Doubango AI

State of the art ANPR/ALPR implementation for embedded devices (ARM) and desktops (X86) using deep learning

https://github.com/DoubangoTelecom/ultimateALPR-SDK

Table of Contents

This is a short technical guide to help developers and integrators take the best from our ALPR/ANPR SDK. You don't need to be a developer or expert in deep learning to understand and follow the recommendations defined in this guide.

1 Intro

Have you ever seen a deep learning based ANPR/ALPR (Automatic Number/License Plate Recognition) engine running at 47fps on ARM device (Android, Snapdragon 855, 720p video resolution) ?

With an average frame rate as high as **47 fps on ARM** devices (Snapdragon 855) this is the fastest ANPR/ALPR implementation you'll find on the market. Being fast is important but being accurate is crucial. We use state of the art deep learning techniques to offer unmatched accuracy and precision. As a comparison this is **#33 times faster than OpenALPR on Android** (see benchmark section for more information).
No need for special or dedicated GPUs, everything is running on CPU with SIMD ARM NEON optimizations, fixed-point math operations and multithreading.
This opens the doors for the possibilities of running fully featured ITS (Intelligent Transportation System) solutions on a camera without soliciting a cloud. Being able to run all ITS applications on the device will **significantly lower the cost to acquire, deploy and maintain** such systems. Please check “Device-based versus Cloud-based solution” section for more information about how this would reduce the cost.
We're already working to bring this frame rate at 64fps and add support for CMMDP (Color-Make Model-Direction-Prediction) before march 2020. We're confident that it's possible to have a complete ITS (license plate recognition, CMMDP, bus lane enforcement, red light enforcement, speed detection, congestion detection, double white line crossing detection, incident detection...) system running above 40fps on ARM device.

On high-end NVIDIA GPUs like the **Tesla V100 the frame rate is 315 fps** which means 3.17 millisecond inference time. On low-end CPUs like the **Raspberry Pi 4 the average frame rate is 12fps.**

Don't take our word for it, come check our implementation. **No registration, license key or internet connection is needed**, just clone the code from Github and start coding/testing: https://github.com/DoubangoTelecom/ultimateALPR-SDK. Everything runs on the device, no data is leaving your computer. The code released on Github comes with many ready-to-use samples to help you get started easily. You can also check our online cloud-based implementation (no registration required) at https://www.doubango.org/webapps/alpr/ to check out the accuracy and precision before starting to play with the SDK.

2 Supported countries

Unlike other companies we don't segment our implementation by region but are grouping them by charset (e.g. Latin, Arabic, Chinese...). The reference models provided on Github at https://github.com/DoubangoTelecom/ultimateALPR-SDK are trained on **Latin charset ([A-Z0-9])** using license plates from more than **150 countries**.
The dataset predominantly contains European license plates as this is where our company is based and most of our customers are using this SDK in Europe. The implementation will work with any country using Latin charset like **USA, Canada, Russia, Armenia, Monaco, India, UK, Turkey, Argentina, Mexico, Indonesia, Philippines, New Zealand, Australia, Brazil, South Africa, Mauritania, Senegal...** If you have any accuracy issues with your country please let us know and we'll add more samples in the dataset. If you can provide your own dataset it would be great.

Starting version 2.7.0 we support Korean license plates.

We can pack all the charsets and provide a single model but the accuracy will drop by 17% and this is why we've to keep them separated.

We have a **"write-once-and-train-everywhere"** implementation which means the current code used with Latin charset will work with Japanese, Chinese, Arabic or any other language without single modification. You even don't need to update the SDK (or your code), just drop the newly trained data and start testing.

The license plate detector is agnostic and supports all countries.

The recognizer is agnostic and supports all countries but you have to provide the right trained data (same model as Latin). If you have a dataset with non-Latin charset and want it included in the SDK please contact us and we'll do it for free.

3 Architecture overview

3.1 Supported operating systems

We support any OS with a C++11 compiler. The code has been tested on Android, iOS, Windows, Linux, Raspberry Pi 3 and many custom embedded devices (e.g. Cameras).

The Github repository (https://github.com/DoubangoTelecom/ultimateALPR-SDK) contains binaries for Android and Raspberry Pi as reference code to allow developers to test the implementation. This reference implementation comes with both Java and C++ APIs. The API is common to all operating systems which means you can develop and test your application on Android or Raspberry Pi and when you're ready to move forward we'll provide the binaries for your OS.

3.2 Supported CPUs

We officially support any ARM32 (AArch32), ARM64 (AArch64), X86 and X86_64 architecture. The SDK have been tested on all these CPUs.

MIPS32/64 may work but haven't been tested and would be horribly slow as there is no SIMD acceleration written for these architectures.

Almost all computer vision functions are written using assembler and accelerated with SIMD code (NEON, SSE and AVX). Some computer vision functions have been open sourced and shared in CompV project available at https://github.com/DoubangoTelecom/CompV.

3.3 Supported GPUs

We support any OpenCL 1.2+ compatible GPU for the computer vision parts.

For the deep learning modules:

The desktop/cloud implementation uses TensorRT which requires NVIDIA CUDA.

The mobile (ARM) implementation works anywhere thanks to the multiple backends: OpenCL, OpenGL shaders, Metal and NNAPI.

Please note that for the mobile (ARM) implementation a GPU isn't required at all. Most of the time the code will run faster on CPU than GPU thanks to fixed-point math implementation and quantized inference. GPU implementations will provide more accuracy as it rely on 32-bit floating-point math. We're working to provide 16bit floating-point models for the coming months.

3.4 Supported programming languages

The code was developed using C++11 and assembler but the API (Application Programming Interface) has many bindings thanks to SWIG.

Bindings: **ANSI-C**, **C++**, **C#**, **Java**, **ObjC**, **Swift**, **Perl**, **Ruby** and **Python**.

3.5 Supported raw formats

We supports the following image/video formats: **RGBA32, BGRA32, RGB24**, **NV12**, **NV21**,

Y(Grayscale), YUV420P, **YVU420P**, **YUV422P** and **YUV444P**. NV12 and NV21 are semi-planar formats also known as **YUV420SP**.

3.6 Optimizations

The SDK contains the following optimizations to make it run as fast as possible:

- Hand-written assembler
- SIMD (SSE, AVX, NEON) using intrinsics or assembler
- GPGPU (CUDA, OpenCL, OpenGL, NNAPI and Metal)
- Smart multithreading (minimal context switch, no false-sharing, no boundaries crossing...)
- Smart memory access (data alignment, cache pre-load, cache blocking, non-temporal load/store for minimal cache pollution, smart reference counting...)
- Fixed-point math
- Quantized inference
- ... and many more

Many functions have been open sourced and included in CompV project: https://github.com/DoubangoTelecom/CompV. More functions from deep learning parts will be open sourced in the coming months. You can contact us to get some closed-source code we're planning to open.

3.7 Thread safety

All the functions in the SDK are thread safe which means you can invoke them in concurrent from multiple threads. But, you should not do it for many reasons:

- The SDK is already massively multithreaded d in an efficient way (see the threading model section).
- You'll end up saturating the CPU and making everything run slower. The threading model makes sure the SDK will never use more threads than the number of virtual CPU cores. Calling the engine from different threads will break this rule as we cannot control the threads created outside the SDK.
- Unless you have access to the private API the engine uses a single context which means concurrent calls are locked when they try to write to a shared resource.

4 Device-based versus Cloud-based solution

There are many cloud-based ANPR/ALPR solutions. We also have one and it's hosted at https://www.doubango.org/webapps/alpr/. Based on our experience we always recommend our customers to go with device-based solutions unless they have money and time to waste.

Lets take a scenario where a company have #500 cameras to install to monitor high speed roads **in realtime**.

In order to be able to capture license plates from a car running at any speed we need a solution working at 25fps or more.

If you're going for cloud-based solutions like OpenALPR you'll face the following issues:

1. According to their websites a high-end dedicated server can run up to 17fps which means we're already missing our 25fps target.
2. A cloud-based solution is hosted on remote site which means you'll have to send the data over the network for processing. This will require high speed bandwidth and off course you'll have to pay for it. Sending #500 video streams at 720p resolution 24/7 would cost a lot.
3. To barely meet the target fps you'll need a dedicated-server per camera. Again, we want a realtime solution which means the video frames cannot be stored and processed later.
4. Even if you go with #1 server for #5 cameras you'll end up with hosting #100 servers. Hosting such number of servers will cost a lot as you'll have to pay the maintenance, the electricity, the bandwidth...

If you're going for device-based solution like ours:

1. Every camera will do the detection and recognition at up to 47fps without sending a single byte to a remote server.
2. Once a license plate is recognized you only send the plate number, GPS coordinates and eventually the image representing the plate (e.g. 100x100 cropped region). This represent few KB of data.
3. A single server can handle the recognition results from the #500 cameras in realtime.
4. No server maintenance or large electricity bills to deal with. Minimal bandwidth usage which mean minimal cost

In fact, the scenario described here represent a real project we had to deliver for one of our customers. They are now paying €53.99 per month to host a single server in order to monitor their #500+ cameras. Previously they had to pay several thousand € per months which include the hosting costs and royalties to the ANPR provider.

Our licensing model is royalty-free and lifetime which means you pay once regardless the number of images you want to process.

5 Configuration options

The configuration options are provided when the engine is initialized and **they are case-sensitive.**

Name	Type	values	Description
debug_level	STRING	verbose info warn error fatal	Defines the debug level to output on the console. You should use verbose for diagnostic, info in development stage and warn in production. Default: info
debug_write_input_image_enabled	BOOLEAN	true false	Whether to write the transformed input image to the disk. This could be useful for debugging. Default: false
debug_internal_data_path	STRING	Folder path	Path to the folder where to write the transformed input image. Used only if debug_write_input_image_enabled is true. Default: ""
license_token_file	STRING	File path	Path to the file containing the license token. First you need to generate a Runtime Key using requestRuntimeLicenseKey() function then activate the key to get a token. You should use license_token_file or license_token_data but not both.
license_token_data	STRING	BASE64	Base64 string representing the license token. First you need to generate a Runtime Key using requestRuntimeLicenseKey() function then activate the key to get a token. You should use license_token_file or license_token_data but not both.
num_threads	INTEGER	Any	Defines the maximum number of threads to use. You should not change this value unless you know what you're doing. Set to -1 to let the SDK choose the right value. The right value the SDK will choose will likely be equal to the number of virtual core. For example, on an octa-core device the maximum number of threads will be #8. Default: -1

gpgpu_enabled	BOOLEAN	true false	Whether to enable GPGPU computing. This will enable or disable GPGPU computing on the computer vision and deep learning libraries. On ARM devices this flag will be ignored when fixed-point (integer) math implementation exist for a well-defined function. For example, this function will be disabled for the bilinear scaling as we have a fixed-point SIMD accelerated implementation: https://github.com/DoubangoTelecom/compv/blob/master/base/image/asm/arm/compv_image_scale_bilinear_arm64_neon.S . Same for many deep learning parts as we're using QINT8 quantized inference. Default: `true`
assets_folder	STRING	Folder path	Path to the folder containing the configuration files and deep learning models. Default value is the current folder. The SDK will look for the models in "$(assets_folder)/models" folder. Default: . Available since: 2.1.0
charset	STRING	latin korean	Defines the charset (alphabet) to use for the recognizer. Default: `latin` Available since: 2.6.0
detect_roi	FLOAT[4]	Any	Defines the Region Of Interest (ROI) for the detector. Any pixels outside region of interest will be ignored by the detector. Defining an WxH region of interest instead of resizing the image at WxH is very important as you'll keep the same quality when you define a ROI while you'll lose in quality when using the later. Format: `[left, right, top, bottom]` Default: `[0.f, 0.f, 0.f, 0.f]`
detect_minscore	FLOAT	]0.f, 1.f]	Defines a threshold for the detection score. Any detection with a score below that threshold will be ignored. Range: `]0.f, 1.f]` Default: `0.3f` `0.f` being poor confidence and `1.f` excellent confidence.
detect_gpu_backend	STRING	opengl	Defines the GPU backend to use. This entry is only meaningful when

		opencl nnapi metal none	gpgpu_enabled=true . You should not set this value and must let the SDK choose the right value based on the system information. On desktop implementation, this entry will be ignored if support for CUDA is found.This value is also ignore when detect_quantization_enabled=true as quantized operations are never executed on a GPU.
detect_quantization_enabled	BOOLEAN	true false	Whether to enable quantization on ARM devices. Please note that quantized functions never run on GPU as such devices are not suitable for integer operations. GPUs are designed and optimized for floating point math. Any function with dual implementation (GPU and Quantized) will be run on GPU if this entry is set to false and on CPU if set to true. Quantized inference bring speed but slightly decrease the accuracy. We think it worth it and you should set this flag to true. Anyway, if you're running a trial version, then an assertion will be raised when you try to set this entry to false. Default: true
pyramidal_search_enabled	BOOLEAN	true false	Whether to enable pyramidal search. Pyramidal search is an advanced feature to accurately detect very small or far away license plates. Default: true
pyramidal_search_sensitivity	FLOAT	[0.f, 1.f]	Defines how sensitive the pyramidal search anchor resolution function should be. The higher this value is, the higher the number of pyramid levels will be. More levels means better accuracy but higher CPU usage and inference time. Pyramidal search will be disabled if this value is equal to 0. Default: 0.28.
pyramidal_search_minscore	FLOAT	]0.f, 1.f]	Defines a threshold for the detection score associated to the plates retrieved after pyramidal search. Any detection with a score below that threshold will be ignored. 0.f being poor confidence and 1.f excellent confidence. Default: 0.8f
pyramidal_search_min_image_size_inpixels	INTEGER	[0, inf[	Minimum image size (max[width, height]) in pixels to trigger pyramidal search. Pyramidal search will be disabled if the image

			size is less than this value. Using pyramidal search on small images is useless. Default: 800.
pyramidal_search_quantization_enabled	BOOL		Whether to enable quantization on ARM devices. Please note that quantized functions never run on GPU as such devices are not suitable for integer operations. GPUs are designed and optimized for floating point math. Any function with dual implementation (GPU and Quantized) will be run on GPU if this entry is set to `false` and on CPU if set to `true`. Quantized inference bring speed but slightly decrease the accuracy. We think it worth it and you should set this flag to `true`. Anyway, if you're running a trial version, then an assertion will be raised when you try to set this entry to `false`. Default: `true`
recogn_score_type	STRING	min mean median max minmax	Defines the overall score type. The recognizer outputs a recognition score ([0.f, 1.f]) for every character in the license plate. The score type defines how to compute the overall score. `min`: Takes the minimum score. `mean`: Takes the average score. `median`: Takes the median score. `max`: Takes the maximum score. `minmax`: Takes (`max + min`) `* 0.5f` The `min` score is the more robust type as it ensure that every character have at least a certain confidence value. The `median` score is the default type as it provide a higher recall. In production we recommend using `min` type. Default: `median`. Recommended: `min`
recogn_minscore	FLOAT	]0.f, 1.f]	Define a threshold for the overall recognition score. Any recognition with a score below that threshold will be ignored. The overall score is computed based on `recogn_score_type`. Range: `]0.f, 1.f]` Default: `0.3f` `0.f` being poor confidence and `1.f` excellent confidence.
recogn_rectify_enabled	BOOLEAN	true	Whether to add rectification layer between the detector's output and the recognizer's input. A

		false	rectification layer is used to suppress the distortion. A plate is distorted when it's skewed and/or slanted. The rectification layer will deslant and deskew the plate to make it straight which make the recognition more accurate. Please note that you only need to enable this feature when the license plates are highly distorted. The implementation can handle moderate distortion without a rectification layer. The rectification layer adds many CPU intensive operations to the pipeline which decrease the frame rate. Default: `false`
recogn_rectify_polarity	STRING	both dark_on_bright bright_on_dark	This entry is only used when `recogn_rectify_enabled=true`. In order to accurately estimate the distortion we need to know the polarity. You should set the value to `both` to let the SDK find the real polarity at runtime. The module used to estimate the polarity is named the polarifier. The polarifier isn't immune to errors and could miss the correct polarity and this is why this entry could be used to define a fixed value. Defining a value other than `both` means the polarifier will be disabled and we'll assume all the plate have the defined polarity value. Default: `both`
recogn_rectify_polarity_preferred	STRING	both dark_on_bright bright_on_dark	This entry is only used when `recogn_rectify_enabled=true`. Unlike `recogn_rectify_polarity` this entry is used as a “hint” for the polarifier. The polarifier will provide more weight to the polarity value defined by this entry as tie breaker. Default: `dark_on_bright`
recogn_gpu_backend	STRING	opengl opencl nnapi metal none	Defines the GPU backend to use. This entry is only meaningful when `gpgpu_enabled=true` . You should not set this value and must let the stack choose the right value based on the system information. On desktop implementation, this entry will be ignored if support for CUDA is found. This value is also ignore when `recogn_quantization_enabled=true` as quantized operations are never executed on a GPU.
recogn_quantiz	BOOLEAN	true	Whether to enable quantization on ARM devices.

ation_enabled		false	Please note that quantized functions never run on GPU as such devices are not suitable for integer operations. GPUs are designed and optimized for floating point math. Any function with dual implementation (GPU and Quantized) will be run on GPU if this entry is set to `false` and on CPU if set to `true`. Quantized inference bring speed but slightly decrease the accuracy. We think it worth it and you should set this flag to `true`. Anyway, if you're running a trial version, then an assertion will be raised when you try to set this entry to `false`. Default: `true`

6 Sample applications

The source code comes with #4 sample applications: **Benchmark, VideoParallel**, **VideoSequential** and **ImageSnap**.

6.1 Benchmark

This application is used to check everything is ok and running as fast as expected. The imformation about the maximum frame rate (**47fps**) on Snapdragon 855 devices could be checked using this application. It's open source and doesn't require registration or license key.

6.2 VideoParallel

This application should be used as reference code by any developer trying to add ultimateALPR to their products. It shows how to detect and recognize license plates in realtime using live video stream from the camera.

ultimateALPR running on ARM device

6.3 VideoSequential

Same as VideoParallel but working on sequential mode which means slower. This application is provided to ease comparing the modes: Parallel versus Sequential.

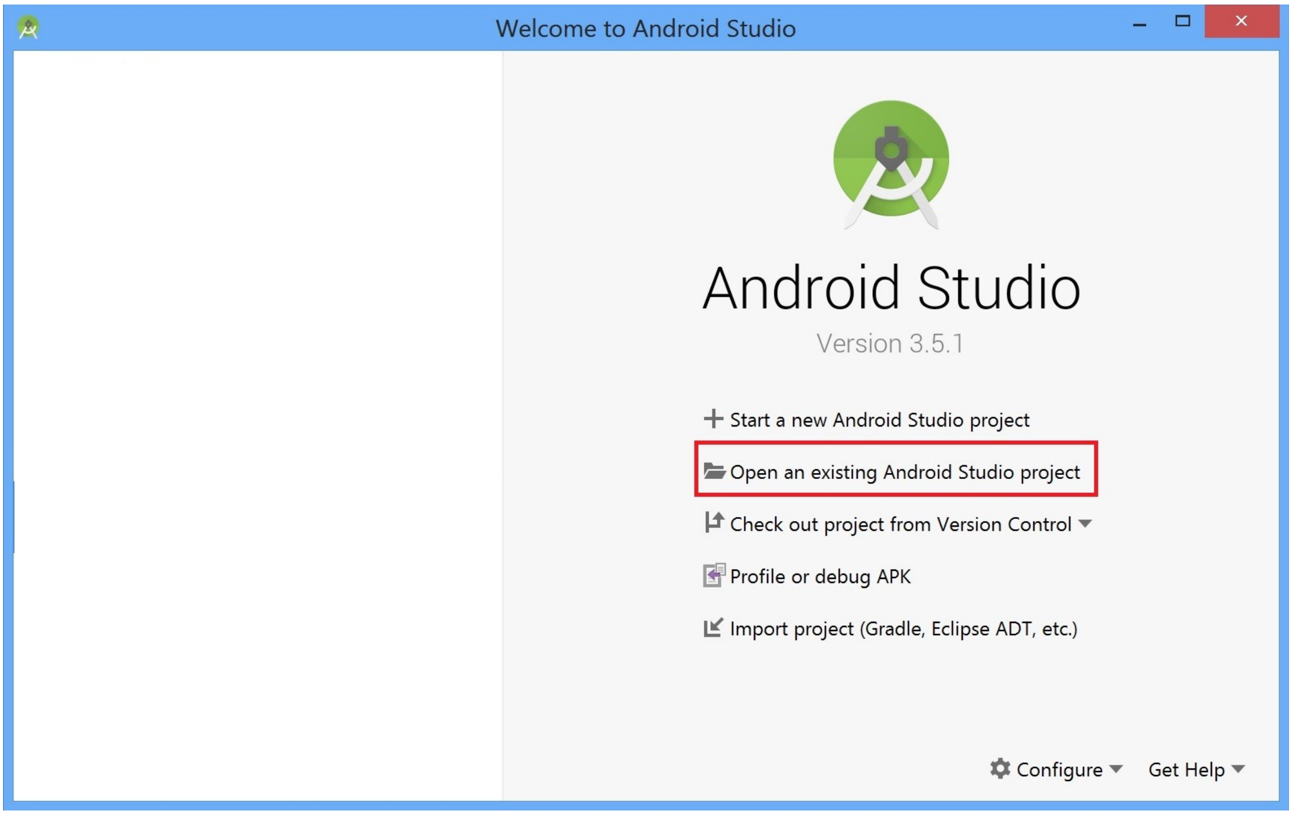
6.4 ImageSnap

This application reads and display the live video stream from the camera but only recognize an image from the stream on demand.

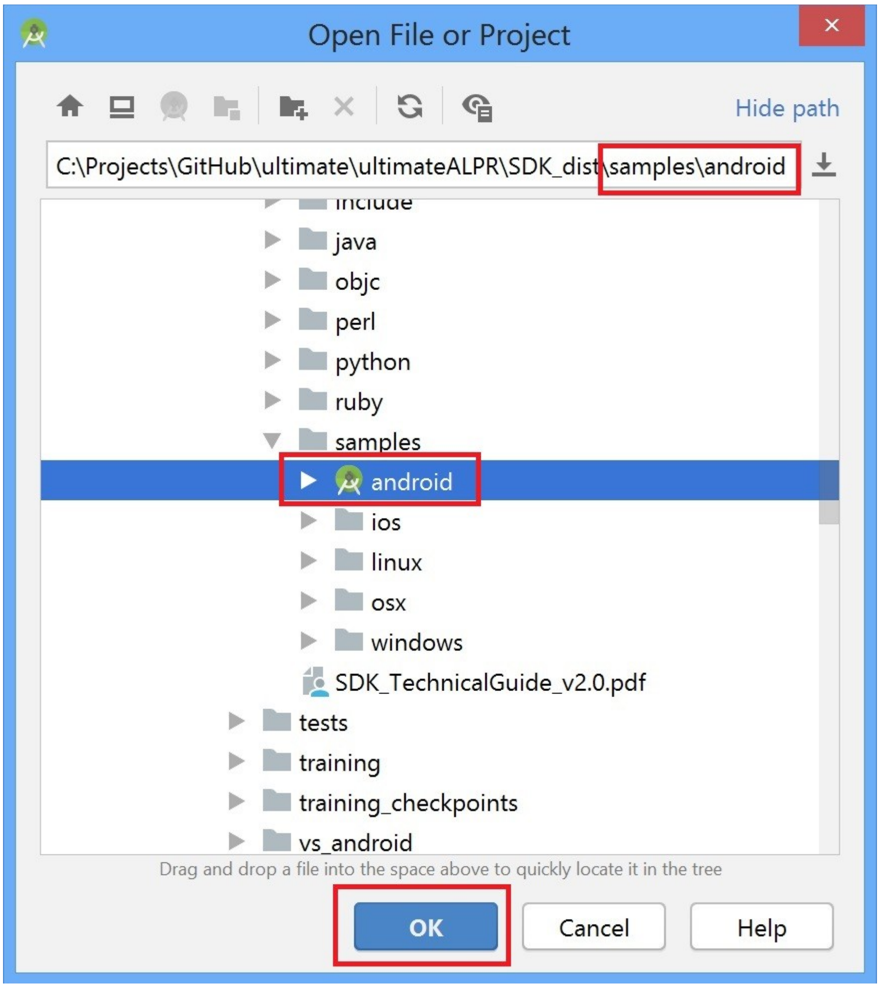
6.5 Trying the samples

To try the sample applications on Android:

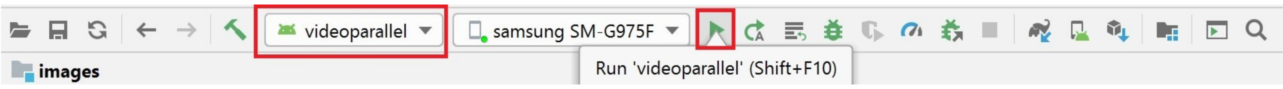
1. Open Android Studio and select "Open an existing Android Studio project"



2. Navigate to "<ultimateALPR-SDK>/samples", select "android" folder and click "OK"



3. Select the sample you want to try (e.g. "videoparallel") and press "run". Make sure to have the device on landscape mode for better experience.



7 Getting started

The SDK works on many platforms using many programming languages but this section focus Android and Java. Please check the previously section for more information on how to use the sample applications.

7.1 Adding the SDK to your project

The SDK is distributed as an Android Studio module and you can add it as reference or you can also build it and add the AAR to your project. I'm more comfortable with C++ on Visual Studio or Xcode and I find Android Studio buggy and very hard to configure. For example, when I downloaded OpenALPR for Android project for the benchmark (speed comparison) it failed to build because the gradle version wasn't install and I had to change it. Then, the maven repository was missing and I had to correct it. Then, the build task failed because the value defined in "compileSdkVersion" was not supported. Fixed the version and I still have warnings (mostly deprecated features) in *build.gradle*.

So, to make your life easier we'll not recommend referencing the SDK project in your application but just add references to the sources:

In your ***build.gradle*** file add:

```
android {
     ....

   sourceSets {
      main {
         jniLibs.srcDirs += ['path-to-your-ultimateALPR-SDK/binaries/android/jniLibs']
         java.srcDirs += ['path-to-your-ultimateALPR-SDK/java/android']
         assets.srcDirs += ['path-to-your-ultimateALPR-SDK/assets/models']
      }
   }
     ....
}
```

If you prefer adding the SDK as reference, then we assume you're an experimented developer and will find how to do it by yourself or just check the sample applications (they are referencing the SDK project instead of including the sources).

7.2 Using the API

It's hard to be lost when you try to use the API as there are only 3 useful functions: init, process and deInit.

.... add sample code here

Again, please check the sample applications for more information.

8 Parallel versus sequential processing

The ANPR/ALPR detector uses Convolutional Neural Networks (ConvNet/CNN). The shape for the input layer is [300,300,3] which means height=300, width=300 and NumBytesPerSample = 3 (R, G, B). Any image will be resized to a fixed 300x300 resolution and converted to RGB_888 (a.k.a RGB24) to match the input layer. The output layer is more complex and one important element is the prediction boxes which interest us in this section.

The CNN will always output #100 prediction boxes regardless the input. Here comes the "post-processing" operation which has the role to filter and fuse these boxes. The filtering is based on the scores/confidences and the fusion is based on the anchors. At the end of the post-processing operation you'll have the real detections which could be zero or up to #10 (arbitrary number used on training stage).

As you may expect the post-processing operation is very CPU intensive and makes the detection very slow, this is a bad news. For example, one operation executed in the post-processing stage is the NMS (non-maximum suppression).

The good news about the post-processing operation is that we can do it in #2 passes, the first one being very fast and allowing to have the real predictions with 98% accuracy. The second pass is very slow and we can schedule it to be executed in parallel to the next detection.

Here is the idea behind the parallel processing:

1. The decoder accepts a video frame N with any size, convert it to 300x300 RGB_888 and pass it to the CNN as input

2. Predicts #100 bounding boxes representing possible license plates

3. Run first pass post-processing operation to get candidate boxes with 98% accuracy

4. Asynchronously schedule second pass post-processing operation using a parallel process and register the result for recognition

5. Return the first pass result to the user. At this step the recognition isn't done yet but the user can use the first pass result to determine if the frame potentially have license plates. For example, the user can crop the frame using the license plate bounding box coordinates and save it for later.

6. The user provides video frame N+1 to the decoder.

6. While the user is preparing frame N+1 the decoder is running the second pass on the background and passed the result to the recognizer

7. Frame N+1 will have the same fate as frame N (see steps #**1** to #**5**)

As you have noticed, the preparation and detection operation for frame N+1 will overlap (parallel execution) with second pass detection and the recognition of frame N. This means you'll have the recognition result for frame N while you're in mid-process for frame N+1. When the pipeline is running at 47fps this means you'll have the recognition result for frame N within 5 to 10 milliseconds interval after providing frame N +1 for detection.

Please note that, if the first pass outputs **K** boxes and the second pass outputs **M** boxes then:

- every box **m** in **M** is in **K**, which means the second phase will never add new boxes to the prediction

On Android devices we have noticed that parallel processing can speedup the pipeline by up to 120% on some devices while on Raspberry Pi there the gain is marginal.

9 Rectification layer

The rectification layer is a dynamic module you can plug/unplug between the detector's output and the recognizer's input to rectify a license plate in order to suppress the distortion. A plate is distorted if it's slanted or skewed.

A license plate is attached on a moving car and is supposed to be undeformable. The camera could be also moving even if this is not the case for most scenarios. In such situation, every position the plate can take could be estimated using a 3x3 matrice. **This is the homography matrice.**

The goal for the rectification layer is to estimate the homography matrice using linear regression, compute it's inverse, multiply it with every pixel from the detector's output and provide the warped pixels to the recognizer's input.

As you may expect, this process is time consuming and is disabled by default when the SDK is running on ARM devices. You should not worry about its absence as the default code can already handle moderately distorted plates.

See the configuration section on how to enable/disable the rectification layer.

The next image shows how the rectification layer transforms an image to remove the skew and slant.

Left: before rectification. **Right:** After rectification

9.1 Polarity

There are two polarities: *DarkOnBright* and *BrightOnDark*.

DarkOnBright: The numbers on the license plate are darker than the background. Example: Black numbers on white background (European license plates).

BrightOnDark: The numbers on the license plate are brighter than the background. Example: White numbers on blue background (Chinese license plates).

Unlike other implementations we don't use the four corners from the license plate to estimate the homography matrice because such implemention wouldn't be robust to high distortions or heavy noise. Instead, we use every edge on the plate to estimate the skew and shear/slant angles. Theses angles are combined with the x/y scales (size normalization) to build a 3x3 rectification matrice (our homography matrice).

The edges directions are very important and this is why we need to know what the polarity. The code contains a polarifier which can estimate the polarity but it's not immune to errors. To help the polarifier you can define a preferred polarity and even better you can restrict it if your country uses single polarity. See the configuration section for more info.

See the configuration section on how to give a “hint” for the polarity.

10 Pyramidal search

As explained in the previous sections, the detector uses a Convolutional Neural Network with a **[300, 300, 3] input layer**. This means any image presented to the detection pipeline will be resized to 300x300 and converted to RGB_888 format regardless its resolution.

Using a low resolution speedup the inference function and using a fixed shape instead of ratio-based scaling improves generalization and speedup the training process. This is obviously an issue when the image is very large and the license plates very small or far away. Small or far away license plates on large images tend to disappear when the image is downscaled with a 2+ factor.

To fix the above issue we could choose an input layer with higher resolution (e.g. [512, 512, 3] instead of [300, 300, 3]) as done by many ANPR solutions. The problem is that higher resolution comes with higher latency and memory usage. **The ANPR solutions we tried barely reach 1fps (detection only) on Raspberry Pi 4 while our implementation can run at 12fps**. To keep this frame rate while being able to accurately detect small and far away plates we spent huge amount of time in R&D to come with a very fast and accurate solution. To make it short: **Scale the features not the image**.

You don't need to understand how the pyramidal search works in order to use it (see "`pyramidal_search_enabled`" config to enable/disable) but some basic technical information may help you debug issues:

1. The features are extracted from the input image and defined as base layer.
2. If quantization is enabled, then the features are converted from float32 to int8 and normalized ([-127, 128]).
3. The detection pipeline is partially executed on the base layer without the Non-max Suppression (NMS) step. The variable **n** is initialized with integer value 1. **This is the first pass (1st-pass)**.
4. A refinement function with binary output is done on the result (bounding boxes and scores) from nth-pass. If the output is 0 or n>6 then, the process is stopped and we move to step 7. Otherwise (output is 1 and n <=6), we move to step 5.
5. The features from nth-pass are scaled by X and the detection pipeline is partially executed again. This execution is almost 7 times faster than nth-pass as there are less layers and neurons (lower depth multiplier).
6. Variable n is incremented and we loop back to step 4.
7. Resume the detection pipeline with NMS and other post-processing operations.

As you've noticed, **there are 6 pyramidal** levels and "`pyramidal_search_sensitivity`" configuration entry controls how many are needed. The sensitivity also controls the depth multiplier which defines the number of neurons. The higher the sensitivity is, the higher the number of pyramidal levels and neurons will be. More levels means better accuracy but higher CPU usage and inference time. Default value: 0.28.

For example, you can use **pyramidal search to monitor at realtime a 5-lane highway using a single long-range camera**.

Let's be very concrete and try with a sample image. The next image is from The Verge article titled "Privacy advocate held at gunpoint after license plate reader database mistake, lawsuit alleges". You can find it here.

Original image:

ANPR result using Pyramidal search:

ANPR result without Pyramidal search:

You can clearly see that we miss the two furthest plates when pyramidal search is disabled.
The same test could be done using our online cloud-based demo web application hosted at https://www.doubango.org/webapps/alpr/ . You can also use the original image with ALPR / ANPR products provided by other companies for comparison.

11 Muti-threading design

No forking, minimal context switch. Doubango vs Others

12 Memory management design

This section is about the memory management design.

12.1 Memory pooling

The SDK will allocate at maximum 1/20th of the available RAM during the application lifetime and manage it using a pool. For example, if the device have 8G memory, then it will start allocating 3M memory and depending on the malloc/free requests this amount will be increased with 400M (1/20th of 8G) being the maximum. Most of the time the allocated memory will never be more than 5M.

Every memory allocation or deallocation operation (malloc, calloc, free, realloc...) is hooked which make it immediate (no delay). The application allocates and deallocates aligned memory hundreds of time every second and thanks to the pooling mechanism these operations don't add any latency.

We found it was interesting to add this section on the documentation so that the developers understand why the amount of allocated memory doesn't automatically decrease when freed. You may think there are leaks but it's probably not the case. Please also note that we track every allocated memory or object and can automatically detect leaks.

12.2 Minimal cache eviction

Thanks to the memory pooling when a block is freed it's not really deallocated but put on the top of the pool and reattributed at the next allocation request. This not only make the allocation faster but also minimize the cache eviction as the fakely freed memory is still hot in the cache.

12.3 Aligned on SIMD and cache line size

Any memory allocation done using the SDK will be aligned on 16bytes on ARM and 32bytes on x86. The data is also strided to make it cache-friendly. The 16bytes and 32bytes alignment values aren't arbitrary but chosen to make ARM NEON and AVX functions happy.

When the user provides non-aligned data as input to the SDK, then the data is unpacked and wrapped to make it SIMD-aligned. This introduce some latency. Try to provide aligned data and when choosing region of interest (ROI) for the detector try to use SIMD-aligned left bounds.

```
(left & 15) == 0; // means 16bytes aligned
(left & 31) == 0; // means 32bytes aligned
```

12.4 Cache blocking

To be filled

13 Improving the accuracy

The code provided on Github (https://github.com/DoubangoTelecom/ultimateALPR-SDK) comes with default configuration to make everyone almost happy. You may want to increase the speed our accuracy to match your use case.

13.1 Detector

This section explains how to increase the accuracy for the detection layer.

13.1.1 Far away or very small plates

This section explains how to improve accuracy on very small or far away plates.

13.1.1.1 Region of interest

As explained in the previous sections, the detector expects a 300x300 image as input. Regardless the input size the detector will always downscale it at 300x300 and convert it to RGB_888.

When a plate is far away or very small and the image too large, then downscaling it to 300x300 make such plates almost disappear.

Let's consider the next 1280x720 image:

The license plates on the Renault and Mercedes-Benz are correctly detected but not the one on the volkswagen (VW).

The issue is that the license plate on the VW is far away or relatively small compared to the image size. Let's resize the image at 300x300 and see what the CNN have as input:

We can clearly see that at 300x300 the plate on the VW is undetectable.

In fact the issue isn't that the plate is small in terms of pixels but in percentage relative to the image size.

To fix the issue, select a region of interest (see previous sections on how to define a ROI) to make the plate size in percentage higher. Let's take a 1100x333 ROI:

The 1100x333 ROI defines a region where we expect to have a license plate and ignore everything else (the sky, the buildings...).

Let's crop the ROI:

Let's resize the cropped ROI at 300x300:

Now you can see that the license plate on the VW is clear and can be reliably detected.

Another solution would be detecting the car first which will always work as its size is large relative to the overall image:

Then, resizing the car at 300x300 and detecting the license plate:

All the steps described in this section are automatically done by the SDK when you define a ROI. You don't need to write a single line of code to crop or resize the input image.

Another elegant way to detect license plates with any size is to enable pyramidal search. See next section for more information.

13.1.1.2 Pyramidal search

This function was added in version 2.4 and is the recommended way to detect small or far away plates on large images.

We recommend enabling pyramidal search if the video resolution is equal or more than 720p.

13.1.2 Score threshold

The configuration section explains how to set the minimum detection score.

If you have too many false-positives, then increase the detection score in order to increase the precision.

If you have too many false-negatives, then decrease the detection score in order to increase the recall.

13.1.3 Matching training data

The training data for the detection predominantly contains license plate mounted on a car. There are very few images of license plates alone. To increase the detection accuracy you should provide images showing both the license plate and the car.

For example, detecting license plate on the next image will be done with the highest accuracy possible (99.99%):

While detecting the license plate on the next image will be done with very low accuracy or even fail:

The fact that the training data predominantly contains images showing both the license plate and the car while there are few images with isolated plates is done on purpose. When you're filming an outdoor scene, then there are many traffic signs or billboards looking very similar to license plates (strong borders with regular text inside). Adding a car as precondition helps get ride of false positives. When the SDK is correctly configured you'll almost never see false-positives.

13.2 Recognizer

This section explains how to increase the accuracy for the recognizer layer.

13.2.1 Adding rectification layer

When the license plates are highly distorted (skewed and/or slanted) you'll need to activate the rectification layer to remove the distortion. The configuration section explains how to activate the rectification layer.

13.2.2 Score threshold

The configuration section explains how to set the minimum recognition score.

If you have too many false-positives, then increase the detection score in order to increase the precision.

If you have too many false-negatives, then decrease the detection score in order to increase the recall.

13.2.3 Restrictive score type

The configuration section explains the different supported score types: "min", "mean", "median", "max" and "minmax".

The "min" score type is the more restrictive one as it ensures that every character on the license plate have at least the minimum target score.

The "max" score type is the less restrictive one as it only ensures that a least one of the characters on the license plate have the minimum target score.

The "median" score type is a good trade-off between the "min" and "max" types.

We recommend using “min” score type.

14 Improving the speed

This section explains how to improve the speed (frame rate).

14.1 Parallel mode

Activate the parallel mode as explained in the previous sections. Please note that this won't change the accuracy while your application will run up to #2 times faster than the sequential mode.

14.2 Memory alignment

Make sure to provide memory aligned data to the SDK. On ARM the preferred alignment is 16bytes while on x86 it's 32bytes. If the input data is an image and the width isn't aligned to the preferred alignment size, then it should be strided. Please check the memory management section for more information.

14.3 Landscape mode

When the device is on portrait mode, then the image is rotated 90 or 270 degree (or any modulo 90 degree). On landscape mode it's rotated 0 or 180 degree (or any modulo 180 degree). On some devices the image could also be horizontally/vertically mirrored in addition to being rotated.

Our deep leaning model can natively handle rotations up to 45 degree but not 90, 180 or 270. There is a pre-processing operation to rotate the image back to 0 degree and remove the mirroring effect but such operation is time consuming on mobile devices. We recommend using the device on landscape mode to avoid the pre-processing operation.

14.4 Removing rectification layer

On ARM devices you should not add the rectification layer which introduces important delay to the inference pipeline. The current code can already handle moderately distorted license plates.

If your images are highly distorted and require the rectification layer, then we recommend changing the camera position or using multiple cameras if possible. On x86, there is no issue on adding the rectification layer.

14.5 Disabling pyramidal search

Pyramidal search function is used to detect very small and far away plates. The refiner (explained above) is smart enough to not build additional pyramidal levels when there is no small or far away plate on the image. But, its smartness have bounds: additional levels are built when the image is too complex (many edges). More levels means higher CPU usage and inference time.

Disable this feature if the license plates on the video stream are close and large enough. Unless you're filming a highway using a long-range camera you don't need this feature.

14.6 Planar formats

Both the detector and recognizer expect a RGB_888 image as input but most likely your camera doesn't support such format. Your camera will probably output YUV frames. If you can choose, then prefer the planar formats (e.g YUV420P) instead of the semi-planar ones (e.g. YUV420SP a.k.a NV21 or NV12). The issue with semi-planar formats is that we've to deinterleave the UV plane which takes some extra time.

14.7 Reducing camera frame rate and resolution

The CPU is a shared resource and all background tasks are fighting each other for their share of the resources. Requesting the camera to provide high resolution images at high frame rate means it'll take a big share. It's useless to have any frame rate above 25fps or any resolution above 720p (1280x720) unless you're monitoring a very large zone and in such case we recommend using multiple cameras.

15 Benchmark

It's easy to assert that our implementation is the fastest you can find without backing our claim with numbers and source code freely available to everyone to check.

See the next section for more information.

15.1 UltimateALPR versus OpenALPR on Android

We've found 3 OpenALPR repositories on Github:

1. https://github.com/SandroMachado/openalpr-android [708 stars]
2. https://github.com/RobertSasak/react-native-openalpr [338 stars]
3. https://github.com/sujaybhowmick/OpenAlprDroidApp [102 stars]

We've decided to go with the one with most stars on Github which is [1]. We're using recognizeWithCountryRegionNConfig(country="us", region="", topN = 10).

Rules:

- We're using Samsung Galaxy S10+ (Snapdragon 855)
- For every implementation we're running the recognition function within a loop for #1000 times.
- The positive rate defines the percentage of images with a plate. For example, 20% positives means we will have #800 negative images (no plate) and #200 positives (with a plate) out of the #1000 total images. This percentage is important as it allows timing both the detector and recognizer.
- All positive images contain a single plate.
- Both implementations are initialized outside the loop.

	0% positives	**20% positives**	**50% positives**	**70% positives**	**100% positives**
ultimateALPR	21344 millis 46.85 fps	25815 millis 38.73 fps	29712 millis 33.65 fps	33352 millis 29.98 fps	37825 millis 26.43 fps
OpenALPR	715800 millis 1.39 fps	758300 millis 1.31 fps	819500 millis 1.22 fps	849100 millis 1.17 fps	899900 millis 1.11 fps

One important note from the above table is that the detector in OpenALPR is very slow and 80% of the time is spent trying to detect the license plates. This could be problematic as most of the time there is no plate on the video stream (negative images) from a camera filming a street/road and in such situations an application must run as fast as possible (above the camera maximum frame rate) to avoid dropping frames and loosing positive frames. Also, the detection part should burn as less as possible CPU cycles which means more energy efficient.

The above table shows that **ultimateALPR is up to 33 times faster than OpenALPR.**

To be fair to OpenALPR:

1. The API only allows providing a file path which means for every loop they are reading and decoding the input while ultimateALPR accepts raw bytes.
2. There is no ARM64 binaries provided and the app is loading the ARMv7 versions.

Again, our benchmark application is open source and doesn't require registration or license key to try. You can try to make the same test on your own device and please don't hesitate to share your numbers or any feedback if you think we missed something.

15.2 Intel® Xeon® E3 1230v5 CPU with GTX 1070 GPU (Untuntu 18)

We recommend using a computer with NVIDIA GPU unleash ultimateALPR speed. Next numbers are obtained using GeForce GTX 1070 GPU and Intel® Xeon® E3 1230v5 CPU on Ubuntu 18.

	0% positives	**20% positives**	**50% positives**	**70% positives**	**100% positives**
Intel® Xeon® E3 1230v5 GTX 1070	9516 millis **105.07 fps**	9963 millis 100.36 fps	10701 millis 93.44 fps	11109.millis 90.01 fps	11704 millis 85.43 fps

15.3 Core i7 (Windows)

Both i7 CPUs are 6yr+ old (2014) to make sure everyone can easily find them at the cheapest price possible.

	0% positives	**20% positives**	**50% positives**	**70% positives**	**100% positives**
i7-4790K (Windows 7)	4251 millis **23.52 fps**	4598 millis 21.74 fps	4851 millis 20.61 fps	5117 millis 19.54 millis	5553 millis 18.00 fps
i7-4770HQ (Windows 10)	6040 millis **16.55 fps**	6342 millis 15.76 fps	7065 millis 14.15 fps	7279 millis 13.73 fps	7965 millis 12.55 fps

15.4 Raspberry Pi

The Github repository contains Raspberry Pi benchamark application to evaluate the performance pi version 3 and later.

More information on how to build and use the application could be found at https://github.com/DoubangoTelecom/ultimateALPR-SDK/blob/master/samples/c++/benchmark/README.md.

Please note that even if Raspberry Pi 4 have a 64-bit CPU Raspbian OS uses a 32-bit kernel which means we're loosing many SIMD optimizations.

	0% positives	**20% positives**	**50% positives**	**70% positives**	**100% positives**
Raspberry Pi 4	81890 millis **12.21 fps**	89770 millis 11.13 fps	115190 millis 8.68 fps	122950 millis 8.13 fps	141460 millis 7.06 fps

16 Best JSON config

Here is the best config we recommend:

- Enable parallel mode: Regardless your use case this is definetly the mode to use. It's faster and provide same accuracy as the sequential mode. If you think it's not suitable for you, then please let us know and we'll explain how to use it.
- YUV420P image format as input: In fact the best format would be RGB24 but your camera most likely dosen't support it. You should prefer YUV420P instead of YUV420SP (NV12 or NV21) as the later is semi-planar wich means the UV plane is interleaved. De-interleaving the UV plane take some extra time.
- 720p image size: Higher the image size is better the quality will be or the recognition part. 720P is a good trade-off between quality and resource consumption. Higher image sizes will give your camera a hard time wich means more CPU and memory usage.
- Define a 800x600 region of interest: As explained in the previous sections, regardless the input size the image will always be rized to 300x300 for the detector. Defining a region of interest as close as possible to 300x300 will improve the detection of small or far plates. JSON config: *"detect_roi": [240.f, 1040.f, 60.f, 660.f]*
- 10% for minimum detection score: This looks low but it'll improve your recall. Off course it's decrease your precision but you should be worry about it as the score from the recognizer will be used to get ride of these false-positives. Please don't use any value lower than 5% as it'll increase the number of false-positives which means more images to recognize which means more CPU usage. JSON config: *"detect_minscore": 0.1*
- 30% for minimum recognition score: This score is very low and make sense if the score type is "min". This means every character on the license plate have an accuracy at least equal to 0.3. For example, having a false-positive with #5 chars and each one is recognized with a score > 0.3 is very unlikely to happen. If you're planning to use "median", "mean", "max" or "minmax" score types, then we recommend using a mimimum score at 70% or higher. JSON config: *"recogn_score_type": "min", "recogn_minscore": 0.3*

The configuration should look like this:

```
{
	"debug_level": "warn",
	"detect_roi": [240.f, 1040.f, 60.f, 660.f],
	"detect_minscore": 0.1,
	"recogn_score_type": "min",
	"recogn_minscore": 0.3
}
```

17 Debugging the SDK

The SDK looks like a black box and it may look that it's hard to understand what may be the issue if it fails to recognize an image.

Here are some good practices to help you:

1. Set the debug level to "verbose" and filter the logs with the keyword "doubango". JSON config: *"debug_level": "verbose"*
2. Maybe the input image has the wrong size or format or we're messing with it. To check how the input image looks like just before being forward to the neural networks enable dumping and set a path to the dump folder. JSON config: *"debug_write_input_image_enabled": true, "debug_internal_data_path": "`<path to dump folder>`"*. Check the sample applications to see how to generate a valid dump folder. The image will be saved on the device as "`ultimateALPR-input.png`" and to pull it from the device to your desktop use adb tool like this: `adb pull <path to dump folder>/ultimateALPR-input.png`
3. The computer vision part is open source and you can match the lines on the logs to https://github.com/DoubangoTelecom/compv

18 Frequently Asked Questions (FAQ)

18.1 Why the benchmark application is faster than VideoParallel?

The VideoParallel application has many background threads to: read from the camera, draw the preview, draw the recognitions, render the UI elements... The CPU is a shared resource and all these background threads are fighting against each other for their share of the resource.

18.2 Why the image is 90% rotated?

When the device is on portrait the image from the camera is rotated 90% to match the screen size. I swear it's not our fault :). We could rotate the image back -90% to have the image on the right orientation but this would increase the processing time and we prefer to let the end user decide what to do. Anyways, put the device on landscape and the image will have the right orientation. Rotating an image by 90% is very easy, $f: (x,y) \rightarrow (y, x)$

18.3 Why does the detector fail to accurately detect far away or very small plates?

As explained in the documentation, the input layer will resize any image to a fixed 300x300x3 resolution and small or far away plates may become undetectable after such process. You can enable pyramidal search feature (will add some delay to the detection pipeline) to fix such issue.

19 Known issues

On ARM32 devices multi-threading is partially disabled. See https://github.com/DoubangoTelecom/ultimateALPR-SDK/issues/3. This have very low priority as all modern ARM devices have 64-bit CPUs.

Please use the issue tracker to open new issue: https://github.com/DoubangoTelecom/ultimateALPR-SDK/issues