Remi HARDY – DSTI S18 – Deep Learning on GPU

I started to work initially on one of the listed subjects (image captioning) but faced too many work flow issues (libraries dependencies, incompatibility of several python versions on my laptop, training time). Also, I did not want to run existing code only but experiment my own workflow.
In parallel, I investigated a more basic topic (object detection) that I was expecting to run end to end more easily.
Eventually, I could achieve this goal at least, which is the reason why I am presenting it here. Moreover, the results led me to get some interest on other side but related topics: following this first part, while reading articles and papers, I started to get some interest in OCR, and more specifically "text recognition in the wild".

Demo Video: https://vimeo.com/309118940

In this document, I present:

1. A quick review of state-of-the-art object detection models.
2. Some experiments using 2 pre-trained CNNs
3. An example of training my own detector, to detect the main text headers of newspaper or magazines
4. A performance analysis of YOLO on a tablet using different frameworks and methods
5. A review of state-of-the-art "text in the wild" recognition
6. An experiment using Tesseract OCR

1. Quick review of object detection models

While browsing the existing models for object detection, I ran into the following most popular proposals:

R-CNN: is based on the selective search method (alternative to exhaustive search) to capture object localization. Small initial regions are merged according to a variety of color spaces and similarity metrics. Eventually, the proposal (output) comprises a reduced number of regions that could contain an object.

Fast R-CNN: was developed to reduce the analysis time from standard R-CNN. Selective search is applied only on regions of interest that are identified by a feature map produced by a std CNN.

Faster R-CNN: uses another method called RPN (Region Proposal Method) to avoid using selective search that is too costly.

R-FCNN: is another variation/combination of the methods above

YOLO (You Look Only Once): is very popular, it predicts boxes and classes with a single network (as opposed to 2) and in a single evaluation. It implements 24 convolutional layers and 2 fully connected layers. It uses a grid where a maximum number of boxes is predicted with a certain confidence score (depending on object detection probability and the matching of the bounding box against the labelled one), then only the most relevant ones are kept.

YOLO Tiny: is a lighter thus faster version of YOLO. It implements 9 convolutional layers and a fewer number of filters, 2 fully connected layers

YOLO is now at its 3rd version

SSD: there are various flavors of Single Shot Detector. The method is quite similar to YOLO

The models are either accurate or fast for inference. They all have complex and heavy architectures. Reduction in size while keeping the same performance is an active field of research, to embed deep learning models into embedded platforms (such as mobile devices)

2. Hands-on with TF and Open CV using pre-trained models

My first goal was to run a pretrained model on a mp4 video flux in 2 configurations

a. Using faster_rcnn_inception_V2 trained on coco 2018
b. Using ssd_mobilenet_V1 trained on coco 2018

I selected 2 models with a significant difference in term of speed.
Faster_RCNN_inception_V2 is supposed to have a good performance on detection. SSD_mobilenet_V1, while having a lower performance on detection, is supposed to be faster.

These 2 models are pre-trained on the COCO dataset developed by Microsoft and used for multiple challenges (80 classes, 120 000 images for training and validation)

The theoretical performance of the 2 models are given below.

Model	**Speed**	**mAP (mean avg precision)**
Ssd mobilenet V1	30ms (33 fps)	21
Faster RCNN inception V2	58ms (17 fps)	28

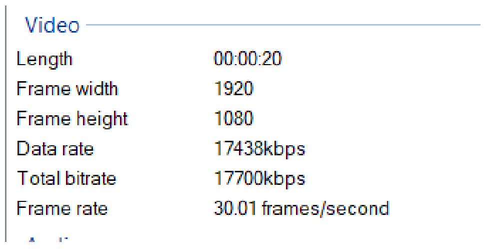
These numbers are given based on 600x600 images, using an NVIDIA TITAN X graphic card.
They can of course be only relative compared to mine, as it depends greatly on the hardware and the images themselves.
From these numbers, we may expect the mobilenet to be twicefaster and slightly less accurate.

I recorded a MP4 video from the street, to feed a Python script applying the detection models on this flux.
On my video flux, as expected, Mobilenet was significantly faster **(~8fps vs ~2fps, so a ratio of 4**) and does not show visually a major detection performance degradation.

Model	**Speed**
Ssd mobilenet V1	8 fps
Faster RCNN inception V2	2 fps

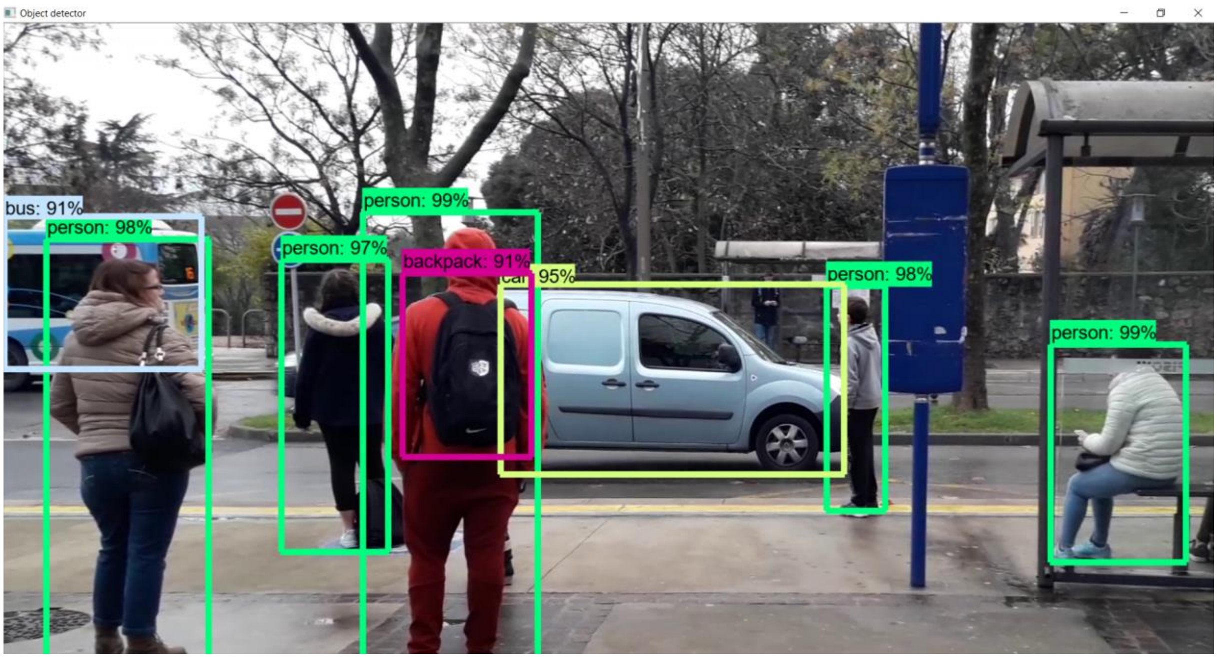
Environment: Anaconda, Windows 10
Framework: Tensorflow
Hardware configuration: CPU i5 7th gen, GPU GTX1050 2Gb

The limited real time performance is mainly due to the size of the video flux (1920x1080x30fps).
Video was taken with my smartphone in the street.


Video

Length	00:00:20
Frame width	1920
Frame height	1080
Data rate	17438kbps
Total bitrate	17700kbps
Frame rate	30.01 frames/second

Below a snapshot of the video flux:



However, compared to YOLO, these 2 models detect less objects.

3. **<u>Train my own detector</u>**

My second goal was to work out the complete pipeline, from dataset creation and annotation, to training the detector and infer on new test images (static images/webcam/mp4 flux).

Although a detector like EAST in Open CV can do the same easily, I wanted to test my own solution and work out the complete pipeline.

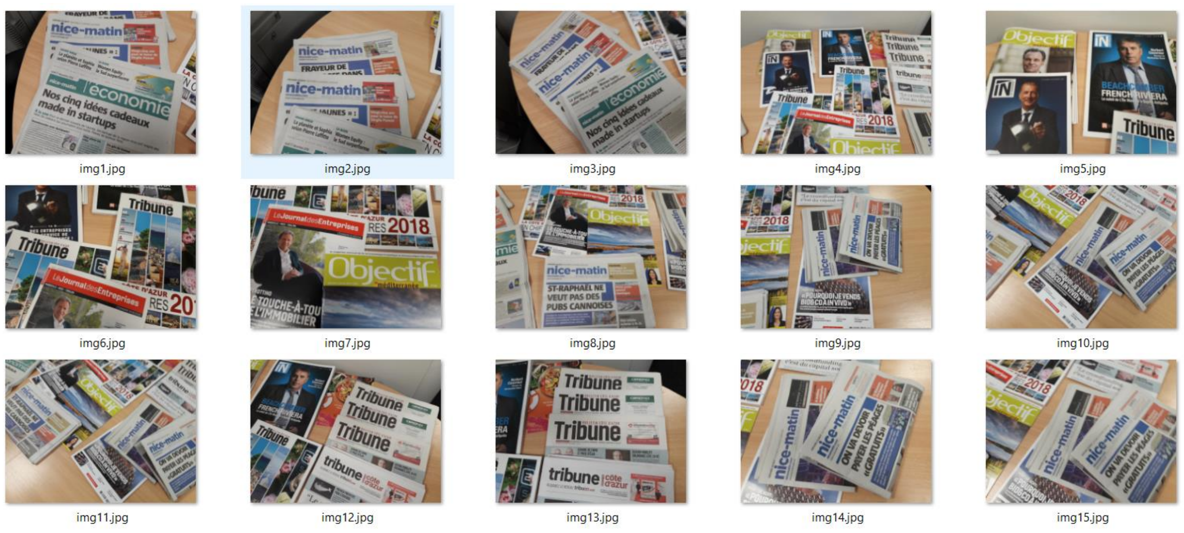
Of course, not everything went as well as described below: before the whole flow can run smoothly, I spent several days fixing various scripts to make it work, due to obsolete functions, missing libraries, deprecated if not invalid options etc …

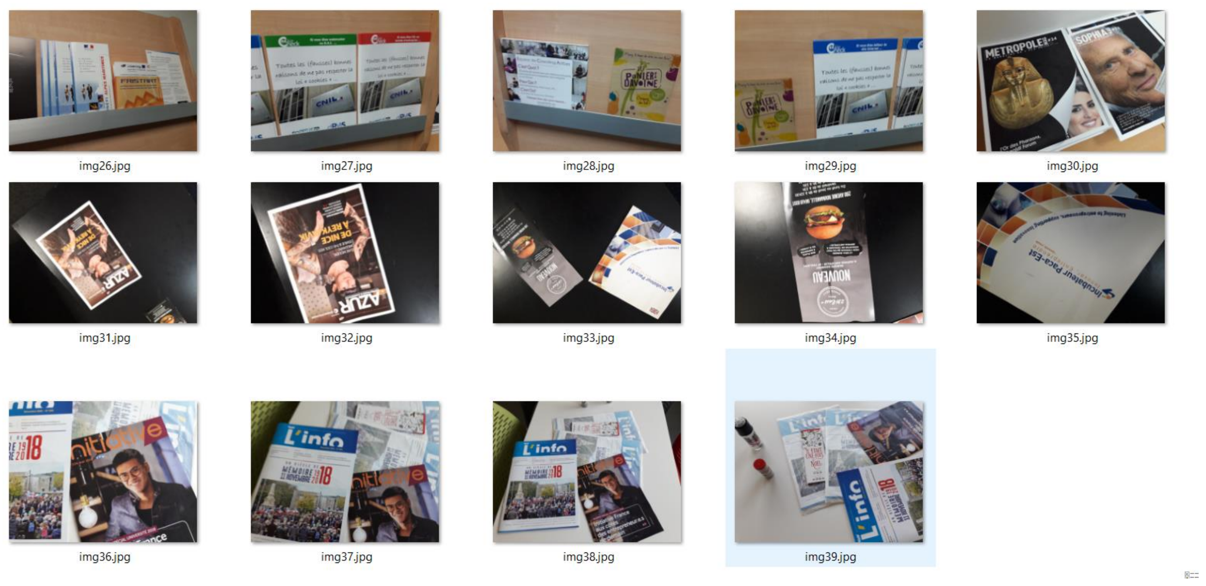
a. <u>Decide on an application</u>
As I had a lot of different newspaper and magazines available at the work place, I decided to build a model to detect the main and big text headers.

b. <u>Choose a pre-trained model</u>
I choose the faster_rcnn_inception_V2 to get good detection results out of the box.
I was planning to run on images that are less demanding as a video file, so performance was not a concern (eventually the speed on a webcam flux is really fine).

c. <u>Build my own image dataset</u>
I took 53 photos of various magazines and newspaper with my mobile phone, in different environments and different orientations.
With a short script, all the images were renamed and resized to 1/8th of original size to be more manageable by the training flow.

Below a snapshot of the images directory:

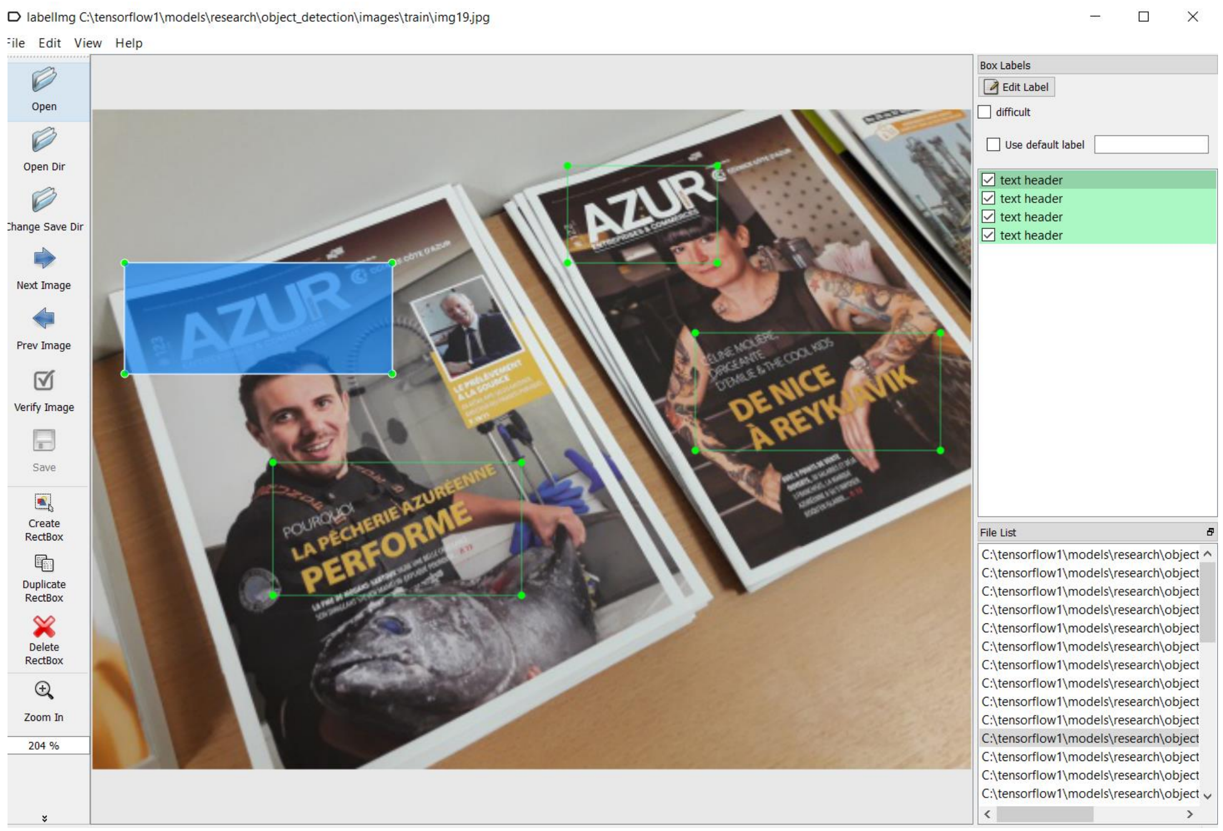




d. <u>Build the annotation files</u>

I used a small application called **<u>labelimg</u>** and annotated ONLY the biggest text headers, not the smaller size text parts.

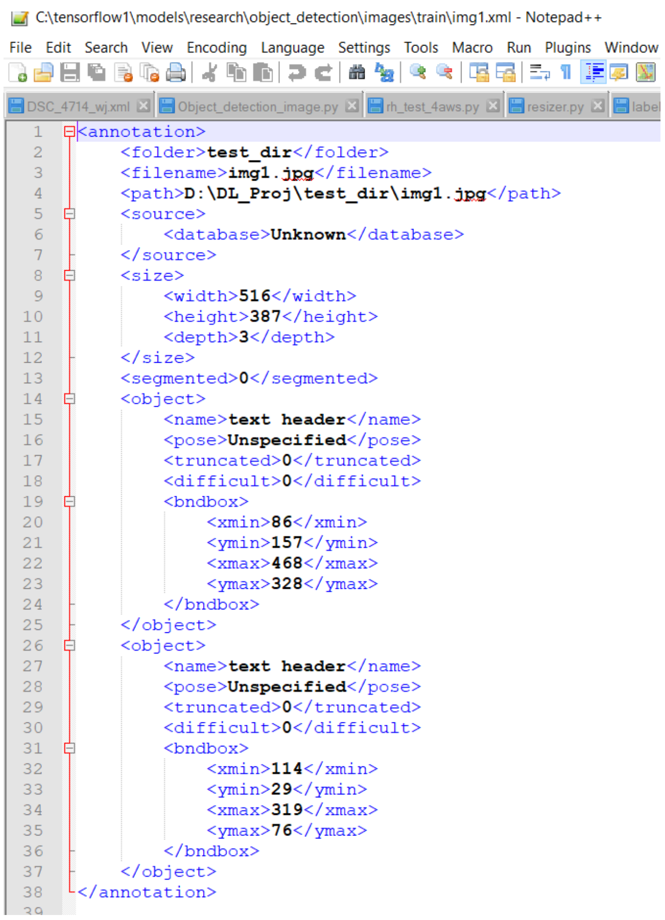
I had 2 to 8 annotations per image leading to 225 annotations in total.


labelImg C:\tensorflow1\models\research\object_detection\images\train\img19.jpg

☑ text header
☑ text header
☑ text header
☑ text header

```
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
C:\tensorflow1\models\research\object
```

It generates the xml files comprising the coordinates of the bounding boxes


C:\tensorflow1\models\research\object_detection\images\train\img1.xml - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window
DSC_4714_wj.xml
Object_detection_image.py
rh_test_4aws.py
resizer.py

```
<annotation>
    <folder>test_dir</folder>
    <filename>img1.jpg</filename>
    <path>D:\DL_Proj\test_dir\img1.jpg</path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>516</width>
        <height>387</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>text header</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>86</xmin>
            <ymin>157</ymin>
            <xmax>468</xmax>
            <ymax>328</ymax>
        </bndbox>
    </object>
    <object>
        <name>text header</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>114</xmin>
            <ymin>29</ymin>
            <xmax>319</xmax>
            <ymax>76</ymax>
        </bndbox>
    </object>
</annotation>
```

Of course, the tool allows to define several classes and annotate multiple objects of multiple classes. Here I had only 1 class to keep it simple, so called "text header".

Several steps are then required to generate the TFrecords files used to train the TF model.

e. Partition the dataset

I partitioned as follows:

- 40 images went to the training directory,
- 13 images went to the validation directory

f. Run the training

It took 3 hours until the loss function is constantly below 0.05. I stopped the training and used the last checkpoint generated by TF.

Hardware configuration:

CPU i5 7th gen: 100% utilization during the training

GPU GTX1050 2Gb: 70 to 80% utilization during the training

g. Generate the frozen graph

From the TF checkpoint, a Python script is used to generate the frozen graph, that is used for inference.

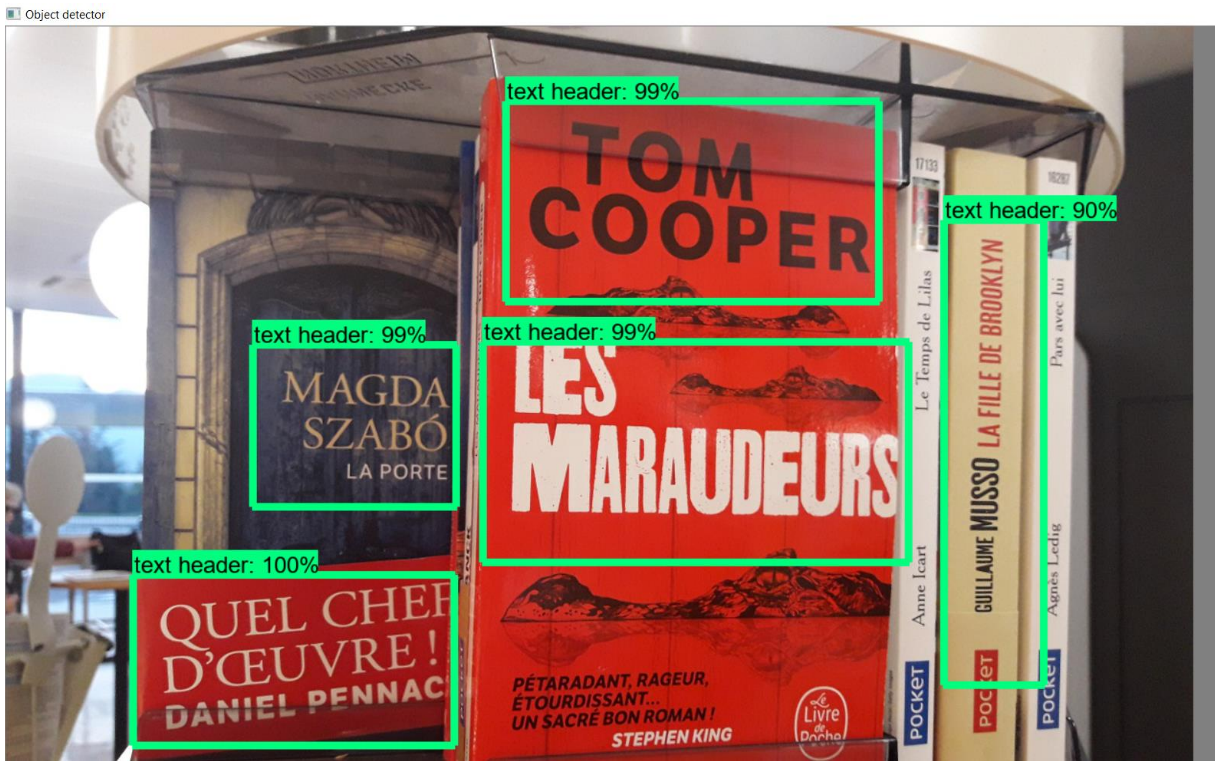
h. Run some inference on test examples

Please check the video for a demo using the webcam flux on VIMEO: https://vimeo.com/309118940

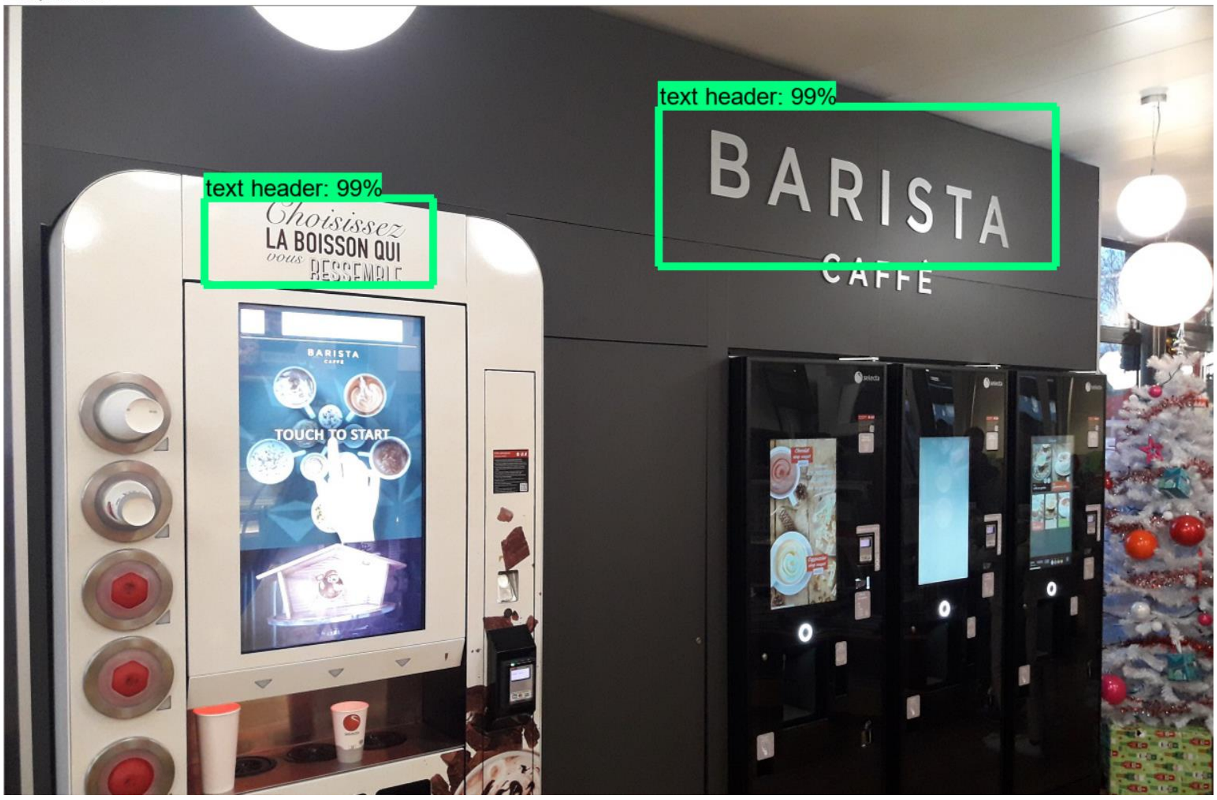
Please note that none of the magazines used in the test are present in the original dataset.

i. Static photos examples

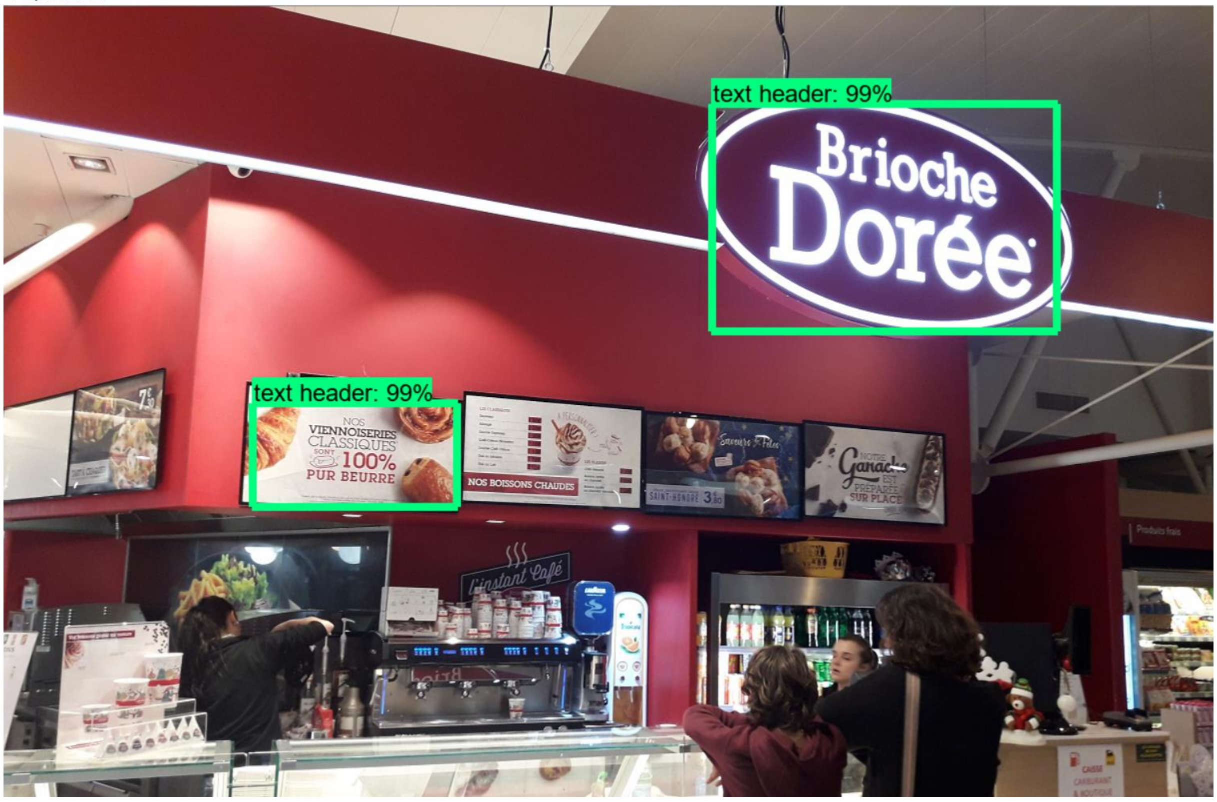
Beside the webcam flux test, I ran the model on several static images from a totally different environment (and it works ok).
Threshold is set at 80% (only bounding boxes with confidence above 80% are depicted)



Object detector



Object detector



4. Performance analysis of YOLO on a tablet using different frameworks and methods

YOLO is popular because it is faster than a faster-R-CNN. While the 2 stages detectors propose object regions first and then investigate the regions for localization and classification, YOLO combines the 2 stages into 1 neural network.

There are several flavors of the YOLO detector, among which we find at least:

- Standard YOLO
- A lighter version YOLO Tiny

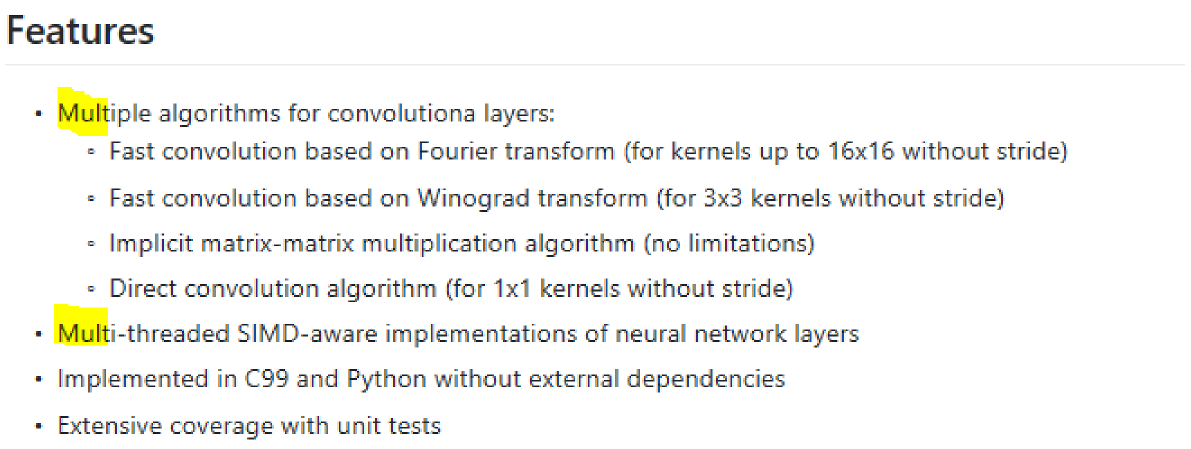
They are both in their 3rd version: V3

As YOLO is popular, there are many papers and blogs describing its concept, implementation and usage. I will skip this here, as I would end-up only paraphrasing these articles.

The YOLO model was developed for the darknet framework. Darknet is written in C and does not have any other programming interface, so if you need to use another programming language or framework due to the platform specifications or your own preferences, you need to adapt it yourself.
As far as my researches went, beside Darknet, I could find YOLO model implementation for OpenCV and Tensorflow beside Darknet.

The idea was to evaluate the speed performance of the YOLO Tiny on a NVIDIA Tablet TEGRA. GPU is a nice add-on to accelerate deep learning computations, but not all embedded platforms provide such an option. TEGRA K1 SoC is the only mobile GPU to support NVIDIA CUDA. It implements 192 CUDA cores. The CPU is a quad core ARM A15 running at 2.3GHz. it supports a maximum of 8Gb of memory.

NNPACK is an accelerator package for neural networks computations, it aims to provide high performance implementations for multicore CPUs. It is available for several frameworks such as Pytorch, Caffee2, Mxnet and Darknet (**that has a dedicated fork with NNPACK support**). From hardware point of view, it supports configurations such as ARM + NEON, ARM64 and x86 architectures.



Features

- Multiple algorithms for convolutiona layers:
  - Fast convolution based on Fourier transform (for kernels up to 16x16 without stride)
  - Fast convolution based on Winograd transform (for 3x3 kernels without stride)
  - Implicit matrix-matrix multiplication algorithm (no limitations)
  - Direct convolution algorithm (for 1x1 kernels without stride)
- Multi-threaded SIMD-aware implementations of neural network layers
- Implemented in C99 and Python without external dependencies
- Extensive coverage with unit tests

Environment: UNIX
Framework: Darknet, OpenCV

I ran the same video testbench on the following configurations:

Detector	Configuration	# frames	Time	FPS
YolotinyV3	Darknet + CPU	126	10min 35 sec	0.2
YolotinyV3	Darknet + CPU + NNPACK no threading	126	2min 45 sec	0.75
YolotinyV3	OpenCV + CPU	126	1min 57sec	1.1
YolotinyV3	Darknet + NNPACK + threading	126	1min 29sec	1.4
YolotinyV3	Darknet + GPU	126	24sec	5.25

Attention: these numbers cannot be compared with the performance from the first study in section 1: the frame size (frame size is smaller by a factor of 7) and the hardware (laptop I5 7th gen vs tablet quad core ARM A15, not the same league at all) are different. They can only be compared relatively to each other in this table.

Here we can see a major performance increase thanks to the Nvidia GPU embedded in the tablet. Unfortunately, not all the Android devices do implement such a GPU.

Running only out of a CPU, the NNPACK library brings a significant performance increase.
We could argue that open CV brings ALSO a relatively good performance. HOWEVER, the Darknet solution detects many more objects than OpenCV (it is visually obvious).
Attention: this sounds weird to me, but I did not have time to investigate this point further. This difference may come from different settings or different implementations while transcoding the YOLO model from Darknet to OpenCV.

OpenCV was initially much faster, but with a lower detection performance than Darknet. OpenCV already uses internally a similar library to NNPACK, reason for its performance. Having NNPACK on top of OpenCV seems useless … and anyhow a difficult task as many functions would need to be updated. Darknet code is much more compact and specialized.

As a conclusion, NNPACK with Darknet allows to have a similar performance compared to OpenCV, but with a better detection performance.

5. A review of state-of-the-art of “text in the wild” recognition

Training my own detector to get main and big text headers detection, I started to think about recognizing the text captured in the bounding boxes, and got some interest in the modern OCR techniques, especially the “text detection in the wild”.

The problem of detecting and recognizing text in images or videos has become a major research topic in the last years. There are regularly new and better solutions to this problem.
Recognizing text from scanned documents is considered as solved: some OCR SW allow to get 99% scores for texts in English.
However, recognizing text in images is another league of a problem: low resolution or degraded images, multi-orientation, complicated or noisy backgrounds. Although the best systems allow scores in the range of 65-75% for Coco-text, a margin of progress remains.

There are now a large variety of databases to test the methods: ICDAR, MSRA, Coco-Text, SVT

a) Detection and extraction:

While the first solutions were based on traditional image processing techniques (Stroke-Width Transform, Maximally Stable Extremal Regions), rapidly the new proposed solutions were based on neural network. While only horizontal text could be recognized initially, the most recent solutions offer multi-orientation text recognition.

There are 3 main solution categories:

- Characters detection: this detection requires a massive image dataset to train the network. A post-processing phase is needed to transform characters into words
- Words detection: although this is simpler than character detection, it does not work for Chinese or Japanese (for ex)
- Lines detection: this method is sometimes not appropriate for short text lines or isolated characters

We can list the following methods:

- CRPN
- PixelLink
- TextBoxes ++
- EAST
- SSTD
- SegLink
- CTPN
- FOTS

b. Recognition

Recognition methods are also based on neural networks. The main difference between the different methods is the need to have a dictionary or not. A pre-defined dictionary allows a faster recognition but may require a pre-processing phase to figure out which dictionary to use.
Standard OCR-only systems are usually lacking performance on this problem.

We can list the following methods:

- CRNN
- DictNet
- PhotoOCR from Google

c. End to end systems

In the last years, systems associating detection and recognition in a single pipeline have been created. Most of them rely on Neural Nets and Deep Learning.

We can list the following methods:

- FOTS
- textboxes++ and CRNN
- Deep TextSpotter
- TextProposals and DictNet
- SEE

Latest I found (Nov 2018): https://arxiv.org/ftp/arxiv/papers/1811/1811.07432.pdf

- **Sources**

Databases :

ICDAR 2015 (1500 images) [Lien]
COCO-Text (63 686 images) [Lien]
MSRA-TD500 (500 image) [Lien]
SVT ou Street View Text (350 images from Google Street View) [Lien]
IIT5k (5000 words)[Lien]
SynthText [Lien]

Solutions :

TextProposals : Article - Code
TextBoxes ++ : Article - Code
EAST : Article - Code
Deep TextSpotter : Article - Code
CurvedTextDetector : Article - Code
SSTD : Article - Code
CRPN : Article - Code
CTPN : Article - Code

SegLink : Article - Code
PixelLink : Article - Code
FOTS : Article
CRNN : Article - Code
Jaderberg 2014 : Article - Code
PhotoOCR : Article
Tesseract : Code - OCR

6. An experiment using Tesseract OCR

Eventually, I felt like testing an OCR method on my text detector described in the section 3.
Running out of time, the goal was to test (quick and dirty) a solution as straightforward as possible. The implementation of one of the latest end-to-end method was out of scope for this assignment.

I installed Tesseract OCR from Google and tested it on the command line.
Then, I installed its Python Pytesseract API and included it in the object detection Python script used to run “text header” detection on static images.

Method:

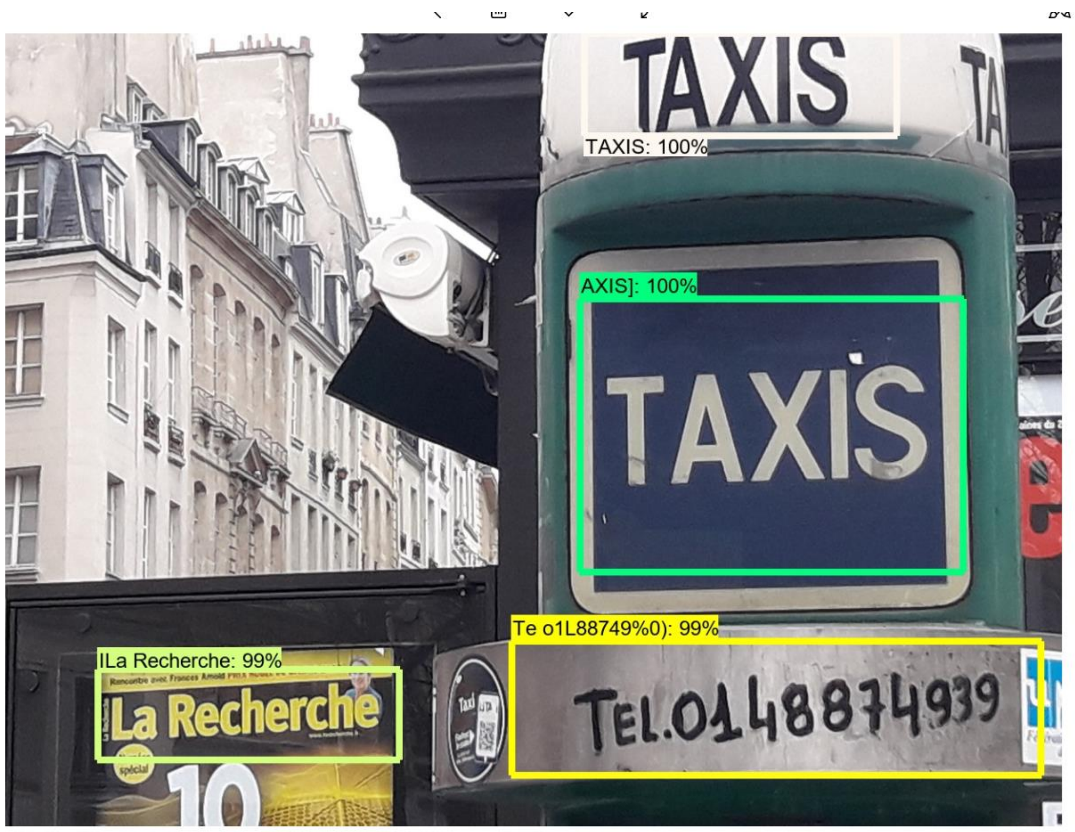
1. Run “text header” detection from section 3
2. Crop all the bounding boxes (where some text is detected)
3. Run Tesseract API on bounding box crops
4. Update bounding box tags with Tesseract result
5. Update image with bounding boxes and new tags
6. Print the final image into a JPG file

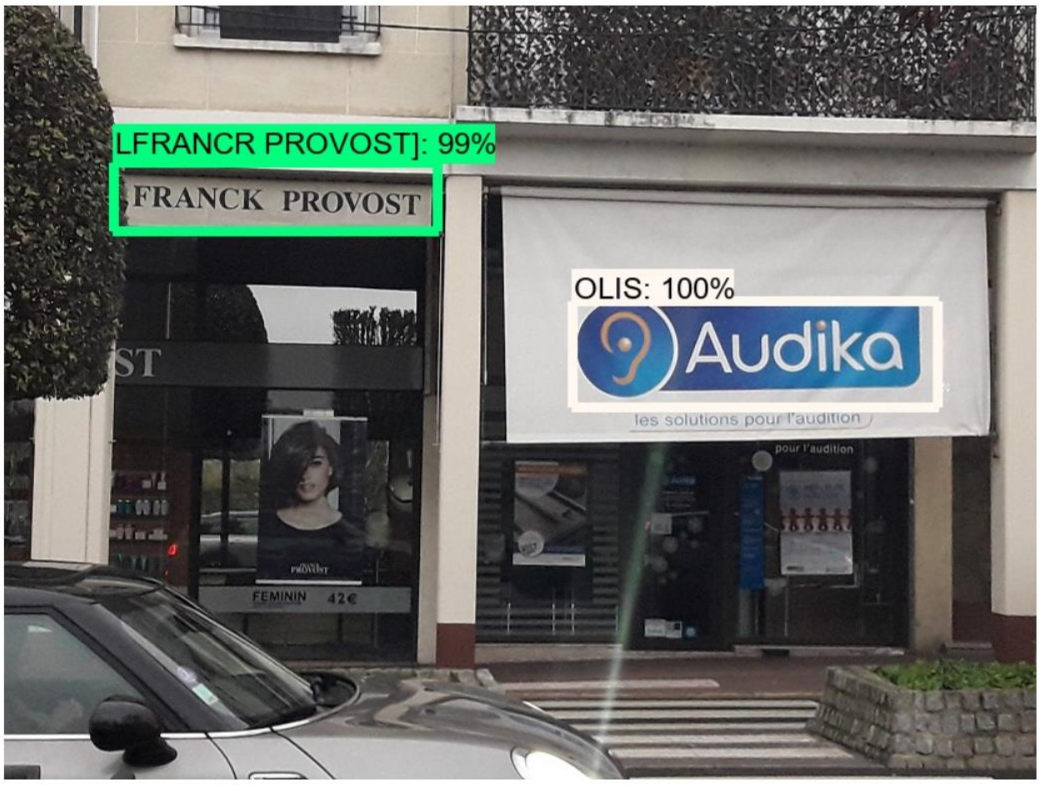
Tesseract is often used as a benchmark for “text in the wild” detection and recognition.
Personally, I found the following limitations:

- It is sensitive to character size and resolution (the bigger the characters and the bigger the image, the better)
- It is sensitive to the PSM option (Page Segmentation Mode) that cannot be configured in advance in the case of “text in the wild”
- It is not very good for curvy or exotic text polices

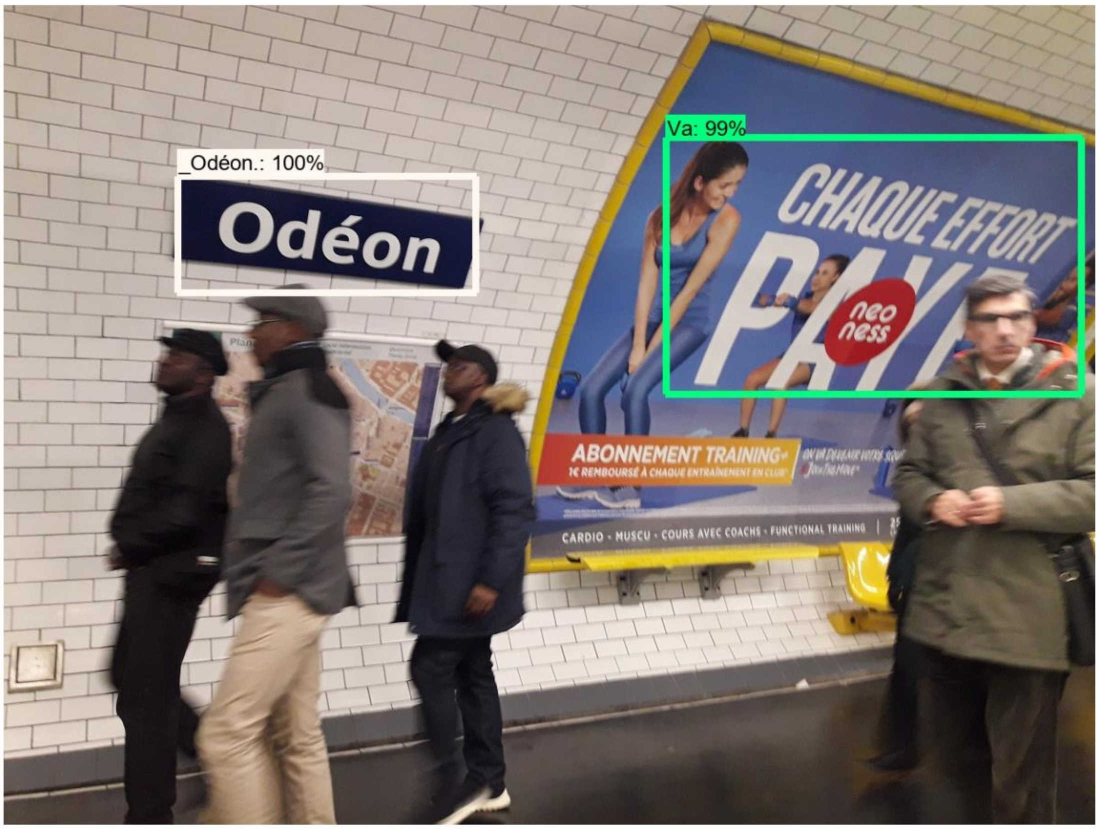
For an industrial project, I would rather try to implement one of the end-to-end method.

Below some test samples:

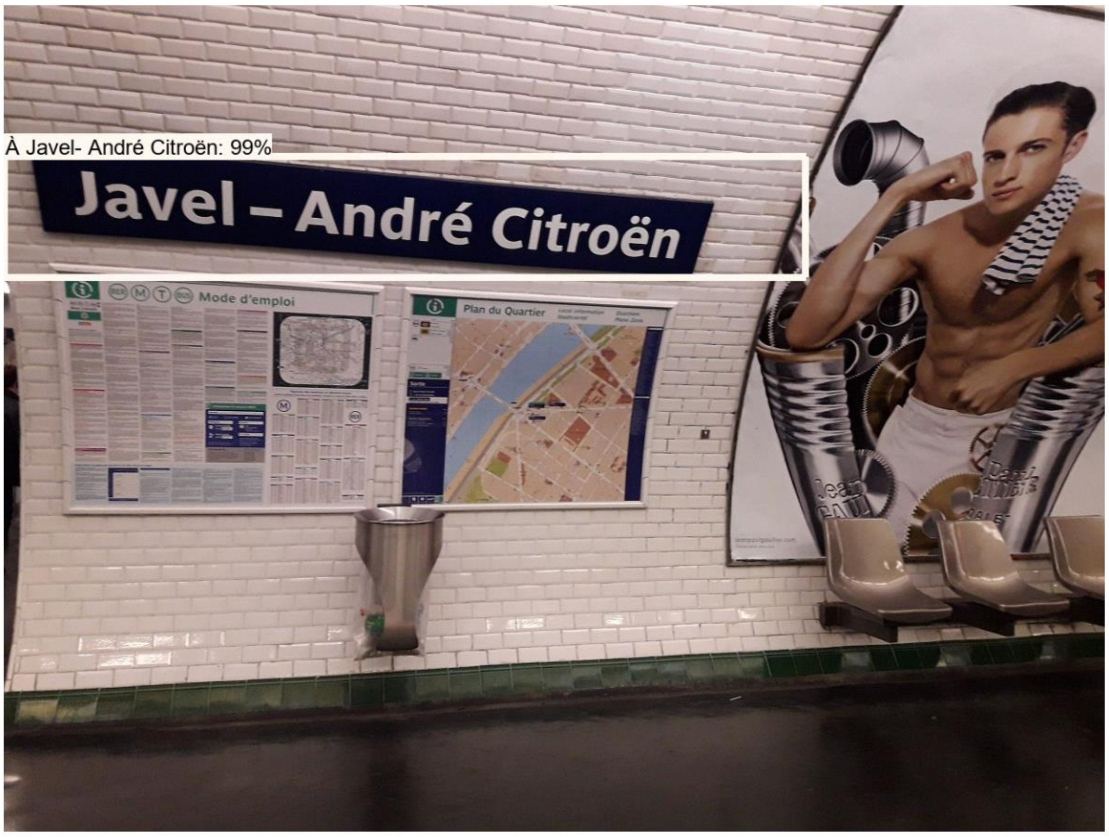


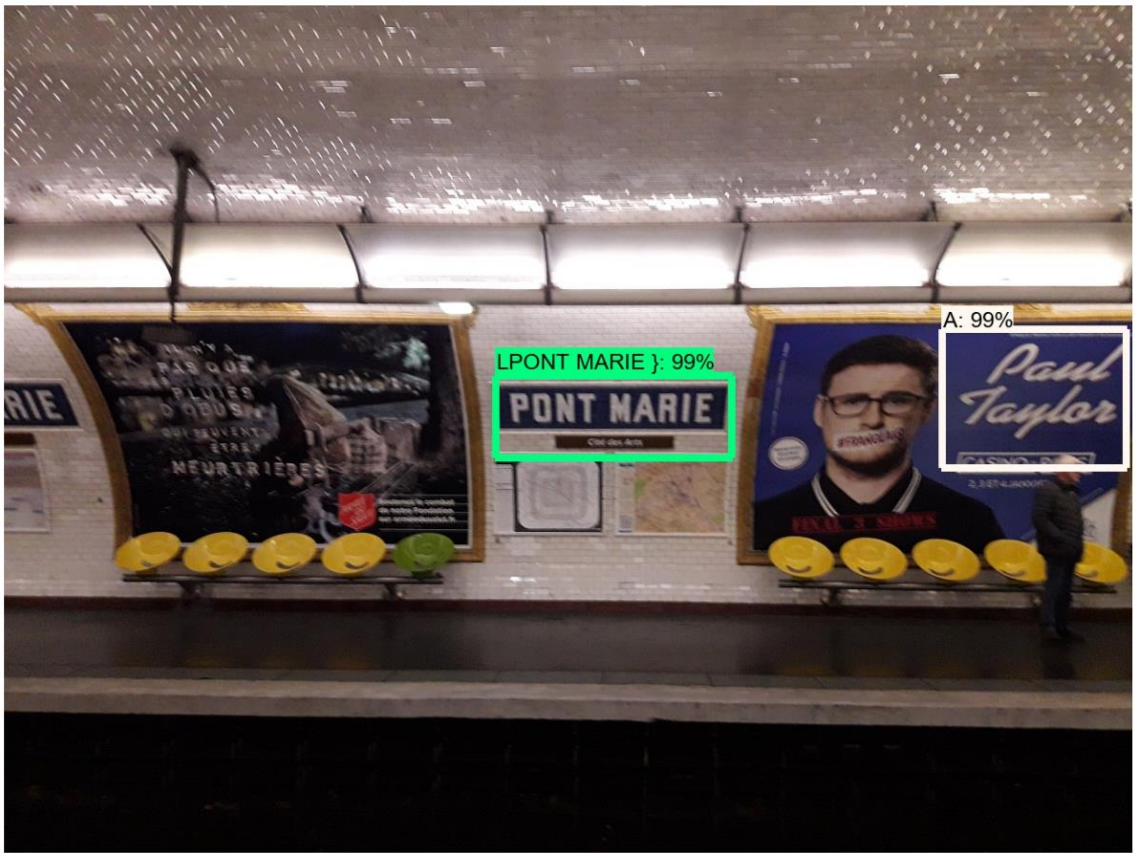


Incorrect result on AUDIKA

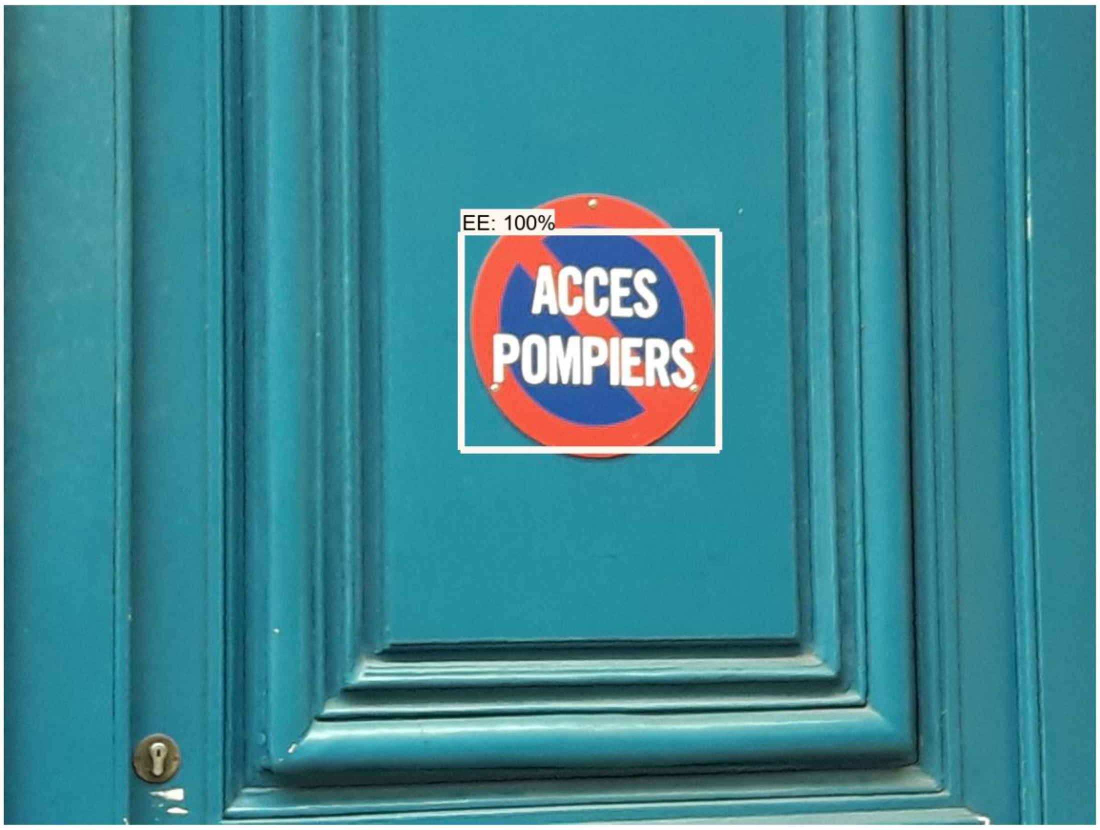

_Odéon.: 100%
Odéon
Va: 99%
CHAQUE EFFORT
PAYE
neo ness
ABONNEMENT TRAINING
1€ REMBOURSÉ À CHAQUE ENTRAÎNEMENT EN CLUB
CARDIO - MUSCU - COURS AVEC COACHS - FUNCTIONAL TRAINING

Incorrect result on the large adv. panel, that could be expected. OK result for “Odeon”.





Does not like curvy writing!



2 lines, missed ! requires a different –psm option