Brieuc Popper x Hi! PARIS : *[Link to GitHub repo]*

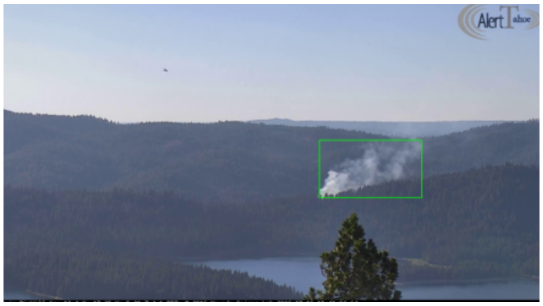

Fig. 1. A bounding box delimits whenever there is smoke caused by a fire

***Abstract*—This document is a report as part of the technical test [Brieuc Popper interviewing for the role of ML Research Engineer at Hi! PARIS]**

I. Introduction

The goal was to develop a model that can predict bounding boxes whenever there is smoke caused by a fire in pictures of forests. This is illustrated in figure 1. For this a dataset of 6703+1699 (train+val) images is provided, among which there are 6053+1529 (train+val.) images with at least one bounding box indicating smoke. There are up to 5 bounding boxes of smoke per image, with some image having none.

II. Chronological timeline of my work

It is important to keep in mind that this project was done under a time constraint and with compute constraints. Section IV details what I would have done with more time.

A. Getting familiar with the data

This is done with an exploratory notebook. The notebook enables the user to draw the bounding boxes over images in the training/validation sets, to see what images don't have any label (these are images for which the ground truth is that there is no smoke), and to do some simple statistics like counting how many bounding boxes there are per image. We can also manually view some outliers.

B. Choice of model

For this task, I choose to use YOLO11 in detection mode as it is very efficient and fast, and because it has publicly available pre-trained weights that can be used for transfer learning. It is overall very well-suited for this exact task. Using the ultralytics python library is very helpful to have streamlined code and run experiements with the training hyper-parameters.

C. YOLO Training

I referred mostly to the documentation here : YOLO training documentation. Because YOLO is so well-suited for the task, it is probably possible to achieve decent performance by training YOLO well, which means we have to find a good set of hyperparameters for training. I fixed some hyperparameters to specific values from the start : these include the model (I used yolo11-small), the batch size (64), the optimizer (AdamW, which is versatile and powerful). This was done to limit the time it would take to experiment, and these reasonable values are probably not holding back the performance of the YOLO's final performance (except of course the model size).

What I then did was a manual iterative refinement of the hyperparameters. The first one I wanted to get right was the learning rate scheduling as a whole, this includes the initial and final learning rate, the warmup, the number of epochs, the *close_mosaic* argument, the optimizer's momentum... This process was done by first using only 15% of the dataset to save some compute time. I iteratively found values that worked well, by monitoring the loss curves (looking for plateaus, unstability) and monitoring overfitting.

I then tuned the parameters regarding data augmentation and regularization to limit overfitting : especially the weight decay and various augmentation coefficients. Once I had a good idea of how these hyper-parameters worked together and what range they should approximately be at, I finally used 50% of the dataset (this means runs take a lot longer) and obtained my best model . To compare models, I compared based on the mAP50, mAP50-95, and the Precision-Recall curve on validation data.

III. My code

On this [link] you can access my code, the README.md there explains the organization of the code. The table below shows the main metrics that my model achieves on validation data.

Metric	mAP@0.5	mAP50-95	P / R (as reported in YOLO validation)
Value	0.531	0.287	0.643 / 0.468

IV. If I had a month to work on this

With more time and more computation available, it is very likely that it is possible to achieve better performance. I would use a bigger YOLO11 model (or another more powerful backbone), and train with the whole training set. If possible it is always better to collect more high quality data to add to the training set (and make sure existing labeled data is high-quality). Also with more time and compute we can afford a higher resolution which will likely yield some small gain in performance.

I would also run a computationally intensive random hyper parameter search (or a Bayesian search), which is very costly but can lead to better final results. It could also be interesting to manually look at examples where the trained YOLO fails, and take steps to mitigate this with perhaps custom pre or post processing of the images and the predictions.

Finally some *active learning* and some *ensembling* can always help a bit in the end to boost the performance a bit.