FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE

DEPARTMENT OF CYBERNETICS

LOCALIZATION OF POSITION MARKERS IN CAMERA IMAGE USING NEURON NETWORKS

BACHELOR'S THESIS

Egor Ulianov

Open Informatics - Artificial Intelligence and Computer Science

Supervisor: Ing. Martin Klíma, Ph.D.

May 2022

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Ulianov Egor** Personal ID number: **491923**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Localization of Position Markers in Camera Image Using Neuron Networks

Bachelor's thesis title in Czech:

Lokalizace orientačních značek v obrazu pomocí neuronových sítí

Guidelines:

Elaborate a state of the art analysis of object location methods for object localisation in a camera images using neuron networks (NN). Design and implement a solution for localization of ArUco markers in a live video stream of a mobile device. Choose a platform, either Android or iOS and demonstrate the functionality and asses the performance and feasibility of using this solution for quick detection of ArUco, eventually QR codes. Assess the system performance for zero, one and multiple markers in the view. Use a NN framework of your choice and justify your choice for the given platform.

Bibliography / sources:

[1] Orhan Gazi Yalçın, Applied Neural Networks with TensorFlow 2, APress, 2020
[2] Amit Kumar Sinha, Adarsha Ruwali, Abhilash Jha, Application of Deep Learning in Object Detection: Application of Deep Learning in Object Detection using Tensorflow, LAP LAMBERT Academic Publishing, 2019
[3] Felix M Philip, Bosco Paul Alapatt, Anupama Jims, Real Time Multi Object Detection and Tracking Using Deep Learning, Noor Publishing, 2020
[4] Xiaoyue Jiang, Abdenour Hadid, Yanwei Pang, Eric Granger, Xiaoyi Feng, Deep Learning in Object Detection and Recognition, Springer, 2019

Name and workplace of bachelor's thesis supervisor:

Ing. Martin Klíma, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **12.01.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Martin Klíma, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Prohlášení autora / Autor statement

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne .....
Podpis autora práce .....

I declare that the presented work was developed independently and that I have listed all sources of information used within at in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date .....

Signature .....

Abstrakt / Abstract

This work describes localization of position fiducial markers through modern approaches to the object detection problem using convolutional neural networks on mobile devices. The most current architectures are researched, and some samples are trained and integrated with a step-by-step description.

Keywords: *object detection, convolutional neural network, unity, yolo, pytorch, tensorflow, qr, aruco, fiducial markers*, *augmented reality*

Práce popisuje moderní přistup k detekci objektů typu „fiducial marker“ prostřednictvím konvolučních neuronových sítí na mobilních zařízeních. Nejmodernější typy architektur byly prozkoumané, a některé z nich byly použité pro vytvoření vlastních modelů, které pak byly integrované do aplikaci s podrobným popisem jednotlivých kroků.

Klíčová slova: *detekce objektů, konvoluční neuronová síť, unity, yolo, pytorch, tensorflow, qr, aruco, fiducial markery*, *rozšířená realita*

Acknowledgement

I would like to thank my parents, Sergey and Olga, for providing me all possible support during my studies and for their endless hope and trust, especially when they don't see their son for years.

Also, I'm grateful to my supervisor, Ing. Martin Klíma, Ph.D., whose support not only during my studies helped me to keep the faith that even long-forgotten childhood dreams can become a reality.

Finally, I would like to address my thanks to all the Misterine crew, a team of dreamers and gamechangers, as their energy keeps me in a stream. Special thanks – to Ing. Jan Dupač, Ph. D., for his priceless pieces of advice on computer vision topic.

Contents

1 Introduction

Object Detection is present in Computer Vision for more than 20 years. With methods of Object Detection, it is possible to search people, cars, cancer cells and other things in an image even in real time. And it can be useful in Augmented Reality applications, where a position of an augmentation is needed to be defined.

Generally, some sorts of objects - markers are detected with their orientation and position. They are called "Fiducial markers". QR codes and ArUco markers are widely used for this purpose, as it is possible to pass some information through these objects, for example an index of a shown augmentation scene, and they are designed to be easily detectable.

For this moment, deterministic methods are the most common way to detect these markers. Deterministic methods are introduced in OpenCV library, one of the most used libraries for computer vision.

But in the last years, neural networks methods of object detection are becoming more and more popular. So, is there an option that a solution using convolutional neural networks can also achieve results, which can be applicated in real usage? It is the question, which has been raised by the team of Misterine s.r.o., a company developing AR solutions for industry and education.

The aim of this work is to elaborate a state-of-the-art analysis of object detection methods on mobile devices using convolutional neural networks.

To achieve the aim, a list of tasks was defined:

- Find and process theoretical information on object detection problem in general and with usage of convolutional neural networks
- Find existing solutions of QR codes and / or ArUco markers detection problem
- Elaborate a learning process of some recent object detection neural network models
- Train some models considering the run on a mobile device and introduce them in a mobile app
- Elaborate a way of neural networks integration into Unity engine on the request of Misterine s.r.o.
- Make some recommendations on neural networks usage for the company considering not only QR codes and ArUco's as Fiducial markers in future

2 Theoretical Part

In this part of the work the main theoretical concepts will be described, including definitions, history and current methods.

2.1 What is Object Recognition, Localization and Detection?

Shimon Ullman in his "High-Level Vision: Object Recognition and Visual Cognition" book defines object recognition as an ability to identify an object from a visual input. One of the most important parts of it is a possibility of identifying invariancies of an object, e.g., on different backgrounds. [1]

Object localization, in comparison, is needed to define a position of an instance of a class with its bounding box. And object detection is understood as a localization of several instances of different classes. [2]

As the work considers usage of different types of markers and a possibility of multiple markers presence on an input, the object detection problem is elaborated in this work.

2.2 History of Object Detection

In this section main object detection methods for last 25 years are described, from methods based on AdaBoost to first regional-based convolutional neural networks.

2.2.1 Viola–Jones Object Detection

Viola-Jones object detection is an algorithm which was proposed by Paul Viola and Michael Jones in 2001, Cambridge, MA. Based on a newly introduced image representation "Integral Image", AdaBoost method and a new method for combining complex classifiers in a cascade, it made possible real-time object detection, primarily face detection, in 15 frames per second. [3]

2.2.2 HOG Detector

HOG detector is based on the histogram of oriented gradients (HOG). It was described by Navneet Dalal and Bill Triggs in 2005, but the main concepts had been introduced in 1986 by Robert K. McConnell of Wayland Research Inc. The main idea behind it is that the distribution of intensity gradients and edge directions can help to find local object appearance and shape. [4]

2.2.3 Deformable Part-based Model

This method was introduced by Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester and Deva Ramanan in 2010. The essential idea behind Deformable Part-based Model algorithm is to consider objects as a deformed version of a template, as objects on input can be in different poses and under a wide range of angles. The model is multi-scale, aiming at making possible an effective use of more latent information such as grammar (hierarchal) models and models which involve latent 3D poses. It uses a convexity of positive examples for latent-SVM, a formalism used by authors for a reformulation of MI-SVM in terms of latent variables. [5]

2.2.4 Regional-based Convolutional Neural Network (RCNN)

A new method using convolutional neural networks, Regional-based CNN, was introduced by a team containing Ross Girshick, Jeff Donahue, Trevor Darrell and Jitendra Malik on UC Berkeley in 2014, as a reaction on a slow progress in usage of minorly improved variants of successful methods and renewed interest in convolutional neural networks for an object classification problem. The team of UC Berkeley proposed to extract just 2000 "region proposals" which can be classified and boxed by processing a CNN output using SVM. [6]

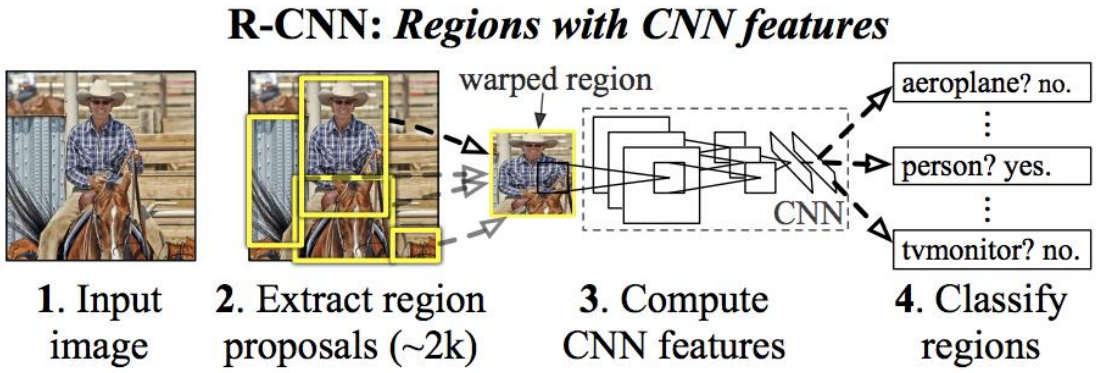


Figure 1. R-CNN principal scheme [6]

The algorithm for the selective search of regions is the following:

1. Initial generation of sub-segments, where many candidate regions are generated
2. Greedy algorithm application to recursively combine similar regions into larger ones
3. The generated regions are used for production of the final candidate region proposals

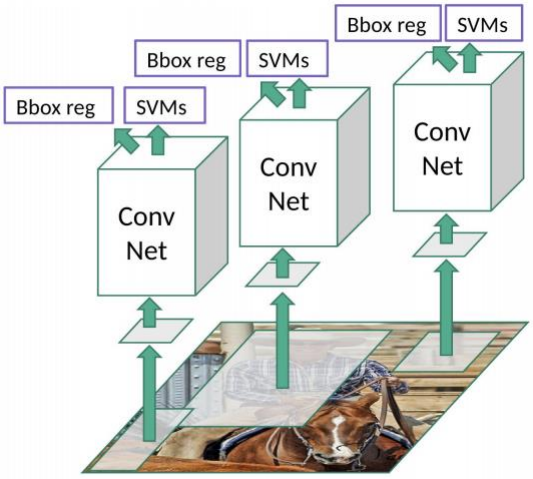


Figure 2. R-CNN regions analysis [6]

Two more improved variations had been developed (Fast RCNN and Faster RCNN), with much faster region proposal, but the main idea behind all the methods of object detection using convolutional networks is the same:

1. Different implementations of an effective regions search algorithm
2. Some sort of classification of those regions.

2.3 State-of-the-Art

As it was mentioned in the previous paragraph, generally there are two steps in object detection using convolutional neural networks. And according to those steps there can be two-stage detectors and one-stage detectors.

As two-stage detectors these examples can be mentioned:

1. RCNN (including Fast RCNN and Faster RCNN)
2. Mask RCNN
3. Feature Pyramid Networks
4. Granulated RCNN

The main one-stage detectors are [7]:

1. You Only Look Once (in different versions)
2. Single-Shot Object Detector
3. EfficientDet [8]
4. SquezeDet
5. Pelee
6. RetinaNet

Two-stage detectors are more accurate, but also slower than one-stage analogs. The question of speed is essential in the desired solution as the implementation should run on mobile devices.

2.3.1 You Only Look Once (YOLO)

A revolutionary method was introduced in 2016 by Joseph Redmon, Santosh Divvala, Ross Girshick and Ali Farhadi. It is a one-stage detector solving single regression problem, predicting bounding boxes and class probabilities.

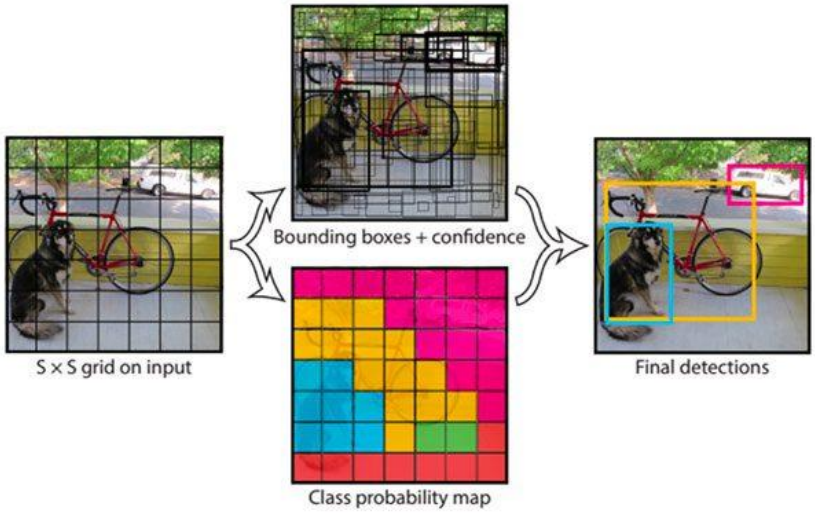


Figure 3. YOLO workflow [9]

Authors show that it is capable of running in 45 frames per second, the Fast version – even in 155 frames per second on Titan X GPU [9], what is, need to mention, still more than 7x more powerful than, for example, the latest Apple's mobile solution, Apple A15, in the meaning of single precision compute power [10] [11].

The idea is following [9]:

1. Firstly, a compressed image is divided into an S x S grid. If the centre of an object is inside the cell, this cell is responsible for detection of this object
2. Secondly, on each cell of this greed B bounding boxes with their confidence scores are predicted. A confidence score here is $Pr(Object) * IOU_{pred}^{truth}$ (IOU – Intersection Over Union). For each box 5 values are got: x and y coordinates, width and height, and confidence. Also it contains C conditional class probabilities, Pr(Class $_i$ | Object), for each grid cell
3. Finally, at test time the conditional class probabilities and the individual box confidence predictions are multiplied to get class-specific confidence scores for each box:
   $Pr(Class_i|Object) * Pr(Object) * IOU_{pred}^{truth} = Pr(Class_i) * IOU_{pred}^{truth}$
   It shows us the class probability and how well the predicted box fits the object.

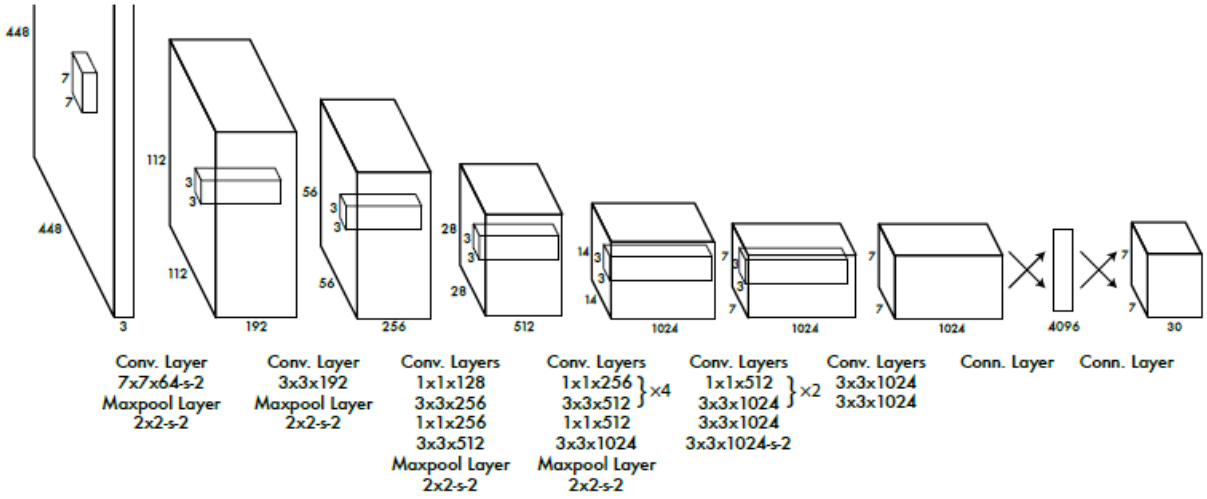


Figure 4. YOLO CNN architecture [9]

The network's model architecture contains 24 convolutional layers and 2 fully connected layers. 1 x 1 reduction layers are followed by 3 x 3 convolutional layers. A fast version of YOLO contains only 9 convolutional layers instead of 24.

As YOLO architecture has been considered as a successful one, newer versions and modifications were made:

- YOLOv2 (including Tiny modification, 2017) [10]
- YOLOv3 (including Tiny modification, 2018) [11]
- YOLOv4 (including Tiny modification, 2020) [12]
- YOLOv5 (including Nano, Small, Medium, Large etc. modifications, 2020) [13]

YOLOv1, v2, v3 and v4 are developed in Darknet framework, YOLOv5 – in Pytorch.

Tiny and Nano modifications are made specially for mobile deployment, with smaller model size and faster evaluation, but lower precision. For example, YOLOv5-nano can be 4 MB large and can run up to 80 frames per second on iPhone 13. In comparison, YOLOv5x is 170 MB large and runs 5 frames per second. [13]

As for the question of models' formats and export possibilities, YOLOv2, YOLOv3 [14] and YOLOv4 [15] are exported in Darknet format by default, which is not widely used in mobile frameworks. There are possibilities to export those models to PyTorch [16] and ONNX [17] formats. YOLOv5 is developed by Ultralytics in PyTorch [13] with options to export this model to different formats, including ONNX and different versions of TensorFlow.

2.3.2 Single-Shot MultiBox Detector

SSD was introduced in the end of year 2016 by Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu and Alexander C. Berg for real-time object detection. It is a one-stage family of detectors. The first version, comparing with already introduced YOLO, was faster and more accurate. [18]

Liu *et al.*

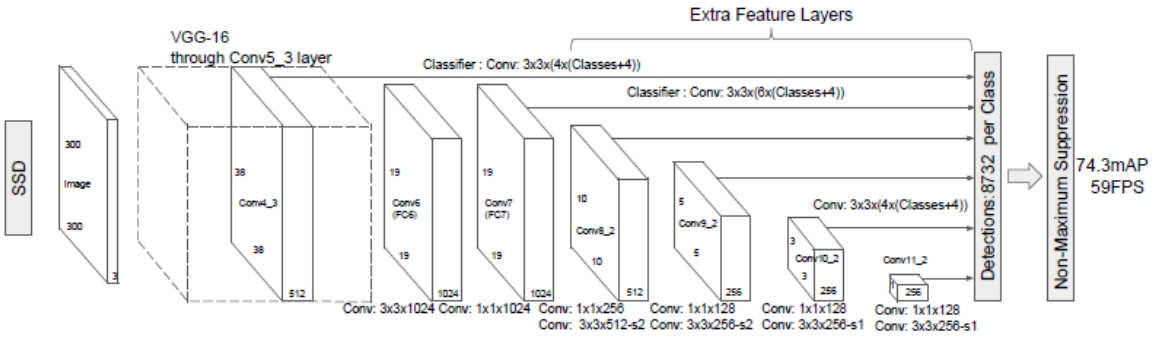


Figure 5. SSD architecture [18]

Like in YOLO, an input image is also divided into a grid of cells, and bounding boxes for cells are predicted. The difference is that class probabilities are made for default sets of bounding boxes connected to cells, not for cells themselves, and there is not the only one grid – multi-scale feature maps are made with cells of several sizes, using small convolutional filters applied on them. Finally, outputs from convolutional layers for each grid are united, and non-maximum suppression is applied. [19]

Some SSD modifications and versions are made. First, versions accepting images of different sizes on input (SSD300 for 300x300 images and SSD500 for 512x512). Also, MobileNet V2 and V3 in combination with SSDLite are needed to be mentioned.

SSD models are exported to Caffe format. There is no officially supported option to convert this kind of models to Tensorflow, ONNX or any other format. But in general, it is possible, as there are some solutions on Github. [20] [21]

2.3.3 EfficientDet

This method was developed by Brain Team of Google Research (Mingxing Tan, Ruoming Pang, Quoc V. Le) and published in the end of July, 2020.

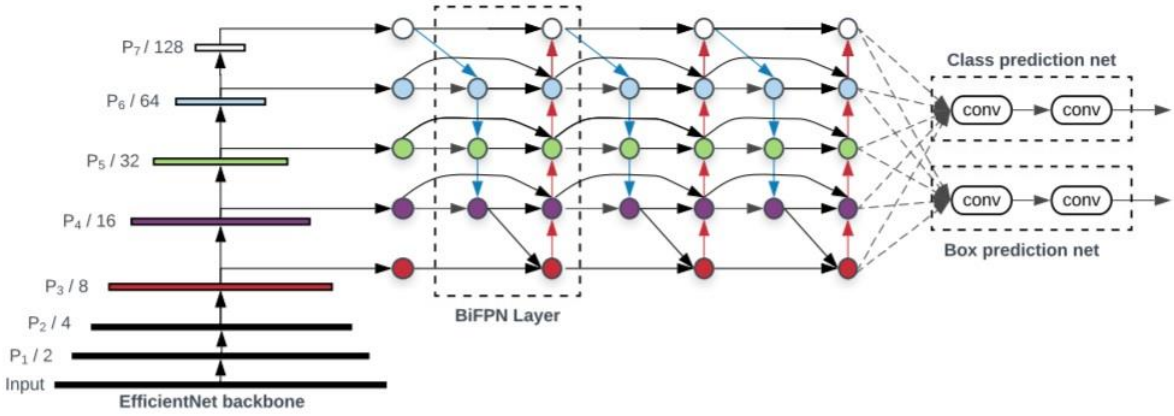


Figure 6. EfficientDet architecture [8]

Generally, it is a one-stage scalable detector, without a region proposal stage. But this time no grid is made. The most significant specialty of this type of detector is that it uses newly introduced Weighted Bi-directional Feature Pyramid Network (BiFPN) between backbone EfficientNet output and classes/boxes networks not only for multiscale feature fusion, but also to enable learnable weights to learn the importance of different input features. These features are used for boxes and classes prediction networks. [8]

TensorFlow and PyTorch implementations exist, with possibilities to export models to ONNX format. [22] [23]

2.3.4 SqueezeDet

SqueezeDet was introduced by a team from UC Berkeley and DeepScale (Bichen Wu, Alvin Wan, Forrest Iandola, Peter H. Jin, Kurt Keutzer) in 2019 for autonomous driving systems with high requirements accuracy, speed, model size and energy efficiency. [24]

The method is inspired by YOLO architecture with some adjustments, including [24]

- Grid with different width and height
- ConvDet layer usage for region proposals generation

ConvDet is a convolutional layer, working as a sliding window, trained to predict bounding boxes coordinates

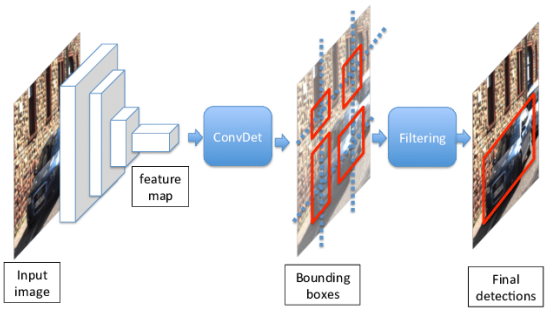


Figure 7. SqueezeDet principal scheme [24]

and probabilities of classes. Using ConvDet instead of 2 fully connected layers in YOLO, it makes possible to use 460x less parameters than YOLO. [24]

Originally this network is implemented in TensorFlow, and Keras implementation also exists. [25] [26]

2.4 Use Case Analysis

Generally, a solution will be potentially used for detection of Fiducial Markers in an application which shows augmented reality scenes.

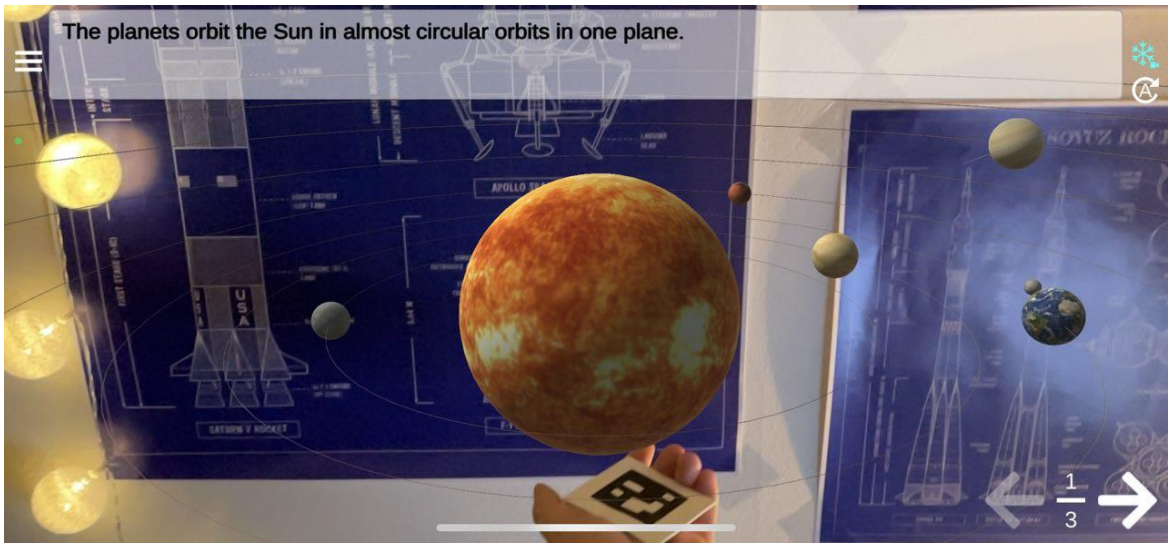


Figure 8. Misterine App

2.4.1 Requirements

The requirements for explored solutions should consider the following scenario from Misterine s.r.o.:

A neural network should run in real-time on a mobile device based on iOS/iPad OS or Android and detect Fiducial Markers - QR Codes and ArUco Markers, creating bounding boxes around them. Potentially the set of Fiducial Markers will be enlarged by adding other types of pictures (for example, pictures from Nreal library). Fiducial Markers are used for Augmentation scenes positioning – as in Misterine App, an application for AR manuals. It would be also useful to provide enough information (coordinates of 4 points in case of QR code) for homography estimation, or another variant of further usage is possible – deterministic detector application on a bounding box.

The solution should be highly likely supported for a long term (a development team of a large and authoritative institution or company

should be behind it) and should potentially accept new kinds of models as object detection networks is a field of study with a high probability of innovations. Ideally the solution should be appliable in Unity engine, because Misterine App is originally developed in this game engine.

Also, the model architecture and integration way should be open source or already bought by Misterine.

So, the following list of functional and non-functional requirements was made.

Functional:

FC1. The detector should make bounding boxes for QR Codes, ArUco markers and potentially some parts of QR Codes
FC2. The detector should classify objects inside bounding boxes
FC3. The model should be easily retrainable to accept new types of Fiducial Markers
FC4. Frames per second metric should be shown and recorded

Non-functional:

NFC1. The detection should be in real-time, using video from back camera
NFC2. The solution should run on Android 8.0 and higher and (potentially) iOS/iPad OS 12 and higher
NFC3. Models should be able to deploy in Unity (at least theoretically, better – with a real example)
NFC4. The solution, in general, should use open-source or already bought technologies

2.5 Functional Analysis

For further development all needed features should be described, with their importance and properties.

2.5.1 Features and their Importance

The described features correspond to mentioned requirements:

1. Bounding boxes and classification generation
   Bounding boxes with class labels are needed for getting region-candidates, on which a homographical algorithm can be applied. Definition of bounding boxes with their labels makes possible a detection of several markers of several types from only one image.
2. Easily retrainable model architecture
   An easy for retrain model architecture (with an easy and clear

training process, widely used input files format etc.) should be used to make further development with new types of markers easy to hold.

3. Statistics generation
   Frames per second statistic should be collected because the resolution if the neural network will be integrated or not should be made, and if yes, the most effective one should be chosen to use in Misterine App in future.
4. Real-time detection
   The detection process should run in real time as it will be potentially used in an AR application. Latency should not be higher than 200 ms, what corresponds to 5 FPS, on the most of target devices (iPhones, iPads, and high-middle Android devices).
5. Unity-deploy availability
   The models used in developed solution should be easy-to-deploy in Unity, as Misterine App and other applications developed by Misterine use Unity as an engine.

3 Existing Solutions for Fiducial Markers Detection

As QR Codes and ArUco Markers are quite popular, some attempts to detect them using neural networks were made.

3.1 Examples

Three examples were found, for ArUco and for QR codes.

3.1.1 ArUco Marker Detection under Occlusion Using Convolutional Neural Network

The solution was introduced in 2020 by a team from Sun Yat-Sen University (Guangzhou, China). In this project ArUco markers are detected for the autonomous drones' position definition. Chinese scientists used several tiny versions of YOLO models on the Nvidia GeForce GT 1030 GPU. It achieved 20% lose rate from 8 meters with latency 0.1s, from 5 meters – with 0.02s latency. Th size of markers was 0.2m x 0.2m. The solution can detect several ArUco markers from one frame. [27]

3.1.2 WeChat QR Code Detector

A Tencent's solution showed in OpenCV contribution version in the end of 2021 (version 4.5.2) uses SSD architecture for QR codes detection. It can detect only one QR code from an input image, and the run was showed on a PC. It cannot detect ArUco codes. [28]

3.1.3 YOLOv3 QR Codes Detector

The solution from 2019 was developed by Portmann, Gabriel Bello. The project uses a YOLOv3-tiny model trained on 1000 of 480px x 480px images. Here no statistic is provided neither on a PC, nor on a mobile. [29]

3.2 Summary

These solutions show that detection of fiducial markers using convolutional neural networks is possible and usable. But some issues were defined:

1. None of these solutions was demonstrated to run on a mobile phone, only on stationary computers or on hardware components comparable with them were demonstrated
2. No solution detecting ArUco markers and QR codes at once was found
3. Found solutions don't use the most recent and modern models provided by the global community

4 Design

This part describes the choice of methods which will be used for the implementation.

4.1 Choice of a Way to Run Model in Unity

There are some ways to run convolutional neural networks in applications based on Unity Engine.

The first and the most common way is to use officially supported open-source Unity Machine Learning Agents (ML-Agents) with the Unity Inference Engine (codenamed Barracuda). A model in this case preferably should be exported to ONNX format of the 9th opset version, as this version is more covered by the package. It can be a problem, because newer versions exist, and development continues. [30] Barracuda can run both on GPU and CPU of a mobile device, and also supports both iOS and Android. [31] The package is regularly updated, the last version (as for May, 2022) was introduced in January, 2022. It is provided under the Unity Companion License, which makes it possible to be used in a commercial product. [32]

The second way is to use PyTorch or TensorFlow API.

For PyTorch it is libtorch C++, and it enables a full functionality of PyTorch. Also, installation with CUDA support is suggested, which makes GPU running possible. But it might be hard to deploy, as drivers, other libraries, etc. should match to some degree and should be controlled with the toolkit's updates. Also, no pre-built versions for Android and/or iOS are provided. [33] PyTorch is an open-source solution under the Modified BSD license, so it is also can be used in this work and in further development for Misterine. [34]

For TensorFlow and its Lite version a C++ API is officially supported. It is also capable of running on GPU, and the team is a part of Google. TensorFlow itself is provided under Apache License 2.0, so it is free and open-source library. [35]

What is good, a pre-built version is provided by the community, which is supported by Android and iOS, and licensed under MIT License. With this project, an example of SSD-MobileNet is provided.

Another option for TensorFlow could be TensorFlowSharp, a library port for .NET developed by community. The problem is that it is not updated for more than a year and does not support many of new TensorFlow features. [36]

Also, it is possible to use OpenCV for this purpose. This open-source solution can run models in Darknet and Caffe formats and has got a Unity version already used by Misterine. The problem is that OpenCV in Unity version cannot run on GPU, which is important for neural networks. And potentially

there can be one more problem, as Misterine does not use the latest version of this library.

	Supported models	Unity support	Misterine usage	License	GPU support	Last update
Barracuda	ONNX opset 9 version	Native	No	Unity Companion License	Yes	2022-01-24
libtorch	PyTorch	Needed to be built for each platform, no pre-built versions	No	Modified BSD license	Yes	2022-03-10
TensorFlow C++ API	TensorFlow, TensorFlow Lite	Needed to be built for each platform, open-source pre-built versions exist	No	Apache License 2.0, MIT	Yes	2022-01-31
TensorFlowSharp	TensorFlow, TensorFlow Lite	Native	No	Apache License 2.0	Yes	2021-05-26
OpenCVForUnity	Darknet YOLO, Caffe, old TensorFlow	Native	Yes	Bought by Misterine	No	2021-12-30 (a version from 2020-05-04 is used)

Figure 9. Integration methods comparison table

For further development Barracuda and TensorFlow C++ API are chosen because of Unity compatibility and relatively new versions updates, and OpenCVForUnity is chosen, as it is already used by Misterine.

4.2 Choice of Models for Training and Deploying

There is a plenty of object detection convolutional neural network models and their modifications, with different purposes and characteristics. The choice of models should correspond to the following set of parameters:

- Size of a model (number of parameters)
- FLOPs as a parameter for latency
- Accuracy

- Ability to export to formats supported by Unity ML-Agents (Barracuda)
- Availability of easy training mechanisms

Small models and versions for mobile solutions are chosen for comparison, with statistics on COCO dataset (for SqueezeDet KITTI dataset statistics are provided, as no information about COCO testing was found).

	Model size, millions of parameters	FLOPs	Integration method support	Training mechanism	Accuracy, in %
EfficientDet-D0 [8]	3.9M	2.54B	Through TensorFlow	Google Colab is provided	34.6
SSDLite + MobileNet [37]	5.8M	2.32B	Exported to older version of TensorFlow	Google Colab is provided	67.0
SqueezeDet [24]	7.9M	9.7B	Through TensorFlow	Script on Github is provided	76.7
YOLOv4 Tiny [38]	6.1M	6.93B	Through OpenCVForUnity	Google Colab is provided	40.2
YOLOv5n [13]	1.9M	4.5B	Through Barracuda, TensorFlow	Provided by Ultralytics	45.7

Figure 10. Models' architecture comparison table

Based on integration availability, model sizes, easiness of training and FLOPS number, EfficientDet-D0 and YOLOv5n were chosen for further training and integration attempts.

4.3 System Architecture

The system, in general, should contain three main parts:

1. Dataset generator
2. Training mechanism
3. Applications for smartphones and tablets with trained models

Firstly, a set of data is generated. It should contain images and an information on bounding boxes with their labels for each image. After that, these data are passed to a training module, where a new neural network model is created. The next step is deploying this model to a smartphone with an application capable of running neural networks on its camera input.

The application shows neural network outputs and collects statistics on the solution's FPS rate.

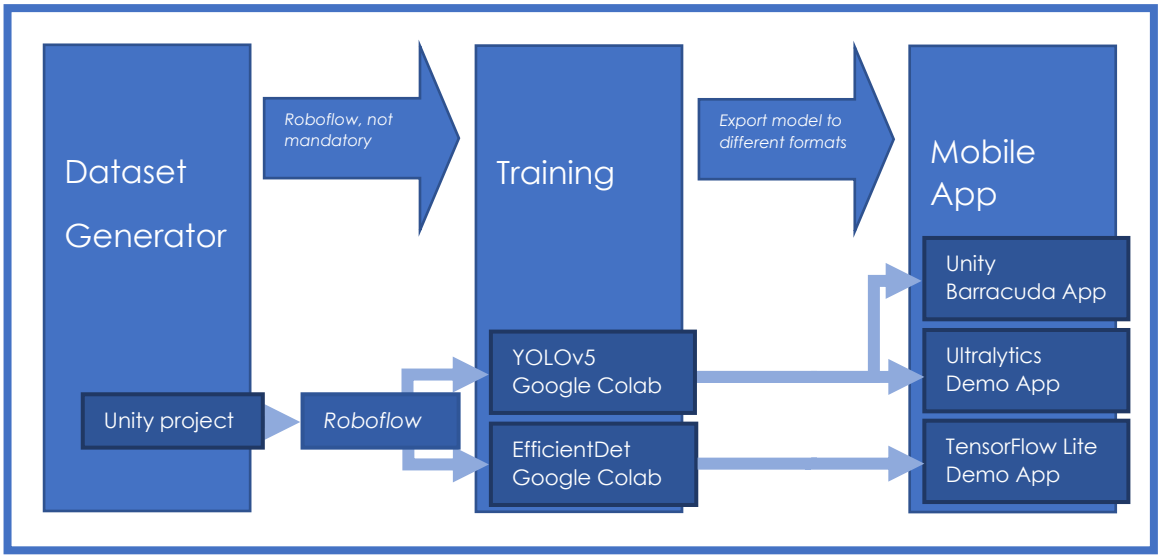


Figure 11. A principal scheme of the solution's architecture

4.4 Details of System Components

Here the components listed above are described, with the technologies used for the implementation and principal connections.

4.4.1 Dataset Generator

This part should automatically generate a set of pictures with bounding boxes and their labels. It should have got a set of backgrounds, objects instances, ability to change objects' brightness and position in space. Also, an automatic convertor to different data formats and manual labels-mapper can be used.

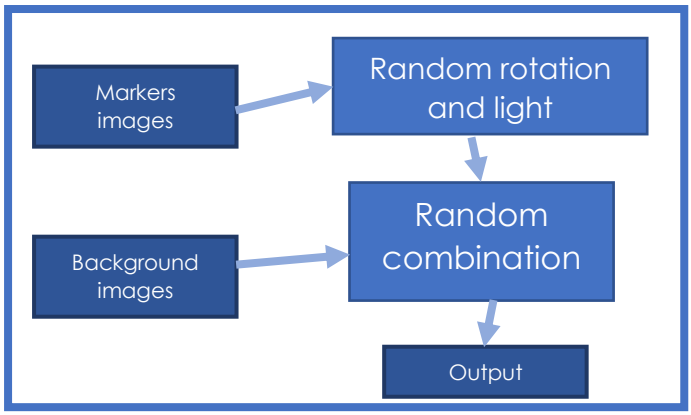


Figure 12. Dataset generator

For the further development, it was decided that the dataset generator will be created in Unity, as it is the simplest way for the author to operate with objects transformation. It will have got a set of backgrounds collected from Google, and also a set of objects' pictures (QR and ArUco). The application should generate a given number of pictures with labeled bounding boxes in one of common formats. In this solution simple YOLO Darknet TXT labeling format is chosen because it is easy to convert it to other types of formats.

After the dataset is generated, QR codes angles can be added manually by one of open-source dataset editors. For this work Roboflow is used, as it is free for personal and scholar use. [39]

4.4.2 Training Mechanism

For different types of models individual training mechanisms are made. Public Google Colab sheets are often provided by research teams. The point of using Google Colab is that it is possible to use efficient hardware (for example, Tesla K80 GPU) freely, what is important for training, as it is a demanding process.

For YOLOv5 an open-source solution with a Colab sheet under GPL-3.0 License [40] is made, with a possibility to import data directly from YOLO Darknet TXT format with some folders structure adjustments, described in [41]. This solution also makes export to different formats possible (ONNX, TensorFlow Lite, PyTorch, etc.)

For EfficientDet there is a free-to-use [42] solution from Roboflow team, which is also a Google Colab sheet. Although, Roboflow itself is free-to use only for personal projects and class assignments [39], so it can be used in this work, but for Misterine purposes datasets will be uploaded manually into Google Colab.

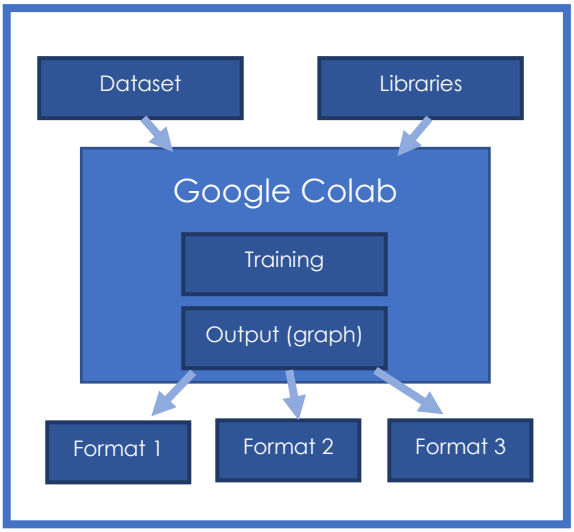


Figure 13. Training mechanism

In each case, a training based on a pre-trained model is possible, so a training time should be smaller.

In general, for each Colab all required libraries are loaded and installed, a dataset is loaded, and after that the training itself starts. After it is finished, the result graph can be downloaded, with a possibility to export a model to different formats.

4.4.3 Testing Applications

Ideally, a Unity solution should be provided for each type of model. But, as time is limited, and the research is made for elaborating the state of the art,

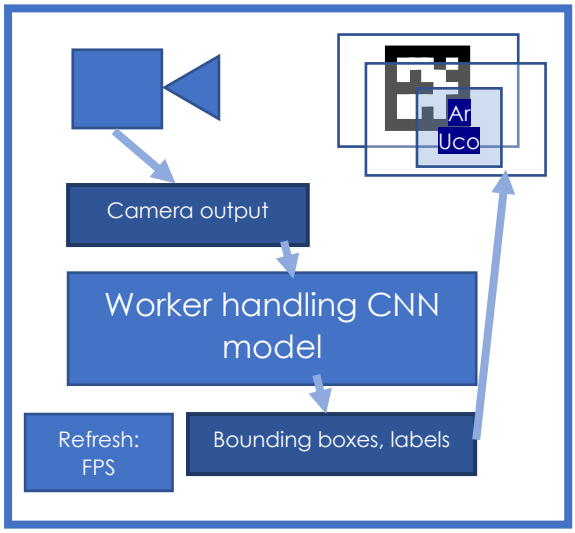


Figure 14. Detector

native TensorFlow solutions are possible, because it is shown that TensorFlow C++ API can be used in Unity.

A custom Unity application for YOLOv5 integration will be created, where a model will be applied to a part of the input texture from a camera. It should be able to run on Android and detect objects within a defined frame. Different object types will be defined by colors (green for ArUco, blue for QR codes, red for their angles). Also, it should provide information about current FPS rate on a screen and to Unity debug to collect the statistics. The statistics will be got from Android Logcat package after the device is connected to a PC.

Also, Ultralytics team provides its own application for models testing, using PyTorch version of your own model. This application exists for Android and for iOS as well.

For the EfficientDet detector an application from TensorFlow public examples will be used. It is a classical Android application written in Java. It shows bounding boxes, their labels and current inference time. But one adjustment will be made – it should be able to print FPS statistics to debug output, accessible through Android Logcat.

But there is a potential problem in the case of TensorFlow and PyTorch, as Unity runs on Mono – the question of scheduling within a Mono solution calling an external neural network library is not documented or explored enough, so in the case of further development using this combination the usage of hardware should be learned in practice.

5 Implementation

5.1 Dataset Generator

For the development of the dataset generator Unity 2020.3.33f1 Personal has been used. The generator is a Unity desktop application made for PC, containing four folders:

- Resources
- Scenes
- Scripts

In the Resources folder there are special folders for ArUco markers, QR codes and backgrounds. All the pictures are imported as Sprites (2D and UI), as Unity requires it to use them as source images for Game Objects.

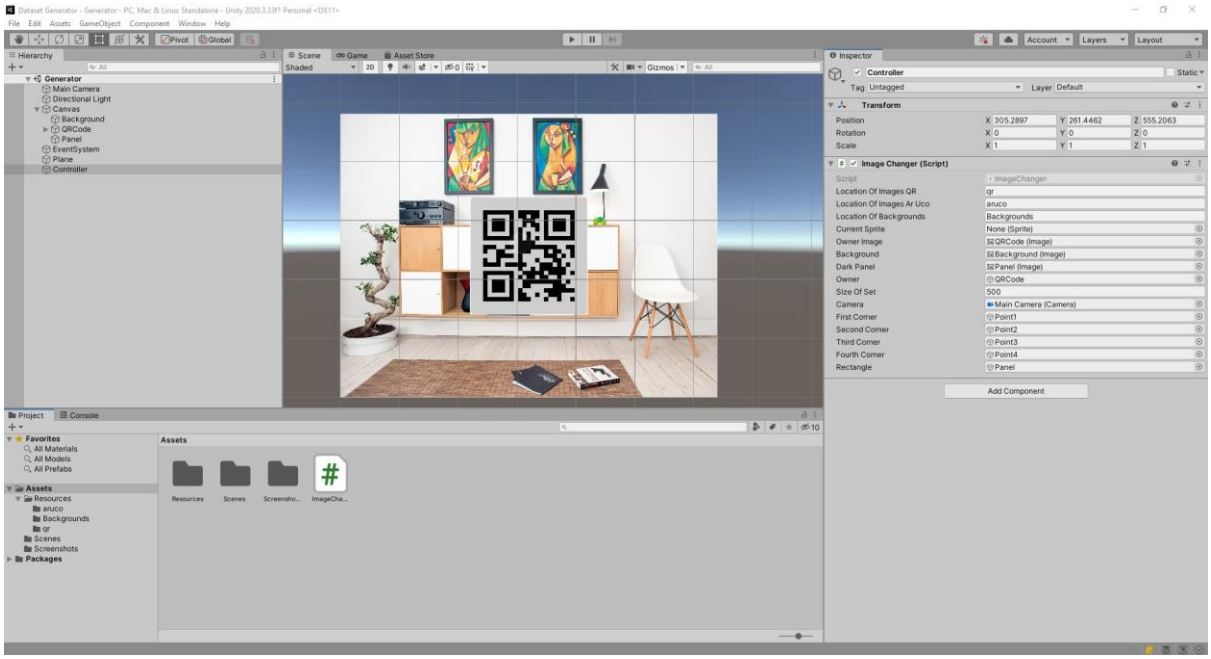


Figure 15.Dataset generator program screenshot

The Scenes folder contains only the main Generator scene. It is the main and only scene of the project, where a canvas with a background, an object and an obscuring panel are located. On the same level with the canvas the Controller is present. It handles a special script which controls changing of background and objects images, rotation of the object and the color of the obscuring panel. It also generates files in YOLO Darknet TXT format.

The C# script *ImageChanger.cs* itself is located in a special folder Scripts. It contains the public class `ImageChanger` derived from Unity's `MonoBehaviour` base class. As it is a MonoBehaviour, it provides two public methods: `Start()` and `Update()`. On the Start images for objects and backgrounds are loaded from the Resources and the initial state is set up.

On each Update a method for image update is called, where the background, the object and its rotation and obscuration are chosen randomly. After that, labels and boxes are generated and written into a defined folder. Images are saved to another folder. Number of updates is defined within the Unity.

In order to make the dataset closer to real sub-optimal scenarios, not only clear QR codes and ArUco markers samples can be used, but also unfocused or noised ones, made manually through one of graphical editors (GIMP or Paint.NET for example) out of clear samples or found on the net.

As the result, images with bounding boxes files are created.

Next, files are loaded to Roboflow, where other bounding boxes can be added (for example, for QR codes angles).

5.2 Training Mechanism

For the chosen model architectures, YOLOv5 and EfficientDet, training mechanism is described in this part.

5.2.1 YOLOv5

For YOLOv5 the training mechanism is provided by Ultralytics team, and it is quite simple. The steps are following:

1. Upload a dataset in a specified format
2. Choose a base model (YOLOv5n in this case)
3. Start the training
4. After the training is done, download the model in a requested format (ONNX, TensorFlow, PyTorch, etc.)

5.2.2 EfficientDet

For the EfficientDet-D0 a Google Colab sheet is provided. [43] Firstly, all data and packages are loaded. Next, the file system structure is modified for different folders.

The next important thing is setting up of training preferences. Here it can be done manually, for example the learning rate, the batch size and the image size can be modified. After this step the training can be started with a given number of epochs.

After the training is completed, the model can be exported to TensorFlow Lite format.

5.3 Application

Three applications are implemented or adapted – a custom Unity Barracuda application, TensorFlowLite demo and Ultralytics demo.

5.3.1 Unity

The Unity application is made in version 2020.3.33f1 and contains only one scene – Detector. It has got a set of GameObjects, the most important of them are:

- Main Camera
  Here the Unity's Camera is located. Also, the Phone Camera Script is attached here, which handles input from the device camera and output from YOLOv5n Detector, which is located in another GameObject
- Canvas
  - Background
    Here the Texture from the device camera is saved and shown
  - FPS
    This component shows current FPS
- Detector
  The Detector itself, which works with the loaded neural network model, is located here

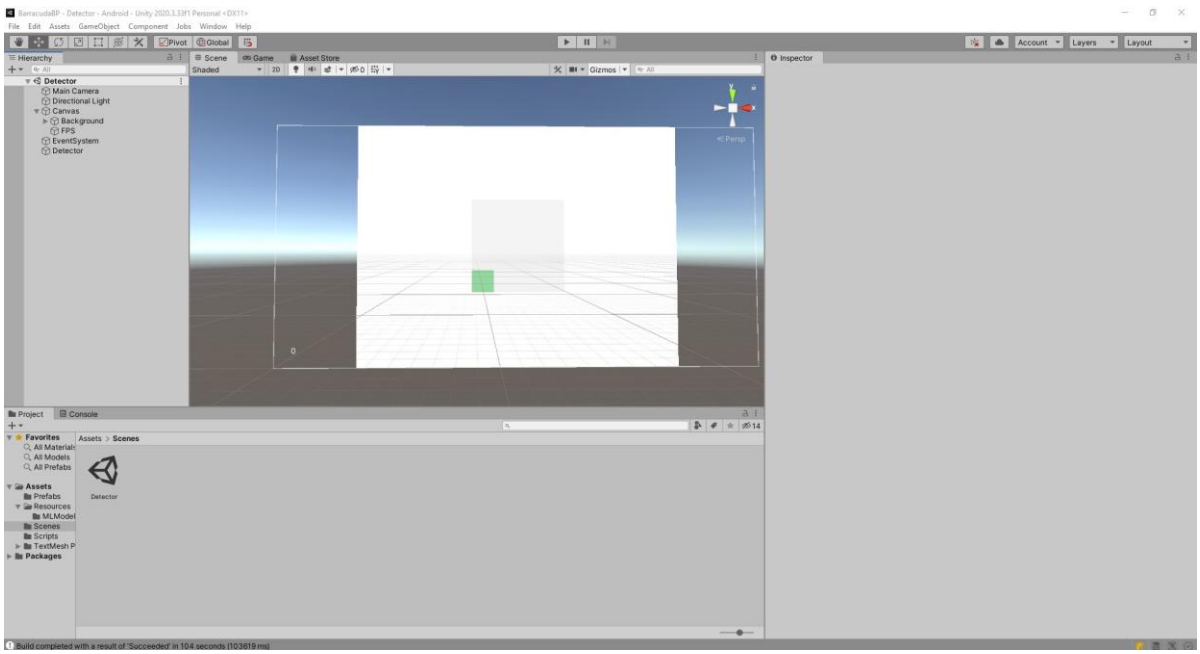


Figure 16. Unity detector screenshot

The C# project associated with the Unity project contains four files:

1. Detector.cs

   `interface Detector` with two provided methods,

`void Start()`,
a standard Unity method,
`IEnumerator Detect (Color32[] picture, int requestedWidth, SystemAction<IList<BoundingBox>> callback)`,
which detects the objects from aa picture represented as an array of `Color32`.

`class BoundingBoxDimensions`
with size and coordinates properties

`class BoundingBox`
with the following properties:
```
BoundingBoxDimensions Dimensions,
string Label,
float Confidence,
Rect Rect
```

The reason of this file existence is that in future other kind of detector can be explored.

2. GraphicsWorker.cs

   Provides an only one static method
   `IWorker GetWorker (Model model)`,
   which returns an instance of IWorker depending on a current platform and GPU availability

3. PhoneCamera.cs

   Contains
   `class PhoneCamera: MonoBehaviour`
   which gets all needed inputs from Unity, including box colors, background, detector, a prefab for box, a text field for FPS, and provides the following methods:
   `void Start()`,
   in which the texture from a camera is got and ratio for detecting frame is set,
   `void Update()`,
   where the input from camera is provided to the Detector, and the detection starts on each frame. Also, bounding boxes are redrawn here, and FPS is counted

4. Yolov5Detector.cs

Contains
`class Yolov5Detector: MonoBehaviour, Detector,`
in which the detector's parameters are handled, such as image size, number of classes, number of the model's output rows, minimal confidence rate, limit of detectable objects, neural network model file and labels file.

It provides the following methods:
`void Start ()`,
in which labels, a model and a worker are loaded,
`IEnumerator Detect (Color32[] picture, int requestedWidth, SystemAction<IList<BoundingBox>> callback)`,
as described in the base Detect class

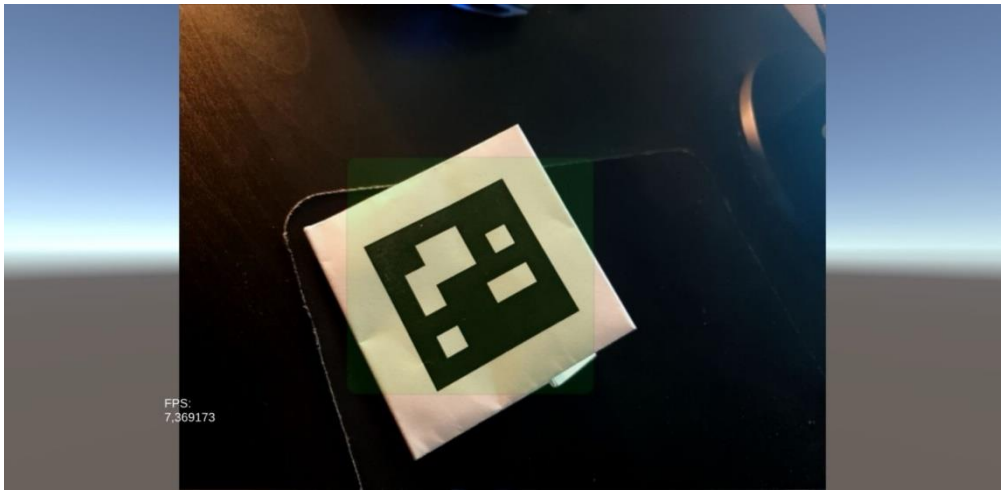


Figure 18. Screenshot from Unity solution

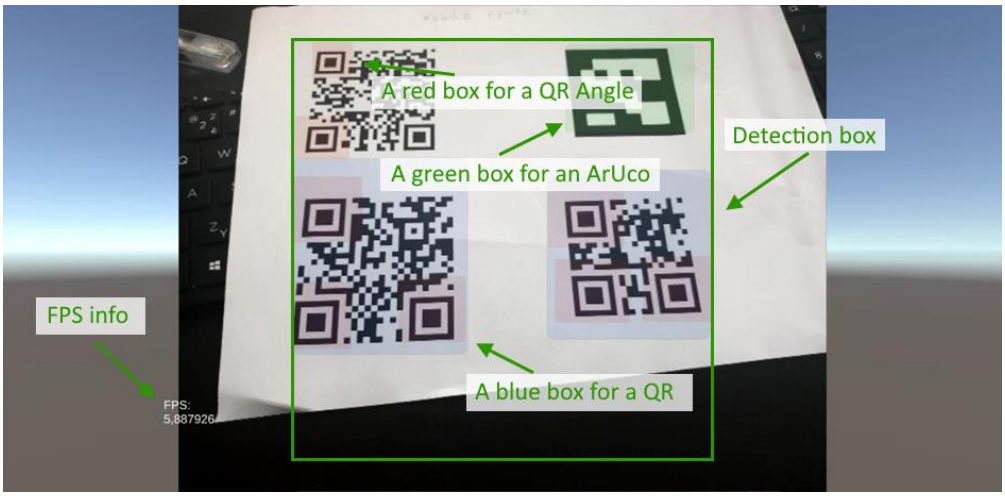


Figure 17. Unity detector run on Android

5.3.2 Other Applications

As for Ultralytics and TensorFlow Demo, no significant implementation details can be provided, as the only thig made manually is the change of the model.

Ultralytics App can run both on Android and iOS, with a possibility to log in and get a model from a user's account from which it was trained. It shows bounding boxes with labels and confidence and provides FPS on the main screen.

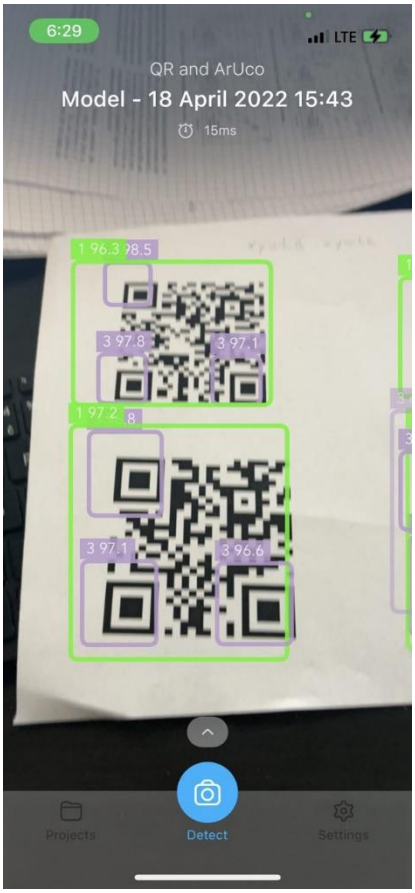


Figure 19. Ultralytics demo application

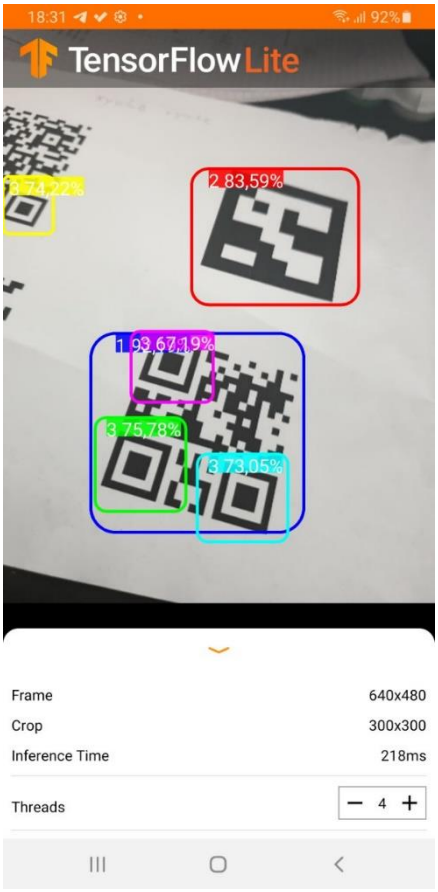


Figure 20. TensorFlow Lite Demo

TensorFlow Demo is provided via the official GitHub and can be built from Android Studio after exchanging the model's file and labels file. In this demo a number of threads can be chosen, and for the EfficientDet only CPU run is possible. It also provides information on the refresh rate as a latency in ms.

6 Evaluation of Tests

For testing purposes, Samsung Galaxy Note 9 128 GB was chosen, as it is already disposed, even if it is not the best solution on the market now. It runs under Android 10 with CPU Samsung Exynos 9810 (Octa-core, 4 x 2.9 GHz, 4 x 1.9 GHz), 6 GB of RAM and Mali G72 MP1 GPU.

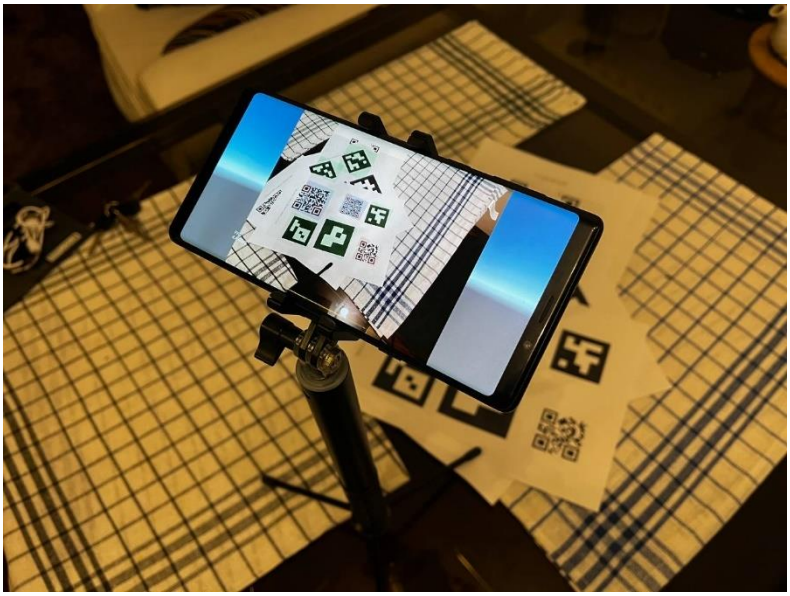


Figure 21. Testing mechanism

The tests are considering run on CPU and on GPU using Vulcan API in Unity application, Ultralytics application and TensorFlowLite Demo application only on CPU. The applications are tested on following criterions:

- FPS on CPU run
- FPS on GPU run
- Number of not-caught objects
- Number of misclassified objects
- Number of false-positive cases

Tests are done on a set of different QR codes and ArUco markers, single or combined together, QR codes with QR codes, ArUco markers with ArUco markers, QR codes with ArUco markers on different pages. Totally the set contains:

- 8 QR codes
- 24 QR codes angles
- 9 ArUco markers

For each run (CPU or GPU) 5 papers with examples are shown. FPS is counted as an arithmetic mean of control points on papers.

For each detector similar min. confidence was used (60%).

The data provided in the table on the next page.

	EfficientDet-D0 under TensorFlow Lite Demo	YOLOv5 under Unity Barracuda	YOLOv5 under Ultralytics Demo (PyTorch)
CPU FPS	6.99 FPS	5.92 FPS	47,61 FPS
GPU FPS	-	5.35 FPS	61,59 FPS
Not-caught objects	14	6	39
Misclassified objects	0	0	1
False-positive	3	2	3
Total number of objects	41	41 x 2	41 x 2

Figure 22. Evaluation tests statistics

As can be seen, the fastest solution is Ultralytics Demo, which uses PyTorch for neural networks running. It achieves more than 60 FPS on GPU using Vulkan API and almost 48 FPS on CPU. But also, it is a solution with the highest number of mistakes – more than 47% of objects were not detected, one object was misclassified, 3 more not existing object were detected.

The most accurate solution is a custom YOLOv5 Unity Barracuda detector, which did not catch only 6 objects out of 82 and detected 2 extra objects, with no misclassification. It is important to say that this solution runs faster on a CPU than on a GPU, achieving almost 6 FPS, but it is still slower than on other solutions.

A TensorFlow Lite solution using EfficientDet-D0 appears to be faster than YOLOv5 Barracuda with almost 7 FPS frequency on CPU. Number of mistakes is lower than on Ultralytics demo, but still, it is much higher than on the Barracuda solution.

7 Conclusion

7.1 Achieved goals

In this work the most part of goals is achieved. A current state of object detection with CNN problem is explored and analysed.

Some of the models described in this work have been re-trained on a custom dataset, what made them possible to detect Fiducial markers, such as ArUco markers and QR codes. Training mechanisms' workflows are scrutinized, so now it is possible to add new types of markers on a request.

The ways of integration current neural network libraries and packages (TensorFlow, PyTorch, Barracuda) to Unity are studied as well. The Barracuda integration option is elaborated on a practical example. For other variants demo applications were used.

Evaluation tests for three variants are provided to make a choice for a potential integration to Misterine App.

7.2 Not achieved goals and problems

Not every one-stage model mentioned in this paper was described or re-trained on a custom dataset, as no well-described training mechanisms were found for some of them. Also, not every possible way of Unity integration was elaborated practically to get more relevant data. Some of the mentioned ways seems to be obsolete.

For some models not the most relevant statistics were collected (for SqueezeDet, for example, because no public data on COCO dataset were found).

Not every way of neural networks integration into Unity had not been tried out, so no practical results were collected. It may be a problem, as Unity Mono scheduling can cause unexpected performance results.

No post-processing was introduced to get precise coordinates and rotation of the objects.

7.3 Further possible development

In the future, a custom model architecture can be developed. It can take an only one grayscale component instead of 3 components in RGB, as the most of fiducial markers have got high contrast. Theoretically, it can increase speed and decrease energy consumption. The model can be based on one of the models' architectures described in this work.

Also, other ways of models' integration can be explored practically, especially the PyTorch's C++ API, as it has shown a very good result in the

meaning of speed. But it should be counted with the issue of the Unity's architecture using Mono as its platform.

The other thing to mention is that usage of CoreML by Apple and NNAPI by Google can be explored. These technologies can not only run neural networks on GPUs, but also, they can optimize the runtime.

As no post-processing was integrated in this work, it can be done as well.

7.4 Recommendations

A solution using neural networks can be theoretically used for the detection of fiducial markers. The speed of PyTorch solution seems to be high enough for mobile real-time detection, but the accuracy of this solution is unacceptably low, so a way to increase it should be found.

The field of object detectors studies is developing in time, so new solutions, if they will be introduced, should be studied as well.

Other objects can be tried out as fiducial markers for a solution using an object detector based on a convolutional neural network.

8 Attachments

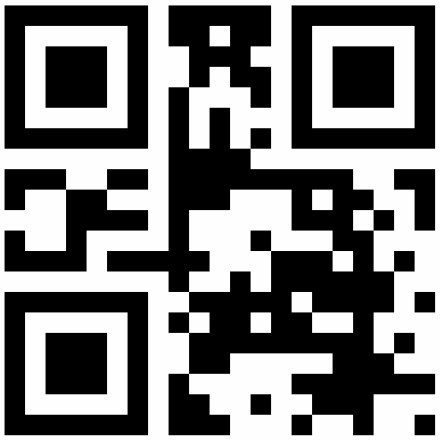
8.1 Links

1. "YOLOv5-Unity" project on GitHub
   https://github.com/egor-ulianov/yolov5-unity
2. Dataset generator on GitHub
   https://github.com/egor-ulianov/dataset-generator-fiducial

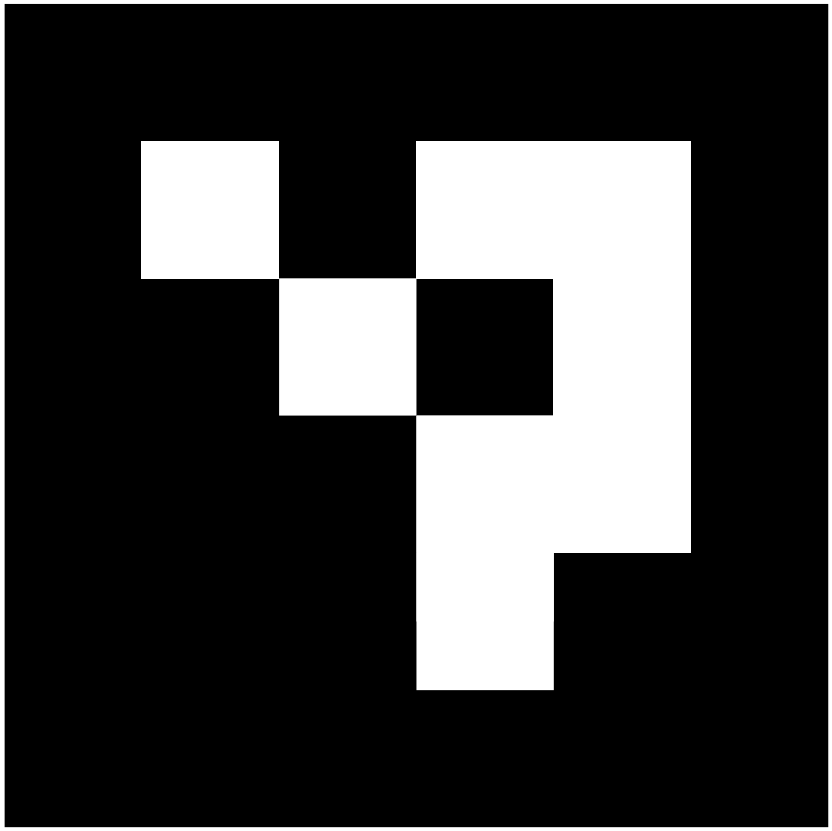
8.2 Data examples

8.2.1 A single QR code

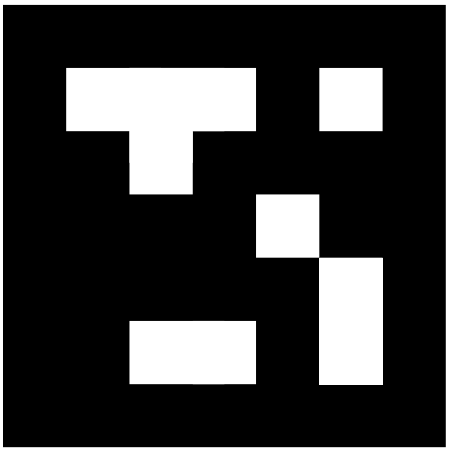
8.2.2 Different QR codes of different sizes

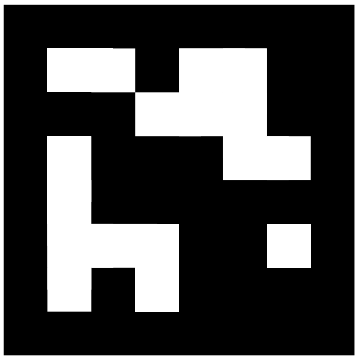


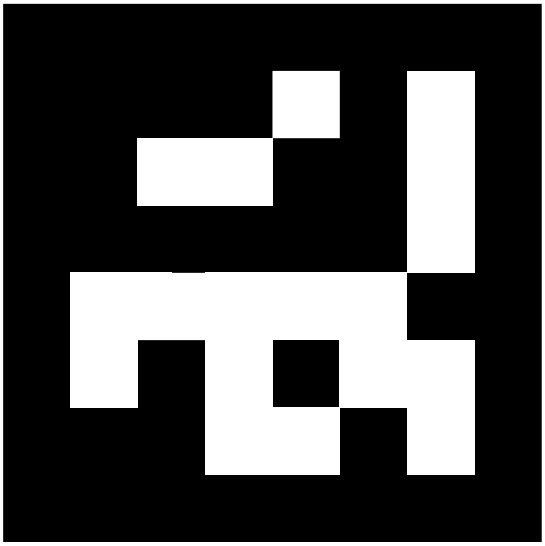
8.2.3 A single ArUco marker

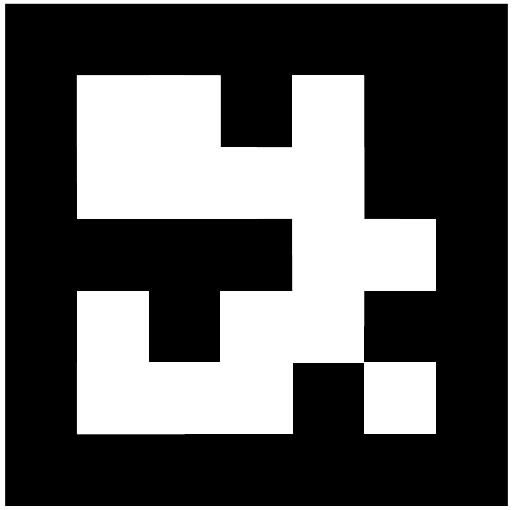


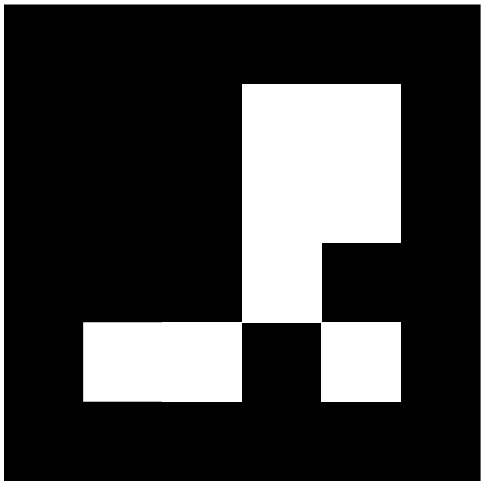
8.2.4 Different ArUco markers of a different size



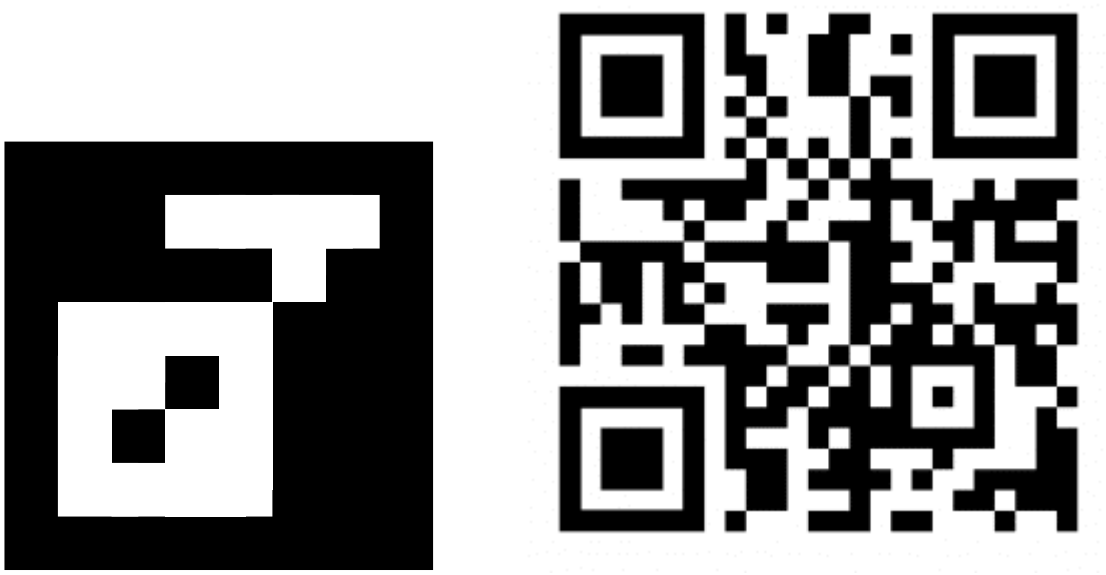


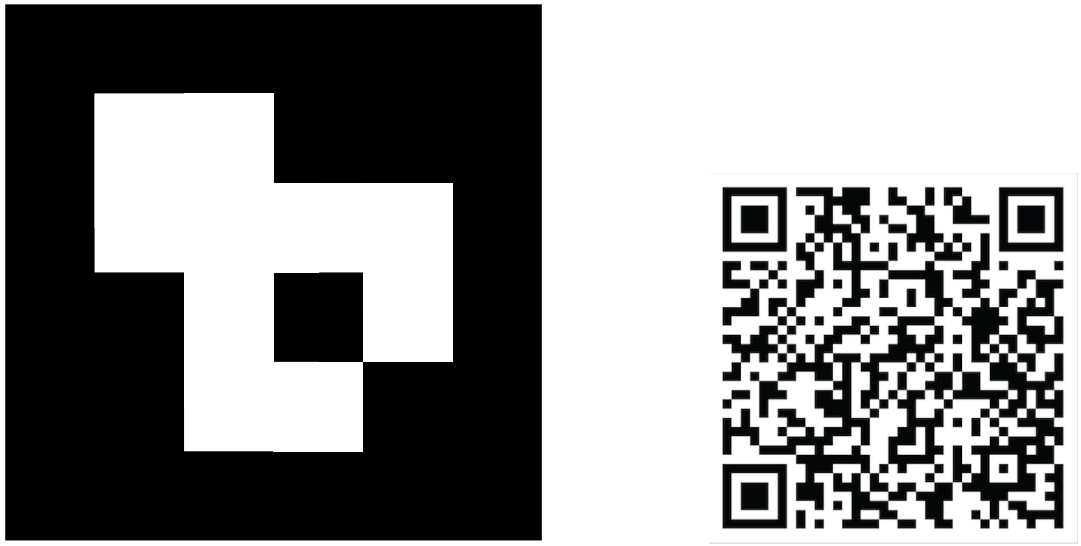






8.2.5 ArUco markers and QR codes altogether





9 Bibliography

[1] S. Ullman, High Level Vision, MIT Press, 1996.

[2] M. H. Q. M. M. Anthony D. Rhodes, Portland, OR: Portland State University, 2016.

[3] M. J. Paul Viola, "Rapid Object Detection using a Boosted Cascade of Simple," Cambridge, MA, 2001.

[4] B. T. N. Dalal, "Histograms of oriented gradients for human detection," San Diego, CA, 2005.

[5] R. B. G. D. M. D. R. Pedro F. Felzenszwalb, *Object Detection with Discriminatively Trained Part-Based Models,* 2009.

[6] J. D. T. D. J. M. Ross Girshick, *Rich feature hierarchies for accurate object detection and semantic segmentation,* Berkeley, CA: UC Berkeley, 2014.

[7] H. L. S. G. B. A. G. B. Y. W. P.-J. K. M. T. V. S. B. C. Yunyang Xiong, *MobileDets: Searching for Object Detection Architectures for Mobile,* arXiv, 2020.

[8] P. R. L. Q. V. Tan Mingxing, *EfficientDet: Scalable and Efficient Object Detection,* arXiv, 2019.

[9] D. S. G. R. F. A. Redmon Joseph, *You Only Look Once: Unified, Real-Time Object Detection,* arXiv, 2015.

[10] "Apple A15 (5 GPU Cores)," [Online]. Available: https://www.cpu-monkey.com/en/igpu-apple_a15_5_gpu_cores-275.

[11] "NVIDIA GeForce GTX TITAN X GPU specs," [Online]. Available: https://www.gpuzoo.com/GPU-NVIDIA/GeForce_GTX_TITAN_X.html.

[12] F. A. Redmon Joseph, "YOLO: Real-Time Object Detection," 2018. [Online]. Available: https://pjreddie.com/darknet/yolo/.

[13] J. a. F. A. Redmon, *YOLOv3: An Incremental Improvement,* arXiv, 2018.

[14] W. C.-Y. L. H.-Y. M. Bochkovskiy Alexey, *YOLOv4: Optimal Speed and Accuracy of Object Detection,* arXiv, 2020.

[15] G. Jocher, "ultralytics/yolov5," Github, [Online]. Available: https://github.com/ultralytics/yolov5.

[16] F. A. Redmon Joseph, "YOLO: Real-Time Object Detection," [Online]. Available: https://pjreddie.com/darknet/yolo/.

[17] B. A. L. H.-Y. M. Wang Chien-Yao, "AlexeyAB/darknet," [Online]. Available: https://github.com/AlexeyAB/darknet.

[18] songzhifei, "how to convert darknet .weight to pytorch .pt file #281," 2020. [Online]. Available: https://github.com/Tianxiaomo/pytorch-YOLOv4/issues/281.

[19] remc, "Darknet model to onnx," 2020. [Online]. Available: https://stackoverflow.com/questions/62673115/darknet-model-to-onnx.

[20] D. A. D. E. C. S. S. R. C.-Y. F. A. C. B. Wei Liu1, *SSD: Single Shot MultiBox Detector,* 2016.

[21] H. Jonathan, "SSD object detection: Single Shot MultiBox Detector for real-time processing," 2018. [Online]. Available: https://jonathan-hui.medium.com/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06.

[22] dhaase-de. [Online]. Available: https://github.com/dhaase-de/caffe-tensorflow-python3.

[23] ethereon. [Online]. Available: https://github.com/ethereon/caffe-tensorflow.

[24] Roboflow, "EfficientDet-D0-D7," [Online]. Available: https://models.roboflow.com/object-detection/efficientdet-d0-d7.

[25] Roboflow, "EfficientDet," [Online]. Available: https://models.roboflow.com/object-detection/efficientdet.

[26] W. A. I. F. J. P. H. ,. K. K. Wu Bichen, "SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving," arXiv, 2016.

[27] BichenWuUCB, "BichenWuUCB/squeezeDet," [Online]. Available: https://github.com/BichenWuUCB/squeezeDet.

[28] E. Christopher, "Fast object detection with SqueezeDet on Keras," 2018. [Online]. Available: https://medium.com/omnius/fast-object-detection-with-squeezedet-on-keras-5cdd124b46ce.

[29] J. W. X. T. B. W. Boxuan Li, *ArUco Marker Detection under Occlusion Using Convolutional Neural Network,* Dalian: IEEE, 2020.

[30] Kukil, "WeChat QR Code Scanner in OpenCV," 23 Nov 2021. [Online]. Available: https://learnopencv.com/wechat-qr-code-scanner-in-opencv/?nowprocket=1.

[31] G. B. Portmann, "LOCALIZING QR CODES WITH YOLOV3," 2019. [Online]. Available: https://www.gabriel-bellport.com/projects/localizing-qr-codes-with-yolov3-2019.

[32] Barracuda, 2021. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.barracuda@3.0/manual/Exporting.html.

[33] Barracuda, 2021. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.barracuda@3.0/manual/FAQ.html.

[34] Barracuda, "License," 2021. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.barracuda@3.0/license/LICENSE.html.

[35] G. Nicholas, "Neural Networks in Unity using Native Libraries," 2020. [Online]. Available: https://www.goodai.com/neural-networks-in-unity-using-native-libraries/.

[36] "PyTorch," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/PyTorch.

[37] "TensorFlow," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/TensorFlow.

[38] migueldeicaza, "migueldeicaza/TensorFlowSharp," [Online]. Available: https://github.com/migueldeicaza/TensorFlowSharp.

[39] chuanqi305, "mobilenet-ssd," 2018. [Online]. Available: https://docs.openvino.ai/latest/omz_models_model_mobilenet_ssd.html.

[40] david8862, "yolo-v4-tiny-tf," 2019. [Online]. Available: https://docs.openvino.ai/latest/omz_models_model_yolo_v4_tiny_tf.html.

[41] Roboflow, "Roboflow Pricing and Plans," [Online]. Available: https://roboflow.com/pricing.

[42] G. Jocher, "YOLOv5 Tutorial," 2022. [Online]. Available: https://colab.research.google.com/github/ultralytics/yolov5/blob/master/tutorial.ipynb.

[43] G. Jocher, "ultralytics/hub," 2021. [Online]. Available: https://github.com/ultralytics/hub#1-create-a-dataset.

[44] J. N. Jacob Solawetz, "Training EfficientDet Object Detection Model with a Custom Dataset," 2020. [Online]. Available: https://blog.roboflow.com/training-efficientdet-object-detection-model-with-a-custom-dataset/.

[45] "Google Colab," [Online]. Available: https://colab.research.google.com/drive/1ZmbeTro4SqT7h_TfW63MLdqbrCUk_1br#scrollTo=KwDS9qqBbMQa.