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

May 24, 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.

Budapest, 2021. május 24.

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, jó megoldásokat kínál eddig nehezen algoritmizálható feladatokra. Ezzel párhuzamosan az okostelefonok robbanásszerű sebességgel fejlődnek. Az ARM CPU-k közelmúltbeli fejlődése és a GPU-k megjelenése az asztali PC-khez hasonló számítási teljesítményt nyújtanak.

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, és javaslatot teszek a valós idejű felhasználhatóságot szem előtt tartó pipeline-ra. Ezután létrehozok és betanítok deep learning algoritmusokat a teljes járműazonosítási folyamat megvalósításához. Később megtervezek egy Android alkalmazást, amely futtatja a kifejlesztett felismerési lépéseket. Ezt követően létrehozok egy szerver alkalmazást is, ahonnan a felhasználók frissíthetik magukat, és ahol bejelenthetik a járműveket. 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 good 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 design an Android application running the developed pipeline. After that, I create a server application from which clients can update themselves and where users can report vehicles. Finally, I explain the applicability and the limitations of the prepared system.

Chapter 1

Introduction

Chapter 2

Technologies

In this section, I present the technologies used to build the deep learning models and to create Android and server applications.

2.1 Deep learning

I used the Python[7] programming language in deep learning-related tasks. Python is an interpreted, dynamically typed, high-level language with an object-oriented approach. Numerous libraries offer deep learning repertoire in Python, however, two libraries stand out from these: PyTorch and TensorFlow. In the following, I only describe the technologies I used during 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 PyTorch

2.1.2 TensorFlow

TensorFlow is an open-source machine learning platform developed by Google Brains. It provides rich Python and C APIs and works well with the popular Keras neural network library. TF also works well with the Colaboratory environment where GPU/TPU-based works are easy to build without local resources. In 2019, TensorFlow 2.0 has been 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[8], where operations are immediately evaluated without building graphs.

TensorFlow uses Google's protobuf[5] 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[6]. 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 aimed at deploying models on smartphones and IoT devices. Trained models can be transformed into this format with the TFLite converter. TFLite uses the Flat Buffer[1] 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. It is also why TFLite models cannot be trained.

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 are the quantization[9] options used to achieve further performance gains (2-3x faster inference, 2-4x smaller networks). I used full integer quantization in which all the model maths are int8 based instead of the original float32.

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

2.1.3 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 either created on or loaded from Google Drive. This is where the project saves its checkpoints and training logs. Realms are the sandboxes of notebook instances, which makes it possible to run them parallel seamlessly. In a realm, two types of projects can be - single training or hyperparameter optimization. For the latter, I used Keras Tuner.

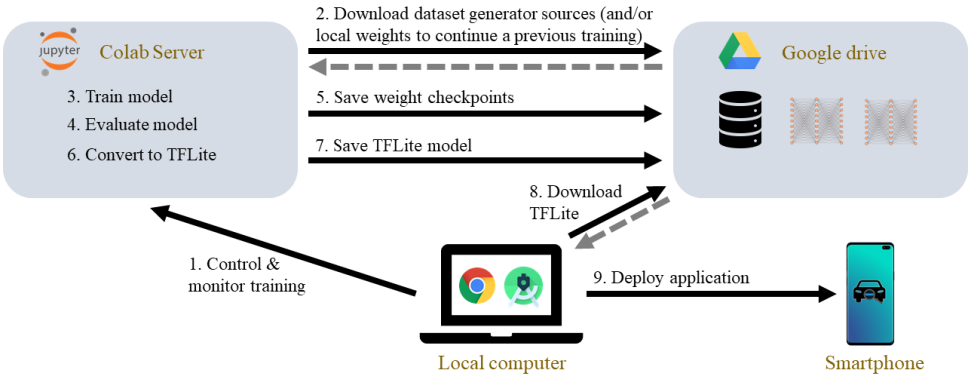

Figure 2.1: High-level overview of the training environment.

2.2 Application

I used the Kotlin programming language[2] in both the frontend and the backend. Kotlin is a statically typed, cross-platform language with type inference. In addition to the object-oriented approach, it also contains functional programming tools.

2.2.1 Frontend

Android provides an extensive application development ecosystem. I used the AndroidX namespace elements, which replaces the previous Support Library since Android 9.0. It is part of Android Jetpack, a collection of components for which the platform promises long-term support. Of these libraries, the application extensively uses the CameraX API to manage device cameras. The ViewModel component acts as a moderator between the user interface and the business logic. The app contains a relational database implemented by the Room Persistence Library, which provides an abstraction layer over SQLite to allow for more robust database access. 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[3] 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

Automatic License Plate Recognition

3.1 Approaches

3.2 Difficulties

Chapter 4

Object Detection

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

4.1 Related Work

4.2 Data

4.3 Model

4.4 Evaluation

Chapter 5

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.

5.1 Related Work

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[21]:

- Annotating real-life datasets on a character level can be 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. What if "GGOO" is the prediction because the "O" is a wide-character? 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[10]. 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[19], 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.[26] proposed an end-to-end model with an attention mechanism. OrigamiNet[25] introduced a one-step model by learning to unfold, transforming existing CRNNs into multiline recognizers. Other approaches still exist, like proposed by WPOD-NET[22], where a modified YOLO detector is used for OCR purposes.

5.2 Data

An adequate amount of data is required to train a deep learning algorithm. There is no generally available de-facto benchmark dataset in ALPR. There were plate sets mainly 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[4]. 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 one-liner vs. multiline versions. Some countries use non-Latin alphabets (like China or Egypt), which makes the task even harder. 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 5.1: 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 or 1-I. 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 solution has the edge in the long run 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 handles text bounding boxes to be converted to output detector training data,

and it can produce multiple text blocks. Creating one 50x500 grayscale image takes 10.934 ms. The generator applies data augmentation techniques to enrich data diversity.

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 [0...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 epoch is set to contain 10,000 images. With this size, there is a maximum of +/-0.015 deviation in loss across multiple validations.

5.3 Model

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

5.3.1 Connectionist Temporal Classification

Connectionist Temporal Classification[10] is used to compute model loss. The cost function has a barely documented Keras implementation, which I wrapped into a layer to

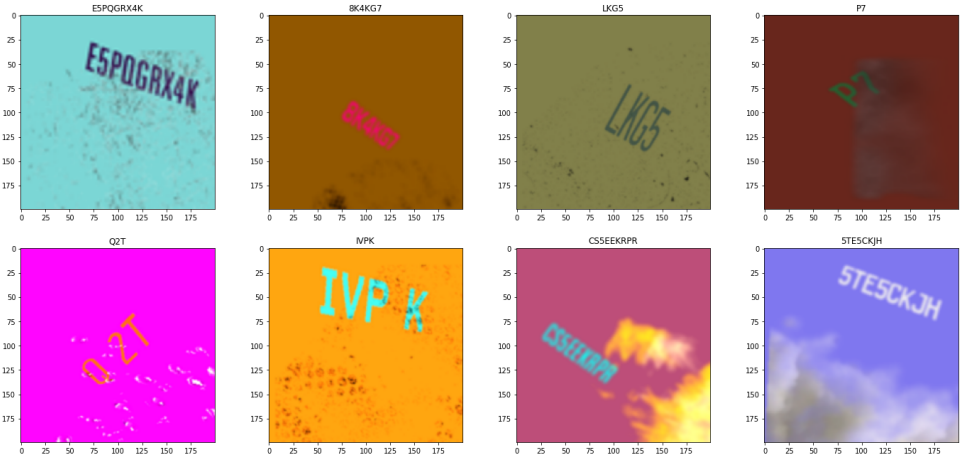

Figure 5.2: One-line generated samples with labels on top. From the validation set, hard overlays have been removed so that the completely illegible characters do not affect results.

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.

5.3.2 Building blocks

A custom convolution layer is defined with Convolution → Nonlinearity → Dropout → Batch Normalization. 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. In

this configuration, a block with input dimensions (height, width, channels) has an output of (*height/2, width/2, channels*2*). My implementation was inspired by the ResNet[16] architecture.

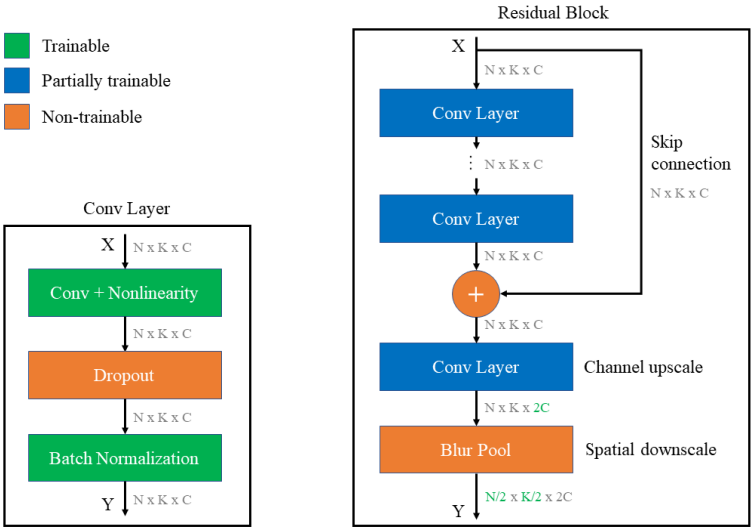

Figure 5.3: Network building blocks with the convolution layer (a) and the residual block (b).

In the recurrent part of the network, two Long Short-Term Memory[15] layers are stacked on top of each other responsible for sequential processing. 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 * downscale_factor, num_characters*), which is the network's output matrix.

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.

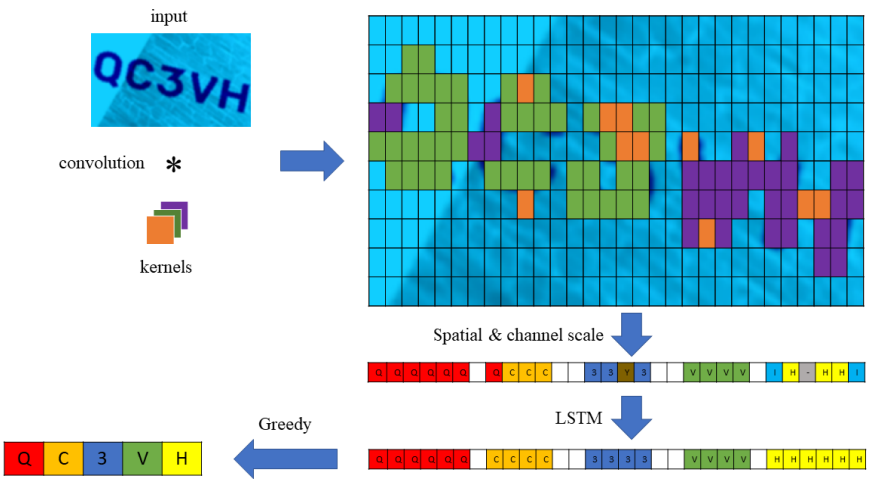

Figure 5.4: Simplified representation of the network's key processing steps.

5.3.3 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[14] 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.

5.4 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[17] is used with the default parameters (1e-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.

Table 5.1: 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.

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

5.4.1 Shift-invariance

Zhang et al.[27] 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. 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.

5.4.2 Hyperband optimization

Hyperparameter optimization has been realized with the Hyperband algorithm[18]. 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.

Table 5.2: Top3 configurations found by Hyperband. Only the top2 were trained until convergence, the 3rd has been stopped earlier. Columns: model, features, kernel, residual, batch norm, dropout, blocks, layer per block, activation, epochs, loss.

	Feat	Ker	Res	BN	Drop	B	L	Act	Epochs	Loss
base	32	3	✓	✓	0.1	2	1	*ReLU*	22	0.1134
1*st*	64	3	✓	✓	0.09	3	3	*ReLU*	39	**0.0753**
2*nd*	64	5	*X*	✓	0.2	2	2	*ReLU*	35	0.0862
3*rd*	64	3	*X*	✓	0.3	2	3	*ELU*	15	0.1554

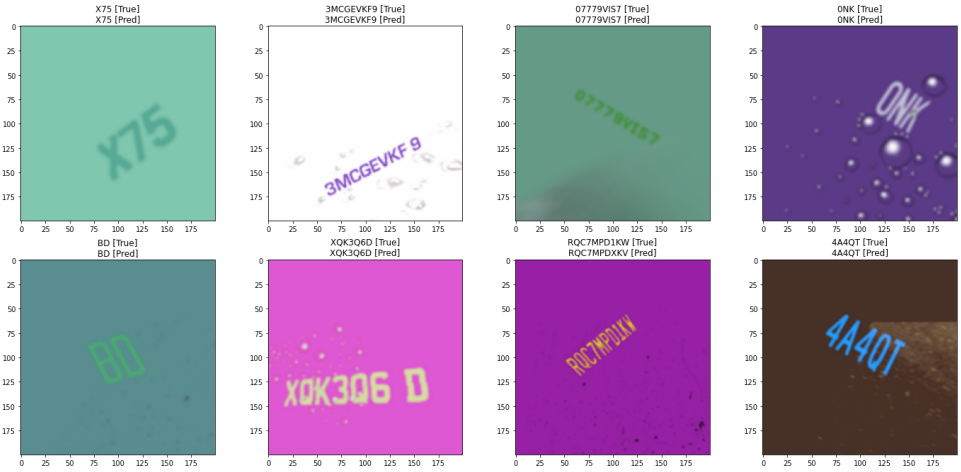

Figure 5.5: Inference results of the best performing network.

5.5 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[24] 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 2-dimensional feature map into a tall, 1-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[25] module that includes convolution layers and resizing with bilinear interpolation.

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.

Chapter 6

System Overview

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

6.1 ALPR pipeline

The applied pipeline aims to compress the traditional tasks into as few steps as possible. The structure is like the one used in WPOD-NET[22], with some differences.

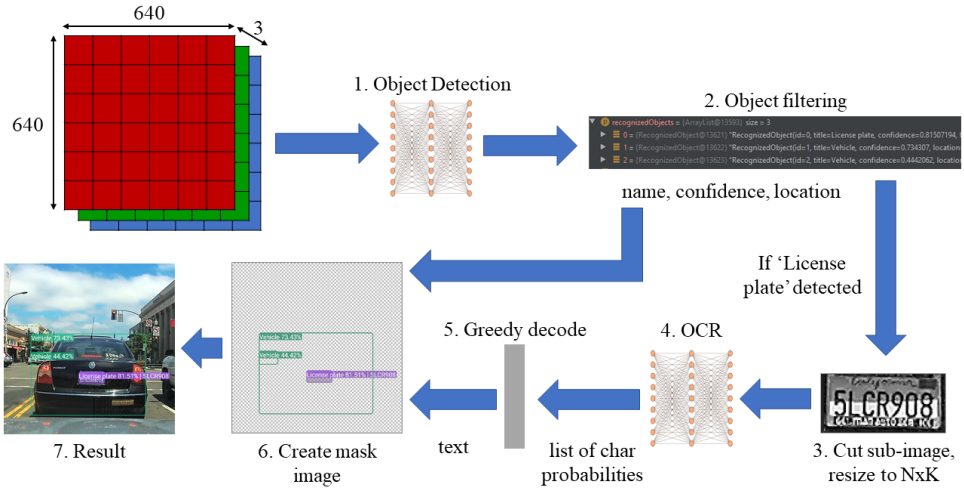

Figure 6.1: The applied ALPR pipeline.

First, an SSD[23] detector is used for both vehicle and plate localization (instead of a YOLO[20] vehicle-only detector). The rectification step is excluded from our pipeline, which is a drawback in plate image quality. WPOD-NET's rectification supports only one plate scale, so adding that would not be a general solution, in my 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.

The pipeline models are dynamically quantized TensorFlow Lite converted algorithms. On a Samsung Galaxy S10 smartphone, where an image contains only one license plate, the average runtime is roughly 160 ms, including image pre-processing (9 ms), detector

inference (75 ms on GPU), plate image part cutting, and resizing (3 ms), OCR inference (37 ms on CPU), Greedy decoding (<1 ms), and 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 sub-image.

Besides the best performing model, 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 (+9.5e-3 loss) to acquire a 1.5x speed-up. As a reference, Google's MLKit Vision Text Recognizer[13] ran 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 6.1: Running time in milliseconds for OCR models.

	Host	Input	Size [KB]	Loss	Inference [ms]
$batch128$	CPU	$50x500x3$	639	0.0848	37
$MLKit$	CPU	$50x500x3$	?	?	40
$batch128$	GPU	$50x500x3$	639	0.0848	50
$Hyper1st$	CPU	$50x500x3$	4447	0.0753	56
$MLKit$	CPU	$200x200x3$	?	?	72
$Unfolding$	CPU	$200x200x3$	3133	0.0991	221
$Unfolding$	GPU	$200x200x3$	3133	0.0991	372

6.2 Frontend

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

6.2.1 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 their 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 its UI state. The report item gets stored in a list wrapped in a LiveData object (which is observable from the Activity). When the user clicks on the send button, the related data is transmitted to the RepositoryService in the model layer. Inside this service, there is the 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 then persists it 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 not successful, the user sees the report as pending. Pending items can be deleted or re-sent at any time.

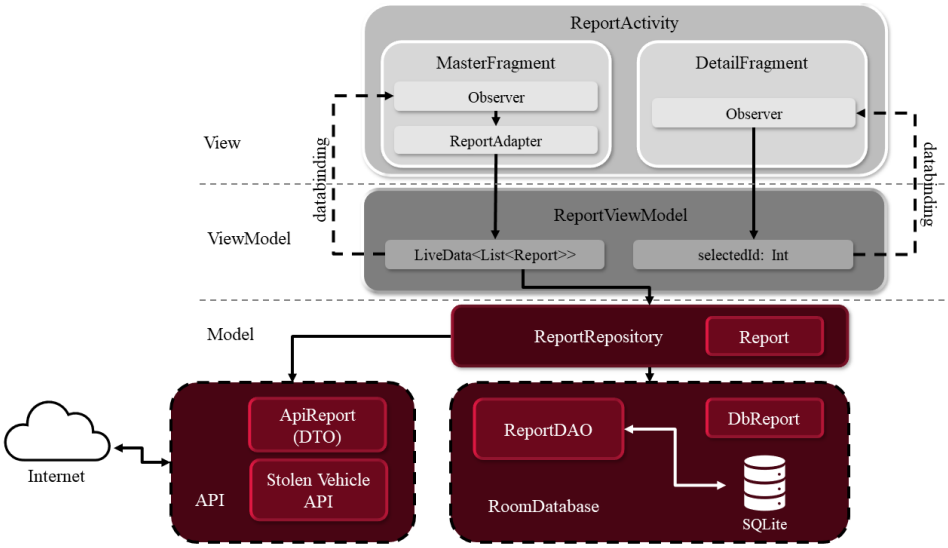

Figure 6.2: MVVM layers in the application.

6.2.2 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.

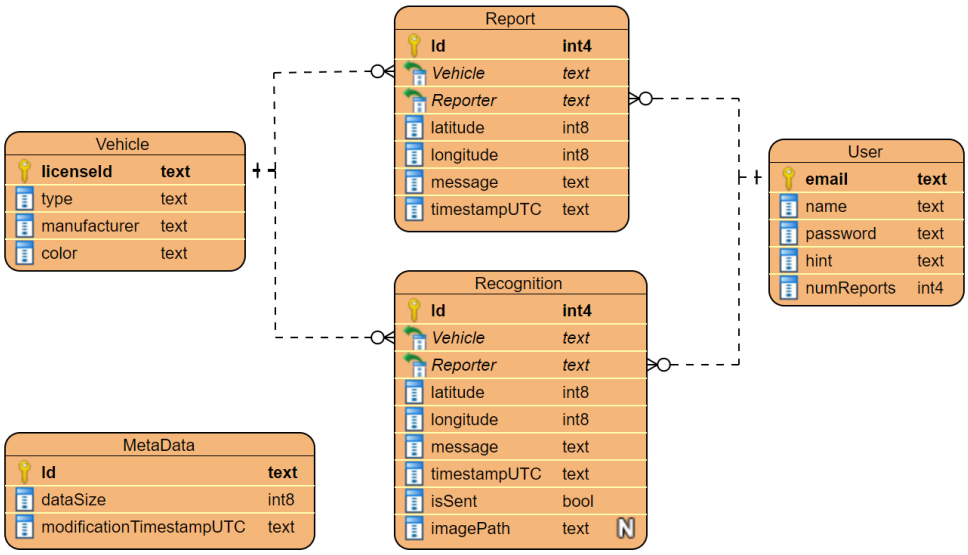

Figure 6.3: Application database schema.

6.3 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 every time querying the server.

6.3.1 Architecture

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

Instead of the UI layer, there is the communication layer through which the API 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.

6.3.2 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). To encode/decode JSON files, the server uses the Gson library.

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

The 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 a large amount of data is never stored on the server (images are not uploaded). To validate this, I examined one item from the largest JSON object type (Report), which is precisely 173 bytes. Multiplied by 1 million, it turns out that the server needs 173 MB memory, which is acceptable. It is a severe overestimation, though, as the stolen vehicles list obtained by web scraping typically has few thousand items. That is the maximum number of records that the in-memory database ever has (if every stolen vehicle has been detected at once). As history content is only stored persistently, extensive API usage does not saturate the memory either.

6.3.3 API

The API is divided into five parts: Vehicles, Reports, Report history, Self, and Users. These names are also the corresponding endpoint prefixes. All parts have similar actions

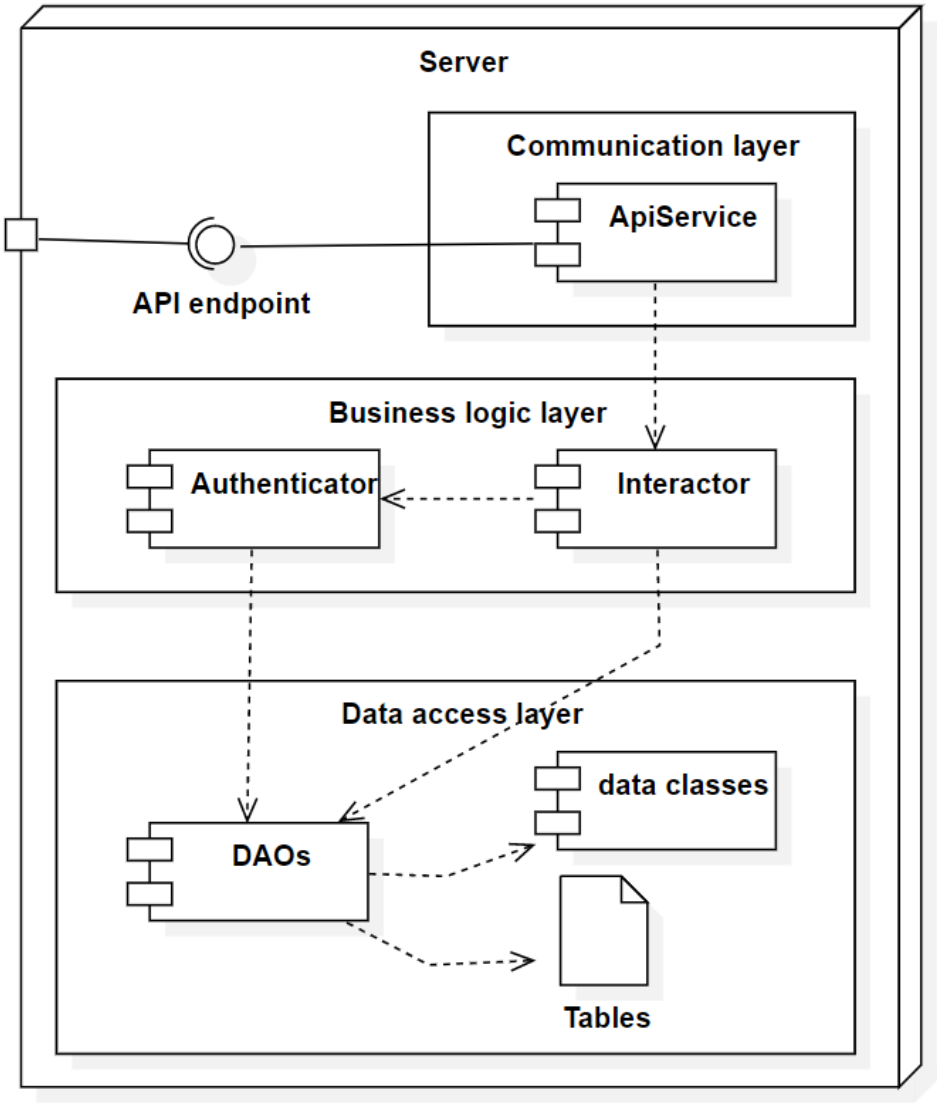

Figure 6.4: Structure diagram of the server.

and a unified calling convention. All actions are subject to specific permission, which is evaluated every time before serving. There is also a status page describing the API.

6.3.4 Permission

As the nature of stored data (location and timestamp of stolen vehicles) could potentially allow abuses, there is strict 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.

- API_GET lets an authorized account to download reports.
- API_SEND makes it possible to send recognitions to the server.
- SELF_MODIFY is needed to prevent blacklisted users from deleting themselves and re-register.

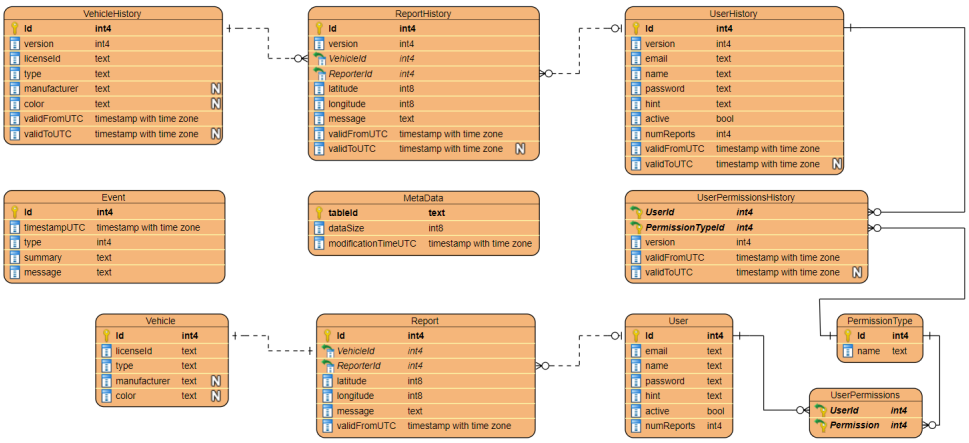

Figure 6.5: Server data schema.

- An ADMINISTRATOR user can modify the server and any user's permissions at any time. If someone's behavior is suspicious, an Admin can revoke permissions, delete a user, or deactivate and blacklist it. An ADMINISTRATOR can register a new user with specific permissions.

- The default/guest user in the client application is an account with only an API_REGISTER role. This way, it is possible for newcomers to register their new accounts. If someone tries to use the application without signing in is, in fact, utilizes this user. As its only API permission is registration, although someone can detect vehicles on-device, he/she cannot report them or see the actual reports. This API_REGISTER role prevents anyone outside the Android app from registering. The default user can create a new account with SELF_MODIFY, API_GET, and API_SEND permissions.

Chapter 7

Summary

In my thesis, I discussed the general task of Automatic License Plate Recognition, then examined best practices for both Object Detection and Optical Character Recognition. I trained deep learning based models to detect number plates and recognize texts on them, then a full ALPR pipeline has been created. Then, I integrated my solution into a stolen vehicle identification system. In the end, I presented the key concepts of my system's frontend and backend applications.

Further work would focus on the license plate detection step, which could be optimized by using better models in detecting small objects like Ultralytic's YOLOv5[12]. Using a detector that recognizes rotated rectangular coordinates could eliminate the need for plate rectification (which the current pipeline lacks); therefore, using a WPOD-NET-like network should also be considered.

Acknowledgements

The author would like to express his thanks to Dániel Pásztor for his valuable support as a scientific advisor. The work presented in this paper has been carried out in the frame of project no. 2018-1.1.2-KFI-2018-00062, which has been implemented with the support provided by the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.1.2 KFI. funding scheme. It was also supported by the BME Artificial Intelligence TKP2020 IE grant of NKFIH Hungary (BME IE-MI-SC TKP2020) and by the Ministry of Innovation and the National Research, Development, and Innovation Office within the framework of the Artificial Intelligence National Laboratory Programme.

Bibliography

[1]

[2] Kotlin programming language. Available at https://kotlinlang.org/ (2021/05/22).

[3] The ktor framework. Available at https://ktor.io/ (2021/05/22).

[4] Platesmania.com. Available at http://platesmania.com (2021/04/22).

[5] Protocol buffers. Available at https://www.tensorflow.org/lite/performance/post_training_quantization (2021/05/22), .

[6] .

[7] Python programming language. Available at https://www.python.org/about/ (2021/05/22).

[8] Tensorflow eager execution. Available at https://www.tensorflow.org/guide/eager (2021/05/22), .

[9] Tensorflow post-training quantization. Available at https://www.tensorflow.org/guide/eager (2021/05/22), .

[10] Faustino Gomez Alex Graves, Santiago Fernandez and Jurgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, 2001.

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

[12] J. Borovec NanoCode012 ChristopherSTAN L. Changyu Laughing-A. Hogan lorenzomammana tkianai yxNONG AlexWang1900 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 G. Jocher, A. Stoken and yzchen. Ultralytic's yolov5. Available at https://github.com/ultralytics/yolov5 (2021/04/23).

[13] Google. Mlkit text recognition. Available at https://developers.google.com/ml-kit/vision/text-recognition (2021/04/23).

[14] Stefan Fiel Harald Scheidl and Robert Sablatnig. Word beam search: A connectionist temporal classification decoding algorithm. Available at https://repositum.tuwien.at/retrieve/1835 (2021/04/23).

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

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

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

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

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

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

[21] Harald Scheidl. An intuitive explanation of connectionist temporal classification. `https://bit.ly/3aUSS3t`.

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

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

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

[25] 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.

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

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

Appendix