Doubango AI

State of the art **MICR (Magnetic ink character recognition)** implementation for embedded devices (ARM) and desktops (x86) using deep learning
https://github.com/DoubangoTelecom/ultimateMICR-SDK

Table of Contents

This is a short technical guide to help developers and integrators take the best from our **MICR** (**M**agnetic **i**nk **c**haracter **r**ecognition) 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

This is state-of-the-art **Magnetic ink character recognition(MICR)** dectector and recognizer using deep learning.

Unlike other solutions you can find on the web, you don't need to adjust the camera/image to define a **R**egion **O**f **I**nterest (ROI). We also don't try to use small ROI to decrease the processing time or false-positives. The whole image (up to 4K supported) is processed and every pixel is checked. No matter if the MICR lines are **small, far away, blurred, partially occluded, skewed** or **slanted**, our implementation can accurately detect and recognize every character.

The detector is agnostic and doesn't decode (recognize/OCR) the text to check it against some pre-defined rules (regular expressions) which means **we support all MICR types** regardless the font, content, shape or country. **Both E-13B and CMC-7 formats are supported.**

Automating Bank account information extraction from MICR (Magnetic ink character recognition) zones on scanned bank checks/document **above human-level accuracy** is a very challenging task. Our implementation reaches such level of accuracy using latest deep learning techniques. We outperform both ABBYY and LEADTOOLS in terms of accuracy and speed (**almost #30 times faster**).

Using a single model we're able to accurately locate the MICR zones, infer the type (**E-13B** or **CMC-7**) and recognize the fields: **one-shot deep model**. The performance gap between us and the other companies is more important for **CMC-7** format which is more challenging than **E-13B**.

VideoRecognizer-CMC7 sample application running on Android (Galaxy S10+)

VideoRecognizer-E13B sample application running on Android (Galaxy S10+)

This technology is a key component of Remote Deposit Capture applications to process bank checks sent using mobile phones or scanners. It's a **must have** technology for any FinTech company.

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/ultimateMICR-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/micr/ to check out the accuracy and precision before starting to play with the SDK.

2 Supported countries

As explained in the intro, we don't try to parse the MICR line to check the validity. Our neural network works like a human brain to detect the MICR lines regardless the content. The big advantage with such method is that we don't need to define a list of supported countries or formats. **All countries and formats are supported.**

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 4** and many custom embedded devices (e.g. scanners).

The Github repository (https://github.com/DoubangoTelecom/ultimateMICR-SDK) contains binaries for **Android, Linux, Windows** and **Raspberry Pi** as reference code to allow developers to test the implementation. These reference implementations come with **Java**, **Obj-C**, **Python** and **C++** APIs. The API is common to all operating systems which means you can develop and test your application on **Android, Linux, Windows** 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 and OCR parts.

In addition to being GPGPU accelerated the implementation is SIMD accelerated. The GPU implementation requires support for 64-bit floating point math (**cl_khr_fp64** extension) which is not available on most of the **ARM Mali GPUs**. On such devices the code is massively multi-treaded and accelerated using **assembler code** and **NEON** instructions.

When you run the code on GPU devices without support for **cl_khr_fp64** extension then, you'll have the next message:

```
W org.doubango.compv: **[COMPV WARN]: function: "newObj()"
W org.doubango.compv: file: "..\source\ml\ultimate_base_ml_predict_rbf.cxx"
W org.doubango.compv: line: "153"
W org.doubango.compv: message: [UltBaseMachineLearningPredictRBF] GPGPU instance
requested but failed as double precision extension (cl_khr_fp64) is missing
```

You can safely ignore the warning.

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**, **Y(Grayscale)**, **NV12**, **NV21**, **YUV420P**, **YVU420P**, **YUV422P** and **YUV444P**. NV12 and NV21 are semi-planar formats also known as **YUV420SP**. Any modern camera will support at least one of these format.

The list of supported formats is wide enough to make sure any camera will work.

3.6 Optimizations

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

- Hand-written assembler (YASM for x86 and GNU ASM for ARM)
- SIMD (SSE, AVX, NEON) using intrinsics or assembler
- GPGPU (OpenCL 1.2+) acceleration
- Massively multithreaded
- Smart multithreading (minimal context switch, no forking, 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
- 8-bit Quantization
- ... 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 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 (no forking). 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 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 cores. For example, on an octa-core device the maximum number of threads will be #8, on quad-core it will be #4. 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. This option could also be ignored when the memory transfer time is high compared to the computation time.
			Default: `true`

`gpgpu_workload_balancing_enabled`	`BOOLEAN`	`true` `false`	A device contains a CPU and a GPU. Both can be used for math operations. This option allows using both units. On some devices the CPU is faster and on other it's slower. When the application starts, the work (math operations to perform) is equally divided: 50% for the CPU and 50% for the GPU. Our code contains a profiler to determine which unit is faster and how fast (percentage) it is. The profiler will change how the work is divided based on the time each unit takes to complete. This is why this configuration entry is named "workload balancing". Default: `false` for x86 and `true` for ARM
`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 "`$(assets_folder)/models`" folder and configuration files in "`$(assets_folder)`". Default: `.`
`format`	`String`	`e13b` `cmc7`	Defines the MICR format to enable for the

		`e13b+cmc7`	detection. Use `"e13b"` to look for E-13B lines only and `"cmc7"` for CMC-7 lines only. To look for both, use `"e13b+cmc7"`. For performance reasons you should not use `"e13b+cmc7"` unless you really expect the document to contain both E-13B and CMC7 lines. Default: `"e13b+cmc7"`
`roi`	`FLOAT[4]`	`Any`	Defines the Region Of Interest (ROI) for the detector. Any pixels outside the 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]`

`segmenter_accuracy`	`STRING`	`veryhigh`
		`high`
		`medium`
		`low`
		`verylow`
			Before calling the classifier to determine whether a zone contains a MICR line we need to segment the text using multi-layer segmenter followed by clustering. The multi-layer segmenter uses **hysteresis** for the voting process using a [min, max] double thresholding values. This configuration entry defines how low the thresholding values should be. Lower the values are, higher the number of fragments will be and higher the recall will be. High number of fragments means more data to process which means more CPU usage and higher processing time.
			Default: `high`

`interpolation`	`STRING`	`nearest`
		`bilinear`
		`bicubic`
			Defines the interpolation method to use when pixels are scaled, deskewed or deslanted. `bicubic` offers the best quality but is slow as there is no SIMD or GPU acceleration yet. `bilinear` and `nearest` interpolations are multithreaded and SIMD accelerated. For most scenarios `bilinear` interpolation is good enough to provide high accuracy/precision results while the code still runs very fast.
			Default: `bilinear`

`score_type`	`String`	`min`	Defines the overall score type. The recognizer

		`mean`
		`median`
		`max`
		`minmax`
			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`

`min_score`	`FLOAT`	`[0.f, 1.f]`	Defines a threshold for the recognition score/confidence. Any recognition with a score below that threshold will be ignored/removed. This value could be used to filter the false-positives and improve the precision. Low value will lead to high recall and low precision while a high value means the opposite. Range: `[0.f, 1.f]` Default: `0.0f` `0.f` being poor confidence and `1.f` excellent confidence.

5 Sample applications

The source code comes with #3 sample applications: **Benchmark**, **MICRVideoRecognizer-E13B** and **MICRVideoRecognizer-CMC7**. All sample applications are open source and don't require registration or license key.

5.1 Benchmark

This application is used to check everything is ok and running as fast as expected. The information about the maximum frame rate on ARM devices could be checked using this application.

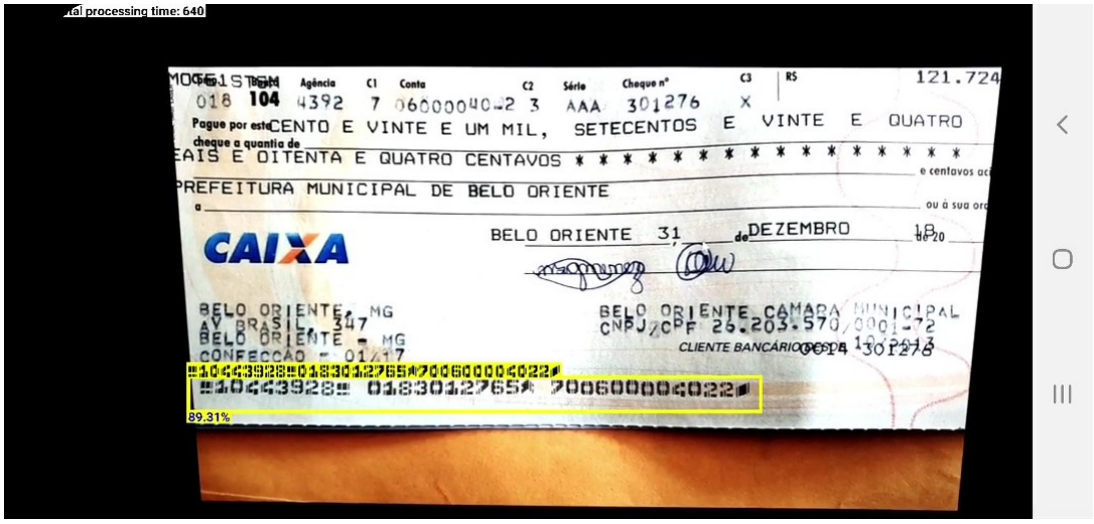
5.2 MICRVideoRecognizer-E13B

This application should be used as reference code by any developer trying to add ultimateMICR to their products. It shows how to detect and recognize **MICR E-13B** lines in realtime using live video stream from the camera.

UltimateMICR (E-13B) running on Android (Galaxy S10+)

5.3 MICRVideoRecognizer-CMC7

This application should be used as reference code by any developer trying to add ultimateMICR to their products. It shows how to detect and recognize **MICR CMC-7** lines in realtime using live video stream from the camera.



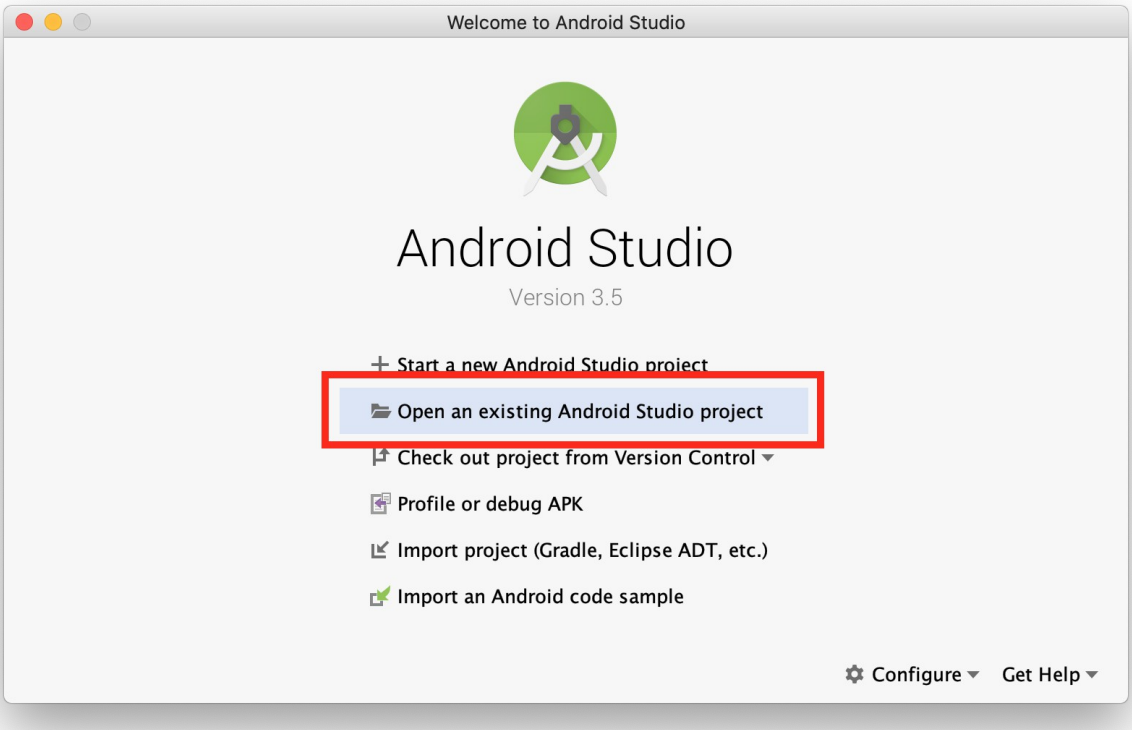
UltimateMICR (CMC-7) running on Android (Galaxy S10+)

5.4 Trying the samples

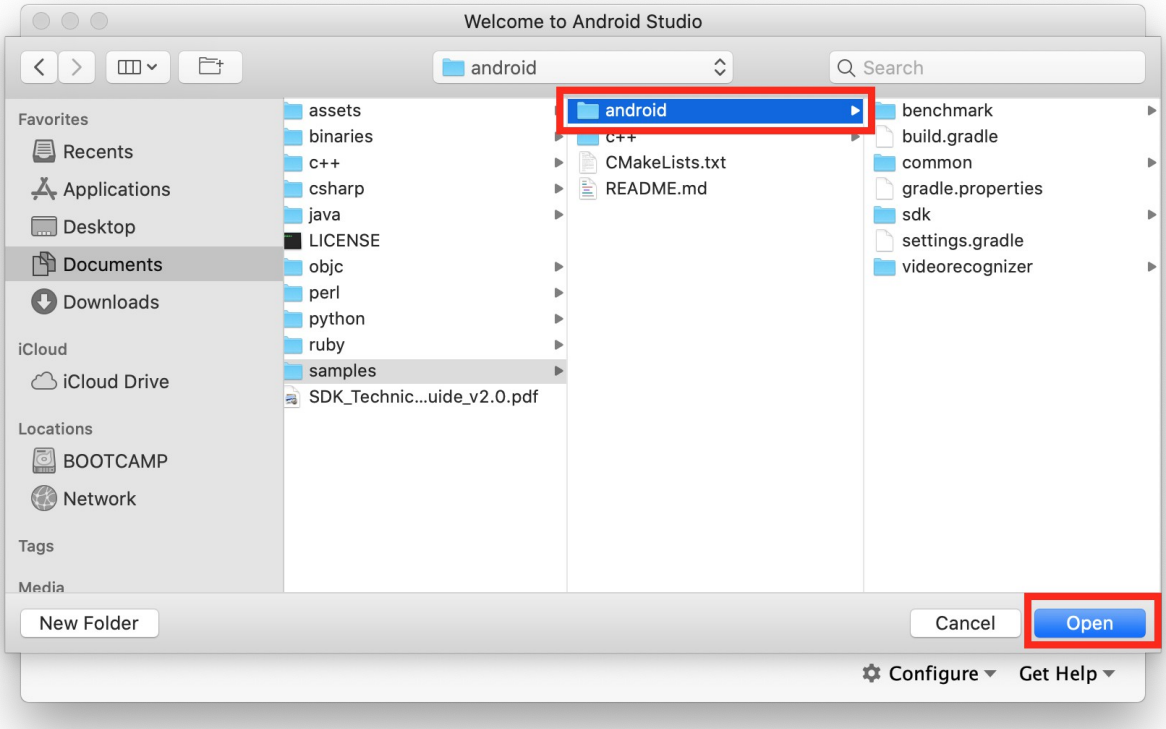
5.4.1 Android

To try the sample applications on Android:

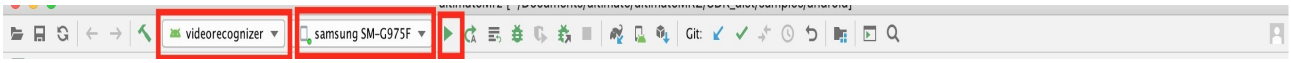
1. Open Android Studio and select "Open an existing Android Studio project"



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



3. Select the sample you want to try (e.g. "videorecognize"), the device (e.g. "samsung SM-G975F") and press "run".



5.4.2 Linux, Windows, Raspberry Pi and Others

For Raspberry Pi and other Linux systems you need to build the sample applications from source. More info at https://github.com/DoubangoTelecom/ultimateMICR-SDK/samples/c%2B%2B/README.md

6 Getting started

Please check the previous section for more information on how to use the sample applications.

6.1 Adding the SDK to your project

The Github repository contains binaries for Android, and Raspberry Pi. The next sections explain how to add the SDK to an existing project.

6.1.1 Android

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. But, the easiest way to add the SDK to your project is by directly including the source.

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

```
android {
        ....

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

6.1.2 Raspberry Pi, Windows and Others

The shared libraries are under "ultimateMICR-SDK/binaries/**<platform>**". The header file at "ultimateMICR-SDK/c++". You can use any C++ compiler.

6.2 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 on how to use the API.

7 Muti-threading design

No forking, minimal context switch. Doubango vs Others

8 Memory management design

This section is about the memory management design.

8.1 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.

8.2 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.

8.3 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
```

8.4 Cache blocking

To be filled

9 Improving the accuracy

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

9.1 Detector

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

9.1.1 Segmenter accuracy

As explained in the configuration section the segmenter accuracy accepts 5 values (JSON strings): **`veryhigh`**, **`high`**, **`medim`**, **`low`**, and **`verylow`**. The default value is **`high`**.

If the SDK fails to detect some MICR lines then, consider using **`veryhigh`** accuracy. With **`veryhigh`** value the detection accuracy will increase but this comes with a cost: *high CPU usage and slow detection*. We recommend increasing the image resolution and making sure that the MICR lines are as straight as possible instead of changing the accuracy level.

9.1.2 Region of interest

Unlike other applications you can find on the market we don't define a region of interest (ROI), the entire frame is processed to look for MICR lines. Setting a ROI could decrease the false-positives and improve the precision score without decreasing the recall value. The deep learning model used for the detection is very accurate (high precision and high recall) and should not output false-positives if you're using the default recommended values.

9.2 Recognizer

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

9.2.1 Interpolation

As explained in the configuration section, the interpolation operation accepts 3 values (JSON strings): **`bicubic`**, **`bilinear`**, and **`nearest`**. The default value is **`bilinear`**.

The interpolation operations are used when pixels are scaled, deskewed or deslanted. **`bicubic`** offers the best quality but is slow as there is no SIMD or GPU acceleration yet. **`bilinear`** and **`nearest`** interpolations are multithreaded and SIMD accelerated. For most scenarios **`bilinear`** interpolation is good enough to provide high accuracy/precision results while the code still runs very fast.

Change the interpolation value to **bicubic** if you're having low recognition score.

9.2.2 Score threshold

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

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

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

9.2.3 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.

10 Improving the speed

Our implementation is massively multithreaded, SIMD and GPGPU accelerated but on some low-end devices it could be slow.

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

10.1 Restrict the format

By default the detector will search for **both E-13B and CMC-7** lines to make sure the application will work for all formats. Depending on your country you'll only need to detect one format.
To speed up the detection process we recommend changing the format (JSON configuration entry: `"format"`) to `"e13b"` or `"cmc7"` instead of `"e13b+cmc7"`.

10.2 GPU / CPU workload balacing

A device contains a CPU and a GPU. Both can be used for math operations. You can use `gpgpu_workload_balancing_enabled` configuration entry to allows using both units. On some devices the CPU is faster and on other it's slower. When the application starts, the work (math operations to perform) is equally divided: 50% for the CPU and 50% for the GPU. Our code contains a profiler to determine which unit is faster and how fast (percentage) it is. The profiler will change how the work is divided based on the time each unit takes to complete. This is why this configuration entry is named "workload balancing".

On x86 there is a known issue and we only recommend enabling this option on ARM devices. **On ARM device this could speedup the detection by up to 100%.**

10.3 Segmenter

As explained in the configuration section, the segmenter accurracy accepts 5 values (JSON strings): `veryhigh`, `high`, `medim`, `low`, and `verylow`. The default value is `high`.

If you're using a low-end mobile device then, consider using `medium` value.

10.4 Interpolation

As explained in the configuration section the interpolation operation accepts 3 values (JSON strings): `bicubic`, `bilinear`, and `nearest`. The default value is `bilinear`.

The interpolation operations are used when pixels are scaled, deskewed or deslanted. `bicubic` offers the best quality but is slow as there is no SIMD or GPU acceleration yet. `bilinear` and `nearest` interpolations are multithreaded and SIMD accelerated. For most scenarios `bilinear` interpolation is good enough to provide high accuracy/precision results while the code still runs very fast.

Make sure the interpolation value is `bilinear` (recommended) or `nearest` (not recommended).

10.5 Region of interest

Unlike the other applications you can find on the market we don't define a region of interest (ROI), the entire frame is processed to look for MRZ lines. The default resolution used in our sample application is HD (720p). If you're using a low end mobile device then, consider setting a region of interest instead of downscaling the resolution.

10.6 Device orientation

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 could be time consuming on some mobile devices. We recommend using the device on landscape mode to avoid the pre-processing operation.

10.7 Memory alignment

Make sure to provide memory aligned data to the SDK. On ARM the preferred alignment is 16-byte (NEON) while on x86 it's 32-byte (AVX). 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.

10.8 Planar formats

Both the detector and recognizer expect a grayscale image as input but most likely your camera doesn't support such format. Your camera will probably output YUV frames. Converting YUV frames to grayscale is a nop (very fast), we just need to map the Y plane.

10.9 Reducing camera frame rate

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. What is very important is the frame resolution. Higher the resolution is better the detection and recognition qualities will be. Try to use very high (2K if possible) resolution but low frame rate.

11 Benchmark

It's easy to assert that our implementation is fast without backing our claim with numbers and source code freely available to everyone to check.

Rules:

- We're running the processing function within a loop for #100 times.
- The **positive rate** defines the percentage of images with #1 MICR line. For example, 20% positives means we will have #80 **negative** images (no MICR lines) and #20 positives (with MICR lines) out of the #100 total images. This percentage is important as it allows timing both the detector and recognizer.
- We're using `high` accuracy for the segmenter and `bilinear` interpolation.
- All positive images contain #1 MICR line.
- Only `E-13B` format is enabled.
- The initialization is done outside the loop.

	0% positives	20% positives	50% positives	70% positives	100% positives
Xeon® E31230v5, GTX 1070 (Ubuntu 18)	1710 millis 58.45 fps	2101 millis 47.57 fps	2735 millis 36.55 fps	3153 millis 31.71 fps	3693 millis 27.07 fps
i7-4790K (Windows 7)	1543 millis 64.80 fps	3642 millis 27.45 fps	6919 millis 14.45 fps	9239 millis 10.82 fps	12392 millis 8.06 fps
i7-4770HQ (Windows 10)	1807 millis 55.31 fps	5709 millis 17.51 fps	11720 millis 8.53 fps	15613 millis 6.40 fps	22136 millis 4.51 fps
Galaxy S10+ (Android 10)	7244 millis **13.80 fps**	18408 millis 5.43 fps	37880 millis 2.63 fps	52165 millis 1.91 fps	62776 millis 1.59 fps
Raspberry Pi 4 (Raspbian OS)	7477 millis **13.37 fps**	31837 millis 3.14 fps	70229 millis 1.42 fps	96170 millis 1.03 fps	133263 millis 0.75 fps

More information on how to build and use the application could be found at https://github.com/DoubangoTelecom/ultimateMICR-SDK/blob/master/samples/c++/benchmark/README.md. For Android the Benchmark application is in samples/android/benchmark folder.

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.

12 Best JSON config

Here is the best config we recommend:

- **YUV420P image format as input**: This format is easy to convert to grayscale (format expected by the recognizer and detector). You should prefer YUV420P instead of YUV420SP (NV12 or NV21) as the later is semi-planar which 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 for the detection and recognition parts. 720p is a good trade-off between quality and resource consumption. Higher image sizes will give your camera a hard time which means more CPU and memory usage.
- **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 MICR line have an accuracy at least equal to 0.3. For example, having a false-positive with #16 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",
	"score_type": "min",
	"min_score": 0.3
}
```

13 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 "`ultimateMICR-input.png`" and to pull it from the device to your desktop use adb tool like this: `adb pull <path to dump folder>/ultimateMICR-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

14 Frequently Asked Questions (FAQ)

14.1 Why the Benchmark application is faster than VideoRecognizer?

The VideoRecognizer application have 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 resources .

15 Known issues

There is no known issue.

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