VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

POČÍTÁNÍ UNIKÁTNÍCH AUT VE SNÍMCÍCH

UNIQUE CAR COUNTING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PETER UHRÍN**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ROMAN JURÁNEK, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Uhrín Peter, Bc.**
Program: Informační technologie Obor: Strojové učení
Název: **Počítání unikátních aut ve snímcích**
Unique Car Counting
Kategorie: Zpracování obrazu
Zadání:

1. Seznamte se metodami detekce aut, zaměřte se na metody využívající hluboké učení
2. Navrhněte metodu pro detekci unikátních vozidel na parkovišti v záběrech z ptačí perspektivy.
3. Implementujte metodu a vytvořte aplikaci, která bude dlouhodobě sbírat statistická data.
4. Vyhodnoťte přesnost metody a demonstrujte její funkčnost.
5. Vytvořte prezentační materiály.
6. Zhodnoťte vaši práci a navrhněte možnosti pokračování.

Literatura:

- Liu et al, SSD: Single Shot MultiBox Detector, ECCV 2016
- Onoro-Rubio and R. J. Lopez-Sastre, Towards Perspective-FreeObject Counting with Deep Learning, ECCV, 2016

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Juránek Roman, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 30. října 2020

Abstrakt

Současné systémy pro počítání aut na parkovištích většinou využívají nějakých specializovaných zařízení, jako jsou například závory při vjezdu na parkoviště. Takový přístup není vhodný pro neplacená, či rezidenční parkoviště. I na těchto parkovištích může být však užitečné udržovat si přehled o jejich obsazenosti a jiných datech. Systém navržený v této práci využívá model *YOLOv4* pro vizuální detekci aut na snímcích z kamer. Následně pro každé auto vypočítá *embedding* vektor, který použije při porovnávání, jestli se na daném parkovacím místě v průběhu času auto změnilo. Výsledné informace ukládá do databáze. Z těchto dat následně systém agreguje různé statistické údaje jako jsou například celkový počet detekovaných aut, průměrná obsazenost parkoviště a průměrná doba parkování jednoho auta. Tyto údaje je možné získat pomocí REST API nebo si je zobrazit ve webové aplikaci.

Abstract

Current systems for counting cars on parking lots usually use specialized equipment, such as barriers at the parking lot entrance. Usage of such equipment is not suitable for free or residential parking areas. However, even in these car parks, it can help keep track of their occupancy and other data. The system designed in this thesis uses the *YOLOv4* model for visual detection of cars in photos. It then calculates an embedding vector for each vehicle, which is used to describe cars and compare whether the car has changed over time at the same parking spot. This information is stored in the database and used to calculate various statistical values like total cars count, average occupancy, or average stay time. These values can be retrieved using REST API or be viewed in the web application.

Klíčová slova

detekce, parkoviště, identifikace aut, počítání aut, YOLOv4, embedding, TripletLoss, siamské sítě, statistika, webová aplikace, REST API

Keywords

detection, parking lot, car identification, car counting, YOLOv4, embedding, TripletLoss, siamese network, statistics, web application, REST API

Citace

UHRÍN, Peter. *Počítání unikátních aut ve snímcích.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Roman Juránek, Ph.D.

Počítání unikátních aut ve snímcích

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doktora Romana Juránka. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Peter Uhrín
16. května 2021

Poděkování

Tímto bych chtěl poděkovat panu doktorovi Romanu Juránkovi, za pravidelné konzultace, rady týkající se tématu a celkové vedení práce. Dále bych chtěl poděkovat svým rodičům a Terce Lebedové především za psychickou podporu, obědy a neuvěřitelnou dávku trpělivosti.

Obsah

1 Úvod **3**

2 Detekce **4**
2.1 Detekce vs. klasifikace 4
2.2 Dostupné metody pro detekci 5

3 Porovnávání aut **10**
3.1 Problémy související s porovnáváním aut 10
3.2 Konvoluční neuronové sítě 12
3.3 Siamské sítě a *embedding* vektory 13

4 Návrh systému **17**
4.1 Požadavky na systém 17
4.2 Architektura systému 19
4.3 Trénovací datasety 20

5 Trénování a vyhodnocení modelů **23**
5.1 Detektor 23
5.2 Embedder 27

6 Aplikace a REST API **32**
6.1 Technologie 32
6.2 Kontejnerová struktura 34
6.3 Databázový model 37
6.4 REST API 40
6.5 Uživatelské rozhraní a funkcionalita 42

7 Zpracování snímků z parkoviště **50**
7.1 Zachování pořadí snímků 50
7.2 Detekce aut a výpočet *embedding* vektorů 53
7.3 Párování parkovacích míst 55
7.4 Zpracování detekovaných aut 58

8 Agregace statistických dat **60**
8.1 Statistiky pro celé parkoviště 61
8.2 Statistiky pro jednotlivá parkovací místa 63

9 Návrhy na vylepšení **64**

10 Závěr **66**

Literatura **68**

A První spuštění aplikace **70**

B Prezentační plakát **71**

Kapitola 1

Úvod

S příchodem automobilů do života běžných lidí se naše okolí změnilo k nepoznání. Krajina začala ustupovat dálnicím a při plánování měst se klade veliký důraz na silniční síť, která městem vede a umožňuje velikému množství lidí dopravovat se během dne z místa na místo. Nedílnou součástí automobilové dopravy je však také parkování. Ať už se jedná o rezidentní parkování, parkování v práci, či v nákupním středisku, hledání parkovacího místa může být nepříjemná záležitost.

V nákupních centrech bývají často k vidění před parkovištěm závory, přes které musí auto jak vjet na parkoviště, tak z něj vyjet, což umožňuje jednoduché počítání aut aktuálně se nacházejících na parkovišti. Poté je velice jednoduše možné zobrazit kupříkladu počet volných parkovacích míst. Takovýto systém je vhodný pro placená, udržovaná parkoviště, ale fakt, že pro jeho fungování jsou potřeba závory nebo jiné speciální zařízení ho dělá méně použitelným pro parkoviště neplacená, kde by se investice do podobných zařízení nevyplatila. Zároveň, proto, že kontroluje a počítá auta pouze při vjezdu a výjezdu, nemá tento systém přehled o jednotlivých parkovacích místech, pouze jen o parkovišti jako celku.

Druhou disciplínou, ve které dříve zmíněné systémy zaostávají, je sbírání statistických údajů z daného parkoviště. Na první pohled se může zdát, že sbírání dat nepřináší viditelné výhody, ale zde pouze záleží je jejich interpretaci. Ukládání si obsazení parkovacích míst na parkovišti může kupříkladu pomoct s plánováním rozšíření daného parkoviště. Pakliže bude takové sbírání dat probíhat delší dobu, je možné z nich vysledovat určité vzory a nabízet řidičům informace nejen o aktuální obsazenosti, ale také o předpokládané obsazenosti v příštích hodinách, což pro ně může být ještě užitečnější informace. Informací, které se dají vydolovat z dlouhodobě sbíraných dat je zkrátka nespočet.

Systém navržený a implementovaný v této práci se zaměří na řešení právě těchto dříve zmíněných nedostatků. Nebude spoléhat na speciální zařízení jako jsou závory na parkovišti. Místo toho bude využívat vizuální detekci aut na snímcích z kamer na parkovišti. Zároveň bude také umožňovat sbírání a vyhodnocování statistických dat pro jednotlivá parkoviště.

Kapitola 2

Detekce

Základním problémem, který je potřeba vyřešit, pakliže chceme počítat auta čistě na základě obrazových dat z parkoviště je samozřejmě jejich detekce. Lidé dokážou na základě zrakových vjemů detekovat obrovské množství objektů, včetně jejich polohy, vizuálních vlastností, kategorie, a podobně. Toho jsme schopni docílit hlavně díky nesmírně komplexnímu vizuálnímu systému, kterým jsme jako lidé obdařeni. Tento systém je určen přesně na tento typ úloh. Není proto divu, že pro počítače, kterým takto úzce zaměřený vizuální systém chybí, je úkol detekce objektů poměrně náročný. Naštěstí ale i přesto existují metody, pomocí kterých je počítačová detekce objektů ve fotografiích nejenom možná, ale v některých oblastech dokonce překonává schopnosti lidí.

2.1 Detekce vs. klasifikace

S oborem počítačového vidění jsou úlohy typu klasifikace a detekce úzce spjaty. Ačkoliv se ale na první pohled můžou zdát tyto dvě úlohy stejné, je potřeba brát v potaz, že se jedná o dvě různé úlohy, které se řeší různými způsoby a jejichž výstupem jsou různé věci.

Klasifikace

Klasifikace je pravděpodobně známější z těchto dvou úloh. Cílem klasifikace je rozhodnout, zda-li se na nějakém obrázku nachází objekty nějakého předem definovaného typu, popřípadě které objekty to jsou.

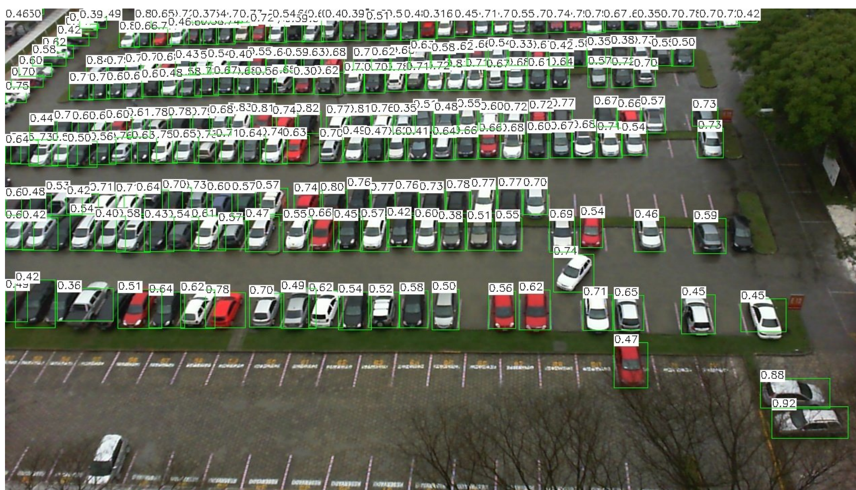
Typický příklad klasifikační úlohy může znít *"Zjisti, jestli se na obrázku nachází slon indický, nebo africký"*. Není podstatné, kde na obrázcích se sloni nachází, ani další dodatečné informace. Jde nám čistě o klasifikování objektu, nacházejícího se kdekoliv na obrázku. Výsledkem takovéto úlohy by tedy byla pravděpodobnostní hodnota značící to, s jakou pravděpodobností se na obrázku nachází slon africký.

Detekce

Detekce z klasifikace částečně vychází a rozšiřuje ji. V detekčních úlohách nestačí pouze klasifikovat objekt na obrázku. Výstupem detekce jsou oproti klasifikaci navíc i detekované rámce, označující oblast na obrázku, kde se daný objekt nachází. Na obrázku samozřejmě může být více stejných i různých objektů.

Typickým příkladem detekční úlohy by tedy mohlo být *"Spočítej auta na parkovišti"*. Výstupem takové úlohy by pak byl rámec ohraničující každé auto na parkovišti spolu s prav-

děpodobnostní hodnotou, že se opravdu jedná o auto. Pakliže bychom chtěli auta spočítat, jednoduše bychom spočítali detekované rámce.



Obrázek 2.1: Příklad detekování aut na parkovišti.

Klasický postup pro detekci objektů na obrázku tvoří několik za sebou jdoucích částí:

- Generování zájmových oblastí na obrázku, kde by se mohl nacházet nějaký klasifikovatelný objekt. Toto je nejsložitější část z celého procesu. Klasickým přístupem bylo generování velikého množství potenciálních oblastí, přičemž většina byla ve skutečnosti na konci klasifikovaná jako pozadí.
- Extrakce příznaků ze zájmových oblastí obrázku.
- Klasifikace objektů v zájmových oblastech obrázku pomocí extrahovaných příznaků.

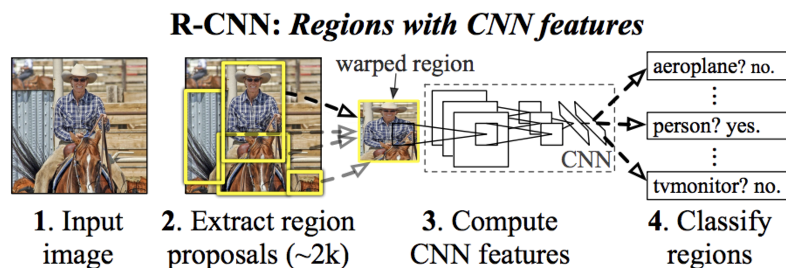
2.2 Dostupné metody pro detekci

Ačkoliv je řešení úloh, jako je detekce nebo klasifikace, často spojované s hlubokými neuronovými sítěmi, pravdou je, že už před nástupem hlubokých neuronových sítí existovaly algoritmy, které byly určené pro řešení právě těchto problémů. Takovými algoritmy jsou například *Scale-invariant feature transform* (SIFT), či *Histogram of oriented gradients* (HOG). Ačkoliv tyto metody fungují dobře, jejich nevýhodou je, že vzory, podle kterých tyto metody vyhledávaly objekty v obrázcích musely být často předem specifikovány. U značně složitých objektů je pak obtížné vymyslet takový mechanismus, který bude extrahovat relevantní charakteristické znaky.

Hluboké neuronové sítě jsou naproti tomu schopné naučit se nejenom detekovat a rozeznávat objekty pomocí předem specifikovaných charakteristických znaků, ale jsou schopny zároveň přijít na to, jak extrahovat nové, potenciálně lepší, charakteristické znaky a vzory. V této kapitole budou popsány především novější, v praxi používané metody pro detekci objektů v obraze založené právě na hlubokém učení.

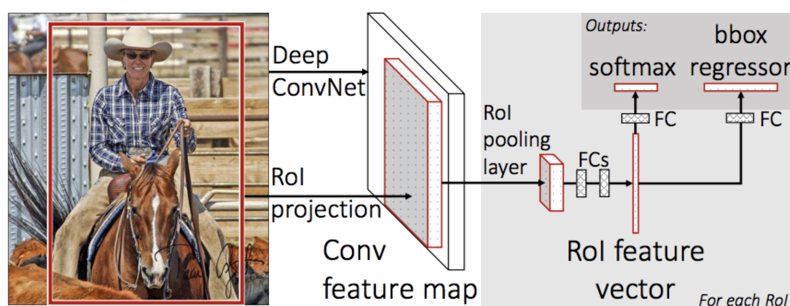
R-CNN, Fast R-CNN, Faster R-CNN

V roce 2014 byla publikována práce *Rich feature hierarchies for accurate object detection and semantic segmentation* [8]. Obsahem této práce byl detekční model, který se sice držel do té doby používaných postupů pro detekci, avšak pro extrakci příznaků používal konvoluční neuronovou síť. Pro generování zajímavých oblastí v obrázku využívala algoritmus *Selective Search*[1] a pro konečnou klasifikaci pak algoritmus *Support-vector machine*[2]. Hlavním nedostatkem tohoto modelu však bylo to, že každá z jeho tří částí byla samostatná. Model tedy nemohl být trénovaný celý najednou.



Obrázek 2.2: Architektura *R-CNN* modelu. Extrakce příznaků je prováděná pomocí konvoluční neuronové sítě. (Zdroj: [8])

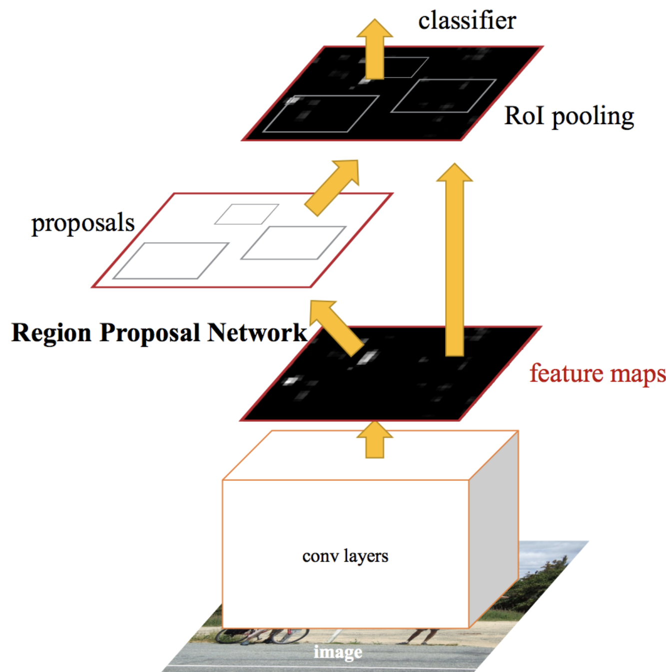
Model *Fast R-CNN* [7] byl představen v roce 2015 a řešil některé z největších nedostatků původního modelu *R-CNN*. Nový model počítal příznaky všech objektů (celého obrázku) najednou, jediným průchodem konvoluční sítí. Navíc bylo nově zavedeno používání vrstvy *Region of interest pooling*[3], která měla za úkol veškeré zajímavé oblasti převést na vektory příznaků o pevné velikosti. To následně vedlo k dalšímu zrychlení zpracování jednotlivých vektorů s příznaky, pomocí klasických hustě propojených vrstev neuronové sítě. I přes toto zrychlení však model *Fast R-CNN* stále využíval poměrně pomalý algoritmus *Selective Search*.



Obrázek 2.3: Architektura *Fast R-CNN* modelu. Příznaky jsou extrahovány pro celý obrázek najednou. Zároveň se před zpracováním jednotlivých zájmových oblastí používá vrstva *ROI pooling*. (Zdroj: [7])

[1] https://www.learnopencv.com/selective-search-for-object-detection-cpp-python/
[2] https://en.wikipedia.org/wiki/Support-vector_machine
[3] https://deepsense.ai/region-of-interest-pooling-explained/

Model *Faster R-CNN* [16] spatřil světlo světa v roce 2017. Zásadním vylepšením oproti modelu *Fast R-CNN* je použití konvoluční sítě *Region proposal network*[4]. Tato síť nahrazuje algoritmus *Selective Search* při generování zájmových oblastí v obrázku. RPN síť využívá konceptu takzvaných *Anchor* boxů, které jsou definované podle předem zvolených hodnot škály a poměru stran. Při procházení mapy příznaků, která je výstupem konvoluční sítě, jsou na základě těchto *anchor* boxů generovány zájmové oblasti. Následují dva průchody přes mapu příznaků. První pro nalezení správných souřadnic výsledných rámců a druhý pro klasifikaci objektů v nich.



Obrázek 2.4: Architektura *Faster R-CNN* modelu. Zájmové oblasti jsou pomocí vrstvy RPN generovány z *anchor* boxů. Zároveň se mapa příznaků vypočítá pouze jednou a je znovu-použita jak pro generování zájmových oblastí, tak pro klasifikaci. (Zdroj: [16])

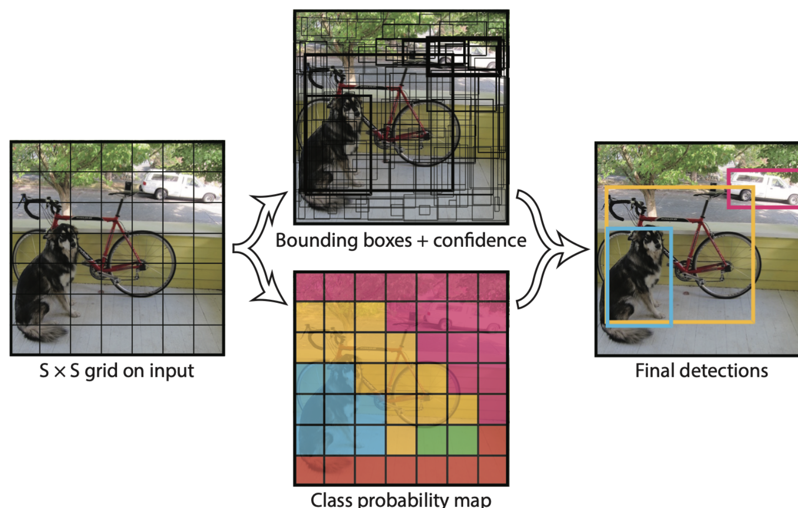
YOLO

Modely *R-CNN*, *Fast R-CNN* a *Faster R-CNN* zpracovávají obrázky ve dvou krocích. Nejprve detekují možné polohy obrázků a následně klasifikují, jaké objekty se na těchto místech nachází. Model *You Only Look Once* [15], neboli *YOLO* byl představen v roce 2015 a jeho hlavním přínosem v době uvedení bylo sjednocení těchto dvou kroků. Detekce i klasifikace objektů probíhá v modelu *YOLO* pomocí jediného průchodu sítí. V době představení tím dosáhl dosud nevídané rychlosti při zpracování vstupních obrázků.

V první verzi modelu *YOLO* [15], uvedené v roce 2015, byl obrázek rozdělen do mřížky $S \times S$ buněk. Pakliže se střed detekovaného objektu nacházel uvnitř nějaké buňky, byla tato buňka zodpovědná za jeho detekci a vygenerovala pro tento objekt B detekovaných rámců, přičemž každý v sobě kromě souřadnic rámce obsahoval také skóre, určující pravděpodobnost, že se v daném rámci nachází objekt. Následuje klasifikace objektu v každé buňce.

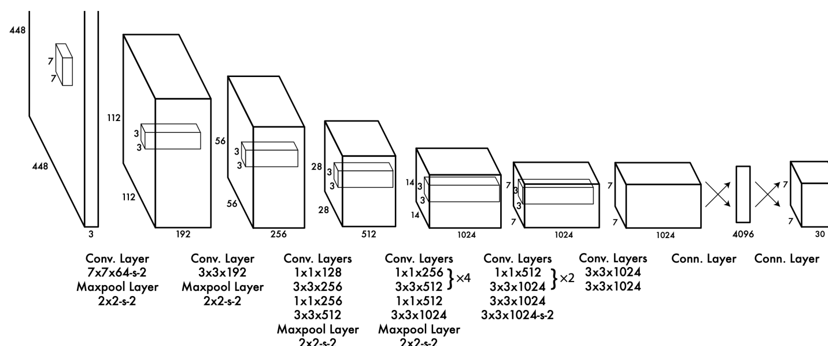
[4] https://medium.com/egen/region-proposal-network-rpn-backbone-of-faster-r-cnn-4a744a38d7f9

Nakonec jsou vygenerované detekční rámce zredukované pomocí algoritmu *Non-maximum Suppression* [5] do výsledné podoby.



Obrázek 2.5: Ukázka způsobu zpracování obrázku původním modelem *YOLO*. Obrázek je rozdělen do mřížky 5 × 5. Následně jsou pro každou buňku vygenerované detekční rámce a nakonec jsou tyto rámce zredukované do výsledné podoby. (Zdroj: [15])

V původním článku [15] autoři navrhli model jako jedinou konvoluční neuronovou síť, jejíž architektura je zobrazená na obrázku 2.6. Síť se dohromady skládá z 24 konvolučních vrstev a dvou plně propojených vrstev na konci.



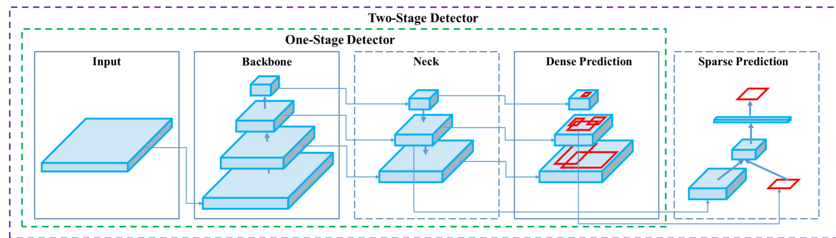
Obrázek 2.6: Architektura původního modelu *YOLO*. (Zdroj: [15])

Hlavní výhodou modelu *YOLO* tedy byla především rychlost, pramenící ze souběžné detekce ve všech buňkách mřížky. Tento systém však měl i své nedostatky. Autoři zvolili mřížku o velikosti 7 × 7 buněk, přičemž každá buňka byla schopna detekovat jeden objekt. Maximální počet objektů, který tedy síť mohla detekovat byl 49. Pokud bylo zároveň v jedné buňce přítomných více objektů najednou, nebylo možné detekovat všechny.

Řešení těchto problémů a spoustu dalších vylepšení bylo v průběhu času zapracováno do několika verzí modelu *YOLO*. V současné době se nejčastěji používá model *YOLOv4* [4].

[5] https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c

Tato verze pracuje na podobném základu jako původní model *YOLO*. Pro detekci nejprve rozdělí obrázek do mřížky a následně pro každou buňku vygeneruje předem daný počet *anchor* boxů, které slouží jako základ pro predikci souřadnic detekovaných rámců.



Obrázek 2.7: Architektura modelu *YOLOv4*. (Zdroj: [4])

Na obrázku 2.7 je architektura modelu rozdělena do několika částí. Část s názvem *backbone* slouží pouze k extrakci příznaků pomocí konvolučních vrstev. Na ni navazuje *neck*, který slouží k lepšímu předávání extrahovaných příznaků do poslední vrstvy s názvem *head*. Tato vrstva na základě příznaků dělá konkrétní predikce rámců a klasifikace objektů v nich.

Kapitola 3

Porovnávání aut

Dalším důležitým krokem při zpracovávání fotografií z parkoviště a dolování relevantních informací z nich je porovnávání aut, které se v sekvenci snímků nachází na stejných parkovacích místech.

3.1 Problémy související s porovnáváním aut

Porovnávání aut na dvou různých snímcích nemusí být tak triviální, jak se na první pohled zdá. Protože se v průběhu času na fotografiích mírně mění jak světelné podmínky, tak v některých případech dokonce i scenérie kolem, je zřejmé, že není možné porovnávat fotografie přímo pixel po pixelu.

Světelné podmínky

Základním problémem, který člověka okamžitě napadne při porovnávání dvou fotografií, které byly pořízeny v určitém časovém rozestupu, je rozdílnost světelných podmínek. Na první pohled je zřejmý kontrast mezi fotografií ve dne a fotografií téhož objektu v noci. Samozřejmě lze namítat, že pakliže je interval pořízení dvou fotografií dostatečně krátký, tyto situace nenastanou. Problém týkající se světelných podmínek je však širší.

Na obrázku 3.1 je možné vidět výraznou změnu ve světelných podmínkách. Fotografie byly přitom pořízeny s rozestupem pouhých pěti minut, což je poměrně krátká doba a i přesto by už zde přímé porovnání aut nefungovalo spolehlivě.

Obrázek 3.1: Příklad výrazné změny vzhledu auta, kvůli změně světelných podmínek.

Okolí aut

Samotná přítomnost jiných aut na parkovišti může při porovnávání také působit potíže. Výstupem detektoru aut jsou rámy, které popisují, kde se auto na snímku nachází. Tyto rámy mají tvar obdélníků a tím pádem neopisují těsně tvar auta. Tato malá nedokonalost způsobuje, že se na okrajích výřezu určeného detekovaným rámem mohou nacházet kousky jiných parkovacích míst. To samo o sobě nevadí, avšak jakmile se na těchto okolních parkovacích místech něco změní, tedy například vedlejší auto odjede, nebo nové naopak přijede, změní to také okraj fotografie, podle které se systém snaží porovnat auta. To jaký vizuální rozdíl může udělat výměna aut parkujících na vedlejších parkovacích místech je demonstrováno na obrázku 3.2.

Obrázek 3.2: Příklad výrazné změny okolí auta.

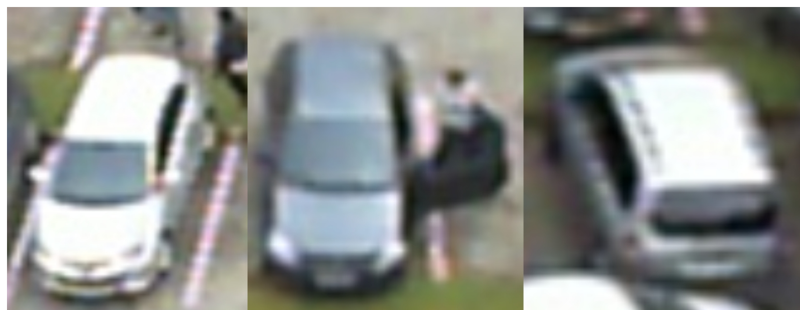
Prvotní nápad, jak by bylo možné tento problém řešit je jednoduché zmenšení rámce, který byl nalezen detektorem tak, aby nezasahoval do okolních parkovacích míst. Tento způsob má však výraznou vadu. V případě, že není auto, které se systém snaží porovnat zaparkováno přesně kolmo ke kameře, pak by oříznutí rámce tak, aby neobsahoval části okolních parkovacích míst, vedlo ke smazání veliké části samotného auta, které je porovnáváno, a tím i ke ztrátě důležitých charakteristických znaků daného auta. Ztráta těchto charakteristických znaků může omezit, ne-li úplně znemožnit spolehlivé porovnávání aut.

Cizí tělesa

Poslední veliký problém při porovnávání aut na parkovišti je fakt, že parkoviště nejsou přesně kontrolovaná prostředí. Může se tedy poměrně jednoduše stát, že se na fotografii parkoviště vyskytnou nějaká cizí tělesa.

Na obrázku 3.3 jsou ukázány některé příklady cizích těles, které se můžou na parkovišti vyskytovat. Obecně platí, že pakliže se cizí objekt nepohybuje přímo před některým autem, tak při detekci a porovnávání nehraje roli. V případě opačném však může částečně zakrývat porovnávané auto a některé jeho významné charakteristiky. To se může opět podepsat na snížené přesnosti porovnávání dvou stejných aut.

Jak je vidět na obrázku 3.3, nejběžnějšími cizími objekty na parkovišti jsou ještě nezaparkovaná auta a lidé, pohybující se před už zaparkovanými vozidly. Pakliže je snímek zachycen v nevhodnou chvíli, mohou otevřené dveře automobilu a lidé, kteří na fotografii zrovna vystupují z auta, zcela změnit geometrii porovnávaného automobilu.



Obrázek 3.3: Příklad cizích těles zachycených na snímku auta.

3.2 Konvoluční neuronové sítě

Problémy popsané výše nejsou specifické jen pro porovnávání aut na parkovišti. Asi nejznámější úlohou, ve které bylo potřeba překonat popsané problémy je úloha klasifikace obrázků.

Cílem klasifikace je zařazení vstupních vzorků do jedné nebo více z předem definovaných kategorií. Příkladem takovéto úlohy muže být rozpoznávání ručně psaných číslic [13]. Jak je vidět na ilustračním obrázku 3.4, je potřeba při řešení této konkrétní úlohy zohlednit problém, kde každá ručně psaná číslice patřící do stejné třídy vypadá trochu jinak. Proto, podobně jako u problémů popsaných výše, nemohlo být použito jednoduché porovnávání pixel po pixelu.

Obrázek 3.4: Příklad ručně psaných číslic z datasetu MNIST. (Zdroj: [13])

Místo toho se pro řešení podobných úloh osvědčilo použití konvolučních neuronových sítí [17], které jsou schopné tyto problémy při klasifikaci překonat.

Konvoluční neuronové sítě určené pro klasifikaci se zpravidla skládají ze dvou hlavních částí. První částí je sada konvolučních vrstev. Druhou pak samotný klasifikátor, většinou tvořený jednou nebo více hustě propojenými vrstvami [5].

To, proč jsou konvoluční neuronové sítě tak úspěšné při klasifikačních úlohách, tkví právě v první části těchto sítí, a tedy v konvolučních vrstvách. Tyto vrstvy využívají operace konvoluce[1] k úpravě vstupních fotografií. Správně naučená konvoluční síť je pomocí konvolučních vrstev schopna ze vstupní fotografie extrahovat vzory, které jsou nějakým

[1]https://en.wikipedia.org/wiki/Convolution

způsobem výrazné a charakteristické pro daný objekt. Nejdůležitější výhody vzorů extrahovaných pomocí konvolučních vrstev jsou následující [5]:

- Extrahované vzory jsou invariantní vůči poloze, na které se v obrázku nachází. Jinými slovy tedy nezáleží na tom, kde na obrázku se daný objekt vyskytuje, nebo jak je natočen.
- Vzory naučené pomocí série za sebou následujících konvolučních sítí mají hierarchickou strukturu. Vzory extrahované první vrstvou jsou většinou velice jednoduché. Vzory z každé další vrstvy jsou skládány ze vzorů předchozích a jsou tedy čím dál složitější. Pakliže je vrstev dostatečné množství, jsou vzory, extrahované poslední vrstvou schopny popsat i značně komplexní objekty.

Druhá část konvolučních neuronových sítí je samotný klasifikátor. Tato část je naučená tak, aby ze vzorů extrahovaných pomocí konvolučních vrstev určila pravděpodobnost, s jakou se na daném obrázku nachází objekt příslušící k jednotlivým třídám.

Problém porovnávání aut však není tak docela klasifikační problém. Klasifikátory fungují dobře, pokud mají předem definovanou množinu tříd, do kterých můžou klasifikovat a nad kterými jsou naučeny. V případě aut parkujících na parkovišti ale tohle není prakticky proveditelné, protože pro každé auto bychom museli mít vytvořenou samostatnou třídu. Pakliže by přijelo nějaké nové auto, musel by být klasifikátor přetrénován, aby zahrnoval novou třídu.

To, že pro problém porovnávaní aut není možné použít klasifikátor, ale neznamená, že není možné použít konvoluční vrstvy. Ty z obrázků extrahují vzory nezávisle na tom, jestli jsou pak používány klasifikátorem, nebo jako vstup pro jinou metodu.

3.3 Siamské sítě a *embedding* vektory

Klasifikátor používá vzory, nalezené konvolučními vrstvami jako jakési abstraktní popisy vstupního obrázku. Během trénování se klasifikátor naučí, které z těchto vzorů jsou nejdůležitější a nejvíce přispívají ke správnému určení třídy, která je na obrázku zastoupená. Výsledek tohoto učení je zakódován do vektoru, který obsahuje pravděpodobnost výskytu jednotlivých tříd na obrázku.

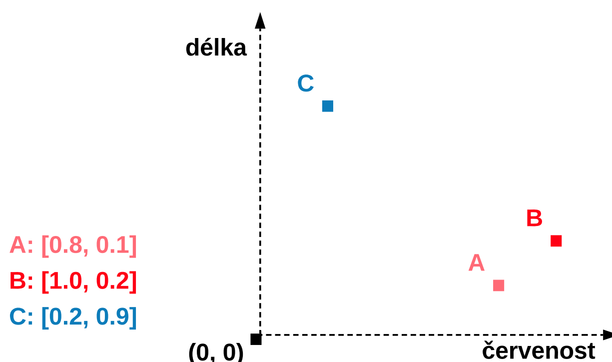
Při porovnávání dvou obrázků sice nemůžeme použít klasifikaci jako vhodnou metodu, ale můžeme se jí inspirovat.

Embedding vektor

Pokud vzory, které konvoluční vrstvy naleznou, stačí pro klasifikaci, mohly by stačit i pro porovnání. Podobně jako se klasifikátor musí naučit, které vzory nejvíce přispívají k určení dané třídy a tento výsledek zakóduje do vektoru pravděpodobností, je možné naučit síť, aby zakódovala nejdůležitější vzory do vektoru, popisujícího nějaké charakteristické vlastnosti daného objektu.

Pro zjednodušení si lze představit například dvourozměrný vektor popisující auto. První souřadnice daného vektoru označuje, jak moc je auto červené a druhá souřadnice, jak je dlouhé. Pokud tedy zakódujeme několik obrázků aut do těchto vektorů, měla by mít všechna auta, která jsou červená, vysokou hodnotu na první souřadnici ve vektoru a všechna auta, která jsou dlouhá, vysokou hodnotu na druhé souřadnici ve vektoru. Tento příklad je vyobrazen na obrázku 3.5. Jednotlivá auta jsou zde popsaná jako body v dvourozměrném

prostoru, definované svými vektory. Je zde patrná důležitá vlastnost tohoto zakódování. Dvě podobná auta mají také podobné vektory.



Obrázek 3.5: Příklad vlastností embedding vektorů

Porovnání těchto aut se poté může provést velice jednoduše, jako výpočet euklidovské vzdálenosti dvou bodů v prostoru[2]. Pakliže je vzdálenost menší, než předem zvolený práh $\Theta_{similarity}$, pak jsou auta prohlášena za stejná. V rovnici 3.1 označují X a Y obrázky porovnávaných aut a f je funkce počítající výsledný vektor.

$$\|f(X) - f(Y)\| < \Theta_{similarity} \tag{3.1}$$

Této technice zakódování obrázku do vektoru tak, aby dva podobné obrázky měly také podobné vektory se říká *embedding*[3]. Obecně jsou však tyto vektory delší než v uvedeném příkladě a význam jednotlivých os nemusí být vůbec zřejmý.

Embedding vektory byly v roce 2015 využity v síti *FaceNet* [18] k zakódování obrázků lidských tváří a k jejich porovnávání. V práci *Vehicle Re-Identification: an Efficient Baseline Using Triplet Embedding* [12], která vyšla v roce 2019, byla tato technika použita přímo na porovnávání aut.

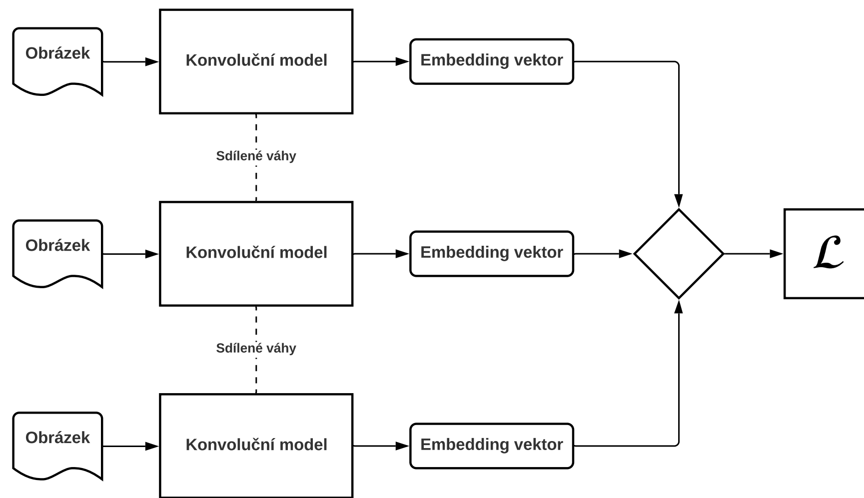
Siamské sítě

Opět je zde ale problém. Klasické neuronové sítě potřebují ke trénování sadu trénovacích dat, která jsou náležitě anotována. V případě klasifikace mají anotace formu vektoru, ve kterém jsou správně zaznačené pravděpodobnosti příslušnosti obrázku do jednotlivých tříd. V případě *embedding* vektorů však dopředu nevíme, jaké hodnoty má tento vektor nabývat, protože to, podle kterých atributů obrázku jsou *embedding* vektory vypočítané je dáno právě až učením. Nemáme tedy možnost, jak manuálně vytvořit anotace ve formě *embedding* vektorů.

Řešením tohoto problému může být použití takzvaných siamských neuronových sítí [11]. Princip siamských neuronových sítí spočívá v použití dvou nebo i více neuronových sítí, které jsou na konci spojeny jednou společnou ztrátovou funkcí. Sítě mají navzájem sdílené váhy. Pokud se tedy během trénování jejich hodnota upravuje, projeví se tato úprava ve všech. Příklad schématu takové sítě je vidět na obrázku 3.6. Architektura siamské sítě nám

[2]https://en.wikipedia.org/wiki/Euclidean_distance
[3]https://en.wikipedia.org/wiki/Embedding

umožňuje na vstup vložit několik obrázků najednou s tím, že všechny budou zpracovány stejně.



Obrázek 3.6: Schéma siamské neuronové sítě

To co je ale při této architektuře nejpodstatnější, je právě ztrátová funkce. Vhodně zvolená ztrátová funkce umožní transformování problému učení se vytváření *embedding* vektorů na problém porovnávání výstupů jednotlivých sítí v siamské architektuře, podle jejich předem známé příslušnosti ke třídě. Jednou z běžně používaných ztrátových funkcí pro siamské sítě je funkce TripletLoss [18].

TripletLoss

Intuice za funkcí TripletLoss je velice jednoduchá. Jak již bylo popsáno výše, *embedding* je v podstatě taková funkce f, jejíž výstupem je vektor, popisující obrázek x jako bod v N-dimenzionálním prostoru $\mathbb{R}^n$, přičemž vzdálenost bodů v tomto prostoru vyjadřuje podobnost daných obrázků. Obecně tedy platí, že $f(x) \in \mathbb{R}^n$ a vzdálenost *embedding* vektorů reprezentujících podobné obrázky by měla být malá, kdežto vzdálenost *embedding* vektorů reprezentujících odlišné obrázky by měla být veliká.

Máme-li tři obrázky nazvané *Anchor* (A), reprezentující vztažný obrázek, *Positive* (P), reprezentující obrázek, který je podobný vztažnému obrázku A a nakonec *Negative* (N) obrázek, který se vztažnému obrázku A nepodobá, pak můžeme tento požadovaný vztah formálně zapsat rovnicí 3.2. Symbol α zde značí vynucenou vzdálenost mezi vzájemně nepodobnými snímky.

$$\|f(A) - f(P)\|^2 + \alpha < \|f(A) - f(P)\|^2 \tag{3.2}$$

Funkce TripletLoss pracuje právě s těmito trojicemi *embedding* vektorů na vstupu. Úkolem ztrátové funkce je penalizovat takové případy, kdy je *embedding* vektor $f(P)$ vzdálený vektoru $f(A)$ nebo když je vektor $f(N)$ příliš blízko vektoru $f(A)$. Výsledná ztrátová funkce TripletLoss, která byla použita v síti *FaceNet* [18], je znázorněná rovnicí 3.3.

$$\mathcal{L}(A, P, N) = min(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0) \tag{3.3}$$

Je zřejmé, že pro co nejrychlejší učení je potřeba volit takové trojice obrázků A, P, N, které nejvýrazněji porušují vztah uvedený rovnicí 3.2. Nejrychlejší učení však nemusí vyústit v nejlepší výsledky. Pakliže bychom skutečně vybrali takové trojice, kde se obrázky A a P podobají nejvíce a zároveň se obrázky A a N podobají nejméně v celém datasetu, vyvstalo by riziko, že budou neustále voleny takové obrázky N, které bychom za normálních okolností označili jako odlehlé hodnoty. Síť by se tedy mohla učit na extrémech, což by nevedlo k dobré generalizaci.

Naštěstí existuje několik možných technik, které se v praxi využívají pro řešení tohoto problému [18]:

- Takzvaným *offline* generováním trojic se rozumí technika, kdy jsou jsou trojice nalezeny před samotným trénováním sítě. Všechny obrázky z datasetu jsou postupně před-rozděleny do trojic, avšak výběr trojic může probíhat pouze nad podmnožinou obrázků, která se postupně mění. Tím se zajistí, že se pro obrázek N nevyberou vždy jen ty stejně odlehlé hodnoty.

- Během *online* generování trojic nejsou trojice vytvořeny z celého datasetu před zahájením trénování. Místo toho jsou generovány dynamicky z mini-dávek během samotného procesu trénování. Podobně jako u *offline* generování se i tady původní problém odlehlých hodnot řeší použitím mini-dávek místo celého datasetu najednou.

Kapitola 4

Návrh systému

Jak již bylo nastíněno v úvodu, výsledný systém musí být schopen bez použití speciálních zařízení, jako jsou závory při vjezdu na parkoviště a při výjezdu z něj, detekovat auta, která se na parkovišti nachází. Samotná detekce aut na parkovišti však nestačí. Druhý požadavek na systém, který byl zmíněný v úvody se týká sbírání statistických dat. Taková data je možné sbírat pouze za předpokladu, že máme k dispozici časovou řadu záznamů z daného parkoviště. Následně je navíc potřeba, aby byl systém schopen poznat, zda se na stejném místě na parkovišti nachází pořád stejné auto, nebo přijelo auto nové.

Je tedy zřejmě, že systém musí provádět řadu různých úkonů. V této kapitole bude systém popsán jako celek a důraz bude kladen hlavně na to, jakým způsobem by měly jednotlivé části spolupracovat, aby dosáhly požadovaného výsledku.

4.1 Požadavky na systém

Před samotným návrhem architektury systému je potřeba specifikovat, co se od systému očekává. Tato sekce proto slouží jako definice požadavků na systém.

Detekce aut na fotografiích

Princip počítání aut standardních systémů využívajících závory při vjezdu a výjezdu je velice jednoduchý. Pokaždé, když přijede nějaké auto a závora se pro něj tedy musí otevřít, započítá se toto auto k celkovému součtu aut, aktuálně parkujících na parkovišti. Pokud auto naopak odjíždí, musí opět opustit parkoviště skrze závoru a systém si jej tedy může odečíst z parkujících aut.

Pakliže je jedním z hlavních kritérií to, aby systém nevyžadoval instalaci speciálních zařízený, jako jsou například závory při vjezdu a výjezdu, nabízí se vizuální detekce pomocí obrazu z kamery zabírající celé parkoviště. Systém detekce a počítání aut není tak jednoduchý a přímočarý jako počítání aut za použití závor, má však několik výrazných výhod.

Kamery se v průběhu času staly velice dostupné. Ve světě, kde každé lyžařské středisko má několik kamer, které přenáší živé záběry ze sjezdovek na internet, není problém nainstalovat jednoduchou kameru v podstatě kamkoliv. Kamera sama o sobě navíc nemusí být ničím speciální. Na rozdíl od počítání aut při vjezdu na parkoviště a jejich výjezdu z něj za pomoci závor, je však v případě záznamů z kamer parkoviště potřeba fotografie zpracovat. Prvním a základním krokem při zpracování fotografie zachycující celé parkoviště proto je detekce částí, na kterých se nachází auta.

Detekcí objektů na obrázku se myslí nalezení takového regionu na obrázku, ve kterém se daný objekt nachází. Nejčastěji se tedy jedná o obdélníkový výřez z fotografie, který je specifikovaný buď souřadnicemi jednotlivých bodů, nebo kombinací souřadnic jednoho předem specifikovaného bodu (například levého horního bodu) a šířky a výšky výřezu. Takových výřezů na fotografii může být hned několik a každý by měl reprezentovat jeden detekovaný objekt. V případě systému na detekci aut na parkovišti nebude potřeba detekovat nic jiného než auta.

Konkretní techniky, pomocí kterých lze detekovat objekty na fotografiích či videu, byly popsány v kapitole 2.

Porovnání aut

Jakmile systém na fotografii parkoviště detekuje jednotlivá auta a jejich pozice, mohlo by se zdát, že je vše hotovo. Přeci jenom, jakmile je detekce hotová, stačí nám spočítat počet detekovaných objektů a máme počet aut, aktuálně se nacházejících na parkovišti. Taková data jsou jistě užitečná, ale není možné z nich vydolovat všechny informace, které by systém mohl nabídnout.

Pakliže bychom například chtěli zjistit, kolik aut celkem bylo na parkovišti daný den, jednoduché spočítání aut na každém snímku z daného dne by nám neřeklo to, co jsme chtěli vědět. Na prvním snímku by systém napočítal dejme tomu 100 aut. Na druhém pak 101. Znamená to tedy, že přibilo pouze jedno auto? Co když ale v čase mezi pořízením prvního a druhého snímku pět aut odjelo a šest aut přijelo? Potíž je v tom, že pokud by systém pouze počítal detekovaná auta, nebyl by schopen tyto situace rozlišit.

Řešením tohoto problému může být porovnávání detekovaných aut na stejných pozicích na parkovišti mezi dvěma pořízenými snímky. Pokud bude systém schopen určit, že ve dvou snímcích bylo na stejné pozici na parkovišti detekováno auto, a porovnat tyto dvě auta, bude také schopen určit, jestli se jedná o totéž auto a nemělo by být znovu započítáno, nebo se jedná o auto nové a v tom případě se korektně jako nové započítá.

Na první pohled se jedná o jednoduchou úlohu. Prvotní instinkt by mohl říkat, že stačí porovnat výřezy aut přímo, pixel po pixelu. Takhle jednoduché to však není. Stačí, aby mezi pořízením dvou snímků přiletěl mrak a celá fotografie může mít jiné vyvážení bílé a tedy i jiné barevně odstíny, nebo se na fotografii můžou objevit nové stíny tam, kde předtím nebyly. Tyto změněné podmínky můžou navíc ovlivnit už i detekci a tudíž ani zarovnání detekovaných aut nemusí mezi dvěma snímky přesně souhlasit.

Detaily těchto problémů a jejich řešení při porovnávání aut na snímcích byly podrobněji vysvětleny v kapitole 3.

Sbírání a uchovávání statistických údajů

Většina údajů, které by měl systém shromažďovat bude generována už v předchozích dvou částech procesu. Tato okamžitá data získaná pouze z jednoho snímku najednou však neříkají nic o trendech a dlouhodobých statistikách. Dokonce už ve fázi porovnávání aut potřebuje systém informace o předchozím stavu parkoviště.

Je tedy zřejmé, že veškerá potřebná data bude třeba uchovávat jak k pozdějšímu využití samotným systémem, tak pro dolování statistických informací z nich.

Rozhraní systému

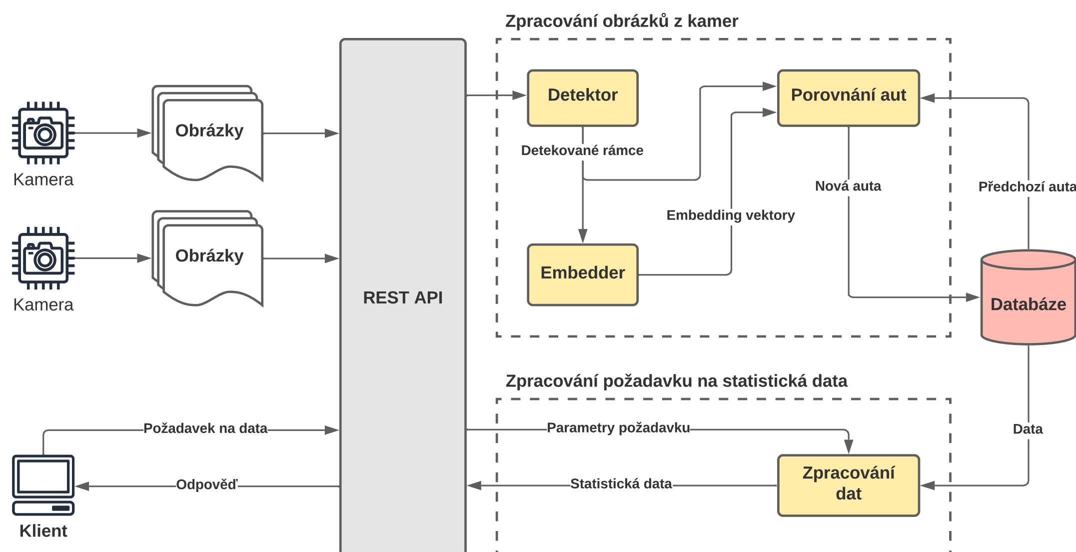
Již v úvodu byly popsané některé výhody vyplývající se systému, který využívá ke sbírání dat o zaplnění parkoviště obrazový výstup jednoduché kamery. Jedna z velikých výhod tohoto přístupu však v úvodu zmíněná nebyla. Tím je možnost vzdáleného zpracování obrazových dat.

Systém sám o sobě, na místě kde se parkoviště nachází, nepotřebuje vyhodnocovat žádná data. Pakliže vezmeme v úvahu možnost, že systém bude využíván na několika parkovištích současně, znamená to další ušetřené výdaje pro zprovozněni. Není totiž na každé lokalitě potřeba samostatná stanice, která by data zpracovávala. Místo toho je možné mít pouze jednu centrální stanici, která bude přijímat data z kamer na jednotlivých parkovištích a zpracovávat je.

K tomu, aby však bylo možné data spolehlivě odesílat ke zpracování, nebo naopak získávat agregovaná statistická data, musí systém disponovat rozhraním, které bude přesně definovat strukturu dotazů a dat.

4.2 Architektura systému

Jakmile jsou požadavky na systém dostatečně definované, je možné zkonstruovat návrh architektury tohoto systému tak, aby zohlednil všechny aspekty požadavků. Architektura systému, který je výsledkem této práce je zobrazena na obrázku 4.1.



Obrázek 4.1: Architektura systému na počítání unikátních aut na parkovišti a sbírání statistických dat.

Na diagramu je zobrazeno několik aktérů, kteří budou se systémem komunikovat. Prvním jsou kamery (obecně jich může být libovolné množství), které do systému odesílají obrazová data z parkovišť. Druhým aktérem se systémem je následně klient, který pracuje se statistickými údaji. Typické požadavky klienta zahrnují výběr nějakých konkrétních hodnot z databáze, či jejich agregování.

Rozhraní systému

Komunikace aktérů se systémem probíhá přes REST API. Vzhledem k jednoduchosti a rozšířenosti REST API všude na internetu je tento způsob komunikace vhodný pro jednoduché propojení aktérů a systému. Struktura požadavků a odpovědí je přesně definovaná, je proto možné dokonce poskytnou specifikaci třetím stranám, které by se na systém chtěly napojit taky.

Zpracování obrázků z kamer

Prvním a zároveň nejkomplexnějším případem užití zobrazeném na diagramu 4.1 je zpracování obrázků z kamer.

Po přijetí obrázku parkoviště je nejprve provést detekci aut na něm. Principy detekce objektů na obrázcích byly popsané v kapitole 2. Tuto činnost zajišťuje modul označený na diagramu jako *detektor*. Vstupem do tohoto modulu jsou tedy obrázky parkovišť a výstupem jsou pak rámce, ohraničující detekovaná auta.

Detekované rámce slouží jako vstup do modulu nazvaného *embedder*. Modul *embedder* vypočítá pro každé auto, které je ohraničeno detekovaným rámcem, *embedding* vektor, popisující jeho charakteristiky. Význam *embedding* vektorů a intuice za nimi byla popsaná v kapitole 3.3.

Modul označený jen jako *porovnání aut* přijímá na vstupu jak detekované rámce z modulu *detektor* a *embedding* vektory z modulu *embedder*, označující auta na novém snímku, tak detekované rámce a *embedding* vektory aut z předchozího snímku toho samého parkoviště. Tato historická data jsou vybraná z databáze. Úkolem modulu *porovnání aut* je porovnání aut detekovaných na novém snímku s auty na minulém snímku, která se nachází na stejných pozicích na parkovišti. V kapitole 7.3 je proces párování stejných parkovacích míst a porovnávání aut detailně popsán.

Posledním krokem zpracování nového obrázku z parkoviště je uložení veškerých informací o nových autech do databáze, odkud můžou být opět použity při zpracování dalšího snímku, nebo při dolování statistických dat.

Zpracování požadavku na statistická data

Klientský případ použití systému je co do zpracování jednodušší. Klient posílá v předem specifikovaném formátu požadavek na data na příslušný *endpoint* v REST API. Systém tento požadavek zpracuje v modulu nazvaném na diagramu jako *zpracování dat* a vybere z databáze veškerá relevantní data. Podle požadavku může data dále zpracovat, nebo nad nimi provést různé agregační funkce. Následně požadovaná data odesílá klientovi. Opět v předem specifikovaném formátu, jako odpověď na jeho požadavek.

4.3 Trénovací datasety

Moduly pro detekci aut na obrázku a pro vytváření *embedding* vektorů využívají hlubokých neuronových sítí. Ty je samozřejmě potřeba natrénovat na konkrétní problém a k tomu je zapotřebí veliké množství trénovacích dat. Naštěstí existují kvalitní množiny těchto trénovacích dat, které se dají s malými úpravami použít.

PKLot

Dataset *PKLot* [2] byl publikován v roce 2015. Obrázky obsažené v tomto datasetu byly pořízeny ze dvou parkovišť. Federal University of Parana (UFPR) and the Pontifical Catholic University of Parana (PUCPR). Dohromady dataset obsahuje 12 417 obrázků parkovišť, na nichž se nachází celkově 695 899 parkovacích míst. Snímky byly pořízeny standardní kamerou umístěnou buď na přilehlé výškové budově nebo na sloupu vedle parkoviště, v různých světelných a povětrnostních podmínkách.

Ke každému snímku je vytvořen XML soubor obsahující strukturovaná data o pozicích parkovacích míst a informaci o jejich obsazenosti autem. Mírnou nevýhodou může být fakt, že anotace neobsahují informace o všech autech přítomných na snímcích. Z tohoto důvodu není dataset příliv vhodný pro trénování *YOLOv4* detektoru.

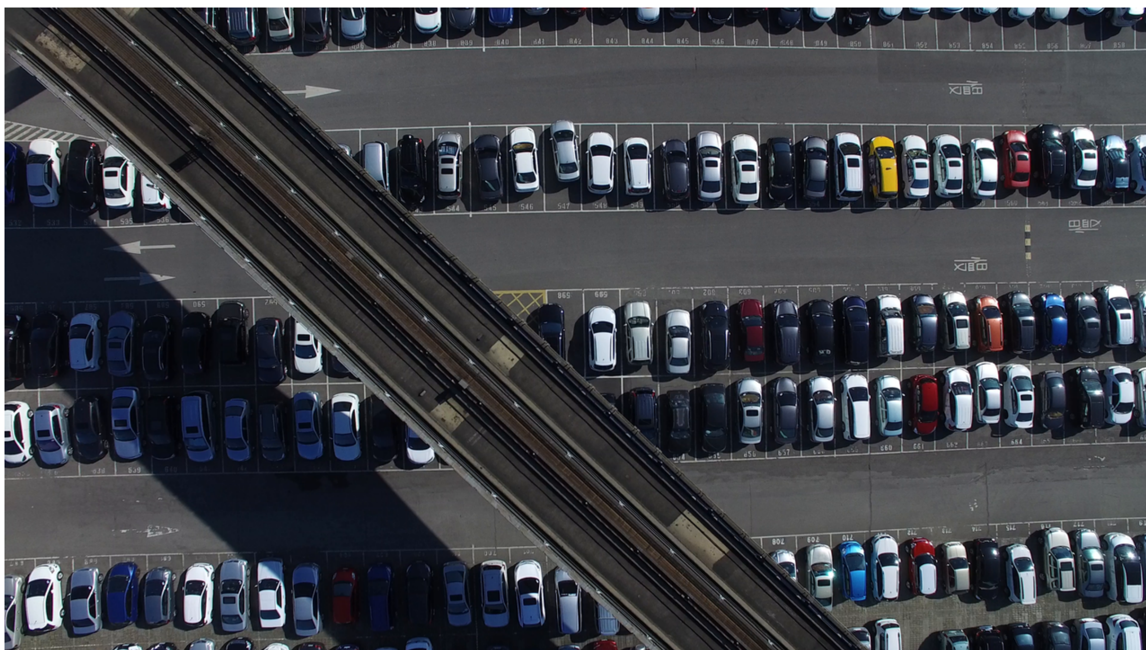
Výhodou je naopak to, že snímky byly na jednotlivých parkovištích snímány v intervalu 5 minut, což se blíží zamýšlenému použití pro systém vytvářený v této práci. Proto je možné zároveň z fotek parkoviště vyříznout pouze fotky jednotlivých aut a použít k trénování modelu pro výpočet *embedding* vektorů jednotlivých aut.

Obrázek 4.2: Ukázka snímku parkoviště PUCPR, pocházejícího z datasetu *PKLot*. (Zdroj: [2])

CARPK

V roce 2017 byla spolu s prací *Drone-based Object Counting by Spatially Regularized Regional Proposal Network* [9], která se zabývá počítáním aut na parkovištích, uveden také nový dataset nazvaný *CARPK*. Dataset *CARPK* obsahuje snímky parkovišť, obdobně jako dataset *PKLot*, přičemž některé snímky jsou dokonce z datasetu *PKLot* převzaty (konkrétně snímky z parkoviště PUCPR). Dohromady dataset obsahuje 1 573 snímků parkovišť na nichž se nachází 89 777 aut.

Na rozdíl od datasetu *PKLot* jsou snímky focené z ptačí perspektivy pomocí dronu. Velikou výhodou tohoto datasetu je fakt, že všechna auta, která se na fotkách nachází jsou řádně anotovaná, protože se jednalo o dataset, který byl v původní práci používán na počítání aut. Na druhou stranu ale fakt, že se nejedná o snímky ze statické kamery, jejíž poloha ani úhel záběru se mezi snímky nemění, představuje výrazný problém pro hledání stejných aut, mezi několika snímky, které by mohly být následně použity pro trénování modelu na *embedding*. Tento dataset bude tedy použit pouze na trénování *YOLO4* detektoru.



Obrázek 4.3: Ukázka snímku pocházejícího z datasetu *CARPK*. (Zdroj: [9])

Kapitola 5

Trénování a vyhodnocení modelů

S samotném jádru systému pro detekci a počítání aut na parkovišti jsou dvě neuronové sítě, přičemž každá má na starosti jinou část zpracování obrázku. První síť provádí detekce aut na každém snímku a je tedy zodpovědná za správné určení pozic aut na snímcích. Druhá sít poté pro jednotlivá detekovaná auta počítá *embedding* vektory. Před tím, než je možné tyto sítě v systému použít, je však potřeba je natrénovat na dostatečném množství trénovacích dat, které musí odpovídat úloze, kterou má výsledná síť řešit.

5.1 Detektor

V kapitole 2 byly popsány různé modely pro detekci objektů na obrázcích. Každý z těchto modelů detekoval trochu jiným způsobem a měl různé výhody a nevýhody. Pro systém pro detekci a počítání unikátních aut na parkovišti byl nakonec použit model *YOLOv4* [4].

Darknet *framework*

Implementace celého *YOLOv4* modelu a zároveň nástrojů pro jeho trénování a vyhodnocení by byla velice náročná. Spolu s modelem *YOLO* však vzniká také open source *framework* Darknet [14]. Ten nejenom že implementuje model *YOLO* pro detekci objektů na snímcích, ale zároveň obsahuje sadu nástrojů právě pro jednoduchou úpravu specifikace *YOLO* modelu, jeho trénování a vyhodnocení.

Existuje několik verzí Darknet *frameworku*, přičemž autorem nejnovější verze, která zároveň podporuje i model *YOLOv4*, je Alexey Bochkovskiy. Tato verze je dostupná v jeho GitHub repositáři [3], který neobsahuje pouze zdrojové kódy, ale také podrobný návod na kompilaci *frameworku* a užitečné rady, jak nastavit model a jeho trénování pro optimální výsledky na vlastní datové sadě.

Úprava konfigurace modelu

Model *YOLOv4* je poměrně složitý a jeho architektura je nejlépe popsaná v původní práci [4], ve které byl představen. Pro samotné natrénování a využívání *YOLOv4* však není třeba model příliš měnit, protože autoři Darknet *frameworku* ve svém repositáři poskytli konfigurační soubory, které strukturu modelu předdefinují, přičemž v těchto konfiguračních souborech není doporučeno příliš měnit samotnou strukturu sítě. Je v nich však možné nastavit určité parametry tak, aby byl model lépe uzpůsoben konkrétnímu problému a datové sadě, na které bude trénován.

Následující parametry byly změněny oproti základnímu konfiguračnímu souboru, obsahujícímu strukturu sítě *YOLOv4*.

- ***width, height*** - Všechny obrázky vstupující do sítě musí mít předem definovanou velikost. Parametry *width* a `height` definují tuto velikost a oba jsou nastaveny na hodnotu 416.
- **`classes`** - Konfigurační soubor pro síť *YOLOv4* obsahuje konfiguraci tří detekčních vrstev. Každá tato vrstva obsahuje parametr `classes`, který určuje počet tříd, jenž budou klasifikovány. Vzhledem k tomu, že systém má detekovat pouze auta a nic jiného, je parametr `classes` ve všech těchto vrstvách nastaven na 1.
- **`filters`** - Před každou ze tří detekčních vrstev se nachází také konvoluční vrstva, která obsahuje parametr `filters` určující počet konvolučních filtrů pro danou vrstvu. Počet těchto filtrů je doporučeno změnit a to podle vztahu $(\texttt{classes} + 5) \cdot 3$. Počet filtrů pro tyto vrstvy byl tedy nastaven na 18.
- **`max_batches`** - Nastavení maximálního počtu iterací, po který bude trénování sítě probíhat. Tento parametr je opět doporučeno změnit podle vztahu $\texttt{classes} \cdot 2\,000$, neměl by však být menší než 6 000. Tento parametr byl tedy nastaven na hodnotu 6 000.
- **`steps`** - Druhý parametr upravující učení je parametr `steps`. Na začátku trénování je učící konstanta nastavena na hodnotu 0.001. Parametr `steps` specifikuje, ve kterých iteracích má být učící konstanta zmenšena, aby se zajistilo správné konvergování sítě. Parametru `steps` byly tedy nastaveny hodnoty 4800 a 5400, odpovídající 80% a 90% celkového počtu iterací.

Příprava trénovacích dat

Datasetů, které obsahují fotky parkovišť je celá řada. Neexistuje však jednotný formát, ve kterém by byly kódované anotace všech těchto datasetů. Jak již bylo uvedeno v kapitole 4.3, byl pro trénování detektoru použit dataset CARPK [9], především proto, že všechna auta, která jsou na fotkách zachycena, jsou řádně anotovaná. Ke každému obrázku náleží jeden textový soubor, ve kterém se na každém řádku nachází anotace pro jedeno auto na daném obrázku. Formát, který dataset CARPK využívá pro anotace je následující:

```
<x_left> <y_top> <x_right> <y_bottom>
```

Každý rámec pro auto je definován souřadnicemi levého horního a pravého dolního bodu. Tyto souřadnice jsou navíc uvedené v pixelech a jsou tedy přímo navázané na konkrétní velikost obrázků. Darknet *framework* však vyžaduje anotace ve formátu:

```
<class> <x_center> <y_center> <width> <height>
```

Anotace tedy využívají k definici rámce jeho střed, šířku a výšku. Vzhledem k tomu, že síť *YOLOv4* automaticky všem vstupním obrázkům mění velikost na předem definovanou hodnotu, nemůžou být souřadnice zapsané pomocí pixelů. Každá souřadnice je tedy normalizovaná výškou, nebo šířkou celého obrázku do intervalu $\langle 0, 1 \rangle$. Před samotným trénováním bylo tedy nejprve potřeba upravit anotace datasetu CARPK pomocí skriptu `parse_annotations.py`, který je součástí zdrojových kódů.

V neposlední řadě očekává Darknet ještě soubory, které mu lépe popíšou, kde má trénovací a validační data hledat a co se v datech konkrétně nachází. Soubor `obj.data`, obsahuje počet detekovaných tříd, cestu k souboru `obj.names`, ve kterém se nachází jména jednotlivých tříd a následně pak cestu k souborům `train.txt`, `test.txt` a `valid.txt`, kde každý obsahuje seznam cest k trénovacím, testovacím a validačním obrázkům, které budou použity pro trénování a testování sítě.

Trénování detektoru

Posledními kroky před spuštěním samotného trénování detektoru je kompilace *frameworku* Darknet, ke které je možné použít nástroj `cmake` a stažení před-trénovaných vah konvolučních vrstev. Trénování je možné začít i bez těchto před-trénovaných vrstev, není to však doporučeno, neboť takové trénování je časově výrazně delší.

Jakmile je připraven konfigurační soubor s architekturou sítě a všechna trénovací a validační data, samotné trénování už je na systému, kde byl zkompilován Darknet, pouze otázkou spuštění příkazu:

```
./darknet detector train obj.data yolov4-obj.conf yolov4.conv
```

Parametr `obj.data` odkazuje na soubor popisující data, parametr `yolov4-obj.conf` pak na konfigurační soubor se strukturou sítě a nakonec parametr `yolov4.conv` odkazuje na stažené před-trénované váhy konvolučních vrstev.

Vyhodnocení detektoru

Pro vyhodnocení detekčních úloh se nejčastěji využívají metriky *precision*, neboli přesnost a *recall*, neboli výtěžnost. Hodnota *precision* určuje míru přesnosti detektoru, tedy podíl správně detekovaných a zařazených objektů, proti všem detekcím. Pokud tedy detektor generuje veliké množství detekcí, ale jen málo z nich je oprávněných, jeho přesnost bude nízká. Hodnota *recall* určuje s jakou mírou se detektoru daří nalézt všechny objekty na obrázku, které má detekovat. Tato hodnota je tedy vypočítaná jako poměr správně detekovaných a zařazených objektů proti všem objektům žádaného typu, které se na obrázku opravdu nachází a měly být detekované. Pokud tedy detektor nalezne pouze zlomek objektů, které nalézt měl, je hodnota *recall* malá i přesto, že všechny jím nalezené objekty správně zařadil a jeho přesnost je tedy maximální.

Pokud detektor není naprosto dokonalý, je potřeba najít kompromis mezi přesností a výtěžností. Při provádění detekce vypočítá detektor pro každý detekovaný objekt takzvanou míru jistoty. Jedná se o pravděpodobnost, s jakou se dle detektoru skutečně jedná o hledaný objekt. Práh jistoty, nebo také *confidence threshold* pevně určuje hranici, od které je rozhodnuto, že bude detekce prohlášena za platnou. Pakliže je práh jistoty nastavený na 0.5, pak všechny detekce, u kterých je míra jistoty nižší, než je tento práh, nebudou počítané jako detekce. Práh jistoty je také spjat s hodnotami *precision* a *recall*. Vysoký práh jistoty způsobí, že výsledné detekce budou s velikou pravděpodobností správné a *precision* tedy bude vysoká, ale těchto detekcí bude pravděpodobně málo a *recall* tedy bude nízký. S nízkým prahem jistoty pak bude naopak detekcí veliké množství a *recall* tak bude vysoký, ale většina detekcí nemusí být tak kvalitních a *precision* se tedy sníží.

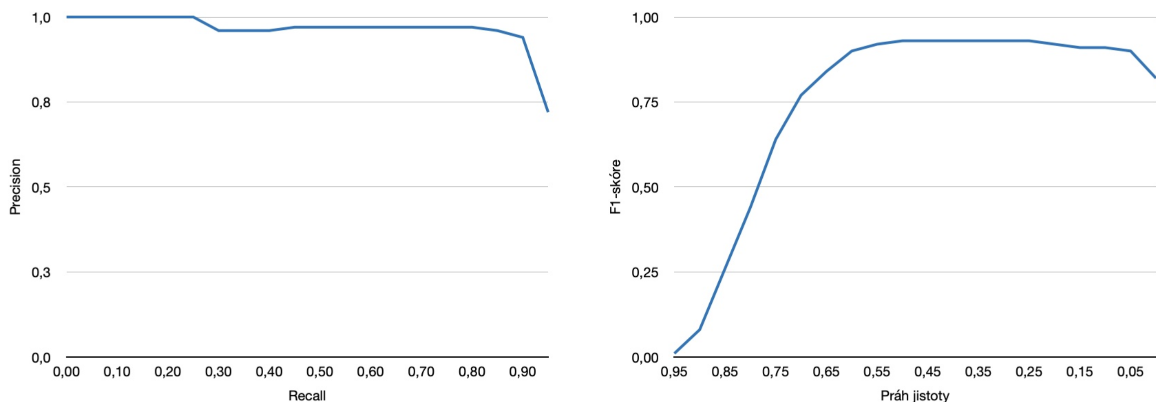
Proto se hodnoty *precision* a *recall* pro daný detektor nepočítají pouze jednou. Místo toho jsou spočítány několikrát, vždy pro jiné nastavení prahu jistoty. Jejich vztah je následně zobrazen pomocí *Precision-Recall* křivky. Pro detektor *YOLOv4*, který byl trénován na

detekci parkovacích míst je tato křivka zobrazena v levé části obrázku 5.1. Metriky *precision* a *recall* byly počítány nad 100 obrázky z původního datasetu, které byly náhodně vybrány ještě před trénováním a síť je tedy nikdy předtím nezpracovávala. Hodnota prahu jistoty byla postupně nastavována v intervalu $\langle 0, 0.99 \rangle$ s krokem 0.05. Jak je vidět na obrázku, detektor si zachovává velice vysokou přesnost i při poměrně vysoké míře výtěžnosti a celkově tak dosahuje *mean average precision*[1] (mAP) **91.9%**.

V pravé části obrázku 5.1 je následně vykreslen vztah prahu jistoty a takzvaného F1-skóre, což je kombinace hodnot *precision* a *recall* a určuje tak celkovou přesnost detektoru pro daný práh. Výpočet F1-skóre je dán vztahem 5.1.

$$F1_{score} = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{5.1}$$

Z výsledné křivky v pravé části obrázku 5.1 je zřejmé, že nejlepší úspěšnosti dosahuje detektor s prahem jistoty z intervalu $\langle 0.25, 0.55 \rangle$. Pro použití v systému na detekci aut na parkovišti, byl tedy zvolen práh **0.3**.



Obrázek 5.1: *Precision-Recall* křivka vlevo ukazuje, že výsledný model si zachovává vysokou přesnost i při poměrně vyšších hodnotách výtěžnosti. V pravé části je ze vztahu mezi prahem jistoty a F1-skórem možné vyčíst optimální nastavení prahu jistoty, které bylo nakonec zvoleno na hodnotu 0.3.

Realizace všech výpočtů nad celou sadou testovacích obrázků by mohla být poměrně náročná. Naštěstí *framework* Darknet opět poskytuje nástroj, pomocí nějž je možné nad testovacími daty provést jak výpočet metrik *precision* a *recall*, tak počítání F1-skóre a mAP. Slouží k tomu následující příkaz:

```
./darknet detector map obj.data yolov4-obj.cfg yolov4-obj.weights \
-thresh 0.3
```

Parametry `obj.data` a `yolov4-obj.cfg` odkazují, stejně jako při trénování, na soubor cestami k datům a konfigurační soubor sítě. Parametr `yolov4-obj.weights` odkazuje na již natrénované váhy detektoru a nakonec je pomocí parametru `-thresh` možné nastavit požahovaná práh jistoty.

[1]https://towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f623a52

5.2 Embedder

Kapitola 3 předestřela některé z problémů, které se můžou při porovnávání aut vyskytnout. Zároveň tam však bylo popsáno řešení těchto problémů v podobě sítě, která z obrázku auta vygeneruje *embedding* vektor, který by měl být ideálně invariantní vůči veškerým vnějším vlivům, které mohou ovlivňovat vzhled totožného auta. Ve stejné kapitole byl zároveň představen i koncept siamských sítí a ztrátová funkce TripletLoss, která byla nakonec v sytému pro detekci a počítání aut použita jako ztrátová funkce *embedderu*.

Generování trojic

Ztrátová funkce TripletLoss potřebuje pro optimalizaci sítě trojici obrázků najednou. První dva by měly být obrázky stejného auta a poslední obrázek pak jiného auta. Pro trénování sítě je tedy potřeba generovat veliké množství těchto trojic. Problém ovšem spočívá v tom, že datasety s obrázky aut většinou nemají více obrázků stejného auta nebo tyto obrázky nejsou anotované jako stejné auto. Prvním krokem je tedy výběr alespoň dvou odlišných aut, což se jeví jako jednodušší úloha. Pakliže bychom ale měli pouze veliké množství obrázků jednotlivých aut, teoreticky by existovalo riziko, že vybereme dvě stejná auta.

Proto byl pro trénování *embedderu* vybrán dataset PKLot [2]. Tento dataset obsahuje veliké množství samostatných snímků parkovišť, na nichž jsou auta. To, že se jedná o snímky celých parkovišť a ne o snímky jednotlivých aut je v případě generování trojic pro TripletLoss výhodou. Je totiž možné bez obav vybrat na stejném snímku parkoviště dvě auta, o kterých můžeme s naprostou jistotou říct, že se nejedná o totožné auto, protože se na stejném snímku nachází na dvou různých místech na parkovišti. První krok výběru dvou odlišných aut je tedy hotový. K jednomu z takto vybraných aut bychom však navíc potřebovali najít další snímek, na kterém se nachází, což není možné jednoduše udělat bez příslušných anotací, určující, že dvě auta jsou totožná.

První možností jak tedy najít stejné auto by bylo vybírat obrázky totožných aut ručně a ručně je také anotovat. Tento postup by byl časově velice náročný a při velikém množství dat prakticky nemožný. Alternativní přístup je další snímek stejného auta vůbec nehledat, ale místo toho z původního snímku vygenerovat množství dalších, které budou mírně pozměněny. Změny by však neměly být ledajaké, ale měly by odpovídat tomu, co chceme, aby sít při generování *embedding* vektorů nebrala v úvahu. To, které jevy můžou působit problémy při porovnávání aut, bylo ukázáno v kapitole 3.1. Knihovna imgaug [10] poskytuje sadu nástrojů pro pohodlnou úpravu fotek, které je možné použít i pro augmentaci obrázků aut. Při generování nových augmentovaných snímků se prování několik operací:

- **Afinní transformace** - První úpravou, která se v rámci augmentace obrázku auta děje, je úprava pozice auta na obrázku. Toto je provedeno jednoduše pomocí zvětšení, či zmenšení fotografie a následného posunutí fotografie po obou osách. Tímto je simulováno nedokonalé zarovnání detekovaného rámce na parkující auto. Všechny tyto afinní transformace mají rozsah $\langle -20\%, 20\% \rangle$ z celkové velikosti snímku auta.

- **Úprava jasu** - Pro simulaci různých světelných podmínek je potřeba změna několika hodnot. Jednou z nich je samozřejmě celkový jas. Změna jasu probíhá v rozsahu $\langle 50\%, 150\% \rangle$ původního jasu snímku.

- **Úprava kontrastu** - Druhou složkou, podílející se na simulaci světelných podmínek, je změna kontrastu na fotografii. Kontrast je upravován v rozsahu $\langle 50\%, 200\% \rangle$.

- **Úprava vyvážení bílé** - Simulace různých světelných podmínek je završena změnou teploty fotografie. Výsledná teplota fotografie po upravení spadá do rozmezí $\langle 3000K, 9000K \rangle$

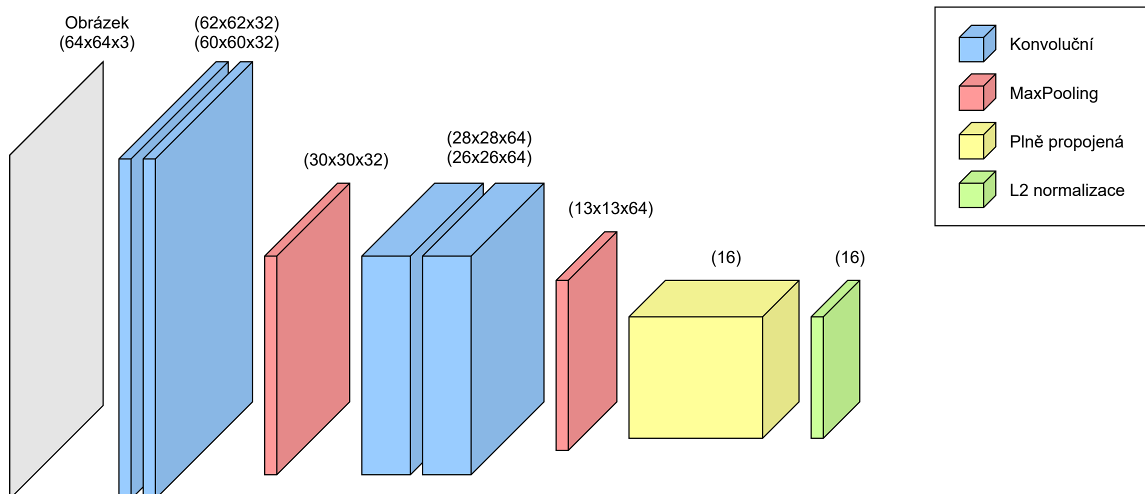
Obrázek 5.2: Augmentace snímku jednoho auta. Posouváním původního obrázku a jeho zvětšením, či zmenšením se simuluje nepřesnost detektoru. Úpravou vyvážení bílé a jasu jsou simulované různé světelné podmínky.

Pro každé auto na jednom snímku parkoviště je vygenerováno několik takto augmentovaných obrázků, přičemž jak originálnímu obrázku, tak všem z něj vygenerovaných, je přiřazena anotace, označující, že se jedná o stejné auto. Získání trojice je poté otázka vybrání dvou obrázků se stejnou anotací a jednoho obrázku z jinou anotací. Z jednoho snímku parkoviště je tedy možné sestavit hned několik trojic a každý snímek parkoviště může tedy sloužit jako základ pro celou mini-dávku pro trénování sítě.

Obrázek 5.3: Trojice obrázků vygenerovaná pro trénování *embedder* sítě pomocí ztrátové funkce TripletLoss. V trojici se nachází dva upravené snímky stejného auta a jeden snímek jiného auta.

Architektura modelu

V práci *FaceNet: A Unified Embedding for Face Recognition and Clustering* [18] řešili autoři problém porovnávání fotek lidských obličejů. I když se předmět zájmu v případě porovnávání aut liší, pořád se v podstatě jedná o tentýž problém. Proto byl i model pro výpočet *embedding* vektorů pro porovnání aut inspirován architekturou původní sítě FaceNet. Vzhledem k tomu, že porovnávání aut na stejném parkovacím místě je o poznání jednodušší, než porovnávání lidských obličejů, mohl být i samotný model značně zjednodušen. Na obrázku 5.4 je architektura modelu graficky znázorněna.



Obrázek 5.4: Architektura sítě, která pro obrázky aut počítá *embedding* vektory. U každé vrstvy jsou uvedeny rozměry výstupního tenzoru dané vrstvy.

Model na vstupu očekává obrázky o fixní velikosti 64×64 pixelů. Pro snímky celého parkoviště nelze očekávat, že by měly výřezy jednotlivých aut o mnoho větší rozlišení a zároveň tím pádem nemusí být síť tak veliká. První dvě vrstvy sítě jsou konvoluční vrstvy. Každá z těchto vrstev má 32 filtrů o velikosti 3×3 a využívá aktivační funkci `relu`. Výsledek je zpracován max-pooling vrstvou, která využívá krok 2×2 a čtyřnásobně tak sníží počet pixelů, přičemž zachovává významné hodnoty z konvolučních vrstev. Následuje druhý blok dvou konvolučních vrstev. Tentokrát však každý využívá 64 filtrů o velikosti 3×3 s aktivační funkcí `relu`. Tyto vrstvy umožňují detekci složitějších vzorů z původního obrázku. Samotné zpracování obrázku je zakončeno opět max-pooling vrstvou o kroku 2×2, která opět čtyřnásobně sníží počet pixelů.

Příznaky extrahované z obrázku pomocí konvolučních vrstev jsou poté vloženy do hustě propojené vrstvy. Síť FaceNet využívala pro popis lidského obličeje *embedding* vektor o rozměru 128. Jak již bylo zmíněno, porovnávání dvou aut na stejném parkovacím místě, navíc z konstantního úhlu záběru, je o poznání jednodušší úloha. Obrázky aut jsou navíc v menším rozlišení a obsahují také méně detailů. Také je třeba pamatovat na to, že vypočítané *embedding* vektory by se měly uchovávat někde v databázi pro pozdější vyhodnocení, kde by příliš dlouhé vektory postupem času zabíraly zbytečně moc místa. Po přihlédnutí ke všem těmto faktům byla pro *embedding* vektor popisující parkující auta zvolena délka **16**. Plně propojená vrstva má proto 16 neuronů a jelikož se v podstatě jedná o regresní úlohu, nevyužívá žádnou aktivační funkci. Ztrátová funkce TripletLoss využívá pro výpočet chyby L2 vzdálenosti mezi vektory. Výpočet *embedding* vektoru je tedy ukončen L2 normalizací[2] hodnot z plně propojené vrstvy.

Model je implementován pomocí knihovny Keras[3] a využívá před-implementovanou ztrátovou funkci `TripletSemiHardLoss`[4], která se nachází v knihovně TensorFlow Addons. Tato implementace TripletLoss ztrátové funkce odstiňuje od manuálního vytváření siamské sítě pro trénovaní modelu. Zároveň automaticky provádí *online* výběr vhodných trojic pro optimální trénování.

[2] https://machinelearningmastery.com/vector-norms-machine-learning/
[3] https://keras.io
[4] https://www.tensorflow.org/addons/tutorials/losses_triplet

Trénování *embedderu*

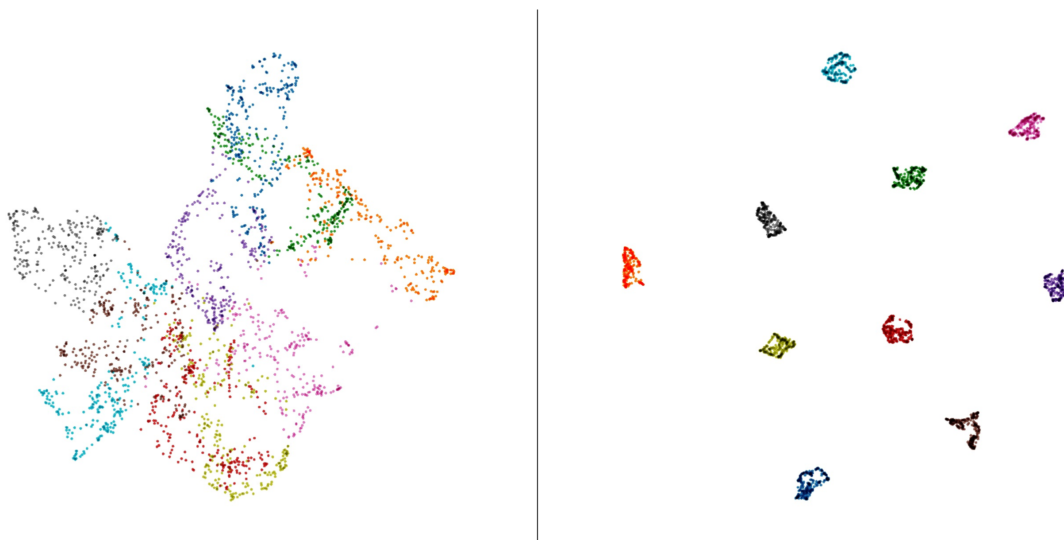
Trénování je za použití knihovny Keras opět velice jednoduché. Nad vytvořeným modelem je zavolaná metoda `fit`, které je předán generátor, generující z každého obrázku parkoviště várku augmentovaných obrázků jednotlivých aut a příslušné anotace k nim. Jak již bylo zmíněno, ztrátová funkce `TripletSemiHardLoss` se už během trénování sama stará o skládání vhodných trojic, pomocí kterých síť optimalizuje.

Síť byla trénovaná s velikostí mini-dávky 256 po 3 500 iterací na epochu a po dobu 5 epoch. Na jednu epochu to tedy činí necelých 300 000 trojic obrázků aut.

Vyhodnocení *embedderu*

Během trénování sítě je sice možné sledovat hodnotu chyby, počítané ztrátovou funkcí. Z té však není moc poznat, zda se síť opravdu naučila řešit daný problém. K prvnímu experimentu bylo tedy náhodně zvoleno deset různých aut z validační části datasetu a pro každé z těchto aut bylo vygenerováno 250 alternativních obrázků pomocí technik popsaných výše.

Pro ustanovení referenčního bodu je vhodné pokusit se porovnat přímo celé snímky aut mezi sebou. Ačkoliv sice jednotlivé pixely snímků nejsou přímo souřadnicemi v *embedding* prostoru, pořád platí vlastnost, že pokud bychom tak s nimi pracovali, měly by dva totožné snímky nulovou euklidovskou vzdálenost mezi pixely a čím rozdílnější by snímek byl, tím větší by vzdálenost byla. Tyto obrázky byly tedy zaneseny do nástroje Embedding Projector[5], který je pomocí algoritmu UMAP[6] schopný zredukovat dimenzionalitu vložených dat a promítnout tak tyto vektory do 2-D prostoru. Výsledek je zobrazen na levé půlce obrázku 5.5. Promítnutá data působí velice chaoticky a není možné podle vzdálenosti bodů jednoduše odlišit, na kterých obrázcích se nachází stejná auta.



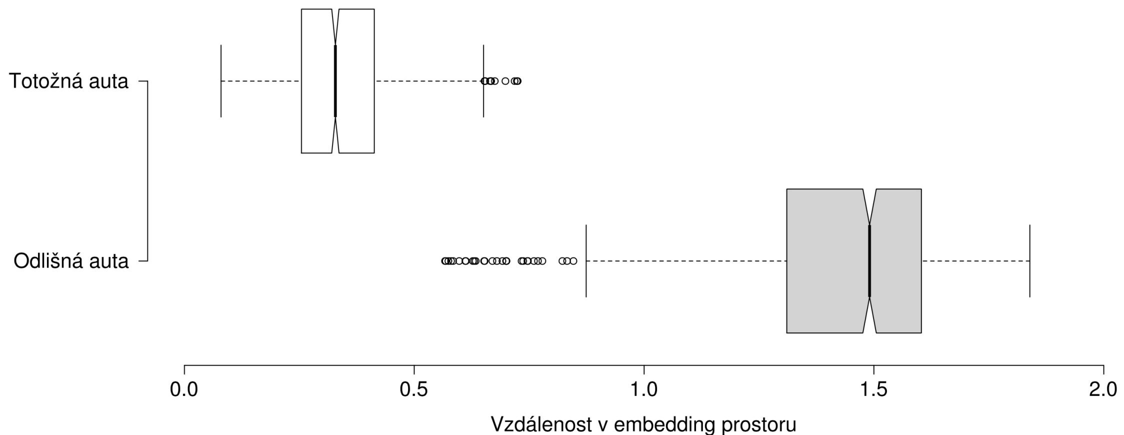
Obrázek 5.5: Promítnutí *embedding* prostoru na 2-D plochu. Vlevo je výsledek přímého promítnutí obrázků aut. Vpravo je promítnutí vypočítaných *embedding* vektorů. Okamžitě je vidět, že *embedding* vektory stejných aut jsou seskupeny.

[5] https://projector.tensorflow.org
[6] https://umap-learn.readthedocs.io/en/latest/how_umap_works.html

Ze stejných obrázků byly následně pomocí naučené sítě vygenerovány *embedding* vektory, popisující dané auto. Pomocí nástroje Embedding Projector byla opět provedena redukce dimenzionatily. Výsledek je také na obrázku 5.5, tentokrát na pravé polovině. Okamžitě je vidět, že se *embedding* vektory stejných aut seskupily. Všech deset původních aut je nyní jasně rozlišitelných. Toto samotné dokazuje, že natrénovaná síť řeší problém poznávání stejných aut na lehce změněných snímcích přinejhorším lépe, než přímé porovnání snímků aut.

Pro optimální fungování porovnávání dvou *embedding* vektorů je však ještě vhodné zjistit průměrnou vzdálenost dvou *embedding* vektorů patřících stejným autům, a následně stejnou vzdálenost pro vektory patřící dvěma odlišným autům. Znalost těchto hodnot pak poslouží k vhodnému zvolení maximální vzdálenosti v *embedding* prostoru, při které je ještě možné dvě auta prohlásit za totožná.

Z validační části datasetu bylo pro tento experiment náhodně vybráno 100 aut. Ke každému snímku auta bylo vygenerováno 100 alternativních snímků a pro každý snímek byl vypočítán příslušný *embedding* vektor. Pro každý vektor byla následně vypočítaná euklidovská vzdálenost se všemi ostatními vektory. Výsledky byly zaneseny do grafu 5.6. Graf potvrzuje, že mírně se lišící snímky totožných aut jsou si v *embedding* prostoru blíže než snímky odlišných aut. Průměrná vzdálenost totožných aut je **0.3482** a průměrná vzdálenost odlišných aut je **1.4324**. Na základě těchto dat byla jako práh pro určení totožnosti auta zvolena hodnota **0.7**. Při nastavení tohoto prahu je pak úspěšnost sítě při porovnávání aut na stejném parkovacím místě přibližně **98.6%**



Obrázek 5.6: Hodnoty vzdáleností v *embedding* prostoru při porovnávání vektorů totožných aut a odlišných aut. Průměrná vzdálenost pro totožná auta je 0.3482 a průměrná vzdálenost pro odlišná auta je 1.4324.

Kapitola 6

Aplikace a REST API

Ačkoliv jsou model pro detekci aut na parkovišti a model pro výpočet *embedding* vektorů pro jednotlivá auta nedílnou součástí fungujícího systému pro počítání aut na parkovišti, zdaleka nejsou jedinou. V kapitole 4.2 je architektura systému jako celku znázorněná pomocí schématu. V této kapitole bude struktura systému popsaná detailněji, včetně použitých technologií, databázového modelu, rozdělení aplikace do funkčních komponent a popis komunikace těchto komponent. A v neposlední řadě budou popsané jak REST API, pomocí kterého je možné s aplikací komunikovat, tak grafické uživatelské rozhraní aplikace.

6.1 Technologie

Kromě samotných natrénovaných modelů, které se používají pro detekci aut a výpočet *embedding* vektorů pro jednotlivá auta a technologií s nimi spojenými, využívá systém pro počítání unikátních aut na parkovišti celou řadu dalších technologií.

Pro lepší přehled toho, jak systém funguje je vhodné nejprve přiblížit tyto použité technologie a způsob, jakým jsou v systému využity.

Flask

Ať už se jedná o REST API, či webovou aplikaci, existuje spousta úkonů, kterou je potřeba vykonávat nad rámec samotné aplikační logiky. Takovými úkony je třeba směřování požadavků na aplikaci na správné místa, zpracovávání těchto HTTP požadavků, správu statických zdrojů, jako jsou HTML stránky, obrázky a podobně. Ve většině případů jsou však tohle úkony, které jsou shodné ať už se jedná aplikaci, která zobrazuje například jízdný řády nebo o aplikaci na počítání unikátních aut na parkovišti. Existuje proto celá řada knihoven, která poskytují nástroje, jež tvůrce aplikace od těchto repetetivních úkonů do značné míry odstíní a tvorba webové aplikace je pak o poznání jednodušší.

Flask[1] je Python *framework*, který poskytuje nástroje, které umožňují právě tohle. Jeho použití je jednoduché a existuje pro něj veliké množství rozšíření, které implementují například logiku potřebnou pro přihlašování a udržování *session*, zjednodušenou práci s databází a podobné. To z něj učinilo v současnosti jeden z nejpoužívanějších *frameworků* pro vytváření webových aplikací v jazyku Python.

Systém pro detekci unikátních aut na parkovišti využívá *framework* Flask jak pro implementaci REST API, pomocí kterého je se systémem možné komunikovat především na

[1] https://flask.palletsprojects.com

strojové úrovni, tak k implementaci webové aplikace, která poskytuje grafické uživatelské rozhraní, pomocí kterého je možné se systémem pracovat přímo v libovolném moderním webovém prohlížeči.

PostgreSQL

Veškeré informace ze zpracovaných obrázků parkovišť je samozřejmě potřeba ukládat do perzistentní paměti, ať už pro budoucí vyhodnocování, jestli se na daném parkovacím místě vyměnila auta, tak pro agregaci nějakých užitečných statistických dat. K tomuto účelu využívá aplikace databázový systém PostgreSQL[2].

PostgreSQL open source databázový systém a řadí se mezi nejpoužívanější databázové systémy na světě. Je využívám všude od malých projektů, až po profesionální sféru, což svědčí o jeho spolehlivosti. I tento fakt hrál svou roli při jeho výběru pro použití v systému pro detekci a počítání unikátních aut na parkovišti. Zároveň však toto kritérium nebylo jediné.

Druhým stejně důležitým kritériem pro výběr PostgreSQL databázového systému byly vestavěné datové typy pro ukládání geometrických objektů, jako jsou polygony, obdélníky a podobně. Těmito datovými typy je možné reprezentovat například detekované rámce parkovacích míst a navíc umožňují efektivní výpočty jejich ploch, průniků a podobně. Databázový model, včetně toho, které atributy jsou uloženy jako geometrický datový typ je popsaný v kapitole 6.3.

SQLAlchemy

Programovací jazyk Python má pro připojování k databázím specifikované takzvané DB API[3]. Knihovny, které slouží pro připojení ke konkrétním databázovým systémům musí toto API dodržovat. Ve skutečnosti se s databází čistě pomocí DB API nepracuje moc pohodlně. Veškeré dotazy se do databáze zadávají pomocí řetězců s SQL příkazy a odpovědi jsou ve formě základních Python datových typů. Data v databázi ale většinou představují nějaké objekty v reálném světě, se kterými aplikace dále pracuje. Je proto přirozené, že nad Python DB API vznikly různé knihovny, které s daty z databáze pracují na vyšší úrovni abstrakce a zaobalují je do specifických objektů, které je ve výsledné aplikaci možné jednodušeji používat. Této abstrakci se říká objektově relační mapování, zkráceně jen ORM.

Jednou z nejpoužívanějších Python knihoven, která umožňuje pracovat s databází ve formě ORM se nazývá SQLAlchemy[4]. Každá databázová tabulka je pomocí SQLAlchemy definovaná jako Python třída, přičemž atributy této třídy jsou mapované jako sloupce v databázové tabulce. Tyto třídy v sobě obsahují nejen jednoduché atributy tabulek, ale také vztahy mezi různými tabulkami a další pomocné metody. Tabulky tak již na aplikační úrovni není potřeba manuálně propojovat pomocí cizích klíčů. Místo toho je možné využít mapovací atributy přímo v Python třídách, představujících dané tabulky.

Jak již bylo zmíněno dříve, databázový systém PostgreSQL podporuje také některé méně tradiční datové typy, jako jsou například geometrické typy `polygon`, čí `box`. Vzhledem k tomu, že je knihovna SQLAlchemy navržená tak, aby fungovala s velikým množstvím databázových systémů, někdy datové typy, či SQL funkce specifické pro jeden konkrétní systém chybí. Ani tohle však není nepřekonatelný problém, protože SQLAlchemy je navrženo tak, aby bylo uživatelsky rozšiřitelné.

[2]https://www.postgresql.org
[3]https://www.python.org/dev/peps/pep-0249/
[4]https://www.sqlalchemy.org

V rámci aplikace pro detekci a počítání aut na parkovišti byly tedy specifické datově typy `polygon`, `box` a `point` databázového systému PostgreSQL doimplementovány, včetně operací nad nimi prováděnými v rámci databázového systému tak, aby byly kompatibilní s knihovnou SQLAlchemy a bylo je tedy možné jednoduše používat.

Celery

Existují typy úloh, které při běhu aplikace nevyžadují okamžité zpracování. Obecně se většinou jedná o úlohy, které jsou velice výpočetně náročné a není přímo nutné provádět je hned, nebo to mohou být úlohy, u kterých je pozdější zpracování dokonce žádoucí. Příkladem takové úlohy může být naplánování odeslání emailu na určitý čas, naplánování čištění starých záznamů z databáze, nebo odložené zpracování požadavků. V případě aplikace na detekci a počítání unikátních aut na parkovišti takové úlohy existují také a pro jejich plánování a následné vykonávání slouží systém Celery[5].

Celery je nástroj pro jazyk Python, který umožňuje nejen vytvářet fronty, do kterých je možné vkládat a plánovat úlohy, jenž mají být postupně zpracovávány, ale zároveň se stará o synchronizaci těchto front a také o zpracovávání jednotlivých úloh v nich. Velikými výhodami tohoto nástroje je jeho jednoduchost používání, kdy naplánování úlohy spočívá pouze ve vložení funkce, jejich parametrů a případného plánovaného času spuštění do fronty a Celery se postará o zbytek.

6.2 Kontejnerová struktura

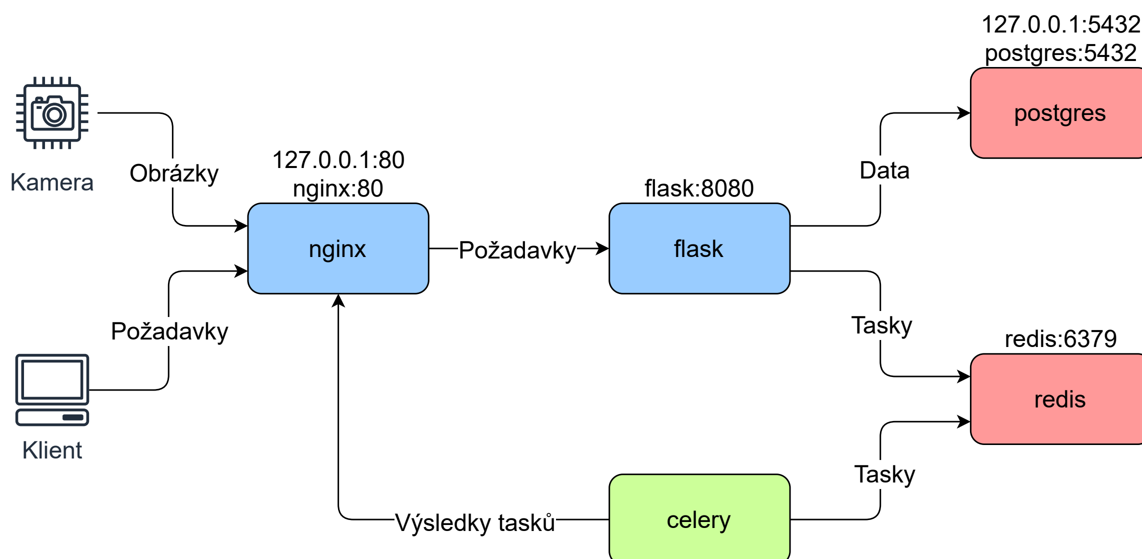
Ze schématu 4.1 je zřejmé, že systém pro počítání aut na parkovišti se skládá z několika součástí, které jsou sice všechny propojeny, ale každá může vyžadovat jiné technologie, běhové prostředí, či jiné náležitosti. Pokaždé, kdy by bylo potřeba celý systém spustit v novém prostředí by bylo nejprve nutné toto prostředí přesně nakonfigurovat podle potřeb systému, což může být náročný a časově zdlouhavý proces, přičemž stačí zapomenout na zdánlivou maličkost a systém pak nemusí pracovat správně.

Přesně v takovýchto situacích je vhodné použít virtualizované prostředí, které je potřeba nakonfigurovat pouze jednou. Systém následně běží ve virtuálním prostředí, které je pro něj upraveno na míru a toto virtuální prostředí (nebo alespoň jeho konfigurace) je přenositelná.

V současnosti velice rozšířeným systémem, který se stará právě o virtualizaci běhových prostředí, za účelem jednoduchého nasazení a přenositelnosti aplikací je Docker[6]. V Docker terminologii se hotové virtualizované prostředí nazývá kontejner. Jedna aplikace může využívat libovolné množství kontejnerů. To umožňuje rozdělení jedné aplikace do několika částí podle toho, jaké prostředí tyto části potřebují ke svému běhu. Zjednodušuje se tak celá konfigurace a zároveň i správa závislostí, protože změna konfigurace jednoho kontejneru nemá přímý dopad na konfigurace jiných kontejnerů.

Docker využívá i systém na počítání unikátních aut na parkovišti navržený a implementovaný v rámci této diplomové práce. Na obrázku 6.1 je vidět obdobné schéma aplikace, jako je vyobrazeno na obrázku 4.1, avšak zde jsou jednotlivé komponenty rozděleny do několika samostatných kontejnerů, které mezi sebou komunikují.

[5] https://docs.celeryproject.org
[6] https://www.docker.com



Obrázek 6.1: Architektura Docker kontejnerů systému na počítání unikátních aut na parkovišti. Jednotlivé kontejnery jsou zvýrazněny modře.

NGINX

Prvním kontejnerem, který je vyobrazen na schématu 6.1, je kontejner nazvaný `nginx`. Tento kontejner slouží jako hlavní komunikační brána mezi vnějším světem a zbytkem aplikace, protože v něm běží webový server NGINX[7].

Kontejner `nginx` je na hostitelském stoji dostupný na adrese `127.0.0.1:80` a veškeré požadavky vystavené REST API jsou směřovány na Flask aplikaci, která se nachází v druhém samostatném kontejneru.

Flask

Druhým a největším kontejnerem je kontejner nazvaný `flask`. V tomto kontejneru běží Flask aplikace, která zpracovává veškeré požadavky, které přichází z NGINX serveru, ať už se jedná o požadavky na REST API, nebo o požadavky na zobrazení uživatelského rozhraní webové aplikace. Jak už bylo zaznačeno v schématu 4.1, mezi hlavní funkce Flask aplikace patří zpracovávání požadavků obsahujících snímky z parkoviště. Detailně se tomuto tématu věnuje kapitola 7. Další důležitou funkcí, kterou Flask aplikace zastává je správná agregace dat z databáze, do smysluplných statistických údajů. Vzhledem k tomu, že Flask aplikace je jediná komponenta systému, která má přímý přístup do databáze nacházející se v kontejneru `postgres`, musí veškeré požadavky na získávání dat z databáze, či naopak ukládání nových dat procházet právě přes ni. Tato funkcionalita je podrobněji popsaná v kapitole 8.

Přímý přístup k `flask` kontejneru není z hostitelského stroje možný. V rámci Docker sítě je však tento kontejner dostupný na adrese `flask:8080`. Tím je zajištěno, že veškeré požadavky přicházející z vnější sítě na REST API a webovou aplikaci budou skutečně procházet přes NGINX server. Zároveň je však NGINX server schopen tyto požadavky úspěšně předat Flask aplikaci na vnitřní Docker síti.

[7] https://www.nginx.com

Postgres

Pro ukládání perzistentních dat využívá aplikace databázový systém PostgreSQL. Tato databáze běží v samostatném kontejneru nazvaném `postgres`.

Ačkoliv s databází komunikuje v rámci aplikace pouze kontejner `flask`, je databáze umístěná v samostatném kontejneru z důvodu jednodušší konfigurace prostředí. V rámci Docker sítě je kontejner dostupný na adrese `postgres:5432`. Samostatný kontejner pro databázi ale navíc umožňuje zpřístupnění databáze i v hostitelském systému na adrese `127.0.0.1:5432`, aniž by musel být zároveň vystavený i kontejner `flask`.

Veškeré změny, které se odehrají v kontejneru během toho, co je spuštěný nejsou standardně trvalé a po restartu kontejneru se tedy smažou. Tohle samozřejmě v případě databáze není žádoucí. Proto jsou data v databázi ukládána na Docker *volume*[8] s názvem `postgres_data`. Toto *volume* je uloženo na hostitelském systému. Tím pádem je zajištěna persistence databázových dat i po restartu kontejneru.

Konkrétní databázový model, který aplikace pro počítání unikátních aut na parkovišti používá je detailněji popsaný v sekci 6.3.

Celery

Většina aplikační logiky a zpracování dotazů na REST API se nachází v kontejneru `flask`. Některé úlohy jsou však poměrně výpočetně náročné a mohly by API výrazně zpomalovat. Tyto náročné úlohy se proto neprovádí přímo při zpracovávání požadavku na API, ale jsou naplánované jako úlohy, k pozdějšímu zpracování. Zpracování dotazu na API tak může proběhnout rychle. Výhody tohoto přístupu jsou detailněji popsané v kapitole 7.

Služba Celery, využívaná právě na takové plánování a postupné zpracovávání naplánovaných úloh, běží v samostatném kontejneru s názvem `celery`.

Oddělení výpočetně náročných úloh od kontejneru `flask` navíc opět zjednodušuje konfiguraci jednotlivých prostředí, protože spousta knihoven je potřeba pouze pro tyto náročné výpočty. Typickým příkladem můžou být knihovny OpenCV, TensorFlow, Keras a podobné. Vyčleněním náročných výpočtů do samostatného `celery` kontejneru je možné tyto prerekvizitní knihovny nainstalovat pouze tam a zpřehlednit tak prostředí pro Flask aplikaci ve `flask` kontejneru.

Redis

Flask aplikace v kontejneru `flask` tedy zadává úlohy pro zpracování v kontejneru `celery`. K tomuto zadávání úloh a zpětnému informování, že úloha byla dokončena, je však potřeba prostředník, který zajistí předávání zpráv mezi těmito dvěma kontejnery. Na to slouží databáze Redis[9], která běží v kontejneru s názvem `redis` a v Docker síti je dostupná na adrese `redis:6379`.

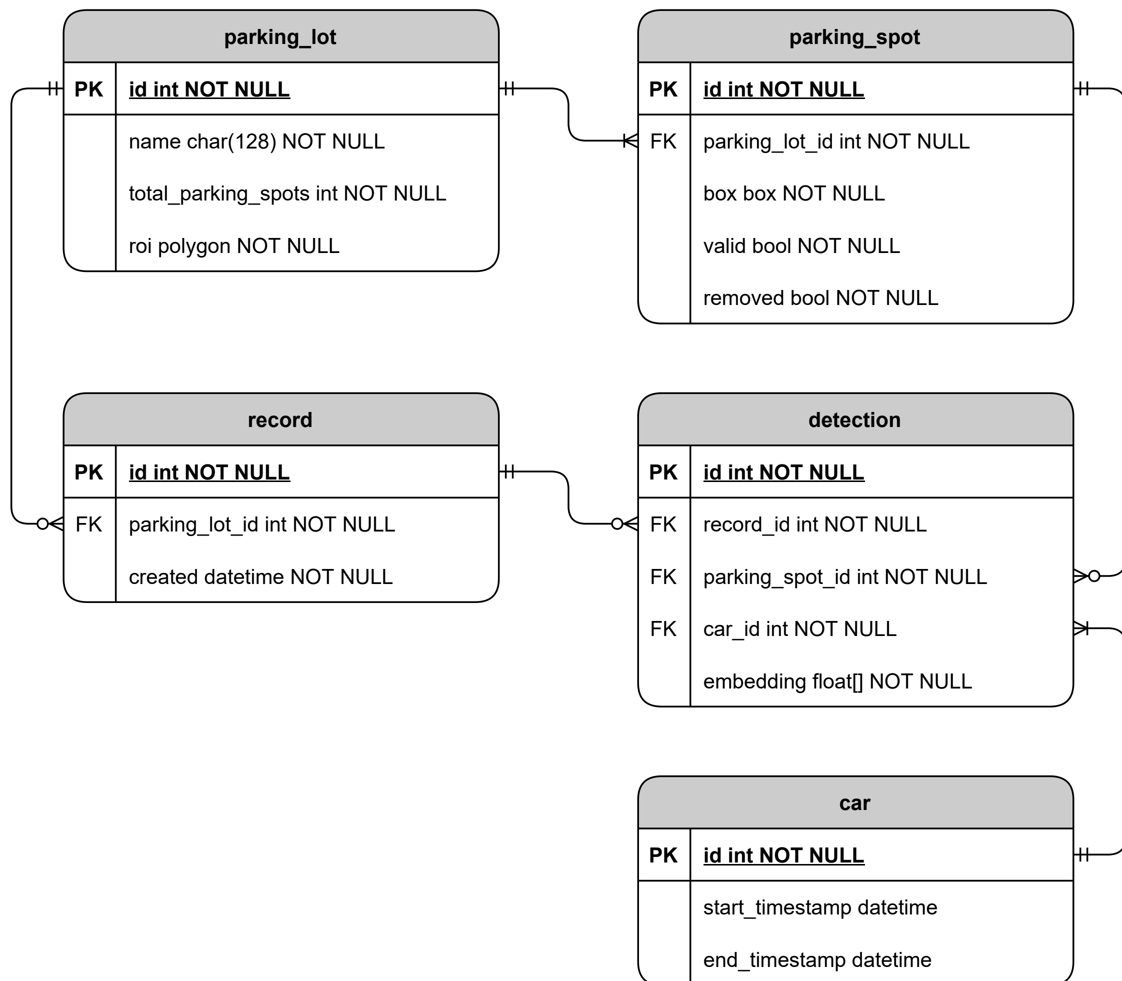
Flask aplikace se pomocí Celery klienta na její straně připojí k Redis databázi a vkládá do ní definice úloh, které mají být zpracovány. Celery *worker* v kontejneru `celery` je pak k této databázi připojen také, a úkoly si z ní vytahuje a zpracovává, načež u nich v Redis databázi aktualizuje stav a případný výsledek.

[8] https://docs.docker.com/storage/volumes/
[9] https://redis.io

6.3 Databázový model

Během zpracovávání snímků z parkoviště vznikají strukturovaná data. Databázový model musí tedy strukturou odpovídat datům, která během zpracování snímku vznikají. Zároveň ale databáze neslouží pouze jako prosté skladiště dat. Je potřeba dbát na to, aby byla data uložená v databázi způsobem, který umožní efektivní agregování statistických údajů, jako je například celkový počet unikátních aut na parkovišti, průměrný čas, který jedno auto na parkovišti stráví, nebo jaká je průměrná naplněnost parkoviště.

Výsledný databázový model, který systém pro počítání unikátních aut na parkovišti využívá je zachycen na obrázku 6.2.



Obrázek 6.2: Schéma databázového modelu, který využívá systém na počítání unikátních aut na parkovišti.

Tabulka `parking_lot`

Základní tabulkou databázového modelu je tabulka `parking_lot`. Systém pro počítání unikátních aut na parkovišti podporuje zpracovávání obrázků z libovolného počtu parkovišť. Je proto potřeba, aby byly základní informace o těchto parkovištích uložené v databázi.

Základní informace o parkovištích se následně buď zobrazují v aplikaci, nebo jsou použity pro různé statistické výpočty.

- `id` - Primární klíč, který jednoznačně určuje parkoviště. Tento identifikátor se následně používá i pro jednoznačné určení, ke kterému parkovišti patří data extrahovaná z jednotlivých obrázků daného parkoviště.
- `name` - Jméno parkoviště. Atribut se používá hlavně k zobrazení v uživatelském rozhraní aplikace a pro uživatelsky přívětivější identifikaci parkoviště než pomocí atributu `id`.
- `total_parking_spots` - Celočíselná hodnota určující celkový počet parkovacích míst na parkovišti. Toto číslo se využívá k výpočtu procentuální obsazenosti parkoviště.
- `roi` - Polygon tvořený seznamem bodů označující oblast zájmu na fotografiích daného parkoviště. Umístění kamery na parkovišti nemusí být ideální a detekce aut na parkovacích místech, která jsou například částečně skrytá stromem nemusí být ideální. Proto je pro každé parkoviště možné specifikovat oblast zájmu. Statistické údaje, které systém agreguje jsou následně počítány pouze z parkovacích míst uvnitř oblasti zájmu.

  Datový typ `polygon` databázového systému PostgreSQL umožňuje jednoduchou kontrolu toho, jestli se parkovací místo nachází v oblasti zájmu rovnou v SQL dotazu.

  Souřadnice jednotlivých bodů polygonu jsou v rozsahu $\langle 0, 1 \rangle$ a vypočítají se jako podíl mezi skutečnými souřadnicemi bodů na obrázku a rozměry obrázku. Tím se zajistí, že systém nebude ovlivněn zpracováváním fotografií téhož parkoviště, které mají rozdílné rozlišení.

Tabulka `parking_spot`

Každé parkoviště má nějaká parkovací místa. Ať už se jedná o přesně ohraničená místa na parkovišti, nebo o plochy, která se zkrátka jen využívají k parkování. Pokud má systém úspěšně počítat auta na parkovišti a shromažďovat informace o daném parkovišti, potřebuje k tomu znát polohu jednotlivých parkovacích míst, aby bylo možné porovnávat, jestli se dvě auta ze dvou obrázků téhož parkoviště nachází na stejném parkovacím místě, či nikoliv.

Systém si parkovací místa detekuje a vytváří sám, na základě opětovného detekování totožného auta na stejném místě.

- `id` - Primární klíč, který jednoznačně určuje parkovací místo. Tento identifikátor se následně používá i pro jednoznačné určení toho, na kterém parkovacím místě bylo detekováno auto.
- `parking_lot_id` - Cizí klíč, jednoznačně určující parkoviště, na kterém se dané parkovací místo nachází.
- `box` - Obdélník definovaný levým horním a pravým dolním bodem. Tento obdélník přesně vymezuje hranice parkovacího místa na parkovišti.

  Souřadnice jednotlivých bodů obdélníku jsou v rozsahu $\langle 0, 1 \rangle$ a vypočítají se jako podíl mezi skutečnými souřadnicemi bodů na obrázku a rozměry obrázku. Tím se zajistí, že systém nebude ovlivněn zpracováváním fotografií téhož parkoviště, které mají rozdílné rozlišení.

- `valid` - Označení, zda bylo parkování validováno. Validace je provedena na základě detekce totožného auta na stejném místě na dvou časově následujících obrázcích. Tímto způsobem je možné odfiltrovat takové detekce, kde se nejedná o skutečné parkovací místo, ale o auto, které bylo zachyceno během popojíždění po parkovišti.
- `removed` - Označení, zda bylo parkovací místo manuálně odstraněno. Pakliže uživatel manuálně odstraní parkovací místo, veškeré detekce na tomto místě budou ignorovány.

Tabulka `record`

Systém pro počítání aut na parkovišti je navržen tak, že očekává, že obrázky parkoviště, které přijímá jsou pořizovány v časové posloupnosti v přibližně stejných časových intervalech. Informace o každém takto přijatém a zpracovaném snímku je v databázi uložená v tabulce `record`.

- `id` - Primární klíč, který jednoznačně určuje záznam vzniklý při přijetí a zpracování snímku z parkoviště.
- `parking_lot_id` - Cizí klíč, jednoznačně určující parkoviště, ze kterého přijatý snímek pochází.
- `created` - Časový údaj přijatý spolu s pořízeným snímkem, který označuje datum a čas vniku daného snímku.
- `filename` - Jméno zdrojového snímku, pod kterým jej lze dohledat v úložišti.

Tabulka `detection`

V tabulce `record` jsou uloženy základní informace o jednom pořízeném snímku. Tabulka `detection` naproti tomu slouží pro uložení informací o každé jednotlivé detekci, která vznikla při zpracovávání daného snímku. Každá detekce je tedy navázaná na tabulku `record` a označuje detekované auto na parkovišti.

- `id` - Primární klíč, který jednoznačně určuje detekci vzniklou při zpracování snímku z parkoviště.
- `record_id` - Cizí klíč, jednoznačně určující záznam (snímek) ke kterému daná detekce patří.
- `parking_stop_id` - Cizí klíč, jednoznačně určující parkovací místo na parkovišti, ze kterého byl snímek pořízen a na kterém bylo detekováno auto.
- `car_id` - Cizí klíč, určující, auto, které bylo na daném parkovacím místě v rámci snímku detekováno.
- `embedding` - *Embedding* vektor popisující vizuální vlastnosti detekovaného auta, tak jak je popsáno v kapitole 3.3. *Embedding* vektor je uložený u každé detekce zvlášť, protože v průběhu času se situace na parkovišti může měnit i tím pádem se ze snímku na snímek může lehce měnit také vizuální reprezentace auta (například když se postupně stmívá). Embedding vektor každé nové detekce se proto porovnává s posledním a tedy nejnovějším předchozím *embedding* vektorem na stejném parkovacím místě, aby se zamezilo veliké kumulaci vizuálních změn.

Tabulka `car`

Pakliže má systém počítat auta na parkovišti a sbírat o nich další statistické údaje, musí být schopen si informace o jednotlivých autech ukládat. Tabulka `car` slouží jako pomocná tabulka k tabulce `detection`.

Ačkoliv by změnu auta na parkovacím místě bylo možné z databáze zjišťovat opětovným porovnáváním *embedding* vektorů a hledáním výrazné změny, jednalo by se o poměrně náročnou operaci. Vezmeme-li v úvahu, že na jednom parkovacím místě se může vystřídat několik aut denně a tento počet se kumuluje v čase, pak by získávání informací o počtu aut, či o příjezdu a odjezdu jednotlivých aut, bylo výpočetně náročné.

Tabulka `car` celou tuto operaci značně zjednodušuje, protože jak změnu auta na parkovacím místě, tak časy příjezdu a odjezdu daného auta ukládá přímo.

- **`id`** - Primární klíč, který jednoznačně určuje auto, detekováno při zpracování snímku z parkoviště.
- **`start_timestamp`** - Časový údaj určující kdy bylo dané auto poprvé detekováno na parkovacím místě. Slouží tedy jako přibližný čas příjezdu auta.
- **`end_timestamp`** - Časový údaj určující kdy už dané auto nebylo detekováno na parkovacím místě. Slouží tedy jako přibližný čas odjezdu auta.

6.4 REST API

Architekturu *Representational State Transfer*, neboli REST popsal poprvé Roy Fielding v roce 2000 ve své dizertační práci [6]. Jedná se o sadu principů, pomocí kterých lze navrhovat aplikační rozhraní. Tato architektura se využívá hlavně při návrhu webových aplikačních rozhraní.

Základní entitou v REST architektuře je zdroj (*resource*). Zdrojem v REST architektuře mohou být jak stavy aplikace, tak samotná data, přičemž zdroje mají vždy předem definovanou strukturu a každý zdroj má zároveň vlastní identifikátor, pomocí kterého se na něj lze dotazovat, nebo jej upravovat. Zdroje můžou být navíc uspořádané v hierarchické struktuře.

REST API, které bylo navrženo pro aplikaci na detekci a počítání unikátních aut na parkovišti obsahuje několik zdrojů, na které je možné se dotazovat voláním na příslušný *endpoint*. V rámci této technické zprávy nebude každý **`endpoint`** rozebrán dopodrobna, včetně jeho parametrů a odpovědí, protože podrobná specifikace každého z nich se nachází ve formátu Open API[10] na přiloženém paměťovém médiu. Nicméně je vhodné uvést alespoň výčet jednotlivých *endpointů* a krátký popis toho, k čemu se tyto *endpointy* využívají.

Parkoviště

Základním zdrojem pro systém na detekci a počítání unikátních aut na parkovišti jsou parkoviště. Každé parkoviště obsahuje nejen základní informace, které jsou uloženy v databázové tabulce `parking_lot`, ale také informaci o tom, ze kdy pochází poslední zpracovaný snímek tohoto parkoviště, či kolik parkovacích míst bylo na parkovišti skutečně detekováno.

- **`POST /parking-lots`** - Vytváření nových parkovišť v systému.

[10] https://spec.openapis.org/oas/v3.1.0

- `GET /parking-lots` - Získání seznamu všech parkovišť v systému a patřičných detailů ke každému z nich.
- `GET /parking-lots/<id>` - Získání detailů ke konkrétnímu parkovišti v systému, které odpovídá zadanému identifikátoru.
- `PATCH /parking-lots/<id>` - Úprava údajů o konkrétním parkovišti v systému, které odpovídá zadanému identifikátoru.

Oblasti zájmu

Oblast zájmu je sice v databázi uložená v tabulce `parking_lot` jako atribut `roi`, spolu s ostatními informacemi o parkovišti, v REST API aplikace se však jedná o samostatný zdroj, který je hierarchicky pod parkovištěm. Důvodem tohoto oddělení je způsob, jakým se s oblastí zájmu v aplikaci pracuje. Je tedy možné jej načítat, nebo upravovat nezávisle na parkovišti, ke kterému náleží.

- `GET /parking-lots/<id>/regions-of-interest` - Získání oblasti zájmu patřící konkrétnímu parkovišti v systému, které odpovídá zadanému identifikátoru.
- `PATCH /parking-lots/<id>/regions-of-interest` - Úprava oblasti zájmu patřící konkrétnímu parkovišti v systému, které odpovídá zadanému identifikátoru.

Parkovací místa

Parkovací místa jsou opět zdroj, který se hierarchicky nachází pod parkovištěm. S parkovacími místy je však opět možné manipulovat bez ohledu na parkoviště. Jedná se proto o samostatný zdroj.

- `GET /parking-lots/<id>/parking-spots` - Získání všech parkovacích míst patřící konkrétnímu parkovišti v systému, které odpovídá zadanému identifikátoru. Vracená data obsahují i informaci o poloze každého parkovacího místa na parkovišti.
- `GET /parking-lots/<id>/parking-spots/<spot_id>` - Získání informací o konkrétním parkovacím místě, specifikovaném pomocí identifikátoru `spot_id`.
- `DELETE /parking-lots/<id>/parking-spots/<spot_id>` - Mazání konkrétního parkovacího místa, specifikovaného identifikátorem `spot_id`.

Snímky parkoviště

Po vytvoření parkoviště v systému se očekává, že do něj začnou proudit snímky daného parkoviště. Snímek parkoviště je tedy další samostatný zdroj, který v REST API aplikace vystupuje.

- `POST /parking-lots/<id>/frames` - Odeslání snímku parkoviště ke zpracování. Zpracování snímku, které je iniciováno odesláním snímku na tento *endpoint* je poměrně složité a je podrobněji popsané v kapitole 7.
- `GET /parking-lots/<id>/frames/latest` - Získání posledního zpracovaného snímku parkoviště, za účelem jeho zobrazení v uživatelském rozhraní aplikace.

Statistiky

Systém pro detekci a počítání unikátních aut na parkovišti by měl detekci a počítání nejen provádět, ale také ukládat a agregovat do užitečných statistických údajů. Ačkoliv je možné si tyto statistiky získat pomocí SQL dotazu přímo z databáze, není to však příliš uživatelsky přívětivé. Proto se o agregaci statistických údajů stará sama aplikace. Tyto statistiky jsou opět zdrojem v REST API.

Statistiky jsou vždy počítané jak pro parkoviště jako celek, tak pro každé parkovací místo zvlášť. Je tedy možné zjisti například jak průměrný čas, který libovolné auto stráví na parkovišti, tak to, na kterých parkovacích místech je tento strávený čas vyšší. O tom, jaké konkrétní statistické údaje systém agreguje a jakým způsobem jsou počítané, pojednává kapitola 8.

- **`GET /parking-lots/<id>/cumulative-statistics`** - Získání kumulativních statistických údajů. Tyto údaje jsou průměrné hodnoty agregovaných veličin za celou dobu.
- **`GET /parking-lots/<id>/full-statistics`** - Získání statistických údajů, které zahrnují jak kumulativní hodnoty, tak jednu hodnotu každé agregované veličiny pro jeden časový krok. Tento časový krok může být v rozsahu jedné minuty, až jednoho roku. Je tedy možné získat si například histogram obsazenosti parkoviště s časovým krokem jedné hodiny.

Detekce

Jediným, čistě interním, zdrojem, který se v REST API aplikace nachází je zdroj detekce. Detekce vzniká při předzpracování snímku parkoviště a ve formě tohoto zdroje je předávaná na jiná místa v aplikaci k dalšímu zpracování. Detail zpracování snímků v aplikaci, včetně popisu nutnosti existence tohoto zdroje je popsán v kapitole 7.

- **`POST /detections/<record_id>`** - Odeslání detekcí, vzniklých při předzpracování snímku parkoviště. Tento *endpoint* je aplikací využíván pouze interně a není zamýšlen pro externí využití.

6.5 Uživatelské rozhraní a funkcionalita

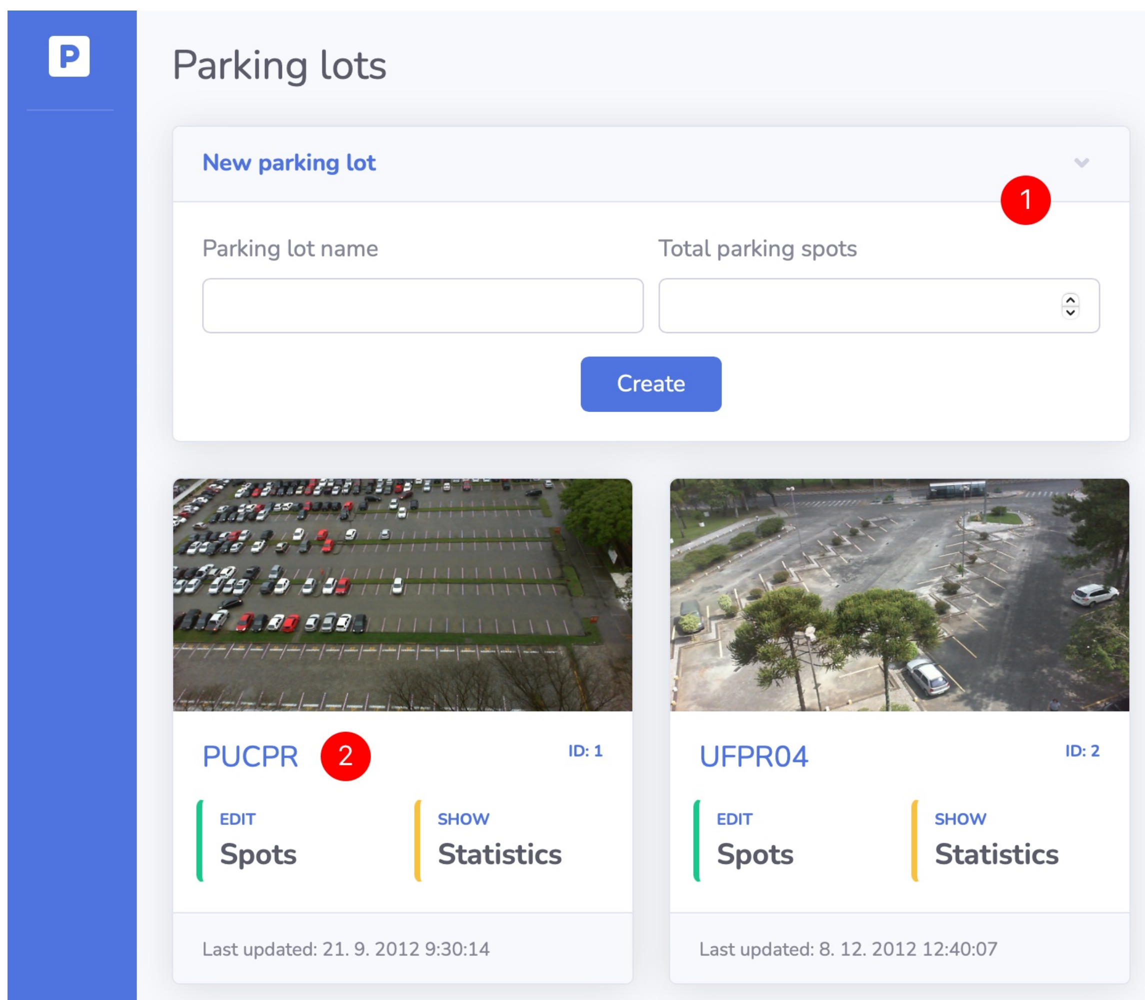
Získávání čí úprava informací pomocí různých *endpointů* REST API aplikace, je výborný prostředek pro komunikaci systémů na strojové úrovni. Pakliže je třeba z kamer na parkovišti automaticky odesílat snímky ke zpracování, nebo naopak automaticky stahovat statistická data ze serveru, vystavené REST API naplňuje tyto potřeby. Většinou však se systémem pracují lidé, pro které není strukturovaná textová odpověď ze serveru moc dobře čitelná. V takovém případě je možné použít webovou aplikaci s grafickým uživatelským rozhraním.

Uživatelské rozhraní aplikace na detekci a počítání unikátních aut na parkovišti je založeno na open source šabloně SB Admin 2 [1], která využívá framework Bootstrap 4[11]. Z původní šablony byly však nakonec využity pouze kaskádové styly upravující vzhled stránky. Pro množství interaktivních prvků, které se v uživatelském rozhraní nachází byla dále použita JavaScript knihovna p5.js[12], která usnadňuje práci s HTML *canvasem*.

[11] https://getbootstrap.com
[12] https://p5js.org

Přehled a vytváření parkovišť

Jak již bylo zmíněno při popisu REST API, před tím, než je možné začít zpracovávat snímky z nějakého parkoviště, musí být toto parkoviště nejprve v systému vytvořeno. Základní stránka po načtení webové aplikace proto patří právě přehledu již vytvořených parkovišť. Navíc se na této stránce nachází formulář pro vytvoření nového parkoviště. Tato stránka je zachycená na obrázku 6.3.



Obrázek 6.3: Stránka pro přehled a vytváření parkovišť. Horní dlaždice (1) obsahuje formulář pro vytváření nového parkoviště. Pod ním jsou dlaždice pro jednotlivá již existující parkoviště (2).

V horní části stránky se nachází formulář pro vytvoření nového parkoviště. Tento formulář je na obrázku 6.3 označen číslem **1**. Pro vytvoření nového parkoviště v systému není třeba mnoho informací. Formulář tedy vyžaduje pouze jméno nového parkoviště a počet parkovacích míst na parkovišti.

Poté, co uživatel klikne na tlačítko "Create", jsou tato data odeslána na *endpoint* `POST /parking-lots`. Po obdržení odpovědi o úspěšném vytvoření parkoviště v systému je novému parkovišti dynamicky vytvořená nová karta v přehledu parkovišť. Stránku tedy není

potřeba načítat znovu. V případě chybně zadaných dat se chybové hlášky zobrazí přímo u formulářových polí, ke kterým patří.

Existující parkoviště v systému jsou zobrazeny jako karty pod formulářem na vytvoření nového parkoviště. Na obrázku 6.3 je jedna z těchto karet označená číslem **2** Aby nebylo načítání stránky při velikém množství parkovišť příliš dlouhé, probíhá načítání existujících parkovišť a vytváření jejich karet na stránce opět dynamicky. Data jsou získávána z *endpointu* `GET /parking-lots`.

Každá karta obsahuje náhled parkoviště, pro který se používá poslední zpracovaný snímek daného parkoviště. Následně je na kartě název a ID parkoviště, datum a čas, ze kdy pochází poslední zpracovaný snímek parkoviště a nakonec tlačítka, která odkazují na detail parkoviště se správou parkovacích míst a zobrazení statistik o daném parkovišti.

Detail parkoviště

Po kliknutí na tlačítko "Edit Spots" na kartě parkoviště, je uživatel přesměrován na stránku pro editaci údajů o parkovišti a správu parkovacích míst. Na horním okraji stránky se zobrazuje hlavička se jménem a ID parkoviště, které je právě editováno. Na postranním panelu se nachází logo, které zároveň slouží jako jako odkaz pro návrat na předchozí stránku s přehledem všech parkovišť.

Obrázek 6.4: Stránka s detailem parkoviště, kde lze editovat základní informace o parkovišti (1), mazat falešné parkovací místa (2) a upravovat oblast zájmu na parkovišti (3).

Karta, těsně pod hlavičkou, která je na obrázku 6.4 označená číslem **1**, vypadá totožně, jako karta s formulářem pro vytvoření nového parkoviště. V tomto případě jsou však již hodnoty formulářových polí předvyplněné hodnotami daného parkoviště. Tento formulář

tedy slouží pro jejich editaci. Po zmáčknutí tlačítka "Save" jsou data dynamicky uložena pomocí *endpointu* `PATCH /parking-lot/<id>`.

Prvek označený číslem **2** na obrázku 6.4 je rozklikávací karta, kde je možné prohlížet si detekované parkovací místa a také mazat parkovací místa, která byla detekovaná nesprávně. Detail této karty je zachycen na obrázku 6.5.

Posledním prvkem na této stránce je opět rozklikávací karta, označená na obrázku 6.4 číslem **3**. Na této kartě jsou také zobrazená parkovací místa, avšak tentokrát není možné je vybírat a mazat. Místo toho se v této kartě nastavuje oblast zájmu, která určuje, který úsek parkoviště má být brán v potaz při agregaci statistických údajů. Karta s oblastí zájmu je zobrazena na obrázku 6.6.

Mazaní falešných parkovacích míst

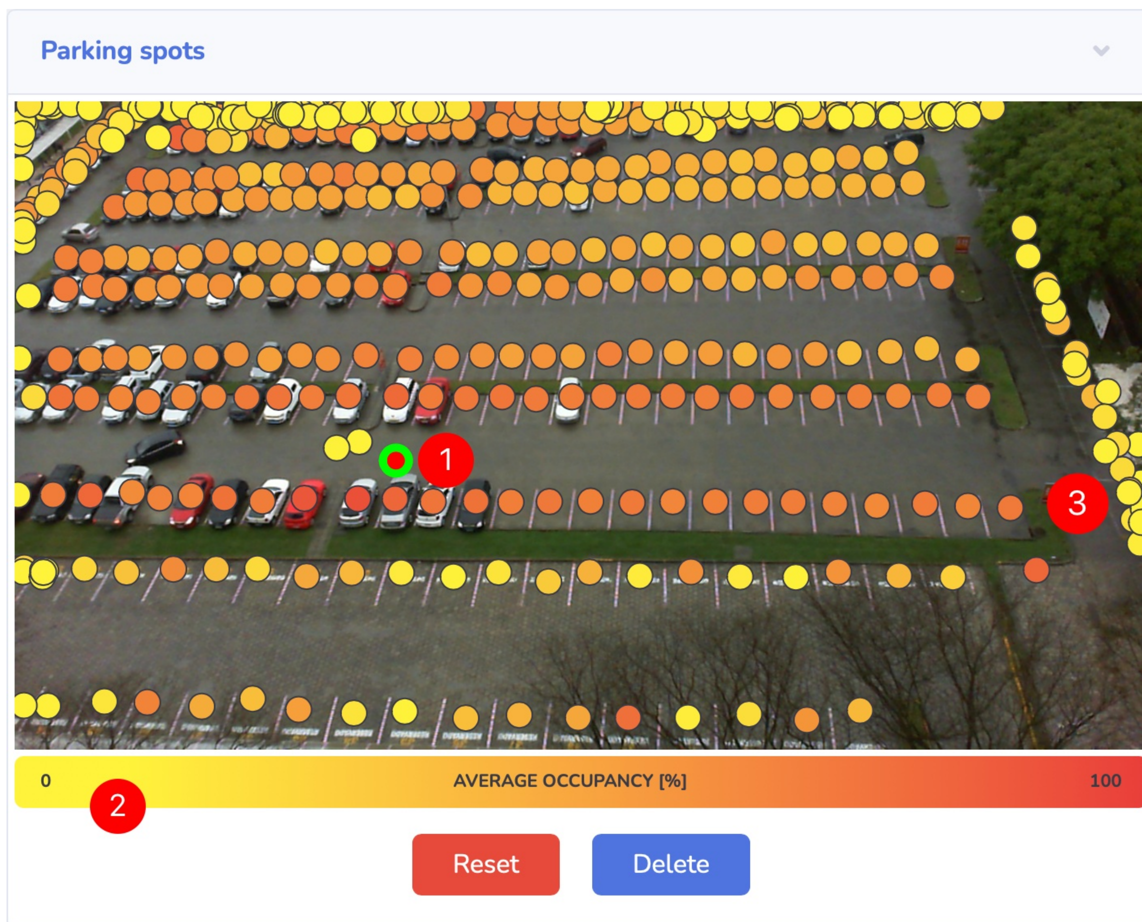
Systém pro detekci a počítání unikátních aut na parkovišti závisí na správné detekci parkovacích míst na parkovišti. Někdy se však může stát, že detektor detekuje jako parkovací místo oblast, která ve skutečnosti parkovacím místem není. Systém má mechanizmy, kterými se podobným falešným detekcím snaží zabránit. Tyto mechanizmy jsou spolu s celým procesem zpracování snímků parkoviště popsané v kapitole 7. V případě, že však není falešná detekce jedním z těchto mechanizmů odchycena, je možné toto parkovací místo manuálně smazat.

Karta pro mazání falešných parkovacích míst se nachází na stránce detailu parkoviště a je vyobrazená na obrázku 6.5. Polohy parkovacích míst jsou dynamicky získávány z *endpointu* `GET /parking-lots/<id>/parking-spots`. Každé parkovací místo je na snímku parkoviště označeno jako barevný kruh, který je vykreslený na jeho místě. Celá karta je interaktivní a na každé parkovací místo je možné kliknout a označit ho tak. Označená místa jsou zbarvená do sytě červené barvy. Naposledy označené místo má navíc sytě zelený okraj. Takto vybrané místo je na obrázku 6.5 označené číslem **1**. Po opětovném kliknutí na označené parkovací místo bude toto místo odoznačeno.

Aplikace se s identifikací falešných parkovacích míst snaží uživateli pomoct. Pomocí endpointu `GET /parking-lots/<id>/cumulative-statistics` získá kumulativní statistiky pro jednotlivá parkovací místa. Tyto informace jsou také načítány dynamicky, aby načítání samotné stránky zbytečně nezpomalovaly. Součástí těchto statistik je také průměrná obsazenost každého parkovacího místa. Pokud je parkovací místo obsazeno často a po dlouhé časové úseky, dá se předpokládat, že je to skutečné parkovací místo. Pokud je ale naopak parkovací místo obsazeno jen zřídka, může se jednat o falešnou detekci. Každé parkovací místo, vykreslené na této kartě má proto přiřazenou barvu, odpovídající průměrné obsazenosti daného parkovacího místa. Číslem **2** je na obrázku 6.5 označena škála, sloužící jako legenda pro toto barevné kódování. Parkovací místa, které jsou téměř nepoužívané jsou tedy zabarvené žlutě a ty, které jsou používané často mají odstín červené. Výsledek pak připomíná jakousi teplotní mapu[13], kde je však místo teploty vyznačená průměrná obsazenost.

Díky tomuto grafickému znázornění pak uživatel okamžitě vidí, která parkovací místa jsou podezřelá a po vlastním vyhodnocení je pak může odstranit. U značky s číslem **3** na obrázku 6.5 se nachází příklad parkovacích míst, která mají nízkou průměrnou obsazenost a jsou tedy adepty pro odstranění.

[13]https://en.wikipedia.org/wiki/Heat_map

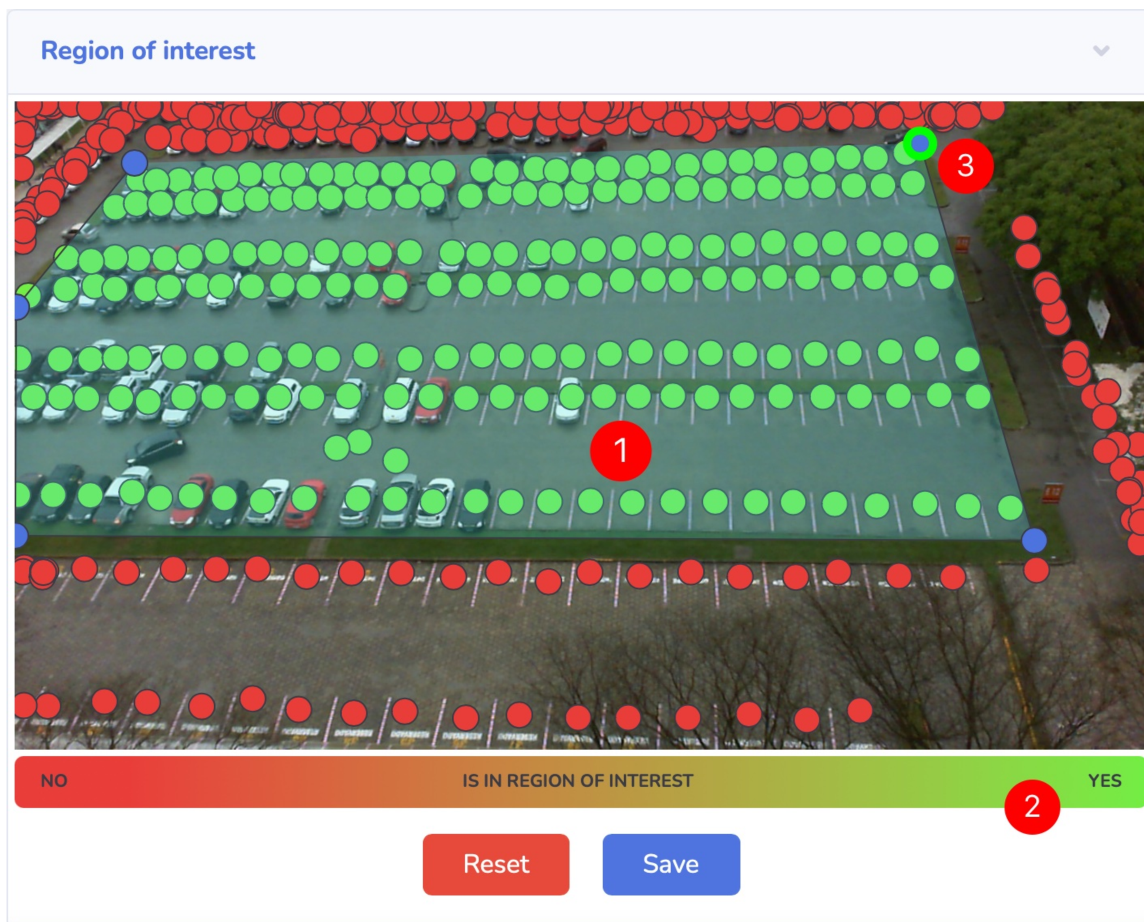


Obrázek 6.5: Zobrazení parkovacích míst na parkovišti. Parkovací místa je možné vybírat (1) a mazat. Každé parkovací místo je barevně kódováno (2) podle jeho průměrné obsazenosti. Podezřelá místa jsou pak jasně rozlišitelná (3).

Úprava oblasti zájmu

Množství dat, které je systémem pro detekci a počítání unikátních aut na parkovišti generováno může být veliké. Někdy však uživatelé nemusí chtít zobrazit statistiky pro celé parkoviště najednou, ale stačila by pouze část. Někdy také uživatel nemusí chtít odstraňovat falešné parkovací místa manuálně, ale přesto nechce, aby mu data z nich ovlivňovala statistiku. A někdy může na parkoviště mířit více kamer, přičemž úhel záběru těchto kamer se v nějaké oblasti překrývá, což by způsobilo dvojí započítání stejných aut. Zkrátka někdy je žádoucí definovat si oblast zájmu, ze které jsou statistiky počítány. Tento přístup má zároveň tu výhodu, že data mimo oblast zájmu sice nejsou započítána do statistiky, ale nejsou ztracena. Kdykoliv je tedy možné oblast zájmu změnit a veškerá data k vypočítání statistik z nové oblasti zájmu stále existují.

Webová aplikace nastavení a používání oblasti zájmu podporuje a úprava oblasti zájmu je pro každé parkoviště možná na stránce detailu daného parkoviště. Karta ve které je možné oblast zájmu upravovat je zachycená na obrázku 6.6.



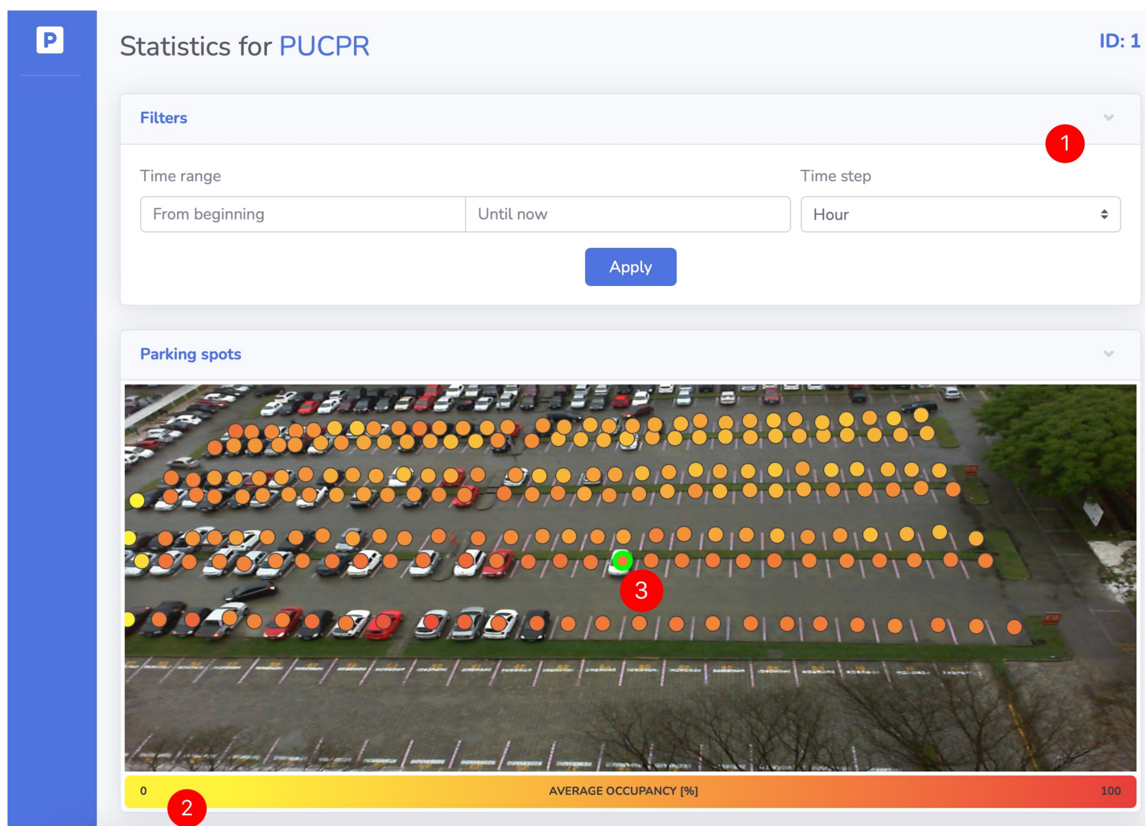
Obrázek 6.6: Zobrazení a editace oblasti zájmu (1) na parkovišti. Podle barevného zvýrazněni parkovacích míst (2) je okamžitě vidět, která parkovací místa do oblastí zájmu patří. Klikáním na obrázek parkoviště je možné přidávat nové body polygonu. Klikáním na zvýrazněné body (3) je možné je odstranit.

Pod číslem **1** na obrázku 6.6 je možné vidět oblast zájmu, která byla pro parkoviště nastavena. I oblast zájmu je pro rychlejší počáteční načtení stránky, načítána dynamicky z *endpointu* `GET /parking-lots/<id>/regions-of-interest`. Obdobně jako v případě karty na mazání parkovacích míst, jsou i zde jednotlivá parkovací místa barevně kódovaná. V tomto případě však barva označuje jednoduše to, jestli se parkovací místa nachází, či nenachází uvnitř oblasti zájmu.

Oblast zájmu je v podstatě polygon, tvořen jednotlivými body. Pomocí přidávání, či odstraňování těchto bodů je tedy možné i měnit tvar oblasti zájmu. Kliknutím kdekoli na obrázek parkoviště může uživatel přidat nový bod, který bude spojen s předchozím aktivním bodem. Nově vložený bod se stane označeným, což je indikováno sytě zeleným okrajem, což je na obrázku 6.6 vyznačeno číslem **3**. Pokud uživatel opět klikne na aktivní bod, bude tento bod smazán. Nemít žádnou oblast zájmu, nebo mít oblast zájmu s nulovou plochou je nicméně nevalidní. Pokud si tedy uživatel nepřeje započítávat do statistik pouze nějakou podoblast celého parkoviště, měl by oblast zájmu nastavit na celý snímek. Toho lze dosáhnout buď manuálně, přidáním jednotlivých bodů, nebo pomocí tlačítka “Reset”.

Zobrazení statistik

I když je prohlížení parkovacích míst s možností jejich mazání, či upravování oblasti zájmu užitečná funkcionalita, která vylepšuje celkovou použitelnost aplikace, největším přínosem grafického uživatelského rozhraní je vizuální reprezentace statistických údajů, které se systému podařilo nasbírat. Z původního menu se seznamem všech parkovišť je možné se na stránku se statistikami některého parkoviště dostat pomocí tlačítka “Show Statistics” na kartě daného parkoviště. Stránku se statistikami můžeme pomyslně rozdělit na dvě části. První část je převážně interaktivní a druhá pouze zobrazuje data.

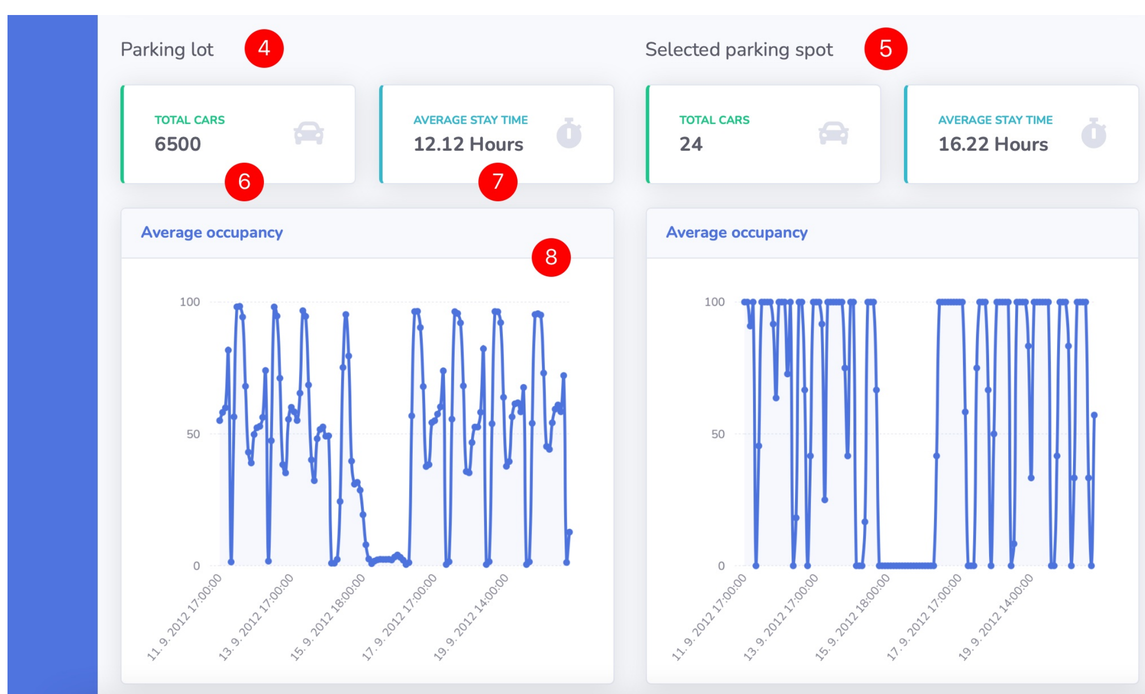


Obrázek 6.7: Stránka se statistikami pro zvolené parkoviště. Statistiky je možné filtrovat podle času (1). Parkovací místa jsou barevně kódovaná podle průměrné obsazenosti (2). Pro detailnější statistiky konkrétního parkovacího místa je možné toto místo vybrat (3).

Na samotném vrchu stránky se opět nachází hlavička se jménem parkoviště a jeho identifikátorem. Pod hlavičkou je pak karta s filtry. Tu je na obrázku 6.7 možné vidět pod číslem **1**. Filtrovat lze podle data pořízení snímku a to buď zadáním minimálního data pořízení, maximálního data pořízení, nebo obou dat najednou. Nejzajímavější položka ve filtru je však položka s názvem “Time step”. Hodnoty tohoto pole lze nastavit v rozsahu jedné minuty, až jednoho roku a jedná se o časové rozlišení statistik. Pokud si tedy uživatel prohlíží data za celý rok, pravděpodobně nebude chtít tyto data prohlížet po minutách, ale mohly by mu stačit měsíce, případně dny. Opět jsou veškerá statistická data načítaná dynamicky z *endpointu* `GET /parking-lots/<id>/full-statistics`.

Parkovací místa jsou opět, barevně kódovaná podle průměrné obsazenosti, avšak nezobrazují se už všechna, ale pouze ty, které spadají do nastavené oblasti zájmu. Na obrázku 6.7 je hned vidět, že parkovací místa jsou podle průměrné obsazenosti kódovaná do škály zobrazené u čísla **2**. Už i toto kódování může sloužit ke statistickému vyhodnocování. Kamera ze které je parkoviště snímáno se v tomto případě nachází na budově. Na zbarvení jednotlivých parkovacích míst je vidět. Že řada parkovacích míst nejblíže budově bývá nejčastěji obsazená. Každá další řada dále od budovy pak bývá obsazená čím dál tím méně.

Většina agregovaných statistických údajů se však nachází až pod obrázkem parkoviště, a to ve dvou sadách. První sada, na obrázku 6.8, označená číslem **4**, zobrazuje data pro parkoviště jako celek. Vedle ní se nachází druhá sada, označená číslem **5**, ve které se nachází statistiky pro jedno, aktuálně zvolené, parkovací místo. Parkovací místa lze vybírat klikáním na ně na obrázku parkoviště. Aktivní parkovací místo je na obrázku 6.7 zobrazeno u čísla **3**.



Obrázek 6.8: Statistické údaje jak pro celé parkoviště (4), tak pro aktuálně vybrané parkovací místo (5). Parkoviště i parkovací místo mají stejnou sadu statistik. Celkový počet aut (6), průměrná doba stání jednoho auta (7) a graf obsazenosti (8).

Každá sada, jak pro parkoviště, tak pro parkovací místo obsahuje stejné typy údajů, samozřejmě však specifická pro ně. Pod číslem **6** se nachází celkový počet aut na parkovišti, případně na parkovacím místě, za zadaný časový úsek. Pod číslem **7** je pak průměrná doba parkování jednoho auta. A nakonec je u čísla **8** vidět graf obsazenosti parkoviště, případně parkovacího místa. Na konkrétním příkladě na obrázku 6.8 je vidět opakující se denní vzorec naplněnosti parkoviště, přičemž dne 16.9.2012 byla neděle, proto bylo parkoviště téměř prázdné. O tom, jak jsou všechny tyto statistické údaje vypočítané pojednává kapitola 8.

Kapitola 7

Zpracování snímků z parkoviště

Základním předpokladem pro detekci aut na parkovišti, jejich počítání a sbírání jiných statistických údajů o parkovišti je schopnost řádně zpracovat snímky parkoviště, které přichází v časové posloupnosti. Při tomto zpracování snímků vzniká většina údajů, ze kterých je poté systém schopen agregovat strukturovaná statistická data.

Už v kapitole 4.2 byl zjednodušeně popsán princip zpracování snímků. V této kapitole budou jednotlivé operace popsané podrobněji.

7.1 Zachování pořadí snímků

Z jediného snímku parkoviště se moc užitečných statistických dat vytěžit nedá. Lze na něm detekovat auta, ale už není jednoduše možné určit, jestli detekovaná auta parkují, nebo jen popojíždí po parkovišti. Nelze ani určit čas příjezdu a odjezdu auta a tím ani dobu strávenou na parkovišti. Většina těchto informací se ale začne vynořovat, jakmile budeme mít snímků víc.

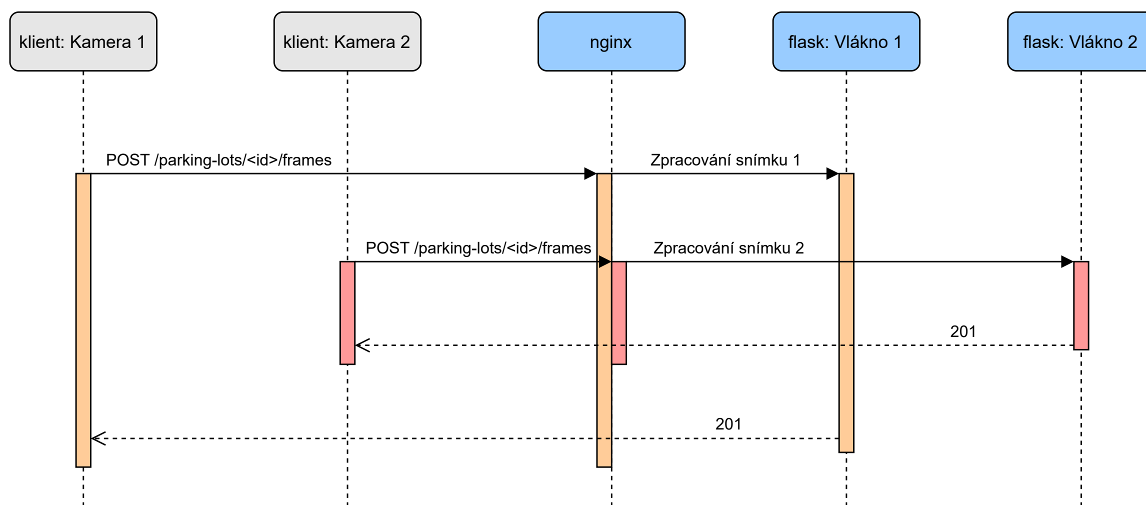
Pakliže na více snímcích detekujeme auta na stejných místech, dá se předpokládat, že se jedná o parkovací místa. Pokud je naopak auto detekováno na určitém místě pouze na jednom snímku z mnoha, tak se pravděpodobně o parkovací místo nejedná a tímto místem akorát auto při pořízení snímku zrovna projíždělo.

Známe-li navíc čas pořízení těchto snímků a seřadíme-li je podle tohoto času, pak je možné z těchto snímků extrahovat ještě větší množství informací. Pokud je na jednom snímku parkovací místo prázdné a na následujícím na něm parkuje auto, pak víme, že auto na toto místo muselo přijet časově někdy mezi pořízením těchto dvou snímků. To stejné platí i opačně. Je takhle tedy možné získat přibližnou představu o příjezdu a odjezdu každého auta. Zároveň je také možné počítat auta přesněji. Pokud se stejné auto nachází po několik snímků na tomtéž parkovacím místě, nemusíme ho počítat více-krát. Pokud se naopak auto na parkovacím místě v čase změní, je potřeba ho započítat jako nové.

Je tedy zřejmé, že pro vytěžení co největšího počtu informací ze snímků z parkoviště potřebujeme nejen znát časový údaj o pořízení snímku, ale také zachovat správné pořadí snímků, podle času pořízení, protože výsledek zpracování konkrétního snímku je přímo závislý na výsledcích zpracování předchozích snímků. V případě přijímání snímků přes REST API, může být ale zachování správného pořadí problematické. Je tedy potřeba zpracování snímků nějakým způsoben synchronizovat tak, aby se další snímek v časové posloupnosti začal zpracovávat až poté, co byl zpracován snímek předchozí.

Předpokládejme, že na vystavené REST API, které přijímá ke zpracování jednotlivé snímky sekvence, posílá klient, nebo několik klientů snímky v pořadí, v jakém byly skutečně pořízeny, avšak nečeká nutně na odpověď, která značí dokončení zpracování. Může se jednoduše stát, že klient pošle další snímek v pořadí ještě v době, kdy zpracování prvního nebylo dokončeno. Server tedy začne v novém vlákně zpracovávat i druhý snímek. Pakliže se zpracovávání druhého snímku dokončí rychleji, než zpracovávání prvního, nastává problém, protože pro správné fungování systému je kritické, že se zpracování snímků dokončí ve správném pořadí. Tato situace je vyobrazena na sekvenčním diagramu 7.1.

Modře zabarvené objekty odpovídají stejně pojmenovaným Docker kontejnerům, které byly blíže popsané v kapitole 6.2. Požadavek na API tedy dorazí nejprve na NGINX server, který tento požadavek distribuuje mezi jednotlivá vlákna, ve kterých běží Flask aplikace.



Obrázek 7.1: Příklad chyby při zpracování snímků, která může nastat, pakliže se pozdější snímek v sekvenci zpracovává kratší dobu, než dřívější snímek v sekvenci.

V případě, že by nebylo třeba zpracovávat obrázky postupně, přesně v pořadí, v jakém byly pořízeny, bylo by toto chování naprosto v pořádku. Pokud je ale potřeba zachovat pořadí zpracovávání, nabízí se dvě možná řešení.

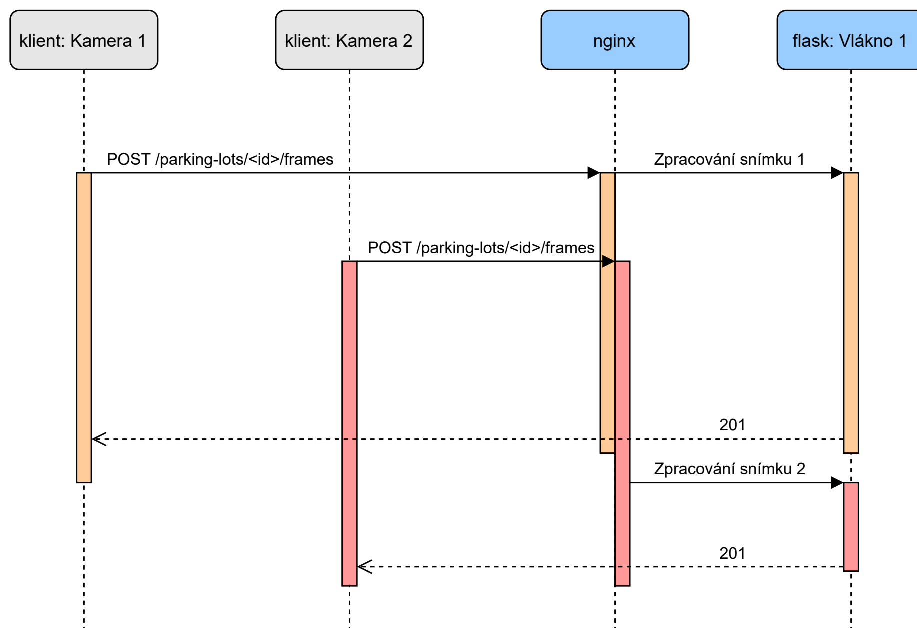
Zpracovávání požadavku na API v jednom vlákně

První a nejjednodušší možné řešení daného problému spočívá v eliminaci paralelního zpracování jednotlivých požadavků, které na API přichází. Tato možnost je zobrazená na sekvenčním diagramu 7.2.

Jak je viděl na diagramu, problém se zachováním pořadí zpracování snímků je zde úspěšně vyřešen. Požadavek od klienta *Kamera 1*, dorazí na API jako první a také jako první se dokončí jeho zpracovávání. Okamžitě je však z diagramu zřejmé, v čem spočívá nevýhoda tohoto řešení.

Ačkoliv přijde požadavek od klienta *Kamera 2* krátce poté, co přišel požadavek od klienta *Kamera 1*, čeká, až se první požadavek kompletně zpracuje. Teprve poté se začne zpracovávat druhý obrázek a po jeho dokončení je vrácena odpověď klientovi *Kamera 2*. V případě, že by takovýmto způsobem přišlo veliké množství požadavků krátce po sobě, musel by klient, který poslal poslední požadavek, čekat až se dokončí zpracovávání všech

požadavků před ním. Pokud by zpracování předchozích požadavků trvalo příliš dlouho, riskujeme dokonce, že dotaz skončí chybou `408 Request Timeout Error`. Je tedy zřejmé, že i když tento způsob řeší původní problém zachování pořadí zpracování snímků, nevyhovuje v jiných ohledech.



Obrázek 7.2: Zpracovávání požadavků na API sekvenčně v jednom vlákně.

Oddělení zpracování snímku od požadavku na API

Na předchozím příkladu je vidět, že tím, že bude server zpracovávat snímky sekvenčně v rámci zpracování požadavků na API, se výrazně prodlužuje doba odpovědi na daný požadavek. Na jednu stranu tedy předchozí způsob vyřešil původní problém se sekvenčním zpracováním snímků parkoviště, na stranu druhou ale představil nový problém, dlouhého čekání na odpověď. Je tedy možné zaměřit se pouze na řešení nově vzniklého problému.

Výpočetně a tedy i časově nejnáročnější částí zpracování požadavku se snímkem z parkoviště je samotné zpracování snímku. Pokud by bylo možné v rámci zpracování požadavku na API pouze naplánovat toto zpracování snímku do nějaké synchronní fronty, dal by se každý požadavek na zpracování snímku přes API zpracovat přibližně stejně rychle a celkově za daleko kratší čas, než kdyby celé zpracovávání snímku probíhalo přímo v něm.

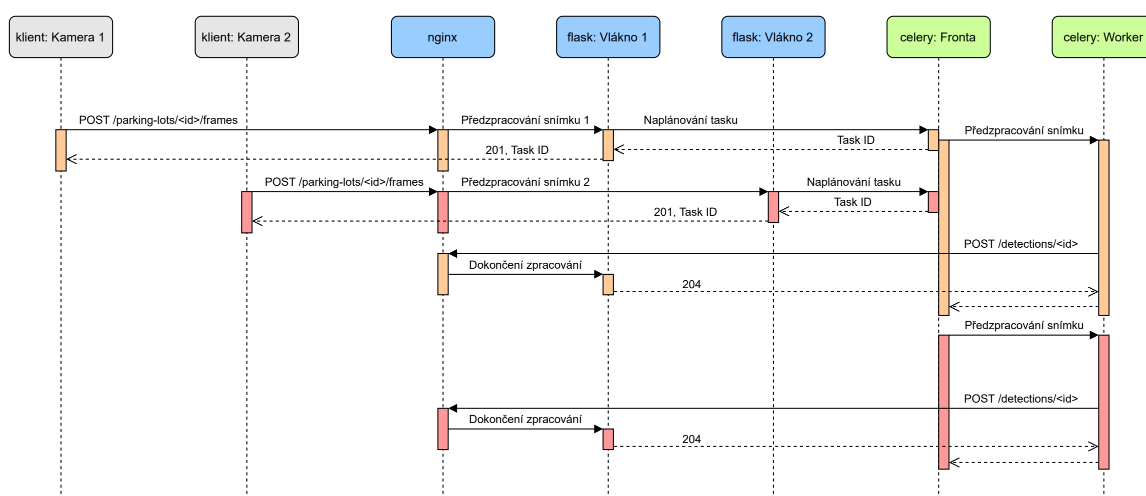
Aplikace pro detekci a počítání aut na parkovišti má pro tento případ speciální kontejner `celery`, ve kterém běží synchronní Celery *worker*, což je služba dedikovaná právě pro předzpracovávání snímků z parkoviště, jakožto výpočetně nejnáročnější operace. Na sekvenčním diagramu 7.3 je zaznačen celý proces synchronizace příchozích požadavků se zárukou, že snímky budou zpracovány v pořadí kdy přišly, bez ohledu na to, jak dlouho trvá jejich jednotlivé zpracování.

Podobně jako v prvním řešení, je i zde zaručeno zpracování ve správném pořadí tím, že jednotlivé snímky se zpracovávají sekvenčně. Rozdíl je zde však patrný na první pohled. Zpracování snímku se nyní v rámci zpracování požadavku pouze naplánuje jako úloha do

sdílené synchronní fronty. Samotný požadavek je tedy vyřízen velice rychle. A server jich takhle může obsloužit daleko více.

Celery *worker* si pak naplánované úlohy ze sdílené synchronní fronty postupně přiřazuje a zpracovává jeden po druhém. V rámci úlohy však dochází pouze k před-zpracování snímků, tedy k detekci aut na snímku a k následnému výpočtu *embedding* vektorů těchto detekovaných aut. Pro další zpracování těchto hodnot je potřeba je nejprve porovnat s hodnotami detekcí na předchozích snímcích, vyhodnotit toto porovnání a nové hodnoty do databáze vložit. Vzhledem k tomu, že kontejner `flask` je jediný, který má přímý přístup do databáze, je potřeba hodnoty vzniklé před-zpracováním snímku v Celery úloze poslat zpátky do Flask aplikace k finálnímu porovnání a uložení. Celery *worker* čeká, dokud toto porovnání a uložení ve Flask aplikaci proběhne a následně začne zpracovávat další snímek v pořadí.

Tímto způsobem je zaručeno správné pořadí zpracovávání snímků i rychlá odezva API.



Obrázek 7.3: Synchronizace zpracování příchozích snímků za využití Celery fronty s jedním *workerem*. Tento způsob je také využíván aplikací pro počítání aut na parkovišti.

7.2 Detekce aut a výpočet *embedding* vektorů

Detekce aut na snímcích parkoviště i výpočet *embedding* vektorů probíhá v kontejneru `celery` jako Celery úloha, zejména kvůli své výpočetní náročnosti. Celý proces sestává hlavně z detekce pomocí detektoru *YOLOv4* a následně výpočtu *embedding* vektorů pro jednotlivá auta pomocí modelu, který byl popsán v kapitole 5. Mezi detekcí a výpočtem *embedding* vektoru se však provádí několik mezikroků, které stojí za zmínku. Celý proces tedy může být shrnut do několika kroků.

1. **Načtení modelů** - Pomocí knihovny OpenCV a její třídy `dnn_DetectionModel` je načten konfigurační soubor architektury sítě *YOLOv4* pro detekci aut a následně i její váhy. Rozměry vstupní vrstvy detektoru jsou nastaveny na 416×416 pixelů. Pomocí knihovny Keras je pak načten i model pro výpočet *embedding* vektorů aut. Toto načítání jak knihoven, tak konkrétních modelů a vah je poměrně časově náročné, probíhá však pouze jednou, při startu `celery` kontejneru.

2. **Detekování aut na snímku parkoviště** - Samotná detekce probíhá pomocí metody `dnn_DetectionModel.detect`. Tato metoda vrací mimo jiné seznam všech detekovaných rámců, ohraničujících detekovaná auta. Tyto rámce jsou ve formě $(x, y, width, height)$. První dvě hodnoty jsou souřadnice levého horního rohu detekovaného rámce. Poslední dvě hodnoty jsou šířka a výška detekovaného obdélníku v pixelech. Bez ohledu na to, jaké jsou nastavené rozměry vstupní vrstvy detektoru, vrácené hodnoty jsou vždy navázané na původní rozměry snímku.

3. **Úprava formátu souřadnic detekovaných rámců** - Původní formát detekovaných rámců $(x, y, width, height)$, které jsou výsledkem detektoru není vyhovující pro budoucí práci s detekcemi a pro uložení do databáze. Jak již bylo popsáno v kapitole 6.3, PostgreSQL datový typ `box`, který se používá pro uložení rámce parkovacích míst, používá pro definici obdélníku levý horní a pravý dolní bod. Navíc jsou souřadnice těchto bodů normalizované tak, aby nebyly závislé na rozlišení fotografie.

   Rovnice 7.1 ukazuje způsob převodu souřadnic detekovaných rámců do požadovaného formátu, přičemž hodnoty $frame_width$ a $frame_height$ označují rozměry původního snímku.

$$\begin{aligned} point_{left_top} &= \left(\frac{x}{frame_width}, \frac{y}{frame_height}\right) \\ point_{right_bottom} &= \left(\frac{x + width}{frame_width}, \frac{y + height}{frame_height}\right) \end{aligned} \tag{7.1}$$

4. **Příprava aut pro výpočet *embedding* vektorů** - Před výpočtem *embedding* vektorů pro jednotlivá auta musí přípravou projít i snímek parkoviště. Model pro výpočet *embedding* vektorů na vstupu očekává snímky o rozměrech 64×64 pixelů s tím, že se na tomto snímku nachází pouze jedno auto. Prvním krokem je tedy z jednoho snímku parkoviště vyříznutí úseků, na kterých byly detekovaná auta. K tomu poslouží souřadnice rámců, které vrátil detektor.

   Tyto rámce však většinou nemají čtvercový tvar o velikosti 64×64 pixelů. Proto jsou vyříznuté obrázky aut nejprve zvětšeny, či zmenšeny tak, aby jejich delší hrana měřila 64 pixelů a následně jsou do těchto obrázků doplněny černé okraje tak, aby měl výsledný obrázek tvar čtverce.

   V neposlední řadě je ve snímku prohozen červený a modrý barevný kanál, protože knihovna OpenCV, kterou je obrázek načítán, používá uspořádání kanálů BGR, ale knihovna Keras naopak více konvenční uspořádání RGB.

5. **Výpočet *embedding* vektorů pro jednotlivá auta** - Po před-připravení výřezů jednotlivých aut jsou tyto snímky uspořádány do seznamu za sebou. Celý tento seznam je následně vložen do modelu, který pro každé auto zvlášť vypočítá *embedding* vektor. Vložení všech aut do modelu najednou, jako jednotlivé vzorky, je co do celkového času potřebného na zpracování výrazně rychlejší, že provádění výpočtu *embedding* vektoru po jednom.

6. **Odeslání výsledku do Flask aplikace** - Detekcí aut na snímku a výpočtem *embedding* vektorů pro tato auta končí před-zpracování snímku parkoviště. Detekce a *embedding* vektory je však ještě potřeba porovnat s daty z předchozích snímků a uložit je do databáze. Jsou proto odeslány zpátky do Flask aplikace na speciální dedikovaný REST API endpoint `POST /detections/<id>`.

7.3 Párování parkovacích míst

Pakliže systém zpracovává pouze fotografie a nemá k dispozici žádné dodatečné informace o parkovacích pozicích, může i určení konkrétního parkovacího místa znamenat problém. S každým novým snímkem se podmínky při pořizování tohoto snímku mohou změnit. Kvůli těmto změnám může mít rámec označující stejné detekované auto na dvou různých snímcích mírně lišící se proporce či být nepatrně posunutý. Pakliže se na stejném parkovacím místě auta na různých snímcích vyměnila. Můžou být tyto změny detekovaného rámce ještě výraznější. Je proto potřeba implementovat mechanismus, který při porovnávání aut na stejných parkovacích místech nejprve určí, že se skutečně jedná o stejné parkovací místo bez ohledu na absolutní souřadnice detekovaného rámce.

Párování parkovacích míst

Po zpracování obrázku Celery *workerem* v kontejneru `celery` a následným předáním detekovaných rámců a *embedding* vektorů označujících detekovaná auta zpátky do Flask aplikace, přichází na řadu párování parkovacích míst.

Ačkoliv na absolutní souřadnice detekovaných rámců se při párování stejných parkovacích míst nelze spolehnout, pořád platí, že pokud na dvou snímcích bylo detekováno auto na stejném parkovacím místě, měly by být souřadnice detekovaného rámce blízké k souřadnicím detekovaného rámce v původním snímku.

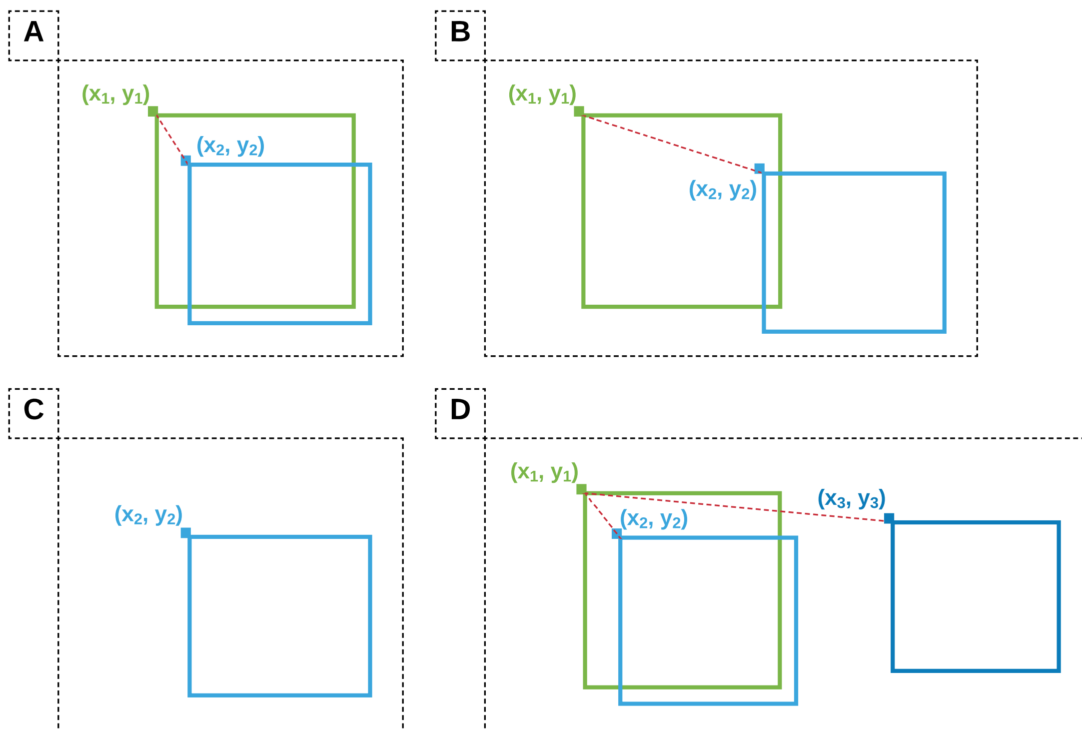
Prvním krokem pro párování parkovacích míst je tedy získání veškerých parkovacích míst, detekovaných na předchozích snímcích a jednoduché nalezení nejbližšího existujícího parkovacího místa, pro každý detekovaný rámec z nového snímku. Fotografie je pouze dvourozměrná plocha. Proto je hledání nejbližšího parkovacího místa realizováno jen na základě euklidovské vzdálenosti počátečních souřadnic parkovacích míst a jednotlivých nově detekovaných rámců. Po spárování rámců může nastat několik situací:

- Nově detekovaný rámec, spárovaný na základě minimální vzdálenosti skutečně odpovídá spárovanému parkovacímu místu. Toto je znázorněno v příkladě **A** na obrázku 7.4. Rámce se navzájem z veliké části překrývají.

- Nově detekovaný rámec, spárovaný na základě minimální vzdálenosti neodpovídá stejnému parkovacímu místu. Na obrázku 7.4 se jedná o příklad **B**. Rámce sice byly spárovány na základě minimální vzdálenosti, překrývají se však málo, nebo vůbec. Tato situace vede k vytvoření nového parkovacího místa na místě, kde byl detekovaný nový rámec. Toto nově vytvořené místo však nebude platné, dokud nebude potvrzeno druhou validní detekcí na dalším snímku.

- Na novém snímku byl detekován rámec, ale v databázi žádné parkovací místa dosud nejsou. Nový rámec tedy nemůže být s ničím spárovaný a porovnaný. Příklad **C** na obrázku 7.4 je příklad tohoto stavu.

- Na novém snímku bylo detekováno více rámců, než kolik bylo v databázi uloženo parkovacích míst. To tedy znamená, že více rámců z nového snímku bude podle vzdálenosti spárováno se stejným parkovacím místem z databáze. Tato situace je znázorněna v příkladě **D** na obrázku 7.4.



Obrázek 7.4: Různé stavy při párování parkovacích míst podle nejmenší vzdálenosti detekovaných rámců.

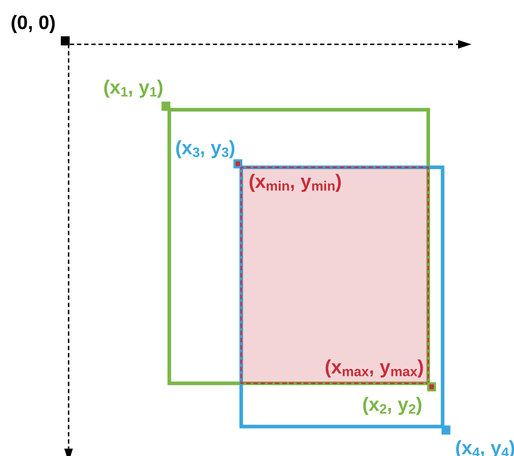
Jednoznačné určení parkovacího místa a porovnání aut

Jak je zřejmé z obrázku 7.4, samotné spárování detekovaných rámců podle jejich vzájemné nejmenší vzdálenosti, nestačí pro jednoznačné určení, že se jedná o stejné parkovací místo a měly by se na něm tedy porovnávat zaparkovaná auta. U popisu jednotlivých situací však už byl naznačen způsob, jak by bylo možné jednoznačně určit, zda se jedná o stejné parkovací místo pomocí překrývající se plochy spárovaných rámců. Pakliže bude společná plocha obou rámců dostatečně veliká, můžeme tato místa prohlásit za stejná.

Každý rámec označující parkovací místo je zadán levým horním a pravým dolním bodem obdélníku $((x_{left}, y_{top}), (x_{right}, y_{bottom}))$, kde (x, y) jsou souřadnice těchto bodů. Na obrázku 7.5 jsou vyobrazeny dva vzájemně se překrývající rámce. Pro zjištění překrývající se plochy tedy stačí určit body (x_{min}, y_{min}) a (x_{max}, y_{max}). Výpočet těchto souřadnic je uveden v rovnici 7.2.

$$\begin{aligned} x_{min} &= max(x_1, x_3) \\ y_{min} &= max(y_1, y_3) \\ x_{max} &= min(x_2, x_4) \\ y_{max} &= min(y_2, y_4) \end{aligned} \tag{7.2}$$

$$intersect_area = (x_{max} - x_{min}) * (y_{max} - y_{min}) \tag{7.3}$$



Obrázek 7.5: Zjištění plochy překryvu dvou detekovaných rámců.

Vypočítání plochy překryvu je pak uvedeno v rovnici 7.3. Nakonec je ještě potřeba porovnat tuto plochu s plochou sjednocení dvou porovnávaných rámců. Pakliže je poměr plochy překryvu dvou porovnávaných rámců a jejich plochy sjednocení větší než nějaký předem zvolený práh Θ_{area}, pak můžeme prohlásit, že se jedná o stejné parkovací místo.

$$\frac{intersect_area}{union_area} > \Theta_{area} \tag{7.4}$$

Prevence nežádoucích detekcí

Kamery, které snímají parkoviště nemusí mít vždy ideální polohu. Některé parkovací místa můžou být zabraná pouze z části, některé parkovací místa můžou být od kamery poměrně vzdálená a jevit se tak na snímku jako velice malá. V takových případech není práce detektoru naprosto spolehlivá a může vyústit ve falešné detekce na místech, kde žádné parkovací místo není, nebo se tam sice parkovací místo nachází, ale není detekováno spolehlivě pokaždé, když na něm stojí auto.

Systém pro detekci a počítání aut na parkovišti implementuje dva mechanismy, jak předejít, nebo alespoň omezit vznikání falešných parkovacích míst. Prvním mechanismem je validace parkovacího místa pomocí příznaku `valid`, který se ukládá ke každému parkovacímu místu v databázi. Každé nově vytvořené parkovací místo má tento příznak v základu nastavený na negativní hodnotu, čímž je toto parkovací místo označeno jako nevalidní a při agregaci statistických údajů se nebere v úvahu. Validním se parkovací místo stane ve chvíli, když je na stejném parkovacím místě minimálně na dvou po sobě následujících snímcích detekováno auto. Tímto způsobem se odfiltrují všechny detekce pouze projíždějících aut, nebo falešné detekce, které občas můžou náhodně vzniknout.

Druhým mechanismem pro čištění parkovacích míst od falešných detekcí je možnost uživatelského mazaní parkovacích míst. Nejedná se však o prosté mazání záznamů z databáze. Pokud by uživatel falešnou detekci odstranil a systém by danou detekci pouze smazal z da-

tabáze, nezachovala by se žádná informace o tom, že na místě, kde falešná detekce původně byla se už žádné nové detekce vytvářet nemají.

Z tohoto důvodu má každé parkovací místo, které je v databázi uloženo, příznak `removed`. Při operaci "mazání" parkovacího místa se tedy parkovací místo z databáze fyzicky nesmaže, pouze se mu nastaví tento příznak. Nové detekce jsou tedy párovány i s parkovacími místy, která byla předem označená jako smazaná. Pokud je nová detekce spárovaná se smazaným parkovacím místem, jednoduše se ignoruje a do databáze se neukládá.

Tímto způsobem je možné práci systému pro detekci a počítání aut na parkovišti v průběhu času uživatelsky zpřesňovat. Postupným mazáním špatně detekovaných parkovacích míst se zamezí novým detekcím na těchto místech, které by s největší pravděpodobností byly opět falešné.

7.4 Zpracování detekovaných aut

Po úspěšném detekování aut na snímku, výpočtu jejich *embedding* vektorů a následném spárování detekovaných rámců s existujícími parkovacími místy, už nic nebrání ve vyhodnocení všech extrahovaných informací a následném uložení nových detekcí do databáze. Opět existuje několik scénářů, které se během vyhodnocování mohou objevit a ovlivnit tak průběh vyhodnocování. Každá detekce je vyhodnocena jako jeden z následujících stavů.

- **Nové auto přijelo na prázdné parkovací místo** - Pokud na spárovaném parkovacím místě na předchozím snímku nebylo detekováno žádné auto, je právě zpracovávaná detekce označená jako auto, které nově přijelo na parkoviště.

    1. Do tabulky `car` v databázi je vložen nový záznam označující nové auto na parkovišti. U nového záznamu je navíc zaznamenán `start_timestamp`, který představuje čas příjezdu auta.
    2. Záznam pro zpracovávanou detekci je pomocí cizího klíče spojen se záznamem nově vytvořeného auta.

- **Auto se na parkovacím místě nezměnilo** - Pakliže bylo na spárovaném parkovacím místě detekováno nějaké auto i na předchozím snímku a navíc porovnání *embedding* vektorů minulé a současné detekce spadá do předem zvoleného intervalu, pak je aktuálně zpracovávaná detekce vyhodnocená jako stejné auto, které na místě parkuje i nadále.

    1. Záznam pro zpracovávanou detekci je pomocí cizího klíče spojen se záznamem auta v tabulce `car`, který patřil i detekci na předchozím snímku.
    2. Pokud bylo spárované parkovací místo dosud označeno jako nevalidní, je mu nastaven atribut `valid` na kladnou hodnotu, čímž dané místo validuje.

- **Auto se na parkovacím místě změnilo** - Tato situace se podobá předchozímu scénáři, avšak s tím rozdílem, že při porovnání *embedding* vektorů minulé a současné detekce výsledek nespadá do definovaného intervalu značící podobnost aut. V takovém případě systém vyhodnotí situaci jako nahrazení jednoho auta na parkovacím místě jiným.

    1. Do tabulky `car` v databázi je vložen nový záznam označující nové auto na parkovišti. U nového záznamu je navíc zaznamenán `start_timestamp`, který představuje čas příjezdu auta.

2. Záznam pro zpracovávanou detekci je pomocí cizího klíče spojen se záznamem nově vytvořeného auta.
3. Záznamu auta v tabulce `car`, který je spojený s předchozí detekcí na stejném parkovacím místě, je nastavena hodnota atributu `end_timestamp`, která označuje čas odjezdu daného auta.

- **Auto z parkovacího místa odjelo** - Posledním scénářem, který během zpracovávání detekce může nastat je situace, kdy se na parkovacím místě na předchozím snímku nacházelo auto, avšak na stejném místě nyní žádné auto detekováno nebylo. Tato situace je vyhodnocena jako auto, které z parkoviště odjelo.

  1. Záznamu auta v tabulce `car`, který je spojený s předchozí detekcí na stejném parkovacím místě, je nastavena hodnota atributu `end_timestamp`, která označuje čas odjezdu daného auta.

Všechny databázově změny provedené během zpracovávání všech detekcí jednoho snímku se dějí v rámci jediné transakce. Na konci zpracovávání těchto detekcí je transakce dokončena a změny jsou najednou uloženy do databáze. Tímto je proces zpracovávání snímku a dat z něj vyextrahovaných úspěšně dokončen.

Kapitola 8

Agregace statistických dat

Zpracování jednotlivých snímku parkoviště při jejich přijetí do systému vytváří poměrně veliké množství dat, která se postupem času v databázi hromadí. Už při zběžném pohledu do databáze je tedy zřejmé, že pro to, aby bylo možné z těchto nashromážděných dat vyvodit nějaké závěry, či zkrátka vydolovat nějaké užitečné údaje, je potřeba data nejprve správným způsobem agregovat.

V kapitole 6.5 bylo popsáno, které statistiky se pro parkoviště a parkovací místa zobrazují ve webové aplikaci. Jednalo se o celkový počet aut, detekovaných na celém parkovišti nebo na konkrétním parkovacím místě, Průměrnou dobu stání auta na parkovišti nebo konkrétním parkovacím místě a v neposlední řadě průměrnou obsazenost parkoviště a parkovacího místa. Každý z těchto tří typů údajů je však ze systému možné získat ve dvou variantách i když některé z těchto variant v aplikaci zobrazené nejsou.

- **Kumulativní** varianta pro daný údaj označuje jednu hodnotu tohoto údaje, za celý časový úsek, který je vymezený pomocí časových filtrů. V případě celkového počtu aut je tedy kumulativní varianta součet všech unikátních aut detekovaných za celé časové období vymezené pomocí filtrů. V případě průměrné doby stání je to celkový průměr během celého časového úseku. A pro průměrnou obsazenost se pak opět jedná o průměr za celou dobu odpovídající filtrům.

- **Kroková** varianta pro daný údaj označuje uspořádaný seznam hodnot daného údaje pro každý krok v čase přes celý časový rozsah odpovídající filtrům. Velikost časového kroku může být od jedné minuty, až po jeden rok. Pro celkový počet unikátních aut a krok o velikosti jedné hodiny se pak bude jednat o seznam počtu detekovaných aut v rámci jedné hodiny pro každou hodinu celého časového rozsahu. Obdobně pak pro průměrnou dobu stání a průměrnou zaplněnost bude pro krok o velikosti jedné hodiny, kroková varianta obsahovat seznam průměrných hodnot vypočítaných jako průměr z každé hodiny zvlášť.

Každý ze tří zmíněných typů údajů je dostupný v kumulativní i krokové variantě a to jak pro celé parkoviště, tak pro každé parkovací místo zvlášť. Ačkoliv je tedy základní výpočet pro všechny varianty jednoho typu údaje velice podobný nebo totožný, jejich implementace a agregace v databázi se liší a to zejména kvůli optimalizaci. Ze stejného důvodu byla co největší část výpočtu implementovaná přímo pomocí databázového dotazu, aby se předešlo náročným výpočtům ve Flask aplikaci, která požadavky na statistické údaje zpracovává.

8.1 Statistiky pro celé parkoviště

Již v kapitole 6.3 bylo zmíněno, že vhodně zvolená struktura databázového modelu, může značně zefektivnit nejen ukládání dat do databáze, ale také práci s těmito daty. Databázový model navržený v kapitole 6.3 má takovou strukturu, aby bylo jednoduše možné agregovat právě dříve zmíněnou trojici statistických údajů a to jak pro celé parkoviště, tak pro jednotlivá parkovací místa.

Prvním krokem je získání identifikátorů všech parkovacích míst, která budou započítávaná do statistik. Základem jsou všechna parkovací místa, která se nachází na parkovišti a pro která jsou aktuálně statistiky počítané. V kapitole 7 je však uvedeno, že v systému existují mechanismy, které pomáhají při prevenci zkreslení statistik falešnými detekcemi parkovacích míst. Tyto mechanismy využívají atributů `valid` a `removed` v databázové tabulce `parking_spot` a je proto potřeba je zohlednit i při výběru parkovacích míst. Poslední vlastností parkovacího místa, která se při výběru zohledňuje je přítomnost parkovacího místa v nastavené oblasti zájmu parkoviště. Databázový systém PostgreSQL podporuje operátory, pomocí nichž je tato kontrola přímočará. Operátor `@@` nejprve vypočítá střed obdélníku, kterým je ohraničeno parkovací místo a vrátí tedy jeden bod. Operátor `@>` poté kontroluje, zda se tento bod nachází v polygonu oblasti zájmu. Celý SQL dotaz, vybírající parkovací místa, která se použijí pro výpočet statistik, je uveden ve zdrojovém kódu 8.1.

```sql
SELECT
    parking_spot.id AS parking_spot_id
FROM
    parking_spot
    JOIN parking_lot ON parking_lot.id = parking_spot.parking_lot_id
WHERE
    parking_lot.id = '{parking_lot_id}'
    AND parking_spot.removed IS FALSE
    AND parking_spot.valid IS TRUE
    AND (parking_lot.roi @> (@@ parking_spot.box));
```

Zdrojový kód 8.1: SQL dotaz pro výběr identifikátorů všech parkovacích míst, jejichž data budou započítána do statistik. Parkovací místa se vybírají vždy pro konkrétní parkoviště určené identifikátorem `parking_lot_id`.

Po úspěšném vyfiltrování parkovacích míst pak už nic nebrání agregování statistických údajů. SQL dotaz, který v databázi agreguje statistické údaje pro celé parkoviště je uveden ve zdrojovém kódu 8.2. Pro získání dat pouze z požadovaného časového rozsahu jsou v klauzuli `WHERE` ještě podle data pořízení snímku vyfiltrovány jen ty záznamy, jenž do zadaného časového rozsahu patří a následně pouze ty parkovací místa, která byla vyfiltrovaná v předchozím dotazu.

Celkový počet aut na parkovišti odpovídající zadanému časovému rozsahu je možné zjistit z pomocí uvedeného dotazu přímo. V databázové tabulce `car` jsou uloženy identifikátory všech aut, které byly na parkovišti detekovány. Spočítání aut je tedy pouze otázka použití agregační funkce `count(DISTINCT car.id)`. Obdobně je to i s průměrnou dobou

stání aut na parkovišti. V databázové tabulce `car` jsou zároveň uloženy i časy příjezdu a odjezdu konkrétního auta. Jedná se o atributy `start_timestamp` a `end_timestamp`. Dobu stání jednoho konkrétního auta je tedy možné vypočítat jako `end_timestamp - start_timestamp` a pro získání průměrné hodnoty všech aut lze pak opět použít SQL agregační funkci `avg(car.end_timestamp - car.start_timestamp)`. Ve výsledném dotazu je navíc použita funkce `coalesce`, která použije čas pořízení posledního zpracovaného snímku v případě, že je atribut `car.end_timestamp` prázdný.

Pro výpočet hodnoty průměrné obsazenosti parkoviště je však situace trochu odlišná. Tato hodnota není vypočítaná přímo v databázovém dotazu. Místo toho je z databáze vybraný celkový počet zpracovaných snímků, pomocí funkce `count(DISTINCT record.id)`, označený jako *records_count* a následně, pomocí funkce `count(DISTINCT detection.id)`, celkový počet detekcí, označený jako *cumulative_cars_count*. Průměrná obsazenost parkoviště je poté vypočítaná podle vzorce 8.1, kde je v proměnné *total_spots* celkový počet validních parkovacích míst na parkovišti.

$$average_occupancy = \frac{cumulative_cars_count}{total_spots \cdot records_count} \tag{8.1}$$

```
SELECT
    count(DISTINCT record.id) AS records_count,
    count(DISTINCT detection.id) AS cumulative_cars_count,
    count(DISTINCT car.id) AS distinct_cars_count,
    avg(coalesce(car.end_timestamp, '{last_timestamp}')
        - car.start_timestamp) AS average_stay_time
FROM
    parking_lot
    JOIN record ON parking_lot.id = record.parking_lot_id
    JOIN detection ON record.id = detection.record_id
    JOIN parking_spot ON parking_spot.id = detection.parking_spot_id
    JOIN car ON car.id = detection.car_id
WHERE
    parking_spot.id IN '{parking_spot_ids}'
    AND record.created >= '{start_timestamp}'
    AND record.created <= '{end_timestamp}';
```

Zdrojový kód 8.2: SQL dotaz pro získání statistických údajů pro celé parkoviště. Do statistik jsou započítány pouze záznamy, které časově spadají do zadaného rozsahu a které se týkají předem vybraných parkovacích míst.

Dotaz pro získání krokových statistických údajů pro celé parkoviště je téměř totožný. V klauzuli `SELECT` se však vybírá nová hodnota, určující časovou známku kroku. Za použití funkce `date_trunc('{scale}' record.created)` se zprava vynuluje část časové známky podle zadaného kroku označeného jako `scale`. Kupříkladu pro krok jeden den tak budou

v časové známce vynulované minuty a vteřiny. Všechny záznamy z jednoho dne tak budou mít stejnou časovou známku, bez ohledu na to, kdy přesně v ten den byly pořízeny. Tato vlastnost je pak využita i v klauzuli `GROUP BY`, kde se podle stejné funkce a ořezané časové známky všechny záznamy shluknou podle zadaného kroku a všechny agregační funkce budou tedy počítat statistiky pro daný krok.

8.2 Statistiky pro jednotlivá parkovací místa

Databázový dotaz pro výpočet kumulativních statistických údajů pro každé parkovací místo zvlášť je opět velice podobný dotazu pro výpočet kumulativních dat pro celé parkoviště. V klauzuli `SELECT` je pouze navíc identifikátor parkovacího místa a v klauzuli `GROUP BY` se podle tohoto identifikátoru záznamy seskupují a agregační funkce tedy počítají statistiky v rámci tohoto seskupení.

Výpočet průměrné obsazenosti konkrétního parkovacího místa se však liší od výpočtu průměrné obsazenosti celého parkoviště. Pro tento výpočet už není třeba celkový počet validních parkovacích míst. Stále se však používá hodnota *records_count*, označující celkový počet zpracovaných snímků parkoviště, získaná z původního dotazu pro kumulativní statistiky celého parkoviště. Naproti tomu použitá hodnota *cumulative_cars_count* se nyní vztahuje pouze k počtu detekcí pro konkrétní parkovací místo. Průměrná obsazenost parkovacího místa je pak vypočítaná podle vzorce 8.2.

$$average_occupancy_{spot} = \frac{cumulative_cars_count_{spot}}{records_count} \tag{8.2}$$

Krokové statistiky pro jednotlivá parkovací místa je možné získat pomocí databázového dotazu, který je opět téměř totožný s dotazem pro získání kumulativních statistik parkovacích míst. Stejně jako v případě krokových statistik pro celé parkoviště, se i v tomto případě pouze do klauzulí `SELECT` a `GROUP BY` přidá časová známka zpracovaná pomocí funkce `date_trunc('{scale}', record.created)`. Výsledné záznamy jsou tedy seskupeny podle parkovacích míst a zároveň podle zadaného časového kroku.

Kapitola 9

Návrhy na vylepšení

I když je systém navržený a implementování v rámci této práce funkční a samostatně použitelný, vždy je možné najít oblasti, které by bylo možné ještě vylepšit nebo nové funkce, které by bylo možné implementovat. Následující seznam návrhů potenciálních nových funkcí systému by mohly vylepšit jeho celkovou použitelnost v praxi a zpříjemnit uživatelskou práci s ním.

Slučování parkovacích míst

V kapitole 6.5 je ukázka možnosti uživatelského mazání falešně detekovaných parkovacích míst. Tato postupná uživatelská sanitace sbíraných dat je jedním z mechanizmů systému, jak udržovat vypočítané statistiky správné a pro očividně falešné detekce, například uprostřed silnice, funguje výborně. V některých případech však systém nemusí detekovat parkovací místo, které je vyloženě falešné a v takových případech může smazání tohoto parkovací místa přinést nové problémy.

Typickým příkladem takové situace může být duplicitní detekce stejného parkovacího místa. Tomuto problému se systém snaží předejít jak na úrovni detekce pomocí algoritmu *Non-maximum Suppression*, tak na úrovni párování parkovacích míst. I tak se však může stát, že systém tuto duplicitní detekci nezachytí. Tato situace nastává hlavně na okrajových částech snímku, kde jsou detekované rámce malé, nebo jsou auta na snímku zachycená kupříkladu jen z půlky. Problém následně spočívá v tom, že obě detekce jsou v podstatě validní parkovací místo. Odstranění jedné z nich by pak mohlo způsobit, že některé budoucí detekce nebudou pro toto parkovací místo správně započítané.

Řešením tohoto problému by mohlo být slučování duplicitních parkovacích míst dohromady. Podobně jako má uživatel možnost parkovací místa vybírat a označovat jako smazaná, měl by možnost manuálně vybrat duplicitní parkovací místa a nechat systém sloučit je dohromady. Rámec takto sloučeného parkovacího místa by mohl být vypočítán jako obdélník, který těsně obepíná všechny slučované detekce.

Více kamer pro jedno parkoviště

Druhým možným vylepšením systému, které by mohlo vylepšit jeho použitelnost v praxi je podpora více kamer na jednom parkovišti. V současné době je systém navržen tak, že snímky parkoviště jsou na REST API posílány s identifikátorem konkrétního parkoviště a jsou s tímto parkovištěm tedy pevně spjaty. Ve skutečnosti však může jedno parkoviště snímat několik kamer současně. Každá z jiného úhlu, nebo jinou část daného parkoviště.

I v současné verzi systému je samozřejmě možné zpracovávat snímky z těchto různých kamer. Pro každou kameru by však bylo potřeba založit zvlášť pseudo-parkoviště, ke kterému by byly navázané. Pro každé pseudo-parkoviště by se pak počítaly statistiky zvlášť. Pomocí nastavení oblasti zájmu u jednotlivých pseudo-parkovišť by bylo možné alespoň zabránit duplicitnímu počítání aut na parkovacích místech, které zabírá několik kamer současně a statistiky pak pro všechny pseudo-parkoviště ručně sečíst. Tento způsob však není příliš uživatelsky přívětivý.

Místo toho by bylo možné do databáze systému přidat prostředníka mezi parkovištěm a zpracovávanými snímky. Tento prostředník by byl navázaný na parkoviště a hrál by roli konkrétní kamery na parkovišti. Snímky odesílané ke zpracování by se pak odkazovaly na konkrétní kameru, místo parkoviště. Samotné zpracování snímku by se v zásadě nezměnilo. Samozřejmě že by se však k párování parkovacích míst musely použít jen parkovací místa detekovaná stejnou kamerou. Změna by však nastala při výpočtu statistik. Ty by se pořád počítaly vzhledem k jednomu parkovišti, nyní by se však zpracovala nasbíraná data ze všech kamer. Za předpokladu, že by pro každou kameru byla nastavena oblast zájmu tak, aby se parkovací místa nezapočítávala několika-násobně kvůli překrývajícímu se záběru kamer, bylo by takto možné vypočítat statistiky pro celé parkoviště současně.

Změnu struktury databáze a možnost nastavení více kamer na jednom parkovišti by následně musela doprovázet také úprava uživatelského rozhraní.

Predikce naplněnosti parkoviště a detekce anomálií

Dlouhodobé zpracovávání a ukládání sekvenčních dat je výborný základ pro predikci budoucích dat pro stejnou úlohu. Taková data jsou i dlouhodobé statistiky o naplněnosti parkoviště. Možnost predikce naplněnosti parkoviště na základě minulých dat by přitom mohla být velice užitečná. Při zjištění dlouhotrvajících trendů ve zvyšování naplněnosti parkoviště je možné například začít plánovat jeho rozšíření. Pro případné dočasné uzávěrky parkoviště je naopak možné díky dlouhodobým predikcím naplněnosti vybrat vhodné datum, ve kterém se nepředpokládá veliká naplněnost. V neposlední řadě sebou predikce naplněnosti nese i možnost detekce případných anomalit.

Složitost implementace takové funkce do značné míry závisí na konkrétních požadavcích na přesnost predikce a detekce anomalit. Nejjednodušší varianta může být jednoduše deterministické vypočítání budoucí hodnoty na základě hodnoty předchozí. Složitější varianta pak může brát v úvahu několik hodnot dohromady a k výpočtu použít nějaký komplexnější regresní algoritmus. Za zvážení by stálo i použití rekurentních neuronových sítí.

I přesto, že tento návrh není nic, co by zpřesnilo samotné zpracovávání snímků, detekci aut, ani sbírání statistických údajů o parkovišti a trochu se tedy vymyká původnímu směru práce, byla by tato nová funkce užitečným nástrojem v systému a opět by vylepšila jeho použitelnost v praxi.

Kapitola 10

Závěr

Ve městech existuje spousta parkovišť. Většina těchto parkovišť má společné to, že skrývají potenciálně veliké množství statistických dat, která by mohla být využita pro cokoliv od plánování uzávěrek dopravy, až po efektivnější řízení městské infrastruktury. Tato data ale častokrát nikdo nesbírá, protože neexistuje mnoho systémů, které by byly schopné udržovat si přehled o naplněnosti parkoviště bez použití drahých závor, nebo jiných zařízení podobného typu. Cílem této diplomové práce bylo prozkoumat možnosti počítání unikátních aut na parkovišti, pouze za použití obrazových dat, a navrhnout a implementovat konkrétní metodu, která tak bude činit.

Práce čistě s obrazovými daty skýtá dva hlavní problémy. Prvním je spolehlivá detekce aut na snímku parkoviště. Dostupných metod na detekci objektů na obrázku, využívajících hluboké učení, je celá řada. Nakonec byl však pro svou rychlost a spolehlivost vybrán model *YOLOv4* [4], který byl následně trénován na datasetu CARPK [9] a po natrénování dosáhl při detekci parkujících aut mAP 91.9%.

Druhým problémem je porovnávání detekovaných aut s předchozími detekcemi na stejném parkovacím místě, aby nebylo jedno auto, které na parkovišti stojí dlouho, započítáno do statistik znovu při každém zpracování nového snímku v sekvenci. Řešením, které je pro tento problém použito v navržené metodě je výpočet *embedding* vektorů, které kompaktně popisují klíčové charakteristiky vzhledu jednotlivých aut. Pro výpočet těchto vektorů z obrázků aut byla navržena a implementovaná konvoluční neuronová sít, inspirovaná sítí FaceNet [18], která řeší obdobnou úlohu porovnání lidských obličejů. Podobně jako model FaceNet, využívá i model navržený v této diplomové práci ztrátovou funkci TripletLoss, ale v rámci této práce byla metoda trénovaná na obrázcích aut z datasetu PKLot [2]. Po konečném nastavení rozhodovacího prahu pro porovnávání, je možné rozhodovat, zda se na parkovacím místě auto změnilo, či nikoliv, s úspěšností až 98.6%.

K tomu, aby bylo možné nově detekovaná auta porovnávat s předchozími detekcemi je třeba mít tyto detekce k dispozici i zpětně. Dalším krokem byl tedy návrh databázového modelu, pro ukládání všech relevantních dat vznikajících při zpracování snímků.

Pro snadnou komunikaci se systémem bylo navrženo REST API, na které je možné nejen odesílat obrázky ke zpracování, ale získávat z něj také statistické údaje agregované z dat uložených v databázi. Komunikace pomocí REST API však není příliš uživatelsky přívětivá. Vzhledem k tomu, že jeden z cílů této práce bylo systém navrhnout tak, aby byl v praxi co nejpoužitelnější, bylo pro něj vytvořeno také grafické uživatelské rozhraní ve formě webové aplikace, ve kterém je možné pohodlně spravovat dostupná parkoviště a především přímo vyhodnocovat všechny statistické udaje, které je systém schopen nabídnout.

Výsledná aplikace tedy zaobaluje veškerou komplexní funkcionalitu systému do jednoduše použitelného a uživatelsky přívětivého balíčku, který lze navíc nasadit téměř okamžitě. I přesto je zde však prostor pro budoucí vylepšení. Možnost zpracovávat obrázky z různých kamer na stejném parkovišti tak, aby byly výsledné statistiky agregovány pro všechny kamery najednou, nebo predikce naplněnosti parkoviště a detekce anomalit by dále rozšířily už tak široké možnosti systému a opět by tak zvýšily jeho použitelnost v praxi.

Literatura

[1] *SB Admin 2 - Free Bootstrap Admin Theme.* Dostupné z: https://startbootstrap.com/theme/sb-admin-2.

[2] Almeida, P. R. L. de, Oliveira, L. S., Britto, J., Silva, J. a Koerich, A. L. PKLot – A robust dataset for parking lot classification. *Expert systems with applications.* 2015, sv. 42, č. 11, s. 4937–4949. ISSN 0957-4174.

[3] Bochkovskiy, A. *Yolo v4, v3 and v2 for Windows and Linux* [https://github.com/AlexeyAB/darknet]. GitHub.

[4] Bochkovskiy, A., Wang, C.-Y. a Liao, H.-Y. M. YOLOv4: Optimal speed and accuracy of object detection. *ArXiv [cs.CV].* 2020. Dostupné z: http://arxiv.org/abs/2004.10934.

[5] Chollet, F. *Deep learning with python.* Manning Publications, 2017. ISBN 9781617294433.

[6] Fielding, R. T. *Architectural styles and the design of network-based software architectures.* 2000. Publication. University of California, Irvine. Dostupné z: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[7] Girshick, R. Fast R-CNN. In: *2015 IEEE International Conference on Computer Vision (ICCV).* IEEE, 2015. ISBN 9781467383912.

[8] Girshick, R., Donahue, J., Darrell, T. a Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition.* IEEE, 2014. ISBN 9781479951185.

[9] Hsieh, M.-R., Lin, Y.-L. a Hsu, W. H. Drone-based Object Counting by Spatially Regularized Regional Proposal Networks. In: IEEE. *The IEEE International Conference on Computer Vision (ICCV).* 2017.

[10] Jung, A. *Imgaug* [https://github.com/aleju/imgaug]. GitHub.

[11] Koch, G., Zemel, R. a Salakhutdinov, R. Siamese Neural Networks for One-shot Image Recognition. 2015. Dostupné z: https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf.

[12] Kumar, R., Weill, E., Aghdasi, F. a Sriram, P. Vehicle re-identification: An efficient baseline using triplet embedding. *ArXiv [cs.CV].* 2019. Dostupné z: http://arxiv.org/abs/1901.01015.

[13] LeCun, Y. a Cortes, C. MNIST handwritten digit database. 2010. Dostupné z: http://yann.lecun.com/exdb/mnist/.

[14] Redmon, J. *Darknet: Open Source Neural Networks in C* [http://pjreddie.com/darknet/]. 2013–2016.

[15] Redmon, J., Divvala, S., Girshick, R. a Farhadi, A. You only look once: Unified, real-time object detection. *ArXiv [cs.CV]*. 2015. Dostupné z: http://arxiv.org/abs/1506.02640.

[16] Ren, S., He, K., Girshick, R. a Sun, J. Faster R-CNN: Towards real-time object detection with region proposal networks. *IEEE transactions on pattern analysis and machine intelligence*. 2017. ISSN 0162-8828.

[17] Saha, S. *A comprehensive guide to convolutional neural networks — the ELI5 way*. Dec 2018. Dostupné z: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[18] Schroff, F., Kalenichenko, D. a Philbin, J. *FaceNet: A unified embedding for face recognition and clustering*. 2015. Dostupné z: http://arxiv.org/abs/1503.03832.

Příloha A

První spuštění aplikace

Systém pro detekci a počítání unikátních aut na parkovišti využívá ke svému provozu virtuální prostředí Docker kontejnerů. Spuštění systému by tedy mělo být možné kdekoliv, kde je nainstalovaný Docker a Docker Compose.

Veškerá konfigurace virtuálních prostředí je pro jednotlivé kontejnery uložena souborech `Dockerfile` a pro celou aplikaci pak v souboru `docker-compose.yml`, který je přiložený ke zdrojovým souborům aplikace. Před prvním spuštěním kontejnerů je potřeba kontejnery vytvořit:

```
docker-compose -f <path/to/docker-compose.yml> up --build
```

Příkazu je potřeba jen zadat cestu ke správnému `docker-compose.yml` souboru a všechny kontejnery budou vytvořeny a následně i spuštěny. Pokud jsou už kontejnery vytvořeny, je pro pouhé spuštění kontejnerů možné z příkazu vynechat přepínač `--build`.

Po prvním vytvoření kontejnerů se vytvoří nejenom kontejnery samotné, ale také *volumes*, ve kterých si Docker ukládá například perzistentní data z databáze. Vytvořena je také samotná databáze s následujícími údaji:

```
USER:     pktracker
PASSWORD: password
```

Databázové tabulky však v databázi ještě vytvořené nejsou. Je proto potřeba se pomocí nástroje na správu Docker kontejnerů připojit do kontejneru `flask` a spustit skript pro vytvoření databázových tabulek:

```
python /app/create_db.py
```

Po vytvoření databázových tabulek je už aplikace připravená na používání a je dostupná na adrese 127.0.0.1:80. Před odesláním prvních zpracovaných snímků je třeba ještě vytvořit v systému nějaké parkoviště.

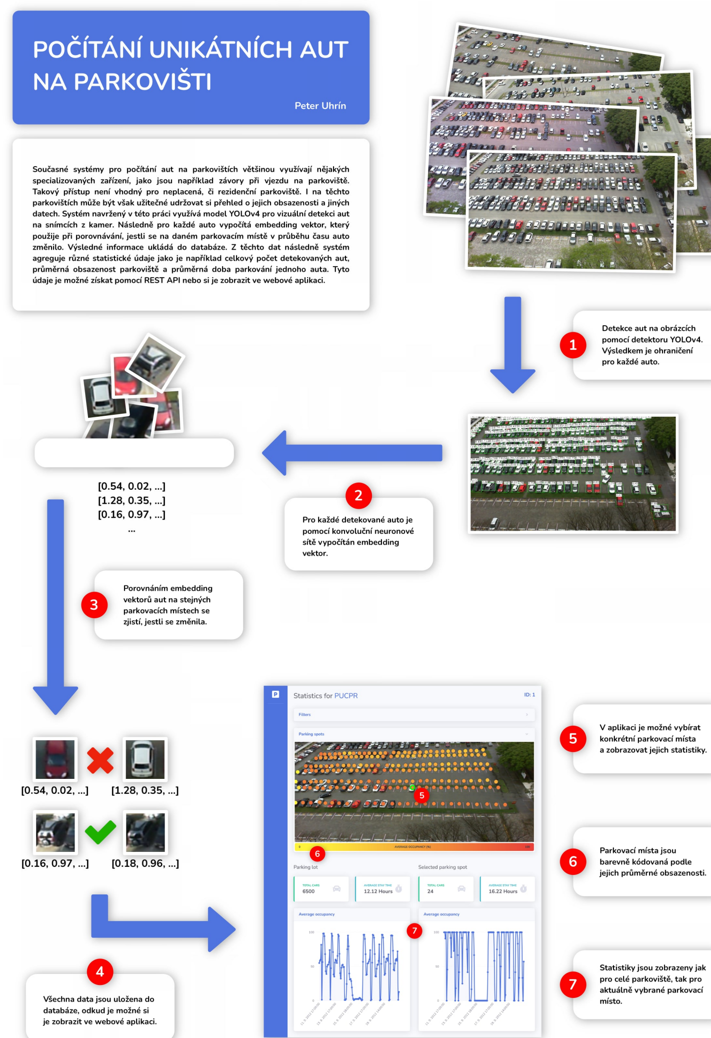
K posílání testovacích snímků parkoviště je možné využít skript `send_images.py`. Tento skript má vlastní *requirements.txt*, aby bylo možné připravit pro něj prostředí na hostitelském stroji. Samotné odeslání obrázků je možné realizovat pomocí příkazu:

```
python send_images.py --src=<path/to/images> --parking_lot_id=<id>
```

Jména obrázků v odkazované složce musí být pojmenované podle časové známky jejich pořízení, ve formátu `"%Y-%m-%d_%H_%M_%S"` a mít příponu `*.jpg`, nebo `*.png`.

Příloha B

Prezentační plakát



Obrázek B.1: Plakát pro prezentaci práce. Zamýšlená velikost je formát A2.