Doubango AI

State-of-the-art Bank credit card layout analysis, fields extraction and recognition (Number, Holder's name, Validity, Company...) using deep learning
https://github.com/DoubangoTelecom/ultimateCreditCard-SDK

Table of Contents

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

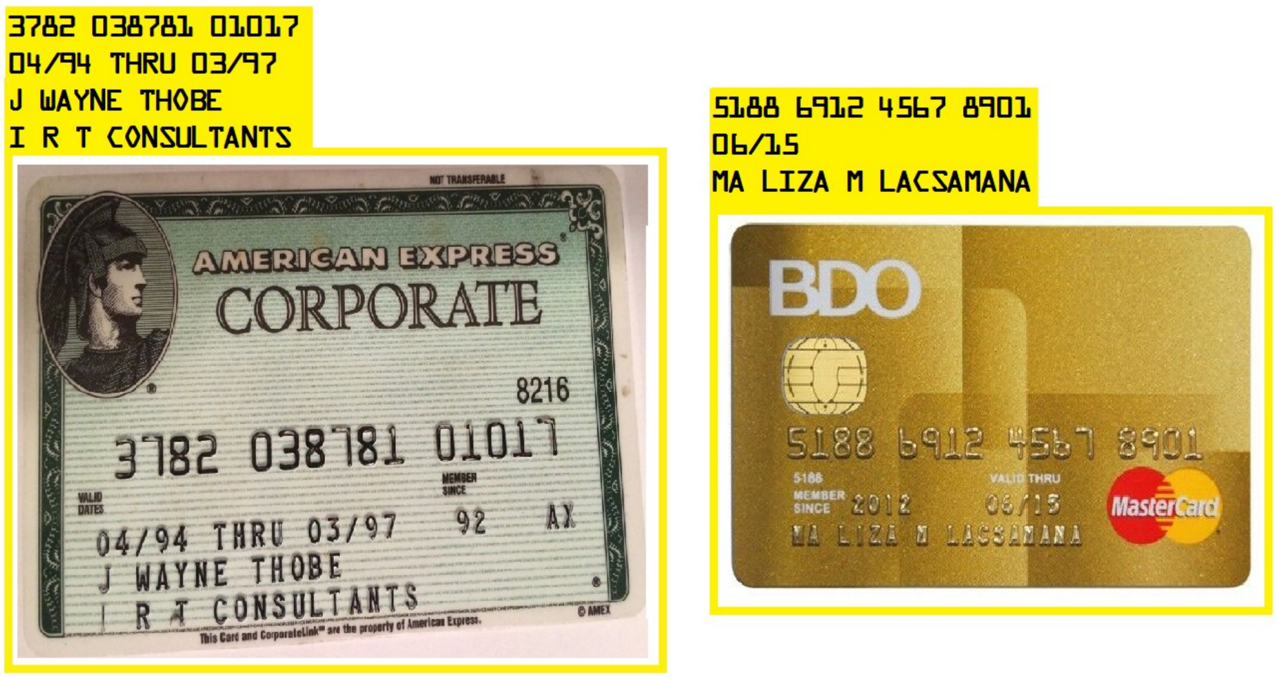
1 Intro

This is a state-of-the-art **DeepLayout Analysis** implementation based on Tensorflow to accurately detect, qualify, extract and recognize/OCR every field from a bank credit card using a single image : **Number, Holder's name, Validity, Company...**

Our implementation works with all cards (**credit, debit, travel, prepaid, corporate...**) from all payment networks (**Visa, MasterCard, American Express, RuPay, Discover...**).

Both **Embossed** and **UnEmbossed** formats are supported.

Unlike other implementations we're not doing brute force OCR (trying multiple images/parameters until match). You only need a single image to get the correct result. There is no template matching which means the data could be malformed or at any position and you'll still have the correct result (WYSIWYG).



Sample results using the SDK on Android

You can reach **100% precision** on the credit card number recognition using data validation process.

The number of use cases in FinTech industry are countless: **Scan To Pay, Helping visually impaired users, Online shopping speed-up, payment forms auto-filling, better user experience, reducing typing errors, process automation...**

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/ultimateCreditCard-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/credit-card-ocr to check out the accuracy and precision before starting to play with the SDK.

2 Supported formats

Our implementation works with all cards (**credit, debit, travel, prepaid, corporate...**) from all payment networks (**Visa, MasterCard, American Express, RuPay, Discover...**).

It's very hard to find credit card images to train a deep learning model. We tried to find as many as possible from Google, Bing, Instagram, Pinterest... Some formats are more represented than others wich means you may have poor accuracy on recent cards. Please let us know if you have low accuracy scores with some formats.

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. Scanners).

The Github repository (https://github.com/DoubangoTelecom/ultimateCreditCard-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**,

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 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: `.`
`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, width, top, height]` 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`
		
		`opencl`
		
		`nnapi`
		
		`metal`
			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 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

		`none`	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`

`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 credit card 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 has 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`	`BOOLEAN`	`true`	Whether to add rectification layer between the

`_enabled`		`false`	detector's output and the recognizer's input. A 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 credit cards 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_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`

5 Sample applications

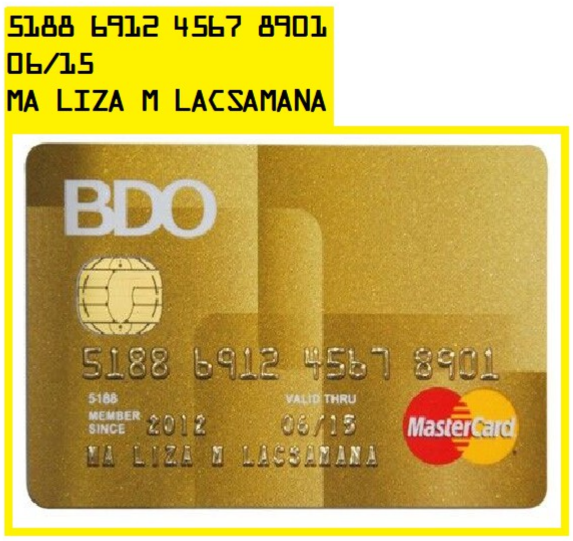
The source code comes with #2 sample applications: **Benchmark** and **CCardVideoRecognizer**.

5.1 Benchmark

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

5.2 CCardVideoRecognizer

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



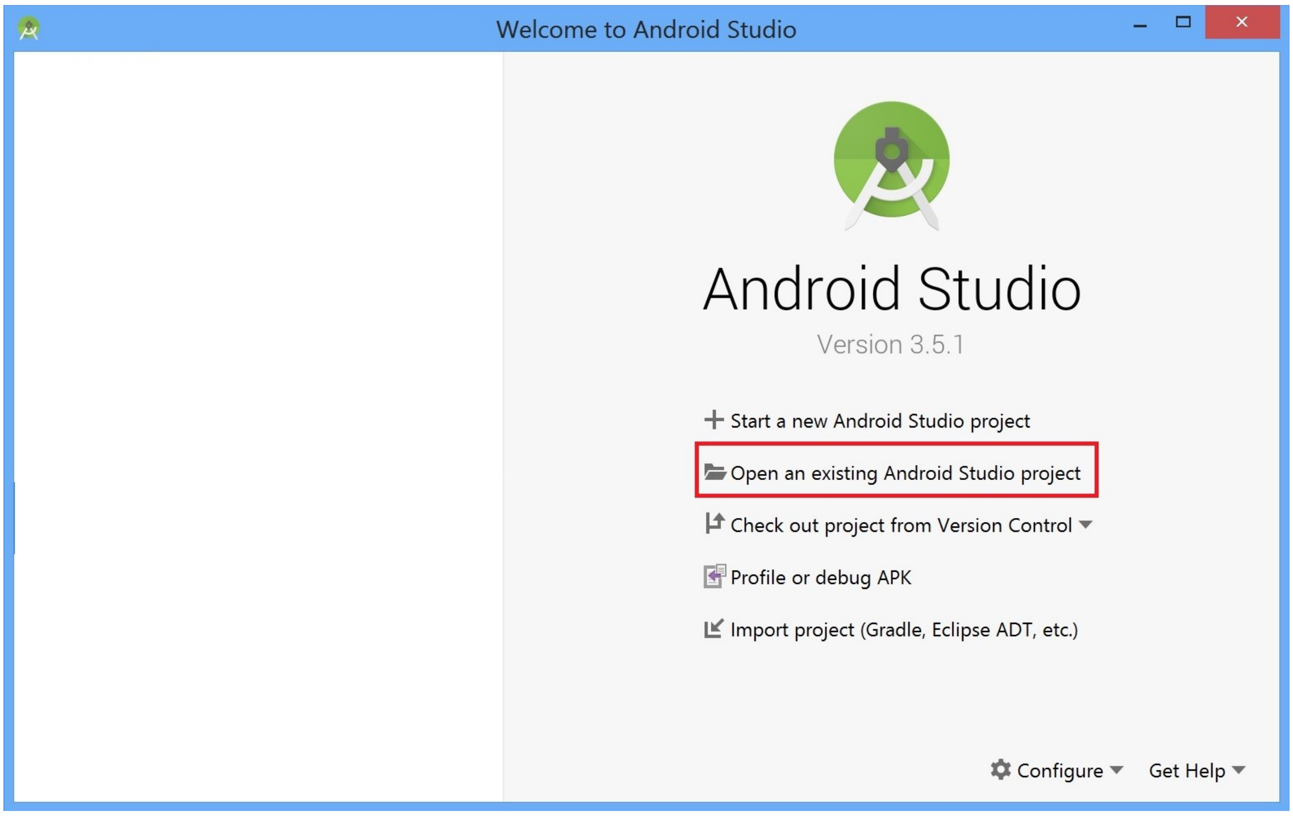
```
5188 6912 4567 8901
06/15
MA LIZA M LACSAMANA
```

ultimateCreditCard running on ARM device

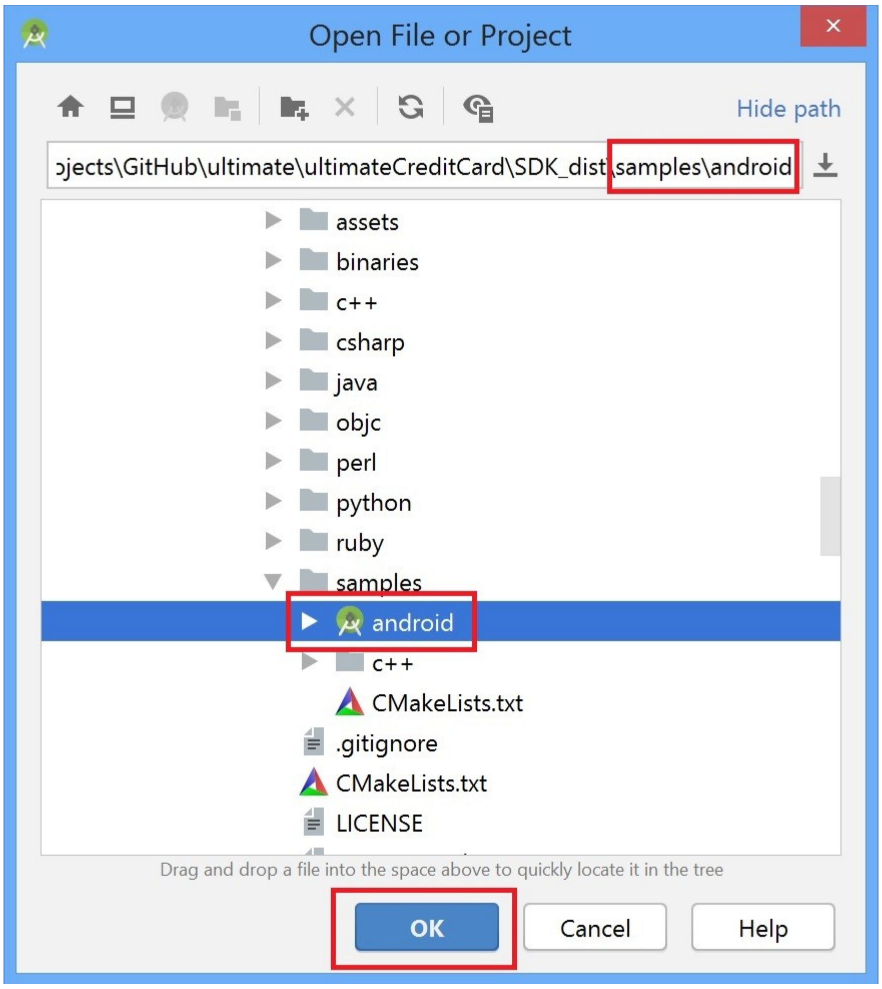
5.3 Trying the samples

To try the sample applications on Android:

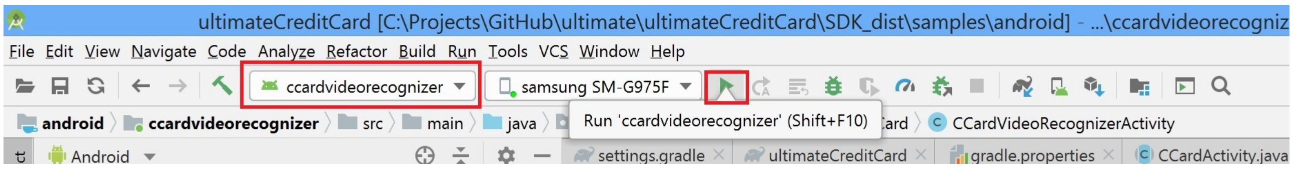
1. Open Android Studio and select "Open an existing Android Studio project"


Welcome to Android Studio
Android Studio
Version 3.5.1
Start a new Android Studio project
Open an existing Android Studio project
Check out project from Version Control
Profile or debug APK
Import project (Gradle, Eclipse ADT, etc.)
Configure
Get Help

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



3. Select the sample you want to try (e.g. "ccardvideorecognizer") and press "run".


ultimateCreditCard [C:\Projects\GitHub\ultimate\ultimateCreditCard\SDK_dist\samples\android] - ...\ccardvideorecogniz
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
ccardvideorecognizer
samsung SM-G975F
android ccardvideorecognizer src main java
Run 'ccardvideorecognizer' (Shift+F10)
CCardVideoRecognizerActivity
Android
settings.gradle
ultimateCreditCard
gradle.properties
CCardActivity.java

6 Getting started

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

6.1 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. 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-ultimateCreditCard-SDK/binaries/android/jniLibs']
         java.srcDirs += ['path-to-your-ultimateCreditCard-SDK/java/android']
         assets.srcDirs += ['path-to-your-ultimateCreditCard-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).

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.

7 Data validation

For every JSON result returned by the SDK you will have a boolean field (`"valid"`) defining whether the credit card number is correct based on the check digit. The validation process is done using Luhn algorithm. You can safely stop the processing when this boolean field is equal to true.

Use case to reach 100% precision: process all frames from the camera until the validation field is equal to true.

8 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.

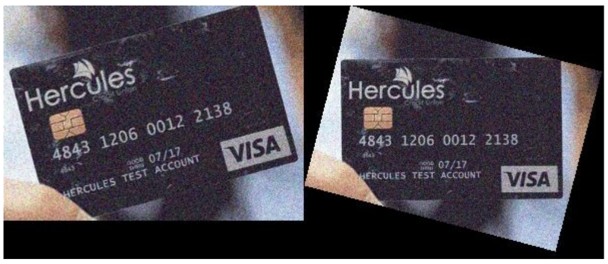
When a user tries to scan a credit card using his/her phone then, both the card and the camera are moving. The credit card is an undeformable object. In such situation, every position the card can take could be estimated using a 3x3 matrice. **This is the homography matrice.**

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

8.1 Polarity

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

DarkOnBright: The text on the credit card is darker than the background. Example: Black numbers on white background (e.g. American Express cards).

BrightOnDark: The text on the credit card is brighter than the background. Example: White numbers on blue background (e.g. UnEmbossed new Visa cards).

Unlike other implementations we don't use the four corners from the credit card 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.

9 Muti-threading design

No forking, minimal context switch. Doubango vs Others

10 Memory management design

This section is about the memory management design.

10.1 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.

10.2 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.

10.3 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
```

10.4 Cache blocking

To be filled

11 Improving the accuracy

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

11.1 Detector

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

11.1.1 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 valid credit cards. 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.

11.1.2 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.

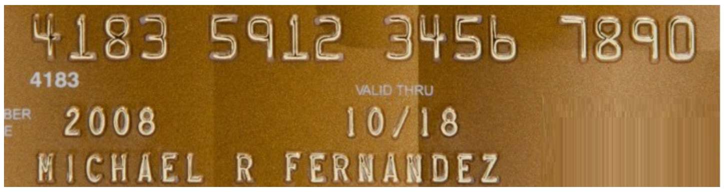
If you have too many false-negatives, then decrease the detection score in order to increase the recall.

11.1.3 Matching training data

The training data for the detection predominantly contains credit cards with clear borders. To increase the detection accuracy you should provide images showing the entire credit card: borders, chip and payment network (Visa, MasterCard, AMEX...) logo included.

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

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



The fact that the training data predominantly contains images showing the entire credit card while there are few images with partial cards is done on purpose to decrease the number of false-positives.

11.2 Recognizer

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

11.2.1 Data validation

For every JSON result returned by the SDK you will have a boolean field (`"valid"`) defining whether the credit card number is correct based on the check digit. The validation process is done using Luhn algorithm. You can safely stop the processing when this boolean field is equal to true.

11.2.2 Adding rectification layer

When the credit cards 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.

11.2.3 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.

11.2.4 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 credit card field has 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 credit card field has 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.

11.2.5 Score threshold

The SDK returns a score/confidence value for each field (Number, Holder's name, Validity, Company...). These scores are in percentage and within [0.0, 100.0]. When the score is **>=80%** then, you're sure that every character in the field is correct. A score **>=70%** means almost every character in the field is correct.

We recommend using a threshold value equal to 70% and using the validation process to make sure the credit card number is correct.

12 Improving the speed

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

12.1 Memory alignment

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

12.2 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.

12.3 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.

12.4 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.

12.5 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.

13 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 a valid credit card. For example, 20% positives means we will have #80 **negative** images (no credit card) and #20 positives (with credit cards) out of the #100 total images. This percentage is important as it allows timing both the detector and recognizer.
- All positive images contain a single credit card.
- The rectification layer is disabled.
- The initialization is done outside the loop.

The concept of **negative** and **positive** images is very important because in most use cases you'll:

1. Start the application.
2. Move the application to the credit card to recognize the data.

You only need a single "good frame" to recognize the credit card. But, between step #1 and step #2 the application has probably processed more than #200 frames (40fps * 5sec). So, in such scenario the application have to process #201 frames before reaching the "good frame": #200 negatives and #1 positive. If processing negative frame is very slow then, the application won't be able to catch this "good frame" at the right time. A slow application will do one of these strategies:

1. **Drop frames to keep the impression that the application is running at realtime:** In such scenario the positive frames will most likely be dropped (probability = 1/201 = **0.49%**) which means reporting the time when this single "good frame" is caught.
2. **Enqueue the frames and process them at the application speed:** This is the worse solution because you could run out of memory and when the application is running slowly then, you can spend several minutes before reaching this single "good frame".

A fast application will run at 40fps and catch this "good frame" as soon as it's presented for processing. This offers a nice user experience.

	0% positives	**20% positives**	**50% positives**	**70% positives**	**100% positives**
Galaxy S10+ (Android 10)	3673 millis **27.22 fps**	13053 ms 7.66 fps	28304 ms 3.53 fps	38175 ms 2.61 fps	52685 ms 1.89 fps
Raspberry Pi 4 (Raspbian OS)	9731 millis **10.21 fps**	35449 ms 2.82 fps	74698 ms 1.33 fps	101665 ms 0.98 fps	128269 ms 0.77 fps

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

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.

14 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.
- 50% for minimum detection score: This looks low but it'll improve your recall. Off course it'll 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 10% as it'll increase the number of false-positives which means more images to recognize which means more CPU usage. JSON config: *"detect_minscore": 0.5*
- 20% for minimum recognition score: This score is very low and make sense if the score type is "min". This means every character on the credit card have an accuracy at least equal to 0.2. For example, having a false-positive with #5 chars and each one is recognized with a score > 0.2 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_minscore": 0.5,
        "recogn_score_type": "min",
        "recogn_minscore": 0.2
}
```

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

16 Frequently Asked Questions (FAQ)

16.1 Why the benchmark application is faster than CCardVideoRecognizer?

The CCardVideoRecognizer 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.

17 Known issues

There is no known issue.

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