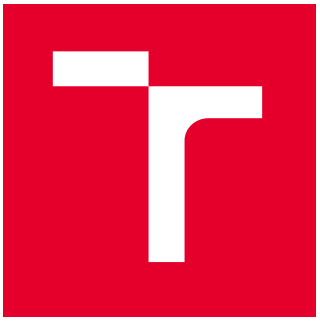


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

DETEKCE OBJEKTŮ A SLEDOVÁNÍ TRASY POHYBU ÚČASTNÍKŮ PROVOZU PRO POTŘEBY INTELIGENTNÍCH DOPRAVNÍCH UZLŮ

DETECTION OF OBJECTS AND TRACKING THE ROUTE OF MOVEMENT OF TRAFFIC PARTICIPANTS FOR THE NEEDS OF INTELLIGENT TRANSPORT NODES

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. Tomáš Vymazal

AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. Radim Burget, Ph.D.

SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Tomáš Vymazal ***ID:*** 214411

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Detekce objektů a sledování trasy pohybu účastníků provozu pro potřeby inteligentních dopravních uzlů

POKYNY PRO VYPRACOVÁNÍ:

Pořiďte několik videozáznamů dopravního uzlu pro potřeby experimentů vyhodnocení přesnosti (podrobnosti diskutujte s vedoucím). Datovou množinu rozdělte na část trénovací a na část testovací. Seznamte se s problematikou detekce s využitím neuronových sítí a zpracujte rešerši a srovnání algoritmů pro detekci objektů (EfficientDet, Scaled-YOLOv4, YOLOR, YOLOv5, …). Proveďte srovnání z pohledu přesnosti, latence, paměťových nároků a výpočetních nároků. Vytvořte experiment, který porovná přesnost vybraných detektorů a vykreslí trajektorii pohybu jednotlivých objektů.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 19.5.2023

Vedoucí práce: doc. Ing. Radim Burget, Ph.D.

prof. Ing. Jiří Mišurec, CSc.
předseda rady studijního programu

ABSTRAKT

Diplomová práce se zaměřuje na problematiku detekce objektů. Je navržen experiment, který posuzuje detekční modely YOLOv5, YOLOR, Scaled-YOLOv4 a EfficientDet a porovnává jejich vlastnosti (rychlost detekce, pamětové nároky, přesnost a jistotu detekce). K tomuto účelu je vytvořena vlastní datová sada, na které jsou tyto parametry zkoumány. Ze studie vyplývá, že nejlépe je na tom síť YOLOv5. Pro trasování objektů je použit deep SORT, který je důležitý pro následné získání trénovacích dat z videozáznamu pro predikci pohybu objektů. Přidanou hodnotou je návrh predikčního algoritmu, který je založený na polynomiálním regresním modelu.

KLÍČOVÁ SLOVA

Python3, PyTorch, YOLOv5, YOLOR, Scaled-YOLOv4, EfficientDet, deep SORT, LSTM, GRU, RNN, Polynomiální regresní model, pandas, Detekce, Trasování, Predikce

ABSTRACT

The master's thesis is focused on the object detection. The aim of this thesis is to desine an experiment to assess the detection models YOLOv5, YOLOR, Scaled-YOLOv4 and EfficientDet and to compare their properties (detection speed, memory requirements, accuracy and certainty of detection). For this purpose a custom data set is created to investigate these parameters. The study shows that the YOLOv5 network is performd as the best solution. Deep SORT is used for object tracking which is important for the subsequent extraction of training data from video footage for object movement prediction. The added value is the design of the prediction algorithm which is based on a polynomial regression model.

KEYWORDS

Python3, PyTorch, YOLOv5, YOLOR, Scaled-YOLOv4, EfficientDet, deep SORT, LSTM, GRU, RNN, Polynomial regression model, pandas, Detection, Tracking, Prediction

VYMAZAL, Tomáš. *Detekce objektů a sledování trasy pohybu účastníků provozu pro potřeby inteligentních dopravních uzlů*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 82 s. Diplomová práce. Vedoucí práce: doc. Ing. Radim Burget, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Tomáš Vymazal
VUT ID autora:	214411
Typ práce:	Diplomová práce
Akademický rok:	2022/23
Téma závěrečné práce:	Detekce objektů a sledování trasy pohybu účastníků provozu pro potřeby inteligentních dopravních uzlů

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu doc. Ing. Radimu Burgetovi Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Diplomová práce se zabývá detekcí objektů a sledováním trasy pohybu účastníků provozu pro potřeby inteligentních dopravních uzlů. Nejčastěji lze nalézt články, které cílí na potřeby autonomního řízení automobilů. I když se jedná o obdobný problém, výsledky nejsou tak jednoduše aplikovatelné na situaci, kdy jsou použity stacionární kamery. Na druhou stranu pevně stanovené pozice pro snímání obrazu umožňují zjednodušit situaci v tom, že objekty se často vyskytují ve stejných místech (auta obvykle nejezdí po chodníku). Tato výhoda umožní snadnější učení neuronové sítě.

Když už články s tématikou existují, nespojují detekci s trasováním objektů. Převážně se zaměřují pouze na rychlost detekce. [11] [12] [13] Proto je práce, která spojuje tyto témata velice vzácná a přínosná pro vědeckou komunitu.

Práce se především zaměřuje na detekci objektů v dopravních křižovatkách pro potřeby inteligentních dopravních uzlů. Pro testovací účely byla zvolena jedna brněnská křižovatka (ulice Koliště a Milady Horákové), kterou snímá dopravní kamera. Z této jedné kamery jsou pořízeny snímky, které jsou použity pro trénování a validaci zadaných neuronových sítí. Hlavním cílem práce je porovnat výsledky těchto neuronových sítí a vybrat jednu, která by byla nejoptimálnější pro trasování objektů (dopravních prostředků). Pro tyto účely je napsaný jednoduchý program, který vyhodnocuje výsledky a generuje grafy, které jsou součástí tohoto dokumentu.

Pro účely výzkumu je nutno vytvořit vlastní datovou sadu, na které je potřeba odzkoušet, jestli detekce budou vůbec možné a pokud, tak na kolik budou přesné. Další možností, která by se dala využít, je veřejná datová sadu **COCO**. Protože porovnání výsledků neuronových sítí na **COCO** je volně dostupné a nepřináší nové informace, není potřeba tyto výsledky znovu ověřovat. Podstatnější jsou výsledky, kterých je možno dosáhnout na vlastní datové sadě.

V druhé polovině se práce zaměřuje na vlastní implementaci predikčního mechanizmu, který by dokázal předpovídat polohu objektů v budoucnosti. Aby se dal tento predikční algoritmus vytvořit je nutné, aby práce byla schopna sledovat objekty mezi jednotlivými snímky. K tom je použita jedna z implementací ***deep SORT***. Schopnost trasovat a predikovat pohyb objektů je důležitou funkcionalitou moderních řídících aplikací. Jsou to například autonomní automobily, inteligentní řízení dopravních uzlů nebo prevence proti kolizím objektů.

1 Teorie

Detekce objektů v obraze je oblast počítačového vidění, která se vyvíjí rychlým tempem. Spousta informací a údajů v této oblasti se stává rychle zastaralými nebo neplatnými. Důvodem, proč je tato oblast aktivní je především potřeba zefektivnit autonomní řídící systémy. Dále je to lepší dostupnost výkonnějších grafických karet (GPU), bez kterých by tento výzkum nebyl na takové úrovni. Důkazem rychlého rozvoje je například nedávná publikace nových sítí **YOLOv6**, **YOLOv7** a **YOLOv8**. Jejich použití a začlenění do této práce není plánováno, protože jejich výsledky jsou dobře zpracovány. Z repozitářů také vyplývá, že hlavním cílem nových implementací je optimalizace výpočtů [14] [15] [16].

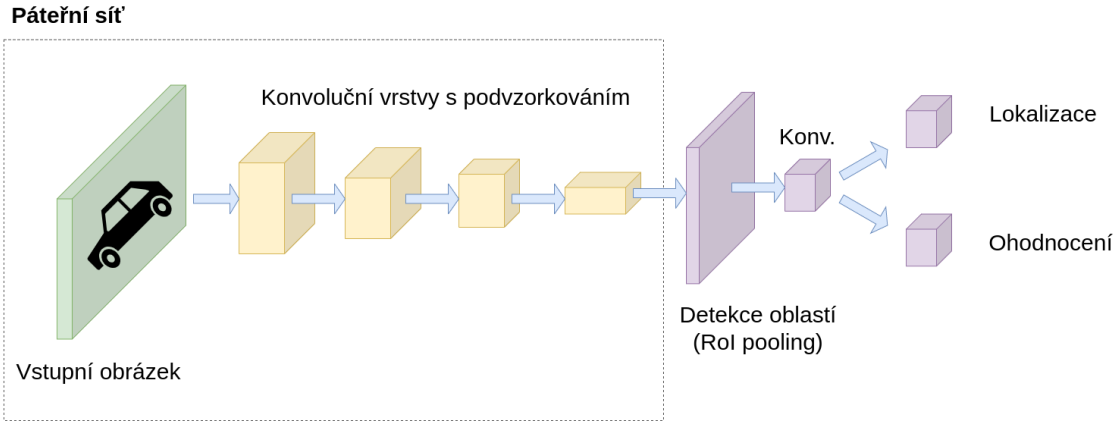
Teoretická část se zaměřuje na jednotlivé neuronové sítě a klade důraz na to, aby byly vyzdviženy rozdílné přístupy, které byly použity při jejich návrhu. Práce se zaměřuje pouze na jednostupňové detektory s kotvou. Problém nastává u neuronové sítě YOLOv5, ke které neexistují žádné oficiální dokumenty. Proto je shánění ověřených informací k této implementaci velice obtížné.

1.1 Neuronové sítě pro detekci objektů

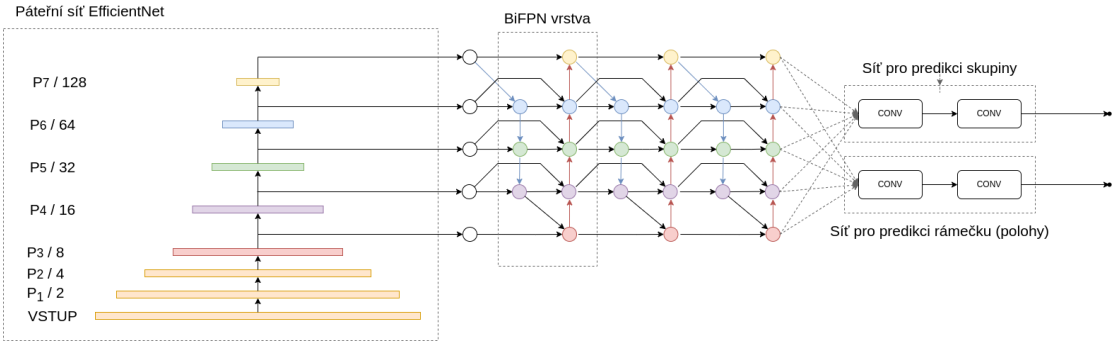
Jednou z částí, kterou se práce zabývá, je detekce objektů v obrazových snímcích. Jedná se o oblast počítačového vidění [1], která je velice dynamická a rychle se rozvíjející. Práce se především zaměřuje na detekční sítě, které jsou následně použity v praktické části. Všechny tyto sítě patří do skupiny jednostupňových detektorů s kotvou, jejíchž výhodou je velká účinnost a jednoduchost [2]. Tyto sítě jsou konstruovány pro klasifikaci a lokalizaci objektů v obrazech. Tím se chápe nalezení objektu nebo více objektů v obraze a jejich označení pomocí obdélníkového rámečku. Tomuto rámečku se také říká ohraničující rámeček [1]. Obecná struktura jednostupňového detektoru je zobrazena na obrázku 1.1.

1.1.1 EfficientDet

Jedná se o implementaci neuronové sítě pro detekci objektů od společnosti **Google**. Jejím hlavním cílem je zmenšit výpočetní nároky (snížení parametru FLOP) a zároveň zachovat přesnost detekce. Referenční sítí, se kterou byla implementace srovnávána, je síť YOLOv3 [17]. Základním stavebním kamenem sítě EfficientDet je EfficientNet, která je použita jako páteřní síť. Na páteřní síť je následně napojena síť funkcí pojmenovaná BiFPN (síť pyramidových funkcí). Tato síť se v modelu opakuje třikrát až osmkrát podle dané verze (D0 až D6). [2]



Obr. 1.1: Obecná struktura jednostupňového detektoru [1]



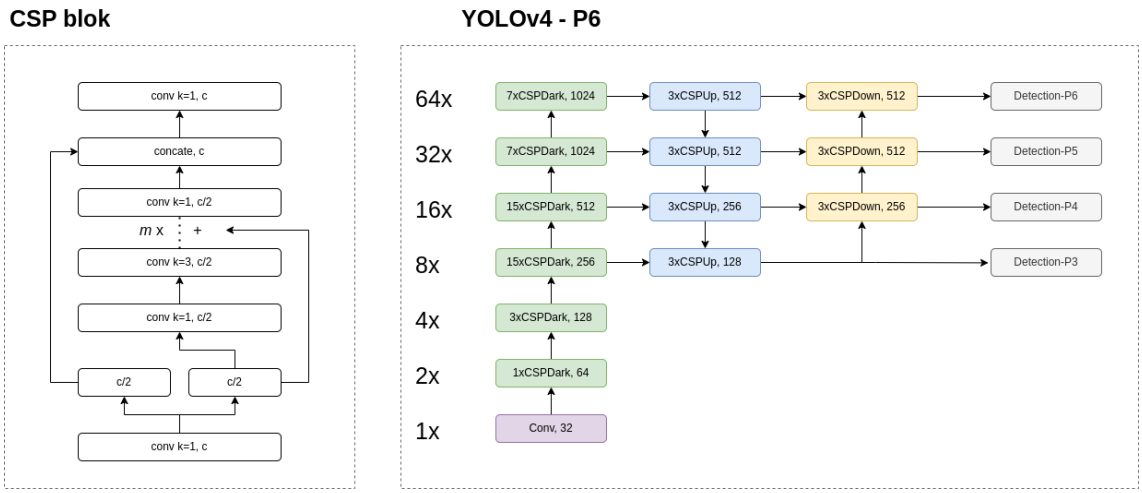
Obr. 1.2: Struktura neuronové sítě EfficientDet [2]

1.1.2 Scaled-YOLOv4

Jedná se o implementaci navrženou pro detekci objektů v reálném čase. Tato verze vychází z YOLOv4 a přepracovává tuto verzi na YOLOv4-CSP. Při návrhu je především kladen důraz na snížení výpočetních nároků při zachování přesnosti detekce tak, aby detekce bylo možné provádět i na GPU s nižším výpočetním výkonem. Páteřní sítí v tomhle případě je CSPOSANet s architekturou PCB, na rozdíl od původní verze YOLOv4, která používá CSPDarkNet53 [18]. Dalšími stavebními bloky jsou prvky CSP se škálováním nahoru (up) a dolů (down). Jejich použití je hlavním přínosem dané implementace, protože při správném návrhu sítě se dají snížit výpočetní nároky oproti YOLOv4. [3] Struktura sítě je zobrazena na obrázku 1.3, který popisuje, jak vypadá síť s CSP bloky pro verzi P6.

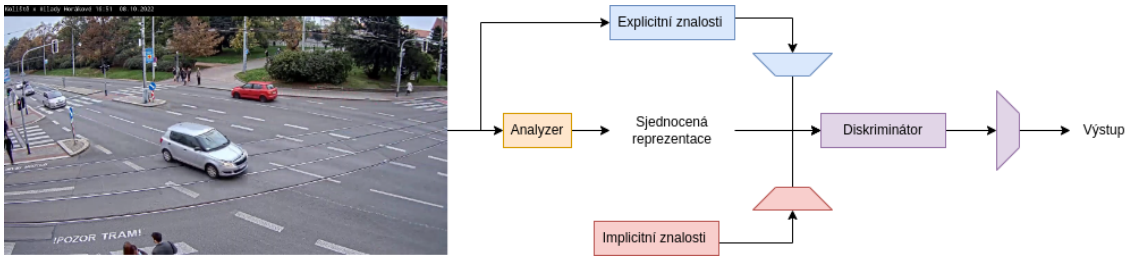
1.1.3 YOLOR

YOLOR je detekční síť, která vychází ze Scaled-YOLOv4 a rozšiřuje tuto implementaci o implicitní znalosti. V této implementaci jde především o zakomponování



Obr. 1.3: Struktura neuronové sítě Scaled-YOLOv4-P6 a CSP bloku [3]

implicitních znalostí pro dosažení lepších výsledků. Tato myšlenka se snaží napodobit procesy, které pravděpodobně provádí i lidský mozek. Výsledná síť má o trochu lepší výsledky než Scaled-YOLOv4, a navíc bylo experimentálně ověřeno, že tato síť se může učit mnohem rychleji [4]. Architekturu této sítě lze vidět na obrázku 1.3, proto je v tomto případě nutné se zaměřit na zpracování implicitní znalosti v YOLOR, viz obrázek 1.4.



Obr. 1.4: Proces zpracování implicitní znalosti v síti YOLOR. [4]

1.1.4 YOLO

Implementace **YOLOv5** byla publikována dva měsíce po vydání YOLOv4, a to formou github repositáře a do dnešního dne neexistuje oficiální dokument k této síti. Jedná se o implementaci, která byla jako první celá vytvořena v modulu PyTorch [19]. Tato síť nabízí čtyři základní velikosti modelů (YOLOv5s, YOLOv5m, YOLOv51 a YOLOv5x), rozdělené podle koncového použití. Například nejmenší model YOLOv5s byl navržen pro mobilní telefony, které nemají takový výpočetní

výkon. Nevýhodou tohoto malého modelu je, že není tak přesný [19][20]. Výhodou této implementace je její rychlost a přesnost, která je podle informací nalezených v dokumentech v repositáři velice zajímavá, dokonce v některých případech jde o zrychlení z 50 FPS u YOLOv4 na 140 FPS u YOLOv5 [21]. Struktura použité sítě v projektu je vyexportována ve formátu ***onnx*** a do obrázku byla převedena pomocí webové stránky ***https://netron.app/***. Je přiložena v repositáři pod jménem ***last-yolov5.onnx.png***.

V posledním roce také byly publikovány i nové varianty **YOLO**. Jedná se o **YOLOv6**, **YOLOv7** a **YOLOv8**. Tyto novější implementace jsou napsány pomocí Python modulu *PyTorch*. Hlavním přínosem těchto prací je zvýšení rychlosti nebo navýšení přesnosti o jednotky procent. Především jde o modifikaci za cílem snížit výpočetní nároky na nerunovou síť tak, aby byla použitelná i na běžných mobilních zařízeních [22] [23] [5]. Informace o páteřních sítích a rocích vydání jsou uvedeny v tabulce 1.1 [5].

Detekční model	**Rok vydání**	**Páteřní síť**
Scaled-YOLOv4	2021	CSPDarknet
YOLOR	2021	CSPDarknet
YOLOv5	2020	Modified CSP v7
YOLOv6	2022	EfficientRep
YOLOv7	2022	RepConvN
YOLOv8	2023	YOLO v8

Tab. 1.1: Tabulka porovnávající použité páteřní sítě u sítí typu YOLO [5].

1.2 Problematika trasování objektů

Detekce objektů je pouze jedním z mnoha problematik, které musí tato práce pokrýt. Dalším problémem je trasování objektů. S tím souvisí problém trasování objektů na dvou po sobě navazujících snímcích. Detektor neumožňuje přiřadit objekt z první fotografie k objektu z následujícího snímku, i když lidským okem je jasné, že se jedná o ten samý objekt. Tento problém je v práci řešen dopočítáváním vzdáleností mezi středy detekčních rámečků. Následně jsou k sobě přiřazeny objekty stejné skupiny s nejmenší vzdáleností. Nejedená se o nejlepší řešení, ale pro porovnání výsledků je to dostačující. Mnohem lepším řešením by bylo použití sofistikovanějšího nástroje, jakým je například **deep SORT**.

1.2.1 Deep SORT

První možností je hloubková asociace metriky známá taky pod názvem **deep SORT**. Jedná se o rozšíření prvotní verze SORT (Simple Online Realtime Tracker) o neuronovou síť, která má za úkol vyrovnat chybějící detekce. Základem této metody stále zůstává původní SORT, který se skládá z Kalmanova filtru. Tato samotná verze je již schopná provádět sledování v reálném čase. Implementace používá standardní verzi Kalmanova filtru, který vyhodnocuje rychlost objektu a odhaduje budoucí pozici objektu [24].

Deep SORT je moderním rozšířením původní metody SORT, která obsahuje převážně Kalmanův filtr. Přidává do architektury hloubkovou neuronovou síť pro extrakci příznaků pro identifikaci a sledování objektů v reálném čase. Tato neuronová síť pomáhá zlepšit přesnost trasování a dovoluje sledovat a rozpoznávat objekty i když mění vzhled a překrývají se.[25].

1.3 Problematika predikce pohybu

Předchozími kroky zajistily detekci objektů a jejich trasování. Dalším logickým krokem, který bude následovat, je predikce pohybu. Problematika predikce pohybu je velice obsáhlý pojem. Jeho metody je možné rozdělit podle různých parametrů. Především je to umístění pozorovatele, množství vstupních dat a potřebné výpočetní zdroje.

Aktuální problém vyvstávající u předpovědi pohybu objektů je, že jsou často reprezentované jako výseče, které udávají, kde se bude objekt v budoucnu nacházet[26]. Tyto předpovědi, pro naše účely detekce objektů v dopravních křižovatkách, jsou dostačující, přesto pro potřeby autonomního řízení automobilů nikoli. Jedná se o situaci, která může vést k častému zbytečnému brždění a zpomalení rychlosti, což by mělo efekt na plynulost jízdy[26]. Článek od *Dooseop Choi a KyoungWook Min* přichází s novými metodami, jak tyto výseče převést na množinu bodů značících předpokládaný pohyb [26], čehož by tato práce mohla využít například pro předpověď kolizí objektů.

Modely založené na **LSTM**, které generují predikce pohyby jako N nejpravděpodobnějších trajektorií se ukázaly jako výrazně přesnější než konvenční metody [27]. Také bylo zjištěno, že tento model je velice vhodný pro predikování změny směru vozidla v jízdních pruzích [28].

Zajímavou možností, jak predikovat směr pohybu u objektů je například ST-LSTM, který modifikuje strukturu **LSTM**. Především se zaměřuje na problém mizejícího gradientu, kvůli kterému je nemožné **LSTM** trénovat na dlouhé časové řady. Mizející gradient a nemožnost trénovat dlouhé řady způsobuje velké problémy při

predikci pohybu. Jedná se o modernější a přesnější model, který navíc pracuje s interakcemi mezi vozidly. Hlavním rozdílem oproti **LSTM** jsou spojení mezi vstupy a výstupy, tím je eliminováno mizení gradientu [29].

2 Srovnání detekčních neuronových sítí

Praktickou část této práce je možné rozdělit na jednotlivé na sebe navazující kroky. Prvním z nich je vytvoření datové sady, po té následuje spuštění trénovacích skriptů, které generují soubor s "váhami"(soubor definující natrénovanou neuronovou síť). Následným krokem je ověření přesnosti a paměťových nároků jednotlivých testovaných detekčních metod. Každý z těchto kroků je možné rozdělit na další podúkoly, které jsou často specifické podle dané neuronové sítě.

Pro detekci jsou použity neuronové sítě ze zadání práce. Jedná se o **EfficientDet** [30], **Scaled-YOLOv4** [31], **YOLOR** [32] a **YOLOv5** [20]. Vlastní tvorba neuronových sítí by byla časově náročná, proto jsou použity již vytvořené implementace, které bylo možné nalézt ve veřejných repositářích na stránce *www.github.com*. Obsah těchto repositářů byl zduplikován a vložen do repositáře tohoto projektu. Přesto je potřeba připomenout, že tyto implementace byly vystaveny pod určitými licencemi, jejíž podmínky je potřeba dodržet.

Při vývoji aplikací v jazyce Python dochází k častým problémům s verzemi modulů, proto je vždy potřeba přesně definovat běhové prostředí. Nejhorší je absence jakýchkoli informací o použité verzi Pythonu, verzích modulů a závislých dynamických knihoven. Proto tato práce klade velký důraz na správnou definici vývojového prostředí.

2.1 Definice prostředí

Vývoj je prováděn na operačním systému Ubuntu pomocí programovacího jazyku Python ve verzi 3.7.0. Pro instalaci a správu modulů projekt využívá *pip* ve verzi 20.3.3. Dalším modulem je *python-venv*, který umožňuje vytvořit virtuální vývojová prostředí. Potřebné základní moduly pro Python se instalují do Ubuntu pomocí příkazu, který je ve výpisu 2.1.

Výpis 2.1: Příklad instalace vývojového prostředí pro Ubuntu [6] [7] [8]

```
sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt install python3.7
sudo apt install python3-pip
sudo apt install python3.7-dev
sudo apt install python3.7-venv
```

Veřejně dostupné implementace není možné po nainstalování souboru se závislostmi (*requirements.txt*) spustit bez chyb. Důvodem jsou chybějící verze závislých

modulů, které nejsou v *requirements.txt* definovány. V některých případech dokonce chybí definice verzí i u použitých modulů. Z tohoto důvodu projekt obsahuje nové a funkční soubory se závislostmi. Tyto soubory obsahují kompletní výpis všech nainstalovaných modulů i těch, které nejsou použity. Jedná se o mnohem funkčnější variantu než v případě, kdy závislé moduly chybí. Soubor se závislostmi byl vygenerován pomocí následujícího příkazu, který je ve výpisu 2.2.

Výpis 2.2: Příklad exportu závislostí [9]

```
python3 -m pip freeze > requirements.txt
```

Instalaci vývojového prostředí je možné provést pomocí modulu *venv*. Takto vytvořené prostředí je následně potřeba aktivovat. Aktivní prostředí je vidět na terminálu, protože název aktivovaného prostředí se zobrazuje v kulatých závorkách před názvem uživatele a počítače. Po aktivaci prostředí je možné nainstalovat závislosti z vygenerovaného souboru *requirements.txt*. Tím se docílí, že moduly s určitými verzemi se nainstalují pouze v daném virtuálním prostředí. Deaktivaci prostředí je následně možné provést pomocí příkazu *deactivate*. Ukázka instalace, aktivace, instalace modulů a deaktivace je zobrazena na následujícím výpisu 2.3.

Výpis 2.3: Příklad práce s virtuálním prostředím

```
xvymaz11@tesla:~$ python3 -m venv prostredi                    1
xvymaz11@tesla:~$ source prostredi/bin/activate                2
(prostredi) xvymaz11@tesla:~$ pip install -r requirements.txt  3
(prostredi) xvymaz11@tesla:~$ deactivate                       4
xvymaz11@tesla:~$                                              5
```

Přestože použití novější verze Pythonu by mohlo také fungovat, je silně doporučeno použít stejnou verzi, jaká byla použita pro export, tedy 3.7. S každou verzí Pythonu dochází ke změně podporovaných modulů. V komerčních řešeních se využívají jiné metody pro definici závislostí, exportují se pouze ty moduly, které jsou opravdu použity pro daný projekt, a to vždy s příslušnou verzí. Definice *requirements.txt* bez uvedení patřičných verzí je často nespolehlivá a neexistuje žádné nekomerční řešení, které by bylo schopno vyexportovat pouze použité závislosti. Je potřeba využít jiný méně dokonalý nástroj. Tím je příkaz *pip freeze*, který vytvoří export všech nainstalovaných modulů a jejich verzí a pokryje tím veškeré nedostatky v původních *requirements.txt*. Veškeré potřebné soubory se závislostmi jsou uloženy ve složce *requirements* v přílohách.

2.1.1 Použité výpočetní zdroje

Detekce objektů není výpočet, který by se dal dělat na libovolném počítači. Nutnou výbavou by měl být stroj, který disponuje grafickou kartou NVIDIA. Výkonnost této

karty velice ovlivňuje rychlost detekce. Pro trénování je velká paměť grafické karty výhodou.

Tento výkonný stroj se nachází v interní síti VUT, proto je potřeba se nejprve připojit na VUT VPN a následně se do něj přihlásit pomocí SSH tunelu. Server disponuje výpočetními parametry 48 jader a 200 GB paměti RAM. Nejzásadnější jsou parametry grafických karet. Počítač obsahuje 3 totožné karty typu TESLA V100 s 32 GB grafické paměti. Výpis GPU podrobností, jako je například verze ovladače nebo verze CUDA je možné vidět na obrázku 2.1.

```
xvymaz11@tesla:~$ nvidia-smi
Tue Nov 29 16:52:43 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.141.03   Driver Version: 470.141.03   CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100S-PCI...  Off  | 00000000:25:00.0 Off |                    0 |
| N/A   29C    P0    24W / 250W |      8MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100S-PCI...  Off  | 00000000:81:00.0 Off |                    0 |
| N/A   34C    P0    38W / 250W |   5413MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100S-PCI...  Off  | 00000000:C1:00.0 Off |                    0 |
| N/A   28C    P0    26W / 250W |      8MiB / 32510MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1497      G   /usr/lib/xorg/Xorg                  4MiB |
|    1   N/A  N/A      1497      G   /usr/lib/xorg/Xorg                  4MiB |
|    1   N/A  N/A    613687      C   ...s/venv_pytorch/bin/python     5405MiB |
|    2   N/A  N/A      1497      G   /usr/lib/xorg/Xorg                  4MiB |
+-----------------------------------------------------------------------------+
```

Obr. 2.1: Výpis GPU parametrů školního serveru

2.2 Implementace

Prvním úkolem této práce je získat data, která jsou použita pro učení a testování. Z tohoto důvodu je implementována jednoduchá webová aplikace v jazyce Python. Tato aplikace umožňuje stahování datového proudu z platformy *youtube.com*. Tento datový zdroj je rozsekán na jednotlivé fotografie, a ty jsou následně jednotlivě zobrazovány na webové stránce. Tato aplikace disponuje automatickým ukládáním fotek

ve formátu JPG v časovém intervalu pěti minut. Takto získané fotografie se následně použijí pro další krok. Implementace tohoto rozhraní je přiložena v git repositáři tohoto projektu. Implementace se také nachází v přílohách ve složce *web_interface*.

Hlavním cílem této práce je srovnání jednotlivých neuronových sítí a jejich výsledků. Z tohoto důvodu bylo potřeba vytvořit program, pomocí kterého by bylo možné detekcí algoritmy porovnat. Proto byl implementován jednoduchý program v jazyce Python. Vstupními parametry jsou: cesta do složky, ve které jsou uloženy jednotlivé výsledky z detekcí, lokace složky s trénovacími daty a textový soubor, který obsahuje definice názvů fotek, které patří do testovací množiny. Program je umístěn v příloze ve složce *compare_results* a jmenuje se *compare.py*. Před spuštěním je potřeba si nejprve vytvořit virtuální prostředí, nainstalovat závislosti, a až poté je doporučeno program spustit. Pokud nejsou definovány výše zmíněné parametry, budou použity výchozí hodnoty.

Výstupem tohoto programu jsou grafy, které jsou vytvořeny pomocí modulu *mathplotlib*, který umožňuje generovat grafy. Program obrázky sám neukládá. Pro export je potřeba tyto grafy uložit jako fotografie, a to pomocí předchystaných možností v dialogovém okně.

2.2.1 Příprava vlastní datové sady

Prvním krokem je nasbírání dostatečného množství fotografií, na kterých se nachází objekty, které jsou použity pro trénování detekčních modelů. Za tímto účelem je vytvořena webová aplikace, která sbírá fotografie z brněnské křižovatky (ulice Koliště a Milady Horákové). Dalším krokem bylo označení objektů ve fotografiích. K tomu byl požit program *Label Studio*. [33]

Program *Label Studio* je vystaven pod licencí Otevřený Software (Open Source). Slouží k označování jednotlivých objektů ve fotografiích. Pro vývoj byla tato aplikace spuštěna jako *Docker* kontejner s perzistentním diskem. Díky tomu nedojde ke ztrátě dat po vypnutí kontejneru. Značnou nevýhodou tohoto programu je neintuitivní prvotní nastavení, ve kterém je potřeba nastavit formát ukládaných dat a definice skupin objektů. Následné označovaní objektů je již velice snadný proces. Je třeba provést kontrolu, jestli objekt opravdu koresponduje s označeným objektem. Občas se stává, že uživatel omylem označí již vytvořený obdélník a klikne na tlačítko skupiny, čímž změní skupinu již vytvořeného objektu. Přesnost tohoto kroku je důležitá, protože kvalita označených objektů je zásadní jak pro proces trénovaní, tak pro proces ověření přesnosti.

Detekce rozlišuje tyto objekty:

- auto,
- vlak (tramvaj),
- kolo,
- motorka.

2.2.2 Rozdělení datové sady

Ve všech neuronových sítích, kde probíhá proces trénovaní, je potřeba rozdělit datovou sadu na dvě až tři množiny. Jedná se o množinu trénovací a testovací, popřípadě validační. Pro tento účel je naimplementován jednoduchý program, který umí data rozdělit podle nastavených parametrů. V řešeném případě je zvoleno 20% množiny pro testování, 5% pro validaci a zbylých 75% pro trénovací sadu.

Tento program je napsaný v Pythonu a jeho výstupem je textový soubor v **YAML** formátu, který je potřebný pro spuštění většiny implementací. Příklad výsledného souboru je ukázán v následujícím výpisu 2.4. Dalšími výstupy jsou textové soubory *train.txt*, *validate.txt* a *test.txt*, ve kterých jsou definovány cesty k fotkám.

Výpis 2.4: Ukázka vstupního souboru pro neuronovou síť YOLOv5

```
                                                    1
path: ../datasets/cross1-test1                      2
train: train.txt                                    3
val: validate.txt                                   4
test: test.txt                                      5
names:                                              6
  0: bike                                           7
  1: bus                                            8
  2: car                                            9
  3: train                                          10
```

Takto získaný soubor je potřeba nakopírovat do složky, ve které se nachází daná implementace neuronové sítě. Bohužel některé sítě vyžadují další dodatečné informace, které je potřeba do tohoto souboru doplnit. Pro zjednodušení je dobré si prohlédnout, jak vypadají soubory, které byly použity pro trénování, a podle nich doplnit, co v souboru chybí nebo co je potřeba zadefinovat jinak. Tyto soubory jsou pojmenovány jako *cross1-test1.yaml*.

Naprostou výjimkou je neuronová síť *EfficientDet*, která nepracuje s datovou sadou YOLO, ale s datovou sadou ve formátu JSON. Proto je pro převod z jedné datové sady na druhou použit projekt *Yolo-na-COCO* (Yolo-to-COCO) [34]. Při použití tohoto programu je potřeba držet se jednotlivých kroků, které jsou uvedeny na stránce projektu.

2.2.3 Proces trénovaní datové sady

Tento bod vývoje je potřeba rozdělit na dvě části. Těmi jsou příprava datové sady a proces trénování. Kvalita datové sady je v tomto případě velice rozhodující, proto se dbá zvýšené pozornosti při její přípravě. Je běžnou praxí, že příprava dat pro neuronovou síť je časově nejnáročnější proces.

Důležitou informací je, jaký formát datové sady očekává neuronová síť. Po pár experimentech bylo zjištěno, že data z *Label Studia* musí být vyexportována ve formátu YOLO. Některé implementace detekčních modelů vyžadují definici parametrů, které nejsou součástí YOLO formátu. Proto je nutné tyto parametry doplnit ručně. Jedná se například o informaci, kolik typů objektů má model podporovat. Jde o duplicitní hodnotu, protože ta se musí shodovat s velikostí slovníku objektů, viz výpis 2.4.

Neuronové sítě typu YOLO potřebují jako vstupní parametr cestu k YAML souboru, ve kterém jsou definovány další parametry, především textové názvy skupin, cesty k textovým souborům, ve kterých jsou definovány lokace do jednotlivých datových množin (testovací, trénovací a validační).

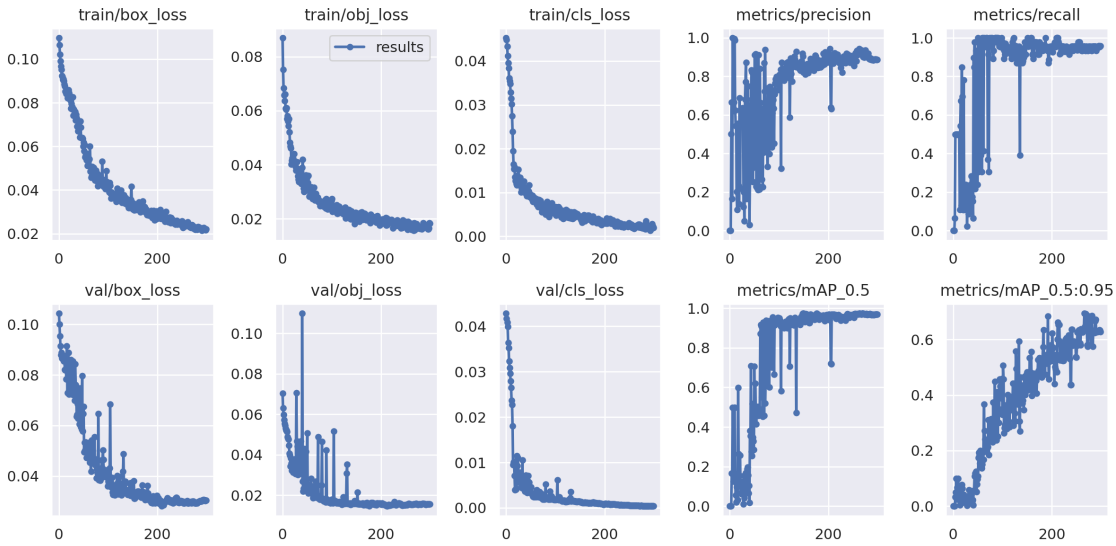
Trénování je možné spustit pomocí následujících příkazů, výpis 2.5. Pro ukázku je zde uvažováno se spuštěním trénování neuronové sítě YOLOv5.

Výpis 2.5: Script pro trénování neuronové sítě typu YOLOv5

```
#!/bin/bash

source vyma-yolov5-env/bin/activate

cd yolov5
python3 train.py    --imgsz 1280 \
                    --data cross1-test1.yaml \
                    --cfg models/yolov5s.yaml \
                    --weights '' \
                    --batch-size -1 \
                    --device 0

deactivate
```

Výsledkem trénování je soubor s váhami (*weights.pl*). Součástí některých implementací je export grafů, které popisují změnu parametrů během trénování, viz obrázek 2.2. Díky těmto grafům se dá optimalizovat počet epoch, které jsou nutné pro trénování. Z grafu je patrné, že charakteristiky mají exponenciální průběh. Proto nemá smysl trénovat déle, než je potřeba. S parametrem počet epoch se v projektu

neexperimentovalo a zůstal nastavený na 300 epochách. Jedinou výjimkou byla síť EfficientDet, kde bylo použito 500 epoch.



Obr. 2.2: Graf popisující změnu parametrů během trénování pro neuronovou síť Scaled-YOLOv4

2.2.4 Proces validace výsledků

Validace výsledků práce je rozdělena do dvou kategorií. Jedná se o subjektivní zhodnocení výsledků a o výstup z validačního skriptu. Po dokončení trénovaní program spustí validační funkci a provede detekce na validní množině fotek. V tomto případě to znamená, že se vloží do obrázku obdélníky s detekcemi. Tyto objekty jsou často doplněny o informaci, o jakou skupinu objektů se jedná a na kolik procent si je jistý. Takto získané informace jsou porovnány pouze subjektivně. Přesto jsou tyto výstupy velice užitečné. Validační výsledek vytvořený po natrénovaní Scaled-YOLOv4 je vyobrazen na obrázku 2.3.

2.3 Výsledky experimentu

Vypočítaná objektivní porovnání jednotlivých neuronových sítí mají jednu nevýhodu. Vždy nám umožňují porovnat jen několik parametrů. To nám může dát určitou informaci, která metoda je například rychlejší a nebo přesnější. Porovnáním všech testovaných parametrů je možné říci, že první síť je lepší než druhá.

Obr. 2.3: Detekované objekty na validační sadě, pro neuronovou síť Scaled-YOLOv4

2.3.1 Metoda vyhodnocování výsledků

Volitelným výstupem detekčního programu je textový soubor. Spuštění této funkcionality musí být zapnuto při detekci. U některých neuronových sítí se nastavuje ještě jeden parametr, který do souboru přidává informaci o procentuálním ohodnocení detekovaného objektu. Tento parametr představuje procentuální hodnotu, s jakou se jedná o hledaný objekt. Dalšími výstupy můžou být například fotky s označením detekovaných objektů, atd.

Pro následné zpracování jsou nejdůležitější textové soubory. Jména souborů musí korespondovat s názvy vstupních obrázků, jen s tím rozdílem, že mají příponu textového souboru TXT. Struktura tohoto souboru je velice jednoduchá, každý řádek značí detekovaný objekt. Na každém řádku se nachází 6 čísel oddělených mezerami. První číslo na řádku je z množiny celých čísel a znamená skupinu detekovaného objektu. Dále se zde nachází 5 čísel z množiny reálných čísel. Tyto hodnoty udávají v pořadí: střed objektu v ose x, střed objektu v ose y, velikost objektu v ose x, velikost objektu v ose y a pravděpodobnost, na kolik si je neuronová síť jistá, že se jedná o daný objekt. Hodnoty pozic a velikostí jsou udány v procentuálním zápisu, proto je pro převod na velikost nebo pozici potřeba znát rozměry vstupních obrázků. V tomto případě se jedná o velikost 1280 na 640 pixelů.

Programu pro zpracování výsledků je potřeba nastavit cestu do složky s výsledky. V této složce by se měly nacházet další složky, které jsou pojmenovány podle neuronové sítě, kterou byly vytvořeny. Názvy složek jsou při zpracování použity jako

názvy detekčních metod. V těchto složkách musí být umístěny textové soubory, které byly vytvořeny během detekce.

Dalším parametrem musí být cesta do složky, ve které se nachází soubory, které byly vytvořeny během přípravy datové sady. Názvy souborů se musí, jako v předchozím případě, shodovat s názvy fotek. Struktura těchto souborů je stejná se strukturou souborů z detekce, jen s tím rozdílem, že na řádcích v těchto souborech chybí poslední hodnota udávající přesnost detekce.

Pro vizualizaci naměřených výsledků byl použit modul *mathplotlib*. Jednotné grafy byly vytvořeny pomocí návodů na oficiálním webu tohoto modulu. [35] [36] [37]

2.3.2 Vyhodnocení prostorové přesnosti detekce

Pro zjednodušení výpočtu je potřeba prostorovou přesnost rozdělit do dvou skupin. První z nich porovnává, jak daleko je střed detekovaného objektu od označeného objektu. Druhou je porovnání velikosti detekovaných objektů. Je nutné dodat, že metoda nezohledňuje, jestli detekovaný objekt je ve stejné skupině jako označený, a už vůbec se nepřihlíží k procentuálnímu parametru, který určuje přesnost detekce.

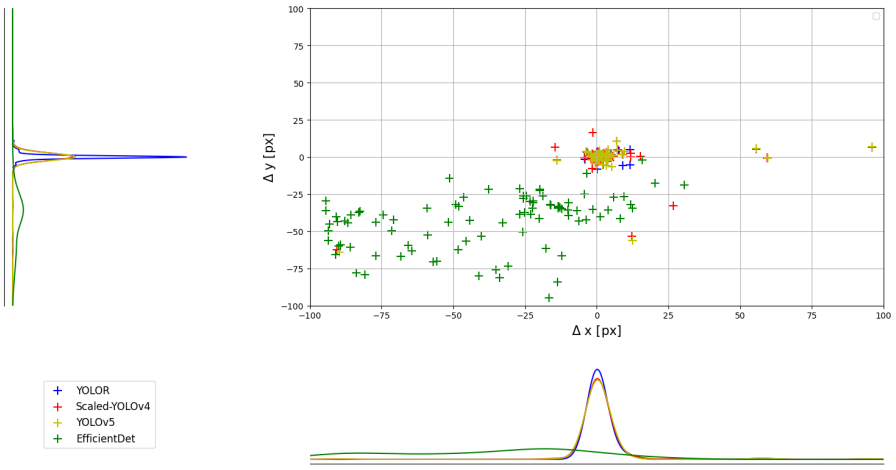
Problémem, který je potřeba vyřešit, je přidělení objektů z detekce k objektům vytvořeným ručně v programu *Label Studio*. K tomu je použit jednoduchý algoritmus, který každý objekt převede na jeden bod, v tomto případě je to střed objektu. Následně prochází všechny objekty z detekované množiny a hledá objekt, který je k němu nejblíže v množině ručně vytvořených. Po nalezení nejbližšího objektu algoritmus zkontroluje, jestli jsou objekty stejného typu. Pokud ano, výsledek je zaznačen do grafu. Tato metoda dostačuje pro měření přesností, ale pro následné trasování není vhodná.

Na prvním obrázku 2.4, který udává rozdíl vzdáleností v pixelech mezi ručně vytvořeným a detekovaným objektem v obou osách, lze vidět, že sítě typu YOLO jsou velice přesné. Tento subjektivní závěr potvrzují Gaussovy křivky, které popisují rozložení daných rozdílů. Nejhůře je na tom síť *EfficientDet*, u které se dá pozorovat, že detekované objekty jsou výrazně posunuty od středu a mají velký rozptyl (malá strmost Gaussovy křivky).

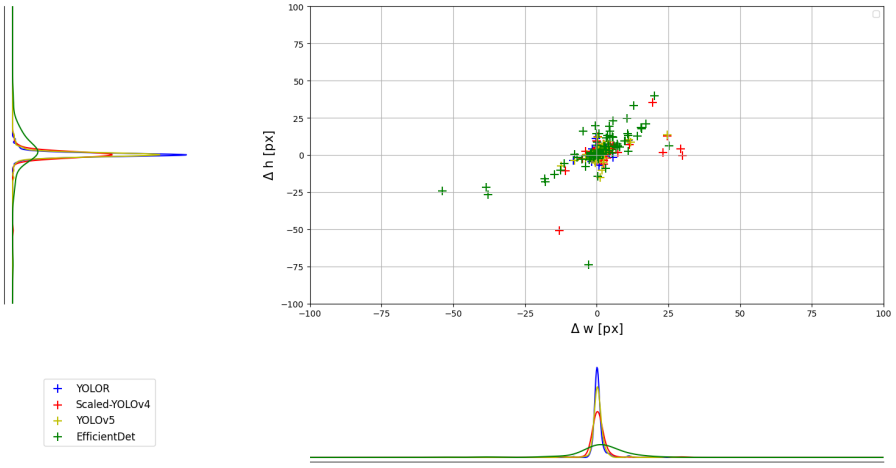
Při porovnání obrázku 2.5, popisující rozdíl velikosti objektů, si lze povšimnout, že síť *EfficientDet* už tak velké problémy s přesností nemá, přesto dopadla nejhůře.

2.3.3 Vyhodnocení rychlosti detekce

Měření rychlosti detekce není jednoduché. Rychlost je ovlivněna mnoha faktory, především záleží na dané implementaci, na použitém hardwaru a velkosti obrázků. Přesto jsou tyto parametry měřeny a dají se snadno dohledat. Nejčastěji jsou součástí



Obr. 2.4: Graf naměřené prostorové přesnosti středu detekovaných objektů v závislosti na zvolené neuronové síti



Obr. 2.5: Graf naměřené prostorové přesnosti rozměrů detekovaných objektů v závislosti na zvolené neuronové síti

dokumentace k dané neuronové síti a demonstrují, o kolik je daná implementace rychlejší než ostatní. Tyto výsledky jsou nejčastěji měřeny na datové sadě COCO [30][32][31], která zaručuje, že měření bylo provedeno na stejných datech. Častým parametrem, který se pak následně udává, je rychlost detekce jednoho obrázku. Tyto obrázky jsou nejčastěji ve velikosti 640 na 640 pixelů.

V práci je použit jiný postup pro porovnání rychlosti detekce. Rychlost detekce

jednoho obrázku je velice důležitý parametr, který byl v práci změřen. Ovlivňuje výslednou celkovou rychlost detekce. Mnohem důležitějším parametrem je rychlost celé detekce, která zohledňuje i režie potřebné pro danou implementaci. Z tohoto důvodu byl změřen i čas vykonávání celé detekce, tedy množiny 195 fotek. Tato metoda zohledňuje také například to, jak daná implementace pracuje s pamětí. Pro měření byl opět použit školní server a výpočty byly provedeny na jednom z jeho GPU.

Režie detekce byly vypočítány podle následujícího vzorce 2.1, kde R je režie detekce, N počet fotografii (195 ks), V_{Det} je rychlost detekce jednoho obrázku a V_{Imp} je rychlost celé implementace.

$$R = V_{Imp} - (V_{Det} * N) \tag{2.1}$$

Takto vypočtené a změřené parametry jsou udány v následující tabulce 2.1. Pro výpočet rychlosti detekce jednoho obrázku byl zvolen průměr z naměřených hodnot detekcí, podrobnější udaje je možné nalézt v přiloženém excelovém souboru, viz přílohy. Rychlost implementace byla změřena pomocí Python modulu *scalene*. Záznamy z měření jsou umístěny v gitu a v přílohách ve složce *runs*.

Neuronová síť	Rychlost implementace [s]	Rychlost detekce [ms]	Režie detekce [s]
EfficientDet	50,27	33,33	43,77
Scaled-YOLOv4	24,85	37,68	17,50
YOLOR	20,06	27,30	14,74
YOLOv5	16,22	6,91	14,87

Tab. 2.1: Tabulka porovnávající rychlost detekce

Zvolená metoda je inovativní a lépe demonstruje časové nároky potřebné na detekci. Z výsledků vyplývá, že čas potřebný na jednu detekci není hlavním hodnotícím kritériem, který je potřeba v této kategorii sledovat. Především co se týče implementace *EfficientDet*, kde 87% času zabírají pouze režie potřebné pro detekci.

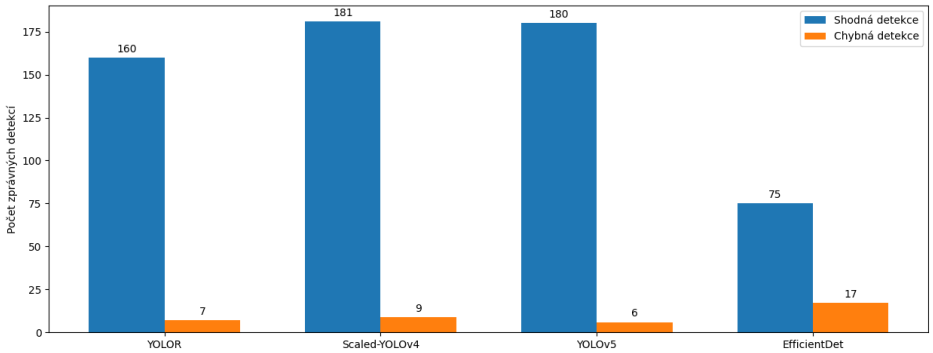
Závěrem objektivního pozorování je, že při porovnání časové náročnosti je na tom nejlépe síť *YOLOv5* především proto, že má velice rychlou detekci a nízké režie.

2.3.4 Vyhodnocení přesnosti rozpoznávání objektů

Validace výsledků byla provedena pouze na testovací množině fotek. Výstupem tohoto testování byla sada textových souborů, které bylo potřeba zpracovat. Zpracované výsledky jsou zobrazeny na následujících obrázcích 2.6, 2.7. Důležitou podmínkou proto, aby byly výsledky srovnatelné, je použití stejných parametrů při spuštění detekčních algoritmů. Především se jedná o parametr, který se může přeložit jako

hranice detekce (*threshold*). Tímto parametrem je možné nastavit minimální hodnotu jistoty detekce potřebnou pro to, aby byl detekovaný objekt zapsán do textového souboru.

První sloupec v obrázku 2.6 popisuje celkový počet detekovaných objektů, ke kterým byla správně přiřazena skupina objektů. Druhý sloupec v obrázku 2.6 udává počet špatných detekcí, tedy hodnotu, kdy se skupiny neshodovaly. Tato metoda zanedbává informaci o tom jak moc si je neuronová síť jistá, že se jedná o daný objekt nebo jak prostorově přesně je objekt detekován. Tyto informace jsou velice důležité, protože nám dávají představu o tom, jak citlivá detekce je u jednotlivých neuronových sítí. Objektivním vyhodnocením byl učiněn závěr, že v tomto porovnávaném aspektu si nejlépe vedou neuronové sítě typu YOLO, především novější implementace tedy *YOLOv5* a *Scaled-YOLOv4*.

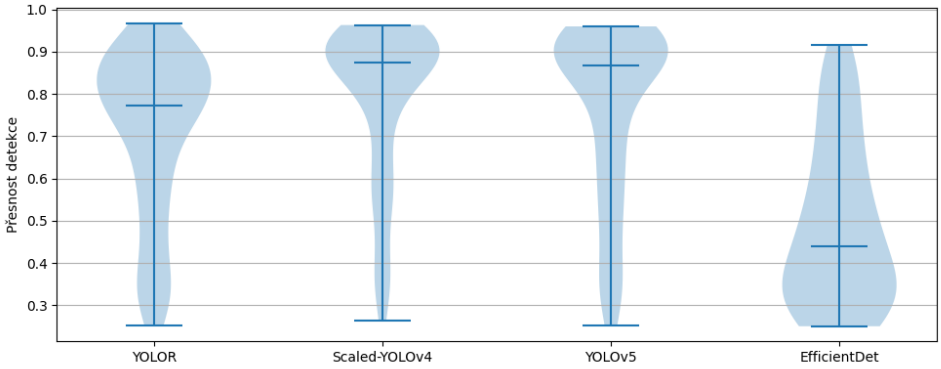


Obr. 2.6: Množství detekovaných objektů v závislosti na zvolené neuronové síti

Obrázek grafu 2.7 neporovnává množství detekovaných objektů, ale zaměřuje se na jistotu detekce neboli parametr, který nám udává s jakou jistotou se jedná o detekovaný objekt. Struktura grafu byla schválně vybrána tak, aby byly zobrazeny i mediány měřených dat. Tyto hodnoty byly vytvořeny pouze z jistot detekce, u kterých neuronová síť správně přiřadila skupinu objektů. Objektivním porovnáním byl učiněn závěr, že i v tomto případě jsou na tom neuronové sítě typu YOLO nejlépe.

2.3.5 Souhrn naměřených výsledků

Porovnání naměřených výsledků je provedeno pouze na testovací množině fotek. Výjimku tvoří měření rychlosti detekce a použité grafické paměti. Jedná se o případy, kdy obsah fotek je irelevantní k měřeným výsledků. Tyto parametry jsou především ovlivněny počtem zpracovaných obrázků, proto bylo příhodné tyto parametry testovat na co největším počtu snímků.



Obr. 2.7: Přesnost detekce v závislosti na zvolené neuronové síti

Parametrem, kterému zde není věnován žádný odstavec, je množství použité paměti na GPU při detekci. Přesto jsou tyto hodnoty uvedeny v následující tabulce 2.2. V souhrnné tabulce 2.2 jsou zobrazeny všechny výsledky, které byly získány v této práci. V každé hodnocené kategorii je zvýrazněna hodnota, která je nejlepší v daném srovnání. Po krátkém prozkoumání tabulky je patrné, že nejlépe se umístily sítě typu YOLO. Nejhorší výsledky má implementace *EfficientDet*.

Naměřené výsledky jsou zpracovány pomocí skriptu napsaného v Pythonu, který je zmiňován v kapitole "Metoda vyhodnocování výsledků". Pro jednodušší implementaci byly tyto výsledky vypsány do konzole a následně opsány do tabulky 2.2. Data o časové náročnosti jsou vyhodnocena v excelové tabulce, která je součástí příloh. Množství použité paměti na GPU bylo získáno z výstupu modulu *scalene*, jehož výpisy jsou uloženy v repositáři s v přílohách ve složce *runs*.

Porovnávaný parametr	**EfficientDet**	**Scaled-YOLOv4**	**YOLOR**	**YOLOv5**
Verze sítě	s efficientnet-b0	yolov4-p6	yolor-p6	yolov5s
Rychlost detekce na množině fotek [s]	50,27	24,85	20,06	**16,22**
Průměrná rychlost detekce jedné fotky [ms]	33,33	37,68	27,30	**6,91**
Režie detekce [s]	43,77	17,50	14,74	**14,87**
GPU pamět [GB]	2,53	2,41	2,32	**2,08**
Medián vzdáleností středů v ose X [px]	-26,84	**0,04**	0,06	0,06
Medián vzdáleností středů v ose Y [px]	-38,6	0,05	**-0,01**	**-0,01**
Medián rozdílů velikostí objektů v ose X [px]	1,19	0,31	**0,17**	0,29
Medián rozdílů velikostí objektů v ose Y [px]	2,04	0,11	**0,06**	0,21
Medián jistoty detekce [%]	45,46	**87,57**	77,49	86,69
Počet správně detekovaných objektů [ks]	75	**181**	160	180
Počet špatně detekovaných objektů [ks]	17	9	**7**	9

Tab. 2.2: Tabulka porovnávající výsledky testovaných neuronových sítí

3 Nasazení algoritmu pro trasování objektů

Jedním z cílů diplomové práce je vykreslit trajektorie pohybu jednotlivých objektů. K vykreslení trajektorie je potřeba algoritmu, který je schopný přiřadit objekty z dvou po sobě jdoucích snímkům k objektům v následujícím snímku. K dosažení tohoto cíle je možné využít mnohé přístupy. První z nich je použit při porovnání parametrů jednotlivých detekčních modelů. Jedná se o případ jednoduchého přiřazení vycházejícího z hledání nejmenší vzdálenosti mezi středy objektů v dvou po sobě následujících snímcích. Dalšími možnostmi jsou metody založené na neuronových sítích, jako například ***deep SORT*** nebo ***ByteTrack***.

Deep SORT je metoda vhodná pro sledování více objektů. Využívá kombinaci algoritmů pro detekci objektů a sledování. Hlavním stavebním blokem tohoto modelu je *Kalmanův filtr*, který se používá pro predikci pohybu. Díky tomu je ***deep SORT*** vhodný pro sledování více objektů, například vozidel v křižovatce.

Oproti tomu ***ByteTrack*** využívá **RPN** (region proposal network), který provádí detekci v jednom místě snímku. Následně vytváří ***ByteTrack*** stínový model objektu a tím umožňuje sledování. Tato metoda je vhodná pro sledování jednoho objektu, například pro sledování pohybu právě jednoho vozidla.

První metoda, která byla využita pro vyhodnocení, je velice vhodná pro účely měření, protože se vyznačuje jednoduchým použitím a není výpočetně náročná a její výsledky budou vždy deterministické. Ovšem pro účely trasování je tato metoda nedostačující. Proto se práce dále zaměřuje na trasování objektů pomocí neuronových sítí.

3.1 Nasazení trasovacího algoritmu

Z teorie vyplývá, že je logické se hned zaměřit na metodu ***deep SORT***, protože je vhodná pro trasovaní více objektů najednou. Výsledky s použitím ***ByteTrack*** [38] byly objektivně výrazně horší než výsledky dosažené pomocí ***deep SORT***.

Rozhodujícím faktorem pro získání validních výsledků v metodě ***deep SORT*** je optimalizace parametrů, modelu. Práce používá následující hodnoty při volání konstruktoru ***deep SORT***. Který je vidět na výpisu 3.1. Nejdůležitějšími parametry jsou *max_age*, *n_init* a *embedder*. Parametr *embedder* nejvíce ovlivňuje následnou přesnost trasování. Tímto parametrem se nastavuje neuronová síť, která bude použita pro zpracování sledovacích parametrů objektu. Nastavení *embedder=mobilenet* je výchozí hodnota parametru.

Pro použitou implementaci ***deep SORT*** [39] a její parametr *embedder* je možné zvolit jeden z následujících modelů:

- mobilenet,
- torchreid,
- clip_RN50,
- clip_RN101,
- clip_RN50x4,
- clip_RN50x16,
- clip_ViT-B/32,
- clip_ViT-B/16.

Pro použití jiných hodnot je potřeba doinstalovat další závislé moduly, které se nepovedlo dohledat a nebo doinstalovat na danou verzi Python.

Výpis 3.1: Vytváření instace deepSort s nastavenými parametry

```
self.tracker = DeepSort(
    max_iou_distance=1,
    max_age=10,
    n_init=3,
    nms_max_overlap=1.0,
    max_cosine_distance=0.2,
    nn_budget=None,
    gating_only_position=True,
    override_track_class=None,
    embedder="mobilenet",
    half=True,
    bgr=True,
    embedder_gpu=use_cuda,
    embedder_model_name=None,
    embedder_wts=None,
    polygon=False,
    today=None
)
```

Hodnota parametru *max_age* určuje, jak dlouho má objekt zůstávat v paměti trasovače. Proměnnou *n_init* je možné nastavit, kolik záznamů je potřeba pro inicializaci trasovaného objektu, neboli kolikrát musí být objekt detekován, aby ho trasovač začal trasovat.

Trasování objektů je možné pomocí volání metody *update_tracks*, viz výpis 3.2. Metoda očekává jako vstupní parametry datovou strukturu, ve které jsou uloženy

detekované objekty a obrázek, na kterém byla tato detekce provedena. Datová struktura musí být datový typ list. Každý prvek listu je jedna detekce, která je tvořena uspořádanou n-ticí. Na prvním místě n-tice je list s detekcemi (střed v ose x, střed v ose y, velikost objektu v ose x a velikost objektu v ose y). Tyto hodnoty musí být typu **int**. Na druhém místě je hodnota přesnosti detekce, která je typu **float**. Posledním prvkem je hodnota znamenající typ objektu, která má datový typ **int**.

Výpis 3.2: Volání metody pro trasování objektů

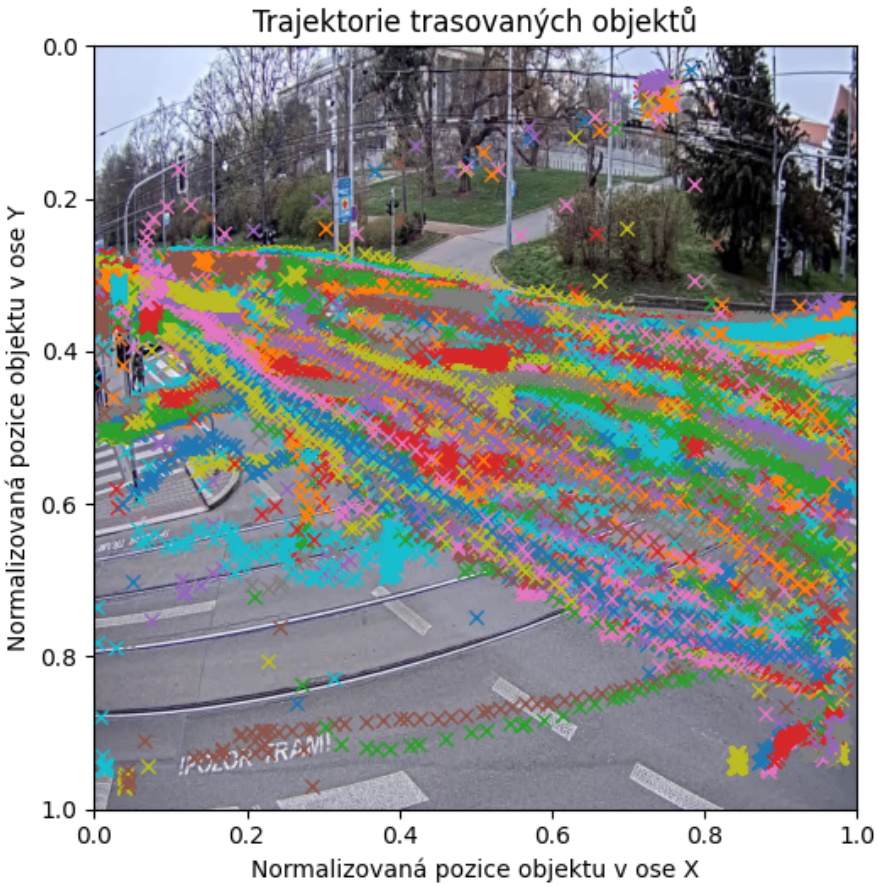
```
# bbs: List[(List[int], float, int)]

tracks = self.tracker.update_tracks(bbs, frame=im0)
```

Návratovou hodnotou je datová struktura, která je složena z listu, ve kterém jsou uloženy instance tříd pro reprezentaci objektu. Proto je následně potřeba vyfiltrovat záznamy, které jsou pro trasování nepoužitelné. Toho může být dosaženo pomocí kontroly, jestli je objekt platný a jestli nemá nulovou přesnost detekce. V ***deep SORT*** dochází k velké ztrátě informací, obzvláště co se přesnosti a falešných detekcí týče. Dalším faktorem, který výrazně ovlivňuje použitelnost trasovače, je přesnost předchozího detektoru. Především se ukázalo, že model vytvořený pro porovnání parametrů detekčních modelů není moc spolehlivý. Pro trasování byl použit již natrénovaný model pro YOLOv5 (yolov5s6).

3.2 Detekované trasy objektů

Pro ověření funkčnosti trasování je potřeba vytvořit záznam z dopravní kamery, který je dlouhý 21 minut a 53 sekund. Tento záznam je později v práci také použit jako zdroj trénovacích dat. Následně je potřeba nad tímto záznamem provést detekce spolu s trasovacím algoritmem. Takto získané informace jsou následně uloženy do souboru s názvem *dataset_train.csv*. Tento soubor obsahuje 268 457 detekcí objektů a trasu 5 921 objektů. Počet objektů není reálný, velká část je tvořena falešnými detekcemi. Pro následnou vizualizaci jsou data z tohoto souboru filtrována. To znamená ignorování všech tras, které mají méně vzorků než 100. Po filtraci zůstane pouze 720 objektů.

Na obrázku 3.1 je vidět graf s pozadím snímané křižovatky. Do tohoto grafu jsou vyznačeny vyfiltrované detekované trasy vozidel. Toto zobrazení pracuje pouze se středy objektů, nikoli s velikostí objektu. Z grafu 3.1 je patrné, že ne všechny trasy dávají smysl především co se týče shluku bodů v pravém horním rohu. Zde se jedná o falešné detekce způsobené trasovacím algoritmem ***deep SORT*** a nebo detekční metodou.



Obr. 3.1: Graf s detekovanými trasami objektů.

4 Implementace predikce pohybu objektů

Cílem návrhu predikčního modelu bylo docílit co nejnižší výpočetní náročnosti. Proto se práce zaměřuje pouze na řešení, která nejsou nijak složitá.

Podmínkou pro další zpracování je schopnost trasovat objekty mezi jednotlivými snímky. Pro tento účel není možné použít ten samý algoritmus jako při ověřování přesnosti, protože pro další zpracování je potřeba mít co nejpřesnější výsledky. Hlavní předností takového algoritmu by měla být na prvním místě vysoká přesnost, která zaručí, že na dvou po sobě jdoucích snímcích se nachází opravdu ty samé objekty.

Úplné kódy z následujícího výzkumu jsou součástí příloh. Je možné je nalézt ve složce *prediction_research*. Tento vývoj byl prováděn na počítači, který není osazen **GPU**. Z tohoto důvodu nejsou kódy optimalizovány pro výpočty na grafické kartě.

4.1 Výzkum zkoumající využití jednoduchých neuronových sítí pro predikci

Práce se v prvních fázích zaměřovala na možnost využití modelu pro predikování pohybu. Především se zaměřuje na jednoduché modely a zkoumá, jestli by bylo možné nějaké z těchto modelů využít v praxi. Jedná se o stavební bloky neuronových sítí LSTM (*Long short-term memory*), GRU (*Gated recurrent unit*) a RNN (*Recurrent neural network*). Přestože již v teoretické části bylo zjištěno, že pro predikce se hojně používají složitější modely, je potřeba tuto možnost prozkoumat. Zástupcem složitějšího modelu je například ST-LSTM [29], který vypadá slibně, co se predikcí pohybu týče.

Následující modely je možné rozdělit podle toho, jak pracují s daty. První skupinou jsou neuronové sítě zpracovávající sekvenci dat. Jedná se o modely **LSTM**, **GRU** a **RNN**, které pracují pouze s jednou osou souřadnic a pohlíží na ní jako na časovou posloupnost. Zástupcem druhé skupiny je **Polynomiální regresní model** kde neuronová síť pohlíží na problematiku trochu jinak. Podstatou je neuronovou síť naučit přiřazovat pozici v ose **y** ke zvolené souřadnici na ose **x**.

První skupina modelů predikuje budoucí pozici podle předchozí hodnoty predikce, k tomu účelu byla vybrána trajektorie souřadnice osy **x**. Pro trénování je použit objekt s ID 36 a pro testování byl vybrán objekt s ID 39. Soubor, ze kterého je možné načíst tyto data, se jmenuje *dataset_train.csv*. **Polynomiální regresní model** využívá stejná trénovací data jako ostatní modely. Přesto je potřeba načítaná data reprezentovat v jiném formátu.

Tato studie využívá k trénování a testování dvě jedinečné trasy objektů (ID 36 a 39), které spadají do stejného směru jízdy dvou unikátních vozidel. Jedná se o situaci, kdy je možné vizualizovat pohyb dvou vozidel na tom samém jízdním pruhu.

4.1.1 LSTM po jednotlivých vzorcích

LSTM je druh paměti neuronové sítě. Často se používá pro predikování hodnot. Typickým příkladem je předpověď periodických jevů nebo předpověď pohybu cen na trhu. Nejčastějším použitím je situace, kdy síť je trénována na velkém množství předchozích hodnot časové řady za účelem získání nejbližších budoucích hodnot posloupnosti.

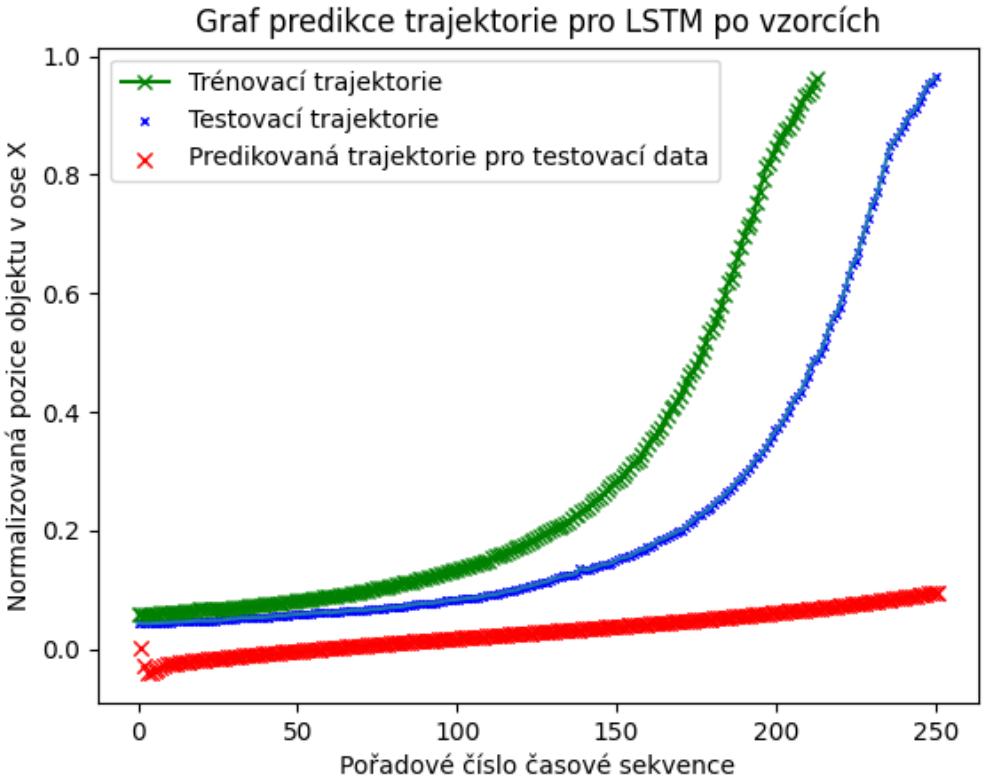
Vytvořený model, který je vidět na výpisu 4.1, používá dvě vrstvy. Vstupní vrstva je **LSTM** s parametry *input_size* = 1 a *hidden_size* = 10. Poslední vrstvou je **Linear**, která má parametry *hidden_size* = 10 a *output_size* = 1. Jedná se o velice jednoduchý model. Výstupní stavy **LSTM** vrstvy jsou reprezentovány proměnnou *hidden_prev*, která je jak vstupním tak výstupním parametrem v metodě *forward*. Za účelem otestování byla tato síť trénovaná 6000 epoch.

Výpis 4.1: Definice modelu pro LSTMperSample.

```
                                                                    1
class LSTMperSample(nn.Module):                                     2
    def __init__(self):                                             3
        super(LSTMperSample, self).__init__()                       4
                                                                    5
        self.lstm = nn.LSTM(input_size=input_size,                  6
                            hidden_size=hidden_size,                7
                            num_layers=1, batch_first=True)         8
        self.linear = nn.Linear(hidden_size, output_size)           9
                                                                    10
    def forward(self, x, hidden_prev):                              11
        out, hidden_prev = self.lstm(x, hidden_prev)                12
        out = out.reshape(-1, hidden_size)                          13
        out = self.linear(out)                                      14
        out = out.unsqueeze(dim=0)                                  15
        return out, hidden_prev                                     16
```

Na obrázku 4.1 je možné porovnat, jak vypadá predikovaná trajektorie pomocí **LSTM**. Zelené body symbolizují průběh trajektorie, která byla použita pro učení modelu. Modré body ukazují trajektorii vozidla, které bylo použito pro testování modelu. Predikované hodnoty jsou zobrazeny červeně. Z predikované trajektorie také

vyplývá že predikce nebyla příliš úspěšná. Je to především způsobeno neinicializovanou proměnnou pro výstupní stavy **LSTM** vrstvy. Této vlastnosti je možné si všimnout hned při predikci prvních dvou bodů, kdy se navrácené hodnoty výrazně liší od skutečné polohy objektu. Dalším problémem u tohoto modelu je skutečnost, že je po modelu požadována predikce jednoho budoucího bodu a to pouze ze znalosti předchozí souřadnice. Na takové použití ale **LSTM** konstruovaná není. Teoreticky by **LSTM** mohla být úspěšná pokud by do ní vstupovalo více předchozích souřadnic objektů. To by mělo za následek, že predikce by byla možná až po delším čase. V tomto případě by se mohlo stávat, že trasování by bylo možné až po průjezdu křižovatkou, tudíž takové použití nedává smysl.



Obr. 4.1: Ukázka predikce trajektorie pomocí LSTM po jednotlivých vzorcích.

4.1.2 GRU

Jedná se o obdobný model jako v případě **LSTM**. Rozdílem je, že byla nahrazena vrstva **LSTM** za **GRU**. Paměť v **GRU** je považována ze méně robustnější než v **LSTM**. Zásadním rozdílem oproti **LSTM** je definice paměti pro skryté výstupní stavy. Rozdíl v definicích je vidět na výpisu 4.2. Proměnná definovaná pro **LSTM** je složena ze dvou parametrů.

Na výpisu 4.3 je definice modelu využívající vrstvu **GRU**. Parametry definující následují model jsou *input_size* = 1, *hidden_size* = 10 a *output_size* = 1, jako v předchozím případě. Obdobně je tento model natrénován na 6000 epoch.

Výpis 4.2: Definice proměnných pro skryté stavy u LSTM, GRU a RNN

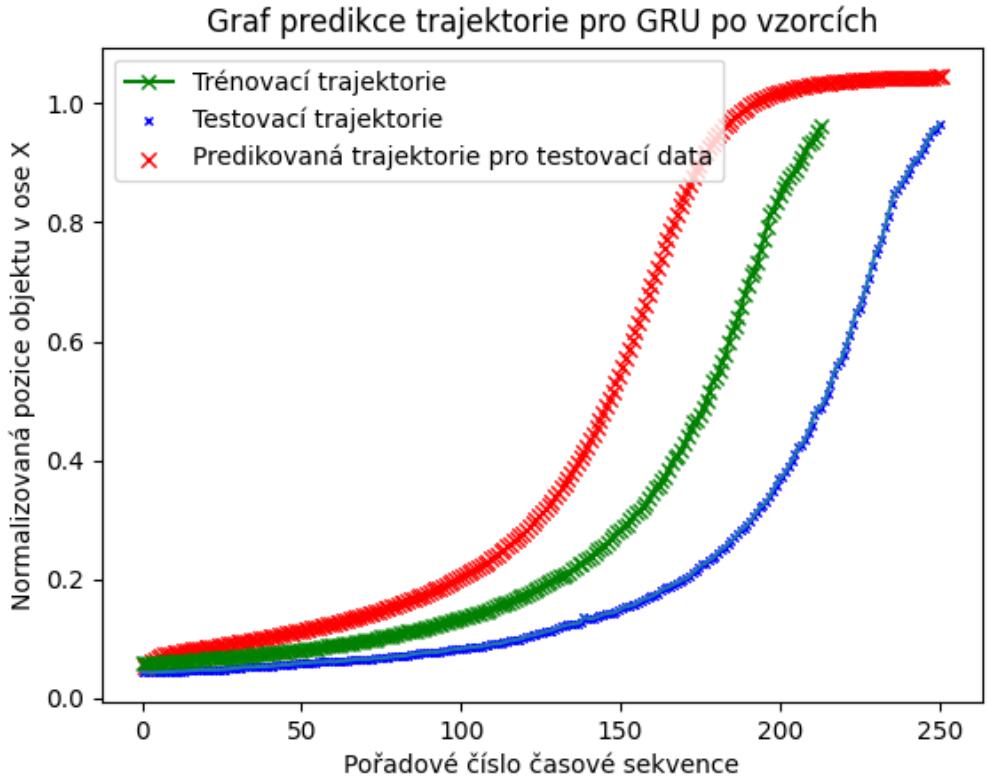
```
# Definice hidden_prev pro model s LSTM
h_t = torch.zeros(1, 1, hidden_size, dtype=torch.float32)
c_t = torch.zeros(1, 1, hidden_size, dtype=torch.float32)
hidden_prev = (h_t, c_t)

# Definice hidden_prev pro modely s GRU a RNN
hidden_prev = torch.zeros(1, 1, hidden_size)
```

Výpis 4.3: Definice modelu pro GRUperSample.

```
class GRUperSample(nn.Module):
    def __init__(self):
        super(GRUperSample, self).__init__()

        self.gru = nn.GRU(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=1,
            batch_first=True
        )
        self.linear = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden_prev):
        out, hidden_prev = self.gru(x, hidden_prev)
        out = out.reshape(-1, hidden_size)

        out = self.linear(out)
        out = out.unsqueeze(dim=0)
        return out, hidden_prev
```

Obrázek 4.2 zobrazuje, jak vypadá predikce trajektorie pomocí modelu s **GRU** vrstvou. Jako v předchozím případě, zelené body vyznačují trajektorii, která byla použita pro trénování. Modrá trajektorie ukazuje pohyb testovacího objektu a červená indikuje predikovanou trajektorii. V optimálním případě by se červená a modrá křivka měly překrývat. Z grafu je patrné, že **GRU** si objektivně vede lépe než

LSTM. To je pravděpodobně způsobeno menší robustností paměťových buněk. Obdobně jako u **LSTM** je vidět odskok hned v prvních pár bodech, který způsobuje odklonění predikované hodnoty od skutečných hodnot. To je způsobeno vynulováním proměnné pro skryté výstupní stavy, jako v případě **LSTM**. Model s **GRU** vrstvou je vhodnější pro predikci pohyby objektů v případě, kdy známe pouze poslední stav a chceme predikovat velké množství budoucích pozic objektu. Přesto tato metoda není příliš spolehlivá. Přínosem modelu s **GRU** vrstvou je lepší předpověď pohybu v případě, kdy je predikována nejbližší budoucnost.



Obr. 4.2: Ukázka predikce trajektorie pomocí GRU po jednotlivých vzorcích.

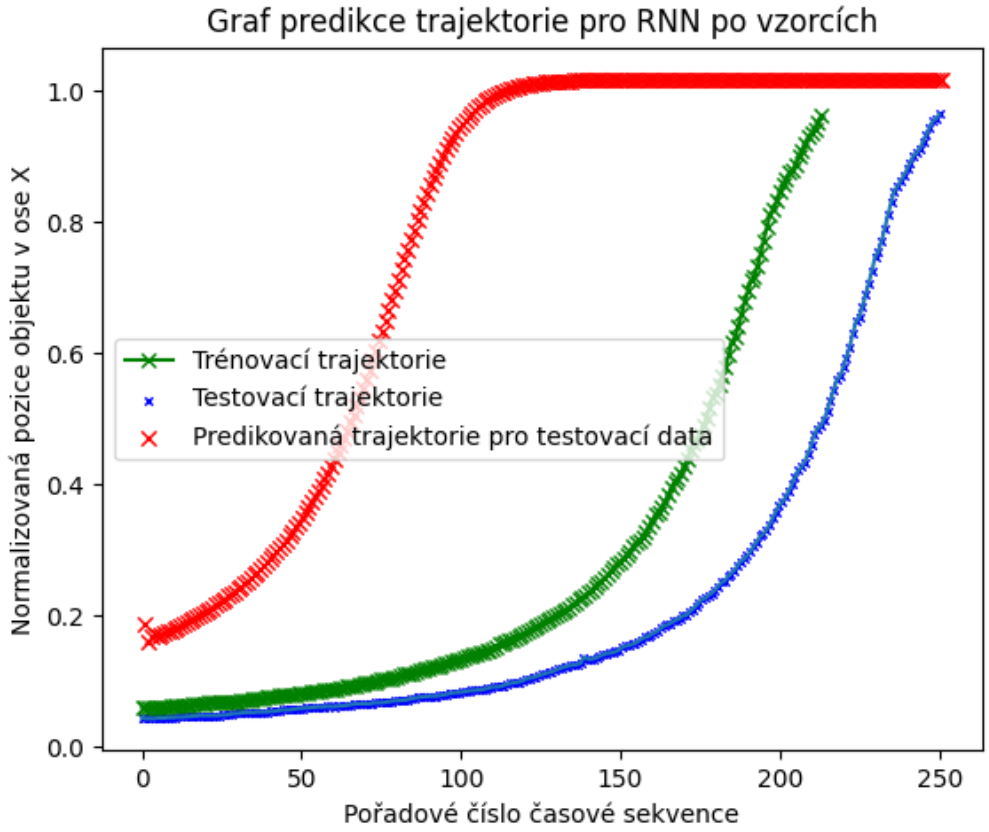
4.1.3 RNN po jednotlivých vzorcích

RNN je rekurentní neuronová síť, která pracuje se skrytými stavy jako předchozí modely s **LSTM** a **GRU**. Struktura modelu vypadá obdobně. Je tvořena z vstupní vrstvy **RNN** a výstupní vrstvy **Linear**, která slouží pro převedení výstupů z **RNN** na jedno číslo, které značí predikovanou hodnotu. Parametry, které definují tento model jsou $input\_size = 1$, $hidden\_size = 10$ a $output\_size = 1$. Úplnou definici modelu je možné vidět na výpisu 4.4. Za účelem ověření schopnosti predikovat hodnoty je model trénován na 6000 epoch, jako v předchozích případech.

Na obrázku 4.3 jsou zobrazeny křivky s trajektoriemi pro trénování, testování a predikované hodnoty. Z grafu je patrné, že daný model není vhodný pro predikci trajektorie. Chyba při predikci prvních hodnot je vetší než u předchozích modelů. Tato chyba je způsobena vynulováním skrytých stavů jako v předchozích případech. **RNN** vrstvy jsou vhodné pro zpracování sekvenčních dat, jako jsou řečové signály nebo textové sekvence. Pro predikci pohybu objektů se nehodí. Jako v předchozím případě je použita zelená barva pro trénovací datovou sadu, modrá pro testovací. Červená ukazuje predikovanou trajektorii.

Výpis 4.4: Definice modelu pro RNNperSample.

```
                                                                    1
class RNNperSample(nn.Module):                                      2
    def __init__(self):                                             3
        super(RNNperSample, self).__init__()                        4
        self.rnn = nn.RNN(input_size=input_size,                    5
            hidden_size=hidden_size, num_layers=1,                  6
            batch_first=True)                                       7
        self.linear = nn.Linear(hidden_size, output_size)           8
                                                                    9
    def forward(self, x, hidden_prev):                              10
        out, hidden_prev = self.rnn(x, hidden_prev)                 11
        out = out.reshape(-1, hidden_size)                          12
        out = self.linear(out)                                      13
        out = out.unsqueeze(dim=0)                                  14
        return out, hidden_prev                                     15
```



Obr. 4.3: Ukázka predikce trajektorie pomocí RNN po jednotlivých vzorcích.

4.1.4 Polynomiální regresní model

Použití **polynomiálního regresního modelu** vyžaduje jiný přístup. V předchozích případech se predikuje pouze jedna souřadnice pro osu **x**, čímž je ztracena informace o poloze vozidla. To by bylo možné kompenzovat například zdvojením modelu, což by přineslo výrazné zvětšení chyby při predikci.

Návrh **polynomiálního regresního modelu** je zobrazen ve výpisu 4.5. Model je složen z parametrů **p1** až **p6**, které reprezentují parametry polynomu. Vstupní hodnotou je normalizovaná pozice objektu na ose **x**. Návratovou hodnotou modelu je normalizovaná hodnota popisující pozici objektu v ose **y**. Takto vytvořený model je trénován na 100 000 epoch.

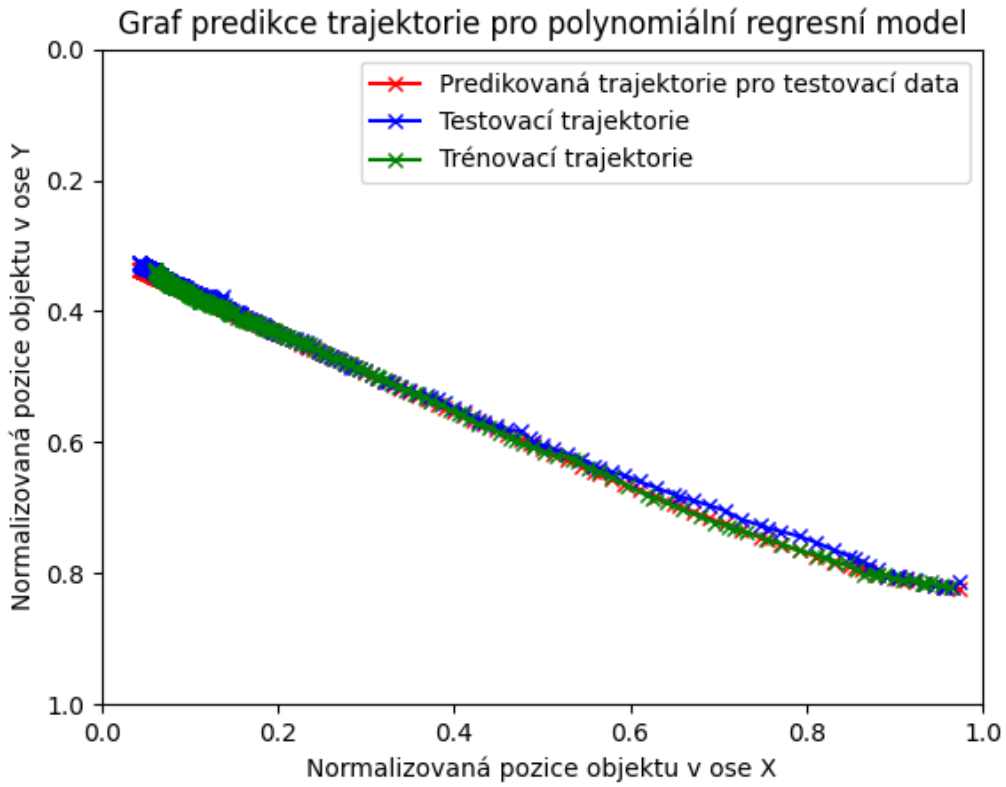
Výpis 4.5: Definice modelu pro PolynomialRegressionModel [10].

```
                                                                    1
class PolynomialRegressionModel(nn.Module):                         2
    def __init__(self):                                             3
        super().__init__()                                          4
```

```
        self.p1 = nn.Parameter(                                    5
            torch.rand(1, requires_grad=True,                      6
                       dtype=torch.float32))                       7
        self.p2 = nn.Parameter(                                    8
            torch.rand(1, requires_grad=True,                      9
                       dtype=torch.float32))                       10
        self.p3 = nn.Parameter(                                    11
            torch.rand(1, requires_grad=True,                      12
                       dtype=torch.float32))                       13
        self.p4 = nn.Parameter(                                    14
            torch.rand(1, requires_grad=True,                      15
                       dtype=torch.float32))                       16
        self.p5 = nn.Parameter(                                    17
            torch.rand(1, requires_grad=True,                      18
                       dtype=torch.float32))                       19
        self.p6 = nn.Parameter(                                    20
            torch.rand(1, requires_grad=True,                      21
                       dtype=torch.float32))                       22
                                                                   23
    def forward(self, x):                                          24
        return self.p1*x**5 + self.p2*x**4 +                       25
               self.p3*x**3 + self.p4*x**2 +                       26
               self.p5*x + self.p6                                 27
```

Polynomiální regresní model není schopný predikovat pozici, ale je možné procházet nejbližší prostor objektu a tak určit, po jakých trajektoriích se pohybovaly předchozí objekty. Pomocí kódu ve výpisu 4.5 dojde k pevné fixaci trajektorie. Voláním modelu je možné zjistit jakou pozici by měl sledovaný objekt mít v daném místě detekce. To umožňuje porovnat vypočítanou hodnotu se skutečnou pozicí v ose **y**. Pokud se objekt pohybuje po testované trajektorii, potom se v optimálním případě rozdíl mezí skutečnou a vypočtenou pozicí bude blížit nule.

Na obrázku 4.4 jsou vidět tři překrývající se křivky. Zelená křivka zobrazuje trénovací množinu dat. Modrá zobrazuje trajektorii testovacího objektu. Červená ukazuje, jak vypadají souřadnice v ose **x** a **y**, které jsou přiřazeny k testovacímu objektu podle jeho aktuální **x** pozice. Z definice modelu ve výpisu 4.5 vyplývá, že výstup z modelu by měl být totožný s trénovacími daty. Tato vlastnost je na obrázku 4.4 vidět díky překrývajícím se křivkám s predikcí a s trénovací množinou.



Obr. 4.4: Ukázka predikce trajektorie pomocí polynomiálního regresního modelu.

4.2 Algoritmické řešení

Tato část se zaměřuje na návrh vlastního řešení pro predikční metodu. Velká část veřejného výzkumu je proprietární a není možné se bezplatně dostat ke kvalitním datovým sadám a k nejnovějším strukturám neuronových sítí. Dalším problémem je stále se zvyšující výpočetní náročnost těchto nových modelů. Z tohoto důvodu se práce zaměřuje na řešení, které by bylo funkční a přitom by nebylo náročné na výpočetní výkon. Toto řešení vychází ze zjednodušené implementace neuronové sítě vytvořené pomocí polynomů.

Výstupem této části je třída *MovementPrediction*, kterou je možné nalézt v příloze v souboru *prediction.py* ve složce *server*. Podpůrným souborem, který definuje trasy polynomů je *poly.data*, který také umístěn ve složce *server*.

4.2.1 Teoretický návrh algoritmu

Cílem je vytvořit predikční algoritmus, který je rychlý a nezvyšoval by výpočetní náročnost celého systému. Algoritmické řešení vychází principiálně z návrhu neuronové sítě využívající polynomy pro predikci pohybu, protože jako jediný ze zkoumaných

modulů dokáže efektivně předpovídat pohyb vozidel. Samotný model pro předpověď nestačí, proto je nutné do tohoto řešení přidat další mechanizmy, které umožňují filtraci výsledných množin predikcí podle předpokládaného úhlového intervalu pohybu pro jednotlivé skupiny směrů.

Celý návrh je postaven na statistickém modelu a testování, zda daný objekt spadá do určité skupiny, v tomto případě do určitého směru pohybu. Každý směr je reprezentován polynomem 3. řádu. Souřadnice $\mathbf{x}$ je vstupní parametr do daného polynomu a výsledek polynomu je souřadnice $\mathbf{y}$. Tedy pro každý detekovaný objekt v jednom snímku jsou spočítány hodnoty souřadnice $\mathbf{y}$, a to pomocí všech polynomů reprezentujících jednotlivé směry. Následně jsou porovnány absolutní hodnoty rozdílů vypočtených a skutečných hodnot v ose $\mathbf{y}$. Pokud je absolutní hodnota z rozdílu menší než stanovená hodnota pro parametr ***diff***, je tento směr považován za možný a je vrácen predikčním algoritmem.

Následnou dodanou funkcionalitou je filtrování směru podle intervalu úhlů. To funguje na principu, že každý polynom reprezentující jeden směr obsahuje informaci o předpokládaném směru pohybu objektu po tomto polynomu. Tato část vyhodnocuje vektor pohybu objektu ve dvou po sobě jdoucích snímcích. Funkcionalita filtrace je předřazena výpočtu souřadnice $\mathbf{y}$, proto jsou pro každý objekt počítány pouze polynomy, u kterých objekt spadá do intervalu úhlů daného polynomu. Díky tomu jsou odfiltrovány irelevantní směry pohybu (nepředpokládáme jízdu vozidla v protisměru). Dalším benefitem této metody je možnost zvětšení parametru *diff*, a tím docílit nižší chybovosti predikce.

4.2.2 Příprava trénovacích dat

Prvním krokem je využití nějakého algoritmu pro rozřazení trajektorií do skupin, které budou reprezentovat jednotlivé směry. Proto i v tomto případě je experimentováno s různými metodami pro strojové učení, jako je například *k-means*. Ty bohužel nebyli úspěšné. Jedná se o podobný problém jako u jednoduchým modelů. Není možné pomocí nich bezpečně a přesně předpovídat pohyby objektů a nebo je rozřadit do určitých skupin ani při kombinaci různých parametrů pro vyhodnocování (vektor pohybu, počátek, predikce, atd.). Proto je potřeba jednotlivé směry rozřadit manuálně. K tomu je použit skript, který požívá modul *pandas* a *matplotlib*. Tento skript je možné vidět na výpisu 4.6. Kód je také součástí příloh a je možné ho nalézt v souboru s názvem *generarate_trajectory_graphs.py*.

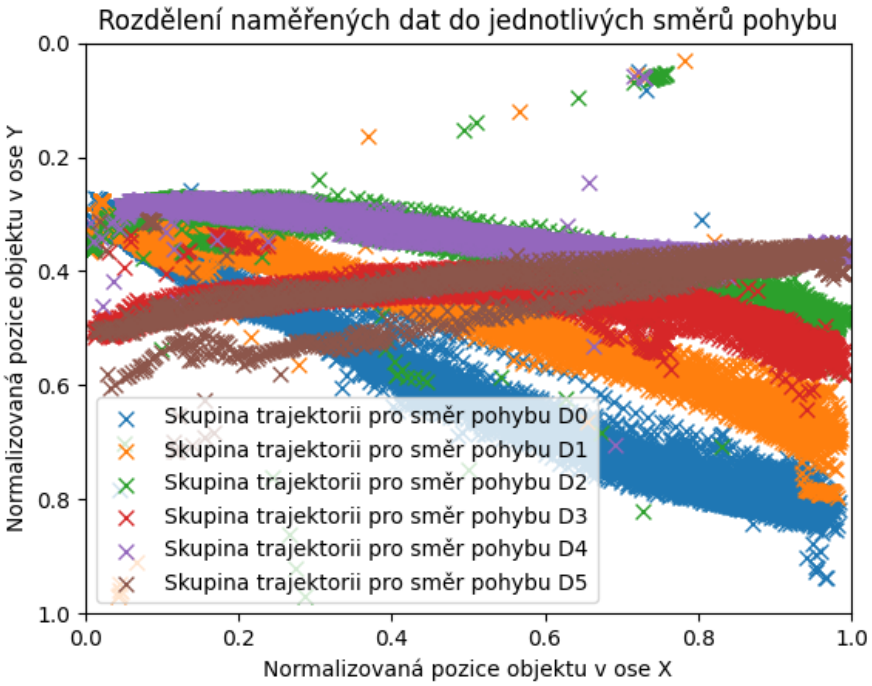
Výpis 4.6: Kód pro vygenerování trajektorií.

```
                                                                    1
import pandas as pd                                                 2
import matplotlib.pyplot as plt                                     3
                                                                    4
                                                                    5
data = pd.read_csv('dataset_train.csv', sep=';')                    6
                                                                    7
                                                                    8
data["xx"] = data['x']/1280                                         9
data["yy"] = data['y']/720                                          10
data = data.filter(items=['xx', 'yy', 'id'])                        11
                                                                    12
                                                                    13
for i in range(1, 6000):                                            14
    fi = data[data['id'] == i].filter(items=['xx', 'yy'])           15
    if len(fi) > 100:                                               16
        data_n = fi.to_numpy()                                      17
        plt.scatter(                                                18
            list(data_n[:, 0]),                                     19
            list(data_n[:, 1]),                                     20
            label=str(i),                                           21
            marker='x',                                             22
            linewidths=1)                                           23
        plt.xlim(0, 1)                                              24
        plt.ylim(1, 0)                                              25
        plt.savefig(F"SOMEFOLDER/output_{i}.png")                   26
        plt.close()                                                 27
```

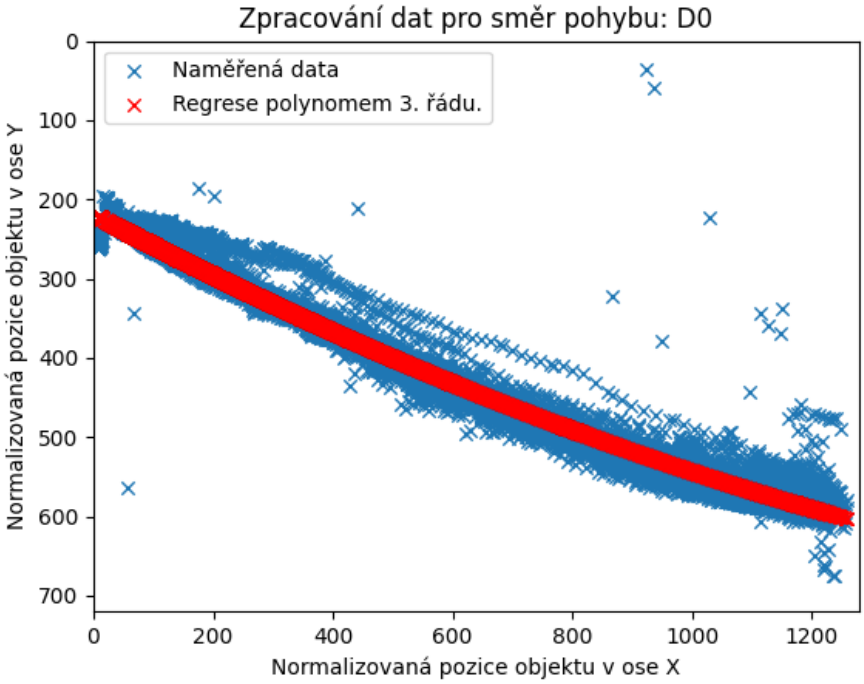
Při přípravě trénovacích dat je potřeba provést detekce a trasování na trénovacích datech. K tomu je vytvořen video záznam. Jedná se o ten stejný jako v kapitole 3. Data s informacemi o detekcích jsou pomocí detekčního programu vyexportována do textového souboru *dataset_train.csv*. Graficky je tento soubor vizualizován na obrázku 3.1. Takto získaná data bylo nutné rozřadit do jednotlivých směrů. Toho bylo docíleno manuálním rozřazením do jednotlivých skupin, a to podle tvaru grafu. Celkově tak vzniklo 6 skupin, kde každá reprezentuje jeden směr. Jednotlivé datové skupiny jsou zobrazeny na následujících obrázcích 4.6 4.7 4.8 4.9 4.10 4.11.

Rozdělení do skupin je provedeno pomocí skriptu, který pro každé vozidlo vykresluje jeho směr do jednoho grafu a tento graf je následně uložen do obrázku. Název obrázku identifikuje daná ID vozidla z detekce. Takto získané grafy jsou ručně rozřazeny do jednotlivých složek podle průběhu funkce. Nevhodné průběhy

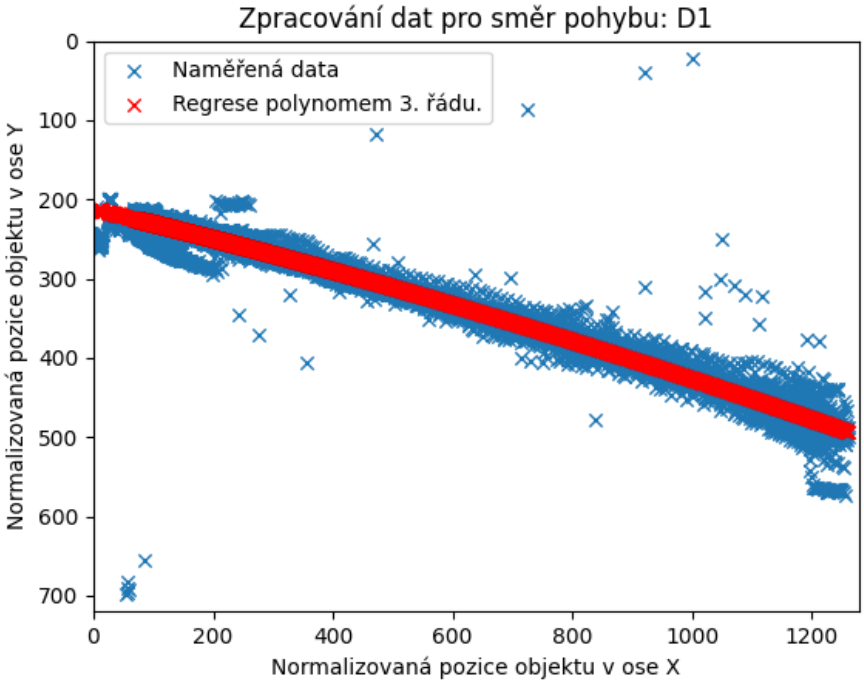
nebo chybné detekce, které se nepovedlo kvantitativně odfiltrovat jsou odstraněny, aby neovlivňovaly následné výpočty polynomů. Celkem tak bylo získáno 334 trajektorií vozidel, které jsou rozházeny do 6 skupin viz obrázek 4.5. Pro zjednodušení práce byly tyto ručně rozřazené skupiny převedeny zpět do formátu ***CSV***. Jedná se o soubory s názvy *polynoms_0.csv* až *polynoms_5.csv*. Tyto soubory obsahují pouze informace o souřadnicích **x** a **y**



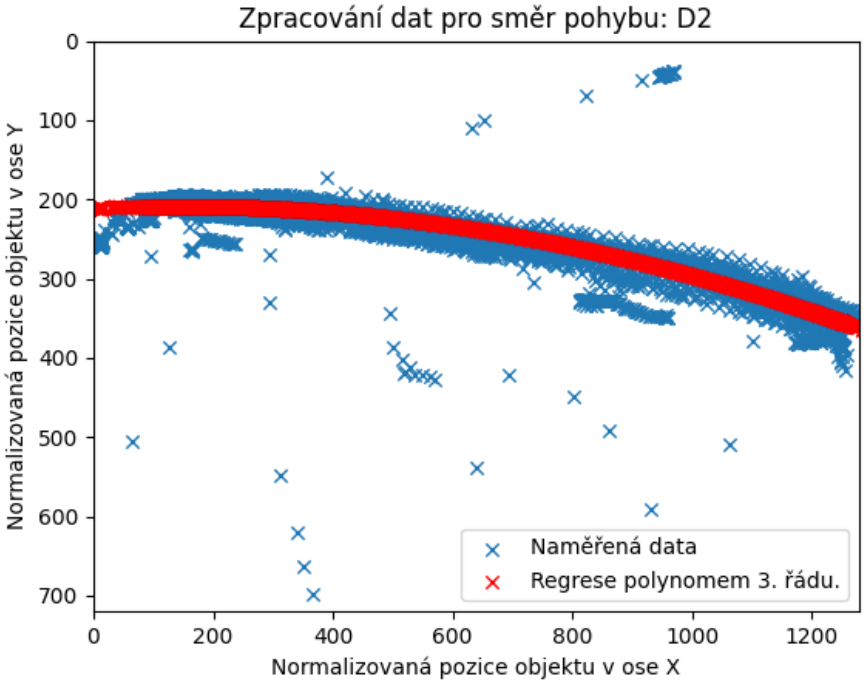
Obr. 4.5: Graf ukazující rozdělení datové množiny na jednotlivé směry.



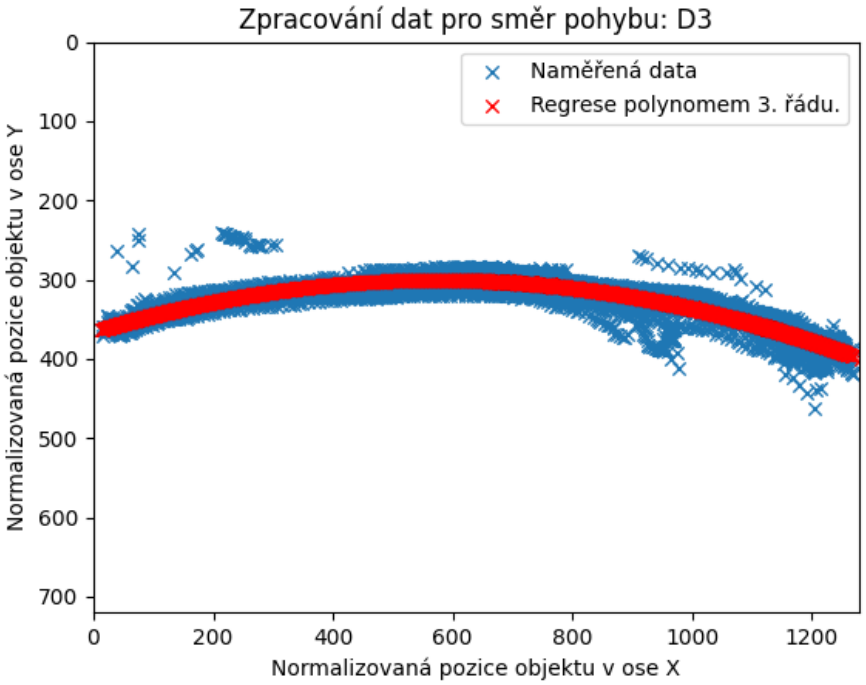
Obr. 4.6: Graf s datovou množinou pro směr označený pořadovým číslem 0



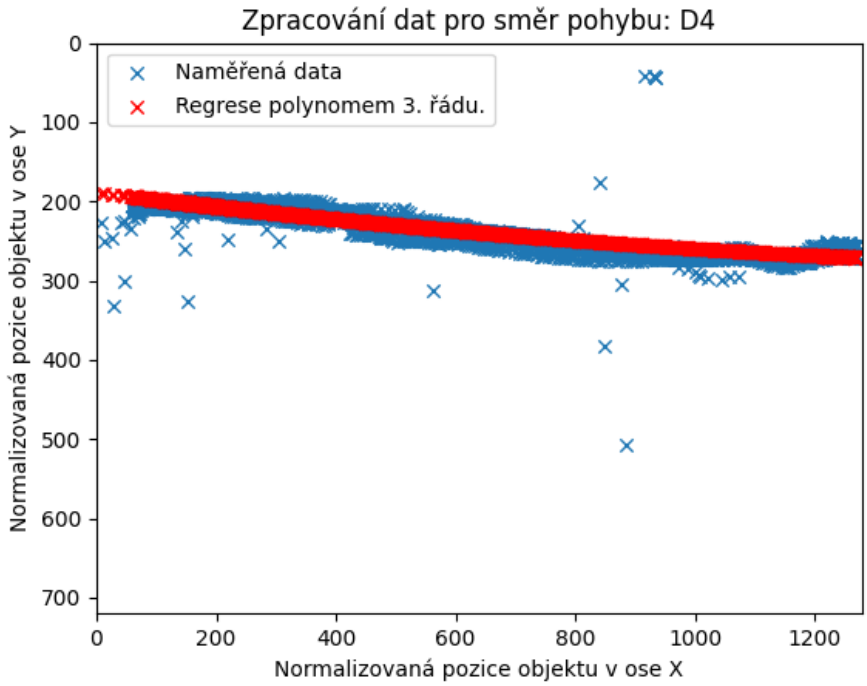
Obr. 4.7: Graf s datovou množinou pro směr označený pořadovým číslem 1



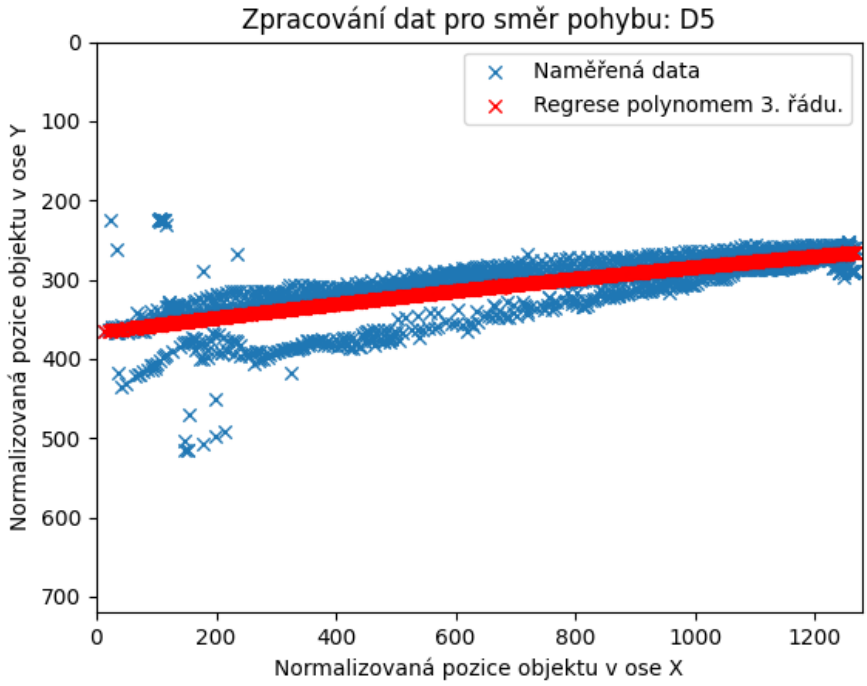
Obr. 4.8: Graf s datovou množinou pro směr označený pořadovým číslem 2



Obr. 4.9: Graf s datovou množinou pro směr označený pořadovým číslem 3



Obr. 4.10: Graf s datovou množinou pro směr označený pořadovým číslem 4



Obr. 4.11: Graf s datovou množinou pro směr označený pořadovým číslem 5

V obrázcích 4.6 4.7 4.8 4.9 4.10 4.11 je možné si povšimnout červených bodů, které jsou body polynomu 3. řádu. Tyto polynomy jsou vypočítány pomocí python modulu **sklearn**, kde je využito již implementované metody pro lineární regresi. Následně jsou z těchto bodů určeny jednotlivé parametry daného polynomu pomocí třídy **PolynomialFeatures**, která je součástí již zmíněného modulu **sklearn**. Takto získané parametry je možné vidět v následujícím výpisu 4.7. Především se jedná o poslední tři čísla na jednotlivých řádcích. První dvě čísla reprezentují interval vektorů, pro daný polynom.

Výpis 4.7: Parametry definující vygenerované polynomy (Obsah souboru poly.data)

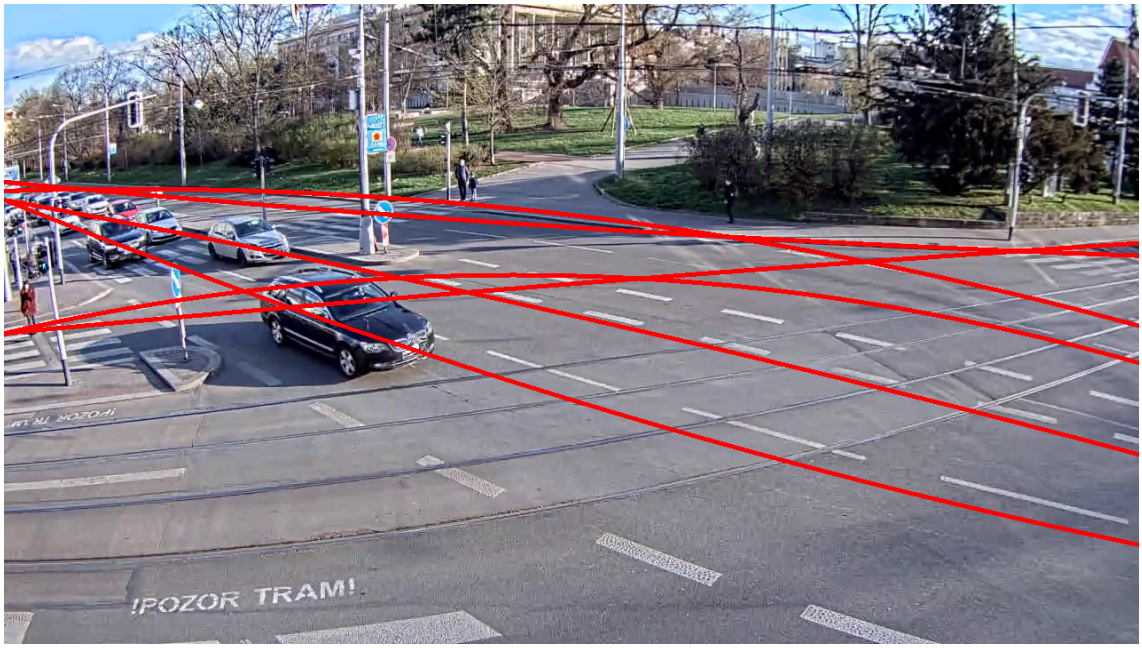
```
                                                                    1
335 ;350 ; 220;   4.0013295e-01 ;  -7.5796481e-05                   2
340 ;355 ; 220;   1.8209648e-01 ;   3.2565149e-05                   3
173 ;180 ; 210;  -0.03409461    ;   0.00011928                      4
170 ;190 ; 370;  -2.2808814e-01 ;   1.9855809e-04                   5
170 ;180 ; 200;   9.394166e-02  ;  -2.291033e-05                    6
180 ;190 ; 370;  -9.4564773e-02 ;   1.1976886e-05                   7
```

Ve výpisu 4.7 první dvě čísla definují interval úhlů, ve kterém se objekt musí pohybovat, aby byla daná trasa vyhodnocována. Zbytek hodnot definuje parametry polynomu. Například první řádek z výpisu 4.7 definuje interval úhlů <335 ; 350> a polynom, viz vzorec 4.1.

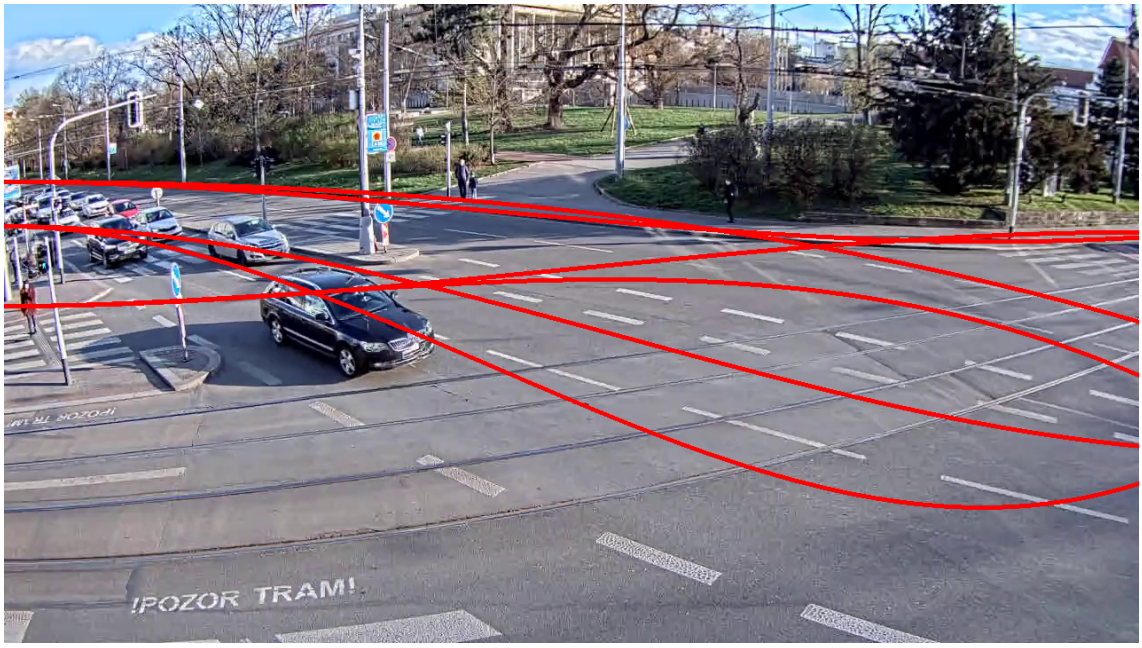
$$y_{vyp} = 220 \cdot x^0 + 4.0013295 \cdot 10^{-1} \cdot x^1 - 7.5796481 \cdot 10^{-5} \cdot x^2 \tag{4.1}$$

Důležitým faktorem, který může ovlivňovat výslednou přesnost je, že první tři hodnoty z řádku nebyly vypočítány pomocí statistického modelu, ale jsou určeny experimentálně. Jedná se o interval a nultý prvek polynomu, který definuje posun v ose **y**. Provedení optimalizace těchto parametrů není dokončené, jedná se tedy o hrubé odhady, jak by tyto hodnoty mohly vypadat. Lepším nastavením by bylo s velkou pravděpodobností možné zefektivnit danou predikční metodu. Přesto tyto výchozí hodnoty pro potřeby práce bohatě dostačují především díky tomu, že je možné tyto nedostatky kompenzovat nastavením větší hodnoty parametru *diff*.

Práce také experimentovala s použitím polynomu vyšších řádů. Bohužel se ukázalo, že použití vyššího řádu než tři může vnášet další nepřesnosti do predikce, které jsou způsobeny velikostí trénovací sady. Tedy model by začal brát v potaz anomálie, které jsou obsaženy v trénovacích datech. Jak by vypadali trajektorie čtvrtého řádu oproti použitým je možné porovnat na obrazcích 4.12 a 4.13. Při srovnání je vidět že polynom 4. řádu není vhodný, protože jeho trajektorie nekopíruje přirozený pohyb vozidel.



Obr. 4.12: Vykreslení polynomů 3. řádu do snímku videa (vhodné pro predikci)



Obr. 4.13: Vykreslení polynomů 4. řádu do snímku videa (nevhodné pro predikci)

4.2.3 Implementace predikčního algoritmu

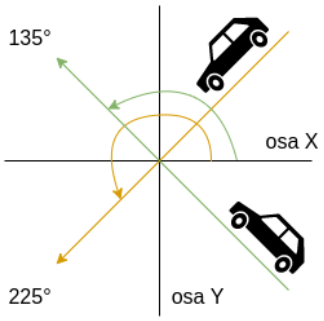
Implementace je provedena v programovacím jazyce Python. Jedná se o jednoduchou třídu s názvem *MovementPrediction*, která obsahuje metodu pro načtení souboru s definovanými polynomy, viz soubor ve výpisu 4.7. Další metodou je *_calculate_one_* pro výpočet předpokládané souřadnice **y** podle ukázky výpočtu 4.1 a metoda, která vrací možné směry pohybu objektu. Definice predikční funkce je vidět na výpisu 4.8.

Výstupem funkce *calculate* je pár hodnot. První je struktura obsahující budoucí pozice objektu. Jedná se o strukturu typu **list listů**, ve kterém jsou uloženy n-tice obsahující předpovídané souřadnice **x** a **y**. První list v struktuře reprezentuje možný směr, podřadný list obsahuje již predikované pozice objektu. Pokud objekt není přiřazen ani k jednomu polynomu, hodnota této struktury bude prázdný list. Dalším parametrem, který byl dodán pro kvantitativní optimalizaci směru pohybu je proměna *degree*, která vrací aktuální hodnotu úhlu pohybu.
Vstupními parametry do predikční funkce jsou:

- input_x - aktuální pozice objektu v ose **x** (levý horní roh detekce),
- input_y - aktuální pozice objektu v ose **y** (levý horní roh detekce),
- diff_x - rozdíl v ose **x** mezi dvěma po sobě jdoucími predikcemi,
- diff_y - rozdíl v ose **y** mezi dvěma po sobě jdoucími predikcemi,
- car_center_y - pozice středu objektu v ose **y**.

Samotný predikční algoritmus si neukládá žádné záznamy o předchozích polohách objektu, proto je potřeba, aby tato data byla uložena a zpracována mimo tuto třídu. Tohle řešení je vybráno kvůli tomu, aby bylo možné jednodušeji testovat výpočetní metodu.

Na obrázku 4.14 je zobrazeno, jakým způsobem jsou vypočítávaný úhly pohybu. K tomu jsou použity goniometrické funkce uvnitř metody *calculate* v výpisu 4.8. Základem je určení kvadrantu a poté dopočet přesného úhlu pomocí funkce *arkus tangens*. Během výpočtu je potřeba také odchytit nedefinované stavy, které můžou nastat při dělení nulou.



Obr. 4.14: Obrázek demonstrující výpočet úhlu pohybu objektu.

Výpis 4.8: Ukázka metody pro predikci pohybu objektů.

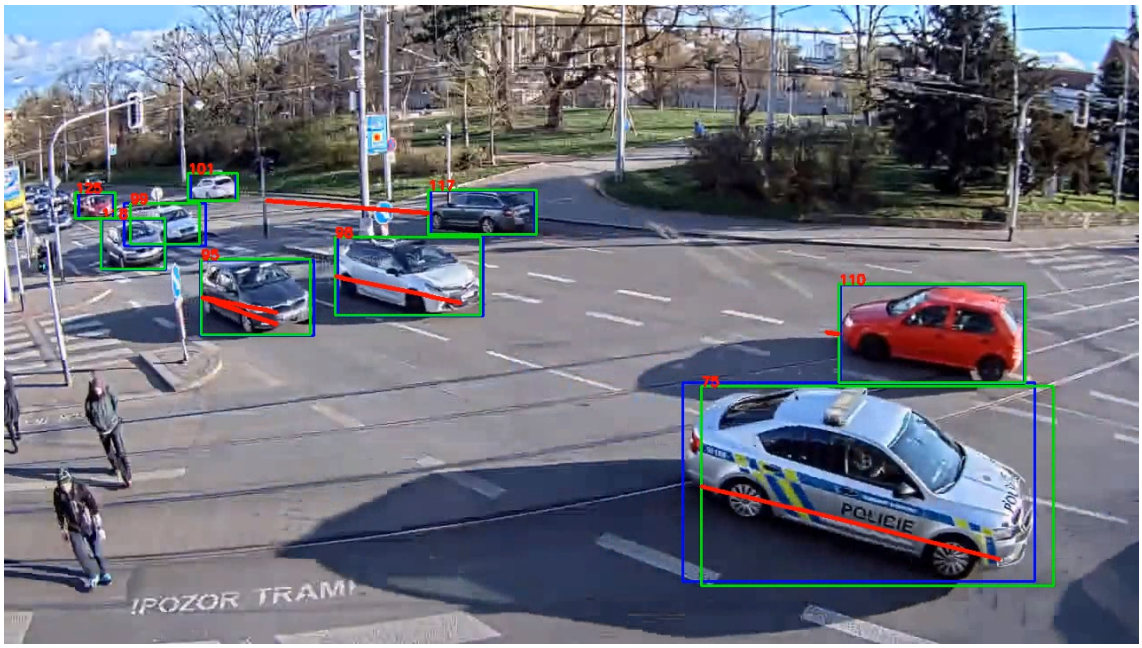
```
                                                                    1
def calculate(self, input_x: float, input_y: float,                 2
              diff_x: float, diff_y: float,                         3
              car_center_y: float):                                 4
    # vytvoření návratové struktury                                 5
    ret_list = []                                                   6
    # výpočet úhlu pohybu zkounamého objektu                        7
    if diff_y == 0:                                                 8
        diff_y += 0.0000000000000001                                9
    sigma = math.atan(diff_x / diff_y) * 180 / math.pi              10
                                                                    11
    if diff_y > 0:                                                  12
        degree = 270 + sigma                                        13
    else:                                                           14
        degree = 90 + sigma                                         15
    # Iterace přes polynomy a hledání shody                         16
    #    a výpočet budoucí pozice.                                  17
    for pos, params in enumerate(self.data):                        18
        y_calculated = self._calculate_one_(input_x, params)        19
        if abs(y_calculated - input_y) < self.diff and \            20
                ((self.directions[pos][0] < degree) and \           21
                 (degree < self.directions[pos][1])):               22
            sub_list = []                                           23
            first = True                                            24
            for i in range(self.forward):                           25
                if first:                                           26
                    y = self._calculate_one_(                       27
                        input_x + (diff_x * i),                     28
                        params)                                     29
                    y_offset = car_center_y - y                     30
                    first = False                                   31
                sub_list.append(                                    32
                    [int(round(input_x + (diff_x * i))),            33
                     int(round(                                     34
                         self._calculate_one_(                      35
                             input_x + (diff_x * i),                36
                             params) + y_offset                     37
                     ))])                                           38
            ret_list.append(sub_list)                               39
    return ret_list, degree                                         40
```

4.3 Výsledky predikčního algoritmu

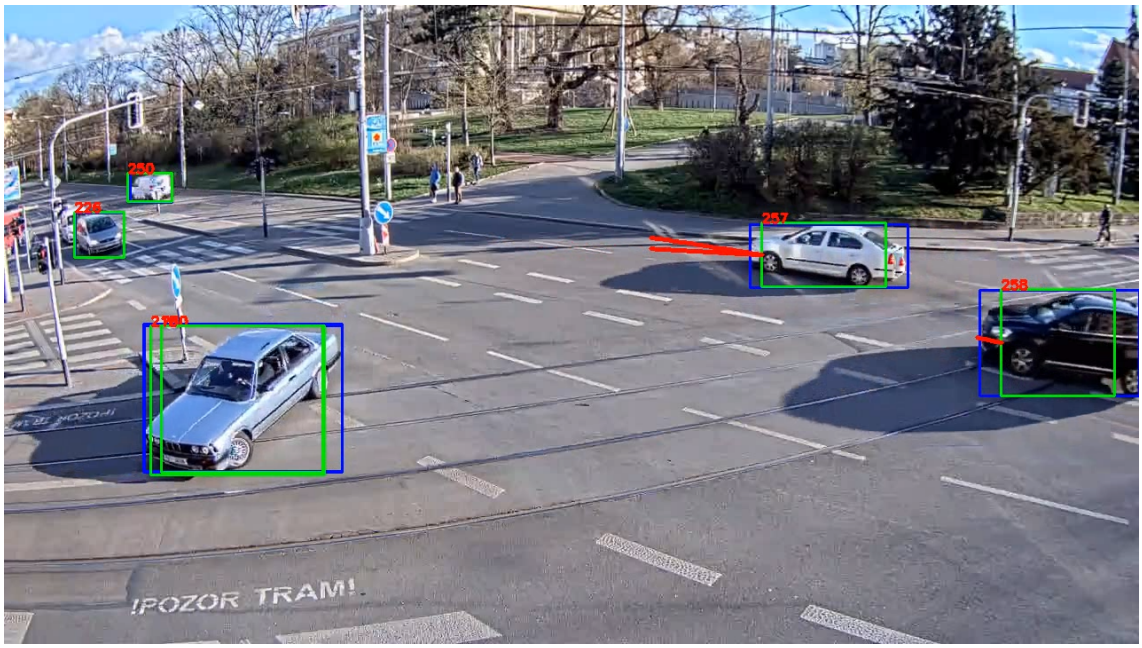
Vyhodnocení výsledků predikce naráží na problém, protože práce nedisponuje žádným referenčním algoritmem, se kterým by se nová metoda dala srovnat. Překážkou je, že predikční algoritmy, které již existují jsou často proprietárním řešením nějaké komerční aplikace. U otevřených implementací je chybou, že neexistuje standart, který by definoval, jak se mají parametry vyhodnocovat a porovnávat. Další komplikací je, že neexistuje univerzální bezplatná datová sada, která by se dala použít pro referenční měření. Existuje několik velkých datových sad, ale přístup k nim je zpoplatněn.

Obrázek 4.15 ukazuje, jak vypadá predikce pomocí vytvořeného algoritmu. Modrý obdélník vytyčuje detekované objekty pomocí detekčního modelu **YOLOv5**. Zelené obdélníky ohraničují objekt, který je vrácen z trasovacího modelu ***deep SORT***. Červené číslo nad zeleným rámečkem informuje pozorovatele o ID daného objektu. Červené trajektorie zobrazují predikované směry pohybu. Jejich délka závisí na rychlosti objektu, proto nejsou stejně dlouhé u všech objektů. Vždy nám zobrazují, kde se bude objekt nacházet v příštích 30 snímcích obrazu. Jde o hrubý odhad, protože snímaný obraz neodpovídá skutečné realitě. Predikční algoritmus neumí pracovat s perspektivou, a proto u vozidel, které se pohybují směrem ke kameře predikuje pomalejší rychlost, než u vozidel jedoucích opačným směrem.

Ze subjektivního vyhodnocení predikčního algoritmu vyplývá, že pro vozidla standardních rozměrů jedoucích v uvažovaných jízdních pruzích a směrech je účinný. Problémem pro tento detektor je například neschopnost predikovat změnu jízdního pruhu vozidla a nebo vozidla pohybujícího se po nestandardních drahách. Taková situace je vyobrazena na obrázku 4.16. Nevýhodou je také predikce pohybu pro velké objekty, jako jsou například autobusy nebo nákladní vozidla, které mají posunutý střed objektu. Proto jejich přiřazení k nějakému polynomu selhává.



Obr. 4.15: Ukázka výsledků predikčního algoritmu.



Obr. 4.16: Ukázka selhání predikčního algoritmu.

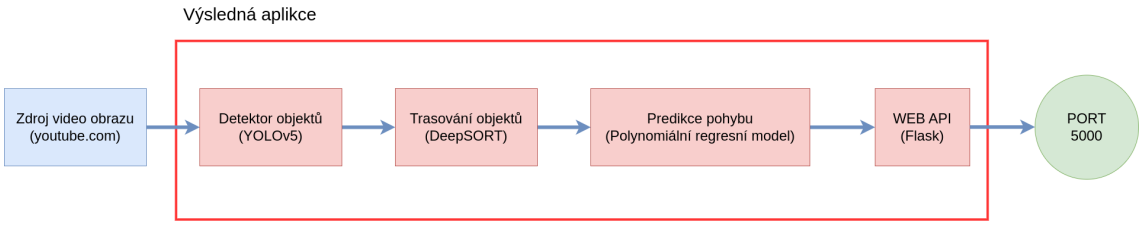
5 Výsledky a diskuze

5.1 Finální aplikace

Výsledkem práce je ucelený program, jehož výstupem je webové rozhraní, na kterém je možné vizualizovat výsledky predikce. Protože bylo nutné pro tuto stránku povolit port 5000, z bezpečnostních důvodů jde výsledný program spustit pouze na školním serveru *gravy.utko.feec.vutbr.cz*. Jelikož se jedná o server, který nedisponuje takovým výpočetním výkonem, detekce a trasování není možné provádět v reálném čase. Proto může občas dojít k nenadálému skoku v obrazu. To je zapříčiněno vyprázdněním zásobníku nezpracovaných snímků a začátkem predikce nejnovějších snímků.

Výsledná aplikace je postavena na implementaci **YOLOAIR**. Je složena ze souborů *api.py*, *detection.py* a *prediction.py*. Tyto soubory je možné nalézt ve složce *server* v přílohách.

Struktura výsledné aplikace je schématicky vyobrazena na obrázku 5.1



Obr. 5.1: Schéma výsledné aplikace pro detekci objektů a predikci pohybu.

5.2 Dosažení cílů diplomové práce

V zadání práce jsou definovány následné cíle:

- pořídit několik videozáznamů dopravního uzlu pro potřeby experimentů,
- rozdělit datovou množinu na část trénovací a na část testovací,
- zpracovat rešerši a srovnat algoritmy pro detekci objektů,
- provést srovnání z pohledu přesnosti, latence, paměťových nároků a výpočetních nároků,
- vytvořit experiment, který porovná přesnost vybraných detektorů a vykreslí trajektorii pohybu jednotlivých objektů.

Pro potřeby vývoje a testování vznikly dvě datové množiny, jedna se skládá z databáze fotografií a druhá je tvořena dvěma videozáznamy. Všechny jsou pořízeny ze stejné kamery, která je umístěn na křižovatce ulic Koliště a Milady Horákové v Brně. Datová sada fotografií je rozdělena na testování (20%), trénování (75%) a validační

(5%) množinu. Tato datová sada je použita pro trénování, validaci a vyhodnocení výsledků. Video záznamy jsou pouze dva a jsou použity v druhé části práce pro vykreslení trajektorií a pro predikci pohybu. Větší z videozáznamů je dlouhý 21 minut a 53 sekund, je použit jako trénovací množina. Druhý video záznam je dlouhý 3 minuty a 5 sekund a je aplikován k subjektivnímu vyhodnocení přesnosti predikční metody. Místo validační množiny je využito webové rozhraní, na kterém je možné sledovat a ladit parametry jednotlivých algoritmů. Vykreslení trajektorie je provedeno v obrázku 3.1. Tato data jsou získána pomocí trasovacího modelu ***deep SORT***. V obrázku 3.1 jsou jednotlivé trajektorie vozidel rozlišeny barevně.

5.3 Zjištěné nedostatky

V průběhu implementace bylo zjištěno, že původní vytvořený model pro porovnání parametrů detekčních modelů není dostačující pro účely trasování. V práci je z tohoto důvodu použit natrénovaný model **yolov5s6**. Lepším řešením by bylo vzít natrénovaný model pro **YOLOv5** a dotrénovat ho pomocí vytvořené datové sady. K této možnosti nedošlo, protože model **yolov5s6** a trénovací sada jsou spolu nekompatibilní. Důvodem je rozdílné značení definování ID skupin detekovaných objektů.

Výsledný trasovací model vykazuje jistou míru nepřesnosti. To je pravděpodobně způsobeno špatnou volbou neuronové sítě v **deep SORT**. Jiné konfigurace **deep SORT** se nepovedlo otestovat tak, aby byla tato hypotéza potvrzena. Problém byl v chybějících modulech, které se nepovedlo dohledat nebo nainstalovat na danou verzi Python. Falešné detekce v **deep SORT** můžou být způsobeny již v detekční části **YOLOv5**. To by se teoreticky dalo vyřešit přetrénováním modelu **yolov5s6**.

Při volbě lokace byla stanovena podmínka, že křižovatkou musí projíždět tramvaje. Z tohoto důvodu byla vybrána křižovatka ulic Koliště a Milady Horákové. Přestože touto křižovatkou projíždí tramvaje, datová sada neobsahovala dostatečné množství vzorků pro jejich vyhodnocení.

V tomto silničním uzlu existuje pouze jedna veřejná kamera, která se na začátku práce zdála vhodná pro potřeby srovnání neuronových sítí. To bohužel neplatí pro predikční algoritmus. Predikční metoda je silně závisí na umístění kamery v křižovatce. Například změna záběru nebo posunutí kamery by mělo za následek absolutní nefunkčnost dané metody. Stejně tak tato metoda není vhodná pro případy, kdy by se vozidla pohybovaly po složitějších křivkách (kruhové objezdy). Obecně musí být splněna podmínka, že pro každou souřadnici **x** musí existovat právě jedna souřadnice **y**.

Závěr

Zadáním práce bylo vytvořit datovou sadu, otestovat detekce a vyhodnotit výsledky několika zvolených neuronových sítí. Datová sada byla vytvořena za pomocí programu, který sbíral data z veřejné dopravní kamery. Data byla následně zpracována v aplikaci *Label Studio* a tím vznikla datová sada. Poté byla tato datová sada rozdělena na tři množiny (trénovací, validační a testovací). Pomocí trénovací množiny byly natrénovány modely jednotlivých implementací. Následně byla získaná data z detekcí vyhodnocena programem, ze kterého bylo možné získat výstupy formou grafů. K vyhodnocování byla použita testovací množina.

Z výsledků vyplývá, že vytvořená datová sada není optimální. Je to patrné z detekcí, protože modely dokáží přesně rozeznávat pouze automobily. Důvodem je, že datová sada obsahuje převážně tento jeden druh objektů. Proto jsou detekce jiných objektů téměř nemožné. Naopak jistota, s jakou jsou automobily detekovány je vysoká u všech sítí. Tento jev může být způsoben tím, že se objekty objevují pouze v určité části fotky (silnice) a mají vždy podobné rozměry.

Vypočítané a změřené výsledky neuronových sítí jsou součástí diplomové práce. Obecně je možné říci, že mladší implementace jsou při porovnání výsledků lepší. Z vybraných implementací se nejlépe umístila síť YOLOv5, která je velice rychlá, má nízké nároky na GPU paměť, a to bez výrazného ovlivnění přesnosti detekce.

V druhé části se práce zabývá trasováním objektů. K tomu je použit sledovací algoritmus *deep SORT*. Jeho výstupem je množina objektů, ke kterým je přiřazeno unikátní ID, pomocí kterého je možné objekt sledovat napříč snímky obrazu. Díky tomu je možné provést detekce spolu s trasováním na pořízeném videozáznamu. Tento videozáznam je součásti trénovací sady pro následnou predikci pohybu. Detekované trajektorie jsou zobrazeny v grafu.

V poslední části se práce zabývá možností predikovat směr pohybu objektů. Jedná se o přidanou hodnotu této práce, protože zvolená metoda je svým způsobem unikátní a přitom subjektivně dosahuje překvapivě dobrých výsledků. Vytvořená implementace vychází z polynomiálního regresního modelu, který byl vytvořen z trénovací množiny, která vznikla při trasovaní objektů. Ukázky úspěšných předpovědí a situací, kde metoda selhává jsou zobrazeny v práci.

Literatura

[1] Jihong Yan and Zipeng Wang. Yolo v3+ vgg16-based automatic operations monitoring and analysis in a manufacturing workshop under industry 4.0. *Journal of Manufacturing Systems*, 63:134–142, 2022.

[2] Mingxing Tan, Ruoming Pang, and Quoc V Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10781–10790, 2020.

[3] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-yolov4: Scaling cross stage partial network. In *Proceedings of the IEEE/cvf conference on computer vision and pattern recognition*, pages 13029–13038, 2021.

[4] Chien-Yao Wang, I-Hau Yeh, and Hong-Yuan Mark Liao. You only learn one representation: Unified network for multiple tasks. *arXiv preprint arXiv:2105.04206*, 2021.

[5] Juan Terven and Diana Cordova-Esparza. A comprehensive review of yolo: From yolov1 to yolov8 and beyond. *arXiv preprint arXiv:2304.00501*, 2023.

[6] Linuxize. *How to Install Python 3.7 on Ubuntu 18.04*, Listopad 2022. URL: `https://linuxize.com/post/how-to-install-python-3-7-on-ubuntu-18-04/`.

[7] Linuxize. *How to Install Pip on Ubuntu 18.04*, Listopad 2022. URL: `https://linuxize.com/post/how-to-install-pip-on-ubuntu-18.04/`.

[8] zoomadmin.com. *How To Install "python3.7-venv"Package on Ubuntu*, Listopad 2022. URL: `https://zoomadmin.com/HowToInstall/UbuntuPackage/python3.7-venv`.

[9] The pip developers. *pip documentation v22.3.1*, Listopad 2022. URL: `https://pip.pypa.io/en/stable/cli/pip_freeze/`.

[10] HeleneAthene. Creating a pytorch model to fit a polynomial distribution. *https://discuss.pytorch.org/t/creating-a-pytorch-model-to-fit-a-polynomial-distribution/161595*, 2023.

[11] Jing Tao, Hongbo Wang, Xinyu Zhang, Xiaoyu Li, and Huawei Yang. An object detection system based on yolo in traffic scene. In *2017 6th International Conference on Computer Science and Network Technology (ICCSNT)*, pages 315–319. IEEE, 2017.

[12] Ala Mhalla, Thierry Chateau, Sami Gazzah, and Najoua Essoukri Ben Amara. An embedded computer-vision system for multi-object detection in traffic surveillance. *IEEE Transactions on Intelligent Transportation Systems*, 20(11):4006–4018, 2018.

[13] Rita Cucchiara, Costantino Grana, Metal Piccardi, and A Prati. Statistic and knowledge-based moving object detection in traffic scenes. In *ITSC2000. 2000 IEEE Intelligent Transportation Systems. Proceedings (Cat. No. 00TH8493)*, pages 27–32. IEEE, 2000.

[14] meituan. YOLOv6: By meituan vision department. *github https://github.com/meituan/YOLOv6*, 2023.

[15] WongKinYiu. YOLOv7: Official yolov7. *github https://github.com/WongKinYiu/yolov7*, 2023.

[16] ultralytics. YOLOv8: Ultralytics yolov8. *github https://github.com/ultralytics/ultralytics*, 2023.

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

[18] Jimin Yu and Wei Zhang. Face mask wearing detection algorithm based on improved yolo-v4. *Sensors*, 21(9):3263, 2021.

[19] Petr MIKULSKÝ. *Detekce pohybujících se objektů ve videu s využitím neuronových sítí.* 69 s. Diplomová práce. Vedoucí práce: Ing. Vojtěch Myška, Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací., Brno, 2021.

[20] ultralytics. *yolov5*, Listopad 2022. URL: `https://github.com/ultralytics/yolov5`.

[21] Yu Zhao, Yuanbo Shi, and Zelong Wang. The improved yolov5 algorithm and its application in small target detection. In *International Conference on Intelligent Robotics and Applications*, pages 679–688. Springer, 2022.

[22] Chuyi Li, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, et al. Yolov6: A single-stage object detection framework for industrial applications. *arXiv preprint arXiv:2209.02976*, 2022.

[23] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:2207.02696*, 2022.

[24] Xinyu Hou, Yi Wang, and Lap-Pui Chau. Vehicle tracking using deep sort with low confidence track filtering. In *2019 16th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2019.

[25] Ricardo Pereira, Guilherme Carvalho, Luís Garrote, and Urbano J Nunes. Sort and deep-sort based multi-object tracking for mobile robotics: evaluation with new data association metrics. *Applied Sciences*, 12(3):1319, 2022.

[26] D. Choi and K. Min. Hierarchical latent structure for multi-modal vehicle trajectory forecasting. In *Eur. Conf. Comput. Vis.*, 2022.

[27] Seong Hyeon Park, ByeongDo Kim, Chang Mook Kang, Chung Choo Chung, and Jun Won Choi. Sequence-to-sequence prediction of vehicle trajectory via lstm encoder-decoder architecture. In *2018 IEEE intelligent vehicles symposium (IV)*, pages 1672–1678. IEEE, 2018.

[28] Nachiket Deo and Mohan M Trivedi. Multi-modal trajectory prediction of surrounding vehicles with maneuver based lstms. In *2018 IEEE intelligent vehicles symposium (IV)*, pages 1179–1184. IEEE, 2018.

[29] Shengzhe Dai, Li Li, and Zhiheng Li. Modeling vehicle interactions via modified lstm models for trajectory prediction. *IEEE Access*, 7:38287–38296, 2019.

[30] Signatrix GmbH. A pytorch implementation of efficientdet object detection. *https://github.com/signatrix/efficientdet*, 2020.

[31] iscyy. YOLOAir: Makes improvements easy again. *github https://github.com/iscyy/yoloair*, 2022.

[32] WongKinYiu. Yolor. *https://github.com/WongKinYiu/yolor*, 2022.

[33] labelstud.io. *Label Studio*, Listopad 2022. URL: `https://labelstud.io/`.

[34] Taeyoung96. *Yolo-to-COCO-format-converter*, Listopad 2022. URL: `https://github.com/Taeyoung96/Yolo-to-COCO-format-converter`.

[35] mathworks.com. *Help Center*, Listopad 2022. URL: `https://uk.mathworks.com/help/stats/scatterhist.html`.

[36] matplotlib.org. *Box plot vs. violin plot comparison*, Listopad 2022. URL: `https://matplotlib.org/stable/gallery/statistics/boxplot_vs_violin.html#sphx-glr-gallery-statistics-boxplot-vs-violin-py`.

[37] matplotlib.org. *Stacked bar chart*, Listopad 2022. URL: https://matplotlib.org/stable/gallery/lines_bars_and_markers/bar_stacked.html#sphx-glr-gallery-lines-bars-and-markers-bar-stacked-py.

[38] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. Bytetrack: Multi-object tracking by associating every detection box. 2022.

[39] levan92. Python implementation of deep sort. *github https://github.com/levan92/deep*$_s$*ort*$_r$*ealtime*, 2023.

Seznam symbolů a zkratek

atd	A tak dále
YOLO	Podíváš se jen jednou – You Only Look Once
SORT	Sledování v reálném čase s jednoduchou online metodou – Simple Online Realtime Tracking
LSTM	Dlouhá krátkodobá paměť – Long Short-Term Memory
GRU	Brána rekurentní jednotky – Gated Recurrent Unit
RNN	Rekurentní neuronová síť – Recurrent Neural Network
COCO	Běžné objekty v kontextu – Common Objects in Context
GPU	Grafický procesor – Graphics Processing Unit
FLOP	Operace s plovoucí desetinnou čárkou – Floating Point Operations
FPS	Snímky za sekundu – Frames Per Second
PCB	Pseudokonvoluční páteř – Pseudo Convolutional Backbone
ST-LSTM	Prostorově-časová LSTM – Spatio-Temporal LSTM
pip	Instalátor balíčků pro Python – Package Installer for Python
venv	Virtuální prostředí – Virtual Environment
txt	Přípona textových souborů
VPN	Virtuální privátní síť – Virtual Private Network
SSH	Zabezpečený shell – Secure Shell
RAM	Paměť s libovolným přístupem – Random Access Memory
CUDA	Jednotná architektura pro výpočty na zařízení – Compute Unified Device Architecture
YAML	Přípona strukturovaného textového souboru
JSON	Přípona textového souboru. Notace objektů v JavaScriptu – JavaScript Object Notation

RPN	Síť pro předkládání návrhů v regionu – Region Proposal Network
csv	Přípona textové souboru. Hodnoty oddělené čárkou – Comma-Separated Values

Seznam příloh

A Obsah elektronické přílohy

- / — Kořenový adresář přiloženého archivu
  - compare_results — Složka programu pro vyhodnocení dat
    - compare.py — Program pro vyhodnocení dat
    - graph_generator.py
  - dataset_train.csv — Datová sada s trasováním objektů
  - detects.effDet.sh — Skript pro spuštění detekce
  - detects.Syolov4.sh
  - detects.yolor.sh
  - detects.yolov5.sh
  - docker-compose.yml — Docker compose soubor pro spuštěné Label Studio
  - generarate_trajectory_graphs.py — Skript pro vytvoření grafu
  - last-yolov5.onnx.png — Export ONNX YOLOv5
  - polynoms_0.csv — Datový soubor pro směr pohybu D0
  - polynoms_1.csv
  - polynoms_2.csv
  - polynoms_3.csv
  - polynoms_4.csv
  - polynoms_5.csv
  - prediction_research — Složka s výzkumem
    - GRU
      - GRU.py
    - LSTM
      - lstm.py
    - POLYREG
      - poly.data — Výsledný soubor s definicí polynomů
      - POLY.py
    - RNN
      - rnn.py
  - requirements — Složka se závislostmi
    - EfficientDet.requirements.txt
    - Scaled-YOLOv4.requirements.txt
    - YOLOR.requirements.txt
    - YOLOv5.requirements.txt
  - runs — Exporty z měření vytíženosti GPU
    - effRun.txt
    - Syolov4Run.txt
    - yolorRun.txt
    - yolov5Run.txt
  - server — Implementace výstupní aplikace
    - api.py
    - detection.py
    - poly.data
    - prediction.py
    - requirements.txt

- `time_measure.xls` ..... Soubor se zpracováním rychlostí detekcí
- `web_interface` ..... Složka programu pro sbírání dat
  - `data_out` ..... Prázdná složka pro výstupní data
  - `Dockerfile`
  - `interface.py`
  - `requirements.txt`
  - `templates` ..... Předlohy pro webové stránky
    - `index.html`