Nástroj pro zpracování obrazu a vyhodnocení experimentů s roboty typu Kilobot s využitím OpenCV a jazyka C/C++

Bc. Petr Svoboda

Diplomová práce
2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Petr Svoboda**
Osobní číslo:	**A20149**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Nástroj pro zpracování obrazu a vyhodnocení experimentů s roboty typu Kilobot s využitím OpenCV a jazyka C/C++**
Téma práce anglicky:	**A Tool for Image Processing and Evaluation of Experiments with Kilobot Robots Using OpenCV and C/C++ Language**

Zásady pro vypracování

1. Seznamte se se základními principy knihovny OpenCV se zaměřením na detekci objektů v obraze.
2. Navrhněte metodu automatické kalibrace obrazu a určení její vzdálenosti od podložky.
3. Navrhněte algoritmus pro detekci a sledování robota v obraze kamery.
4. Zaměřte se na možnosti určení vektoru natočení robota a sledování trajektorie.
5. Proveďte ukázkovou implementaci navržených algoritmů do podoby knihovny jazyka C/C++.
6. Experimentálně ověřte výslednou vytvořenou knihovnu.

Forma zpracování diplomové práce: **Tištěná**

Seznam doporučené literatury:

1. RUBENSTEIN, Michael, Christian AHLER a Radhika NAGPAL. Kilobot: A low cost scalable robot system for collective behaviors. In: 2012 IEEE International Conference on Robotics and Automation [online]. IEEE, 2012, 2012, s. 3293-3298 [cit. 2021-11-26]. ISBN 978-1-4673-1405-3. Dostupné z: doi:10.1109/ICRA.2012.6224638

2. RUBENSTEIN, Michael, Christian AHLER, Nick HOFF, Adrian CABRERA a Radhika NAGPAL. Kilobot: A low cost robot with scalable operations designed for collective behaviors. Robotics and Autonomous Systems [online]. 2014, 62(7), 966-975 [cit. 2021-11-26]. ISSN 09218890. Dostupné z: doi:10.1016/j.robot.2013.08.006

3. REINA, Andreagiovanni, Alex J. COPE, Eleftherios NIKOLAIDIS, James A. R. MARSHALL a Chelsea SABO. ARK: Augmented Reality for Kilobots. IEEE Robotics and Automation Letters [online]. 2017, 2(3), 1755-1761 [cit. 2021-11-26]. ISSN 2377-3766. Dostupné z: doi:10.1109/LRA.2017.2700059

4. VIRIUS, Miroslav. Programování v C++: od základů k profesionálnímu použití. Praha: Grada Publishing, 2018. Myslíme v.. ISBN 978-80-271-0502-1.

5. KAEHLER, Adrian a Gary R. BRADSKI. Learning OpenCV 3: computer vision in C++ with the OpenCV library. Sebastopol: O'Reilly, 2016. ISBN 978-1-4919-3799-0.

Vedoucí diplomové práce: **Ing. Michal Pluháček, Ph.D.**
Ústav automatizace a řídicí techniky

Konzultant diplomové práce: **Ing. Peter Janků, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **3. prosince 2021**
Termín odevzdání diplomové práce: **23. května 2022**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne ……………………….

podpis studenta

ABSTRAKT

Práce popisuje možnosti OpenCV v oblasti detekce a sledování objektů v obraze pro účely experimentů s roboty typu Kilobot. Teoretická část se zaměřuje na popis vybraných metod pro detekci objektů v obraze, určení trajektorie, úhlů natočení a vzdálenosti od kamery, zejména pak na konvoluční neuronové sítě a algoritmus YOLO v5. V praktické části je popsáno trénování modelu YOLO v5 a výsledné řešení navržené open-source knihovny v jazyce C++.

Klíčová slova: Kilobot, OpenCV, C++, Python, DNN, Microsoft Visual Studio, Google Colab, Detekce objektů v obraze, Sledování bojektů v obraze

ABSTRACT

The work describes the possibilities of OpenCV in the field of object detection and tracking for the purpose of experiments with Kilobots robots. The theoretical part focuses on the description of selected methods for detecting objects in the image, determining the trajectory, rotation angles and distance from the camera, especially on convolutional neural networks and the YOLO v5 algorithm. The practical part describes the training of the YOLO v5 model and the resulting solution of the designed open-source library in C++.

Keywords: Kilobot, OpenCV, C++, Python, DNN, Microsoft Visual Studio, Google Colab, Object detection, Object tracking

Tímto děkuji pánům Ing. Michalovi Pluháčkovi Ph.D. a Ing. Peterovi Janků Ph.D. za vedení a rady při tvorbě této práce.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD **9**

I TEORETICKÁ ČÁST **10**

1 STROJOVÉ UČENÍ **11**

1.1 Učení s učitelem 11

1.2 Učení bez učitele 11

1.3 Kombinované 11

1.4 Model a data 11

2 HEJNOVÁ INTELIGENCE **12**

2.1 Kilobot 12

2.2 Design Kilobota 13

3 NEURONOVÉ SÍTĚ **14**

3.1 Struktura 14

3.2 Normalizace dat 15

3.3 Umělý neuron 15

3.4 Konvoluční neuronové sítě 16

3.4.1 Konvoluční vrstva 16

3.4.2 Pooling vrstva 16

3.4.3 Plně propojená vrstva 17

3.4.4 Vrstva Softmax 17

4 DETEKCE OBJEKTŮ V OBRAZE POMOCÍ HLUBKOÝCH NEURONOVÝCH SÍTÍ **18**

4.1 Dvoufázové metody detekce 18

4.2 Jednofázové metody detekce 18

4.2.1 YOLO 19

4.2.1.1 Historie 19

4.2.1.2 Princip 19

5 YOLO V5 .... **20**

5.1 Google Colab 20

5.2 Lokální přístup 20

6 OPENCV **21**

6.1 Moduly 21

6.2 Vybrané třídy 22

6.2.1 Point 22

6.2.2 Rect 22

6.2.3 Scalar 22

6.2.4 Mat 22

6.3 Vybrané funkce 23

6.3.1 threshold 23

6.3.2 findContours a drawContours 23

6.3.3 cvtColor 23

6.3.4 line, rectangle, ellipse a circle 24

6.3.5 Bitové operace 24

6.4 DETEKCE KILOBOTŮ POMOCÍ OPENCV ... 24
6.4.1 Detekce kontur ... 24
6.4.2 Spárování s šablonou ... 24
6.4.3 Houghova kruhová transformace ... 25
6.4.4 Kaskádové klasifikátory ... 25
6.4.5 Hluboké neuronové sítě ... 25
6.5 MODULY TRACKING A LEGACY ... 26
6.6 MODUL DNN ... 26
6.6.1 Třída Net ... 26
6.6.2 blobFromImage ... 26
6.6.3 NMSBoxes ... 26
7 URČENÍ VZDÁLENOSTI OD KAMERY ... 28
7.1.1 Model dírkové komory ... 28
7.1.2 Určení vzdálenosti od kamery ... 29
8 TRACKING ... 30
8.1 TRACKERY OPENCV ... 30
8.2 TRACKING POMOCÍ NEJMENŠÍ EUKLEIDOVSKÉ VZDÁLENOSTI ... 31
8.2.1 Výpočet vzdálenosti ... 31
8.2.2 Algoritmus ... 32
9 KALIBRACE OBRAZU ... 35
II PRAKTICKÁ ČÁST ... 37
10 SBĚR A PREPROCESSING DAT ... 38
10.1 ZÁKLADNÍ ÚPRAVA ... 38
10.2 ZNAČENÍ ... 38
10.3 STRUKTURA ... 40
11 TRÉNOVÁNÍ MODELU YOLOV5 ... 41
11.1 ZAVEDENÍ YOLO V5 ... 41
11.2 PŘÍPRAVA SOUBORŮ ... 41
11.3 TRÉNOVÁNÍ MODELU ... 42
11.4 EXPORT MODELU DO FORMÁTU ONNX ... 43
11.5 POUŽITÁ DATA ... 43
12 KNIHOVNA KILOLIB ... 44
12.1 KALIBRACE KAMERY ... 44
12.1.1 Nalezení bodů potřebných pro výpočet ... 44
12.1.2 Uložení a načtení vypočtených matic ... 47
12.2 STRUKTURA KILOBOT ... 48
12.3 MĚŘENÍ VZDÁLENOSTI ... 48
12.3.1 Naměřená data ... 48
12.3.2 Funkce measureDistance ... 49
12.3.3 centerKilobotDetection ... 50
12.4 NALEZENÍ LED ... 52
12.4.1 Odfiltrování vnější části mimo Kilobota ... 52
12.4.2 Odfiltrování vnitřní části Kilobota ... 53
12.4.3 Aplikace prahové hodnoty ... 53

12.4.4 Vyhledání kontur v oblasti zájmu 54
12.4.5 Určení kontury zaujímající největší oblast 54
12.5 DETEKCE POMOCÍ NEURONOVÉ SÍTĚ 55
12.5.1 LoadNet 56
12.5.2 _format 57
12.5.3 Detect 58
12.6 TRACKING 62
12.6.1 Reset 63
12.6.2 _findObject 63
12.6.3 Track 64
12.7 UKÁZKA POUŽITÍ 66
12.7.1 Kalibrace 66
12.7.2 Měření vzdáleností 67
12.7.3 Tracking 68
13 VÝSLEDKY 71
13.1 KALIBRACE KAMERY 71
13.2 DETEKCE KILOBOTA 71
13.3 MĚŘENÍ VZDÁLENOSTI 72
13.4 TRAJEKTORIE KILOBOTA 73
13.5 DETEKCE LED DIODY 74
ZÁVĚR 76
SEZNAM POUŽITÉ LITERATURY 77
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 82
SEZNAM OBRÁZKŮ 83
SEZNAM TABULEK 84
SEZNAM PŘÍLOH 85

ÚVOD

Aniž bychom si to uvědomovali, neustálý pokrok v oblasti strojového učení se denně dotýká každého z nás. Setkáme se s ním například při vyhledávání na Googlu, u zobrazovaných reklam na YouTube nebo při vyhodnocování spamu v elektronické poště. Mnoho algoritmů v této oblasti se inspiruje jevy z biologie, fyziky či chemie, řečeno jinými slovy z přírody. Díky jejich jednoduchosti, flexibilitě a použitelnosti při řešení optimalizačních problémů obliba těchto algoritmů i celé oblasti strojového učení nabývá na popularitě. Zmíněné algoritmy se stávají základem pro mnoho moderních aplikací napříč různými odvětvími.

Tato práce se dotýká problematiky tzv. Swarm Intelligence, neboli hejnové či rojové inteligence, která se snaží navrhovat algoritmy inspirované chováním živočichů, kteří nejsou schopni plnit pro ně obtížné úkoly sami, ale jako skupina si s nimi dokáží poradit, díky lokální komunikaci, aniž by byla skupina centrálně řízena.

Výzkumem v této oblasti se již řadu let zabývají též výzkumníci z Univerzity Tomáše Bati ve Zlíně, v posledních letech i díky experimentů s roboty typu Kilobot.

Kilobot je malý robot o velikosti 3,3 cm navržený tak, aby byl levný, lehce sestavitelný, schopný pohybu a interakce s okolními Kiloboty [1]. Díky zmíněným vlastnostem jsou výzkumníci schopni realizovat experimenty s řádově desítkami i stovkami těchto malých robotů. Vyhodnocování experimentů, je vzhledem k počtu robotických jednotek poměrně složitě a v současné době neexistuje univerzálně použitelný nástroj. Existující nástroje jsou obvykle navrženy pro specifické hardwarové vybavení dané laboratoře a nepočítají např. se změnou umístění kamery nebo sklonem obrazu [2]

Práce si klade za cíl implementovat řešení detekce Kilobota v obraze pro různé pozice kamery a dát výzkumníkům možnost získávat z těchto experimentů data jako je trasa, kterou při experimentu Kilobot urazil nebo úhel natočení robota v konkrétním bodě trajektorie. Takto vznikne univerzálněji použitelný nástroj pro využití v dalším výzkumu.

Výsledkem práce je open source knihovna implementovaná za pomocí jazyka C++ a knihovny OpenCV, kterou budou moci výzkumníci dále pro své potřeby upravovat.

I. TEORETICKÁ ČÁST

1 STROJOVÉ UČENÍ

Jedná se o oblast umělé inteligence, která se řídí myšlenkou, že počítačový systém se může učit na základě poskytnutých dat. Algoritmy z této oblasti jsou schopny pracovat s konkrétními daty a vyrábět predikce. Například u již zmíněné filtrace spamu v emailech můžeme za pomocí velkého množství emailů, kde rozdělíme příchozí poštu na vyžádanou a nevyžádanou, naučit určitý algoritmus je od sebe rozeznávat. Nejčastější způsoby strojového učení označujeme jako učení s učitelem, bez učitele, nebo jejich kombinaci. [3]

1.1 Učení s učitelem

Tento typ učení potřebuje znát k jemu zadaným vstupům i výstupy. Pokud budeme mít úlohu, kde se budeme snažit rozlišit od sebe psi a kočky, tak algoritmus potřebuje vědět, na kterých snímcích jsou kočky a na kterých psi. Cílem algoritmu by pak mělo být nalezení pravidla, podle kterého bude schopen od sebe jednotlivá zvířata rozlišit. [3]

1.2 Učení bez učitele

U učení bez učitele jsou algoritmu dána pouze vstupní data bez jakéhokoliv označení. U výše zmíněného problému klasifikace psů a koček by tedy algoritmus musel nejprve nalézt společné rysy jednotlivých zvířat a poté je až klasifikovat. Společnými rysy může být prakticky cokoliv například jejich velikost, tvar těla nebo barva. [3]

1.3 Kombinované

Oba předchozí způsoby lze samozřejmě zkombinovat. V praxi to znamená, že označíme pouze část vstupních dat. Algoritmy tohoto typu pak mají na čem zakládat, i tak ale musí být schopny určit společné vlastnosti dat. [3]

1.4 Model a data

Algoritmy strojového učení pracují tak, že prochází dostupná trénovací data a snaží se je vhodně generalizovat. Jejich cílem je jim porozumět a nalézt mezi nimi vzory a souvislosti. Jinými slovy se dá říct, že se snaží vytvořit z dostupných dat určitý model, který je spustitelný ve formě počítačového kódu a který může být následně použitý pro klasifikace, predikce a odhadování i na datech, která jsou různá od dat, na kterých byl model trénován. [4]

2 HEJNOVÁ INTELIGENCE

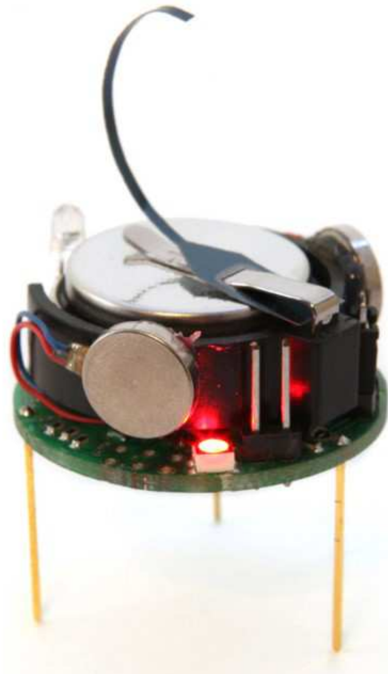
Hejnová inteligence (někdy též překládáno jako Rojová) se týká kolektivního chování skupiny „agentů“, kteří následují určitá pravidla. Každý z agentů může být samostatně považován za neinteligentního, kdežto kolektiv těchto neinteligentních agentů může vykazovat známky inteligentního chování. Společnými vlastnostmi algoritmů založených na hejnové inteligenci jsou sdílení informací mezi jednotlivými agenty, jejich schopnost samoorganizace a kolektivní evoluce, vysoká efektivita díky společnému učení a jednoduchá paralelizace pro praktické využití na reálných problémech. [5]

2.1 Kilobot

Se zvyšující se mírou pokroku v oblasti robotiky začali v roce 2010 na Harvardské Univerzitě vznikat roboti nesoucí název Kilobot. Ti vznikli zejména proto, jelikož výzkum v oblasti hejnové inteligence se do té doby dal provádět pouze za pomocí simulací nebo testování na jednotkách cenově nedostupných robotů. Pro účely ověření správnosti některých navržených algoritmů, je však potřeba je testovat na větším množství robotů. Ti jsou totiž schopni jako velká skupina dosáhnout cílů, kterých by jednotlivec nebo pár jedinců sami nedosáhli. Inspirací navržených algoritmů pro tyto účely je sama příroda. Tam můžeme vidět například mravence, kteří jsou společnými silami schopni přemístit předmět, který je 50x těžší než hmotnost celé jejich skupiny. [6]

Kilobot byl navržen tak, aby za nízkou cenu poskytoval dostatečnou funkcionalitu. Oproti podobným robotům jsou výhodami jeho jednoduchost a nízká cena, necelých 14 dolarů za kus. Podobní roboti jako e-puck nebo Jasmine stojí desetkrát i stokrát více. [6]

2.2 Design Kilobota



Obrázek 1. Kilobot [7]

Základní motivací designu Kilobota bylo, aby byl schopen realizovat algoritmus S-DASH, pomocí kterého je hejno schopné samo sestavit a opravit požadovaný tvar. Pro tyto účely je potřeba, aby byli roboti schopni pohybu dopředu, otáčení, komunikace a měření vzdálenosti k okolním robotům a zároveň mít dostatek paměti na uchování a běh zmíněného algoritmu. Pohyb Kilobota zajišťují dva vibrační motory po stranách a komunikace je zajištěna infračerveným vysílačem a přijímačem ve spodní částí. Kvůli větší univerzálnosti byla přidána i další kritéria jako měření úrovně okolního osvětlení, zobrazení interního stavu pro účely debuggování a umožňovat škálovatelnost operací. [6]

3 NEURONOVÉ SÍTĚ

Jedním z typů algoritmů spadajících do odvětví zvaného hluboké učení, které je součástí strojového učení jsou neuronové sítě. Jsou vhodné zejména pro řešení problémů jako rozpoznávání objektů v obraze nebo zpracování jazyka. Oproti konvenčním algoritmům se neuronové sítě snaží přistupovat k problémům podobně jako neurony v lidském mozku a to tak, že se učí na bázi poskytnutých příkladů. Stejně jako lidský mozek se neuronové sítě z velkého množství propojených uzlů pracujících paralelně. [3]

3.1 Struktura

Neuronové sítě se skládají ze skupin neuronů, které nazýváme vrstvy. Každá ze sítí má alespoň jednu vstupní a jednu výstupní vrstvu. Program přivede na vstup vstupní vrstvy vzorek, který neuronová síť zpracuje a vyprodukuje vzorek na výstupu. To, co se děje mezi vstupní a výstupní vrstvou je dáno architekturou sítě, která může být pro konkrétní problémy různá. Na obrázku XX je tato část označena jako „Hidden Layers“ – skryté vrstvy. [8]

Obrázek 2. Neuronová síť [8]

Zmíněnými vzorky, které jsou vstupem a výstupem neuronové sítě, rozumíme určité pole desetinných čísel. Počet neuronů v jednotlivých vrstvách udává velikost těchto polí a je neměnný. Abychom mohli neuronovou síť použít pro řešení konkrétního problému, je ho nejprve potřeba vyjádřit jako pole desetinných čísel. Jediné, co neuronová síť dokáže je transformovat toto pole do jiného pole desetinných čísel. Jinak řečeno, neuronová síť je schopna na základě naučených znalostí z množství předchozích dat predikovat výstup na dalších, pro ni kompletně nových datech. [8]

Příklad vstupu a výstupu neuronové sítě, kde je vstupní vrstva složena ze tří neuronů a výstupní ze dvou:

Vstup: [0.14, 0.14, 0.1]

Výstup [0.25, 0.84]

3.2 Normalizace dat

Jak lze vidět na příkladu výše, u vstupu a výstupu jsou pouze data v rozsahu hodnot 0 až 1. U reálných problémů se ale mohou vykytovat rozsahy zcela odlišné. K transformaci dat do rozsahu 0 až 1 slouží tzv. normalizace. [8]

Pokud bychom v originálních datech měli například hmotnost, která by byla v rozmezí 200 – 10 000 kg, tak konkrétní hodnotu 3000 kg můžeme normalizovat relativně jednoduchým výpočtem:

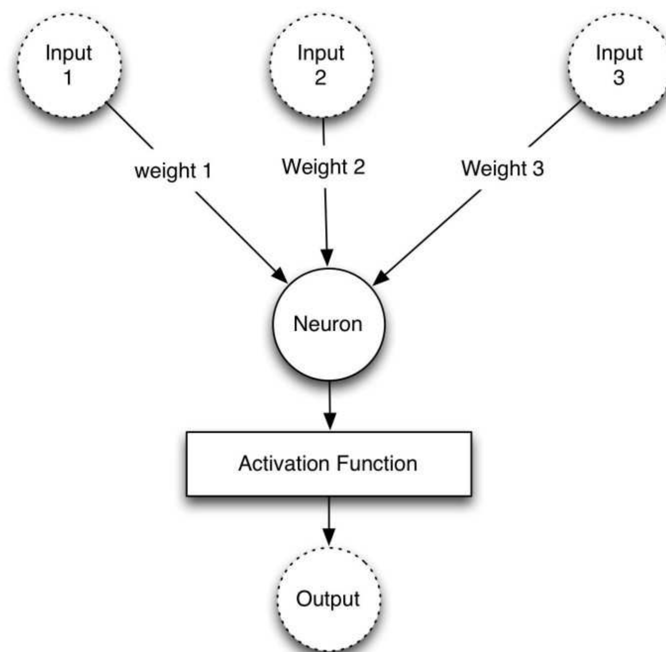
Hodnota od začátku rozsahu: 3 000 - 200 = 2800

Celkové rozmezí hodnot: 10 000 – 200 = 9 800

Procentuální hodnota jako desetinné číslo: 2 800 / 9 800 = 0.29

Na vstup neuronové sítě bychom pro hodnotu 3000 kg předali normalizovanou hodnotu 0.29.

3.3 Umělý neuron



Obrázek 3. Schéma umělého neuronu [8]

Jak již bylo zmíněno, celá neuronová síť se skládá z velikého množství umělých neuronů. Ty pracují tak, že po obdržení vstupů od programu nebo dalších neuronů je nejprve podělí jim předělenými váhami, výsledné hodnoty spolu sečte a jejich sumu předá na vstup aktivační funkce. [8]

3.4 Konvoluční neuronové sítě

Jedná se o druh neuronových sítí, složený zejména z párů konvolučních a pooling vrstev dále vedoucích do vrstev plně propojených a vrstvy softmax. Při využití konvoluce společně s neuronovými sítěmi bylo dosaženo vyšších přesností hlavně v problematice rozpoznávání obrazu. Konvoluční neuronová síť pracuje tak, že snímek rozdělí na části, na kterých se dále pokouší identifikovat hrany nebo binární data (označovány jako BLOB = Binary large object). [8]

3.4.1 Konvoluční vrstva

Primárním úkolem konvoluční vrstvy je detekovat rysy jako jsou hrany, čáry nebo jiné vizuální prvky. Je toho schopna díky použití filtrů, což jsou vlastně čtvercové mřížky, které postupně prochází snímek zleva doprava. Konvoluční vrstva bývá z pravidla definována několika parametry, kterými jsou počet filtrů, velikost filtru, krok, zarovnání a aktivační funkce. [8]

Konvoluční vrstvy pracují tak, že aplikují zadané filtry na vstupní snímek a „zmapují" výskyt hledaných rysů ve snímku. Jsou velice efektivní zejména díky možnosti skládání více vrstev za sebou, kde vrstvy blíže ke vstupnímu snímku nejprve rozlišují detailní vlastnosti objektů (například čáry) a vrstvy následující za nimi jsou již schopny rozlišit abstraktnější rysy jako tvary nebo konkrétní objekty. Limitace zmíněného mapování rysů je zejména ta, že je velice náchylné na minimální změny jako jsou posun, otočení nebo ořezaní vstupního snímku. Z tohoto důvodu je vhodné využít výhod downsamplingu, který bývá nejčastěji realizován pomocí vyvedení výstupu konvolučních vrstev do tzv. pooling vrstev. [9]

3.4.2 Pooling vrstva

Přidání pooling vrstvy za vrstvu konvoluční je u konvolučních neuronových sítí velice časté. Ta většinou pracuje tak, že vytvoří mřížku o velikosti 2x2 pixelů, kterou ve snímku posouvá o 2 pixely a podle vybraného způsobu buď z aktuálních 4 pixelů vybere maximální hodnotu (Maximum Pooling) nebo je zprůměruje (Average Pooling). Postupně tak zredukuje výstup konvoluční vrstvy na polovinu – z výstupu 4x4 udělá výstup o velikosti 2x2 atd. [9]

V tabulkách níže je znázorněn princip poolingu pomocí maximální hodnoty. V první z nich jsou různými barvami označeny vždy 4 pixely, ze kterých je vybírána maximální hodnota.

1	4	6	5
5	1	2	1
7	8	1	4
1	2	2	5

Tabulka 1. Výstup konvoluční vrstvy

V druhé tabulce je znázorněn výsledek zmíněné redukce, kdy ze 4 pixelů je vždy vybrán ten s největší hodnotou.

5	6
8	5

Tabulka 2. výstup pooling vrstvy

3.4.3 Plně propojená vrstva

Za konvoluční a pooling vrstvy bývají umístěné vrstvy plně propojené. Jsou zde zejména proto, jelikož výstup konvolučních vrstev většinou náleží pouze části původního snímku, protože jejich receptivní pole není schopné pokrýt celý prostorový rozměr obrazu. [10]

Jak lze již z názvu poznat, vrstvy plně propojené propojují všechny vstupy z jedné vrstvy na všechny aktivační jednotky vrstvy následující. Tyto vrstvy mají v konvoluční neuronové síti za úkol sestavit extrahovaná data z konvolučních vrstev. [11]

3.4.4 Vrstva Softmax

Tato vrstva obsahuje aktivační funkci, která se používá ke klasifikaci nalezených objektů do tříd, kde každá z nich je reprezentována jedním neuronem. Výstupem této vrstvy jsou hodnoty pravděpodobnosti, s kterými neuronová síť určila, že se jedná o jednu ze známých tříd. Při rozlišování 3 různých tříd může být výstupem této vrstvy například to, že vstup klasifikovala s jistotou 80 % jako první třídu, 15 % jako druhou třídu a 5 % jako poslední třídu. Součet musí vždy dávat 100 %. [8]

4 DETEKCE OBJEKTŮ V OBRAZE POMOCÍ HLUBKOÝCH NEURONOVÝCH SÍTÍ

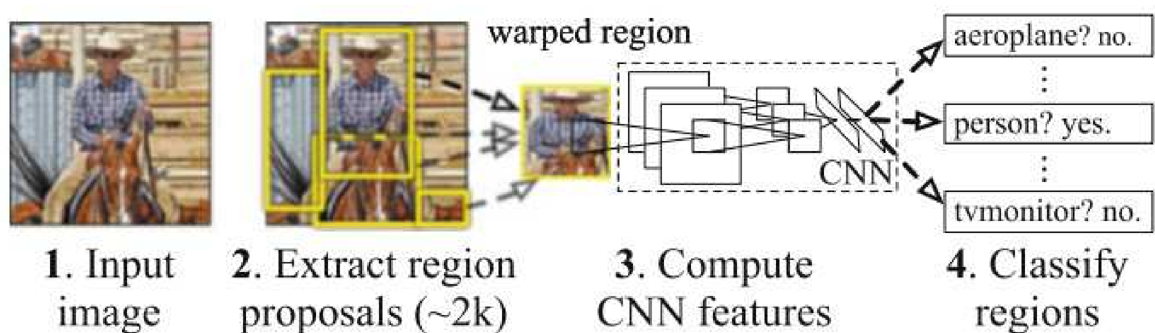
V posledních letech stále více dochází k využití hlubokých neuronových sítí pro účely detekce a rozpoznávání objektů v obraze, které jsou klíčovými prvky v oblasti počítačového vidění a zpracování obrazu. Během fáze trénování jsou schopny sami rozpoznat vhodné vlastnosti z poskytnutých dat, a to i na datech získaných v různých světelných podmínkách nebo s jiným úhlem natočení. [12]

Metod, které lze pro tyto účely použít je více. Z pravidla zejména rozlišujeme metody jednofázové a metody dvoufázové.

4.1 Dvoufázové metody detekce

Jak již název napovídá, dvoufázové metody přistupují k detekci objektů ve dvou krocích. Ze vstupního snímku nejprve detekují potenciální objekty, které se v druhém kroku pokusí klasifikovat do kategorií, které neuronová síť zná. Výhodou je redukce velkého množství potenciálních objektů, protože klasifikace probíhá pouze v oblastech detekovaných v prvním kroku. Z toho také plyne, že okolí má na klasifikaci detekcí jen minimální vliv. [12]

Typickými zástupci tohoto způsobu detekce jsou sítě typu RCNN, SPPnet, Fast RCNN a Faster RCNN. Princip RCNN síti lze vidět na obrázku XX. [12]



Obrázek 4. Princip RCNN [12]

4.2 Jednofázové metody detekce

Na rozdíl od dvoufázových metod, kde detekce a klasifikace probíhá ve dvou krocích, u jednofázových metod se neuronová síť snaží zároveň určit pozici objektu ve snímku i jeho kategorii. Metody tohoto typu jsou podobně přesné jako metody dvoufázové, ale celá

klasifikace je výrazně rychlejší. Typickými zástupci těchto metod jsou OverFeat, RetinaNet, SSD a YOLO. [12]

4.2.1 YOLO

Název algoritmu YOLO je zkratkou věty „You only look once" v překladu „Hledáš jen jednou" nebo „Díváš se jen jednou". Jeho základním principem je rozdělení snímku na souřadný systém, kde každá buňka systému je sama zodpovědná za detekování objektů uvnitř sebe sama. [13]

4.2.1.1 Historie

Původním autorem je Joseph Redmon, který tento algoritmus poprvé zmínil ve své práci z roku 2015 nesoucí název „You Only Look Once: Unified, Real-Time Object Detection". V následujících letech autor vydal v letech 2016 a 2018 ještě další dvě práce, které se zabývaly zrychlením a zvýšení přesnosti tohoto algoritmu. Algoritmy zmíněné v pracích autor rozlišil pomocí verzování, kdy za název YOLO přidal písmeno v a číslo verze. [13]

V dubnu roku 2020, kdy autor originální práce již v dalším vývoji algoritmu nepokračoval, vyšel článek, jehož autory jsou Alexey Bochoknovskiy, Chien-Yao Wang, a Hong-Yuan Mark Liao, kteří představili další verzi algoritmu s číslem 4. Nedlouho poté, v květnu 2020, byla představena pátá verze algoritmu, se kterou přišel Glenn Jocher, jejíž implementace využívající framework Pytorch je volně dostupná na gitlabu. [13]

4.2.1.2 Princip

Principem této metody je rozdělení snímku na mřížky o velikosti k x k, kde každá z mřížek predikuje B ohraničení s určitou mírou jistoty. U těchto predikcí je poté schopna určit pravděpodobnost, s kterou se jedná o konkrétní objekt. Jednotlivé predikce se skládají z parametrů x, y, w, h a s, kde první čtyři udávají pozici a velikost predikovaného ohraničení s tím, že levý horní roh se nachází v bodě A[x, y] a šířka a výška ohraničení je dána parametry w a h. Poslední parametr s udává již zmíněnou míru jistoty, že se opravdu jedná o objekt. Ve výstupu se za těmito pěti parametry ještě nachází pole o velikosti shodné s počtem tříd, které se snažíme na snímcích detekovat, které udává míru pravděpodobnosti, s kterou se jedná o konkrétní třídu objektů. [12]

5 YOLO V5

Pro trénování modelu YOLO V5, který byl zvolen pro účely této práce, lze kromě lokálního přístupu využít i některé z online běhových prostředí. Jedním z nich je například Google Colab, ve kterém lze najít tutoriál přímo od autorů YOLO V5. Na něm si lze vyzkoušet detekci na COCO val 2017 datasetu, který se skládá z 80 různých tříd již označených objektů zahrnující celkový počet 5000 různých obrázků. Mezi nimi najdeme třídy jako člověk, kolo, auto, letadlo atd.

5.1 Google Colab

Jedná se o online běhové prostředí od firmy Google, které umožňuje psát a spouštět Python kód v prohlížeči. Uživatel nepotřebuje nic složitě nastavovat a velkou výhodou je bezplatný přístup ke grafickému procesoru. [14]

Je nabízen ve 3 tarifech. Bezplatný Colab, který trpí na časté odpojování prostředí a není tedy příliš použitelný, dále Colab Pro v ceně 9.99$ měsíčně, který je výpočetně rychlejší a nedochází tak často k odpojení jako u Colab verze, nebo Colab Pro + za 49.99$ měsíčně, jehož hlavní výhodou je, že jsou sešity schopny běžet i po zavření prohlížeče. [15]

5.2 Lokální přístup

Pro možnost trénování modelu lokálně je potřeba mít nainstalovanou distribuci Pythonu, Visual Studio Build Tools, a Pytorch. Lze využít i NVIDIA CUDA Toolkit a za jeho pomocí spouštět běh trénování na grafickém procesoru. Po zavedení všech zmíněných součástí se již postup v Google Colab a lokálním přístupu nijak neliší.

6 OPENCV

Knihovna OpenCV se primárně zaměřuje na zpracování obrazu, je multiplatformní a volně dostupná i pro komerční účely. Je vysoce optimalizovaná pro použití v real-time aplikacích. [16]

Jedná se o knihovnu napsanou v C/C++, která se těší oblibě zejména díky použitelnosti na platformách Windows, Linux, Android i MacOS, dále díky její optimalizaci, která klade důraz zejména na rychlost a výkon, i díky rozhraní, které umožňuje její využití v jazycích C++, Python, Java a MATLAB. Obsahuje více než 2500 optimalizovaných algoritmů zejména z oblasti strojového učení a zpracování obrazu. [17]

OpenCV je využívána v aplikacích po celém světě například společnostmi jako Google, Microsoft, Intel, IBM, Sony nebo Honda. Využívají ji i odborníci z univezit jako je MIT nebo Stanford. Její oblíbenost zdůrazňuje počet stažení, kterých je více než 14 milionů. [17]

6.1 Moduly

Celá knihovna je rozdělena do několika různých modulů, kde každý z nich pokrývá určitou oblast počítačového vidění.

core – základní funkcionalita, datové struktury a funkce, které využívají ostatní moduly

imgproc – zpracování obrazu, geometrické transformace obrazu, konverze mezi barevnými modely

imgcodecs – čtení a zápis snímků

videoio – rozhraní s video kodeky a práci s videi

highgui – rozhraní pro tvorbu a manipulaci s okny, zobrazení obrázků, tvorbu trackbarů, zpracování událostí myši a klávesnice

video – analýza videa, odečítání pozadí, odhadování pohybu, algoritmy pro detekci objektů

calib3d – kalibrace kamery a 3D rekonstrukce obrazu

feautures2d – detekce rysů a 2D deskriptorů pro kategorizaci obrazu

objdetect – detekce objektů a instancí předdefinovaných tříd

dnn – modul pro práci s hlubokými neuronovými sítěmi, obsahuje Api pro tvorbu vrstev a neuronových sítí nebo načtení modelů neuronových sítí z jiných frameworků

ml – sada tříd a metod z oblasti strojového učení využitelných pro klasifikaci nebo regresi

flann – shlukování a vyhledávání v multidimenzionálním prostoru

photo – modul s funkcemi pro oblast výpočetní fotografie

stitching – modul umožňující aplikaci techniky sešívání snímků

shape – práce s tvary, určování jejich vzdálenosti, shody

superres – sada tříd a metod vhodných pro vylepšení rozlišení

videosatb – stabilizace videa

viz – 3D vizualizér, umožňuje interakci se scénou a widgety

Kromě výše zmíněných modulů obsahuje knihovna také experimentální algoritmy a podporu testů či jiných souborů, které je možno sestavit společně se základní množinou modulů OpenCV. Lze je najít pod výrazem opencv_contrib a opencv_extra.

6.2 Vybrané třídy

6.2.1 Point

OpenCV obsahuje několik variant třídy Point, která je určena pro usnadnění práce s body. Lze využít varianty Point, Point2i, Point2f, Point2d, kde i značí datový typ int, f značí float a d značí double. Stejné ekvivalenty najdeme pro body ve 3D prostoru, kde je základní třída označena jako Point3. [18, 19]

6.2.2 Rect

Pro ukládání oblasti, která bude obepínat Kilobota bude použita třída Rect, která obsahuje body, ke kterým lze přistoupit metodami tl(), která značí levý horní bod čtverce a br(), který značí jeho pravý dolní roh. Dále lze zjistit jeho šířku a výšku pomocí width() a height(). [20]

6.2.3 Scalar

Třída Scalar se v OpenCV hojně používá k reprezentaci barev, většinou ve formátu RGBA. Jedná se o potomka třídy Vec, který reprezentuje vektor se čtyřmi prvky. [21]

6.2.4 Mat

Při zpracování obrazu se velmi často používá třída Mat. Jedná se o n-dimenzionální pole, které může být použito pro ukládání vektorů, matic, obrázků nebo například histogramů. Lze

s ní velice jednoduše zacházet a je použita v drtivé většině operací týkajících se zpracování obrazu. [22]

6.3 Vybrané funkce

6.3.1 threshold

V OpenCV, ale i obecně v oblasti zpracování obrazu, se velice často využívá tzv. prahování. To lze aplikovat na snímek (ve stupních šedi) a vyfiltrovat tak pouze části snímku, kde je hodnota pixelu pod určitým prahem. Jednoduše řečeno, pokud je hodnota pixelu menší než práh, tak ji změníme na 0, pokud je větší, tak ji změníme na maximum, které je také parametrem této funkce. OpenCV nabízí i funkci adaptiveThresholding, která se hodí zejména na snímky, které jsou v různých částech jinak nasvícené. [23]

6.3.2 findContours a drawContours

Funkce findContours slouží k nalezení obrysů nebo křivek se stejnou barvou nebo intenzitou. Princip jejího použití závisí na konkrétním problému, ale pro koncové aplikace je vhodné předem snímek, ve kterém se snažíme křivky najít, konvertovat na stupně šedi a rozostřit jej nebo upravit pomocí prahové hodnoty funkcí threshold.

Parametry funkce jsou třída Mat, která značí obrázek, ve kterém křivky hledáme, vektor jednotlivých křivek, které se skládají z vektoru nalezených bodů a poté dva parametry značící jakým způsobem se metoda pokusí křivky najít a jakým způsobem je bude aproximovat.

Nalezené křivky můžeme poté vykreslit pomocí funkce drawContours. Její parametry jsou třída Mat se snímkem, ve kterém budeme vykreslovat, vektor nalezených křivek, id křivky (pozice ve vektoru), třída Scalar s barvou pro vykreslení a šířka vykreslované čáry. [24]

V praktické části bude metoda využita pro detekci LED diody kilobota, podle kterého se určuje jeho úhel natočení.

6.3.3 cvtColor

Pomocí funkce ctvColor lze jednoduše transformovat snímky mezi různými druhy kódování barev jako jsou RGB, HSV, grayscale apod. [25]

6.3.4 line, rectangle, ellipse a circle

Pro potřeby kreslení jednoduchých objektů lze využít funkce line, rectangle, ellipse a circle. Společné parametry jsou třída Mat, se snímkem, na který kreslíme, třída Scalar s barvou pro vykreslení, šířka čáry a typ čáry, který značí způsob vykreslení. Ostatní parametry jsou vztažené ke konkrétnímu vykreslovanému objektu a udávají například střed a šířku kresleného kruhu. [26]

6.3.5 Bitové operace

Pomocí bitových operací můžeme velice jednoduše extrahovat část snímku a použít ji jako masku pro další operace. V praktické části bude využita při detekci LED pro odstranění barevných pixelů mimo oblast zájmu, ve které se LED může nacházet. [27]

6.4 Detekce Kilobotů pomocí OpenCV

Pro detekci objektů za pomocí OpenCV lze využít více různých a jinak složitých přístupů. V aktuálním řešení, se kterým pracují výzkumníci na UTB ve Zlíně je použita metoda detekce kružnic pomocí Houghovi kruhové transformace, která se pro tyto účely jeví jako velice efektivní, avšak má i svoje stinné stránky. Jednou z nich je počáteční nastavení metody, kde je potřeba nastavit 5 různých parametrů v závislosti na přiblížení/oddálení kamery. Pokud bychom během experimentu přiblížení kamery změnili, bude opět potřeba přenastavit parametry detekce. Snahou této práce je tedy navržení takové metody, která by byla více odolná vůči takovým změnám a zároveň více uživatelsky přívětivá. Níže je popis a krátké porovnání zvažovaných metod pro účely detekce.

6.4.1 Detekce kontur

Výše byla zmíněna metoda *findContours*, která je schopna detekovat ve snímku kontury. Tato metoda se jeví jako použitelná pouze v případě, pokud se na pracovní ploše nachází pouze Kiloboti a nic jiného. Ze zmíněného důvodu se pro detekci Kilobotů nejeví jako vhodný kandidát.

6.4.2 Spárování s šablonou

Velice jednoduchým způsobem můžeme vyhledávat objekty na snímcích použitím metody spárování se šablonou. Tato metoda funguje na principu 2D konvoluce, kde jednoduše

vezmeme hledaný objekt a poté jej postupně posouváme ve snímku a porovnáváme šablonu s jeho aktuálně vybranou částí. [28]

Z praktického hlediska by tato metoda měla být efektivní zejména při vyhledávání objektů v 2D animacích nebo tam, kde potřebujeme vyhledat nám již známý objekt. Pro praktické využití, do kterého vstupují prvky náhody (úhel objektu, nasvícení…) není tato metoda příliš vhodná.

6.4.3 Houghova kruhová transformace

Funkce houghCircles je schopna na snímcích ve stupních šedi najít kružnice. Má několik parametrů, ze kterých jsou pro praktickou aplikaci nejdůležitější param1, který určuje úroveň prahu pro Cannyho prahový detektor, param2, který určuje preciznost detekce – zde při nízkých hodnotách detekujeme více duplicitních kružnic. Dále minDist, který značí minimální vzdálenost mezi hledanými kružnicemi a poté minRadius a maxRadius, kterým lze ovlivnit velikost nalezených kružnic. [29]

Jak již bylo zmíněno, tato metoda je použita u stávajícího řešení detekce Kilobotů a pro tyto účely se jeví jako více než vhodná. Z tohoto důvodu zůstane ve výsledné knihovně zachována a díky tomu budou mít výzkumníci více možností detekce Kilobotů a mohou sami zvážit, kterou z metod detekce použijí na konkrétní experiment.

6.4.4 Kaskádové klasifikátory

Další zvažovanou metodou podporovanou OpenCV je metoda detekce pomocí kaskádových klasifikátorů. První zmínka o ní je z roku 2001 a to v práci nesoucí název „Rapid Object Detection using a Boosted Cascade of Simple Features“, jejíž autory jsou Paul Viola a Michael Jones. Jedná se o metodu z oblasti strojového učení, která je založena na trénování z mnoha pozitivních a negativních snímků. Pozitivními snímky jsou myšleny snímky obsahující objekt, který hledáme, negativní jej neobsahují. Tato metoda se snaží z pozitivních snímků rozpoznat společné rysy jako hrany a čáry ze kterých poté vytvoří výsledný klasifikátor. [30]

6.4.5 Hluboké neuronové sítě

Jak už bylo výše naznačeno, pro potřeby detekce Kilobotů byla zvolena neuronová síť. V dnešní době se v podobné problematice jedná prakticky o standard. Je závislá hlavně na kvalitě datasetu. Pokud ve snímcích, na kterých neuronovou síť učíme zahrneme všechny

možné rozdíly, co se týče například úhlů, z kterých jsou objekty nasnímány, záběry z odlišné vzdálenosti nebo za jiných světelných podmínek kamery, jsme schopni odbourat nepříznivé vlivy, které by nám mohly kazit výsledky u ostatních zmíněných metod.

6.5 Moduly tracking a legacy

Pro účely sledování objektů v obraze obsahuje knihovna OpenCV moduly tracking a legacy. V nich lze najít mnoho různých algoritmů určených k trackování objektů. Určitou nadstavbou těchto algoritmů je třída s názvem MultiTracker. Pomocí ní lze trackovat více objektů zároveň, kde na každý z nich můžeme aplikovat kterýkoliv z dostupných trackovacích algoritmů. [31, 32]

6.6 Modul DNN

Modul DNN zahrnuje většinu funkcí a tříd, které souvisí s hlubokými neuronovými sítěmi. Podporuje širokou škálu různých modelů neuronových sítí a je schopna s nimi pracovat například na mobilních zařízení s operačním systémem Android nebo ve webových prohlížečích. [33]

6.6.1 Třída Net

Jak název napovídá, třída Net slouží k vytváření a práci neuronovými sítěmi. OpenCV přistupuje k neuronovým sítím jako k orientovanému acyklickému grafu, kde jednotlivé vrcholy reprezentují vrstvy a hrany, které zmíněné vrcholy grafu propojují, znázorňují vztahy mezi vstupy a výstupy jednotlivých vrstev. Každá vrstva neuronové sítě má své vlastní id a unikátní jméno typu string. [34]

6.6.2 blobFromImage

V modulu DNN nalezneme funkci blobFromImage, která převádí snímek, ve kterém se snažíme Kiloboty detekovat, do formátu, kterému neuronová síť rozumí. Výstup této metody se poté zavede na vstup neuronové sítě pomocí metody *setInput* a zavoláním metody *forward* se neuronová síť pokusí objekty ve snímku detekovat. [35]

6.6.3 NMSBoxes

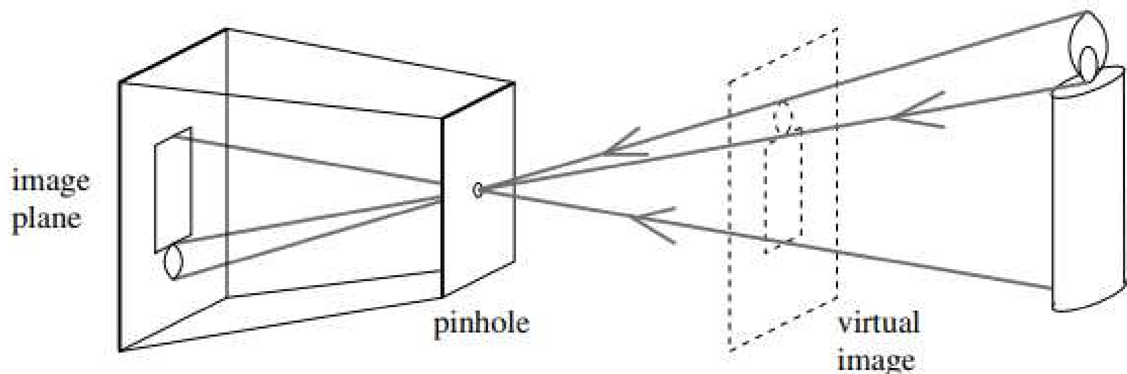
Výstupem třídy Net může být větší množství detekcí okolo jednoho objektu. Tyto duplicitní detekce je potřeba zredukovat na jednu jedinou. K tomu slouží technika zvaná NMS – Non Maximum Suppression. Jedná se o algoritmus, který je schopen z mnoha překrývajících se

entit vytvořit jednu jedinou a to tak, že na ně aplikuje techniku zvanou „Jaccardův koeficient podobnosti“ nebo také „průnik nad sjednocením“ zkráceně IoU jejíž výstupem je poměr průniku a sjednocení. Samotný algoritmus tedy nejprve vybere z vektoru všech detekcí tu s nejlepším skóre a tu poté porovnává s ostatními ze seznamu pomocí výše zmíněné techniky. [36]

7 URČENÍ VZDÁLENOSTI OD KAMERY

7.1.1 Model dírkové komory

Model dírkové komory vysvětluje princip, který říká, že pokud ve slabě osvětlené místnosti vezmeme krabici, kde v jedné z jejich stran uděláme malou díru, protilehlou stranu nahradíme průsvitnou deskou a namíříme stranu s dírou na nějaký světelný zdroj, například svíčku, na průhledné desce se objeví obrácený obraz světelného zdroje (svíčky). [34]



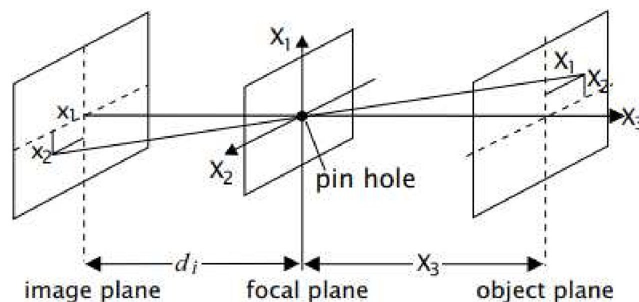
Obrázek 5. Model dírkové komory [37]

Velmi často se tato metoda používá tak, že díru uvažujeme jako bod [0,0,0] ve 3D prostoru a v tomto případě lze vypočítat polohu jakéhokoliv z promítnutých bodů pomocí rovnic:

$$x_1 = -\frac{d_i * X_1}{X3} \text{ [38]}$$

$$x_2 = -\frac{d_i * X_2}{X_3} \text{ [38]}$$

Kde proměnné psané velkými písmeny jsou hodnoty určující reálnou polohu a vzdálenost objektu od kamery a proměnné psané malými písmeny určují polohu a vzdálenost promítnutého obrazu. Níže je zobrazen zmíněný princip:



Obrázek 6. Dírková komora jako 3D prostor [38]

Důležitým faktem této metody, objevené už počátkem 15. století je to, že velikost objektu zobrazeného na průhledné desce je závislá na vzdálenosti objektu. [37]

Pokud tuto metodu aplikujeme na kamerový záznam, kdy bod [0,0,0] bude zaujímat čočka kamery, můžeme ze zmíněných rovnic vypočítat vzdálenost obrazu d_i, která pro konkrétní kameru bude vždy konstantní. Pro tento výpočet budeme potřebovat znát reálnou velikost objektu, vzdálenost kamery od objektu a velikost objektu v obraze v pixelech. Jakmile zjistíme hodnotu konstanty d_i, můžeme ji použít i na dalších záznamech pro stejný objekt umístěný v jiné vzdálenosti a z rovnice dopočítat reálnou vzdálenost označenou jako X_3.

7.1.2 Určení vzdálenosti od kamery

Princip lze ukázat na následujícím příkladu. Máme-li objekt, který má reálnou šířku 5 cm, natočený kamerou v laboratoři, kde reálná vzdálenost objektu od kamery je 86 cm a na kamerovém záznamu zaujímá 23 pixelů můžeme hodnotu d_i podle rovnice

$$x_2 = -\frac{d_i * X_2}{X_3}$$

Ze které vyjádříme proměnnou d_i jako

$$d_i = -\frac{x_2 * X_3}{X_2}$$

Po dosazení do rovnice zjistíme hodnotu

$$d_i = -\frac{23 * 86}{5} = -395{,}6$$

Pokud bychom poté chtěli dopočítat vzdálenost stejného objektu ve videu, kde se nacházel blíž/dál od kamery, stačí opět změřit šířku objektu v obraze a dosadit reálnou šířku objektu a proměnnou d_i. Pro velikost 32 pixelů by se například jednalo o rovnici:

$$X_3 = -\frac{d_i * X_2}{x_2}$$

$$X_3 = -\frac{-395{,}6 * 5}{32} = 61{,}8125\ cm$$

8 TRACKING

OpenCV obsahuje dva moduly, tracking a legacy, které umožňují trackování objektů. Využití těchto modulů se na první pohled jeví jako velice vhodné, hlavně díky tomu, že jejich použití je opravdu triviální. Je každopádně nutné, aby případný vybraný trackovací algoritmus byl dostatečně přesný a rychlý, a to i pro použití na řádově desítkách až stovkách objektů.

8.1 Trackery OpenCV

Pro potřeby práce byly testovány trackery MIL, Boosting, TLD, CSRT, KCF, MedianFlow a MOSSE. Jedná se o trackery, které potřebují znát pouze polohu objektu ve snímku a na tomto základě jsou poté schopny jej dohledat na snímku následujícím. K testování byla použita videa, která obsahovala okolo 100 Kilobotů. Test probíhal tak, že aplikace nejprve načetla první snímek videa, pomocí neuronové sítě v něm detekovala Kiloboty a poté se snažila Kiloboty trackovat na následujících snímcích.

Nejprve byly trackery testovány samostatně pro detekci prvního detekovaného Kilobota. Při této zkoušce se nejlépe projevily trackery Boosting a MedianFlow. Oba detekovaly Kilobota velice přesně, Boosting to zvádal s rychlostí 55 snímků za vteřinu, MedianFlow s rychlostí 70 snímků za vteřinu. Další úspěšný byl tracker MIL, který ale nebyl tak přesný jako dva dříve zmíněné. Navíc ani nedosahoval takové rychlosti jako oni, když trackoval Kilobota s rychlostí 17 snímků za vteřinu. Trackery CSRT, MOSSE a KCF běžely s rychlostí 80-110 snímků za vteřinu, Kilobota ale vůbec nebyly schopny trackovat. Poslední testovaný tracker TLD nebyl schopný Kilobota trackovat a byl schopný zpracovat pouze 5 snímků za vteřinu.

Další experiment probíhal za pomocí třídy MultiTracker, která je schopná pracovat s více trackery najednou. Pro všech 100 Kilobotů byl vytvořen vlastní tracker, s kterými byli následně Kiloboti trackováni. V tento moment došlo k drastickému snížení výkonu u všech tří v prvním testu úspěšných trackerů, kdy nejrychlejší z nich byl tracker MedianFlow, který byl schopný pracovat s rychlostí zpracování jednoho snímku za vteřinu. Tracker Boosting stíhal zpracovat jeden snímek za 2 vteřiny a trackeru MIL to trvalo 10 vteřin.

Protože žádný z trackerů neumožňuje v jedné instanci trackeru trackovat více objektů, bylo jasné, že pro účely trackování bude muset být navržen jiný algoritmus, který bude schopen rychlého běhu i při vysokém počtu trackovaných objektů.

8.2 Tracking pomocí nejmenší Eukleidovské vzdálenosti

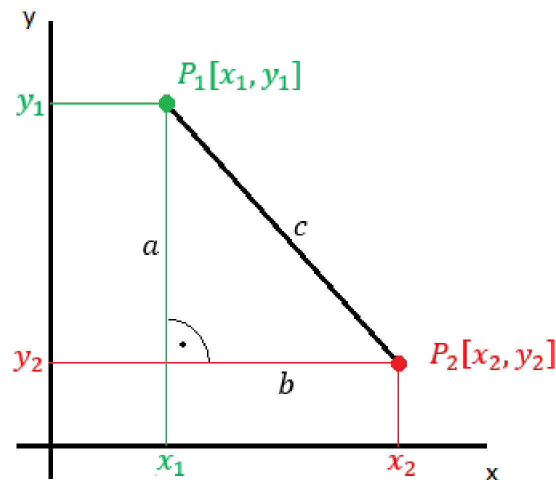
Pro potřeby trackování Kilobotů byl zvolen jednoduchý algoritmus, který pracuje s detekcemi na 2 po sobě jdoucích snímcích. Je vhodný zejména díky tomu, že plochu zaznamenáváme z pohledu shora a tím pádem nemůže nastat situace, kdy by se v kamerovém záznamu navzájem dva Kiloboti překryli. Pokud bychom tento algoritmus použili například v záznamu, který by plochu zabíral z pohledu z boku, stane se vysoce neefektivním, jelikož jeden z robotů by se mohl za druhého na chvíli skrýt a na následujících snímcích, kdy by se robot znovu objevil na scéně, by mohlo dojít k prohození jejich identifikátorů. [39]

8.2.1 Výpočet vzdálenosti

Jelikož v našem případě předpokládáme scénu snímanou statickou kamerou, můžeme zabranou plochu považovat za kartézskou soustavu, rovinu o 2 osách x a y. Poté lze za pomocí rovnice odvozené z Pythagorovi věty dopočítat vzdálenost mezi středy detekovaných objektů na dvou po sobě následujících snímcích podle rovnice $c^2 = a^2 + b^2$.

Pythagorova věta říká, že obsah čtverce nad přeponou pravoúhlého trojúhelníku je roven součtu obsahu čtverců nad jeho odvěsnami. Pokud obsah čtverce nad přeponou odmocníme, získáme délku přepony, která v kartézské soustavě odpovídá vzdálenosti mezi dvěma body. Tuto vzdálenost pak můžeme spočítat tak, že z původní rovnice popisující Pythagorovu větu vyjádříme c a dosadíme do ní výpočet délky odvěsen za pomocí dvou bodů kartézské soustavy $P_1[x_1, y_1]$ a $P_2[x_2, y_2]$. [40]

Na obrázku XX je zobrazen výše zmíněný princip výpočtu délky odvěsen pravoúhlého trojúhelníku, kde délky těchto stran určíme jako $a = |y_2 - y_1|$ a $b = |x_2 - x_1|$.



Obrázek 7. Určení vzdálenosti mezi dvěma body kartézské soustavy

Výše zmíněnou úpravou vyjádříme rovnici $c^2 = a^2 + b^2$ nejprve jako

$$c = \sqrt{a^2 + b^2}$$

A po dosazení výpočtu délek stran za *a* a *b* získáme

$$c = \sqrt{|x_2 - x_1|^2 + |y_2 - y_1|^2}$$

Dále díky tomu, že druhá mocnina záporného čísla je vždy kladná můžeme z výpočtu odstranit absolutní hodnoty, čímž získáme výsledný vzorec pro výpočet vzdálenosti mezi dvěma body.

$$c = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

8.2.2 Algoritmus

Základní myšlenka navrženého algoritmu je jasná. Na každém snímku je potřeba detekovat roboty a poté na následujícím snímku najít robota takového, který je k robotovi ze snímku předešlého nejblíže. Bere v úvahu velikost Kilobotů a rychlost s jakou se pohybují. Zohledňuje i fakt, že detektoru se nemusí povést detekovat všechny Kiloboty na všech snímcích. To se může stát například vlivem špatného nasvícení plochy. Je proto potřeba zavést určitá omezení, která algoritmu pomůžou roboty správně rozlišovat. Prvním z nich je počet po sobě jdoucích snímků, na kterých se může robot ztratit a druhým maximální vzdálenost, do které považujeme robota za jednoho a toho samého. Hodnota maximální vzdálenosti je dále v popisu označována jako práh nebo prahová hodnota. [39]

Jakmile tyto parametry vhodně nastavíme, je poté algoritmus schopen roboty spárovat i poté, co je detektor znovu po určité době detekuje. Zde ale musíme brát v úvahu, že by se mělo opravdu jednat o stejného Kilobota a parametry by měly být nastaveny na krátký časový úsek (počet snímků v řádu stovek) a vzdálenost. Jejich hodnota záleží na rychlosti pořízeného záznamu. Ve skutečnosti jsou totiž Kiloboti schopni na záběru pořízeném HD kamerou urazit mezi snímky vzdálenost odpovídající posunu ve snímku maximálně 1 px.

Pokud zmíněné parametry nastavíme například na hodnoty 180 a 5, říkáme, že při záznamu pořízeném s rychlostí 30 snímků za vteřinu, pokud se robot ztratí, tak ho po 6 vteřin udržíme v paměti. Pokud se ho povede do uplynutí 180 snímků detekovat a posun bude maximálně 5 pixelů, budeme ho stále považovat za toho samého. V opačném případě mu vygenerujeme nový identifikátor a prezentujeme ho jako nový objekt.

Samotný algoritmus se skládá z několika částí, které by se daly shrnout následujícími body:

Krok 1: Výpočet středů detekovaných objektů

Do algoritmu vstupují data, která popisují polohu a velikost detekovaného objektu ve snímku. Z těchto dat je nejprve potřeba určit střed detekovaného objektu.

Krok 2: Nalezení nejkratší vzdálenosti k objektům z předchozího snímku

Pro každý detekovaný objekt najdeme nejméně vzdálený objekt na předchozím snímku, který nebyl dosud spárovaný. Učiníme tak pomocí výpočtu odvozeného v předchozí kapitole.

Krok 3: Spárování objektů

Poté, co nalezneme objekt, který je na předešlém snímku nejblíže, určíme, jestli se jedná o již trackovaný objekt nebo o nově detekovaný objekt. Pokud je hodnota vzdálenosti menší než maximální možná vzdálenost posunu mezi dvěma snímky, tak jej spárujeme s objektem aktuálně detekovaným a znemožníme jeho další přiřazení. Pokud je hodnota větší nejedná se o stejný objekt.

Krok 4: Přiřazení identifikátoru nově detekovaným objektům a uchování ztracených objektů

Pokud jsme detekovali na aktuálním snímku více objektů než na snímku předchozím, vygenerujeme všem nespárovaným objektům nový identifikátor. V případě, kdy jsme detekovali objektů méně, tak ztracený objekt nadále po určitou dobu uchováváme v paměti.

Krok 5: Odstranění nedetekovaných objektů z paměti

Pokud jsme nebyli schopni ztracený objekt detekovat na několika po sobě jdoucích snímcích, teprve tehdy jej z paměti vymažeme. Pokud by následně došlo k opětovné detekci, bude tento objekt považován za nový a bude mu vygenerován nový identifikátor.

Jedná se o velice zjednodušený popis navrženého algoritmu. Jeho samotná implementace je popsána v praktické části.

9 KALIBRACE OBRAZU

Kalibrace obrazu slouží k odstranění zkreslení, které může být způsobeno více různými vlivy. OpenCV bere v potaz zejména radiální a tangenciální zkreslení. Díky kalibraci lze také určit vztah mezi reálnými jednotkami, například centimetry a jednotkami kamery – pixely. Radiální zkreslení se projevuje ve formě soudkového efektu nebo efektu rybího oka. Tangenciální je způsobeno tím, že čočka, která snímá obraz není dokonale rovnoběžná se zobrazovací rovinou. [41]

Vztah mezi body ve zkresleném a nezkresleném snímku způsobeným vlivem radiálního zkreslení lze dopočítat podle rovnic:

$$x_{distorted} = x(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

$$y_{distorted} = y(1 + k_1r^2 + k_2r^4 + k_3r^6)$$

Podobně lze přepočítat i body ve snímcích ovlivněných tangenciálním zkreslením, a to podle rovnic:

$$x_{distorted} = x + [2p_1xy + p_2(r^2 + 2x^2]$$

$$y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2xy]$$

S chybějícími parametry OpenCV pracuje jako s maticí o pěti sloupcích zvanou koeficienty zkreslení:

$$distortion\ coeficients = (k_1\ k_2\ p_1\ p_2\ k_3)$$

Abychom byli schopni odstranit zkreslení, je potřeba znát tzv. matici kamery. Ta vypadá následovně:

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Neznámé parametry f_x a f_y jsou ohniskové vzdálenosti kamery a c_x a c_y což jsou optické středy vyjádřené v pixelech. Každá matice kamery je specifická ke konkrétní kameře a jakmile ji jednou spočítáme, lze jí použít na dalších snímcích zaznamenaných touto kamerou.

Kalibrace kamery je vlastně proces, při kterém se snažíme najít všechny chybějící parametry a koeficienty zkreslení a matice kamery jsou jejím výstupem. OpenCV aktuálně podporuje kalibraci pomocí symetrického či asymetrického kruhového vzoru a také pomocí

šachovnice. V podstatě stačí pomocí kamery zachytit několik snímků obsahujících tyto vzory a nechat OpenCV dopočítat potřebné matice. Pro kvalitní výsledky se doporučuje zaznamenat minimálně 10 snímků se zmíněnými vzory. [41]

II. PRAKTICKÁ ČÁST

10 SBĚR A PREPROCESSING DAT

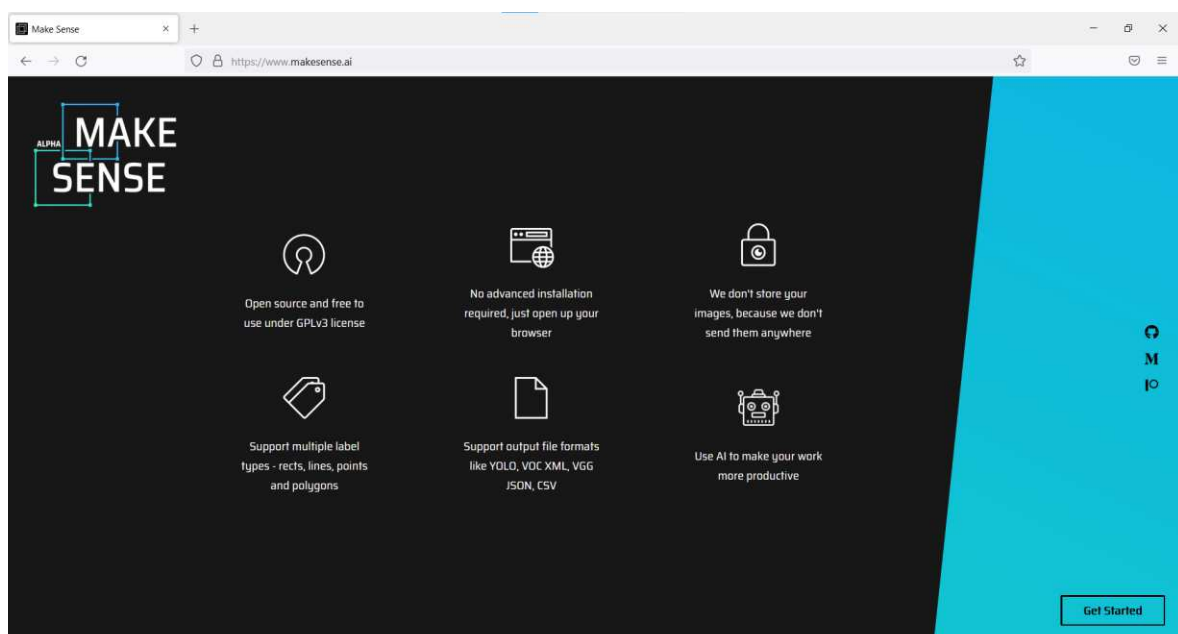
Pro natrénování neuronové sítě je potřeba velké množství dat. Ta byla, ve formě videí, či sérii obrázků z experimentů, získána v laboratoři A.I.Lab na UTB, z veřejně dostupných zdrojů, příp. s povolením převzata v anonymizované podobě od zahraničních robotických laboratoří.

10.1 Základní úprava

Jelikož většina dat byla ve formě videí, bylo potřeba z nich nějakým způsobem vytáhnout jednotlivé snímky. Pro tyto potřeby byl použit online nástroj [42], který z videa vyrobil jpg soubory. Většina změn ve videích byla relativně pomalá, tudíž z konvertovaných dat byla podle uvážení autora velká část odstraněna a zbyly pouze snímky takové, na kterých byla vidět alespoň nějaká změna. Ze zbylých snímků byly následně odstraněny ještě ty, na kterých byli Kiloboti rozmazáni nebo špatně nasvíceni.

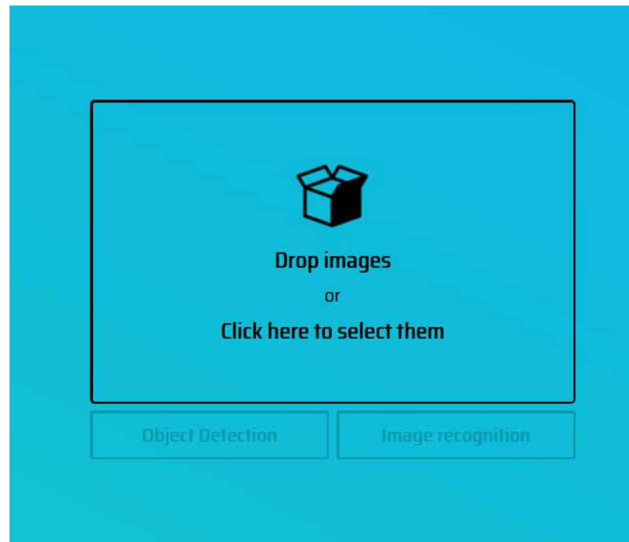
10.2 Značení

Po vytřídění dat bylo potřeba projít a označit na nich Kiloboty. K tomu lze využít více nástrojů. Osobně jsem našel nástroje DarkMark [43] a MakeSense [44]. Zvolil jsem druhý zmíněný, a to z jednoduchého důvodu, jelikož není potřeba cokoliv instalovat a lze jej použít přímo v prohlížeči. Níže je zobrazena úvodní stránka, na které stačí v pravém dolním rohu zvolit možnost *Get Started.*



Obrázek 8. Úvodní stránka MakeSense

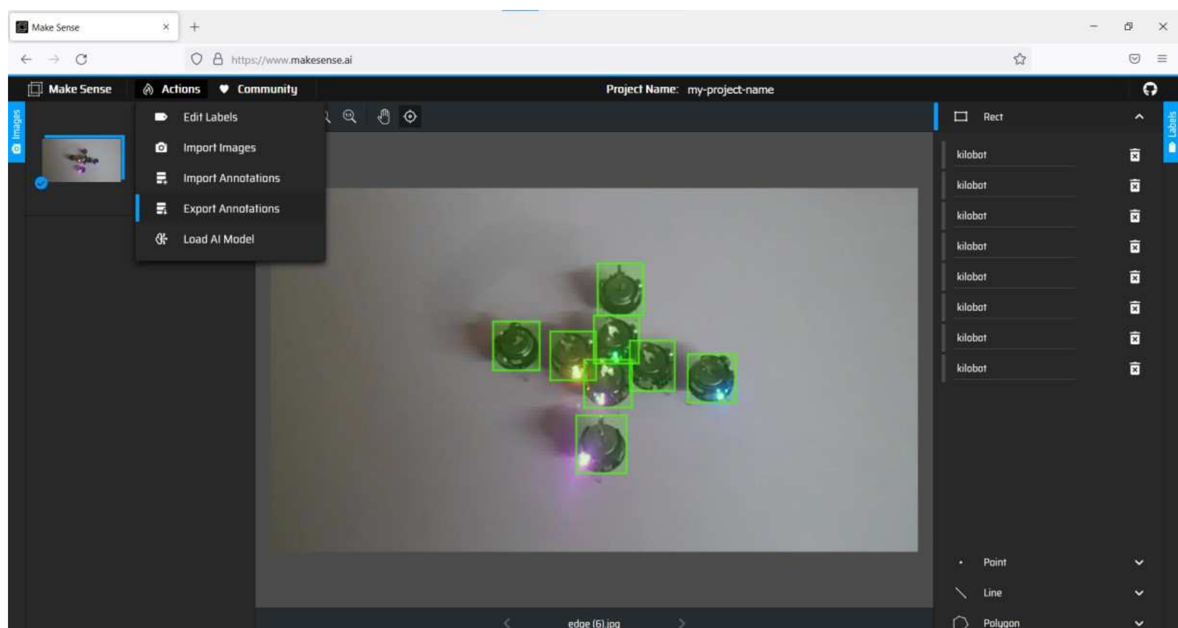
Poté se zobrazí stránka, kde můžeme importovat data. Jakmile je nahrajeme, tak zvolíme možnost *Object Detection.*



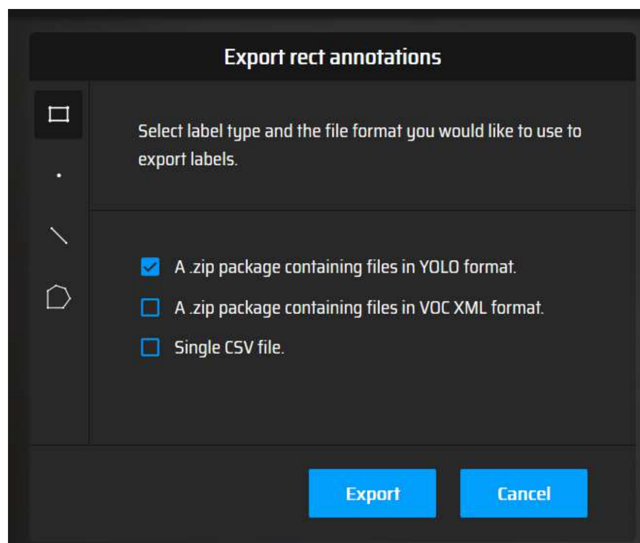
Obrázek 9. Import dat určených k označení

Dostaneme se na další stránku, kde přidáme název označovaných tříd v obrázcích. Pokud označujeme pouze kiloboty, nemusíme název přidávat a ve výsledných souborech se nám exportují štítky s id 0.

Postupně tedy označíme na všech snímcích všechny kiloboty a klikneme na *Actions* v levém horním rohu, tam zvolíme *Export Annotations* a vybereme možnost *A .zip package containing files in YOLO format* a klikneme na *Export.*



Obrázek 10. Označení Kilobotů



Obrázek 11. Export anotací

10.3 Struktura

Jakmile máme označené kiloboty na obrázcích a vyexportované soubory s jejich štítky, je potřeba je vložit do složky s určitou strukturou, jejíž podobu budeme definovat v další kapitole. V tutoriálu přímo od autorů YOLO defaultně pracuje s rozdělením na *images* a *labels* a v nich rozlišuje podsložky *train* pro trénování a *val* pro validaci. Tato struktura byla pro účely trénování dodržena a je zobrazena níže.

```
data/
├── images/
│   ├── train/
│   │   ├── img1.jpg
│   │   └── img2.jpg
│   └── val/
│       └── img3.jpg
└── labels/
    ├── train/
    │   ├── img1.txt
    │   └── img2.txt
    └── val/
        └── img3.txt
```

11 TRÉNOVÁNÍ MODELU YOLOV5

Jak již bylo zmíněno, samotné natrénování modelu lze provádět lokálně na našem zařízení nebo online například za pomocí Google Colab, ze kterého je i následující postup. Zdrojové kódy Yolo v5 jsou volně dostupné na platformě github. [45]

11.1 Zavedení YOLO v5

Nejdříve je potřeba spustit blok, který vyklonuje z gitu soubory yolov5, nainstaluje PyTorch a ověří platnost GPU.

```
!git clone https://github.com/ultralytics/yolov5  # clone
%cd yolov5
%pip install -qr requirements.txt  # install

import torch
from yolov5 import utils
display = utils.notebook_init()  # checks
```

11.2 Příprava souborů

Poté nahrajeme připravená data v .zip formátu do adresáře na stejnou úroveň, kde jsou soubory yolov5 a rozbalíme je pomocí příkazu níže.

```
!unzip -q ../data.zip -d ../
```

Dále do složky yolov5/data nahrajeme soubor s názvem kilobot.yaml, ve kterém definujeme počet tříd, jejich názvy a cestu k souborům pro trénování a validaci. Zde se vlastně jedná o popis struktury zmiňovaný v kapitole 6.3.

```
train: ../train_data/images/train  # train images (relative to 'path')

val: ../train_data/images/val  # val images (relative to 'path')


# Classes

nc:  1 # number of classes

names: ['kilobot']  # class names
```

11.3 Trénování modelu

Pro natrénování modelu použijeme příkaz train.py ze složky yolov5.

```
!python train.py --img 640 --batch 16 --epochs 180 --data ki-
lobot.yaml --weights yolov5s.pt --cache
```

--img – určuje velikost vstupních dat pro neuronovou síť

--batch – velikost dávky

--epochs – určuje počet etap pro trénování

--data – název souboru, ve kterém jsou definovány cesty k obrázkům a štítkům pro trénování a validaci a také počet tříd a jejich názvy

--weigths – soubor yolov5 s definovanými váhami

--cache – parametr pro urychlení výpočtu pomocí využití cache paměti

[46]

```
     Epoch   gpu_mem       box       obj       cls    labels  img_size
   174/179     3.34G   0.02479   0.05404         0      1439       640: 100% 10/10 [00:09<00:00,  1.07it/s]
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 1/1 [00:00<00:00,  2.81it/s]
                 all         41        333      0.976      0.994      0.984      0.832

     Epoch   gpu_mem       box       obj       cls    labels  img_size
   175/179     3.34G   0.02512    0.0523         0       884       640: 100% 10/10 [00:09<00:00,  1.07it/s]
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 1/1 [00:00<00:00,  2.86it/s]
                 all         41        333      0.976      0.994      0.983      0.834

     Epoch   gpu_mem       box       obj       cls    labels  img_size
   176/179     3.34G   0.02386   0.04981         0      1311       640: 100% 10/10 [00:09<00:00,  1.08it/s]
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 1/1 [00:00<00:00,  2.80it/s]
                 all         41        333      0.979      0.993      0.983      0.831

     Epoch   gpu_mem       box       obj       cls    labels  img_size
   177/179     3.34G   0.02397   0.05214         0       995       640: 100% 10/10 [00:09<00:00,  1.07it/s]
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 1/1 [00:00<00:00,  2.81it/s]
                 all         41        333      0.976      0.997      0.982       0.83

     Epoch   gpu_mem       box       obj       cls    labels  img_size
   178/179     3.34G   0.02416   0.05159         0      1547       640: 100% 10/10 [00:09<00:00,  1.08it/s]
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 1/1 [00:00<00:00,  2.86it/s]
                 all         41        333      0.976      0.994      0.982      0.831

     Epoch   gpu_mem       box       obj       cls    labels  img_size
   179/179     3.34G   0.02432   0.05246         0      1141       640: 100% 10/10 [00:09<00:00,  1.08it/s]
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 1/1 [00:00<00:00,  2.81it/s]
                 all         41        333      0.976      0.994      0.984      0.831

180 epochs completed in 0.502 hours.
Optimizer stripped from runs/train/exp/weights/last.pt, 3.8MB
Optimizer stripped from runs/train/exp/weights/best.pt, 3.8MB

Validating runs/train/exp/weights/best.pt...
Fusing layers...
Model Summary: 213 layers, 1760518 parameters, 0 gradients, 4.2 GFLOPs
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 1/1 [00:00<00:00,  1.66it/s]
                 all         41        333      0.976      0.997      0.984      0.842
Results saved to runs/train/exp
```

Obrázek 12. Trénování modelu v Google Colab

11.4 Export modelu do formátu onnx

Pro potřeby načtení modelu do třídy OpenCV bylo potřeba vyexportovat natrénovaný model do formátu, který OpenCV zná. Dostupné jsou formáty Caffe, TensorFlow, Torch, Darknet, TLD a ONNX. K poslednímu zmíněnému formátu byl autorem yolov5 vytvořen skript, který jej do něj exportuje. Pro tyto potřeby stačí spustit v příkazové řádce s adresářem yolov5 příkaz:

```
!python export.py --weights runs/train/exp20/weights/best.pt
--include onnx –simplify
```

Kde za parametrem –*weights* uvedeme cestu k modelu, který chceme exportovat.

11.5 Použitá data

Model byl trénován na snímcích, které dohromady obsahovaly přibližně 9400 Kilobotů. Většina použitých snímků byla v HD kvalitě a průměr Kilobota se pohyboval v rozmezí 20 až 60 pixelů. Výjimkou byly snímky ze 2 videí, na kterých měli Kiloboti velikost 80 a 120 pixelů. Šestina všech použitých snímků byla použita pro validaci a zbytek pro trénování.

12 KNIHOVNA KILOLIB

Nyní, když máme již natrénovaný, ověřený a exportovaný model, můžeme se vrhnout na popis navrženého řešení.

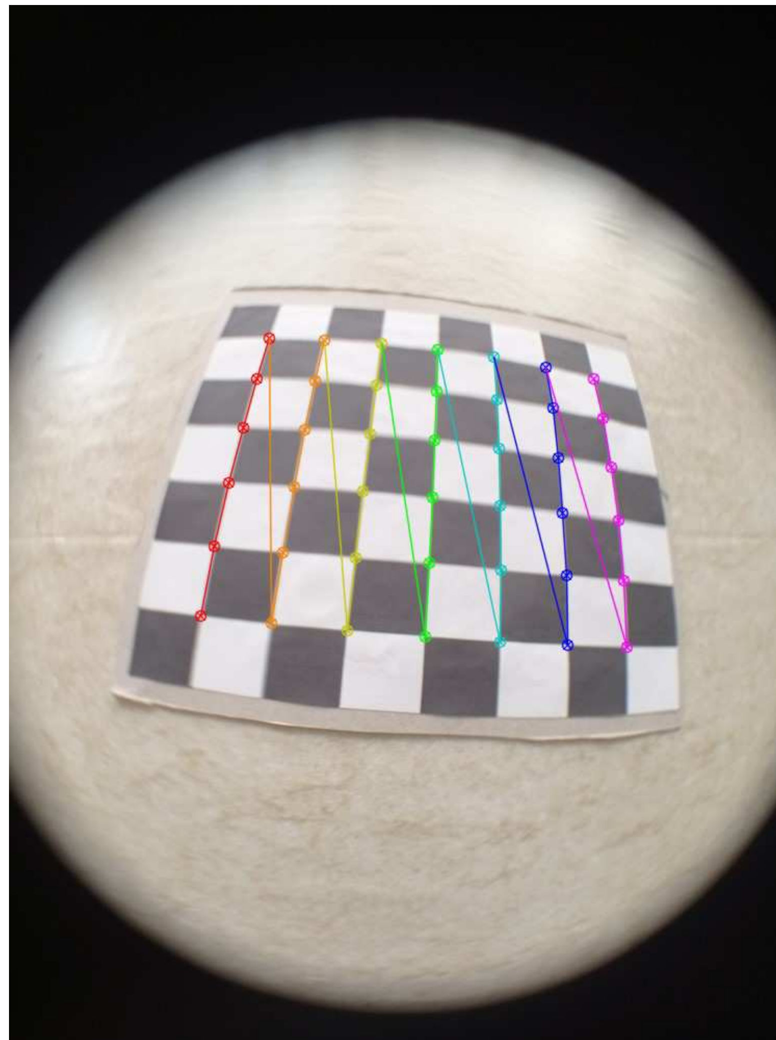
12.1 Kalibrace kamery

Programů na kalibraci pomocí OpenCV již byla napsána spousta. Nebylo proto potřeba vymýšlet nic nového a pro účely práce byl pouze upraven jeden z nich, který využívá šachovnici. [47]

12.1.1 Nalezení bodů potřebných pro výpočet

Pro potřeby kalibrace byla navržena metoda, jejíž vstupními parametry jsou cesta do složky, ve které se nachází snímky kalibračního objektu (šachovnice), přípona obrazového formátu, ve kterém jsou snímky uloženy (png, jpg …), šířka hledané šachovnice, výška hledané šachovnice, vektory, do kterých se mají uložit body pro výpočet, a rozlišení snímků použitých pro kalibraci.

Zmíněná šířka a výška šachovnice znázorňuje počet hledaných bodů. Ten je vždy o 1 menší než počet čtverců na jedné straně šachovnice. Pro šachovnici 8x8 bude OpenCV hledat 7 bodů na každé straně. Názorně to lze vidět na obrázku níže, kde je použita šachovnice s rozměry 8x7. Nalezených bodů je 7 na delší straně šachovnice a 6 na kratší.



Obrázek 13. Nalezené body šachovnice

```
bool CalculatePoints(string folderPath, string suffix, int
cbWidth, int cbHeight,
    vector<vector<Point3f>>& objPoints, vector<vector<Point2f>>&
imgPoints, Size& size) {
```

Nejprve je potřeba vytvořit body reálného prostoru *X*, *Y* a *Z*. Jelikož šachovnice v tomto případě ležela bez jakéhokoliv posunu na stole a měnila se pouze poloha kamery, ze které byla šachovnice snímána, můžeme říct, že hodnota *Z* byla vždy rovna 0. Za hledané hodnoty *X* a *Y* lze poté dosazovat body [0, 0], [1, 0], [2, 0], čímž získáme výsledky zmenšené v poměru k délce hrany jednoho čtverce šachovnice. Pokud délku této hrany známe, můžeme za body dosazovat například při hraně 30 mm body [0, 0], [30, 0], [60, 0] a výsledky ke kterým se dobereme budou odpovídat reálným jednotkám. [48]

```
    vector<Point3f> objp;
    for (int i = 0; i < cbHeight; i++)
    {
        for (int j = 0; j < cbWidth; j++)
            objp.push_back(Point3f(j, i, 0));
    }
```

Poté pomocí funkce glob projdeme zadaný adresář a do vektoru *images* zapíšeme cesty ke všem souborům v daném adresáři.

```
    vector<String> images;

    stringstream stream;
    stream << "./" << folderPath << "*" << suffix;

    string path = stream.str();

    glob(path, images);
```

Dále postupně projdeme všechny snímky z adresáře s danou příponou a pokusíme se v nich najít šachovnici pomocí funkce *findChessboardCorners*.

```
    Mat frame, gray;

    vector<Point2f> corner_pts;
    bool success;

    for (int i = 0; i < images.size(); i++)
    {
        std::cout << "processing img: " << i + 1 << std::endl;

        frame = imread(images[i]);
        cvtColor(frame, gray, COLOR_BGR2GRAY);

        success = findChessboardCorners(gray, Size(cbWidth,
cbHeight), corner_pts);
```

Pokud zmíněnou funkcí nalezneme na snímku šachovnici o rozměrech daných parametry funkce, pomocí funkce *cornerSubPix* zpřesníme souřadnice nalezených rohů a vykreslíme je funkcí *drawChessboardCorners*.

```
        if (success)
        {
            std::cout << "accepted" << std::endl;
```

```
            TermCriteria criteria(TermCriteria::EPS | TermCrite-
ria::MAX_ITER, 30, 0.001);

            cornerSubPix(gray, corner_pts, Size(11, 11), Size(-1,
-1), criteria);

            drawChessboardCorners(frame, Size(cbWidth, cbHeight),
corner_pts, success);

            objPoints.push_back(objp);
            imgPoints.push_back(corner_pts);
        }
    }
```

Nejprve zapíšeme rozlišení snímků, na kterých byla šachovnice hledána a pokud se v adresáři nachází alespoň dva snímky, na kterých se povedlo šachovnici úspěšně najít vrátíme hodnotu *true*. V opačném případě funkce vrací *false*.

```
    if (!gray.empty())
        size = gray.size();

    if (objPoints.size() > 2)
        return true;

    return false;
}
```

12.1.2 Uložení a načtení vypočtených matic

Jakmile je matice vypočítána, lze ji jednoduše uložit ve formátu xml nebo yaml pomocí následující funkce. Ta využívá třídu *FileStorage*, která je velice jednoduše schopna zapsat obě spočtené matice do souboru.

```
void StoreCalibration(string fileName, Mat cameraMatrix, Mat dist-
Coeffs) {
    FileStorage fs(fileName, FileStorage::WRITE);

    fs << "Camera_matrix" << cameraMatrix;
    fs << "Distortion_coeficients" << distCoeffs;
}
```

Obdobným způsobem lze uložené matice pomocí stejné třídy opět načíst a poté pomocí funkce *undistort* zkreslený snímek napravit.

```
void LoadCalibration(string fileName, Mat& cameraMatrix, Mat&
distCoeffs) {
```

```
    FileStorage fs(fileName, FileStorage::READ);

    fs["Camera_matrix"] >> cameraMatrix;
    fs["Distortion_coeficients"] >> distCoeffs;
}
```

12.2 Struktura Kilobot

Pro účely zaznamenání dat byla navržena následující struktura:

```
struct Kilobot
{
    float confidence;
    Rect box;
    int id;
    int undetected;
    std::vector<Point> led;
    std::vector<Point> trajectory;
    Scalar color;
};
```

Jednotlivé položky určují:

confidence – míra jistoty neuronové sítě, že se jedná o Kilobota

box – část snímku, ve které se Kilobot nachází

id – přidělené ID

undetected – počet snímků, kolikrát již nebyl detekován

led – vektor polohy detekované LED diody

trajectory – vektor středů detekovaných oblastí Kilobota (z položky box)

color – položka pro určení barvy pro vykreslení

12.3 Měření vzdálenosti

Pro účely určení přibližné vzdálenosti Kilobota od kamery byla použita metoda využívající principu dírkové komory a podobnosti trojúhelníků.

12.3.1 Naměřená data

Při měření v laboratoři bylo potřeba ověřit, že velikost Kilobota je skutečně 3,3 cm jak uvádějí zdroje. Poté proběhlo měření ve třech různých vzdálenostech od kamery, kde byl

Kilobot umístěn přímo pod ní. Ze všech tří měření vzniklo video, z kterého byla zjištěna velikost Kilobotů v pixelech. V tabulce X lze vidět potřebná data:

Číslo měření	Vzdálenost od kamery [cm]	Velikost ve snímku [px]
1	116	35
2	102,2	40
3	83,8	50

Tabulka 3. Měření vzdáleností od kamery

Vzdálenost kamery lze pomocí trojúhelníkové podobnosti získat velice jednoduchým výpočtem podle rovnice:

$$d_i = -\frac{x_2 * X_3}{X_2}$$

Kde x_2 značí velikost v pixelech, X_3 vzdálenost od kamery a X_2 reálnou šířku objektu. Pro naměřená data lze tedy spočítat hodnotu vzdálenosti d_i jako:

$$d_{i1} = -\frac{35*116}{3,3} = -1230,30$$

$$d_{i2} = -\frac{40*102,2}{3,3} = -1238,79$$

$$d_{i3} = -\frac{50*83,8}{3,3} = -1269,7$$

Jak lze vidět, pro měření vyšly hodnoty v rozmezí -1269,7 až -1230,3, což je způsobeno chybou měření.

Výpočet, který obsahuje knihovna ale hledá reálnou vzdálenost, tudíž proměnnou X_3. Po jejím vyjádření tudíž dostaneme:

$$X_3 = -\frac{d_i * X_2}{x_2}$$

12.3.2 Funkce measureDistance

```
static const double di = -1240;
static const double kilobotSize = 3.3;
```

Nejprve bylo potřeba definovat konstanty zjištěné z měření. Pro účely práce jsem používal lehce zaokrouhlenou hodnotu prostřední naměřené vzdálenosti na -1240.

Na prvním řádku funkce je samotný výpočet, který vznikl dosazením konstant a průměru Kilobota v pixelech do rovnice. U některých méně přesných detekcí se o pár pixelů lišila detekovaná šířka a výška rámečku, ve kterém se nachází Kilobot. Proto pro účely výpočtu používáme průměr obou hodnot, což by mělo v těchto případech výpočet zpřesnit.

```
double measureDistance(Rect box, Mat frame, bool display, Scalar
textColor)
    {
        double distance = - di * kilobotSize / ((box.width +
box.height) / 2);
```

Pokud uživatel funkci zavolá s proměnnou display nastavenou na true, tak se vykreslí naměřená hodnota zaokrouhlená na 2 desetinná místa. O to se stará podmínka níže.

```
        if (display) {
            std::ostringstream streamObj;
            streamObj << std::fixed;
            streamObj << std::setprecision(2);
            streamObj << distance;
            std::string strObj = streamObj.str();

            cv::putText(frame, strObj, box.tl(),
cv::FONT_HERSHEY_SIMPLEX, 0.4, textColor, 1);
        }

        return distance;
    }
```

12.3.3 centerKilobotDetection

Pro potřeby měření vzdáleností byla navržena funkce centerKilobotDetection, která načte snímek do paměti a poté se v jeho středu snaží najít Kilobota. Velikost hledané oblasti je dána v procentech a lze ji ovlivnit parametrem *percentage*. Pro potřeby samotného nalezení Kilobota je použita funkce *findContours*, kterou můžeme ovlivnit parametrem *thresh*. Poslední parametr, pokud je nastaven na *true*, tak u detekovaného Kilobota vykreslí změřenou vzdálenost od kamery v centimetrech. Výsledný snímek uloží pomocí reference do proměnné *frame*.

```
void centerKilobotDetection(Mat& frame, string imgPath, int per-
centage, int thresh, bool measureDist = true) {
```

Nejdříve je potřeba načíst snímek, který bude funkce dále upravovat. Dále je vytvořen další snímek o stejné velikosti jako originál a je plně vykreslen bílou barvou. Poté je

provedena extrakce části originálního snímku tak, že funkce vyjme část zadanou v procentech a umístí ji do bílého snímku, který dále upravuje.

```
Mat original = imread(imgPath);

frame = Mat(original.rows, original.cols, original.type(),
Scalar(255, 255, 255));

int w = original.size().width / 100 * percentage;
int h = original.size().height / 100 * percentage;

int tx = original.size().width / 2 - w / 2;
int ty = original.size().height / 2 - h / 2;

Rect r = Rect(tx, ty, w, h);

original(r).copyTo(frame(r));
```

Dále je provedeno nalezení kontur pomocí funkce findContours s požadovanou úrovní thresholdu.

```
Mat gray;
cvtColor(frame, gray, COLOR_BGR2GRAY);

Mat th;
threshold(gray, th, thresh, 255, THRESH_BINARY_INV);

vector<vector<Point>> contours;
findContours(th, contours, RETR_TREE, CHAIN_APPROX_SIMPLE);
```

V nalezených konturách najdeme tu, která pokrývá největší oblast a vykreslíme ji v upravovaném snímku. Pokud uživatel stál o změření vzdálenosti, zavolá se funkce *measureDistance* a vykreslí se spočtená vzdálenost.

```
int biggestSize = -1;
Rect biggestBox;

for (int c = 0; c < contours.size(); c++) {
	Rect cBox = boundingRect(contours[c]);

	double area = (double)cBox.width * (double)cBox.height;

	if (area > biggestSize) {
		biggestSize = area;
		biggestBox = cBox;
	}
}
```

```
      if (biggestSize != -1) {
            rectangle(frame, biggestBox, Scalar(0, 255, 0));

            if (measureDist)
                  measureDistance(biggestBox, frame, true,
Scalar(255, 0, 255));
      }
}
```

12.4 Nalezení LED

Navržený algoritmus určený k nalezení LED u detekovaného Kilobota kombinuje několik funkcí OpenCV popsaných v teoretické části. V principu se jedná vyhledání největší lesklé oblasti na obvodu Kilobota.

Algoritmus je navržen v knihovně jako funkce, kterou najdeme v souboru kilobot_functions.h a jejími parametry jsou celý snímek, ve kterém byl Kilobot detekován, oblast, kterou ve snímku zabírá, prahová hodnota pro vyfiltrování lesklých částí a minimální velikost oblasti, kterou budeme považovat za LED diodu.

```
Point findLED(Mat frame, Rect box, double thresh, double minArea)
```

Nejprve nastavíme hodnoty *x* a *y*, které udávají polohu detekované diody na -1. Pokud diodu nenalezneme, tak funkce vrátí třídu Point s těmito parametry a uživateli bude tímto indikováno, že dioda nebyla nalezena. Dále vybereme pouze část snímku, na které se kilobot nachází a uchováme si ji v proměnné typu *Mat*.

```
int x = -1;
int y = -1;
Mat kilobot = frame(box);
```

12.4.1 Odfiltrování vnější části mimo Kilobota

V prvním kroku je potřeba odfiltrovat vše vnějšího okolo kilobota. Pokud při pohybu Kilobotů dochází k jejich shlukování, tak se v těchto částech snímku mohou nacházet LED diody dalších Kilobotů.

Postup je takový, že vytvoříme masku ve tvaru kruhu, o stejném poloměru jako je poloměr Kilobota, kterou vyplníme bílou barvou a poté tuto masku aplikujeme na snímek s Kilobotem. Ve výsledném snímku tak zůstane pouze oblast, nad kterou se nacházela bílá barva a zbylé části jsou překryty černou barvou. Toto překrytí kopíruje kruhový tvar Kilobota.

```
Mat mask = Mat::zeros(kilobot.rows, kilobot.cols, kilobot.type());
double radius = kilobot.rows / 2;

if (kilobot.rows > kilobot.cols)
    radius = kilobot.cols / 2;

circle(mask, Point(kilobot.rows / 2, kilobot.cols / 2), radius,
Scalar(255, 255, 255), FILLED);

Mat res;
bitwise_and(kilobot, mask, res);
```

Obrázek 14. Odfiltrování vnější části Kilobota

12.4.2 Odfiltrování vnitřní části Kilobota

Vnitřní oblast Kilobota je tvořena zejména baterií, která při špatném nasvícení odráží světlo. Abychom tuto část v detekci LED nebrali v úvahu, je jí potřeba také odfiltrovat. To lze udělat jednoduchým překrytím pomocí černého kruhu. Pro tento účel jsem metodou pokus – omyl určil poloměr kruhu, který zaujímá třetinu celkové šířky Kilobota.

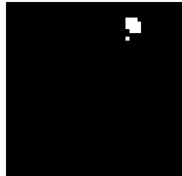
Obrázek 15. Odfiltrování vnější části Kilobota

12.4.3 Aplikace prahové hodnoty

Nyní, když jsme již odstranili části snímku, ve kterých by se LED dioda neměla nacházet, můžeme přejít k samotnému vyhledání této diody. To je řešeno pomocí aplikace prahové hodnoty a funkce *threshold*. Pro tyto účely je ale nejprve nutné obrázek převést na stupně šedi. Velikost prahu je argumentem funkce *findLED* a její hodnotu určuje uživatel. V experimentech byly často využívány hodnoty v rozmezí 150 až 230.

```
Mat gray;
cvtColor(res, gray, COLOR_BGR2GRAY);
```

```
Mat th;
threshold(gray, th, thresh, 255, THRESH_BINARY);
```



Obrázek 16. Aplikace prahové hodnoty

12.4.4 Vyhledání kontur v oblasti zájmu

Jak lze vidět na snímku výše, pomocí aplikace prahové hodnoty jsme nenašli pouze jednu lesklou oblast, ale v tomto konkrétním případě jednu větší a jednu menší. Pro nalezení jejich polohy ve snímku, lze využít funkci findContours.

```
vector<vector<Point>> contours;
findContours(th, contours, RETR_TREE, CHAIN_APPROX_SIMPLE);
```

12.4.5 Určení kontury zaujímající největší oblast

Po nalezení všech lesklých části na obvodu Kilobota určíme tu, která je z nich největší a porovnáme ji s parametrem, který určuje, jakou minimální velikost musí LED dioda zabírat. Pokud oblast, kterou zabírá je větší než tento parametr, tak ji považujeme za detekovanou diodu, pokud ne, tak se pravděpodobně jedná pouze o falešnou detekci a dioda v tento moment nesvítí. Nastavení této hodnoty je čistě na uživateli, bude ovlivněna zejména přiblížením kamery. V experimentech se jednalo o desetiny až jednotky procent.

Výsledný algoritmus:

```
int biggestId = -1;
int biggestSize = -1;
Rect biggestBox;
```

Projdu všechny nalezené kontury a určím hranice prostoru, kde se nachází. Vypočtu velikost oblasti, kterou zaujímají a pokud je větší než doposud největší nalezená oblast, tak aktualizuji proměnné týkající se oblasti největší kontury.

```
for (int c = 0; c < contours.size(); c++) {
    Rect cBox = boundingRect(contours[c]);

    double ledArea = (double)cBox.width * (double)cBox.height;
```

```
    if (ledArea > biggestSize) {
        biggestId = c;
        biggestSize = ledArea;
        biggestBox = cBox;
    }
}
```

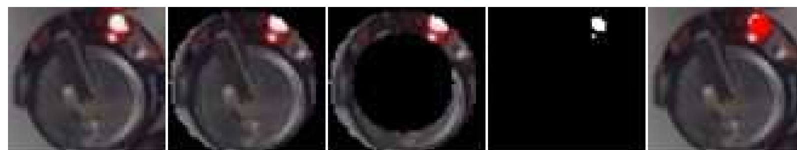
Pokud počet kontur byl nenulový, tak u největší kontury zjistím, kolik procent z celkové oblasti snímku Kilobota dioda zabírá a jestli je tato hodnota větší než požadovaná minimální velikost, tak spočtu střed dané LED diody a přepíšu hodnotu proměnných *x* a *y*.

```
if (biggestId != -1) {
    double kilobotArea = (double)th.rows * (double)th.cols;
    double ledArea =
        (double)biggestBox.width * (double)biggestBox.height;
    double percentage = (ledArea * 100) / kilobotArea;

    if (percentage > minArea) {
        x = box.tl().x + biggestBox.tl().x + biggestBox.width / 2;
        y = box.tl().y + biggestBox.tl().y + biggestBox.height /2;
    }
}
```

Na konci funkce už zbývá jen vrátit nalezenou pozici *x* a *y* pomocí

```
return Point(x, y);
```

Na obrázku níže je zobrazen postup úprav části snímku obsahující Kilobota, kde na posledním z nich je červeně vyznačen střed detekované oblasti diody.



Obrázek 17. Časová posloupnost hledání LED diody

Předpokladem je, že tato funkce bude využita pro uložení pozice LED diody v každém snímku, a to konkrétně do vektoru ve struktuře Kilobot.

12.5 Detekce pomocí neuronové sítě

Navržená třída vychází z tutoriálu volně dostupného na webu. [49]

Pro účely detekce za pomocí neuronové sítě byla navržena třída YoloDetector. Je závislá na třech hlavičkových souborech – <opencv2/opencv.hpp>, <opencv2/dnn.hpp>, <kilobot.h> a <random>.

Třída obsahuje 2 veřejné metody a 1 privátní. Veřejnými jsou LoadNet, pomocí které lze načíst soubor s natrénovanou neuronovou sítí a Detect, která ve snímku detekuji Kiloboty. Privátní metodou je metoda _format, která upravuje snímek pro potřeby neuronové sítě.

Dále třída obsahuje 2 privátní proměnné, první typu cv::dnn::Net s názvem _net určuje neuronovou síť a druhá typu std::vector<cv::Mat> s názvem _outputs ukládá výstupy z neuronové sítě.

```
class YoloDetector {

public:
    YoloDetector();

    void LoadNet(string pathToFile, bool is_cuda);

    void Detect(cv::Mat& image, std::vector<Kilobot>& output,
float scoreVal, float confVal, float nmsVal);

private:
    cv::Mat _format(const cv::Mat& source);

private:
    cv::dnn::Net _net;
    std::vector<cv::Mat> _outputs;
};
```

12.5.1 LoadNet

Metoda LoadNet slouží k načtení neuronové sítě ze souboru daného parametrem *pathToFile*, který udává cestu k souboru s neuronovou sítí. Druhým parametrem je proměnná typu boolean, která udává, jestli chceme, aby se síť pokusila využít běhu na grafickém procesoru za pomocí CUDA. Pokud bude tento parametr nastaven na true, ale nepodaří se z nějakého důvodu spustit běh na grafickém procesoru, tak se automaticky síť spustí na CPU.

```
void YoloDetector::LoadNet(string pathToFile, bool is_cuda)
{
    _net = cv::dnn::readNet(pathToFile);
    if (is_cuda)
    {
        std::cout << "Running with CUDA\n";
```

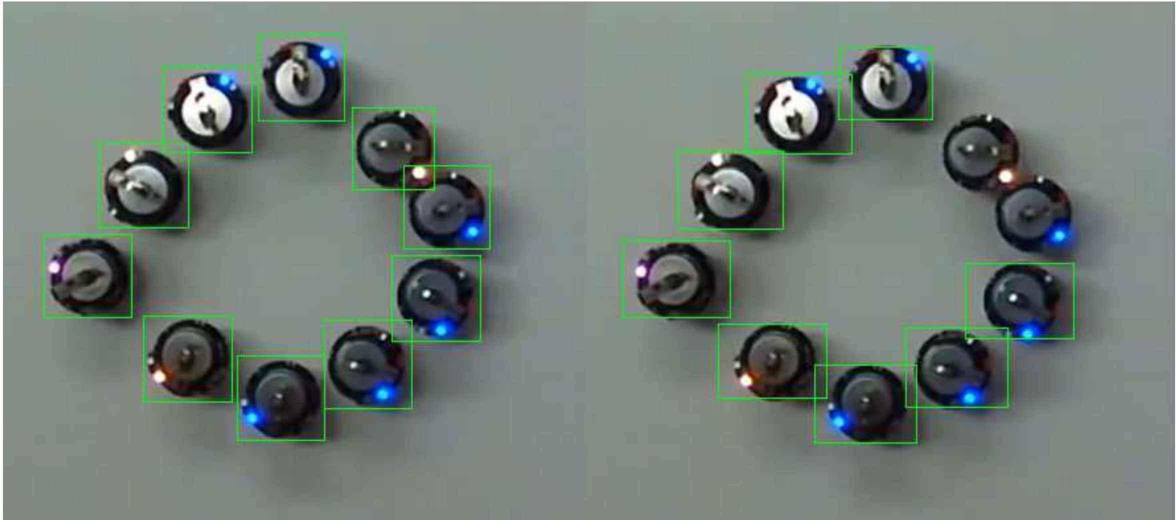
```
        _net.setPreferableBackend(cv::dnn::DNN_BACKEND_CUDA);
        _net.setPreferableTarget(cv::dnn::DNN_TARGET_CUDA_FP16);
    }
    else
    {
        std::cout << "Running on CPU\n";
        _net.setPreferableBackend(cv::dnn::DNN_BACKEND_OPENCV);
        _net.setPreferableTarget(cv::dnn::DNN_TARGET_CPU);
    }
}
```

12.5.2 _format

Metoda _format slouží k úpravě snímku pro neuronovou síť, která byla natrénována na 640x640 pixelů. Zarovnává snímek tak, aby jeho šířka a výška byla shodná a to tak, že doplní chybějící pixely.

```
cv::Mat YoloDetector::_format(const cv::Mat& source)
{
    int col = source.cols;
    int row = source.rows;
    int _max = MAX(col, row);
    cv::Mat result = cv::Mat::zeros(_max, _max, CV_8UC3);

    source.copyTo(result(cv::Rect(0, 0, col, row)));
    return result;
}
```

Na obrázku níže je vidět rozdíl, kde vlevo byla detekce provedena na zarovnaném a vpravo na nezarovnaném snímku.



Obrázek 18. Rozdíl v detekci mezi zarovnaným a nezarovnaným snímkem

Z obrázku je jasné, že provedená úprava zvyšuje přesnost detekce pomocí neuronové sítě. Oproti nezarovnanému snímku je detekce přesnější, u nezarovnaného se dokonce neuronové síti nepodařilo detekovat se stejným nastavením všechny Kiloboty a tam kde je detekovala je šířka detekcí zbytečně velká a zabírá i oblast, ve které se už Kiloboti nenachází.

12.5.3 Detect

Metoda Detect slouží k hledání Kilobotů ve snímku *image*. Detekované Kiloboty uloží do výstupního vektoru *output*. Ten můžeme filtrovat pomocí parametrů score, conf a nms, kterými můžeme odstranit duplicitní detekce a detekce, u kterých si nebyla neuronová síť příliš jistá.

Na začátku metody ověříme, že byla načtena neuronová síť a v opačném případě vyvoláme chybovou hlášku. Poté upravíme vstupní snímek metodou _format a vyrobíme z něj binární data pomocí funkce blobFromImage. Zadané parametry funkce upraví obrázek do rozlišení neuronové sítě (640x640 px), normalizují pixely na 0 nebo 1 a prohodí kanály červené a modré barevné složky.

```
void YoloDetector::Detect(cv::Mat& image, std::vector<Kilobot>&
output, float score, float conf, float nms)
{
    if (_net.empty()) {
        std::cout << "There is no Net available for detection!\n";
        return;
    }
```

```
    cv::Mat blob;

    auto input_image = _format(image);

    cv::dnn::blobFromImage(input_image, blob, 1. / 255., 
cv::Size(INPUT_WIDTH, INPUT_HEIGHT), cv::Scalar(), true, false);
```

Poté zadáme vytvořený BLOB na vstup neuronové sítě a pomocí *_net.forward()* provedeme samotnou detekci. Výstupy uložíme do proměnné *_outputs*.

```
    _net.setInput(blob);
    _net.forward(_outputs, _net.getUnconnectedOutLayersNames());
```

Dále spočteme poměr mezi upraveným snímkem a rozlišením neuronové sítě, o který budeme později posouvat a zvětšovat detekovanou oblast okolo kilobota.

```
    float x_factor = input_image.cols / INPUT_WIDTH;
    float y_factor = input_image.rows / INPUT_HEIGHT;
```

Uložíme si do proměnné data ukazatel na začátek dat, které budeme dál procházet. Definujeme počet dimenzí výstupního pole, které je dáno parametry *x, y, šířka, výška, míra jistoty, skóre.* Definujeme počet řádků výstupu, kterých je pro rozlišení 640x640 px 25200.

```
    float* data = (float*)_outputs[0].data;
    const int dimensions = 6;
    const int rows = 25200;

    std::vector<float> confidences;
    std::vector<cv::Rect> boxes;
```

Projdeme celou matici s výstupy a vyfiltrujeme pouze ty, které mají míru jistoty a skóre větší než zadané parametry metody. U takto vyfiltrovaných detekcí si uložíme data, která nás zajímají – míru jistoty a ohraničení Kilobota. Ohraničení ještě před uložením vynásobíme dříve spočtenými velikostními poměry mezi rozlišením neuronové sítě a rozlišení formátovaného snímku.

```
    for (int r = 0; r < rows; ++r) {
        if (data[4] >= conf) {
            if (data[5] > score) {

                confidences.push_back(data[4]);

                float x = data[0];
```

```
                float y = data[1];
                float w = data[2];
                float h = data[3];
                int left = int((x - 0.5 * w) * x_factor);
                int top = int((y - 0.5 * h) * y_factor);
                int width = int(w * x_factor);
                int height = int(h * y_factor);
                boxes.push_back(cv::Rect(left, top, width,
height));
            }
        }

        data += dimensions;
    }
```

Aktuálně máme k dispozici velké množství duplicitně detekovaných Kilobotů jak lze vidět na obrázku níže. Abychom tyto duplicitní detekce odstranili, aplikujeme na vyfiltrované detekce funkci potlačující nemaximální hodnoty *NMSBoxes*, která vybere z překrývajících se hodnot tu nejpřesnější.

Obrázek 19. Duplicitně detekovaní Kiloboti

```
    std::vector<int> nms_result;
    cv::dnn::NMSBoxes(boxes, confidences, score, nms, nms_result);
```

Po aplikaci této funkce již máme každý objekt detekovaný pouze jednou, jak ukazuje obrázek níže.

Obrázek 20. Detekovaní Kiloboti po aplikace funkce NMSBoxes

Nyní už zbývá pouze umístit nalezená data do vektoru definované třídy Kilobot. Defaultně nastavíme id na -1, čímž indikujeme, že mu zatím žádné nebylo určeno, vynulujeme proměnnou značící počet snímků, na kterých nebyl detekován a vygenerujeme náhodnou barvu pro vykreslení.

```
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distr(0, 255);

    for (int n = 0; n < nms_result.size(); n++) {
        int idx = nms_result[n];
        Kilobot result;
        result.confidence = confidences[idx];
        result.box = boxes[idx];
        result.id = -1;
        result.undetected = 0;

        result.color = Scalar(distr(gen), distr(gen), distr(gen));

        output.push_back(result);
    }
}
```

12.6 Tracking

Pro účely sledování Kilobotů v obraze byla navržena třída EuclideanTracker. Třída obsahuje dvě veřejné metody. První z nich, metoda Reset slouží k přenastavení parametrů pro trackování. Lze díky ní nastavit počáteční id, od které začneme Kiloboty číslovat, maximální vzdálenost, kterou může Kilobot mezi snímky urazit, abychom ho stále považovali za toho samého, a maximální počet snímků, na kterých nebyl detekován.

Druhá metoda s názvem Track se stará o samotné trackování. Obsahuje parametry pro detekci LED diody, jejíž poloha je zaznamenávána spolu s trajektorií a díky tomu lze zpětně v konkrétních bodech určit i úhel natočení.

Dále třída obsahuje privátní metodu *_findObject*, ve které se vyhledává konkrétní Kilobot v zadaném vektoru Kilobotů.

Pokud Kilobota na několika po sobě jdoucích snímcích nejsme schopni detekovat, tak jej stále udržujeme v paměti po dobu, dokud je počet snímků menší než maximální povolený počet, a to konkrétně ve vektoru s názvem *_prevUndetected*. Po dosažení maximálního počtu snímků, na kterých se nepodařilo Kilobota detekovat jej teprve vyřadíme. Při případné další detekci už neznáme jeho původní identitu a je mu vygenerované nové id.

```
class EuclideanTracker {
public:
    EuclideanTracker(int lastId = defId, int maxDist = defDist,
int maxUndetected = defUndetected);

    void Reset(int lastId = defId, int maxDist = defDist, int ma-
xUndetected = defUndetected);
    void Track(Mat frame, std::vector<Kilobot>& output,
std::vector<Kilobot>& prevOutput, double LEDthreshVal, double LED-
PercentageVal);

private:
    void _findObject(Mat& frame, Kilobot& obj, std::vector<Kilo-
bot>& objects, double LEDthreshVal, double LEDPercentageVal);

private:
    int _maxDist;
    int _maxUndetected;

    int _lastId;

    std::vector<Kilobot> _prevUndetected;
};
```

12.6.1 Reset

Metoda reset podle zadaných hodnot přenastaví proměnné _lastId, _maxDist a _maxUndetected a vyčistí vektor obsahující nedetekované Kiloboty z předešlých snímků.

```
void EuclideanTracker::Reset(int lastId, int maxDist, int maxUnde-
tected)
{
    _lastId = lastId;
    _maxDist = maxDist;
    _maxUndetected = maxUndetected;
    _prevUndetected.clear();
}
```

12.6.2 _findObject

V této metodě hledáme Kilobota v daném vektoru pomocí určení nejmenší Eukleidovské vzdálenosti.

```
void EuclideanTracker::_findObject(Mat& frame, Kilobot& obj,
str::vector<Kilobot>& objects, double LEDthreshVal, double LEDPer-
centageVal) {
```

Nejdříve nastavíme index na -1, což značí, že se Kilobota nepodařilo najít, spočteme střed detekovaného Kilobota a nejmenší nalezenou vzdálenost nastavíme na maximální možnou.

```
    Rect box = obj.box;

    int x = box.br().x - box.width / 2;
    int y = box.br().y - box.height / 2;

    double minDistance = _maxDist;
    int index = -1;
```

Poté projdeme vektor s Kiloboty a u těch, kterým doposud nebylo přiřazené id spočteme vzdálenost od středu aktuálně hledaného Kilobota. Pokud je výsledná vzdálenost menší než doposud nejmenší nalezená vzdálenost, tak si uložíme index tohoto kilobota ve vektoru.

```
    for (int o = 0; o < objects.size(); o++) {
        if (objects[o].id == -1)
            continue;

        Rect prevR = objects[o].box;
        Point p = Point(prevR.tl().x + prevR.width / 2,
prevR.tl().y + prevR.height / 2);
```

```
        Point q = Point(box.tl().x + box.width / 2, box.tl().y +
box.height / 2);

        double distance = sqrt(abs(pow(q.x - p.x, 2) + pow(q.y -
p.y, 2)));

        if (distance < minDistance) {
            minDistance = distance;
            index = o;
        }
    }
```

Po projití celého vektoru, pokud jsme v něm Kilobota nalezli, tak mu přiřadíme id, dříve používanou barvu, detekovanou trasu a polohy LED diody a přidáme hodnoty z aktuálního snímku. Také označíme id ve vektoru jako -1, čímž dáme najevo, že tento objekt byl už přiřazen.

```
    if (index != -1) {
        obj.id = objects[index].id;
        objects[index].id = -1;

        obj.trajectory = objects[index].trajectory;
        obj.trajectory.push_back(Point(x, y));

        obj.led = objects[index].led;
        obj.led.push_back(findLED(frame, obj.box, LEDthreshVal,
LEDPercentageVal));

        obj.color = objects[index].color;
}
```

12.6.3 Track

```
void EuclideanTracker::Track(Mat frame, std::vector<Kilobot>& out-
put, std::vector<Kilobot>& prevOutput, double LEDthreshVal, double
LEDPercentageVal)
{
```

V metodě track nejprve zkusíme Kilobota najít v detekcích z předešlého snímku, pokud ho tam nenajdeme, tak se ho pokusíme vyhledat ještě ve vektoru s Kiloboty, které se na předchozích snímcích nepodařilo trackovat.

```
    std::vector<Kilobot> undetected;

    for (int o = 0; o < output.size(); o++)
```

```
    {
        auto detection = output[o];

        auto box = detection.box;

        _findObject(frame, output[o], prevOutput, LEDthreshVal,
LEDPercentageVal);

        if (output[o].id == -1) {
            _findObject(frame, output[o], _prevUndetected, LED-
threshVal, LEDPercentageVal);
```

Pokud Kilobota nenajdeme ani v jednom z vektorů, tak mu přiřadíme nové id. Pokud by hodnota counteru dosáhla maximální hodnoty integeru, tak ji vyresetujeme.

```
            if (output[o].id == -1) {
                output[o].id = _lastId;
                _lastId++;


                if (_lastId == INT_MAX)
                    _lastId = 1;
            }
        }
    }
```

Veškeré Kiloboty, které se nepovedlo spárovat uchováme v paměti, zvýšíme jim počet nedetekovaných snímků o 1 a pokud je tento počet menší než maximální povolený počet snímků, na kterých mohou být nedetekování, tak je přiřadíme do vektoru *_prevUndetected*.

```
    for (int po = 0; po < prevOutput.size(); po++) {
        if (prevOutput[po].id != -1)
            undetected.push_back(prevOutput[po]);
    }

    for (int pu = 0; pu < _prevUndetected.size(); pu++) {
        if (_prevUndetected[pu].id != -1)
            undetected.push_back(_prevUndetected[pu]);
    }

    _prevUndetected.clear();

    for (int u = 0; u < undetected.size(); u++) {
        undetected[u].undetected++;

        if (undetected[u].undetected < _maxUndetected)
            _prevUndetected.push_back(undetected[u]);
```

```
    }
};
```

12.7 Ukázka použití

12.7.1 Kalibrace

V prvotní fázi je potřeba provést kalibraci kamery. Pro další použití vypočtených matic je pomocí metody *StoreCalibration* lze uložit na disk.

```
int main()
{
    vector<vector<Point3f>> objpoints;
    vector<vector<Point2f>> imgpoints;

    Mat cameraMatrix, distCoeffs, R, T;

    Size size;

    if (CalculatePoints("images/", ".jpg", 6, 7, objpoints, img-
points, size)) {
        calibrateCamera(objpoints, imgpoints, size, cameraMatrix,
distCoeffs, R, T);

        StoreCalibration("calibration.xml", cameraMatrix, distCo-
effs);

        Mat original;
        Mat undistorted;
```

Po výpočtu matic je lze aplikovat na zkreslený snímek a upravit ho.

```
        while (true) {

            original = imread("images/test.jpg");

            undistort(original, undistorted, cameraMatrix, distCo-
effs);

            imshow("original", original);
            imshow("undistorted", undistorted);

            if (waitKey(1) != -1)
            {
                std::cout << "finished by user\n";
                break;
            }
```

```
        }
    }
    else
        std::cout << "Not enough good images for camera calibra-
tion.";

    return 0;
}
```

Po prvotní kalibraci už není potřeba ji při každém použití provádět znovu. Místo volání funkcí *CalculatePoints*, *calibrateCamera* a *StoreCalibration* stačí pouze načíst uložený soubor pomocí metody *LoadCalibration* a aplikovat matice stejně jako po kalibraci.

```
int main()
{
    cv::Mat cameraMatrix, distCoeffs;

    Mat original;
    Mat undistorted;

    LoadCalibration("calibration.xml", cameraMatrix, distCoeffs);

    if (cameraMatrix.empty() || distCoeffs.empty()) {
            std::cout << "matrices were not loaded correctly" <<
std::endl;
            return 0;
    }

    original = imread("images/test.jpg");

    undistort(original, undistorted, cameraMatrix, distCoeffs);

    imshow("original", original);
    imshow("undistorted", undistorted);

    waitKey(0);

    return 0;
}
```

12.7.2 Měření vzdáleností

Při měření vzdálenosti kamery od podložky je potřeba umístit přímo pod kameru přibližně do středu jednoho Kilobota. V aplikaci pak stačí zavolat funkci *centerKilobotDetection*, které předáme parametry *percentage* a *threshold* nastavený trackbary.

```
int main() {
```

```
	int percentage = 50;
	int thresh = 120;

	namedWindow("Tracks", WINDOW_AUTOSIZE);
	createTrackbar("percentage", "Tracks", &percentage, 100);
	createTrackbar("threshold", "Tracks", &thresh, 255);

	Mat frame;

	while (true) {

		if (percentage == 0)
			percentage = 1;

		if (thresh == 0)
			thresh = 1;

		centerKilobotDetection(frame, "images/test.png", per-
centage, thresh, true);

		imshow("output", frame);

		if (waitKey(1) != -1)
		{
			std::cout << "finished by user\n";
			break;
		}
	}

	return 0;
}
```

12.7.3 Tracking

Níže je ukázka zahrnující detekci a následovné trackování Kilobotů.

```
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>

#include "kilobot.h"
#include "euclidian_tracker.h"
#include "yolo_detector.h"
#include "kilolib_functions.h"

using namespace kilolib;
using namespace cv;
using namespace std;
```

```
int main(int argc, char** argv)
{
```

Nejdříve je potřeba vytvořit detektor, tracker a načíst soubor obsahující model neuronové sítě a pokusíme se otevřít soubor s videem.

```
    YoloDetector detector;
    detector.LoadNet("res/bestCUDA.onnx", true);

    EuclidieanTracker tracker;

    cv::VideoCapture capture("res/vid1.mp4");

    if (!capture.isOpened())
    {
        std::cerr << "Error opening video file\n";
        return -1;
    }
```

Poté vytvoříme trackbary, abychom byli schopni manipulovat s parametry detekce a trackingu.

```
    int nms = 0;
    int score = 5;
    int conf = 5;
    int t = 200;
    int per = 50;

    namedWindow("Controls", WINDOW_AUTOSIZE);
    createTrackbar("NMS", "Controls", &nms, 10);
    createTrackbar("Score", "Controls", &score, 10);
    createTrackbar("Confidence", "Controls", &conf, 10);
    createTrackbar("Threshold", "Controls", &t, 255);
    createTrackbar("Percentage", "Controls", &per, 50);
```

Dále je potřeba definovat vektory pro ukládání detekovaných Kilobotů. Ve vektoru *output* budou detekce na aktuálním snímku a v *prevOutput* budou detekovaní Kiloboti ze snímku předchozího.

```
    std::vector<Kilobot> output;
    std::vector<Kilobot> prevOutput;
```

Poté postupně načítáme jednotlivé snímky a nejprve voláme funkci pro detekci Kilobotů s parametry nastavenými pomocí trackbarů. Detekované Kiloboty uložíme do vektoru

output a poté je pokusíme spárovat s Kiloboty z předchozího snímku pomocí zavolání metody *Track*.

```
    cv::Mat frame;

    while (true)
    {
        capture.read(frame);

        if (frame.empty())
        {
            std::cout << "End of stream\n";
            break;
        }

        detector.Detect(frame, output, (float)(nms + 1) / 10,
(float)(conf + 1) / 10, (float)(score + 1) / 10);

        tracker.Track(frame, output, prevOutput, (double)t + 1,
per / 10);
```

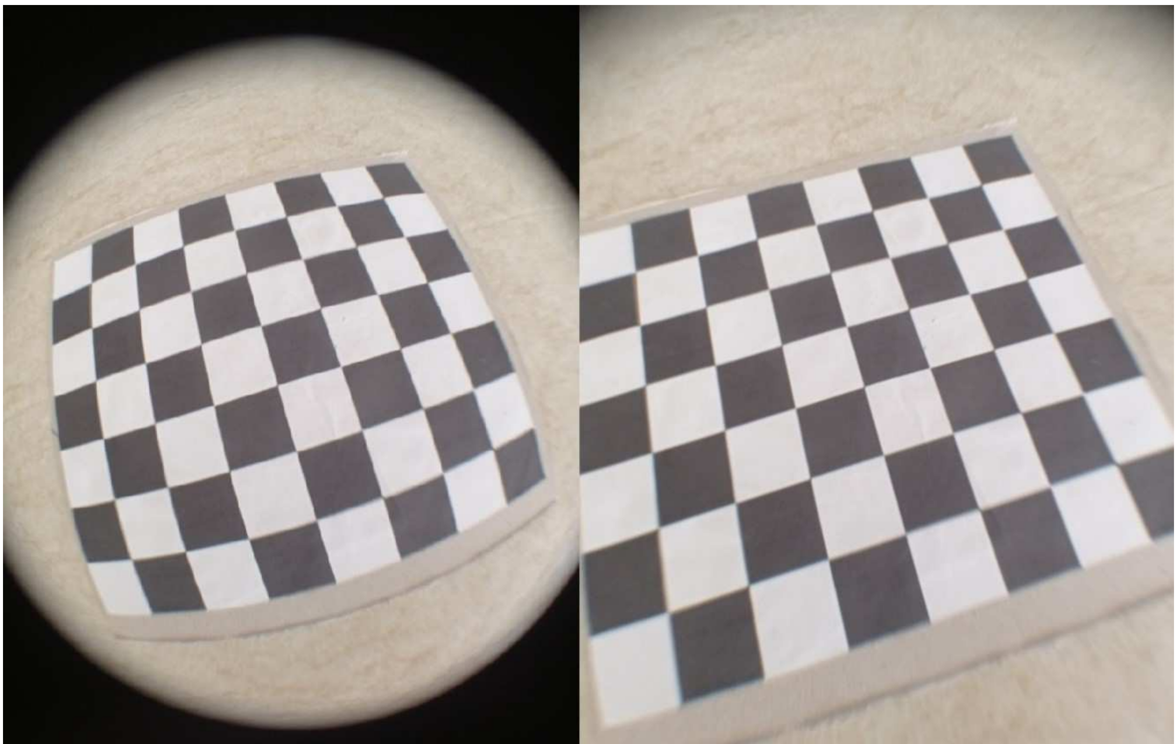
Pak už zbývá pouze vykreslit snímek, předat detekované Kiloboty z aktuálního snímku do vektoru *prevOutput* a vyčistit vektor *output* pro detekci na následujícím snímku.

```
        imshow("output", frame);

        if (cv::waitKey(1) != -1)
        {
            capture.release();
            std::cout << "finished by user\n";
            break;
        }

        prevOutput = output;
        output.clear();
    }

    cv::waitKey(0);

    return 0;
}
```

13 VÝSLEDKY

13.1 Kalibrace kamery

Kalibrace byla testována na 180° čočce, přes kterou je obraz snímaný s efektem rybího oka. Vlevo je zobrazena šachovnice na neupraveném snímku, kde je vidět, že hrany jednotlivých čtverců na šachovnici, zejména ve spodní částí snímku, jsou výrazně zaobleny. Vpravo je zobrazen snímek upravený pomocí výše zmíněných funkcí. Na něm je vidět, že spodní hrana je více zarovnaná, každopádně levý spodní roh stále působí lehce oble.



Obrázek 21. Porovnání zkresleného a nezkresleného snímku

13.2 Detekce Kilobota

Na obrázku níže jsou zelenou barvou vyznačeny oblasti, které neuronová síť označila jako Kiloboty. U některých Kilobotů síť netrefila přesnou hranu, chyba je ale tak minimální, že se většinou jedná o 1 nebo 2 pixely, což je stále celkem působivé.

Obrázek 22. Detekovaní Kiloboti

13.3 Měření vzdálenosti

Při použití funkce *centerKilobotDetection* je nejprve potřeba odfiltrovat prostředí okolo Kilobota. Pomocí trackbaru nastavíme hodnotu *percentage* tak, abychom odstranili okolní části snímku.

Obrázek 23. Původní snímek

Obrázek 24. Odfiltrovaná část snímku se změřenou vzdáleností

Níže je zobrazeno porovnání měření na videích, z kterých byla počítána konstanta podobnosti. Původní vzdálenosti jsou 82.8, 102,2 a 116 cm.

Obrázek 25. Programově změřené vzdálenosti před korekcí

Průměrná hodnota koeficientu vycházela na -1246,26. Chyba byla v prvním případě 0,13, ve druhém 0,62 a ve třetím 1,5.

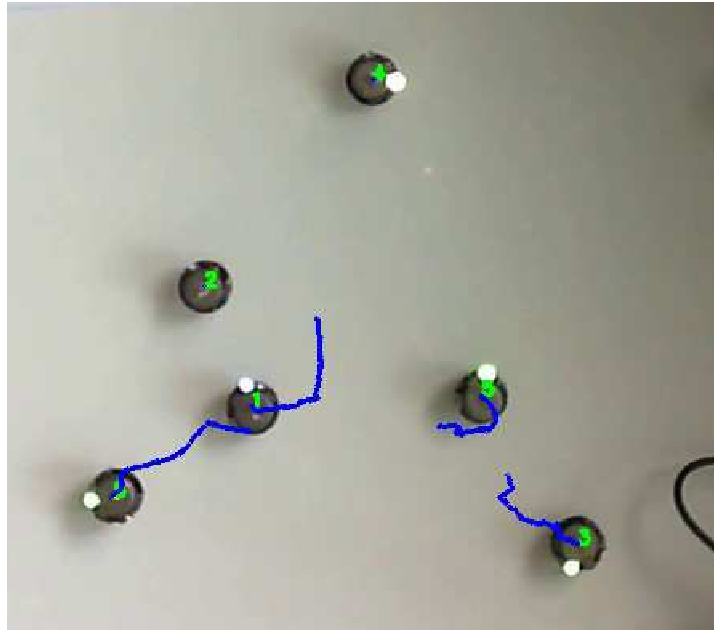
Provedl jsem malou korekci a snížil hodnotu na -1240. Při této hodnotě algoritmus naměřil vzdálenosti 83,51, 102,3 a 116,91, čímž se sice zhoršila přesnost prvního měření, ale zlepšily se hodnoty u dalších dvou. Chyba je zde pro první měření 0,29, pro druhé 0,1 a pro třetí 0,91.

Obrázek 26. Programově změřené vzdálenosti po korekci

Další úprava by měla smysl až tehdy, pokud se výrazně změnila vzdálenost, ze které bychom Kiloboty nahrávali.

13.4 Trajektorie Kilobota

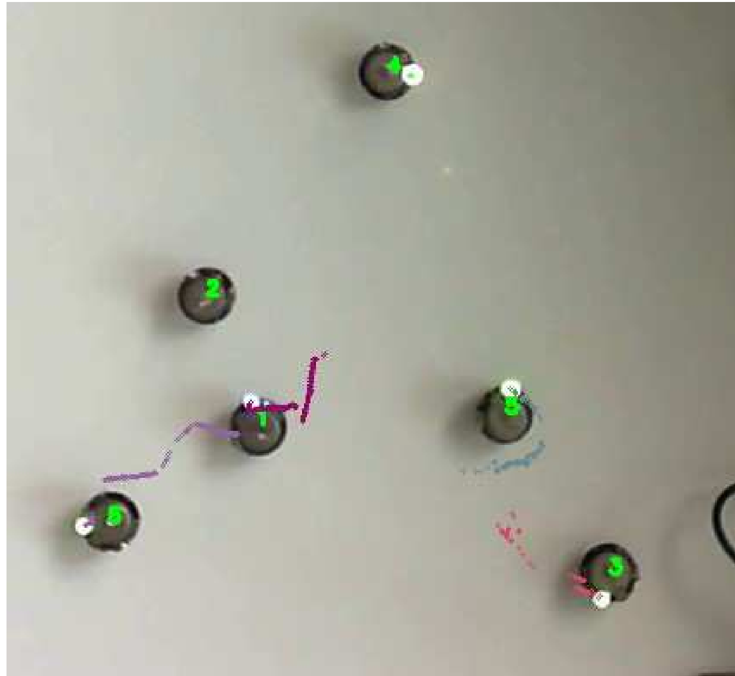
Díky určení identity každému z Kilobotů, můžeme ze získaných dat určit jejich trajektorii. Na obrázku níže je zobrazeno přiřazené id každému z nich a modrou barvou vykreslena trasa, kterou urazili za cca 15 s videa.



Obrázek 27. Trajektorie Kilobotů

13.5 Detekce LED diody

Na rozdíl od trajektorie, detekce LED se nemusí povést v každé iteraci, protože LED nemusí zrovna svítit. Pokud se tak stane, je do vektoru uložena hodnota Point(-1,-1), abychom udrželi znalost konkrétního natočení v konkrétním bodě trajektorie. Zejména na Kilobotech vpravo s číslem 3 a 5 můžeme vidět, že se LED v určitých bodech detekovat nepodařilo a ve výsledném vykreslení LED diody v čase tak vznikly mezery.



Obrázek 28. Detekované LED diody Kilobotů

Na obrázku níže je zobrazeno porovnání detekce LED (náhodně vygenerovaná barva) a detekce trajektorie (modrá barva).

Obrázek 29. Porovnání detekovaných trajektorií a LED diod

ZÁVĚR

Cílem práce bylo navrhnout open source knihovnu, kterou by výzkumníci v oblasti hejnové inteligence mohli využívat pro vyhodnocování experimentů s Kiloboty. Při jejím použití je nyní možné z kamerového záznamu experimentu získat data, která mohou být výzkumníky dále zpracována. V knihovně byly navrženy veškeré funkcionality dané zadáním práce a knihovna jako taková je výzkumníkům k dispozici pro další rozšiřování. Nalezne uplatnění nejen pro rozvoj výzkumu hejnové inteligence na Univerzitě Tomáše Bati ve Zlíně, ale i v dalších laboratořích na světě. Obrovským přínosem je univerzálnost použití bez ohledu na konkrétní hardwarovou vybavenost, která doposud nebyla možná.

Rychlost zpracování experimentů je závislá zejména na detekci Kilobotů pomocí modelu umělé neuronové sítě a pohybuje se okolo 18 snímků za vteřinu. Je zde ale možné využít schopnost OpenCV spustit běh modelu za pomocí CUDA, a tím rychlost zpracování zvýšit na cca 38 snímků za vteřinu.

Výsledná knihovna je použitelná pro sběr a vyhodnocení dat z kamerového záznamu experimentů s Kiloboty, ale způsob jejich uchování po ukončení běhu nechává již v rukou uživatele, tak aby bylo možno knihovnu implementovat i do již existujících aplikací s definovaným formátem vstupu.

SEZNAM POUŽITÉ LITERATURY

[1] RUBENSTEIN, Michael; AHLER, Christian; NAGPAL, Radhika. Kilobot: A low cost scalable robot system for collective behaviors. In: 2012 IEEE international conference on robotics and automation. IEEE, 2012. p. 3293-3298.

[2] Reina, A., Cope, A. J., Nikolaidis, E., Marshall, J. A., & Sabo, C. (2017). ARK: Augmented reality for Kilobots. IEEE Robotics and Automation letters, 2(3), 1755-1761.

[3] TAYLOR, Michael. Neural Networks: A Visual Introduction for Beginners. Blue Windmill Media, 2017. ISBN 1549869132.

[4] KANTARDZIC, Mehmed. Data Mining: Concepts, Models, Methods, and Algorithms. 2nd edition. Wiley-IEEE Press, 2011. ISBN 1118029143.

[5] HASSANIEN, Aboul Ella a Eid EMARY. Swarm Intelligence: Principles, Advances, and Applications. CRC Press, 2015. ISBN 1498741061.

[6] RUBENSTEIN, Michael, Christian AHLER a Radhika NAGPAL. Kilobot: A low cost scalable robot system for collective behaviors. In: 2012 IEEE International Conference on Robotics and Automation [online]. IEEE, 2012, 2012, s. 3293-3298 [cit. 2022-05-18]. ISBN 978-1-4673-1405-3. Dostupné z: doi:10.1109/ICRA.2012.6224638

[7] Kilobot - ROBOTS: Your Guide to the World of Robotics. ROBOTS: Your Guide to the World of Robotics [online]. [cit. 2022-05-16]. Dostupné z: https://robots.ieee.org/robots/kilobot/

[8] HEATON, Jeff. Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks. CreateSpace Independent Publishing Platform, 2015. ISBN 1505714346.

[9] A Gentle Introduction to Pooling Layers for Convolutional Neural Networks. Machine Learning Mastery [online]. [cit. 2022-05-16]. Dostupné z: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

[10] BASHA, S.H. Shabbeer, Shiv Ram DUBEY, Viswanath PULABAIGARI a Snehasis MUKHERJEE. Impact of fully connected layers on performance of convolutional neural networks for image classification. Neurocomputing [online]. 2020, 378, 112-119 [cit. 2022-05-17]. ISSN 09252312. Dostupné z: doi:10.1016/j.neucom.2019.10.008

[11] Fully Connected Layer: The brute force layer of a Machine Learning model. OpenGenus IQ: Computing Expertise & Legacy [online]. [cit. 2022-05-16]. Dostupné z: https://iq.opengenus.org/fully-connected-layer/

[12] JIANG, Xiaoyue, Abdenour HADID, Yanwei PANG, Eric GRANGER a Xiaoyi FENG. Deep Learning in Object Detection and Recognition. Singapore: Springer, 2019. ISBN 9811051526.

[13] YOLOv5 Documentation. Yolov5 and Vision AI - Ultralytics [online]. [cit. 2022-05-16]. Dostupné z: https://docs.ultralytics.com/

[14] Welcome To Colaboratory. Google [online]. [cit. 2022-05-16]. Dostupné z: https://colab.research.google.com/

[15] Google Colab. Google [online]. [cit. 2022-05-16]. Dostupné z: https://colab.research.google.com/signup/pricing?utm_source=dialog&utm_medium=link&utm_campaign=settings_page

[16] OpenCV [online]. [cit. 2022-05-16]. Dostupné z: https://opencv.org/

[17] VILLÁN, Alberto Fernández. Mastering OpenCV 4 with Python: A Practical Guide Covering Topics from Image Processing, Augmented Reality to Deep Lear-ning with OpenCV 4 and Python 3.7. Packt Publishing, 2019. ISBN 1789349753.

[18] OpenCV: cv::Point_< _Tp > Class Template Reference. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/db/d4e/classcv_1_1Point__.html

[19] OpenCV: cv::Point3_< _Tp > Class Template Reference. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/df/d6c/classcv_1_1Point3__.html

[20] OpenCV: cv::Rect_< _Tp > Class Template Reference. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d2/d44/classcv_1_1Rect__.html

[21] OpenCV: cv::Scalar_< _Tp > Class Template Reference. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d1/da0/classcv_1_1Scalar__.html

[22] OpenCV: cv::Mat Class Reference. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/4.x/d3/d63/classcv_1_1Mat.html

[23] OpenCV: Image Thresholding. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html

[24] OpenCV: Contours: Getting Started. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html

[25] OpenCV: Color Space Conversions. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d8/d01/group__imgproc__color__conversions.html

[26] OpenCV: Drawing Functions. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d6/d6e/group__imgproc__draw.html

[27] OpenCV: Arithmetic Operations on Images. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d0/d86/tutorial_py_image_arithmetics.html

[28] OpenCV: Template Matching. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html

[29] OpenCV: Feature Detection. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/4.x/dd/d1a/group__imgproc__feature.html

[30] OpenCV: Cascade Classifier. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html

[31] OpenCV: Tracking API. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d9/df8/group__tracking.html

[32] OpenCV: Legacy support. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d5/dc3/group__cudalegacy.html

[33] OpenCV: Deep Neural Networks (dnn module). OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/4.x/d2/d58/tutorial_table_of_content_dnn.html

[34] OpenCV: cv::dnn::Net Class Reference. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/db/d30/classcv_1_1dnn_1_1Net.html

[35] OpenCV: Deep Neural Network module. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/3.4/d6/d0f/group__dnn.html

[36] Non Maximum Suppression: Theory and Implementation in PyTorch. AI Education For All, From Your First Step To Mastery! [online]. [cit. 2022-05-16]. Dostupné z: https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/

[37] FORSYTH, David A. a Jean PONCE. Computer vision: A modern approach. 2nd edition. Prentice Hall, 2011. ISBN 013608592X.

[38] JAHNE, Bernd. Practical handbook on image processing for scientific and technical applications. 2nd edition. CRC Press, 2004. ISBN 0849319005.

[39] Simple object tracking with OpenCV. PyImageSearch - You can master Computer Vision, Deep Learning, and OpenCV. [online]. [cit. 2022-05-16]. Dostupné z: https://pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/

[40] AUFMANN, Richard N., Vernon C. BARKER a Richard D. NATION. College Trigonometry. 6th edition. Brooks Cole, 2007. ISBN 061882507X.

[41] OpenCV: Camera calibration With OpenCV. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/4.x/d4/d94/tutorial_camera_calibration.html

[42] Video to JPG - Online Converter. Online Converter [online]. [cit. 2022-05-16]. Dostupné z: https://www.onlineconverter.com/video-to-jpg

[43] DarkMark. C Code Run [online]. [cit. 2022-05-16]. Dostupné z: https://www.ccoderun.ca/darkmark/Summary.html

[44] MakeSense [online]. [cit. 2022-05-16]. Dostupné z: https://www.makesense.ai/

[45] Ultralytics/yolov5: YOLOv5 in PyTorch > ONNX > CoreML > TFLite. GitHub [online]. [cit. 2022-05-16]. Dostupné z: https://github.com/ultralytics/yolov5

[46] How to Train YOLOv5 On a Custom Dataset. Roboflow Blog [online]. 10. června 2020 [cit. 2022-05-16]. Dostupné z: https://blog.roboflow.com/how-to-train-yolov5-on-a-custom-dataset/

[47] Camera Calibration using OpenCV. AI Education For All, From Your First Step To Mastery! [online]. [cit. 2022-05-16]. Dostupné z: https://learnopencv.com/camera-calibration-using-opencv/

[48] OpenCV: Camera Calibration. OpenCV documentation index [online]. [cit. 2022-05-16]. Dostupné z: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

[49] Detecting objects with YOLOv5, OpenCV, Python and C++. >Medium – Where good ideas find you. [online]. [cit. 2022-05-16]. Dostupné z: https://medium.com/mlearning-ai/detecting-objects-with-yolov5-opencv-python-and-c-c7cf13d1483c

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

SSD	Single Shot Detector
YOLO	You Only Look Once
RGB	Red Green Blue
RGBA	Red Green Blue Alfa
HSV	Hue Saturation Value
LED	Light-Emitting Diode
UTB	Univerzita Tomáše Bati
DNN	Deep Neural Network
BLOB	Binary Large Object
NMS	Non-Maximum Suppression
IoU	Intersetion Over Union
MIL	Multiple Instance Learning
TLD	Tracking Learning Detection
CSRT	Channel and Spatial Reliability Tracking
KCF	Kernelized Correlation Filter
MOSSE	Minimum Output Sum of Squared Error
A.I. Lab	Artificial Intelligence Laboratory
GPU	Graphics Processing unit
ONNX	Open Neural Network Exchange
HD	High Definition
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture

SEZNAM OBRÁZKŮ

Obrázek 1. Kilobot [7] 13
Obrázek 2. Neuronová síť [8] 14
Obrázek 3. Schéma umělého neuronu [8] 15
Obrázek 4. Princip RCNN [12] 18
Obrázek 5. Model dírkové komory [37] 28
Obrázek 6. Dírková komora jako 3D prostor [38] 28
Obrázek 7. Určení vzdálenosti mezi dvěma body kartézské soustavy 32
Obrázek 8. Úvodní stránka MakeSense 38
Obrázek 9. Import dat určených k označení 39
Obrázek 10. Označení Kilobotů 39
Obrázek 11. Export anotací 40
Obrázek 12. Trénování modelu v Google Colab 42
Obrázek 13. Nalezené body šachovnice 45
Obrázek 14. Odfiltrování vnější části Kilobota 53
Obrázek 15. Odfiltrování vnější části Kilobota 53
Obrázek 16. Aplikace prahové hodnoty 54
Obrázek 17. Časová posloupnost hledání LED diody 55
Obrázek 18. Rozdíl v detekci mezi zarovnaným a nezarovnaným snímkem 58
Obrázek 19. Duplicitně detekovaní Kiloboti 60
Obrázek 20. Detekovaní Kiloboti po aplikace funkce NMSBoxes 61
Obrázek 21. Porovnání zkresleného a nezkresleného snímku 71
Obrázek 22. Detekovaní Kiloboti 72
Obrázek 23. Původní snímek 72
Obrázek 24. Odfiltrovaná část snímku se změřenou vzdáleností 73
Obrázek 25. Programově změřené vzdálenosti před korekcí 73
Obrázek 26. Programově změřené vzdálenosti po korekci 73
Obrázek 27. Trajektorie Kilobotů 74
Obrázek 28. Detekované LED diody Kilobotů 75
Obrázek 29. Porovnání detekovaných trajektorií a LED diod 75

SEZNAM TABULEK

Tabulka 1. Výstup konvoluční vrstvy 17
Tabulka 2. výstup pooling vrstvy 17
Tabulka 3. Měření vzdáleností od kamery 49

SEZNAM PŘÍLOH

Příloha P I: Obsah přiloženého CD-ROM disku

PŘÍLOHA P I: OBSAH PŘILOŽENÉHO CD-ROM DISKU

- Elektronická verze diplomové práce
- Složka se zdrojovými kódy vytvořené knihovny a modelem umělé neuronové sítě