License Plate Detector

Documentation

By Dimitar Dyulgerov

Date: 14.03.2024

Abstract

This document describes the development process and decision making points throughout the development cycle of the license plate detector. It also describes and provides clarification for the source code, providing an insight which may be useful for extending or implementing the code in other projects

Table of contents

Introduction

The license plate detector project's aim is to take an input image, containing a vehicle with a visible and unobstructed license plate, and output its contents as a string. The project has been made with Dutch car license plates in mind, i.e. the license plate format is expected to be in one line. The project can be used for other license plates as well, however accuracy may vary.

It is worth noting that this is by no means the first realisation of a license plate detector, as there are plenty of working examples that can be found online. All the information online regarding the topic, however, is mostly in the context of Python. One of the main goals and challenges of this assignment is to develop a solution in a Windows/C++ environment. That in itself adds a considerable layer of complexity, especially given the documentation limitations and lack of example code.

Procedure

Project structure

- **OS**: Windows 10
  Windows is the most popular operating system, used worldwide. Linux may be more predominant in the embedded world, however its global use is still quite a bit lower. Windows also may require some additional setup steps, such as linux-only library equivalents, a big chunk of which are less well documented, hence making the research phase more difficult (and thus creating a bigger learning possibility).
- **Code Editor**: Visual Studio Code
- **Language**: C++
- **Compiler**: MSVC19
- **Build tool**: CMake 3.10+
- **Library manager**: vcpkg
- **Computer Vision API**: OpenCV
- **OCR Engine**: Tesseract
- **Version Control**: git/GitHub
- **ML framework**: YOLOv8
- **ML format**: ONNX
- **ML Library**: OpenCV 4.x+
- **ML model**: pretrained model (link to original GitHub)

One of the goals for the project is to run the application in a Windows 10 environment, with C++. There are plenty of **python** projects that share more or less the same sentiment as this one, however not many, if any at all, examples can be found for Windows/C++. That added layer of complexity serves to improve researching skills and hopefully be useful for other people who are working on vision in a similar environment, be it as a hobby or professionally. The main challenge, as discussed in the aptly named section, is parsing and interpreting the outputs of the ML models.

Source code documentation

The license plate detector source code is mostly self-documenting, though it is useful to have a document with references and more detailed explanations of different parts of interest, the main ones being the DNN output parser, the OCR parser, and the driver code.

DNN Output Parser

The DNN output parser is the layer of the program that is used to leverage a pre-trained DNN model, in this case a YOLOv8 model in ONNX format, and translate its outputs to more meaningful information that can be used later in the program. For instance, one of the translated outputs is the coordinates of the bounding box containing the detected license plate region(s) that exceed the pre-set confidence threshold (50%). That is vital information for the rest of the application, as it isolates a region of interest (ROI) that can be better preprocessed for the OCR step.

The bounding box coordinates are the main output of interest of the DNN model, however further information is also available that can serve for debugging or verbose outputs.
Detected class of the object: in this case the model will always return “0”, as there is only one class that the model is trained to recognize (a license plate on a vehicle);
Detection confidence: the model’s confidence of the current detection - a number between 0.0 and 1.0 (higher is better).

YOLOv8Detector (DNN output parser class) high-level description

YOLOv8Detector is a class that encapsulates the DNN parsing required for the project.

Initialize

This method is responsible for initialization of the class, as opposed to having it performed by a constructor. The reasoning being is that this approach gives a better control of when the DNN is initialised, even if the object for the class is created beforehand (i.e. to be used as a global variable/passed as a dependency). DNN initialization can be relatively time-consuming, so having it be part of a startup sequence with control over when it happens is preferable.
The arguments for the method are the file path to the model and the number of classes in the model. There is a possibility for an extension, where a map of the class ids to their names is defined.

SetConfidenceThreshold

As the name suggests, sets the confidence threshold for the model (non-inclusive);

Detect

This is the method housing the main logic of the class. It takes in a source image (an OpenCV Mat object) and runs it through the model. It is worth noting that the source image is converted to 640x640px. The outputs are rescaled accordingly, however, so there is no need to worry about a mismatch with the original image.
The method returns a vector of type Detection, a struct containing basic detection information: class id, confidence (0.0 to 1.0), and the bounding box (OpenCV rectangle).

YOLOv8/ONNX output deep dive

The output of the model is a matrix of size 1 x 5 x 8400, where:
(1st dimension) 1: batch size, always 1
(2nd dimension) 5: 5 elements, 0, 1, 2, 3 are x, y (both x and y describing the **centre** of the box), width, height, respectively. 4 is the confidence for each class. This number can change, depending on the number of classes. In the license plate detection model, there is only 1 class, the license plate, hence why there are 5 elements
(3d dimension) 8400: the maximum number of possible detections/bounding boxes

Further explanation, from one of the developers from a GitHub Issue:
Yes, each row in the matrix represents a different detection, but the total number of detections depends on how many bounding boxes the model has detected and kept after applying confidence thresholds and non-maximum suppression.

No, there are not "1400 pieces" per row. In the output matrix, each row corresponds to one detection, which typically includes the `center_x`, `center_y`, the width `w`, height `h` of the bounding box, the objectness score indicating the confidence that the box contains an object, and a set of class probabilities reflecting the model's confidence for each class. The exact size and structure of the prediction matrix depend on post-processing steps used to filter and sort these detections.

OCR Parser

The OCR parser layer of the program is responsible for using Optical Character Recognition (OCR) to extract text from an image. In this context, it is used to extract the text from the (preprocessed) region of interest detected by the DNN model in the previous layer. It outputs the detected contents of the image as a string. Accuracy on some license plate formats, i.e. ones that are considerably different to the Dutch ones, can vary. The language for the OCR parser is English, therefore accuracy on Asian license plates can be especially poor.

Input image characteristics for best performance of Tesseract

- Black text on white background
- Tresholded image (only black and white, no greys)
- As much noise from the image has to be removed, i.e. elements that are not text
- 300dpi (or more) resolution
- Recommended text size is 10-30px for a lowercase “x”, alternatively a 25-30px high lowercase “d”
- Text should not be too “squished” - letters should have a balanced aspect ratio

TesseractOCR (ocr wrapper class)

This class is responsible for housing the logic related to the use of OCR. It follows much the same design pattern as the DNN parser class.

Initialize

Initializes the OCR engine, with some optional parameters, such as the path to the language model, the language id (a string with the language abbreviation, as defined by the Tesseract API), and the OCR engine mode. All of these parameters are set with default values, so changing them is not recommended unless custom behaviour is required, such as changing the OCR language.

SetPageSegMode

Sets the page segmentation mode for the OCR, meaning it tells the OCR how the text is structured - on a single line, on multiple lines, in multiple columns, etc.
The default value is PSM_AUTO, i.e. letting Tesseract decide what to use.

GetText

This is the method used to extract text from an image. It expects an openCV Mat object, containing the image data. It returns the detected text, in an std::string, taking care of memory safety issues that could arise from the Tesseract API’s output - the output that the API itself provides is stored in an allocated piece of memory that needs to be freed afterwards. Having the output be stored in an std::string and freeing the resources after avoids this potential memory leak.

Driver code

The driver code is the layer of the program that is responsible for running the whole project. It makes use of the previous 2 layers, doing some small steps in between as well, such as basic preprocessing of the ROI detected by the DNN model, as well as some output cleanup, such as getting rid of invalid/unexpected characters due to poor input image quality for the OCR.

Conclusion

In conclusion, I believe that the implementation of a license plate detector in a windows environment and using C++ instead of Python has been an overall success. The main challenge was implementing the necessary steps in the context of C++. Python is by far the more popular programming language when it comes to implementing ML algorithms, with plenty of already made libraries that do most of the heavy lifting under the hood, something that is lacking in C++. However, that creates the need to go a little deeper and get more acquainted with how the whole project works. Of course, libraries were used, as listed in the Project Structure section, however a lot of the parsing and preprocessing needed to be done manually. I believe that this project has been a great challenge and opportunity to further improve not only my C++ programming skills, but research, sifting through irrelevant and/or outdated information, code translation, debugging and problem solving altogether. The project has also been a good introduction to the field of computer vision, and its implementation for a concrete use-case, such as a license plate detector.

Appendix

GitHub repository containing setup guides and project notes

References

OpenCV image processing documentation
OpenCV Fontys Sharepoint
OpenCV Eduresources

Tesseract output quality improvement
OCR with Tesseract article
Tesseract C++ API example
Tesseract API documentation

YOLOv8/ONNX parser reference
YOLOv8/ONNX output discussion (1)
YOLOv8/ONNX output discussion (2)
YOLOv8 raw output discussion
YOLOv8 license plate recognition model
YOLOv8 export to ONNX
YOLOv8/ONNX in C++ (ONNX runtime)
YOLOv8/ONNX in C++ (OpenCV)
Number plate recognition in Python
Steps to train a YOLOv8 model (custom dataset)