Sem vložte zadání Vaší práce.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Detekce a sledování vozidel z video záznamu pro analytické účely

Bc. Petr Pilař

Katedra aplikované matematiky
Vedoucí práce: Ing. Lukáš Brchl

3. srpna 2020

Poděkování

V první řadě chci poděkovat vedoucímu Ing. Lukáši Brchlovi za rady a praktické postřehy při konzultacích a při tvorbě diplomové práce. Následně děkuji všem, kteří mě po dobu psaní této práce podporovali a motivovali.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 3. srpna 2020 . . . . . .

Odkaz na tuto práci

Pilař, Petr. *Detekce a sledování vozidel z video záznamu pro analytické účely.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Aplikací detekce a sledování vozidel se v posledních letech stále více využívá ke sledování dopravních situací v zabydlených oblastech i dálnicích, plánování provozu po komunikacích i k různorodému civilnímu využití. Spojením záznamů z kamer a moderních algoritmů pro detekci a sledování objektů můžeme agregovat statistiky o počtu vozidel v určitém úseku, průměrné rychlosti, typu, barvě, značce, trajektorií pohybu a spoustu dalšího.

Tato práce se zabývá problematikou detekce a sledování vozidel v běžném provozu z video záznamů. V první části se zaměřuje především na analýzu algoritmů, struktur, modelů a existujících řešení této problematiky a popisuje jednotlivé druhy neuronových sítí využitých v pokročilých modelech. Na základě analýzy implementuje v druhé části práce řešení detekce a sledování vozidel pomocí modelu DETR, který je absolutní novinka mezi modely pro detekci a klasifikaci objektů, která využívá model hlubokého učení Transformer. K trénování a evaluaci používám v implementaci dataset VisDrone.

Klíčová slova Počítačové vidění, neuronové sítě, supervizované učení, detekce objektů, předzpracování dat, sledování objektů, DETR, Transformers, SORT, Centroid Tracker, VisDrone dataset, detekce vozidel z video záznamu, Python

Abstract

In recent years, the applications of vehicle detection and tracking have been increasingly used for monitoring traffic situations in populated areas and motorways, planning traffic on the roads and for various civil uses. Combining recordings from surveillance cameras and modern algorithms for detection and tracking objects, we can aggregate statistics on the number of vehicles in each area, average speed, type, colour, brand, trajectories and more.

This work deals with the issue of detection and monitoring of vehicles in normal operation from the video recordings. The first part focuses mainly on the analysis of algorithms, structures, models and existing solutions to the given topic; describes different types of neural networks used in advanced models. Based on the analysis, in the second part of the work it implements a solution for vehicle detection and tracking using the DETR model, which is an absolute novelty among models for object detection and classification, which uses the Transformer deep learning model. In the implementation for training and evaluation, the VisDrone dataset was used.

Keywords Computer vision, neural networks, supervised learning, object detection, data preprocessing, object tracking, DETR, transformers, SORT, Centroid Tracker, VisDrone dataset, vehicle detection from video recording, Python

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Počítačové vidění přináší nový vhled do zpracovávání informací. Díky algoritmům a modelům pro detekování objektů je možné určovat pozici daných objektů na obrázcích či video záznamech. Detekované objektu je možné třídit do různých tříd a takto sledovat. Pro sledování se používá další skupina algoritmů, které asociují objekty napříč jednotlivými snímky videa. To má v dnešní době obrovské využití ve všech možných odvětvích i civilním sektoru. Mezi ty nejzásadnější patří detekce a sledování vozidel, neboť umožňuje získávat informace a statistiky z určitých oblastí v určitém čase. K tomu se nejčastěji využívá záznamů z běžných kamer, ať už jsou to záznamy z pouličních kamer, kamer na dálnicích, dronů, které se nehýbou, či například z mobilních zařízení.

V této práci si nejdříve v kapitole 2 popíšeme, co je to strojové učení a z jakých klíčových komponent se skládá. Následně v kapitole 3 projdeme neuronové sítě a složitější modely, na kterých jsou založené komplexnější modely. V této kapitole věnujeme pozornost také Transformerům, které využívá DETR. Tyto neuronové sítě následně využívají komplexnější modely právě pro detekci a sledování objektů, jak si popíšeme v kapitole 4.

Mezi zmíněnými modely, které v práci budeme popisovat, například patří SSD, R-CNN, Fast R-CNN, Faster R-CNN, tři verze YOLO, DETR a další. Podobným způsobem se práce věnuje i algoritmům pro sledování objektů. V práci se v kapitole 5 budeme následně zabývat analýzou existujících řešení pro tuto problematiku. Tyto řešení si popíšeme a zhodnotíme. Následně práce v kratší kapitole 6 popíše jednoduché i složitější evaluační metriky, které modely používají.Po analytické části si zvolíme, jaké modely a algoritmy použijeme k implementaci vlastní aplikace, která bude detekovat, sledovat a vytvářet statistiky o vozidlech. To, jakým způsobem budeme postupovat při implementaci, si vysvětlíme v kapitole implementace 7, kde detailně rozebíráme zvolené řešení a kód samotný. V kapitole s experimenty 8 provedeme několik experimentů a pozorování, která nám napoví, jak používat parametry detektorů a trackerů. Kapitola s experimenty obsahuje i jejich výsledky.

Kapitola 1

Cíl práce

Cílem rešeršní části diplomové práce je analýza existujících řešení pro problémy zabývající se detekcí a sledováním různých objektů z video záznamů, s čímž se pojí i detekce v jednotlivých obrázcích. Zaměřuje se na objekty vozidel zachycené běžnou pouliční kamerou. Analyzuje jednotlivé neuronové sítě a jejich komponenty využité v komplexnějších modelech. Dále prozkoumává typy a řešení výstupních statistik ze sledování objektů.

Cílem praktické části je implementace kódu aplikace, která bude detekovat objekty vozidel z video záznamů s využitím algoritmů počítačového vidění, které jsou popsány v rešeršní části práce. Detekované objekty bude aplikace vhodným algoritmem sledovat a odvozovat o nich výstupní statistiky. Implementaci je nutné provést v programovacím jazyce Python.

Kapitola 2

Strojové učení

Pro vysvětlení pojmu strojového učení si nejdříve představíme normální program. Tento program má daný vstup, který určeným způsobem zpracuje a jako výsledek zpracování vrátí nějaký výstup. Pro stejné vstupy a vstupní parametry vždy vrátí stejné výstupy. Algoritmy strojového učení nám umožňují sestavit program, který využívá svých předchozích zkušeností, výstupů či chyb tak, aby získal lepší vnitřní představu o tom, jak zadaný problém lépe vyřešit. Oproti normálnímu programu, který musí vždy vědět, co má přesně dělat. Strojové učení propojuje statistiku a informatiku. V průběhu učení si program postupně buduje vnitřní interpretaci statistických vzorců ze vstupních dat. Zásadní rozdíl je, že tyto vzorce, podle kterých určuje výsledek, si program vymýšlí sám a nejsou předem definované programátorem. Dá se říci, že čím více vstupních dat máme, tím lépe se může program naučit.

2.1 Klíčové komponenty

Strojové učení se neobejde bez vstupních dat, definice modelu, jeho ztrátové funkce a optimalizačního algoritmu [1]. Následně si popíšeme tyto komponenty v bodech:

Dataset

Data jsou typicky reprezentována kolekcí příkladů v počítačově lehce zpracovatelném formátu. Nejčastěji v numerické či textové podobě. Jeden příklad z kolekce se skládá z jednotlivých vlastností, které ho popisují. U učení s učitelem je u každého příkladu uvedený ještě jeho cílový výsledek. Jedna vlastnost napříč různými příklady ze stejné datové sady se může lišit velikostí a pro optimální výsledek je třeba k tomu přizpůsobit model. Všechna data by měla být v jednotném formátu a počet jejich vlastností stejný, aby se dali jednoduše převést na vektory se stejnou dimenzí. Proto je v některých případech nutné nejdříve data předzpracovat.

Data je vhodné rozdělit na trénovací, validační a testovací část. Na trénovací části, jak z názvu napovídá, se bude vybraný model, se specifickými hyper-parametry, trénovat (učit). Na validační části se následně ověřuje, jak správně model predikuje výsledek. Také nám pomůže vybrat selekci nejlepších hyper-parametrů, nebo celých modelů. Testovací část se používá k predikci výkonu modelu na reálných datech. Trénovací část musí být z principu větší než validační a testovací. Pokud máme dostatečné množství dat, doporučuje se rozdělit 50 % pro trénovací část a 25 % pro obě validační a testovací části, jak je uvedeno ve slidu 10 v [2].

Pokud nemáme dostatečné množství dat, doporučuje se náhodně rozdělit data na několik stejně velkých částí. Jednu část (první) zvolíme za validační a ostatní za trénovací a trénujeme model. Po natrénovaní tento postup se stejnými částmi opakujeme, avšak jako validační část zvolíme nějakou jinou, doposud nepoužitou, dokud nenatrénujeme model postupně na všech. Vždy s výjimkou jedné části uvádí slide 21 v [2].

Model

Model strojového učení je struktura určující posloupnost výpočetních mechanik a proměnných. Tato struktura je určená k postupnému zpracování vstupních dat až k předem určenému počtu výstupů, které mohou být jiného typu. Model představuje soubor, který si ukládá tuto strukturu, její proměnné a konstanty. Při trénování modelu dochází k rozpoznávání skrytých vzorců a upravují se proměnné jeho struktury. Naučený model se dá použít k určování výsledků i dříve neznámých vstupních dat. Při učení modelu se využívá ztrátové funkce.

Ztrátová funkce

Funkce, která dokáže změřit, jak dobrý (nebo špatný) je zvolený model. Mapuje tedy predikci výsledku z modelu a cílový výsledek na skutečné číslo. Typicky se ji snažíme při optimalizaci minimalizovat. Je rozdíl, pokud porovnáváme výsledek modelu na trénovacích, validačních či testovacích datech. Proto uvádíme trénovací chybu a validační či testovací chybu.

Křížová entropie (anglicky cross entropy) se určuje pro dvě odlišné pravděpodobnostní rozdělení. Měří výkon klasifikačního modelu, jehož výstupem je jedno pravděpodobnostní rozdělení mezi 0 a 1. Výsledek ztrátové funkce s užitím křížové entropie se postupně zvětšuje, čím více se hodnoty odlišují. Pro ideální model dává hodnotu 0. Pro binární klasifikaci, kde máme dvě třídy, se křížová entropie počítá jako:

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Ve vzorci je y binární identifikátor toho, zda je třída predikována dobře nebo ne a p je pravděpodobnost konkrétní predikce.

Podobně můžeme sestavit i křížovou entropii pro více tříd. Jedná se o sumu logaritmů pravděpodobností. Protože pravděpodobnost je mezi 0 a 1, tak její logaritmus vrací záporné číslo a tak musíme celou sumu znegovat.

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Optimalizace

Způsob redukce chyby predikce modelu. Optimalizace se snaží najít co nejlepší hyper-parametry modelu pro minimalizování ztrátové funkce. Pro neuronové sítě se nejčastěji používá optimalizační algoritmus gradientního klesání, či nějaká z jeho vylepšených variant (SDG, Adagrad, Adam, AdaMax) [3].

2.2 Učení s učitelem

Ačkoliv se strojové učení dělí na několik typů podle zadání úlohy, typu vstupních dat a toho, jak moc programu s učením pomáháme, tato práce je zaměřená a zabývá se pouze učením s učitelem a jeho dělením, nikoliv učení bez učitele, kde u vstupních dat neznáme cílový výsledek.

Učení s učitelem využívá zpětné vazby vstupních dat, kde pro každý prvek známe jeho cílový výsledek. Program pomocí modelu předpovídá pro vstupní data výsledek a následně v roli učitele kontroluje predikovaný výsledek s výsledkem daným.

Regrese patří k nejjednodušším úkolům učení neuronových sítí s učitelem. Jedná se o predikci skalární číselné hodnoty. Její řešení tedy nalézá odpověď na otázku *Jak moc*. Například určení ceny automobilu, jestliže známe jeho kupní cenu, počet najetých kilometrů a další vlastnosti. Na regresní úlohu se dá namapovat spousta problémů, avšak ne vždy to jde. Chyba se většinou počítá pomocí střední kvadratické chyby:

$$l(y, y') = \sum_i (y_i - y'_i)^2.$$

Při klasifikaci predikujeme pro jeden vstupní příklad s vlastnostmi jednu kategorii z množiny předem definovaných kategorií. Kategoriím se často v anglické literatuře říká labels neboli jmenovka. Model výslednou kategorii určí tak, že pro každou kategorii z množiny určí pravděpodobnost, že právě tato kategorie je výsledek. Pokud máme pouze dvě třídy, hovoříme o binární klasifikaci. Nejznámější příklad klasifikace s více než dvěma třídami je rozpoznávání ručně psaných čísel s více než 30000 záznamy [4]. Zde také zjišťujeme, že ne

všechny chyby jsou si sobě rovné. Druhy chyb a evaluační metriky si rozebereme později.

Další úkol je takzvané tagování. Jedná se o klasifikaci více tříd na jednom vstupu, které nejsou vzájemně se vylučující. Příkladem tagování je rozpoznávání více objektů na obrázku při strojovém vidění. O tom více v kapitole se strojovým viděním. Kromě již zmíněných úloh existují ještě úlohy doporučovacích systémů a řazení. Těmi se však tato práce nezabývá.

Kapitola 3

Neuronové sítě

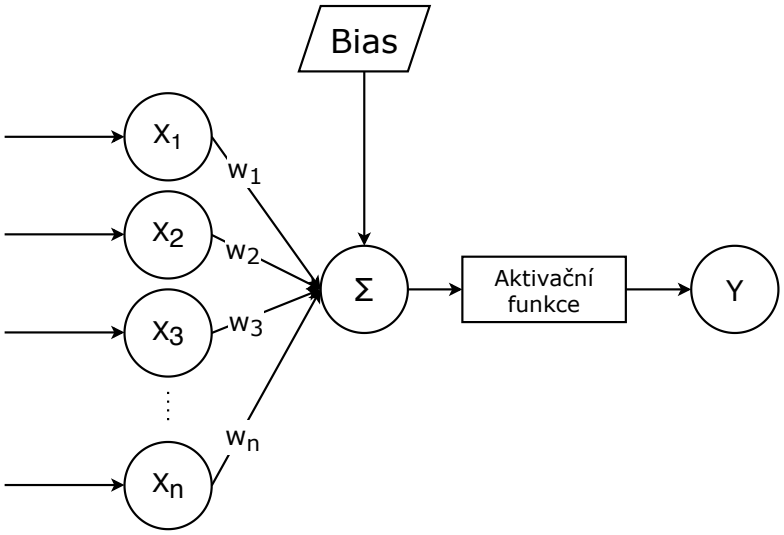
Abychom porozuměli aktuální situaci ve sféře neuronových sítí a mohli vybrat nejlepší způsob, jak vyřešit zadaný problém detekce a sledování vozidel, představíme si typy neuronových sítí. Existuje mnoho typů umělých neuronových sítí, z nichž každá má své silné stránky.

Neuronová síť se dá chápat jako graf s výpočetními uzly a hrany mezi nimi, značícími kudy data putují. Tyto výpočetní uzly jsou uspořádány do vrstev a mohou pracovat paralelně. Uzly se dělí do tří základních kategorií a to na vstupní, skryté a výstupní. Existují však i uzly rekurentní, paměťové a kernelové [5]. Vstupní neurony jsou v první vrstvě neuronové sítě a přijímají vstupní data. U většiny neuronových sítí se počet vstupních neuronů rovná počtu vlastností příkladu neboli dimenzi vstupních dat. Vstup každé vrstvy jsou výstupní data z předchozí úrovně (předchozích úrovní), zpracuje je a opět předá vrstvě za sebou. Na konci neuronové sítě je výstupní vrstva s výstupními neurony. Těch je stejné množství jako počet výstupních parametrů. U jednoduché regresní úlohy klidně i jeden výstupní parametr. U klasifikační úlohy pravděpodobnost pro každou třídu. Uzly mohou být mezi sousedními vrstvami plně propojené, to znamená každý s každým. Při vstupu informace do neuronu si neuron určuje, jak je pro něj důležitá a nastavuje si u něj specifickou váhu. Typicky reálné číslo mezi 0 a 1. Nejvyšší váhu mají hrany, které nejvíce přispívají ke správnému výstupu. Analogicky nejnižší mají ty, co přispívají nejméně. Sítě, kde informace putuje pouze směrem od počátku do konce, se jmenují dopředné sítě.

3.1 Perceptron

Perceptron je nejjednodušší lineární neuronová sít, která propaguje informaci od začátku do konce. Základní perceptron se používá k binární klasifikaci. To však v základním znění lze pouze pokud jsou data lineárně rozdělitelná. Dá se na něm však velice přehledně ukázat, jak funguje neuronová síť, přestože jde pouze o jeden neuron. Jak můžeme vidět v následujícím obrázku, skládá se ze

vstupních hodnot X_i, jejich vah w_i , *bias*, aktivační funkce a sumy. Hodnota sumy součinů X_i a w_i pro všechny vstupy i je sečtena s *biasem* a předána aktivační funkci, která rozhodne, zda propustí výstupní signál.



Obrázek 3.1: Perceptron

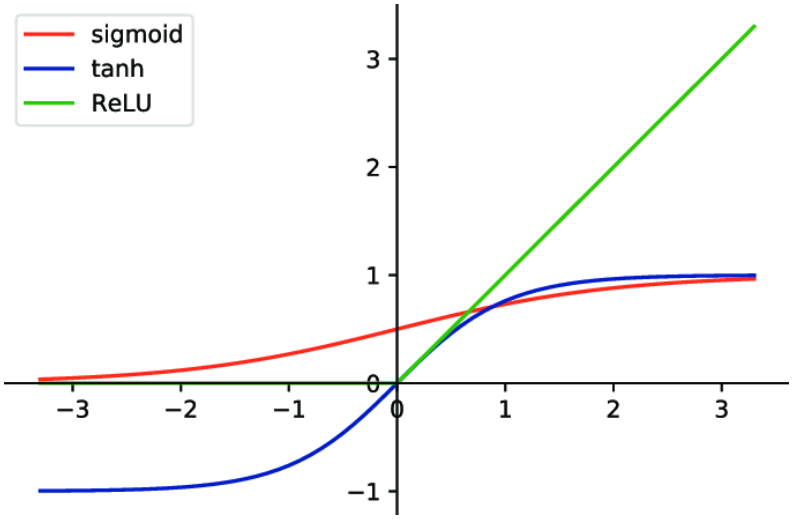
3.2 Vícevrstvý perceptron

Vícevrstvý perceptron má minimálně jednu vstupní, jednu skrytou a jednu výstupní vrstvu. Nejmenší počet jeho vrstev jsou tedy tři, avšak počet skrytých vrstev může být i větší. Protože má skryté vrstvy, bavíme se již o hlubokém učení, které v prací využijeme, a proto je tato část relevantní. Jeho hlavní využití je u klasifikace dat, která nejsou lineárně separabilní. Každá vrstva je plně propojena s vrstvou další, to znamená, že jsou všechny neurony jedné vrstvy propojeny se všemi neurony vrstev vedlejších, stejně jako úplný bipartní graf. Používá se jako základ do složitějších neuronových sítí a je vhodný pro úlohy překladu a rozpoznávání řeči.

3.2.1 Aktivační funkce

Klíčová vlastnost vícevrstvého perceptronu je nelineární aktivační funkce, která je aplikována v každém skrytém neuronu. Aktivační funkce rozhoduje o aktivaci neuronu pomocí sumy vážených vstupů a biasu. Nejvíce používané aktivační funkce jsou funkce ReLU a sigmoid. Funkce ReLU zajišťuje velmi jednoduchou nelineární proměnu. Pro každý skalární vstup je její výstup roven maximu z tohoto vstupu nebo nuly. Když je vstup záporný, derivace funkce ReLU je 0, a v opačném případě, když je vstup kladný, derivace je

1 [6]. Používá se, protože se jednoduše a rychle počítá její derivace v bodě. Její nevýhoda nastává při zpětné propagaci, kdy nedochází k aktualizaci vah a biasů, a tak se z neuronu stane takzvaný mrtvý neuron. Funkce signoid představuje další aktivační funkci široce užívanou, hlavně díky tomu, že je všude diferencovatelná a jednoduše se na ní určují aktivační prahy. Výstup funkce sigmoid je reálné číslo ležící na intervalu od 0 do 1. . Do třetice uvedu funkci Tanh neboli hyperbolický tangent. Má tvar podobný sigmoidu, avšak její výstup se pohybuje mezi hodnotou –1 a 1. Funkce můžeme vizuálně porovnat v obrázku 3.2 ze zdroje [7].



Obrázek 3.2: Relu, sigmoid a tanh

3.3 Hluboké neuronové sítě

Dosud jsme si představili perceptron, jednoduché dopředné neuronové sítě a vícevrstvý perceptron a řešili jsme jednotlivé neurony. Můžeme si však všimnout, že celá neuronová síť a jeden neuron mají podobné vlastnosti – mají dané vstupy a spočítají na ně odpovídající výstupy. Stejně tak při procesu učení se provádí zpětná propagace přes jeden neuron, tak stejnou myšlenkou přes celou síť (až na výjimky). Můžeme tedy na neuronové sítě pohlížet jako na "černé krabičky". Díky tomu jsme schopni abstrakce a porozumět i komplexním modelům složených z více různých neuronových sítí. Aby takový model fungoval, musíme však zajistit validní návaznost jednotlivých výstupních a vstupních vrstev. Například model DETR ResNet101, který se hojně využívá ve strojovém vidění i jiných odvětvích, je 101 vrstev hluboký a tyto vrstvy se skládají z dalších podvrstev.

Hluboké neuronové sítě vyžadují hluboké učení. Hluboké učení může opět

být s učitelem, bez učitele, nebo se semi-učitelem. Hluboké učení dále patří následující architektury: konvoluční, rekurentní, neuronové sítě, hluboké belief sítě a další.

3.4 Konvoluční neuronové sítě

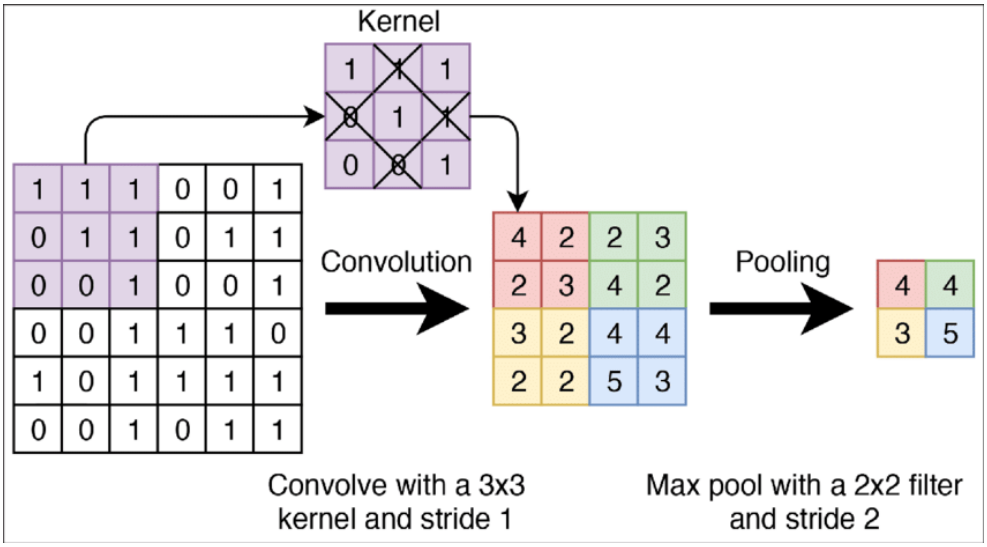
Konvoluční neuronová síť neboli CNN je nejčastěji používána pro zpracovávání vizuálních snímků [8]. Používají se v případech, kdy není výpočetně nebo časově možné zpracovat obrovské množství parametrů, které pro stejný úkol vyžadují husté architektury jako vícevrstvý perceptron.

Konvoluční vrstvy jsou sice inspirovány vícevrstvým perceptronem, ale nemusí být plně propojeny. Tyto vrstvy používají konvoluční operace, díky kterým může být výsledná síť mnohem hlubší a to ještě s mnohem méně parametry. Dá se říci, že se vrstvy na začátku celé sítě zaměřují na lokální regiony bez zájmu o celý obsah ve vzdálených regionech. Postupně se i tyto regiony potkají jako sousedé a mohou vytvořit představu o celém obrázku. Vrstvy na začátku reagují vždy stejně na podobné vlastnosti v lokálních regionech. Díky tomu jsou velice efektivní při úkolech rozpoznávání z obrázků a video záznamů i dalších odvětví. Tyto operace a vrstvu, na které se provádí, budeme dále považovat za totéž. Konvoluční vrstvy jsou většinou v sérii za sebou spolu s pooling, plně propojenými a normalizačními vrstvami.

Matematicky se konvoluční operací myslí křížová korelace. Vstupní vektor má velikost rovnou součinu šířky, výšky a hloubky obrázku. Hloubkou obrázku myslíme barevnou hloubku, tedy počet čísel určujících jeden pixel. Například jeden pixel v jednobarevném (černobílém) obrázku může být definován pouze jedním číslem určujícím intenzitu dané barvy. Pokud má obrázek tři barvy (červenou, zelenou a modrou) má tedy i hloubku tři. Konvoluční vrstva promění vstup do méně dimenzionální mapy vlastností.

Pooling (sdružovací) vrstva zajišťuje redukci dimenze dat. Určitým předem definovaným způsobem, řekněme maximem (max pooling), vytvoří z dlaždic o určité velikosti, většinou 2x2, jen jedno číslo (viz obrázke 3.3). Existuje mnoho způsobů, jak vybrat nebo spočítat tuto hodnotu. Výběr maxima extrahuje důležité vlastnosti, jako jsou hrany, avšak zbytek informace ztrácí v redukci. Spočítání průměrné hodnoty se zdá v některých případech efektivnější, záleží však na vstupních datech [9] ze zdroje [10].

Pokud v konvolučních a pooling vrstvách používáme dlaždice určité délky, je potřeba k tomu mít odpovídající velikost obrázku. Také si můžeme všimnout, že pixely na okraji obrázku nejsou tak využívány jako ostatní, to může mít za následek nižší míru predikce u okrajů obrázku. Řešení je jednoduché a to přidání prázdných pixelů na okraj obrázku o šířce podle velikosti dlaždice. Této metodě se říká padding. Jako velikost dlaždic se často používají malé liché hodnoty 1, 3, 5 nebo 7. To zachovává poměr stran.



Obrázek 3.3: Konvoluční a pooling vrstva

Přestože mají konvoluční sítě mnohem méně parametrů než vícevrstvý perceptron, mohou být stále dražší na výpočet. Je to z důvodu více výpočtů, neboť jeden parametr se může podílet na velkém množství dílčích násobení. Moderní konvoluční neuronové sítě vyžadují méně parametrů než husté architektury a jsou jednodušeji parallelizovatelné pomocí GPU jader.

3.4.1 Plně konvoluční neuronové sítě (FCN)

Plně konvoluční neuronové sítě se využívají v Faster R-CNNs (rychlejší konvoluční neuronové sítě založené na regionech) pro detekci objektů v obrázcích. Dají se také využít pro další úlohy jako je sémantická či panoptická segmentace, které však nespadají do náplně této práce, a tak je řešit nebudeme. Plně konvoluční neuronové sítě využívají klasické konvoluční neuronové sítě k transformaci obrázku pixel po pixelu. Následně však mapují svůj výstup zpět do původní velikosti. K tomu se používá transponovaná konvoluční vrstva, která právě zajistí, že výsledný obrázek bude stejné velikosti (šířky a výšky) jako obrázek na vstupu, přičemž barva pixelu bude odpovídat jeho třídě. Proto se můžou a pravděpodobně budou lišit barvy. Pojďme si popsat jejich architekturu od začátku do konce. Na počátku je normální konvoluční neurovná síť, která přijímá obrázek a extrahuje charakteristiky obrázku. Poté 1 x 1 konvoluční síť transformuje počet kanálů pixelu na počet kategorií. Následně transponovaná konvoluční vrstva přemění šířku a výšku mapy charakteristik do velikostí původního obrázku, a proto je výstupní velikost stejná jako vstupní [11].

Transponovaná konvoluční vrstva dělá přesný opak, než čekáme od klasických konvoluční vrstev, nebo pooling vrstev, které redukují dimenzi. Ano, transponovaná konvoluční vrstva zvětšuje dimenzi vstupní matice. K tomu používá kernel o určité velikosti, který jako klouzavé okénko putuje po vstupní matici a generuje hodnoty do nové větší matice, kde se v jednotlivých buňkách postupně agregují.

3.5 Rekurentní neuronové sítě

Pro lepší pochopení kapitoly o transformerech a mechanismech pozornosti je nutné si vysvětlit, jak fungují rekurentní neuronové sítě. Je to typ neuronové sítě, ve které se vyskytuje nějaká forma paměti. Výstupy konkrétní vrstvy jsou namapovány na další vrstvu, ale zároveň i znovu do té původní. Jinak je model tvořen jako dopředná síť. První a poslední vrstva se tvoří stejně jako normálně a reprezentují vstupy a výstupy, avšak u skrytých vrstev dochází (nemusí u všech) k rekurenci vstupů. Zmíněné rekurentní chování pomáhá predikovat výsledky vrstev. U těchto neuronových sítí je zásadní práce s časovými kroky. Každý uzel funguje jako paměťová buňka s extrémně krátkou pamětí a to přesně jeden krok. Tato paměť se v každém kroku nastaví na novou hodnotu. Od každého časového kroku do dalšího si každý uzel zapamatuje některé informace, které měl v předchozím časovém kroku. Neuronová síť si tedy pamatuje informace, které může potřebovat později.

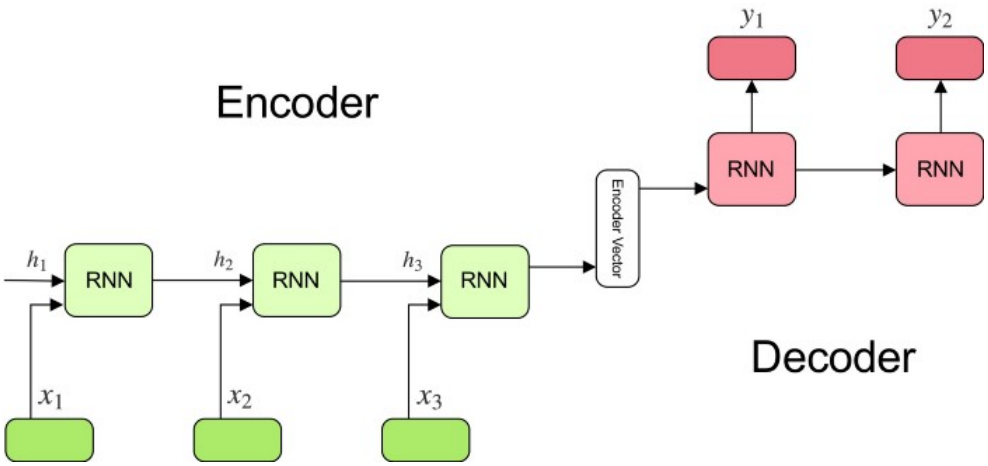
Jejich využití je hlavně při řešení problémů, které jsou závislé na sekvenci po sobě jdoucích stavů. Trénují se pomocí zpětné propagace skrz čas - takzvané BPTT. To vyžaduje, abychom rozšířili rekurentní neuronovou síť o jednu časovou periodu, tedy abychom získali závislosti mezi proměnnými modelu a parametry. Poté na základě řetězového pravidla použijeme backpropagation pro výpočet a uložení gradientů. Skryté vrstvy RNN mohou zachycovat historické informace o sekvenci určité délky, dané velikostí modelu, až do aktuálního času. Počet parametrů modelu se nezvyšuje se zvyšujícím se počtem časových intervalů.

3.5.1 Enkodér-dekodér

Populární architektura pro predikci textových sekvencí například v úlohách překladu jazyků. Na první pohled nemusí být zřejmé, že i tento druh architektury neuronových sítí spadá do rozsahu této práce. Potřebujeme ovšem vědět, jak funguje a co od něho čekat, protože se ho, i když v poupravené verzi, využívá v detekci objektů pomocí transformerů, kterou v práci využíváme k implementaci.

Skládá se ze dvou hlavních částí a to česky enkodéru a dekodéru (viz 3.4 z [12]). Enkodér se stará o zakódování vstupní sekvence do stavu určeným několika tensory. Síť enkodéru se skládá z několika rekurentních vrstev, kde každá přijímá jednu část vstupní sekvence, udržuje o ní v paměti informaci

a propaguje ji dále do dalších vrstev. Stav je poté předán pomocí kódového vektoru do dekodéru. Kódový vektor je poslední část enkodéru a snaží se zapouzdřit informaci od všech vstupních prvků [13]. Dekodér je síť s obvykle stejnou architekturou jako enkodér. Vezme kódový vektor a predikuje co nejlepší možnou sekvenci, aby měla stejný význam jako vstupní sekvence. Existují však i typy enkodérů/dekodérů, které mají architekturu konvoluční neuronové sítě, nebo jiné.



Obrázek 3.4: Enkodér a dekodér

3.6 Transformeři

Transformeři v současnosti, tedy v roce 2020, patří mezi nejlepší modely pro úlohy zpracování jazyka. Jak si však později ukážeme, dají se využít i v jiných odvětvích a to třeba právě v detekci a rozpoznávání objektů z obrázků. Modely založené na transformerech (BERT, GPT,..) se vyskytují na předních příčkách různých soutěží. Tato architektura byla představena v roce 2017 v práci "Attention Is All You Need" a od té doby se staly velice populární [14]. Abychom porozuměli transformerům, musíme nejdříve rozumět mechanismům pozornosti (attention mechanism).

3.6.1 Mechanismy pozornosti

V podkapitole enkodér-dekodér jsme si vysvětlili, jak zakódujeme vstupní sekvenci v rekurentních vrstvách a předáme ji dekodéru, aby predikoval cílovou sekvenci. Pozornost nám pomáhá využívat vztahy různých částí vstupní sekvence, a tak lépe predikovat sekvenci výstupní. Pozornost vezme dvě sekvence (například v různém jazyku), promění je v matici, kde slova jedné sekvence

tvoří sloupce a slova druhé sekvence tvoří řádky. Následně vytváří shody, identifikující relevantní kontext. To je velmi užitečné při strojovém překladu. Pokud pozornost bere dvě sekvence, které jsou stejné, říkáme jí self-attention. Například ve větě *"Martin jel do školy na červeném kole."* víme, že přídavné jméno *"červeném"* se vztahuje k podstatnému jménu *"kolu"* a ne k ničemu jinému z věty. Jádrem mechanismu je jedna vrstva pozornosti. Má paměť ve tvaru párů klíč-hodnota. Pomocí lineární algebry (stejně jako u všech neuronových sítí) můžeme počítat velké množství výpočtů najednou a díky tomu je v tomto případě možné vypočítat vztahy mezi každým slovem sekvencí.

Vstupy se označují jako dotazy. Pro získání výstupu se používá ohodnocovací funkce, která měří podobnost dotazu s klíči v paměti, tedy pro každou část vstupního dotazu máme nějaké ohodnocení. Následně se použije funkce softmax, abychom získali z ohodnocení pozornostní váhy mezi 0 a 1. Konečný výstup je vážená suma pozornostních vah, podle hodnot z dvojic v paměti. Používají se různé ohodnocovací funkce - skalární součin, vícehlavá funkce [14] a vícevrstvý perceptron. Pozornost se skalárním součinem předpokládá, že dotaz má stejnou dimenzi jako klíče v paměti. Dále počítá ohodnocení jako skalární součin mezi dotazem a klíčem. Pro omezení nevyžádaných vztahů se výsledek dělí druhou odmocninou dimenze, neboť při násobení hodnot se můžou data extrémně rychle zvětšovat. K pozornostní vrstvě se přidává i vrstva výpadku (dropout) [14]. Tato vrstva zařídí náhodné vyhození částí vrstvy, a tak se pracuje s menší více regularizovanou vrstvou a ta nemůže spoléhat na jeden konkrétní vstup, protože ho občas nedostane.

Vícehlavá ohodnocovací funkce se používá pro self-attention. Vstup ve formě slov z dotazu, klíčů z paměti a hodnot z paměti se vloží do tří rozdílných plně propojených lineárních vrstev, které z každé části vstupu vytvoří vektory. Vektory z dané kategorie tvoří matici. Matice slov dotazu a klíčů se dále maticově vynásobí a získá se matice ohodnocení. Ta udává, jaké vztahy mezi sebou mají slova ze vstupní sekvence. Toto se děje pro každou jednotku času, tím je myšleno, když vkládáme do modelu slova vstupní sekvence postupně. Stejně jako u ohodnocovací funkce se skalárním součinem se tento součin, zde ale v matici, dělá druhou odmocninou dimenze. Dochází k tomu ze stejných důvodů. Nyní tedy máme přeškolovanou matici ohodnocení. Na matici aplikujeme **softmax**. Vysoké prvky matice budou upřednostňovány a menší potlačovány. Dále tuto matici s pozornostními vahami vynásobíme maticí hodnot z paměti a získáme matici výstupu. Samotný výstup ve formě vektoru nám vydá poslední lineární vrstva. Trik vícehlavé pozornosti je v tom, že 3 kategorie vstupu ve formě vektorů rozdělíme na menší vektory stejné velikosti. Proces probíhá úplně totožně, ale malé vektory vstupují postupně. Každému takovému procesu se říká hlava. Protože jsme vstupní vektory rozdělili, musíme na konec přidat ještě jednu vrstvu, konkrétně na předposlední místo před poslední lineární. Tato přidaná vrstva zajišťuje zřetězení výstupních vektorů, abychom měli stejný výstup jako v předchozím případu. Každá hlava má možnost naučit se něco jiného, a tak toto řešení dává modelu více informace

o reprezentaci sekvence.

3.6.2 Moduly transformerů a jejich sítě

Nyní si můžeme vysvětlit mechaniky fungování transformerů krok po kroku. Transformer se skládá ze dvou hlavních modulů enkodéru a dekodéru. První hlavní modul, který se v obrázku vyskytuje na levé straně, se nazývá enkodér. Ovšem jedná se o složitější modul než dříve zmíněný enkodér, a tak v této práci budeme tomuto modulu říkat první hlavní modul. Obdobně s druhým hlavním modulem, kterému se říká dekodér, ovšem je mnohem složitější. Ten se vyskytuje v pravé části.

Nejdříve si popíšeme první hlavní modul. Počínaje tím, že vložíme vstup do embedding vrstvy, která každé vstupní slovo ze sekvence přetvoří na vektor čísel. Poté musíme přidat ke zmíněnému vektoru i zakódovanou pozici daného slova. Je to jednoduší než vytvářet rekurentní neurony pro všechny vstupy. K zakódování pozice slova se používá funkce založená na sinu nebo cosinu. Jsou vybrané schválně, protože mají lineární části, které se může model dobře naučit. Pro každý lichý krok se používá funkce s cosinem a pro každý sudý se sinem. To úspěšně zařídí, že se vektory vstupů rozšíří o jejich vstupní pozici. Nyní vektory vstupují do vrstvy enkodéru. Ten má za úkol namapovat všechna slova vstupní sekvence do abstraktní kontinuální reprezentace, která si udržuje naučenou informaci pro celou sekvenci. První hlavní modul má dva podmoduly: vícehlavou pozornostní síť, za níž je dopředná plně propojená neuronová síť. Nezávisle kolem obou podmodulů existují residuální propojení napojené na dvě vrstvy normalizace, kde každá je za jedním podmodulem. Další podmodul v pořadí podle směru zpracování dat je vícehlavá pozornostní síť, která počítá pozornostní váhy vstupů a produkuje výstupní vektory s kódovanou informací o vztahu slov mezi sebou. V další normalizační vrstvě se výstup vícehlavé pozornostní sítě sčítá se vstupem předaným pomocí residuálního propojení. Následuje modul s dopřednou sítí, ve které jsou lineární vrstvy a ReLu aktivace mezi nimi. Výstup tohoto modulu se opět v normalizační vrstvě sčítá se vstupem. Již několikrát se v různých částech transformeru používá normalizační vrstva a residuální propojení. Normalizace napomáhá stabilizaci a redukuje čas trénování. Residuální připojení vstupu pomáhá síti trénovat, protože gradientní zpětná propagace může celou sítí jít přímo. Celý proces prvního hlavního modelu se dá zopakovat několikrát, aby byla informace jasnější.

Druhý hlavní modul se zabývá generováním výstupních sekvencí slov. Vzpomeňme, že tato slova nemusí být pouze textového charakteru. Může se jednat například o výstup vlastností z konvoluční vrstvy, která zpracovala nějaký obrazový vstup. Tuto konkrétní mechaniku si vysvětlíme podrobněji dále. Nyní však zpět k popisu druhého hlavního modulu transformeru. Má podobné podmoduly jako první hlavní modul. Můžeme si všimnout, že má vlastně dva vstupy. První vstup se napojuje do druhé vícehlavé pozornostní

sítě uprostřed celého modulu. Druhý vstup znovu zpracovává předchozí výstup z celého transformeru. Druhá hlavní část je tedy autoregresní. Podle směru od druhého vstupu po celkový výstup obsahuje nejdříve posunutí předchozí výstupní (nyní vstupní) sekvence doprava. To zajistí, aby model předpovídal další slovo v pořadí. Stejně jako v prvním hlavním modulu používá embedding a zakódované pozice, aby získal informaci o pozici jednotlivých slov. Vektory z embedding vrstvy obohacené o pozici vcházení do první vícehlavé pozornostní sítě. Ta však pracuje trochu jinak, než očekáváme. Protože je tento hlavní modul autoregresivní a generuje sekvenci slovo po slově, je nutné zabránit podmiňování od slov budoucích, které ještě nezná. Pro vysvětlení uveďme příklad. Při jednoduché sekvenci *" [začátek] jak se máš"* má druhý hlavní modul například tuto vnitřní reprezentaci viz 3.1.

	[začátek]	jak	se	máš *
[začátek]	0.6	0.1	0.1	0.1
jak	0.2	0.7	0.2	0.1
se *	0.1	0.3	0.6	0.1 *
máš	0.1	0.3	0.2	0.5

Tabulka 3.1: Ukázka možné vnitřiní reprezenatce modulu

Pokud se zaměříme na slovo *"se"*, nemělo by znát pozornost ke slovu *"máš"*, protože to je budoucí slovo, které bylo vygenerované až po *"se"*. Metoda, jak zabránit počítání s pozorností k budoucím slovům, je klasická maska.

V první vícehlavé pozornostní síti je tedy nutné přidat tuto maskovací vrstvu na místo před aplikací softmaxu. Masku je nutné si vyjádřit dopředu, je ovšem vždy stejná. Reprezentuje jí čtvercová matice o šířce a délce sekvence. Matice je v horním trojúhelníkovém tvaru, kde jsou hodnoty mínus nekonečno. Tato na první pohled zvláštní hodnota zaručuje, že při sečtení s maticí přeškálovaných ohodnocení se odpovídající buňka matice vymaskuje, a tak zbydou jen požadované hodnoty. Následný softmax poté změní hodnoty mínus nekonečna na 0 a pak se model nebude vůbec zaměřovat na maskovaná slova. Výstupem první vícehlavé pozornostní sítě je maskovaný vektor s informací, jak by model měl pracovat s prvním vstupem (z první hlavní části modelu).

Druhá vícehlavá pozornostní vrstva má, jak bylo již řečeno, jako vstup dotaz a v paměti dvojice klíč-hodnota. Dotaz je dán vektory z prvního vstupu z první hlavní části a jako klíče a hodnoty jsou použity výstupní vektory z první vícehlavé pozornostní sítě. Tento proces porovnává vztahy prvního vstupu v závislosti na výstupu první vícehlavé pozornostní sítě. Následuje dopředná vrstva, která výstup dále zpracuje. Poté lineární vrstva přistupuje ke klasifikátoru. Klasifikátor má velikost danou maximálním počtem tříd při jednom tagování, které máme k dispozici. Většinou se udává i více. Například CoCo dataset má na jednom obrázku maximálně okolo 70 z celkových 91 tříd,

tak se jako velikost klasifikátoru doporučuje číslo 100, protože od 70 nechává ještě dostatečnou rezervu [15]. Následně se na klasifikátor použije softmax, a tak dostaneme pravděpodobnostní ohodnocení pro každou třídu. Při použití na textu reprezentuje počet tříd počet slov ve slovníku. Při použití na obrázcích zase počet různých kategorií objektů v datasetu (osoba, automobil, židle...). Podle indexu třídy s nejvyšší pravděpodobností nalezneme odpovídající pojmenování. Jedná se tak vlastně o klasifikaci, přestože to není na první pohled znát.

Na konci sekvence je značka konce, a tak transformer ví, kdy přestat propagovat svůj výstup opět na druhý vstup.

Vzhledem k počtu podmodulů a vrstev je trénování transformerů obtížné. V roli programátora se o to však nemusíme tolik starat, protože to zařizují příslušné třídy a metody v použitém frameworku. O frameworcích pro implementaci si povíme později. Při trénování druhého hlavního modelu transformeru se snažíme o to, aby co nejlépe předpovídal další krok ze sekvence, kterou již předtím zpracovával.

Kapitola 4

Počítačové vidění

Velmi populární odvětví vědy zabývající se tím, jak počítače vnímají digitální obrázky nebo videa. Z pohledu člověka se jedná o na první pohled velice jednoduchou úlohu, jednoduše vidí, detekuje, sleduje a dělá závěry o objektech a událostech, co má před svýma očima. Z pohledu počítače se ale jedná o velice komplexní úlohy, neboť bez pomoci počítačového vidění nevidí objekty a události, ale pouze hodnoty barev pixelů z kamer, či jiných dat z různých senzorů. Ačkoliv si to nemusíme uvědomovat, tak se s ním v dnešní době setkáváme každý den a do budoucnosti bude jeho užití ještě rapidně stoupat. Technologie strojového vidění nás obklopuje například na letištích, kde sleduje podezřelé osoby a osamocená zavazadla, v chytrých autech, které si samy hlídají okolní silnici nebo ve zdravotnictví, kde kontrolují snímky rentgenů. Dříve se jednoduché algoritmy počítačového vidění dali implementovat i bez hlubokého učení, ale v dnešní době jsou již tyto dvě odvětví téměř neoddělitelné.

V předchozích kapitolách jsme si představili, jak fungují hluboké neuronové sítě různých typů, abychom snáze pochopili oblast počítačového vidění. Nyní si uvedeme, jakým způsobem s vizuálními daty pracovat. Přestavíme si metody rozpoznávání objektů z obrázků.

4.1 Předzpracování obrázků

V našem případě budeme pracovat s daty, které obsahují obrázky a video záznamy z běžných kamer. Přestože se jedná o běžné kamery, není specifikováno, jak velká data mohou být. V roce 2020 má téměř každý člověk chytrý mobilní telefon. Takový telefon je schopen pořizovat snímky a video záznamy o vysokém rozlišení od jednotek až po desítky mega pixelů. Proto je nutné umět pracovat i s daty ve vysokém rozlišení, umět si je přizpůsobil (zmenšit), ale tak, aby docházelo k co nejmenší ztrátě informace. Hluboké neuronové sítě mají přesně daný počet vstupních parametrů, které přebírá vstupní vrstva a dále je propaguje do sítě. Počet vstupní parametrů tedy musí přesně od-

povídat číslu, které dostaneme vynásobením šířky, výšky obrázku a hloubky jednoho jeho pixelu. Z toho plyne, že obrázky je třeba nějakým způsobem upravit do stejného tvaru, abychom je mohli propagovat skrz neuronovou síť. V našem případě budeme tedy většinou obrázky zmenšovat.

Při každém učení neuronových sítí platí, že čím více dat máme, tím můžeme náš model lépe natrénovat. V některých oblastech je těžké získat více relevantních dat. Ale v odvětví počítačového vidění a hlavně rozpoznávání můžeme data přidělat celkem jednoduše. Musíme však zajistit, aby zůstaly kategorie objektů stejné! Také pokud například u detekování objektů máme v anotacích hranice těchto objektů, musíme stejně upravovat i tyto informace. Uvedeme si příklad. Představte si, že chceme vytvořit nějaký malý dataset obrázků koček, tak vezmeme nějaké zařízení pro sběr obrázků (třeba mobilní telefon) a vyrazíme fotit kočky a po hodině máme dataset hotový. Při řešení problému detekce musíme data ještě řádně anotovat, v našem případě pravděpodobně ručně. Model, který bychom na těchto datech natrénovali, však nebude příliš dobrý při testování na nějakých jiných obrázcích koček. Je to, protože námi vytvořený dataset pokrývá jen velmi úzkou část všech možných vizuálních reprezentací koček. Mluvíme o takzvaném latentním prostoru. Mezi nedostatky, co jsme při sběru dat udělali, můžeme například zařadit, že jsme vůbec nezachytili černé kočky, nebo nemáme fotky koček s bleskem, v noci a šeru, při dešti nebo mlze, vyskytující se v různých částech obrázku, malé, velké, na jiném pozadí nebo ve zvláštních pozicích. Velké množství z těchto věcí se můžeme pokusit přigenerovat, nebude to sice tak perfektní jako mít všechna data, ale určitě to při trénování modelu velice pomůže. Jinými slovy se vyhneme přeučení. Generování obrázků s podobnými změnami probíhá úpravami těch stávajících. Protože obrázek je vlastně matice nebo několik matic, jeho úpravy jsou pouze maticové operace, které jsou dobře počítačově zpracovatelné i paralelně [16]. Mezi úpravy patří:

Převrácení obrázku
: Obrázek převracíme většinou horizontálně, ale můžeme i vertikálně, či obojí. Důležité je, že převrácení obrázku nemění kategorie jeho objektů.

Výřez z obrázku
: Dovoluje nám změnit velikost, avšak zachovává poměr stran.

Náhodné výřezy z obrázku
: Náhodně se po vstupním obrázku rozmístí čtverce zvolené velikosti a jejich obsah se vyřízne (vyplní se černou nebo jinou barvou). To přidává do obrázků nečekané artefakty na náhodných pixelech, a tak na ně nemůže model spoléhat. Díky tomu se učí obecněji.

Škálování obrázku
: Mění kompozici a poměr stran. Musíme si však být jisti, že se nemění kategorie objektů. U kočičího datasetu se kategorie nemění. Pokud bychom

třeba rozpoznávali či klasifikovali jednoduché geometrické obrazce focené kolmo, z kruhů by se nám staly elipsy, a to je nežádoucí.

Pootočení obrázku o nějaký úhel
Často se používá, protože je velice účinné. Pokud obrázek pootočíme o malý úhel doprava, nebo doleva, lidskému oku to přijde stále stejné, ovšem reprezentace hodnot pixelů obrázku v počítači/programu se celá změní. Model se tak snažíme učit rozpoznávat lépe vlastnosti obrázku, než se zaměřovat na konkrétní pixely – to ovšem platí pro všechny způsoby předzpracování obrázků.

Přidání šumu
Může být užitečné přidávat do obrázků šum. Většinou se používá náhodný šum, určený nějakým parametrem určujícím, kolik šumu se má na obrázek přidat. Vyberou se náhodné pixely, kterým se přiřadí náhodná barva.

Mezi další úpravy samozřejmě patří úpravy barev. Změna jasu, kontrastu, odstínu barev (HUE), saturace, value a dalších nám pomáhají tvořit nové obrázky z existujících, čímž se rozšíří náš dataset.

Jaké úpravy jsou ale nejlepší a nejvíce zvýší úspěšnost modelu? Jsou lepší geometrické úpravy jako rotace, otáčení a ořezávání, či úpravy barev? Na tyto otázky jsem našel odpovědi ve studii, kterou připlavili Taylor a Nitschke [17]. Výsledky této studie ukazují, že na modelu z konvolučních neuronových sítí při použití datasetu Caltech101 mají lepší výsledky geometrické úpravy. Jak uvádí převzatá tabulka 4.1, generování úpravou ořezáváním mělo až 18,82 % nárůst výkonosti modelu [17].

	Top-1 Accuracy	**Top-5 Accuracy**
Baseline	$48.13 \pm 0.42\%$	$64.50 \pm 0.65\%$
Flipping	$49.73 \pm 1.13\%$	$67.36 \pm 1.38\%$
Rotating	$50.80 \pm 0.63\%$	$69.41 \pm 0.48\%$
Cropping	$\mathbf{61.95 \pm 1.01\%}$	$\mathbf{79.10 \pm 0.80\%}$
Color Jittering	$\mathbf{49.57 \pm 0.53\%}$	$67.18 \pm 0.42\%$
Edge Enhancement	$49.29 \pm 1.16\%$	$66.49 \pm 0.84\%$
Fancy PCA	$49.41 \pm 0.84\%$	$\mathbf{67.54 \pm 1.01\%}$

Obrázek 4.1: Porovnání výkonosti různých úprav

Závěrem této kapitoly je nutné říci, že nejčastěji se používá více úprav najednou, aby se dataset rozšířil o mnoho nových obrázků. Tedy se zvětšil latentní prostor, který pokrývá a ze kterého se náš model může učit a následně predikovat. Výhodou je, že nejvíce používané frameworky dělají námi zvolené úpravy automaticky při přejímání datasetu v programu. Tím pádem nemusíme

ukládat všechny generované obrázky na disk, což šetří místo, ale je to i rychlejší pro program, protože nemusí z disku načítat takové množství dat.

4.2 Ladění modelu

V kapitole s konvolučními neuronovými sítěmi jsme si řekli, že jejich konvoluční vrstvy určují jednoduché textury, hrany, tvary a obecně kompozici obrázku. Předpokládejme, že učíme nějaký model identifikovat například auta, můžeme si být jisti, že k postupné extrakci kompozic obrázku dochází. Pokud potom chceme udělat podobný model, který bude identifikovat například autobusy, můžeme do určité míry využít předchozího modelu a pak ho ladit. Můžeme ho využít, protože oba modely by se stejně museli naučit například identifikaci hran a jiných drobných struktur, které mají oba pozorované objekty (auta a autobusy) shodné. Můžeme takhle využít složitých modelů, které jsou natrénovány na řádově spodních desítkách milionů obrázků, na nichž je dohromady více než 1000 kategorií.

Nyní si ukážeme, jak postupovat při využití jiného před-trénovaného modelu. Nejdříve si musíme položit otázku, zda náš výsledný model bude pracovat s daty, které se zásadně neliší od před-trénovaného modelu, který chceme využít. Samozřejmě, pokud naše data představují nějakou velmi specifickou doménu, například snímky z rentgenů, můžeme si být téměř jisti, že v takové doméně nenajdeme nějaký před-trénovaný model a měli bychom začít stavět model a trénovat od začátku. V případě, že máme malý počet dat, musíme si dát pozor, aby silný před-trénovaný model doladěný o pár posledních vrstev netíhnul k přeučování. Pokud jsme tedy našli vhodný model, vytvoříme novou (vlastní) neuronovou síť, která replikuje architekturu a parametry sítě před-trénovaného modelu. Nyní je čas na smazání poslední vrstvy. Odstraňujeme pouze poslední vrstvu, protože tam dochází k finální klasifikaci/identifikace a zbytek sítě zachycující obecnou strukturu, můžeme ponechat (proto to děláme). Poslední vrstva samozřejmě nesmí chybět, a tak přidáme vlastní. Musíme brát zřetel na to, aby se počet výstupů rovnal počtu kategorií. Parametry této vrstvy určíme náhodně, protože se nám v průběhu učení přeučí. Samotné trénování výsledného modelu se prování na našich datech, nikoliv na datech před-trénovaného modelu. Poslední vrstva se bude trénovat od začátku a zbytek modelu se bude pouze ladit. Pro trénování se doporučuje menší koeficient učení, protože očekáváme, že před-trénované parametry budou dobře a nechceme je příliš měnit. Na stránce [18] se dokonce tvrdí, že máme zakázat změnu parametrů na prvních pár vrstvách z důvodů, které jsme již zmínili dříve.

Před-trénované modely můžeme hledat v takzvaných modelových zoo, které existují pro každý větší framework jako je TensorFlow, Torch, Keras, MxNet, zastaralejší Caffe a další. V těchto zoo se většinou dají stáhnout před-trénované modely, ale i prázdné architektury, které si můžeme natrénovat od

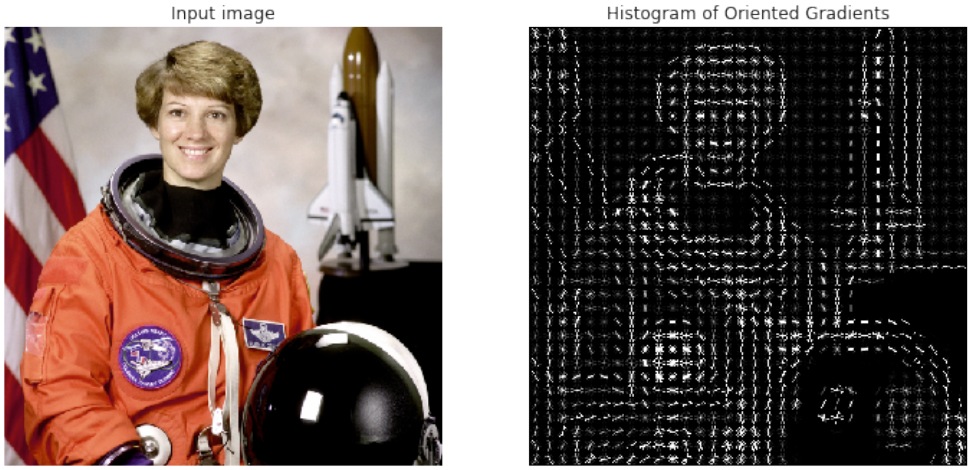
začátku. Modely jsou naučené na obrovských a známých datasetech jako je ImageNet, CIFAR nebo COCO.

4.3 Detekce objektů

Mezi hlavní úlohu počítačového vidění z reálného světa patří detekce objektů. Jedná se o komplexní úlohu, ve které se z obrázků rozpoznává více objektů, u kterých nás zajímá jejich pozice, případně ohraničení a navíc se klasifikují. Detekce objektů se využívá v autonomním řízení, osobních robotech, příslušenství pro lidi s trvalou ztrátou zraku, industriální robotice a mnoha dalších odvětvích.

Staré modely pro detekci objektů byly založené na ručně vyrobených vlastnostech, které se postupně odhalovaly z obrázku pomocí klouzavého okna, nebo dlaždice, jak jsme tento prvek pojmenovali dříve v konvolučních sítích. Pokud byla reakce na výběr okénka pro danou vlastnost dostatečné vysoká, byl detekován objekt. Detailnější vysvětlení nejstarších modelů přeskočíme a podíváme se na metody extrakce struktur obrázků s histogramově orientovanými gradienty (HOG).

Modely používající metody jako je HOG, mají základní architekturu kterou užívjí pro rozpoznávání obrázků, která se skládá z: určení regionů, extrakce příznaků z těchto regionů a podle těchto příznaků následné klasifikace regionů. Tato technika počítá výskyty orientovaných gradientů na husté mřížce uniformě uspořádaných buněk reprezentujících pixely. Hlavní myšlenka spočívá v popisu jednotlivých tříd pomocí histogramů. Nejdříve si pro každý pixel napočítáme dvě čísla. Jedná se o absolutní hodnoty rozdílů pixelů před naším pixelem a po našem pixelu ve směru, pro obrázek přirozených, os x a y. Následně se pomocí eukleidovské normy těchto dvou rozdílů spočítá gradientní vzestup a pomocí $tan^{-}1$ se určí úhel tohoto gradientu. Všechny úhly jsou mezi 0 a 180. Potom, co tyto hodnoty napočítáme pro všechny pixely ve výběru, můžeme z nich udělat histogram. U histogramu se typicky uvádí 9 tříd, protože 9 celočíselně dělí 180. V tomto případě se bude úhel orientovaných histogramů posouvat po 20 stupních. Pro každý výběr máme tedy jeden histogram, do kterého si napočítáme hodnoty gradientního vzestupu podle odpovídajících úhlů. Ačkoliv máme určené hranice úseků v histogramu, přičítáme hodnoty vzestupu váženě. Tedy pokud se úhel gradientu zrovna strefí do hodnoty rozdělující úseky histogramu, jednoduše hodnotu gradientního vzestupu rozpůlíme a do každého úseku přičtete jednu polovinu. Díky histogramu máme 9 čísel, které reprezentují sílu gradientu v daném směru, které normalizujeme. Výstupem metody je vektor popisující strukturu objektů. Nyní už stačí použít klouzavé okénko a kontrolovat, zda dojde ke shodě. Pokud ano, detekujeme hledaný objekt. Hod dobře vystihuje následující obrázek 4.2 ze zdroje [19].



Obrázek 4.2: HOG

4.3.1 Boxy

K určování polohy objektů se v naprosté většině případů používá ohraňující box, anglicky bounding box a zkráceně bbox nebo jen box. Ohraňující box je obdélníkový box, který se určuje pomocí čtyř metrik. Musíme si dát pozor na to, jaké datasety a modely používají jaké formáty.

- Některé uvádí čtveřici jako x1 a y1 souřadnice os pro horní levý roh a x2 a y2 souřadnice bodu pro pravý dolní roh. U tohoto typu se souřadnicím někdy říká minimální x a y a maximální x a y. Toto značení je převzato z kartografie.

- Čtveřice je udána souřadnicí levého horního rohu spolu s šířkou a výškou boxu. Tento formát používá například dataset COCO ve svých anotacích.

- Čtveřice je udána souřadnicemi středu spolu s šířkou a výškou ohraňujícího boxu. To je vhodné v případě, kdy hned po detekci objektu probíhá jeho sledování, které může být založené na sledování pozic centroidů objektů.

U špatně dokumentovaných modelů a datasetů může být obtížné zjistit, jaký formát používají a jak tedy správně namapovat vstupní data. Převádět mezi formáty je jednoduchá matematická operace, takže se může i stát, že před-trénovaný model používající jiný formát ohraňujícího boxu se to přeučí na ten správný a bude fungovat tak jako tak. Je to ovšem přitěžující faktor učení.

4.3.2 Kotvy

Při detekci objektů z obrázků dochází k vytváření velkého množství potencionálních lokací, kde by se mohly vyskytovat hledané objekty. Všechny tyto oblasti se při klasickém způsoby detekce musí procházet. Okraje oblastí se následně upravují tak, aby přesněji predikovali ohraňující box. Pokud možno úplně stejně jako je reálný ohraňující box v anotacích. Zjistit, jaký box nejlépe lokalizuje daný objekt, je těžká úloha, na kterou existuje několik řešení.

Nyní si uvedeme metodu kotev, která generuje více boxů rozdílné velikosti a z nich tvoří finální box. Nejdříve vygenerujeme několik kotev (ohraňujících boxů) s různou velikostí. Tyto velikosti jsou předem dané velikostí vstupního obrázku. Tyto kotvy umístíme na pixely tak, aby na každý pixel mě vždy všechny druhy kotev vyrovnané středem kotvy na sebe. To zajistí, že můžeme detekovat i více objektů, které se překrývají a jejich pomyslný střed na obrázku je ve stejném či hodně blízkém bodě. Pro generování velikostí kotev musíme mít definované dvě množiny: množinu velikostí hran o velikosti n a množinu poměrů stran o velikosti m. Na každý pixel se vždy generují všechny kombinace z těchto množin a každý obrázek bude mít celkem (šířka * výška * n *m) kotev. To je na jeden obrázek obrovské množství a taková je i výpočetní náročnost. Počet kotev se dá zmenšit, když bereme v úvahu pouze podmnožinu ze všech kombinací dvou množin. Zvolíme z množiny velikostí jednu hodnotu a z množiny poměrů také jednu hodnotu. Do výsledné podmnožiny patří pouze takové prvky, které obsahují buď jeden, nebo oba zvolené prvky. Tím jsme omezili počet kotev obrázku na hodnotu (šířka * výška * (n + m – 1)). Při použití kotev můžete vyhodnotit všechny predikce objektů najednou. Kotvy eliminují potřebu skenovat obrázek posuvným oknem, které vypočítává samostatnou predikci v každé potenciální lokaci.

Při učení na trénovacím datasetu určíme pro každou kotvu kategorii cíle, na kterém kotva leží a relativní posun od reálného ohraňujícího boxu. Pohlížíme totiž na každou kotvu jako na malý trénovací objekt. Detekce probíhá tak, že nejdříve vygenerujeme kotvy, kde každá predikuje kategorii a posun. Posuneme kotvy podle hodnot jejich posunů a tím získáme ohraňující boxy, které již mohou být pro finální predikce.

Ukažme si, jakým způsobem převedeme posunuté kotvy na opravdové ohraňující boxy. Označme si kotvy K1, K2 až Kn a opravdové ohraňující boxy B1, B2 až Bm. Musí platit, že kotev je v součtu více než opravdových boxů, to však vzhledem k obrovskému počtu kotev platí téměř vždy. Použitím IOU na všechny kombinace kotev K a opravdových boxů B získáme matici M s rozměry n*m, kde sloupce jsou označeny B1 až Bm a řádky K1 až Kn. Nyní když máme naplněnou matici M, začneme postupně promazávat její prvky, dokud nezůstanou jen osamocené prvky. Promazávání probíhá tak, že nalezneme prvek matice, který má nejvyšší hodnotu. Je na něm kotva, která nejvíce pokrývá nějaký objekt. Tento prvek si v matici označíme a k dané kotvě si bokem přiřadíme pravdivý box. Zaznamenáme si indexy nalezeného

prvku, tedy řádek a sloupec jako i a j. Následně smažeme všechny ostatní prvky v odpovídajícím řádku a sloupci. Tento postup opakujeme, dokud nepromažeme všechny sloupce a nepromazané řádky přeskočíme. Poté postupně projdeme značené prvky matice (nebo ty které jsme si zaznamenávali bokem) a kotvám přiřazujeme odpovídající opravdové boxu, pokud je hodnota IOU prvku matice větší než předem určená hranice.

Nyní můžeme přiřadit kategorie a posuny jednotlivým kotvám. Ty kotvy, které mají přiřazený box, mají kategorii danou podle boxu a obdobně s posunem, který se připisuje relativně k dané kotvě. Protože poměr stran kotvy a odpovídajícího opravdového boxu se může lišit, posun se počítá podle souřadnic středu. Pokud kotvě tímto způsobem nepřiřadíme kategorii, dostane kategorii třídy pozadí [20]. Zmíněná hranice tedy dělí kotvy do dvou kategorií a to na objekty a pozadí. U kotev s hodnotami poblíž rozhodovací hranice tedy není příliš jasné, do které kategorie opravdu patří a pak se většinou udávají hranice dvě. Hranice vysoká nám zaručí, že kotvy, které identifikuje, opravdu obsahují nějaký objekt a hranice nízká určuje maximální hodnotu, od které se již kotva považuje za pozadí. Mezi těmito hranicemi vzniká tedy taková, lidově řečeno, šedá zóna, kde si model není moc jistý s predikcí, a tak ji raději zahazujeme, aby to vypadalo, že model ví, co dělá. Při použití kotev se však šedé zóně nikdy nevyhneme a moderní řešení s ní umí dobře pracovat a hranice se přibližují. Můžeme si všimnout, že malé kotvy uvnitř nějakého detekovaného objektu mají přiřazenou třídu, ale díky malému IOU je považujeme za pozadí, to je v pořádku, protože jejich box nás nezajímá.

To nás přivádí do stavu, kdy máme detekované objekty, avšak každý objekt je detekován více kotvami a mi potřebujeme vybrat jen jednu, nebo je nějak zkombinovat, aby výsledek přesně lokalizoval opravdový ohraničující box. Boxy sice můžeme jednoduše zprůměrovat, ale to bychom přišli o část naučené informace. Existuje často používaná metoda NMS (non-maximum suppression) [21]. Zde si popíšeme, jak metoda NMS funguje. Nejdříve všechny kotvy ohodnotíme pravděpodobností pro každou kategorii, čemuž se také říká ohodnocení spolehlivosti. Seřadíme všechny kotvy podle těchto pravděpodobností. Následuje jádro metody NMS, kde pro každou oblast zájmu s více kotvami vybereme kotvy s nejvyšší hodnotou a ty, které se od ní moc liší, potlačíme. Proto se metoda jmenuje non-maximální potlačení.

Rekapitulace ve zkratce: K vygenerování konečných detekcí objektů se odstraní kotvy (pokrývající celý obrázek), které patří do třídy pozadí, a zbývající se filtrují podle jejich ohodnocení spolehlivosti. Kotvy s největším ohodnocením se vybírají pomocí NMS.

4.3.3 Kotvy a vzorkování pomocí CNN

Detekce objektů pomocí kotev je účinná, ale hodně výpočetně náročná, protože musíme ohodnocovat velké množství kotev. Menší obrázky pořízené průměrným chytrým mobilním telefonem mohou mít kolem 8 mega pixelů. Pokud na

takovém obrázku chceme detekovat objekty, zmenšíme ho například na 800 * 600 pixelů. Přestože použijeme jen 5 rozměrů kotev, získáme téměř dva a půl milionu kotev a to je na jeden rozměr obrázku hodně. Počet kotev však můžeme redukovat pomocí rovnoměrného vzorkování. Rovnoměrně vybereme jen ty pixely, kolem kterých se vygenerují kotvy. Není tedy potřeba generovat kotvy úplně na všech pixelech. Díky tomu si můžeme dovolit generovat i více různých rozměrů kotev. Velikost vzorkování udává míru redukce. Vzorkování pixelů probíhá postupně od jemného vzorku s malými kotvami, přes střední až k velkým kotvám, které jsou vystřeďované na střed celého obrázku.

K výběru obrázků můžeme také využít výstup konvolučních neuronových sítí (CNN), které odhalují struktury v obrázku. Způsob, jakým konvoluční neuronová síť zpracovává obrázek, nám dovoluje pro každou pozici pixelu na vstupu zjistit jeho pozici na výstupu. To lze díky tomu, že agregace pixelů, které se dějí v konvolučních a pooling vrstvách, mají přesně definované pozice svých vstupů a výstupu. Díky tomu může konvoluční neuronová síť extrahovat strukturu obrázku v jednom průchodu. Extrahované struktury pak mohou být přiřazeny na pozici v původním obrázku. a právě na tyto pozice se berou jako vzorky a generují se na ně kotvy.

Prostorové umístění ve vstupu může souviset s prostorovým umístěním ve výstupu. Tato konvoluční korespondence znamená, že CNN může extrahovat obrazové funkce pro celý obraz najednou. Extrahované funkce mohou být poté v daném obrázku přiřazeny zpět k jejich umístění. Použití kotevních boxů nahrazuje a drasticky snižuje náklady na přístup k posuvnému oknu pro extrahování prvků z obrazu. Pomocí kotevních boxů můžete navrhnout efektivní detektory objektů hlubokého učení tak, aby zahrnovaly všechny tři fáze (detekce, kódování prvků a klasifikace) detektoru objektů založeného na posuvném okně. Díky takto vylepšenému použití kotev můžeme elegantně vytvořit model, který nedodržuje klasickou architekturu (detekce, kalkulace vlastností oblastí a klasifikace oblastí) jako například modely založené na posuvném-okénku [22].

4.3.4 VGG

Modely využívající VGG nabízí několik konceptů heuristik, které se začali hojně využívat i v ostatních hlubokých neuronových sítích. V dnešní době se jedná již o starší koncept, takže si ho nebudeme popisovat příliš důkladně. VGG označuje sítě využívající bloky. Tyto bloky se dají skládat za sebe a využívat podle potřeby, a tak architektura sítě může být chápána mnohem abstraktněji.

Základní bloky klasických konvolučních neuronových sítí jsou po sobě jdoucí konvoluční vrstvy, nelineární aktivační funkce ReLU a pooling vrstvy. Hlavní z VGG bloků představuje po sobě jdoucí konvoluční vrstvy s 3*3 kernelem a jednu pooling vrstvu agregující maximum s 2*2 kernelem se stride rovno dvěma. Model VGG se skládá z konvoluční části, kde jsou za sebou

naskládané hlavní bloky a za nimi je několik plně propojených vrstev. Plně propojené vrstvy bývají tři. Originální VGG model měl těchto bloků za sebou pět, kde první a druhý blok měly pouze jednu konvoluční vrstvu a ostatní bloky měly dvě [23]. Celkem se v modelu nacházelo tedy 11 vrstev, a tak vznikl i název jeho první verze. VGG-11 vytváří síť pomocí opakovaně použitelných konvolučních bloků. Různé modely VGG lze definovat rozdíly v počtu konvolučních vrstev a výstupních kanálů v každém bloku.

4.3.5 Single Shot Multibox Detection (SSD)

Do této chvíle jsme při detekci objektů vždy museli nějakým způsobem iterovat přes vstupní obrázek. Single shot detekce, jak název napovídá, dokáže predikovat pravděpodobnosti kategorií a ohraňující boxy objektů v obrázku pouze jedním přečtením obrázku. Modely umožňující tento typ detekce mají architekturu, která se skládá z bloku s hlavní sítí, za nímž následují multiscale feature bloky (MSF), které generují kotvy, podle kterých blok pro predikci kategorie predikuje kategorii a blok pro predikci ohraňujícího boxu predikuje box.

Blok s hlavní sítí většinou představuje hluboká konvoluční neuronová síť, která v obrázku zjišťuje jeho struktury (anglicky feature map FM) ve velkém detailu. Nedochází tedy k takovému zmenšení dimenzí obrázku jako normálně. Její výstup FM je postupně předávám do MSF, kde se FM dále redukuje, generuje kotvy a následně probíhá samotná detekce v příslušných blocích. MSF je za sebou hned několik, a tak se postupným průchodem vstupního obrázku redukuje jeho vnitřní interpretace (FM). V každém kroku se generuje rozdílné množství kotev různých velikostí. Podle toho, jak daleko v modelu jsme, dochází k detekci postupně větších a větších objektů v obrázku počínaje od malých.

Blok predikce kategorie rozlišuje zadané třídy a přidává ještě třídu pozadí. Vstup tohoto bloku je aktuální velikost FM (šířky a výšku redukovaného obrázku) a jeho hodnoty. Počet vygenerovaných kotev můžeme opět rapidně zmenšit při použití triku s hlubokými konvoluční neuronovými sítěmi, jak jsme si vysvětlili v minulé kapitole. SSD v tomto bloku dokonce používá speciální konvoluční vrstvy, které mají stejný počet vstupů jako výstupů, a tak je mapování pozic vstupních pixelů na pozice těch výstupních triviální.

Trénovaní single shot multi-scale detektoru. Nejdříve inicializujeme všechny parametry na náhodné hodnoty a pomocí gradientního poklesu model optimalizujeme. Ze začátku trénování může dělat modelu problém zjistit, o jaké se jedná třídy a pak může predikovat špatně. Optimalizování kategorizace používá jako ztrátovou funkci křížovou entropii. Pro ohraňující boxy překvapivě používáme IOU, ale L1 ztrátovou funkci. Zjednodušeně se jedná o součet všech rozdílů predikce od pravdy v absolutní hodnotě. Při počítání ztrátové funkce je třeba nezapočítávat kotvy pozadí (negativní kotvy) a kotvy z přidaného okraje [24].

4.3.6 Region-based CNNs (R-CNNs)

Hlavní myšlenka regionálně-založené konvoluční neuronové sítě je provedení selektivního hledání ještě před konvoluční vrstvou. Selektivní hledání vytvoří množinu ohraňujících boxů. V práci [25] jsou označené jako návrhové regiony (proposal regions). Zmíněný výběr pomocí kotev je jeden z možných způsobů. Tyto návrhové regiony se obohatí o jejich třídu a ohraňovací box. Poté přichází na řadu konvoluční síť a provedením dopředného výpočtu se extrahují charakteristiky z každého návrhového regionu, abychom predikovali jejich kategorie a ohraňující boxy. R-CNN architektura se skládá kromě selektivního hledání a před-trénované konvoluční neuronové sítě ještě ze dvou dalších částí, které mohou pracovat paralelně. V první části dochází ke zkombinování charakteristik návrhového regionu s jeho kategorií jako příkladu pro trénování několika SVMs (support vector machines) pro klasifikaci objektů. Ve druhé části dochází ke zkombinování charakteristik návrhového regionu s jeho ohraňujícími boxy jako příkladu pro trénování pomocí regrese [26].

Nyní se většinou používá softmax vrstva. Porovnání suport vector machines a softmaxu se přímo uvádí ve článku Fast R-CNNs [27]. Na nízkých rozměrech vstupních dat je o trochu lepší SVM a na středních také. Na velkých datech jsou výsledky téměř stejné, ale nejlepší je sotmax. Musíme brát v ohledu, že se jedná o porovnávání jejich konkrétní implementace.

R-CNNs modely jsou velice efektivní, ale na druhou stranu jsou velice pomalé. Trvá jim obrovské množství času, aby se natrénovaly, protože musí zpracovávat 2000 návrhových regionů pro jeden obrázek. Přitom se tyto regiony z velké části překrývají, a tak dochází k opakování té samé práce. Tím pádem nemohou být implementované pro úlohy vyžadující real-timové řešení, protože predikovat jeden obrázek trvá řádově desítky sekund. Další nevýhodou je, že při jejich učení vlastně nedochází k učení všech čtyř částí, protože algoritmus selektivního hledání je daný a nemění se. Nyní si popíšeme další členy rodiny a známé regionálně-založených konvolučních neuronových sítí a poté bude následovat jejich přehledné porovnání.

4.3.7 Fast R-CNNs

Jednoduše řečeno Rychlé R-CNNs vylepšují normální R-CNNs pouze dopřednou konvoluční sítí, která zpracovává charakteristiky obrázku pro celý obrázek najednou. Charakteristiky obrázku namapujeme zpět na obrázek, a tak získáme body, kam umístíme návrhové regiony. Návrhové regiony následně prochází RoI (region of interest) pooling vrstvou, která změní jejich tvar na tvar fixní velikosti, abychom je mohli vložit do plně propojené vrstvy. Její výstup se označuje jako RoI vektor charakteristik [28]. Ten vstupuje paralelně do softmax vrstvy pro určení klasifikace a regresní vrstvy pro určení ohraňujících boxů. To znamená, že FR-CNNs predikuje kategorii a ohraňující box pro každý návrhový region.

Aby nedošlo ke zmatení, tak zmíněná RoI pooling vrstva není stejná jako klasická pooling vrstva z konvolučních neuronových sítí. RoI pooling vrstva nám umožňuje přímo specifikovat šířku a výšku výstupu pro každý region. V tomto regionu se pak podle velikosti zvoleného okénka provádí výběr maximálního prvku.

Rychlé R-CNNs jsou opravdu rychlejší než klasické R-CNNs při trénování i testování. Zpracování jednoho obrázku trvá síti už jen řádově pár sekund, avšak na realtimové použití je to stále moc. Jejich největší nedokonalostí je, že pro kvalitní detekci obrázků vyžadují stále velké množství návrhových regionů, generovaných selektivním hledáním. Právě to nahrazuje další model Faster R-CNN.

4.3.8 Faster R-CNNs

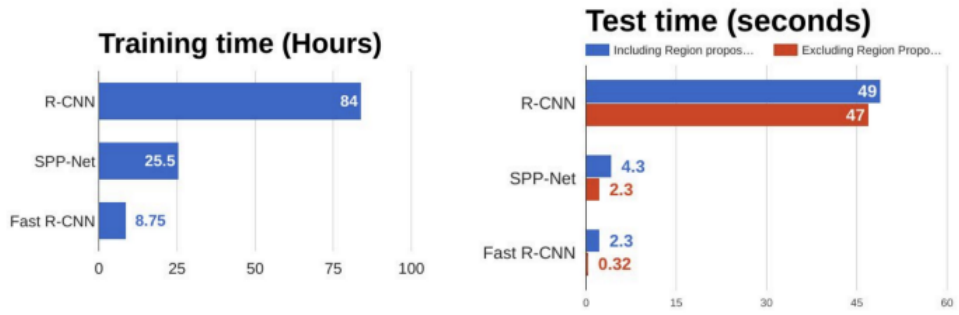
Ještě rychlejší R-CNN model představuje další vylepšení. Oba zmíněné předchozí modely (klasický R-CNNs a rychlý R-CNN) používají selektivní hledání k nalezení návrhů regionů. Selektivní hledání je ale poměrně pomalé, a proto ovlivňuje rychlost celé sítě. Z celkového času testování modelů zabírá nejvíce času (viz obrázek [obrázek rozložení casu vypočtu]).

Rychlejší R-CNN nahrazuje selektivní hledání novou sítí navrhující regiony (region proposal network RPN). Ta zajišťuje, že při zachování výkonosti predikce potřebujeme méně návrhů regionů. Ostatní části modelů jsou stejné jako u jejích předchůdců. Síť navrhující regiony (RPN) vezme vstupní obrázky libovolné velikosti a na výstupu vrátí množinu obdélníkových návrhů a u každého je ohodnocení objektivnosti. Jedná se o plně konvoluční neuronovou síť (FCN)[29].

Při trénování dochází k trénování celé sítě najednou, ne jako v předchozích případech. Tedy má několik objektivních funkcí a to pro klasifikaci objektů, ohraňujících boxů objektů, binární klasifikace návrhů regionů a jejich boxů. Díky tomu, že se RPN může učit, může predikovat návrhy regionů velice kvalitně a pak snižuje jejich počet.

Do rodiny R-CNNs patří ještě Maskovací R-CNN, která klasifikuje vstupní obrázek na hladině jednotlivých pixelů, čímž se dostáváme k sémantické segmentaci. Ta je velice zajímavá, protože nám umožňuje rozpoznávat každý pixel, a tak si utvořit mnohem komplexnější představu o tom, co kde na obrázku je. Sémantická segmentace však již nepatří do obsahu této práce, a proto se jí nebudeme zabývat.

V následující tabulce 4.3můžete přehledně vidět porovnání rychlostí detekce výše zmíněných modelů [30]. Z daleka nejpomalejší je model R-CNN s 49 vteřinami na jeden obrázek. Model SPP-Net jsme si neuváděli. Vylepšená verze Fast R-CNN je více než dvacetkrát rychlejší než ta původní a detekuje rychlostí 2,3 vteřiny na obrázek. U Faster R-CNN se dá říci, že se svou rychlostí detekce 0,2 vteřiny na obrázek téměř vstupuje do realtimových detektorů
.



Obrázek 4.3: Porovnání rychlostí modelů R-CNN

4.3.9 YOLO v1

YOLO V1 (You Only Look Once) je model, který nezaujímá jen chytlavým názvem a rychlostí, ale i této doby (2015) téměř kompletně novým pohledem na detekci objektů z obrázků. Doposavad zmíněné algoritmy rozpoznávání objektů fungují na principu upraveného klasifikátoru, který provádí detekci. YOLO se staví k detekci problémů jako k regresní úloze prostorově oddělit ohraňující boxy a zároveň jim přiřazovat pravděpodobnostní ohodnocení jednotlivých tříd. K detekci ohraňujících boxů a pravděpodobností tříd dochází YOLO neuronovou sítí přímo z obrázku na pouze jednu evaluaci. Protože toto celé dělá pouze jedna síť, která je sice komplexní, ale dá se optimalizovat od začátku do konce. Na rozdíl od předchozích modelů je mnohem rychlejší, a tak se dostává do sféry kvalitních realtimových modelů pro detekci objektů. V oficiálním článku [31] se uvádí, že normální YOLO model stíhá predikovat 45 snímků za vteřinu a jeho rychlejší verze Fast YOLO dokonce 155 snímků za vteřinu. K tomu všemu dosahuje až dvounásobných hodnot evaluační metriky mAP (mean Average Precision) oproti ostatních realtimových detektorů objektů.

YOLO nejdříve změní (většinou zmenší) velikost vstupního obrázku na 448 * 448 pixelů a pak následuje celkem 24 konvolučních vrstev, za kterými jsou dvě plně propojené. Na konci celého modelu je non-max potlačení, které jsme si již popisovali. Zmíněné konvoluční vrstvy jsou před-trénovány pro klasifikaci na polovičním rozlišení a následně zdvojnásobují svůj výstup, a tak se zachovává velikost, zatímco se velice vylepšuje rychlost. YOLO používá charakteristiky celého obrázku pro predikci každého ohraňujícího boxu. Také predikuje všechny boxy napříč všemi třídami. To znamená, že síť predikce zdůvodňuje globálně a pouze v určitém regionu jako předchozí modely. YOLO rozdělí obrázek na čtvercové mřížky. Za detekci objektu má zodpovědnost buňka mřížky, do které spadá střed detekovaného objektu. Každá buňka mřížky predikuje ohraňující box a ohodnocení příslušnost tříd pro maximálně nějaké

předem určené množství objektů. Pokud se v nějaké buňce žádný objekt nevyskytuje (nemá v ní svůj střed), tak jsou všechny ohodnocení klasifikace rovny nule. YOLO používá formát boxů ve tvaru x a y souřadnice středu objektu a šířku a výšku. Souřadnice středu objektu jsou ještě k tomu relativní pouze k buňce mřížky, kam spadá.

Trénování probíhá rozděleně. Nejdříve se před-trénuje prvních 20 konvolučních vrstev na ImageNet 1000-class datasetu. Těchto 20 vrstev se obohatí o average-pooling a plně propojenou vrstvu. Po přibližně týdnu trénování na grafické kartě Titan X dosahuje 88 % úspěšnosti a je srovnatelná s ostatními modely detekce objektů. Následně se za před-trénované vrstvy přidají další 4 konvoluční vrstvy, které dělají trik se zmenšením výšky a šířky vstupních dat na polovinu a na výstupu je opět zvětší. Také se přidají dvě plně propojené vrstvy s náhodně inicializovanými hodnotami. Poslední vrstva používá lineární aktivační funkci. Všechny ostatní používají takzvanou propouštěcí (leaky) lineární aktivační funkci, která pro vstup funkce větší než nula, vstup pustí dál, a pokud je menší než nula, tak ho změní na jeho jednu desetinu. YOLO model se optimalizuje přes součet druhých mocnin chyb (MSE). To zajišťuje stejnou váhu chyby pro velké i malé objekty. Malé odchylky ohraňovacích boxů u velkých objektů by ale měly vadit méně než u malých objektů. Kvůli tomu YOLO predikuje odmocninu šířky a výšky boxu místo přímo šířky a výšky.

4.3.10 YOLO v2

Populární model YOLO se dočkal velkých změn a po necelém roce vyšel článek "YOLO9000: Better, Faster, Stronger" [32]. Označení YOLO9000 se shoduje s V2. Číslo 9000 se uvádí, protože tvůrci modelu si všimli, že existující datasety pro detekci objektů z obrázků neobsahují zdaleka tolik tříd jako datasety určené pro trénování čisté klasifikace, a tak datasety spojili a výsledný dataset obsahoval právě přes devět tisíc tříd, které YOLO v2 umí klasifikovat. Trénuje se na datasetech ImageNet a COCO. Ačkoliv byla původní verze pokroková, ukázalo se, že dělá výrazně více lokalizačních chyb než rychlé konvoluční sítě založené na návrzích regionů (Fast R-CNNs).

Prvním vylepšením je zavedení Batch normalizace. Ta normalizuje vstupní vrstvu pomocí mírné změny a škálování aktivačních funkcí. Zvyšuje stabilitu neuronové sítě. Přidáním této normalizace do konvolučních vrstev došlo ke zlepšení o 2 % [32]. Batch normalizace dovoluje YOLO v2 odstranit vyhazovací vrstvu (drouput). V první verzi docházelo k zmenšování a zvětšování rozlišení vstupu. To se však postupem času ukázalo jako neefektivní a YOLO v2 se učí jen na obrázcích s vysokým rozlišením 448 * 448 pixelů (vzhledem k ostatním modelům). Model se inspiruje u konkurenčního modelu Faster R-CNN a využívá vlastností sítí navrhujících regiony RPN neboli kotvy. YOLO v2 vytváří mřížku stejně jako předchůdce, ale vždy o rozměrech 13 * 13, což je jemnější mřížka, než se používala v předchůdci, kde byla většinou 5 * 5, nebo 7 * 7. Protože počet objektů detekovaných jednou buňkou mřížky je

omezený, může druhá verze lépe detekovat menší objekty a hlavně celkově více objektů. Model je učený na obrázcích s náhodně vybranými rozměry (samozřejmě v určitém intervalu) a díky tomu je robustnější. Architektura obou verzí je založená na Darknetu. Druhá verze se skládá z 19 po sobě jsoucích konvolučních vrstev, 5 max pooling vrstev a na konci je softmax vrstva pro klasifikaci. Se všemi těmito vylepšeními se druhá verze zdá jako velice kvalitní model.

4.3.11 YOLO v3

V roce 2018 vychází nová a to třetí verze algoritmu YOLO. Její model se skládá z více vrstev, než u předchůdců a je přesnější. Třetí verze se zaměřuje především na přesnost a rychlost detekce. V této verzi se objektům dává ohodnocení za každý ohraňovací box. Pro predikci těchto ohodnocení používá logistickou regresi. Ohodnocení by se mělo rovnat 1, pokud predikovaný ohraňující box překrývá opravdový box nejvíce ze všech ostatních boxů. Pokud je hodnota menší než předem určená hranice, tak predikovaný ohraňující box zahazujeme. Jako ztrátová funkce pro trénování klasifikace se používá křížová-entropie.

Třetí verze již nepoužívá vrstvu softmaxu, protože ji shledává pro dobrý výkon modelu nepotřebnou a uvádí, že místo ní používá nezávislé logistické klasifikátory [33]. To se uvádí za účelem, aby síť mohla klasifikovat objekt do více tříd. Může se sice zdát, že je to kontraproduktivní, jako po správném klasifikátoru přece požadujeme, aby predikoval pouze jednu jmenovku třídy a ne více. To přesně zařizuje vyjmutý softmax, který výstup z neuronové sítě převedl na jednotlivé pravděpodobnosti. YOLO v3 díky logistickým kvalifikátoru předpovídá zvláště pravděpodobnost pro každý objekt. Uveďme si příklad. Na obrázku je skupina lidí, kde jsou muž, žena a další skupiny lidí. V množině tříd, které určujeme, máme jmenovku pro osobu, ale i konkrétněji pro muže a ženu. Normální klasifikátor nebo klasifikační část z detektoru objektů bude mít problémy rozhodnout se, zda se jedná o osobu nebo například ženu a kvůli tomu dostanou obě třídy ve výstupu jejich pravděpodobnostní ohodnocení velmi zredukované. Tomu se přesně snaží YOLO v3 s jednotlivými ligistickými klasifikátory zabránit. Ty se v našem případě zachovávají tak, že logistický klasifikátor pro třídu osoby vrátí vysokou hodnotu, stejně tak jako logistický klasifikátor pro třídu ženy, protože se navzájem nevylučují.

YOLO v3 implementuje predikci napříč rozlišeními. Pro kontext uvádím, že již existuje celkem 5 verzí YOLO, ale poslední dvě si nebudeme uvádět. Vraťme se zpět ke třetí verzi. V již zmíněném dokumentu [33] se uvádí, že třetí verze predikuje ohraňující boxy třemi různými rozlišeními. Charakteristiky se extrahují z každého rozlišení. Užívá se podobných metod jako u sítí pyramid charakteristik (feature pyramid network FPN). Jednotlivé charakteristiky z různých vrstev se spojují dohromady pomocí zřetězení. Díky tomu má síť jemnější představu o jednotlivých objektech.

Model třetí verze YOLO je opět navržený ve frameworku Darknet. Stejně jako druhá verze od první, tak tato třetí navyšuje počet vrstev. Používá síť Darknet-53, která má 53 konvolučních vrstev. Převzatá tabulka [obrázek tabulka 2 z https://arxiv.org/pdf/1804.02767.pdf] ukazuje, že Darknet-53 je přesnější než její předchozí verze Darknet-19 a drží se na stejné příčce jako ResNet-101 a ResNet-152, co se síly predikce týče. Přitom je síť YOLO v3 mnohem rychlejší a na testovacím vzorku si udržela 78 snímků za sekundu. Testovací prostředí sestávalo z obrázků o stejné velikosti 256 * 256 a čas měřený na grafické kartě Titan X.

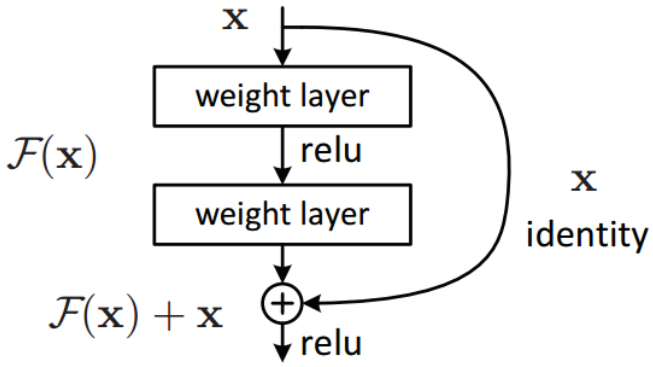
YOLO je kvalitní nástroj pro rozpoznávání a vždy patřil a patří ve své době k těm nejrychlejším modelům, a přesto dosahuje téměř stejných výsledků jako jeho nejsilnější konkurenti. Trénování se však zdá komplikované, protože jednotlivé vrstvy se musí před-trénovat samostatně a pak ladit dohromady. Tuto vlastnost však sdílí i s konkurenční rodinou modelů/sítí ResNet. Pro určování identity se tak síť vlastně musí naučit jen to, aby její vrstvy měny na výstupu nulu, která se sečte se vstupem, což je jednoduší než ji učit vracet x. Navíc pokud se chce naučit ještě něco jiného než identitu, stále na to má prostor u výstupů s nulou.

4.3.12 Residuální neuronové sítě (ResNet)

Residuální neuronové sítě určitě spadají do kategorie hlubokých sítí, neboť mají mnoho skrytých vrstev. Jak vlastně ResNet funguje. Dalo by se říci, že čím je síť hlubší, tak tím je lepší. To je však zavádějící, protože právě s rostoucí hloubnou neuronových sítí dochází od určitého bodu k poklesu výkonosti. K tomu dochází, protože síť ztrácí informaci o mapování identit tedy funkce f(x) = x. Abychom síti s tímto problémem pomohli, vytvoříme zkratkové propojení. To prakticky obejde celou síť od začátku až na konec, kde se pouze sečte s výstupem sítě. Na tomto principu můžeme sestavit model residuální neuronové sítě s jakoukoliv posloupností konvolučních sítí. Autoři ResNetu však zvolili model, který popíšu níže. a tak se od této chvíle v této kapitole pod pojmem residuální neuronové sítě bude myslet konkrétní ResNet.

Jejich model začíná dvěmi 7 * 7 konvolučními vrstvami, kde má každá na výstupu 64 kanálů a stride o velikosti 2. Následuje pooling vrstva, která svým posuvným okénkem (kernelem) o velikosti 3 * 3 projde celý jejich vstup a dál posílá maximum z okénka. Všechny pooling vrstvy v modelu musí dodržovat následující dvě pravidla. První je, že pro počet extrahovaných charakteristik musí mít vždy stejný počet filtrů. Druhá je, že pokud vrstva přijímá poloviční vstup, její počet filtrů bude dvojnásobný, aby se zachovala časová náročnost na vrstvu[34]. Díky tomu již není potřeba měnit šířku a výšku obrázku na vstupu. Následují konvoluční sítě míchané s residuálními bloky a na konci modelu je globální pooling vrstva, která agreguje průměry. Za tou ještě následuje plně propojená softmax vrstva a tak je celkem 34 vrstev, kde měníme parametry.

Residuální sítě používají za každou svou konvoluční vrstvou batch normalizaci ještě před aktivační funkcí další konvoluční sítě v pořadí. Residuální síť používá čtyři hlavní residuální bloky (viz 4.4 ze zdroje [34]), kde každý z nich používá několik residuálních bloků se stejným počtem výstupních kanálů. Residuální bloky se vyznačují zkratkovými propojeními, které začínají po zmíněné pooling vrstvě na začátku sítě a pak propojují oblasti vždy před a po dvou konvolučních sítích dohromady. Pro použití se vstupem do další vrstvy však musí mít stejnou dimenzi jako tento původní vstup. Na začátku každého hlavního bloku, kromě toho prvního, je zkratkové propojení, které dimenzi zvětšuje a přitom stále zachovává mapování identity charakteristik. Zvětšování dimenze vzniká přidáním nulových prvků.



Obrázek 4.4: Ukázka residuálního bloku

Residuální neuronové sítě v základním podání mají menší počet filtrů a nižší složitost než VGG sítě. Trénovat můžeme efektivně, protože díky residuálním blokům prochází datové kanály napříč bloky.

Existuje několik verzí ResNetu, které se dělí podle toho, kolik mají vrstev. Jejich princip je jinak úplně stejný a všechny mají ve své architektuře na začátku zmíněné 7 * 7 konvoluční a 3* 3 max pooling vrstvy. Uvádí se tedy modely pro 18, 34, 50, 101, 152 a 200 vrstev.

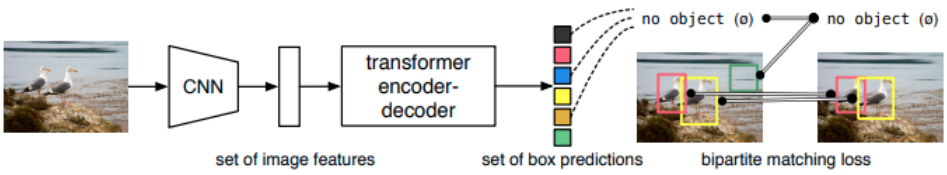
Nyní by bylo vhodné si ukázat, jak rychlé jsou jednotlivé verze s porovnáním výkonosti. V následující tabulce 4.1 ze zdroje [35] můžeme vidět ve spodní části přehledné porovnání. Metriky Top-1 chyby a Top—5 chyby se používají právě pro porovnávání výkonnosti hlubokých neuronových sítí. ResNet modely se s přibývajícími vrstvami postupně zlepšují v obou metrikách počínaje u hodnoty Top-5 chyby na 10,76 až do 5,79 pro síť ResNet-200. Spolu s množstvím vrstev však roste i jejich výpočetní náročnost, a tak sítě s více než 50 vrstvami přestávají být použitelné pro realtimové využití.

Network	Vrstev	Top-1 error	Top-5 error	Rychlost (ms)
AlexNet	8	42.90	19.80	14.56
Inception-V1	22	-	10.07	39.14
VGG-16	16	27.00	8.80	128.62
VGG-19	19	27.30	9.00	147.32
ResNet-18	18	30.43	10.76	31.54
ResNet-34	34	26.73	8.74	51.59
ResNet-50	50	24.01	7.02	103.58
ResNet-101	101	22.44	6.21	156.44
ResNet-152	152	22.16	6.16	217.91
ResNet-200	200	21.66	5.79	296.51

Tabulka 4.1: Porovnání výkonosti a rychlostí modelů

4.3.13 DETR

Doslova nyní, v květnu letošního roku 2020, publikoval Výzkumný ústav pro umělou inteligenci společnosti Facebook práci Ent-to-End Object Detection with Transformers [36], v níž představuje DETR. DETR je úplně nová metoda rozpoznávání objektů. Díky novému způsobů použití Transformerů, které se předtím používali pouze při práci s textem, již nepotřebuje ručně navržené komponenty modelů, jako jsou metody non-maximálního potlačení a generování kotev. DETR je označení pro DEtection TRansformer. Je to architektura složená z enkodé-dekodér architektury transformerů a z globální ztrátové funkce založené na množinách, která pomocí bipartního přiřazování dělá unikátní predikce. Jedná se tedy o velice jednoduchou architekturu (viz 4.5 ze zdroje [36]) složenou ze dvou hlavních modulů na rozdíl od modelů, o kterých jsme si říkali v předchozích kapitolách.



Obrázek 4.5: Architektura DETR

Nejdříve si udělejme představu o tom, jak vlastně tento model funguje. Řekněme, že máme nějaký obrázek, ve kterém máme detekovat veškeré objekty, určit kde přesně jsou a jaká je jejich kategorie. Obrázek dáme první části modelu. Jedná se o konvoluční neuronovou síť, která nám najde množinu charakteristik našeho obrázku. Nezapomeňme, že obrázek není jen jedna ma-

tice hodnot, ale hned tři, jedna pro každou barvu. Konvoluční neuronová síť z těchto tří matic vytvoří několik (více než tři) kanálů charakteristik neboli množinu charakteristik obrázku. Jedná se tak o detailnější reprezentaci obrázku než jen popis podle barev, jako tomu bylo předtím. Tato množina charakteristik následně putuje do modulu transformeru, který si více do detailu rozebereme později. Transformer predikuje množinu dvojic ohraňujících boxů a tříd. Tato množina je vždy stejné velikosti, která by měla být o něco větší než maximální množství objektů, co jsou najednou v obrázku. Je nutné zdůraznit, že transformer pracuje s třídou prázdného objektu. Nyní teoreticky máme všechno, co potřebujeme.

Otázka je, jak model natrénovat, protože anotované objekty v datasetu neobsahují třídu prázdného objektu. Transformer také detekuje více objektů na jeden a ten samý objekt s mírně rozdílnými ohraňujícími boxy. Model musí nějak zajistit správné přiřazení výstupů transformeru na anotované informace. Na to DETR používá bipartitní přiřazování. Toto přiřazování k sobě musí nějak napárovat N výstupních dvojic z transformeru a N dvojic z anotací. Anotací určitě bude méně než N a tak se (až ve fázi přiřazování) doplní o prázdné záznamy. Bipartitní přiřazování nemůže jednoduše porovnat prvky postupně, protože se jedná o množiny a na pořadí by nemělo záležet. Také chceme nějakým způsobem zajistit, aby se transformer nezaměřil jen na nejlépe rozpoznatelný objekt a ignoroval ostatní objekty, ale aby již detekované objekty neměli takovou šanci na to být znovu detekované, třebaže s trochu jiným ohraňujícím boxem.

Pro přiřazení potřebujeme definovat ztrátovou funkci (viz 4.6 ze zdroje [36]), která vezme dvojici (třída, ohraňující box) z transformeru spolu s dvojicí z anotací a určí jejich podobnost. Pokud se u této ztrátové funkce sejdou dvě dvojice, jejichž třída je bezobjektová, tak nás již nezajímá porovnání ohraňujících boxů. Sejdou-li se dvě dvojice s tou samou třídou (jinou než bezobjektovou) a jejich ohraňující boxy si odpovídají, tak je predikce dobrá a hodnota bude velice nízká (můžeme určit i negativní záleží na implementaci). Přiřazování chceme udělat tak, aby ohodnocení touto přiřazovací funkcí pro všechny přiřazení bylo co nejmenší. Tímto způsobem je přiřazování odolné vůči zmíněným problémům. Konkrétní implementace přiřazování není tak důležitá. Většinou se používá Hungarian matching (HM).

$$\hat{\sigma} = \underset{\sigma \in \mathfrak{S}_N}{\arg\min} \sum_{i}^{N} \mathcal{L}_{\text{match}}(y_i, \hat{y}_{\sigma(i)})$$

Obrázek 4.6: Vzorec ztrátové funkce DETRu

Druhá část Hungarian přiřazování vyžaduje ztrátovou funkci konkrétně pro ohraňující rámeček (viz 4.7 ze zdroje [36]). DETR predikuje boxy přímo

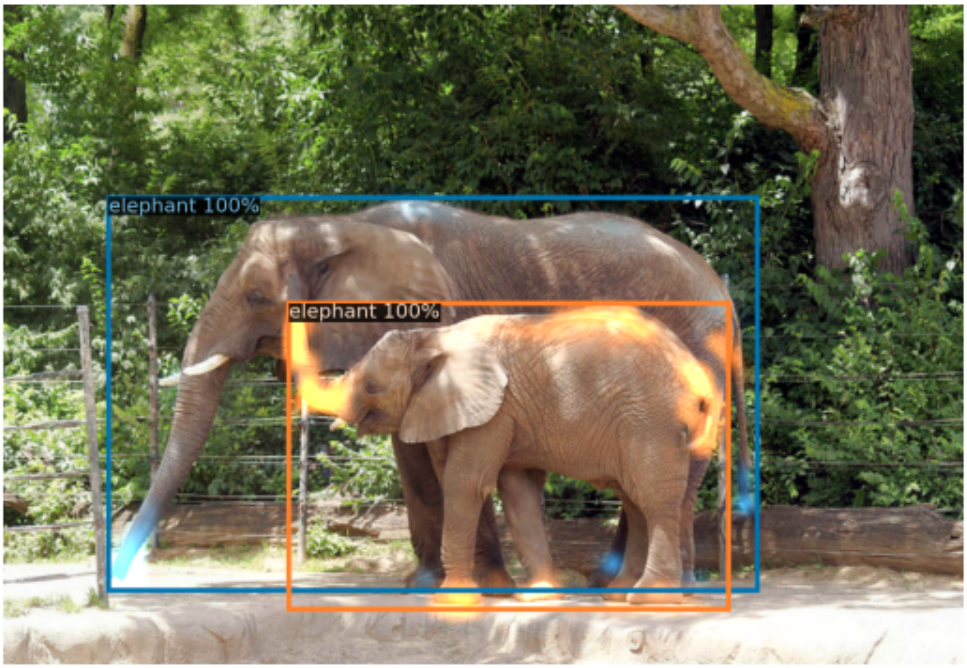
$$\mathcal{L}_{\text{Hungarian}}(y, \hat{y}) = \sum_{i=1}^{N} \left[-\log \hat{p}_{\hat{\sigma}(i)}(c_i) + \mathbb{1}_{\{c_i \neq \varnothing\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\hat{\sigma}}(i)) \right]$$

Obrázek 4.7: Hungarian matching

a nevytváří spoustu návrhů boxů a pak vybírá, který by mohl být ten reálný. Přestože takový přístup zjednodušuje implementaci, způsobuje to problém s relativním škálováním ztrátové funkce. Nejčastěji používaná L1 ztrátová funkce by měla rozdílné ztráty ro malé a velké objekty i přes to, že jejich relativní chyba je stejná. Proto DETR používá kombinaci L1 s GIOU (Generalized Intersection over Union). Obě jsou normalizované pro velikost objektů v jednom batchi.

Nyní se podíváme na architekturu více do hloubky. Konvoluční síť na začátku modelu vytvoří kanály charakteristik, které jsou stále 2D jako obrázek. Ještě před vstupem do enkodéru transformeru z těchto matic vytvoříme vektor sekvencí o velikosti (počet kanálů charakteristik) * (šířka * výška jedné mapy charakteristik) a obohatíme ho o zakódované pozice. Jak jsme si řekli v kapitole s transformery, jsou to modely, které zpracovávají sekvence, proto musíme mít vektor jakožto sekvenci. Transformeři pracují s pozorností (attencion) a díky tomu si mohou asociovat části vstupního vektoru, a tak si utváří představu o objektech v obrázku. Výstup enkodéru má stejnou velikost jako jeho vstup a následně vstupuje do dekodéru jako boční vstup. Transformer dekodér obecně funguje tak, že má na vstupu sekvenci a vrací sekvenci. Vstupní sekvence má velikost N – maximálního počtu tříd detekovaných na jednom obrázku a jednotlivým částem sekvence se říká sloty pro předpověď. Jedná se o sekvenci dvojic třída a ohraňující box. Po průchodu dekodérem reprezentuje tedy každý slot pro predikci stejnou dvojici, avšak již se správnými hodnotami. Na začátku mohou být sloty pro predikci náhodně ohodnocené, ale pro lepší výsledek se před-trénují. Sloty si můžeme představit jako skupinu lidí, kteří se zaměřují na určité části obrázku, které jsou však naučené a ne předem dané. Tito lidé společně vytvoří globální myšlenku o tom, kde a co jsou hledané objekty. Slovo globální je zde na místě, protože díky mechanismům pozornosti se mohou více či méně navzájem ovlivňovat. Toto se dá dobře vysvětlit na obrázku slonů 4.8 z [36], který se nachází ve zmíněné práci. Na obrázku jsou dva sloni, kteří se překrývají. Transformer (dekodér) se naučil věnovat pozornost chobotu, hřbetu, ocásku a koncům nohou slonů. Typicky se věnuje pozornost více částem poblíž ohraňujícího boxu než uprostřed.

Ve výzkumném oddělení pro umělou inteligenci ve Facebooku se rozhodli trénovat DETR na COCO 2017 datasetu, který obsahuje okolo 118 tisíc trénovacích obrázků a 5 tisíc validačních obrázků. V průměru je na jednom obrázku 7 ob-



Obrázek 4.8: Vizuální zobranení částí obrázků, kterým je věnována pozornost.

jektů a maximálně 63. Proto se počet slotů predikcí nastavuje na 100. Samotné trénování se provádí po 300 epoch a na jednom počítači s 8 grafickými kartami NVIDIA V100 [36] trval přes 6 dní, kde jedna epocha trvá 28 minut [37].

Následující tabulka 4.9 udává porovnání různých verzí DETR modelů a jeho konkurentů z rodiny R-CNN [36]. DETR ve všech metrikách AP (Avarage Precision) vítězí, kromě detekce na malých obrázcích a AP75.

4.4 Sledování objektů

Sledování objektů přirozeně navazuje na jejich detekci, na které zcela závisí. Má spoustu možných využití od strojového vidění, sledování osob z kamer, vozidel, chování davu, posunů a sesuvů půdy a spoustu dalších. Pro praktické aplikace by měla být technika sledování objektu nejen přesná, ale také rychlá, aby šla použít v reálném čase. Existuje několik různých algoritmů pro sledování, které se dělí na sledování pouze jednoho objektu nebo na sledování objektů více. Také se dělí podle toho, zda v sobě již mají nějaký detektor, nebo jsou od detektoru odděleni. Ti, co mají detektor zabudovaný, mají velkou výhodu v tom, že můžou využívat vnitřní reprezentace aktuálního snímku v sítí detektoru, a tak mohou vlastně predikovat budoucí pozici objektu a nejen jeho ohraňující box a například kategorii. Jsou také velice pomalé, a tak se

Model	GFLOPS/FPS	#params	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Faster RCNN-DC5	320/16	166M	39.0	60.5	42.3	21.4	43.5	52.5
Faster RCNN-FPN	180/26	42M	40.2	61.0	43.8	24.2	43.5	52.0
Faster RCNN-R101-FPN	246/20	60M	42.0	62.5	45.9	25.2	45.6	54.6
Faster RCNN-DC5+	320/16	166M	41.1	61.4	44.3	22.9	45.9	55.0
Faster RCNN-FPN+	180/26	42M	42.0	62.1	45.5	26.6	45.4	53.4
Faster RCNN-R101-FPN+	246/20	60M	44.0	63.9	**47.8**	**27.2**	48.1	56.0
DETR	86/28	41M	42.0	62.4	44.2	20.5	45.8	61.1
DETR-DC5	187/12	41M	43.3	63.1	45.9	22.5	47.3	61.1
DETR-R101	152/20	60M	43.5	63.8	46.4	21.9	48.0	61.8
DETR-DC5-R101	253/10	60M	**44.9**	**64.7**	47.7	23.7	**49.5**	**62.3**

Obrázek 4.9: Porovnání rychlosti verzí DETRu s ostatními modely.

nedají používat v reálném prostředí. Výhodou sledování bez vlastního detektoru je, že můžeme poměrně jednoduše vyměňovat detektory a modely jako moduly a celá koncepce je tak přehlednější.

Úkolem sledování je asociace jednotlivých detekovaných objektů napříč po sobě jsoucími snímky. Při detekování z nového snímku tedy musíme nějakou metodou přiřadit nové detekce k těm již existujícím a tím identifikovat konkrétní objekt. Touto identifikací není na mysli klasifikace, ale opravdu identifikace, tedy přiřazení unikátního indexu, který si daný objekt drží napříč snímky a nemění se.

Při rešerši jsem nenarazil na ucelený soupis všech, nebo alespoň většiny, metod sledování objektů. Musel jsem se tak dříve, než jsem čekal, zanořit do jednotlivých prací zabývajících se sledováním v nich a jim podobných hledat jednotlivé způsoby.

Konkrétní způsoby:

Scale-invariant feature transform (SIFT)

Jedná se o starší techniku, která byla vynalezena v roce 1999 [38]. Propojuje sledování s detekcí. Extrahuje charakteristiky snímku a ukládá je do databáze. V následujícím snímku dochází také k extrakci a následnému porovnání s tou původní. Charakteristiky převede na vektory. Charakteristiky po sobě jdoucích snímků na sebe mapuje pomocí Euklidovské vzdálenosti jejich vektorů. Z množiny mapovaných charakteristik některé vybere a určí je za klíče, pokud odpovídají danému objektu svou lokací, měřítkem a orientací. Následně stanovuje konzistentní skupiny klíčů pomocí algoritmu zobecněné Hougovy transformace. Skupina je uznána, pokud se v ní vyskytují tři a více klíčů. Pro uznané skupiny se spočítá pravděpodobnost obsahování objektu.

Generic Object Tracking Using RNN (GOTURN)

Sledovací algoritmus implementovaný původně v jazyku c++. Existují však i jeho verze v Pythonu (PY-GOTURN) [39]. Zaměřuje se na problematiku sledování pouze jednoho objektu, kterému byl v prvním snímku videa určen ohraňující box a jmenovka. Ačkoliv v této práci budeme implementovat sledování více objektů, GOTURN přináší zajímavé metody, kterých můžeme využít.

Používá hluboké regresní sítě. Ty trénuje na obrázcích s anotacemi ohraňujících boxů. U natrénovaného modelu se zmrazí parametry (váhy) a následně se používá ke sledování. Mezi sekvencemi obrázků jsou i takové, které se mírně rotují, kamera se hýbe, nebo se mění světlo a stín. Proto s těmito jevy umí model pracovat. Práce se snaží dokázat, že je možné naučit obecný model sledování objektů off-line a následně ho používat v realtimových situacích. GOTURN je schopen pracovat s rychlostí 100 snímků za vteřinu. Toho model dosahuje z velké části kvůli tomu, že ořezává region pro určování pohybu sledovaného objektu. Region se zmenší na poměrně malé okolí kolem lokace objektu z minulého snímku. Model je náchylný k rychle se pohybujícím objektům, protože se cílová pozice objeví mimo region.

Vstupem modelu je celý obrázek a region. Následuje několik po sobě jdoucích konvolučních vrstev, které zachytí reprezentaci obrázku pomocí charakteristik. Nyní data prochází několika plně propojenými vrstvami, jejichž úkol je porovnat charakteristiky objektu a s charakteristikami aktuálního snímku a posoudit, kam se objekt pohnul. Konvoluční vrstvy jsou převzaté z CAFFE zoo a konkrétně se jedná o CaffeNet. Výstupem regresní sítě je lokace objektu v aktuálním snímku ve formátu souřadnic levého horního rohu a pravého dolního rohu ohraňujícího boxu.

Multi-Domain Network Tracker (MDNet)

Vítěz soutěže VOT2015. Architektura modelu MDNet má 5 skrytých vrstev, ze kterých jsou první tři konvoluční [40]. Za nimi následují dvě plně propojené vrstvy. Síť se následně dělí na K větví, kde má každá ještě jednu plně propojenou vrstvu. Těchto K větvím se říká domény, a proto název Multi-Domain. Domény vznikají kvůli problému rozpoznávání pozadí obrázku. Pozadí objektů v jednom videu může být totiž sledovaný objekt v druhém videu. Model přijímá obrázky o velikosti 107 * 107 pixelů při se třemi kanály (RGB). Tři konvoluční vrstvy ho postupně sníží na 51*51, 11*11 a 3*3 s počtem kanálu postupně 96, 256 a 512. Těchto 512 vstupů následně přijímá první plně propojená vrstva, která používá ReLU a dropout. Plně propojená síť na každé z větví je binární klasifikátor se softmax vrstvou. Jinými slovy MDNet dělá chytře to, že rozdělil svůj model na dvě části, kde první část je sdílená a druhá nezávislá pro každou doménu. Nezávislé části se pro lepší sledování dají ladit online (za běhu).

Hlavní úlohu při trénování zastává rozdělení objektů a pozadí v každé doméně. Konvoluční vrstvy se trénují pomocí stochastického poklesu gradientů (SGD). MDNet využívá tvou typů aktualizací – dlouhodobých a krátkodobých. Dlouhodobé se provádí opakovaně s použitím v průběhu nasbíraných dat a krátkodobé se provádí vždy, když selže sledování objektu. Díky tomu si MDNet dokáže udržovat kvalitní představu o všech objektech z videa a nejen o těch co jsou vidět. V obrázku z webových stránek MDNetu[41] je vidět, že dosahuje perfektních výsledků.

Recurrent YOLO (ROLO)

ROLO [42] je založený na modelu YOLO, kterého využívá pro získání bohatých charakteristik a detekcí objektů. Těchto charakteristik následně využívá rekurentní neuronová síť s dlouhou krátkodobou pamětí (LSTM). Díky regresní schopnosti LSTM je ROLO, jak prostorově, tak časově, schopen interpretovat řadu vizuálních prvků na vysoké úrovni přímo do souřadnic sledovaných objektů. ROLO se trénuje ve třech fázích. Předtrénování konvolučních vrstev pro extrakci charakteristik na datasetu ImageNet, YOLO trénovaní pro detekci objektů a trénování LSTM pro sledování objektů. Jako ztrátovou funkci používá střední kvadratické chyby (MSE). Implementovaná verze pomocí jakyku Python s použitím frameworku TensorFlow pracuje s rychlostí YOLO 20 a LSTM 60 snímků za vteřinu.

Simple Online and Realtime Tracking (SORT)

Jednoduchý algoritmus sledování objektů. Nasazuje se na detektor a pracuje s jeho výstupem. Dá se tak využít u různých modelů. Na rozdíl od ostatních sledovacích algoritmů není založen na snaze odchytnout různé okrajové případy, ale na spolehlivosti jednoduchého přiřazování napříč snímky a rychlosti. SORT dokáže sledovat více objektů najednou rychlostí až 260 Hz. To je, jak se v práci [43] uvádí, až 20x rychlejší než ostatní sledovací algoritmy z roku 2018. Problémy se zmíněnými okrajovými SORT přesouvá na detektor. Poprvé byl SORT nasazen na model Faster Region Convolutional Neural Network (Fr R-CNN), kde dosáhnul zmíněných výsledků [43].

SORT má tři parametry a jsou to `max_age` určující maximální možný počet snímků, kdy objekt není detekován, aby k němu bylo možné přiřadit stejný identifikátor. Druhý parametr je `min_hits` určující kolikrát se minimálně musí objekt detekovat, aby k němu byla přidána asociace napříč snímky značená identifikátorem. Poslední ze tří je `iou_threshold`, který určuje minimální hladinu IOU pro asociaci dvou na sobě ležících (asi stejných) objektů. Tyto parametry je dobré nastavit přímo pro konkrétní sekvenci či video, protože pokud se například vozidlo pohybuje rychle, dochází k velkým změnám jeho pozice napříč snímky, a tak může být jeho OIU velice malé.

Centroid Tracker

Tento detektor se podobně jako SORT využívá v případě, kdy máme oddělený modul s modelem detekce a modul pro sledování. Centroid Tracker je postaven na vzdálenosti středů (center) jednotlivých ohraňujících boxů. Nepracuje tak s celými ohraňujícími boxy, ale pouze s jejich středy. K ukládání a práci s identifikovanými objekty používá strukturu OrderedDict, která pracuje stejně jako normální slovník (Dict), ale pamatuje si pořadí vložení prvků. Při iteraci jsou položky vráceny v pořadí, v jakém byly nejprve přidány jejich klíče. Po každém snímku je tracker ve stavu, kdy ve svém slovníku uchovává centroidy a jejich id, kde id je klíč a centroid je hodnota. Při volání jeho funkce update, která má na starosti přiřazení nových centroidů k těm existujícím, objevení těch, které mají být do trackeru registrovány, a smazání těch, které již nebyly delší dobu vidět. Doba pamatování si centroidu, i když není vidět, je určená vstupním parametrem `maxDisappear` při inicializaci trackeru. Druhý parametr `maxDistance` určuje, o kolik nejvíce pixelů se může centroid posunout, aby k němu bylo možné přiřadit stejné ID. Výsledné sledování je velice citlivé na tyto parametry a je ideální upravit je pro konkrétní sekvenci obrázků či video. Samotné přiřazování nových centroidů na ty staré funguje na euklidovské vzdálenosti [44]. Protože je přiřazování založené na vzdálenosti, je možné, že si dva překrývající objekty prohodí identifikátor, či se jeden z nich potlačí a zapomene. Centroid Tracker je jeden z nejjednodušších a díky tomu nejrychlejších trackerů.

4.5 Statistiky ze sledování objektů

Pokud máme k dispozici detekované objekty s identifikátory, můžeme o nich dělat různé statistiky. Které statistiky však záleží na tom, jaké informace o těchto objektech uchováváme. Také záleží na způsobu pořízení sekvence vstupních obrázků či video záznamu. Sledování vozidel, dopravních situací, pohybu osob po komunikaci a spousty dalších informací patří mezi žádané výstupy strojového vidění čím dál více žádané výstupy strojového vidění.

První statistika, kterou můžeme provádět i na jednom snímku, je **počet objektů**. Větší smysl ovšem dává měřit počet objektů při sledování napříč snímky. V každém snímku tak může být (například v rohu snímku) vyznačeno, kolik se na něm právě vyskytuje objektů. Pokud se kamera nehýbe, je tedy statická, můžeme pozorovat **trajektorie pohybu objektů**. Trajektorie lze sledovat, i když se kamera pohybuje, avšak pohyb kamery mění pozici na snímku pro všechny objekty, a tak vznikají velké nepřesnosti v datech. Je málo případů, kdy se dá využít statistik trajektorií pro záběry s různými změnami úhlu záběru a přiblížení, či při pohybu. Sledování trajektorií se zdá ideální nejen pro vozidla. Při sledování trajektorií lze odhalit i vozidla, která změnila

směr své jízdy o určitý **úhel**.

Můžeme také sledovat **vzdálenost pohybu objektu** napříč snímky. Primárně je vzdáleností myšlena vzdálenost v pixelech na obrazovce. Pokud však máme přesně změřenou oblast z video záznamu, můžeme vzdálenost v pixelech převádět na **reálnou vzdálenost pohybu objektu**, například v metrech. K tomu je však zapotřebí připravit statistiky pro konkrétní video záznam. Pokud měříme vzdálenost, můžeme nad jejími hodnotami vytvářet spoustu pozorování. Například objekt, který se pohyboval **nejdelší/nejkratší vzdálenost**. Dále je možné **třídit objekty do kategorií podle ujeté vzdálenosti**, či zkoumat **průměry vzdáleností objektů**.

S využitím vzdáleností objektů za snímek můžeme počítat jejich **rychlost** v pixelech po snímku, či v reálných jednotkách, pokud známe oblast záběru. Můžeme dělat záznamy o **nejrychlejším/nejpomalejším objektu** či objektech, nebo měřit jejich **průměrnou rychlost**. Užitečné je dělat záznamy, kdy, kde a proč docházelo k **prudkému zpomalení, či naopak zrychlení objektů**. Z takovýchto informací zle odhadnout místa či situace, které například zdržují provoz nebo jsou nebezpečné pro okolní objekty (vozidla, chodce,...). Na statických záběrech zle ručně vyznačit **linka**, pro kterou můžeme sledovat počet objektů, které linku překročily. Dokonce zle sledovat dvě hodnoty, neboť linka se dá překročit dvěma způsoby. Nejčastější použití linky při sledování vozidel je k počítání, kolik vozidel ji v jakém směru či pruhu překročilo, a tak si utvářet představu o dopravní situaci. **Linek se dá jistě použít i více**, tím se může měřit například na statických záznamech z kruhového objezdu, kolik vozidel opouští objezd, jakým výjezdem či naopak, jakým vjezdem se na kruhový objezd dostanou. Díky tomu je pak možné tuto sledovanou oblast přizpůsobit, či dokonce přestavět, aby se zvýšila plynulost provozu.

Stejně tak jako linky se dají ručně vyznačit i **oblasti**. Tyto oblasti jsou vlastně podmnožiny původního obrázku, a tak na nich lze sledovat a vytvářet statistiky úplně stejně jako na původním obrázku. Mezi hlavní využití oblastí patří sledování **vjezdu do zakázaných zón**, či počítání času objektu v této sledované zóně (například parkování na omezenou dobu). Stejně tak se dá zjistit informace o tom, jak dlouho se objekt vyskytuje na celém obrázku, tedy uvádět statistiky o **čase stráveném objektem v záznamu**.

Pokud máme k dispozici celý ohraňující box objektu, nikoliv pouze jeho střed, můžeme porovnávat pro tento konkrétní objekt **velikost ohraňujícího boxu napříč snímky**. Při záběrech z kamery umístěných na kapotě vozidla může být tato statistika velice užitečná a odhalovat prudké zabrzdění vozidla nacházejícího se před tímto vozidlem.

Kapitola 5

Existující řešení zabývající se detekcí a sledováním vozidel

V této kapitole si popíšeme existující projekty implementující řešení pro detekci vozidel, sledování vozidel nebo obojí. Na začátek je nutné říci, že detekce a sledování vozidel je úzké zaměření z celého počítačového vidění. Přesto, že se omezuje pouze na jednu třídu objektů, kterou jsou vozidla, může se jednat i o tříd více, jako jsou: auta, nákladní auta, autobusy, trolejbusy, cyklistická kola, tříkolky a povozy. Jak jsme si uvedli v kapitole o ladění modelu, můžeme vzít před-trénovaný model a upravit ho na detekci vlastních tříd. Stejně tak můžeme využít před-trénovaného modelu, který umí pracovat s obrovskou množinou tříd, kde budeme většinu z nich ignorovat. Tímto rozšířením lze do této kapitoly zařadit téměř všechny implementace modelů, které jsme si v předchozích kapitolách důkladně popsali.

Nutno zdůraznit, že v průběhu analýzy existujících řešení bylo nalezeno mnoho prací, které se problematikou zabývaly, nicméně jen zlomek z nich poskytoval nějakou formu implementace, která byla dohledatelná. Nalezené práce také často měly implementaci v jazyku c++. Z těchto projektů se mi nepodařilo najít žádný, který by implementoval mnou zvolený model DETR, pokud neberu v úvahu DETR samotný, který je natrénovaný na datasetu COCO, který obsahuje i vozidla.

Nyní se ale podívejme na konkrétní existující projekty.

Faster R-CNN od Yao Xiao

Mezi první mnou nalezené se řadí implementace Faster R-CNN od Yao Xiao [45], o které jsem se dozvěděl z práce [46]. Je založená na frameworku TensorFlow a Keras a vyžaduje balíčky cython, opencv, numpy, matplotlib a scipy. Použití frameworku TensorFlow odůvodňuje autor práce tím, že je jednoduší při nasazování na více zařízení, čehož se v průmyslu hojně využívá. Používá před-trénovaný model Faster R-CNN na datasetu Pascal_VOC a COCO, avšak umožňuje ho trénovat i od

začátku. Trénování a evaluace probíhá přesně podle oficiálního řešení modelu. V práci se uvádí, že pro různé extraktory charakteristik (VGG-16, ResNet-50, ResNet-101) se AP pohybuje od 57 % do 68 %, nejvyšších procent dosahují různé verze ResNet-101.

Vehicle detection with Keras

Další nalezená práce se zabývá detekcí vozidel ze záznamů získaných kamerou, která je umístěná na palubní desce v automobilu [47]. Využívá konvolučních neuronových sítí a je založená na frameworku Keras. Model konvoluční sítě se skládá z:

- Normalizační vrstvy Lambda
- 3x 2D konvoluční vrstva s aktivační funkci ReLU a strides rovno 2
- 2x 2D konvoluční vrstva s aktivační funkcí ReLU
- Flatten - zplošťovací vrstva
- Dropout - vyhození signálu s 50
- 4x Plně propojená vrstva s výstupními dimenzemi postupně (100, 50, 10 a 1)

Na trénování používá neznámý dataset, který obsahuje téměř 9 tisíc obrázků bez vozidel a přes čtyři tisíce obrázků vozidel.

Jak si můžeme všimnout z poslední plně propojené vrstvy, má pouze jeden výstup, tedy tento model predikuje pravděpodobnost, že na obrázku je, či není vozidlo. S přidáním anotací do datasetu a lehkou úpravou modelu, by však mohl predikovat i více tříd vozidel.

Vehicle Detection and Tracking

Práce s názvem "Vehicle Detection and Tracking"[48] se naneštěstí zabývá pouze detekcí. Zabývá se binární klasifikací vozidlo/ne-vozidlo. K tomu využívá část KITTI dataset s vozidly. Trénuje na stejném počtu obrázků z obou kategorií. Jak autor uvádí, model měl problémy s detekcí bílých aut a rozšířil dataset o dalších 200 obrázků s bílými auty. Protože se jedná o záběry z kamery v autě, může si dovolit omezit oblast pro detekci ořízntutím horní poloviny obrázků, protože tam se ve většině případů nevyskytuje žádné vozidlo. Používá následující architekturu:

- Normalizační vrstvy Lambda
- Konvoluční vrstvu s 16 filtry, velikostí kernelu 3 *3 a aktivační funkcí ReLU
- Dropout - vyhození signálu s 50
- Konvoluční vrstvu s 32 filtry, velikostí kernelu 3 *3 a aktivační funkcí ReLU

- Dropout - vyhození signálu s 50
- Konvoluční vrstvu s 64 filtry, velikostí kernelu 3 *3 a aktivační funkcí ReLU
- Pooling vrstvu, která agreguje podle maximum o výstupní velikosti 8 * 8
- Dropout - vyhození signálu s 50
- Konvoluční vrstvu pouze s 1 filtrem, velikostí kernelu 8 * 8 a aktivační funkcí ReLU

Nečekané využití poslední konvoluční vrstvy s jedním filtrem jako způsobu, jak získat pouze jednu hodnotu. Velikost datasetu udává na 7,5 tisíce. Natrénovaný model se chlubí se správností (accuracy) 99,4 %. Tato hodnota je až příliš dobrá, a tak je důležité se zamyslet proč. Pravděpodobně kvůli tomu, že správnost je citlivá na vyváženost dat, může tak i u poměrně špatné predikce určit vysoké procento.

VB vehicle detection in highway scenes

Práce s celým názvem Vision-based vehicle detection and counting system using deep learning in highway scenes [49] vypadá velice slibně. Bohužel se k ní nevztahuje žádný kód implementace, ale přesto jsem se rozhodl ji zde uvést. Zabývá se detekcí a počítáním vozidel na dálnicích. Má k dispozici dataset videí z dálnic z kamer používaných pro sledování stavu provozu. Ve videích se mění velikost objektů, protože vozidla se postupně přibližují ke kameře a pozorovací úhel. Porovnává různé datasety pro detekci vozidel, ale nakonec si vytváří vlastní, protože se ty existující zdají nevyhovující. Tento dataset má na rozdíl od velkých datasetů jako je PASCAL VOC, ImageNet a COCO největší počet zastoupení, vzhledem k velikosti obrázků, na kterých jsou 3 až 8 vozidel. Využívá modelu YOLO v3, který detekuje vozidla a následně díky ORB extrakci charakteristik dochází ke sledování více objektů. Model YOLO v3 jsme si již popsali v předchozích kapitolách. Protože je kamera stále ve stejné pozici, může jednoduše určit oblast videa, která se nehýbe, a tak označit oblast silnic, kde bude detekovat vozidla. Tímto maskováním snímků se velice omezí oblast detekování a model bude rychlejší.

Ke sledování objektů využívá predikovaných ohraňujících boxů, které získává z detekce objektů modelem YOLO. Následně používá algoritmus ORB a tvrdí, že s perfektní alternativou k podobným algoritmům jako je SIFT a SURF. Jeho proces sledování vypadá následovně:

- Získá z detektoru vstupní ohraňující boxy
- ORB algoritmus extrahuje charakteristiky
- Pokud se objekt shoduje s již existujícím objektem

  - Aktualizuj jeho pozici
  - Aktualizuje jeho stopu a udržuje její informaci pro posledních 10 snímků
- Jinak čekej 10 snímků, jestli se nenajde objekt, který by se s ním shodoval
- Jinak odstraň objekt

V práci se provádí analýza trajektorií, díky vlastnímu datasetu již dopředu vědí, co očekávat. Na všech video záznamem dálnice putují auta buď směrem ke kameře, nebo od kamery. Doprostřed jednotlivých snímků umisťují sledovací linii. Pokud detekované vozidlo linii překročí, jednoduše přičtou k čítači. Uvádí, že podle směru překročení čáry mohou určit na jaké straně dálnice dané vozidlo je, a tak mají pro každou stranu jeden čítač. Dokonce mají čítač i pro každou kategorii vozidel.

Dataset o celkové velikosti přes 11 tisíc snímků rozdělují na trénovací a testovací podle 80 % a 20 %. Výsledky jsou velmi pozitivní s detekcí na 87 % a sledováním kolem 92,5 %. Tato práce působí uceleným a kvalitním dojmem.

OpenCV Vehicle Detection, Tracking, and Speed Estimation"

Práce od Adriana Rosebrock z roku 2019 [44] je přehledně vypracovaný článek návodného charakteru popisuje implementaci detekování a sledování vozidel s odhadem jejich rychlosti. Vše probíhá v realtimovém video streamu. Implementace je založená na frameworku OpenCV. Zajímavostí této práce je, že je i hardwarově podložená, protože k zachycování videí probíhá na modulu Raspberry Pi a výpočet na Intel Movidius Neural Compute Stick. Video obohacené o anotace potom ukládá na dropbox.

Pro měření rychlostí využívají toho, že mají kameru stále na stejném místě, a tak mohou vyznačit body v prostoru a asociovat je přímo s konkrétními pixely, to se zde provádí ručně. Pro sledování používá Centroid Traker, který si pamatuje po určitou (předem zvolenou jako parametr) dobu pozice středů detekovaných vozidel a porovnávání podle vzdálenosti středů identifikuje objekty. Implementace využívá před-trénovanou síť MobileNet SSD, kterou lze stáhnout z Caffe model zoo. Tento model byl před-trénovaný na obrázcích s celkem 21 třídami. Implementace po detekci modelem zahodí všechny objekty, které nejsou označeny jako automobil. To je velké plýtvání výpočetních prostředků, přesto model stíhá predikovat v přiměřeném realtimovém čase. Práce neuvádí výkonost modelu, ale podle videoukázky na stránkách projektu pracuje celkem dobře.

Build your own Vehicle Detection Model using OpenCV and Python

Práce nebo spíše návod [50] z roku 2020 uvádí nejen stručný popis problematiky, ale i implementaci v jazyce Python s použitím frameworku

OpenCV. K detekci nepoužívá neuronovou síť, ale pouze analyzuje obraz po sobě jdoucích snímků. Tyto snímky převádí z barevných na černobílé a následně je pomocí funkce OpenCV od sebe odečte. Tím získá mapy lokací na obrázku, kde došlo ke změně barvy pixelů. Na tento pomocný obrázek rozdílu dále aplikuje prahování, kde převede černobílý obrázek z šedé škály barev pouze na dvě (černou a bílou) podle určité hranice. Tím odstraní vzniklý šum a lokace změn jsou jasnější. Následně ořezává horní polovinu, kterou shledává neužitečnou pro detekci. Z pomocného obrázku nalézá kontury objektů, které vykresluje na původní obrázek a tím se zvýrazní "detekuje" vozidlo na obrazovce.

Tento návod je zajímavý tím, že nepracuje s žádným modelem založeným na hlubokých neuronových sítích, jako ostatní práce. Nicméně stále se jedná o detekci objektu ve videu/obrázku. Popisuje jednoduchost práce s frameworkem OpenCV. Jako vstupní video používá dataset, který není anotovaný, nelze tedy určit správnost, ale vzhledem k tomu, že tato metoda označí veškerý pohyb ve videu, tak vlastně detekuje dobře. K tomu však potřebuje, aby se napříč snímky nehýbala kamera, na tom je celá práce postavená.

Vehicle Detection and Tracking

Projekt se nacházejícím v repositáři [51] představuje implementaci a popis detekování a sledování více vozidel s použitím kamery nainstalované uvnitř samořídícího se automobilu. Krom toho se snaží o co největší jasnost implementace postupných operací a modulů tak, aby se daly poměrně jednoduše vyměňovat jednotlivé algoritmy detekce nebo sledování. Program nejdříve inicializuje konkrétní modul detektoru a sledovače. Následně detektor lokalizuje vozidla v každém snímku videa. Modul sledovače v každém snímku zpracuje výstupní hodnoty z detektoru a identifikuje jednotlivé objekty. Na závěr jsou výstupy z obou modulů použity k vykreslení výsledků na původní snímek.

Využívá před-trénovaného modelu SSD MobileNet v1 na datasetu COCO, který obsahuje kolem 90 tříd, kde se prvních 14 zabývá transportem. Tento projekt se bohužel nezabývá samotným trénováním od začátku. Autor se rozhodl detekovat automobily, autobusy a náklaďáky. Odstraňuje anotace s velikostí ohraňujícího boxu menšími než 20 pixelů na šířku i na výšku. To zdůvodňuje redukcí chyby prvního typu (false positive). To je zajímavý nápad, protože u takhle malých detekcí není jisté, že se budou detekovat napříč snímky a docházelo by ke zbytečnému zmatení v modulu sledovače.

Používá Kalmanovy filtry, které popisuje a ručně implementuje. Při sledování porovnává IOU objektu na aktuálním a předešlém snímku a krom základních vlastností sledovačů přidává filtrování, že pokud se ohraňující boxy liší v IOU o více než třetinu, považuje je za rozdílné objekty.

V této kapitole jsme popsali několik existujících řešení a nalezli zajímavé nápady a přístupy, jak dosáhnout některých cílů. V kapitole vlastní implementace si představíme, které z těchto nápadů ji ovlivnili a jakým způsobem.

Kapitola 6

Evaluační metriky

V této kapitole si určíme, jaké existují objektivní metriky pro měření výkonu neuronových sítí, co od nich můžeme očekávat a jaké jsou jejich klady a zápory. Doposud jsme si představovali neuronové sítě s jejich strukturou a algoritmy, ale je potřeba i testovat, jak dobře fungují. Rozebereme si metriky pro neuronové sítě zabývající se klasifikací, detekcí objektů a sledováním objektů.

6.1 Klasifikace

Začneme maticí záměn, nejedná se sice přímo o evaluační metriku, ale potřebujeme znát její části, abychom evaluačním metrikám porozuměli. Pokud mámě klasifikační problém, tak se pro daný vstup snažíme určit, jaké třídě z množiny všech přípustných tříd odpovídá. Pokud si tuto úlohu omezíme pouze na dvě třídy, jedná se o binární klasifikaci. U binární klasifikace určujeme, zda vstup do nějaké kategorie spadá, či nikoliv. Uveďme si tedy příklad a představme si, že máme celkem 1100 obrázků, kde na 1000 obrázcích není auto a na 100 obrázcích auto je. Příklad je inspirovaný naším problémem detekcí vozidel. K tomu máme test, který určí, zda se jedná o obrázek auta, či nikoliv. Pokud otestujeme všech 1100 obrázků, můžeme sestavit matici záměn. Ta má dva sloupce a dva řádky. Indexy řádků jsou určené pro skutečnosti, které známe a indexy sloupců pro výsledek predikce. Indexy jsou vždy značeny pravda a nepravda. V literatuře se řádky a sloupce matice záměn často zaměňují, nebudeme si tedy matici záměn znázorňovat. Pojďme na její obsah pro náš případ. V matici vznikla čtyři okénka, kde každé má jiný význam.

- TP (true positive) = 90 - Označuje počet obrázků s autem, označených jako auto.
- FN (false negative) = 10 - Označuje počet obrázků s autem, neoznačených jako auto.

- TN (true negative) = 940 - Označuje počet obrázků bez auta, označených jako auto.
- FP (false positive) = 60 - Označuje počet obrázků bez auta, neoznačených jako auto.

Můžeme si všimnou, že první uvedená dvojice TP a FN se týká klasifikace na obrázcích, na kterých auto doopravdy bylo a druhá dvojice TN a FP klasifikace na obrázcích, na kterých auto určitě nebylo. Od těchto hodnot se odvíjí výpočet metrik klasifikace.

Správnost (accuracy)

Je možná nejjednodušší metrika, jakou si člověk dokáže představit, a je definován jako počet správných předpovědí dělený celkovým počtem předpovědí. Pro získání procent vynásobíme ještě 100.

$$\text{Správnost} = \frac{TP + TN}{TP + TN + FP + FN}$$

Ve výše uvedeném příkladu je tedy z 1100 vzorků 1030 správně predikováno, což má za následek správnost klasifikace 93,6 %. Správnost je citlivá na vyváženost dat, může tak i u poměrně špatné predikce určit vysoké procento. Dochází k tomu, protože velkou část z čitatele přidá negativní skupina, v našem příkladu jsou to správně klasifikované jako obrázky bez aut obrázky, na kterých auta nejsou. Pokud by test předpovídal všechny obrázky, jako že na nich nejsou auta, tak by stále měl okolo 90% správnosti. Pokud jsou data hodně nevyvážená, můžeme využít dalších metrik.

Přesnost (Precison)

Pohlíží na konkrétní třídu. Tedy buď na pozitivní, či negativní. Říká nám, kolik pozitivních predikcí jsou doopravdy pozitivní a obdobně pro negativní třídu.

$$\text{Přesnost} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

V našem případě vychází pro pozitivní třídu 60 % a pro negativní 98.9 %. Z těchto výsledků můžeme vidět, že přesnost učení auta je velice malá. Přesnost určení negativní třídy nás většinou tolik nezajímá a v tomto případě je velmi vysoká.

Pokrytí (Senzitivita a specificita) (Recall and specificity)

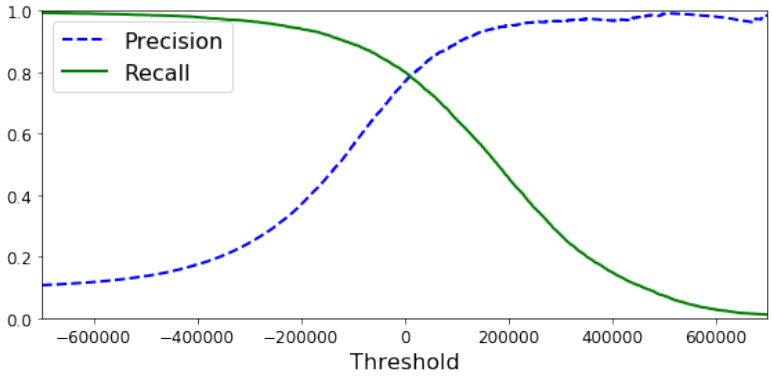
Další metrika pohlížející na konkrétní třídu. Definuje se jako poměr počtu vzorků z třídy, které jsou správně predikované. Český výraz pokrytí pod sebou schovává senzitivitu i specificitu, takže se nejedná o překlad

anglického slova recall, které označuje pouze senzitivitu, zatímco specificitu anglické slovo specificity.

$$Senzitivita = \frac{TP}{TP + FN}$$
$$Specificita = \frac{TN}{TN + FN}$$

V našem příkladu vychází pro pozitivní třídu krásných 90 % a pro negativní 94 %.

Jak jsme si uvedli, tak evaluační metriky mají své nedostatky. Velmi efektivní však je, když je používáme zároveň. V perfektním případě, pokud máme dokonale oddělitelná data, lze docílit, aby obě metriky přesnost a pokrytí byly 1. V reálném světě to zajistit nejde, a tak narážíme na problém kompromisu (viz 6.1 ze zdroje [52]). Musíme si vybrat, zda chceme, aby náš model (test) měl vysoké výsledky přesnosti, nebo naopak pokrytí, nebo můžeme vybrat kompromis.



Obrázek 6.1: Porovnání přesnosti a senzitivity (precision vs recall)

Nyní už rychle k poslední klasifikační metrice zmíněné v této práci. Existují samozřejmě i jiné jako obsah grafu pod ROC křivkou a další.

F1 scóre

Přichází v potaz, pokud chceme zkombinovat hodnoty přesnosti a pokrytí a v našem příkladu vychází 72 %.

$$F1 = \frac{PT}{PT + \frac{FP+FN}{2}} = 2 \cdot \frac{\text{Přesnost} \cdot \text{Pokrytí}}{\text{Přesnost} + \text{Pokrytí}}$$

Top-N

Jedná se o hodnotu určující přesnost klasifikace. N nejčastěji bývá hodnota 1 nebo 5. Pokud používáme Top-1 máme, porovnáváme modelem predikovanou třídu s nejsilnější pravděpodobností s tou opravdovou.

Při použití Top-5, porovnáváme, jestli se opravdová třída nachází mezi 5 nejsilněji predikovanými třídami. Top-5 je tedy benevolentnější než Top-1. Porovnání probíhá tolikrát, kolik děláme predikcí a na konci se zprůměruje.

6.2 Evaluace detekce a sledování objektů

U detekce objektů z obrázků predikuje model dva výstupy. Jeden z nich je třída detekovaného obrázku a druhým je jelo lokace. Tedy při zjišťování přesnosti těchto modelů je zapotřebí nejdříve nějakým způsobem zjistit přesnost těchto dvou predikcí a následně je určitým způsobem zkombinovat. To, jak hodnotit přesnost modelu, který klasifikuje objekt z určité množiny tříd, jsme ji již ukázali v předchozí podkapitole. Jak jsme si již zmínili, tak lokace detekovaných objektů se nejčastěji definuje pomocí ohraňujícího boxu. Máme tedy čtyři číselné hodnoty v nějakém formátu pro predikovaný ohraňující box a čtyři pro ten s pravdivými (anotovanými) hodnotami. Musíme nejdříve zajistit, aby oba formáty byly stejné. Nabízí se možnost tyto čtyři hodnoty sečíst a porovnávat je se sečtenými (nebo jinak matematicky agregovanými, ale v obou případech stejně) pravdivými hodnotami a následně jejich porovnání řešit jako klasickou regresní úlohu pomocí střední čtvercové chyby (MSE) či variací této metody. Tato možnost se v našem případě však nedá použít, neboť model si díky reprezentaci čtyř hodnot v jedné nemůže při trénovaní z této jedné hodnoty vydedukovat ostatní, a tak by nemohl měnit jednotlivé části ohraňujícího boxu. Nabízí se metoda porovnávající predikované a opravdové hodnoty ohraňujících boxů podle jejich obsahu.

6.2.1 Průnik nad sjednocením

Průnik nad sjednocením aneb anglicky intersection over union (IOU) je jedna z nejpřímočařejších metod evaluace detekce obrázků. Porovnává míru shody dvou ohraňujících boxů podle jejich obsahu a využívá při tom znalostí z teorie množin [53]. Na každý pixel obrázku můžeme pohlížet jako na jeden prvek z množiny všech pixelů obrázku. Podobně nám vzniknou dvě podmnožiny pixelů obrázku (pro pixely v predikovaném a pixely v opravdovém boxu). Následně můžeme u těchto dvou množin, z logiky teorie množin, provádět operace průniku a sjednocení. Pokud bychom se při učení modelu snažili maximalizovat průnik, model by se snažil predikovat takové boxy, aby zaplnili co nejvíce prostoru, takže by sice označovali hledaný objekt, ale spolu s ním i široké okolí. Podobně nelze použít pouze minimalizování sjednocení. Řešením je Jaccard index, který používá poměr průniku a sjednocení těchto množin.

$$IOU = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard index nabývá hodnot od 0 do 1, kde nízké hodnoty patří ohraňujícím boxům, které nejsou moc podobné a vysoké těm více podobným. Optimálně se snažíme dosáhnout hodnoty jedna, kdy se boxy rovnají.

6.2.2 Generalizovaný průnik nad sjednocením

Normální způsob evaluace IOU informuje dobře o tom, zda se ohraňující boxy překrývají nebo ne. Předpokládejme, že náš model udělá špatnou predikci ohraňujícího boxu, která se s neprotíná s opravdovým ohraňujícím boxem. Hodnota IOU bude 0. V další iteraci udělá náš špatný, ale už o trochu lepší, model predikci, která je o dost lepší, ale stále se neprotíná s opravdovým ohraňujícím boxem, a tak je stále 0 a model nezjistí, zda se přibližuje nebo ne. Přesně tento problém adresuje GIOU.

$$GIoU = \frac{|A \cap B|}{|A \cup B|} - \frac{|C \backslash (A \cup B)|}{|C|} = IOU - \frac{|C \backslash (A \cup B)|}{|C|}$$

Ve vzorci představují a a B ohraňující boxy predikce a pravdy. C označuje nejmenší konvexní obálku bodů a a B. Pokud se boxy a a B neprotínají, ale přiblížily se, víme, že se zlepšujeme, protože celková velikost C bude menší.

6.2.3 MOT

K porovnání výkonosti pří sledování objektů je potřeba znát opravdové pozice těchto objektů a jejich predikce. U obojího samozřejmě i indexy jednotlivých objektů, abychom je mohli správně porovnávat.

MOTP (multiple object tracking precision) je metrika přesnosti určující celkovou chybu pozicí na celém snímku zprůměrovaná celkovým počtem provedených operací přiřazení aktuálních pozicí k předešlým pozicím. Tato metrika kvalitně odhaduje schopnost modelu odhadnou přesnou lokaci objektu, či zachování konzistentních trajektorií.

Kapitola 7

Implementace

V rešeršní části jsme se zanořili do problematiky technologií plynoucích ze zadání práce. Ukázali jsme si, jakým způsobem pracují jednotlivé druhy neuronových sítí a řešení z nich vyplívajících. Popsali jsme si algoritmy potřebné k správnému trénování sítě a ladění modelu. Nyní je na čase popsat vlastní implementaci. Po analýze modelů detekujících a sledujících objekty jsem se rozhodl zvolit takovou architekturu, která umožňuje jednodušeji vyměňovat jednotlivé moduly. Z toho plyne, že by moduly pro detekci, sledování a vyvozování statistik, měly pracovat co nejvíce samostatně a měly by si předávat jen nutné informace.

Hlavní modul implementace se zabývá detekcí objektů v obrázcích / snímcích video záznamu. Ačkoliv jsme si ukázali hodně různých modelů s vysokou výkonností, nejvíce mě zaujal model DETR (End-to-End Object Detection with Transformers), který k detekci využívá transformery. Zvolil jsem si ho nejen kvůli jeho výsledkům v porovnání s ostatními modely, ale i proto, že je to úplná novinka z letošního roku 2020 od výzkumného oddělení celosvětově známe firmy Facebook.

Jako druhý přichází na řadu modul pro sledování. Z analýzy víme, že existují modely se zabudovaným detektorem a bez něj. V tomto případě jsem si tedy musel vybrat model takový, který se dá použít odděleně. Přesně takový je SORT (Simple online and realtime tracking) a Centroid Tracker, a tak jsem do implementace zakomponoval oba dva. Uživatel si při spouštění programu evaluace videa sám určí argumentem, který z nich chce použít. Tyto moduly pro sledování získávají vstup ve formě predikovaných lokací objektů z modulu pro detekci. Tento vstup zpracují a přiřadí jednotlivým lokacím identifikátor podle toho, jakému patří objektu. Lokace a identifikátory sledovaných objektů jsou následně ukládány do struktur umožňující vyhodnocování statistik.

7.1 Nastavení prostředí

Program musí být implementován v jazyku Python. Toho lze dosáhnout na jakémkoliv operačním systému, nicméně jsem kvůli samotné práci s operačním systémem a instalací závislostí modelů zvolil operační systém Linux. Jako distribuci Linuxu jsem si zvolil Ubuntu 18.04, neboť jsem ji měl již nainstalovanou na sestavě počítače. Později jsem přešel na novější verzi Ubuntu 20.04. Sestava počítače, na které jsem primárně pracoval, má následující parametry:

- Procesor: Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz se 4 jádry a 6MB vyrovnávací paměti L3
- Grafická karta: GeForce GTX 1060 s 6 GB GDDR5 a šířkou sběrnice 192-bit
- RAM: 12 GB

K trénování jsem občas využíval sestavu počítače na Fakulta informačních technologií ČVUT v Laboratoři zpracování obrazu. Její parametry jsou.

- Procesor: Intel(R) Core(TM) i5-7600K CPU @ 3.80GHz se 4 jádry a 6MB vyrovnávací paměti Smart Cache
- Grafická karta: GeForce RTX 2080 Ti s 11 GB GDDR5
- RAM: 32 GB

Jako vývojové prostředí jsem si zvolil PyCharm, neboť byl v několika porovnáních zvolen za nejlepší [54]. Ke stažení je verze zdarma (Community Editions). ČVUT k němu dokonce nabízí aktivaci prémiových služeb, tedy verzi (Professional Edition). Menší části kódu jsem si ověřoval/zpracovával v cloudové aplikaci Google Colab, která zadarmo umožňuje spouštět v Pythonu na přiřazených procesorech a grafických kartách až po dobu 12 hodin.

Implementace programu v Pythonu požaduje po používaném systému, aby měl Python nainstalovaný. Verzí Pythonu existuje několik a pro tuto práci jsem si vybral a nainstaloval Python 3.8.2. K práci na konkrétním projektu je vhodné vytvořit virtuální prostředí, které umožňuje vytvářet svou vlastní nezávislou sadu balíčků Python. Jako virtuální prostředí jsem použil nástroj `venv`, který jsem nainstaloval a vytvořil si prostředí `my_env`, kam jsem instaloval balíčky projektu v průběhu implementace. Balíčky se instalují pomocí manažera balíčků, kterých existuje několik (pip, conda, pipenv, Packagr, wheel,...). Jednotlivé balíčky, které jsem chtěl v průběhu nainstalovat, měly většinou své vlastní webové stránky, na kterých uváděli postup instalace. Většina z nich k instalaci používala pip, a tak jsem si zvolil ten. Manager balíčků pip je určen pro Python verze 2.X.Y a pro verzi Pythonu 3.X.Y je určen pip3. Ve virtuálním prostředí s verzí 3.X.Y zle používat příkaz pip, neboť je nastavený jako odkaz na výchozí příkaz pip3.

7.2 Struktura projektu

Abych zjistil do detailu, jak funguje implementace DETR a abych mohl použít speciální ztrátovou funkci, kterou DETR požaduje, přidal jsem do projektu jeho repositář. To jsem udělal příkazem `git clone`. Stejným způsobem jsem do projektu přidal i repositář s SORT. Následně jsem v projektu vytvořil další soubory a složky, které si dále rozebereme. Struktura projektu je znázorněná zde 7.1.

- `centroid` ..... Složka obsahující Centroid Tracker
  - `centroidtracker.py` ..... Soubor s Centroid Trackerem
- `dataset` ..... Složka pro výchozí načítání datasetu
  - `annotations` ..... Složka s anotačními soubory
  - `images` ..... Složka s obrázky určenými k učení modelu
  - `train.csv` ..... Soubor shromažďující anotace
- `detr` ..... Složka modelu DETR
- `images` ..... Složka s obrázky pro evaluaci
  - `annotated` ..... Složka pro anotované obrázky
  - `toannotate` ..... Složka pro k anotaci
- `models` ..... Složka pro ukládání modelů
- `sort` ..... Složka modulu SORT
- `videos` ..... Složka určená k načítání a ukládání videí
- `classes.py` ..... Soubor s polem názvů tříd
- `convert_annotations.py` ..... Skript pro vytvoření anotačního souboru
- `loss_agregator.py` ..... Soubor s agregátorem ztrát
- `mymodel.py` ..... Soubor pro upravený DETR model
- `test_detector.py` ..... Soubor pro testování detektoru
- `train.py` ..... Soubor pro trénování modelu
- `utils.py` ..... Soubor s pomocnými funkcemi a vykreslováním
- `seqence_test.py` ..... Soubor pro testování na sekvenci obrázků
- `video_test.py` ..... Soubor pro testování na videích
- `visdrone_dataset.py` ..... Soubor s třídou Dataset a transformacemi

Obrázek 7.1: Struktura projektu

7.3 Dataset a jeho zpracování

Pro trénování a testování DETR používám dataset VisDrone [55], který disponuje přes sedmi tisíci obrázků v trénovací a validační části. Protože v programu využívám trénovací i testovací části dohromady, je nutné si obrázky z obou částí překopírovat do jedné složky a stejně tak anotace. K tomu jsou ve struktuře projektu připraveny složky `dataset/images` a `dataset/annotations`, které by obrázky a anotace měly obsahovat pro správný chod programu při

výchozím nastavení. Uživatel si nicméně lokace těchto složek může určit sám pomocí argumentů při spuštění programu. Tento dataset oficiálně obsahuje 12 kategorií: `ignored`, `pedestrian`, `person`, `bike`,`car`, `van`, `truck`, `tricycle`, `awning trycicle`, `bus`, `motor`, `other`. Původně obsahoval o trochu více kategorií, ale VisDrone je kvůli nedostatečnému počtu označilo jako `other` třídu. V souboru `classes.py` je k nalezení pole tříd datasetu VisDrone a datasetu COCO (pro porovnání). V následující ukázce je vidět úryvek anotace obrázku před zpracováním:

871	572	54	92	1	4	0	0
948	592	62	92	1	4	0	0
874	705	67	110	1	4	0	1

Tabulka 7.1: Ukázka části anotačního souboru

Abych mohl jednodušeji přistupovat k určitým anotacím, vytvořil jsem skript `convert_annotations.py`, který ukládá anotace do jednoho souboru a umožňuje je upravovat. Skript se spouští pomocí příkazu z příkazového řádku, nebo s poižitím IDE. s těmito parametry:

- `--new_anns_file` určuje název souboru, kam chce uživatel uložit nový soubor, který vznikl agregováním anotací,
- `--img_dir` určuje název složky s obrázky,
- `--anns_dir` určuje název složky s anotacemi k těmto obrázkům,
- `--min_width` určuje minimální rozměr ohraňujícího boxu anotace, aby byla zpracována programem a uložena do nového souboru s anotacemi,
- `--max_obj_per_image` určuje maximální počet anotací pro jeden obrázek,
- `--only_vehicles` určuje, zda se mají vyfiltrovat pouze anotace obsahující nějaká vozidla.
- `--bbox_format` určuje, v jakém formátu chceme data zapsat. Hodnota může být buď 'xxwh' nebo 'ccwh'.

Minimální rozměr ohraňujícího boxu se používá v případě, kdy nechceme mít mezi výslednými anotacemi miniaturní nic neříkající detekce. Omezení maximální počtem anotací na obrázek je velice důležité, protože DETR může detekovat jen určité množství objektů a počet anotací na jeden vstupní obrázek DETRu by měl být vždy menší. Následně skript kontroluje, zda zmíněné složky a soubor existují. Program zahlásí výjimku, pokud soubor existuje a pokud složky neexistují. Pokud počet obrázků respektive jejich názvy neodpovídají počtu anotací respektive jejich názvům, program opět zahlásí výjimku. Nyní je skript ve stavu, kdy jsou všechny potřebné věci ošetřeny, a tak se pokusí

vytvořit na odpovídajícím místě nový soubor pro anotace. Do výstupního souboru vypíše hlavičku `image_id,width,height,bbox,source` a začne postupně procházet po dvojicích všechny obrázky a jejich anotace. Získá informace o jménu, šířce obrázku, výšce obrázku, ohraňujících boxech a jejich kategoriích. Podle regulárního výrazu:

`^[0-9]+,[0-9]+,[0-9]+,[0-9]+,[0-9]+,[0-9]+,[0-9]+,[0-9]+$`

kontroluje, zda obsah načtených anotačních souborů je validní. Po posledním stažení uvedeného datasetu byl o pár nových obrázků spolu s jejich anotacemi rozšířený, avšak tento regulární výraz jeden soubor anotací nesplňoval! Při procházení souborů se kontroluje minimální rozměr ohraňujících boxů a filtrují se podle kategorie.

Pokud anotace odpovídá filtru, je v jiném formátu přidána do výsledného souboru. Pokud po filtraci jednotlivých záznamů ze souboru anotací byl použit jen jeden nebo dokonce žádný záznam, informuje o tom skript uživatele. Skript vypíše celkový počet anotací přidaných do výstupního souboru. Při určování třídy jsou vozidla označena jako třída 0 a ne-vozidla jak třída 1. Po úspěšném zakončení běhu skriptu by ve zvolené lokaci měl existovat soubor odpovídajícího jména s filtrovanými anotacemi všech obrázků. Může začínat podobně jako na následující ukázce:

image_id	width	height	cat	x	y	w	h
0000001_02999_d_0000005	1920	1080	0	871.0	572.0	54.0	92.0
0000001_02999_d_0000005	1920	1080	0	948.0	592.0	62.0	92.0
0000001_02999_d_0000005	1920	1080	0	874.0	705.0	67.0	110.0

Tabulka 7.2: Ukázka části nového anotačního souboru

Soubor `visdrone_dataset.py` obsahuje třídu `Dataset` která dědí z abstraktní třídy `torch.utils.data.Dataset` přejmenované na `TorchD`. Moje třída `Dataset`, tak musí implementovat metody `__len__` pro zjištění velikosti datasetu a `__getitem__` pro možnost získání i-tého prvku při volání `dataset[i]`. Než budeme tyto metody volat, musí být ve třídě již nadefinované informace o tom, kde hledat anotace a obrázky. Metoda `__getitem__` vezme z načteného souboru s anotacemi pouze ty anotace, které se týkají požadovaného obrázku. Pokud to jde, tak tento obrázek načte, jinak vyhodí výjimku. Pixely načteného obrázku znormalizuje násobením 1/255, tedy jejich hodnoty jsou následně mezi 0 a 1. Z anotací se vezmou ohraňující boxy a spočítají se jejich obsahy. Podobně se z anotací vezmou i označení tříd. Pokud anotace převádíme skriptem `convert_annotations.py` s argumentem `--only_vehicles True`, nebo ho vůbec neuvedeme, budou třídy všech načtených záznamů rovny nule. Pokud uvedeme `False`, budou třídy načtených záznamů rovny 0 nebo 1.

Následuje transformace obrázku, ohraňujících boxů a labelů pomocí Albumentations [56]. Podle toho, jestli se jedná o trénovací, nebo validační

část datasetu se provedou odpovídající transformace. K tomu slouží funkce `get_train_transforms()` a `get_valid_transforms()`. U transformace testovacích dat dochází k několika transformacím obrázku s určitou pravděpodobností p.

- A.RandomGamma(p=0.1),
- A.RandomBrightnessContrast(p=0.1),
- A.ToGray(p=0.01),
- A.HorizontalFlip(p=0.5),
- A.VerticalFlip(p=0.05),
- A.CoarseDropout(num_holes=8, fill_value=0, p=0.1),
- A.Resize(height=512, width=512, p=1),
- ToTensorV2(p=1.0)

Ohraňující boxy a labely se změní automaticky podle změny obrázku. Obrázek se v transformaci přemění na tensor. Po transformaci normalizuji velikosti ohraňujících boxů. Na závěr veškeré transformované informace z anotací převedu na tensory a přidám do slovníku `target`. Funkce `get_valid_transforms()` neprovádí žádné vizuální transformace, pouze změní velikost obrázku a udělá z něj tensor.

7.4 Model

Pro tuto práci jsem si zvolil model DETR, neboť představuje nový způsob řešení problematiky detekce objektů a v rámci závěrečných prací a projektů v České republice jsem nenašel žádné, které by model používali. V souboru `model.py` se nachází třída `DETRModel`, která dědí z třídy torch.nn.Module. Představuje zapouzdření pro model DETR nacházející se v `detr/models/detr.py`. Třída `DETRModel` umožňuje uživateli stáhnout plně před-trénovaný model, tedy již s natrénovanými parametry, nebo nepřed-trénovaný model, bez naučených parametrů. To zařídí následující příkaz v kódu

`self.model = torch.hub.load('facebookresearch/detr', model_name, pretrained=pretrained)`.

Před-trénovaný model je naučený na COCO datasetu, kde objekty spadají do 91 tříd, které umí rozpoznat, tedy predikovat, jejich ohraňující box a třídu. Ze zadání plyne, že implementovaný model má klasifikovat objekty na vozidla a ostatní, máme tedy pouze 1 hlavní třídu a to vozidla. Proto je potřeba v před-trénovaném modelu nahradit lineární vrstvu `class_embed`. Tato vrstva by měla mít počet vstupních parametrů stejný jako předtím

(`class_embed.in_features`), avšak dimenze výstupních parametrů by měla být počet tříd + 1. Třídy se navyšují o 1, protože DETR přidává "no-object" třídu, jak jsme si vysvětlili výše. Nahrazení vrstvy dosáhneme pomocí kódu:

`self.model.class_embed = nn.Linear(in_features=self.hidden_dim, out_features=self.num_classes)`.

Na závěr je nutné zopakovat, že DETR má vždy předem daný počet detekcí. Tento počet určuje proměnná `num_queries`. Jak se uvádí v komentářích DETRu, tento počet by měl být větší než počet objektů v jednom obrázku. Výchozí nastavení hodnoty, se kterou je ověřeno, že DETR funguje, je 100. Maximální počet objektů v jednom obrázku v datasetu VisDrone je ale 540 a tak je potřeba anotace obrázků upravit, aby odpovídaly hodnotě menší než 100, nebo zvětšit `num_queries` na hodnotu větší. Metoda `forward(self, images)` předává obrázky do modelu DETRu, který očekává vstup ve formě `NestedTensor` a vrací predikované pravděpodobnosti pro příslušnost objektu do tříd `pred_logits` a ohraňující boxy `pred_boxes`.

7.5 Trénování

Trénováním modelu pro detekci objektů se zabývá soubor train.py. Tento soubor se dá samostatně spustit pomocí příkazu `Python train.py`, který umožňuje uživateli měnit následující parametry:

- `--img_dir` určuje cestu ke složce, která obsahuje anotované obrázky určené k učení.
- `--anns_file` určuje cestu a název souboru s anotacemi k obrázkům z minulého parametru. Musí odpovídat formátu `image_id, width, height, cat, x, y, w, h`, bez mezer a proto doporučuji použít k jejich vygenerování skript `convert_annotations.py`.
- `--batch_size` určuje velikost jednoho batche. Tedy pro kolik obrázků se má model spustit, než dojde ke zpětné propagaci a učení.
- `--epochs` určuje počet kompletních průchodů trénovacího datasetu.
- `--num_of_queries` určuje maximální počet objektů v jednom obrázku.
- `--learning_rate` určuje míru učení. Čím je hodnota vyšší, tím dochází k větším úpravám.
- `--folds` určuje počet překladů vstupního datasetu. Dataset je rozdělen na tento počet částí, kde při použití křížové validace je jeden vždy označen za testovací část a zbytek je sjednocen do trénovací části.
- `--null_class_coef` určuje velikost relativní váhy aplikované na detekované objekty z přidané třídy no-object (bez objektová třída).

- `--loss_ce` určuje velikost relativní váhy aplikované na pravděpodobnosti příslušnosti tříd.
- `--loss_bbox` určuje velikost relativní váhy aplikované na ohraňující boxy.
- `--loss_giou` určuje velikost relativní váhy aplikované na generalizovaný průnik nad sjednocením GIOU.
- `--detr_model` určuje, jaký model ze zoo facebookresearch/detr chceme využít k trénování. Možné použít `detr_resnet50` s 50 vrstvami, či `detr_resnet101` se 101 vrstvami.
- `--save_model_dir` určuje, do jaké složky se mají ukládat trénované modely.

Na začátku souboru se zkontroluje, zda jsou validní všechny vstupní parametry a složky. Pokud neexistuje složka pro ukládání modelů, je vytvořena.

Následně program načte soubor s anotacemi do pandas dataframu. Z načtených dat se vytvoří pomocný dataset `anns_marked`, který později určí, v jakém překladu (foldu) bude jaký obrázek. U pomocného datasetu se agregují hodnoty podle počtu anotací. Podle toho počtu se vytvoří K rozvrstvených překladů (Stratified Folds) tak, aby počet anotací na obrázek v jednotlivých překladech byl co nejvíce vyrovnaný. Děje se tomu tak díky kódu (viz ukázka 1).

```
skf = StratifiedKFold(
n_splits=args.folds, shuffle=True,
    random_state=utils.SEED)
# Creating K~Stratified Splits (Folds)
# For stratification use number of all anns (vehicles)
splits = skf.split(
    X=anns_marked.index, y=anns_marked['vehicle_anns'])
```

Výpis kódu 1: Stratified Folds

Do pomocného datasetu se zapíše, jakému překladu odpovídají jaká data, a vypíše se pro každý překlad počet odpovídajících obrázků.

Následně si definujeme operaci převodu tensorů na GPU `device = torch.device('cuda')` pro využití vyšší výpočetní síly. Balíček `torch.cuda` přidává podporu pro typy tenzorů CUDA, které implementují stejnou funkci jako tenzory CPU, ale pro výpočet používají GPU. Definujeme si, jaké části komplexní ztrátové funkce DETRu chceme používat `losses = ['labels', 'boxes', 'cardinality']`, kde `labels` značí porovnávání tříd, `boxes` porovnávání ohraňujících boxů a `cardinality` absolutní chybu počtu předpovězených neprázdných ohraňujících boxů. Podle komentáře na řádku 132 v kódu souboru `detr.py` v projektu DETRu se doopravdy nejedná o míru nějaké ztrátové funkce,

ale o způsob logování a nepropaguje gradienty [37]. Poté podle hodnot ze vstupních parametrů od uživatele přiřadíme jednotlivým ztrátám váhu.

Nyní přichází v kódu na řadu křížová validace podle jednotlivých překladů. Jednotlivé části se budou postupně střídat v roli testovacích dat takovým způsobem, že každá bude v této roli právě jednou. Podle zvoleného ohybu jsou jasně definované trénovací a testovací data. Program umožňuje trénovat i testovat na stejných datech, a proto tato část vypadá následovně (viz ukázka 2).

```
# Cross-validation on K~folds
for fold in range(args.folds):
    # Split dataset ids into train and val. parts by folds
    if args.folds == 0:
        train_ids = anns_marked[anns_marked['fold'] == fold]
    else:
        train_ids = anns_marked[anns_marked['fold'] != fold]
    valid_ids = anns_marked[anns_marked['fold'] == fold]
```

Výpis kódu 2: Rozdělení dat na trénovací a testovací

V každé iteraci vytvoříme novou instanci třídy `DETRModel` se všemi potřebnými parametry. Počet tříd je v kódu ručně určen `NUMBER_OF_CLASSES = 2` (0 značí třídu vozidla a 1 značí no-object třídu) a zbytek je určen vstupními parametry od uživatele. Model zakládáme v každé iteraci z principu křížové validace, aby model netestoval obrázky, na kterých se trénoval. Podle zvolených testovacích a trénovacích dat vytvoříme dvakrát instanci třídy `Dataset` (třída `Dataset` v souboru `visdrone_dataset.py`). Pro zopakování, vstupy této třídy při inicializaci jsou: seznam identifikátorů obrázků datasetu, složka, kde se obrázky nalézají, jejich anotace ve formátu pandas dataframe a funkce zvolená pro transformaci.

Pro oba připravené datasety se vytvoří odpovídající `torch DataLoader`, kam se předává samotný dataset, velikost batche, počet vláken ke zpracování a další parametry. Pokud chce uživatel pro trénovací i testovací dataset použít ta samá data, vytvoří se také dva DataLoadery, avšak oba z těchto stejných dat.

DETR používá speciální třídu pro počítání ztrát, kterou musíme inicializovat. Je to třída `SetCriterion`, která má jako parametry: počet reálných tříd (počet všech tříd - 1 no-object třída), algoritmus přiřazování (matcher), váhy ztrát (`weight_dict`), koeficient no-objekt třídy (`eos_coef=args.null_class_coef`) a jaké části ztrátové funkce chceme použít (`losses`). Jako matcher DETR doporučuje Hungarian Matcher [36]. Jako optimalizer používá DETR AdamW, který má výchozí rychlost učení nastavenou na 0.0001, avšak je určena podle vstupních parametrů uživatele.

Program si drží informaci o dosavadním nejlepším výsledku trénování, tedy

nejmenší hodnotě výstupu ztrátové funkce na validačních datech napříč všemi epochami. Pokud se tato hodnota překoná, je model uložen. V každé epoše dochází k trénování na trénovacím datasetu a k validaci na validačním datasetu. K tomu se používají metody `train_epoch()` a `eval_epoch()`. Hodnoty výstup těchto metod se na konci každé epochy vypíší.

Trénovací funkce `train_epoch()` pro jednu epochu je přímočará. Nejdříve se model a criterion připraví do stavu pro trénování. Následně se inicializuje agregátor hodnot ztrátových funkcí, jak se budou postupně přidávat ztráty z každého batche, aby bylo možné po celé epoše zjistit jejich průměrnou hodnotu. Po jednotlivých batch-ích zvolené velikosti se projde celý trénovací dataset, kde se pro každý z nich zjistí predikce modelu: `output = model(images)`. V tomto bodě jsou hodnoty pixelů obrázků normalizované, a tak mají hodnotu mezi 0 a 1. Predikce `output` a požadované hodnoty z anotací `targets` se porovnají v `criterion`, který vrátí hodnoty jednotlivých ztrátových funkcí. Tyto hodnoty se následně vynásobí odpovídajícími váhami. Při používání frameworku PyTorch je nutné resetovat na nulu gradienty všech parametrů modelu ještě před zavoláním zpětné propagace. To se dělá z důvodu, protože PyTorch akumuluje gradienty a není v našem zájmu je mixovat mezi jednotlivými batchi. Proto až po tomto resetování `optimizer.zero_grad()` dochází ke zpětné propagaci `losses.backward()` a `optimizer.step()` pro aktualizaci parametrů na základě aktuálního gradientu. Ztráty se přidají do agregátoru a aktualizuje se ukazatel průběhu. Evaluační funkce `eval_epoch()` je téměř stejná jako testovací. Nedochází zde však ke zpětné propagaci a operacím s ní spojenými, pouze ke zjištění hodnot ztrátových funkcí a jejich agregaci.

Při spuštění programu může začínat jeho výstup například následovně (viz ukázka 3)

7.6 Detekce

Pro využití natrénovaného modelu pro detekci vozidel z obrázků můžeme použít soubory `test_detector.py` a `image_test.py`. První z těchto souborů (test_detector.py) se používá k evaluaci zvoleného detektoru na obrázcích z určité složky. Vstupní parametry tohoto programu jsou:

- `--img_dir` určuje složku, ze které se budou načítat obrázky k evaluaci.
- `--save_img_dir` určuje, kam se uloží obrázky obohacené o nakreslené ohraňující boxy predikce a ty opravdové. Pokud tato složka neexistuje, vytvoří se nová, pokud to jde.
- `--anns_file` určuje cestu a název souboru s anotacemi k obrázkům z minulého parametru. Musí odpovídat formátu `image_id, width, height, cat, x, y, w, h`, a proto doporučuji použít k jejich vygenerování skript `convert_annotations.py`.

```
Exists anns_file =  ./dataset/train.csv
Exists img_dir =  ./dataset/images
Exists save_model_dir =  ./models
Using cache found in /.../torch/hub/facebookresearch_detr_master
Fold 0     have 1369 items.
Fold 1     have 1369 items.
Fold 2     have 1369 items.
Fold 3     have 1368 items.
Fold 4     have 1368 items.
Dataset have 5474 images
Dataset have 1369 images
100%|||||||| 5474/5474 [17:40<00:00,  5.16it/s, loss=1.89]
100%|||||||| 1369/1369 [01:52<00:00, 12.13it/s, loss=2.05]
Time        = 02.08.2020 18:45:45
Fold        = 0
Epoch       = 0
Train loss  = 1.89428
Valid loss  = 2.05137
Saving: model_ID-0_Fold-0_Epoch-0_TL-1.894_VL-2.051.pth
```

Výpis kódu 3: Ukázka možného začátku výstupu programu `train.py`

- `--min_prob` určuje minimální pravděpodobnost pro vykreslení predikovaného ohraňujícího boxu.
- `--max_output_images` určuje maximální možný počet nových obrázků uložených do výstupní složky. Pořadí se bere podle názvu souboru.
- `--model_file` určuje cestu k modelu, který bude využit k predikci.

Tento program načítá soubor s anotacemi, připravuje `device`, Hungarian `matcher` a `criterion`, stejně jako v programu pro trénování modelu. Velikost batche je přímo v kódu nastavená na hodnotu 1, stejně jako koeficient no-object třídy, který má hodnotu 0.5. Protože máme obrázky a k nim anotace s ohraňujícími boxy a třídami, můžeme použít `dataset` instanci třídy `vdd.Dataset` a `data_loader` instanci `DataLoader`. Program načte model podle cesty ze vstupu a začne postupně iterovat přes po sobě jdoucí batche a jejich prvky. Protože nevíme, jak jsou obrázky velké, měníme jejich velikost, a proto si musíme zapamatovat jejich původní dimenzi. Přístup k anotovaným ohraňujícím boxům je k dispozici hned ze vstupního souboru. Hned po založení nového obrázku pro výstup, který je zatím stejný jako vstup, můžeme začít s vykreslováním ohraňujících boxů. Jakmile jsou ohraňující boxy vykreslené, model začne predikovat výsledek. Z predikovaných výsledků a opravdových výsledků jsou pomocí kódu z ukázky 4 získány hodnoty ztrát.

```
with torch.no_grad():
    outputs = model(images)

# Get losses
loss_dict = criterion(outputs, targets)
weight_dict = criterion.weight_dict
losses = sum(loss_dict[k] * weight_dict[k]
             for k in loss_dict.keys() if k in weight_dict)

# Update average validation loss
average_loss.update(losses.item(), BATCH_SIZE)
print(f'Average loss is {average_loss.avg:4f}')
```

Výpis kódu 4: Ukázka počítání průměrných ztrát

Ztráty se uchovávají a průměrují. Tento průměr se vypíše po každém predikovaném batchi, který je v tomto programu 1, takže po každém obrázku. Následně se na výstupní obrázek vykreslí predikované boxy, které se ovšem musí ještě škálovat podle poměru změny velikosti na začátku. Výsledný obrázek se následně uloží a program o tom informuje uživatele. Aby se nepřekročil maximální počet uložených obrázků z parametru od uživatele, kontroluje se zde na konci programu i tato podmínka. Následující text 5 a obrázek [obrazek tested2 nějaky pekny] je ukázka výstupu programu.

```
Exists anns_file =  ./dataset/train.csv
Exists model_file =  ./models/detr_best.pth
Exists img_dir =  ./dataset/images
Exists save_img_dir =  ./images/tested
Loading model from imput./models/detr_best.pth.
Using cache found in /.../torch/hub/facebookresearch_detr_master
Dataset have 6843 images
Number of batches is  6843
Average loss is 2.839127
Saving:  ./images/tested2/0000001_02999_d_0000005.jpg
```

Výpis kódu 5: Ukázka možného začátku výstupu `test_detector.py`, kde se uložil jeden obrázek.

[obrazek tested2 nějaky pekny]

Na obrázky si můžeme všimnout bílých a zelených obdélníků. Ty bíle značí ohraňující boxy, které jsou uvedené v anotacích, a ty zelené značí ohraňující boxy predikované modelem.

7.7 Sledování

Přirozeným vývojem při implementaci bylo po detekování objektů na jednom obrázku, přejít na detekování a sledování objektů na sekvenci obrázků. Tím se zabývá program `sequence_test.py`. Umožňuje postupné čtení sekvence obrázků, na kterých pomocí vlastního či před-trénovaného modelu detekuje objekty, které napříč snímky sleduje. Ke sledování se v implementaci dá použít jeden ze dvou trackerů SORT a Centroid Tracker. Při sledování se vykreslují statistiky přímo na výstupní obrázek a před ukončením programu se vypíší ještě další statistiky. Při spuštění programu je možné měnit následující parametry:

- `--images_dir` určuje složku se sekvencí obrázků. V této složce se zkontrolují všechny soubory při pokusu o načtení všech obrázků sekvence. Je tedy důležité, aby složka obsahovala pouze obrázky sekvence.
- `--save_images_dir` určuje, do jaké složky se mají výstupní obrázky ukládat. Pokud tato složka neexistuje, vytvoří program novou.
- `--anns_file` určuje soubor s anotacemi obrázků sekvence. Na rozdíl od jiných parametrů `--anns_file` z ostatních programů implementace se zde nevyužívá předpřipravených anotací pomocí skriptu `convert_annotations.py`, protože mají jiný formát. Formát, který se zde vyžaduje, je stejný jako z VisDrone MOT datasetu a jeho tvar je: `id snímku sekvence, id objektu, x-ová souřadnice levého horního bodu ohraňujícího boxu, y-ová souřadnice levého horního bodu ohraňujícího boxu, jeho šířka, výška, skóre detekce, kategorii, truncation a occlusion`.
- `--model_file` určuje cestu k modelu, který bude využit k predikci.
- `--use_pretrained_model` určuje pomocí booleovských hodnot Trua a False, zda chceme použít vlastní model či před-trénovaný.
- `--pretrained_model` pokud jsme v minulém parametru určili, že chceme použít před-trénovaný model, zde se dá zvolit, jaký konkrétně ze zoo modelů chceme použít.
- `--tracker` určuje použitý tracker. Na výběr má uživatel ze dvou trackerů (Sort, značící se `sort` a Centroid Tracker, značící se `centroid`). Pro správné použití trackeru je potřeba napsat jedno z těchto dvou značení.
- `--line` definuje řetězec složený ze čtveřice celých čísel oddělených čárkou bez mezery. Tyto hodnoty jsou postupně souřadnice prvního a druhého bodu, který tvoří linku. V dalším a posledním parametru se následně určí, jestli se tato linka použije.
- `--useline` určuje, zda chceme použít měřící linku, či nikoliv.

Program nejdříve zkontroluje vstupní parametry souborů a složek, pokud není vytvořená složka pro zápis výstupních obrázků, je programem vytvořena. Určení parametru linky se provádí test, zda je dobře zapsaný a obsahuje čtyři čísla (dva body).

Program převádí jména obrázků sekvence ze složky do celých cest a načte soubor s anotacemi ve zmíněném formátu. Podle toho, jaký model chce uživatel zvolit, tento model načteme. U před-trénovaných modelů ze zoo dochází k predikci ohraňujících boxů ve formátu souřadnic jejich středu, šířky a výšky.

Po zvolení modelu se podle parametrů volí tracker. Jak je vidět z úryvku kódu 6 , oba trackery mají ladící parametry, které jsme si již popsali v podkapitole se sledováním objektů.

```
# Choose tracker depending on user input [sort, centroid]
if args.tracker == 'sort':
    tracker = Sort(max_age=20, min_hits=3, iou_threshold=0.25)
else:
    tracker = CentroidTracker(maxDisappeared=15, maxDistance=45)
```

Výpis kódu 6: Výběr a inicializace trackerů

Pro zaznamenávání informací, ze kterých se budou vyvozovat statistiky, disponuje kód tohoto programu třídou `Stats`. Tato třída si zaznamenává k jednotlivým id objektů, jejich po sobě jdoucí centroidy. Posloupnost těchto centroidů se pak používá k vypisování některých statistik. Metodou zaručující správnou funkčnost třídy Stats je metoda `update()`. Ta přijímá list trojic (x, y, id) objektů, kde x a y značí souřadnice centra ohraňujícího boxu detekovaného v tomto snímku. Pokud již o objektu s tímto identifikátorem existuje záznam, přidá se jeho centroid nakonec. Pokud záznam ještě neexistuje, přidá se pod uvedeným identifikátorem centroid jako první záznam daného objektu. Metoda `getDistanceAndCount()` pro každý objekt vedený ve třídě `Stats` spočítá vzdálenost ujetou v pixelech od počátečního snímku až do toho aktuálního (posledního). Slovník s uloženými centroidy lze získat pomocí metody `getPaths()`.

Nyní začne program postupně procházet snímky. Tento snímek si načte a rovnou vytvoří výstupní kopii, která se bude upravovat, jak je vidět v ukázce kódu 7. K tomu dochází z důvodu, aby se vstupní obrázek neměnil a bylo možní ho použít opakovaně.

Vstupní obrázek se převede na tensor a nechá se modelem zpracovat. Jelikož DETR používá třídu žádného objektu (no-object class), je vhodné detekce s touto třídou nebrat v úvahu a odstranit je. Odstranění se liší podle zvoleného modelu (viz ukázka 8). Pokud se jedná o model před-trénovaný ze zoo, není v zájmu práce detekovat i jiné objekty než vozidla, a tak se nechají pouze objekty s názvem tříd, které jsou definované proměnné `ALLOWED_COCO_CLASSES` a tím se odstraní i no-object třída, které se předtím nastaví název třídy na

```
# Load image
image = cv2.imread(img_path)
height, width, channels = image.shape

# Prepare output image
output_image = image.copy()
output_name = os.path.join(args.save_images_dir, img_name)
```

Výpis kódu 7: Načtení obrázku a příprava výstupního obrázku

"no-class". U vlastního modelu je postup podobný, neboť model predikuje pouze dvě třídy 0 a 1, ponecháme třídu 0 značící vozidlo.

```
# Filter out non-vehicle classes
vehicle_boxes = []
if args.use_pretrained_model:
    for box, scores in zip(oboxes, all_scores):
        label_id, value = utils.get_max_id_and_max(scores)
        if label_id >= len(classes.CLASSES_COCO):
            label = 'no-class'  # no-class object
        else:
            label = classes.CLASSES_COCO[label_id]
        if label in ALLOWED_COCO_CLASSES:
            vehicle_boxes.append(box)
else:
    # This model has only 2 classes, where 0 is for vehicles
    for box, scores in zip(oboxes, all_scores):
        label_id, value = utils.get_max_id_and_max(scores)
        if label_id == 0:
            vehicle_boxes.append(box)
```

Výpis kódu 8: Část kódu s odstraněním predikcí objektů, které patří do no-class třídy

DETR často přisuzoval k jednomu objektu více podobných ohraňujících boxů (někdy i stejných), a tak jsem vytvořil heuristiku, která na základě podobnosti OIU promaže predikované ohraňující boxy. Volání této heuristiky je vidět v následující ukázce kódu 9.

```
# Delete detections on the same obj - heuristic
unique_vihicle_boxes, removed =\\
    utils.remove_similar(vehicle_boxes, iou_trh=0.5)
```

Výpis kódu 9: Volání funkce pro odstranění podobných ohraňujících boxů

Tato úprava se provádí na každém snímku a může být zajímavé sledovat, kolik objektů odstraňuje. Proto se vždy vypíší tři řádky, kde první značí počet všech predikovaných boxů (měl by odpovídat parametru `num_of_queries` v souboru `train.py` při učení modelu). Druhý řádek vypisuje počet detekovaných vozidel a třetí řádek vypisuje počet unikátních detekovaných vozidel po odstranění duplicit.

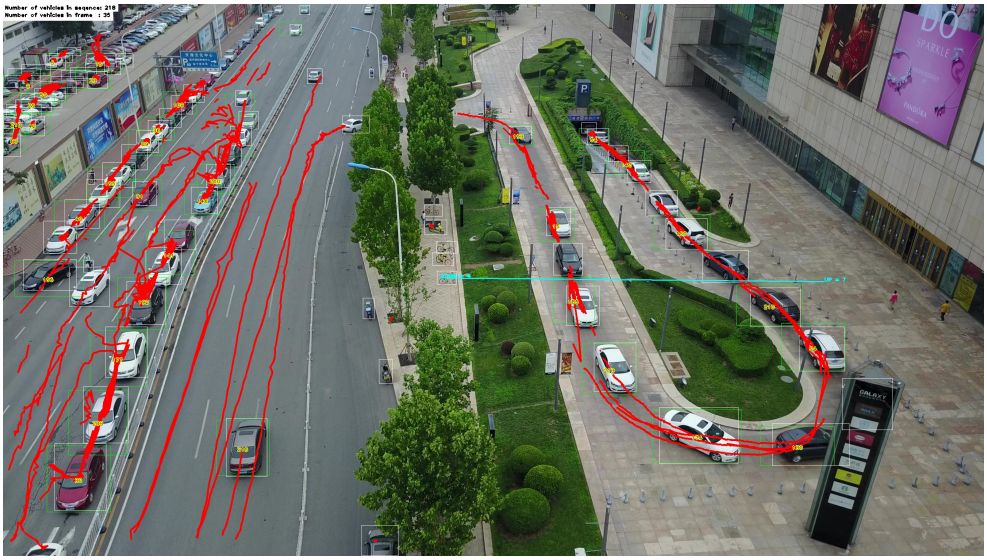
Nyní přichází na řadu sledování. Program je napsaný tak, aby se dal střídat SORT tracker a Centroid Tracker. Data predikovaných objektů se tak převádí do struktur, kterým daný tracker rozumí, a výstup z trackeru se opět převádí do struktur, kterým rozumí třída `Stats`. Pro SORT je potřeba vytvořit list listů pětic ve formátu x min, y min, x max, y max, scóre. Převod do této struktury zařídí funkce `utils.transformToSORT()`. Výstup ze SORTu je ve formátu `x min, y min, x max, y max, id objektu`, ovšem jak SORT uvádí, počet a hodnoty se mohou lišit, což může vyústit v nespolehlivé chování. Výstup je převeden na slovník, kde jako klíče jsou identifikátory objektů a jako hodnoty jsou dvojice určující souřadnice středu ohraňujícího boxu. Centroid tracker vrací výsledek přímo tuto strukturu, a tak není potřeba převod. K tomu, aby se boxy méně překrývaly, a tím pádem lépe sledovaly, jsem implementoval další heuristiku, která ohraňující boxy ořízne ze všech stran o nějakou velikost. Tato velikost se udává relativně k dimenzím ohraňujícího boxu, neboť každý může být jinak velký. To má na starosti funkce `utils.shrink_booxes()`

Nyní, když už máme identifikované objektu (vozidla) napříč snímky od počátečního až po aktuální, můžeme vypisovat statistiky. Program vykresluje přímo do obrázku statistiky pro **počet unikátních vozidel od počátečního do aktuálního snímku. Počet unikátních vozidel v aktuálním snímku**. Tento počet zahrnuje i vozidla, které nejsou zrovna detekovaná, ale zvolený tracker je hodlá sledovat ještě určitý počet snímků, než je smaže. Dále také vykresluje jejich **trajektorie** a na konci programu vypíše jejich **ujetou vzdálenost v pixelech, počet snímků, na kterých byli identifikované a jejich průměrnou rychlost v pixelech za snímek**. K tomu se používají výstupu ze třídy `Stats`. V této třídě se při volání metody `update()` kontroluje, zda se starý a nový centroid objektu nevyskytuje na odlišné straně linky, pokud je definována. Pokud se linka v nějakém směru překročí, je zvýšen adekvátní čítač. Tato linka se v každém snímku vykreslí spolu s oběma čítači světle modrou barvou. To, k jakému čítači (up, down) se přičte jednička, je individuální pro každou sledovanou oblast, ze které je sekvence obrázků či video. To znamená, že doporučuji si přímo v programu určit, kam se má jednička přičíst. V základu k porovnání používám pozici y centroidů. Jako poslední program vykreslí identifikátory jednotlivých objektů. Tomu se děje až nyní z důvodu, aby byly identifikátory dobře čitelné (viz ukázka kódu 10).

Pro každý vstupní snímek sekvence se tedy uloží jeden s nakreslenými ohraňujícími boxy, linkou, čítačemi, trajektoriemi, anotovanými ohraňujícími boxy a všemi zmíněnými statistikami. Program nakonec vypíše statistiky vzdálenosti a rychlosti. Následující obrázek 7.2 zobrazuje výstup programu.

```
# Draw IDs last so we can see them
for key, value in tracker_output.items():
    utils.draw_centroid_and_id(
        output_image, value[0], value[1], str(key))
```

Výpis kódu 10: Ukázka volání funkce pro vykreslování ID centroidů



Obrázek 7.2: Ukázka výstupního obrázku ze sekvence

Následující text 11 zobrazuje část možného výstupu programu.

Soubor s názvem `viedo_test.py` dělá prakticky to samém jako soubor `sekvence_test.py`, avšak nepoužívá sekvenci obrázků, ale video. Z videa si postupně načítá jednotlivé snímky, které upravuje a následně ukládá do nového videa. Důležitou změnou v programu je změna velikosti videa, a tak i škálování linky, ohraňujících boxů a ostatních proměnných závislých na relativní šířce a výšce obrázku. Program má některé vstupní parametry stejné jako ten předchozí, jsou to: `--model_file`, `--use_pretrained_model`, dále také `--pretrained_model`, `--tracker`, `--line` a `--useline`. Nově použitými parametry jsou:

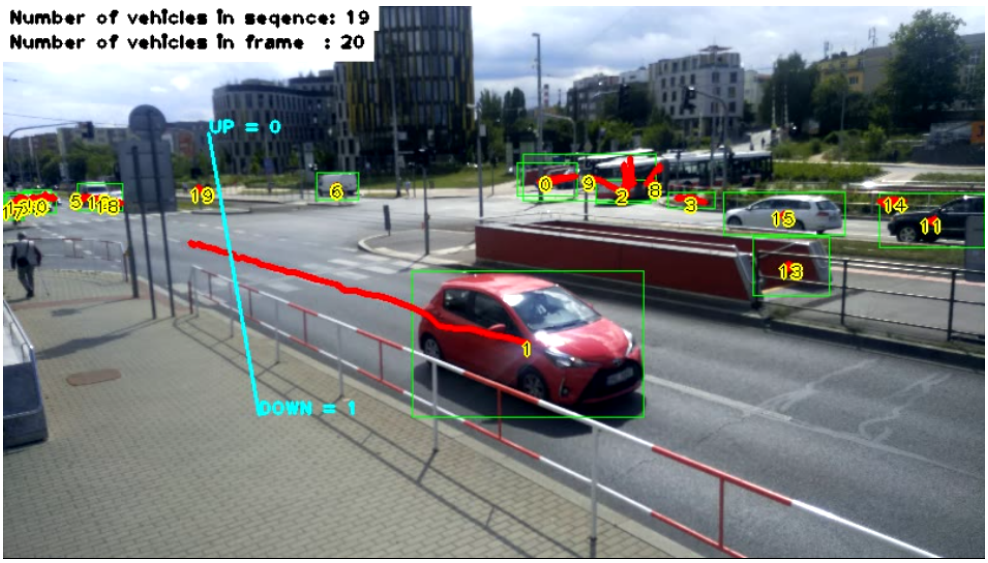
- `--length_f` určující počet snímků videa použitých ve výstupním videu.
- `--max_frame_size` určuje maximální možnou šířku nebo výšku snímku videa, na kterou se škáluje video.
- `--video_file` určuje soubor s video záznamem.

```
Vehicle with id [   0] traveled     95 pixels in 58 frames.
Its avarage speed is:   1.638 pixels per frame.
Vehicle with id [   1] traveled     66 pixels in 58 frames.
Its avarage speed is:   1.138 pixels per frame.
Vehicle with id [   2] traveled     92 pixels in 58 frames.
Its avarage speed is:   1.586 pixels per frame.
Vehicle with id [   3] traveled     41 pixels in 58 frames.
Its avarage speed is:   0.707 pixels per frame.
Vehicle with id [   4] traveled    202 pixels in 58 frames.
Its avarage speed is:   3.483 pixels per frame.
Vehicle with id [   5] traveled     48 pixels in 58 frames.
Its avarage speed is:   0.828 pixels per frame.
Process finished with exit code 0
```

Výpis kódu 11: Ukázka části textového výstupu z programu `sekvence_test.py`

- `--save_video_dir` určuje složku, kam se má zapsat výstupní video obohacené o statistiky a detekování.

Na následujícím obrázku 7.3 je vidět ukázka snímku z videa.



Obrázek 7.3: Ukázka snímku z výstupního videa.

Kapitola 8

Experimenty

Při popisu jednotlivých částí programu jsme si uvedli mnoho různých parametrů, které se dají při trénování modelu či testování detekce a sledování měnit. Pro trénování mezi ně patří počet výstupů DETRu (`num_of_queries`), míra učení (`learning_rate`), počet překladů (`folds`), počet epoch, velikost jednoho batche, koeficient třídy no-object (`null_class_coef`) a jednotlivé váhy pro ztrátovou funkci (`loss_ce`, `loss_bbox` a `loss_giou`). Také se můžeme zaměřit na obsah datasetu, jestli obsahuje pouze vozidla, nebo i třídy bez objektu. To, jaký před-trénovaný model použijeme pro trénování, má také vliv na jeho výsledek. U sledování objektů a statistik můžeme porovnávat mezi trackery SORT a Centroid Trackerem a pro oba zkoušet různé parametry. Při měření výsledků experimentů je důležité zafixovat všechny parametry kromě těch, které testujeme. Nyní si pro některé ze zmíněných parametrů (rozdílů) uvedeme experimenty, které by měly objasnit rozdíly v jejich použití.

1. Experiment: Porovnání velikosti výstupů ztrátových funkcí pro měnící se nastavení míry učení pro Transformer v DETRu (využívající AdamW), při stabilizovaných ostatních parametrů. Jedná se o reálnou hodnotu mezi 0 a 1 a obvykle se pohybuje kolem hodnoty $0.0001 = 1 \cdot 10^{-4}$. Hodnoty, které budeme sledovat, jsou (0.00001, 0.0001, 0.0005 a 0.001). Výsledky by měly odhalit lepší a horší hodnoty parametru, aby proces učení zdárně konvergoval.

   Fixované parametry:

   - Počet výstupů DETRu = 100
   - Velikost batche = 4
   - Počet ohybů = 5
   - Počet epoch = 30
   - Koeficient třídy no-object = 1

- Váhy pro ztrátovou funkci (`loss_ce`, `loss_bbox` a `loss_giou`) všechny = 1
- Trénované pouze na datasetu s anotacemi vozidel.

2. Experiment: O koeficientu třídy no-object je v dokumentaci DETRu velice málo informací. Je to proto ideální kandidát na testování. Jeho hodnota může být nejspíše jakákoliv kladná hodnota, logicky větší než 0. Kód DETRu používá hodnotu 0.1. Nabízí se otestovat trénování pro hodnoty [0.1 a 0.5]

   Fixované parametry:

   - Počet výstupů DETRu = 100
   - Velikost batche = 4
   - Počet ohybů = 5
   - Počet epoch = 30
   - Míra učení = 0.0001
   - Váhy pro ztrátovou funkci (`loss_ce`, `loss_bbox` a `loss_giou`) všechny = 2
   - Trénované pouze na datasetu s anotacemi vozidel.

3. Experiment: Porovnání SORT a Centroid Trackeru na stejném videu pomocí vizuálního pozorování výstupních sekvencí obrázků a podle vypsaných statistik. K tomuto experimentu použiji před-trénovaný model `detr_resnet101` stažený ze zoo, detekce by tak měly být kvalitní, protože byl model trénován více než 6 dní, na tisících obrázcích. Tracker SORT má parametry `max_age=20, min_hits=2, iou_threshold=0.25` a Centroid Tracker má parametry `maxDisappeared=15, maxDistance=45`.

4. Experiment: Porovnání různých hodnot jednoho z parametrů Centroid Trackeru na stejném videu pomocí vizuálního pozorování výstupních sekvencí obrázků a podle vypsaných statistik. Stejně jako v minulém experimentu použiji před-trénovaný model `detr_resnet101` stažený ze zoo, aby se do výsledků nepromítaly špatné detekce díky málo naučenému modelu. Centroid tracker má dva počáteční parametry `maxDisappeared`, `maxDistance`, budeme sledovat sekvenci obrázků vždy s rozlišnými hodnotami parametr `maxDisappeared` jednou s hodnotou nízkou (5) a jednou s hodnotou vysokou (50) a uvedeme si, jaký je jejich reálný vliv na kvalitní sledování vozidel.

8.1 Výsledky

Následuje číslovaný seznam výsledků experimentů, kde čílo výsledku odpovídá číslu experimentu.

1. Výsledek: Program pro trénování jsem si upravil tak, aby si pamatoval průměrné hodnoty ztráty v každé epoše. Tyto hodnoty následně průměruje napříč foldy. Při opakovaném spouštění programu s mírou učení rovnou 0.001 DETR padal. Jak jsem se dozvěděl z fóra, tato míra učení je příliš vysoká [57]. Dostáváme tedy následující tabulky 8.1 a tři jim odpovídající grafy 8.1.

Míra učení	0.00001		0.0001		0.0005	
Epocha	Trénovací ztráta	Validační ztráta	Trénovací ztráta	Validační ztráta	Trénovací ztráta	Validační ztráta
1	1,9007	2,1084	1,9004	2,1002	1,8974	2,0984
2	1,8525	1,9525	1,8726	1,9721	1,8571	1,9286
3	1,8297	1,9225	1,8195	1,9302	1,8353	1,9103
4	1,7878	1,8523	1,8429	1,8775	1,7786	1,8732
5	1,7626	1,8034	1,7951	1,8266	1,7802	1,8299
6	1,7469	1,8126	1,7764	1,8157	1,7585	1,8153
7	1,7602	1,8057	1,7519	1,8012	1,7556	1,8082
8	1,7354	1,7826	1,7434	1,7807	1,7231	1,7673
9	1,7242	1,7951	1,7394	1,7711	1,7137	1,7614
10	1,7156	1,7795	1,7065	1,7625	1,7277	1,7652
11	1,7312	1,7821	1,7164	1,7711	1,7223	1,7625
12	1,7239	1,7811	1,7109	1,7801	1,7104	1,7411
13	1,7147	1,7564	1,6915	1,7624	1,6969	1,7356
14	1,7091	1,7426	1,7209	1,7394	1,6965	1,7233
15	1,6963	1,7515	1,6795	1,7422	1,6799	1,7199
16	1,6923	1,7612	1,6931	1,7404	1,6963	1,7189
17	1,6829	1,7366	1,6924	1,7375	1,6803	1,7156
18	1,6793	1,7355	1,6999	1,7361	1,6724	1,7059
19	1,6844	1,7425	1,6921	1,7312	1,6989	1,7207
20	1,6624	1,7399	1,7108	1,7399	1,6795	1,7120
21	1,6674	1,7422	1,6925	1,7436	1,6691	1,6982
22	1,6699	1,7315	1,6699	1,7255	1,6594	1,6983
23	1,6822	1,7219	1,6558	1,7211	1,6735	1,7064
24	1,6723	1,7248	1,6666	1,7108	1,6729	1,6987
25	1,6694	1,7291	1,6648	1,7121	1,6711	1,7023
26	1,6525	1,7159	1,6677	1,7109	1,6517	1,6984
27	1,6527	1,7042	1,6572	1,7002	1,6505	1,6954
28	1,6533	1,7154	1,6589	1,7008	1,6614	1,6946
29	1,6629	1,7193	1,6631	1,7053	1,6600	1,6834
30	1,6511	1,7005	1,6509	1,7014	1,6591	1,6914

Tabulka 8.1: Tabulka výsledných hodnot 1. experimentu

Z grafu je vidět, že postupně dochází ke zmenšování chyby, avšak velikost

chyby je stále velmi velká, přesto začínají křivku konvergovat ke stálé hodnotě. To naznačuje, že nedochází k dobrému učení. Křivky všech tří grafů si přibližně odpovídají, a tak se zdá, že míra učení pro AdamW nemá (krom spadnutí procesu učení) výrazný vliv na model. To je však zvláštní, neboť by tato hodnota měla efektivně měnit tvar křivky. Možná je to zapříčiněno malým počtem epoch.

2. Výsledek: Hodnoty jsem měřil podobně jako v předchozím experiment, opět přes průměrování hodnot ztrát v jednotlivých překladech a epochách. Můžeme si všimnout, že hodnoty jsou vyšší, to je však zapříčiněno dvojnásobnými váhami prvků ztrátové funkce, které jsou však zafixované a v průběhu se nemění. Po rozšíření svislé osy grafu je vidět pokles ztrát napříč epochami. Hodnoty grafu 8.2 se pohybují mezi 4 až 1.5, jak je patrné z tabulky 8.2.

   U křivek se zdá, že konvergují ke stálé hodnotě. Ovšem po spuštění modelu po 30 epochách nebyl výsledek detekce zdaleka tak dobrý jako u před-trénovaného modelu bez smazaných vah jedné vrstvy. Je tedy možné, že při trénování modelu dochází k nějakým nepřesnostem, díky kterým se model neučí tak dobře. Jak jsem se dočetl na fóru, někteří uživatelé používající DETR mají podobné problémy.

3. Výsledek: Jako výsledek tohoto experimentu přikládám následující dva obrázky 8.3 a 8.4, které jsou ze stejné sekvence. Nutno podotknout, že závěry z tohoto experimentu jsou vytvořené z celých generovaných sekvencí obrázků a dva uvedené obrázky tak slouží pouze k rychlému porovnání pro čtenáře.

   Z porovnání obrázků je vidět, že SORT má lepší výsledky při trackování, pokud jsou nastavené parametry tak, jak byly definované v úvodu experimentu. Na trajektoriích je znát, že jsou mnohem rovnější, a tak vytvářejí jasnější informaci o reálném pohybu vozidel po vozovce ze sekvence obrázků. U Centroid Trackeru se vyskytují výpadky objektů, častější přeskakování identifikace mezi blízkými objekty a některé objekty se po reálném objevení a detekování obrázku začnou až pozdě sledovat. To například zapříčinilo, že čítače se liší hodnoty čítačů linek v porovnávaných sekvencích. Oba obrázky jsou v pořadí 150. ze sekvence a pro SORT ukazuje linka (DOWN=8, UP=6) a u Centroid Trackeru ukazuje linka (DOWN=8, UP=4), tedy dvě vozidla z celkových šesti Centroid Tracker nezačal sledovat včas. To bude zapříčiněnou tím, že při vjezdu vozidla do zabírané oblasti je jeho rychlost příliš vysoká a pak urazí určitou vzdálenost, které je větší než nastavená hranice u Centroid Trackeru a pak tyto vozidla tracker neasociuje.

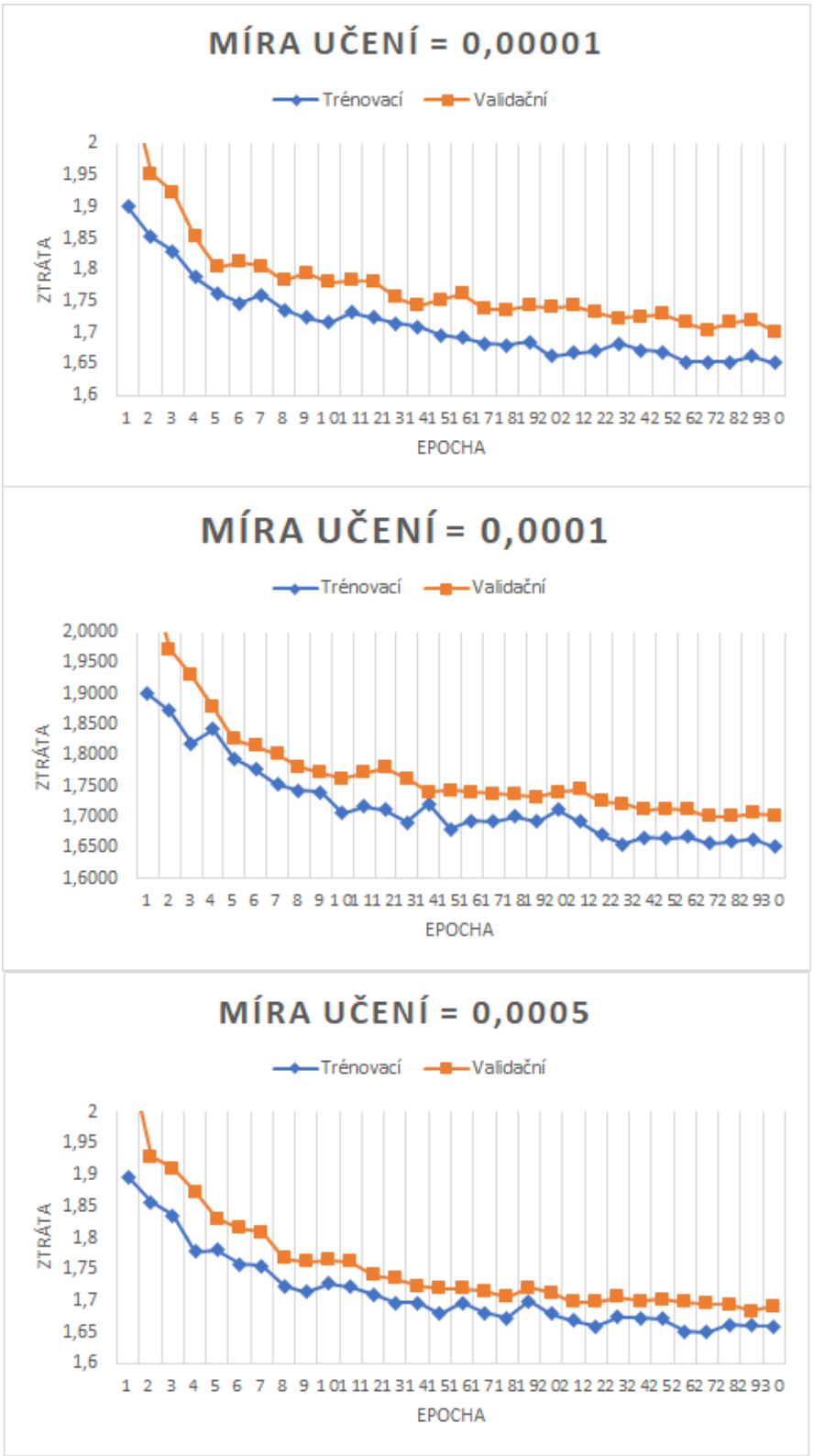
   Jak naznačují výstupní data obou běhů programu, SORT detekoval 247 objektů, zatímco Centroid Tracker pouze 118. To však nemusí znamenat,

Koeficient třídy no-object	Koeficient = 0,1		Koeficient = 0,5	
Epocha	Trénovací ztráta	Validační ztráta	Trénovací ztráta	Validační ztráta
1	3,5563	3,7936	3,5673	3,7836
2	3,3022	3,3822	3,3294	3,4036
3	2,9965	3,2344	2,9831	3,3297
4	2,7911	3,1495	2,7212	3,1548
5	2,5188	3,1493	2,6064	3,1269
6	2,4401	2,8492	2,3451	2,7147
7	2,3921	2,6581	2,4512	2,7314
8	2,3806	2,4743	2,3949	2,5297
9	2,2497	2,3971	2,2763	2,4195
10	2,2088	2,3558	2,2847	2,3794
11	2,2007	2,2717	2,1855	2,3961
12	2,0031	2,2153	2,0243	2,2036
13	2,0195	2,1728	2,0094	2,2021
14	2,0001	2,1947	2,0099	2,2012
15	2,1097	2,2022	2,2125	2,2369
16	2,0073	2,2187	2,2048	2,2492
17	2,1453	2,1913	2,1607	2,2596
18	2,1072	2,2157	2,1194	2,3019
19	2,0806	2,1898	2,0403	2,2149
20	2,0174	2,2074	2,0394	2,2915
21	2,0001	2,1939	2,0101	2,3218
22	2,1333	2,1842	2,0133	2,1535
23	2,1024	2,1999	2,0027	2,1423
24	1,9441	2,2432	1,9527	2,1535
25	1,9228	2,3101	1,9475	2,2862
26	1,9033	2,2447	1,9143	2,2936
27	1,9979	2,2149	1,9024	2,2474
28	1,8774	2,2108	1,9196	2,1903
29	1,8301	2,1663	1,9102	2,1632
30	1,8002	2,1563	1,9089	2,1541

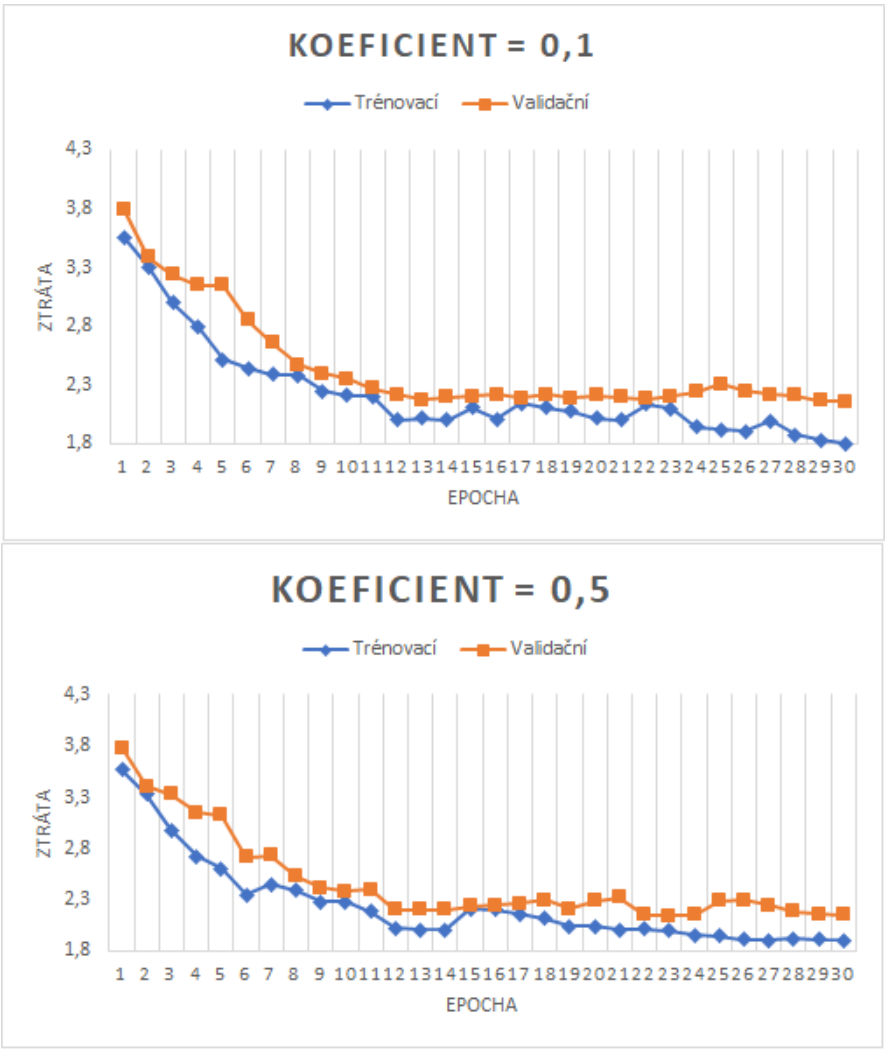
Tabulka 8.2: Tabulka výsledných hodnot 2. experimentu

že Centroid Tracker je v tomto ohledu horší, neboť ve výstupu SORTu má poměrně značná část o velikosti 21 prvků ujetou vzdálenost v pixelech za celých 150 snímků menší než 50 pixelů. Jedná se tak o špatně trackované objektu, které zapomněli svůj identifikátor. Závěr tohoto experimentu je, že oba trackery mají svoje silné a slabé stránky, a tak záleží na konkrétní sekvenci či videu a zvolených parametrech trackeru.

4. Výsledek: Při tomto experimentu uvedu dva snímky z videa, na kterých je jasně vidět rozdíl při nízké a vysoké hodnotě parametru. Toto video bylo staženo ze zdroje [58] a představuje statický pohled na ulici, kde blízko před kamerou jezdí vozidla, a tak po určitý počet snímků nejsou vidět ostatní vozidla. Na prvním obrázku 8.6 je po přejetí automobilu vidět, že zakrytá vozidla si stále drží svůj identifikátor. Přestože zrovna zakrytá vozidla nejsou vidět, tak druhý čítač v levém horním rohu nám dává informaci, že na snímku je stále 18 vozidel. Oproti tomu na druhém obrázku 8.5 je hodnota tohoto čítače pouze 3, to znamená, že za průjezd automobilu blízko před kamerou stihl Centroid tracker zapomenout zakrytá vozidla. V tomto případě po přejetí auta před kamerou byla již jednou rozpoznaná a identifikovaná vozidla znovu identifikovaná a to je chybné.



Obrázek 8.1: Graf experimentu 1

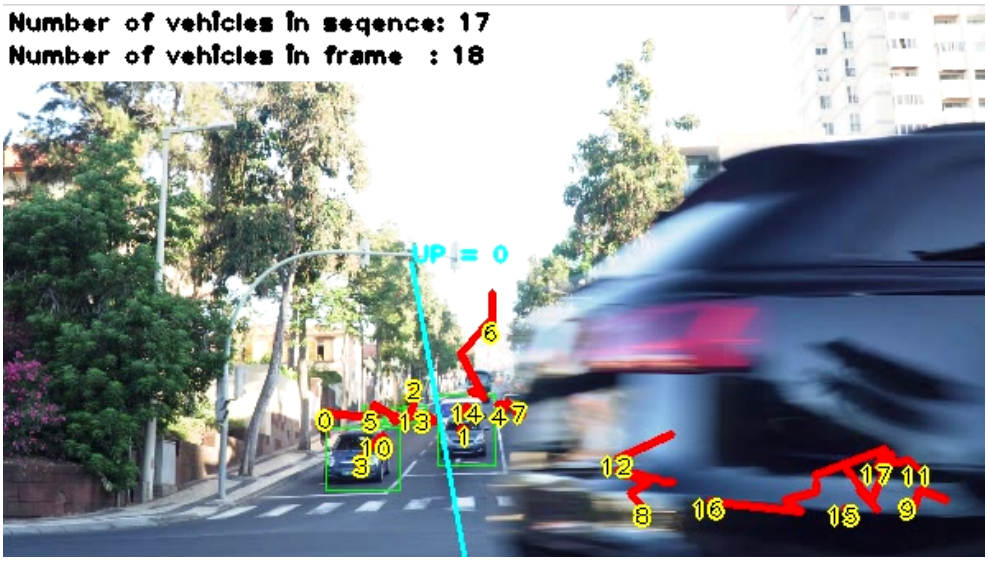


Obrázek 8.2: Graf experimentu 2

Obrázek 8.3: 150. snímek z sekvence při použití trackeru SORT

Obrázek 8.4: 150. snímek z sekvence při použití Centroid Trackeru

Obrázek 8.5: Ukázka snímku z videa, při nízkém parametru `maxDisappeared`



Obrázek 8.6: Ukázka snímku z videa, při nízkém parametru `maxDisappeared`

Závěr

V této práci jsem se zabýval analýzou existujících řešení pro problematiku detekce a sledování objektů z video záznamů se zaměřením na objekty vozidel. Provedl jsem důkladnou analýzu základů strojového učení, neuronových sítí, komplexních modelů strojového vidění a řady algoritmů, které využívají. Na základě této analýzy jsem navrhl implementaci vlastní aplikace na rozpoznávání a sledování vozidel s použitím zmíněných neuronových sítí, modelů a algoritmů počítačového vidění. Struktura implementace se skládá z části pro detekci objektů a z části pro jejich sledování. Tato aplikace používá pro detekci vozidel nový model DETR vyvinutý týmem lidí ve Facebook AI Research. Pro sledování vozidel využívá dokonce dvou trackerů a to SORT a Centroid Tracker. Práce prozkoumává možnosti výstupních statistiky a některé z nich implementuje v aplikaci. K trénování a evaluaci využívá datasety VisDrone OD a VisDrone MOT, avšak zvládá zpracovat i videa natočené běžnou kamerou. Aplikace s využitím před-trénovaného modelu poměrně dobře detekuje a sleduje vozidla, o kterých v průběhu sledování vytváří statistiky. Práce tak plní všechny body zadání.

Při trénování a ladění modelu hodnoty ztrátových funkcí sice konvergují, ale jsou stále moc vysoké. To značí, že se model neučí tak kvalitně. To je nejspíše zapříčiněno malým počtem epoch a dat, které má DETR k dispozici. DETR je velice živý projekt, a tak se doslova mění pod rukama, což není ideální, pokud spoléháme na nějaké jeho funkcionality. Stejně tak použitý SORT pro sledování objektů. Oba zmíněné projekty mají nedostatečně vedenou dokumentaci, a tak jsem byl při implementaci často odkázán pouze na komentáře v jejich kódu či diskusní fóra. Přesto se implementace aplikace povedla a přináší upokojivé statistiky o detekovaných a sledovaných vozidlech.

Do budoucna by bylo možné rozšířit aplikaci o další trackery kromě SORT a Centroid Trackeru, mezi kterými by se dalo porovnávat. Protože se DETR stále vyvíjí a přináší nové možnosti, nabízí se po příštích aktualizacích DETRu zdokonalovat implementaci a přizpůsobovat ji aktuální verzi.

Literatura

[1] Zhang, A.; Lipton, Z. C.; Li, M.; aj.: Dive into Deep Learning. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://d2l.ai

[2] Vašata, D.: MI-ADM lecture 2. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://courses.fit.cvut.cz/MI-ADM/lectures/files/MI-ADM-02-en-slides.pdf

[3] Ruder, S.: An overview of gradient descent optimization algorithms. [online], 2016, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1609.04747

[4] LeCun, Y.; Cortes, C.; J.C. Burges, C.: THE MNIST DATABASE of handwritten digits. [online], [cit. 2020-07-30]. Dostupné z: http://yann.lecun.com/exdb/mnist/

[5] Mehta, A.: A Comprehensive Guide to Types of Neural Networks. [online], 2019, [cit. 2020-07-30]. Dostupné z: https://www.digitalvidya.com/blog/types-of-neural-networks/

[6] Chigozie Nwankpa, A. G. S. M., Winifred Ijomah: Activation Functions: Comparison of trends in Practice and Research for Deep Learning. [online], 2018, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1811.03378

[7] Gao, X.; Shi, M.; Song, X.; aj.: Recurrent neural networks for real-time prediction of TBM operating parameters. *Automation in Construction*, ročník 15, 01 2019: s. 130–140, doi:10.1016/j.autcon.2018.11.013.

[8] Valueva, M.; Nagornov, N.; Lyakhov, P.; aj.: Application of the residue number system to reduce hardware costs of the CNN. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://www.sciencedirect.com/science/article/abs/pii/S0378475420301580?via%3Dihub

[9] pooling in CNN actualy work?, D.: Will, Craig. [online], 2018, [cit. 2020-07-30]. Dostupné z: http://principlesofdeeplearning.com/index.php/2018/08/27/is-pooling-dead-in-convolutional-networks/

[10] Verschoof-van der Vaart, W.; Lambers, K.: Learning to Look at LiDAR: The Use of R-CNN in the Automated Detection of Archaeological Objects in LiDAR Data from the Netherlands. ročník 2, 03 2019: s. 31–40, doi: 10.5334/jcaa.32.

[11] Zhang, A.; Lipton, Z. C.; Li, M.; aj.: Dive into Deep Learning. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://d2l.ai/chapter_computer-vision/fcn.html

[12] Kostadinov, S.: Understanding Encoder-Decoder Sequence to Sequence Model. [online], 2019, [cit. 2020-07-30]. Dostupné z: https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346

[13] Kostadinov, S.: Understanding Encoder-Decoder Sequence to Sequence Model. [online], 2019, [cit. 2020-07-30]. Dostupné z: https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346

[14] Vaswani, A.; Shazeer, N.; Parmar, N.; aj.: Attention Is All You Need. [online], 2017, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1706.03762

[15] Carion, N.; Massa, F.; Synnaeve, G.; aj.: End-to-End Object Detection with Transformers. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/2005.12872

[16] Shorten, C.; Khoshgoftaar, T. M.: A survey on Image Data Augmentation for Deep Learning. [online], 2019, [cit. 2020-07-30]. Dostupné z: https://link.springer.com/article/10.1186/s40537-019-0197-0

[17] Taylor, L.; Nitschke, G.: Improving Deep Learning using Generic Data Augmentation. [online], 2017, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1708.06020

[18] Yu, F.: A Comprehensive guide to Fine-tuning Deep Learning Models in Keras (Part I). [online], 2016, [cit. 2020-07-30]. Dostupné z: https://flyyufelix.github.io/2016/10/03/fine-tuning-in-keras-part1.html

[19] Chopra, E.: Using Histogram of Oriented Gradients (HOG) for Object Detection. [online], 2015, [cit. 2020-07-30]. Dostupné z:

https://iq.opengenus.org/object-detection-with-histogram-of-oriented-gradients-hog

[20] Wovenware: Anchor Boxes in Object Detection: When, Where and How to Propose Them for Deep Learning Apps. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://www.wovenware.com/blog/2020/06/anchor-boxes-in-object-detection-when-where-and-how-to-propose-them-for-deep-learning-apps/

[21] Ng, A.; Katanforoosh, K.; Mourri, Y. B.: Convolutional Neural Networks. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://www.coursera.org/lecture/convolutional-neural-networks/non-max-suppression-dvrjH

[22] MathWorks: Anchor Boxes for Object Detection. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://www.mathworks.com/help/vision/ug/anchor-boxes-for-object-detection.html

[23] Zhang, A.; Lipton, Z. C.; Li, M.; aj.: Dive into Deep Learning. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://d2l.ai/chapter_convolutional-modern/vgg.html

[24] Zhang, A.; Lipton, Z. C.; Li, M.; aj.: Dive into Deep Learning. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://d2l.ai/chapter_computer-vision/ssd.html

[25] Redmon, J.; Divvala, S.; Girshick, R.; aj.: You Only Look Once: Unified, Real-Time Object Detection. [online], 2015, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1506.02640

[26] Girshick, R.; Donahue, J.; Darrell, T.; aj.: Rich feature hierarchies for accurate object detection and semantic segmentation. [online], 2013, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1311.2524

[27] Girshick, R.: Fast R-CNN. [online], 2015, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1504.08083

[28] Zhang, A.; Lipton, Z. C.; Li, M.; aj.: Dive into Deep Learning. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://d2l.ai/chapter_computer-vision/rcnn.html#faster-r-cnn

[29] Shaoqing, R.; Kaiming, H.; Ross, G.; aj.: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. [online], 2015, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1506.01497

[30] Rohith, G.: R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms. [online], 2018, [cit. 2020-07-30]. Dostupné

z: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

[31] Redmon, J.; Divvala, S.; Girshick, R.; aj.: You Only Look Once: Unified, Real-Time Object Detection. [online], 2015, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1506.02640

[32] Redmon, J.; Farhadi, A.: YOLO9000: Better, Faster, Stronger. [online], 2016, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1612.08242

[33] Redmon, J.; Farhadi, A.: YOLOv3: An Incremental Improvement. [online], 2018, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1804.02767

[34] Kaiming, H.; Xiangyu, Z.; Shaoqing, R.; aj.: Deep Residual Learning for Image Recognition. [online], 2015, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1512.03385

[35] Johnson, J.: Benchmarks for popular convolutional neural network models. [online], 2017, [cit. 2020-07-30]. Dostupné z: https://github.com/jcjohnson/cnn-benchmarks

[36] Carion, N.; Massa, F.; Synnaeve, G.; aj.: End-to-End Object Detection with Transformers. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/2005.12872

[37] research team, F. A.: End-to-End Object Detection with Transformers. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://github.com/facebookresearch/detr

[38] Lowe, D. G.: Object Recognition from Local Scale-Invariant Features. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://www.cs.ubc.ca/~lowe/papers/iccv99.pdf

[39] Held, D.; Thrun, S.; Savarese, S.: Learning to Track at 100 FPS with Deep Regression Networks. [online], 2020, [cit. 2020-07-30]. Dostupné z: http://davheld.github.io/GOTURN/GOTURN.html

[40] Hyeonseob, N.; Bohyung, H.: Learning Multi-Domain Convolutional Neural Networks for Visual Tracking. [online], 2015, [cit. 2020-07-30]. Dostupné z: https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Nam_Learning_Multi-Domain_Convolutional_CVPR_2016_paper.pdf

[41] Hyeonseob, N.; Bohyung, H.: Learning Multi-Domain Convolutional Neural Networks for Visual Tracking. [online], 2015, [cit. 2020-07-30]. Dostupné z: http://cvlab.postech.ac.kr/research/mdnet/

[42] Guanghan, N.; Zhi, Z.; Chen, H.; aj.: Spatially Supervised Recurrent Convolutional Neural Networks for Visual Object Tracking. [online], 2016, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1607.05781

[43] Bewley, A.; Ge, Z.; Ott, L.; aj.: Simple online and realtime tracking. [online], 2016, doi:10.1109/icip.2016.7533003, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/abs/1602.00763

[44] Rosebrock, A.: OpenCV Vehicle Detection, Tracking, and Speed Estimation. [online], 2019, [cit. 2020-07-30]. Dostupné z: https://www.pyimagesearch.com/2019/12/02/opencv-vehicle-detection-tracking-and-speed-estimation/

[45] for Object Detection, K. F.: Xiao, Yao. [online], 2018, [cit. 2020-07-30]. Dostupné z: https://github.com/PatrickXYS/Reproduce_frcnn

[46] Xiao, Y.: Vehicle Detection in Deep Learning. [online], 2019, [cit. 2020-07-30]. Dostupné z: https://arxiv.org/ftp/arxiv/papers/1905/1905.13390.pdf

[47] Hunter, D.: Vehicle Detection. [online], 2018, [cit. 2020-07-30]. Dostupné z: https://github.com/thedch/vehicle-detection

[48] Antevis: Vehicle Detection Project. [online], 2017, [cit. 2020-07-30]. Dostupné z: https://github.com/antevis/CarND-Project5-Vehicle_Detection_and_Tracking

[49] Huansheng, S.; Haoxiang, L.; Huaiyu, L.; aj.: Vision-based vehicle detection and counting system using deep learning in highway scenes. [online], 2019, [cit. 2020-07-30]. Dostupné z: https://link.springer.com/article/10.1186/s12544-019-0390-4

[50] Joshi, P.: Build your own Vehicle Detection Model using OpenCV and Python. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://www.analyticsvidhya.com/blog/2020/04/vehicle-detection-opencv-python/

[51] Guan, K.: Vehicle Detection and Tracking. [online], 2017, [cit. 2020-07-30]. Dostupné z: https://github.com/kcg2015/Vehicle-Detection-and-Tracking

[52] InterviewBuddy: What is precision recall tradeoff? [online], 2019, [cit. 2020-07-30]. Dostupné z: https://www.machinelearningaptitude.com/topics/machine-learning/what-is-precision-recall-tradeoff/

[53] Rosebrock, A.: Intersection over Union (IoU) for object detection. [online], 2016, [cit. 2020-07-30]. Dostupné z: https:

//www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/

[54] ©, S.: 12 Best Python IDEs And Code Editors In 2020. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://www.softwaretestinghelp.com/python-ide-code-editors/

[55] Pengfei, Z.; Longyin, W.; Dawei, D.; aj.: Vision Meets Drones: Past, Present and Future. [online], 2020, [cit. 2020-07-30]. Dostupné z: http://aiskyeye.com/

[56] team, A.: Albumentations. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://github.com/albumentations-team/albumentations

[57] LovPe: DETR Issue number 101 - training error. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://github.com/facebookresearch/detr/issues/101

[58] Groeneveld, M.: Traffic on a sloping street. [online], 2020, [cit. 2020-07-30]. Dostupné z: https://mixkit.co/free-stock-video/traffic-on-a-sloping-street-2486/

Příloha **A**

Seznam použitých zkratek

AP Avarage Precision

BERT Bidirectional Encoder Representations from Transformers

BPTT Back Propagation Through Time

CNN Convolutional Neural Network

COCO Common Objects in Context

CIFAR Canadian Institute for Advanced Research

CPU Central Processing Unit

DETR DEtection TRansformer

FCN Fully Convolutional Network

FM Feature Map

FN False Negative

FPN Feature Pyramid Network

FP False Positive

GIOU Generalized Intersection over Union

GPT Generative Pretrained Transformer

GPU Graphics Processing Units

HM Hungarian Matching

HOG Histograms of Oriented Gradients

ID IDentity

IOU Intersection over Union

LSTM Long Short-Term Memory

MDNet Multi Domain Network

MOT Multiple Object Tracking

MSE Mean Squared Error

MSF Multi-Scale Features

NMS Non-Maximum Suppression

ORB Oriented fast and Rotated Brief

R-CNN Region-based Convolutional Neural Networks

RGB Red Green Blue

ROLO Recurrent YOLO

RPN Region Proposal Network

ReLU Rectified Linear Units

SGD Stochastic Gradient Descent

SIFT Scale-Invariant Feature Transform

SORT Simple, Online, and Realtime Tracking

SSD Single Shot Detector

SURF Speeded Pp Robust Features

SVM Support Vector Machine

TN True negative

TP True Positive

YOLO You Only Look Once

Příloha B

Instalace a manuál

Pro správnou funkčnost programu je nutné vložit všechny soubory obsažené na přiloženém médiu ve složce `src/impl/` do jedné složky. Následně do této složky stáhnout balíček DETR do složky `detr`, SORT do složky `sort` a Centroid Tracker do složky `centroid` (pokud již není obsažený mezi soubory). K těmto balíčkům se následně musí přidat systémová cesta Pythonu, abychom je mohli používat ze zmíněné složky. Do složky je nutné přidat ještě další podsložky a do nich souboru obrázků, videí a anotací, jak je popsáno v kapitole s implementací.

Programy byly pouštěné z programovacího IDE PyCharm, avšak mělo by je být možné spustit i bez tohoto IDE. Doporučuji použít virtuální prostředí pro správu balíčků Pythonu a nainstalovat do něj všechny potřebné balíčky, podle souboru req.txt. DETR a SORT jsou stále velice aktivní projekty, a tak je možné že v nich dojde ke změně nějaké funkcionality a přestanou fungovat tak, jak implementace očekává.

Příloha C

Obsah přiloženého média

Na přiloženém médiu se vyskytuje následující adresářová struktura.

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... složka pro text práce
  - `DP_Pilař_Petr_2020.pdf` ..... text práce ve formátu PDF