Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Árpád Fodor

REAL-TIME STOLEN VEHICLE DETECTION ON ANDROID WITH DEEP LEARNING

Students' Scientific Conference Report

SUPERVISOR

Dániel Pásztor

BUDAPEST, 2020

Table of contents

Abstract

Machine and deep learning allow computers to solve complex tasks; object detection, speech synthesis, or time series prediction are good examples. Technological advancement allows these models to be available on multiple devices. Meanwhile, smartphones have become part of our everyday life. Combining machine learning and mobile devices can open new opportunities that help us in our everyday lives.

Although the range of possibilities is broad, the on-device inference is still rarely used. It can raise problems, such as lower availability, Internet dependency, increased network traffic, or personal data release. This report aimed to demonstrate how to create a machine learning-based system capable of providing solutions to these problems - running in real-time, independently, on-device.

I chose stolen vehicle detection as a domain area because it involves numerous tasks (vehicle and license plate detection, optical character recognition) to solve. Using an ordinary smartphone, even as a dashcam, a driver can continuously monitor the traffic and report alerts automatically while driving. Although similar pre-installed camera systems already exist, they typically run on stationary devices. The chosen task is not just one of the first such applications in the smartphone market; it can be easily generalized to other domains.

The steps necessary for composing the model, such as dataset creation, model building, training, and tweaking, are described in detail. Besides, I explain the Android client and the server app broadly to give a complete picture of what is needed to bring such a system to life.

Kivonat

A gépi tanulás segítségével komplex feladatok számítógépes megoldására nyílik lehetőségünk; objektumdetektálás, beszédszintézis, vagy idősor előrejelzés is lehetséges. A technológia fejlődése lehetővé teszi, hogy ennek a tudományágnak a vívmányai egyre több eszközön megjelenjenek. Ezzel párhuzamosan a mobiltelefonok mindennapi életünk részévé váltak. A gépi tanulás és a mobil eszközök ötvözése új lehetőségeket nyithat, melyek kisegítenek minket mindennapjainkban.

Habár a lehetőségek tárháza széles, az eszközön történő futtatás még igen ritka. Ez olyan problémákat okozhat, mint az alacsony rendelkezésre állás, az internettől való függés, megnövekedő hálózati forgalom, vagy a személyes adatok védelmének kérdése. Ennek a dolgozatnak a célja, hogy demonstrálja, hogyan készülhet egy olyan rendszer a gépi tanulás segítségével, ami ezen problémákat kiküszöböli – valós időben, függetlenül futva hétköznapi eszközökön.

Azért a lopott járművek felismerését választottam, mert ez számos megoldandó részfeladatot foglal magába (jármű- és rendszámtábla detektálás, karakterfelismerés). Egy átlagos okostelefont fedélzeti kameraként használva egy autós folyamatosan nézheti a forgalmat és automatikusan bejelentést tehet, akár vezetés közben is. Habár hasonló telepített kamerarendszerek már léteznek, ezek jellemzően helyhez kötött eszközökön futnak. A választott feladat nem csak az egyik első ilyen alkalmazás az okostelefonok piacán, könnyen általánosítható és alkalmazható más problémakörökben is.

Az algoritmus készítéséhez szükséges lépéseket, úgymint az adatok előkészítését, a modell kialakítását, tanítását és finomhangolását részletekbe menően bemutatom. Továbbá az Android kliens és a szerveroldali alkalmazás is röviden ismertetésre kerül, hogy képet kapjon az Olvasó, mi minden szükséges egy ilyen rendszer életre keltéséhez.

1. Introduction

Nowadays, smart devices are an integral part of our everyday lives. The number of gadgets around us is growing, as are the demands towards them. Thanks to continuous advances in technology, these demands can be better served. Mobile phones have become such a part of our daily lives that more than 3.5 billion users now have such a device.

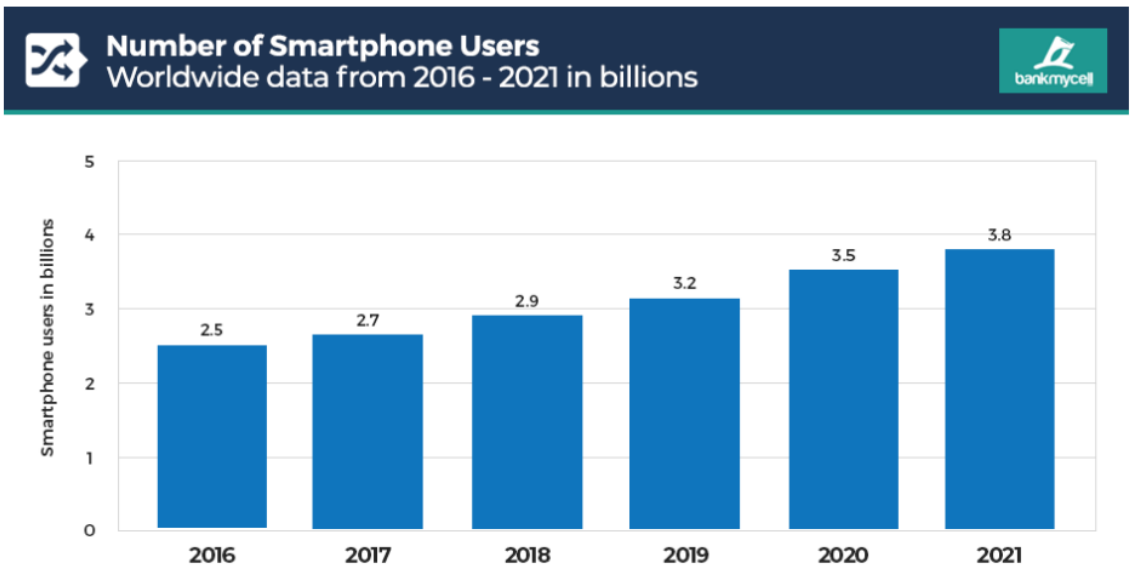


Figure 1: Growth of smartphone users worldwide. Source: [1]

Meanwhile, with machine learning, we can solve increasingly complex tasks by computers; object detection and emotion recognition, speech synthesis, time series prediction, even automated planning are possible. These are complex tasks that would be very time-consuming to program "by hand".

Technological advances help to deploy ML-based algorithms on more and more devices. By combining the two, we can provide solutions to traditionally hard-to-implement tasks accessible from anyone's pocket.

I aim to create an end-to-end stolen vehicle detection system. I chose this domain area because it involves numerous tasks (vehicle and license plate detection, optical character recognition) to solve. Using an ordinary smartphone, even as a dashcam, a driver can continuously monitor the traffic and report alerts automatically while driving. Although similar pre-installed camera systems already exist, they typically run on stationary devices. The chosen task is not just one of the first such applications in the smartphone market; it can be easily generalized to other domains.

The structure of the report is as follows:

- In section 2, the system- and module level specification is described.
- Section 3 contains the presentation of the technologies used during this work.
- Section 4 is about stolen vehicle detection; data preparation and the model construction steps are presented. Besides, theoretical insights are also provided.
- Section 5 contains the Android and server applications' introduction with the most critical design decisions and implementation details.
- Section 6 provides a summary of the work done, where further development possibilities are also covered, as well as the knowledge and experience gained during this work.

2. Specification

In this section, the specification of the whole system and each subpart of it are described.

2.1 System requirements

The main goals are to detect, report, and track stolen vehicles anywhere by portable devices. Thus, it is possible to monitor arbitrary places without the need for pre-installed stationary devices.

Object detection is challenging to algorithmize, so it is worth turning to the toolkit of machine learning. Since detection should run in real-time, the detector needs to run on-device. Although it would be possible to run the model on a server and communicate with it, this would have the following disadvantages:

- Sending images over the network is resource-intensive and would increase data traffic.
- The server may be unavailable or respond slowly due to its load.
- An Internet connection with the necessary bandwidth may not always be available to the user.

Therefore, I decided to choose the solution of on-device inference. This puts the extra computational effort on the client system in exchange for eliminating the problems listed above.

2.2 Architecture

The system is planned to be based on a client-server architecture. The clients are smartphone applications that can detect vehicles independently. The server stores report information (coordinates, messages, timestamps) and provides a REST API for clients to report or query.

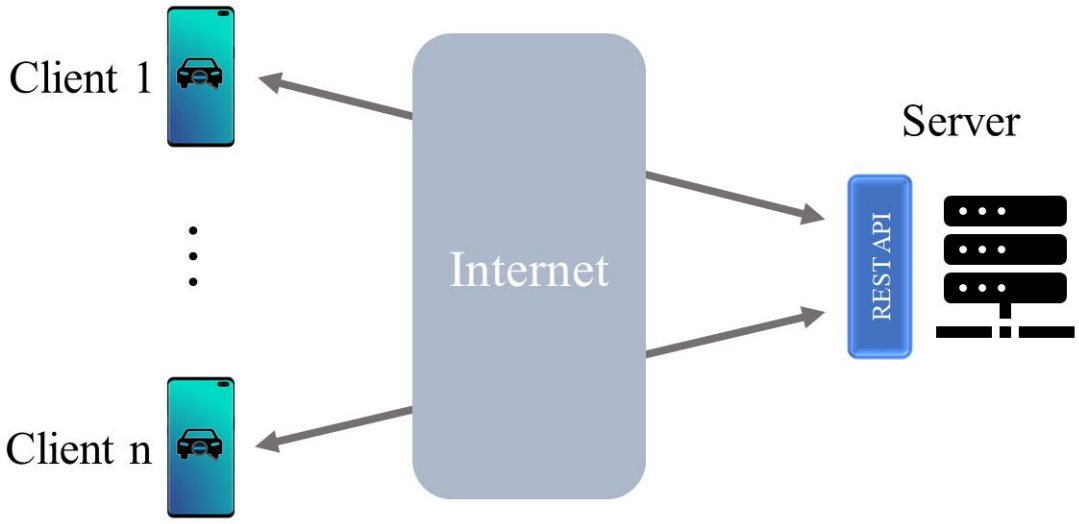


Figure 2: High-level system architecture

2.3 Front end

The client application's two main functionalities are sending reports and viewing them.

To report a suspicious vehicle, the user needs to automatically receive an alert generated when a stolen vehicle appears on an image. The image source can be the camera's live feed or a loaded picture stored on the device. When an alert appears, the user can individually check them (as there can be multiple findings) and add to the recognitions queue (which persists its content and sends them to the server) or delete. To create a valid recognition, two more things are needed: timestamp and coordinates. In the case of live detection, the timestamp is the UTC system time, and the coordinates come from the GPS position of the device. When a loaded image produces an alert, timestamp and coordinates come from the image EXIF metadata. When inspecting a pending alert or a non-delivered recognition, the user can append additional text to his/her finding or discard it but cannot modify its picture/timestamp/coordinates to minimize the potential for abuse.

An interactive map allows users to see the status of reports. It displays valid recognitions on the map, and each recognition details can be seen by clicking on it.

In addition to the main functionalities, Figure 3 represents the general use-cases.

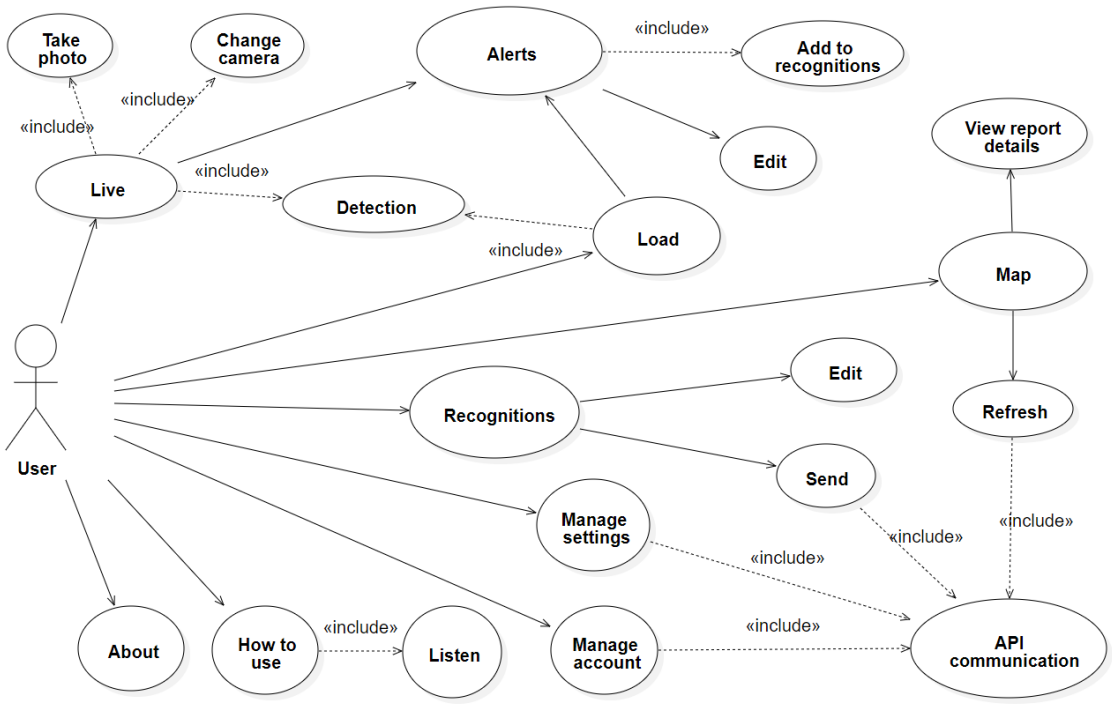


Figure 3: Application use case

2.4 Back end

The server application provides the API for the clients and manages users. The API can retrieve the list of stolen vehicles, list of recent recognitions with location, and is also able to receive new reports.

The data source of stolen vehicles is the Hungarian Police website, from where the data is extracted via web scraping.

Because some of the data stored on the server include user accounts, and some contain license plate and location data for suspicious vehicles, sophisticated authorization management and authentication are required to prevent sensitive data leakage.

The API can only be used by authorized users whose permissions can be changed by administrators anytime. It is crucial that the data stored on the server can be restored in case of an error and that changes to the data and unexpected events can be reviewed afterward.

3. Technologies

This section presents the technologies used to build the detector pipeline and create Android and server applications.

3.1 Deep learning

I mainly used the Python programming language[1] in deep learning-related tasks. Python is an interpreted, dynamically typed, high-level language with an object-oriented approach.

Numerous libraries offer a deep learning repertoire - some of them have built-in object detection capabilities, like FAIR's Detectron, TorchVision, or TensorFlow's Object Detection API. In the following, I only describe the technologies I used during this work. The choice was driven primarily by the desire for Android interoperability, which is currently not widely supported by other tools than the selected ones.

3.1.1 TensorFlow

TensorFlow is an open-source machine learning platform developed by Google Brains. It provides rich Python and C APIs and works well with the popular Keras neural network library. TF also works well with the Colaboratory environment where GPU/TPU-based works are easy to build without local resources.

TensorFlow uses Google's protobuf[8] format to store models. In this case, a .proto file defines a scheme, and Protocol Buffers generates the content. It is a denser format than XML or JSON, and it supports fast serialization, prevents scheme-violations, and guarantees type-safety[10]. In turn, protobuf files are not as human-readable as opposed to JSON or XML.

In September 2019, TensorFlow 2.0 has been released with Eager mode, which broke up with the former "define-and-run" [6] scheme (where a network is statically defined and fixed, and then the user periodically feeds it with batches of training data). Eager uses a "define-by-run" approach[7], where operations are immediately evaluated without building graphs.

3.1.2 TF Lite

TFLite is a lightweight, speed, or storage optimized format aimed at deploying models on smartphones and IoT devices. Trained TensorFlow models can be transformed into this format with the TFLite converter (standalone TFLite files cannot be trained).

TensorFlow Lite uses the Flat Buffer[9] format. It is similar to TensorFlow's protobuf; the main difference is that Flat Buffers do not require deserializing the entire content (coupled with per-object memory allocation) before accessing an item in it. Therefore, these files consume significantly less memory than protobufs[10]. On the other hand, Flat Buffer encoding is more complicated than in JSON/protobuf formats – therefore, TensorFlow does not use it. It is also the reason why TFLite models cannot be trained.

During TFLite conversion, it can be selected whether it is required to minimize model size further (above protobuf -> Flat Buffer conversion) with a slight model accuracy trade-off or not. These are the quantization options used to achieve further performance gains (2-3x faster inference, 2-4x smaller networks). I used full integer quantization[11] in which all the model maths are int8 based calculations instead of the original float32.

3.1.3 TensorFlow Object Detection API

Object Detection API[12] is an open-source framework built on top of TensorFlow to solve complex computer vision tasks, like object detection or semantic segmentation.

The library provides a Model Zoo in which pre-trained models are available. The API supports one-staged meta architectures like *SSD*s and *CenterNet*s as well as two-staged *R-CNN* variants. However, other types like Facebook AI's *YOLO* architecture is missing (which is similar to Google's *SSD* – probably they prefer their in-house solution).

As there are many parameters to tweak during model creation, the API has introduced a configuration language in which it is possible to fine-tune the training pipeline – e.g., data source, pre-processing steps, optimization algorithms, input/output directories. Possible configuration parameters are described in the corresponding proto files. Configuration handles custom components (like a new backbone *CNN* to use) as well.

During this work, support for TF2 has arrived, making it possible to use Keras models in detector architectures. While migrating the project to the new API version, it was possible to compare the two types of TensorFlow as well. The library has actively evolved in the past months, and although reliability issues often arise, rapid implementations of the latest research (e.g., *FPN*, *CenterNet*) helped me understanding novel concepts.

3.1.4 TensorBoard

The TensorBoard component provides a useful visualization tool where users see a dashboard of model performance and training/evaluation details. It can also display the current images fed to the network, the model's answer to it, and many more. I used this tool to monitor the training/evaluation processes.

3.2 Application development

I primarily used the Kotlin programming language[3] in both the Android client and the backend. Kotlin is a relatively new, statically typed, cross-platform language with type inference. In addition to the object-oriented approach, it also contains functional programming tools.

3.2.1 Android

Android provides an extensive application development ecosystem. I mainly used the AndroidX namespace elements, which replaces the previous Support Library since Android 9.0. It is part of Android Jetpack, a collection of components for which the platform promises long-term support. Of these libraries, the application extensively uses the CameraX API to manage device cameras. The ViewModel component acts as a moderator between the user interface and business logic in the architecture. The app contains a relational database implemented by the Room Persistence Library, which provides an abstraction layer over SQLite to allow for more robust database access.

I used a pre-trained ML Kit *OCR* model running on-device to read license plate texts on image snippets. To boost user experience, I decided to use a text-to-speech engine provided by the operating system to read aloud alerts (useful if a user wants notifications while driving or wants to hear how to use the application).

Material Design defines guidelines for building the interface to maximize user experience. On Android, Material elements supported by the platform can be accessed through a library. The application uses its concepts, styles, icons, fonts, and UI elements (e.g., Floating Action Button, Snackbar).

3.2.2 Server

I used Ktor[4] for the back end. It is an open-source, asynchronous framework for creating microservices and web applications. JSON data handling was implemented with the help of Gson. Testing the server API was mainly conducted by Postman, and for web scraping, I used ParseHub.

3.3 Environment

The environment in which I concluded training was Google Colaboratory Pro. It is a cloud-based Jupyter notebook service with resources on demand. I was using an Nvidia Tesla V100 SXM2[5] GPU with 16 GB memory. I synchronized it with a Google Drive account from where the prepared dataset is available and where model checkpoints and output files are stored. The Android application was developed with Android Studio.

4. Stolen vehicle detection

In this section, I discuss the work done related to object detection. In addition to the model development phases, I also describe the related research. Since deep learning requires a greater theoretical background, I assume knowledge of its general principles (backpropagation, convolutional neural networks) for reasons of length. I describe the theoretical background related to object detection in detail.

Before starting the task, it is worth clarifying what the exact requirement is. The main goal is to detect stolen vehicles on input images. This problem can be broken down into two subtasks: object detection and optical character recognition. Related subtasks are also required, such as resizing images. Since this is easy to algorithmize, this will not be the task of the model. In general, a machine learning development process is time-consuming, so it should only be used if it is firmly justified.

Object detection is a difficult task to algorithmize, so it is worth turning to the toolkit of ML. Although there are popular machine learning solutions, like *HOG-Linear SVM* based detectors[13], deep learning-based object detection has been a research hotspot in recent years due to its powerful learning ability[14]. Therefore, I chose this option to create a model with.

In this part, I describe the process of making the detector in detail. Since the application uses a pre-trained Optical Character Recognition model, I will not go into *OCR*s' details.

4.1 Data preparation

The following section describes collecting, preparing, analyzing, transforming, building, and validating the database needed to train the detector.

4.1.1 Requirements

An essential aspect of obtaining the data was to cover as many images as possible with labeled license plates. The assumption was that thousands of images were needed to ensure proper data diversity. The aim was to get not only license plate annotations but vehicle annotations as well. The reason behind this was that I wanted to leave open the

possibility to classify license recognitions based on whether they took place within a vehicle object or not.

4.1.2 Sources

There were not any standalone off the shelf datasets meeting these requirements. The research was conducted mainly on Kaggle and with the Google Dataset Search engine. The most promising sets had a few hundred labels, so I changed the approach.

I examined the *COCO*[15] (Common Objects in Context) and the *OID*[16] (Open Images) datasets. These are famously large sets containing 123,287 and roughly 2 million detection images. My assumption was that if at least one of them contains license plate annotations, the number of these labels would be enough. Of these, *OID* contained a vehicle registration plate class. As the dataset is huge, I used a Python toolkit to download all the 6,867 images containing these items. I also downloaded all the detection annotations in separate CSV files (*OID* train, validation, test). The total size of all files was 2.352 GB.

4.1.3 Pre-processing

The application is not limited to detect only stolen cars. It is necessary because the stolen vehicle database does not just contain cars, nor does the police data source. As a vehicle class was also needed, I decided to build it the following way.

All the annotations of the downloaded images were kept. *OID* has a hierarchical class structure from which I chose nine classes representing the new vehicle class (car, airplane, helicopter, boat, motorcycle, bus, taxi, truck, ambulance). The original *OID* vehicle class was discarded as it contained objects challenging to place in other subclasses like surfboards, wheelchairs. To keep the number of the resulting classes equal, a few less essential classes were also discarded (like aerial vehicle or snowmobile).

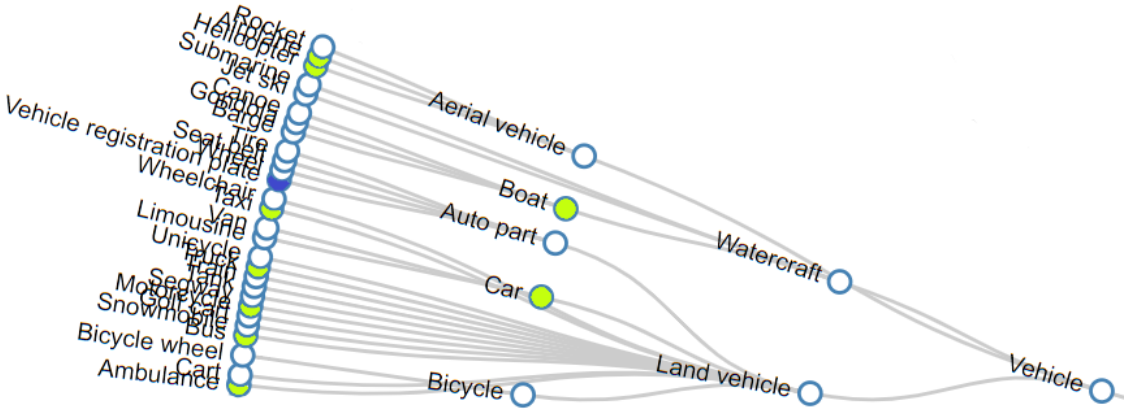
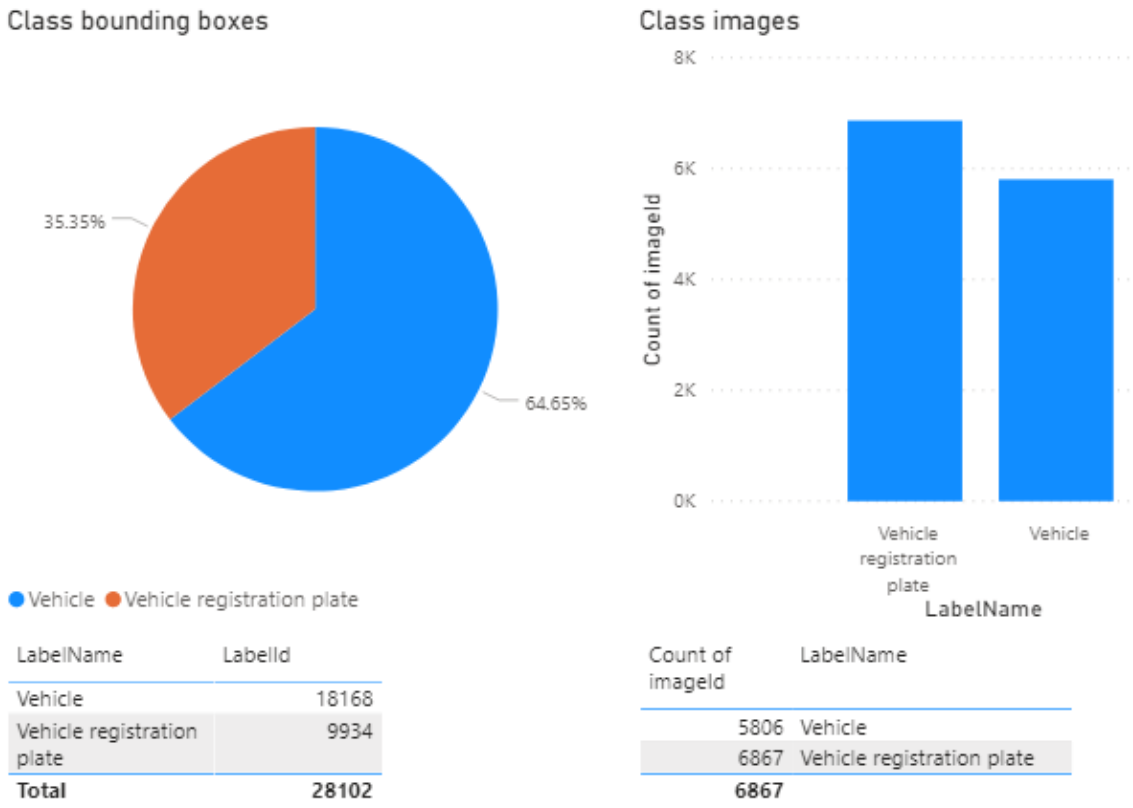


Figure 4: Selected classes from the original OID branch. License plate, vehicle

4.1.4 Analysis

To analyze the downloaded content, a Power BI report was created. It uses CSV files as data sources. The raw dataset contains 6867 images with 28,102 bounding boxes, which is roughly four annotations per image. There are 9934 vehicle registration plate annotations, which is one-third of all the boxes. The report revealed that the average image size is 1005x753 pixels. While most images have a size of 1024 by 768, some exemptions do occur. As the model input is constantly resized, this is not important to examine more thoroughly.

Figure 5 shows that unless more images in which license plates are presented, there are more vehicle boxes. The pie chart on the left shows class multiplicities compared to each other, while the column chart on the right is not intended to compare the values relative to each other but to the maximum number of images.



LabelName	LabelId
Vehicle	18168
Vehicle registration plate	9934
Total	**28102**

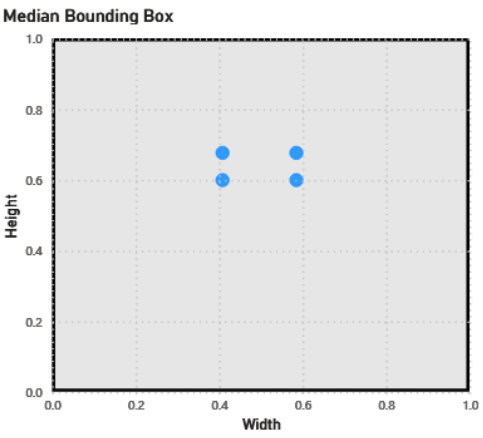
Count of imageId	LabelName
5806	Vehicle
6867	Vehicle registration plate
6867	

Figure 5: Multiplicity of class boxes and the number of images in which the class is presented

The average bounding box dimensions for license plates can be found in Figure 6. They occupy an average of 10% of both the X and Y axes, which means the detector should have wider anchor boxes of this size. It confirms that the data looks like it should.

Median Bounding Box coordinates

Median of XMax	Median of XMin	Median of YMax	Median of YMin
0.58	0.41	0.68	0.60



Avg Bounding Box coordinates

Average of XMax	Average of XMin	Average of YMax	Average of YMin
0.55	0.44	0.65	0.58

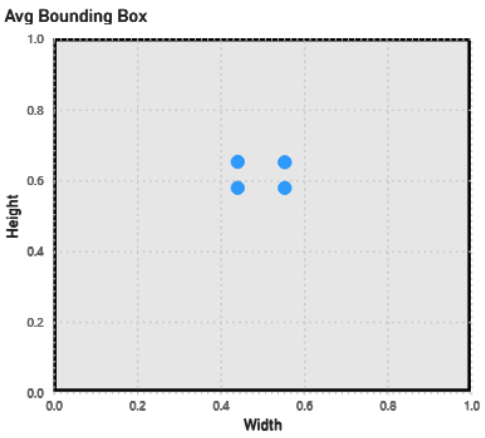


Figure 6: Mean and average bounding boxes of License plates

OID has a separate train, validation, and test set. In the data, it turned out they differ in bounding box position distributions, and there is also a slight difference in the average box numbers per image (4.09 vs. 3.33). As the average license plate is also bigger in the validation set, it may be misleading about the model's real performance. The issue was first indicated when I was comparing the average boxes. A closer look at the symptom confirmed that some outliers did not cause it - the reason was the difference in the distribution of the box sizes across the subsets.

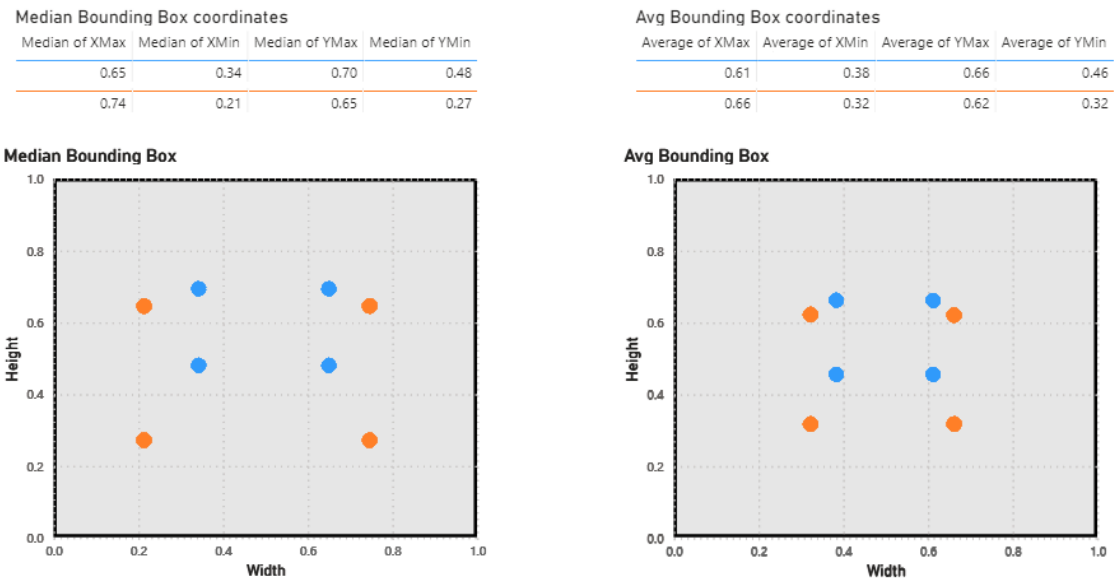


Figure 7: Mean and average bounding boxes of train and validation sets

4.1.5 Transformation

To fix this problem, data aggregation and partition have been applied. First, all the images and annotations have been aggregated then saved to tfrecords[17]. It is a binary format that stores images and their custom labels together. This format's main advantage is that it can be fed rapidly to a model that is not negligible if the dataset has thousands of instances. First, I selected the fields to be serialized, and then an encoder Python script was created. Detail contents of such a record are shown below:

```
features {
  feature {
    key: "image/encoded"
    value {
      bytes_list {
        value: binary encoded image
      }
    }
  }
  feature {
    key: "image/height"
    value {
      int64_list {
        value: 769
      }
    }
  }
…
  feature {
    key: "image/object/bbox/xmax"
    value {
      float_list {
        value: 0.800000011920929
        value: 0.3006249964237213
      }
    }
  }
  feature {
    key: "image/object/class/text"
    value {
      bytes_list {
        value: "Vehicle"
        value: "Vehicle registration plate"
      }
    }
  }
…
}
```

The dataset has been evenly sharded between 14 files using the Euclidean division ($n\%14$). While generating tfrecord batches, I also created the corresponding CSV files to analyze later. A validation and a test batch have been selected, and all the other files became part of the training set.

This way, the problem discussed above has been resolved. Figure 8 shows the average bounding boxes of test (12 batches), validation (1 batch), and train (1 batch) sets. They are almost identical.

Avg Bounding Box coordinates

Average of XMax	Average of XMin	Average of YMax	Average of YMin
0.61	0.38	0.66	0.45
0.61	0.39	0.66	0.47
0.60	0.38	0.68	0.47

Figure 8: Average bounding boxes after redistributing train, validation, and test sets

4.1.6 Evaluation

The final dataset has 5,793 (84%) training, 537 (7.8%) validation, and 537 (7.8%) test images. It has a 23,739 (84%) training, 2,200 (7,8%) validation, and 2163 (7,7%) test bounding box distribution. There are general dataset division guidelines (like the 70-20-10 recommendation), which I deviated from. In my opinion, the validation and test sets are already sufficiently representative after reshuffling them in the order of thousands; therefore, I tried to maximize the size of the train set.

I created a tfrecord viewer script to inspect whether it is possible to decode images and annotations. I found that the data files were restorable. Here are some samples from each subset decoded from batch files.

Figure 9: Training samples with bounding boxes (vehicle registration plate, vehicle)

Figure 10: Validation (1st row) and test (2nd row) samples with boxes (vehicle reg. plate, vehicle)

It is important to note that there are some inconsistencies in the dataset (Figure 11). It turned out that some images were incompletely annotated where:

- Clearly visible license plates are not annotated (top left).
- Vehicle annotations are entirely missing (top right).

One more thing to spot: as a few original *OID* classes were discarded to keep the new vehicle class close to the multiplicity of vehicle registration plates, some types of boxes (like vans) were dropped (bottom row). In part, this is the reason for some of the missing boxes.

Figure 11: Inconsistently missing labels (top row), no vehicle boxes around vans (bottom row)

These are good examples that while the dataset may be appropriate for the task, it has its flaws. Since balancing classes relative to each other is a priority not to mislead the model, I have not made any further changes. Having the dataset prepared, the next step is developing the algorithm.

4.2 Deep Learning algorithm

In the following, I present the preliminary steps of model creation (choosing appropriate metrics), architecture selection, training and fine-tuning, and then post-production (quantization, wrapping).

4.2.1 Evaluation metrics

Choosing the right metrics is crucial to evaluate a model in a manner appropriate to the task. If we solely concentrate on detector loss for object detection, we simply miss out on details like how well the model localizes (Where is the object?) or how well it classifies (Is it a vehicle?). They may suggest different things about the network - broadly speaking, classification depends primarily on the backbone, while detection is mainly the task of the last convolutional layers in an *SSD* architecture.

4.2.1.1 Concepts

Some concepts appear for most protocols, which I briefly describe below[22]. Abbreviations: T_p- true positive, T_n- true negative, F_p- false positive, F_n- false negative.

- $Precision = \frac{T_p}{T_p + F_p}$ How many of the predictions are actually true?
- $Recall = \frac{T_p}{T_p + F_n}$ How many of the ground truth items were hit?
- $IoU\ (Intersection\ over\ Union) = \frac{area\ of\ overlap}{area\ of\ union}$ It is used to measure how much the prediction overlaps with the ground truth. Sometimes, there is a predefined *IoU* threshold (often 0.5) under which a prediction is interpreted as incorrect (F_p), but above as correct (T_p).

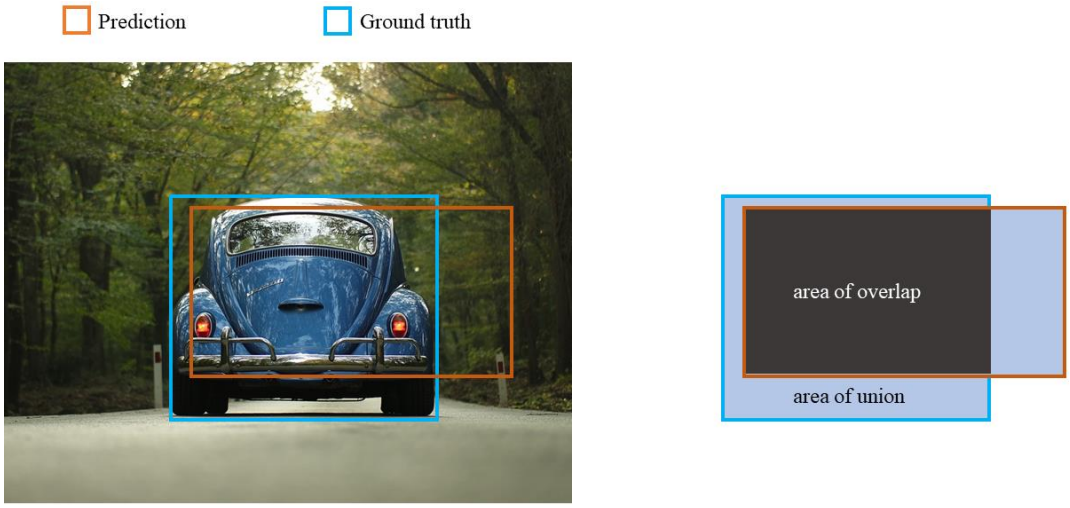


Figure 12: Illustration of *IoU*

- $AP\ (Average\ Precision) = \int_0^1 p(r)dr$ The area under the precision-recall curve. The next example demonstrates how to calculate it. Let the following be the result of some evaluations:

Number	Correct (is IoU ≥ 0.5)?	Precision	Recall
1	True	1.0	0.33
2	True	1.0	0.66
3	False	0.67	0.66
4	True	0.75	1.0
5	False	0.6	1.0

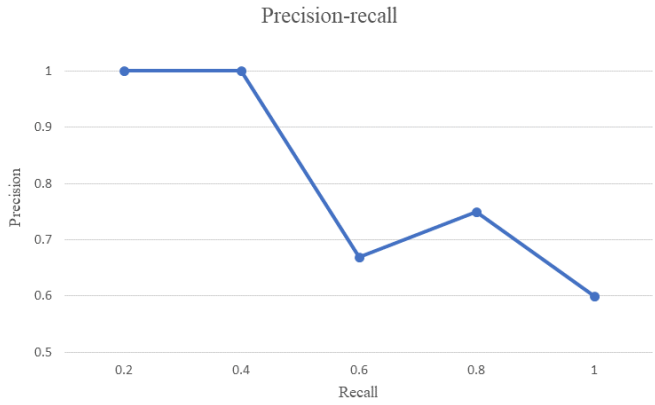


Figure 13: Sample detection results plotted in a tabular and graphical form

For example, calculations of line #4 in the table: Precision is the proportion of T_ps so far $(3/4 = 0.75)$, Recall is the proportion of T_ps out of the possible positives $(3/3 = 1.0)$. Recall continually increases as we go down the prediction ranking. However, precision can follow a zig-zag pattern (decreases with F_ps and increases with T_ps). Average precision is

the area under the precision-recall curve. As precision and recall values are always between 0 and 1, *AP* falls within this boundary too.

- $Interpolated\ Average\ Precision = \frac{1}{n}\sum_{n\in(0.0\ \ldots\ 1.0)} AP_n$ It divides the recall value from 0 to 1.0 into n points. It is common to smooth out the zig-zag pattern (at each recall level, replacing each precision value with the highest precision found to the right of that recall level[23]).

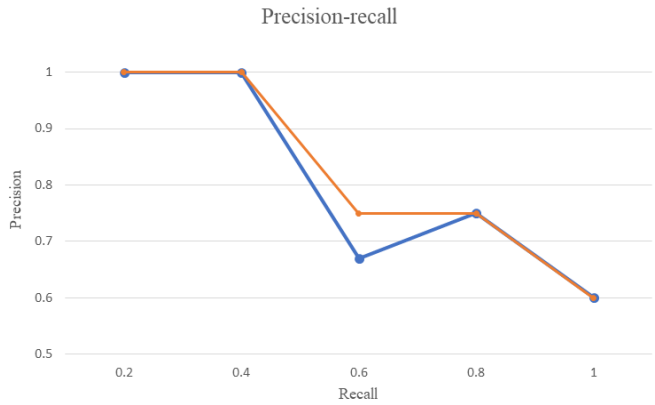


Figure 14: Transformed precision values: original, smoothed

Interpolated AP is calculated based on the area below the smoothed values. It is the basic idea behind the *mAP* variants.

- $mAP\ (mean\ Average\ Precision)$: there are numerous types of *mAP*. In *COCO*, a 101-point *interpolated AP* is used, which is the average over 10 *IoU* levels starting from 0.5 to 0.95 with a step size of 0.05. Hereafter, I refer to the *COCO* implementation by this name.

4.2.1.2 Protocols

There is no consensus about the evaluation metrics of the object detection problem. World-famous competitions such as *PASCAL VOC*[18], *COCO*[19], or Google Open Images Challenge have their ways to measure performance[21]:

- *PASCAL VOC*[18] has introduced *mAP* for evaluating the quality of detectors with an 11-point *interpolated AP* definition.
- *COCO* detection metrics[19] are similar but have additional measures such as *mAP* at different *IoU* thresholds from 0.5 to 0.95. There are also precision/recall statistics for small (area $< 32^2$), medium ($32^2 <$ area $< 96^2$), and large (area $> 96^2$) objects.

- *Open Images V2* detection metrics[20] are focused on average precision for each class and among all classes, but there are no metrics for objects grouped by their size. *OID* classes are organized in a hierarchy (e.g., car groups several specific classes like limousine or van). If a model claims that a van is a car, it is not punished as drastically as if it had claimed it was a cat.

4.2.1.3 Choice

I choose the *COCO* evaluation protocol mainly because it measures performance on different sized objects. As discussed earlier, the vehicle registration plates are relatively small on the images, so I wanted to see how different sizes affect performance. In my opinion, *OID* would not have been suitable for this task because I would have lost size-specific indicators, but I would not have won with the class-level metrics because there are only two classes (one of which is a custom one).

4.2.2 Workflow

Different data subsets have different roles during the development process. The main idea is to train models on the training set (with most images) and validate them on the validation set. When the best model is selected based on its validation performance, it is tested against a not seen before test set to spot if over-fitting occurs – which is the case when the model performs noticeably weaker on the new set. Sometimes, this type of overfitting happens as we select the best model based on the validation set – and it easily remains unnoticed if we do not apply this technique. So, I use the outlined workflow with the different subsets.

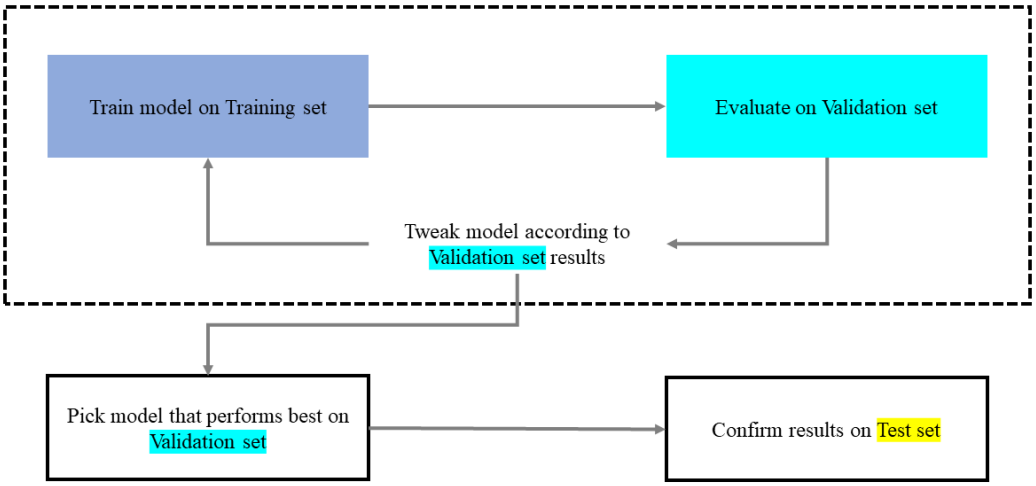


Figure 15: Applied workflow with different subsets[24]

Training is concluded with Python/Jupyter environment in a Colaboratory instance. After installing and testing external dependencies, the described steps are applied each time training is about to start.

4.2.2.1 Model preparation

I download base models from TensorFlow's model zoo with or without pre-trained weights for a fresh start. To continue a previous training, I import saved models from Google Drive.

4.2.2.2 Dataset download

The tfrecords are also stored in a zip file on Google Drive. As it stores data on a different server to the Jupyter instance machine, I download and extract the dataset too (thus, there is no need for network communication during training). Then, it is needed to define the paths to the training, validation, and test batches.

4.2.2.3 Pipeline

To start with, if a pre-trained model has been loaded, its weights and configurations are used for restoration.

During pre-processing steps, it is defined how many possible input classes exist. In my pipeline, I chose to encode the background as a class, too (with zero label, to encode non-object image parts as negative examples). Fixed shape input image resizing has been applied to input images as the dataset has pictures with various resolutions, but the detector expects a static input shape which corresponds to the backbone network's input. Anchor box generator properties are provided here, which I experimented with.

In the architecture part, the backbone network and the box predictor are configured separately (in the case of one-staged architectures, too). I changed the classifier and its convolutional hyperparameters (e.g., activation function, regularization, batch normalization). Box predictor properties (like convolutional kernel size or depthwise convolution, batch normalization, weight initializer, activation function) have been tuned similarly.

It is an exciting topic of what kind of loss functions to use in an object detection problem. I use *Focal loss*[26] for classification and *Huber loss*[25] for localization. It is worth mentioning that I started with sigmoid classification loss and online hard example mining (with three negative object samples per 1 positive) in a way like the original detector architectures. However, after researching the topic, I learned about *Focal loss* and *RetinaNet*[27], using a different approach eliminating foreground-background class imbalance (by down-weighting the loss assigned to well-classified items and by preventing easy negatives from overwhelming the detector during training[27]). Since Focal loss takes care of it, hard example mining is not necessary. *Huber loss* is used for localization because it handles outliers outside a delta value quite well. Both loss functions are included in the calculation of my model's total loss with the same weights.

Post-processing properties, like how many images are allowed on the network's output, are essential questions to be decided. Since there are relatively many cars and license plates on a street scene, I maximized the network output in 100 simultaneously detectable objects (although in the database, a sample image has four objects on average – but it was shown that many of them are incompletely annotated).

Training properties, like batch size, number of steps, variable freezing, optimization algorithm, and the corresponding subsets' path, are defined in the last part of the pipeline. During training, I use three data augmentation techniques (horizontal flip, crop and padding, brightness adjustment) to prevent overfitting. Padding can be interpreted as an out zooming process that reduces bounding box sizes, thus helping one-stage detectors, which are generally poor in localizing small objects. This decision was inspired by the procedure described in the original *SSD*[33] paper. An image and its box coordinates are always augmented with the same transformation.

Figure 16: Augmented image samples during training

During the evaluation, augmentation is not used to measure performance objectively and ensure that the results are independent of random transformations.

4.2.2.4 Prerequisites

Before training, one last inspection is applied to verify that configuration is correct and everything is ready. A TensorBoard instance is also started pointing to the log directory of training.

4.2.2.5 Training and evaluation

Training is usually a long process. Primarily, it can be followed on TensorBoard, which shows a dashboard of the running task. The running Jupyter cell also logs live loss value per hundred steps. At certain specified intervals, training is interrupted, and evaluation is done. On these occasions, model checkpoints are also saved to make the model states restorable (for early stopping or state preservation in case of a failure).

After the training process, a standalone evaluation can be executed to verify the results and display the *COCO* protocol metrics.

4.2.2.6 Export model

After the work is finished, the model can be saved in three different ways:

- First, all the training files and checkpoints are zipped and saved to Google Drive.
- Second, only the latest (or the selected) checkpoint is preserved, and detailed log files are discarded (only metrics are retained like loss; model outputs for specific images are dropped) and saved to Drive.
- The third option is to convert the model to TFLite and save it.

A training iteration includes the steps discussed above. Figure 17 shows the simplified, high-level diagram of this process:

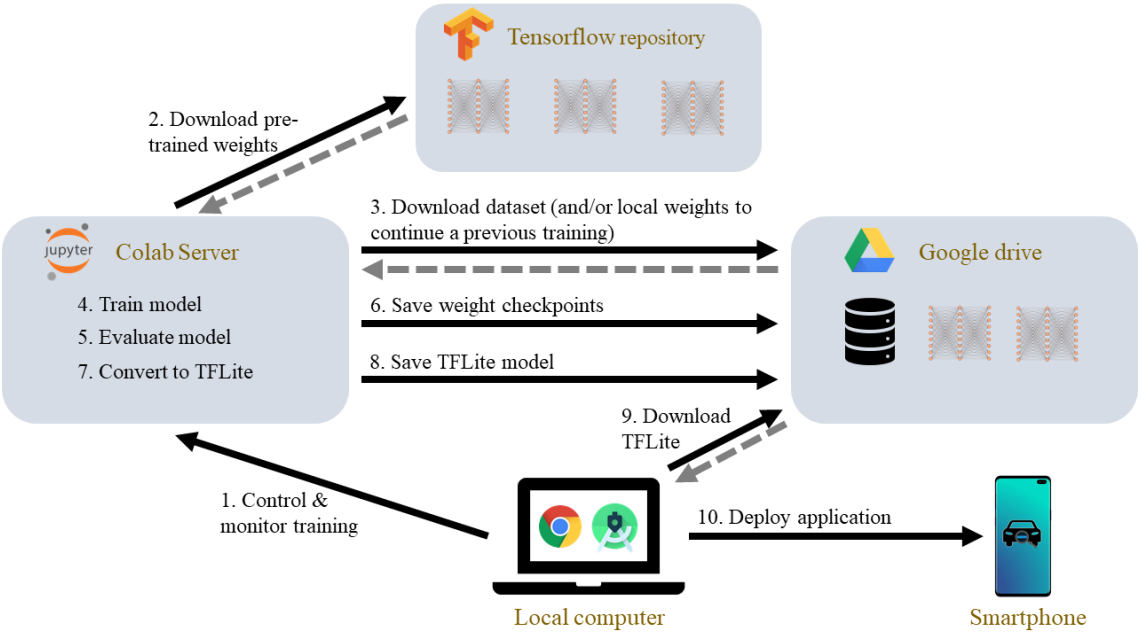


Figure 17: Steps needed to train and deploy a model

After defining the applied workflow, it is time to dig into detector architectures.

4.2.3 Architecture

Generally, there are two types of deep learning-based detector architectures: two-staged (e.g., *R-CNN*) and one-staged (like *CenterNet*, *YOLO*, *SSD*) variants. I briefly describe both types in the following.

4.2.3.1 Two-staged detectors

Until the mid-2010s, complex ensemble machines were the best performing models in object detection. In 2014, *Region-based Convolutional Neural Network*[28] (*R-CNN*) was introduced for detection and segmentation tasks. Its main idea was to use a region proposal for generating category-independent proposals, then feeding them to a feature extractor *CNN*. In this arrangement, there is a third module, which is a set of class-specific linear *SVN*s. In this architecture, the separate modules must be trained independently.

In 2015, *Fast R-CNN*[29] had been introduced. It uses *VGG-16* as a feature extractor and made it possible to train the whole system in one piece. The main architectural difference to the previous version is that it feeds an input image directly to the *CNN* to generate a convolutional feature map. From that, proposals are identified by an *RoI* (Region of Interest, max-pooling) layer, which is fed to dense layers outputting coordinates. There is also a softmax layer from the RoI feature vector, which predicts the class of the proposed region. *Fast R-CNN* is considerably faster than its predecessor because, for 100 region proposals, there is no need to feed all of them independently to the *CNN* – *RoI*s from the same image share computation and memory.

The latest variant, *Faster R-CNN*[30], consists of two parts: a fully convolutional *RPN* (Region Proposal Network) and a *Fast R-CNN* detector using the output of the former. The two networks might share a common set of convolutional layers. RPN is a small network working in a sliding-window fashion, which predicts the regions unlike *R-CNN* or *Fast R-CNN*, where a selective search algorithm is used for this purpose (which is slower).

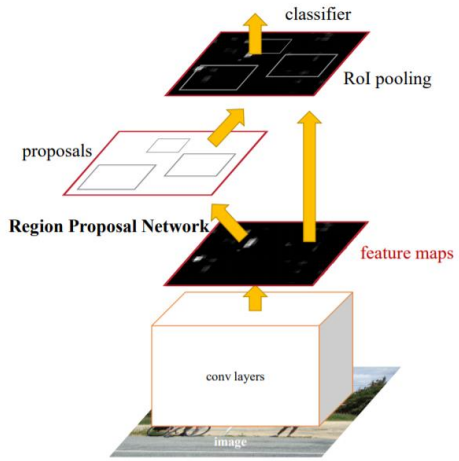


Figure 18: Faster R-CNN architecture. Source: [30]

4.2.3.2 One-staged detectors

You Only Look Once[32] (*YOLO*) has been introduced in May 2016. It is a unified architecture with a single network containing 24 convolutional layers followed by two fully connected ones. The convolutional layers extract features from images, while the dense layers predict coordinates and probabilities. The input image is divided into an $N \times N$ grid in which a cell is responsible for predicting boxes in its area (one box has five values: X_{min}, Y_{min}, $width$, $height$, $confidence$). A cell outputs B boxes and their confidence values for every C classes, so the output shape is $N \times N \times ((B * 5) + C)$. In general, this architecture tends to make more localization mistakes (mainly struggles with small objects, which is partly related to loss functions discussed earlier) but is fast and less likely to predict false positives on the background (compared to two-staged detectors). *YOLO* has various variants (v4 is the latest so far).

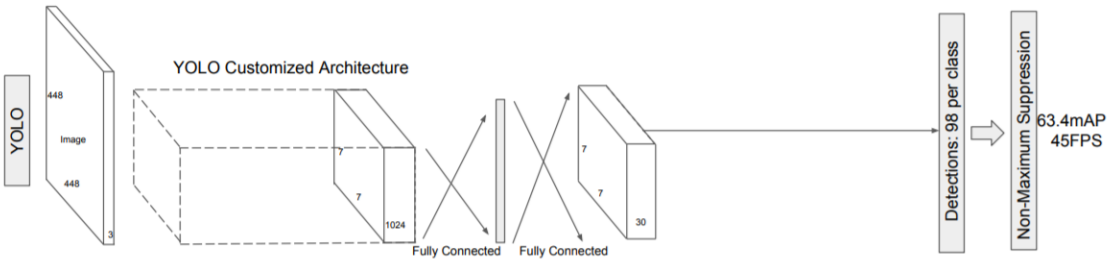


Figure 19: YOLO architecture. Source: [33]

Single Shot MultiBox Detector[33] (*SSD*) is similar to *YOLO*. In the first part of the architecture, there is an image classification *CNN* (originally *VGG-16*) called the base network. After that, multiple convolutional layers are implementing a detector structure for multi-scale object localization. These layers progressively decrease in size towards the end of the architecture, allowing detections at multiple scales (unlike *YOLO*, which initially operates on a single scale feature map). *SSD* uses a set of default bounding boxes for each feature map cells on each feature maps.

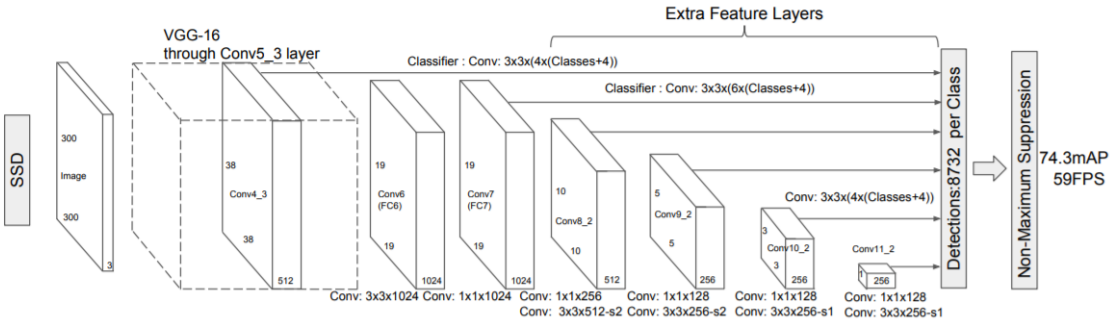


Figure 20: SSD architecture. Source: [33]

CenterNet[34] has been released in 2019, and it uses a different approach compared to other architectures presented so far. It is based on the keypoint-based *CornerNet* network and uses triplets to localize objects. It first generates two coordinates (XY_{min}, XY_{max}) localizing a proposal, then takes its geometric center as the third point. Then, it is inspected whether the center key point's region is predicted as the same class as the whole bounding box. This way, not just the box borders but its central region's visual patterns are also analyzed, providing a more robust approach to reduce false positives.

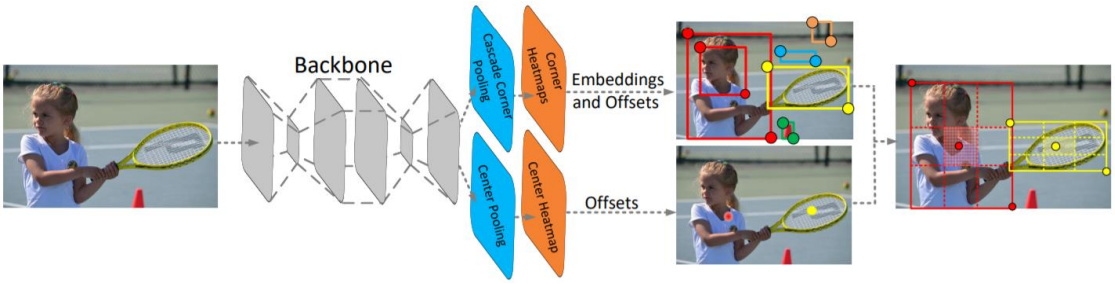


Figure 21: CenterNet architecture. Source: [34]

4.2.3.3 Selection

Generally, a two-staged detector takes a classifier and evaluates it in different locations. These are slower architectures because of the separate steps but performing better with various sized objects. TensorFlow Object Detection API currently supports *Faster R-CNN* from these variants. Although its inference speed is comparable to one-staged architectures, TFLite conversion is not working – therefore, I turned to one-staged systems. From them, *SSD*s and *CenterNet*s are supported (*YOLO* is missing and is not likely to be added at all). Although *CenterNet*s look promising (similar benchmark speed and similar *COCO mAP* to *SSD*s), again, the conversion is not yet implemented. For these reasons, I chose *SSD*, but I noted that *CenterNet* would be a viable option, and it may be worth examining in the future.

4.2.4 Optimization

In this part, I compare the models via summarizing their results in tables. Since multiple values were measured, I only display *mAP*s. All the metrics (training duration, *AP*, *loss*) of the results can be viewed in a comprehensive table in Appendix D.

Once the architecture was selected, I started training the models. In the beginning, to save time, pre-trained weights were used (only on the same *COCO 2017* data set). I

kept the same settings of the networks as they were pre-trained. Cosine decay learning rate was applied in each case to prevent too optimistic gradient change at the beginning (started from $1/40$ of the base learning rate, trained like this for $1/20$ of time, then increased it).

Since training a detector is time-consuming, I drew conclusions based on the actual results and the learning curve's nature after three epochs (one epoch is when the entire dataset is passed forward and backward through the network once), which is 16,000 images. Applying early stopping (training finishes when model performance permanently stagnates/falls back) would be the best solution. However, in this case, less training iteration would have been possible because, at the end of the flattened learning curve, there is a minimal improvement over a long period of time.

4.2.4.1 Backbone

Since the detector is planned to run on smartphones, the priority was to choose a relatively small and fast model. To start with, I tried 3 different options: *ResNet50*, *MobileNet v2* (with 320x320 input), and *EfficientDet D0*.

ResNet[35] (*Residual Network*) was the winner of the ImageNet 2015 challenge. It has introduced skip connections with its residual units. Skip connections are helpful because while training, the input signal can make its way across the network, so even if deeper layers have not quite started learning (their output is close to zero), the network can progress. I used a smaller version of the original model with 50 layers.

*MobileNet*s[36] are special models optimized for mobile/IoT inference. They are relatively small and fast, but their accuracy by no means among the best. *V2* uses linear bottlenecks and inverted residuals. According to the *MobileNet* designers, bottlenecks store all the necessary information, and between them, expansion layers serve for extraction with non-linearities. Thus, bottlenecks are linear layers to prevent non-linearities from destroying the original information. The other novel idea was that as bottlenecks store the information, shortcuts are needed directly between them. I started with the smallest version of this network (320x320 input), which provides outstanding performance due to its use of depthwise separable convolution.

EfficentDet[37] has been introduced in July 2020 by Google Brain and is currently the benchmark on *COCO* (55.1 *AP*). It is a neural network family, especially for object detection. Its paper has introduced the bi-directional feature pyramid network, which

learns to weigh input features with different resolutions before fusing them. I started with the smallest version of this network (512x512 input), which is relatively fast despite its size.

I started with the smallest variants of all the three networks introduced above. Results can be found in Figure 22.

				COCO metrics					
				Precision					
model	size	input	batch	mAP	mAP@.50	mAP@.75	mAP S	mAP M	mAP L
resnet50	241.68 MB	640 x 640	8	0.389	0.678	0.404	0.116	0.394	0.531
efficientdet_d0	42.51 MB	512 x 512	16	0.314	0.581	0.302	0.019	0.321	0.492
mobilenet_v2	19.78 MB	320 x 320	16	0.278	0.504	0.269	0.007	0.248	0.492

Figure 22: Comparison of the smallest models of each type

I found this comparison problematic as the performance was suspiciously proportional to the input size of the models. It is also important to note that *ResNet50* was trained with batch size eight because I had insufficient memory to train it with 16-sized batches. To fix these issues, I switched to *EfficientDet D1* and *MobileNet v2 640*, and the same batch /step values have been applied.

				COCO metrics					
				Precision					
model	size	input	batch	mAP	mAP@.50	mAP@.75	mAP S	mAP M	mAP L
resnet50	241.68 MB	640 x 640	8	0.389	0.678	0.404	0.116	0.394	0.531
efficientdet_d1	63.36 MB	640 x 640	8	0.391	0.671	0.4	0.119	0.396	0.548
mobilenet_v2	19.78 MB	640 x 640	8	0.379	0.678	0.386	0.115	0.386	0.517

Figure 23: Comparison of different types with the same input

This comparison shows more balanced results. Unless *ResNet50* is by far the largest model, its performance is not outstanding. In fact, *EfficientDet* seems to outperform it. *MobileNet* has modest results; however, it is not far from the benchmark.

As inference speed and model size are critical aspects, *ResNet* may not be considered. Even though *EfficientDet* is four times larger than *MobileNet*, the former runs in 54 ms according to TensorFlow's benchmark, while the latter runs in 39 ms (I measured similar inference times). As their *mAP* performance is relatively close, I decided to experiment further with both networks before committing to one.

4.2.4.2 Batch size

Batch size is the number of samples propagated through the network at once. Bigger batches allow computational speedups but reduce the ability to generalize. On the other hand, if a model is trained with too small batch sizes (perhaps caused by too sample-specific gradient updates), performance decreases. There is an ideal range of batch sizes affected by model parameters and the current dataset.

I trained both networks with different batch sizes. The maximum size I could use was 8 in the case of *EfficientDet* and 24 for *MobileNet*. When training with batch size 1, I reduced the learning rate by order of magnitude (from 0.08 to 0.008) to avoid divergence caused by too large gradient updates.

				COCO metrics					
				Precision					
model	size	input	batch	mAP	mAP@.50	mAP@.75	mAP S	mAP M	mAP L
mobilenet_v2	19.78 MB	640 x 640	24	0.368	0.669	0.376	0.108	0.367	0.512
mobilenet_v2	19.78 MB	640 x 640	16	0.372	0.666	0.378	0.101	0.381	0.508
mobilenet_v2	19.78 MB	640 x 640	8	0.379	0.678	0.386	0.115	0.386	0.517
mobilenet_v2	19.78 MB	640 x 640	4	0.377	0.677	0.381	0.116	0.385	0.507
mobilenet_v2	19.78 MB	640 x 640	1	0.261	0.535	0.225	0.09	0.327	0.301
efficientdet_d1	63.36 MB	640 x 640	8	0.391	0.671	0.4	0.119	0.396	0.548
efficientdet_d1	63.36 MB	640 x 640	4	0.388	0.681	0.39	0.119	0.398	0.541
efficientdet_d1	63.36 MB	640 x 640	1	0.291	0.578	0.262	0.1	0.33	0.373

Figure 24: Training results with different batch sizes

Based on the results, batch sizes 4 and 8 were the most ideal. Both types of networks produced the most accurate outputs trained with batch size 8. At larger sizes, a decline is observed, especially on detecting small objects. Observing the other extreme at batch size 1, gradient update per every single image seems to confuse the network and reduce its performance. Since the same trend can be observed between the two models (*MobileNet* lags by 2-3% *mAP* in every configuration), I decided to use *EfficientDet D1* for the rest with batch size 8.

4.2.4.3 Optimization algorithm and learning rate

Optimization algorithms[40] are used to update network parameters (such as weights) to minimize model loss while training. The most popular method is *Stochastic Gradient Descent* and its mini-batch variant. This algorithm is relatively easy and powerful, but it usually results in slow convergence. To overcome this issue, *Momentum* optimization is usually applied, which introduces and updates the velocity of gradients

instead of their specific value. There are different optimizers, such as methods using an adaptive learning rate, such as *RMSprop* or *Adam.*

I tried the *Adam, Momentum,* and the *RMSprop* variants. Initially, I used the same learning rate (8×10^{-2}) in all three cases. However, *Adam* and *RMSprop* training indicators showed signs of divergence, so I changed their values to the default learning rate (2×10^{-3}) based on TensorFlow's optimizer definition. It seems that *Momentum*'s recommended optimum is roughly an order of magnitude larger than what is ideal for the other two.

				COCO metrics					
				Precision					
model	input	learning rate	optimizer	mAP	mAP@.50	mAP@.75	mAP S	mAP M	mAP L
efficientdet_d1	640 x 640	0.002	Adam	0.392	0.682	0.41	0.103	0.412	0.546
efficientdet_d1	640 x 640	0.08	Momentum	0.391	0.671	0.4	0.119	0.396	0.548
efficientdet_d1	640 x 640	0.002	RMS Prop	0.361	0.643	0.374	0.079	0.394	0.512
efficientdet_d1	640 x 640	0.08	RMS Prop	0.155	0.333	0.13	0.007	0.154	0.213
efficientdet_d1	640 x 640	0.08	Adam	0.011	0.031	0.004	0	0	0.018

Figure 25: Comparison of different optimization algorithms

The last two rows show typical cases of too high learning rates. With values closer to their optimums, algorithms could be compared more realistically.

Adam and Momentum produced very similar results, while RMSprop could not perform at their level. However, before concluding, the nature of the learning curves is also worth analyzing to see which algorithm has started to converge and which is still improving.

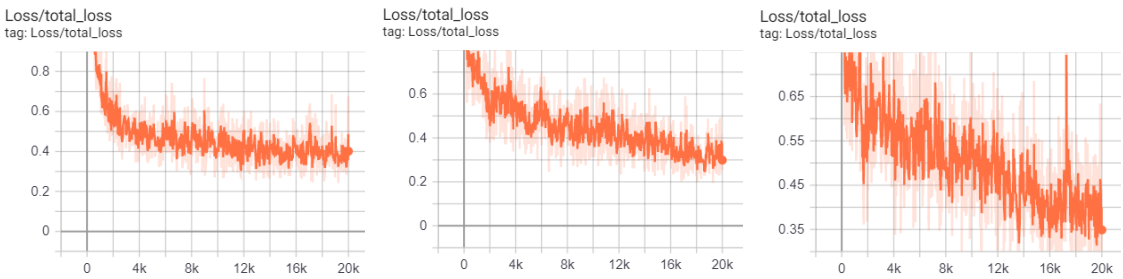


Figure 26: Total loss trends of RMSprop (left), Momentum (middle), and Adam (right)

What can be seen from here is that RMSprop not only lags behind but already converges strongly and just slightly improves after 14,000 steps. On the other hand, Momentum and Adam still strongly improve. Of the two, Adam oscillates more, which may sign a too high learning rate. However, the unbroken improvement after 20,000 steps

(and the fact that I use decay learning rate, but oscillation does not decrease) rather suggests that this is more of a behavior due to the algorithm's nature, which needs further investigation. Another thing to spot: although *Adam* performs better in almost everything, *Momentum* has a significant edge in detecting small objects.

Therefore, I executed another 20,000 steps with the last two algorithms, and I also tried Adam with an order of magnitude lower learning rate (2×10^{-4}). After this iteration, each case surpassed the optimum and started to overfit. It seems that *Momentum* has the edge: it has improved until reaching 0.393 *mAP*, while the best of *Adam* is 0.391, before stagnating around 0.387. The latter could not improve with the lower learning rate either, resulting in 0.38 *mAP*. Thus, the final model to take is the one trained with *Momentum*.

				COCO metrics					
				Precision					
model	input	learning rate	optimizer	mAP	mAP@.50	mAP@.75	mAP S	mAP M	mAP L
efficientdet_d1	640 x 640	0.008	Momentum	0.393	0.686	0.391	0.119	0.392	0.545
efficientdet_d1	640 x 640	0.002	Adam	0.391	0.685	0.404	0.108	0.404	0.543
efficientdet_d1	640 x 640	0.0002	Adam	0.383	0.669	0.38	0.116	0.39	0.528

Figure 27: Comparison of the best results of Momentum and Adam (all three cases indicate the best performing models before overfitting)

4.2.5 Post-processing

After the final model was created, it was time for quantization, TFLite conversion, and to generate its auxiliary structures.

4.2.5.1 TFlite conversion and quantization

During the conversion, it was necessary to note that not all TensorFlow operations are implemented in TFLite. To avoid conversion problems, I checked the compatibility table to see whether the operations in the model (e.g., activation function, depthwise convolution) could be converted. The TFLite file created at the end of the process can be run on Android devices using an interpreter.

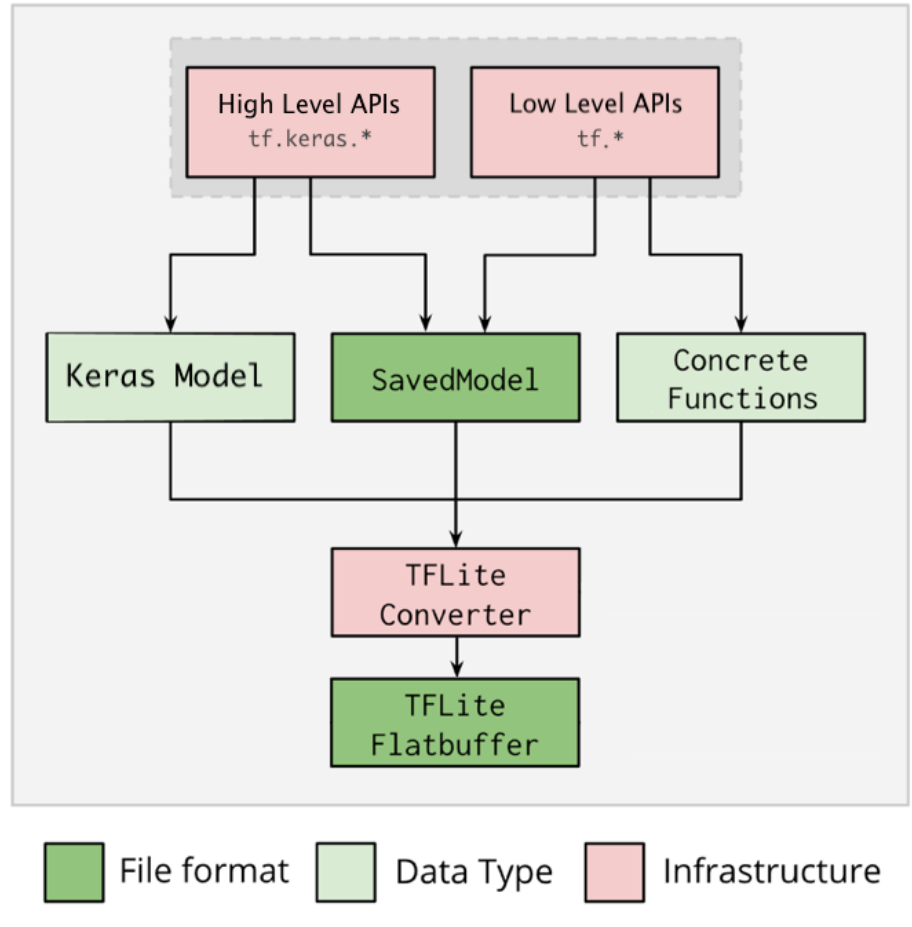


Figure 28: The TFLite conversion process. Source: [38]

4.2.5.2 Input and output data formats

The detector expects 640×640 sized RGB images to work with. The input is channels_last encoded (order of indices: height, width, red, green, blue). The value of one channel of a pixel is encoded in 8 bits (1 byte). On all the three RGB channels, the values are interpreted as 8-bit integer numbers, which must be between 0 and 255.

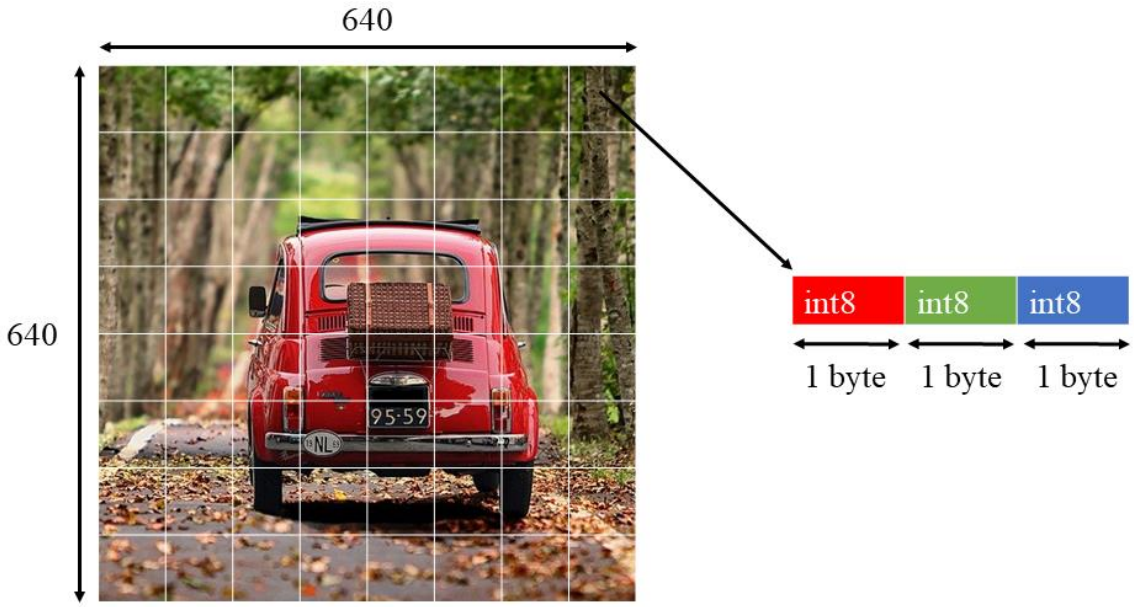


Figure 29: An illustration of the expected input

The model outputs a HashMap containing four arrays mapped to the indices 0-3. Arrays 0, 1, and 2 describe 100 detected objects, with one element in each array corresponding to each object. There are always 100 detections. A brief description of the arrays is as follows:

- Detection boxes: Multidimensional float32 tensor of shape [1, num_boxes, 4] with box locations. Floating-point values are between 0 and 1, the inner arrays representing bounding boxes in the form [top, left, bottom, right].
- Detection classes: A float32 tensor of shape [1, num_boxes] with class indices, each indicating the index of a class label from the labels file.
- Detection scores: A float32 tensor of shape [1, num_boxes] with class scores. Values between 0 and 1 representing the probability of the detected class.
- Number of boxes: float32 tensor of size 1 containing the number of detected boxes.

4.2.5.3 Auxiliary structures

It is also necessary to use an auxiliary structure to interpret the output of the detector. To do this, I created a file called labelmap.txt that contains the names of the output classes line by line (including the background class).

The model is accompanied by a file called model_info.txt. It contains general information about the network, such as the definition and interpretation of the input and output data formats required for use.

4.3 Summary

This section dealt with data preparation, presentation of the theoretical background, architecture selection, training and fine-tuning, and then the detector's post-production.

A total of 184 hours of training (with 23 different configurations) took place. The detailed results of the final model can be observed in Figure 30. Compared to the first initial training from this network family (on *EfficientDet D0*), 8% gain has been reached in *COCO mAP* and 10% gain in *mAP* for small objects. The final detector's size is 35.38 MB, its quantized counterpart is 11.191 MB.

batch	step	learning rate	pre-trained?	duration
8	30000	lr0.08, cos_decay, warm:1000/0.002	COCO	8h 50m

Precision					
mAP	mAP@.50IOU	mAP@.75IOU	mAP (small)	mAP (medium)	mAP (large)
0.393	0.686	0.391	0.119	0.392	0.545

Recall					
AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (med)	AR@100 (large)
0.299	0.504	0.539	0.235	0.554	0.69

Loss			
localization	classification	regularization	total
0.162675	0.959799	0.058981	1.181456

Figure 30: COCO evaluation results of the selected model

After the detector has been created, the next step is to present the system in which it operates.

5. System overview

In this section, the complete system is overviewed. I explain the main design decisions, and some of the more exciting implementation details are also covered.

5.1 Android application

The client application's main task is to detect stolen vehicles, then report them using location and time data. It is possible to run an evaluation on loaded images as well as on the live image feed. The user constantly sees exactly what has been recognized. Stolen vehicle and user data are stored in a local SQLite database, which is synchronized in the background with the API. Camera operations include front/back camera switching, image saving, tap to focus, and pinch to zoom. In the following, I present the Android application's architecture and the stolen vehicle recognition pipeline.

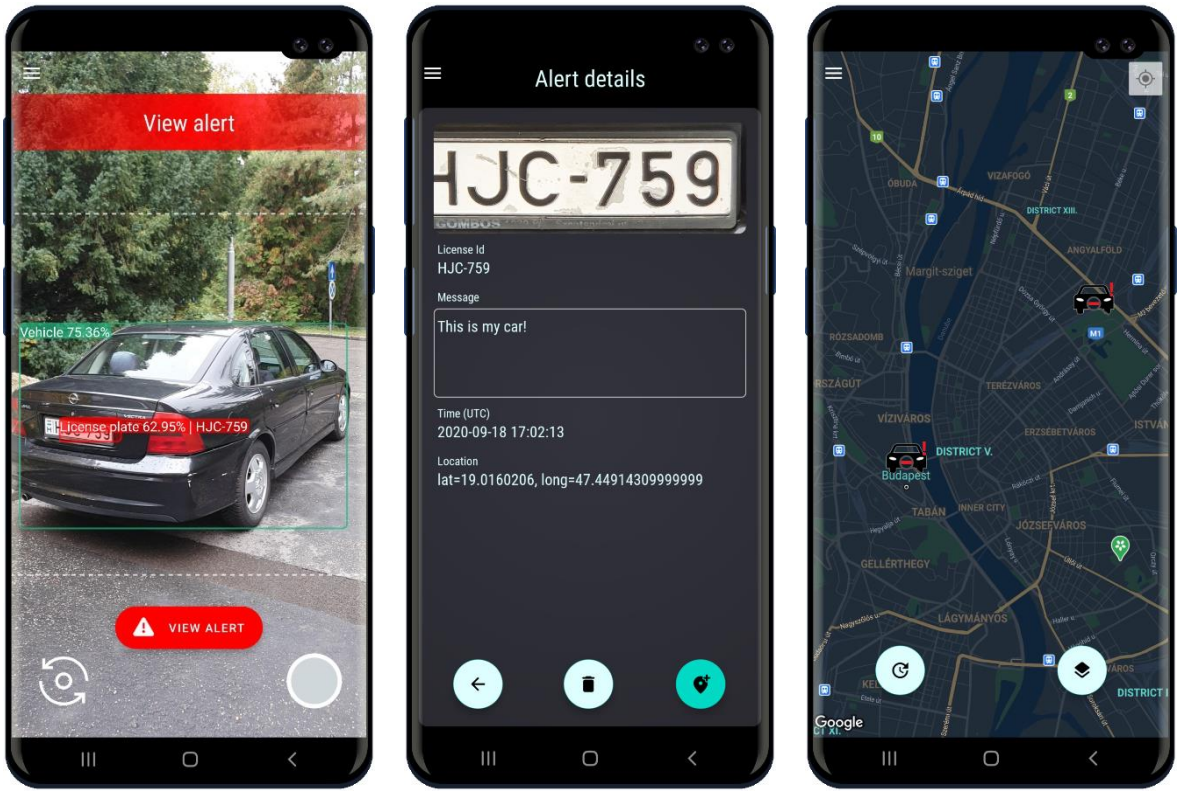


Figure 31: Live inference (left), recognition details (middle), map of detected vehicles (right)

During development, attention has been paid to the user experience. As UX is outside of the scope of this writing, it is not presented in detail. More pictures of the application can be found in Appendix A.

5.1.1 Architecture

I used the *Model View ViewModel* (*MVVM*) UI design pattern. It is an event-driven model, invented by Microsoft to take advantage of data binding capabilities. In *MVVM*, the *View* contains UI descriptive code often in a declarative (XML, XAML, HTML) form, and the connection to the *ViewModel* is realized with explicit data binding. Therefore, there are fewer classic coding tasks in *View*s, and the business logic components can be easily separated.

Figure 32: MVVM components with their relations

There are sub-layers in the model level of the application. I explain their hierarchy through the steps of reporting a single recognition item. Suppose a new stolen vehicle was detected on the live camera feed, and the user selected to report it. In this case, the user sees a *RecognitionActivity*, which has a *ViewModel* storing its UI related state. The recognition item gets stored in a list wrapped in a *LiveData* object (which is observable from the Activity). When the user clicks on the send button, the related data is transmitted to the *RepositoryService* in the model layer. Inside this service, there is the *RecognitionRepository*. It hides further data operations (database handling, API communication) from the outside. When it receives a new recognition, it transforms it into a format stored in the Recognition table and then persists it with *RecognitionDAO* (data access object) to the database. After that, it calls *ApiService* to send recognition to the server. When the success response arrives from the API, *RecognitionRepository* updates the corresponding item in the database. Until the operation is not successful, the user sees that the recognition is pending. Pending items can be deleted or re-sent at any time.

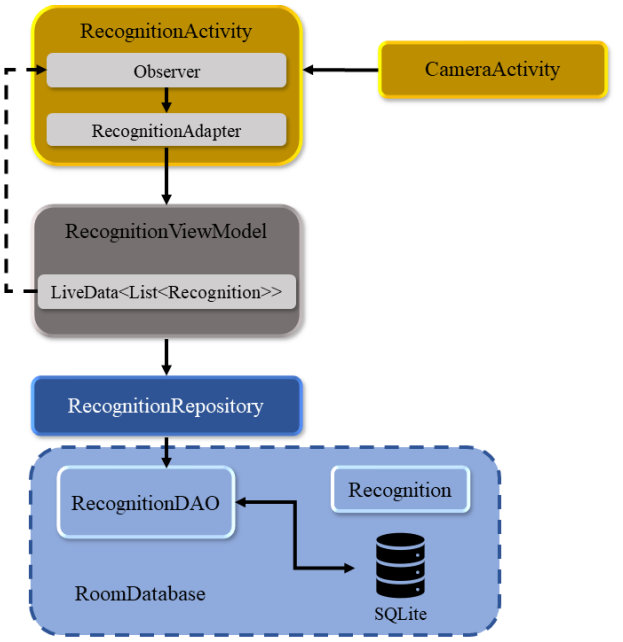


Figure 33: Recognition reporting steps through different architectural layers

Beyond recognition, the application stores several other information in its local relational database (list of stolen vehicles, account information, metadata). For further insight, the data model schema diagram has been added to Appendix B.

5.1.2 Stolen vehicle recognition pipeline

To allow the user to interact with the application, the pipeline calculations run in the background. First, the input image must be in the appropriate format. Any picture with its dimensions (provided by device camera or loaded from storage) is converted to have an $n \times n$ size. It is followed by resizing the image to the detector's input (640×640). Finally, a byte buffer conversion occurs, which is the expected data format to feed the TFLite model with.

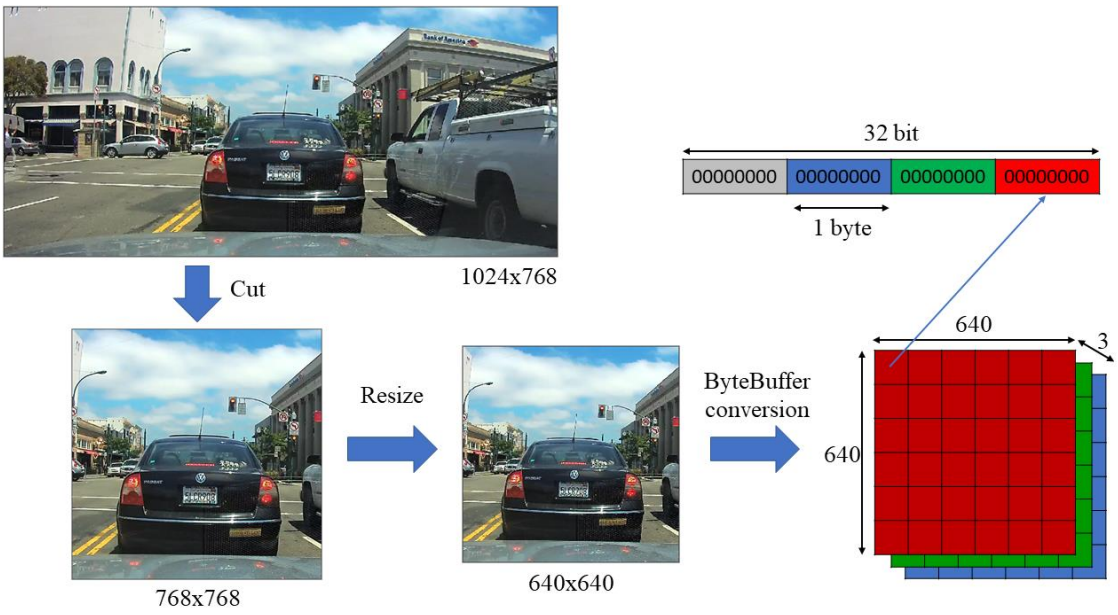


Figure 34: Image pre-processing steps

Once the desired input is generated, evaluation can follow. The byte buffer is first applied to the input, and here the model fixedly outputs the 100 most possible detections. Objects below a minimum certainty threshold are discarded (this value can be set in the application). The detector's output contains the predicted classes of detected things, their certainty values, and their enclosing bounding boxes' coordinates. The resulting items form a list. Suppose one of the elements denotes a license plate. In that case, the associated image snippet's location is cropped from the original image to pass through a text recognition model to obtain the license plate text. Slices are cut out from the original picture because license plates are typically small, and the original image retains information better due to its higher resolution.

Once license plate texts are available, plotting begins. Initially, a transparent image is created of the same size as the image on the device's screen. Bounding boxes are drawn with their information (class name, probability) on that image. The license plate texts are checked before drawing to see if they belong to stolen vehicles. If so, the bounding box is drawn in red, and an alert button appears on the screen. If not, it is drawn like other boxes. Once this mask image is created, placing it on the original image allows the user to see where and what objects were detected and recognized.

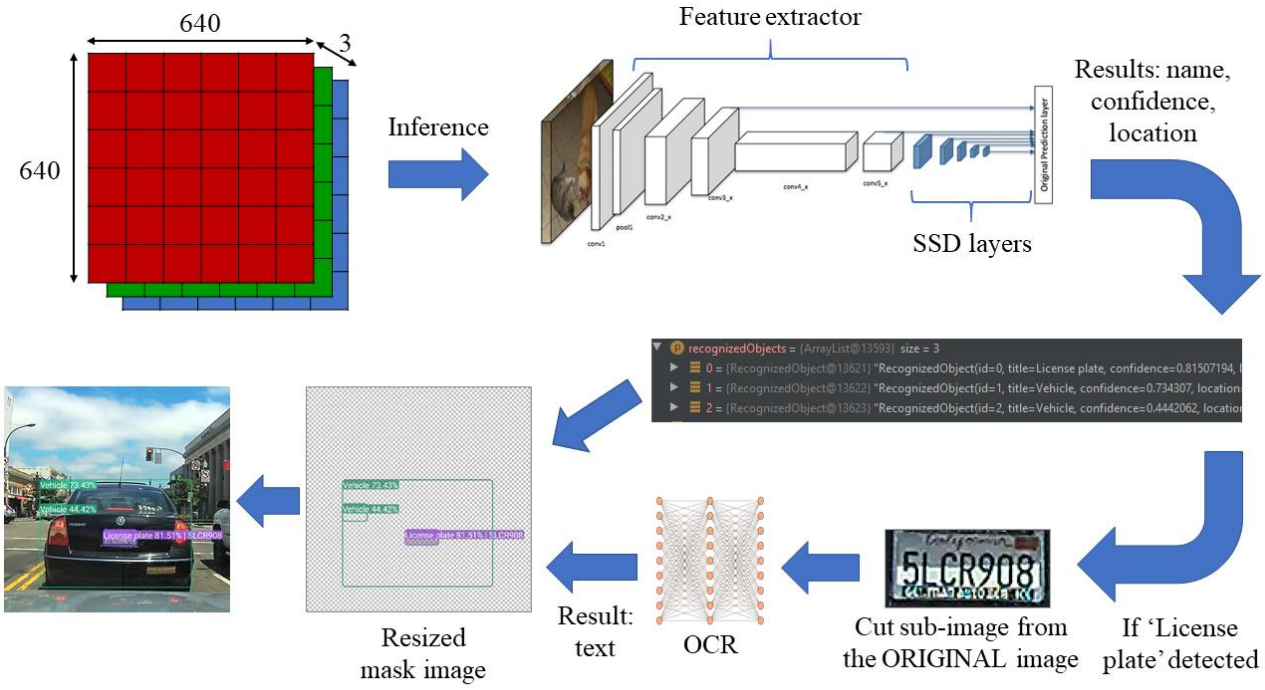


Figure 35: Stolen vehicle recognition steps

I tested the pipeline runtime on a Samsung Galaxy S10 smartphone. The average processing time for the different steps is as follows:

- Image pre-processing: 9 ms (Android code)
- Detector inference: 123 ms (quantized TFLite model, runs on mobile GPU)
- Cut image snippet: 3 ms (Android code)
- *OCR* text recognition: 181 ms (MLKit model)
- Create a mask image with bounding boxes: 8 ms (Android code)

In total, the pipeline averagely runs in 324 ms. It means that three images are processed within a second so that it can work properly, even with a live image feed.

However, the runtime can vary depending on how many license plate sub-images are fed to the text recognizer. It appears that the *OCR* is the bottleneck in the pipeline, as it is the slowest element in the case of numerous license plates (they are fed separately). Creating a custom text recognizer is room for improvement in the future. Another option is to mask the time of the pipeline calculation from the user by applying an object tracking algorithm between two calculations.

All in all, there is room for improvement with the previously presented options. In turn, with the current configuration, the application still works as intended and can already recognize stolen vehicles even through a live camera feed.

5.2 Server application

The server application provides the API for the clients and manages registered users. It has a stateless REST API, so a user needs to authenticate itself every time querying the server.

5.2.1 Architecture

The server has a three-layered architecture. Because it is responsible for API service and data storage, it does not have a separate View layer (only a simple HTML UI is available).

Instead of the UI layer, there is the communication layer through which the API service can be accessed. Because it is a separate layer, it is more loosely coupled with other layers, reducing the chances of leaking information from other parts. User authentication is done with Http basic authentication.

In the business logic layer, the Authenticator module checks requests and does not allow them to be executed when the required permissions are missing. The Interactor contains the main business logic.

The data access layer contains the DAO classes responsible for handling their tables and providing a unified interface for retrieving/writing data.

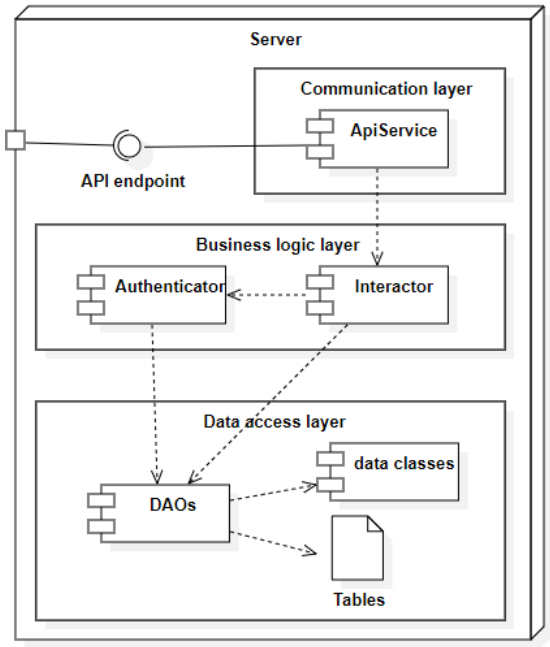


Figure 36: Server structure diagram

5.2.2 Database

Since I decided to use a self-created database for performance reasons, I briefly describe its main guidelines. It is a NoSQL variant with an in-memory approach. The tables store information in an object-oriented manner (instead of relational data). The table contents are in JSON format (like in the case of MongoDB). To encode/decode JSON files, the server uses the Gson library.

There are tables of stolen vehicles, current reports, and user accounts. To these contents, there are history files (write only) as well. They store all items using a timestamp and a version number to support recovery and traceability. History tables are not stored in memory, and when a data table is updated, its corresponding history is automatically updated. There is a meta content storing size and timestamp information of the previous tables. Lastly, there is an Event table that records system logs (also write-only). The data model schema diagram can be found in Appendix D.

When accessing data, the database serves it from memory, making API responses fast because there is no need to wait for table I/O operations. The memory content is synchronized with the corresponding table in the background. It is a viable solution as a very large amount of data is never stored on the server (images are not uploaded). To validate this, I examined one item from the largest JSON object type (Report), which is precisely 173 bytes. Multiplied by 1 million, it turns out that the server needs 173 MB

memory, which is acceptable. It is a severe overestimation, though, as the stolen vehicles list obtained by web scraping typically has a few thousand items. That is the maximum number of records that the in-memory database ever has (if every stolen vehicle has been detected at once). As history content is only stored persistently, extensive API usage does not saturate the memory either.

5.2.3 API

The API is divided into five parts: Vehicles, Reports, Report history, Self, and Users. These names are also the corresponding API call prefixes. All parts have similar actions and a unified calling convention. All actions are subject to specific permission, which is evaluated every time before serving. There is also a status page describing the API. Figure 37 & 38 show the two most common server and client communication types.

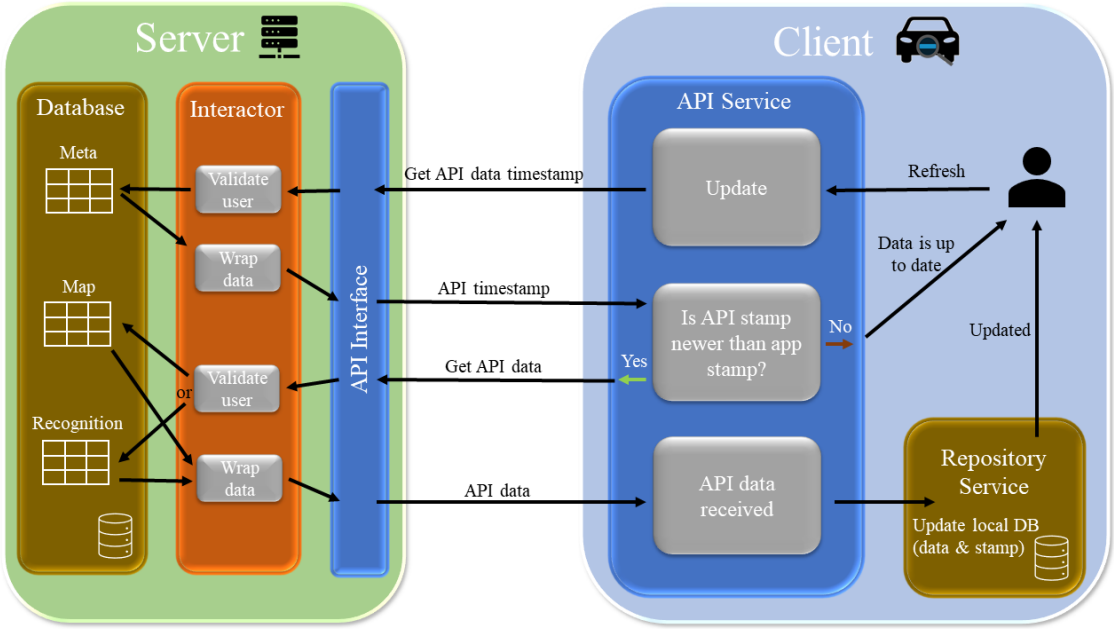


Figure 37: Client updates via the API

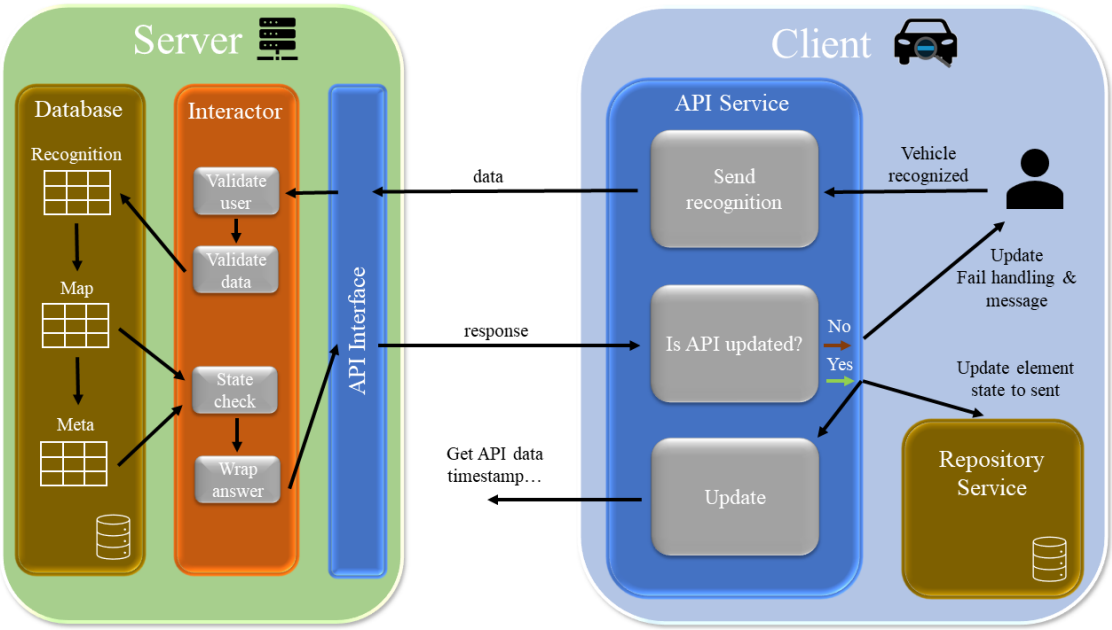


Figure 38: Client reports a vehicle

5.2.4 Permission management

As the nature of the stored data (location and timestamp of stolen vehicles) could potentially allow abuses, there is strict role-based permission management in the system. Users with specific roles are eligible to execute various operations.

There are ADMINISTRATOR, API_REGISTER, SELF_MODIFY, API_GET, and API_SEND permissions.

- API_GET lets an authorized account to download reports.
- API_SEND makes it possible to send recognitions to the server.
- SELF_MODIFY is needed to prevent blacklisted users from deleting themselves and re-register.
- An ADMINISTRATOR user can modify the server and any user's permissions at any time. If someone's behavior is suspicious, an Admin can revoke permissions, delete a user, or deactivate and blacklist it. An ADMINISTRATOR can register a new user with specific permissions.
- The default/guest user in the client application is an account with only an API_REGISTER role. This way, it is possible for newcomers to register their new accounts. If someone tries to use the application without signing in is, in fact, utilizes this user. As its only API permission is registration,

although someone can detect vehicles on-device, he/she cannot report them or see the actual reports. This API_REGISTER role prevents anyone outside the Android app from registering. The default user can create a new account with SELF_MODIFY, API_GET, and API_SEND permissions.

This section gave a brief overview of the complete system. During testing, the functionalities were validated first on the application level (the client on Android phones, the server with Postman), followed by integration testing (aimed to test client and server communication). I used Samsung Galaxy S10 and Samsung Galaxy J6 smartphones, and a Pixel 3XL emulated device to test the application. Runtime values reflect the results measured on the S10 phone. The stolen vehicle was mocked (our car's license plate was used for this purpose).

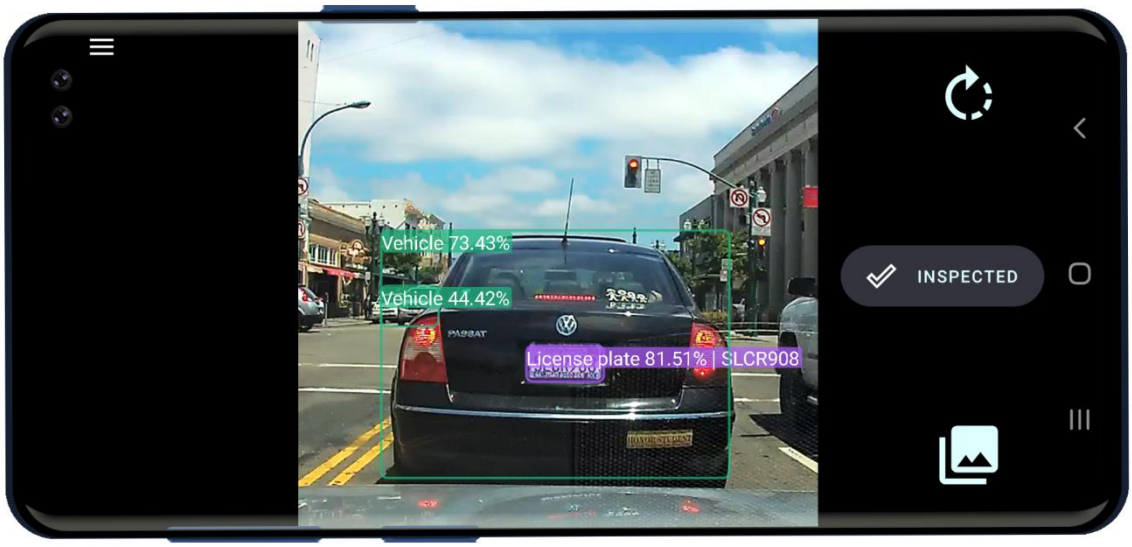


Figure 39: Testing detection on a loaded image on-device

6. Summary

This project is aimed to create a system that recognizes stolen vehicles anywhere and in real-time by ordinary smartphones. It has been achieved with the help of deep learning. I have collected the required dataset, built, and optimized an object detector. After that, I have created an Android application extensively using smartphone features (camera handling, database creation, handling media files, internet communication, GPU capabilities). Finally, I have built a server application providing the necessary API for mobile clients to handle accounts and find and report suspicious vehicles. Since I had not dealt with object detection before, this project was an excellent opportunity to catch up.

I learned a lot while working on this task. To understand object detection, I needed to learn the theoretical background, and I liked that a novel idea could be tried out in practice right away. It was interesting to see how open the community is in the field of machine learning. I think the attitude of developers/researchers to share their results with each other contributes significantly to the current development momentum. I have had previous experience with the Android platform. During this project, I enjoyed gaining insight into the system's more resonant operation (memory management, performance aspects).

The resulting system could be further developed in numerous ways. Training a custom text recognizer and implementing object tracking would speed up the recognizer pipeline. In the detector case, even other combinations could have been tried (e.g., bigger batch size using other hardware resources). I would also develop the server application further to scale well for many user requests.

Overall, I enjoyed working on this project. I feel like I have managed to accomplish the pre-set goals and showcase the Android platform and deep learning frameworks' potential through a novel, community-based, stolen vehicle detection system.

7. Acknowledgements

I would like to thank my supervisor, Dániel Pásztor, for his valuable guidance and advice throughout the project. I would also like to thank my colleague, István Boros, for introducing me to the topic of object detection.

References

[1] BankMyCell. 2020. 1 Billion More Phones Than People In The World. Bankmycell. [online] Available at: <https://www.bankmycell.com/blog/how-many-phones-are-in-the-world#1579705085743-b3697bdb-9a8f> [Accessed 25 October 2020].

[2] Python programming language. 2020. Python.Org. [online] Available at: <https://www.python.org/about/> [Accessed 19 October 2020].

[3] Kotlin programming language. 2020. [online] Available at: <https://kotlinlang.org/> [Accessed 19 October 2020].

[4] Ktor Framework. 2020. Ktor: Build Asynchronous Servers And Clients In Kotlin. [online] Available at: <https://ktor.io/> [Accessed 11 October 2020].

[5] NVIDIA Tesla v100 datasheet. 2020. [online] Available at: <https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf> [Accessed 18 October 2020].

[6] Chainer. 2020. Define-By-Run - Chainer Documentation. [online] Available at: <https://docs.chainer.org/en/stable/guides/define_by_run.html> [Accessed 18 October 2020].

[7] TensorFlow. 2020. Eager Execution | Tensorflow Core. [online] Available at: <https://www.tensorflow.org/guide/eager> [Accessed 18 October 2020].

[8] Google Developers. 2020. Protocol Buffers | Google Developers. [online] Available at: <https://developers.google.com/protocol-buffers> [Accessed 21 October 2020].

[9] Google. 2020. Flatbuffers. [online] Available at: <https://google.github.io/flatbuffers/> [Accessed 21 October 2020].

[10] Medium. 2020. JSON Vs Protocol Buffers Vs Flatbuffers. [online] Available at: <https://codeburst.io/json-vs-protocol-buffers-vs-flatbuffers-a4247f8bda6f> [Accessed 21 October 2020].

[11] TensorFlow. 2020. Post-Training Quantization | Tensorflow Lite. [online] Available at: <https://www.tensorflow.org/lite/performance/post_training_quantization> [Accessed 18 October 2020].

[12] GitHub. 2020. Tensorflow Object Detection API. [online] Available at: <https://github.com/tensorflow/models/tree/master/research/object_detection> [Accessed 18 October 2020].

[13] Medium. 2020. Vehicle Detection With HOG And Linear SVM. [online] Available at: <https://medium.com/@mithi/vehicles-tracking-with-hog-and-linear-svm-c9f27eaf521a> [Accessed 11 October 2020].

[14] Zhao, Z., Zheng, P., Xu, S. and Wu, X., 2019. Object Detection with Deep Learning: A Review. [online] p.17. Available at: <https://arxiv.org/pdf/1807.05511.pdf> [Accessed 11 October 2020].

[15] Cocodataset.org. 2020. COCO - Common Objects In Context. [online] Available at: <https://cocodataset.org/> [Accessed 11 October 2020].

[16] Storage.googleapis.com. 2020. Open Images V6. [online] Available at: <https://storage.googleapis.com/openimages/web/index.html> [Accessed 11 October 2020].

[17] TensorFlow. 2020. Tfrecord And Tf.Train.Example | Tensorflow Core. [online] Available at: <https://www.tensorflow.org/tutorials/load_data/tfrecord> [Accessed 11 October 2020].

[18] PASCAL VOC. 2020. The PASCAL VOC Challenge. [online] Available at: <https://homepages.inf.ed.ac.uk/ckiw/postscript/ijcv_voc09.pdf> [Accessed 17 October 2020].

[19] COCO Detection. 2020. COCO - Common Objects In Context. [online] Available at: <https://cocodataset.org/#detection-2020> [Accessed 17 October 2020].

[20] Google Open Images Dataset Challenge. 2020. Overview Of Open Images Challenge. [online] Available at: <https://storage.googleapis.com/openimages/web/challenge_overview.html#object_detection> [Accessed 17 October 2020].

[21] Evaluation protocols. 2020. Tensorflow/Models. [online] Available at: <https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/evaluation_protocols.md> [Accessed 17 October 2020].

[22] Medium. 2020. mAP (Mean Average Precision) For Object Detection. [online] Available at: <https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173> [Accessed 17 October 2020].

[23] Nlp.stanford.edu. 2020. Evaluation Of Ranked Retrieval Results. [online] Available at: <https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-ranked-retrieval-results-1.html> [Accessed 17 October 2020].

[24] Google Developers. 2020. Validation Set: Another Partition | Machine Learning Crash Course. [online] Available at: <https://developers.google.com/machine-learning/crash-course/validation/another-partition> [Accessed 18 October 2020].

[25] Medium. 2020. Huber Error| Loss Functions. [online] Available at: <https://medium.com/@gobiviswaml/huber-error-loss-functions-3f2ac015cd45> [Accessed 22 October 2020].

[26] TensorFlow. 2020. Tfa.Losses.Sigmoidfocalcrossentropy | Tensorflow Addons. [online] Available at: <https://www.tensorflow.org/addons/api_docs/python/tfa/losses/SigmoidFocalCrossEntropy> [Accessed 22 October 2020].

[27] Yi Lin, T., Goyal, P., Girshick, R., He, K. and Dollár, P., 2018. Focal Loss for Dense Object Detection. [online] Available at: <https://arxiv.org/pdf/1708.02002.pdf> [Accessed 22 October 2020].

[28] Girshick, R., Donahue, J., Darrell, T. and Malik, J., 2014. Rich Feature Hierarchies For Accurate Object Detection And Semantic Segmentation. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1311.2524.pdf> [Accessed 22 October 2020].

[29] Girshick, R., 2015. Fast R-CNN. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1504.08083.pdf> [Accessed 22 October 2020].

[30] Ren, S., He, K., Girshick, R. and Sun, J., 2016. Faster R-CNN: Towards Real-Time Object Detection With Region Proposal Networks. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1506.01497.pdf> [Accessed 22 October 2020].

[31] Medium. 2020. R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms. [online] Available at: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e> [Accessed 22 October 2020].

[32] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A., 2016. You Only Look Once: Unified, Real-Time Object Detection. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1506.02640.pdf> [Accessed 23 October 2020].

[33] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Yang Fu, C. and C. Berg, A., 2016. SSD: Single Shot Multibox Detector. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1512.02325.pdf> [Accessed 23 October 2020].

[34] Duan, K., Bai, S., Xie, L., Qi, H., Huang, Q. and Tian, Q., 2019. Centernet: Keypoint Triplets For Object Detection. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1904.08189.pdf> [Accessed 23 October 2020].

[35] He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep Residual Learning For Image Recognition. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1512.03385v1.pdf> [Accessed 24 October 2020].

[36] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L., 2019. Mobilenetv2: Inverted Residuals And Linear Bottlenecks. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1801.04381.pdf> [Accessed 24 October 2020].

[37] Tan, M., Pang, R. and V. Le, Q., 2020. Efficientdet: Scalable And Efficient Object Detection. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1911.09070.pdf> [Accessed 24 October 2020].

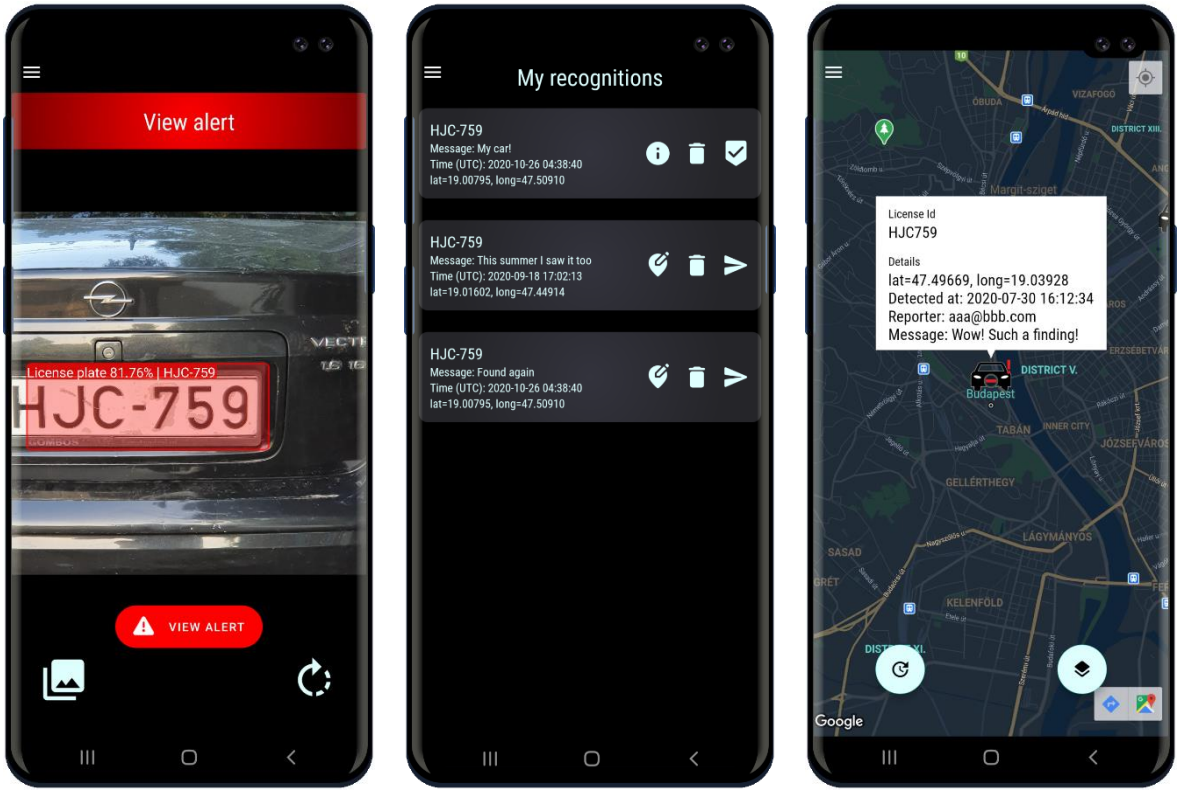
[38] TensorFlow. 2020. Tensorflow Lite Converter. [online] Available at: <https://www.tensorflow.org/lite/convert> [Accessed 25 October 2020].

[39] Ruder, S., 2017. An Overview Of Gradient Descent Optimization Algorithms. [online] Arxiv.org. Available at: <https://arxiv.org/pdf/1609.04747.pdf> [Accessed 25 October 2020].

[40] Hinton, G., Srivastava, N. and Swersky, K., 2014. Neural Networks For Machine Learning: Lecture 6. [online] Cs.toronto.edu. Available at: <http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf> [Accessed 27 October 2020].

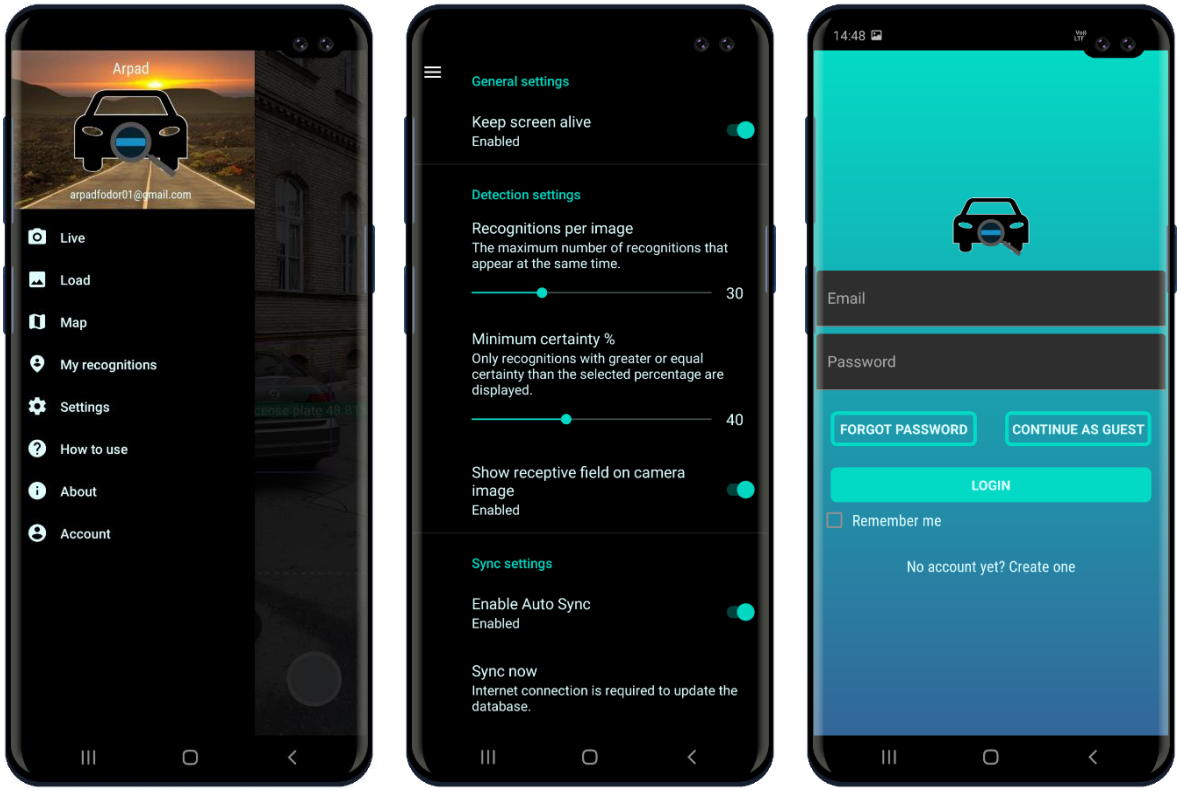
Appendix

Appendix A

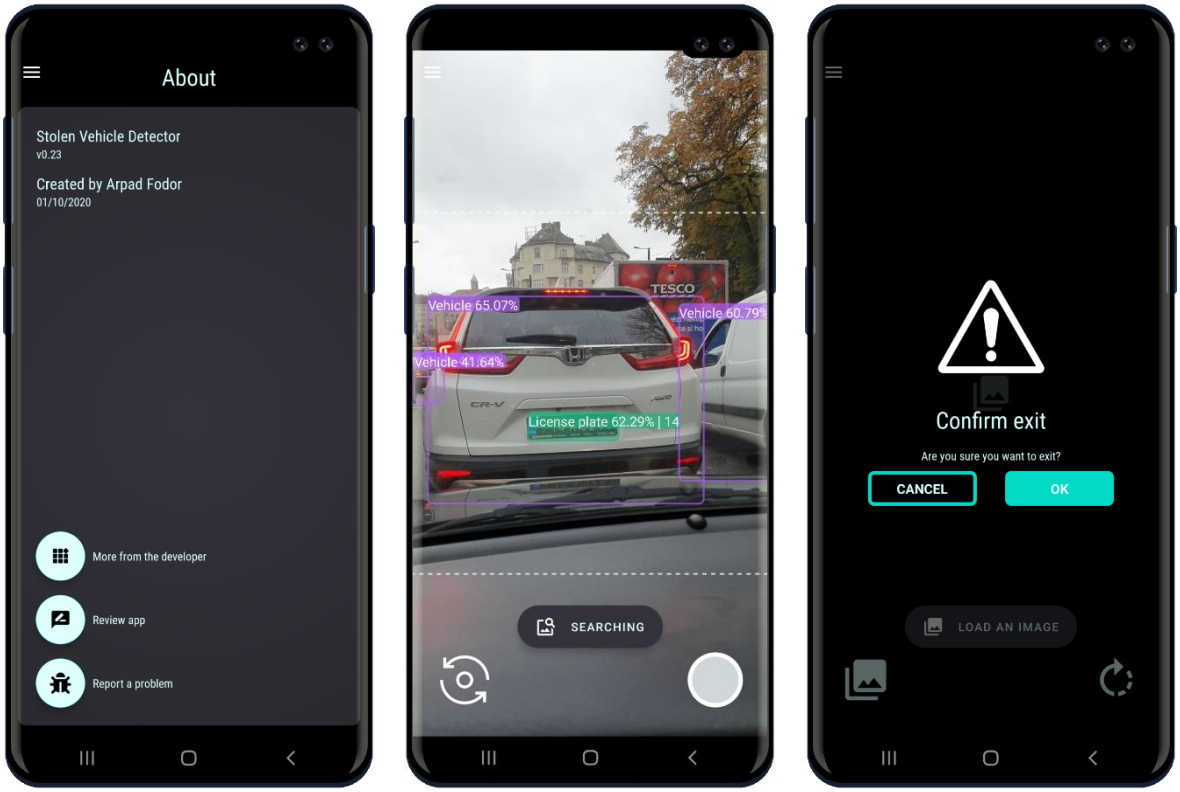


A loaded image (left), list of recognitions with sent and pending items (middle), vehicle information on the map (right)

List of recognitions (landscape device orientation)

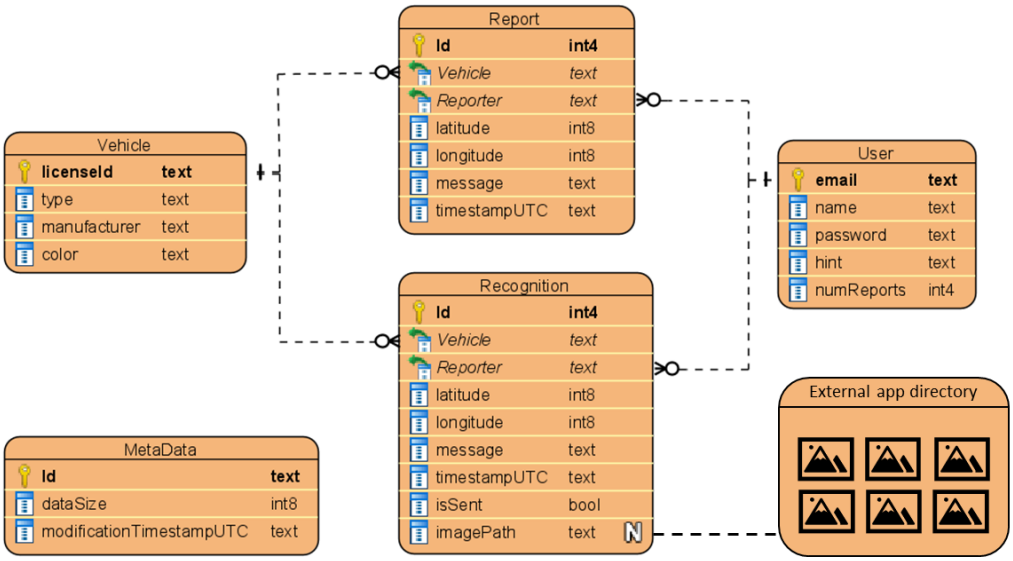


Menu (left), settings screen (middle), login screen (right)



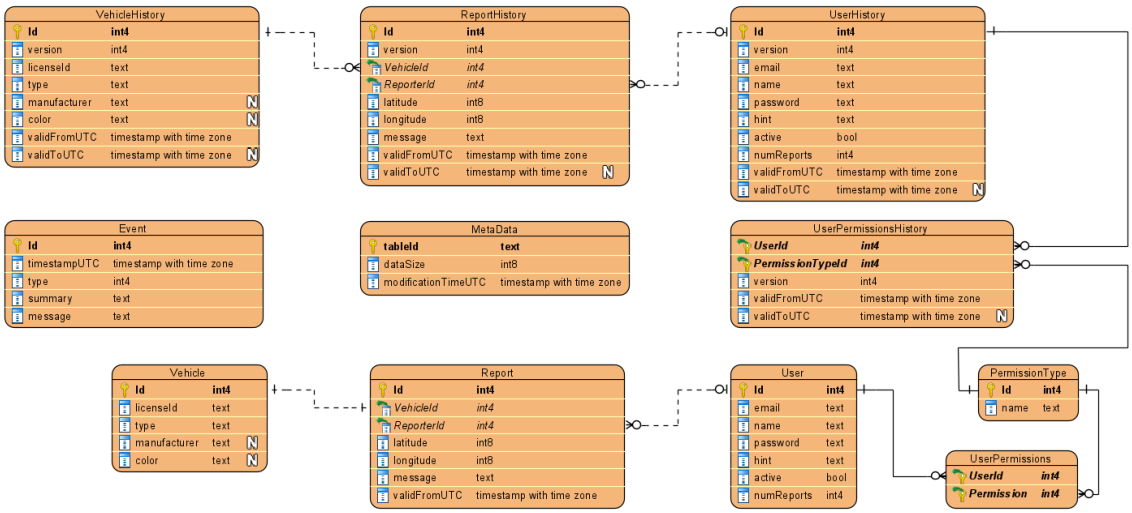
About content (left), live feed without stolen cars (middle), exit screen (right)

Appendix B



Android application database schema

Appendix C



Server data model schema

Appendix D

												Loss function			COCO metrics																		Loss			
															Precision						Recall																
model	ckpt size	input	inference [ms]	batch	step	total imgs fed	optimizer	learning rate	pre-trained?	freeze	duration	total loss	class loss	loc loss	mAP	mAP@.50IOU	mAP@.75IOU	mAP (small)	mAP (medium)	mAP (large)	AR@1	AR@10	AR@100	AR@100 (small)	AR@100 (med)	AR@100 (large)	localization	classification	regularization	total						
ssd_resnet50_v1_fpn_640x640_coco17_tpu-8	241.68 MB	640 x 640	46	8	20000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	3h 54m	0.444	0.1891	0.1335	0.389	0.678	0.404	0.116	0.394	0.531	0.294	0.517	0.565	0.257	0.585	0.709	0.157732	0.369674	0.121819	0.649225				
efficientdet_d0_coco17_tpu-32	42.51 MB	512 x 512	39	16	10000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	2h 19m	0.2153	0.1787	0.005977	0.314	0.581	0.302	0.019	0.321	0.492	0.252	0.424	0.462	0.113	0.484	0.656	0.008246	0.780476	0.30632	0.819354				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	20000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	-	8h 38m	0.4085	0.1888	0.1103	0.39	0.681	0.396	0.114	0.399	0.542	0.297	0.508	0.544	0.215	0.578	0.697	0.157866	0.751188	0.042879	0.951934				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	1	160000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	8h 20m	0.4226	0.276	0.09588	0.291	0.578	0.262	0.1	0.33	0.373	0.256	0.453	0.497	0.204	0.515	0.618	0.222161	0.557912	0.050744	0.830817				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	4	40000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	6h 4m	0.4167	0.2303	0.1308	0.388	0.681	0.39	0.119	0.398	0.541	0.298	0.506	0.543	0.226	0.571	0.693	0.155637	0.657971	0.055557	0.869164				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	20000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	5h 46m	0.3157	0.1555	0.107	0.391	0.671	0.4	0.119	0.396	0.548	0.298	0.51	0.545	0.229	0.57	0.698	0.161812	0.957659	0.053175	1.172645				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	18000	144000	RMS	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	5h 50m	1.048	0.5569	0.4614	0.155	0.333	0.13	0.007	0.154	0.213	0.14	0.288	0.331	0.006	0.354	0.512	0.514635	0.827527	0.029544	1.371705				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	20000	160000	RMS	lr0.002, cos_decay, warm:1000/0.0002	COCO	extractor	5h 55m	0.4009	0.2377	0.1337	0.361	0.643	0.374	0.079	0.394	0.512	0.278	0.487	0.523	0.182	0.577	0.673	0.184088	0.706238	0.029537	0.919863				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	16000	128000	Adam	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	2h 1m	11.95	4.29	0.529	0.011	0.031	0.004	0	0	0.018	0.029	0.04	0.058	0	0.001	0.132	0.635919	8.837397	7.18506	16.65839				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	10000	80000	Adam	lr0.004, cos_decay, warm:1000/0.002	COCO	extractor	2h 50m	0.3987	0.2358	0.114	0.373	0.656	0.371	0.09	0.395	0.527	0.288	0.491	0.531	0.203	0.568	0.688	0.164956	0.570882	0.048987	0.784826				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	20000	160000	Adam	lr0.002, cos_decay, warm:1000/0.0002	COCO	extractor	7h 5m	0.349	0.2113	0.09381	0.392	0.682	0.41	0.103	0.412	0.546	0.291	0.51	0.545	0.219	0.576	0.697	0.155675	0.573626	0.043904	0.773205				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	30000	240000	Adam	lr0.002, cos_decay, warm:1000/0.0002	COCO	extractor	9h 3m	0.3292	0.2097	0.09464	0.391	0.685	0.404	0.108	0.404	0.543	0.295	0.506	0.541	0.212	0.575	0.688	0.154645	0.623379	0.049006	0.82703				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	40000	320000	Adam	lr0.002, cos_decay, warm:1000/0.0002	COCO	extractor	14h 17m	0.3444	0.2054	0.09166	0.387	0.678	0.398	0.11	0.398	0.541	0.293	0.503	0.541	0.219	0.57	0.69	0.154894	0.62614	0.047373	0.828407				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	30000	240000	Adam	lr0.0002, cos_decay, warm:1000/0.00002	COCO	extractor	8h 41m	0.2265	0.1199	0.06424	0.383	0.669	0.38	0.116	0.39	0.528	0.291	0.498	0.532	0.225	0.552	0.678	0.169966	0.990891	0.03271	1.193567				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	40000	320000	Adam	lr0.0002, cos_decay, warm:1000/0.00002	COCO	extractor	14h 13m	0.2466	0.1424	0.07168	0.38	0.666	0.381	0.115	0.387	0.527	0.291	0.498	0.529	0.218	0.549	0.677	0.170887	1.130547	0.032567	1.334001				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	20000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	7h 24m	0.2989	0.1727	0.07949	0.392	0.677	0.4	0.116	0.389	0.553	0.294	0.515	0.548	0.224	0.567	0.707	0.154253	0.758655	0.046646	0.959554				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	30000	240000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	8h 50m	0.29	0.1482	0.06837	0.393	0.686	0.391	0.119	0.392	0.545	0.299	0.504	0.539	0.235	0.554	0.69	0.162675	0.959799	0.058981	1.181456				
efficientdet_d1_coco17_tpu-32	63.36 MB	640 x 640	54	8	40000	320000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	14h 7m	0.3002	0.1477	0.0944	0.39	0.685	0.385	0.12	0.39	0.541	0.298	0.503	0.538	0.236	0.555	0.688	0.162432	1.083678	0.058113	1.304222				
ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8	19.78 MB	320 x 320	22	16	10000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	3h 55m	0.5245	0.1921	0.1986	0.278	0.504	0.269	0.007	0.248	0.492	0.234	0.396	0.438	0.08	0.421	0.668	0.237216	0.47648	0.133786	0.847482				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	1	160000	160000	Momentum	lr0.008, cos_decay, warm:1000/0.002	COCO	extractor	5h 57m	0.7242	0.3947	0.202	0.261	0.535	0.225	0.09	0.327	0.301	0.234	0.439	0.482	0.208	0.507	0.579	0.262299	0.539022	0.127919	0.92924				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	4	40000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	4h 41m	0.4671	0.2499	0.1185	0.377	0.677	0.381	0.116	0.385	0.507	0.285	0.498	0.536	0.246	0.554	0.675	0.170977	0.496225	0.098746	0.765949				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	8	20000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	4h 8m	0.4721	0.2412	0.1133	0.379	0.678	0.386	0.115	0.386	0.517	0.286	0.497	0.533	0.239	0.558	0.663	0.170381	0.518244	0.11756	0.806184				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	16	10000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	4h 8m	0.406	0.1616	0.1303	0.372	0.666	0.378	0.101	0.381	0.508	0.288	0.5	0.548	0.242	0.577	0.683	0.170536	0.395023	0.131221	0.696778				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	24	10000	240000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	extractor	7h 7m	0.3873	0.1658	0.09127	0.368	0.669	0.376	0.108	0.367	0.512	0.287	0.485	0.514	0.225	0.534	0.654	0.181506	0.69993	0.130229	1.011663				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	24	10000	240000	Momentum	lr0.08, cos_decay, warm:1000/0.002	COCO	-	6h 45m	0.3774	0.1405	0.09069	0.373	0.674	0.373	0.112	0.376	0.516	0.287	0.486	0.519	0.235	0.541	0.65	0.180529	0.675216	0.130298	0.986043				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	8	20000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	-	extractor	4h	0.6277	0.3266	0.2518	0.341	0.627	0.337	0.081	0.358	0.476	0.269	0.46	0.497	0.183	0.529	0.638	0.189032	0.522684	0.049284	0.761				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	16	10000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	-	extractor	4h 1m	0.4735	0.2646	0.1654	0.336	0.621	0.327	0.081	0.354	0.475	0.27	0.452	0.488	0.174	0.517	0.641	0.194693	0.537193	0.043559	0.775446				
ssd_mobilenet_v2_fpnlite_640x640_coco17_tpu-8	19.78 MB	640 x 640	39	16	10000	160000	Momentum	lr0.08, cos_decay, warm:1000/0.002	-	-	3h 53m	0.5482	0.3061	0.1984	0.337	0.62	0.333	0.076	0.358	0.475	0.27	0.455	0.493	0.196	0.527	0.638	0.195156	0.537025	0.043644	0.775826				

Summary of all training configurations with results

Appendix E

The source code of the whole project (detector training, Android application, server application) can be found in the following repository: https://github.com/arpadfodor/StolenVehicleDetector