VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DETEKCE VOZIDEL V OBRAZE

VEHICLE DETECTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ADAM PETRÁŠ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAKUB ŠPAŇHEL**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Petráš Adam**
Program: Informační technologie
Název: **Detekce vozidla v obraze**
Detection of Vehicles in Image and Video
Kategorie: Zpracování obrazu

Zadání:

1. Prostudujte základy zpracování obrazu a zorientujte se v současných metodách detekce objektů.
2. Obstarejte si vhodnou datovou sadu pro problematiku detekce vozidel.
3. Vyhledejte literaturu zabývající se detekcí objektů se zaměřením na dopravní scény.
4. Vyberte vhodné metody detekce vozidel použitelné na zařízeních typu nVidia Jetson a experimentujte s nimi.
5. Vhodným způsobem vyhodnoťte vybrané metody a diskutujte dosažené výsledky.
6. Vytvořte plakát a video prezentující vaší práci, její cíle a výsledky.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních tří bodů zadání
- Rozpracovaný 4. bod zadání
- Odevzdání rozepsaného textu práce

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

Vedoucí práce: **Špaňhel Jakub, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 14. května 2020
Datum schválení: 1. listopadu 2019

Abstrakt

Tato bakalářská práce je zaměřena na detekci vozidel v obraze. V práci je rozebrána metoda detekce vozidel pomocí konvolučních neuronových sítí, jejich struktury a modely. Všechny skripty byly realizovány v programovacím jazyce Python s rozhraním Tensorflow Object Detection API. První část bakalářské práce jsem věnoval strukturám populárních neuronových sítí a modelům detekčních neuronových sítí. Další kapitola se zabývá nejznámějšími frameworky, které se používají pro strojové učení. Byly vybrány tři modely neuronové sítě, jež byly natrénovány na datasetu COD20K. Výsledkem jsou statistické údaje, které pojednávají o efektivitě a výkonu jednotlivých modelů na natrénovaném datasetu a porovnání výkonu bez zobrazení videa na zařízeních Nvidia RTX 2060, kdy výkon dosažený sítí SDD MobileNet V2 byl 300FPS a Nvidia Tegra TX2 8GB, jehož výkon dosahoval téměř 44FPS.

Abstract

This bachelor thesis is focused on vehicle detection. The thesis deals with the method of vehicle detection using convolutional neural networks, their structures and models. All scripts were implemented using python programming language with Tensorflow Object Detection API interface. The first part of this thesis was devote to the structures of popular neural networks and models of detection neural networks. The next chapter deals with the most famous frameworks that are used for machine learning. Three neural network models were selected and trained on the COD20K dataset. The result of this thesis is statistics that discuss the efficiency and performance of each model on trained dataset and compare performance without displaying video on Nvidia RTX 2060, where the performace archieved by SSD MobileNet V2 network was 300FPS and Nvidia Tegra TX2 8GB, whose performace reached almost 44FPS.

Klíčová slova

detekce vozidel, konvoluční neuronové sítě, Python, TensorFlow, Nvidia Tegra, TensorRT, kvantizace, R-CNN, SSD, ONNX

Keywords

vehicle detection, convolutional neural networks, Python, TensorFlow, Nvidia Tegra, TensorRt, quantization, R-CNN, SSD, ONNX

Citace

PETRÁŠ, Adam. *Detekce vozidel v obraze*. Brno, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jakub Špaňhel

Detekce vozidel v obraze

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jakuba Špaňhela a uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Adam Petráš
26. května 2020

Poděkování

Touto cestou bych chtěl poděkovat Ing. Jakubu Špaňhelovi za odbornou pomoc a cenné rady při vedení této bakalářské práce.

Obsah

Kapitola 1

Úvod

Cílem této práce je detekce vozidel v běžné dopravní situaci. Použitelných metod pro detekci vozidel existuje mnoho a jejich význam je jediný, a to monitorovat určité vyhrazené zóny. Monitoringem se rozumí sledování buďto dopravní situace, například někde v úsecích, ale také sledování a detekce vozidla v dané, například zakázané, zóně. Cílem tohoto monitoringu je aktuální přehled o dění v provozu, hustotě provozu, měření rychlosti vozidel, počítání dopravních prostředků a zjišťování přestupků. Na základě těchto informací jsou pak počítány stupně provozu, odhad dojezdu na vzdálený cíl, nebo také detekce dopravních přestupků.

Náplní této bakalářské práce je teoretický rozbor metody detekce pomocí konvolučních neuronových sítí. Dále trénování modelů Faster R-CNN Inception v2, SSD Inception v2 a SSD MobileNet v2 na datové sadě COD20k a porovnání výsledků jednotlivých modelů. Dále bude probíhat porovnání výkonů jednotlivých modelů na zařízeních Nvidia Tegra TX2 a Nvidia RTX 2060

V úvodní části bakalářské práce byly popsány nejznámější architektury konvolučních neuronových sítí a nejpoužívanější modely detekčních neuronových sítí. Následuje kapitola, kde jsou nastíněny nejpoužívanějšími frameworky. Existují také metody založené na počítačovém vidění a zpracování obrazu, jako je například rozdíl snímků, rozdíl aktuálního snímku a pozadí, detekce pomocí hranového detektoru a další, ale těmito metodami se tato bakalářská práce nezabývá.

V kapitole implementace s použitím knihoven TensorFlow jsou shrnuty všechny kroky, které je potřeba vykonat. Je zde popsán způsob vytváření datové sady a její struktura, dále pak následná anotace této datové sady, kde je popsán využitý program, jež byl využit k vytváření anotací pro vytvořenou datovou sadu. Následně je popsán postup extrakce anotací, kdy bylo potřeba vytvořit TfRecord soubory, jež jsou potřebné pro trénování modelů. Po části, která se věnuje preprocessingu následovala konfigurace modelů pro učení a popis samotného postupu učení. Poslední částí této kapitoly je postup optimalizace modelů pomocí knihoven TensorRT.

V poslední kapitole této bakalářské práce je uvedena přesnost natrénovaných modelů jejich výkon na zařízeních Nvidia Tegra TX2 a Nvidia RTX 2060. Jsou zde také uvedeny možné optimalizace a vylepšení výkonu pro zařízení Nvidia Tegra TX2.

Kapitola 2

Architektury populárních neuronových sítí a struktury používaných detekčních modelů

V této kapitole kapitole jsem se zabýval strukturou neuronových sítí pyramidové architektury, do které spadají sítě AlexNet, VGG. Dále jsem se věnoval dvěma specifickým architekturám. První z nich nese název Inception a využívá Inception blocky. Druhá ResNet je sestavena z reziduálních bloků. Všechny již zmiňované sítě mají jedno společné, a tím je použití. Jsou použity ke klasifikaci objektů.

Jako další téma této kapitoly jsem zvolil nejpopulárnější modely detekčních konvolučních neuronových sítí. V této části byly vylíčeny modely z rodiny R-CNN, mezi něž patří R-CNN, Fast R-CNN, tyto modely mají společný algoritmus výběru zájmové oblasti Selective Search. Dále pak Faster R-CNN, který je již kompletně postaven na plně konvoluční neuronové síti. Jako poslední z této rodiny je Mask R-CNN, který je schopen obraz i segmentovat. Po R-CNN jsem se věnoval Single shot detektorům, mezi něž patří YOLO a SSD. Tyto modely dokáží běžet v reálném čase, na průměrném zařízení a jsou specifické tím, že se na detekční obrázek "dívají pouze jednou". Následkem toho dokáží běžet i na běžných zařízeních.

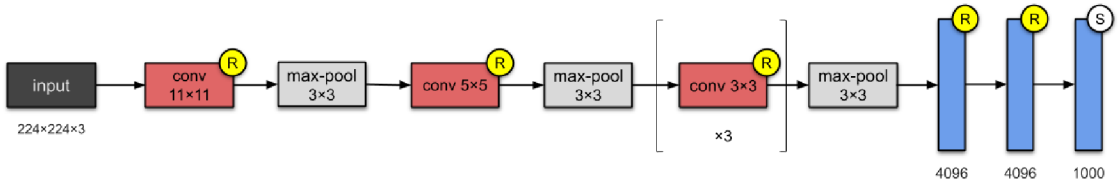
2.1 AlexNet

Neuronová síť AlexNet byla vytvořena v roce 2012 autorem Alexem Krishevskym. Architektura se skládá z osmi vrstev, kde pět vrstev je konvolučních a tři vrstvy jsou plně propojené vrstvy, aktivační funkce jednotlivých vrstev jsou označeny znaky v kolečcích, kde R značí ReLU a S značí SoftMax viz obrázek 2.1. Neuronová síť AlexNet obsahuje 60 milionů parametrů .

AlexNet používá ReLU[1], které jsou dnes již standardem. Výhodou ReLU je doba trénování. CNN[2] používající ReLU byly schopny dosáhnout chyby 25% v datové sadě CIFAR-10 šestkrát rychleji, než CNN používající tanh.

[1] ReLU - aktivační funkce
[2] CNN - Konvoluční neuronová síť

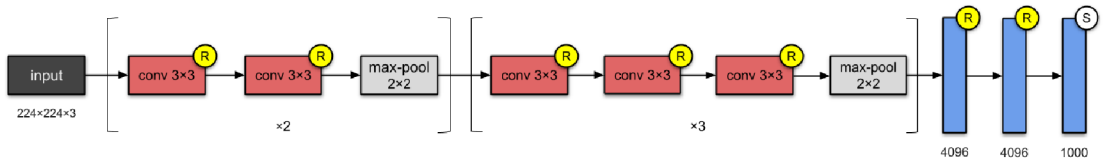
AlexNet umožňuje MultiGPU tak, že polovinu neuronů modelu umístí na první GPU[3] a druhou polovinu na druhé GPU. To znamená nejen že lze trénovat větší model, ale také se zkracuje doba trénování [14].



Obrázek 2.1: Architektura konvoluční neuronové sítě AlexNet, která byla použita na světové soutěži ImageNet [14].

2.2 VGG

Byla vytvořena v roce 2014 autory Simonyan a Zisserman. Existují dva typy VGG-16 a VGG-19, kdy VGG-19 má přidány dvě konvoluční vrstvy. Tyto neuronové sítě jsou charakteristické jednoduchostí a používáním konvolučních jader o velikosti 3x3 a stále se zvyšující hloubky. Snížení velikosti je zajištěno max pooling vrstvami. Za plně propojenými vrstvami, kdy každá obsahuje 4096 uzlů, následuje klasifikátor softmax.



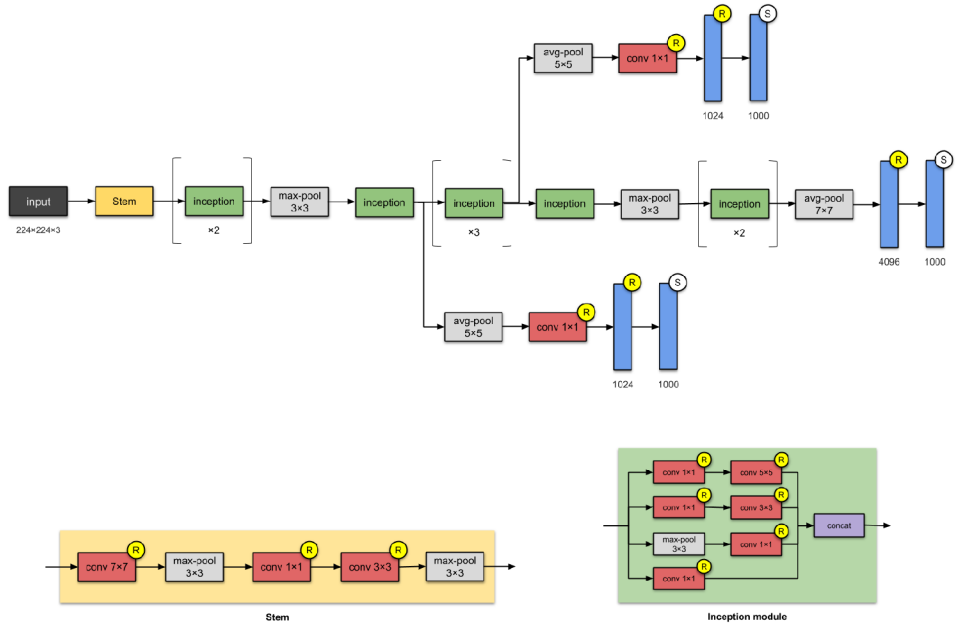
Obrázek 2.2: Architektura konvoluční neuronové sítě VGG-16 [14].

VGG-16 obsahuje 13 konvolučních a 3 plně propojené vrstvy viz obrázek 2.2. Kvůli hloubce sítě a počtu plně propojených uzlů, potřebuje VGG-16 533MB paměti a VGG-19 574MB paměti [14].

2.3 Inception

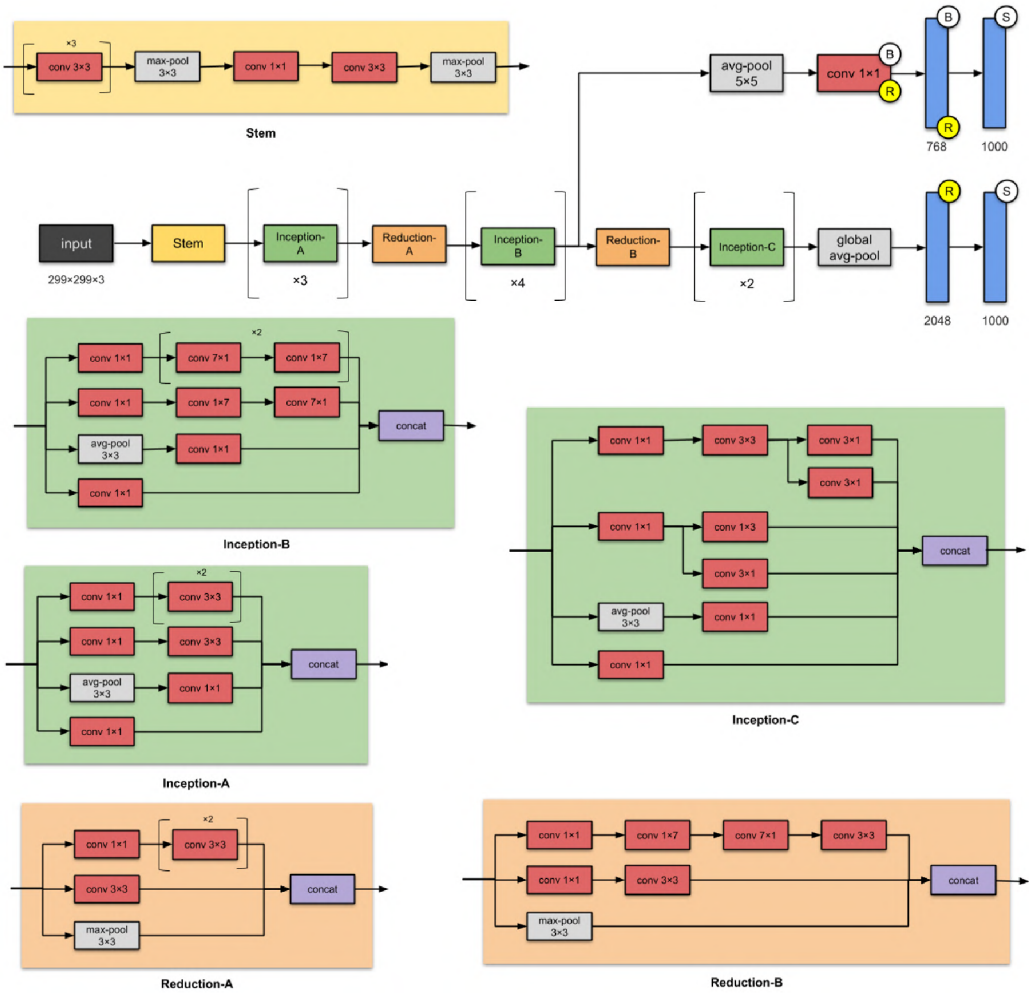
Neuronová síť Inception byla vytvořena již v roce 2014. Architektura má čtyři typy, Inception V1 viz obrázek 2.3, Inception V2 je znázorněna na obrázku 2.4, společně se sítí Inception V3, následuje poslední verze Inception V4, kdy číslo označuje verzi. Neuronová síť se skládá z modulů, které obsahují konvoluční jádra o velikostech 1x1, 3x3 a 5x5. Modul je rozdělena do čtyř větví, kdy v každé větvi provádí konvoluci 1x1.

[3]GPU - Grafický procesor



Obrázek 2.3: Architektura neuronové sítě Inception V1. Tato neuronová síť obsahuje dvě pomocné sítě, které zkracují inferenční čas [14].

V roce 2015 byla vytvořena architektura Inception V2 a v podstatě V3, které jsou si velmi podobné publikoval vědec Christian Szegedy. Neuronová síť Inception V3 obsahuje 22 vrstev a 24 milionů parametrů. Motivace sítí V2 a V3 bylo předejít reprezentativním místům. To znamená drastická redukce vstupní dimenze pro další vrstvu. Incepton V2 a V3 obsahuje efektivnější výpočty použitím faktorizačních metod.



Obrázek 2.4: Architektura Inception V3, tato neuronová síť má pomocnou síť, která zkracuje inferenční čas. Všechny konvoluční vrstvy jsou následovány batch normalizací a aktivačními funkcemi ReLU (označení R ve žlutém kolečku) [14]. Plně propojené vrstvy využivají aktivačnich funkcí ReLU, Softmax a viz označení R a S. Označení vrstev B v kolečku značí batch normalizaci.

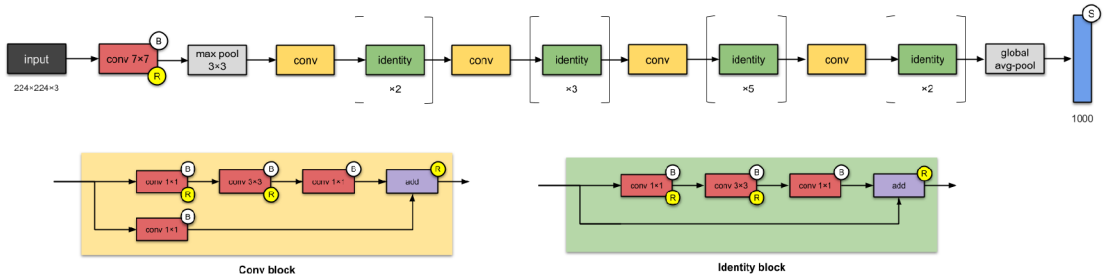
Navrhuje aktualizaci počátečního modulu pro další zvýšení přesnosti klasifikace. Váhy pro Inception V3 jsou menší, než u VGG a Resn, proto velikost paměti, kterou síť bere, je 96MB [14].

2.4 ResNet

Architektura sítě ResNet byla vytvořena v roce 2015 autory ze Standfordu Loffe a Szegedy. Existují čtyři typy ResNet34, ResNet50 viz obrázek 2.5, ResNet101 a ResNet152, kdy číslo je počet vrstev neuronové sítě. Na rozdíl od tradičních sekvenčních síťových architektur jako AlexNet a VGG, je ResNet postavená z reziduálních bloků, těmto blokům se také říká identity blok. ResNet je jedna z prvních sítí, jež prvně obsahovala batch normalizaci.

ResNet obsahuje 26 milionů paramatrů a přestože je ResNet mnohem hlubší než VGG16 a VGG19, velikost modelu je ve skutečnosti podstatně menší kvůli použití spíše globálního

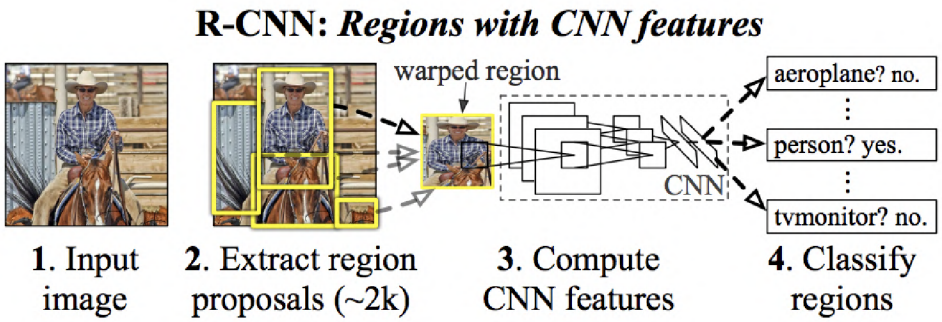
průměrného sdružování než plně propojených vrstev. Tím pádem je velikost modelu ResNet50 rovna 102MB [14].



Obrázek 2.5: Struktura architektury ResNet50 a reziduálních blocků, z nichž je celá neuronová síť stavěna [9].

2.5 R-CNN

Model vytvořen v roce 2014 malým týmem na University of California, Berkeley pod vedením Jitendry Malika [24]. Systém detekce R-CNN je rozdělen do čtyř kroků podle obrázku 2.6.



Obrázek 2.6: Postup vyhodnocení modelu R-CNN krok po kroku od samotného vstupního obrázku až po výstupní bounding box [24].

Prvním krokem je načtení snímku, který bude zpracován. Dále pak vytvoření oblastí, které mají něco společného. Algoritmus, který dokáže seskupit oblasti, se nazývá selective search. Jakmile jsou oblasti vytyčeny, model deformuje oblasti na čtverce a ty předá do konvoluční neuronové sítě AlexNet. Poslední vrstvou CNN je SVM[4], který jednoduše klasifikuje, zda se jedná o objekt a pokud ano, tak o jaký objekt. Algoritmus také predikuje čtyři hodnoty, které jsou jako offset bounding boxu[5] pro zvýšení přesnosti. Například při návrhu oblasti by selective search předpověděl vozidlo, ale část vozidla se v návrhu nevyskytovala. Hodnoty offsetu proto pomáhají při úpravě bounding boxu.

Problémy systému R-CNN:

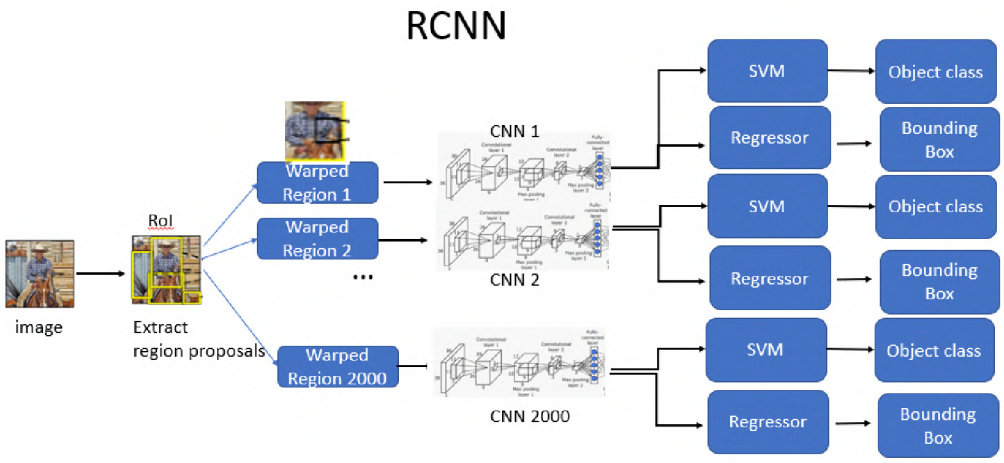
[4]SVM - Support Vector Machines metoda strojového učení s učitelem

[5]bounding box - ohraničující box, jež jež je vytvořen nad objektem a ohraničuje ho

• Trénování sítě trvá dlouho, jelikož musí podle obrázku klasifikovat 2000 návrhů regionů, což můžeme videt na obrázku 2.7 zobrazující architekturu.

• Není možné jej implementovat jako algoritmus, který běži v reálném čase, jelikož zpracování jednoho snímku trvá asi 47 sekund

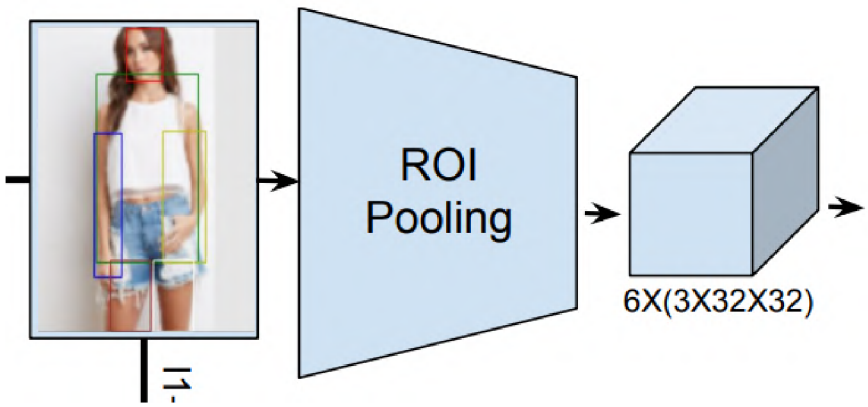
• Selective search není učící algoritmus, takže může vést ke špatnému generování návrhů regionů.



Obrázek 2.7: Kompletní architektura modelu R-CNN, kde v levé části je znázorněn výběr oblastí, ve středu jsou vidět konvoluční neuronové sítě a v právé části výstupy ze všech sítí [16].

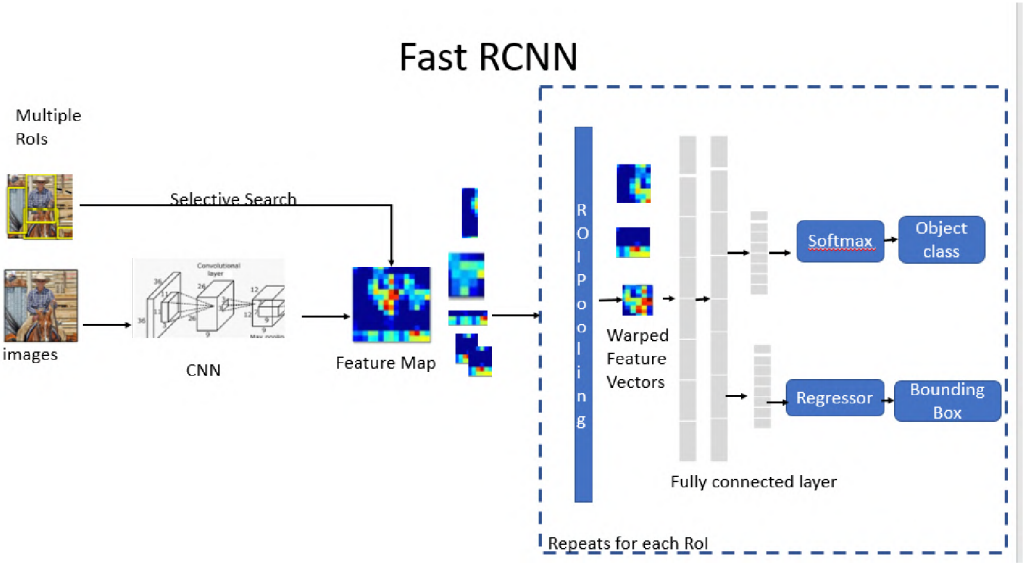
2.6 Fast R-CNN

V roce 2015 Ross Girshink první autor R-CNN vylepšil algoritmus vyhledávání oblastí, kdy rychlost oproti R-CNN stoupla 10x [8]. Hlavní problém byl v tom, že po výběru oblastí se většina z nich překrývala, což způsobilo opakované provádění stejného výpočtu CNN.



Obrázek 2.8: Vrstva ROI Pooling shlukuje překrývající se oblasti a vytváří jednu oblast, která shluk reprezentuje [11].

Jeho představa byla najít způsob jak spustit CNN na obrázek pouze jednou a poté sdílet tento výpočet mezi všechny (2000) oblasti. Přesně to Fast R-CNN dělá, pomocí techniky známé jako RoI[6] Pooling viz obrázek 2.8. Fast R-CNN je navržen tak, aby společně trénoval CNN klasifikátoru a regresoru, který určuje oblasti, kde by se mohl objekt vyskytnout a to všechno v jednom modelu sítě. To znamená, že u R-CNN bylo nutné použít extrakci obrazových dat, poté obrazová data poslat do CNN a následně SVM, kde nám vyšly informace o dění na obrázku. Kompletní architektura ze zobrazena na obrázku 2.9.

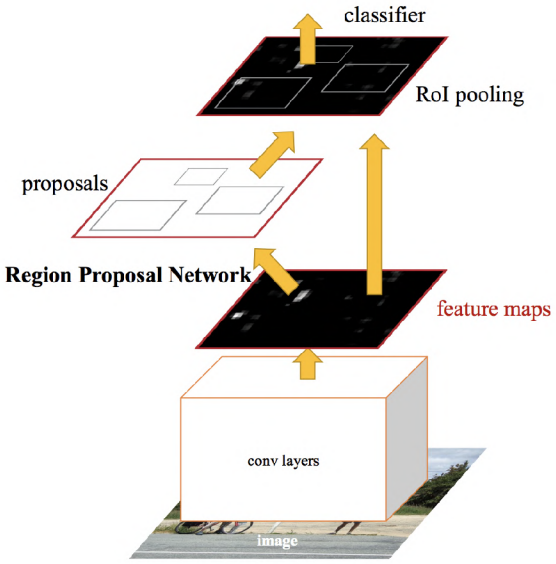


Obrázek 2.9: Kompletní architektura modelu Fast R-CNN, kde v levé části je znázorněn výběr oblasti, ve středu jsou vidět vybrané oblasti a v pravé části se nachází klasifikace [16].

2.7 Faster R-CNN

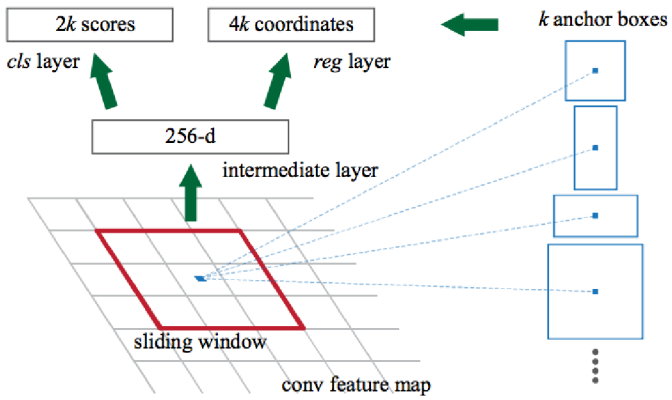
V polovině roku 2015 tým společnosti Microsoft Research složen z členů Shaoqing Ren, Kaiming He, Ross Girshick a Jian Sun vytvořili algoritmus, jehož architektura je znázorněna na obrázku 2.12, který hledá oblasti téměř bezplatným způsobem. Tento model tvořivě pojmenovali Faster R-CNN [25]. Faster R-CNN je hluboká plně konvoluční neuronová síť, která navrhuje oblasti a klasifikuje objekty, které se zde nachází. Hlavní myšlenka tohoto modelu spočívala v tom, že návrhy oblastí závisely na vlastnostech obrazu, které byly vypočteny dopředným průchodem (forward pass) CNN. Faster R-CNN již nepoužívá algoritmus selective seach, který je příliš drahý. Místo algoritmu selective search využívá Region proposal network, dále už jen RPN[7], jak vrstvy prochází tímto algoritmem je zobrazeno na obrázku 2.10.

[6]RoI - Region of Interest zájmová oblast
[7]RPN - Region Proposal Network



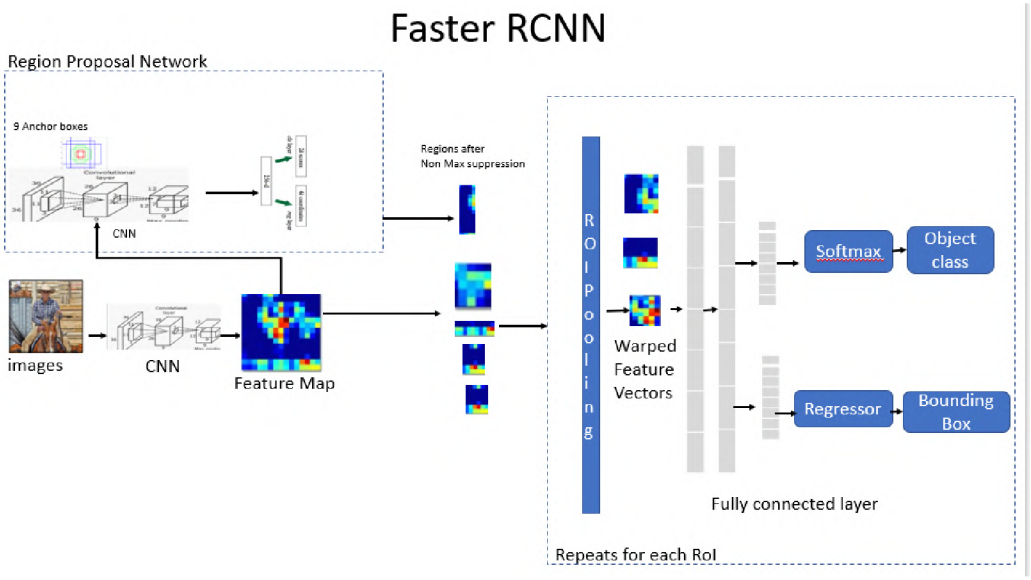
Obrázek 2.10: Na obrázku je znázorněna struktura části RPN, kdy neuronová síť předpovídá možné oblasti, kde by se měl objekt vyskytovat [16].

RPN přijímá obrazová data jako vstup a vytváří sadu návrhů ohraničených objektů, z nichž každý má skóre objektivity. To se provádí přesunem malé neuronové sítě přes feature mapu, která je generována konvoluční neuronovou sítí. Prvek, který vychází z RPN je dále veden do dvou vrstev. První vrstva je regresní a ohraničuje pole. Druhá vrstva je klasifikační, a ta klasifikuje nalezenou oblast.



Obrázek 2.11: Vrstva feature maps, která vybírá oblasti pro RPN [16].

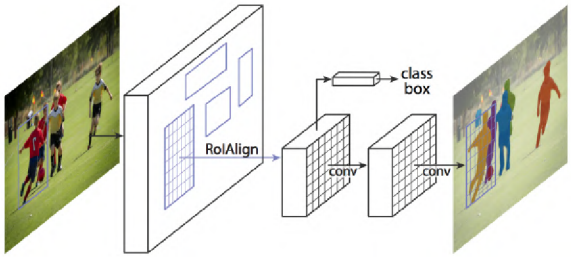
Posun přes feature mapu se provádí tak, že přes obraz jede okno, které má typicky tři velikosti a tři poměry stran. Střed okna se nazývá anchor. V každém umístění posuvného okna současně předpovídáme několik návrhů oblastí viz obrázek 2.11. Maximální počet oblastí označíme k. Poté má regresivní vrstva $4k$ výstupů kódující souřadnice k ohraničení. Vrstva cls má $2k$ výstupů kódujících skóre, které odhadují pravděpodobnost, zda se jedná o objekt či nikoli.



Obrázek 2.12: Celková architektura Faster R-CNN, kde v levé části se nachází výběr oblastí s využitím RPN vrstvy, ve středu jsou znázorněny vybrané oblasti a v pravé části je neuronová síť pro klasifikaci a zpřesnění výsledku [16].

2.8 Mask R-CNN

V roce 2017 tým Facebook AI Research složen z členů Kaiming He, Georgia Gkioxari, Piotr Dollar, Ross Girshick přišel s myšlenkou, jak vyřešit segmentaci instancí pomocí neuronové sítě viz obrázek 2.13. Mask R-CNN je v podstatě rozšířený Faster R-CNN, jež má další větev pro predikci segmentačních masek, v každé zájmové oblasti (RoI) způsobem pixel to pixel. Faster R-CNN neni navržen pro to, aby porovnával objekty pixel to pixel [13].

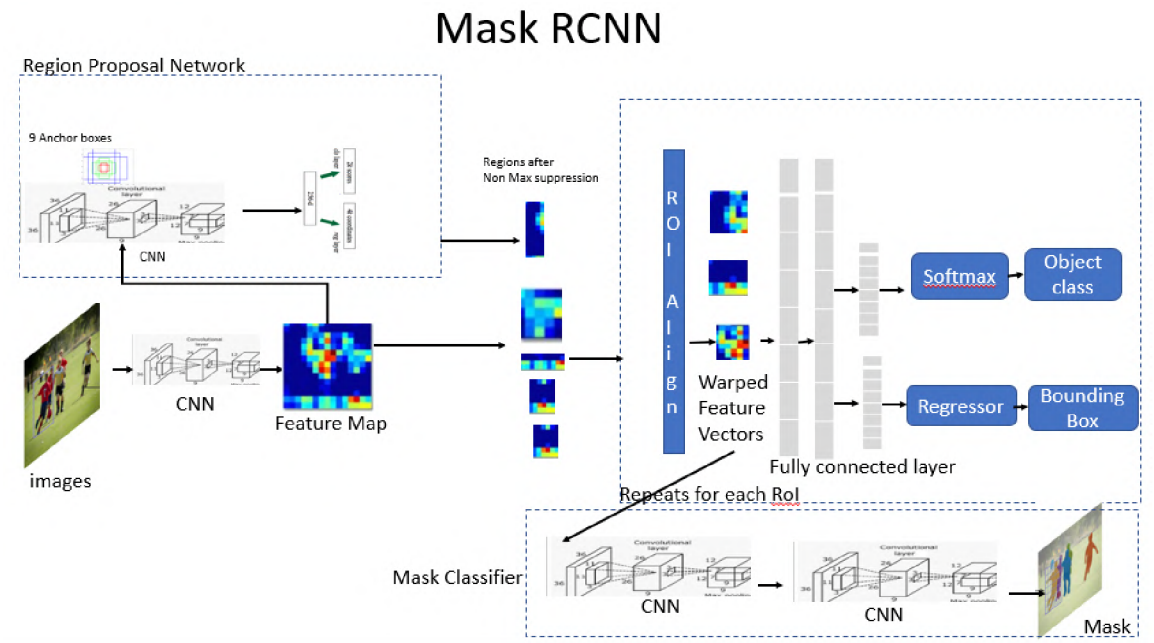


Obrázek 2.13: Architektura modelu Mask R-CNN [13].

Mask R-CNN má dvě fáze. Nejprve generuje oblasti, ve kterých by se na vstupním obrázku mohl objekt nacházet. Poté předpovídá třídu objektu a zpřesňuje ohraničovací rámeček a generuje masku v pixelové úrovni na základě návrhu v první fázi. Obě fáze jsou spojeny páteřní strukturou, kompletní architektura je zobrazena na obrázku 2.14.

Nejprve vstupuje snímek do konvolučních neuronové sítě a generuje feature mapy. Což je v podstatě RPN, kdy pomocí lehkého binárního klasifikátoru vytváří zájmové oblasti. To se provádí pomocí devíti anchor panelů nad obrázkem. Klasifikátor vrací skóre objektu-/skóre NEobjektu. Poté je aplikován algoritmus Non Max Suppression, na anchory s větším skóre objektu. Po vytvoření feature mapy jde na řadu RoI, který vytáhne z mapy všechny

ohraničení a deformuje je na pevné dimenze. Jednotlivé prvky jsou poté vedeny do plně propojených vrstev, aby klasifikace byla provedena pomocí softmax funkce a predikce ohraničujícího pole se dále zpřesňuje pomocí regresního modelu. Tyto prvky jsou také souběžně přiváděny do klasifikátoru masky, který se zkládá ze dvou konvolučních neuronových sítí, pro výstup binární masky pro každý prvek. Klasifikátor, který vytváří masky umožňuje síti generovat masky pro každou třídu bez konkurence mezi třídami.



Obrázek 2.14: Architektura modelu Mask R-CNN [15].

2.9 YOLO-You Look Only Once

V roce 2015 autoři Joseph Redmon, Santosh Divvala, Ross Girshick a Ali Farhadi vytvořili zcela odlišný přístup jak detekovat objekty v obraze [12]. Není to klasický klasifikátor, který se používá jako detektor objektů. YOLO se na obrázek dívá skutečně jen jednou, ale chytrým způsobem. YOLO rozdělí obraz na mřížku 13x13. Poté každá buňka této mřížky odpovídá pěti ohraničením viz obrázek 2.15a.

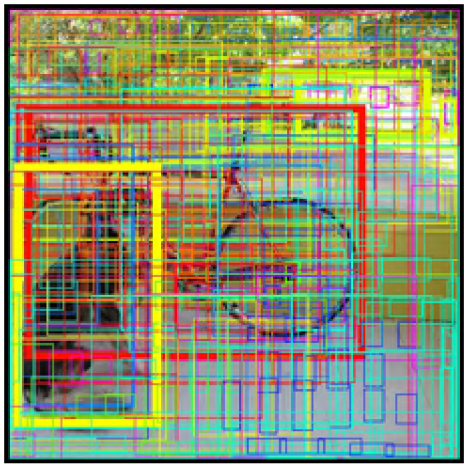
YOLO také vydává skóre, které nám říká, jak jisté je, že předpovězená oblast skutečně ohraničuje objekt. Toto skóre však neříká nic o tom, jaký druh objektu je v ohraničení, toto je zobrazeno na obrázku 2.15b.

Pro každé ohraničení je také předpovězená třída. YOLO dává rozdělení pravděpodobnosti do všech možných tříd podle datasetu. Skóre spolehlivosti ohraničení a predikce třídy jsou sloučeny do jednoho konečného skóre, které nám udává pravděpodobnost, s jakou toto ohraničení obsahuje specifický typ objektu. Kompletní ohraničení je zobrazeno na obrázku 2.15c.

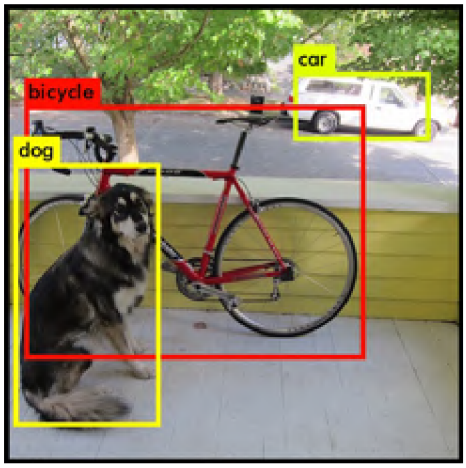
Protože jsme rozdělili obrázek do mřížky 13x13 = 169 buněk a každá buňka obsahuje 5 ohraničení, vyjde nám výsledný počet 845 ohraničení. Ukazuje se, že většina těchto boxů bude mít velmi nízké skóre spolehlivosti, takže ponecháme pouze ty, jejichž konečné skóre je větší, než nějaká předem stanovená hodnota (např. 30% nebo více). Výsledkem je ohraničení detekovaných objektů v obraze viz obrázek 2.15d.

(a) Vstupní obrázek, který je rozdělen do mřížky 13x13 [10].

(b) Vstupní obrázek, s ohraničenímy, kde tloušťka čáry odpovídá pravděpodobnosti, že se vevnitř nachází objekt. Čím tlustší čára, tím větší pravděpodobnost [10].



(c) Vstupní obrázek, kde jsou znázorněny ohraničení, do kterých je brána v potaz i třída objektu. Barva reprezentuje třídu, tloušťka pravděpodobnost [10].



(d) Výsledný obrázek, který vyšel z modelu YOLO, kdy bylo nutné odfiltrovat všechny objekty, které jsou s největší pravděpodobností falešné. To je dáno mezní hodnotou (např. 30%) [10].

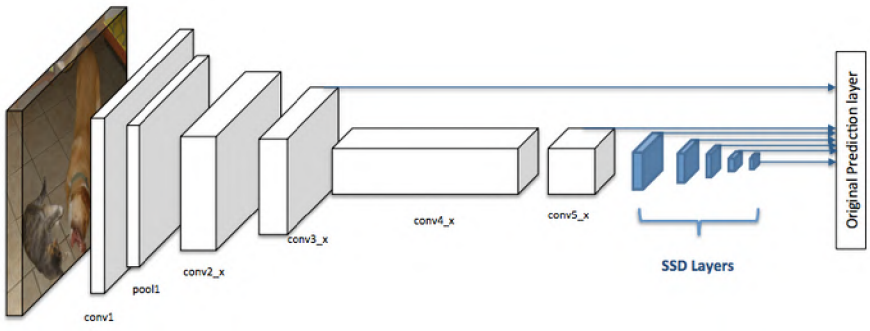
2.10 SSD-Single Shot Multibox Detector

SSD byl vytvořen na konci roku 2016 autory Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg [29]. SSD byl navržen tak, aby co nejlépe běžel v aplikacích, které běži v reálném čase. SSD urychluje proces tím, že eliminuje potřebu RPN viz obrázek 2.15. K poklesu přesnosti používá SSD několik vylepšení, například několik měřítek ohraničení. Toto umožňuje SSD, aby odpovídala s co největší přesností i na obrázcích s nižším rozlišením, což také zvyšuje jeho rychlost. Multi-box také zvažuje okna různých velikostí, a tím zvyšuje robustnost detekce. SSD má dva komponenty **backbone**[8] a **SSD head**[9]. Backbone je obvykle předtrénovaná síť pro kla-

[8] backbone - páteřní síť

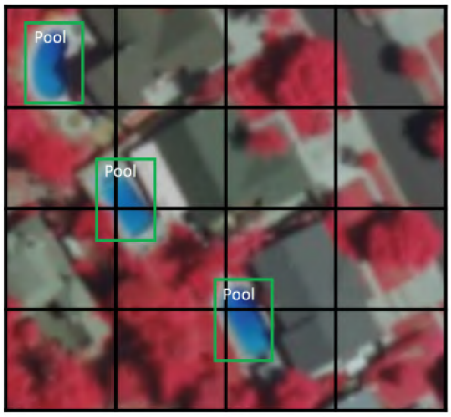
[9] SSD head - jedna konvoluční vrstva přidaná k páteřní síti

sifikaci obrázků jako feature extraktor. Je obvykle tvořena sítí ResNet, která je trénovaná na datasetu ImageNet, ze které byla odstraněna finální plně propojená klasifikační vrstva. SSD head je jedna nebo i více konvolučních vrstev, připojených k backbone a výstupy jsou interpretovány jako ohraničující rámečky a třídy objektů v prostorovém umístění finálních vrstev.



Obrázek 2.15: Architektura SSD, kde bíle je reprezentována část backbone a modře část SSD head [2].

SSD namísto použití posuvného okna rozdělí obraz pomocí mřížky a nechá každou buňku mřížky zodpovědnou za detekci objektů v této oblasti obrazu. Pokud není přítomen žádný objekt, považujeme jej za třídu, která odpovídá pozadí viz obrázek 2.16.

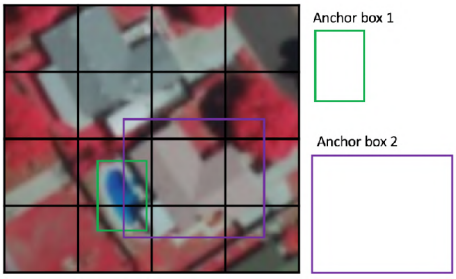


Obrázek 2.16: Vstupní obrázek rozdělen do mřížky o velikosti 4x4, kdy každá mřížka poté odpovídá sama za detekci [2].

K detekci více objektů v jedné buňce SSD využívá takzvaný Anchor box[10]. Tyto boxy jsou předdefinovány a každý z nich je zodpovědný za velikost a tvar v buňce mřížky, což je znázorněno na obrázku 2.17. SSD během trénování přizpůsobuje příslušné anchor boxy k bounding boxům, každého objektu. Za předpovídání třídy a jeho umístění v podstatě odpovídá anchor box s nejvyšším překrytím. Tato vlastnost se používá pro trénování sítě, pro predikci a umístění detekovaných objektů. Každý anchor box je určený poměrem stran a velikostí zvětšení. Ne všechny objekty mají čtvercový tvar. Některé jsou delší, jiné zase

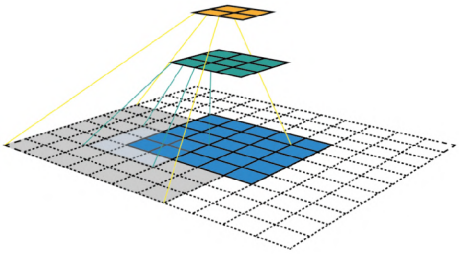
[10]Anchor box - kotevní boxy, které je přiděleny buňkám mřížky

kratší. SSD umožňuje předdefinovat poměry stran anchor boxů tak, aby co nejlépe seděli na daný typ objektu.



Obrázek 2.17: Použití bounding boxů s různým poměrem stran [2].

Receptive fields, neboli receptivní pole, jejíž struktura je znázorněna na obrázku 2.18 je definována jako oblast ve vstupním prostoru, na kterou se dívá konkrétní prvek CNN. Z důvodu operace konvoluce představují prvky v různých vrstvách jiné velikosti oblasti, vycházející ze vstupního obrazu. Jak jdeme hlouběji, velikost představována funkcí se zvětšuje. V tomto příkladu níže začneme spodní vrstvou 5x5, aplikujeme konvoluci, poté jdeme k vrstvě 3x3 aplikujeme konvoluci, následně jdeme do poslední vrstvy 2x2, kde každý prvek odpovídá velikosti 7x7 na vstupním obrázku. Tomuto se říká feature maps, které odkazují na sadu prvků vytvořených použitím stejného extractoru prvků na různých místech. Prvky ve stejné feature mapě mají stejné vnímavé pole a hledají stejný vzor, ale na různých místech. Tím se vytváří prostorová invariance.



Obrázek 2.18: Vizualizace receptivního pole, kde můžeme vidět první vrstvu konvoluce 5x5, následuje vrstva 3x3 a poté je aplikována vrstva 2x2 [2].

Receptivní pole je hlavním kamenem architektury SSD, protože nám umožňuje detekovat objekty v různých měřítcích a vydávat těsnější ohraničení. Další krok SSD spočívá v aplikaci více konvolučních vrstev na backbone a každá z těchto konvolučních vrstev vydává výsledky detekce objektů. Protože starší vrstvy, nesoucí menší vnímavé pole, mohou představovat objekty menší velikosti. Tím predikce z předchozích vrstev napomáhá detekci menších objektů [2].

Shrnutí

V první části sekce jsem prošel velkou část neuronových sítí, jež jsou použity v detekčních modelech, hlavní část této kapitoly jsem věnoval právě modelům. Modely YOLO a SSD jsou pro mé použití na zařízení Nvidia Tegra TX2 nejvyužitelnější nicméně jsem vybral model Faster R-CNN a dva modely SSD. Modely jsou schopny běžet v reálném

čase, a jejich přesnost je pro tuto práci dostačující. Modely Faster R-CNN a SSD jsou nejvhodnější poměřem přesnost/výkon. Model Faster R-CNN byl vybrán hlavně z důvodu porovnání přesnosti modelů SSD, aby se dalo porovnat, jak dokází modely SSD detekovat objekty (vozidla) i vzhledem k rozdílné architektuře modelu.

Kapitola 3

Implementace s použitím knihoven TensorFlow

V této kapitole jsem shrnul populární frameworky, jako například TensorFlow, který nám usnadňuje práci s neuronovými sítěmi. Poskytuje nám rozhraní pro GPU popř. multi GPU nebo multi CPU. Dále jsem se věnoval knihovnám CUDA, cuDNN a TensorRT, kdy knihovny jsou od společnosti Nvidia a akcelerují učení a také samotnou inferenci modelů neuronových sítí a umožňují například optimalizaci, nebo kvantizaci modelů, což zvýší rychlost někdy i několikanásobně a navíc s minimální ztrátou přesnosti. Jako posledním tématem byly frameworky, které nám dávají vyšší úroveň abstrakce, pro vytváření vlastních modelů. Tyto frameworky také využívají již zmíněné knihovny pro optimalizaci modelů a akceleraci výpočtů. Dále jsem se věnoval krokům, které je potřeba provést k detekci vozidel, respektive všech objektů, které chceme detekovat pomocí konvolučních neuronových sítí. Mojí prioritou v další části této kapitoly se stal detailní popis postupu implementace. Jako první jsem proto vybral vytváření datové sady a popsal postup vytváření datasetu[1]. Dataset byl zaměřen na vozidla, jež se mohou vyskytovat kdekoliv v obraze, to znamená, že dataset měl být použitelný na každou scénů. Zahrnuty jsou zde informace o datasetu a kriteria se kterými jsem celou datovou sadu vytvářel. Následuje další sekce, kde větší část je věnována právě anotacím datové sady. Tato část také obsahuje informace o adresářové struktuře celého projektu. Neméně důležitý je také preprocessing, neboli získávání TFRecord a csv souborů z databáze. Proto jsem další část věnoval právě tomuto tématu. Hlavně se v této kapitole zaměřuji na nastavení konfiguračních souborů a samotné učení modelů. Také je zde zmíněn soubor label_map.pbtxt, ze kterého jsou načítaná data, podle niž jsou pak vytvářeny bounding boxy a také účeny modely. V podstatě přiřazuje každé třídě unikátní identifikátor, podle kterého jsou jednotlivé třídy reprezentovány. Na závěr byla provedena optimalizace modelů pomocí knihoven TensorRT. Tuto část mé práce jsem popsal v poslední části kapitoly Implementace s použitím knihoven TensorFlow. Zároveň jsem zde vysvětlil pojmy jako kvantizace a optimalizace a nastínil laickým způsobem, jak kvantizace funguje.

[1] dataset - datová sada

3.1 Použitá zařízení, aplikované nástroje a nejznámější frameworky

Tato sekce je věnována zařízením, jež byla použita pro testování a trénování jednotlivých modelů. Jsou zde jejich specifikace a využití v průmyslu. Dále jsem se zabýval frameworky, které mnozí využívají pro práci s neuronovými sítěmi. Tyto frameworky umožňují optimalizaci a běh na GPU, jako například CUDA, cuDNN nebo TensorRT. Mimo jiné jsem se zabýval i frameworkům, které usnadňují práci s neuronovými sítěmi. Mezi ně patří frameworky jako TensorFlow, Keras či Pytorch.

Nvidia RTX 2060 6GB a Nvidia Tegra TX2 8GB

Specifikace použitých zařízeních jsou na obrázku 3.1. Ze specifikace je možné vyčíst všechny informace, týkající se výkonu jednotlivých zařízení.

Nvidia Tegra TX2		Nvidia RTX 2060 6GB	
GPU Name:	GP10B	GPU Name:	TU106
Codename:	NV13B	Codename:	NV166
Architecture:	Pascal	Architecture:	Turing
Foundry:	TSMC	Foundry:	TSMC
Process Size:	16 nm	Process Size:	12 nm
Transistors:	unknown	Transistors:	10,800 million
Die Size:	unknown	Die Size:	445 mm²
Released:	2016	Released:	2016
Shading Units:	256	Shading Units:	2304
TMUs:	16	TMUs:	144
ROPs:	16	ROPs:	64
SM Count:	2	SM Count:	36
SFUs:	64	SFUs:	576
TPCs:	2	TPCs:	18
GPCs:	1	GPCs:	3
L1 Cache:	48 KB per SM	Tensor Cores:	288
L2 Cache:	512 KB	RT Cores:	36
Max. TDP:	15 W	Tex L1 Cache:	32 KB per SM
		L1 Cache:	64 KB per SM
		L2 Cache:	4096 KB
		Max. TDP:	175 W

Obrázek 3.1: Technická specifikace zařízení Nvidia Tegra TX2 a Nvidia RTX 2060. Již ze specifikace je zřejmé, že chip Nvidie Tegry TX2 je staven spíše na spotřebě a čip grafické karty Nvidia RTX 2060 je staven a preferován spíše na vysoký výkon.

Nvidia RTX 2060 je grafická karta, která je na trhu od ledna 2019. Jádro této grafické karty spadá do architektury Turing, která je byla vyvinuta již v roce 2017. Tato grafická

karta je využívána hlavně pro hraní her, nicméně je také výpočetně výkonná. Také je možné akcelerovat matematické výpočty, explicitně na speciálních jádrech této GPU. Tyto jádra se nazývají Tensor cores. Tensor cores akcelerují maticové výpočty. Bloku těchto jáder se také říká FPU[2]. Grafická karta dosahuje teoretického výkonu 12.90 TFLOPS v poloviční přenosti(FP16), což je 8.6 krát větší teoretický výkon než výkon Nvidie Tegry TX2. Nvidia Tegra TX2 je ARM64 SoC[3] Toto zařízení je cíleno hlavně pro výkon v místech, kde je brán ohled na spotřebu a velikost. Zařízení Nvidia Tegra TX2 je využíváno hlavně na zpracování obrazu. Výkon Nvidia Tegra TX2 je výsoký, nicméně neni dostačující pro aplikace běžící v reálném čase, jež inferencují modely neuronových sítí. Nvidia Tegra TX2 neumožňuje inferenci modelů, jež jsou quantizovány na celočíselnou 8 bitovou přesnost. Teoretický výkon grafické karty Nvidie Tegry TX2 je 1.500 TFLOPS v poloviční přenosti (FP16).

TensorFlow

Tensorflow vznikl ve společnosti Google, původně však pocházel z projektu DistBelief týmů Google Brain, který se začal více používat společností, která spadá pod skupinu jménem Alphabet. Framework umožňuje běh a trénování i inference na více CPU[4] popř. GPU, kdy GPU využívá rozhraní Nvidia CUDA a SYCL/OpenCL. Je možné jej provozovat také v cloudu na její TPU [5] pro vyšší energetickou efektivitu výpočtu. Tyto speciální tensor procesory však nelze koupit, pouze pronajmout v rámci cloudu. API[6] s vyšší úrovní abstrakce se doporučuje použít Keras, ale i přesto samotný Tensorflow obsahuje poměrně vysokou úroveň abstrakce pro vytváření grafů a vrstev neuronových sítí. Vizualizaci trénování a validaci se všemi vytvořenými grafy obstarává program TensorBoard, který pracuje v podstatě jako webová služba, která je spustitelná na počítači lokálně. Git repozitář tensorflow/research, který se nachází na serveru github.com, obsahuje větší množství projektů a také modelů, jako je Tensorflow Object detection API. Právě tento model jsem využil k řešení mé bakalářské práce [7].

CUDA

CUDA, neboli Compute Unified Device Architecture, je hardwarová a softwarová platforma, která umožňuje na specifických GPU spouštět programy v jazyce C/C++ nebo Fortran. Využití této technologie je možné pouze na grafických akcelerátorech a výpočetních kartách společnosti Nvidia, která tuto architekturu vyvinula [19].

cuDNN

CuDNN neboli CUDA Deep Neural Networks knihovna, která je akcelerována grafickou nebo výpočetní kartou společnosti Nvidia, poskytuje uživateli vysoce vyladěnou implementaci standartních metod pro hluboké učení. Implementuje metody jako dopředná, zpětná konvoluce, pooling metody, normalizační metody a aktivační funkce. Knihovnu využívají nejvýznamnější frameworky pro hluboké učení např. Caffe, Keras, Matlab, Tensorflow a Pytorch [20].

[2]FPU - jednotka, která pracuje s čísly plovoucí řádové čárky

[3]SoC - System on Chip je čip, jež zahrnuje všechny součásti počítače nebo jiného elektronického systému do jednoho čipu.

[4]CPU - procesor

[5]TPU - výpočetní čip určený pouze na specifické výpočty pro neuronové sítě

[6]API - aplikční programové rozhraní, jež zprostředkovává široké množství funkcí

TensorRT

TensorRT je SDK[7] pro inferenci modelů neuronových sítí na nejvyšší úrovni s co nejvyšším výkonem. Obsahuje také optimalizátor, který umožňuje optimalizovat a quantizovat modely. Tyto modely pak využívají menší přesnost ale mnohem vyšší rychlost a umožňuje nasadit aplikaci do hyper skalárních datových center, nebo automobilových produktových platforem. Tyto aplikace běží až 40x rychleji než na běžném procesoru. TensorRT je postaveno na platformě CUDA a podporuje paralelní zpracování. TensorRT poskytuje INT8 a FP16 optimalizace pro produkci a rozvinutí aplikací hlubokého učení, například video stream, rozpoznání řeči a zpracování přirozeného jazyka. Redukuje přesnost inference a výrazně redukuje latenci aplikace, která je důležitá pro mnoho služeb, které musí běžet v reálném čase [21].

Keras

Keras je Framework s vyšší úrovní abstrakce, který je přímo určený pro hluboké učení s využitím jazyka Python. Sám však neimplementuje operace s maticemi popř. tensory, ale naopak vyžaduje použití dohromady s jiným podporovaným frameworkem. K frameworku Keras je možné vybrat buďto Tensorflow, CNTK nebo Theano. Od verze 2.0 je jako výchozí doporučeno používat Tensorflow [6].

Pytorch

Pytorch je open source knihovna určená pro strojové učení. Primárně je vyvíjena skupinou Facebook AI Research Lab a je využívána pro aplikace typu počítačového vidění a zpracování přirozeného jazyka. Pytorch je možné využít ve dvou rozhraních a to primárně Python nebo C++. Poskytuje zpracování matic na tensory. Výpočty tensorů jsou akcelerovány grafickými nebo výpočetními kartami [18].

ONNX

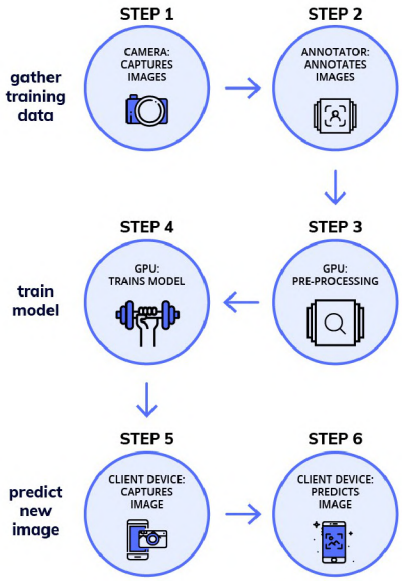
Onnx je otevřený formát, který reprezentuje modely strojového učení. Onnx definuje běžné operátory, jež jsou v podstatě stavebním blokem modelů strojového učení. Dále definuje formát souboru, ve kterém je model uložen, tento umožňuje všem vývojářům používat modely ve vysoké škále frameworků, nástrojů, kompilerů a běhových rozhraních. Hodně lidí pracuje na skvělých nástrojích jako například Tensorflow, Keras, Pytorch a další, ale vývojáři jsou často nucení využívat pouze jeden vývojový software, ve kterém je model vytvořen. Onnx je prvním krokem, který umožňuje prácí více těchto nástrojů tím, že umožní sdílet modely 3.2.

[7] SDK - sada vývojových nástrojů umožňující vytváření aplikací pro určité softwarové balíčky

Obrázek 3.2: Onnx sprostředkovává rozhraní a formát modelů, které je možné použít ve více frameworcích, a také je možné je spouštět na několika možných zařízeních. [22].

3.2 Celkový postup k detekci objektů krok po kroku

Detekce objektů v obraze je komplikovaný problém, který zatím bohužel nelze řešit spolehlivě algoritmicky. Většina detektorů je dnes postavena na hlubokých konvolučních neuronových sítích. Postup od obrázku k detekovaným objektům pomocí neuronových sítí obsahuje zpravidla šest kroků znázorněných na obrázku 3.3. Mezi nejzdlouhavější patří první dva kroky, získávání dat a vytváření anotací. Následují kroky, kdy se z vytvořených dat vytváří řekněme dataset, který je vhodný právě pro trénování sítí a to je krokem čtvrtým. Následně po natrénování sítě máme téměř vyhráno. V krocích 5 a 6 následuje testovací část, kdy je vyhodnocováno, zda je pro nás natrénovaná síť dostatečná nebo je nutné shánět další data, nebo zkoušet jiný model sítě, který je buď rychlejší nebo přesnější.



Obrázek 3.3: Obecný princip detekce objektů pomocí neuronových sítí krok po kroku [3].

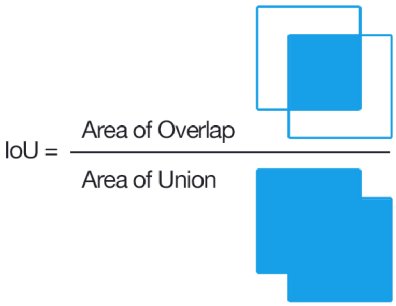
Prvním krokem je vytváření snímků. Je možné i stažení snímků z internetu, ale je nutné si zjistit licenční podmínky. Snímky by měly být rozděleny jako testovací a trénovací. Testovací snímky by měly obsahovat co nejvíce scén, které se mohou vyskytnout a nebo jsou z místa předpokládaného umístění detekčního zařízení. Trénovací snímky musí obsahovat co nejvíce scén, kde se objekty(vozidla) vyskytují. Nejlepší trénovací snímky jsou z míst, kde bude detekční zařízení umístěno, pokud však bude detekční zařízení umístěno na více místech popř. se bude pohybovat, je nutné zachytit co nejvíce případů objektů, které sledujeme. Nikdo není schopen stanovit optimální počet snímků, který je nutný pro to aby neuronová síť pracovala spolehlivě. V případě tvoření databáze snímků se aplikuje pravidlo "čím více tím lépe".

Dále je nutné vytvořit anotace k jednotlivým snímkům. Anotace jsou v podstatě popisným souborem, který popisuje kde se detekovaný objekt nachází a jakého je typu (třídy). Každý snímek má svůj anotační soubor, který má nějaký formát. Existuje mnoho typů formátů anotací pro tuto práci byl vybrán formát Pascal VOC, jež popisuje snímky popisným souborem ve formátu xml.

Třetím krokem je transformace snímků na stanovený rozměr a formát, který neuronová síť potřebuje. Tomuto kroku se také říká preprocessing. Při práci s TensorFlow Object Detection API je do tohoto kroku zahrnuto i vytváření csv souborů z xml nebo txt souborů a následné vytváření TfRecord souborů z csv. TfRecord soubory obsahují binární podobu anotací. TfRecord soubory jsou využity k učení všech přístupných modelů pomocí rozhraní TensorFlow Object Detection API.

Následně po preprocessingu je na řadě práce hardwaru počítače, pokud počítač obsahuje GPU, je možné ji využít a tím pádem zrychlit několikrát výkon. Tato činnost je hodně časově náročná a většinou trvá asi 2 dny. Nechával jsem tedy modely učit vždy dva dny přes noc. Každý model byl natrénován po dobu na 200000 kroků.

Pátý a zároveň šestý krok je vyhodnocování naučených sítí na testovacích datech, kdy vyjde nějaká hodnota na kolik je naše síť přesná. Většinou se vyhodnocují přesnosti v jednotkách mean Average Precission neboli mAP, kdy v jsou porovnávány výsledky neuronové sítě s testovací sadou. Vyhodnocení probíhá výpočtem IoU[8] znazorněné na obrázku 3.4, který vyšel z neuronové sítě a výsledkem z testovací sady.

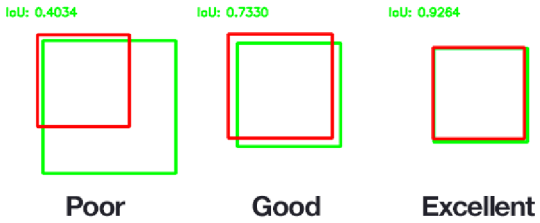


Obrázek 3.4: Výpočet intersekce dvou regionů, je průnik regionů dělený sjednocením dvou regionů [23].

Existuje mnoho typů přesností IoU, podle kterých se určuje přesnost. Všechny přesnosti jsou dány v podstatě thresholdem, který určuje přesnost dvou regionů, které se překrývají na obrázku 3.5. Většinou se setkáváme s hodnotou thresholdu mAP@0.5. Pro to, aby byl

[8]IoU - Intersection over Union

předpoklad pravdivý musí být stanovena správná třída objektu a IoU predikce musí splňovat hodnotu větší než je stanovený threshold. V našem případě je IoU > 0.5. Poté je predikce brána jako správná.



Obrázek 3.5: Příklady IoU, které mohou nastat. Zelenými čtverci jsou zobrazeny ground-truth objekty neboli objekty, jež jsou oanotovány v testovací sadě a jsou v podstatě referencí pro predikci. Červeným čtvercem jsou označeny predikce neuronové sítě. Na levé straně je zobrazeno IoU rovno přibližně hodnotě 0.4, což je velmi nepřesná predikce. V prostředě a v pravé části je zobrazena predikce, která je již velmi přesná a přesahuje hodnotu 0.7 [23].

V tomto kroku probíhá také testování v reálném provozu na reálných datech, se kterými se bude detekční systém setkávat.

3.3 Vytváření datové sady a struktura adresáře

Datová sada se skládá ze dvou sad obrázků a k nim odpovídajících sad anotací, tomuto souborovému uspořádání se říká dataset (datová sada). K trénování se používá trénovací sada obrázků a anotací a k testování se používá testovací sada. Trénovací sada je využívána k samotnému trénování vah neuronové sítě, kdy neuronová síť se těmito obrázky přímo učí. Proto je důležitý výběr obrázků, jež musí býti podobné co nejvíce reálné situaci. Pokud bude detekční zařízení využíváno na několika místech, je nutné sadu rozšířit na co největší možný počet situací a scén. Následuje testovací sada, snímky z testovací sady by měly odpovídat již reálným scénám, ve kterých bude detekce pracovat. Pomocí testovací sady se vyhodnocuje spolehlivost a přesnost naučené sítě.

Obrázek 3.6: Příklad snímků z pozice v úrovni vozidla. Které byly staženy z [26]

Datová sada je sestavena tak, aby pomocí níž bylo možné detekovat vozidla na snímcích, například z dashboard kamer, nebo pouličních kamer. Trénovací sada je rozdělena na 8200 snímků, které jsou vytvořeny jako portréty vozidla, příklady snímků vozidel z datové sady jsou na obrázku 3.6. Dále pak 800 snímků, jak je vidět na obrázku 3.7, jsou vytvořeny z dálničních kamer. Data z kamer nad vozovkou jsem vytvářel pomocí scriptu, který každých n sekund vytvářel snímek některé z vybraných online kamer. Script byl napsán v Pythonu, který na pozadí otevře prohlížeč, vytvoří printscreen a ořeže jej podle pozice a velikosti komponenty. Po ořezání obrázku jej uloží na definované místo a přiřadí unikátní id. Trénovací sada bohužel kvůli časové tísni a náročnosti nebyla dokončena, nicméně bylo zamýšleno vytváření testovacích snímků pomocí dashboardu, kdy bude kamera buďto umístěna ve vozidle, nebo bude sledovat dopravu někde ze stacionárního místa na sloupu. Všechny obrázky mají zpravidla velikost 600x600 se zachováním poměru stran. Tento rozměr obrázků jsem volil hlavně z důvodu rychlosti učení sítě a také z důvodu úspory místa na disku.

Obrázek 3.7: Příklady obrázků z online dostupných kamer. Všechny obrázky byly pořizovány z online kamer z webové stránky [27]

Využil již vytvořenou datovou sadu COD20k, která je volně dostupná pro studijní účely na stránkách fakulty. Tuto datovou sadu jsem využil pro porovnání výsledků přesností, které vyjdou z vybraných modelů.

Struktura adresářů znázorněná na obrázku 3.8 byla daná tím, že jsem využíval více modelů, proto bylo potřeba vytáhnout data do kořenové složky, aby data nebyla duplikována u každého modelu. Dále byly v kořenovém adresáři umístěny všechny scripty, které jsem použil ať už k učení nebo k samotnému testování.

Obrázek 3.8: Adresářová struktura, kde v kořenovém adresáři jsou všechny scripty a do jednotlivých adresářů jsou rozděleny modely. Následně ve dvou adresářích jsou datasety. Při trénování jednoho či druhého datasetu bylo nutné modifikovat cesty v konfiguračním souboru.

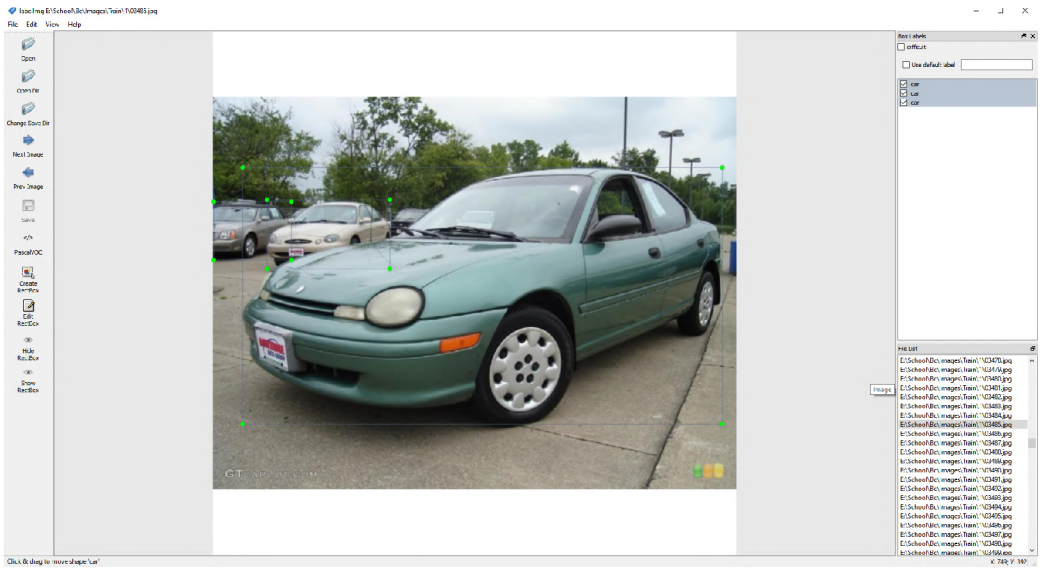
3.4 Anotace obrázků a použité nástroje

Ke každému obrázku musí být soubor, který je reprezentován vybranou strukturou.
V souboru jsou napsána data o objektech vyskytujících se na obrázku, jejich pozice, velikost a třída. Více objektů na jednom obrázku je zaneseno do stejného souboru, který je definován strukturou, která obsahuje informace o všech objektech. Pro vytváření anotací se většinou využívá nějaký grafický software, jež několikanásobně ulehčuje práci. Je možné například označit myší na obrázku oblast, poté vyskočí okno, kde je možné zadat třídu, do které objekt spadá a následně software vytvoří ohraničení definované třídou, pozicí a velikostí. Po uložení software sám vygeneruje anotační soubor ve vybraném formátu.

```
- <annotation>
    <folder>1</folder>
    <filename>07297.jpg</filename>
    <path>E:\School\Bc\Images\Train\1\07297.jpg</path>
  - <source>
      <database>Unknown</database>
    </source>
  - <size>
      <width>600</width>
      <height>600</height>
      <depth>3</depth>
    </size>
    <segmented>0</segmented>
  - <object>
      <name>car</name>
      <pose>Unspecified</pose>
      <truncated>1</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>1</xmin>
        <ymin>144</ymin>
        <xmax>600</xmax>
        <ymax>448</ymax>
      </bndbox>
    </object>
  - <object>
      <name>car</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>43</xmin>
        <ymin>182</ymin>
        <xmax>178</xmax>
        <ymax>253</ymax>
      </bndbox>
    </object>
</annotation>
```

Obrázek 3.9: Struktura PASCAL VOC, kterou jsem zvolil pro anotace svého datasetu.

Pro anotaci obrázku jsem zvolil anotační program LabelImg viz obrázek 3.10, který je dostupný zdarma. Použil jsem jak Linuxovou, tak Windows verzi. Obě verze mají stejnou funkčnost. LabelImg umožňuje nastavit výchozí třídu, což umožnilo vytvářet anotace jedné třídy s velkou rychostí. Dále umí automaticky ukládat vytvořené anotace, kdy dedukuje název anotačního souboru z názvu obrázku, odstraní příponu a přidá příslušnou příponu podle typu anotací. V programu jdou vybrat dva typy anotací. První je plain text, kdy anotace jsou zapisovány do souboru txt ve formátu YOLO. Další typ anotace jsou zapisovány do xml souboru ve formátu PASCAL VOC viz obrázek 3.9.

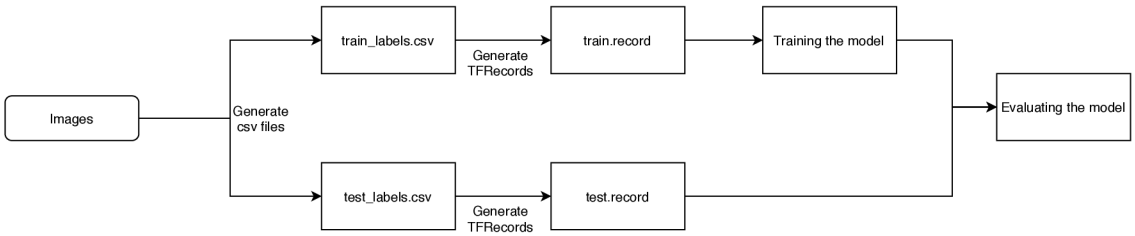


Obrázek 3.10: Anotační program labelImg, který byl využit k anotování vytvářeného datasetu.

Vytvořil jsem si svůj systém označení vozidel. Pokud bylo vidět vozidlo z 50%, tak jsem ho označil. Zároveň jsem bral v úvahu natočení a viditelnost, která část vozidla bylo vidět. Pokud byla vidět pouze část střechy, tak jsem vozidlo neoznačoval, jakmile bylo vidět přední kolo, kousek přední části vozidla, přední část vozidla, zadní část vozidla a většina specifických prvků co vozidla mají, tak jsem je označil. Každý objekt v obraze se musí označit ručně, tím pádem je operace velmi zdlouhavá a je rozdíl, pokud snímky obsahují deset objektů, nebo pouze dva objekty. Trénovací část mého datasetu obsahuje téměř 9000 obrázků s 19000 anotacemi vozidel.

3.5 Extrakce anotací do souborů csv, vytváření TfRecord

Proces přípravy souborů TfRecord pro trénování a testování probíhá tak, že z vytvořeného datasetu, který obsahuje snímky a anotační soubory v případě této bakalářské práce xml, vytvoříme dva csv soubory pro testovací a trénovací část datasetu. Následuje operace, kde se provádí generování souborů TfRecord. Po vygenerování těchto souborů může začít trénování neuronové sítě, popřípadě evaluace, což je znázorněno na obrázku 3.11.



Obrázek 3.11: Proces přípravy TfRecord souborů pro učení nebo evaluaci modelů TensorFlow.

Pro extrakci anotací ze souborů xml do souboru csv byl využit již vytvořený script xml_to_csv.py, který je součástí TensorFlow Object Detection API. Script sjednotí anotace všech xml souborů do jednoho souboru ve formátu csv pro testovací část datové sady a jednoho souboru ve formátu csv pro trénovací část datové sady. Jelikož dataset COD20k byl vytvořen za účelem detekce natočení vozidel, musel být vytvořen script, jež parsoval všechny anotace a filtroval pouze 2D bounding boxy. Následně z textových souborů generoval csv soubor. Po vytvoření csv souboru byly vytvořeny TfRecord soubory, jež obsahují anotace a samotný obrázek v binární podobě. Binární podoba výrazně zrychlí proces čtení dat a umožňuje sekvenční čtení. K vytvoření TfRecord souborů byl využit script generate_tfrecord.py, který je součástí TensorFlow Object Detection API.

3.6 Konfigurační soubory a učení modelů

Modely pro učení je možné stáhnout z github repozitáře [17]. Všechny modely obsahují frozen graph a konfigurační soubor. Po stažení modelů následuje nastavení konfiguračních souborů, podle kterých se učení řídí. Konfigurační soubory obsahují hned několik parametrů od nastavení načítání přes konfiguraci parametrů sítě až ke konfiguraci optimalizačních metod a loss funkcí. Dále je možné nakonfigurovat několik možností augmentace obrazu. TF vytváří sám záchytné body, neboli checkpointy. Na checkpointy se po skončení učení dá navázat, takže není nutné učit celý model od začátku. Z checkpointů se vytváří frozen gra-

phy. Frozen graph je ve své podstatě již naučený model, který při spuštění dokáže vydávat výsledky. Jelikož jsem potřeboval modely, které mají běžet na zařízení Nvidia Tegra, musel jsem vybírat spíše z modelů SSD. Tento model jsem využil hlavně kvůli tomu, že dokáže běžet na průměrných GPU v reálném čase, také jsem chtěl vyzkoušet model Faster R-CNN, hlavně kvůli zvědavosti ohledně výkonu a přesnosti. Nakonec jsem vybral tři modely viz obrázek 3.12, SSD InceptionV2, SSD MobileNetV2 a Faster R-CNN InceptionV2.

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v2_coco	31	22	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes

Obrázek 3.12: Vybrané modely z TensorFlow Object Detection Model ZOO.

K trénování všech sítí byl vytvořen soubor labelmap.pbtxt, v adresáři /annotations do kterého byl zapsán následující kód. V něm by id mělo odpovídat označení v souboru generate_tfrecords.py. Tento skript byl převzán z repozitáře [28]. Soubor labelmap.pbtxt poskytuje informace pro neuronovou síť. Mapuje podle něj, do které kategorie objekt patří. Labelmap pro tuto práci vypadá následovně.

```
item
{
    id: 1
    name: 'car'
}
```

V dalším kroku byl zkopírován konfigurační soubor /<model>/pre-trained/pipeline.config do složky /<model>/training/. <model> definuje vybraný model, jelikož byly učeny tři modely. Konfigurační soubor obsahuje všechno nastavení daného modelu pro trénování a testování sítě. Součástí konfiguračního souboru je také počet kategorií a počet kroků(iterací), po jaky se má model učit. Dále jsou zde cesty k souborům TfRecord. Zkopírovaný konfigurační soubor musíme přejmenovat například <model>.config, jelikož při učení se zde kopíruje také konfigurační soubor pipeline.config.

Trénování modelů bylo prováděno na grafické kartě Nvidia RTX 2060. Všechna trénování probíhala přes noc, jelikož trénování jednoho modelu trvalo něco kolem dvaceti hodin. Po prvních dvaceti tisících krocích jsem vyzkoušel, zda neuronová síť funguje relativně správně a zda dává nějaké výsledky. Pokud ano, tak jsem modely nechával učit až po maximální hodnotu kroku a to je 200 000. K učení byl využit script train.py z knihoven TensorFlow Object Detection API. Následně jak byly natrénovány modely, jsem také zkoušel využít augmentaci. Pro augmentaci byly zvoleny metody náhodné otočení horizontálně, náhodné ořezání obrázku a bounding boxů, náhodné ořezání boxů pro ssd, náhodné zvětšení obrázku, náhodné zmenšení obrázku, náhodná změna světlosti (brightness) a náhodné žvětění hodnoty všech pixelů v obraze. . Po naučení modelů s augmentací jsem porovnával výsledky, zda augmentace k něčemu byla či nikoli. Ve všech případech augmentace obrazu pomohla a neuronová síť byla schopna dávat lepší výsledky.

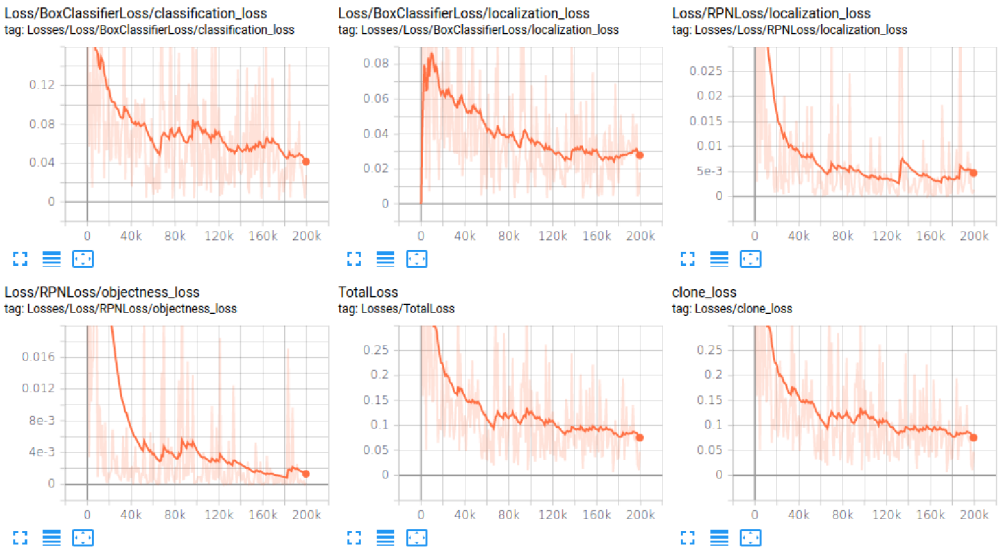
Výhoda modelů SSD Inception V2 a SSD MobileNet V2 je ta, že obsahují regularizační metody, jež nám napomáhají zmírnit přeučení sítě. Regularizace v podstatě reguluje váhy

neuronů a napomáhá síti regulovat velikost váh. Tím pádem má neuronová síť tendenci se pořád učit.

Faster R-CNN

Model Faster R-CNN byl zvolen hlavně z mé zvědavosti, zda je schopný běžet na zařízení Nvidia Tegra Tx2 a také z důvodu porovnání přesnosti s modely SSD. Trénování probíhalo hladce, ze začátku klesala loss velmi prudce poté se však u 80000 téměř zastavila a klesala jen minimálně, což je možné vidět na obrázku 3.13. Tento model byl trénován nejdéle, asi 2 a půl noci.

```
python3 train.py --logtostderr --train_dir=rcnn_training/training
--pipeline_config_path=
rcnn_training/training/faster_rcnn_Inception_v2_coco.config
```

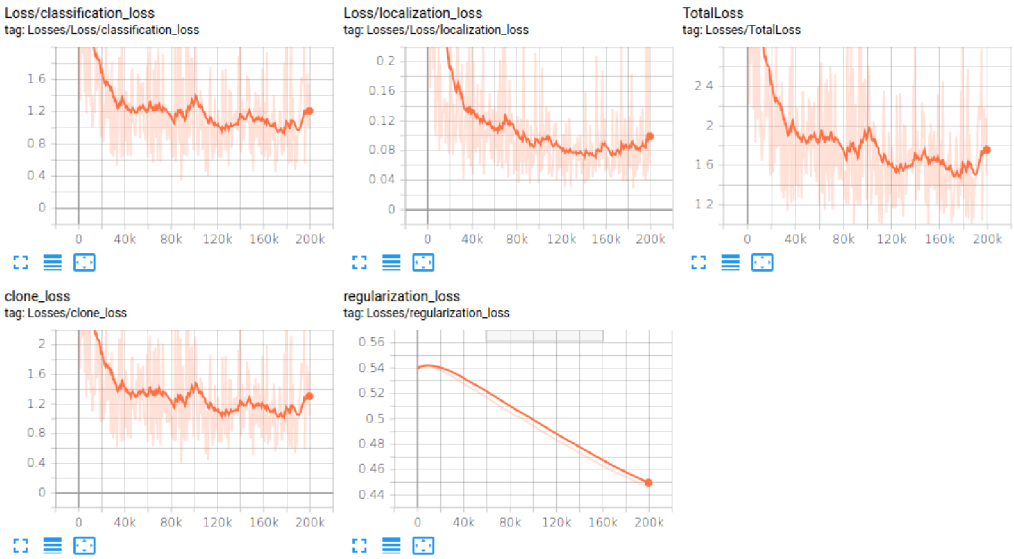


Obrázek 3.13: Průběh trénování modelu Faster R-CNN. Výsledky jsou vyhlazeny hodnotou 0.93.

SSD Inception V2

Trénování modelu SSD Inception V2 se ustálilo již u kroku 40000, kdy poté loss klesala velmi pomalu. Následně v kroku 180000 se křivka obracela a loss se zvedala, což je vidět na obrázku 3.14. U tohoto modelu je toto vidět, kdy v posledních krocích trénování sítě je síť již přeučená a nezabrala ani regularizace, která již ve 20000 krocích začala padat.

```
python3 train.py --logtostderr --train_dir=SSD_Inception_v2/training
--pipeline_config_path=SSD_Inception_v2/training/SSD_Inception_v2.config
```

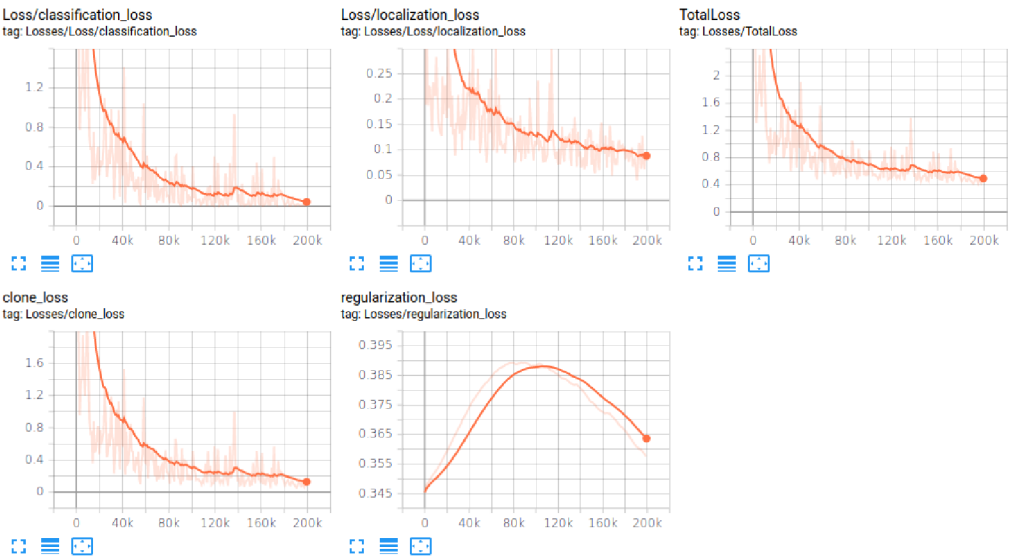


Obrázek 3.14: Průběh trénování modelu SSD Inception V2. Výsledky jsou vyhlazeny hodnotou 0.93.

SSD MobileNet V2

Trénování modelu SSD MobileNet V2 probíhálo hladce loss po celou dobu klesala, kdy je vidět, že i při maximálním kroku má stále tendenci klesat, viz obrázek 3.15.

```
python3 train.py --logtostderr --train_dir=SSD_MobileNet_v2/training
--pipeline_config_path=SSD_MobileNet_v2/training/SSD_MobileNet_v2.config
```



Obrázek 3.15: Průběh trénování modelu SSD MobileNet V2. Výsledky jsou vyhlazeny hodnotou 0.93.

Modely SSD byly problematické z důvodu využitím paměti grafické karty, TF[9] pořád padalo a vyhazovalo tf.stack exception. Zkoušel jsem nainstalovat jak novější, tak starší verze knihoven CUDA, TF, Numpy a CuDNN avšak bez úspěchu. Po přeinstalování všech knihoven jsem se rozhodl nainstalovat nové drivery pro GPU. Po instalaci těchto driverů už to začalo fungovat tak, jak má.

3.7 Optimalizace modelů pomocí TensorRT

Optimalizace modelů se využívá hlavně k zjednodušení složitě navržených struktur, které mohou být sestaveny jednodušeji a tím je proveden výpočet rychleji. Další výhodou optimalizace je paměťová náročnost. Pokud využijeme například model, který je ve FP32[10], jeho jedna váha bude "zakódovaná"v 32 bitech. Pokud bude model ve FP16[11] paměťová náročnost nám klesne téměř o polovinu. Tomuto procesu se říká kvantizace. Většina dnešních grafických karet obsahuje pouze jednotky, které umožňují pracovat s čísly plovoucí řádové čárky. Existují však grafické karty, které mají i jednotky, jež umožňují využití celo číselných operací. V tomto případě je umožněno také quantizovat model na INT8[12], kdy zrychlení tohoto modelu bude znát i na procesoru. Celý proces kvantizace v podstatě mapuje dosavadní váhy a aktivační, loss funkce atd. na vytyčený prostor například u int od 0 do 255, tím pádem budou váhy representovány s relativně malou ztrátou přesnosti. Nicméně prostor, na který se mapuje nemusí být nastaven explicitně. Každý si může prostor implicitně nastavit třeba od -127 do 128 u INT8.

Optimalizaci modelů jsem chtěl provádět pomocí TensorFlow Object Detection API, kdy jsem chtěl využít konverzi modelu na model TensorFlow Lite, dále už jen tfLite. Proto jsem vytvořil a inferencoval model tfLite. Bohužel inference tohoto modelu dokáže běžet pouze na CPU nebo mobilním GPU. Mobilní GPU v tomto případě znamená, GPU jež jsou součástí mobilních telefonů. Bylo tedy nutné se rozhodnout pro zvýšení potenciálu a výkonu jednotlivých modelů. K tomu jsem využil knihoven TensorRT, jež umožňují optimalizovat a quantizovat modely. Pro quantizaci modelu jsem vytvořil script 3.1, ve kterém nadefinuji cestu k modelu, typ modelu po quantizaci, maximální velikost batche, maximální velikost segmentu a další parametry.

```
import tensorflow as tf
from tensorflow.python.compiler.tensorrt import trt_convert as trt
save_dir="./SSD_MobileNet_v2/"
with tf.Session() as sess:
    converter = trt.TrtGraphConverter(
    input_saved_model_dir=save_dir+"my-model/saved_model",
    max_batch_size=1,
    max_workspace_size_bytes=1 << 24,
    precision_mode='FP16',
    minimum_segment_size=50
    )
    trt_graph = converter.convert()
    with open(save_dir+"quantized/frozen_inference_graph.pb", 'wb') as f:
        f.write(trt_graph.SerializeToString())
```

Výpis 3.1: Kvantizace modelu v pythonu s využitím knihoven TensorRT a jejich converteru TRTGraphConverter.

[9] TF - TensorFlow
[10] FP32 - 32bit s plovoucí řádovou čárkou
[11] FP16 - 16bit s plovoucí řádovou čárkou
[12] INT8 - celočíselné 8bit číslo

Kvantizace a optimalizace modelu přinesla pouze vyladění a vyhlazení průběhů snímkové frekvence nikoliv však vylepšení výkonu. Důvod je ten, že modely jsou již od počátku vedeny v přesnosti FP16. Kvůli tomu není možné quantizací modelu z FP16 na FP16 získat nějaký výrazný výsledek. Zkoušel jsem quantizovat modely na INT8, ty však nedokáží běžet s hardwarovou akcelerací na GPU RTX 2060 ani na Tegře TX2, varování je možné vidět na obrázku3.16, jelikož neobsahují výpočetní jednotky pro celočíselné operace [5].

```
2020-04-07 17:45:26.551243: W tensorflow/compiler/tf2tensorrt/utils/trt_logger.c
c:37] DefaultLogger Int8 support requested on hardware without native Int8 suppo
rt, performance will be negatively affected.
```

Obrázek 3.16: Varování při vizualizaci videa na zařízen, které nepodporuje hardwarovou implementaci INT8.

Výkon zařízení Nvidia Tegra TX2 klesl na velmi nizkých 0.7FPS[13]. Přesnost tohoto modelu však neklesla nějak dramaticky. Z toho důvodu je tento proces přínosem, bohužel jsem nemohl vyzkoušet tento model s HW[14] akcelerací. Kvůli tomu musela být využita kvantizace modelu na FP16, která však nebyla přínosem.

Shrnutí

Tato kapitola byla věnována v první části zařízením, na kterých byly použité modely naučeny a také inferencovány, kdy učení bylo prováděno na zařízení Nvidia RTX 2060. Následně byly popsány nejpoužívanější frameworky, jež se využívají pro strojové učení. Také byl popsán již několikrát zmíněný Tensorflow, který jsem využil. V neposlední řadě byly popsány také knihovny CUDA, cuDNN a TensorRT, jež jsou nezbytné pro inferenci a učení neuronových sítí na grafických kartách Nvidia. Tyto knihovny zrychlují jejich výkon a zkracují dobu učení, kdy grafické karty jsou využity na maximum. Po nastínění základních informací o použitých a nejvyužívanějších frameworcích jsem se zabýval vytvářením datasetu, kde byl zmíněn postup, jak byly obrázky získávány. Také jsem se v této části věnoval adresářové struktuře projektu. Dále jsem se zaměřil na vytváření anotací pro vlastní dataset a základní popis softwaru pro vytváření anotací LabelImg. Následně po sekci vytváření anotací došlo na sekci, která se věnovala generování csv souboru a TfRecord souborů, kdy byly využity scripty, jež jsou součásti TensorFlow Object Detection API, Této sekci se říká postprocessing. Po sekci s preprocessingem přišly na řadu konfigurační soubory a jejich nastavení, kdy bylo nutné doplnit cesty k jednotlivým record souborům, cesta k souboru labelmap a také počet kroků, po které se bude model trénovat. Také byly v konfiguračních souborech nastaveny augmentace pro jednotlivé modely. Po nastavení konfigurací mohlo začít samotné trénování modelů. Všechny modely byly trénovány po dobu dvou dnů, tedy 200000 kroků, jež je pro TensorFlow Object Detection API maximum. Následně jsem se věnoval optimalizacím a jejich využití, kdy optimalizace pomocí TensorRT bohužel nepřinesla žádné výkonové vylepšení. Důvod je ten, že všechny modely, jež jsem použil jsou již v přesnosti FP16.

[13] FPS - snímek za sekundu
[14] HW - hardware

Kapitola 4

Dosažené výsledky a možná vylepšení

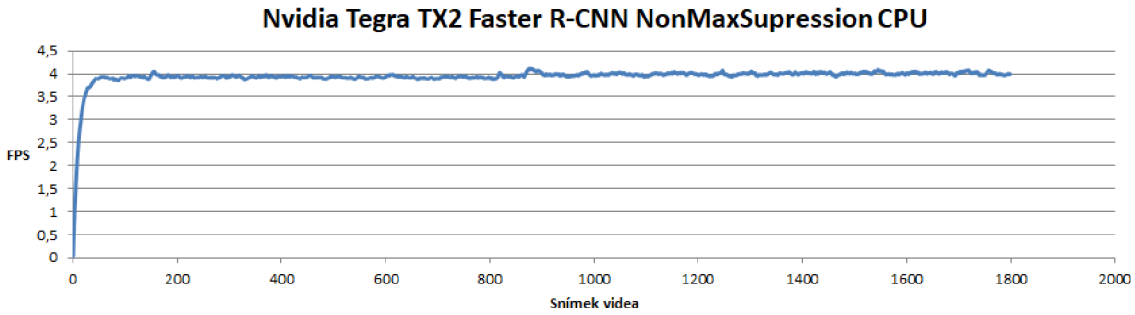
Výsledky byly testovány dvěma způsoby. První způsob byl testování kompletního modelu včetně vykreslení čtverců a další režie spojené s vykreslením. Toto testování probíhalo na videu, které je v rozlišení 1280x720. Délka testovacího videa je jedna minuta. Pro každý snímek je vypočítaná průměrná snímková frekvence, ta je vykreslena do grafu. Druhý způsob testování byl prováděn tak, že probíhalo testování pouze samotné neuronové sítě, kdy snímky byly předem načteny, zmenšeny na velikost 300x300, jelikož modely SSD potřebují tuto velikost a sami explicitně větší/menší obrázky převádí na velikost 300x300. Toto by přineslo režii modelu, kdy by musel sám explicitně každý obrázek zmenšovat. Model Faster R-CNN je schopen přijímat variabilní velikost obrázku s minimální velikostí 600 a maximální velikostí 1024. Proto byla nastavena velikost obrázku na 600x600, aby byly porovnávány výsledky za stejných podmínek. Všechny zmenšené/zvětšené obrázky byly načteny do takzvané batche a poté se na grafické kartě zpracovávala detekce pro celý batch. Prvních pět iterací nebylo počítáno do průměrné frekvence, jelikož probíhá inicializace Tensorflow a dalších závislostí. Výpočet s využitím batche se paralelizoval a je téměř dvakrát rychlejší. U všech výsledků byl hledán maximální batch-size, který grafická karta dokázala zpracovat, jelikož při větším batch-size se zvyšuje využití grafické paměti.

4.1 Přesnost a výkon natrénovaných modelů

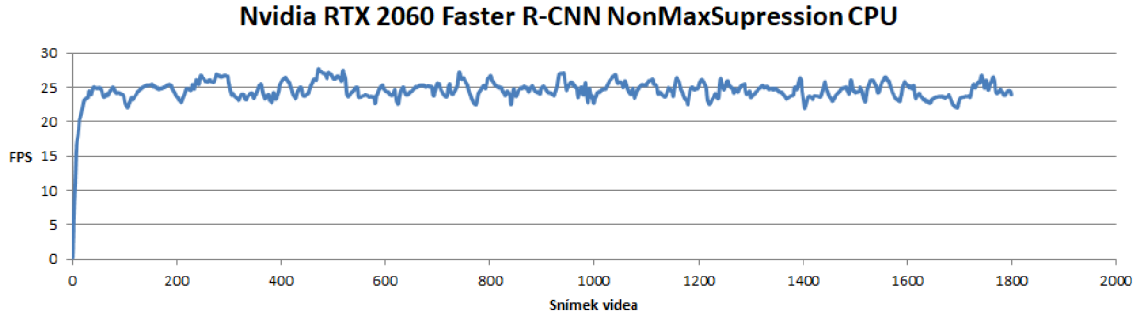
Faster R-CNN

Výkon modelu Faster R-CNN na zařízení Nvidia Tegra TX2 není dostatečný, kdy průměrná snímková frekvence byla bezmála 4 FPS[1], což můžeme vidět na grafu 4.1 znázorňující průběh. Na zařízení Nvidia RTX 2060 si však tento model vedl lépe a jeho průměrná snímková frekvence vyšplhala až na 24.5 FPS viz graf 4.2. Přesnost tohoto modelu byla vyhodnocena jako nejlepší přesností z testovaných modelů, kdy na dvou testovacích snímcích nebylo nalezeno pouze jedno vozidlo. Tento model je použitelný pro zařízení Nvidia RTX 2060 nikoliv však pro zařízení Nvidia Tegra TX2. Nicméně tento model byl vybrán hlavně z důvodu porovnání přesností a to splnil, což můžeme vidět na obrázcích 4.3. Tento model podle očekávání dosáhl největších hodnot přesnosti, takže byl využit a byly vůči němu porovnávány modely SSD Inception V2 a SSD MobileNet V2.

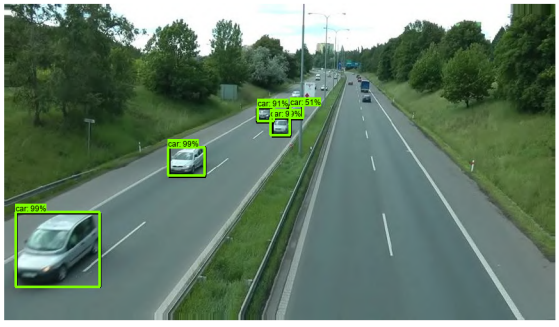
[1]FPS - obrázky za sekundu



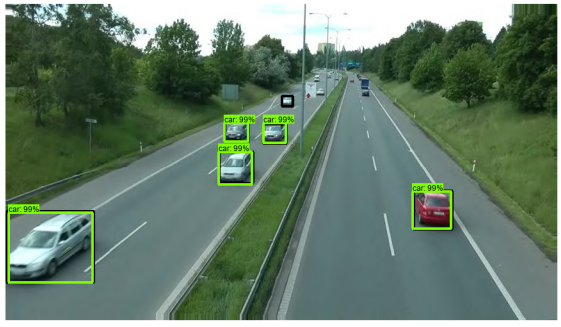
Obrázek 4.1: Model Faster R-CNN, který byl inferencován zařízením Nvidia Tegra Tx2. Průměrná snímková frekvence byla 3.9 FPS. Testování probíhalo na referenčním videu, které je zkráceno na délku jedné minuty. Snímková frekvence zobrazena v tomto grafu zahrnuje i vykreslení.



Obrázek 4.2: Model Faster R-CNN, který byl inferencován na zařízení Nvidia RTX 2060. Testování probíhalo na referenčním videu, které je zkráceno na délku jedné minuty. Snímková frekvence zobrazena v tomto grafu zahrnuje i vykreslení.



(a) Výsledek testovací sady na modelu Faster R-CNN, kde model detekoval všechny projíždějící vozidla.

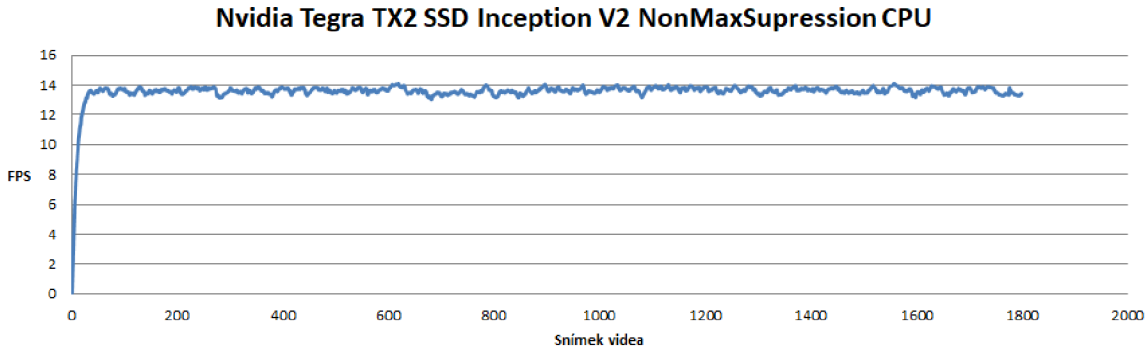


(b) Výsledek testovací sady na modelu Faster R-CNN, kde je vidět jedno nedetekované vozidlo.

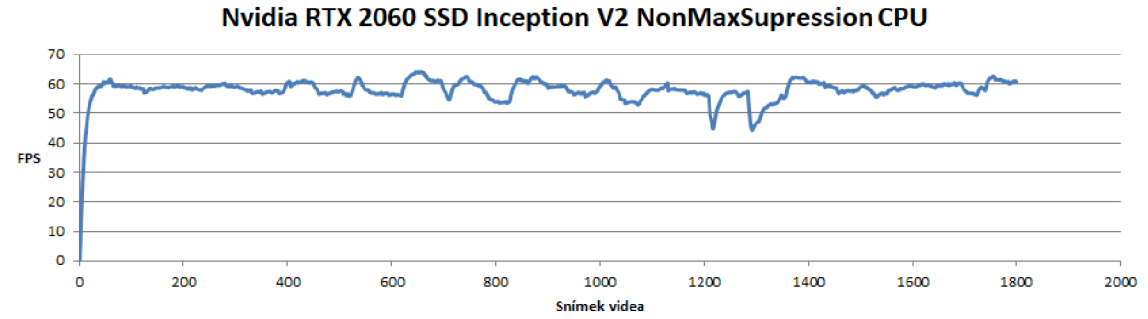
Obrázek 4.3: Výsledky testovací sady na modelu Faster R-CNN.

SSD Inception V2

Výkon modelu SSD Inception V2 byl velmi slušný a použitelný i pro zařízení Nvidia Tegra TX2 viz graf4.4. Bohužel přesnost modelu SSD Inception V2 dosahovala nízkých hodnot přesnosti malých objektů, což je viditelné na obrázku 4.6b z testovací sady. Na dvou obrázcích z testovací sady můžeme vidět, že detektor nenašel čtyři vozidla na obrázku 4.6a. Nicméně výkon na zařízení Nvidia RTX 2060 je obrovský, kdy dosahuje průměrné snímkové frekvence 57.8FPS viz graf 4.5.

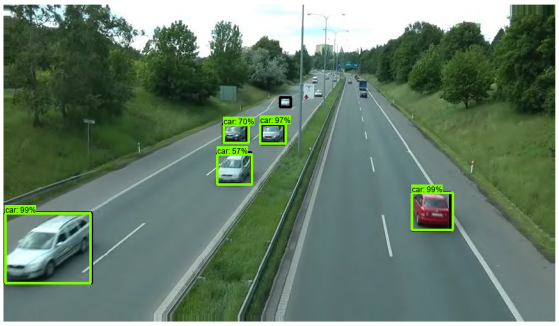


Obrázek 4.4: Model SSD Inception V2, který byl inferencován zařízením Nvidia Tegra Tx2. Průměrná snímková frekvence byla 13.5 FPS. Testování probíhalo na referenčním videu, které je zkráceno na délku jedné minuty. Snímková frekvence zobrazena v tomto grafu zahrnuje i vykreslení.



Obrázek 4.5: Model SSD Inception V2, který byl inferencován zařízením Nvidia RTX 2060. Testování probíhalo na referenčním videu, které je zkráceno na délku jedné minuty. Snímková frekvence zobrazena v tomto grafu zahrnuje i vykreslení.

(a) Výsledek testovací sady na modelu SSD Inception V2, kde jsou vidět tři nedetekovaná vozidla.

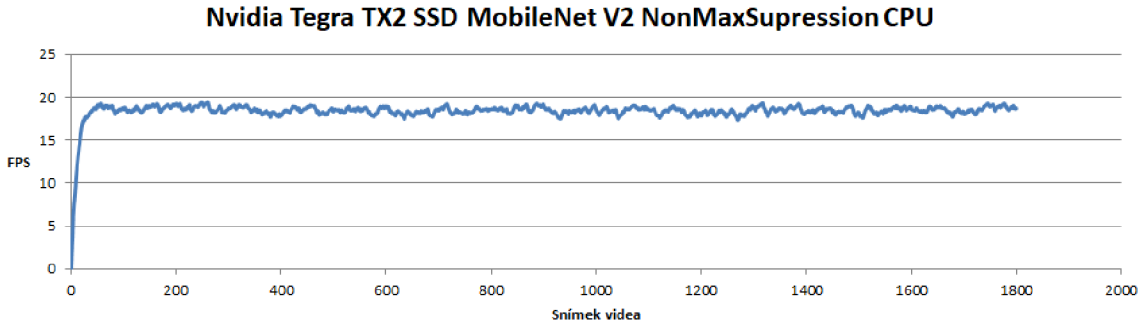


(b) Výsledek testovací sady na modelu SSD Inception V2, kde je vidět jedno nedetekované vozidlo.

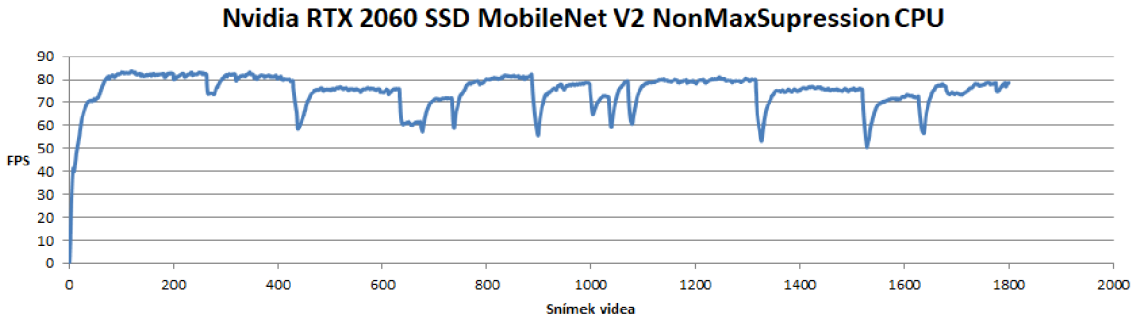
Obrázek 4.6: Výsledky testovací sady na modelu SSD Inception V2.

SSD MobileNet V2

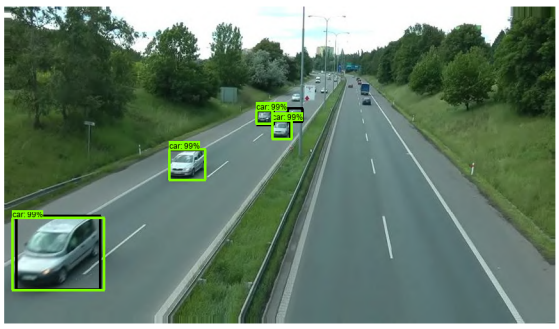
Model SSD MobileNet V2 vyšel pro tuto práci jako nejlepší model pro detekci vozidel v obraze pro zařízení Nvidia Tegra TX2. Tento detektor je dosti spolehlivý i na objekty menší velikosti, což je viditelné na obrázcích 4.9. Navíc se u tento model vyznačil i perfektní přesností vzhledem k výkonu, kdy na datasetu COD20k dosahoval v podstatě stejné přesnosti jako model Faster R-CNN. Výkon tohoto modelu na zařízení Nvidia Tegra TX2 dosahoval nejlepších výsledků, což nám vyplývá z grafu 4.7 průměrná frekvence je 19FPS. Hlavní důvod výběru tohoto modelu byl poměr výkon/přesnost, který byl ze všech použitých nejlepší. Výkon na zařízení Nvidia RTX 2060 byl opravdu vysoký, kdy na grafu 4.8 je videt, že dosahoval průměrné snímkové frekvence 69 FPS. Na obrázcích z testovací sady můžeme vidět, že na dvou obrázcích nenašel pouze dvě vozidla, jež jsou v dálce.



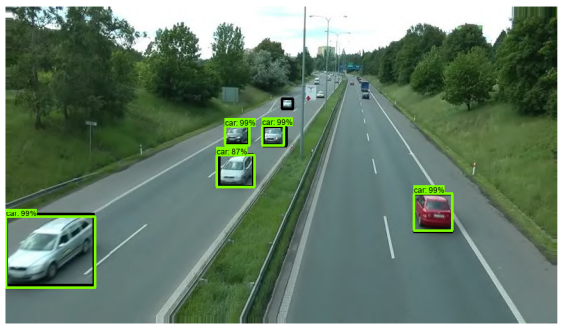
Obrázek 4.7: Model SSD MobileNet V2, který byl inferencován zařízením Nvidia Tegra TX2. Průměrná snímková frekvence je 18.3 FPS. Testování probíhalo na referenčním videu, které je zkráceno na délku jedné minuty. Snímková frekvence zobrazena v tomto grafu zahrnuje i vykreslení.



Obrázek 4.8: Model SSD MobileNet V2, který byl inferencován zařízením Nvidia RTX 2060. Průměrná snímková frekvence byla 69 FPS. Z grafu jsou vidět propady snímkové frekvence, bohužel jsem nezjistil důvod proč se to děje. Nicméně předpokládám, že je to způsobeno pomalým vyčítáním snímků videa. Testování probíhalo na referenčním videu, které je zkráceno na délku jedné minuty. Snímková frekvence zobrazena v tomto grafu zahrnuje i vykreslení.



(a) Výsledek testovací sady na modelu SSD MobileNet V2, kde je vidět jedno vozidlo, které nebylo detekováno.



(b) Výsledek testovací sady na modelu SSD MobileNet V2, kde je vidět jedno vozidlo, které nebylo detekováno.

Obrázek 4.9: Výsledky testovací sady na modelu SSD MobileNet V2

Souhrn přesností a výkonu modelů s aplikací batch-size

Model Faster R-CNN je z testovacích modelů nejpřesnější. Faster R-CNN se vyznačoval hlavně v přesnosti vzdálených vozidel, kdy byl schopen detekovat v podstatě nejmenší objekty z vybraných modelů. SSD Inception V2 vyšel bohužel s přesností na posledním místě. Jeho přesnost hlavně vzdálených(malých) objektů je menší. Bohužel ani přesnost větších objektů není nikterak vysoká. Přesnost modelu SSD MobileNet V2 vzhledem k výkonu je vynikající. SSD MobileNet dosahuje v podstatě stejné, ne-li dokonce větší přesnosti jako model Faster R-CNN.

Model/IoU	Faster R-CNN	SSD Inception V2	SSD MobileNet V2
IoU=0.50:0.95 all	0.396	0.294	0.361
IoU=0.50 all	0.514	0.433	0.547
IoU=0.75 all	0.464	0.352	0.420
IoU=0.50:0.95 small	0.257	0.126	0.223
IoU=0.50:0.95 medium	0.466	0.376	0.440
IoU=0.50:0.95 large	0.442	0.444	0.426

Tabulka 4.1: Výsledné přesnosti modelů, jež byly trénovány na datové sadě COD20k.

Výkon všech modelů byl měřen pouze při inferenci modelu, kdy neprobíhalo žádné vykreslení. Modely byly také optimalizovány pomocí vestavěné sady funkcí TF-TRT, jež jsou součástí TensorFlow tensorflow.python.compiler.tensorrt. Tato součást TensorFlow je vyvíjena v kooperaci Google TensorFlow týmu a Nvidia týmů. Její výhodou je snadné použití na modely, jež jsou součástí TensorFlow. Dále byl vytvořen model, ve formátu ONNX, kdy tento formát je podporován mezi všemi moznými knihovnami. Samotná inference modelu ONNX byla prováděna pomocí rozhraní onnxruntime, kdy byly zapnuty nejvyšší optimalizace modelu.

Zařízení Nvidia RTX 2060 se při inferenci modelu Faster R-CNN vyšplhal nad hranici 30fps i pro batch-size roven jedné. Využití grafické paměti však bylo 5.1GB, po nastavení batch-size na maximální dosaženou hodnotu 3, bylo využití 5.8GB. Po nastavení vyšší batch-size již Tensorflow varoval, že může docházet k poklesu výkonu z důvodu alokace a dealokace zdrojů. To se potvrdilo a výkon byl nižší než při nastavení batch-size 3. Inference modelu ONNX na zařízení Nvidia RTX 2060 přinesla výrazný pokles snímkové frekvence, kdy výkon byl téměř o 35% nižší jako tomu bylo u Tensorflow modelu. Dokonce při zvýšení batch na 3 klesla snímková frekvence oproti batch rovno jedné o 1FPS. Na zařízení Nvidia Tegra TX2 nebyl výkon nikterak vysoký a dosahoval něco málo přes 4FPS, maximální batch-size bylo možné nastavit pouze na hodnotu 1. Po nastavení vyššího batch-size Tensorflow přestal pracovat, s errorem, že není možné alokovat zdroje pro CUDA.

Výkon modelu SSD Inception V2 na zařízení Nvidia RTX 2060 byl velmi dobrý. Na tomto zařízení bylo možné nastavit až 65 batch-size, což naznačuje velmi malému využití grafické paměti neuronové sítě. Při maximálním nastavení batch-size využití grafické paměti dosahovalo 5.8GB. Model ONNX si vedl zase o něco hůř, kdy snímková frekvence na zařízení Nvidia RTX 2060 při batch-size rovno jedné sice stejná, ale po nastavení batch-size na maximální hodnotu byla snímkuvá frekvence o 50% menší než tomu bylo u Tensorflow modelu. Zařízení Nvidia Tegra TX2 s tímto modelem podávala již lepší výsledky, které se pohybovaly kolem 16 FPS. Nicméně bylo možné nastavit hodnotu maximální batch-size na hodnotu 21 což zvýšilo výkon, který přesahoval hranici 30 FPS, což už je velmi slušný výsledek.

Modelu SSD MobileNet V2 na zařízení Nvidia RTX 2060 dosahoval 166 FPS, při nastavení batch-size na hodnotu 1. Využití grafické paměti bylo 2.8GB. Po nalezení maximálního batch-size, kdy nejlepší výsledek byl s hodnotou 41 dosahovalo využití grafické paměti 5.1GB. Po nastavení jakékoliv větší hodnoty Tensorflow varoval, že může docházet k alokování a dealokování zdrojů. Využití grafické paměti při nastavení na hodnotu 42 bylo 5.8GB a výkon klesl. Inference modelu ONNX však nepřinesla srovnatelný výkon, kdy při batch-size nastaveném na jedna byl výkon téměr o 40% nižší než u modelu Tensorflow, problém byl i při nastavení maximálního batche, kdy výkon modelu Tensorflow byl dvakrát větší než modelu ONNX. Nvidia Tegra TX2 si s modelem SSD MobileNet V2 poradila velmi

dobře, výkon i při batch-size 1 je slušný a dosahovala něco málo přes 23 FPS. Nicméně při nastavení batch-size na maximální možnou hodnotu, což v tomto případě bylo 14 se průměrná snímková frekvence vyšplhala na 43 FPS.

Model/Výkon zařízení	Nvidia RTX 2060 [FPS]		Nvidia Tegra TX2 [FPS]	
	Batch=1	Batch=MAX	Batch=1	Batch=MAX
Faster R-CNN	31.98	B3# 40.66	4.19	XXX
TensorRT Faster R-CNN	31.42	B3# 41.12	4.22	XXX
ONNX Faster R-CNN	20.4	B3# 19.12	0.39	XXX
SSD Inception V2	97.61	B65# 252.58	15.60	B21# 32.41
TensorRT SSD Inception V2	96.46	B65# 237.78	15.89	B21# 32.48
ONNX SSD Inception V2	95.12	B65# 156.22	4.37	B21# 4.79
SSD MobileNet V2	175.02	B41# 297.91	22.76	B14# 43.80
TensorRT SSD MobileNet V2	184.47	B41# 300.77	23.17	B14# 43.77
ONNX SSD MobileNet V2	126.29	B41# 154.21	6.06	B14# 7.26

Tabulka 4.2: Výkon zařízení Nvidia RTX 2060 a Nvidia Tegra TX2 na trénovaných modelech. Tyto modely byly také optimalizovány knihovnami TensorRT, kdy z frozen grafu byl vytvořen optimalizovaný graf, jež byl testován stejným scriptem jako neoptimalizovaný graf. Dále byl frozen graph převeden do formátu onnx, který byl inferencován pomocí onnxruntime. Testování probíhalo bez vykreslování, bez cyklického načítání snímků například z videa, kdy snímky byly načteny v inicializaci a poté byly použity. Tím pádem naměřené hodnoty odpovídají opravdu výkon grafické karty byl využit pouze pro výpočty neuronových sítí na jednotlivých modelech.

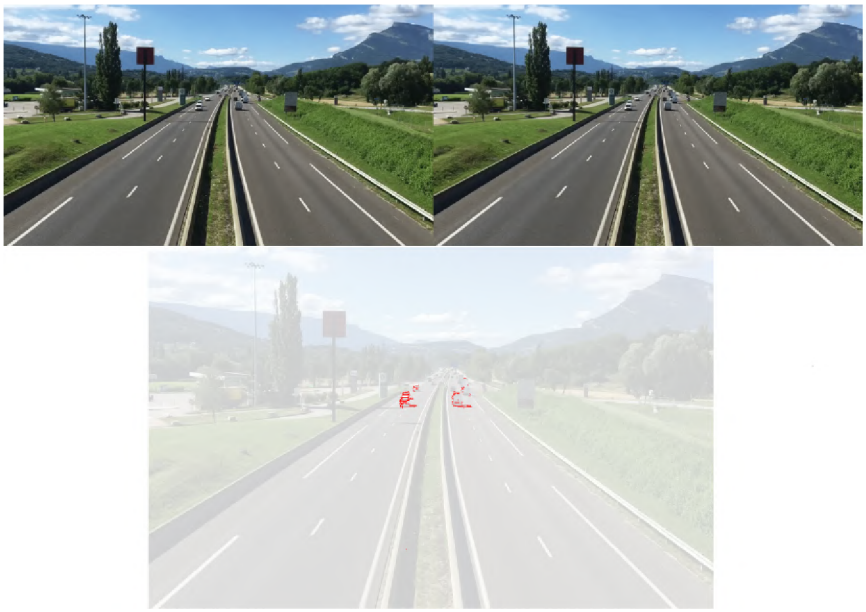
Shrnutí

V této sekci byla nastíněna výkonnost jednotlivých modelů na zařízeních Nvidia RTX 2060 a Nvidia Tegra TX2 a také jejich přesnost detekovat objekty. V první části jsem se věnoval výkonu jednotlivých modelů. V těchto testech bylo zahrnuto i samotné vykreslení do obrazu a další režie s tím spojená. Dále zde byly nastíněny přesnosti jednotlivých modelů, kdy na obrázcích je vidět přesnost vybraných modelů. Nejlépe z hlediska přesnosti vyšel model Faster R-CNN, jež byl schopen detekovat vozidla i ze vzdálenější scény. Naopak model SSD Inception V2 v neuspěl a je vidět, že jeho přesnost detekce není tak dobrá jako u modelu Faster R-CNN. V druhé fázi se ukázal opravdový výkon, kdy byl vytvořen script, jež načte obrázky podle velikosti batch-size s těmi poté pracuje a nevykresluje výsledky. Právě toto testování ukazuje výkon každého modelu na jednotlivých zařízeních. Toto testování přineslo výsledky, ze kterých je již vidět opravdový výkon obou grafických karet. V tomto testování je zřejmé, že výkon grafické karty Nvidia RTX 2060 je přibližně 8x větší než Nvidia Tegra TX2, což potvrzuje teoretickou výkonnost grafických karet, kdy teoreticky je Nvidia RTX 2060 8.6x výkonnější než Nvidia Tegra TX2 ve FP16. Bohužel výkon optimalizovaných modelů je v podstatě stejný jako u modelů neoptimalizovaných. Modely, které byly převedeny do formátu ONNX však nenaplnily očekávání, jejich inference pomocí onnxruntime na grafické kartě Nvidia RTX 2060 ani na Nvidia Tegra TX2 nepřinesla žádné vylepšení, ba naopak dosti velký pokles výkonu oproti inferenci pomocí Tensorflow.

4.2 Možná vylepšení z hlediska výkonu

Detekce pouze lichých nebo sudých snímků

Nvidia Tegra TX2 není dosti výkonné zařízení pro inferenci komplikovanějších neuronových sítí. Na základě toho by mohla být optimalizace například taková, že detekci budeme provádět například na každém lichém snímku, ty budou následně přeneseny do snímků sudých. Nárůst výkonu by byl téměř dvojnásobný, ne roven dvojnásobku. To kvůli tomu, že je nutné snímek přečíst, překreslit detekce a vykreslit. Video, které má snímkovou frekvenci 30 FPS, z toho vyplývá, že změna v obraze se odehraje každých 33ms. Rozdíl těchto po sobě jdoucích snímku je na obrázku 4.10.



Obrázek 4.10: Snímky z videa pro diferenci. Video běží v 30 snímcích za sekundu, to znamená, že časový rozdíl snímků je 33ms.

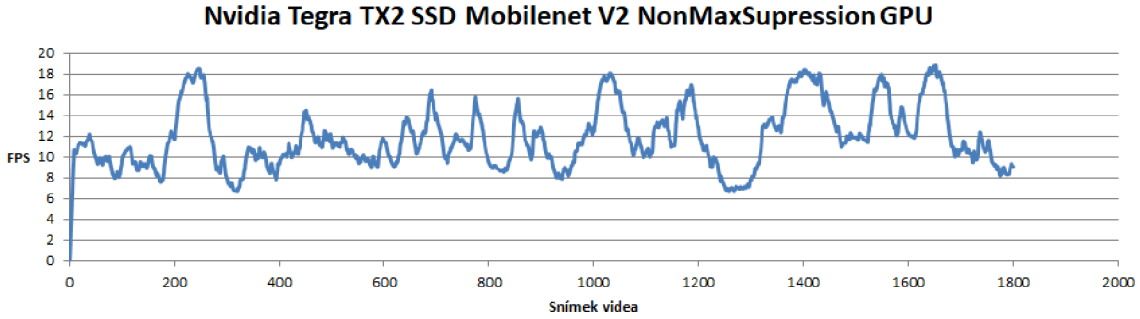
Na obrázku je možné vidět rozdíly, které nastanou za 33ms. Je viditelné, že rozdíl neni nijak drastický. Samozřejmě záleží na situaci, kdy a za jakých okolností má detektor detekovat. Pokud bude detektor na statickém místě a bude detekovat vzdálenější objekty, můžeme se téměř spolehnout na to, že bude fungoval dobře. Pokud se však bude měnit poloha detektoru i detekčních objektů(v mém případě vozidel) bude nutné zvážit, zda k tomu přistoupit. Jinými slovy je důležité vědět, zda potřebujeme výkon či přesnost.

Přesun Non Max Suppression algoritmu na CPU

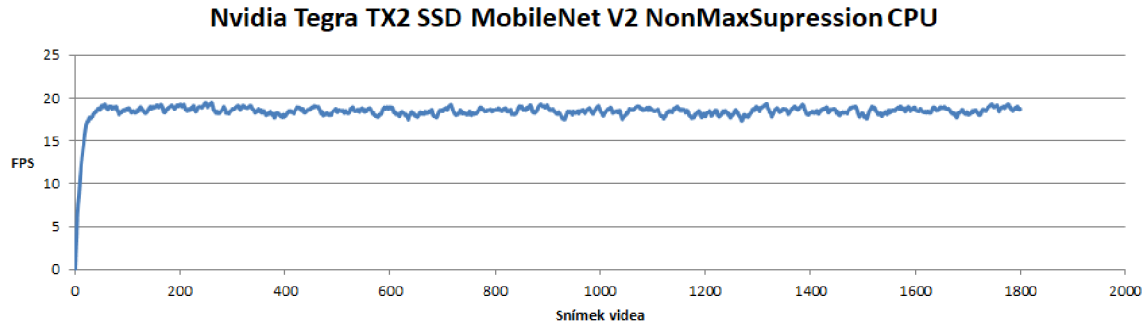
Algoritmus Non Max Suppression sjednotí všechny bounding boxy jednoho objektu, jelikož neuronová síť může vyhodnotit, že vidí dva nebo více objektů. Objekt bude v podstatě jeden a bude ohraničen dvěma nebo více bounding boxy, jako by to byly dva nebo více objektů.

Obrázek 4.11: Ukázka co non max suppression algoritmus dělá. V levé části je snímek, kde jeden objekt má několik ohraničení. V pravé části je snímek, kde ohraničení jsou opraveny algoritmem non max suppression [4]

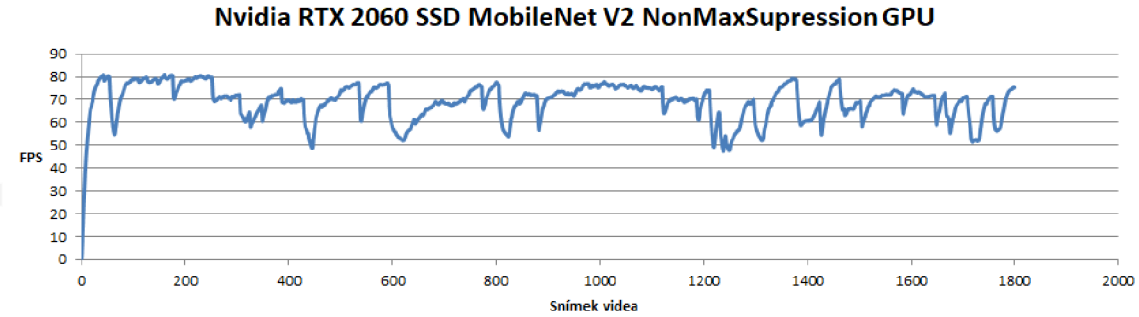
Pro zařízení Nvidia Tegra TX2 mi přišlo vhodné algoritmus Non Max Suppression přesunout na CPU, jelikož grafická karta nedokáže zpracovat tento algoritmus tak rychle jako procesor. Bylo vhodné pro tento algoritmus přesměrovat výpočty na CPU nikoliv na GPU. Kvůli tomu byl naimplementován script v Pythonu, který převede bloky non max suppression v neuronové síti na CPU, každý snímek videa vloží do neuronové sítě. Síť vrátí výsledky, které pak script filtruje pomocí thresholdu. Jakmile jsou bounding boxy vyfiltrované, script jim přiřadí třídu a vykreslí je.



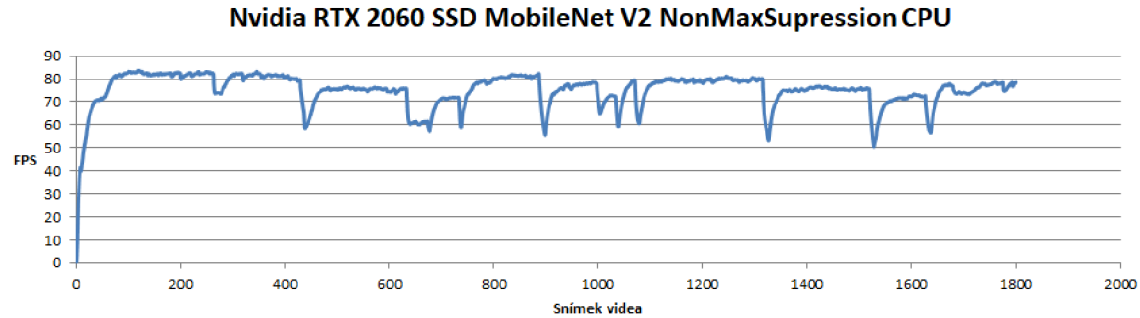
Obrázek 4.12: Inference neuronové sítě, kdy je využit algoritmus Non Max Suppression, který běží na GPU Nvidia Tegra TX2. Snímková frekvence odráží počet detekovaných objektů neuronovou sítí, kdy na ně je aplikován algoritmus Non Max Suppression. Ve špičekách buďto neni detekován žádný objekt, nebo pouze počet objektů v řádu jednotek. Čím více je detekovaných objektů, tím více klesá snímková frekvence.



Obrázek 4.13: Průběh snímkové frekvence, kdy inference probíhá s algoritmem Non Max Suppression, který je přesunut na CPU. Celý průběh je limitován výkonem sítě, tedy grafické karty.



Obrázek 4.14: Inference neuronové sítě, kdy je využit algoritmus Non Max Suppression, který běží na GPU Nvidia RTX 2060. V porovnání s obrázkem 4.15 je průběh snímkové frekvence dosti nestabilní a na více místech padá pod 50 FPS, což je zapříčiněno množstvím vozidel na scéně.



Obrázek 4.15: Průběh snímkové frekvence na zařízení Nvidia RTX 2060, kdy inference probíhá s algoritmem Non Max Suppression, který je přesunut na CPU. Celý průběh je limitován rychlostí vyčítání jednotlivých obrázků videa, kdy vyčítání běži v jednom vlákně. Kvůli tomu jsou viditelné pády snímkové frekvence k 50 FPS.

Kapitola 5

Závěr

Tato Bakalářská práce se zabývala detekcí vozidel v obraze na zařízeních Nvidia Tegra a Nvidia RTX 2060. Kdy výsledkem této práce je přesnost natrénovaných modelů na datové sadě COD20k a výkon jednotlivých modelů na zařízeních Nvidia Tegra TX2 a Nvidia RTX 2060.

V rámci této Bakalářské práce byla vytvořena datová sada. Tato datová sada byla z počátku zamýšlena tak, že by měla být schopná detekovat vozidla v podstatě v jakékoliv scéně. Bohužel datová sada nebyla kompletně dokončena z důvodu velké časové náročnosti získávání dat a hlavně jejich anotací. Kvůli tomu byla vybrána datová sada COD20k, ke které jsou dostupné i anotační soubory testovací sady.

Byly vybrány tři modely z TensorFlow Object Detection API. První model - Model Faster R-CNN porovnával přesnosti dvou dalších modelů, jelikož se vyznačuje velmi slušnou přesností. Dále byl vybrán model SSD Inception V2, a to hlavně kvůli výkonu jelikož model Faster R-CNN nebyl dostatečně výkonný. Následně po natrénování tohoto modelu jsem se rozhodl vyzkoušet ještě model SSD MobileNet V2, jež by měl mít nejlepší poměr výkon/přesnost.

Přesnost natrénovaných modelů byla pro mne překvapením. Byly očekávány horší výsledky, hlavně tedy v oblasti výkonnosti, kdy na Nvidia Tegra TX2 model SSD MobileNet V2 podával velmi dobré výsledky přesnosti a navíc slušný výkon ve videu 720p. Byl natrénován také model SSD Inception V2, který si vedl slušně výkonově, však přesnost vzdálenějších objektů nebyla dostačující. Dále byl natrénován model Faster R-CNN, kdy přesnost tohoto modelu je téměř srovnatelná s modelem SSD MobileNet V2. Velikou nevýhodou tohoto modelu je bohužel jeho výkon, kdy provoz na zařízení Nvidia Tegra TX2 vedl k velmi nízké snímkové frekvenci. Zařízení Nvidia RTX 2060 si vedlo ve všech modelech dobře, nejlépe na tom byl model SSD MobileNet V2. Model Faster R-CNN malinko pokulhával, nicméně tento model není použitelný pro zařízení Nvidia Tegra TX2.

Jelikož hlavním cílem této práce byla detekce vozidel na zařízení Nvidia Tegra TX2, bylo experimentováno hlavně s tímto zařízením. Kvůli tomu muselo dojít k vytvoření scriptu, jež při inferenci modelu využívá algoritmus Non Max Suppression, který je přesunut na CPU, což zařízení přidalo stabilitu snímkové frekvence a navíc i nárust snímkové frekvence až o 50%.

Literatura

[1] *Anaconda* [online]. Dostupné z: "https://www.anaconda.com/distribution/#download-section".

[2] *How single-shot detector SSD works* [online]. Dostupné z: "https://developers.arcgis.com/python/guide/how-ssd-works/".

[3] *How to easily Detect Objects with Deep Learning on RaspberryPi* [online]. Dostupné z: https://mc.ai/how-to-easily-detect-objects-with-deep-learning-on-raspberrypi/.

[4] *Mystery of Object Detection* [online]. Dostupné z: "https://dudeperf3ct.github.io/object/detection/2019/01/07/Mystery-of-Object-Detection/".

[5] *TensorRT Support Matrix* [online]. Dostupné z: "https://docs.nvidia.com/deeplearning/sdk/tensorrt-archived/tensorrt-601/tensorrt-support-matrix/index.html#hardware-precision-matrix".

[6] "François, C. *Keras* [online]. Dostupné z: "https://en.wikipedia.org/wiki/Keras".

[7] gardener" "tensorflower. *TensorFlow* [online]. Dostupné z: "https://github.com/tensorflow/tensorflow".

[8] Girshick", R. *Fast R-CNN* [online]. Duben 2015. Dostupné z: "https://arxiv.org/pdf/1504.08083.pdf".

[9] Gross, S. a Wilber, M. *Training and investigating Residual Nets* [online]. Dostupné z: http://torch.ch/blog/2016/02/04/resnets.html.

[10] Hollemans, M. *Real-time object detection with YOLO* [online]. Dostupné z: https://machinethink.net/blog/object-detection-with-yolo/.

[11] Jong, A. *How to Use ROI Pool and ROI Align in Your Neural Networks (PyTorch 1.0)* [online]. Dostupné z: https://mc.ai/how-to-use-roi-pool-and-roi-align-in-your-neural-networks-pytorch-1-0/.

[12] "Joseph Redmon, R. G. a Farhadi", A. *YOLO - You Look Only Once* [online]. Červen 2015. Dostupné z: "https://arxiv.org/pdf/1506.02640.pdf".

[13] "Kaiming He, P. D. a Girshick", R. *Mask R-CNN* [online]. Březen 2017. Dostupné z: "https://arxiv.org/pdf/1703.06870.pdf".

[14] Karim", R. *Illustrated: 10 CNN Architectures* [online]. Dostupné z: "https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d#6872".

[15] Khandelwal, R. *Computer Vision: Instance Segmentation with Mask R-CNN* [online]. Dostupné z: "https://towardsdatascience.com/computer-vision-instance-segmentation-with-mask-r-cnn-7983502fcad1".

[16] Khandelwal, R. *Computer Vision — A journey from CNN to Mask R-CNN and YOLO -Part 1* [online]. Dostupné z: https://towardsdatascience.com/computer-vision-a-journey-from-cnn-to-mask-r-cnn-and-yolo-1d141eba6e04.

[17] "marksandler2". *Tensorflow detection model zoo* [online]. Dostupné z: "https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md".

[18] "mruberry". *More About PyTorch* [online]. Dostupné z: "https://github.com/pytorch/pytorch#more-about-pytorch".

[19] "Nvidia". *CUDA* [online]. Dostupné z: "https://developer.nvidia.com/cuda-zone".

[20] "Nvidia". *CuDNN* [online]. Dostupné z: "https://developer.nvidia.com/cudnn".

[21] "Nvidia". *TensorRt* [online]. Dostupné z: "https://developer.nvidia.com/tensorrt".

[22] ONNX. *ABOUT* [online]. Dostupné z: "https://onnx.ai/about.html".

[23] Rosebrock, A. *Intersection over Union (IoU) for object detection* [online]. Dostupné z: "https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/".

[24] "Ross Girshick, T. D. a Malik", J. *R-CNN* [online]. Listopad 2013. Dostupné z: "https://arxiv.org/pdf/1311.2524.pdf".

[25] "Shaoqing Ren, R. G. a Sun", J. *Faster R-CNN* [online]. Červen 2015. Dostupné z: "https://arxiv.org/pdf/1506.01497.pdf".

[26] Unknown. *Cars database.zip* [online]. Dostupné z: "https://ulozto.cz/file/4DsvteKpB0CT/cars-database-zip".

[27] "Unknown". *World biggest online cameras directory* [online]. Dostupné z: "https://www.insecam.org/".

[28] VictorDDT. *Raccoon_dataset* [online]. Dostupné z: https://github.com/datitran/raccoon_dataset/blob/master/generate_tfrecord.py.

[29] Wei Liu, D. E. C. S. S. R. C.-Y. F. a Berg, A. C. *SSD - Single shot MultiBox detector* [online]. Prosinec 2015. Dostupné z: "https://arxiv.org/pdf/1512.02325.pdf".

Postup instalace a popis jednotlivých kroků instalace prostředí na x86

Instalace prostředí pro unixový systém Ubuntu 18.4 zahrnuje několik kroků, kdy prvním krokem je instalace ovladačů pro grafickou kartu. Po instalaci grafických ovladačů bylo

nutné zvážit, zda bude vhodné využívat nějaké virtuální prostředí, či nikoliv. Já jsem uznal za vhodné, využít virtuální prostředí Anaconda 3.7 kvůli tomu, že jsem chtěl vyzkoušet více verzí TensorFlow dále již jen TF. Následně po instalaci virtuálního prostředí bylo nutné nainstalovat všechny knihovny, na kterých je TF závislé. Po instalaci knihoven následovala nainstalace softwarové architektury CUDA, která je nutná pro využití grafických karet NVidia v učení a inferenci hlubokých neuronových sítí a dalších. Po instalaci CUDA již mohla proběhnout instalace samotného TF Object Detection API. Po instalaci TF a natrénování několika modelů, následovala instalace TensorRT pro optimalizaci a quantizaci modelů.

Instalace grafických ovladačů

Existuje více metod instalace, já jsem zvolil, instalaci grafického ovladače přes CLI.

```
instalace nvidia driveru
apt search nvidia-driver
sudo apt install nvidia-driver-430
sudo reboot

po restartovani detekce jestli je driver nainstalovan

nvidia-smi
```

Instalace virtuálního prostředí

Jako virtuální prostření se mi nejvíce zamlouvala Anaconda 3.7, z důvodu kompatibility a podpory TF, Numpy a dalších knihoven nutných pro práci s neuronovými sítěmi. Jako první krok byla instalace závislostí pro Anacondu 3.7

```
sudo apt-get install libgl1-mesa-glx libegl1-mesa libxrandr2
sudo apt-get install libxrandr2 libxss1 libxcursor1 libxcomposite1
sudo apt-get install libasound2 libxi6 libxtst6
```

Dalším krokem bylo stažení samotné aplikace Anaconda. [1] Následně bylo nutné ověřit šifrování SHA256 a po kontrole kryptografického hashe jsem začal instalovat.

```
sha256sum <path/filename>
cd ~/Downloads/
bash ./Anaconda3-2019.10-Linux-x86_64.sh
```

Nastavení a vytvoření virtuálního prostředí

Po úspěšné instalaci virtuálního prostředí bylo potřeba vytvořit oddíl virtuálního prostředí, kde se budou instalovat všechny knihovny a závislosti. Jako jazyk virtuálního prostředí byl mnou zvolen Python verze 3.6.

```
conda create -n <virtual\_env\_name> pip python=3.7
```

Po vytvoření virtuálního prostředí proběla aktivace přikazem.

```
conda activate <virtual_env_name>
```

Následovala instalace knihoven TF. Zvolil jsem TF verze 1.15.0, které bylo vydané 16. Srpna 2019. Poté proběhla instalace s využítím programu pip.

```
pip install tensorflow==1.15.0
```

Pokud chceme, aby byl nainstalován TF pro GPU, musíme spustit instalaci příkazem.

```
pip install tensorflow-gpu==1.15.0
```

Testování instalace TensorFlow

Dalším krokem po úspěšné instalaci byla zkouška funkčnosti TF, tzn. vytvoření scriptu, který byl spuštěn pro otestování. Pokud by script nešel spustit nebo byla vyhozená výjimka, je TF špatně nainstalován.

```
import tensorflow as tf
hello = tf.constant('Hello, Tf')
sess = tf.Session()
print(sess.run(hello))
##OUTPUT
'Hello, tf'
```

Instalace CUDA a cuDNN

Pro instalaci knihoven CUDA 10.1.243 byl zvolen software Anaconda.

```
conda install -c anaconda cudatoolkit==10.1.243
```

Po instalaci CUDA proběhla instalace knihovny pro hluboké učení cuDNN 7.6.5. Tyto knihovny obsahují optimalizované metody a funkce pro učení a inferenci hlubokých neuronových sítí.

```
conda install-c anaconda cudnn==7.6.5
```

Následně jsem přidal export cest do souboru /.bashrc, abych se vyhnul spuštění commandů při každém spuštění terminálu.

```
export PATH=/usr/local/cuda-10.1/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda10.1/lib64${}
```

Instalace TensorFlow modelů

Prvním krokem instalace bylo stažení oficiálního repozitáře TensorFlow Models, který obsahuje také knihovny TF Object Detection API. Zvolil jsem opět CLI verzi s nutností doinstalování programu git.

```
sudo apt-get update
sudo apt-get install git
```

Po instalaci programu Git byl stažen repozitář TF Models.

```
mkdir ~/TensorFlow
cd TensorFlow
git clone https://github.com/tensorflow/models
```

Následně po stažení repozitáře bylo zapotřebí program protobuf, který generuje zdrojové kódy ze struktury. Struktura reprezentuje, jak má výsledný kód vypadat. Jsou tři možnosti instalace, kdy první dva typy instalace nepotřebují kompletní sestavení a linkovaní knihoven.

```
conda install -c anaconda protobuf
nebo
pip install protobuf
```

Já jsem volil třetí způsob instalace. Důvodem bylo příliš pomalé načítání modelů v řádu minut v prvnich dvou typech instalace. Když byly všechny knihovny lokálně sestaveny a protobuf nainstalován, načítání modelu trvalo mnohem kratší dobu v řádu sekund.

```
sudo apt-get install autoconf automake libtool curl make g++ unzip -y
git clone https://github.com/google/protobuf.git
cd protobuf
git submodule update --init --recursive
./autogen.sh
./configure
make
make check
sudo make install
sudo ldconfig
```

Instalace trvala poněkud déle, ale to zrychlení načítání modelů stálo za to. Když byl protobuf nainstalován, mohl jsem se vrhnout na instalaci TF Models.

```
cd ~/TensorFlow/models/research
protoc object_detection/protos/*.proto --python_out=.
```

Protobuf vygeneruje source kódy Jakmile protobuf dokončil generování přidal jsem export do souboru /.bashrc. <PATH_TO_TF> značíci cestu k adresáři, kde se TensorFlow vyskytuje.

```
export PYTHONPATH=$PYTHONPATH:<PATH_TO_TF>/TensorFlow/models/research
/object_detection
```

Po zapsání exportu bylo možné buildnout a nainstalovat TF. K tomu jsem použil již vytvořený script setup.py.

```
cd ~/TensorFlow/models/research
python setup.py build
python setup.py install
```

Po dokončení instalace proběhl opět export do souboru /.bashrc.

```
export PYTHONPATH=$PYTHONPATH:<PATH_TO_TF>/TensorFlow/models/research:
<PATH_TO_TF>/TensorFlow/models/research/slim
```

Instalace COCO API

COCO API slouží k vytváření metrik pro neuronové sítě. COCO API se instaluje přes terminál.

```
git clone https://github.com/cocodataset/cocoapi.git
cd cocoapi/PythonAPI
make
cp -r pycocotools <PATH_TO_TF>/TensorFlow/models/research/
```

Instalace TensorRT

Instalace TensorRT byla provedena až po dokončení trénování několika modelů.

```
sudo dpkg -i
nv-tensorrt-repo-ubuntu1804-cuda10.1-trt6.0.1.5-ga-20190913\_1-1\_amd64.deb
sudo apt-key add
/var/nv-tensorrt-repo-cuda10.1-trt6.0.1.5-ga-20190913/7fa2af80.pub
sudo apt-get update
sudo apt-get install tensorrt
sudo apt-get install uff-converter-tf
#### PYTHON 2
sudo apt-get install python-libnvinfer-dev
#### PYTHON 3
```

```
sudo apt-get install python3-libnvinfer-dev
#### verifikace zda je TensorRT nainstalovan
dpkg -l | grep TensorRT
#### VYSTUP
ii graphsurgeon-tf 6.0.1.5-1+cuda10.1 amd64 GraphSurgeon
for TensorRT package
ii libnvinfer-dev 6.0.1.5-1+cuda10.1 amd64 TensorRT development
libraries and headers
ii libnvinfer-samples 6.0.1.51+cuda10.1 amd64 TensorRT
samples and documentation
ii libnvinfer5 6.0.1.5-1+cuda10.1 amd64 TensorRT runtime
libraries
ii python-libnvinfer 6.0.1.5-1+cuda10.1 amd64 Python
bindings for TensorRT
ii python-libnvinfer-dev 6.0.1.5-1+cuda10.1 amd64 Python
development package for TensorRT
ii python3-libnvinfer 6.0.1.5-1+cuda10.1 amd64 Python 3
bindings for TensorRT
ii python3-libnvinfer-dev 6.0.1.5-1+cuda10.1 amd64 Python 3
development package for TensorRT
ii tensorrt 6.0.1.5-1+cuda10.1 amd64 Meta package of TensorRT
ii uff-converter-tf 6.0.1.5-1+cuda10.1 amd64 UFF converter
for TensorRT package
```

Instalace LabelImg pro anotace obrázků

Pro vytváření anotací a anotování dat jsem zvolil nástroj LabelImg, který se mi jevil pro moje využití jako nejlepší volba.

```
cd ~/TensorFlow
mkdir preprocess
cd preprocess
git clone https://github.com/tzutalin/labelImg
sudo apt-get install pyqt5-dev-tools
cd labelImg
sudo pip install -r requirements/requirements-linux-python3.txt
make qt5py3
```