Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Deep learning-based real-time vehicle identification on Android

Master's Thesis

Author
Árpád Fodor

Advisor
Dániel Pásztor

December 19, 2021

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Fodor Árpád*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Stockholm, 2021. december 19.

Fodor Árpád
hallgató

Kivonat

Az elmúlt években a deep learning áttörést hozott különféle alkalmazási területekben, például a számítógépes látás és a természetes nyelvfeldolgozás terén. Ez a gépi tanulás jelenleg is aktívan kutatott területe, amely úgy tűnik, hatékony megoldásokat kínál számítógéppel nehezen algoritmizálható feladatokra. Ezzel párhuzamosan az okostelefonok robbanásszerű sebességgel fejlődnek. Az ARM CPU-k közelmúltbeli fejlődése, valamint a GPU-k megjelenése az asztali PC-khez hasonló számítási teljesítményt tesznek lehetővé.

A járműazonosítás gyakori feladat, amelyet gyakran az ALPR (automatikus rendszámtábla-felismerés) segítségével hajtanak végre. Ezt a technológiát használják az úthasználati engedélyek ellenőrzésére vagy lopott járművek keresésére. Az ilyen rendszerek már régóta léteznek, de fejlesztésük általában drága, működésükhöz egyedi hardver szükséges. A hagyományos ALPR rendszerek kézzel történő tulajdonság kinyerésre támaszkodnak, és független feldolgozási lépéseket tartalmaznak. Egy deep learning algoritmus képes magától megtanulni a szükséges tulajdonságok kinyerését; így robusztusabb megoldást nyújthat még változatos rendszámok felismerésében is.

Ebben a munkában egy valós idejű járműazonosító rendszert hozok létre. Először megvizsgálom az ALPR különböző megközelítéseit, majd javaslatot teszek egy valós idejű felhasználásra kialakított pipeline-ra. Ezután létrehozok deep learning algoritmusokat a teljes járműazonosítási folyamatának megvalósításához. Ezt követően bemutatom az elkészített Android alkalmazást, amely futtatja a kifejlesztett algoritmusokat. Ezután bemutatom a szerver alkalmazást is, melyen keresztül a felhasználók frissítése és a járművek bejelentése történik. Végül ismertetem az elkészített rendszer alkalmazhatóságát és korlátait.

Abstract

In recent years, deep learning has shown breakthroughs in various applications, such as Computer Vision and Natural Language Processing. It is an actively researched area of machine learning that seems to provide effective solutions to tasks that have been difficult to solve with machines. Concurrently, smartphones are evolving at an explosive rate. The recent improvement of ARM CPUs and the advent of GPUs provide computing power comparable to desktop PCs.

Vehicle identification is a common task that is often accomplished through ALPR (Automatic License Plate Recognition). This technology is used to check for road usage permits or to look for stolen vehicles. Such systems have been around for a long time, but they are usually expensive to develop and require unique hardware to operate. Traditional ALPR relies on hand-crafted feature extraction and contains independent processing steps. Deep learning can learn feature extraction on its own; thus, it can provide a more robust solution even for recognizing various license plates.

In this work, I create a real-time vehicle identification system. First, I examine different ALPR approaches and propose a pipeline with real-time usability in mind. Then, I create and train deep learning algorithms for the complete vehicle identification process. Later, I present the Android application running the developed pipeline. After that, I also present the server application from which users can update themselves and report vehicles. Finally, I explain the applicability and the limitations of the prepared system.

Chapter 1

Introduction

In recent years, deep learning has shown breakthroughs in various applications, such as Computer Vision and Natural Language Processing. It is an actively researched area of machine learning that seems to provide effective solutions to tasks that have been difficult to solve with machines. Concurrently, smartphones are evolving at an explosive rate. The recent improvement of ARM CPUs and the advent of GPUs provide computing power comparable to desktop PCs.

Vehicle identification is a common task that is often accomplished through ALPR (Automatic License Plate Recognition). This technology is used to check for road usage permits or to look for stolen vehicles. Such systems have been around for a long time, but they are usually expensive to develop and require unique hardware to operate. Traditional ALPR relies on hand-crafted feature extraction and contains independent processing steps. Deep learning can learn feature extraction on its own; thus, it can provide a more robust solution even for recognizing various license plates.

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

In this work, I describe the main theory related to deep learning-based object detection and sequential processing. I assume knowledge of general deep learning concepts, like backpropagation, loss and activation functions, vanishing/exploding gradient problems, neural networks, convolutional models. For a brief overview, please look at chapter 4 of the author's previous work[19], or for more detailed information, I recommend chapters 6, 7, 8, and 9 of the following literature[27].

The structure of this work is as follows:

- In *Chapter 2*, I present the technologies I used to create deep learning algorithms and the system running them.
- *Chapter 3* discusses the theoretical background of sequential data processing with neural networks.
- In *Chapter 4*, I explain the fundamental ideas behind deep learning-based object detection.
- I present a high-level overview and discuss the general tasks of automatic license plate recognition in *Chapter 5*.
- *Chapter 6* summarizes the work done to implement a vehicle- and license plate detection algorithm.
- *Chapter 7* focuses on my work related to optical character recognition. I propose multiple solutions to solve the same problem, then compare these models and draw conclusions.
- In *Chapter 8*, I overview, explain the most important design decisions, and describe the limitations of the complete system.
- Finally, in *Chapter 9*, I summarize the work done and propose further development ideas.

Chapter 2

Technologies

This chapter presents the technologies used to build the deep learning models and create client and server applications.

2.1 Deep learning

I used the Python[21] programming language 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 in Python. However, two libraries stand out from these: PyTorch and TensorFlow. In this section, I mainly describe the technologies used in this work. My choice was driven primarily by the desire for Android interoperability, which is currently not widely supported by other tools than the selected ones.

2.1.1 TensorFlow

TensorFlow[8] 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. In 2019, TensorFlow 2.0 was released with Eager mode, which broke up with the former "define-and-run" 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[77], where operations are immediately evaluated without building graphs.

TensorFlow uses Google's protobuf[75] 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[41]. In turn, protobuf files are not as human-readable as opposed to JSON or XML.

TensorFlow Lite is a lightweight, speed, or storage optimized format to deploy models on smartphones and IoT devices. Trained models can be transformed into this format with the TFLite converter. TFLite uses the Flat Buffer[3] 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. On the other

hand, Flat Buffer encoding is more complicated than in JSON/protobuf formats - for this reason, TensorFlow does not use it. This is also why TFLite models could not be trained until TensorFlow 2.7[9].

During TFLite conversion, it can be selected whether it is required to minimize model size further with a slight model accuracy trade-off or not. These quantization options are used to achieve further performance gains ($2-3\times$ faster inference, $2-4\times$ smaller networks). One technique is full integer quantization[79] in which all the model maths are int8 based instead of the original float32.

The TensorBoard component provides a helpful 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, and many more. I used this tool to monitor the training/evaluation process.

2.1.2 TensorFlow Object Detection API

TensorFlow's Object Detection API[10] 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, RetinaNet, or CenterNet and two-staged R-CNN variants. However, other types like the YOLO variants are missing.

There are many parameters to tweak during model creation. The API has introduced a configuration language that can fine-tune the training pipeline - e.g., data source, backbone, pre-processing steps, optimization algorithms, and input/output directories. The configuration parameters are described in the corresponding .proto files. The configuration handles architecture level modifications (like different backbone CNNs to use). However, the number of options here is quite limited, customization is cumbersome, and the pipeline is easy to break, which in my opinion, is a big downside.

During this work, support for TF2 has arrived, making it possible to use Keras models in detector architectures. The library has actively evolved in the past years. Although reliability issues often arise, rapid implementations of the latest research (e.g., FPN, CenterNet) helped me understand novel concepts.

2.1.3 PyTorch

PyTorch[6] is an open-source machine learning framework primarily developed by Meta/-Facebook AI Research lab. In the previous years, the community started to use it more extensively, and numerous research implementations were written using it. PyTorch offers a pythonic and object-oriented approach, while TensorFlow has several options to choose from. In my opinion, developing in PyTorch requires more coding than using TensorFlow with Keras. However, it feels more stable and less fragile. The framework also has good interoperability with the Open Neural Network Exchange (ONNX)[4] format, which aims to bridge the compatibility gap between different libraries and frameworks. The PyTorch Mobile module provides both iOS and Android deployment with quantization and optimization. However, it is in the beta stage at the time of this work.

After evaluating the frameworks, I decided to work with TensorFlow/Keras. My primary motivations were finding more documentation for Android, and I assumed it was easier to work with an already matured tool (TensorFlow Lite) than the more risky beta stage

PyTorch Mobile. As a side note: both frameworks also offer C++ APIs, which can also run on Android using the platform's Native Development Kit.

2.1.4 Environment

The environment in which I worked was Google Colaboratory Pro. I was using Nvidia Tesla V100 SXM2 GPUs with 16 GB memory most of the time. At the beginning of a project, a realm is created or loaded from Google Drive. A realm is where the project saves its checkpoints and training logs. Realms are the sandboxes of notebook instances, making it possible to run them parallel seamlessly. Two types of projects/sessions can be: single training or hyperparameter optimization. For the latter, I used Keras Tuner. An overview of the environment can be seen in Figure 2.1.

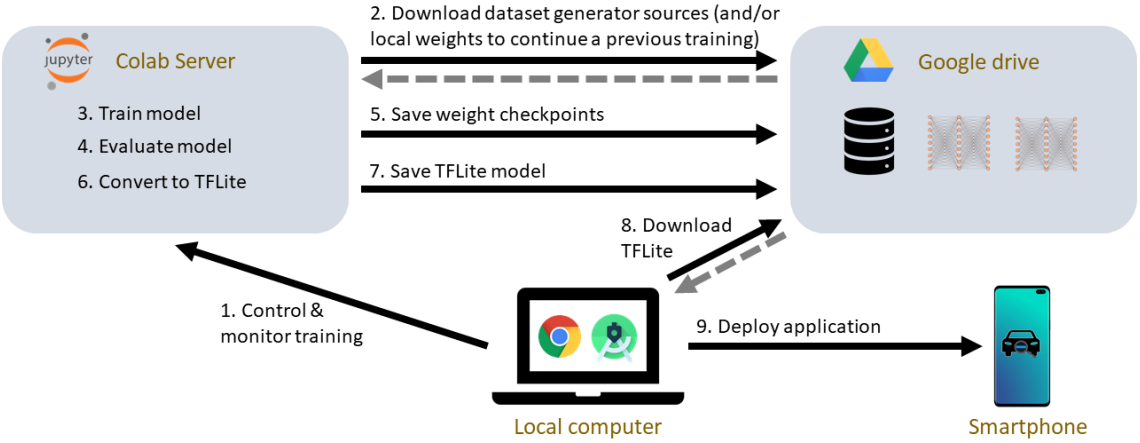

Figure 2.1: High-level overview of a session in the training environment.

Primarily, I monitored the training on TensorBoard, which shows a dashboard of the running session. After every epoch, the current model is evaluated. On these occasions, model checkpoints are saved (for early stopping and state preservation in case of a failure). After the training process, a standalone evaluation can be executed to verify the results and display the selected loss/protocol metrics.

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.

2.2 Application

I used the Kotlin programming language[20] for the frontend and the backend. Kotlin is a statically typed, cross-platform language with type inference. I used the JVM version, where the Kotlin standard library depends on the Java Class Library, but it can also compile to JavaScript. In addition to the object-oriented approach, Kotlin also contains functional programming tools.

2.2.1 Frontend

Android provides an extensive application development ecosystem. I used the AndroidX namespace elements, which replaced 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. The application extensively uses the CameraX API to manage device cameras. The ViewModel component moderates between the user interface and the business logic. The app contains a relational database implemented by the Room Persistence Library, which provides an abstraction layer over SQLite to more robust database access. To boost user experience, I decided to use a text-to-speech engine to read aloud alerts (useful if a user wants notifications while driving or wants to hear how to use tips).

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).

2.2.2 Backend

I used Ktor[37] for the backend. It is an open-source, asynchronous framework for creating microservices and web applications. JSON data handling was implemented with the Gson library. I tested the server API with Postman, and for web scraping, I used ParseHub.

Chapter 3

Sequential modeling

This chapter focuses on the theoretical background of processing sequential data. As relatively simple deep learning-based solutions are surprisingly good with sequential data, I concentrate on neural network-based approaches.

Sequential modeling is a powerful way to solve numerous tasks, like video processing, time series prediction, or natural language processing. Many tasks can be modeled sequentially. The key is to project the domain problem to the proper abstraction. The categorization is partly based on source [55]. There are different types of sequential tasks:

- Vanilla approach processes the input in a *single-input-single-output* (SISO) manner: for example, one frame of a video is classified independently of others. This is the case when the model has a single input image and makes classification.
- In a *single-input-multiple-output* (SIMO) model, there is a single input and multiple possible outputs. For example, when an image contains multiple characters, the input is a single image, and the output is the sequence of characters in it.
- *multiple-input-single-output* (MISO) models receive multiple inputs and output a single prediction. For example, the inputs are words of a textual review, and the prediction is the writer's sentiment.
- *multiple-input-multiple-output* (MIMO) models receive multiple inputs and predict multiple outputs. These approaches can be divided into further subcategories:
  - Some models receive multiple inputs and immediately output the prediction based on that. In video classification, a model usually receives multiple frames and outputs the class for each time frame, but these are predicted simultaneously, depending on each other.
  - In another configuration, popular in machine translation, models receive an input text and predict the current translated text on the fly, based on the current input. Here, input and output sequences are shifted relative to each other (some input characters are needed for the first prediction). For example, Google Translate is a model of this kind.

A summary of the different tasks is illustrated by Figure 3.1. Sequential data can be processed with different network types. In the following sections, I present these variants.

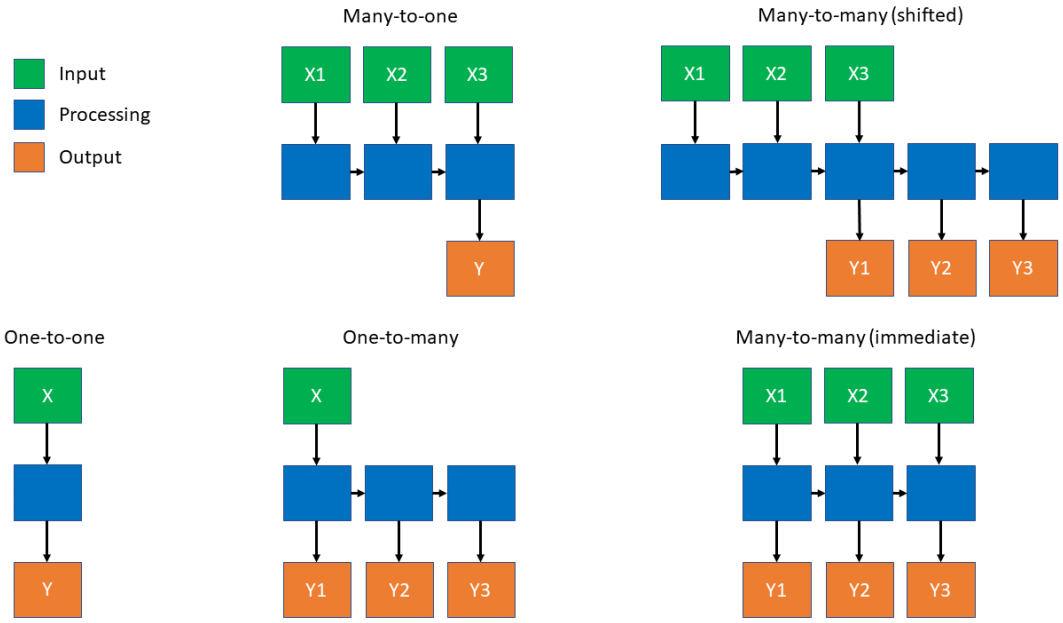

Figure 3.1: The different types of sequential tasks.

3.1 One-dimensional convolution

One-dimensional convolution networks can solve sequential problems by applying convolution to the serial input. These approaches are popular for problems like time-series analysis or audio input processing.

When applying 1D convolution, the corresponding layer only sees the input in slices of the same width as the convolutional kernel. Pooling or strided convolution can be applied to reduce the length of the sequence, thus increasing the visible part of the input toward the deeper convolutional layers. Another approach uses *dilated convolution*, where the kernel is inflated by inserting holes between its elements (Figure 3.2). Dilation rate defines how much the kernel is widened. WaveNet[81] is a successful dilated convolution-based MIMO model for sample-level audio synthesis. Its main data processing idea is illstrated by Figure 3.3.

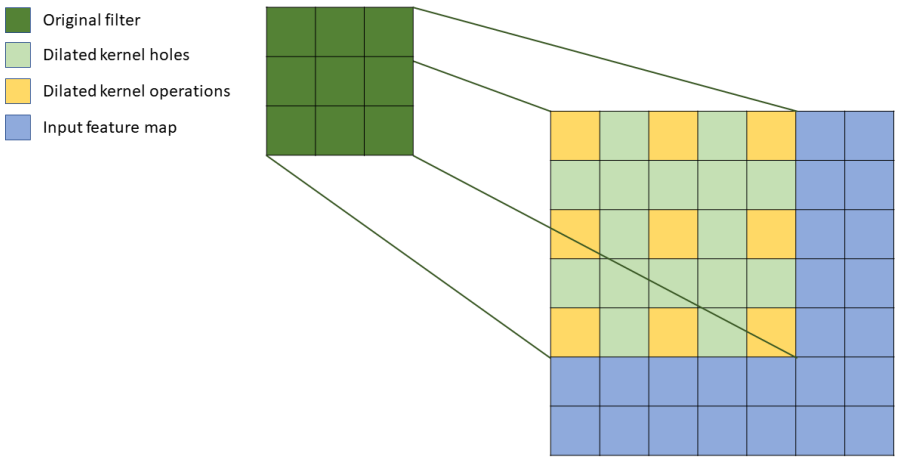

Figure 3.2: Two-dimensional dilated convolution.

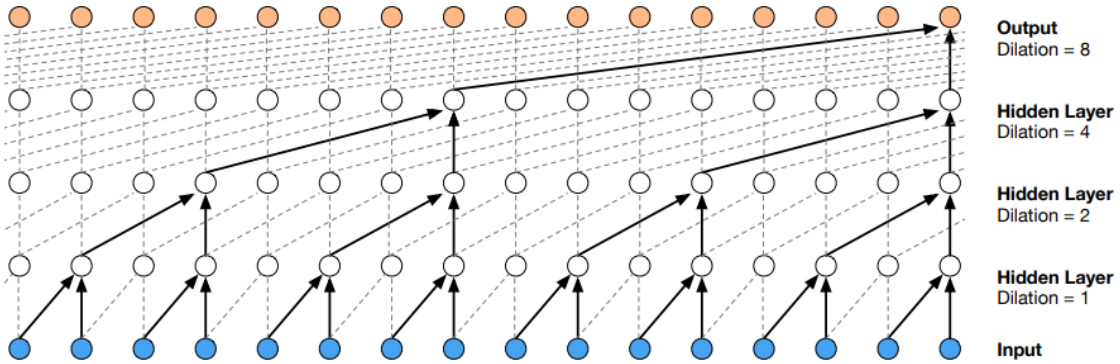

Figure 3.3: Stack of one-dimensional dilated convolutional layers in WaveNet. Source: [81]

These models use solely convolutional layers, thus being relatively simple. While training, the simple backpropagation algorithm is used, and their runtime is only constrained by the convolution operation: in the case of two dimensions, if the input size is $n \times n$ and the kernel size is $m \times m$, the runtime is $\mathcal{O}(n^2 \times m^2)$. However, these models lack one crucial element to process long sequential inputs: memory.

3.2 Recurrent neural networks

Classical neural networks are directed acyclic graphs (DAG), which define computational nodes and their connections. *Recurrent neural networks* (RNN) are dynamic models containing recurrent links (directed loops). These connections can model an inner/hidden state (memory), which makes them useful when sequential data is processed, where later values depend on earlier ones. In every time step, the same function is called, meaning that the RNN's nodes and weights are the same. Figure 3.4 represents the inference steps of an RNN. These models do not require prior knowledge of the input data, and they are generally robust to noise[29]. The section below discusses how such a network is trained.

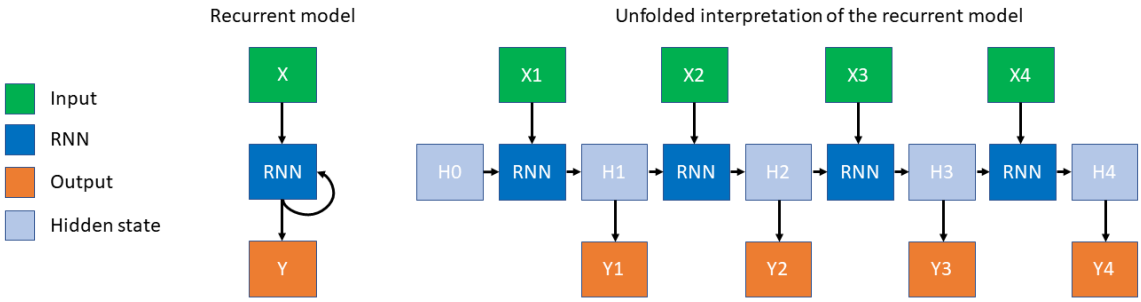

Figure 3.4: Unfolded interpretation of the RNN for a length-4 sequential input.

3.2.1 Backpropagation through time

In the classical backpropagation, a model input is first propagated forward the network, resulting in a prediction. When training the network, it is known what is expected from the model's output (ground truth). The difference between the prediction and the ground

truth is calculated by the loss function, which results in the network's error. The error is then propagated backward the network: for each connection, a gradient is calculated based on how much it contributed to the local error of the following node. When each link has this value, the algorithm updates each connection weight based on its gradient and the learning rate.

Backpropagation through time[55] (BPTT) works similarly. The difference is that the RNN is handled as an unfolded model. For each time step, the backpropagation algorithm is applied in a manner that the input of the *Nth* step is the output of the *N-1th* step (this wording is intentional, as we are moving backward in time). Gradient values are calculated for the whole network independently for each time step. When gradients are computed for a step, values are updated for each weight (Figure 3.5).

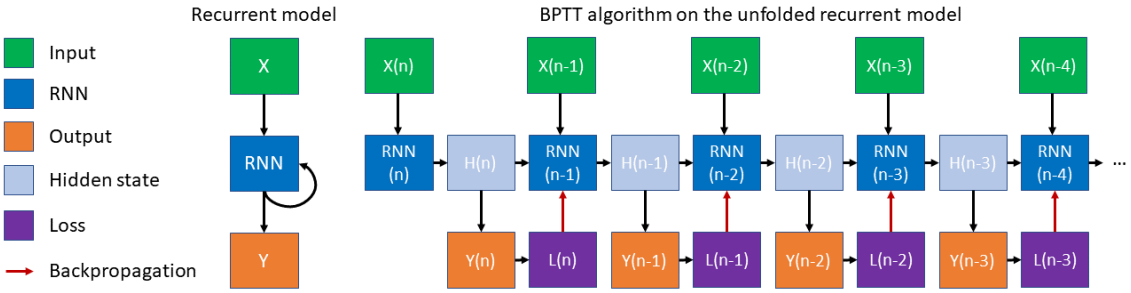

Figure 3.5: Operation of the BPTT algorithm on the unfolded RNN.

Usually, there is a truncation in the BPTT algorithm: the backward calculation only goes back to the last K steps on the network unfolded by time. Otherwise, in the case of long series, the algorithm would take a lot of computational resources, significantly slowing down the training process.

3.2.2 Limitations

Many time steps are needed to create long-term inner memory. During BPTT, the error propagates back to these time steps on the same model. It means that we multiply the model's weight matrix multiple times, which is prone to the following problems:

- When the weight matrix singular value is high, it leads to gradient exploding. The weight updates are too big, so the model cannot converge.
- When the weight matrix singular value is low, gradient vanishing happens. In this case, the gradients are too close to zero; weight updates are too small to improve the model.

For gradient exploding, possible solutions can be using saturating nonlinear activation functions or gradient clipping, where the values cannot go above a maximum. Skip connections can somehow compensate for the gradient vanishing problem, through which the error can flow freely back to the earlier time steps. However, these are not complete solutions.

3.2.3 Long short-term memory

Long short-term memory[33] (LSTM) is a type of RNN cell which uses a gated approach. It implements a short-term memory that can be long enough to handle longer sequences than simple RNNs can do. An LSTM cell has three inputs in each time step:

- C: The cell state is the interpretable, optimized output of the cell.
- H: Hidden state, which is the inner state of the LSTM block.
- X: Input tensor, the input from the earlier network layers (for example, a feature map created by convolutional layers which we want to process sequentially, from left to right).

There are multiple trainable gates within an LSTM cell, manipulating the signal flow and thus modeling what information is needed from a specific input given a state. These gates are trainable to model memory. The gates are the following:

- *forget*: Responsible for deleting certain things from the cell state based on the fused input of the hidden state and the input tensor. This gate was not part of the initial publication; it was added a few years later[23]. Forget gate allows the cell to reset its state and was crucial to the success of LSTM.
- *input*: Decides what items of the fused input should be written to the cleaned cell state.
- *gate*: Applies nonlinearity to the fused input written to the cell state. This is the only gate that uses the tanh activation function instead of the sigmoid. This gate projects the values between [-1, 1] to provide the necessary abstraction switching of the signal while maximizing the output range, retaining the saturating behavior. The other gates use sigmoid activation because they mask the signal to be retained (where the minimum is 0 and the maximum is 1).
- *output*: Masks the hidden state output derived from the cell state.

In an LSTM cell, there are only saturating nonlinear activation functions, which can handle the gradient exploding problem. There is no activation function between the input and the output cell state, only one multiplication (forget gate) and one addition. This means that the input signal can propagate to the output without severe decrease, and thus the gradients can backpropagate without downscaling. This path forms a gradient highway across the unfolded model (containing the same LSTM cell of each time step) during BPTT training. This path is essential in solving the gradient vanishing problem induced by the numerous time steps. The visual representation of an LSTM cell is illustrated by Figure 3.6.

LSTM is a relatively old approach from 1997, a mature standard tool for solving sequential problems. Over time, other solutions have emerged, such as the Gated Recurrent Unit[15] (GRU) from 2014, which similarly models memory, but with fewer parameters (three gates instead of four). I decided to present LSTM because it is easier to understand thanks to its well-separated elements.

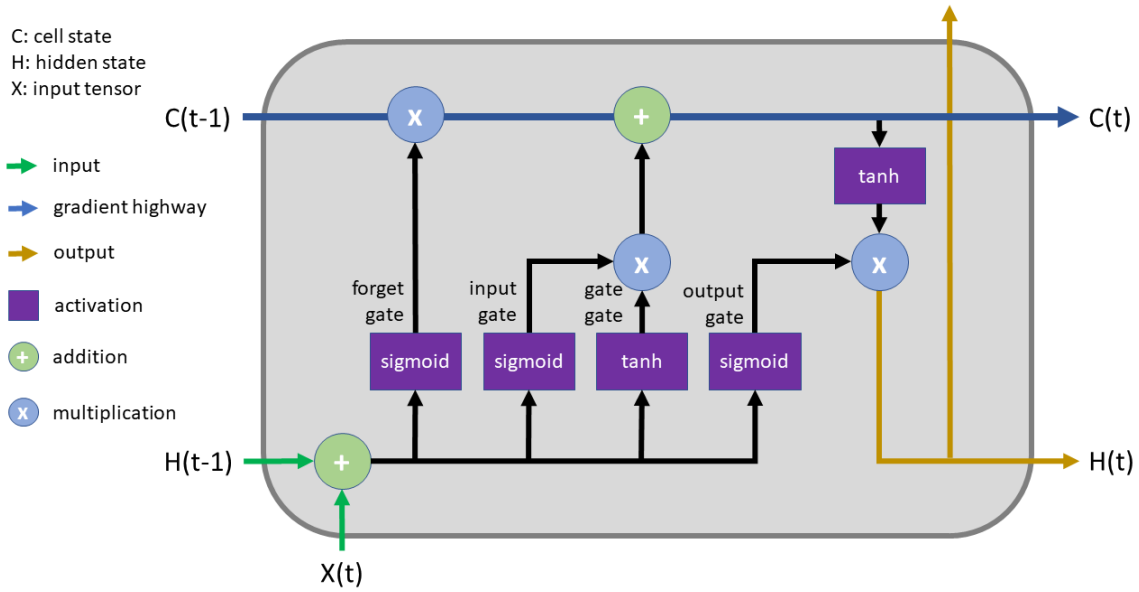

Figure 3.6: Inner structure of an LSTM cell.

3.3 Connectionist Temporal Classification

RNNs are good at sequence learning, but they require thoroughly segmented and labeled data, which is time-consuming. Traditional loss functions like mean squared error cannot work well without properly segmented data because they assume that each item in the sequence is independent. This leads to limited model applicability.

Connectionist Temporal Classification[29] is a loss function to handle sequential input, which is noisy and unsegmented. It assumes that the network output is a conditional probability distribution over the label sequences. This way, the loss function transforms the sequential problem into classification, where the model needs to select the most probable label for a given input sequence.

3.3.1 Example

I illustrate the function's applicability on an example with an input image in which we want to read a text. This example is partly based on [67]. Reading the text in the image can be modeled as a sequential problem, where we want to process the picture from left to right. Only the input and the ground truth text label are needed when training with CTC. We discard the information where the particular characters are. CTC tries the possible alignments of the ground truth label in the image and sums all the scores. This way, it does not matter where the text is in the picture - the algorithm could find a suitable label alignment when the match value is high. This value should be maximized to the alignments outputting the ground truth label.

Because CTC tries to find the optimal label alignment, a good character encoding is needed to help distinguish good and bad ones. The characters are separated by a blank token (marked as "|"), which separates the letters. This way, when a wide character is processed and triggers multiple predictions next to each other, we can merge them between the blank tokens to identify the actual letter. This also solves the problem of segregating the same characters next to each other. The encoding approach makes it possible to represent the

same text multiple ways, allowing different alignments to match. The neural network is trained to predict the output in this encoded format.

- The word "of" is recognized when the output is "|o|ff|", "|oooooo||||ffff||||", or "o|f".
- The word "off" is recognized when the output is "||oo|f||f", "|ooo|fff|f|", or "o|f|f".

To calculate the scores for an alignment, we have to consider all the possible paths and sum the ones producing the same decoded text. The added up probability is the overall score of the path. Each path text is then compared to the ground truth label, and the loss value is calculated weighted by the path probabilities. The loss function needs to be minimized. For that, CTC uses the negative sum of the log probabilities. The probability is maximized when this loss is minimized, so the function motivates the model to find good predictions.

The model outputs a matrix containing a score for each character at each time step. There are different approaches to decode this matrix. One method is called *best path decoding*, where:

- The algorithm takes the maximum prediction at each step, producing the most probable path.
- To decode the prediction, the same characters are removed next to each other, then the blank tokens are also removed. The remaining output is the predicted text.

This method is fast and usually good, but it is just an approximation. There are different decoding approaches where multiple paths can be considered (similarly when calculating the CTC loss for the different paths). One other algorithm is Word Beam Search[68], where the number of nodes to consider at each time step can be defined. It can make decoding more accurate but also scalable.

Chapter 4

Object detection

Computer vision is a scientific field that deals with how computers extract high-level information from digital images or videos. The input can be various (like a three-dimensional point cloud), but for the sake of simplicity, I assume they are two-dimensional images. In general, computer vision tries to automate tasks that the human visual system can do, and there are numerous ones:

- The easiest is *image classification*, where the input is an image, and the output is a simple class. But if the image contains multiple classes at the same time, we cannot find them with this approach.
- *Semantic segmentation* is a task where each input image pixel is classified, and the output is a mask containing pixel-level class predictions. But if there are multiple instances of the same class in the image, we cannot distinguish them.
- The solution to this problem can be *object segmentation*, where each object gets a different Id in the output mask, even if they are from the same class. This task is a more advanced pixel-level classification, but it usually constraints the maximum amount of objects of the same type per image.
- Another approach to distinguish multiple instances of the same type can be *object localization*, where a bounding box is drawn around an object in the image. This can be solved as a regression problem. But if we are interested in the object's location and type at the same time, it is not enough.
- This is where multi-box *object detection* comes into play, unifying the localization (regression) and classification problem. The outputs are bounding boxes having properties $X_{\min}, Y_{\min}, X_{\max}, Y_{\max}$, and the *class*. The location information can be encoded in numerous ways, like $(X_{\min}, Y_{\min}, width, height)$ or $(X_{\text{center}}, Y_{\text{center}}, \frac{width}{2}, \frac{height}{2})$. There are approaches where bounding boxes have extra attributes like rotation angle or further coordinates for 3D object detection. In the following, I discuss 2D bounding box detection without rotation.

Object detection is a difficult task to algorithmize, so it is worth turning to the toolkit of neural networks. Although there are classical machine learning solutions, like HOG linear SVM-based detectors[56], deep learning-based object detection has been a research hotspot in recent years due to its powerful learning ability[88].

In the following part, I discuss the main concepts and explain different metrics and model evaluation protocols. Then building on this knowledge, I present numerous detector architectures and their novel ideas in chronological order.

4.1 Intersection over Union

The first essential operation to understand is *Intersection over Union*. It is calculated as follows:

$$IoU = \frac{area_{\text{overlap}}}{area_{\text{union}}}$$

It measures how much a box overlaps with another box. This simple operation is used extensively in object detection, matching ground truth boxes with anchor boxes (at training). It is also used in non-maximum suppression[39] for removing multiple predictions of the same object (at inference). Recently, it is also used as a differentiable localization loss function (CIoU[40]) between ground truth boxes and predicted boxes (at training). Sometimes, there are predefined IoU thresholds for the different usages. For example, above 0.5, an anchor box is interpreted as positive w.r.t. a ground truth box, but below as a negative sample. These are hyperparameters to tune. Figure 4.1 illustrates IoU values of different bounding box pairs.

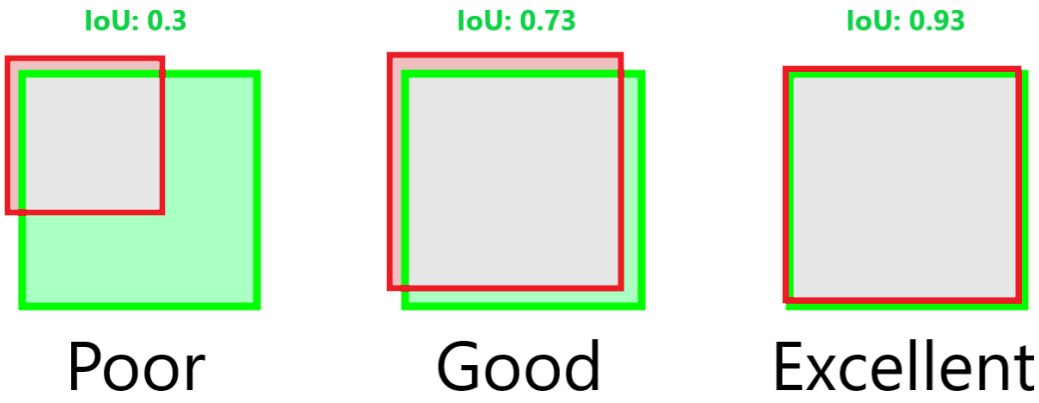

Figure 4.1: Intersection over Union of different bounding boxes.

4.2 Anchor boxes

The anchor box examples and explanation points are partly inspired by the following article[14]. When an image is passed through an object detector, it makes box predictions in the order of thousands, of which only a few are real objects. The following example illustrates and explains why detectors work like this.

Figure 4.2: An example image with different objects.

4.2.1 The problem

Suppose there is a model which can detect two objects in an image. This model gets the input image on Figure 4.2. While training, the model needs to know which predictions are correct to be able to learn. But there can be multiple right solutions:

- *predictor 1*: human, *predictor 2*: bird
- *predictor 1*: bird, *predictor 2*: human

When the detector outputs the following:

- *predictor 1*: bird, *predictor 2*: bird

It is unclear which predictor is correct and which is not. The network needs to know which of its two predictors is responsible for detecting the human or the bird. For this, predictors can specialize in specific things (object size, aspect ratio, objects in different parts of the image). Modern object detectors use all of these specializations with their predictors. In this example, the optimal task sharing is that *predictor 1* is responsible for objects on the left while *predictor 2* is for objects on the right. This way, the correct model output is clear:

- *predictor 1*: human, *predictor 2*: bird

4.2.2 Anchor-based operation

Anchor-based object detectors apply the same solution on a larger scale as follows:

- They generate thousands of predefined anchor boxes; each mapped to a single predictor. One predictor is responsible for finding objects of its anchor box's location, size, and aspect ratio.

- When training, IoU is calculated for each ground truth box and anchor box. Anchor box with the highest IoU w.r.t. the ground truth is responsible for detecting that ground truth box.
- If the calculated highest IoU is above an upper threshold (say 0.5), the corresponding anchor box needs to detect the object. If the highest IoU is below a lower threshold (say 0.3), the anchor box must predict no object. If none is met, the ground truth has too low maximum IoU to be positive but too high to be negative. In this case, the anchor box is ambiguous, so not used in the current loss calculation.
- The predictor of the anchor box outputs the offsets/rescale ratios. In some detector architectures, like YOLO, a predictor is only responsible for a ground truth box if the box's center is in the predictor's grid cell.

This anchor-based approach is used in many detector solutions and works quite well.

4.2.3 Limitations

The number of anchor boxes is in the order of thousands. Some architectures like RetinaNet[51] have more than 100,000 anchor boxes in specific cases. Calculating IoU and then predicting boxes in this order of magnitude is resource-intensive and a potential bottleneck.

The massive number of anchor boxes means a positive-negative class imbalance because there are way more anchor boxes than ground truth boxes. Most of the anchor boxes are assigned as negatives and just a tiny portion of them as positives. Many negative samples suppress the gradients of the positive ones, and the model potentially does not learn. There are various techniques to solve this problem, like hard negative mining[52] (SSD), objectness score-based loss value reduction[63] (YOLO), or a dynamically scaling focal loss function[51] (RetinaNet). I describe each of these solutions in the discussion of the specific architectures.

The last problem is that anchor boxes are parameter-sensitive. Datasets have boxes of different sizes and shapes, and if the predefined anchors do not fit the properties of the ground truth boxes, low IoU values are calculated. Thus, the network does not even know that ground truth objects exist and cannot predict them. There are different approaches to finding ideal anchor configurations. Hand-crafted bounding boxes are used in SSD, while YOLOv3[62] uses a k-means algorithm to estimate ideal values w.r.t. a given dataset.

To illustrate the phenomenon, Figure 4.3 shows 1% of the default anchor boxes (for the COCO dataset) of RetinaNet. The matched bounding boxes are the following when training and evaluating the same model on the WiderFace[85] dataset. Only a few ground truth boxes overlap with any of the anchor boxes. Thus the majority of the faces cannot be detected.

Figure 4.4 shows 1% of the adjusted anchor boxes, where the size of the smallest anchors has been reduced. With this configuration, all face ground truth boxes have been matched by at least one anchor box, so the model can detect them.

Finally, it is also important to note that the IoU value depends on the center of the compared boxes. Even if we use correctly small anchor boxes, if the strides between them are too high, some ground truth boxes can be missed. The upper IoU threshold can be lowered to address this issue, or adding a denser anchor grid-level also helps (increasing the number of anchors and thus is more resource-intensive).

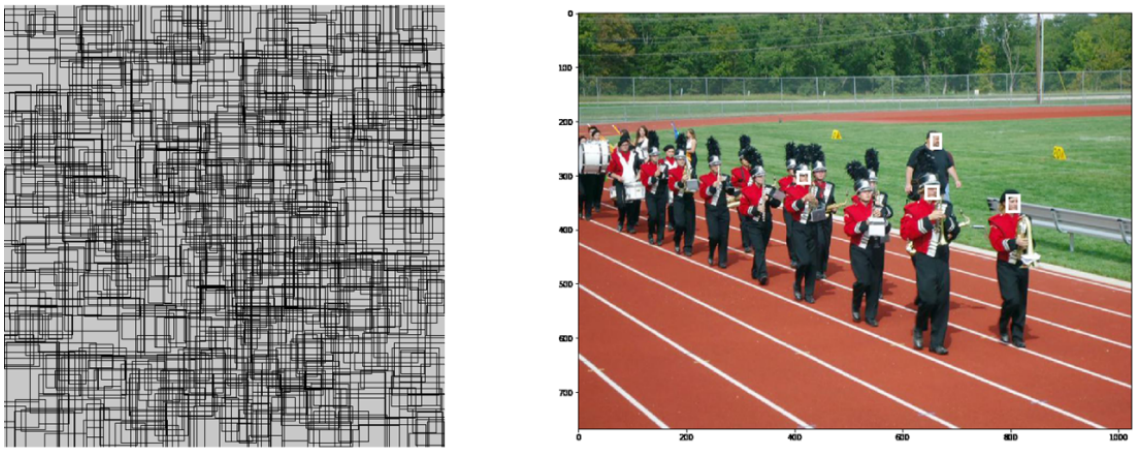

Figure 4.3: Left: 1% of the default RetinaNet anchor boxes. Right: The detector can find only a few faces with this configuration. Source: [14]

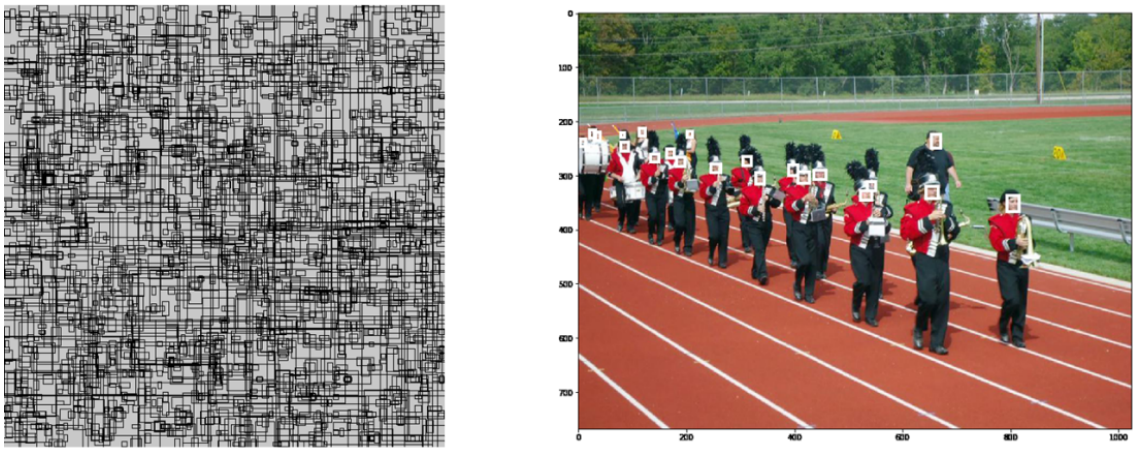

Figure 4.4: Left: 1% of the adjusted (smaller) anchor boxes. Right: With the proper anchors, the detector can localize all of the faces. Source: [14]

4.3 Non-maximum suppression

When there is a specific object in an image, the cell in the grid responsible for finding it outputs high probability predictions. Usually, the adjacent cells also output pretty good proposals for the same object. But we only need one bounding box for one object. The others need to be suppressed. The algorithm to find the best match and discard the others is called non-maximum suppression[39] (NMS). Its steps are the following:

- The method first selects a proposal box with the highest confidence score, removing it from the initial box list B and adding it to output boxes list P.
- Next, for all the other boxes in B, IoU is calculated w.r.t. the box in P. Each box with an IoU value above a threshold is discarded from B.
- This simple algorithm is repeated until B gets empty. When this happens, P contains the final list of the predicted boxes.

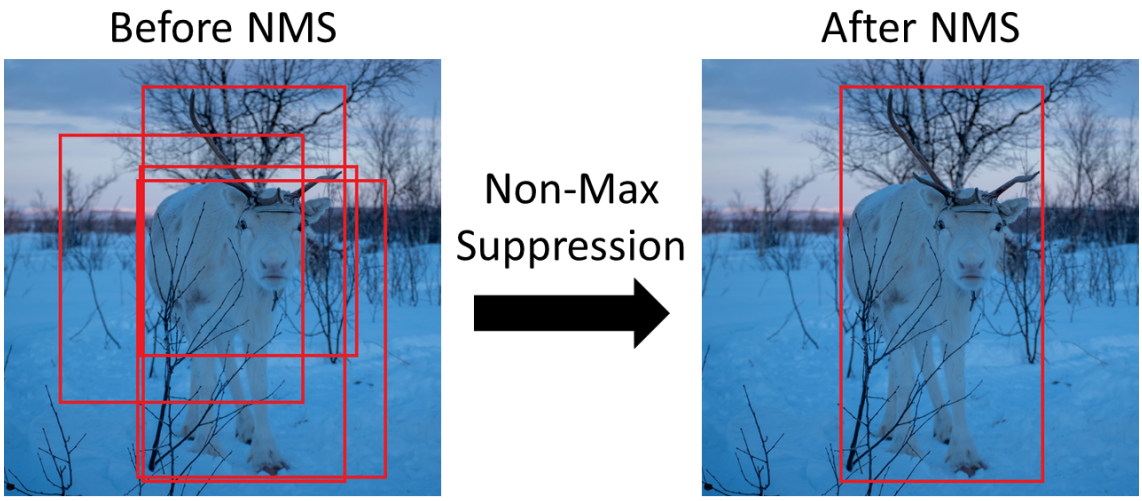

Figure 4.5: The effect of the non-maximum suppression algorithm.

This algorithm has numerous variants, like Combined NMS, where the IoUs are calculated between boxes classified as the same class. This avoids suppressing boxes of different objects close to each other.

4.4 Loss function

Object detection unifies a localization (regression) and a classification problem. These are tasks with different properties: one is in a continuous domain, while the other is in a discrete. In early two-stage detectors like R-CNN[25], the localization and classification tasks are entirely separated. Each module could be trained with the appropriate loss function individually. When training detector systems in an end-to-end manner, we need to optimize both at the same time. For this, object detection usually has a composite loss function, incorporating penalties for both things. The single output value of the cost function is the error, which is backpropagated through the network to optimize the model: calculate the gradients and update weights. This is why a suitable loss function is crucial to train an object detector.

When training in an end to end manner, the cost function incorporates both task penalties. It is important to weigh the classification and localization errors to avoid any of them dominating the error. There are usually hyperparameters to weigh those loss components.

Over time, many loss functions emerged. The original YOLO[63] uses sum-squared error for both problems, which can work by introducing a third simple loss component (objectness score), but has its limitations. SSD[52] uses Softmax for classification and Smooth L1 for localization. It adds an extra empty class and hard negative mining to balance class inequality, different from YOLO. RetinaNet[51] introduced focal loss for classification, a function handling class inequality by downscaling easy examples. YOLOX[22] uses a differentiable IoU function[39] (Complete IoU) for calculating localization loss. There are many different approaches with their pros and cons. I discuss them more thoroughly in the detector architectures section.

4.5 Metrics

Choosing the right metrics is crucial to train and evaluate a detector appropriately for the task. If we solely concentrate on the single loss value, we easily 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. For example, in a single-stage architecture (discussed later), classification depends primarily on the backbone, while detection is mainly the task of the last convolutional layers. Thus, if the classification performance is weak, we may modify the backbone to improve it.

Some metrics appear for most protocols, which I briefly describe here partly based on [34]. The following concepts assume a binary decision problem, but this can be extended to a multi-class decision by saying positive if the target class is predicted or negative in any other case. I use the following abbreviations: T_p: true positive, T_n: true negative, F_p: false positive, F_n: false negative.

Acccuracy is the ratio of trues among all predictions. We can calculate it as follows:

$$Accuracy = \frac{T_\mathrm{p} + T_\mathrm{n}}{Total}$$

Precision gives the ratio of ground truth positives among all positive predictions.

$$Precision = \frac{T_\mathrm{p}}{T_\mathrm{p} + F_\mathrm{p}}$$

The *Recall* metric is the true positive hit ratio among all ground truth positives.

$$Recall = \frac{T_\mathrm{p}}{T_\mathrm{p} + F_\mathrm{n}}$$

Average Precision is the area under the precision-recall curve.

$$AP = \int_0^1 p(r)\,dr$$

The following example demonstrates how to calculate Average Precision. Table 4.1 shows the result of some evaluations:

Table 4.1: Sample detection results in a tabular form.

Id	Correct (IoU>0.5)?	Precision	Recall
1	*True*	1.00	0.33
2	*True*	1.00	0.66
3	*False*	0.67	0.66
4	*True*	0.75	1.00
5	*False*	0.60	1.00

The calculation of detection 4 of Table 4.1 is the following: Precision is the proportion of T_ps so far $\frac{3}{4} = 0.75$, Recall is the proportion of T_ps out of the possible positives $\frac{1}{3} = 1$. Recall continually increases as we go down on 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. The AP of the current example is on Figure 4.6. As precision and recall values are always between 0 and 1, AP falls within this boundary, too.

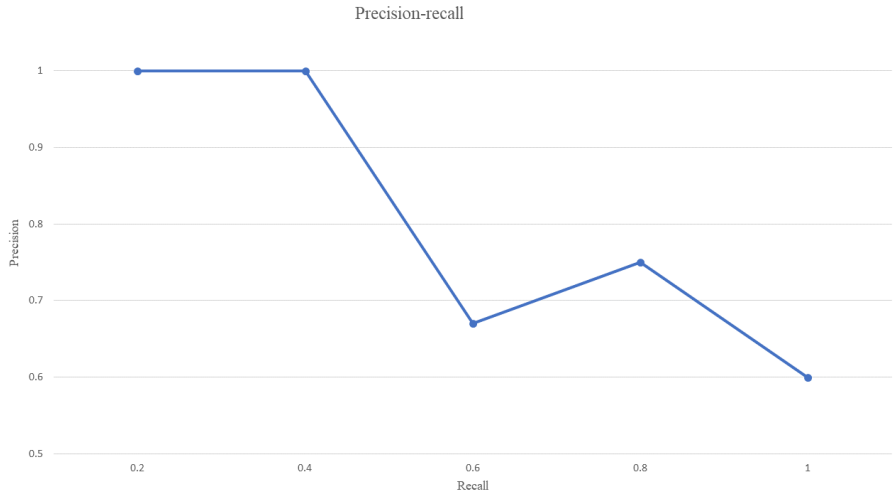

Figure 4.6: Sample detection AP in a line chart.

Interpolated Average Precision divides the recall value from 0 to 1 into n points. It is used to smooth out the zig-zag pattern of AP (at each recall level, replacing each precision value with the highest precision found to the right of that recall level[54]). The IAP of the example is on Figure 4.7. Interpolated AP is calculated based on the area below the smoothed values. It is the basic idea behind the mAP metric variants.

$$IAP = \frac{1}{n} \sum_{i \in (0.0...1.0)} AP_{\mathrm{i}}$$

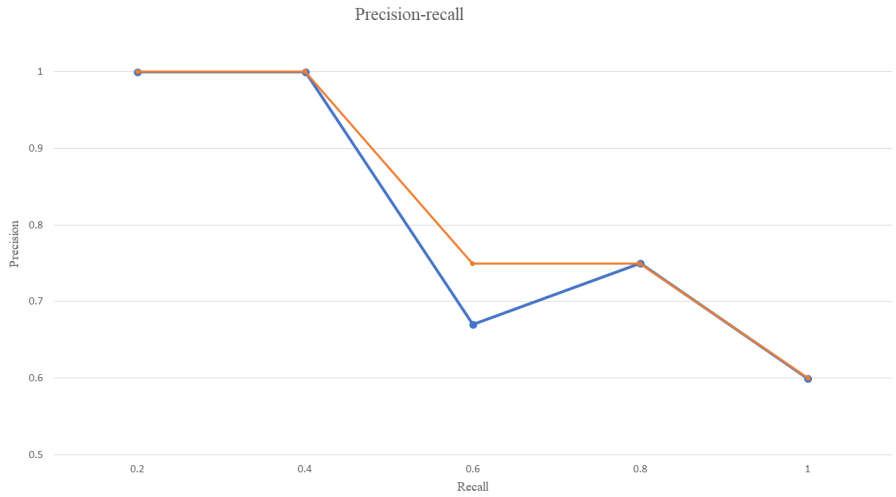

Figure 4.7: Metric comparison on the previous sample: AP, IAP.

The last metric discussed here is *mean Average Precision (mAP)*. There are numerous types of mAP. In the COCO[50] protocol, 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.

4.6 Protocols

There is no consensus about the evaluation metrics of the object detection problem. Different competitions such as PASCAL VOC[17], Common Objects in Context[50] (COCO), or Google Open Images Challenge[45] (OID) have their ways to measure performance[34].

PASCAL VOC[17] has introduced mAP to evaluate detectors' quality with an 11-point interpolated AP definition.

COCO detection metrics[50] 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 ($32^2 \leq area < 96^2$), and large ($area \geq 96^2$) objects.

Open Images V2 detection metrics[45] focus 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.

4.7 CNN architectures

Before we deep-dive into the detector architectures, I discuss some of the most popular CNN versions popular in object detectors.

Convolutional Neural Network (CNN) models are widely used in deep learning. They usually stack multiple convolutional layers with nonlinearity (usually ReLU) and pooling layers. Because of pooling, the input feature spatially gets smaller, but by convolution, the dimensionality increases. Object detection architectures usually use CNNs as feature extractor backbones. Other approaches, like transformer-based detectors, exist, but current real-time systems use CNNs. Therefore, I present some of the most popular variants used in object detection.

Early convolutional networks were primarily engineered for image classification problems. The first stepping stone I mention is LeNet-5[48] from 1998, created by Yann LeCun for classifying hand-written digits. The next one is AlexNet[44], which won the 2012 ImageNet ILSVRC classification challenge[65]; this model stacked multiple convolutional layers on top of each other without pooling and is a much deeper network. GoogLeNet[73] won the ILSVRC challenge in 2014; the novel idea of this network was the inception module, where each block contains multiple different convolutional operations to extract features of more diverse patterns. These models are not usually used for object detection, but the ideas they introduce have been extensively used.

VGG-16[71] was the first popular CNN used in object detectors. In 2014, it was the runner-up behind GoogLeNet in the ILSVRC challenge. This model has a simple structure with 2-3 convolutional layers between each pooling layer (16 layers in total). VGG-16 became a popular detector backbone due to its power and simplicity and the fact that the first detector architectures (Fast R-CNN, SSD) emerged around that time.

ResNet[30] won the ImageNet 2015 challenge. It has introduced skip connections with 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 started learning (their output is close to zero), the network can progress. These residual units help resolve the vanishing gradient problem, making it possible to create deeper trainable networks (around 50-100 layers).

MobileNets are special models optimized for mobile/IoT inference. They are small and fast, but their accuracy is relatively high. Version 2 uses linear bottlenecks and inverted residuals. According to the MobileNetV2 paper[66], bottlenecks store all the necessary information, and between them, expansion layers serve for extraction with nonlinearities. Thus, bottlenecks are linear layers to prevent nonlinearities from destroying the original information. The other novel idea is that as bottlenecks store the information, shortcuts are only needed directly between them.

There are other popular CNNs used in object detection, like DarkNet, CornerNet, or EfficientNet. Since these are models optimized for detection, I discuss them in the description of the detector architecture associated with them.

4.8 Detector architectures

Generally, there are two types of deep learning-based detector architectures: two-stage (e.g., R-CNN) and one-stage variants (like YOLO, SSD, RetinaNet, CenterNet). I briefly describe the two-stage versions, then more comprehensively the one-stage detectors because they are faster, thus, more suitable for smartphone inference.

4.8.1 Two-stage detectors

Two-stage detectors have this name because first, they propose object candidate regions, then classify these regions. Those steps are executed with different algorithms separately. Until the mid-2010s, complex ensemble machines were the best performing models in object detection.

4.8.1.1 Region-based Convolutional Neural Network

In 2014, Region-based Convolutional Neural Network[25] (R-CNN) was introduced for detection and segmentation tasks. Its main idea was to use a region proposal for generating category-independent object candidates, then feeding them to a feature extractor CNN. In this arrangement, there is a third module, which is a set of class-specific linear SVMs. In this architecture, the separate modules must be trained independently.

4.8.1.2 Fast R-CNN

In 2015, Fast R-CNN[24] was introduced. It used VGG-16 as a feature extractor and made it possible to train the whole system at once. 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. Proposals are identified by an RoI (Region of Interest, max-pooling) layer, 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 quicker than its predecessor because, for 100 region proposals, there is no need to feed all of them independently to the CNN - RoIs from the same image share computation and memory.

4.8.1.3 Faster R-CNN

The Faster R-CNN[64] variant from 2016 consists of a fully convolutional RPN (Region Proposal Network) and a Fast R-CNN detector using the output of the former. The RPN is a fully convolutional module predicting offsets for hand-picked prior bounding boxes (anchor boxes). 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 significantly slower selective search algorithm is used for this purpose.

4.8.1.4 Mask R-CNN

The next variant of the R-CNN family is the Mask R-CNN[31]. It extends Faster R-CNN by adding a small fully-connected network as a third output branch predicting object masks (parallel with localization and classification). In addition, this variant introduces the RoIAlign layer, which accurately maps a region of interest to a smaller feature map without rounding up to integers. This architecture is suitable for both detection and segmentation.

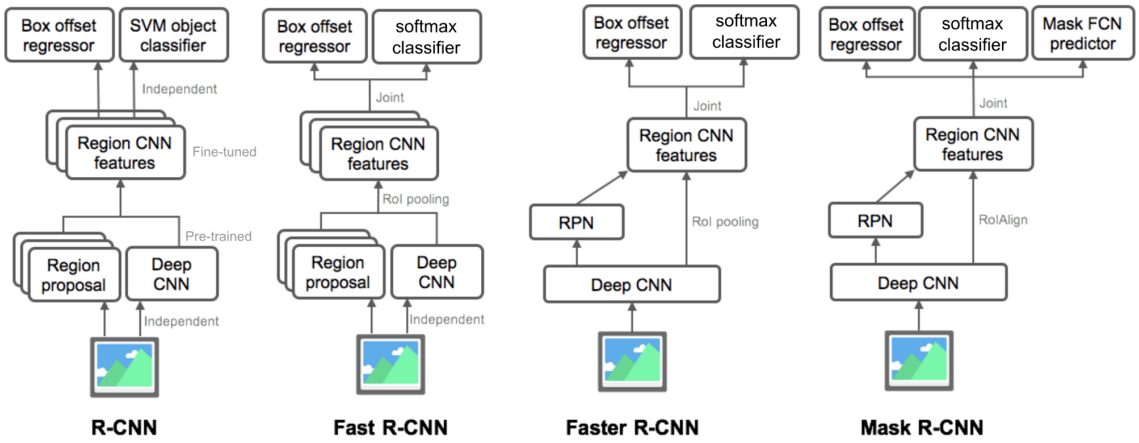

Figure 4.8: R-CNN family variants. Source: [82]

Further two-stage detectors, like Libra R-CNN[58], apply balanced IoU sampling, FPN, and L1 loss to address the class imbalance problem during training. Powerful anchor-free variants also exist, like RepPoints. However, I do not discuss these particular approaches, as my main goal is to present the basic ideas behind two-staged detectors.

4.8.2 One-stage detectors

Two-stage detectors can achieve high accuracy, but they are usually too slow for real-time application. One-stage detectors are not region-based algorithms; they merge the localization and classification tasks into one network - running detection directly over a dense sampling of possible locations. One-stage detectors are fast by design, but they are usually less accurate.

4.8.2.1 You Only Look Once

The first one-stage detector aiming for real-time inference was You Only Look Once[63] (YOLO). It was introduced in June 2015, initially written in the DarkNet framework. YOLO is a unified architecture with a single network containing 24 convolutional layers and 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. In YOLO, one box has six properties: $X_{\text{center}}, Y_{\text{center}}, \frac{width}{2}, \frac{height}{2}, objectness$ and multi-hot encoded *class confidence*. The localization values $(X_{\text{center}}, Y_{\text{center}}, \frac{width}{2}, \frac{height}{2})$ are normalized and directly predicted using fully connected layers. A single cell outputs two boxes and their confidence values for every C classes, so the total output shape is $N \times N \times ((2 \times 5) + C)$. This means that one grid cell can detect maximum one object. In YOLO, the loss function is a sum-squared error for both classification and localization. This loss function has the following limitations:

- It weights regression (localization) and classification errors the same way, which is not optimal because they are fundamentally different problems (continuous and discrete domains).
- The high number of negative samples (in every image, many grid cells do not contain any object) tends to overpower the positive ones, pushing the confidence scores towards zero. This is handled in YOLO by predicting the objectness score and down-scaling bounding box losses with low objectness values.
- The loss function equally weights errors in small and large boxes, but the loss should reflect that slight deviation in large boxes matters less than in small boxes. To address this, YOLO predicts the square root of the bounding box width and height.

These problems regarding the loss function have been addressed in later architectures. When it was introduced, YOLO was considered fast and straightforward compared to the state-of-the-art and less likely to predict false positives on the background than two-staged detectors. On the other hand, it makes more localization mistakes (partly related to the loss function) and struggles with small objects.

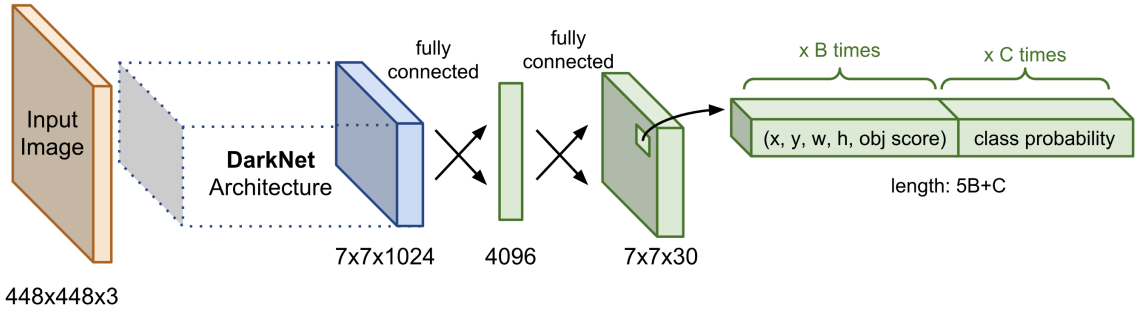

Figure 4.9: The YOLO architecture. Source: [83]

4.8.2.2 Single Shot MultiBox Detector

Single Shot MultiBox Detector[52] (SSD) was introduced in December 2015. In the first part of this architecture, a general-purpose image classification CNN (VGG-16) is used as the backbone. After that, multiple convolutional layers implement a pyramidal structure for multi-scale object localization as shown in Figure 4.10. These layers progressively decrease in size towards the end of the architecture, allowing detections at various scales (unlike YOLO, which operates on a single-scale feature map). This makes it possible to detect objects of different sizes better at the cost of more resource intensity.

SSD uses a set of default bounding boxes for each cell on each feature map. For training, anchor boxes of different sizes and IoU calculations are needed. This is a fundamentally different approach than YOLO. The box properties are $X_{\text{center}}, Y_{\text{center}}, \ln \frac{width}{2}, \ln \frac{height}{2}$, and $confidence$, where the first four values are offsets and resizing ratios of the corresponding anchor box. The predictor outputs the logarithm of the vertical- and horizontal rescaling factors, which is an easier representation for neural networks and thus speeds up training. Unlike YOLO, SSD has no objectness score prediction and loss downscaling, but encodes non-objects as an empty class. The loss function is the sum of the classification loss (softmax over the class confidences) and the localization loss (Smooth L1; a quadratic function below a threshold and an absolute function above). SSD has numerous anchor boxes in multiple scales, and the majority of them do not contain any object for a single image. There is no objectness score-based loss downscaling, so the positive-negative class imbalance is a significant problem. As a remedy, hard negative mining is applied after each matching step, selecting the top negative samples with the highest losses so that the positive-negative sample ratio is 1:3. The other negatives are discarded, leading to faster and more stable training.

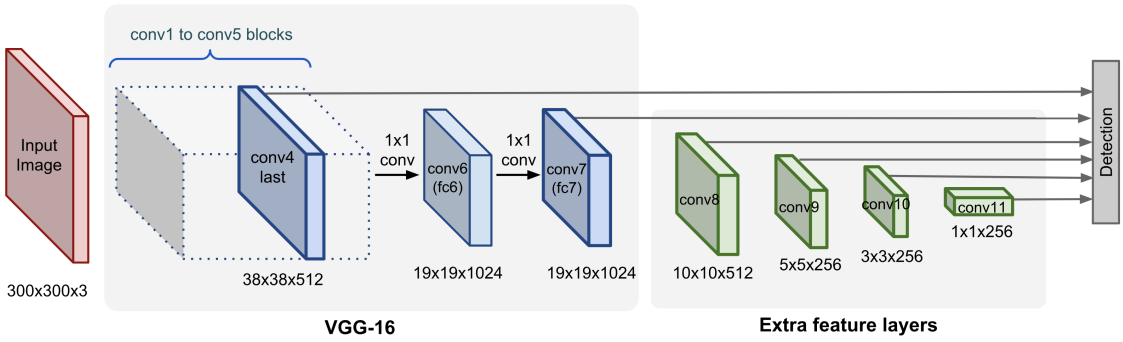

Figure 4.10: The SSD architecture. The extra feature layers are responsible for multiple object size detection. Source: [83]

4.8.2.3 YOLOv2

In 2016, YOLOv2[61] was introduced and brought numerous improvements over the first version. The v2 architecture uses anchor boxes similar to SSD. Thus, fully connected layers are eliminated. But instead of offsets, YOLOv2 predicts location coordinates relative to the predictor's grid cell location (between 0 and 1). Coordinate (0, 0) is the top left, while (1, 1) is the bottom right of the predictor's cell. This constraint prevents anchor boxes from ending up at any point in the image, stabilizing early training. The default anchor boxes are chosen by running K-Means clustering on the training set. SSD uses various

feature maps to get objects of different resolutions. YOLOv2 uses the last convolutional feature map and adds one extra passthrough connection from a previous, higher-resolution layer. The high-resolution features are downscaled by stacking the neighboring features into different channels, turning the spatially bigger feature map to the same size as the last layer, making them concatenable. YOLOv2 also introduced a new CNN backbone, Darknet-19, an ImageNet pre-trained model with batch normalization. It has significantly fewer operations than SSD's VGG-16. During training, the authors used multi-scale training and different augmentations.

YOLO9000 is a particular v2 variant trained on the COCO (localization+classification) and the ImageNet (classification) datasets with a WordTree model and conditional probability. It can distinguish 9000 different classes thanks to a hierarchical label assignment technique.

4.8.2.4 RetinaNet

RetinaNet[51] is an architecture published in 2017 addressing the previously stated negative-positive class imbalance issue.

This solution uses Smooth L1 loss similarly to SSD for box localization but proposes focal loss for classification. Generally, the class imbalance encountered during training can overwhelm cross-entropy loss. SSD uses hard example mining to resolve this. Focal loss is a dynamically scaled cross-entropy loss. The scaling factor decays to zero as confidence in the correct class increases so that the function can weigh, thus handling class imbalance. The publication uses binary cross-entropy (BCE, also called sigmoid cross-entropy), a sigmoid activation plus a cross-entropy loss. Unlike softmax, BCE is independent for each vector component (class), meaning that the loss computed for every output vector is not affected by others. That is why BCE is used for multi-label classification, where an element belonging to a specific class should not influence the decision for another class. Binary cross-entropy is faster than multi-class cross-entropy, and the name comes from that it sets up a binary classification problem between $C = 2$ types for every class in C. This loss is also helpful to avoid exponential overflow.

RetinaNet uses the ResNet-50 network as a backbone and takes three different convolutional layer outputs to build a Feature Pyramid Network (FPN). The FPN has a bottom-up path with feature downscaling (the selected ResNet layers are the key levels), then a top-down path with upscaling by the nearest neighbor algorithm. There are lateral connections (element-wise addition) between the corresponding levels of the two pathways. The FPN module generates semantically stronger feature maps on all pyramidal levels, resolving the problem of too low-level features and thus weak earlier layer predictions (responsible for smaller objects). Finally, for each FPN level output, a decoupled classification- and localization head is connected (convolutional layers) to generate outputs for each anchor box level. The high-level structure is shown in Figure 4.11.

RetinaNet uses anchor boxes similarly to SSD, with multiple level anchors. A single predictor outputs cell offsets and anchor rescaling ratios. The architecture introduced novel concepts; it was accurate and robust. However, the deep backbone, the FPN module, and the massive amount of multi-level anchor boxes result in slower running times.

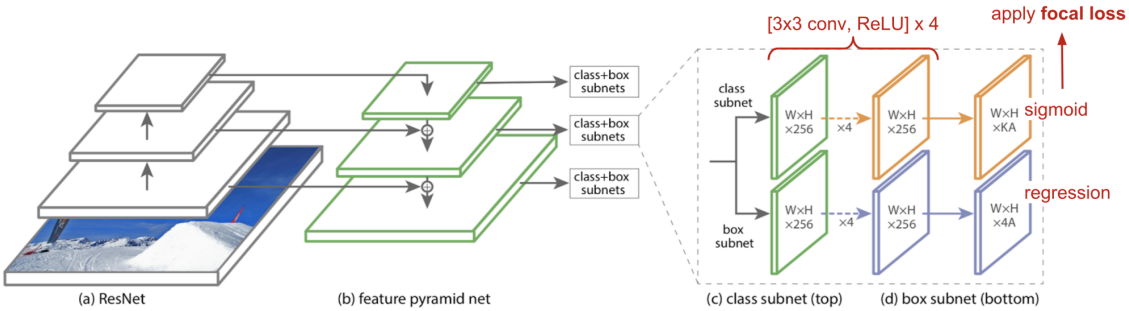

Figure 4.11: The RetinaNet architecture. The (a) part is the backbone network whose different level features serve as inputs to the upscaling FPN module (b). A separate predictor head (c-d) is associated with each FPN layer. Source: [83]

4.8.2.5 YOLOv3

YOLOv3[62] was published in 2018, presenting incremental improvements over YOLOv2. In this version, the softmax activation for class prediction is replaced by binary cross-entropy, making class prediction values for an object independent from each other. Also, the objectness score is predicted by logistic regression (binary cross-entropy) instead of the sum of squared errors in the previous YOLOs. Focal loss is not used as the YOLO variants already output objectness scores and downscale losses based on them.

YOLOv3 incorporates the idea of FPN to make predictions at three different scales similar to RetinaNet. The backbone has been replaced with a new model named DarkNet-53, similar to ResNet (deeper network with skip connections).

YOLOv3 compromises speed and accuracy: the deeper backbone and the FPN make it slower than its predecessor. Still, it is more than three times faster than the original RetinaNet and just slightly less accurate, which made YOLOv3 a de-facto industrial solution for years.

4.8.2.6 EfficientDet

EfficientDet[74] was introduced in July 2020 by Google Brain and was the benchmark on the COCO dataset in 2020 (55.1 mAP). It is an anchor-based architecture family of object detectors. The paper has proposed the bidirectional Feature Pyramid Network (biFPN), which learns to weigh input features with different resolutions before fusing them. EfficientDet uses EfficientNet CNN backbones, which were engineered with the help of Neural Architecture Search (an automatic process of designing neural networks). The EfficientDet architecture is similar to RetinaNet. It is illustrated in Figure 4.13.

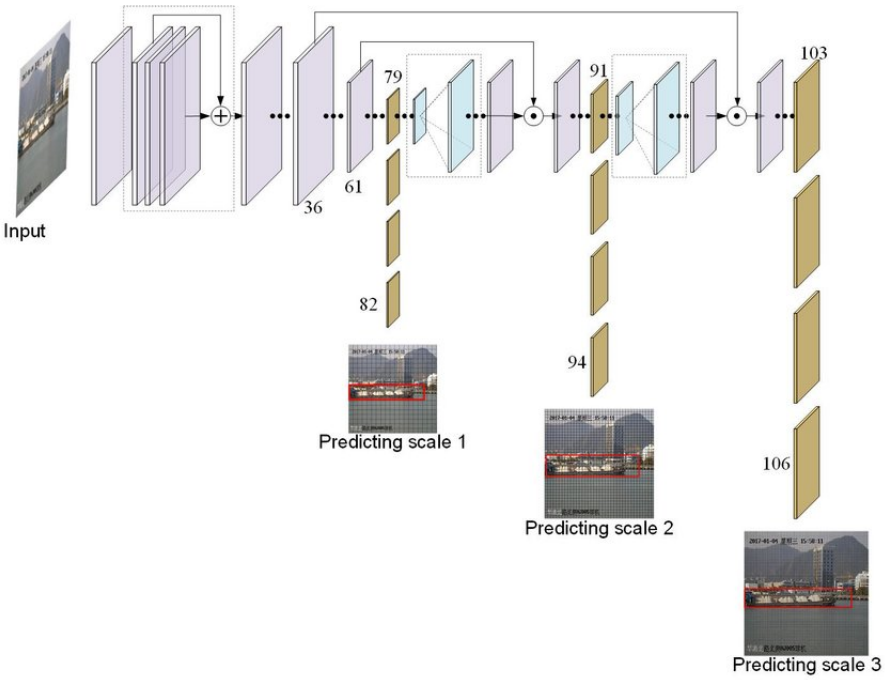

Figure 4.12: The YOLOv3 architecture is similar to RetinaNet. For each scale, a coupled predictor head provides the detector outputs. Source: [57]

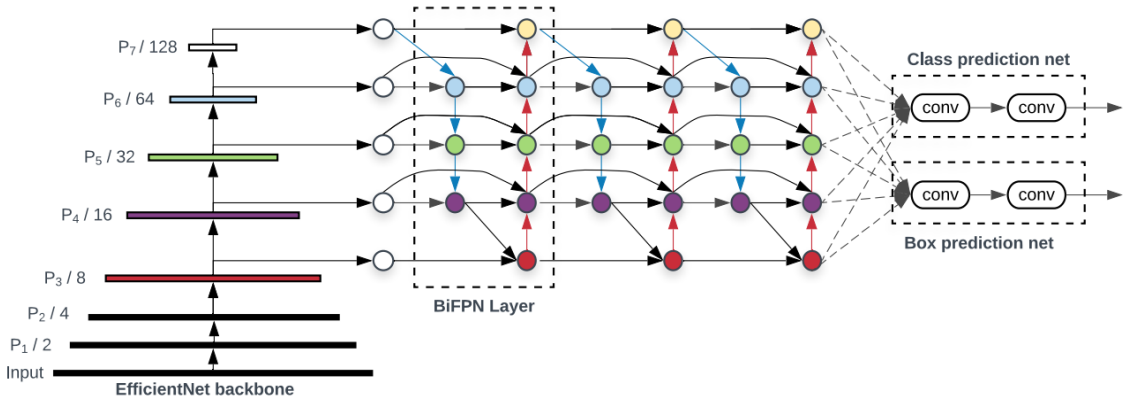

Figure 4.13: The EfficientDet architecture learns to weigh features from different FPN levels. Source: [74]

Over time, a massive number of new solutions emerged. The state-of-the-art solutions on the COCO dataset (as of December 2021) use transformer-based neural networks (like SwinV2-G(HTC++) or Florence CoSwin-H), different from the CNN-based approaches. These are computationally heavy algorithms, making them unsuitable for real-time inference.

The fast-running YOLOv4[13] introduced further data augmentation and regularization techniques, a smaller backbone, and the Mish activation function. A month after YOLOv4, YOLOv5[38] was released. These architectures, however, do not contain significant modifications. Personally, it feels like they are a bit overly optimized to the anchor-based pipeline. For this reason, I do not discuss them. Instead, I cover anchor-free detectors in the following part, which has been the subject of numerous publications in recent years.

4.8.2.7 CenterNet

CenterNet[89] was released in 2019, and it uses a different approach compared to other architectures presented so far. It is based on the keypoint-based CornerNet backbone 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 inspects whether the center key point's region is predicted as the same class as the whole bounding box. This way, the box borders and its central region's visual patterns are analyzed, providing a more robust approach to reduce false positives. CenterNet does not use anchor boxes.

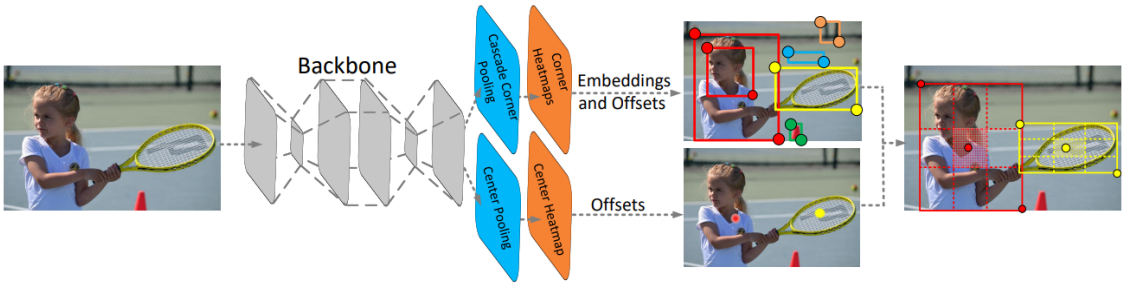

Figure 4.14: The CenterNet architecture. Source: [89]

4.8.2.8 YOLOX

YOLOX[22] is a publication from August 2021 presenting an anchor-free version of YOLOv3. The authors chose v3 as the starting point because it is widely used, while later variants are overly-optimized, and insufficient software support cannot leverage their capabilities. This architecture slightly modifies v3's loss function, using Complete IoU loss[40] for the regression (localization) branch. For both the classification- and objectness branches, it uses binary cross-entropy.

YOLOX replaces the coupled detection head with decoupled heads, one each for classification and for localization per feature-pyramid level. The head structure can be seen in Figure 4.15. Anchor mechanism increases detection head complexity, and the massive number of predictions for each image makes it a bottleneck. YOLOX does not use anchor boxes. Instead, each grid location predicts a single bounding box by four offsets: $Xmin, Ymin, width, height$. While training, each ground truth box's center location is assigned to the grid cell where it falls in as the positive sample. For each FPN level, a different scale range is predefined in which the current level grid cell can offset its prediction. Center sampling is also used in YOLOX, assigning the central portion cells of ground-truth boxes as positive samples. The label assignment strategy applied is called SimOTA. It first calculates the pairwise matching value for each prediction-ground truth pair by the loss function. Then it selects the top k predictions within the center region as positive samples, while the others are negatives.

In this part, I presented a few key object detector solutions. The list is not exhaustive, as I didn't describe many other architectures, like EfficientDet, FCOS, NanoDet, and the current state-of-the-art Transformer detectors. The aim of this section was to give an illustration of the basic ideas behind real-time object detection systems.

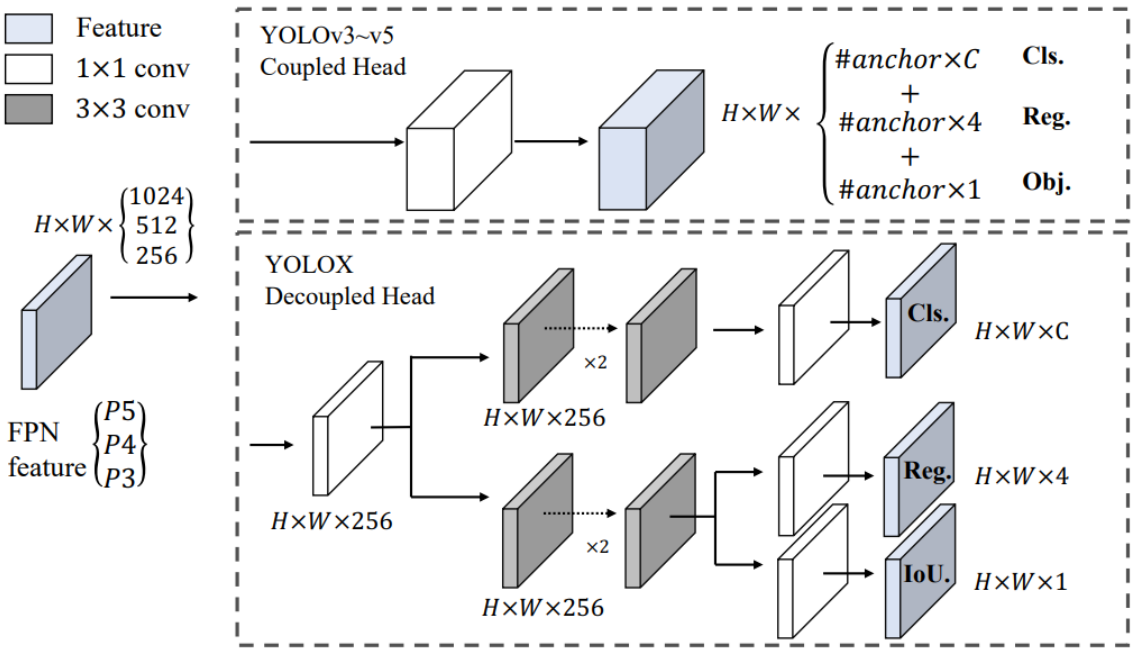

Figure 4.15: Decoupled head of YOLOX, which slightly increases runtime but improves accuracy by delegating classification and regression to different branches. Source: [22]

Chapter 5

Automatic license plate recognition

Automatic license plate recognition (ALPR) refers to a technology that identifies vehicles based on their number plates. It is based on Optical character recognition. Traditionally, these systems are used to find stolen vehicles, check for road usage permits, measure vehicle speed, or operate parking garages. The technology is also suitable for tracking vehicles and collecting location data. This may be to the benefit of the authorities, but in Europe, it raises privacy concerns, as drivers have the right to data privacy.

ALPR systems can be categorized according to several aspects. There are fixed (pre-installed) and mobile (cameras in vehicles) systems based on their mobility. According to another aspect, some systems perform on-device image evaluation, while others process them remotely (like on a central computer or a server farm). The variants are illustrated in Figure 5.1. Both solutions have pros and contras in terms of network bandwidth- and hardware requirements.

Figure 5.1: Pre-installed closed-circuit ALPR system[42] (left) and a police car equipped with mobile ALPR[18] (right).

5.1 History

The technology began to spread worldwide in the 2000s; however, the first ALPR systems were in service as early as the late 1970s. The first system was created in 1976 in the UK by the Police Scientific Development Branch. Their solution has been installed to monitor the A1 road's traffic and to look for stolen vehicles. The first arrest based on ALPR identification of a stolen car happened in 1981[35].

As the technology evolved, more sophisticated solutions emerged. Fixed cameras began to form coherent networks, and thanks to hardware developments, previously expensive and cumbersome systems became affordable and widely available. The proliferation of the systems was further facilitated by changing license plates in many countries (like in the Netherlands) to help ALPR operation[2].

During the 1990s, mobile ALPR systems also appeared. This was due to the elimination of special hardware requirements, and the more robust solutions no longer required certain view angles or conditions to work. A challenging task, in this case, is solving the power supply requirements of the hardware from a battery. Providing an internet connection while moving is also a requirement. These have been challenging issues in the past; nowadays, they are no longer limiting factors.

5.2 Components

There are many versions of ALPR solutions that regularly differ from each other. Still, below I try to give a general picture of what image processing tasks arise in an ALPR system[1] (without claiming completeness):

- Plate localization is the process of finding and isolating the license plates. This can be either done with object detection or semantic segmentation.
- Plate resizing and orientation try to minimize the skew of an image and adjust the dimensions to the required size. Various heuristics (like Projection Profile Method[5]) exist to determine skew and apply projection afterward. A recent solution is the use of attention-based transformer neural networks[36].
- Image manipulation is a collective concept for pixel-level transformations based on its statistical properties in the present work's context. The process can either be normalization (rescale values into a range of [0, 1]), standardization (rescale values to have 0 mean and a standard deviation of 1), grayscale conversion, a combination of these, or other. Care must be taken with these operations because the image quality significantly affects the effectiveness of subsequent steps.
- Optical character recognition consists of character segmentation and classification - more about this in the OCR chapter.
- Syntactic inspection is the process where country-specific information is taken into consideration (position of specific characters, length of the recognized sequence). The goal here is to extract and validate the correct identifier.
- Averaging multiple images can significantly improve performance. It helps by averaging unwanted protruding effects, such as blur or reflection, that often occur in pictures of moving vehicles.

Not all ALPR systems explicitly separate the above points (for example, the OCR character segmentation- and classification can be done at once or as separate steps). These factors influence the usability of a solution - the key is the implementation and coordination of them.

5.3 Challenges

In the case of an ALPR system, there are numerous difficulties for which there are different solutions.

The variance of the images is quite large. Accurate operation at different times (day or night) and weather conditions (sunny, snowy, foggy, rainy) is also expected in most cases. Image manipulation techniques can overcome this problem to some extent. Devices can see the license plates in different sizes and angles depending on their installation. This is where plate localization, then proper resizing and orientation can help - although this cannot provide a solution for too distant, low-quality license plate images. Blurring caused by the movement of cars is also a problem. The answer to this is to use short-shutter cameras (1/1000 second) with a global shutter. The effect of different shutter speeds are illustrated by Figure 5.2. Modern smartphones are generally capable of the required shutter speeds. However, the presence of the rolling shutter can still be an issue.

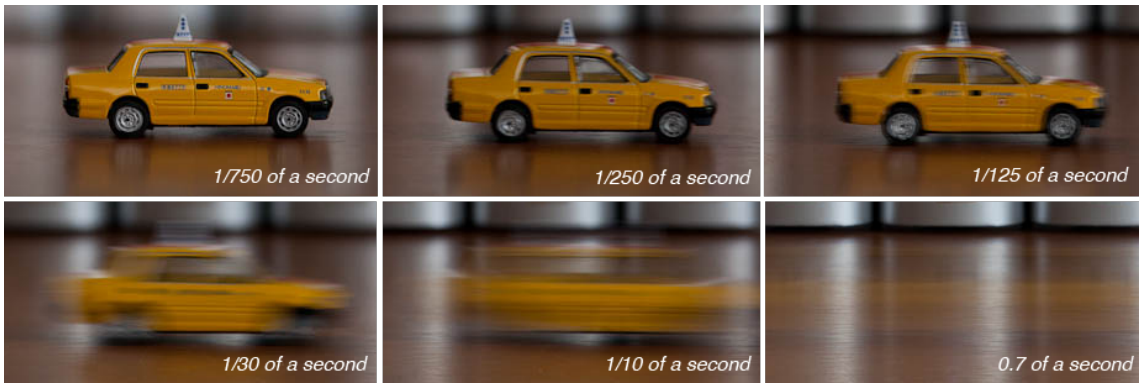

Figure 5.2: The effect of shutter speed illustrated by a model car[80].

Another common difficulty is the variety of number plates. License plates can have various structures, colors, and fonts, and their shape can also vary. It is typical for single and multirow license plates to be used within a country. For these reasons, most of the current ALPR systems can only be used satisfactorily in a given country. However, the situation within the European Union has improved in recent years with the proliferation of EU number plates. Although ALPR processability is now considered in the design of most license plates, in some countries, few characters have almost the same symbol (like 0 and O in the United Kingdom). Outside of Europe, characters outside the Latin alphabet may also appear. Character variability is discussed in more detail in the OCR chapter. Dirt may also be present on license plates, or other objects obscure their text. Deliberate circumvention attempts can be an additional difficulty for ALPR systems. This can be, for example, covering certain characters or using a surface that impairs visibility. I do not address this issue in this work.

5.4 Evaluation

Most ALPR system publishers provide rough metrics about their solutions, making it difficult to compare them. I found two de-facto benchmark datasets for license plate identification. The first one is the SSIG SegPlate Database[26], with 101 on-track vehicles captured by a static camera. The other one is the UFPR-ALPR Dataset[46], containing 4,500 images with 30,000 number plate characters.

Figure 5.3: Samples from the SSIG SegPlate[26] (left) and the UFPR-ALPR[46] (right) datasets.

Sighthound[69] and OpenALPR[72] are considered to be widespread market players. These solutions have been compared to a YOLO-based ALPR system by Laroca et al.[47]. Based on their performances on the SSIG dataset, Sighthound had 89.80%, OpenALPR 93.03%, and YOLO-ALPR 93.53% accuracy. On the more challenging UFPR-ALPR dataset, Sighthound scored 47.39%, OpenALPR 50.94%, and YOLO-ALPR 64.89% validation accuracy.

Chapter 6

License plate localization

This chapter discusses the work done on vehicle and number plate detection. The main goal of this section's algorithm is to localize number plates on input images. I used deep learning-based object detection for this task. I chose object detection over semantic segmentation because multiple license plates on a single image can be possible. Semantic segmentation does pixel-level classification, thus does not help in distinguishing objects of the same class. Object segmentation would hardly constrain the maximum number of things to find in the same type on the image.

Although the main task is number plate detection, I also decided to include vehicle detection. License plates are usually on vehicles that we want to identify. They have properties like type, make and color. If a stolen number plate is identified, we are primarily interested in the vehicle it belongs to. We can also infer additional information (like number plates without vehicles, which are false positives) by doing vehicle detection.

6.1 Data

The following section describes collecting, analyzing, cleaning, and validating the dataset needed for training. An essential aspect of obtaining the data was to cover as many images as possible with labeled license plates. Thousands of photos are necessary to ensure proper data diversity. The aim was to get not only license plate annotations but also vehicle annotations.

6.1.1 Sources

There were not many standalone datasets meeting these requirements. I conducted my research mainly on Kaggle and the Google Dataset Search engine. The most promising sets had only a few hundred labels.

I examined the Common Objects in Context[50] (COCO) and the Open Images[45] (OID) datasets. These are large sets containing 123,287 and roughly 2 million detection images. I assumed that if at least one of them had license plate annotations, the number of these labels would be enough. Of these, OID has 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.

6.1.2 Pre-processing

The application is not limited to detecting only cars. It is necessary because the stolen vehicle database and the police data source contain cars, trucks, motorbikes, and other vehicles.

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). I discarded the original OID vehicle class because it contained objects challenging to place in other subclasses like surfboards and wheelchairs. A few less essential types were discarded (like aerial vehicles or snowmobiles) to keep the cardinality of the resulting classes equal. Figure 6.1 shows all the classes used from the OID Vehicle branch.

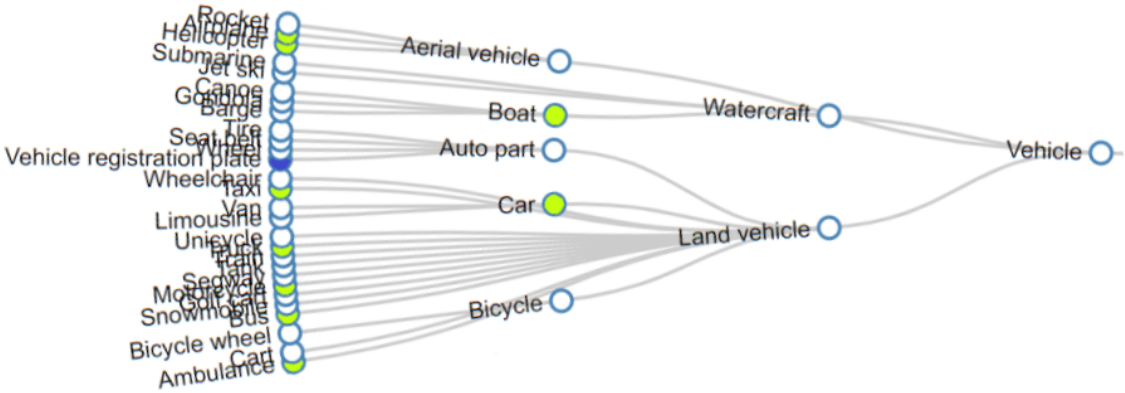

Figure 6.1: Selected classes from the original OID Vehicle branch: license plate, vehicle.

6.1.3 Analysis

I used Power BI to analyze and visualize the downloaded content. My project uses CSV files as data sources. The raw dataset contains 6,867 images with 28,102 bounding boxes, roughly four annotations per image. There are 9,934 vehicle registration plate annotations, one-third of all the boxes. The report revealed that the average image size is 1005×753 pixels. While most images have a dimension of 1024×768 pixels, some exemptions occur. As the model input is always resized, this is not important to examine more thoroughly.

Table 6.1 and Figure 6.2 show that there are more vehicle boxes in the dataset. The left chart shows cardinalities compared to each other, while the right one corresponds to cardinalities relative to the number of images.

Table 6.1: Multiplicity of class boxes (left) and the number of images in which a class is present (right).

class	box cardinality
vehicle	18,168
vehicle reg. plate	9,934
total	**28,102**

class	image cardinality
vehicle	5,806
vehicle reg. plate	6,867
total	**6,867**

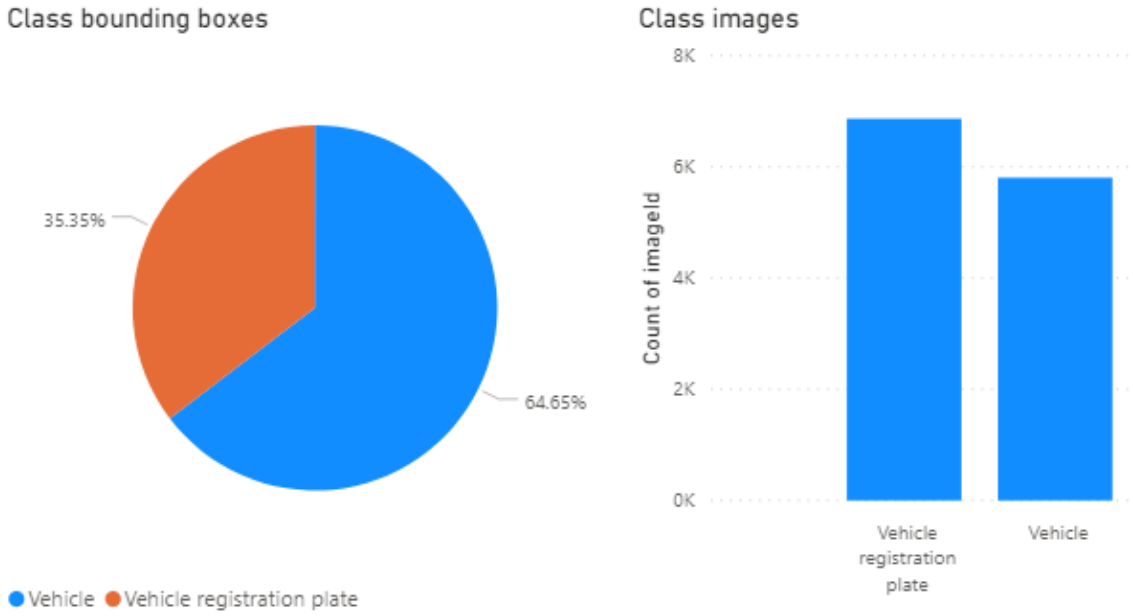

Figure 6.2: Multiplicity of class boxes and the total number of images per class.

Table 6.2 and Figure 6.3 show the average bounding box dimensions for license plates. They occupy an average of 10% of the x-axis and 5% of the y-axis, which means the detector needs wider anchor boxes of this size. The average number plate box is an elongated rectangle, confirming that the data looks like it should (both vehicles and number plates are usually wider along the x-axis).

Table 6.2: Dataset box properties.

type	X_{min}	X_{max}	Y_{min}	Y_{max}
median	0.41	0.58	0.60	0.68
average	0.44	0.55	0.58	0.65

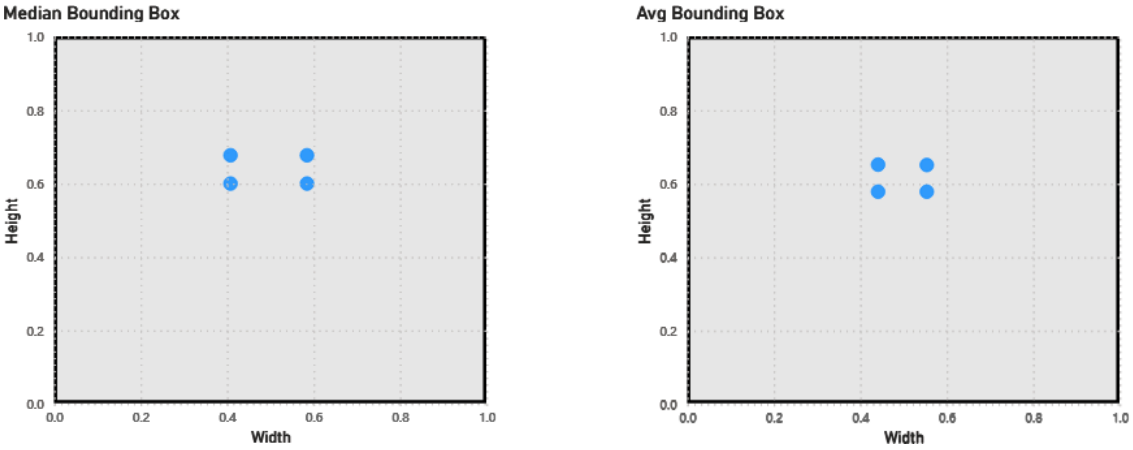

Figure 6.3: Median and average bounding boxes of license plates.

OID has a separate train, validation, and test set. It turned out that 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 to compare the performance of the models based on that subset. I discovered the phenomenon when I was comparing the average boxes. The phenomenon is illustrated by Tables 6.3 and 6.4 and Figure 6.4. A closer look confirmed that some outliers did not cause it - the reason was the difference in the distribution of the box sizes across different subsets.

Table 6.3: Average box coordinates across different subsets.

subset	X_{min}	X_{max}	Y_{min}	Y_{max}
train	0.34	0.65	0.48	0.70
validation	0.21	0.74	0.27	0.65

Table 6.4: Median box coordinates across different subsets.

subset	X_{min}	X_{max}	Y_{min}	Y_{max}
train	0.38	0.61	0.46	0.66
validation	0.32	0.66	0.32	0.62

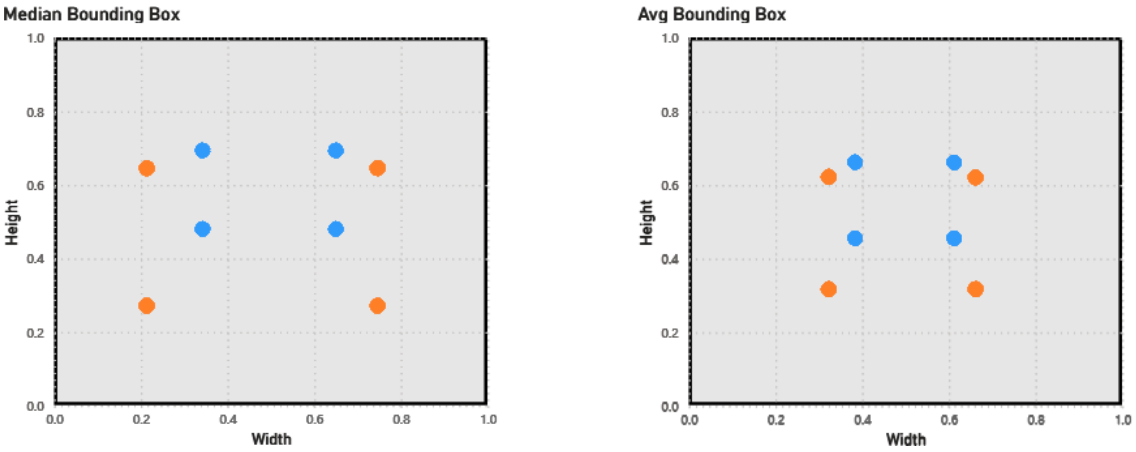

Figure 6.4: Median and average bounding boxes of train and validation sets.

I applied data aggregation and partition to fix this problem. First, all the images and annotations have been aggregated, then saved to TFRecord[7] format. It is a binary file format that stores images and custom labels together. This format's main advantage is that it can be processed rapidly, which is not negligible if the dataset has thousands of instances. I created an encoder Python script to generate the dataset files. Detailed 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 Euclidean division *(n % 14)*. While generating TFRecord batches, I created the corresponding CSV files to analyze later. I selected the validation and test batches, and all the other files became part of the training set. This way, the problem discussed above has been resolved. Table 6.5 shows the average bounding boxes of the test (12 batches), validation (1 batch), and train (1 batch) sets. They are almost identical.

Table 6.5: Average bounding boxes after redistributing the train, validation, and test sets.

subset	X_{min}	X_{max}	Y_{min}	Y_{max}
train	0.38	0.61	0.45	0.66
validation	0.39	0.61	0.47	0.66
test	0.38	0.60	0.47	0.68

6.1.4 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 2,163 (7.7%) test bounding box distribution. There are general dataset division guidelines (like the 70-20-10 rule), from which I deviated. 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 validate image and annotation decoding. Figure 6.5 and 6.6 show some samples from each subset.

Figure 6.5: Training samples with bounding boxes: license plate, vehicle.

Figure 6.6: Samples with boxes: license plate, vehicle. 1st row: validation, 2nd row: test samples.

It is important to note that the dataset is not perfect - there are some inconsistencies. It turned out that some images were incompletely annotated. The problems can be categorized in two ways:

- Visible license plates are not annotated (Figure 6.7 top left).
- Vehicle annotations are entirely missing (Figure 6.7 top right).

One more thing to spot: as a few original OID classes were discarded to keep the new vehicle class close to the cardinality of vehicle registration plates, some types of boxes (like vans) were dropped (bottom row of Figure 6.7). This is partly the reason for some of the missing bounding boxes.

Figure 6.7: Top row: inconsistently missing labels. Bottom row: no vehicle boxes around vans.

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 to avoid bias, I have not made further changes.

6.2 Algorithm

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

I chose 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.

6.2.1 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 evaluate them on the

validation set. When the best model is selected based on its validation performance, it is re-evaluated on an unknown test set to spot if overfitting 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 remains unnoticed if we do not apply this technique. I used the workflow with the different subsets outlined by Figure 6.8.

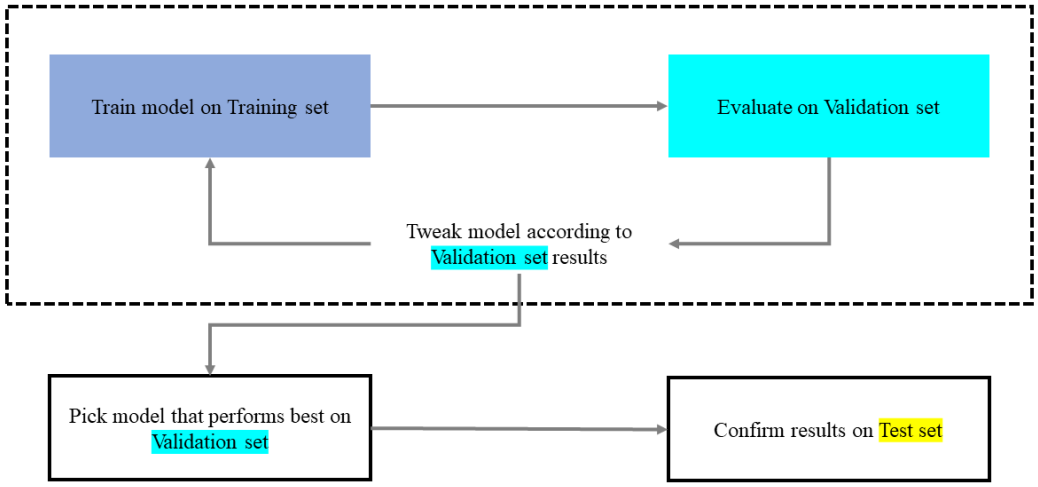

Figure 6.8: Workflow with different subsets. Source: [16]

Training is concluded in a Python/Jupyter environment on a Colaboratory instance. After installing and testing external dependencies, the described steps are applied to each training session. I download base models from TensorFlow's model zoo with or without pre-trained weights (depending on the training configuration). To continue a previous session, I import saved models from Google Drive. The TFRecords are also placed as zip files on Google Drive. They store data on a different server than where the Jupyter instance runs. I download and extract the dataset every time a new session starts (thus, there is no need for network communication during training).

During preprocessing, it is defined how many possible input classes exist. I encode the background as a class (with zero labels to encode non-object image parts as negative examples). I apply fixed shape input image resizing as the dataset has pictures with various resolutions, but the detector expects a static input shape corresponding to the backbone network's input. The anchor box generator properties are also defined during preprocessing.

In the architecture part, the backbone network and the box predictor are configured separately. I experimented with 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, activation function) have been tuned similarly.

It is an exciting topic of what loss functions to use in an object detection problem. I use focal loss[76] for classification and Huber loss (which is very similar to Smooth L1 loss[60]) for localization. It is worth mentioning that I started with sigmoid classification loss and online hard example mining (with three negative object samples per one positive) in a way

like the original SSD paper[52] suggests. However, after researching the topic, I learned about focal loss and RetinaNet[51], 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). Since focal loss takes care of it, hard example mining is unnecessary. Huber loss is used for localization because it handles outliers outside a delta value quite well. Both loss functions contribute equal weight to the total loss.

Post-processing properties, like how many images are allowed on the network's output after non-maximum suppression, are essential questions to be decided. Since there are potentially many cars and license plates on a street scene, I maximized the network output in 100 detectable objects (although an average image from the dataset has four objects).

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. I used four data augmentation techniques (horizontal flip, crop, 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 SSD paper[52]. Image augmentation samples can be seen in Figure 6.9. An image and its bounding box coordinates are always augmented with the same transformation. During an evaluation, augmentation is not used to measure performance objectively and ensure that the results are independent of random transformations.

Figure 6.9: Randomly augmented image samples during training. The same transformation is needed for an image and its bounding boxes.

6.2.2 Optimization

In this part, I compare the models by summarizing their results in tables. Since multiple values were measured, I only display mAPs. All the trained models' metrics (training duration, mAP, loss) can be viewed in a comprehensive table in the Appendix. For length reasons, I use some abbreviations in the Tables of this section. The metrics *@.50, @.75, small, medium, large* are the mAP values at the corresponding IoU thresholds and object sizes based on the COCO evaluation metrics[50]. Model sizes are provided in megabytes.

The single numeric value in the input columns is the pixel width and length of the input image. The lr columns stands for learning rate.

Once the input configurations were selected, I started training the models. In the beginning, to save time, I used pre-trained weights (on the COCO 2017 dataset). I kept the same settings of the networks as they were pre-trained. I applied cosine decay learning rate in each case to prevent too optimistic gradient change at the beginning (started from $\frac{1}{40}$ of the base learning rate, trained like this for $\frac{1}{20}$ of time, then increased it).

Since training a detector is time-consuming, I drew conclusions based on the results and the learning curve's nature after three epochs (around 17,000 images). One epoch is when the entire dataset is passed forward and backward through the network. The best solution is to apply early stopping (training finishes when model performance permanently stagnates/falls back). However, fewer training sessions would have been possible in this case because there is a minimal improvement over a long period at the end of the flattened learning curve.

I used the same anchor box configuration for all the networks. I used five different-sized grids with a total of 12,804 anchor boxes.

6.2.2.1 Architectures

Generally, a two-stage detector takes a classifier and evaluates it in different locations. Because of the separate steps, these are slower architectures but perform better with various-sized objects. TensorFlow Object Detection API currently supports Faster R-CNN from these variants. Although the inference speed is comparable to one-staged architectures, TFLite conversion is not working – therefore, I solely concentrated on one-stage detectors. SSD, RetinaNet, EfficientDet are supported (YOLO is missing and not likely to be added). Since the detector needs to run on smartphones, choosing a relatively small and fast model was the priority. To start with, I tried 3 different options: RetinaNet (ResNet50), RetinaNet (MobileNetV2), and EfficientDet (EfficientNetD0). I did not try VGG-16 because it is no longer used in modern architectures due to its too many parameters. I used all the networks with the smallest possible input size. Table 6.6 shows the comparison of the starter architectures.

Table 6.6: Comparison of the starter architectures.

model	size	input	batch	mAP	@.50	@.75	small	medium	large
ResNet50	241	640	8	**0.389**	**0.678**	**0.404**	**0.116**	**0.394**	**0.531**
EfficientNetD0	42	512	16	0.314	0.581	0.302	0.019	0.321	0.492
MobileNetV2	19	320	16	0.278	0.504	0.269	0.007	0.248	0.492

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 EfficientNetD1 and MobileNetV2 640, and the same batch/step values have been applied. Table 6.7 shows the results.

This comparison shows more balanced results. Unless ResNet50 is the largest model, its performance is not outstanding anymore; EfficientNet seems to outperform it. MobileNet has modest results; however, it is still close to the benchmark.

As inference speed and model size are critical aspects, ResNet should not be considered. EfficientNet is four times larger than MobileNet: the former runs in 54 ms according to

Table 6.7: Comparison of different types with the same input size.

model	size	input	batch	mAP	@.50	@.75	small	medium	large
ResNet50	241	640	8	0.389	**0.678**	**0.404**	0.116	0.394	0.531
EfficientNetD1	63	640	8	**0.391**	0.671	0.400	**0.119**	**0.396**	**0.548**
MobileNetV2	19	640	8	0.379	**0.678**	0.386	0.115	0.386	0.517

TensorFlow's benchmark, while the latter runs in 39 ms. I measured around 150 ms for EfficientDet and 80 ms for MobileNet on a Samsung Galaxy S10 smartphone running on a CPU with dynamic quantization. Their mAPs are relatively close, so I experimented with both networks before committing to one.

6.2.2.2 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. An ideal range of batch sizes is 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 EfficientNet and 24 for MobileNet. When training with batch size 1, I reduced the learning rate by an order of magnitude (from 8×10^{-2} to 8×10^{-3}) to avoid divergence caused by too large gradient updates. Results can be observed in Table 6.8.

Table 6.8: Training results with different batch sizes.

model	size	input	batch	mAP	@.50	@.75	small	medium	large
MobileNetV2	19	640	24	0.386	0.669	0.376	0.108	0.367	0.512
MobileNetV2	19	640	16	0.372	0.666	0.378	0.101	0.381	0.508
MobileNetV2	19	640	8	0.379	0.678	0.386	0.115	0.386	0.517
MobileNetV2	19	640	4	0.377	0.677	0.381	0.116	0.385	0.507
MobileNetV2	19	640	1	0.261	0.535	0.225	0.090	0.327	0.301
EfficientNetD1	63	640	8	**0.391**	0.671	**0.400**	**0.119**	0.396	**0.548**
EfficientNetD1	63	640	4	0.388	**0.681**	0.390	**0.119**	**0.398**	0.541
EfficientNetD1	63	640	1	0.291	0.578	0.262	0.100	0.330	0.373

Based on the results, batch sizes 4 and 8 were the most optimal. Both networks produced the most accurate outputs trained with batch size 8. At larger sizes, a decline is observed, especially in detecting small objects. Regarding the other extreme at batch size 1, gradient update per every image seems to confuse the network and reduce its performance. The trend is MobileNet lags by 2-3% mAP in every configuration behind EfficientNet. On the other hand, MobileNet is four times smaller and significantly faster but slightly less accurate. For these reasons, I decided to use MobileNetV2 for the rest with batch size 8.

6.2.2.3 Optimization algorithm and learning rate

Optimization algorithms[32] 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. Momentum optimization is generally applied to

overcome this issue, introducing and updating gradients' velocity instead of their specific value. Different optimizers exist, such as methods using an adaptive learning rate like RMSprop or Adam.

I tried the Adam, Momentum, and the RMSprop algorithms. Initially, I used the same learning rate (8×10^{-2}) in all three cases. However, training indicators in the case of Adam and RMSprop showed signs of divergence, so I changed their values to their 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. Table 6.9 shows the results.

Table 6.9: Comparison of the different optimization algorithms. All the models are MobileNetV2 variants.

input	lr	optimizer	mAP	@.50	@.75	small	medium	large
640	0.002	*Adam*	**0.392**	**0.682**	**0.410**	0.103	**0.412**	0.546
640	0.08	*Momentum*	0.391	0.671	0.400	**0.119**	0.396	**0.548**
640	0.002	*RMSprop*	0.361	0.643	0.374	0.079	0.394	0.512
640	0.08	*RMSprop*	0.155	0.333	0.130	0.007	0.154	0.213
640	0.08	*Adam*	0.011	0.031	0.004	0.000	0.000	0.018

The last two rows of Table 6.9 show the 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 6.10: Total loss trends of RMSprop (left), Momentum (center), and Adam (right).

Figure 6.10 shows that RMSprop not only lags behind but already converges strongly and slightly improves after 14,000 steps. On the other hand, Momentum and Adam still improve. Out of the two, Adam oscillates more, which may sign a too high learning rate. However, the constant improvement after 20,000 steps (and the fact that I use decay learning rate, but oscillation does not decrease) instead suggests that this is more of 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 training detectors to find 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. Table 6.10 shows the results. 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, resulting in 0.383 mAP. Thus, the final model to take is the one trained with Momentum.

Table 6.10: Comparison of the best results of Momentum and Adam. All three cases indicate the best-performing models before overfitting. All the models are MobileNetV2 variants.

input	lr	optimizer	mAP	@.50	@.75	small	medium	large
640	0.008	*Momentum*	**0.393**	**0.686**	0.391	**0.119**	0.392	**0.545**
640	0.002	*Adam*	0.391	0.685	**0.404**	0.108	**0.404**	0.543
640	0.0002	*Adam*	0.383	0.669	0.380	0.116	0.390	0.528

6.2.3 Model deployment

When the model training is ready, it is time for quantization, TFLite conversion, and to generate the auxiliary structures for deployment.

6.2.3.1 TensorFlow Lite conversion and quantization

During model conversion, it is 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 resulting TensorFlow Lite file at the end of the process can run on Android devices using an interpreter.

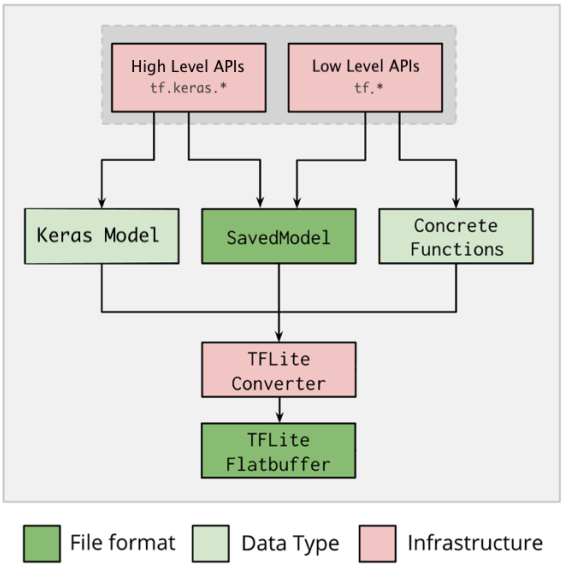

Figure 6.11: TensorFlow Lite conversion process. Source: [78]

6.2.3.2 Input- and output formats

The detector expects 640×640 pixel RGB images on its input. 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 three RGB channels, the values are interpreted as 8-bit integer numbers, which must be between 0 and 255.

The model outputs a Hash Map 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 detection proposals. A brief description of the arrays is as follows:

- *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].
- *Classes*: A float32 tensor of shape [1, num_boxes] with class indices, each indicating the index of a class label from the labels file.
- *Scores*: A float32 tensor of shape [1, num_boxes] with class scores. Values between 0 and 1 represent the probability of the detected class.
- *Number of boxes*: float32 tensor of size 1 containing the number of detected boxes.

6.2.3.3 Auxiliary structures

It is necessary to use an auxiliary structure to interpret the output of the detector. To do this, I created a file containing the names of the output classes in the correct order line by line, including the background class.

The file model_info.txt explains the corresponding model. It contains general information about the network, such as the definition and interpretation of the input and output data formats required for use.

6.2.3.4 Summary

This section dealt with data preparation, architecture selection, training and fine-tuning, and the detector post-production steps. I concluded a total of 184 hours of training with 23 different configurations. The detailed results of the final model can be seen in the table below. Compared to the first initial training from this network family, 1.4% gain has been reached in COCO mAP. The final detector's size is 35.38 MB, its quantized counterpart is 11.19 MB. Some properties of the selected model are shown in Table 6.11.

Table 6.11: Final model training properties. The architecture is MobileNetV2 trained with the Momentum optimizer (learning rate=2×10^{-4}, batch size 8).

input	steps	pre-trained?	duration	mAP	@.50	@.75	small	medium	large
640	$30,000$	Yes (COCO)	8h 50m	0.393	0.686	0.391	0.119	0.392	0.545

Chapter 7

Optical character recognition

This chapter focuses on the OCR (Optical character recognition) part of the ALPR process. I propose a lightweight algorithm that is part of a vehicle identification pipeline capable of running on Android smartphones.

Optical character recognition consists of character localization- and classification. It is responsible for producing machine-encoded text from a text image. First, character segmentation is applied, separating individual characters on the image. The name is a bit misleading as it can be object detection or semantic segmentation. Then follows classification, which is about outputting one machine-encoded character based on an image of a single character.

There are two main approaches regarding Optical Character Recognition with deep learning. Object detection can be used to locate individual characters on images. This solution outputs a character score for each detected box, instantly recognizing tokens in multiline plates. However, there are significant drawbacks in the context of OCR[67]:

- Annotating real-life datasets on a character level is time-consuming.
- Detectors usually struggle with small objects (like characters in text).
- The outputs are character scores, and therefore further pre-processing is needed to get the final text from it.
- When using fixed boxes, a single character can trigger multiple positions. It is possible that "GGOO" is the prediction because the "O" is wide. In that case, we must remove all duplicate tokens. Nevertheless, it can be that the original text would have been "GGO". Then removing the duplicates produces a wrong result.

Another approach can be the sequential feature extraction with convolutional and recurrent layers guided by Connectionist Temporal Classification[29]. In this solution, the convolutional part extracts the necessary features. Then the recurrent part processes them sequentially to output a probability for each time step. The main advantage over the object detection approach is that only an image and the target text need to be provided; CTC handles the rest. Therefore, character positions and width are ignored, which makes it easier to label real-life datasets. An output sequence of a CRNN can be easily translated into a text with either the Greedy or Beam search algorithms, as CTC also indicates the end of the characters with a specific token. However, these solutions can process images along one axis (practically horizontally); therefore, they cannot recognize multiline blocks in standalone versions.

Different approaches exist to convert CRNNs into multiline recognizers. The classical solution can be the Scale Space Technique for Word Segmentation[53], which outputs block positions. It performs relatively well (87%) on handwritten papers. For multiple text block recognition, both an object detector or an image segmentation model can be used first, and their outputs then fed into the CRNN. However, this solution is wasteful as an image must be processed multiple times, and low-level feature extractions are not shared across individual networks. Recently, Wojna et al.[84] proposed an end-to-end model with an attention mechanism. OrigamiNet[86] introduced a one-step model by learning to unfold, transforming existing CRNNs into multiline recognizers. Other approaches still exist, like proposed by WPOD-NET[70], where a modified YOLO detector is used for OCR purposes.

7.1 Data

An adequate amount of data is required to train a deep learning algorithm. There were only a few number plate datasets for object detection (which were missing text labels). The OCR sets neither satisfied the search as they were too specific in domains like handwriting recognition or housing numbers. A good option where there are 15 million annotated images is the database of platesmania.com[59]. However, their API access proved to be expensive even for chunks of 50,000 images, as is their offline license plate generator script.

The nature of license plates varies between countries or even provinces. There are plenty of structures, colors, and fonts (like the modified Mittelschrift in Hungary or the Mandator font in the UK). There can be single-line vs. multi-line versions. Some countries use non-Latin alphabets (like China or Egypt), which makes the task even harder. Figure 7.1 shows some example license plates. Taking these into account, I aim to recognize Latin alphabet characters; 17 currently used number plate fonts have been collected from Australia, North America, and Europe.

Figure 7.1: License plates from Hungary (1st row), Europe (2nd row), and other continents (3rd row).

A few characters are traditionally hard to distinguish in the Latin alphabet, like 0-O-Q, 1-I, or 8-B. This issue is often eliminated by prohibiting letters and numbers on the same character position. However, as plate structures can vary even in a single country, this is not a general solution. Moreover, two different characters may be the same in a font (like 0-O in the Mandatory) or across different fonts. Therefore, I decided that the unique items to distinguish are 0-9 and A-Z excluding O, plus the hyphen (10 + 25 + 1 characters). Other difficulties arise from picture quality and visibility aspects (lighting, contrast, rotation, motion-blur). This diversity suggests a general OCR model has the edge over systems based on hand-crafted feature extraction for each plate type.

For this reason, I implemented a data generator, which approaches the task at the character level. It uses three types of resources to create an output: a list of characters to generate random text, fonts, and overlay images to mimic dirt, ice, and other phenomena. The generator also handles character-level bounding boxes to be converted to output detector training data, producing multiple text blocks. Creating one 200×200 RGB image takes roughly 30 ms without bounding boxes and 61 ms with them. The generator applies data augmentation techniques to enrich data diversity. Algorithm 1 shows the individual steps of the pipeline.

Algorithm 1 Data generator pipeline

procedure Generate($args$)
random font and image overlay from $args$
$bckgColor$ = random value [0...255] in every channel
$image$: create with defined dimensions from $args$ and $bckgColor$
$generatedTexts$ = $args[numBlocks]$ random texts in the range of 1...10.
$lastBoxCoords$ = [0,0,0,0]
$insertedTextIndices$ = []
for $text$ in $generatedTexts$ **do**
    $textColor$ = text and background colors must have at least 20% diff in one channel (randomly selected which channel satisfy this)
    $textImage$: create text image from $text$, with a background color identical to $bckgColor$
    if $textImage$ is bigger than $image$ in any dimension **then**
        $textImage$: resize to fit onto the $image$
    $textImage$: resize ratio with a value in the resize range (typically 0.85...1)
    $textImage$: random aspect ratio change, heigh, and width offset
    if ($lastBoxCoords[3] < boxYstart$) || (($lastBoxCoords[2] < boxXstart$) && ($lastBoxCoords[1] < boxYstart$)) **then**
        $image$ += $textImage$
        $lastBoxCoords$ = [$boxXstart$, $boxYstart$, $boxXend$, $boxYend$]
        $insertedTextIndices$ += currentBoxIndex
$image$: random rotation and perspective transformation
$image$: resize to the required size
$image$ += random overlay with offset and rotation
$image$: random brightness, contrast, sharpness, gaussian blur (within $args$ constraints)
$image$: downscale between a range from $args$, then scale back
$image$: normalize from [0...255] to [-1...1), convert to float32
$label$ += $text$ which Id is in $insertedTextIndices$
$label$: encode $label$ to a number label with dictionary from $args$
return $image$, $label$

As the validation samples are also generated, the images must be equally distributed across different evaluation sets. A single evaluation epoch is set to contain 10,000 images. With this size, there is a maximum of +/-0.015 deviation in loss across multiple validations. Figure 7.2 shows some random one-line generated sample.

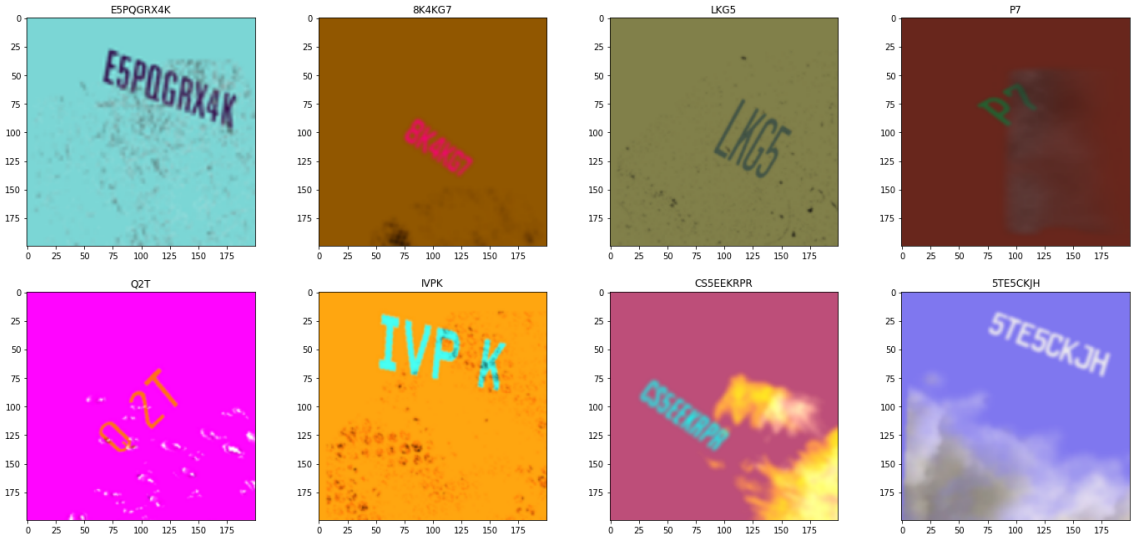

Figure 7.2: One line generated samples with labels on top. Hard overlays have been removed from the validation set so that the completely illegible characters do not affect results.

7.2 Sequential approach

My first OCR approach is based on feed-forward recurrent convolutional networks (CRNN) that produce fixed-size arrays of character probabilities.

7.2.1 Connectionist Temporal Classification

Connectionist Temporal Classification[29] is used to compute model loss. The cost function has a barely documented Keras implementation, which I wrapped into a layer to make it easily detachable after training. This layer has two inputs: model prediction (*batch_size, max_timesteps, num_characters*) and ground truth encoded as numbers (*batch_size, max_len*). Besides the 36 identifiable items, two more numbers are indicating the non-character token and the unknown character. Inputs must be the same length in a single batch. If a label's text is shorter than the maximum length, padding is expected with non-character tokens. The padding ensures that the model can be trained with greater batches than one, as multiple labels can be merged into a single, same dimensional array. The layer deduces the length of each label at runtime before computing its cost. To do that with arbitrary batches, I had to put a loop into the layer. The implemented TensorFlow loop is 30% slower than the Python counterpart, but it can be placed on a TensorFlow graph.

7.2.2 Building blocks

A custom convolution layer is defined with Convolution → Nonlinearity → Dropout → Batch Normalization (Figure 7.3). These types of layers are stacked on top of each other in the residual blocks. The implementation allows to specify the type of activation function, the proportion of dropout, and whether to use Batch Normalization or not (when enabled, it adds bias to the output, so the convolution operation's bias is disabled in that case).

Every convolution operation uses "SAME" padding. It applies padding so that the whole input gets fully covered by the filter and specified stride. The name comes from that, for stride 1, the size of the output is the same as the input. This partly solves the skewed kernels and feature-map artifacts issue induced by the operation[11].

The main parts of the network are the residual blocks which consist of three components. The residual stage contains an arbitrary number of convolution layers with the same input/output size and channels. This block of layers is defined with a skip connection between the input and the output to overcome the vanishing gradient problem, which that means the block's input directly influences its output through a "gradient highway", not just through convolutions. After that, an upscaling convolution layer doubles the number of channels (it is the only layer that gradients cannot bypass; therefore, it is a bottleneck). It is followed by spatial downscaling by a factor of two with pooling. The structure is illustrated by Figure 7.3. In this configuration, a block with input dimensions (height, width, channels) has an output of $(\frac{height}{2}, \frac{width}{2}, channels \times 2)$. My implementation was inspired by the ResNet[30] architecture.

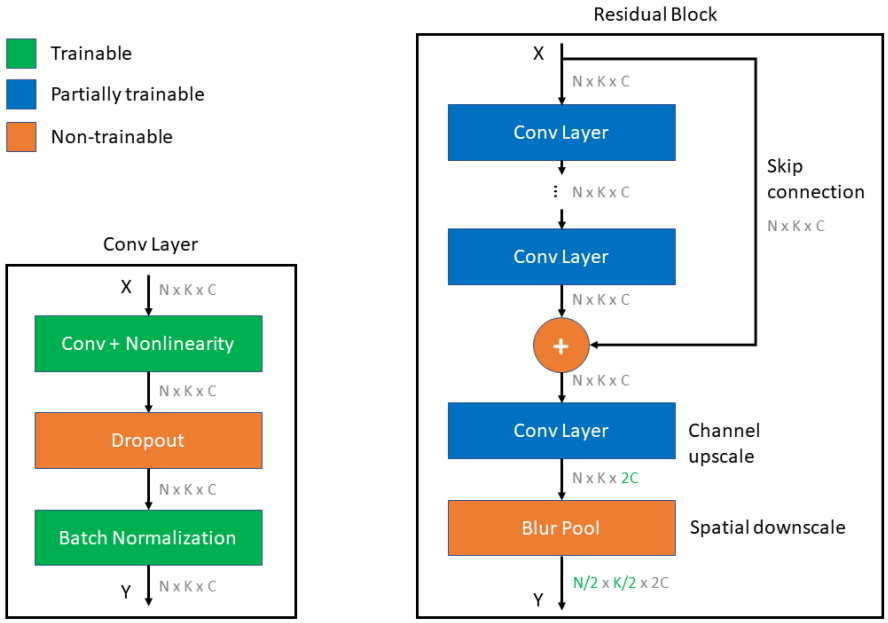

Figure 7.3: Network building blocks with the convolution layer (left) and the residual block (right).

In the recurrent part of the network, two Long Short-Term Memory[33] layers are stacked on top of each other responsible for sequential processing. At the time of this work, GRU TensorFlow Lite support was unavailable, so I could only use LSTM for smartphone inference. In the network's first layer, the height and width dimensions of the input are swapped so that the width is the first dimension in the model. Thus, recurrent layers process inputs horizontally. A dense layer follows the last LSTM layer with a dimension of $(input\_width \times downscale\_factor, num\_characters)$, which is the network's output matrix. The high-level operation of the model is illustrated in Figure 7.4.

The best model has a downscale factor of three (number of residual blocks) and an input image width of 500 pixels, which means 63 steps overall. One window has a width of 8 pixels. Each window must be assigned to a real or an empty character token. It means that net 31 characters can be recognized to have enough space for characters and empty tokens in the output matrix.

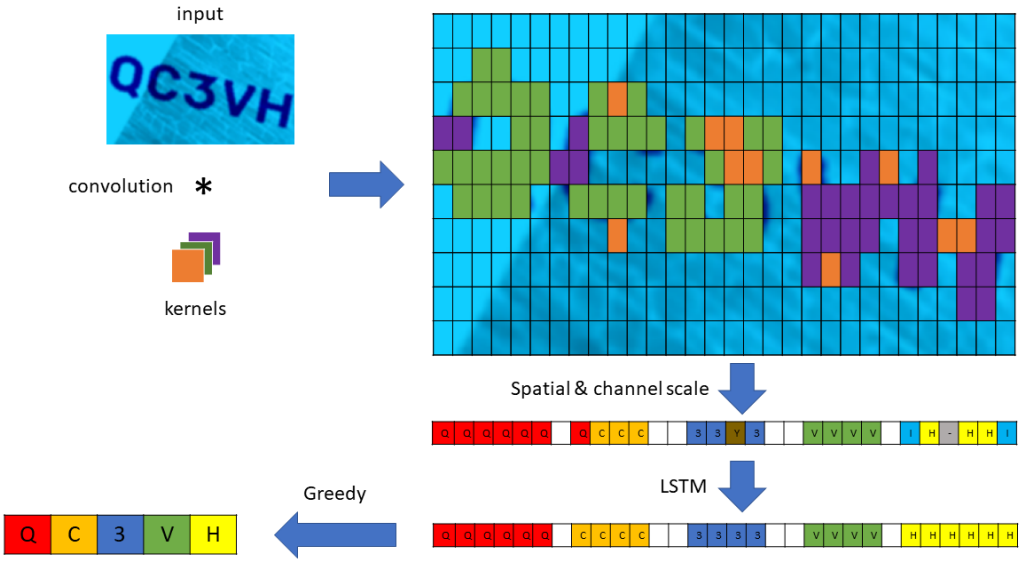

Figure 7.4: Simplified representation of the network's key processing steps.

7.2.3 Greedy search

In training, the CTC layer handles raw model outputs and ground truth labels, then calculates loss to backpropagate. However, this is not required for inference. Multiple approaches are available to decode raw outputs of the last dense layer. The Greedy solution keeps the maximum index at each step, then merges the same tokens between empty tokens. This way, multiple recognitions triggered by the same character are eliminated. The next step is to remove the empty tokens and decode the sequence with the model's vocabulary. That step produces the final text. Other approaches, like Beam search, iteratively create text candidates, arranges them based on their probability scores, and keeps the best ones after every iteration. The most probable beam at the end is returned as a result. This algorithm can be highly computation-intensive depending on the number of beams. Word Beam Search[68] takes this idea to another level: it constrains beams with a prefix tree of existing words. This solution can be helpful in situations where the words to recognize are from a specific vocabulary, but in the case of plate recognition, such a semantic assumption does not help. For these reasons, I implemented the Greedy algorithm.

7.2.4 Experiments

As a starting point, the model is trained with two residual blocks with a single convolution layer in each. The first block starts with 32 input channels, and the batch size is 8. Adam optimizer[43] is used with the default parameters (1×10^{-3} learning rate). One epoch is equal to 100,000 images, and early stopping is applied with five epoch patience. This model reached the optimum after 13 epochs with a validation loss of 0.2298. I tried different batch sizes to find the most optimal as shown in Table 7.1.

Table 7.1: The effect of batch sizes. Smaller batches converged faster at the beginning of training, but their improvement flattened out earlier at worse sub-optimal points. Batch size 128 was the largest that could fit into the memory.

batch	epochs	loss
8	8	0.2121
16	29	0.1545
32	22	0.1134
64	55	0.0855
128	51	**0.0848**

7.2.4.1 Shift-invariance

Zhang et al.[87] pointed out that modern CNNs lack shift-invariance since max-pooling, average-pooling, and strided-convolution ignore Nyquist's sampling theorem. The proposed solution is the use of blur-pooling, where spatial downsampling happens (Figure 7.5). I implemented blur-pooling with separable kernels and replaced all the other pooling layers with them. Although they claim that the solution is compatible with the layers listed above, I wanted to keep the number of layers as low as possible. The shift-invariant model slightly improved in loss, but a lot in the speed of convergence. Therefore, I continued to work with this model as the new benchmark.

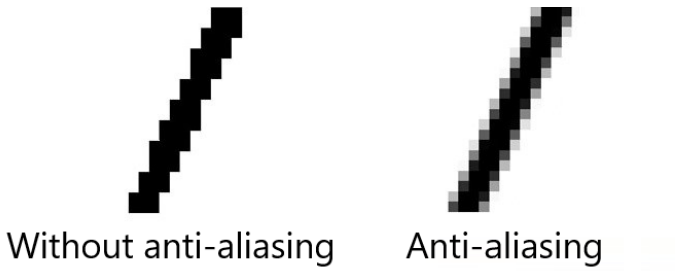

Figure 7.5: The effect of anti-aliasing when scaling down a sloped line. Blur-pooling prevents the appearance of too sharp artifacts.

7.2.4.2 1D convolution

I experimented with using 1D convolutional layers instead of the LSTM cells. This way, the model does not have any recurrent part, only residual blocks with convolutions. The more straightforward structure without loops results in faster runtimes. However, the training became significantly longer, and the final result lagged behind the benchmark (0.08 vs. 0.1). The primary source of error was that one character triggered multiple different classifications based on the sequential processing's actual position. While LSTM can effectively smooth out 1D feature maps ("OCOO" → "OOOO"), it looked like 1D convolution had a more challenging time. Therefore, I no longer used this approach.

7.2.4.3 Hyperband optimization

Hyperparameter optimization has been realized with the Hyperband algorithm[49]. Another option considered was Bayesian optimization. I chose the former because adaptive

Bayesian methods do not handle discrete, independent, unordered values well, and training would have lasted longer (since one configuration is done when the model training stops). Hyperband is a fast guided random search that works like a knockout sports competition: it generates combinations and trains them for a few epochs. After that, the top-performing part of the models is trained further, while the others are discarded. The iteration repeats until all the remaining models stop learning or there is a clear winner.

Hyperband strongly filters models based on early training performances. Therefore, variable batch size or learning rate was not possible with this algorithm because smaller batches tend to outperform larger ones initially, but not in the long run. The selected parameters to optimize were the number of features of the first block (which is then doubled by every consequent block), kernel size, whether residual connections and batch norm layers are needed, dropout rate, number of blocks, convolutional layers in blocks, and type of activation function. A batch size of 32 was applied during training with every model, as instances with more blocks and layers should also fit into memory. Therefore, the previous batch 32 model served as a baseline. The best configurations found by the optimization algorithm are shown in Table 7.2. Sample inference results can be seen in Figure 7.6.

Table 7.2: Top 3 configurations found by Hyperband. Only the top 2 and the baseline were trained until convergence (3rd model training has been stopped earlier). Columns: model, features, kernel, residual, batch norm, dropout, blocks, layer per block, activation, epochs, loss.

model	feat	ker	res	bn	drop	b	l	act	epochs	loss
1st	64	3	✓	✓	0.09	3	3	ReLU	39	**0.0753**
2nd	64	5	*X*	✓	0.2	2	2	ReLU	35	0.0862
3rd	64	3	*X*	✓	0.3	2	3	ELU	15	0.1554
baseline	32	3	✓	✓	0.1	2	1	ReLU	22	0.1134

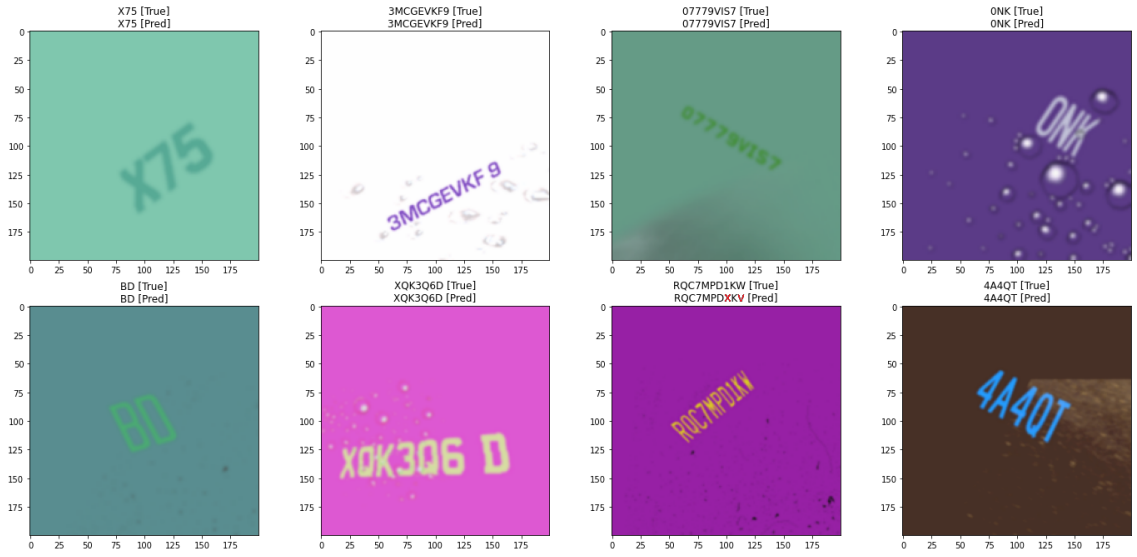

Figure 7.6: Inference results of the best performing network. Incorrect characters are marked as red.

7.2.5 Multiline recognition

There are numerous approaches to process number plates with multiple lines. One can be the usage of object detection to detect individual text blocks on an image. One powerful model based on this approach is the CRAFT[12] text detector. Another approach is text segmentation, which is not necessarily done by neural networks. In both cases, however, the recognition of text blocks and characters are separated steps and the weights used for them are not shared. I experimented with a model that merges these steps. The key idea is to add some layers to the end of the network that learns to unfold the two-dimensional feature map into a tall, one-dimensional feature sequence. This approach changes the main axis (the horizontal reading of text changes to process vertical features). To accomplish this, I implemented a modified Origami[86] module that includes convolution layers and resizing with bilinear interpolation. The pseudo code of the module is shown by Algorithm 2.

Algorithm 2 Pseudo code of the Unfolding module.

```
procedure Unfolding(inDim, inChannel, outChannel, numBlocks, numLayers)
vertical = inputDim[0]
horizontal = inputDim[1]
for i = 0; i <= numBlocks; i++ do
    vertical *= 2
    horizontal /= 2
    Resizing(vertical, horizontal, interpolation=bilinear)
    for j = 0; j <= numLayers; j++ do
        Conv(inChannel, inChannel)
Conv(inChannel, outChannel, (1, 1), (1, 1))
xDownFactor = pow(2, numBlocks)
xRemain = inDim[1] % xDownFactor > 0
xLastKernelSize = (inDim[0] // xDownFactor) + xRemain
Conv(outChannel, outChannel, (1, xLastKernelSize), (1, xLastKernelSize))
```

The original solution was not ideal for this task. I had to remove the recurrent part of the original network (LSTM layers) because they introduced instability while training. I replaced pooling with strided convolution, increased the number of resizing steps, reduced the spatial rescaling ratio, and increased the number of convolution layers. After these modifications, the module started working as intended. The best loss was 0.9517 on images with a maximum of five rows of texts after 73 epochs (0.0991 on single line images). It is far from an optimal solution, but all I had to do is to add a module to the end of an existing model. Unfolding also scales well, as the runtime does not change even when processing multi-row text images.

However, unfolding has its drawbacks. The original model had a size of 640 KB, while this module increased it to 3.138 MB. Multiplying the size of a model already reading single-line texts five times seems wasteful (even so, it is less than using a separate detector and a single line recognizer). Another problem is that unfolding struggles with rotated texts. It jumps through the lines while reading when rotation is close to 45 degrees. Therefore, in my opinion, this solution is not adequate for this task unless I apply an extra horizontal transformation step.

7.3 Object detection approach

RCNNs are fast and efficient on single-line texts but not ideal for multiline ones. To solve this, text block segmentation or detection can be used, which means an extra operation in the pipeline. Histogram-based text block segmentation can work, as the blocks are well separated in the case of license plates, but this is less effective on rotated number plate images. To remedy this, a license plate rectification operation is needed. To keep the number of processing steps low, I tried to solve the OCR task by character-level object detection, which can immediately handle multi-line texts.

I implemented the object detector from scratch for this task instead of using the TensorFlow Object Detection API. The API constraints a few things, like data encoding, available architectures, and usable backbone networks. With my implementation, I could understand the underlying concepts more thoroughly, and I could fine-tune the whole process at a lower level.

I implemented a RetinaNet-based solution. I choose this architecture because the FPN provides multi-scale feature map outputs and the loss function that handles the class imbalance problem. It is also beneficial because of the decoupled classifier- and regressor heads, which add minimal latency but improve accuracy. I started with predictor heads each containing two convolutional layers, similar to the original publication. RetinaNet is an anchor-based detector, so I gained insight into this kind of operation during the implementation. The object detector training minimizes the loss function, but the evaluation protocols help understand how the algorithm performs at localization, classification, and general detection for different object sizes. Object Detection API's evaluation protocol implementation is deeply embedded in the API. It cannot be used as a standalone module. The whole API would have been added as a dependency to use their protocols, which I wanted to avoid. Due to time constraints, I could not implement a proper evaluation protocol, I used localization- and classification losses to draw conclusions.

7.3.1 Experiments

As a baseline, I created a small custom CNN as a backbone. I considered the original ResNet50 too complex for this task because we only need to detect characters that are not as complex and diverse objects as vehicles. The CNN has the following structure: at the beginning, a few feature blocks deepen the feature map but keep the input size. The subsequent spatial blocks keep upscaling the input feature maps by two while downscaling them spatially by the same factor. Each spatial block has an arbitrary number of convolutional layers inside with a residual connection between them. The last convolutional layer does the channel upscaling, meaning that the skip connection cannot bypass that layer, similarly to Figure 7.3. I used the last three different-sized convolutional feature maps from the backbone as the inputs of the FPN. I used the input of 256×256 because, with the power of two input sizes, both pooling and nearest-neighbor downscaling (within the FPN) have the appropriate output size (the residue is treated differently, which is a problem in the case of inputs not in the power of two). An overview of the model is illustrated in Figure 7.7.

7.3.1.1 Loss and anchor configuration

The default RetinaNet implementation uses anchor boxes of sizes *32, 64, 128, 256, 512*. As the input images are only 256×256 pixels, I had to modify this. I used anchor boxes

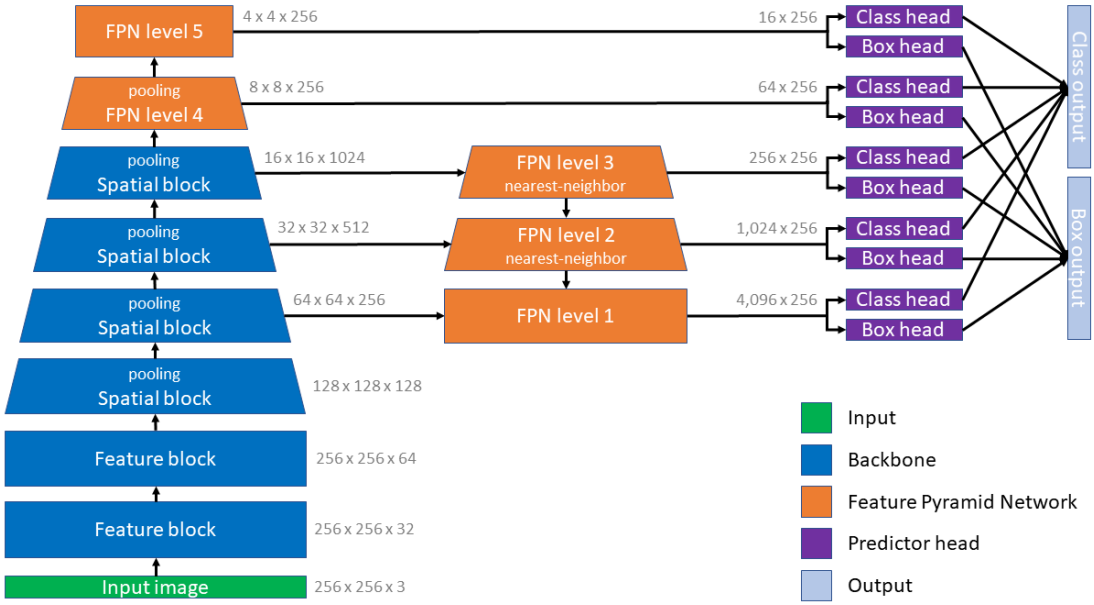

Figure 7.7: The starter detector model. The predictor heads multiply the number of predictions by nine because of three different aspect ratios and three different scale ratios per anchor position.

of sizes *16, 32, 64, 128*, and *256*. Similarly to the original configuration, I used three different aspect ratios ($\frac{1}{2}$, *1, 2*) and three scale ratios ($0, \frac{1}{3}, \frac{2}{3}$) for each anchor box size. With this configuration, the model had 49,104 anchor boxes for 256×256 inputs. The smallest anchor box the model can find is 16 pixels wide or long. This is $\frac{1}{16}$th of the input size. I consider it small enough because the input image is a license plate snippet, roughly 10-14 times as wide as one character on it. Thanks to the scale ratio, slightly smaller boxes can still be detected.

The models did not output any box prediction at the first few training sessions. It turned out that the loss function needed other parameters. The first problem was that the localization loss dominated the total value: there are 36 different classes in the OCR task, but the original COCO dataset has 80. An erroneous classification is more severely penalized if there are more classes by focal loss. I multiplied the localization penalty by 0.6 for downscaling. The other problem was that values for well-classified samples and wrongly empty-classified ones were very similar. The models learned to output nothing, thus minimizing the error. I tried different values to widen this gap and chose 0.85 and 1.2 for alpha and gamma parameters (instead of the original 0.25 and 2.0). After these modifications, the models could learn predicting boxes.

7.3.1.2 Backbone

I wanted to use as small network as possible. I tried a backbone with four convolutional layers as a starting point. This model could only localize huge boxes properly. This was because two convolutional layers could not generate a feature map that could be used to localize and classify objects. As I increased the number of layers, the detector got better. I added two feature blocks with one convolutional layer at the beginning of the network. After four layers, the starter model could output proper predictions (the last feature map

responsible for huge objects). Increasing the number of layers helped the model converge, but the number of parameters got huge: I constrained the maximum output feature map in 512 dimensions to keep it low. In my opinion, this amount was enough to distinguish 36 classes properly.

I tried different feature map dimensions in the first convolutional layer; this value is then doubled for each block. The dimensions were *16, 32*, and *64*. The best results were with the value 32, but a model with only a 16-dimensional feature map produced similar results. Since this value significantly affects the size of the network, I chose the lowest because the deeper feature maps did not substantially affect performance as shown in Table 7.3.

Table 7.3: Detectors with different 1st layer feature dimensions.

1st feature map dimension	epochs	loss
16	22	1.4052
32	26	**1.4026**
64	28	1.4112

Next, I tried different feature map dimension constraints because this greatly influences the number of parameters. I experimented with values *512, 256*, and *128*. Training a model with 256 maximum feature dimensions did not decrease performance, but the model with only 128 could not achieve the same error rate. Hence, I chose 256 as the upper limit.

7.3.1.3 Shift-invariance

Initially, I used all the models with max-pooling. I experimented with the custom implemented blur-pooling layers, which worsened the performance in this case. I don't know the exact reason of this, but it could probably be that the feature edges of different objects got too blurred. This was not a problem with sequential models, where a character usually triggered multiple parts in a sequence, which CTC loss could handle. In object detection, it caused trouble in object localization. Therefore, I used max-pooling in the following models. Figure 7.8 shows the effect of different pooling layers applied to an image containing larger objects.

7.3.1.4 Predictor heads

My implementation uses decoupled classifier- and localizer heads to produce the network output. I experimented with different input feature dimensions (coming from the FPN) and the number of layers.

I tried the predictor head input dimensions of *256, 128*, and *64*. The highest value matches the maximum dimensionality in the CNN backbone. Lower values can be interpreted as a feature downscaling in the FPN. Value 128 was the best choice because it could produce the best error rate in pair with dimension 256, leading to a significantly smaller network. Dimension 64 was too few for this problem. Table 7.4 summarizes the results.

Table 7.4: Comparison of predictor heads with different input dimensions.

input dimension	epochs	loss
64	22	1.9347
128	26	**1.4033**
256	28	1.4052

Figure 7.8: The effect of different pooling layers. The two variants were applied to the same 900×865 image eight times. The 2nd, 4th, 6th, and 8th downscaled images are shown. Blur-pooling was used with kernel size three and stride two (top) and max-pooling with stride two (bottom).

RetinaNet uses two inner convolutional layers in the predictor heads before the last one reshaping the features to the output. I tried one, two, and three inner layers. A single layer was not enough. The loss was higher than with the other two approaches. The deepest predictor heads performed similar, but I chose the ones with two inner layers because of the similar performance but lower number of parameters. Results can be seen in Table 7.5.

Table 7.5: The effect of different number of inner predictor head layers on performance.

number of inner layers	epochs	loss
1	20	1.4456
2	26	1.4033
3	22	**1.3967**

Sample inference results of the best model can be seen in Figure 7.9. False positive predictions mainly cause the error. In the output processing, I used the Combined NMS algorithm with an IoU value of 0.5. To solve the false positive phenomenon, I could use the original NMS. Still, in that case, rotated images are likely to produce IoU values greater than the threshold among different objects, which carry the risk of dropping true positive objects close to each other.

The TensorFlow Object Detection API model runs in 80 ms, while the proposed model's inference time is 317 ms. This phenomenon is caused by several things. The first is the number of anchor boxes. The API model uses around 12,000 anchor boxes, while the own implementation has four times more. This is because small anchor boxes are needed to detect small characters, drastically increasing the number of anchors. The other reason is that my implementation reuses the same predictor heads across different feature levels on the computational graph, leading to a smaller network size but longer inference time.

A development option could be to implement a YOLO-like detector, which uses objectness score to handle class imbalance, thus using a simpler, less parameter-sensitive loss function. Also, implementing an anchor-free version of the model could speed up inference time because the output tensor with roughly 50,000 items is a performance bottleneck. The model size can also be sacrificed using separate predictor heads for each feature level for better runtime.

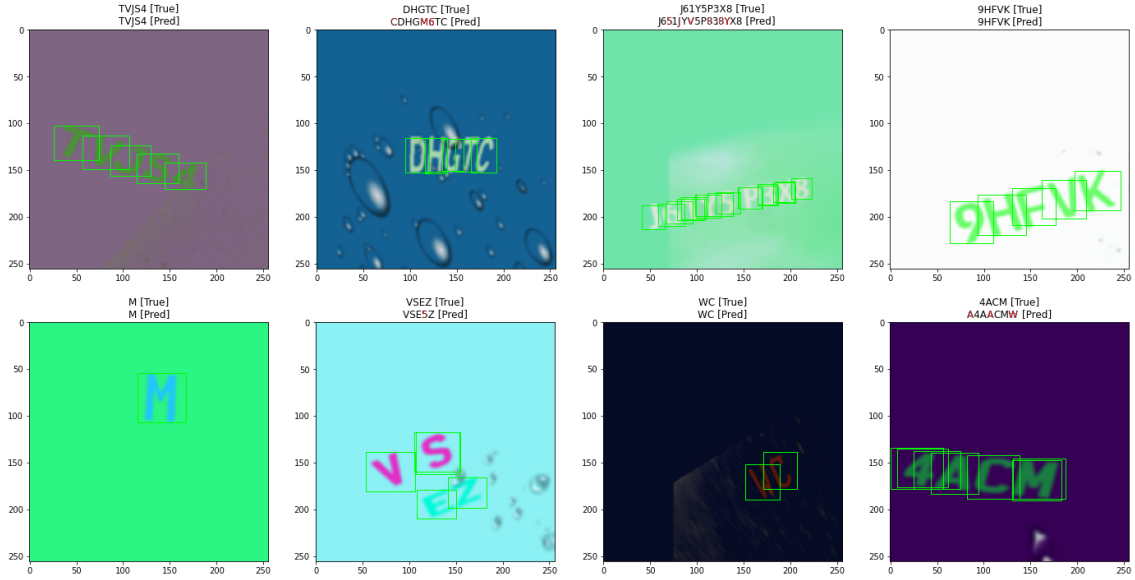

Figure 7.9: Inference results of the best performing network. Incorrect detections are marked as red.

7.4 Comparison

This chapter provided a solution for the OCR task with both the sequential- and object detection approach. The sequential model is fast and simple but cannot process multiline texts efficiently. Object detection can handle them easily but requires roughly 2-3 times more parameters. However, its most significant problem is the inference time. With an anchor-free approach, the detector can be significantly faster, with which it can approach the runtime of the RCNNs. This is a future development opportunity. However, the present anchor-based detector is an order of magnitude slower than the sequential model, and more error prone due to the false positive predictions, so I chose the RCNN approach. The comparison of the two models can be seen in Table 7.6.

Table 7.6: Comparison of sequential and object detection OCR models.

model	host	input	parameters	size [KB]	inference [ms]
sequential	CPU	50x500x3	1,325,734	639	37
object detection	CPU	256x256x3	3,557,484	4,541	317

Chapter 8

System Overview

In this chapter, I overview the complete system. I explain the main design decisions, and some more exciting implementation details.

8.1 ALPR pipeline

The applied pipeline is shown by Figure 8.1. It compresses the traditional tasks into as few steps as possible. The structure is like the one used in WPOD-NET[70], with some differences. First, a RetinaNet[51] detector is used for both vehicle and plate localization (instead of a YOLOv2[61] vehicle-only detector). The rectification step is excluded from our pipeline, which is a drawback in plate image quality. However, WPOD-NET's rectification supports only one plate scale, so adding it would not be a general solution, in our opinion. On the other hand, it would increase the computational load as it must run separately for each detected vehicle. The second step is to filter possible plate objects based on location and confidence scores. The third is to cut plate objects from the original image.

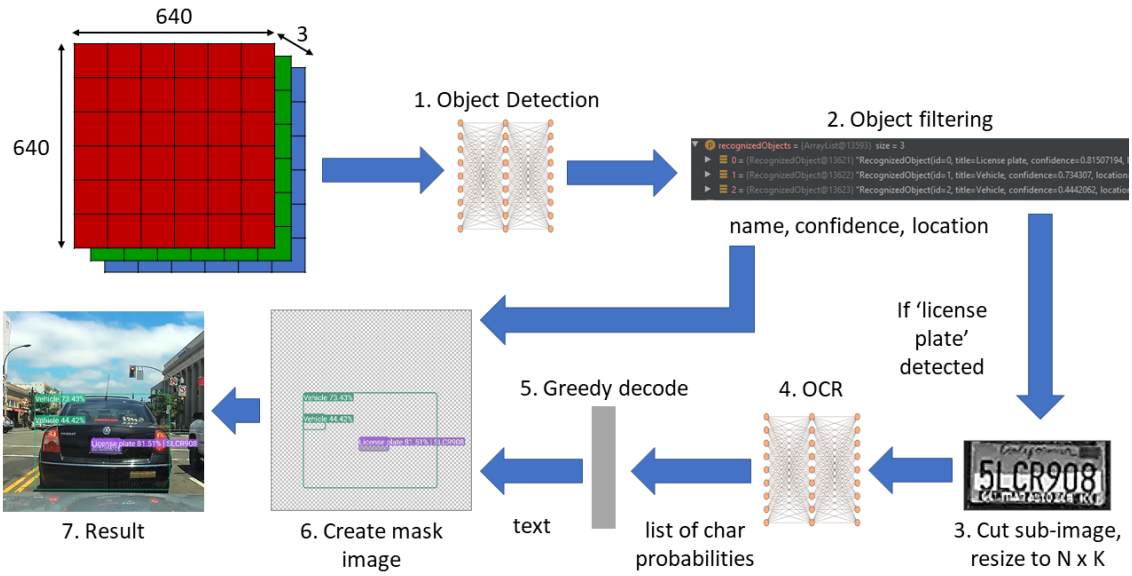

Figure 8.1: The applied ALPR pipeline.

The pipeline models are dynamically quantized TensorFlow Lite converted networks. On a Samsung Galaxy S10 smartphone, where an image contains only one license plate, the average runtime is roughly 160 ms. The processing steps are as follows:

- image preprocessing (9 ms)
- detector inference (75 ms on GPU)
- cutting and resizing the license plate image segment (3 ms)
- OCR inference (37 ms on CPU)
- Greedy decoding (<1 ms)
- creating mask image (33 ms)

This runtime may change depending on the number of plates on a picture, as the OCR steps must be repeated on each license plate image segment.

Table 8.1 summarizes the runtimes of different OCR models. Besides the best-performing network, I also tested the initial architecture (batch 128). It runs in 37 ms on the device's GPU. Keeping in mind the runtime, it is worth sacrificing marginal quality ($+9.5 \times 10^{-3}$ loss) to acquire a 1.5× speed-up. As a reference, Google's MLKit Vision Text Recognizer[28] runs in 40 ms on the same plate image with a single line of text. The MLKit model is a multiblock recognizer. However, when more than one block is present, its runtime increases drastically.

Table 8.1: Running time in milliseconds for OCR models.

model	host	input	size [KB]	loss	inference [ms]
batch 128	CPU	50x500x3	639	0.0848	37
MLKit	CPU	50x500x3	?	?	40
batch 128	GPU	50x500x3	639	0.0848	50
Hyper1st	CPU	50x500x3	4,447	0.0753	56
MLKit	CPU	200x200x3	?	?	72
Unfolding	CPU	200x200x3	3,133	0.0991	221
Unfolding	GPU	200x200x3	3,133	0.0991	372

8.2 Frontend

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

The application can receive inputs from two different sources. The first option is to load an image from the storage. In this case, the image is processed only once, and the meta-information (GPS location, UTC timestamp) is extracted from the image's Exif (Exchangeable image file format) content. This format is supported by almost any type

of smartphone (if not, fallback values are used, which cannot be reported to the server). This is important because otherwise, a stolen vehicle image from the past could overwrite a more recent recognition on the server when reporting the match. The other option is to process the live camera feed. For this, the current camera frame is fed into the pipeline. When the results are ready, the current live frame is processed, meaning that frames are discarded while the pipeline is processing. The live feed results are accompanied by the device's location and system UTC timestamp information. If they are unavailable, fallback values are used. In both cases, image preprocessing steps are applied shown by Figure 8.2. Some images of the application are shown in Figure 8.3.

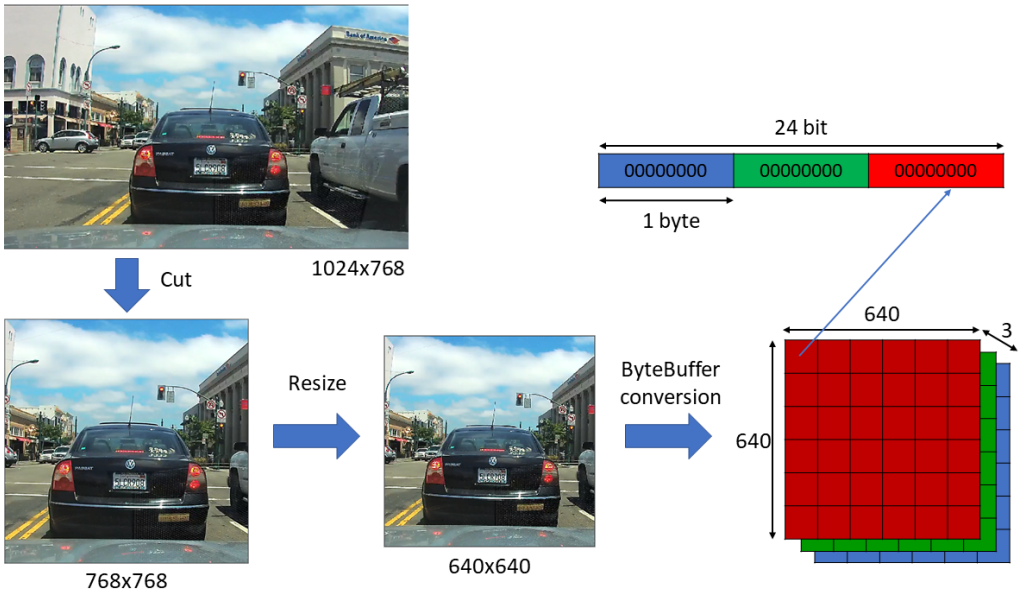

Figure 8.2: Image preprocessing steps before feeding the pipeline.

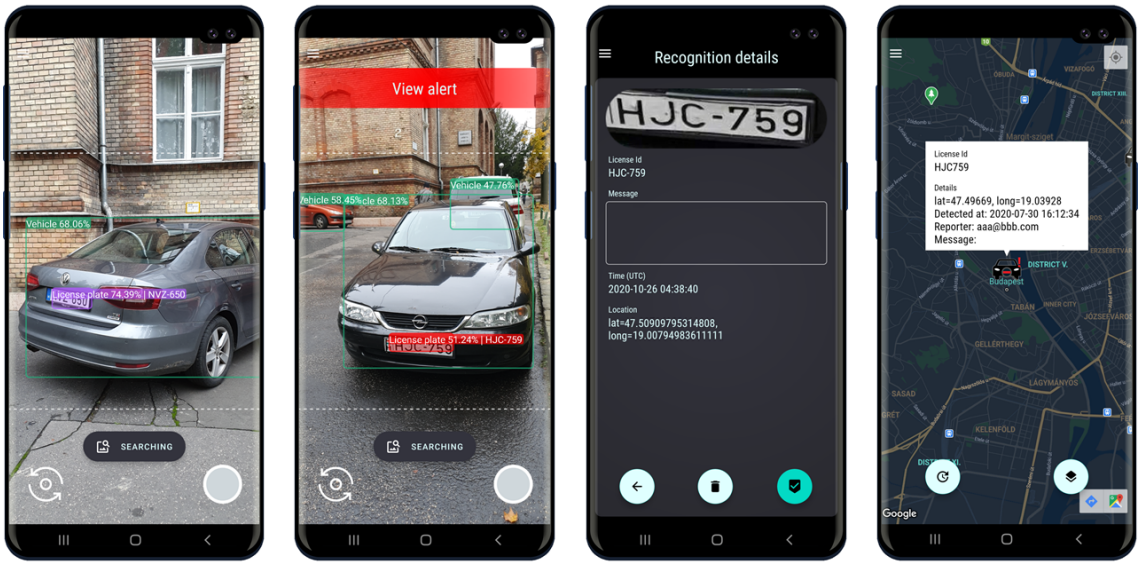

Figure 8.3: Live feed recognition (1st screen) and an identified stolen vehicle (2nd screen). Details can be inspected and a message can be added to a recognition (3rd screen). When a vehicle is reported, its last known position is visible on the map (4th screen).

8.2.1 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 Views, and the business logic components can be easily separated.

There are sub-layers in the model level of the application. I explain the hierarchy through the steps of reporting a single 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 ReportActivity, which has a ViewModel storing the UI state. The report item gets stored in a list wrapped in a LiveData object (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 ReportRepository. It hides further data operations (database handling, API communication) from the outside. When it receives a new report, it transforms it into a format stored in the Report table and persists with ReportDAO (data access object) to the database. After that, it calls ApiService to send the report to the server. When the response arrives from the API, ReportRepository updates the corresponding item in the database. Until the operation is unsuccessful, the user sees the report as pending. Pending items can be deleted or re-sent at any time. Figure 8.4 shows a high-level overview of the sub-layers and modules.

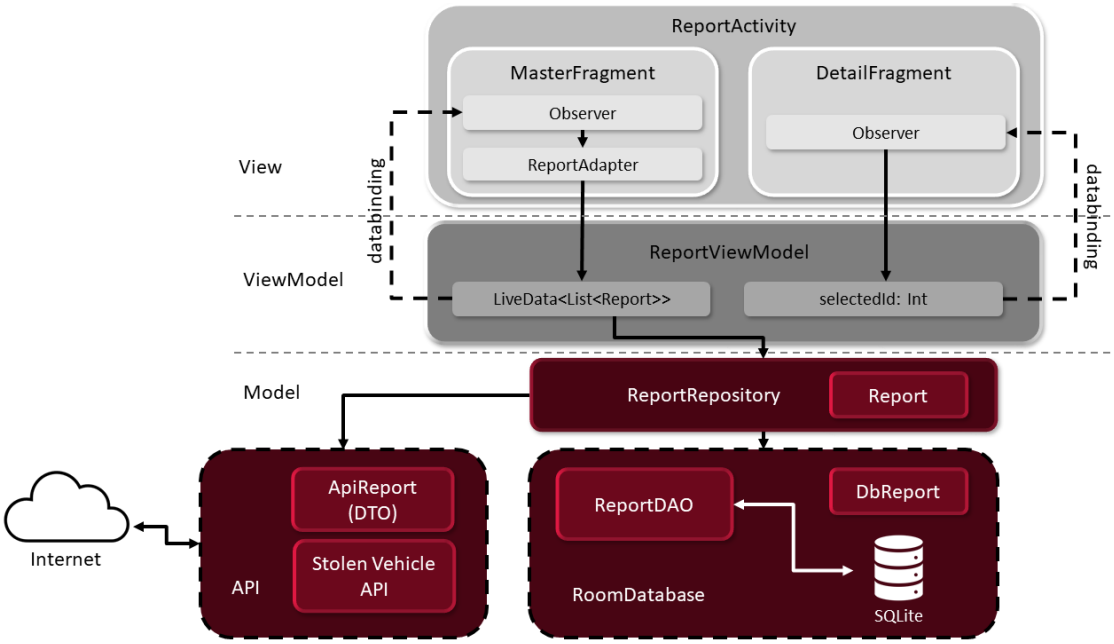

Figure 8.4: MVVM layers in the application.

8.2.2 Database

Beyond reports, the application stores several other information in its local relational database (list of stolen vehicles, account information, metadata). Outside the relational database, the application stores user preferences as key-value pairs. The complete database schema can be seen on Figure 8.5. The persisted data schema differs from the in-memory data representation in order to be independent of higher-level modules. The schema of the data transfer objects (DTOs) is different for the same reason. This way, in case of API- or database schema changing, only the corresponding module needs to be updated, resulting in maintainable code.

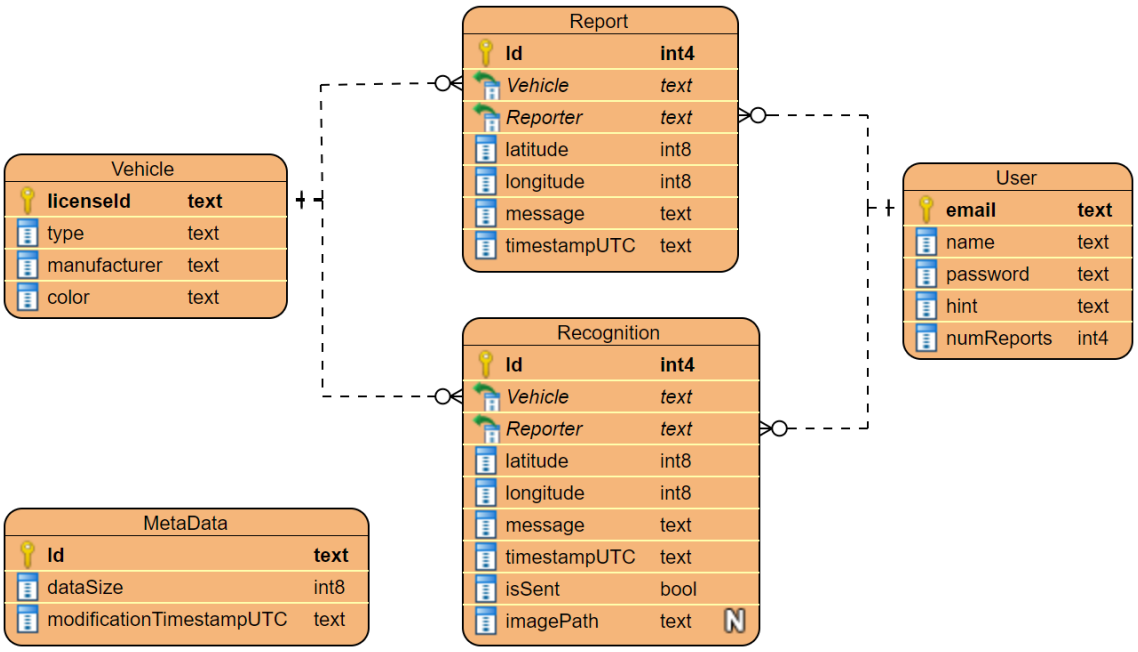

Figure 8.5: Client application database schema.

8.2.3 Vulnerabilities

I would like to draw attention to some vulnerabilities of the application. By manipulating the Exif information of an image (changing coordinates, setting a timestamp to a later time), false recognition can be uploaded to the server, which can overwrite the current vehicle position record. Unfortunately, there is no robust solution to detect whether an image's Exif content has been modified or not by the time of this work. Also, an image editor can easily edit a recent photo (with valid Exif information) to contain a stolen vehicle in it. When using the live camera feed, pointing to an image of a stolen vehicle can also trigger a false report. These are opportunities for abuse due to the nature of image-based recognition. To prevent them, images inducing the report could also be uploaded to the server for further inspections. When a report's metadata is fallback or invalid, the application refuses to send them to the server.

8.3 Backend

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 when querying the server.

8.3.1 Architecture

The server has a three-layered architecture (Figure 8.6). Because it is responsible for API service and data storage, it does not have a separate View layer (only a simple web-based UI is available). Instead of the UI layer, the communication layer through which the API services can be accessed. It is loosely coupled with other layers. 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.

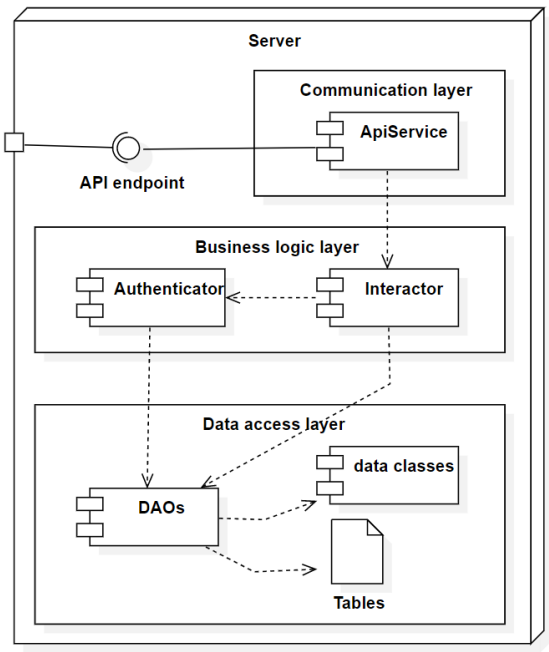

Figure 8.6: Structure diagram of the server.

8.3.2 Database

Since I decided to use a self-created database, 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. The table contents are in JSON format (like in the case of MongoDB). The server uses the Gson library to encode and decode JSON files. Figure 8.7 represents the schema of the stored objects.

There are tables of stolen vehicles, current reports, and user accounts. To these contents, history files(write only) are available. 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, an Event table records system logs (also write-only).

The database serves requests from memory, making API responses fast because there is no need to wait for I/O operations. The memory content is synchronized with the corresponding table in the background. It is a viable solution as too many data records are never stored on the server (the images are not uploaded). I examined the most significant JSON object type (report) to validate this. One record takes up 173 bytes in the memory. The server needs 173 MB memory when 1 million records exist. It is a severe overestimation, though, as the stolen vehicles list obtained by web scraping typically has just a few thousand items (in the case of Hungary). That is the maximum number of records that the in-memory database ever has (when every stolen vehicle was detected). As the history content is stored persistently, extensive API usage neither saturates the memory. Given the scalability of the database, one server instance could handle all European countries.

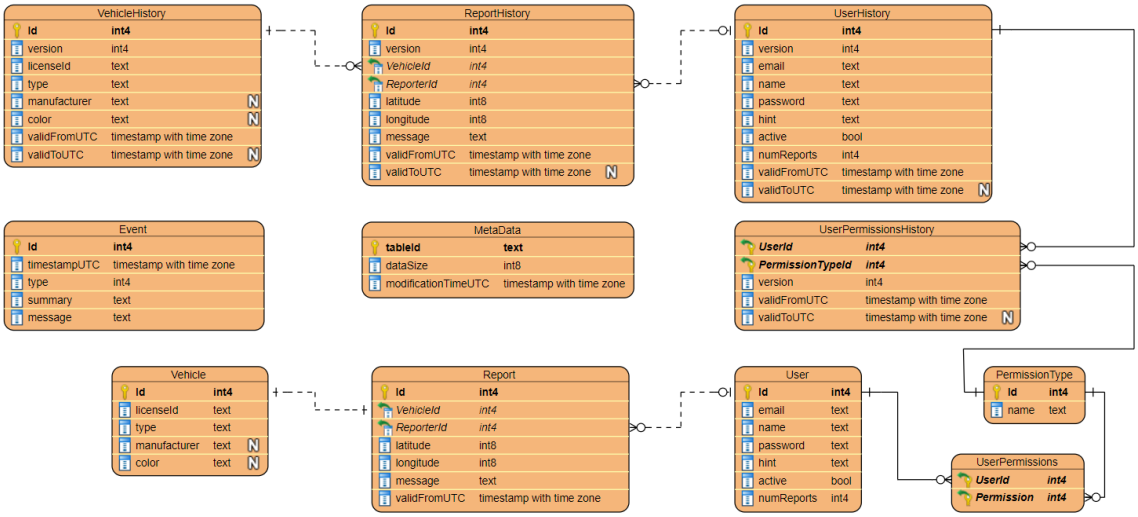

Figure 8.7: Server data schema.

8.3.3 API

The API is divided into five parts: *Vehicles, Reports, Report history, Self*, and *Users*. These names are also the corresponding endpoint prefixes. All query endpoints have similar actions and a unified calling convention. All actions are subject to specific permissions, evaluated before serving. There is also a status page describing the API and its interface.

8.3.4 Permission

The nature of stored data (location and timestamp of stolen vehicles) could potentially allow abuses, so there is an allowlisting role-based permission model. Users with specific roles are eligible to execute various operations. There are ADMINISTRATOR, API_REGISTER, SELF_MODIFY, API_GET, and API_SEND permissions. Users can be uniquely identified by their inner server Id or email address.

- API_GET lets an authorized account query report.
- API_SEND makes it possible to send recognitions to the server.

- SELF_MODIFY is needed to prevent blocklisted 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 administrator can revoke permissions, delete a user, or deactivate and blocklist it. An administrator can also register a new user with specific permissions.
- The default user (e.g., client application users) has an account with 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, this user is utilized. The only API permission is registration. Although someone can detect vehicles on-device, they cannot report them or see current reports. This API_REGISTER role prevents anyone outside of the client application from registering. The default user can create a new account with SELF_MODIFY, API_GET, and API_SEND permissions.

8.3.5 Vulnerabilities

The server stores sensitive data so attempts to abuse should be considered. Passwords are stored on the server in an encrypted format. No one can access user passwords - although administrators can block or delete users. Data access is based on an allowlisting model, meaning that anything which is not explicitly allowed is forbidden. The sensitive API endpoints (user actions, data uploads, and queries) are only available through HTTPS communication. Every database access is through predefined functions with a predefined list of allowed parameters, meaning that injection attacks are not possible. Users outside the client application cannot register, as the front-end contains a protected default user solely used to register new accounts.

If the attacker is an inner user, its behavior can be monitored: if there are too many reports from the same account, or the report locations/timestamps are suspicious, administrators can revoke the API_SEND permission or even block the user. When an account is blocked, it cannot query or upload. A blocked user cannot delete itself or re-register, meaning that it cannot access the server with the same email address anymore. If a badly-formatted report is received, the server discards it. If the report's metadata is invalid (non-existing GPS coordinates, future UTC timestamps), the server also discards it. If an attacker is an administrator, it can delete the current database state. However, the state can always be restored with the help of the history content, which cannot be deleted. Administrators cannot access other users' sensitive content (although they can delete accounts). Other administrators can block an administrator.

Chapter 9

Summary

In this work, I discussed the general task of automatic license plate recognition, then examined best practices for both object detection and optical character recognition. I created and trained deep learning-based algorithms to detect number plates and recognize texts on them. Then an entire ALPR pipeline was constructed. Then, I put together the different modules to form a complete vehicle identification system. In the end, I presented the fundamental concepts of my system's frontend and backend.

Further work would focus on many aspects of the proposed system. For example, the optical character recognition step could be improved by using the object detection approach instead of sequential processing. This way, a single algorithm could solve multi-line text recognition. It is necessary to optimize the detector for inference speed: explicit predictor heads for each feature map on the computational graph would speed up the runtime. Another crucial step would be the usage of an anchor-free approach, like YOLOX[22], which would reduce the output tensor computation by orders of magnitude and simplify the pipeline. In my opinion, RetinaNet[51] loss is less robust than the YOLO approach, where a standalone objectness score is used to address the class imbalance problem. Therefore, I would like to try that variant and compare the results. The license plate and vehicle localization step could be optimized using a custom detector instead of the TensorFlow Object Detection API when these improvements are complete. Appropriate evaluation metrics are required to compare different models fairly. Therefore, implementing the COCO or a custom metric is also needed to develop low-level detectors further. Using a network that recognizes the rotation angle of number plates could be a significant improvement because this would eliminate the need for a standalone plate rectification step. The OCR model could be much easier and faster, as it would not need to handle rotated characters anymore. Another development option would be to identify the number plate's country of origin and use a structural presumption. The current system is not constrained to license plates of specific countries. Still, if the territorial information could be extracted, that would make further processing easier (exactly how many characters are needed, whether a character is an "O" or a "0" based on the location on the plate).

A fair comparison of widespread ALPR systems and my solution is also essential. At the time of this work, I have not been granted access to the benchmark datasets (SSIG SegPlate and UFPR-ALPR).

Acknowledgements

I would like to express my deepest thanks to Dániel Pásztor for his valuable support as a scientific advisor and ideas throughout the whole project. I am incredibly thankful to all the people who took the time to review this work out of their busy lives. In particular, I would like to thank Daniel Benedí García and António Elias for reviewing the chapters. I am infinitely grateful to my mother, Szilvia, for encouraging and supporting me. By listening while presenting the concepts, she helped me clarify my thoughts and improve this work. Last but not least, I would like to thank my grandfather Péter for all his incredible support and guidance, without whom I would not have gotten this far.

Bibliography

[1] Automatic number-plate recognition. https://en.wikipedia.org/wiki/Automatic_number-plate_recognition. Accessed: 2021-09-19.

[2] Vehicle registration plates of the netherlands. https://gaz.wiki/wiki/pt/Vehicle_registration_plates_of_the_Netherlands. Accessed: 2021-09-19.

[3] Flatbuffers. https://google.github.io/flatbuffers/. Accessed: 2021-05-22.

[4] Open neural network exchange. https://onnx.ai/. Accessed: 2021-11-28.

[5] Projection profile method. https://www.geeksforgeeks.org/projection-profile-method/. Accessed: 2021-09-19.

[6] Pytorch. https://pytorch.org/. Accessed: 2021-11-28.

[7] Tfrecord and tf.train.example. https://www.tensorflow.org/tutorials/load_data/tfrecord. Accessed: 2021-12-03.

[8] Tensorflow. https://www.tensorflow.org/, . Accessed: 2021-11-28.

[9] On-device training with tensorflow lite. https://www.tensorflow.org/lite/examples/on_device_training/overview, . Accessed: 2021-12-17.

[10] Tensorflow object detection api. https://github.com/tensorflow/models/tree/master/research/object_detection, . Accessed: 2021-11-28.

[11] Bilal Alsallakh, Narine Kokhlikyan, Vivek Miglani, Jun Yuan, and Orion Reblitz-Richardson. Mind the pad – cnns can develop blind spots. *arXiv:2010.02178v1*, 2020.

[12] Youngmin Baek, Bado Lee, Dongyoon Han, Sangdoo Yun, and Hwalsuk Lee. Character region awareness for text detection. *arXiv:1904.01941v1*, 2019.

[13] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv:2004.10934v1*, 2020.

[14] Anders Christiansen. Anchor boxes - the key to quality object detection. https://towardsdatascience.com/anchor-boxes-the-key-to-quality-object-detection-ddf9d612d4f9, 2018. Accessed: 2021-12-01.

[15] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[16] Google Developers. Validation set: Another partition. https://developers.google.com/machine-learning/crash-course/validation/another-partition. Accessed: 2021-12-03.

[17] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. http://host.robots.ox.ac.uk/pascal/VOC/, jun 2010. Accessed: 2021-12-01.

[18] Flickr. Image of a mobile alpr device. https://c1.staticflickr.com/9/8284/7737315772_d3b5dcd9ee_b.jpg. Accessed: 2021-09-19.

[19] Arpad Fodor. Image recognition by machine learning and application on android platform. https://diplomaterv.vik.bme.hu/hu/Theses/Kepfelismeres-gepi-tanulassal-es-alkalmazasa, 2019. Accessed: 2021-12-01.

[20] Kotlin Foundation. Kotlin programming language. https://kotlinlang.org/, . Accessed: 2021-05-22.

[21] Python Software Foundation. Python programming language. https://www.python.org/about/, . Accessed: 2021-05-22.

[22] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. *arXiv:2107.08430v2*, 2021.

[23] F.A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2, 1999. DOI: 10.1049/cp:19991218.

[24] Ross Girshick. Fast r-cnn, 2015.

[25] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2014.

[26] Gabriel Resende Gonçalves, Sirlene Pio Gomes da Silva, David Menotti, and William Robson Schwartz. Benchmark for license plate character segmentation. *Journal of Electronic Imaging*, 25(5):1–5, 2016. URL http://smartsenselab.dcc.ufmg.br/wp-content/uploads/2019/02/JEI-2016-Benchmark.pdf.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[28] Google. Mlkit text recognition. https://developers.google.com/ml-kit/vision/text-recognition. Accessed: 2021-04-23.

[29] Alex Graves, Santiago Fernandez, Faustino Gomez, and Jurgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, 2006.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv:1512.03385v1*, 2015.

[31] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018.

[32] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning: Lecture 6. pages 5–30, 2014.

[33] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computation 9(8):1735-1780*, 1997.

[34] Jonathan Hui. map (mean average precision) for object detection. https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173, 2018. Accessed: 2021-12-01.

[35] ANPR International. History of anpr. http://www.anpr-international.com/history-of-anpr/. Accessed: 2021-09-19.

[36] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. *arXiv:1506.02025v3*, 2016.

[37] JetBrains. The ktor framework. https://ktor.io/. Accessed: 2021-05-22.

[38] G. Jocher, A. Stoken, J. Borovec, NanoCode012, ChristopherSTAN, L. Changyu, Laughing, A. Hogan, lorenzomammana, tkianai, AlexWang1900 yxNONG, L. Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, Hatovix, J. Poznanski, L. Y., changyu98, P. Rai, R. Ferriday, T. Sullivan, W. Xinyu, YuriRibeiro, E. R. Claramunt, hopesala, pritul dave, and yzchen. Ultralytic's yolov5. https://github.com/ultralytics/yolov5, 2020. Accessed: 2021-04-23.

[39] Sambasivarao. K. Non-maximum suppression (nms). https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c, 2019. Accessed: 2021-12-01.

[40] Renu Khandelwal. Different iou losses for faster and accurate object detection. https://medium.com/analytics-vidhya/different-iou-losses-for-faster-and-accurate-object-detection-3345781e0bf, 2021. Accessed: 2021-12-01.

[41] Kartik Khare. Json vs protocol buffers vs flatbuffers. https://codeburst.io/json-vs-protocol-buffers-vs-flatbuffers-a4247f8bda6f. Accessed: 2021-05-22.

[42] Achraf Khazri. Image of a fixed alpr system. https://towardsdatascience.com/automatic-license-plate-detection-recognition-using-deep-learning-624def07eaaf. Accessed: 2021-09-19.

[43] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980v9*, 2017.

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[45] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *IJCV*, 2020.

[46] R. Laroca, E. Severo, L. A. Zanlorensi, L. S. Oliveira, G. R. Goncalves, W. R. Schwartz, and D. Menotti. A robust real-time automatic license plate recognition based on the YOLO detector. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, July 2018. DOI: 10.1109/IJCNN.2018.8489629.

[47] Rayson Laroca, Evair Severo, Luiz A. Zanlorensi, Luiz S. Oliveira, Gabriel Resende Gonc¸alves, William Robson Schwartz, and David Menotti. A robust real-time automatic license plate recognition based on the yolo detector. *arXiv:1802.09567v6*, 2018.

[48] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. DOI: 10.1109/5.726791.

[49] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv:1603.06560v4*, 2018.

[50] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context. https://cocodataset.org/, 2015. Accessed: 2021-12-01.

[51] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018. Accessed: 2021-11-28.

[52] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. *arXiv:1512.02325v5*, 2016.

[53] R. Manmatha and Nitin Srimal. Scale space technique for word segmentation in handwritten documents. http://ciir.cs.umass.edu/pubfiles/mm-27.pdf. Accessed: 2021-04-24.

[54] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-ranked-retrieval-results-1.html.

[55] Szemenyei Márton. Deep learning in visual informatics: Lecture 8. pages 3, 11–20, 2021.

[56] Mithi. Vehicle detection with hog and linear svm. https://medium.com/@mithi/vehicles-tracking-with-hog-and-linear-svm-c9f27eaf521a, 2017. Accessed: 2021-12-01.

[57] Xin Nie, Meifang Yang, and Ryan Wen Liu. Deep neural network-based robust ship detection under different weather conditions. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 47–52, 2019. DOI: 10.1109/ITSC.2019.8917475.

[58] Jiangmiao Pang, Kai Chen, Jianping Shi, Huajun Feng, Wanli Ouyang, and Dahua Lin. Libra r-cnn: Towards balanced learning for object detection, 2019.

[59] PlatesMania. Platesmania.com. http://platesmania.com. Accessed: 2021-04-22.

[60] PyTorch. torch.nn.smoothl1loss. https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html. Accessed: 2021-12-03.

[61] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. *arXiv:1612.08242v1*, 2016.

[62] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv:1804.02767v1*, 2018.

[63] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *arXiv:1506.02640v5*, 2015.

[64] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016.

[65] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[66] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[67] Harald Scheidl. An intuitive explanation of connectionist temporal classification. https://towardsdatascience.com/intuitively-understanding-connectionist-temporal-classification-3797e43a86c. Accessed: 2021-04-22.

[68] Harald Scheidl, Stefan Fiel, and Robert Sablatnig. Word beam search: A connectionist temporal classification decoding algorithm. https://repositum.tuwien.at/retrieve/1835. Accessed: 2021-04-23.

[69] Sighthound. Sighthound. https://https://www.sighthound.com/. Accessed: 2021-09-19.

[70] Sergio Montazzolli Silva and Claudio Rosito Jung. License plate detection and recognition in unconstrained scenarios. *ECCV 2018*, 2018.

[71] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[72] Rekor Systems. Openalpr. https://www.openalpr.com/. Accessed: 2021-09-19.

[73] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[74] Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. *arXiv:1911.09070v7*, 2020.

[75] TensorFlow. Protocol buffers. https://www.tensorflow.org/lite/performance/post_training_quantization, . Accessed: 2021-05-22.

[76] TensorFlow. tfa.losses.sigmoidfocalcrossentropy. https://www.tensorflow.org/addons/api_docs/python/tfa/losses/SigmoidFocalCrossEntropy, . Accessed: 2021-12-03.

[77] TensorFlow. Tensorflow eager execution. https://www.tensorflow.org/guide/eager, . Accessed: 2021-05-22.

[78] TensorFlow. Tensorflow lite converter. https://www.tensorflow.org/lite/convert, . Accessed: 2021-12-04.

[79] TensorFlow. Tensorflow post-training quantization. https://www.tensorflow.org/guide/eager, . Accessed: 2021-05-22.

[80] Shoot Tokyo. How i shoot. https://shoottokyo.com/shoot/. Accessed: 2021-09-19.

[81] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio, 2016.

[82] Lilian Weng. Object detection for dummies part 3: R-cnn family. https://lilianweng.github.io/lil-log/2017/12/31/object-recognition-for-dummies-part-3.html, 2017. Accessed: 2021-12-02.

[83] Lilian Weng. Object detection part 4: Fast detection models. https://lilianweng.github.io/lil-log/2018/12/27/object-detection-part-4.html, 2018. Accessed: 2021-12-02.

[84] Zbigniew Wojna, Alex Gorban, Dar-Shyang Lee, Kevin Murphy, Qian Yu, Yeqing Li, and Julian Ibarz. Attention-based extraction of structured information from street view imagery. *arXiv:1704.03549v4*, 2017.

[85] Shuo Yang, Ping Luo, Chen Change Loy, and Xiaoou Tang. Wider face: A face detection benchmark. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[86] Mohamed Yousef and Tom E. Bishop. Origaminet: Weakly-supervised, segmentation-free, one-step, full page text recognition by learning to unfold. *arXiv:2006.07491v1*, 2020.

[87] Richard Zhang. Making convolutional networks shift-invariant again. *arXiv:1904.11486v2*, 2019.

[88] Zhong-Qiu Zhao, Peng Zheng, Shou tao Xu, and Xindong Wu. Object detection with deep learning: A review. *arXiv:1807.05511v2*, 2019.

[89] Xingyi Zhou, Dequan Wang, and Philipp Krähenbühl. Objects as points. In *arXiv preprint arXiv:1904.07850*, 2019. Accessed: 2021-11-28.