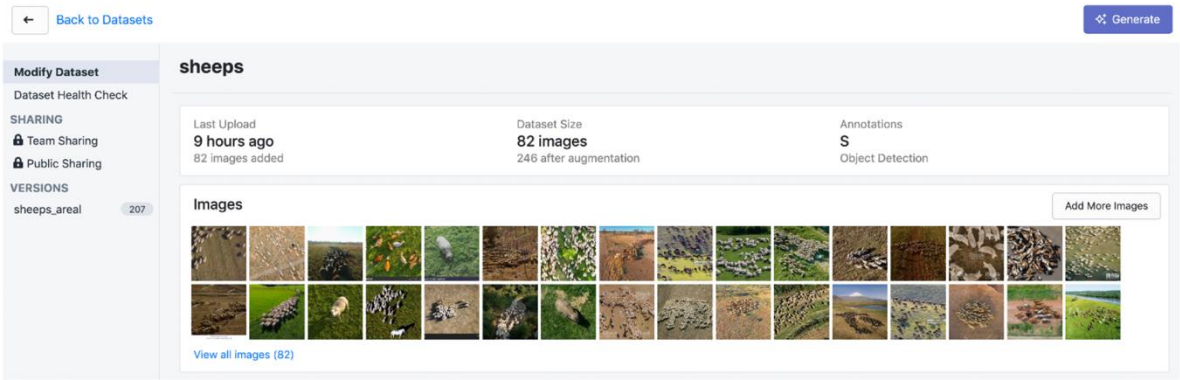
Data Augmentation and Pre Processing

For Data Augmentation, we used the great platform Roboflow. We performed the following augmentation:

- Rotating pictures (from few degress to 90 degrees);
- Editing brightness (from -35% to 35%)

The main reasons were that drone shooting comes from any angle and thus these would still be relevant images to feed in the model. On the platform, the datasets were transformed within minutes and ready to be downloaded:



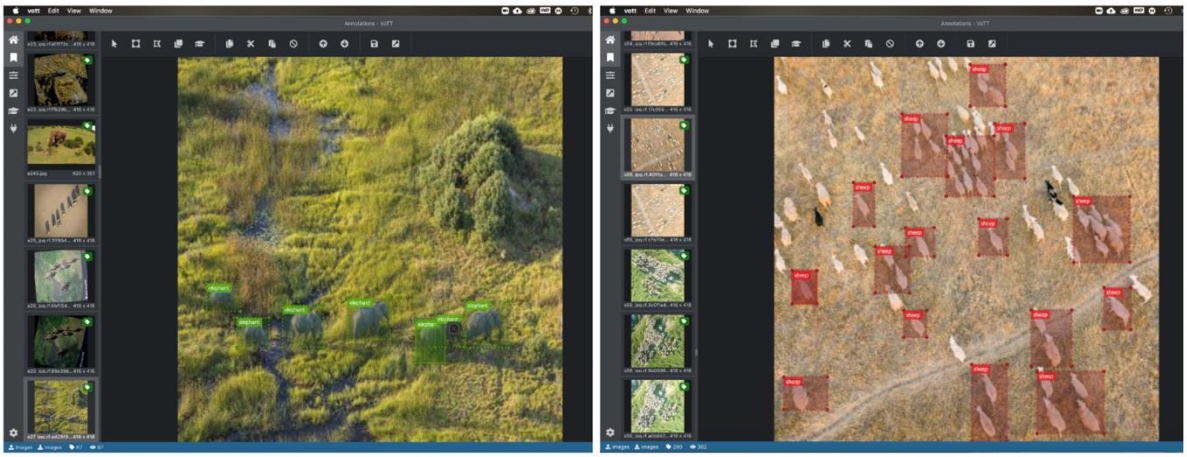


Data Annotation and labelling

We then followed the same procedure explained in the first pipeline, we annotated:

- 220 images for elephants
- 220 images for sheep
-

Again, we tried our best to be as accurate as possible in order to have performant results. The interface when we annotated looked like the following:



Recloning the Repository

With the outputs of the first pipeline now saved in the folder Pipeline 1, we re-used the structure of the initial repository to run the second pipeline. We thus uploaded the new dataset just described, initialized the weights again and retrained the new pipeline.

Training our Own YOLO using Transfer Learning

Again, we train the model on the data that we have annotated with VoTT. We followed the same procedure explained in greater above for pipeline 1.

Training Performance Evaluation

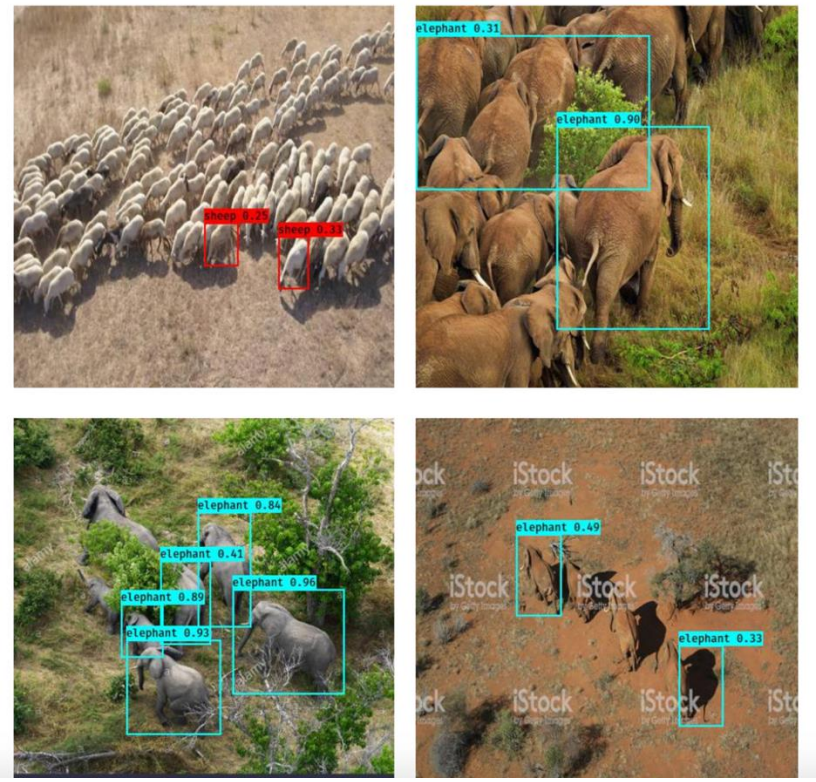
The loading and converting of the YOLO weights took us around 20 minutes again. And then for training the actual model, it took us around 90 minutes, slightly less than the first time. We set the training parameter at 51 epochs for training and the same for testing. And as can be seen from the graph outputted by the Tensorboard in the notebook, there was an early stopping after the 77th epoch, point after which the model was not learning anymore.

Running the Detector on unseen images

Finally, we are able to run our trained model with custom data on unseen test images. We have left 34 pictures to be tested on.

Some of the good results on a first look are:

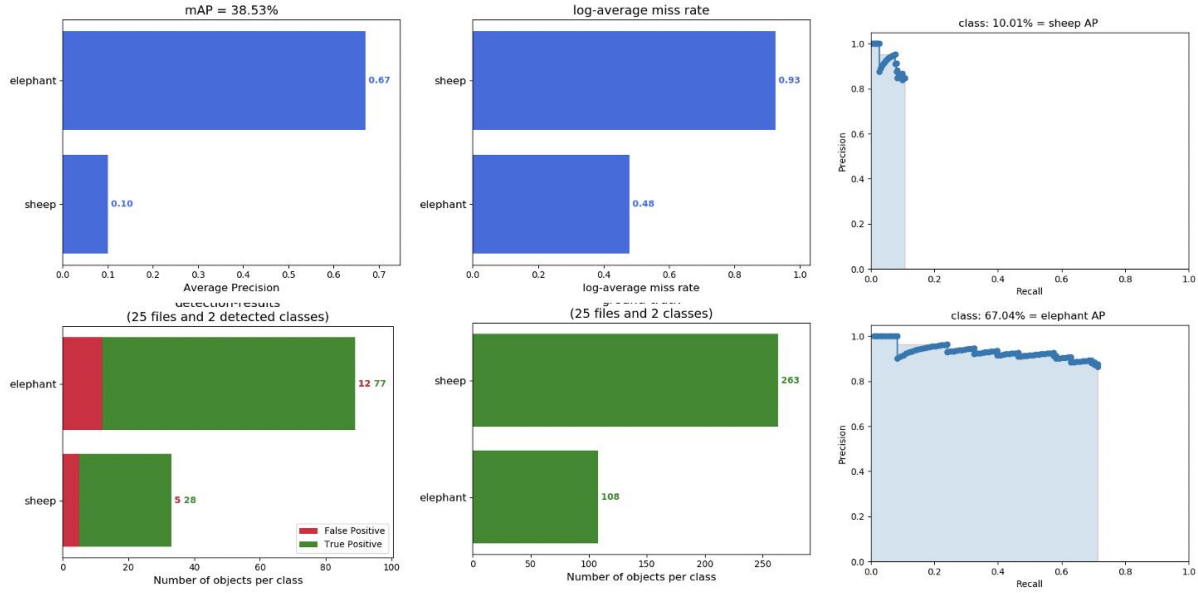
Some of the bad results on a first look are:



We then looked, like above at the metrics.

Calculating the mean Average Precision on Tested Images

We again ran the same procedure as in the first pipeline and got the following measures:



Conclusion on Pipeline II

Again here, elephants are easier to detect than sheeps. This, we believe, might be due to their distinctive shapes. Because even from an areal view, the outline of the animal looks somewhat similar. This is not really the case for sheeps seen from above.

To expand on these observations, we built the 3rd pipeline to fine tune our approach to our use case and provide a more robust solution.

Pipeline III

We built a final pipeline to prove one point: no matter the animal to be detected from an aerial point of view, with a defined environment and limited object classes to be detected, the model could be very powerful.
Because sheeps were the most misclassified class in the previous pipelines, we focused on implementing this last model on sheep detection from areal views. But again, the same procedure could be applied to any other animal.

Preparation of the Dataset for Pip. III

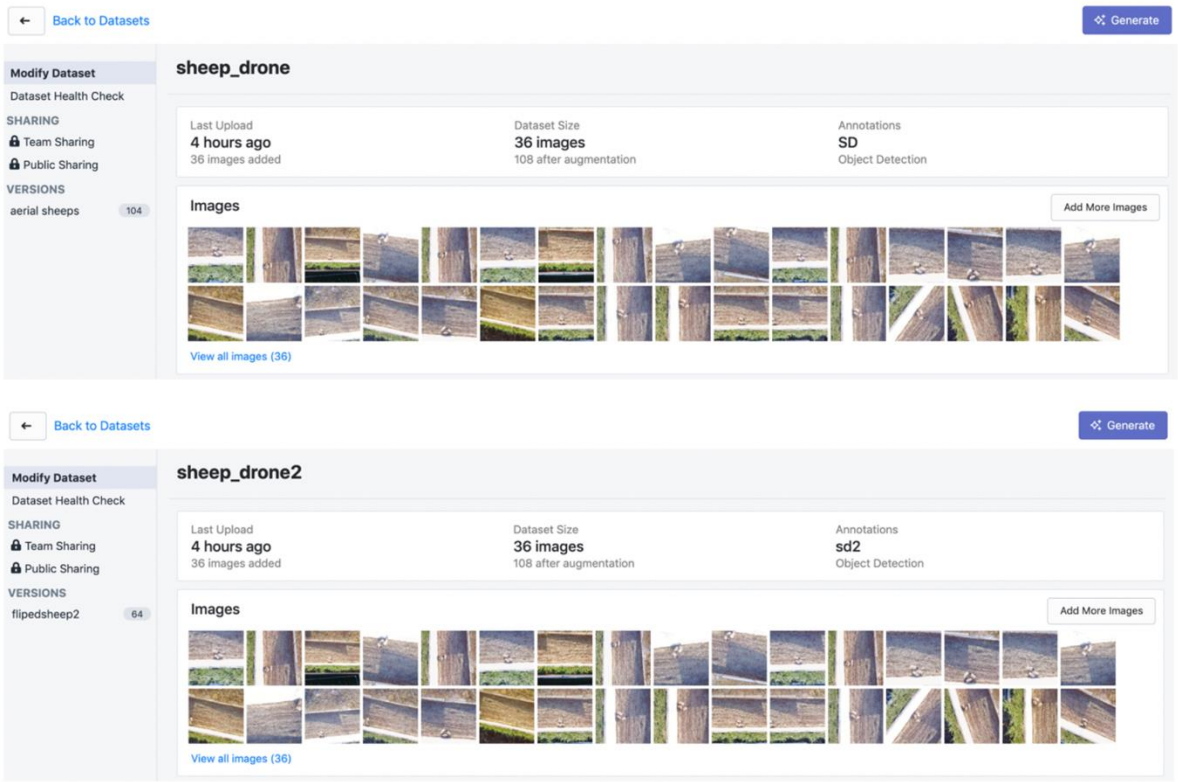
Data Sources

So as mentioned, we ourselves took pictures with a drone in a farm with sheeps. The goal is to have one defined environment and one animal to train on. This could later on be scaled to national parks and any animal of interest.

Over the course of 2 days, we took videos and pictures of the sheep and the farm environment for training the model.

Data Augmentation and Pre-Processing

For Data Augmentation, as in the previous datasets, we used the platform Roboflow.



We performed the following augmentation randomly:

- Rotating pictures (from few degress to 90 degrees);
- Flipping the pictures;
- Editing exposure (from -25% to 25%).

We ended up with a little over 200 images that we annotated the same way we did in the previous 2 cases.

Data Annotation and labelling

We then followed the same procedure explained in the first and second pipeline, we annotated:

- 200 images for sheeps.

Again, we tried our best to be as accurate as possible in order to have performant results.

Recloning the Repository

With the outputs of the first and second pipeline now saved in the folder Pipeline 1 and Pipeline 2, we re-used the structure of the initial repository to run the last pipeline. We thus uploaded the new dataset just described, initialized the weights again and retrained the new pipeline.

Training our Own YOLO using Transfer Learning

Again, we train the model on the data that we have annotated with VoTT. We followed the same procedure explained in greater above for pipeline 1 and 2.

Training Performance Evaluation

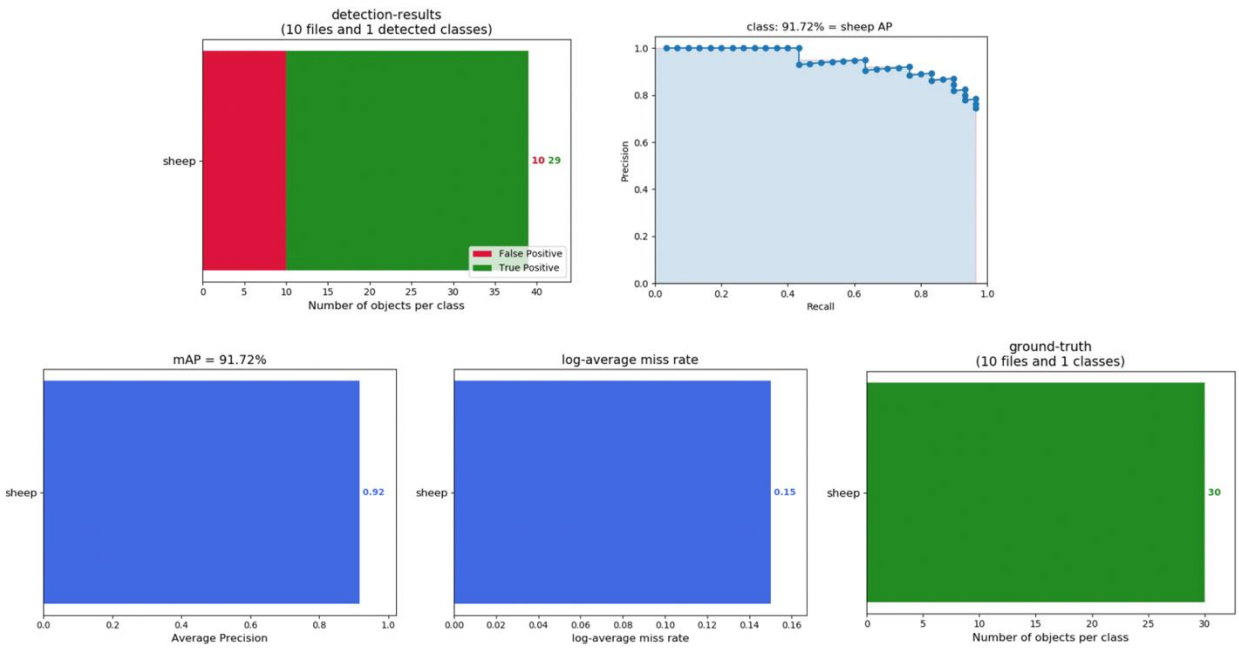
The loading and converting of the YOLO weights took us around 20 minutes again. And then for training the actual model, it took us around 60 minutes, less than the two first time due to the reduction in the number of trained images. We set the training parameter at 51 epochs for training and the same for testing. And as can be seen from the graph outputed by the Tensorboard in the notebook, there was an early stopping after the 70th epoch, point after which the model was not learning anymore.

Running the Detector on unseen images

Finally, we are able to run our trained model with custom data on unseen test images. We have left 10 pictures to be tested on.
All of the tested images turned out to be accurate as shown below:

Calculating the mAP on Tested Images

We again ran the same procedure as in the first pipeline and got the following measures:



The highest numbers and best performance are thus achieved here. While it might be true that we have trained our model on images coming from the same environment than the tested images, this was our goal here. We wanted to highlight that if we define an environment (here farm but could be a national park) and a specific animal to follow, we could easily achieve decent performance with transfer learning.

A video tested with our model can be found here.

Conclusions

Conclusion on Pipeline III and Use Case

A simple but robust mini prototype

The pipeline 3 approaches the most the scenario that would be needed for our elephant rangers talked about in the introduction.
We believe that training a model on one environment with detecting one species for the model would be the way to go for a simple and very efficient object detector for monitoring the endangered animals.
We believe that our small prototype could be scaled for being used in a National Park if data is properly collected, correctly annotated, and potentially pre processed.

Limitations and Future Improvements

Our current approach, and especially the 3rd pipeline is limited by the following factor:

- The need for more data images for a more robust model;
- The need of having a larger environment to be able to generalize better;

- The inclusion of few classes and only one in the last pipeline.

Further improvements that we would suggest are:

- Do more feature engineering and preprocessing depending on the specific case and environment;
- Test other models and weights for transfer learning.

Linking the model to a livestream

For our presentation, we will be trying to implement the full prototype: that is, running a live demo with a live stream from our drone that will be analyzed in real time during the presentation.