VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2020 Tomáš Macko

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEPARTMENT OF CONTROL AND INSTRUMENTATION

NÁVRH UČÍCÍHO A VYBAVOVACÍHO MODULU S VYUŽITÍM AI NA PLATFORMĚ RASPBERRY PI A INTEL MOVIDIUS

DESIGN OF LEARNING AND EQUIPMENT MODULE USING AI ON RASPBERRY PI AND INTEL MOVIDIUS PLATFORM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — Tomáš Macko

AUTHOR

VEDOUCÍ PRÁCE — Ing. Ilona Janáková, Ph.D.

SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Tomáš Macko ***ID:*** 197721

Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Návrh učícího a vybavovacího modulu s využitím AI na platformě Raspberry PI a Intel Movidius

POKYNY PRO VYPRACOVÁNÍ:

Úkolem je návrh učícího a vybavovacího modulu jako rozšíření stávajícího řešení kamerového checkeru na platformě Raspberry PI a import externích již naučených modelů. Programovací jazyk je Python s knihovnami OpenCV, Keras a TensorFlow. Předpokládá se použití platformy Intel Movidius.

1. Seznamte se s metodami a aplikacemi strojového učení v počítačovém vidění.
2. Seznamte se s platformami Intel Movidius a Raspberry PI. Seznamte se s programováním v jazyce Python a vybranými knihovnami.
3. Pro danou úlohu proveďte sběr obrazových dat pro trénování a validaci. Případně proveďte vhodné předzpracování obrazových dat.
4. Vyberte vhodný model pro klasifikaci. Model natrénujte a naimportujte do Intel Movidius. Ověřte úspěšnost modelu a zhodnoťte rychlost vyhodnocení.
5. Navrhněte nebo rozšiřte stávající obslužnou aplikaci tak, aby využívala Movidius a naučený model pro klasifikaci a zároveň, aby splňovala standardní definované průmyslové požadavky.
6. Výsledky zpracujte, stanovte omezující podmínky, zhodnoťte.

DOPORUČENÁ LITERATURA:

Intel® Neural Compute Stick. Intel Software [online]. 22.8.2018 [cit. 2019-09-13]. Dostupné z: https://software.intel.com/en-us/neural-compute-stick

ONDRÁŠEK, David. Implementace algoritmu hlubokého učení na embedded platformě. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, FEKT. Vedoucí práce Karel Horák.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 8.6.2020

Vedoucí práce: Ing. Ilona Janáková, Ph.D.

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

ABSTRAKT

Táto bakalárska práca sa venuje implementácii naučeného modelu neurónovej siete na platformu akcelerátoru neurónových sietí - Intel Movidius. Prvá kapitola sa venuje teórii strojového učenia, počítačového videnia a technikám, ktoré tieto dva obory využívajú. Druhá kapitola popisuje výber vhodných nástrojov v podobe programovacieho jazyku a dodatočných knižníc na programovanie konvolućných neurónových sietí. Tretia a štvrtá kapitola spolu pomerne úzko súvisia. Popisujú použitý hardware, problematiku jeho inštalácie a všetky vyžadované balíčky. V ďalšej kapitole sú vyobrazené ukážky dát a spôsob ich spracovania. Ďalej nasleduje popis modelov neurónovej siete a všetkých skriptov, ktoré pri tejto práci vznikli. Poslednými kapitolami sú spracovanie nameraných dát pri trénovaní a testovaní modelov a ich celkové vyhodnotenie.

KĽÚČOVÉ SLOVÁ

Konvolučná neurónová sieť, umelá inteligencia, strojové učenie, počítačové videnie, Raspberry Pi, Intel Movidius Neural Compute Stick, klasifikácia dát

ABSTRACT

This bachelors thesis describes the process of implementing trained neural network model to AI accelerator - Intel Movidius. The first chapter is about machine learning and computer vision theory. The second chapter describes the options which can be chosen for programming of convolutional neural networks as programming language or related libraries which suit the most. The third and fourth chapters are highly connected. They describe the whole process of hardware installation and troubleshooting of software issues during installation. The next chapter shows previews of images, which are used as data input for neural network. Next pages describe used scripts and models of neural networks which were created from scratch. The last chapters are all about measured datas during the training or testing of neural networks and its evaluation.

KEYWORDS

Convolutional Neural Network, Artificial intelligence, Machine Learning, Computer Vision, Raspberry Pi, Intel Movidius Neural Compute Stick, Image Classification

MACKO, Tomáš. *Návrh učícího a vybavovacího modulu s využitím AI na platformě Raspberry PI a Intel Movidius*. Brno, 2020, 81 s. Bakalárska práca. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedúci práce: Ing. Ilona Janáková, Ph.D.

VYHLÁSENIE

Vyhlasujem, že svoju bakalársku prácu na tému „Návrh učícího a vybavovacího modulu s využitím AI na platformě Raspberry PI a Intel Movidius“ som vypracoval samostatne pod vedením vedúceho bakalárskej práce, s využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej bakalárskej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto bakalárskej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora

POĎAKOVANIE

Veľmi rád by som poďakoval vedúcej bakalárskej práce pani Ing. Ilone Janákovej, Ph.D. za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Obsah

Úvod **12**

1 Strojové učenie v počítačovom videní **14**
1.1 Strojové učenie . . . 14
1.1.1 Učenie s učiteľom . . . 15
1.1.2 Učenie bez učiteľa . . . 15
1.1.3 Spätnoväzebné učenie . . . 16
1.2 Neurónové siete . . . 16
1.2.1 Konvolučné neurónové siete . . . 17
1.3 Počítačové videnie . . . 21
1.4 Vzťah medzi strojovým učením a počítačovým videním . . . 22
1.4.1 Klasifikácia obrazu . . . 22
1.4.2 Klasifikácia obrazu s lokalizáciou objektu . . . 22
1.4.3 Detekcia objektov . . . 23
1.4.4 Rekonštrukcia obrazu . . . 24
1.4.5 Trekovanie objektov . . . 25

2 Software-ové nástroje pre strojové učenie **26**
2.1 Výber vhodného programovacieho jazyka . . . 26
2.1.1 Python ako programovací jazyk . . . 26
2.2 Prehľad najpoužívanejších knižníc pre strojové učenie . . . 27
2.2.1 Keras . . . 27
2.2.2 TensorFlow . . . 27
2.2.3 Caffe . . . 28

3 Použitý hardware **30**
3.1 Akcelerátory neurónových sietí . . . 30
3.1.1 Intel Movidius Neural Compute Stick . . . 31
3.1.2 Neural Compute Software Development Kit . . . 31
3.1.3 Implementácia modelu na Intel Movidius Neural Compute Stick 1 . . . 32
3.2 Hlavná výpočetná jednotka . . . 32
3.2.1 Raspberry Pi . . . 33

4 Implementácia Neural Compute SDK **35**
4.1 Spôsoby inštalácie NCSDK . . . 35
4.1.1 Virtual Machine . . . 35
4.1.2 Virtual Environment . . . 35
4.1.3 Docker . . . 36
4.2 NCSDK pre Ubuntu 16.04 . . . 36
4.3 NCSDK pre Raspbian Stretch . . . 37

4.3.1 Inštalácia pre Raspberry Pi 3 Model B+ . . . 37
4.3.2 Nekompatibilita s Raspberry Pi Model 1 B+ . . . 38
4.4 Používanie NCSDK . . . 38
4.4.1 Aktivácia NCSDK . . . 39
4.4.2 Nástroje NCSDK . . . 39
4.4.3 Výnimky vyvolané NCSDK . . . 41

5 Dáta použité na natrénovanie modelu neurónových sietí 43
5.1 Rozdelenie použitých dát . . . 43
5.1.1 Kategória *535* . . . 43
5.1.2 Kategória *535-2* . . . 44
5.1.3 Kategória *blue* . . . 44
5.1.4 Kategória *dacar524* . . . 45
5.1.5 Kategória *HT* . . . 45
5.1.6 Kategória *manual_cut* . . . 46
5.2 Formát použitých dát . . . 46
5.3 Vytvorenie validačných dát . . . 47
5.4 Zvýšenie počtu tréningových dát - data augmentation . . . 47

6 Modely neurónových sietí 49
6.1 InceptionV3 model neurónovej siete od firmy Tyco Electronics Czech s.r.o. . 49
6.1.1 Skript scrape_hdf5.py . . . 49
6.1.2 Skript keras_to_tf.py . . . 50
6.1.3 Kompilácia modelu a vytvorenie graph súboru . . . 50
6.2 Sequential model neurónovej siete . . . 51
6.2.1 Prehľad modelu . . . 51
6.2.2 Implementácia modelu . . . 54
6.2.3 Kompilácia modelu a vytvorenie graph súboru . . . 55
6.3 VGG16 model neurónovej siete . . . 55
6.3.1 Prehľad modelu . . . 56
6.3.2 Implementácia modelu . . . 57
6.3.3 Kompilácia modelu a vytvorenie graph súboru . . . 58

7 Pomocné skripty 60
7.1 plot.py . . . 60
7.2 evaluate_on_cpu.py . . . 61
7.3 evaluate_on_movidius.py . . . 63

8 Výsledky 64
8.1 Proces trénovania . . . 64
8.2 Proces testovania . . . 66
8.2.1 Úspešnosť jednotlivých konfigurácií modelov . . . 66
8.2.2 Čas potrebný na testovanie pre jednotlivé platformy . . . 69

Závěr **73**

Literatúra **74**

Zoznam symbolov, veličín a skratiek **81**

Zoznam obrázkov

1.1 Neurónová sieť . . . 17
1.2 Konvolučná funkcia . . . 18
1.3 InceptionV3 architektúra . . . 20
1.4 VGG16 architektúra . . . 20
1.5 Graf *Počítačové videnie – Computer Vision* (CV) . . . 21
1.6 Klasifikácia vs. klasifikácia s lokalizáciou . . . 23
1.7 Detekcia objektov . . . 24
1.8 Trekovanie objektov . . . 25
2.1 Architektúra knižnice TensorFlow . . . 28
3.1 NCS . . . 31
3.2 NCS1 kroky . . . 33
3.3 Raspberry Pi 3 Model B+ s Intel Movidius . . . 34
4.1 mvNCCheck . . . 40
4.2 mvNCProfile_textfile . . . 41
5.1 Ukážky z kategórie *535* . . . 44
5.2 Ukážky z kategórie *535-2* . . . 44
5.3 Ukážky z kategórie *blue* . . . 45
5.4 Ukážky z kategórie *dacar524* . . . 45
5.5 Ukážky z kategórie *HT* . . . 46
5.6 Ukážky z kategórie *manual_cut* . . . 46
6.1 model.summary . . . 50
6.2 ValueError: Unsupported shapes . . . 51
6.3 Architektúra vlastného sequential modelu . . . 52
6.4 Ukážka plne prepojených neurónových vrstiev . . . 53
6.5 Architektúra VGG16 modelu . . . 56
6.6 VGG16 mvNCCheck . . . 59
7.1 Accuracy a Loss graf . . . 60
7.2 Accuracy/Loss graf . . . 61
7.3 Confusion Matrix . . . 62
8.1 Loss/Accuracy graf Sequential model s 0,001 learning rate . . . 65
8.2 Loss/Accuracy graf Sequential model s 0,001 learning rate . . . 65
8.3 Sequential confusion matrix . . . 68
8.4 VGG16 confusion matrix . . . 68

Zoznam tabuliek

8.1 Accuracy/Loss 64
8.2 Doba trénovania modelu 65
8.3 Úspešnosť testovania jednotlivých konfigurácií modelov 1/2 66
8.4 Úspešnosť testovania jednotlivých konfigurácií modelov 2/2 67
8.5 Úspešnosť testovania InceptionV3 architektúry 67
8.6 Čas potrebný pre testovanie dát na MacBook Pro 69
8.7 Čas potrebný pre testovanie dát na HP Elite Book 69
8.8 Čas potrebný pre testovanie dát na Raspberry Pi 3B+ 70
8.9 Čas potrebný pre testovanie InceptionV3 architektúry 70
8.10 Čas potrebný pre testovanie dát na HP Elite Book s použitím Intel Movidius 71
8.11 Čas potrebný pre testovanie dát na Raspberry Pi 3B+ s použitím Intel Movidius 71

Zoznam výpisov

4.1 Bash príkazy pre inštaláciu *dask 2.0.0* zo zdroja. 38
5.1 Objekt triedy *ImageDataGenerator* 48
6.1 Freeze modelu VGG16 57

Úvod

Počítačové videnie a všeobecne umelá inteligencia sú oborom, ktorý je čoraz viac žiadaný a obľúbený. Slovné spojenie umelá inteligencia sa prvýkrát skloňovalo v roku 1956 na konferencii, kde prednášal John McCarthy. Avšak s podobnou myšlienkou, akou sa umelá inteligencia zaoberá, sa ľudia pohrávali dávnejšie. Napríklad Alan Turing s Davidom Champernowne v roku 1948 vydali prácu, ktorá popisovala algoritmus v šachu, ktorým by mohol byť stroj schopný poraziť človeka. Jeho výpočetná náročnosť však spôsobila, že pokus vytvoriť na základe algoritmu spustiteľný program, stroskotal. [1]

Za posledné desaťročia využitie umelej inteligencie výrazne vzrástlo. Dôvodom je práve prudký nárast výpočetnej sily, ktorá umožňuje spracovávať čoraz zložitejšie algortimy. Počítačové videnie má využitie v rôznych oboroch. Začínajúc v zdravotníctve, bezpečnosti, alebo automatizácii, až po robotiku, alebo šport. Vstupom pre tento obor je obrazový materiál v podobe fotiek, obrázkov, kamerových záznamov alebo kresieb, na ktorých sú detekované objekty, alebo rozpoznávané tvary. Často však tieto dáta musia byť spracované čo najrýchlejšie, a s čo najväčšou presnosťou. S tým veľmi úzko súvisí aj *hardware*, respektíve technológie používané na tieto výpočty.

V poslednej dobe je pomerne veľký trend vyvýjať a vyrábať stále menšie a menšie zariadenia, s čoraz väčšou výpočetnou silou. Pokiaľ to technológia umožňuje, tak je stále viac populárny aj prenájom virtuálnych strojov a presun výpočtov na *cloud*. Lenže to nieje vhodné vo všetkých situáciách a pre každé využitie. Prístroj, ktorý má za úlohu spracovať dáta a následne ich vyhodnotiť, nemusí mať pripojenie k internetu. Alebo miesto, kde sa tieto aplikácie využívajú, môžu byť limitované rozmerami. Peniaze hrajú tak isto významnú rolu. Môže sa stať, že spoločnosť, ktorá chce používať svoju kameru a následne obraz spracovávať a vyhodnocovať, má prístup k internetu. Rovnako tak má aj veľké priestory, kde by bol HW uložený. Ale nemôže si dovoliť nákup takého zariadenia.

Všetky tieto okolnosti zobrali do úvahy niektoré veľké korporáty, ako napríklad *Google LLC*, alebo *Intel Corporation*. Začali preto vyvýjať zariadenia, ktorých úlohou bolo, v čo najmenšom tele ponúknuť veľkú výpočetnú silu pre spracovanie neurónových sietí. To všetky pri zachovaní rozumnej ceny. Postupne tak vznikali *akcelerátory neurónových sietí - AI accelerators*. Ich hlavnou úlohou je ponúknuť svoju výpočetnú silu zariadeniu, ku ktorému sú pripojené. Takže nieje potrebné mať zakúpený HW za vysoké čiastky. Stačí mať akcelerátor pripojený k zariadeniu. Na akcelerátori, ako na optimalizovanom a pre neurónové siete uspôsobenom zariadení, sa následne spracovávajú potrebné výpočty.

Tejto téme sa venuje práve táto práca. Hlavnou úlohou je dosiahnuť spoluprácu a komunikáciu medzi akcelerátorom a svojími rozmerami malým zariadením - *Raspberry Pi*. Táto kombinácia následne môže byť využívaná vo vyššie spomenutých odvetviach, za úče-

lom prekonania daných limitácií. Súčasťou práce je aj naprogramovanie a natrénovanie modelov neurónových sietí na klasifikáciu dát do šiestich kategórií. Za účelom porovnania kombinácie *akcelerátoru* a *Raspberry Pi* s ostatnými platformami, boli stanovené dáta ako doba trénovania modelu neurónovej siete, jeho úspešnosť a doba testovania. Z týchto hodnôt potom bolo možné odvodiť, či a poprípade aký veľký prínos má použitie akcelerátoru neurónových sietí.

1 Strojové učenie v počítačovom videní

Strojové učenie – Machine Learning (ML) v počítačovom videní je odvetvím, ktoré vzbudzuje čoraz väčší záujem ľudí pracujúcich v informatike a oboroch jej príbuzným. Nemusia to byť len vedci, ale aj ľudia, ktorí si chcú založiť firmu - *startup*, pretože samotné ML ponúka veľmi veľké možnosti. Zameriava sa na riešenie celej rady problémov z reálneho života o rôznej obtiažnosti. Algoritmy, ktoré využíva sú veľmi blízke ľudskému biologickému videniu. [2]

1.1 Strojové učenie

ML je jednou z čaští alebo podkategóriou *umelej inteligencie*. Využíva sa v systémoch, ktoré nemajú presne definovaný postup a inštrukcie ako daný problém vyriešiť. Namiesto toho pracuje so *štatistickými modelmi* a v kombinácii s vhodnými algoritmami rieši špecifický problém. Hlavne aplikácia vhodného algoritmu pre danú úlohu je kritickou časťou. Nie všetky algoritmi sa dajú aplikovať pre riešenie daného problému. Medzi často používané je možné zaradiť:

- Lineárna regresia:
  Algoritmus, ktorý popisuje vzťah dvoch premenných na základe rovnice $Y = a*X + b$, pričom Y je závislá premenná a X naopak nezávislá. Takže algoritmus lineárnej regresie predpovedá hodnotu závislej premennej na základe premennej nezávislej. Rovnica reprezentuje priamku.
- Rozhodovacie stromy:
  Tento druh algoritmu môže byť použitý ako na klasifikáciu, tak aj na regresiu. Má stromovú štruktúru a jeho koreňom je atribút, ktorý nosí najväčšie množstvo informácií o danom probléme. Tréningová množina dát je ďalej rozdelená na podmnožiny. To závisí na obsiahnutých informáciách v rámci daných množín. Proces sa opakuje až pokiaľ sa všetky dáta neklasifikujú a pokiaľ sa nenájde list každej vetvy.
- Naive-Bayes algoritmus:
  Tento algoritmus vychádza z *Bayes*-oveho teorému, ktorý je používaný poväčšine pre vysoko sofistikované klasifikačné metódy. Učí sa pravdepodobnosť, z akou skúmaný objekt s danými vlastnosťami patrí do žiadanej skupiny. Pri tejto technike je výskyt každej vlastnosti nezávislí na výskyte ostatných. [3]
- Neurónové siete:
  Tento algoritmus sa môže zaradiť medzi absolútne najpopulárnejšie ML algoritmy súčastnosti. Spolu s faktom, že *neurónové siete* boli naimplementované v tejto práci, bude im v ďalšej časti venovaná podkategória.

Na to aby *Umelá inteligencia – Artificial Intelligence* (AI) mohla napredovať sa musí vedieť učiť. To je presne úloha s ktorou jej pomáha ML. ML preto nachádza využitie v

software-ovom inžinierstve, počítačovom videní alebo rozpoznávaniu vzorov - *pattern recognition*. Podľa *typu učenia* vieme ML rozdeliť do nasledujúcich kategórií:

- Učenie s učiteľom
- Učenie bez učiteľa
- Spätnoväzebné učenie

1.1.1 Učenie s učiteľom

Tento typ učenia môžeme považovať za najrozšírenejší, pretože je pomerne jednoduché si predstaviť jeho funkcionalitu. Rovnako najjednoduchšie zo všetkých kategórií sa dá daný algortimus naimplementovať.

Vstupom pre algoritmus sú dáta v tvare - *príklad* toho na čo má byť algoritmus natrénovaný a *označenie* daného príkladu. Algoritmus následne vyhodnotí kam tento príklad patrí a v zápetí dostane spätnú väzbu k jeho predpovedi, či bola správna alebo nie. Postupným iteratívnym vyhodnocovaním vstupných dát a spracovávaním spätnej väzby ktorá sa mu dostane je schopný stále viac aproximovať vzťah medzi samotnými príkladmi a ich označeniami. Takto plne naučení algoritmus potom dokáže predpovedať označenie príkladu, ktorý mu nikdy predtým nebol predložený.

Avšak tento typ učenia je vhodný na spracovanie iba jednej konkrétnej úlohy na ktorú je trénovaný. Ako príklad sa dá uviezť *cielená reklama*, ktorá sa zobrazuje uživateľom v aplikáciách alebo na webe, alebo *rozpoznávanie tváre* spracovávané rôznymi sociálnymi sieťami. [4]

1.1.2 Učenie bez učiteľa

Učenie bez učiteľa na rozdiel od kategórie spomenutej vyššie, je špecifické tým, že vstupné dáta nemajú označenie podľa ktorého by sa algoritmus učil. Tým pádom nemá ani fázu trénovania. Výstupom takýchto algoritmov teda niesú konkrétne označenia do ktorých dáta patria.

Algoritmy využívajúce metódu *učenia bez učiteľa* sa snažia nájsť podobnosti vo vstupných dátach a na základe nich ich porozdeľovať do kategórií. V zásade najbežnejšou metódou je *clustering* - zhlukovanie podľa podobných vlastností.

Tento typ algorimtu je používaný v bioinformatike na *sekvenčnú analýzu*, v medicíne na *segmentáciu obrazu* alebo v počítačovom videní na *rozpoznávanie objektov*. [5]

1.1.3 Spätnoväzebné učenie

Spätnoväzebné učenie je druh trénovania modelu tak, aby bol schopný vykonať postupnosť rozhodnutí, ktoré ho privedú do *cieľa*. Cieľom sa myslí napríklad naučiť autonómne šoférovať vozidlo alebo poraziť súpera v nejakej hre.

Za každý krok, ktoré model urobí je buď odmenený alebo penalizovaný. Na dosiahnutie správneho výsledku model maximalizuje svoju odmenu za krok, ktorý spravil. Okrem ohodnotenia v podobe penalizácie alebo odmeny programátor nedáva žiadne ďalšie rady modelu ako sa dostať k cielenému výsledku. Ako už z toho vyplýva, model začína z úplne náhodného štádia.

Spätnoväzebné učenie patrí medzi najefektívnejšie spôsoby ako naučiť model kreativite. Pri porovnaní so živým tvorom, model má možnosť byť trénovaný paralelne na tisíckach rôznych situáciách a na veľmi výkonných strojoch. [6]

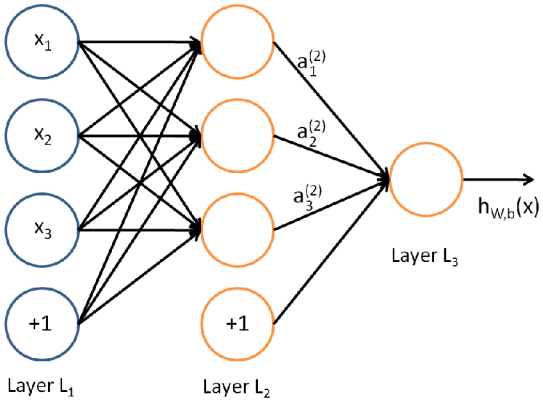
1.2 Neurónové siete

Ako už bolo spomenuté, *Neurónové siete – Neural Networks* (NN) patria medzi najobľúbenejšie a najpoužívanejšie techniky *strojového učenia*. V priebehu času bolo postupne dokázané, že NN prekonávajú iné algoritmy či už v presnosti, ale aj rýchlosti. Momentálne existujú rôzne variácie NN. Začínajúc *Convolučné neurónové siete – Convolutional Neural Networks* (CNN), *Rekurentné neurónové siete – Recurrent Neural Network* (RNN) až po neurónové siete pre *deep learning*.

Ako už z názvu vyplýva, NN sú inšpirované biologickou nervovou stavbou ľudského mozgu. Rovnako ako pri človeku, základnou stavebnou jednotkou je *neurón*. Ten je definovaný aktivačnou funkciou. V princípe, na vstupe sa objaví nejaká hodnota. Na ňu sa následne aplikuje aktivačná funkcia a tento výsledok sa následne objaví na výstupe. Kolekcia neurónovom následne tvorí *vrstvy* NN. Obrázok 1.1 ukazuje príklad veľmi jednoduchej *neurónovej stiete*.

Vrstva na ľavej strane je *vstupnou vrstvou*. Na pravej strane je to *výstupná vrstva*. V tomto prípade ide iba o jeden neurón. Ostatné vrstvy medzi tým sa nazývajú *skryté vrstvy*, pretože ich hodnoty niesú obsiahnuté vo vstupných dátach. Každá neurónová vrstva obsahuje vstupnú a výstupnú vrstvu. Počet skrytých vrstiev zasa záleží na zložitosti riešeného problému. Rovnako je dobré spomenúť, že nemusí mať každá skrytá vrstva rovnaký typ *aktivačnej fukncie*.

Aby NN dokázala správne reagovať na vstup, musí pre každú vrstvu nastaviť vhodné *váhy*. Váhou sa rozumie parameter, ktorým je vynásobený výstup predošlej vrstvy a následne je táto hodnota predaná ako vstup pre nasledujúcu vrstvu. Neurónové siete, ktoré



Obr. 1.1: Jednoduchý príklad architektúry neurónovej siete. [7]

obsahujú viacero skrytých vrstiev sa často označujú ako *Deep Neural Networks*. [7]

Keďže na riešenie problému, ktorým sa zaoberá táto práca bola použitá CNN, je vhodné ju tak isto trochu priblížiť.

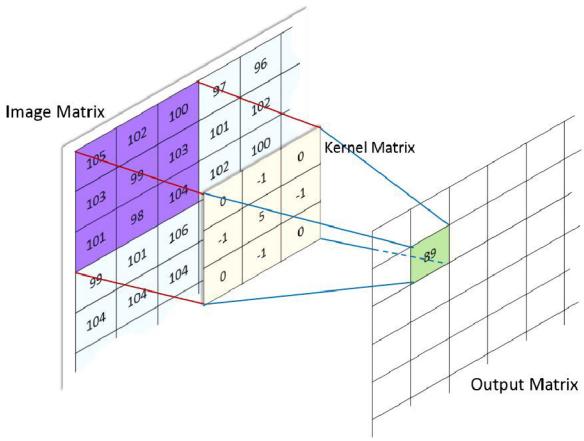
1.2.1 Konvolučné neurónové siete

Konvolučné neurónové siete sú jendou z variánt neurónových sietí hlbokého učenia - *deep-learning*, používaných prevažne v *počítačovom videní*. Ich názov je odvodený od typu skrytých vrstiev. Typicky pozostávajú z *konvolučných*, *pooling*-ových, *plne prepojených (fully-connected)*, alebo *normalizačných* vrstiev. V praxi to znamená, že sa miesto váh používajú konvolučné filtre. V nasledujúcich odstavcoch sú spomenuté tie najpoužívanejšie vrstvy.

Konvolučná vrstva využíva aplikáciu *konvolučnej* funckie na dáta. *Konvolučná* funkcia operuje v 1D na dvoch signáloch, pre dvojrozmerný priestor na obraze a jadre. Jednou časťou je obraz, tou druhou je práve *jadro*, ktoré pripomína akýsi filter veľkosti napríklad *3x3* alebo *5x5*. Ten sa následne posúva po celom obraze. Matematicky je to vyjadrené ako skalárny súčin vstupnej funkcie a funkcie jadra. Graficky je znázornená na obrázku 1.2.

Cieľom **poolingovej vrstvy** a *pooling*-ovej funkcie je znížiť rozmery vstupného vektoru a následne môcť predpokladať, aké vlastnosti obsahujú jednotlivé podoblasti. Existuje viac hlavných typov *pooling*-ových funkcií. Môžu nimi byť *max*, *min*, alebo *average* pooling. Pri použití *max pooling* sa vyberá maximálna hodnota z danej oblasti, pre *min pooling* tá minimálna, a pre *average* sa spočíta priemerná hodnota pre danú oblasť. [7]

Úlohou **normalizačnej vrstvy** je normalizovať hodnoty z vrstvy predchádzajúcej. Pomáha tak isto urýchľovať proces učenia a v niektorých prípadoch aj zlepšuje úspešnosť predikcie celého modelu. Je veľmi obľúbená v moderných konvolučných neurónových sieťach. [51]



Obr. 1.2: Konvolučná funkcia pre spracovávanie obrazu [7]

Dropout vrstva pomáha predchádzať pretrénovaniu neurónovej siete. Jej názov je odvodený od slovného spojenia - *dropping out*, ktorý v preklade znamená vylúčiť. Táto vrstva náhodne nastavuje výstupné hodnoty neurónov skrytých vrstiev na *0*. Pravdepodobnosť, s akou sa hodnoty vynulujú môže byť v intervale *<0; 1>*. Tento úkon sa deje pri každej tréningovej epoche. [55]

Podrobnejšie popísané vrstvy, ktoré sa používajú v konvolučných neurónových sieťach sú spomenuté v kapitole 6.2.1. V danej kapitole je rovnako spomenutý aj proces implementácie modelov. Už zaužívaných a optimalizovaných architektúr CNN, existuje veľké množstvo. Keďže v tejto práci boli použité dve takéto architektúry, na ktorých základoch vznikol celý model, sú popísané len tie. Spoločne s nimi sú spomenuté a vysvetlené najčastejšie používané aktivačné a optimalizačné funkcie.

ReLu aktivačná funckia patrí medzi základné aktivačné funkcie, kedy na výstup vráti presne tú istú hodnotu, ktorú dostala na vstup za predpokladu, že je kladná. V opačnom prípade vráti hodnotu *0*. Z matematického hľadiska je definovaná rovnicou 1.1

$$f(x) = max(x, 0) \tag{1.1}$$

pričom x je hodnota, ktorá vstupuje do aktivačnej funkcie. [59]

Softmax aktivačná funkcia sa snaží normalizovať hodnotu každého neurónu do intervalu *<0; 1>*. Súčet hodnôt jednotlivých neurónov je potom rovný *1*. *Softmax* sa v matematike označuje aj ako *normalizovaná exponenciálna funkcia*. Tá vezme vstupný vektor, ktorý má rovnakú dĺžku ako je počet neurónov v danej vrstve. Následne sú hodnoty normalizované do pravdepodobnostného rozdelenia. Štandardná *softmax* funkcia σ je teda definovaná rovnicou 1.2

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{1.2}$$

pričom K je počet hodnôt, z je daná hodnota na pozícii i. [59] Poslednými parametrami, ktoré je dôležité spomenúť, a pre čo najväčšiu efektivitu správne nastaviť, sú *loss funkcia*, *learnign rate (krok učenia)* a *optimalizačný algoritmus*.

Loss funkcia má za úlohu určiť, ako presne model vyhodnocuje dáta. Čím menší je rozdiel medzi skutočnou a modelom vyhodnotenou hodnotou, tým menšia je aj hodnota *loss* funkcie. Znalosť loss funkcie je nevyhnutná pre korektné učenie modelu. V prípade tejto práce pre všetky modely bola použitá *Cross-Entropy Loss* funkcia, alebo inak nazývaná aj *logaritmická* chybová funkcia. Jej matematikcý predpis je 1.3

$$H(y,p) = -\frac{1}{n}\sum_{i=1}^{n}[y^i log(p^i) + (1-y^i) log(1-p^i)] \tag{1.3}$$

pričom p je pravdepodobnosťou triedy a y odpovedá ground truth. [60]

Learning rate je hyperparameter, ktorý vyjadruje, aké veľké kroky sa robia počas každej iterácie, smerom k čo najväčšiemu minimalizovaniu hodnoty *loss* funckie. Je to pomerne dôležitý parameter a preto je dobré si uvedomiť dve veci. Pokiaľ je *learning rate* hodnota príliš malá, tak sa proces tréningu predĺži, poprípade sa môže úplne zastaviť. V opačnom prípade, keď programátor zvolí hodnotu príliš malú, tak sa celý proces tréningu môže stať nestabilný, váhy sú upravené príliš rýchlo. [61]

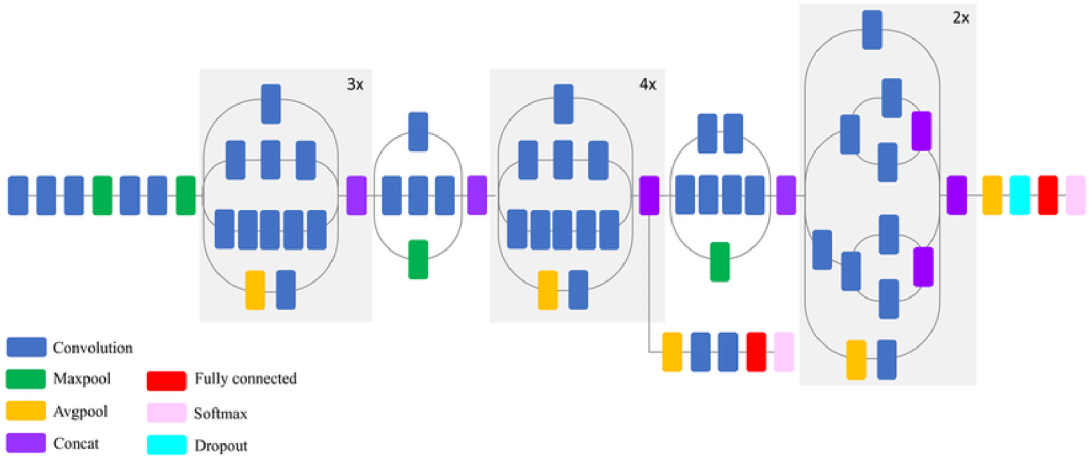
Optimizátor, alebo optimizačný algoritmus pomáha znižovať hodnoty *loss* funkcie na základe úpravy váh. Existuje viacero algoritmov, napríklad *SGD*, *RMSProp*, alebo *Adam*.

InceptionV3

Prvou použitou architektúrou v tejto práci bola *InceptionV3*. Táto architektúra bola vyvinutá spoločnosťou *Google LLC* a je v poradí už treťou architektúrou, ktorá patrí do ich série *Deep Learning Convolutional Architectures*. Pozostáva zo *42* vrstiev a jej graf je zobrazený na obrázku 1.3 Bola natrénovaná na vyše jednom milióne obrázkov, ktoré dokáže rozdeliť do *1000* tried. Dáta čerpala z *ImageNet*-u. *TensorFlow* verzia tohoto modelu klasifikuje dáta do *1001* kategórii, tou poslednou pridanou je "trieda pozadia". Trénovanie prebiehalo počas súťaže *ImageNet Large Visual Recognition Challenge* a dosahovala vyše *78.1%* úspešnosť na sto tisíc testovacích dátach. [46]

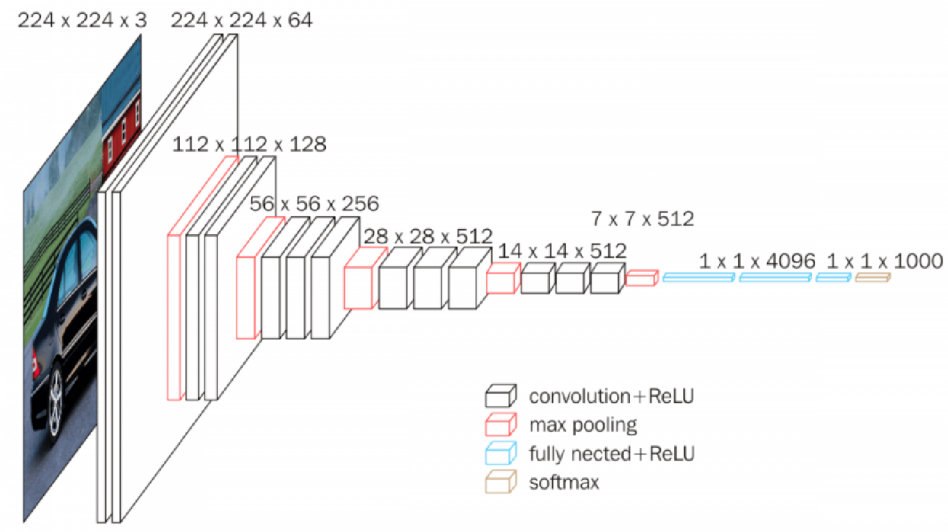
VGG16

Druhou architektúrou, ktorej implementácia v rámci tejto práce je popísaná v nasledujúcich kapitolách je *VGG16*. *VGG16* je konvolučnou neurónovou sieťou prvýkrát predstavenou profesormi *K. Simonyan* a *A. Zisserman* z Oxfordskej univerzity. Rovnako ako



Obr. 1.3: Architektúra modelu InceptionV3 [46]

InceptionV3, aj táto bola trénovaná na *ImageNet* databáze. Vykazovala však o niečo nižšiu úspešnosť. Trénovanie tohoto modelu je veľmi pomalé. Dokazuje to aj fakt, že pri prvom trénovaní ceý proces trval niekoľko týždňov, pričom sa používali vtedy veľmi výkonné grafické karty *NVIDIA Titan Black*. Architektúra tejto siete je zobrazená na obrázku 1.4 [47].



Obr. 1.4: Architektúra modelu VGG16 [47]

Keďže v kapitole 8 sa táto práca venuje porovnávaniu a zhodnocovaniu výsledkov, ktoré jednotlivé modely neurónových sietí dosiahli, je dobré popísať základné pojmy, ktoré sa tam budú používať. V grafoch a tabuľkách sa porovnávajú nasledujúce veličiny.

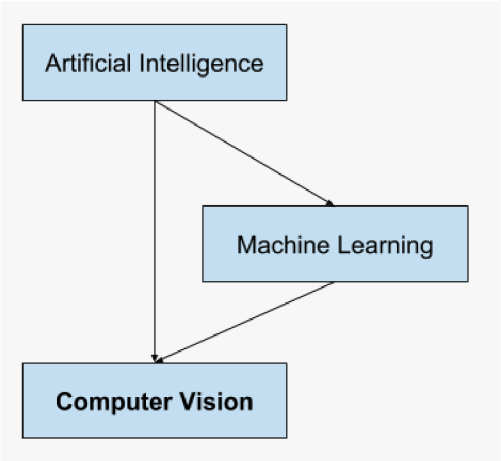
Accuracy odpovedá, s akou percentuálnou úspešnosťou neurónová sieť správne klasifi-

kovala tréningové dáta do daných kategórií. Druhým veľmi blízkym pojmom je **val_accuracy**. Tá vyjadruje, s akou percentuálnou úspešnosťou neurónová sieť po skončení epochy klasifikovala *validačné* dáta do daných kategórií. Prerekvizitou je, že validačné dáta nikdy predtým nevidela. Pokiaľ je *accuracy* výrazne vyššia ako *val_accuracy*, tak ide o *overfitting*.

Loss odpovedá hodnote *loss* funkcie pri tréningu, **val_loss** odpovedá hodnote *loss* funkcie pri validácii na konci epochy. Výrazný rozdiel je v tom, že pri validácii nejde o fázu učenia, ale o *recall* - fázu spomínania. V tomto prípade sa nevyužívajú *Dropout* vrstvy. Naopak pri tréningu sú *Dropout* vsrtvy aktívne, aby sa vo výsledku pridalo trochu šumu a eliminoval sa *overfitting*.

1.3 Počítačové videnie

Počítačové videnie - Computer Vision (CV) je oborom, ktorý sa snaží pomáhať počítačom a iným zariadeniam *vidieť*. Je možné ho zaradiť ako jednu z kategórií *spracovávania obrazu*. *Klasické spracovávanie obrazu* sa zameriava predovšetkým na úlohy bez nutnosti využitia strojového učenia. Rieši typické úlohy ako napríklad odstránenie šumu v obraze, nastavenie kontrastu alebo transofrmácia obrazu. Rovnako ale medzi techniky klasického spracovávania obrazu je možné zaradiť detekciu objektov, ktorá je založená na morfologických operáciách, alebo prahovaniu. Na druhej strane by sa CV dalo zaradiť ako podkategória AI a rovnako aj ML. Nutnosť používať sofistikovanejšie algoritmy v počítačovom videní však vyplynula z čoraz zložitejších úloh na vyriešenie, alebo nárokov na rýchlosť spracovania.



Obr. 1.5: Vzťah medzi AI, ML a CV

Cieľom CV je spracovať obsah digitálneho obrazu. Zvyčajne to zahŕňa metódy reprodukujúce schopnosti ľudského videnia. [8] Inou definíciou by mohlo byť - *Počítačové videnie je automatizované extrahovanie informácií z obrazu. Tými informáciami môže byť čokoľvek od 3D modelov, detekcie a rozpoznávanie objektov až po zoskupovanie a vyhľadávanie*

obsahu obrázkov. [9]

1.4 Vzťah medzi strojovým učením a počítačovým videním

Ako bolo spomenuté vyššie, ML a CV sú dve oblasti, ktoré spolu veľmi úzko súvisia. ML zdokonalilo rozpoznávanie obrazu, trekovanie a detekovanie objektov. Ďalej dokázalo ponúknuť efektívnejšie metódy ziskavania a spracovania obrazu ktoré sa používajú pri CV. CV naopak rozšírilo rozsah použiteľnosti ML. V nasledujúcich odstavcoch sú spomenuté aplikácie, ktoré sú využívané v bežnom živote.

1.4.1 Klasifikácia obrazu

Klasifikácia je proces predpovedania špecifickej triedy, alebo označenia skúmaného objektu, ktorý je definovaný množinou bodov.

Klasifikácia obrazu je teda podmožinou samotnej klasifikácie. Celému obrazu, ktorý je vyhodnocovaný, patrí jedno označenie. Podľa toho označenia a obsahu obrazu sa algoritmus môže rozhodovať, či je na obrázku pes alebo mačka, alebo aký model auta parkuje pred domom. Pri zamyslení nad tým, že takýchto kategórií a vstupných dát sú stovky tisíc a navyše nie stále je úplne jasné, čo na tom obraze hľadať, bolo viac ako potrebné tento proces automatizovať. Manuálne by bolo priam nemožné to spracovať dostatočne rýchlo, presne a efektívne.

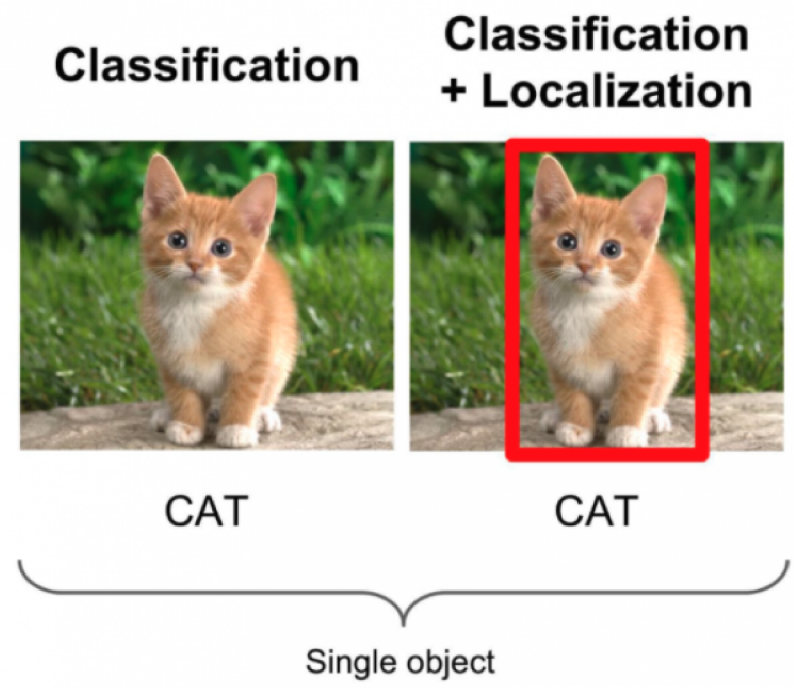
Architektúra modelu na klasifikáciu obrazu poväčšine zahrňuje *konvolučné vrstvy*, ktoré dohromady vytvárajú CNN. Na to aby vôbec CNN dokázala správne klasifikovať obraz, je potrebné jej nastaviť viacero parametrov, čo nieje najtriviálnejšia záležitosť. K tomu sa táto práca dostane v ďalších častiach. Avšak na začiatok je možné vziať tieto hodnoty už s existujúcich modelov, ktoré boli ocenené poprednými odborníkmi na túto oblasť na svetových podujatiach. K takým patrí napríklad *AlexNet*, ktorá vyhrala v roku 2012 cenu *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC). Ďalším vhodným kandidátom môže byť *Residual Neural Network* (ResNet) za rovnakú cenu o rok neskôr akú vyhrala *AlexNet* a rovnako aj za súťaž *Microsoft Common Objects in Context* (MS COCO) z roku 2015. [10]

1.4.2 Klasifikácia obrazu s lokalizáciou objektu

Ďalšou aplikáciou je špecifický druh *klasifikácie obrazu* a to konkrétne *klasifikácia obrazu* s následným *lokalizovaním objektu*. Tento problém je výrazne náročnejší, keďže je najprv potrebné zistiť koľko príslušných objektov sa nachádza v obraze. Po tomto splnenom predpoklade sa musia všetky objekty lokalizovať. Následne sa okolo každého vytýčia 4 body, ktoré tvoria *bounding box* každého z objektov.

Túto úlohu je rovnako možné implementovať pomocou modelov spomenutých v predchádzajúcom odstavci. Pomocou úpravy plne prepojenej - *fully connected* vrstvy sa potom vytvorí *bounding box*. Predpokladom na to, aby bolo možné model natrénovať na tento typ úlohy, je mať dostatočné množstvo tréningových dát obsahujúcich ako označenie alebo triedu kam objekt patrí, tak aj jednoznačne definovanú *bounding box*.

Náročnosť úlohy prichádza pri dátach, ktoré obsahujú neznámy počet objektov. Napríklad obrázky, ktoré vznikali na verejných miestach spravidla obsahujú veľkú diverzitu objektov. Ľudia, zvieratá alebo stromy, to je len malá časť toho, čo všetko môže byť klasifikované. Preto sa tento typ úlohy skôr stáva problémom *detekcie objektov*. [10]



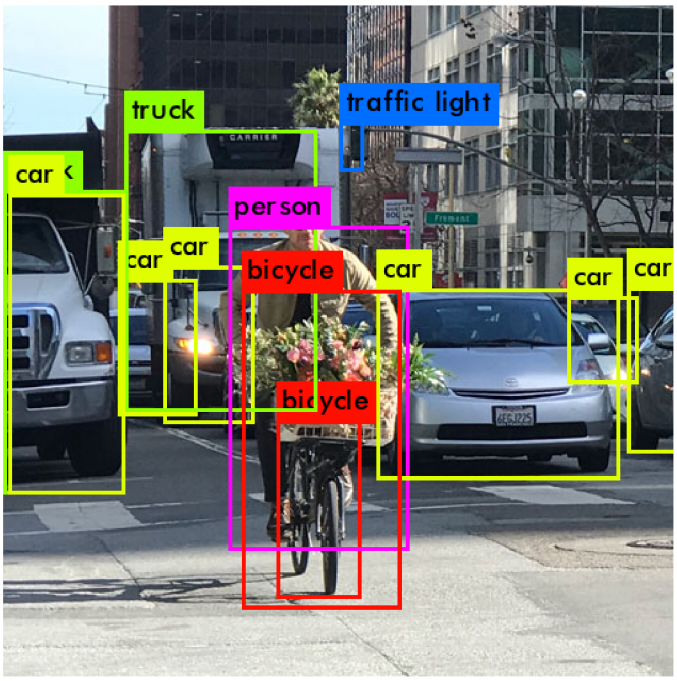
Obr. 1.6: Rozdiel medzi *klasifikáciou obrazu* na ľavej strane a *klasifikácie obrazu s lokalizáciou objektu* na pravej strane. [10]

1.4.3 Detekcia objektov

Detekcia objektov je vlastne *klasifikácia obrazu s lokalizáciou objektu* len pre obrazy, ktoré obsahujú neurčitý počet objektov na scéne. Toto je dôležitá oblasť, pretože systémy využívajúce CV v robotike alebo autonómnych vozidlách musia dokázať správne spracovávať komplexné obrazy. Detekovanie všetkých objektov je pre ich použitie preto kritickou časťou.

Architektúry modelov spomenutých v predošlých odstavcoch sa pre detekovanie objektov výrazne líšia. Nieje možné dopredu určiť veľkosť vstupného vektoru, pretože nevieme

koľko objektov sa na scéne vyskytne. Takže v niektorých prípadoch je nutné nastavovať parametre modelu až počas jeho chodu. Jednou z možností je použiť modely zo skupiny *Region-Based Convolutional Neural Network* (R-CNN). Tie ako jedny z prvých riešia práve problémy *lokalizácie*, *detekcie* a *segmentácie* obrazu. Inou možnosťou sú modely z rodiny *You Only Look Once* (YOLO). Tie nedosahujú takú presnosť ako R-CNN ale na druhú stranu ponúkajú ďaleko vyššiu rýchlosť vyhodnocovania, prakticky *real-time*. [11]



Obr. 1.7: Detekcia objektov v reálnom svete. [12]

1.4.4 Rekonštrukcia obrazu

Rekonštrukcia obrazu je aplikácia, ktorej úlohou je buď prerobiť časti obrazu do pôvodného stavu, alebo ich doplniť, pokiaľ úplne chýbajú tak, aby vytvorili pôvodný obraz. Nieje možné dávať obraz do pôvodného stavu bez toho, aby bola známa aspoň nejaká skutočnosť, ako obraz vyzeral na začiatku. Preto výsledok rekonštrukcie veľmi závisí na tom, koľko sa algoritmus dokáže naučiť z pôvodného obrazu.

Jednými z modelov, ktoré túto problematiku riešia sa nazývajú *Pixel Recurrent Neural Networks*. Používajú RNN na predpovedanie hodnoty chýbajúcich pixelov pozdĺž oboch osí obrazu. Táto aplikácia nachádza využitie napríklad vo filmovom priemysle pri prerábaní čierno-bielych filmov na filmy farebné. Rovnako aj v *autonómnych vozidlách*, kedy je potrebné rekonštruovať obraz v prípade, že medzi objektom, ktorý je dôležitý pre korektný chod vozidla a treba ho rozpoznať, a kamerou vozidla sa nachádza iný objekt. [10]

1.4.5 Trekovanie objektov

Všetky vyššie spomenuté aplikácie dokázali pracovať s jedným obrazom, v čase nemenným. Avšak pri tomto probléme je cieľ trochu iný. Je dôležité dokázať rozpoznať a následne sledovať udalosť meniacu sa v čase. Pre lepšie pochopenie ako daná udalosť prebieha a ako sa mení scéna obrazu je potrebné mať sériu obrázkov, ktoré na seba nadväzujú.

Trekovanie objektov začína tak, že v prvom obraze sa objekt detekuje a následne sa okolo neho vytvorí *bouding box*. Trekovací algoritmus potom vytvára *bouding box* okolo daného objektu pre každý nasledujúci obraz. Ideálne by mal *bouding box* obkolesovať trekovaný objekt tak dlho, pokiaľ je viditeľný. Rovnako to platí aj pre prípad, kedy objekt z jedného snímku zmizne, ale v ďalších sa znovu objaví.

Využitie sa dá nájsť vo viacerých aplikáciách. Pri *autonómnych vozidlách* je veľmi dôležité trekovať objekty ako napríklad chodci alebo iné vozidlá tak, aby nedošlo k nehodám. [10]

Obr. 1.8: Príklady trekovania objektov na sérii obrázkov. [10]

2 Software-ové nástroje pre strojové učenie

V predchádzajúcej kapitole sa táto práca venovala aplikáciám a využitiu *strojového učenia* v *počítačovomo videní*. Aj keď algoritmus ML nemá presne definované inštrukcie programátorom ako daný problém vyriešiť, tak implementácia samotného modelu pre danú úlohu je veľmi dôležitá. S tým je spojený výber vhodného programovacieho jazyka.

2.1 Výber vhodného programovacieho jazyka

Tým ako sa ML stáva stále viac a viac populárnym, rastie aj jeho podpora v rôznych programovacích jazykoch. V súčastnosti je možné naprogramovať model ML prakticky ktorým koľvek programovacím jazykom. Avšak programovať každú jednu časť aplikácie *od nuly* keď už existuje, nieje najrozumnejšie. Je, respektíve môže to byť časovo veľmi náročné a zároveň je tam veľký predpoklad, že to nebude tak optimalizované a efektívne ako už existujúce riešenia. Preto pri výbere vhodného programovacieho jazyka je dosť podstatné sa pozerať na to, aké *knižnice* pre ML obsahuje a akú má podporu z komunity dátových analytikov.

Github minulý rok vydal článok, ktorý obsahoval 10 najlepších programovacích jazykov pre ML na základe príspevkov od ľudí z *Github* komunity. Pre ukážku, na prvom mieste sa umietnil *Python*, nasledovaný jazykmi *C++* a *JavaScript* a na štvrtom a piatom mieste sa umiestnili *Java* a *C#*. [13]

Keďže celá funkcionalita vytvorená pre potreby tejto práce je implementovaná v jazyku *Python* respektíve niečo bolo písané priamo v terminále, čiže bol použitý *Shell*, tak nasledujúce podkapitoly sa budú venovať práve *Python*-u.

2.1.1 Python ako programovací jazyk

Python je *interpretovaný*, *objektovo-orientovaný*, *vyšší* programovací jazyk s *dynamickou sémantikou*. Obsahuje vstavané datové štruktúry a v kombinácii s *dynamickým typovaním* je *Python* veľmi populárnym programovacím jazykom. Jeho využitie je veľmi široké. Od písania skriptov, ktoré vykonávajú určitú funkcionalitu, cez webové stránky, grafické aplikácie alebo hry až práve po AI.

Python obsahuje množstvo knižníc a modulov, ktoré podporujú myšlienku *znovupoužiteľnosti*. Jednoduchá syntax a ľahká čitateľnosť povzbudzuje množstvo začiatocníkov do programovania. Tým, že program písaný v *Python*-e sa nekompiluje, tak užívateľ veľmi rýchlo, ako urobí zmenu v kóde zistí, či sa to správa a či to funguje tak ako má, alebo nie. Rovnako pohodlné je aj *debuggovanie*, keďže *interpreter* vyhodí výnimku za každou chybou

na ktorú narazí. Ak táto výnimka nieje zachytená samotným programom tak interpreter vypíše názov výnimky a kde nastala - *stack trace*. [14]

2.2 Prehľad najpoužívanejších knižníc pre strojové učenie

Keďže okrem *modulov*, ktoré ponúkajú *štandardné knižnice* vyššie spomenutých programovacích jazykov, tak existujú aj desiatky ďalších, ktoré si uživateľ môže stiahnuť a nainštalovať. V roku *2019* vydala spoločnosť *Codete.com* rebríček najpoužívanejších knižníc pre ML. Na prvých troch miestach sa umiestnili *TensorFlow*, *Keras* a *Scikit-learn*. Za nimi nasledovala knižnica vyvinutá spoločnosťou *Facebook, Inc.* - *PyTorch*, rovnako napísaná v Python-e, avšak obsahujúca *frontend*-ovú časť, ktorú je možné implementovať v *C++*. Ďalšími obľúbenými knižnicami sú *H2O*, *Caffe* alebo *MXNet*. [16] Nasledujúce odstavce budú venovať práve popisu ML knižníc použitých v tejto práci.

2.2.1 Keras

Keras je *Application Programming Interface* (API) pre neurónové siete napísané v jazyku *Python*. Pôvodne bol vyvinutý ako súčasť výskumu s názvom *Open-ended Neuro-Electronic Intelligent Robot Operating System* (ONEIROS). Je navrhnutý tak, aby dokázal komunikovať a spracovávať výpočty nad knižnicou *TensorFlow*, ktorá bude spomenutá ďalej. Hlavnou motiváciou a cieľom vyvinutia *Keras*-u bola myšlienka získania výsledku z čo najmenšou odozvou po urobení zmeny v kóde.

Keras umožňuje rýchle a zároveň jednoduché vytváranie prototypov vďaka vysokej modularite a uživateľskej prívetivosti. Podporuje ako CNN, tak aj RNN respektíve kombináciu týchto dvoch typov sietí. Pomocou *Keras*-u je možné vykonávať výpočty buď na *centrálnej procesorovej jednotke* - *CPU* alebo na *grafickej karte* - *GPU* čiže je tam veľký potenciál na dostatočne rýchle a pohodlné trénovanie modelov neurónovej siete. [15]

2.2.2 TensorFlow

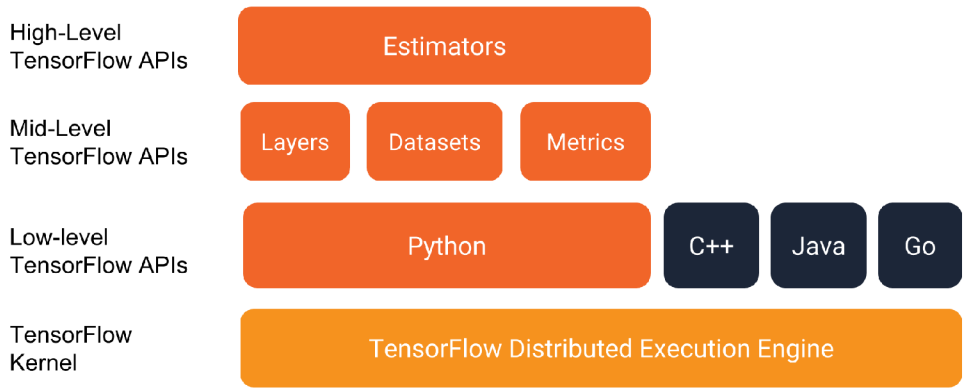
Ďalšou knižnicou, ktorú je potrebné spomenúť pri programovaní ML aplikácií v *Python*-e je *TensorFlow*.

TensorFlow je *open-source* knižnica spracovávajúca numerické výpočty, založená na grafovej štruktúre. Pôvodne bola vyvinutá inžiniermi so spoločnosti *Google LLC*, konkrétne *Google Brain* tímom v rámci *Google's Machine Intelligence* výskumu, ktorý sa zaoberal ML a výskumom neurónových sietí. No využitie tejto knižnice nemusí ostať len v tejto oblasti. Príkladom môže byť trieda *Solver()* z modulu *ode*, ktorej inštancia umožňuje výpočet *diferenciálnych rovníc* [17].

Veľkou prednosťou *TensorFlow*-u je jeho prenositeľnosť. Funguje *multiplatformne*, čo znamená, že funguje takmer na takmer všetkých platformách a zariadeniach. Spúšťanie

na GPU je samozrejmosť. Rovnako tak aj na rôznych *embedded-platformách*. Jedným z príkladov môže byť práve *Raspberry Pi*. Základom tejto knižnice je jazyk *C++* na ktorom je postavená. Užívateľ si potom následne môže vybrať, či chce pre vývoj ML aplikácie používať *C++* alebo *Python*.

Nad *frontend*-ovou časťou *TensorFlow*-u sa nachádzajú 3 API rozhrania. Najprv rozhranie *vrstiev*, ktoré ponúka pohodlnú manipuláciu s bežne používanými vrstvami v modeli. Nad tým sú postavené *Keras* API a *Estimator* API, ktoré pomáha s trénovaním a ohodnocovaním natrénovaného modelu.



Obr. 2.1: Architektúra knižnice *TensorFlow* [19]

Rýchlosť a prenositeľnosť zabezpečuje používanie grafovej štruktúry. Po spustení kódu sa vytvorí *graf*, ktorý je následne spustiteľný. Je to dátová štruktúra, ktorá obsahuje všetky potrebné informácie o danom modeli. Je prenositeľná a spustiteľná na ktorejkoľvek inej platforme, ako bolo spomenuté vyššie. Výpočetná náročnosť je nižšia a existuje tam stále možnosť ďalšej optimalizácie už existujúceho *grafu*. [18]

2.2.3 Caffe

Poslednou knižnicou, ktorej sa práca v tejto teoretickej časti venuje, je *Caffe*. Táto knižnica bola vyvinutá vedcami s *Berkeley Vision*, ktorý patrí pod *Berkeley*-skú univerzitu v Kalifornii. Podobne ako *TensorFlow*, *Caffe* je nástroj určený pre prácu s ML. Kód jadra je napísaný v jazyku *C++* a rýchlosť, presnosť a modularita boli hlavnou motiváciou pre vznik tejto knižnice. Ponúka možnosť spracovávať výpočty ako na *CPU* tak aj na *GPU*, čo je zabezpečené pomocou *CUDA* - API od spoločnosti *Nvidia* podporujúce paralelné výpočty na *GPU*. [20]

Caffe podporuje rozsiahly počet vrstiev a stratových funkcií - *loss functions*, ktoré už sú v knižnice naimplementované. Nastavujú sa pomocou *acyklických grafov* v konfiguračnom súbore. Rovnako, *Caffe* už obsahuje predtrénované známe modely, ako napríklad

AlexNet a umožnuje tak priestor pre ďalšie experimenty. [21]

Aj keď táto knižnica nebola priamo použitá v tejto práci, bolo ju potrebné nainštalovať. Cesta k nej následne musela byť pridaná do systémových premenných aby bolo možné nakonfigurovať *Neural Compute SDK* (NCSDK), ktorému sa bude venovať ďalšia kapitola.

3 Použitý hardware

Jednotlivé časti tejto kapitoly sa budú venovať popisu *hardware*-ových akcelerátorov, následne konkrétnemu akcelerátoru od firmy *Intel*, ktorý bol použitý v tejto práci a nakoniec zoznámením sa so zariadením *Raspberry Pi*.

3.1 Akcelerátory neurónových sietí

Akcelerátory neurónových sietí alebo *AI akcelerátory* sú jednou z kategórií *hardware*-ových akcelerátorov. Sú nadizajnované tak, aby zefektívnili a urýchlili aplikácie zamerané hlavne na CV, ML a všeobecne používajúce neurónové siete. Dokážu presunúť určité výpočetné procesy na špecializované *HW* komponenty v systéme za účelom dosiahnutia vyššej výpočetnej efektívnosti a rýchlosti, ako by tomu bolo pri spracovávaní danej úlohy na CPU.

Hardware-ové akcelerátory kombinujú použitie bežných procesorov ako je CPU a silu HW prispôsobeného na daný typ úlohy ako napríklad GPU alebo *Zákaznícky integrovaný obvod – Application-Specific Integrated Circuit* (ASIC). Tým sa následne niekoľkonásobne zvýši výpočetná sila. Príkladom môžu byť aplikácie na správu a spracovávanie videa. To dostane za úlohu na spracovanie GPU a nielen že sa tento proces zrýchli, ale sa aj uvoľní pamäť v CPU na vykonávanie iných činností.

Medzi najpoužívanejšie *hardware*-ové akcelerátory sa môžu zaradiť:

- GPU: Pôvodne nadizajnované na spracovanie sekvencií obrazu a videa, avšak momentálne nachádzajú využitie pri spracovávaní a výpočte veľkého množstva dát. Veľkou výhodou je, že architektúra GPU umožňuje paralelizmus, čiže spracovávanie množstva rôznych operácií v ten istý čas.

- FPGA: *programovatelné hradlové pole – Field Programmable Gate Array* (FPGA) je typ logického obvodu, ktorý sa vyznačuje tým, že môže byť naprogramovaný na vykonávanie rôznych úloh. Programuje sa pomocou jazyka HDL. Je možné ho využiť na zefektívnenie algoritmu, pričom výpočetná sila je delená práve medzi FPGA a CPU.

- ASIC: ASIC je integrovaný obvod vyrobený na spracovávanie jednej konkrétnej úlohy. V moderných ASIC akcelerátoroch sa nachádza viac ako 100 miliónov logických hradiel.

[22]

3.1.1 Intel Movidius Neural Compute Stick

Movidius, spoločnosť, ktorú v roku 2016 odkúpil gigant na výrobu čipov - Intel, vyvinula *AI akcelerátor*, ktorý nepotrebuje vlastné napájanie, nepotrebuje sa pripájať na cloud, celá optimalizácia sa deje na integrovanom obvode v zariadení pripojenom cez USB zbernicu.

Základom tohoto zariadenia je *Myriad Vision Processing Unit* (VPU) čip, ktorý je navrhnutý na akceleráciu úloh CV využívajúcich CNN. Podľa vyjadrení *Intel*-u, Myriad VPU obsahuje dedikovanú architektúru na vysoko kvalitné spracovávanie obrazu, CV a neurónových sietí. Tieto vlastnosti ho robia vhodným na zvládnutie náročných úloh zameraných na CV v moderných chytrých zariadeniach. Je používaný v chytrých bezpečnostných kamerách po celom svete, alebo dronoch ovládaných pomocou gest ruky.

Myriad je sám o sebe integorvaný obvod, ktorý sa Movidius rozhodol zakomponovať do ich *Movidius Neural Compute Stick* (NCS). Keďže, ako bolo spomenuté vyššie, NCS sa pripája pomocou USB rozhrania, je to veľmi jednoducho pripojiteľné ako periféria k iným zariadeniam, v prípade tejto práce k *Rapsberry Pi*.

Momentálne sú na trhu dve verzie NCS a to konkrétne *NCS1* a *NCS2*. Obe zariadenia majú plnú podporu od *Intel*-u. V tejto práci je použitá prvá generácia tohto zariadenia - *NCS1*.

Vnútro *NCS1* tvorí *Myriad 2 VPU*, ktorý pozostáva z dvanástich *Streaming Hybrid Architecture Vector Engine* (SHAVE) jadier. Tie zabazpečujú paralelizáciu výpočtov. *NCS1* dokáže komunikovať s dvomi platformami a to konkrétne *Ubuntu 16.04*, alebo *Raspbian Stretch*. Nepredpokladá sa, že by *Intel* pridal podporu pre ďalšie operačné systémy. [23]

Obr. 3.1: NCS od firmy *Intel Corporation* pripojiteľný cez USB zbernicu [23]

3.1.2 Neural Compute Software Development Kit

NCSDK je vývojárským balíčkom, ktorý umožňuje rýchle vytváranie a následný presun modelu v kompatibilnom formáte do zariadenia *NCS1*. Rovnako ako na kompiláciu, tak obsahuje nástroje aj na profilovanie a kontrolu validity modelu. V neposlednom rade NCSDK

ponúka *neural compute* API pre vývoj aplikácii v jazykoch *Python* a *C/C++*.

Podľa dokumentácie, ktorá sa nachádza na *github*-e v repozitári NCSDK, je možné tento vývojársky balíček použiť iba s *Movidius*-om pvrej generácie. Pri *NCS2* je nutné použiť *OpenVINO* nástroj. [25]

Aktuálne existujú dve verzie NCSDK. Prvá - v1 sa nachádza na hlavnej vetve repozitáru. Druhá - v2 sa nachádza na vetve s názvom *ncsdk2*. NCSDK v2 obsahuje oproti prvej verzii niektoré funkcie akým je vylepšené API, ktoré podporuje uloženie viacerých grafov v jednom zariadení, alebo vstup a výstup v dátovej štruktúre typu *fronta*. Na obrázku 3.2 sú vyobrazené jednotlivé štádiá, ktorým si model musí prejsť, aby bol následne kompatibilný s *NCS1*. [26]

3.1.3 Implementácia modelu na Intel Movidius Neural Compute Stick 1

V nasledujúcej časti sú popísané kroky, ktoré je potrebné vykonať aby bolo možné použitie akcelerátoru modelov neurónových sietí.

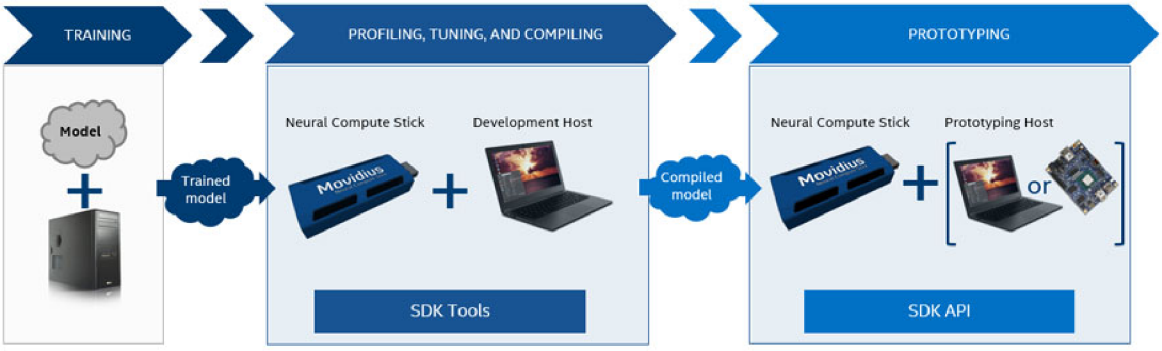
Tým prvým je natrénovanie modelu na určitú funkcionalitu. Toto sa deje buď na *cloud*-e, GPU alebo inom zariadení, ktoré je na to výpočetne prispôsobené. Model by mal byť vytvorený použitím *TensorFlow* alebo *Caffe* knižnice.

V momente, kedy je model natrénovaný a je vyhodnotená jeho presnosť a rýchlosť, prechádza sa k druhému štádiu. Model je potrebné skompilovať a zoptimalizovať pre použitie na NCS1. Kompiláciou vznikne súbor typu *graph*. Táto časť sa deje za pomoci NCSDK - *open-source* projektu, ktorý prichádza s rôznymi knižnicami a nástrojmi, ktoré umožňujú vytvoriť graf z *TensorFlow* alebo *Caffe* modelov.

V tomto kroku má programátor dve možnosti. Buď to model skompilovať na operačnom systéme *Ubuntu 16.04*, alebo *Raspbian Stretch*. Efektívnejšie je ale využitie linuxovej distribúcie, keďže tam je predpoklad, že to beží na výkonnejšom zariadení ako je *Raspberry Pi*. Avšak nič nebráni kompilovaniu modelu na Raspbian-e. Ako bude spomenuté ďalších častiach tejto práce, aj táto možnosť funguje. Po vygenerovaní grafu je následne nahraný do NCS. Tento proces prebieha za podpory knižníc naimplementovaných v jazyku *Python* a *C++* v NCSDK repozitári. [24]

3.2 Hlavná výpočetná jednotka

Ako bolo vyššie spomenuté, trénovanie a kompilácia modelu by za ideálnych podmienok mali byť vykonávané na zariadeniach s adekvátnou výpočetnou silou. V závislosti na veľkosti projektu, preferenciách programátora a požiadavkách zákazníka sa naskytujú viaceré



Obr. 3.2: Kroky, ktoré zahrňuje použitie NCS1. [26]

možnosti. Veľmi zaujímavou možnosťou je trénovanie modelu na serveroch od *Google LLC*. Existuje možnosť prenájmu *virtuálneho stroja*, ktorý sa následne dá nakonfigurovať, napríklad na využívanie GPU od spoločnosti *Nvidia*. Konkrétne *high-end*-ová rada s grafickým čipom od spoločnosti *Tesla* a jej rôzne modely. Ďalšou, a pre požiadavky tejto práce vhodnejšou voľbou je však trénovanie na PC. [27]

Modely, ktoré boli použité v tejto práci, boli trénované na notebooku od spoločnosti *Apple Inc.*. Zariadenie obsahuje osem jadier *Intel Core i9*, každé s taktovacou frekvenciou *2,3GHz*. Ďalej obsahuje *16GB* pamäte RAM typu DDR4, čo zabezpečuje viac ako dostatočný výkon a pamäť pre trénovanie modelov o danej náročnosti. Na vrchu celého hardware-u je operačný systém *macOS Catalina*.

Ďalším použitým zariadením bol notebook *HP EliteBook Folio 1040 G3*. Jeho špecifikácie sú, na rozdiel od predchádzajúceho zariadenia, nedostačujúce na trénovanie. Použitie tohoto zariadenia bolo však nevyhnuté, keďže, ako bolo spomenuté vyššie, NCSDK je možné nainštalovať buď na linuxovú distribúciu *Ubuntu 16.04*, alebo *Raspbian Stretch*. Obsahuje štyri *Intel Core i7* jadrá šiestej generácie, každé o taktovacej frekvencii *2,5GHz*. Tieto čísla samy o sebe niesú vôbec zlé. V čom má ale veľké nedostatky, ktoré nechal pocítiť, je o polovicu nižšia kapacita pamäte RAM. Pri pokusoch o trénovanie a pár krát aj pri komunikovaní s NCS1 bol jediným výstupom error typu **Segmentation fault**.

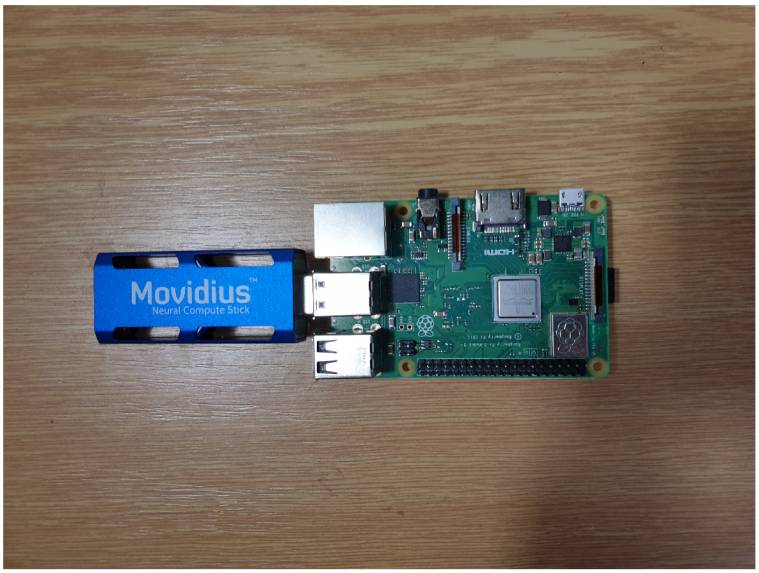
3.2.1 Raspberry Pi

Raspberry Pi je pomerne lacné zariadenie o veľkosti platobnej karty. Tento jednodoskový počítač bol vyvinutý organizáciou *Raspberry Pi Foundation*, ktorá ako charita si dáva za cieľ pomáhať so šírením vzdelania v oblasti informatiky vo Veľkej Británii. Prvý model, *Raspberry Pi 1 Model B* vyšiel v roku 2012, no odvtedy sa vydalo ďalších 13 modelov. Momentálne je možné z oficiálnych stránok Raspberry zakúpiť 9 týchto zariadení. Väčšinou modely, ktoré boli vydané na začiatku a prešli revíziou, už niesú na trhu k dispozícii

na zakúpenie. Najnovšie *Raspberry Pi 4 Model B* vyšlo na trh v júni 2019. Základ tvorí štvorjadrový procesor *Broadcom BCM2711* architektúry *ARMv8* o taktovacej frekvencii *1,5GHz*. Uživateľ má možnosť si vybrať veľkosť pamäte RAM a to medzi *1*, *2* a *4GB*. Ďalej ponúka až dva micro-HDMI porty, každý z nich umožňuje prehrávanie videa v kvalite až *4k*. Samozrejmosťou sú *2xUSB3.0* a *2xUSB2.0* porty. [28]

V tejto práci bolo však použité *Raspberry Pi 3 Model B+*, keďže iba tento model je podporovaný *Movidius*-om. Jeho parametre sú o niečo horšie ako parametre Raspberri Pi štvrtej generácie. Konkrétne štvorjadrové procesor *Broadcom Cortex A53* architektúry *ARMv7* o taktovacej frekvencii *1,4GHz*. Veľkosť operačnej pamäte je fixne daná - a to *1GB*. Ako bude v práci ďalej spomenuté, bolo to skúšané aj na *Raspberry Pi 1 Model B+*, ktoré však bohužiaľ nebolo kompatibilné. [29]

V nasledujúcich častiach tejto práce je popísaný postup ako naimplementovať natrénovaný model neurónových sietí, ktorý klasifikuje obrazy do šiestich kategórii na platformu *Raspberry Pi* s použitím akcelerátoru neurónových sietí *Intel Movidius Neural Compute Stick 1*. Zapojenie je znázornené na obrázku 3.3.



Obr. 3.3: Raspberry Pi 3 Model B+ s pripojeným akcelerátorom Intel Movidius.

Jednotlivé sekcie sú zoradené podľa toho, ako sa v práci postupovalo. Okrem fuknčnej časti projektu sú v práci spomenuté aj všetky chyby a problémy, ktoré pri implementácii nastali. Snahou je vysvetliť, čo v takom prípade robiť a ako tieto chyby odstrániť, aby sa na konci dosiahol cielený výsledok.

4 Implementácia Neural Compute SDK

V tejto kapitole je popísaný postup, ako nainštalovať vývojársky balíček od *Movidius*-u a všetky knižnice, ktoré potrebuje k jeho funkčnosti. Inštalácia prebehla ako na PC *HP EliteBook Folio*, tak aj na *Raspberry Pi Model 3 B+*. Spomenutá bude aj neúspešná inštalácia na *Raspberry Pi Model 1 B+* a to z toho dôvodu, aby sa programátor, inšpirujúci sa touto prácou, vyhol podobnému problému s kompatibilitou.

Keďže je v tejto práci použitý Movidius NCS prvej generácie, nieje možné použiť *OpenVINO Toolkit*. Na druhej strane, pôvodný *Neural Compute Software Development Kit* má stále dostatočnú podporu od *Intel*-u. Inštalovaná bola druhá verzia NCSDK s označením *v2*. Ako už bolo spomenuté predtým, prvá a druhá verzia niesú spätne kompatibilné, preto ako aj na *Raspberry Pi*, tak aj na PC *HP EliteBook Folia* sú nainštalované rovnaké verzie NCSDK.

V druhej časti tejto kapitoly je spomenuté, ako využívať práve nainštalovaný NCSDK. Je vhodné sa najprv zoznámiť s možnosťami tohoto *development kit*-u, aby sa následne bolo možné v ďalších častiach tejto práce odvolávať na jednotlivé príkazy.

4.1 Spôsoby inštalácie NCSDK

Medzi veľkú výhodu druhej verzie NCSDK oproti tej prvej sú rôzne spôsoby inštalácie. Programátor nieje viazaný na inštaláciu do celého svojho zariadenia, ale si môže vybrať medzi spustením v rámci *docker image*-u, vytvorením *virtual environment*-u pre izolované prostredie jazyka Python, alebo inštaláciu na *Virtual Machine* (VM).

4.1.1 Virtual Machine

Tento spôsob má oproti ostatným pomerne veľkú výhodu a to v tom, že je možné ju spustiť aj na nepodporovaných verziách operačných systémov. Oficiálne testovanými operačnými systémami na *hostiteľských - host* zariadeniach, na ktorom VM beží, sú *Windows 10*, *OS X Yosemite 10.10.5* alebo *Ubuntu 16.04*. Je však pravdepodobné, že to bude kompatibilné aj s inými operačnými systémami, ktoré testované neboli. Doporučeným virtualizačným software-om je *VirtualBox*. Podstatným krokom pri nastavovaní VM je aktivovanie *USB3.0* rozhrania a *whitelist*-ovanie USB portu, ktorým bude Movidius NCS1 komunikovať. Táto možnosť je ale náročnejšia na výkon použitého hardware-u. [34]

4.1.2 Virtual Environment

Virtual Environment označené ako *virtualenv* je nástroj, ktorý vytvára izolované prostredie pre použitie jazyka Python a jeho kompatibilných knižníc. Predchádza konfliktom, ktoré sú spôsobené vzájomnou nekompatibilitou medzi jednotlivými verziami knižníc.

Pre použitie tejto možnosti je potrebné ešte pred samotnou inštaláciou nastaviť v konfiguračnom súbore hodnotu premennej **USE_VIRTUALENV=yes**. V tejto práci bola použitá práve táto možnosť, ktorá elegantne oddeľuje knižnice a rôzne verzie jazyka Python od zvyšku systému. Pokiaľ dôjde k nekompatibilite jednotlivých závistlostí, stále je možné virtuálne prostredie preinštalovať a skúsiť to odznova. Na akej ceste je možné nájsť konfiguračný súbor, v ktorom je potrebné nastaviť danú premennú, bude ukázané v ďalšej časti práce. [35]

4.1.3 Docker

Docker je software, ktorý sa stará o izolovanie konfiguračných súborov, aplikácií a ich knižníc a ostatných závislých súborov do kontajnerov. Každý kontajner je vlastne samostatný proces. Zároveň zabezpečuje, že aplikácie môžu byť spustené v ktoromkoľvek prostredí. Výhodou je, že štart kontajneru je okamžitý a zaťaženie pamäte RAM je minimálne, pretože kernel je zdieľaný. Ďalšiou prácou Docker-u je starať sa o celkový životný cyklus kontajnerov, od ich vzniku, naštartovania až po ich zánik.

NCSDK môže byť nainštalované a spustené priamo v Docker kontajneri, čiže všetky závislosti potrebné na spustenie NCSDK sú izolované od zvyšku systému. Rovnako je predpoklad, že Movidius NCS1 bude pripojený cez USB rozhranie. [36]

4.2 NCSDK pre Ubuntu 16.04

Táto verzia linuxovej distrubúcie operačného systému je stále veľmi jednoducho dohľadateľná a stiahnuteľná z oficiálnych stránok Ubuntu. Jej celé označenie je *Ubuntu 16.04.6 LTS (Xenial Xerus)*. Je dôležité, aby to bola *64 bitová* verzia pre *AMD64* architektúru. Ak je tento predpoklad splnený, je možné prejsť k samotnému stiahnutia repozitára NCSDK a jeho inštalácii.

Predtým, ako dôjde k samotnému naklonovaniu vetvy, na ktorej je *NCSDK v2* uložený, je potrebné nainštalovať *Git*. To je možné spraviť príkazom **sudo apt install git**. O zvyšok sa postará nástroj na menežovanie balíčkov. Po úspešnom nainštalovaní verzovacieho nástroja je možné prejsť k samotnému naklonovaniu repozitáru. Použitím príkazu **git clone -b ncsdk2 http://github.com/Movidius/ncsdk** sa do priečinku, kde sa programátor momentálne nachádza, naklonujú všetky dáta z vetvy *ncsdk2* ktorá vychádza z projektu *http://github.com/Movidius/ncsdk*. Následne je potrebné sa presunúť do novo vzniknutého priečinku s názvom *ncsdk*.

Keďže pre túto prácu bola použitá inštalácia v rámci *virtuálneho prostredia*, je potrebné pred samotnou inštaláciou modifikovať súbor na ceste **ncsdk/ncsdk.conf**. Jedinou zmenou je nastavenie premennej **USE_VIRTUALENV=yes** z pôvodnej verzie, ktorá mala hodnotu **no**. Posledným krokom v rámci inštalácie NCSDK je stiahnutie a spustenie

binárneho súboru, prekopírovanie potrebných súborov na svoje miesta a doinštalovanie ostatných vyžadovaných knižníc. O túto časť sa postará príkaz **make install**. Ten vyžaduje *sudo* práva, čiže bude potrebné ešte zadať heslo do systému. [35]

V prípade, že je inštalácia neúspešná a skončí sa chybou s návratovou hodnotou **131** pri spustení príkazu **python setup.py egg_info**, exituje spôsob, ako to napraviť. Táto práca sa s týmto problémom stretla, preto je vhodné spomenúť jeho riešenie. Pravdepodobnou príčinou je zastaralá verzia knižníc *setuptools* a *pip*. Zavolaním príkazu **pip3 install –upgrade <názov knižnice>** sa nainštaluje jeho najnovšia verzia. Ďalšou záležitosťou, na ktorú ale musí programátor myslieť je to, že *virtuálne prostredie* už bolo vytvorené. Inštalácia vrátila chybu až ďaleko po tomto kroku. Bohužiaľ *virtualenv* používa stále pôvodnú verziu týchto knižníc a inštalácia nových verzií sa ho netýka. Takže riešenie, ktoré sa použilo v tejto práci, bolo vymazať priečinok *ncsdk*, urobiť *update* týchto knižníc a znovu reprodukovať kroky, ktoré boli popísané vyššie.

4.3 NCSDK pre Raspbian Stretch

Inštalácia NCSDK pre operačný systém Raspbian Stretch mala pôvodne prebiehať rovnako ako pri Ubuntu 16.04. Keďže obsahom tejto práce nieje popisovať, ako prebieha inštalácia OS pre Raspberry Pi na SD kartu, tieto kroky sú vynechané. Na oficiálnych stránkach sa táto verzia distrubúcie už nenachádza, preto je potrebné stiahnuť súbor v formáte *zip* z neoficiálnych stránok. Najnovšia verzia pochádza z apríla roku 2019. [37]

4.3.1 Inštalácia pre Raspberry Pi 3 Model B+

Prvý problém nastáva hneď na začiatku. Rovnako, ako pri predošlej inštalácii NCSDK, tak aj v tomto prípade, debugovanie a hľadanie riešenia trvalo pomerne dlho. Táto verzia OS prichádza s Python verziou *3.5.2*. Ten sa už síce radí medzi Python 3 verzie, ale niektoré knižnice, ktoré sú definované v súbore **requirements.txt** vyžadujú Python *3.6* alebo vyšší. Vedľa pôvodnej verzie bolo odskúšané doinštalovať aj novšiu verziu Python-u a zároveň prenastaviť systémové cesty k nemu, no neúspešne. Jediný výsledok, ktorý sa dosiahol, bolo ďalšie obmedzenie funkčnosti, poprípade pri úplnej odinštalácii pôvodnej verzie Python-u prestali fungovať viaceré systémové veci.

Konkrétnym zdrojom problému sa stala knižnica *dask*. Tá je vyžadovaná knižnicou *scikit-image*. A na koniec, *scikit-image* je vyžadovaná inštaláciou NCSDK. To samo o sebe problémom nieje, pretože ako aj pre *dask*, tak aj pre *scikit-image* existujú verzie pre nižšie verzie Python-u. Čo ale problémom už je, v **requirements.txt** je presne nadefinovaná aj verzia, respektíva interval, v akom je akceptovaná inštalácia danej knižnice. Pre *scikit-image* je je to verzia medzi *0.13.0* a *0.14.0* vrátane. Pri tomto rozmedzí je vyžadovaný

dask vyšší ako *0.9.0* a ten vyžaduje Python vyšší ako je verzia, ktorá prišla s inštaláciou OS.

Jedno z riešení, ktorému sa hodí označenie *workaround* je nasledovný postup. Prvým krokom je staihnutie celej knižnice *dask 2.0.0* zo zdroja. Na *github.com/dask/dask* je možné nájsť všetky verzie, ktoré kedy boli vydané. [38] Fungovať by to malo aj s predošlými verziami, pre túto konkrétnu to ale bolo odskúšané. Teraz je všetko pripravené k inštalácii tejto knižnice. Dohromady to tvoria tieto príkazy:

Výpis 4.1: Bash príkazy pre inštaláciu *dask 2.0.0* zo zdroja.

```
wget https://github.com/dask/dask/archive/2.0.0.zip
unzip dask-2.0.0.zip
cd dask-2.0.0
python3 -m pip install ".[complete]"
```

Po *dask*-u je potrebné nainštalovať *scikit-learn*. Odskúšaná verzia je *0.15.0* a je ju možné nainštalovať pomocou manažéra balíčkov a knižníc. V tomto prípade bol použitý *pip*. Konkrétny príkaz je **pip3 install scikit-learn==0.15.0**. Po tomto kroku je však nainštalovaná verzia *scikit-learn* knižnice ignorovaná NCSDK. Je nutné trochu poupraviť zdrojové súbory. Konkrétne shell-ovský skript, ktorý je možné nájsť na ceste **./install.sh**. V príkaze **$PIP_PREFIX pip3 install $PIP_QUIET –trusted-host files.pythonhosted.org scikit-learn>=0.13.0,<=0.14.0** je nutné zmeniť verziu knižnice *scikit-learn* z *0.14.0* na *0.15.0*.

Posledným problémom, ktorému inštalácia čelila, bola inštalácia *Caffe* knižnice. Medzi jej vyžadovanými knižnicami je aj knižnica *Pillow* a práve jej verzia vyvolávala výnimku. Konkrétne *Caffe* nedokázala nájsť jednu metódu, ktorú *PIL* mala obsahovať. Riešením bola manuálna inštalácia zo zdroja, ktorá bola zvládnuteľná pomocou návodov na oficiálnych stránkach *Caffe*. [39] Následne už nič nebránilo úspešnej inštalácii NCSDK pomocou príkazu **make install**.

4.3.2 Nekompatibilita s Raspberry Pi Model 1 B+

Inštalácia na tento model Raspberry Pi prebiehala úplne rovnako ako pre model *3 B+*. Jediným rozdielom bolo, že v tomto prípade inštalácia skončila neúspešne. Vyvolaná bola **Illegal Instruction** výnimka. Tá vzniká v momente, kedy je snaha o spustenie inštrukcií na inom type procesoru, pre aký boli tieto inštrukcie dizajnované. [40] V tomto prípade je hlavným problémom to, že základ *Raspberry Pi Model 1 B+* tvorí procesor zkonštruovaný na architektúre *ARMv6*, ale Intel vyvýjal NCSDK pre procesory *ARMv7*.

4.4 Používanie NCSDK

V prvom prípade, kde sa NCSDK inštalovalo pre *Ubuntu 16.0.4* bolo použitá *virtual environment* pre Python. Preto, aby tento balíček bolo možné využívať, je potrebné virtuálne

prostredie najprv aktivovať.

4.4.1 Aktivácia NCSDK

Aktivácia sa deje spustením príkazu **source /opt/movidius/virtualenv-python/bin/activate**. **Source** umožnuje spustiť spustiteľný súbor na ceste, ktorá je definovaná ako argument príkazu. Je dôležité si uvedomiť, že všetky knižnice, ktoré sú doinštalované do aktivovaného virtuálneho prostredia, sú dosiahnuteľné len v rámci tohoto prostredia. Presne to je najväčšou výhodou vytvárania virtuálnych prostredí pre každý projekt zvlášť, aby sa navzájom nemohli ovplyvňovať jednotlivé verzie knižníc. Deaktivovať virtuálne prostredia je možné pomocou príkazu **deactivate**. [35]

4.4.2 Nástroje NCSDK

Intel Movidius Neural Compute SDK obsahuje nástroje pre *profilovanie*, *vylaďovanie* a *compiláciu* modelov neurónových sietí. Na nasledujúcich odstavcoch bude spomenuté, čo jednotlivé príkazy vykonávajú, respektíve na čo sú určené. Na druhej strane však nebudú úplne do podrobna popísané všetky argumenty daných príkazov. Rovnako sú vynechané aj všetky výnimky, ktoré môžu byť vyvolané. Tieto informácie sú však veľmi ľahko dohladateľné v oficiálnej dokumentácii Movidius-u. Každý príkad nižšie popísaný je odcitovaný, aby dohľadanie dokumentácie bolo čo najjednoduchšie.

mvNCCheck

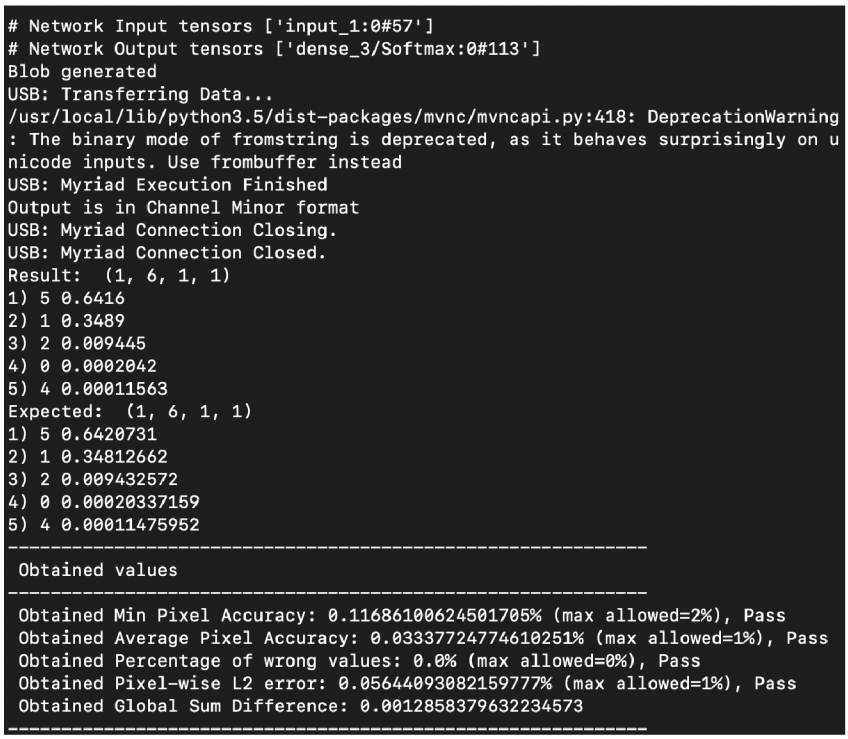
mvNCCheck je nástrojom príkazového riadku, ktorý zabezpečuje kontrolu validity *Caffe* a *TensorFlow* modelov neurónových sietí na akcelerátore neurónových sietí. Vyhodnocovanie modelu prebieha ako aj na Movidius-e, tak aj na zariadení, na ktorom je spustený. Výsledky sa následne porovnajú a NCSDK rozhodne, či neurónová sieť uspela (**PASS**), alebo nie (**FAIL**). Prvých päť najlepších výsledkov je výstupom tohoto porovnávania. [41]

Ako príklad bol použitý príkaz **mvNCCheck TF_Model/tf_model.meta -in=input_1 -on=dense_3/Softmax**. Výstup je možné vidieť na obrázku 4.1.

Prvým argumentom príkazu je ceste k *meta* súboru neurónovej siete. Druhým argumentom je *vstupná* vrstva. Tým tretím zasa *výstupná* vrstva. Tieto argumenty sú povinné.

mvNCProfile

mvNCProfile je nástroj, ktorý skompiluje model neurónovej siete, aby bol použiteľný na Movidius-e. Zároveň, už skompilovaný model spustí na danom akcelerátore a vygeneruje reporty v textovej a HTML podobe. Takto vygenerované reporty obsahujú záznamy a štatistiky o výkonnosti daného modelu vrstvu po vrstve. Tieto informácie ďalej dokážu pomôcť pri vylaďovaní modelu. Programátor dokáže vidieť, koľko času trvá spracovanie tej danej vrstvy a to je vhodná informácia pre následnú optimalizáciu. [42]



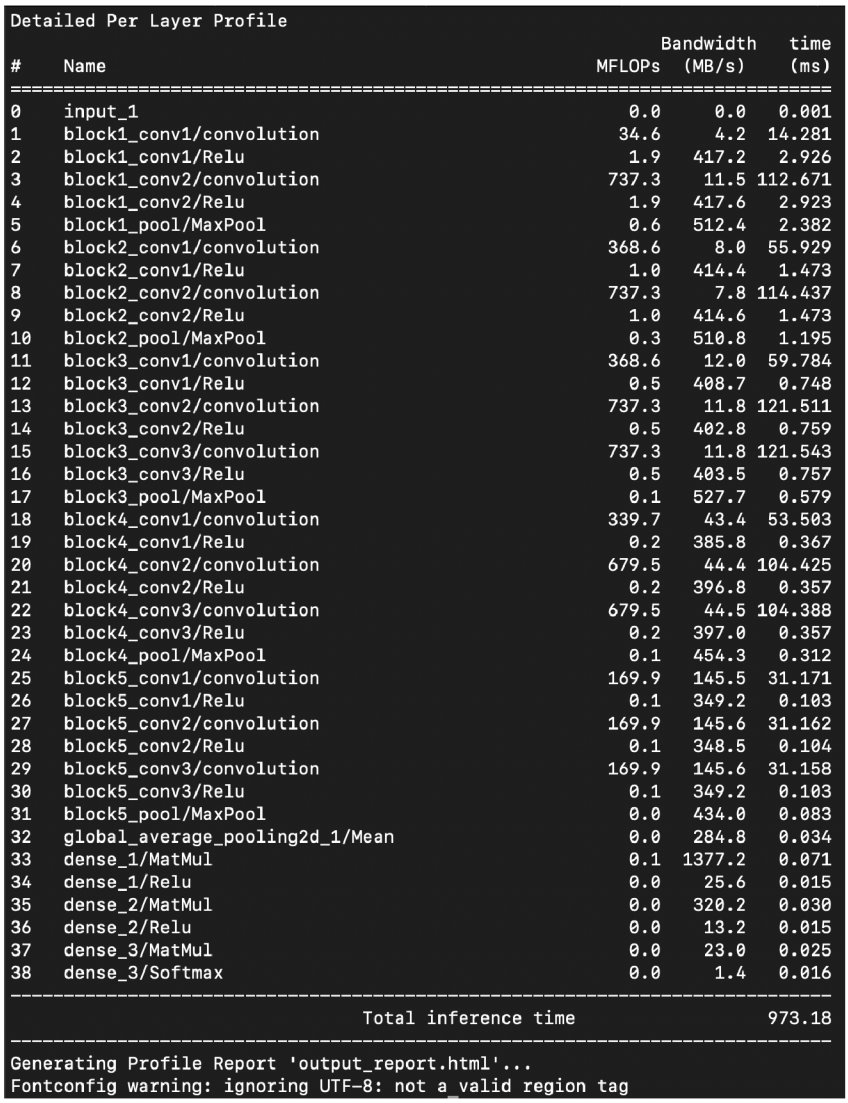
Obr. 4.1: Príklad výstupu príkazu **mvNCCheck**

Vzorový príkaz má tvar **mvNCProfile TF_Model/tf_model.meta -in=input_1 -on=dense_3/Softmax** a argumenty majú rovnaký význam ako v prípade vyššie. Výstupom je teda textová podoba, znázornená na obrázku 4.2, ako aj *HTML*, *numpy* a *csv* report.

mvNCCompile

Posledným z nástrojov je **mvNCCompile**, ktorý kompiluje model a váhy neurónovej siete do súboru vo formáte *graph* pre Movidius.

Príkladom použitia môže byť napríklad nasledujúci príkaz **mvNCCompile TF_Model/tf_model.meta -in=input_1 -on=dense_3/Softmax**. Rovnako ako pre oba predchádzajúce príklady, aj pre tento príkaz sú argumenty rovnaké. Výstupom je *graph* súbor, ktorý je priložený v prílohe. Výstup z terminálu nieje dostatočne zaujímavý na to, aby bol priložený ako samostatný obrázok.



```
Detailed Per Layer Profile
                                                                  Bandwidth   time
#    Name                                                MFLOPs  (MB/s)    (ms)
==============================================================================
0    input_1                                              0.0     0.0   0.001
1    block1_conv1/convolution                            34.6     4.2  14.281
2    block1_conv1/Relu                                    1.9   417.2   2.926
3    block1_conv2/convolution                           737.3    11.5 112.671
4    block1_conv2/Relu                                    1.9   417.6   2.923
5    block1_pool/MaxPool                                  0.6   512.4   2.382
6    block2_conv1/convolution                           368.6     8.0  55.929
7    block2_conv1/Relu                                    1.0   414.4   1.473
8    block2_conv2/convolution                           737.3     7.8 114.437
9    block2_conv2/Relu                                    1.0   414.6   1.473
10   block2_pool/MaxPool                                  0.3   510.8   1.195
11   block3_conv1/convolution                           368.6    12.0  59.784
12   block3_conv1/Relu                                    0.5   408.7   0.748
13   block3_conv2/convolution                           737.3    11.8 121.511
14   block3_conv2/Relu                                    0.5   402.8   0.759
15   block3_conv3/convolution                           737.3    11.8 121.543
16   block3_conv3/Relu                                    0.5   403.5   0.757
17   block3_pool/MaxPool                                  0.1   527.7   0.579
18   block4_conv1/convolution                           339.7    43.4  53.503
19   block4_conv1/Relu                                    0.2   385.8   0.367
20   block4_conv2/convolution                           679.5    44.4 104.425
21   block4_conv2/Relu                                    0.2   396.8   0.357
22   block4_conv3/convolution                           679.5    44.5 104.388
23   block4_conv3/Relu                                    0.2   397.0   0.357
24   block4_pool/MaxPool                                  0.1   454.3   0.312
25   block5_conv1/convolution                           169.9   145.5  31.171
26   block5_conv1/Relu                                    0.1   349.2   0.103
27   block5_conv2/convolution                           169.9   145.6  31.162
28   block5_conv2/Relu                                    0.1   348.5   0.104
29   block5_conv3/convolution                           169.9   145.6  31.158
30   block5_conv3/Relu                                    0.1   349.2   0.103
31   block5_pool/MaxPool                                  0.0   434.0   0.083
32   global_average_pooling2d_1/Mean                      0.0   284.8   0.034
33   dense_1/MatMul                                       0.1  1377.2   0.071
34   dense_1/Relu                                         0.0    25.6   0.015
35   dense_2/MatMul                                       0.0   320.2   0.030
36   dense_2/Relu                                         0.0    13.2   0.015
37   dense_3/MatMul                                       0.0    23.0   0.025
38   dense_3/Softmax                                      0.0     1.4   0.016
------------------------------------------------------------------------------
                                 Total inference time                  973.18
------------------------------------------------------------------------------
Generating Profile Report 'output_report.html'...
Fontconfig warning: ignoring UTF-8: not a valid region tag
```

Obr. 4.2: Príklad textového výstupu príkazu **mvNCProfile**

4.4.3 Výnimky vyvolané NCSDK

Rovnako ako pri hociktorých iných nástrojoch, alebo príkazoch, aj v tomto prípade je nevyhnutné používať vyššie spomenuté nátroje korektne. Inak hrozí, že sa vyvolajú výnimky, ktoré následne bránia v ďalšom chode programu. Je potrebné spomenúť najbežnejšie prípady, kedy sa *shell*-ovská inštancia ukončila s kódom iným, ako *0*. Zoznam všetkých kódov aj s ich významom, je možné nájsť v dokumentácii Movidius-u na oficiálnych stránkach Intel-u. [44] V rámci implementácie neurónovej siete na Movidius sa táto práca stretla s nasledujúcimi výnimkami.

Toolkit Error: USB Failure

Táto situácia nastáva v momente, kedy nedokáže Movidius komunikovať s NCSDK. Buď nieje vôbec pripojený cez USB rozhranie do zariadenia, alebo je samotný USB port za-

blokovaný, respektíve pokazený. V týchto prípadoch ho nieje možné rozpoznať. Vyvolaná výnimka má nasledujúci tvar: **[Error 7] Toolkit Error: USB Failure. Code: No devices found**

Toolkit Error: Stage Details Not Supported

Táto chyba je pravdepodobne jedna z najhorších. Znamená, že model neurónovej siete s touto architektúrou je nepodporovaný Movidius-om. Samozrejme je možné, že s ďalšími verziami NCSDK budú tieto vrstvy pridané. No v tomto momente daná vrstva musí byť z modelu odstránená, aby ho bolo možné skompilovať. Za názvom chyby je uvedená konkrétna vrstva, ktorá momentálne nemá podporu: **[Error 5] Toolkit Error: Stage Details Not Supported: batch_normalization_1:sub_1**

Toolkit Error: Input/Output Node Name Doesn't Exist

Táto chyba bola v rámci tejto práce veľmi častá. Vypovedá o tom, že aspoň jedna zo zadaných vrstiev v argumentoch **-in**, alebo **-on**, je v nesprávnom tvare, alebo neexistuje. Veľmi častou chybou bolo, že za názvom výstupnej vrtsvy musí nasledovať znak "/" a následne názov aktivačnej funkcie s prvým písmenom veľkým. Pre ukážku - výstupná vrtsva má názov *dense_3*, ale v tomto tvare ako argument pre nástroje NCSDK nieje akceptovaná. Správny tvar je *dense_3/Softmax*, pričom aktivačná fukncia v tejto vrstve bola *softmax*. Vyvolaná výnimka má nasledujúci tvar: **[Error 13] Toolkit Error: Provided Output-Node/InputNode name does not exist or does not match with one contained in model file Provided: dense_3/**

5 Dáta použité na natrénovanie modelu neurónových sietí

Dáta použité v tejto práci boli poskytnuté firmou *Tyco Electronics Czech s.r.o.* so sídlom v meste Kuřim. Firma sa zaoberá výrobným programom zameraným na káblovú a konektorovú techniku pre automobilový priemysel. Príkladom môžu byť káblové zväzky, poistkové skrine, kabeláže pre pripojenie AirBagu a rada ďalších produktov. [30]

5.1 Rozdelenie použitých dát

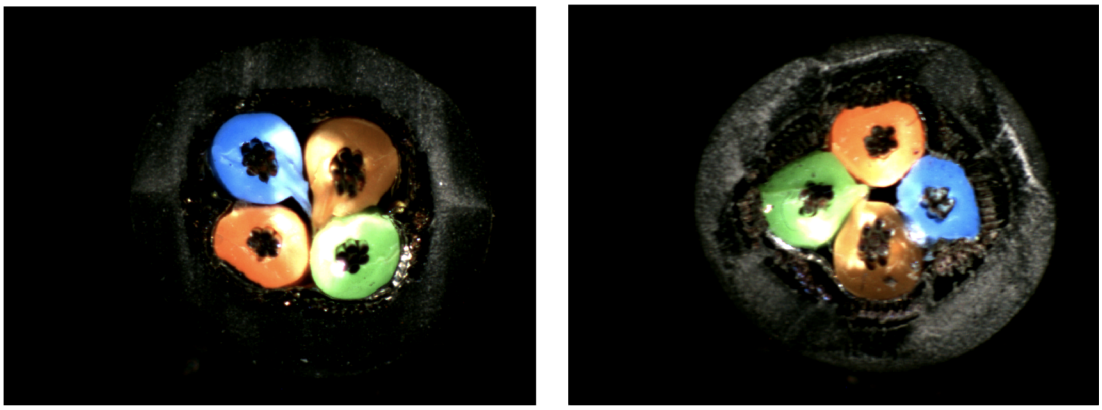
Dáta reprezentujú snímky prierezu káblov, ktoré ďalej putujú po pásovom dopravníku. Kategórie sú buď pomenované podľa typu rezu, alebo druhu káblu. Kategórií je dohromady šesť, konkrétne:

- 535
- 535-2
- blue
- dacar524
- HT
- manual_cut

Všetkých obrazových dát je dohromady 8925. Okrem šiestich vyššie spomenutých kategórií, podľa ktorých ich neurónová sieť klasifikuje, sú prvotne rozdelené do dvoch skupín. Skupina *tréningových* a skupina *testovacích* dát. V ďalšom odstavci bude potom spomenuté, že sa vytvára ešte aj tretia skupina, skupina *validačných* dát. Ale fyzicky boli poskytnuté v prvých dvoch vyššie spomenutých skupinách.

5.1.1 Kategória *535*

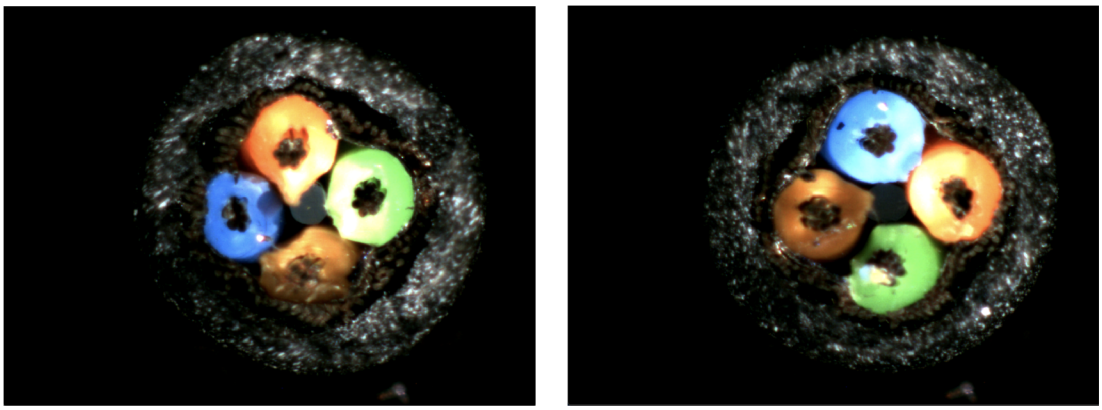
- Počet tréningových vzorkov: 800
- Počet validačných vzorkov: 200
- Počet testovacích vzorkov: 500
- Vzorová ukážka dát:



Obr. 5.1: Ukážky dvoch obrazov z kategórie *535*

5.1.2 Kategória *535-2*

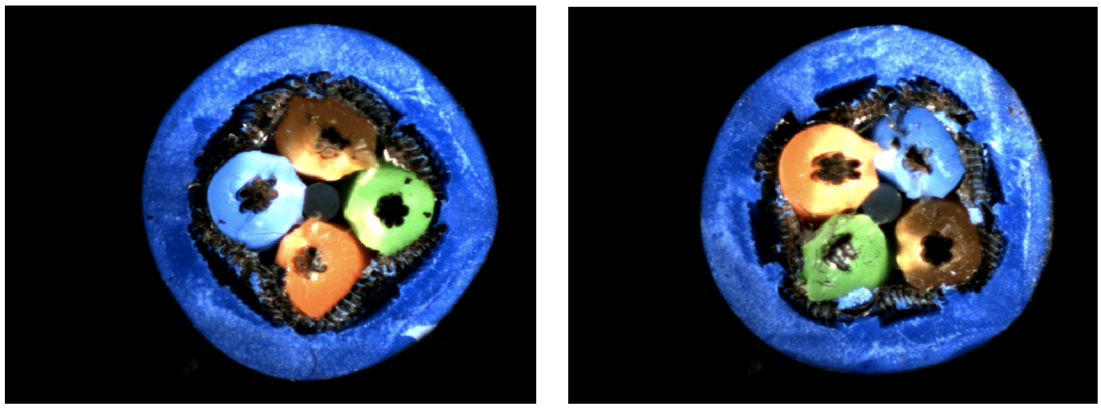
- Počet tréningových vzorkov: 800
- Počet validačných vzorkov: 200
- Počet testovacích vzorkov: 500
- Vzorová ukážka dát:



Obr. 5.2: Ukážky dvoch obrazov z kategórie *535-2*

5.1.3 Kategória *blue*

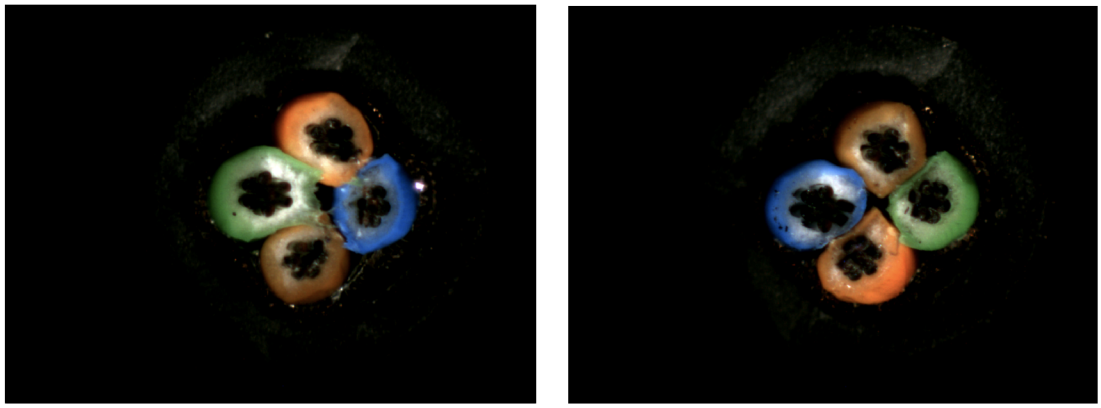
- Počet tréningových vzorkov: 800
- Počet validačných vzorkov: 200
- Počet testovacích vzorkov: 500
- Vzorová ukážka dát:



Obr. 5.3: Ukážky dvoch obrazov z kategórie *blue*

5.1.4 Kategória *dacar524*

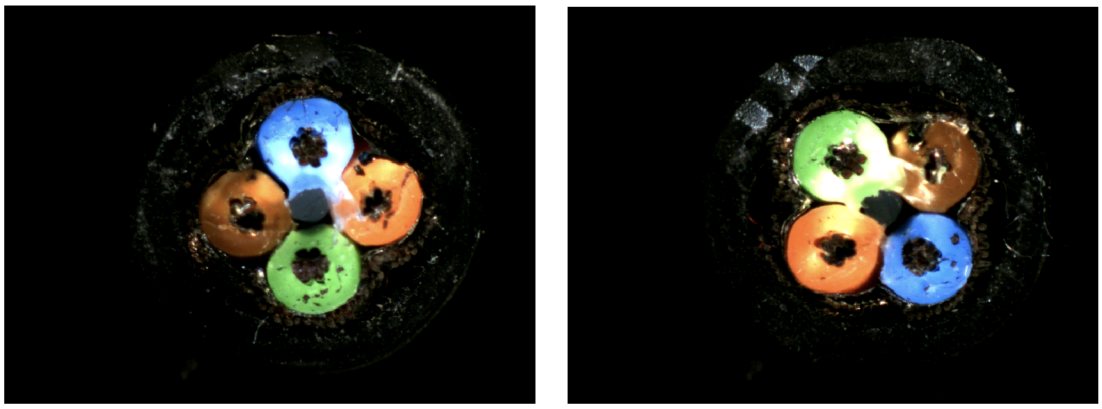
- Počet tréningových vzorkov: 800
- Počet validačných vzorkov: 200
- Počet testovacích vzorkov: 500
- Vzorová ukážka dát:



Obr. 5.4: Ukážky dvoch obrazov z kategórie *dacar524*

5.1.5 Kategória *HT*

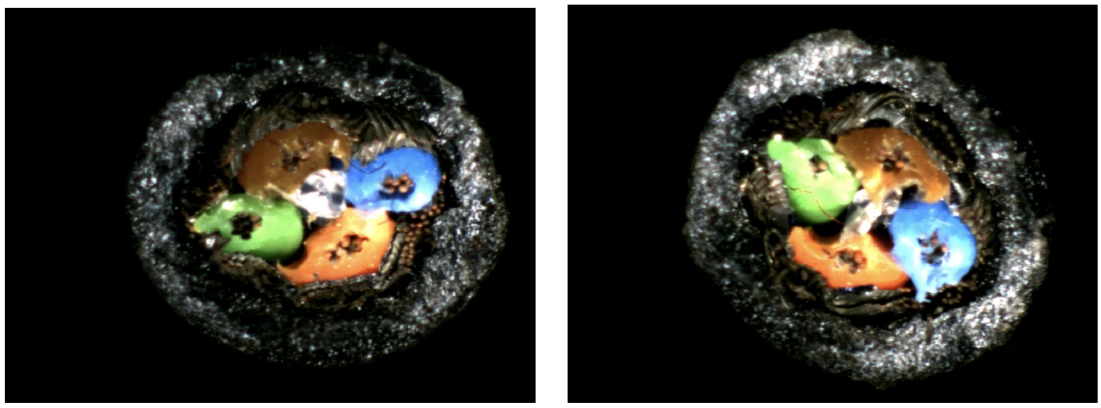
- Počet tréningových vzorkov: 800
- Počet validačných vzorkov: 200
- Počet testovacích vzorkov: 500
- Vzorová ukážka dát:



Obr. 5.5: Ukážky dvoch obrazov z kategórie *HT*

5.1.6 Kategória *manual_cut*

- Počet tréningových vzorkov: 725
- Počet validačných vzorkov: 200
- Počet testovacích vzorkov: 500
- Vzorová ukážka dát:



Obr. 5.6: Ukážky dvoch obrazov z kategórie *manual_cut*

5.2 Formát použitých dát

Každý jeden z poskytnutých obrázkov je vo formáte *bitmap image file* (BMP). BMP je pomerne rozšírených formátom pre ukladanie rástrových obrázkov. Informácia o hodnote pixelov sa ukladá pre každý jeden pixel zvlášť a bez následnej kompresie. Preto sú BMP obrázky vysoko kvalitné, ale pri vyšších rozlíšeniach zaberajú viac miesta ako iné typy rástrových formátov - JPEG alebo GIF. [31].

Rozlíšenie obrazových dát je *800x600* pixelov. Keďže na natrénovanie modelu neurónovej siete nebolo potrebné použiť pôvodnú veľkosť obrazov, upravovali sa na *100x100*,

respektíve *200x200* pixelov. Pôvodná veľkosť by znamenala značné predĺženie doby trénovania modelu a ako bude ďalej možné vidieť, na samotnú presnosť by to vplyv nemalo.

5.3 Vytvorenie validačných dát

Aby bolo možné presnejšie vyjadriť, ako veľmi sa darí natrénovanému modelu rozpoznávať nové dáta, s ktorými sa nikdy predtým nestretol, je na validáciu vhodné použiť metódu *cross-validation*. Pokiaľ model vykazuje vynikajúcu úspešnosť pre dáta na ktorých bol trénovaný, ale pre nové dáta, s ktorými sa ešte nikdy nestretol je úspešnosť oveľa nižšia, ide o pretrénovanosť - *overfitting*. Je možné povedať, že model až príliš dobre dokáže rozpoznať známe dáta, no pre použitie v reálnej aplikácii sa už nehodí, keďže nedokáže dostatočne reagovať na nové situácie. Opakom sa dá pomenovať podtrénovanie - *underfitting*. To nastáva v okamihu, keď úspešnosť pri validačných dátach je vyššia ako na tréningových. Tak isto tým vieme označiť celkovú situáciu, keď má model nízku úspešnosť. [32]

Už od začiatku, pri trénovaní vlastného modelu, sa táto práca stretla s *overfitting*-om. Po pár tréningových epochách bola jej úspešnosť voči tréningovým dátam veľmi vysoká, nad *97%*, ale pri testovaní na testovacích dátach sa nedokázala dostať cez *20%*. Tento problém sa podarilo detekovať metódou *cross-validation*, ktorá je spomenutá v tomto odstavci, následne zväčšením množstva tréningových dát - *data augmentation*, ktorá je spomenutá v nasledujúcom odstavci, to bolo možné odstrániť. Samozrejme pomohli aj úpravy nastavenia parametrov modelu, no to bude spomenuté až v ďalsích kapitolách.

Aj keď existuje viac typov *cross-validation* metód, koncept je pre všetky druhy veľmi podobný a vyzerá nasledovne. V prvom kroku sa rozdelia tréningové dáta na množinu *tréningových* a množinu *validačných* dát v určitom pomere. Validačné dáta sa dajú stranou, nepoužívajú sa na trénovanie modelu. Cieľom je odhaliť chybu predikovania modelu tým, že sa overí úspešnosť predikcie modelu skupinou dát, s ktorou sa model nikdy predtým nestretol - s validačnými dátami.

Na rozdelenie skupiny dát na tréningové a validačné bola zo začiatku v rámci kódu použitá metóda *train_test_split()*. Táto metóda sa nachádza v *Python*-ovskej knižnici *scikit-learn* a jej module *model_selection*. Následne však pre lepšiu prehľadnosť a hlavne možnosť jednoduchšie nájsť chybu a odkrokovať program, bola *validačná* skupina vytvorená ručne a prestala sa používať vyššie spomenutá metóda. Dáta boli rozdelné približne v pomere *1/4*, kde tréningových bolo dohromady *4725* a validačných *1200*.

5.4 Zvýšenie počtu tréningových dát - data augmentation

Ako bolo spomenuté v odstavci vyššie, ďalšou z možností, ako predísť *overfitting*-u je zvýšenie počtu tréningových dát. Najlepšie, pokiaľ sú tieto dáta ešte poupravené. Príkladom

môže byť priblíženie podstatnej informácie v obraze, pridanie šumu alebo pootočenie. *Keras* v module *preprocessing.image* ponúka triedu *ImageDataGenerator*. Tá vracia objekt - *iterátor*. Iterátor umožňuje aplikovanie funkcií úprav na dáta, ktoré boli predané konštruktoru tejto triedy a následné prechádzanie cez nich. Ďalej je potrebné zavolať na inicializovaný iterátor metódu *flow* alebo *flow_from_directory*. Tá vytvorí generátor, ktorý vracia (yield) batch-e upravených dát v každom kroku trénovania modelu. Dáta sa generujú tak dlho, pokiaľ prebieha tréning. Proces generovania dát je náhodný, samozrejme v rámci možností používaného HW a to práve zabezpečuje diverzitu dát a zabraňuje *overfitting*-u.

Výpis 5.1: Objekt triedy *ImageDataGenerator*

```
train_datagen = ImageDataGenerator(
      rescale = 1/255.0,
            rotation_range=25,
            width_shift_range=0.1,
            height_shift_range=0.1,
            shear_range=0.2,
            zoom_range=0.2,
            horizontal_flip=True,
            fill_mode='nearest'
        )
```

Argumenty použité v konštruktore triedy *ImageDataGenerator* sú nasledovné:

- rotation_range: Typ - *int*. Rozsah pootočenia obrazu v stupňoch.
- width_shift_range: Typ - *float*. Posunutie v smere šírky.
- height_shift_range: Typ - *float*. Posunutie v smere dĺžky.
- shear_range: Typ - *float*. Veľkosť skrútenia proti smeru hodinových ručičiek v stupňoch.
- zoom_range: Typ - *float*. Rozsah priblíženia.
- horizontal_flip: Typ - *bool*. Preklopenie v horizontálnom smere.
- fill_mode: Body mimo hranice v obraze sú vyplnené podľa vzoru, ktorý je argumentom.

Pomocou týchto argumentov sú náhodne aplikované funkcie na pôvodný obraz. Cieľom tejto aplikácie je zovšeobecnenie modelu a lepšie a presnejšie predikovanie nových dát.

6 Modely neurónových sietí

Nasledujúca kapitola sa venuje jednotlivým modelom, použitým v rámci tejto práce. Modely už buď existovali a cieľom ich bolo skompilovať a naimplementovať na Movidius, alebo vznikali od samého začiatku. Kapitola sa rovnako tak aj venuje ostatným skriptom, ktoré mali či už priamy, alebo nepriamy vplyv na implementáciu modelu na Movidius.

6.1 InceptionV3 model neurónovej siete od firmy Tyco Electronics Czech s.r.o.

Prvým z modelov, ktorý bol testovaný, bol od firmy *Tyco Electronics Czech s.r.o.*. Táto firma model natrénovala na dátach, ktoré boli spomenuté v kapitole 5, na klasifikáciu obrazov. Pre potreby tejto práce bol už natrénovaný model poskytnutý ako súbor vo formáte *hdf5* s názvom **weights-improvement-06-0.9983.hdf5**. Je to súbor označovaný ako *checkpoint*, ktorý vznikol ako výstup *callback* metódy pri trénovaní modelu. Uchováva v sebe informácie o modeli ako počet a typ jednotlivých vrstiev a váhy, ktoré boli upravené behom procesu trénovania.

Ako už bolo spomenuté, aby bolo možné model skompilovať a vytvoriť z neho súbor vo formáte *graph*, ktorý je kompatibilný s Movidius-om, je nutné poznať architektúru daného modelu. Najpodstatnejšou informáciou je *vstupná* a *výstupná* vrstva, ktoré sú povinnými argumentami pri volaní jednotlivých nástrojov NCSDK. Keďže *checkpoint* vo formáte *hdf5* nieje čitateľný pre človeka, tak pre túto prácu som vytvoril skript, ktorého úlohou je získať informácie zrozumiteľné pre človeka.

6.1.1 Skript scrape_hdf5.py

Tento skript pozostáva z troch metód, s ktorých každá má svoju špecifickú úlohu. Na vrchu celého stojí metóda **main**, v ktorej sa vykonáva celá funkcionalita.

Prvá metóda **get_model_architecture** si berie ako vstupný parameter objekt typu *model*. Následne z neho pomocou metódy **load_model**, ktorá sa nachádza v API *Keras*-u vráti celkovú architektúru modelu v *JSON* formáte. S tým je pomerne úzko spojená aj ďalšia metóda **load_json**, ktorá sa síce nevolá v **main** metóde, ale je ju možné využiť pre jednoduché načítanie obsahu JSON súboru.

Metóda **parse_and_save_model_data** si berie ako vstupné parametre cesty k modelu v *hdf5* súbore, cestu k *JSON* súboru, kam sa uloží architektúra modelu a cestu k *h5* súboru, kde sú uložené aktuálne váhy modelu. V rámci tejto metódy je na model volaná aj metóda **summary**, ktorá má pomerne veľký význam. Na štandardný výstup vypíše názov modelu a následne jeho jednotlivé vrstvy. Tak isto aj informácie o tvare výstupu

vrstiev, počte parametrov a na akú ďalšiu vrstvu je aktuálna napojená. Na obrázku 6.1 je možné vidieť časť výpisu ako ukážku. Presne tieto informácie sú potrebné na to, aby mohli byť použité nástroje NCSDK. Rovnaké informácie sa nachádzajú aj v *JSON* súbore s architektúrou modelu, táto verzia je však prehľadnejšia.

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 800, 600, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 399, 299, 32)      864
_________________________________________________________________
batch_normalization_1 (BatchNor (None, 399, 299, 32)   96
_________________________________________________________________
activation_1 (Activation)    (None, 399, 299, 32)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 397, 297, 32)      9216
_________________________________________________________________
batch_normalization_2 (BatchNor (None, 397, 297, 32)   96
```

Obr. 6.1: Príklad výpisu metódy **summary**

V poslednej časti tejto metódy prebieha ukladanie informácií o modele do odpovedajúcich súborov.

Za pomoci tohoto skriptu boli získané nasledujúce informácie. Vstupná vrstva sa volá *input_1* a výstupná vrstva *output_layer* s aktivačnou funkciou *softmax*. To znamená, že argumenty príkazov, ktoré sa volajú v rámci NCSDK majú tvar **-in=input_1** a **-on=output_layer/Softmax**. Na to, aby bolo možné model skompilovať na súbor *graph*, je potrebné previezť *Keras* model na *TensorFlow*. O túto funkcionalitu sa stará skript **keras_to_tf.py**.

6.1.2 Skript keras_to_tf.py

Tento skript je pomerne krátky. Na začiatku sa z *JSON* a *h5* súborov vytvorí model. Nastaví sa mu fáza učenia pomocou metódy **set_learning_phase** na nulu. To znamená, že model nebude použitý na učenie. Vytvorí sa objekt TensorFlow modulu *train* a jeho triedy **Saver()**. Ten má za úlohu uložiť model aj so všetkými informáciami. Tento krát je to však *TensorFlow* a nie *Keras* model, uložený vo formáte *meta*. Týmto bol získaný posledný argument pre NCSDK nástroje. V tomto momente je všetko pripravené na skompilovanie modelu.

Rovnako ako aj predošlý skript, oba sú zdokumentované a priložené ako príloha k tejto práci.

6.1.3 Kompilácia modelu a vytvorenie graph súboru

Podmienkou na vytvorenie *graph* súboru z aktuálneho *TensorFlow* modelu, je zadanie vtupnej a výstupnej vrstvy v správnom tvare a so správnou aktivačnou funkciou. Ako bolo spomenuté v kapitole 4.4.3, v prípade, že niektorá zo zadaných vrstiev v danom tvare neexistuje, vyvolá sa výnimka. Pri kompilovaní tohoto modelu však nastal iný problém.

To, že NCSDK nevyvolal žiadnu výnimku znamená, že vstupná aj výstupná vrstva boli zadané v správnom tvare. Iná výnimka však bola vyvolaná knižnicou *Numpy* v súbore **FusedBatchNorm.py**, ktorý sa nachádza medzi ovládačmi NCSDK. Je znázornená na obrázku 6.2.

```
  File "/usr/local/bin/ncsdk/Controllers/Parsers/TensorFlowParser/FusedBatchNorm.py", li
ne 36, in load
    scale = np.reciprocal(np.sqrt(variance)) * scale_param
ValueError: operands could not be broadcast together with shapes (0,) (32,)
```

Obr. 6.2: ValueError výnimka vyvolaná pri kompilácii *InceptionV3* siete.

Podľa *Numpy* dokumentácie, **broadcasting** má za úlohu opísať, ako *Numpy* vykonáva aritmetické operácie na dvoch poliach s rôznymi dimenziami. Medzi všeobecné pravidlá platí, že 2 polia sú kompatibilné, pokiaľ majú buď *rovnakú veľkosť*, alebo jedno z nich má rozmer veľkosti *1*. V ostatných prípadoch je vyvolaná *ValueError* výnimka. [45] V našom prípade ma jedno z polí veľkosť *0*, čiže je prázdne.

S menšou úpravou zdrojového súboru, z ktorého bola výnimka vyvolaná, bolo možné zistiť pravdepodobnú príčinu tejto chyby. Po vytvorení modelových polí s potrebnými rozmermi bola splnená **broadcasting** podmienka a žiadna *ValueError* výnimka vyvolaná nebola. Je nutné zdôrazniť slovíčko z prvej vety - *pravdepodobná*. Dôvodom je, že nebolo možné dohľadať príčinu chyby z overených zdrojov. Až takýmto experimentálnym postupom sme sa dopracovali k výnimke vyvolanej NCSDK - **Toolkit Error: Stage Details Not Supported**. Aj z tohoto dôvodu v rámci tejto práce vznikli ďalšie dva modely neurónových sietí, ktorým sa budú venovať nasledujúce odstavce. Bohužiaľ, túto sieť nebolo možné na Movidius naimplementovať.

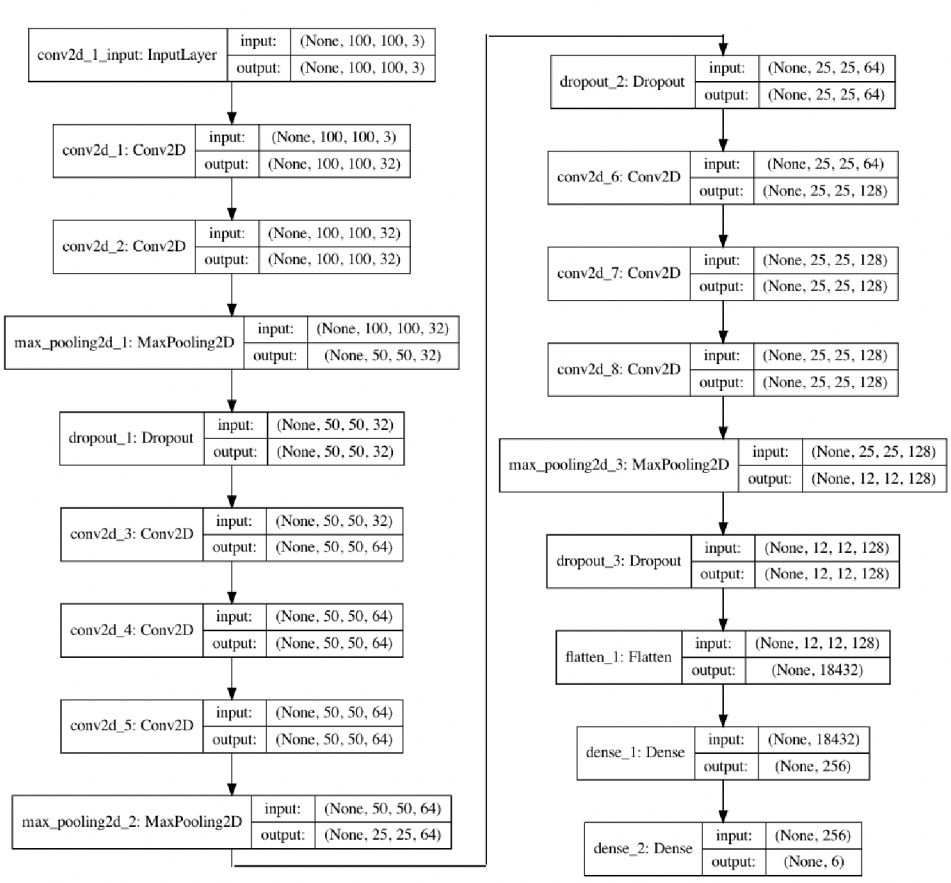
6.2 Sequential model neurónovej siete

V poradí celkovo druhý model bol vytvorený od samotných základov. Rozhodol som sa, že pre tento model využijem knižnicu *Keras* a konkrétne **Models** API. Toto pomerne jednoduché API ponúka viacero spôsobov, ako postupovať pri vytvorení modelu. Pre potreby tejto práce bol vybraný spôsob implementácie pomocou *Sequential* triedy, ktorá je pre klasifikáciu dát do skupín dostačujúca. Pre zložitejšie úlohy by však pravdepodobne mala isté limitácie. Sekvenčný model je vhodný pre aplikácie, ktorých vrstvy na seba hladko naväzujú a zároveň, každá vrstva má presne jeden vstupný a jeden výstupný tenzor.

6.2.1 Prehľad modelu

Model sa skladá celkovo z *18* vrstiev, ktoré sú znázornené na obrázku 6.3.

Základom naprogramovania sekvenčného modelu je vytvorenie objektu triedy *Sequential*. Jednotlivé vrtsvy sa buď zadajú priamo do konštruktora triedy, alebo pomocou druhého spôsobu, postupného pridávania vrstiev. V druhom prípade sa využíva metóda *add*. Tá vrstvy pridáva podobne ako do zásobníku. Posledne pridanú vrstvu je potom



Obr. 6.3: Graf znázorňujúci architektúru vlastného *sequential* modelu.

možné odstrániť pomocou metódy *pop*. [48]

Teória konvolučných neurónových sietí bola spomenutá v kapitole 1.2.1. Je ale vhodné spomenúť, ako sa vytvárajú jednotlivé objekty - vrstvy, definujúce model.

Conv2D vrstva

Rozmery a počet konvolučných jadier sú predané ako parametre konštruktoru triedy *Conv2D*, ktorá sa nachádza v **Layers** API. V našom prípade sú rozmery jadier *3x3*. Ich počet závisí od poradia vrstvy. Prvá *Conv2D* vrstva ich obsahuje *32* a posledná *128*.

MaxPooling2D vrstva

Táto vrstva väčsinou nasleduje za vrstvou konvolučnou. Nazýva sa *poolingová* alebo tak isto aj *subsamplingová* vrstva. Rovnako ako pri predchádzajúcej vrstve, aj pri *MaxPooling2D* sa rozmery poolingového jadra predávajú ako parametre konštruktoru triedy. V našom prípade sú rozmery jadra *2x2*. Ako už vyplýva z názvu, ide o poolingovú vrstvu, využívajúcu *max pooling* techniku.

Dropout vrstva

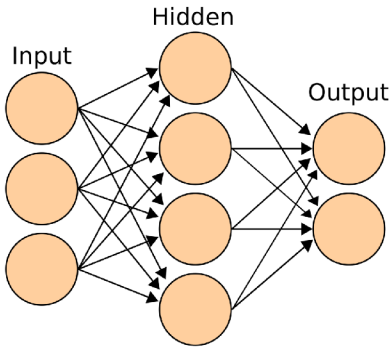
Nachádza sa za každou *poolingovou* vrstvou. Pre potreby tohoto modelu boli *dropout* hodnoty nastavené na *0.25*, čiže s pravdepodobnosťou jednej štvrtiny sa pri trénovaní nastavuje výstup z neurónu na nulu.

Flatten vrstva

Flatten vrstva sa zvyčajne nachádza medzi *konvolučnými* a *plne prepojenými* vrstvami. Jej hlavnou úlohou je pretransformovať dvojrozmerné pole hodnôt do jednorozmerného vektoru.

Dense vrstva

Dense vrstva už nepatrí medzi *konvolučné*, ako tomu bolo v predošlých vrstvách, ale medzi *plne prepojené* vrstvy. Každý neurón predchádzajúcej vrtvy je teda spojený s každým neurónom *plne prepojenej*, v tomto prípade, *Dense* vrstvy. Výstupný vektor z *Dense* vrstvy má rovnakú veľkosť, ako je počet neurónov v plne prepojenej vrstve. [57] Pre predstavu, *plne prepojené* neurónové vrstvy sú vyobrazené na obrázku 6.4.



Obr. 6.4: Ukážka plne prepojených neurónových vrstiev. [58]

Activation vrstva

Úlohou tejto vrstvy je aplikovanie aktivačnej funkcie na výstup. V prípade tohoto modelu bola použitá *ReLu - rectified linear unit* a *Softmax* aktivačná funkcia. *Softmax* je použitá v poslednej plne prepojenej vrstve modelu. Pre problém, ktorým sa zaoberá táto práca je využitie tejto aktivačnej funkcie najvhodnejšie.

Pre tento model bol ako **optimizačný algoritmus** využitý algoritmus *Adam*. *Adam* kombinuje metódu *gradientného zostupu* spoločne s metódou *kvadratického priemeru*. Ako **loss funckia** bola použitá *Cross-Entropy Loss* funkcia.

6.2.2 Implementácia modelu

Proces vytvorenia, kompilovania a trénovania tohoto modelu bol definovaný v celkovo dvoch skriptoch. Ten prvý, **run_custom_model.py** je spustiteľný skript, volajúci jednu metódu - *main*. Jej úlohou je nastaviť jednotlivé parametre modelu, ktoré predá konštruktoru triedy *ImageClassifier*. Následne na novo vytvorený objekt zavolá metódy, ktorých definície sú v druhom skripte a implementujú celkovú funkcionalitu modelu.

ImageClasiffier trieda

Touto triedou je definovaný celý model neurónovej siete pre klasifikáciu dát do jednotlivých kategórii. Obsahuje všetky potrebné metódy, definované nižšie, až na metódu testovania úspešnosti modelu. Tá je obsiahnutá v inom skripte.

V **build_model** metóde vzniká objekt triedy *Sequential*, ktorý tvorí kostru modelu. Pomocou metódy *add* sa postupne pridávajú do modelu jednotlivé vrstvy. Model je následne touto metódou vrátený. Na úplnom začiatku je ešte volaná metóda *image_data_format*, ktorá sa nachádza v *backend* module knižnice *Keras*. Každá knižnica inak spracováva viacrozmerné polia ktoré dostane ako vstupné dáta. Napríklad *TensorFlow* si ukladá počet farebných kanálov obrazu ako posledný rozmer, *Theano* naopak ako prvý. Úlohou *image_data_format* metódy je teda zistiť ako vstupné dáta ukladať a vrátiť *tuple*, obsahujúci poradie rozmerov obrazu.

compile_and_save_model je najkomplexnejšou metódou z popisovaného skriptu. V tejto časti prebiaha kompilácia a tréning modelu. Vstupnými parametrami metódy sú dáta, rozdelené na validačné a tréningové, iterátor triedy *ImageDataGenerator*, iterujúci cez jednotlivé *batch*-e *augmentovaných* dát (batch size - počet tréningových dát s ktorými pracuje neurónová sieť pri jednej iterácii) a samotný model. Kompilovanie modelu zabezpečuje volanie metódy *compile*. V tomto momente je možné nakonfigurovať *loss funkciu* a *optimizačný algoritmus*.

Po úspešnej kompilácii modelu nasleduje proces učenia. Keras ponúka vo svojom API tri metódy, ktoré programátor môže využiť. Pri používaní *data augmentation* je jedinou vhodnou metódou na trénovanie metóda *fit_generator*. Podstatnou časťou je vytvorenie objektu triedy *ImageDataGenerator*, aplikujúci *data augmentation*, ako bolo spomenuté v kapitole 5.4. Proces trénovania sa opakuje, pokiaľ sa nevykoná daný počet epoch. Druhou možnosťou je definovať si *callback* funkciu, ktorá učenie zastaví ak sa splnia určité podmienky. Pri trénovaní vlastného *sequential* modelu bola využitá práve druhá možnosť. Boli odskúšané rôzne nastavenia parametrov, ale najviac sa osvedčili nasledovné. Učenie sa ukončí, pokiaľ 10 po sebe idúcich epoch nemalo zlepšenie v hodnote *loss* funkcie pri validácii na konci epochy. Za zlepšenie sa považuje, pokiaľ hodnota *loss* funkcie klasne aspoň o *0.0001*.

Na natrénovaný model je následne volaná metóda *evaluate*, ktorá ho otestuje a vyhodnotí na validačných dátach. Takto naučený model sa ukladá na miesto definované cestou, o čo sa stará ďalšia metóda, ktorú obsahuje tento skript - **save_weights**. Poslednými dvoma vytvorenými metódami sú **preprocessing_data** a **plot_graph**. Prvá menovaná spracováva tréningové dáta. Iteratívne prechádza všetky priečinky na ceste *data/train* a jednotlivé obrazy spracováva pomocou knižnice *opencv-python* a *numpy*. Tie sú uložené do dvoch objektov typu *list*. Prvý obsahuje informácie o hodnotách jednotlivých pixelov - trojrozmerné polia typu *numpy.array* pre každý obraz. Druhý obsahuje označenie, do ktorej kategórie jednotlivé dáta z prvého listu patria. Druhá metóda **plot_graph**, vytvára grafy naučeného modelu, aké boli *loss* a *accuracy* hodnoty pre jednotlivé epochy. Tie sú následne uložené na disk.

6.2.3 Kompilácia modelu a vytvorenie graph súboru

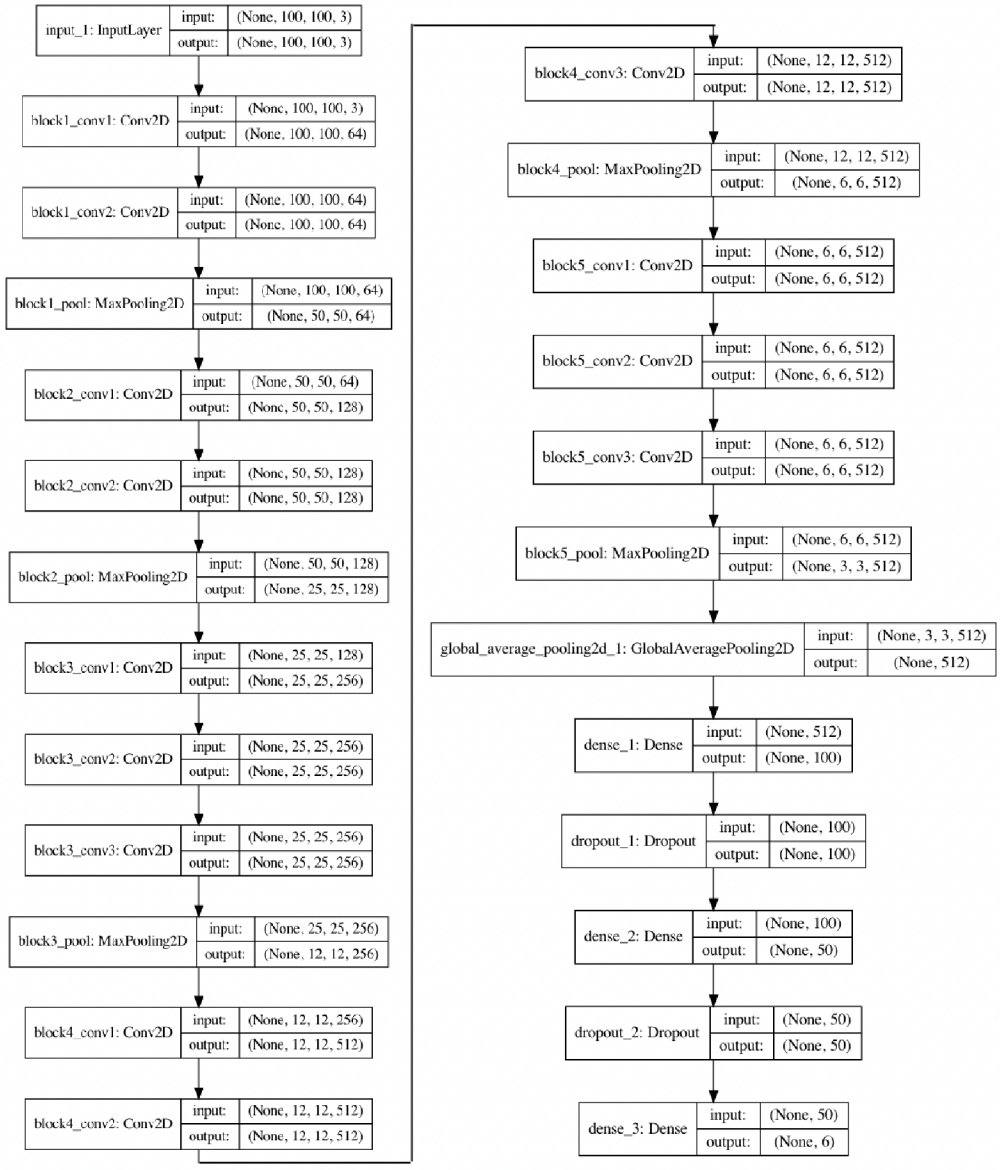
Rovnako ako pri predošlom modele, aj v tomto prípade bolo potrebné previezť *Keras* modely na *TensorFlow*. Opäť to bolo dosiahnuté pomocou skriptu **keras_to_tf.py**. Vstupná vrstva má názov *conv2d_1*. Názov výstupnej vrstvy je *dense_2* a tá používa aktivačnú funckiu *Softmax*. Pre úspešnú kompiláciu a vytvorenie *graph* súboru, potrebného pre komunikáciu s Movidius-om je potrebný príkaz **mvNCCompile TF_Model/tf_model.meta -in=conv2d_1 -on=dense_2/Softmax**. *Graph* súbor bol úspešne vytvorený. Pre úplnú istotu, že vytvorený súbor bude bez akýchkoľvek problémov komunikovať z Movidiusom, bol ešte vyskúšaný príkaz **mvNCCheck**. Ako bolo spomenuté v kapitole 4.4, za normálnych okolností by výstupom v termináli bola tabuľka hodnôt, označujúca správnosť a kompatibilitu testovaného modelu. V tomto prípade bola ale výstupom vyhodená výnimka - **Status.ERROR**. Informácia **dispatcherEventReceive() Read failed -1** bola už trochu konkrétnejšia. Podľa oficálneho fóra Intel-u, kde na problémy zákazníkov reagujú zamestnanci technickej podpory tejto spoločnosti som narazil na rovnaký problém ako nastal v tomto prípade. V ich oficiálnom stanovisku stálo: *"Skontrolovali sme firmware a aj API, oba vyzerajú v poriadku. V momente kedy sa z firmware-u zavolá kód CNN, nieje vrátená žiadna hodnota, a po určitom čase sa stratí spojenie so zariadením (Movidius). Bohužiaľ nevidíme riešenie, ktorým by sa tento problém dal obísť."* [62] Tento model rovnako nieje použiteľný pre Movidius.

6.3 VGG16 model neurónovej siete

Posledný vytvorený model obsahoval základ v už existujúcej architektúre - *VGG16*. Rovnako sa tak už použili aj váhy, ktoré boli upravené pre klasifikáciu dát z databázy *ImageNet*. Pridaná bolo vlastná - plne prepojená vrstva, ktorej váhy boli natrénované od začiatku. Rovnako ako v predošlých prípadoch, aj teraz bolo využité *Keras* API. Architektúra, spôsob, akým bol model trénovaný a stručný popis vytvorených skriptov je popísaný v nasledujúcich odstavcoch.

6.3.1 Prehľad modelu

Tento model sa skladá z celkovo *25* vrstiev. Prvých *19* patrí VGG16 architektúre, zvyšok sú vrstvy plne-prepojené, patriac rovnako do **Layers** API Keras-u. Aké vrstvy celý model obsahuje je možné vidieť na obrázku 6.5. Objekt, ktorý obsahuje VGG16 architektúru vzniká z triedy *VGG16*, ktorá sa nachádza v module *applications*. Okrem VGG16 obsahuje plno ďalších známych architektúr, ktoré sú pomerne jednoducho implementovateľné. Medzi nimi rovnako aj *InceptionV3*.



Obr. 6.5: Graf znázorňujúci architektúru VGG16 modelu s plne prepojenými vrstvami.

V základe tento model pracuje s dátami - RGB obrázkami, každý o veľkosti *224x224*. V prípade, že programátor chce pracovať so vstupnými dátami iných rozmerov, musí konštruktoru predať premennú **include_top=False**. Štandardne je nastavená na hodnotu

True. Vtedy sa zároveň za konvolučné vrstvy pridajú tri preddefinované plne prepojené vrstvy. Keďže v tejto práci bola snaha o rôzne konfigurácie a zároveň o pridanie plne prepojených vrstiev podľa vlastného uváženia, **include_top** je nastavené na hodnotu *False*. Ďalším dôležitým parametrom je **weights**. Pomocou tohoto parametru sa inicializujú počiatočné váhy modelu. Štandardne sú nastavené na náhodné hodnoty, ale pre prípad tejto práce sa zvolili váhy, ktoré sa upravili trénovaním na *ImageNet* databáze. Týmito nastaveniami vzniká základ modelu.

Za konvolučnými vrstvami nasleduje jedna **GlobalAveragePooling2D** vrstva, ktorá podobne ako **Flatten** vrstva z predošlého modelu vytvorí z *4D* vektoru *2D* vektor, ktorý sa následne rozmerovo zhoduje so vstupným vektorom prvej plne prepojenej vrstvy. Za **GlobalAveragePooling2D** ďalej nasledujú **Dropout** a **Dense** vsrtvy.

6.3.2 Implementácia modelu

Podobne ako predošlý model, aj v pri tomto platí, že jeho implementácia je rozdelená do dvoch skriptov. Prvý, **run.py**, vytvára objekt triedy *ImageClassifier* a pomocou konštruktoru jej predáva paramtetre definované na začiatku *main* metódy. Na objekt sa potom volá metóda *run*, ktorej funkcionalita je popísaná nižšie.

Aj keď oba modely majú veľa podobností, tak štýl písania jednotlivých skriptov sa výrazne líši. Hlavným dôvodo je to, že vznikali nezávisle na sebe a v rôznych obdobiach, a tak isto pôvodne používali odlišné techniky. Skript, v ktorom je definícia celého modelu, spoločne so všetkými potrebnými metódami, aby ho bolo možné natrénovať na potreby tejto práce, sa nazýva **vgg16.py**. Začiatok skriptu obsahuje metódy **get_test_data** a **transform_array**. Obe sa používajú na vytvorenie testovacích dát - *data* a ich označení *label*, ktoré sú využívané technikami hodnotiacimi celkovú úspešnosť modelu, ako aj vytváraním grafov, ktoré sú spomenuté v nasledujúcej kapitole. Nasleduje metóda **get_input_shape**, ktorá podobne ako v predošlom príklade, určuje tvar vstupného vektoru.

create_model

Keďže všetky potrebné parametre sú buď predané objektu modelu v konštruktore, alebo vytvárané priamo v tejto metóde, nemusia sa jej predať žiadne argumenty. Veľmi podstatnú úlohu zohráva príkaz

Výpis 6.1: Freeze modelu VGG16

```
self.vgg16.trainable = False
```

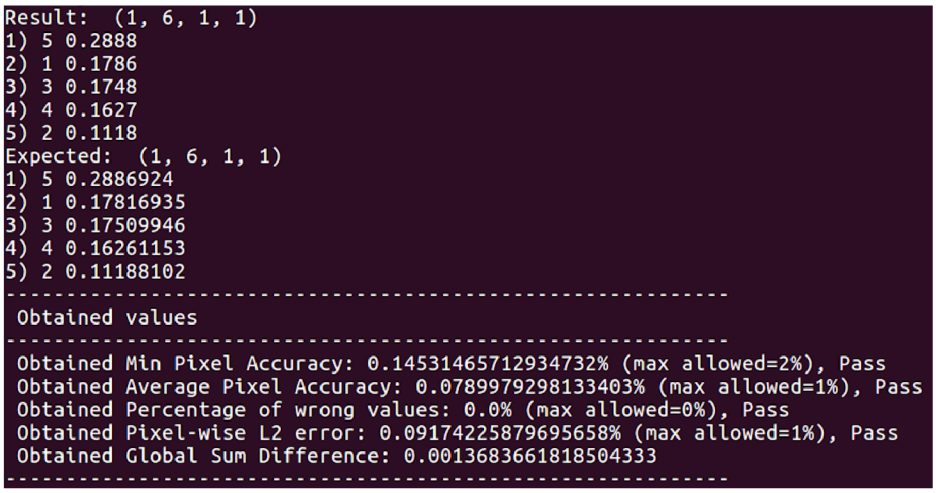
Na jednotlivé vrstvy modelu *VGG16*, ktorý je síce natrénovaný na klasifikáciu dát do tisíc kategórií, ale jeho váhy niesú úplne náhodne generované je možné zavolať metódu *trainable*. Pokiaľ sa jej hodnota nastaví na *False*, rovnako ako v našom prípade, vrstvy modelu sa zmrazia. To spôsobí, že následné trénovanie celého modelu (VGG16 spoločne

s plne prepojenými vrstvami) sa bude týkať len plne prepojených vrstiev. Ako bolo spomenuté na začiatku práce, samotné trénovanie *VGG16* je veľmi dlhou záležitosťou aj za predpokladu, že sa používa najvýkonnejší HW. Ako bude možné vidieť v kapitole spracovávajúcej výsledky, úspešnosť je stále veľmi vysoká.

Za touto časťou nasleduje proces trénovania. Aktivačné funkcie plne prepojených vrstiev sú rovnaké, ako v predošlom modeli. *ReLu*, respektíve pre úplne poslednú **Dense** vrstvu to je *Softmax*. Ako optimizačný algoritmus bol zvolený *SGD* optimizér. Učenie tohoto modelu neprebiehalo stále rovnaký a presne daný počet epoch, ale mu boli nastavené *callback* funkcie, rovnako ako pre *Sequential* model. Nastavenie sa však trochu líši. Keďže je použitá architektúra s už čiastočne predtrénovanými váhami, učenie nemusí prebiehať tak dlhý čas, ako pri predošlom modeli. Už po štyroch až piatich epochách je model prakticky natrénovaný pre riešenie problematiky tejto práce a ďalšie zlepšenia niesú až také razantné. Kontrolujú sa hodnoty *loss* funkcií pri validácii na konci každej epochy. Čaká sa na zlepšenie po dobu *5* epoch, ale za zlepšenie sa tentokrát považuje situácia, kedy je rozdiel hodnôt *loss* funkcií minimálne *0.001*. Pokiaľ zlepšenie nastane, model aj s upravenými váhami sa uloží na cestou definované miesto. Celý proces trénovania je stopovaný pomocou modulu *datetime* a výsledky sú porovnávané s ostatnými konfiguráciami. Poslednou metódou v skripte je metóda **load_model**, ktorá, ako už z názvu vyplýva, načíta model z cesty, ktorá je predaná ako argument metóde. Následne vráti objekt modelu.

6.3.3 Kompilácia modelu a vytvorenie graph súboru

Kompilácia a vytvorenie *graph* súboru prebehla rovnako ako pri predošlých modeloch. Spustením skriptu **keras_to_tf.py** sa vytvorili *TensorFlow* modely a následným príkazom **mvNCCompile** vnikol *graph* súbor. Argumentom **-in** bola vstupná vrstva s názvom *input_1* a **-on** zasa výstupná vrstva s názvom *dense_3/Softmax*. V tomto prípade žiadna chyba pri kompilácii nenastala. Pre istotu bol spustený ešte príkaz **mvNCCheck**, ktorý pre všetky merané veličiny vrátil **PASS**, čo znamená, že sa model správa tak, ako má. Situácia je znázornená na obrázku 6.6.



Obr. 6.6: Kontrola korektnosti VGG16 modelu.

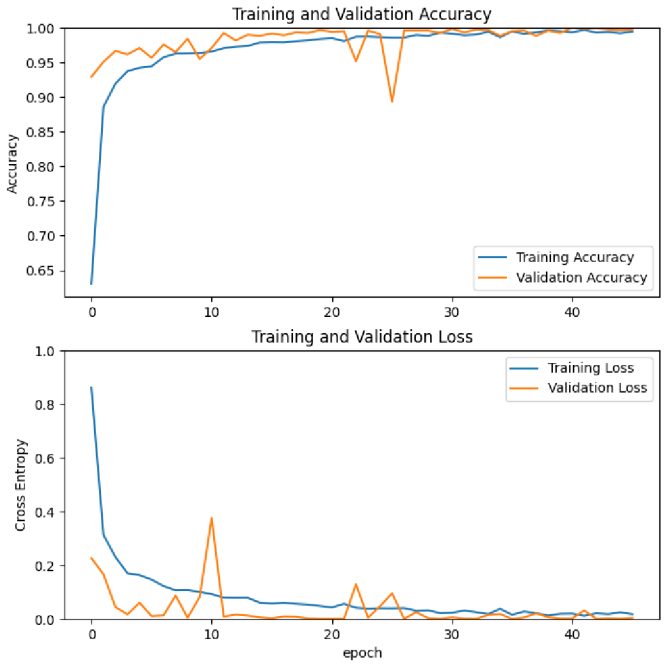
Tento model sa podarilo implementovať na *Raspberry Pi* s pripojeným *Movidius*-om.

7 Pomocné skripty

Táto kapitola sa v krátkosti venuje skriptom, ktoré nemali na zkonštruovanie a implementáciu modelu až taký vplyv. Sú to skôr skripty, ktoré vizuálne zobrazia priebeh učenia jednotlivých modelov, alebo ich samotné testovanie. Všetky spomenuté skripty v tejto kapitole, rovnako ako v predošlých, sú zdokumentované a priložené ako príloha k tejto práci a boli vytvorené od samotného začiatku. Jazyk, ktorý bol pri ich písaní použitý je Python. Sú buď spustiteľné priamo z príkazového riadku, alebo obsahujú definíciu triedy daného objektu, ktorý sa následne vytvára a inicializuje v iných skriptoch.

7.1 plot.py

V tomto skripte je definovaná trieda *Plot*, ktorá okrem konštruktoru obsahuje ešte jednu metódu - *create_plot*. Úlohou tejto triedy, ako už názov napovedá, je vytvoriť graf, ktorý znázorňuje ako prebiehalo učenie daného modelu. Dokopy sa vytvoria 3 grafy. Prvé 2 sú znázornené na obrázku 7.1. Horný graf (číslo 1) na ose **y** porovnáva presnosť predikcie výsledku - *accuracy* počas procesu trénovania a validácie. Na ose **x** sú znázornené epochy. Spodný graf (číslo 2) porovnáva hodnoty *cross entropy loss* funkcie - os **y**. Na osy **x** znázorňuje, rovnako ako v predošlom grafe, jednotlivé epochy.



Obr. 7.1: 2 grafy znázorňujúce *accuracy* a *loss* hodnoty počas trénovania modelu.

Tretí graf prakticky neposkytuje žiadne nové informácie, ktoré by neboli zaznamenané

na predošlých dvoch grafoch. Hlavnou myšlienkou tohoto grafu je znázorniť aj *accuracy* aj *loss* hodnoty do jedného grafu. V niektorých prípadoch sa to môže hodiť viac. Tento graf je znázornený na obrázku 7.2.

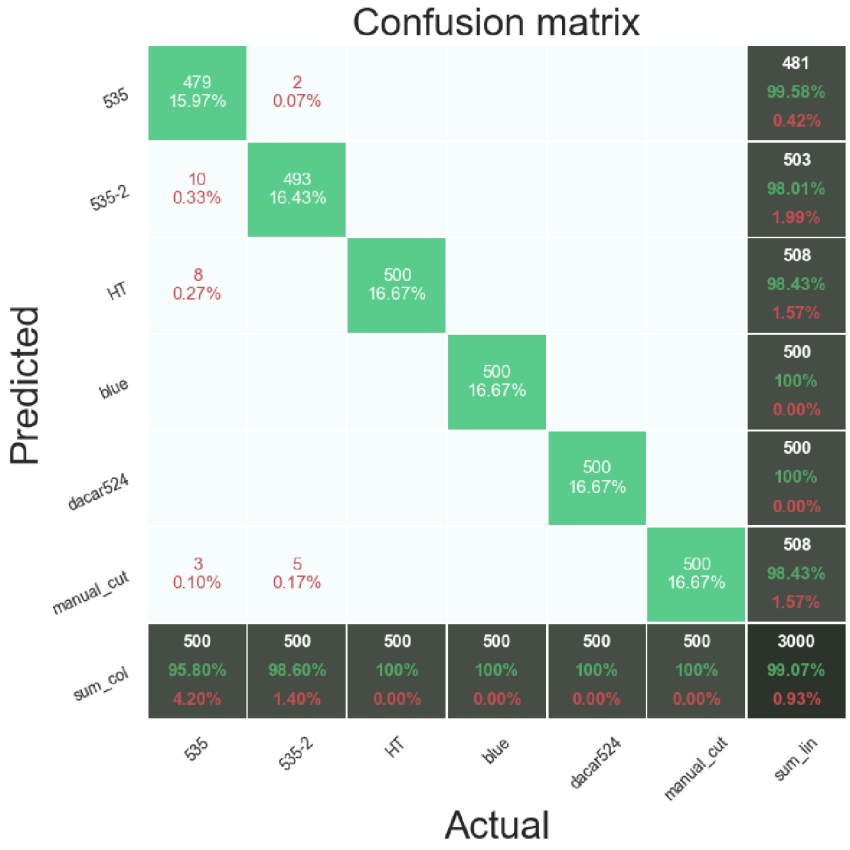
Obr. 7.2: Graf znázorňujúci *accuracy* a *loss* hodnoty počas trénovania modelu.

Hlavnou úlohou nasledujúcich dvoch skriptov je testovanie úspešnosti natrénovaného modelu na *testovacích* dátach, ktoré sieť nikdy pred tým nevidela. Aj keď oba skripty v konečnom dôsledku vykonávajú rovnakú funkcionalitu, tak implementácia je rozdielna. Je to z toho dôvodu, že **evaluate_on_cpu.py** primárne využíva *Keras* API a všetky výpočty sa dejú na CPU bez použitia *Movidius*-u. Na strane druhej **evaluate_on_movidius.py** používa primárne *mvnc* API, ktoré je súčasťou *NCSDK* a výpočty prebiehajú na *Movidius*-e.

7.2 evaluate_on_cpu.py

Skript sa skladá z troch metód. Metóda *prepare_model* načíta a vráti inštanciu modelu. V závislosti na uložení informácií o modeli sa buď načíta architektúra definovaná v *JSON* súbore a váhy v *h5* súbore zvlášť, alebo je celý model definovaný v *hdf5* súbore a pomocou metódy *load_model* z *Keras* API sa model vytvorí a vráti. Druhá definovaná metóda, *preprocess_data*, pretransformuje obrazové dáta na objekt typu *numpy.array*. Týmto sa zabezpečí kompatibilita so vstupnou vrstvou modelu a dáta môžu byť vyhodnocované. Poslednou metódou je metóda *predict*. V nej sa interatívne prechádzajú všetky testovacie dáta. Tie vstupujú ako parameter do *Keras* metódy *predict*, ktorá sa volá na model. Výstupom je pole o rovnakej dĺžke, ako je počet kategórií, do ktorých sa dáta klasifikujú. Z tohoto poľa sa vyberie maximálna hodnota, vytvorí sa objekt typu *dict*, ktorý ako kľúče obsahuje indexy poľa a ako hodnoty ich odpovedajúce kategórie. Následne sa

porovná hodnota daného kľúču slovníka s označením testovacích dát a pokiaľ sa zhoduje, tak model klasifikoval dáta správne. Okrem samotnej klasifikácie, v tejto metóde prebieha aj stopovanie času potrebného na otestovanie modelu a celková úspešnosť v podobe logov, dohromady uložené do textového súboru. Tieto namerané dáta sú potom ďalej spracované a vyhodnocované v kapitole spracovávania výsledkov. Posledná funkcionalita, ktorú obsahuje tento skript, je import triedy *ConfusionMatrix*. Pomocou nej vzniká *matica zámeny - confusion matrix* a ukladá sa do formátu *.png*. Aby ju bolo možné zkonštruovať, tak počas každej iterácie procesu predikcie, sa do dvojrozmerného *list*-u veľkosti *6x6* ukladá z každej testovanej kategórie počet testovaných dát a počet kategórií, ktoré neurónová sieť predikovala. Ukážka *confusion matrix* je na obrázku 7.3



Obr. 7.3: Confusion Matrix obsahujúca výsledky predikovania modelu.

Každý riadok *confusion matrix* odpovedá jednej kategórií, ktorá bola výstupom neurónovej siete. Stĺpce matice označujú skutočnú kategóriu, kam skúmaný obraz patrí. Na diagonále sú správne predpovedané dáta pre jednotlivé kategórie. V prípade, že je testovaný objekt predpovedaný nesprávne, tak sa pozícia v matici, kde sa pretína riadok označený predpovedanou kategóriou so stĺpcom, obsahujúcim správnu kategóriu inkrementuje o 1. Z matice je následne vidieť, s čím najčastejšie sa zamieňali dáta s jednotlivých kategórii a

aký bol pomer správnych predpovedí modelu.

7.3 evaluate_on_movidius.py

Tento skript bol pri jeho vzniku rozdelený do viacerých metód, ktoré nebudú úplne do hĺbky rozobraté. Rovnako ako aj ostatné skripty, je zdokumentovaný a priložený ako príloha k tejto práci. Ako bolo spomenuté vyššie, pracuje hlavne s *mvnc* API. Na začiatku sa testuje, či je *Movidius* pripojený k zariadeniu. V prípade, že nie, je vyvolaná výnimka. Nasleduje krok načítania súboru typu *graph*, ktorý bol vytvorený pomocou príkazu **mvNCCompile**, do vyrovnávajúcej pamäte. Ďalším krokom je vytvorenie dátovej štruktúry typu *fronta*, obsahujúca vstupy a výstupy načítaného *graph* súboru. Testovacie dáta sa spracovávajú rovnako ako v predošlom prípade. Ak sú v správnom formáte, pomocou metódy volanej na *graph* objekt *queue_inference_with_fifo_elem* vznikne komunikácia medzi *Movidiusom* a *mvnc* API. Model neurónovej siete začne predikovať do akej kategórie vstupné dáta patria a výsledok vráti ako pole a rovnakej dĺžke, ako je počet klasifikačných kategórií. Tieto údaje sú rovnako spracovávané ako v **evaluate_on_cpu.py**. Tak isto je meraný čas predikovania výsledku a úspešnosti klasifikácie ukladaný v podobe logov. Posledným krokom je vykonanie *cleanup*-u, kde je potrebné aby komunikácia so zariadením bola ukončená a zariadenie odstránené. [63]

8 Výsledky

Táto kapitola sa venuje porovnávaniu a vyhodnocovaniu nameraných dát. Namerané dáta vieme rozdeliť do dvoch kategórií. Prvá spracováva dáta, ktoré vznikali pri trénovaní modelov. Tréning prebiehal a dáta boli zozbierané na oboch vytvorených modeloch, ako aj na *Sequential* modele, tak aj na *VGG16*. Fáza trénovania sa vykonávala na notebooku *MacBook Pro*, ktorého špecifikácie sú spomenuté v kapitole 3.2. Druhá kategória sa zasa venuje testovaniu natrénovaných modelov. Testy prebiehali na piatich rôznych HW konfiguráciách a to konkrétne: *MacBook Pro*, *HP Elite Book*, *HP Elite Book + Intel Movidius*, *Raspberry Pi* a *Raspberry Pi + Intel Movidius*. Všetky dáta, ktoré boli zozbierané pri testovaní, sú uložené aj v textových súboroch, ktoré môžeme nazvať ako *logy*. Z procesu trénovania však bohužiaľ logy niesú, dĺžka celkového procesu trénovania a záznam o epochách bol len vypísaný do terminálu. Miesto textovej podoby sú ale vyobrazené v podobe grafov, spomenutých v kapitole 7.

8.1 Proces trénovania

Porovnávať dobu trénovania na rôznych platformách nedáva veľmi zmysel. V praxi sa na trénovanie využíva ten najvýkonnejší HW, ktorý je aktuálne k dispozícii. Rovnako tak aj trénovanie jedného nastavenia modelu sa nedeje pred každým jeho používaním. Preto som sa v práci skôr zameral na porovnávanie doby tréningu viacerých konfigurácií. Pre oba modely vzniklo dohromady 8 rôznych nastavení. Menili sa tri premenné a to konkrétne:

- veľkosť vstupných dát - **100x100 px** / **200x200 px**
- batch size - **16** / **32**
- learning rate - **0,001** / **0,0001**

Na trénovanie modelov bolo použitých celkovo *5925* dát. Z tohoto množstva sa na validáciu použilo *1200* dát, ako bolo spomenuté v kapitole 5.3. Pri každom tréningu sa merali hodnoty loss, accuracy, validation loss a validation accuracy zaznamenané v tabuľke 8.1. V tabuľke 8.2 je možné vidieť celkový čas potrebný na natrénovanie, poradie epochy v ktorom bol dosiahnutý najlepší výsledok a celkový počet epoch, ktoré sa model trénoval.

Počiatočné nastavenia modelov			Loss [-]		Accuracy [-]		Validate loss [-]		Validate accuracy [-]	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16
100x100	16	0.001	0,0239	0,0298	0,993	0,9924	1,65E-04	6,55E-07	0,9967	1
100x100	16	0.0001	0,0202	0,0986	0,9924	0,9695	0,002	0,0092	1	0,9992
100x100	32	0.001	0,0425	0,0359	0,9884	0,9922	4,59E-04	2,74E-06	0,9967	1
100x100	32	0.0001	0,0586	0,0688	0,979	0,9803	0,0381	2,00E-05	0,9958	1
200x200	16	0.001	0,0401	0,0313	0,9869	0,9913	6,61E-05	9,01E-05	1	1
200x200	16	0.0001	0,0247	0,0906	0,9915	0,9733	0,0047	8,11E-05	1	1
200x200	32	0.001	0,0125	0,0696	0,9947	0,981	1,05E-05	3,06E-05	1	1
200x200	32	0.0001	0,0277	0,2477	0,9922	0,9122	0,0148	0,0075	0,9983	1

Tab. 8.1: Tabuľka hodnôt *loss* a *accuracy* pri trénovaní a validácii modelu.

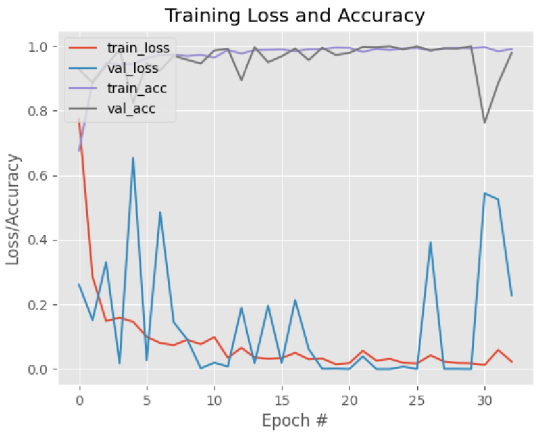
Prvým predpokladom, z ktorého som vychádzal bola veľkosť vstupných dát. Zaujímalo ma, ako sa zmení doba tréningu a celková úspešnosť modelu, pokiaľ na vstup pošlem 4-krát

Počiatočné nastavenia modelov			Čas potrebný na trénovanie [hh:mm:ss]		Priemerný čas trénovania na epochu [mm:ss]		Epocha s najlepším výsledkom [-]		Posledná epocha [-]	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16
100x100	16	0.001	01:06:49	00:49:25	02:34	06:11	22	8	26	8
100x100	16	0.0001	01:36:35	00:57:15	02:12	06:22	41	6	44	9
100x100	32	0.001	01:10:24	01:02:48	02:12	06:59	20	5	32	9
100x100	32	0.0001	00:51:01	01:53:26	02:13	05:58	20	10	23	19
200x200	16	0.001	04:09:33	03:42:28	08:55	24:43	19	7	28	9
200x200	16	0.0001	02:39:11	04:03:01	08:23	33:40	18	8	19	9
200x200	32	0.001	05:49:40	03:23:18	08:20	25:25	41	5	42	8
200x200	32	0.0001	03:53:06	05:41:58	08:38	22:48	24	11	27	15

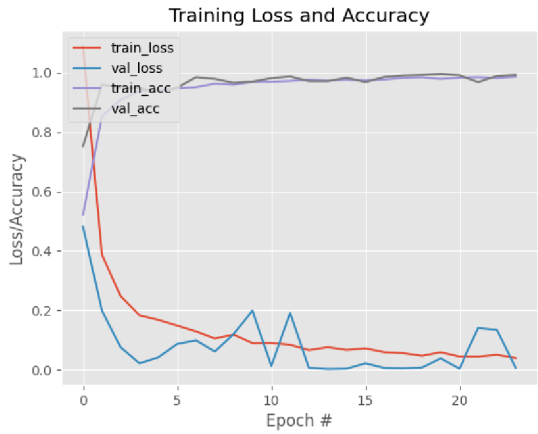
Tab. 8.2: Tabuľka doby trénovania modelu.

väčšie dáta - o veľkosti **200x200 px**. Z tabuľky 8.2 je možné si všimnúť, že priemerná doba trvania jednej epochy sa rovnako, ako veľkosť vstupných dát, zvýšila približne štvornásobne. To platí ako aj pre *Sequential*, tak aj pre *VGG16* model. Nemyslím si však, že pri väčšom rozmere vstupných dát sme získali aj väčšiu tréningovú, alebo validačnú presnosť. Pre oba rozmery je hodnota *validate loss* veľmi blízka *0*. Tak isto aj *validate accuracy* je takmer *1*.

Ďalším parametrom, od ktorého som očakával vplyv pri trénovaní je *learning rate*. Ako už bolo spomenuté v kapitole 1.2.1, príliš vysoká hodnota môže spôsobiť osciláciu hodnoty *loss* funkcie, naopak príliš nízka predlžuje dobu trénovania. Nedá sa povedať, že by modely s nastavením *learning rate* - **0,0001** potrebovali výrazne dlhšiu dobu na natrénovanie. Čo ale vieme povedať, a je to možné vidieť práve v grafoch, pri nižšej hodnote *learning rate* - **0,001**, hodnota *loss* funkcie pomerne dosť výrazne oscilovala, oproti jemnejšiemu kroku učenia. Túto situáciu je možné pozorovať na grafe 8.1, ktorý odpovedá nastaveniu *learning rate* - **0,001**. Na ľavej strane na grafe 8.2 je potom nastavenie s *learning rate* - **0,0001**. Rozmery vstupných dát sú **100x100 px** a *batch size* je **32**.



Obr. 8.1: Graf Sequential modelu s 0,001 learning rate



Obr. 8.2: Graf Sequential modelu s 0,0001 learning rate

Modrá krivka určuje hodnotu *validation loss* funkcie. Na grafe 8.1 je prekmitov o niečo viac, rovnako v peaku dosahujú hodnoty nad *0,5*. Graf s menšou hodnotou *learning rate* má peaky približne pri *0,2*. Tieto závislosti platia aj pre *VGG16* model. Však tým, že potreboval menej epoch na celkový tréning, je to viac viditeľné práve pre *Sequential* model. Ostatné grafy sú priložené ako príloha k tejto práci.

Voľba *batch size* nemala výrazný vplyv na tréningový proces modelov. Je možné, že pri väčšom rozdiely hodnôt, napríklad **16** a **64** by bol viditeľný rozdiel v *accuracy*. To bol aj predpoklad, že menšie hodnoty *batch size* prekonajú tie väčšie v presnosti. Čo sa však potvrdilo bolo, že model *VGG16* potreboval menej epoch na natrénovanie. To je spôsobené práve použitím už existujúcich váh a následným dolaďovaním, pre konkrétne využitie. Z tabuľky 8.2 je možné vyčítať, že pre *VGG16* architektúru bolo potrebných o približne polovicu menej epoch na natrénovanie.

8.2 Proces testovania

Pri testovaní som sa zameriaval na dva hlavné ciele. Prvým bolo zistiť, ako úspešné sú jednotlivé natrénované konfigurácie. Záznamy o úspešnosti sú pre každú jednú kategóriu, a tak isto aj ako celková úspešnosť modelu. Druhým cieľom bolo odmerať, ako dlho trvá proces testovania na rôznych platformách, ktoré som mal k dispozícii. Meranie času prebiehalo medzi každým testovaným obrázkom a rovnako bol zaznamenaný aj celkový čas, potrebný pre otestovanie všetkých *3000* testovacích dát. Okrem toho bol zaznamenaný aj čas potrebný na predspracovávanie dát, aby ich bolo možné použiť na testovanie.

8.2.1 Úspešnosť jednotlivých konfigurácií modelov

V tabuľkách 8.3 a 8.4 je možné vidieť, s akou presnosťou každé nastavenie modelu klasifikovalo jednotlivé dáta. Keď sa pozrieme na jednotlivé kategórie, tak najviac nesprávnych predikcií bolo pre kategórie **535** a **535-2**. Tento fakt nieje až taký prekvapujúci, dáta z oboch kategórií vyzerajú naozaj podobne. Zvyšné 4 kategórie mali vo väčšine konfigurácií *100%* úspešnosť. Jediná hodnota, ktorá ma prekvapila, bola úspešnosť **blue** kategórie, kedy v dvoch prípadoch bola nižšia ako *90%*.

Percentuálna úspešnosť testovania jednotlivých konfigurácií modelov Sequential a VGG16 [%]										
Počiatočné nastavenia modelov			HT		dacar524		manual_cut		Celková úspešnosť	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16
100x100	16	0.001	99,8	100	100	100	100	100	95,73	99,17
100x100	16	0.0001	100	100	100	99,8	100	100	99,07	97,2
100x100	32	0.001	100	100	100	100	100	100	99,53	99,23
100x100	32	0.0001	99,8	99,2	100	100	98	100	98,03	99
200x200	16	0.001	100	100	100	100	100	100	98,97	99,47
200x200	16	0.0001	100	100	100	100	99,8	100	98,9	99,43
200x200	32	0.001	100	100	100	100	100	100	97,73	97,5
200x200	32	0.0001	100	100	100	99,8	99,8	100	99,67	99,13

Tab. 8.3: Tabuľka úspešnosti testovania jednotlivých konfigurácií modelov 1/2

Percentuálna úspešnosť testovania jednotlivých konfigurácií modelov Sequential a VGG16 [%]								
Počiatočné nastavenia modelov			535		535-2		blue	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16
100x100	16	0.001	79,4	98,6	95,2	100	100	96,4
100x100	16	0.0001	95,8	87,2	98,6	96,2	100	100
100x100	32	0.001	97,8	95,8	99,4	99,6	100	100
100x100	32	0.0001	95,2	94,8	95,2	100	100	100
200x200	16	0.001	95	96,8	98,8	100	100	100
200x200	16	0.0001	95	98,4	98,6	98,6	100	99,6
200x200	32	0.001	98,6	96,2	98,4	100	89,4	88,8
200x200	32	0.0001	98,6	96,4	99,6	98,6	100	100

Tab. 8.4: Tabuľka úspešnosti testovania jednotlivých konfigurácií modelov 2/2

Oba vzniknuté modely majú veľmi vysokú celkovú úspešnosť na testovacích dátach, okolo *99%*. Najväčšiu má *Sequential* model, pri rozmeroch vstupných dát **200x200 px**, *batch size* - **32** a *learning rate* - **0,0001** a to konkrétne *99,67%*. Je však potrebné podotknúť, že tá istá konfigurácia, len s veľkosťou vstupných dát 4-krát menšou a rovnako tak aj dobou trénovania, mala úspešnosť len o 15 stotín nižšiu - *99,53%*. Čas potrebný na natrénovanie však bol výrazne nižší - *1 hodina a 10 minút* oproti *5 hodín a 49 minút*.

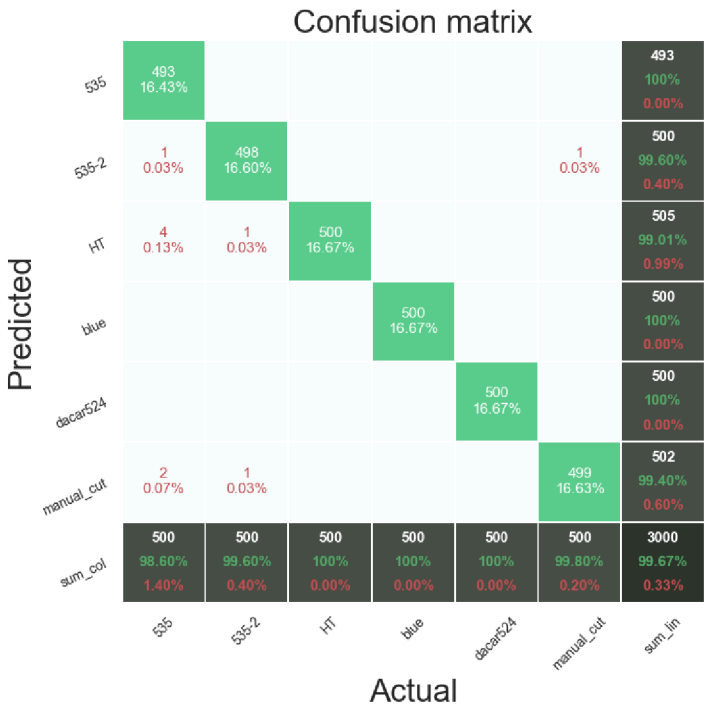
Pre zachovanie úplnosti som na tých istých dátach otestoval aj *InceptionV3* architektúru, ktorú som už natrénovanú dostal. Jej úspešnosť je možné vidieť v tabuľke 8.5.

Percentuálna úspešnosť testovania modelu InceptioV3 [%]							
535	535-2	blue	HT	dacar524	manual_cut	tréningová úspešnosť	testovacia úspešnosť
97,6	99,4	86	99,8	83,2	99,6	99,83	94,27

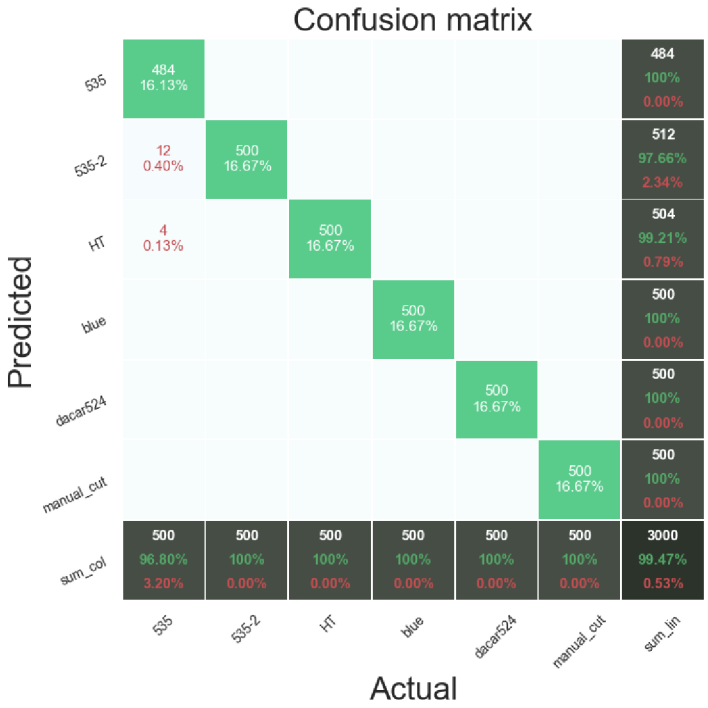
Tab. 8.5: Tabuľka úspešnosti testovania už predtrénovaného modelu InceptionV3

Podľa názvu *.hdf5* súboru - **weights-improvement-06-0.9983.hdf5**, ktorý obsahuje váhy a architektúru *InceptionV3* modelu som usúdil, že pravdepodobná presnosť po natrénovaní bola *99,83%*. Po testovaní mi však vyšla hodnota *94,27%*. O dosť nižšie hodnoty úspešnosti boli dosiahnuté hlavne pri kategóriách **blue** a **dacar524**. Pre názornosť nasledujúce dva grafy, obsahujúce *confusion matrices* znázorňujú, ktoré kategórie s ktorými sa najviac zamieňali pri predikcii. Graf 8.3 odpovedá *confusion matrix* pre *Sequential* model s najvyššou úspešnosťou, graf 8.4 zasa nastaveniu *VGG16* architektúry s najvyššou presnosťou.

V oboch prípadoch sa potvrdilo, že kategória s najväčšou chybovosťou je kategória **535**. Čo je ale pomerne zaujímavé, hlavne pri grafe 8.4, v 12-tich prípadoch bola táto kategória zamenená za kategóriu **535-2**. Avšak ani v jednom prípade to neplatilo opačne, aby dáta z kategórie **535-2** boli klasifikované do kategórie **535**.



Obr. 8.3: Confusion matrix *Sequential* modelu s najvyššou úspešnosťou



Obr. 8.4: Confusion matrix *VGG16* modelu s najvyššou úspešnosťou

8.2.2 Čas potrebný na testovanie pre jednotlivé platformy

V predchádzajúcej podkapitole bola zhrnutá úspešnosť jednotlivých konfigurácií. Tá je medzi jednotlivými platformami nemenná. Na druhej strane čas potrebný na otestovanie jednotlivých dát sa s každým použitým HW líši. Predpokladal som, že čím výkonnejší, alebo pre danú aplikáciu vhodnejší HW je použitý, tým je kratšia doba potrebná na spracovanie a otestovanie jednotlivých dát. Na meranie časových úsekov bol použitý Python modul *time* a konkrétne metóda *process_time()*. Táto metóda vracia na výstup čas v zlomkoch sekundy, ktorý spotreboval systém a CPU na vykonanie daného procesu. Do tejto doby nieje započítaný *sleep time* [64].

V nasledujúcich troch tabuľkách je možné vidieť, akú dlhú dobu trvalo CPU predikovať *jeden* obrázok, rovnako ako všetkých *3000*. Zaznamený je aj čas, potrebný na predspracovanie a predikciu jedného obrázku, spoločne s celkovým časom. *Predspracovanie* dát označuje proces načítania dát nachádzajúcich sa na danej ceste, následná úprava rozmerov obrázku na žiadanú veľkosť, pretypovanie na objekt typu *numpy.array* a normalizácia každej farebnej zložky a každého pixelu obrazu do intervalu *<0, 1>*. Tabuľka 8.6 odpovedá HW *MacBook Pro*, 8.7 HW *HP Elite Book* a v tabuľke 8.8 je možné vidieť čas predikcie dát na *Raspberry Pi 3B+*.

MacBook Pro										
			Bez predspracovávania vstupných dát				S predspracovávaním vstupných dát			
Počiatočné nastavenia modelov			Priemerný čas [ms]		Celkový čas [s]		Priemerný čas [ms]		Celkový čas [s]	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16
100x100	16	0.001	41,73	233,00	125,19	699,00	44,23	236,95	132,68	710,85
100x100	16	0.0001	41,68	213,72	125,03	641,16	44,20	217,24	132,61	652,03
100x100	32	0.001	42,32	222,61	126,96	667,84	44,85	226,38	134,55	679,13
100x100	32	0.0001	39,25	226,21	117,76	678,64	41,59	230,08	124,77	690,24
200x200	16	0.001	138,88	704,94	416,64	2114,81	141,60	708,77	424,79	2126,32
200x200	16	0.0001	145,68	690,11	437,05	2070,32	148,55	693,88	445,64	2081,63
200x200	32	0.001	136,72	719,56	410,16	2158,69	139,42	723,48	418,25	2170,45
200x200	32	0.0001	143,42	728,17	430,25	2184,50	146,25	732,13	438,76	2196,38

Tab. 8.6: Tabuľka doby trvania predikcie jednotlivých dát na *MacBook Pro*

HP Elite Book										
			Bez predspracovávania vstupných dát				S predspracovávaním vstupných dát			
Počiatočné nastavenia modelov			Priemerný čas [ms]		Celkový čas [s]		Priemerný čas [ms]		Celkový čas [s]	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16
100x100	16	0.001	53,27	169,46	159,80	508,38	56,38	171,89	169,13	515,67
100x100	16	0.0001	53,62	171,36	160,85	514,07	56,76	173,79	170,28	521,36
100x100	32	0.001	53,24	236,31	159,73	708,92	56,34	239,09	169,02	717,26
100x100	32	0.0001	53,48	235,72	160,45	707,16	56,60	238,48	169,81	715,44
200x200	16	0.001	210,09	916,41	630,27	2749,24	213,15	919,44	639,45	2758,32
200x200	16	0.0001	211,33	915,42	633,98	2746,27	214,63	918,45	643,88	2755,36
200x200	32	0.001	209,90	914,54	629,70	2743,63	212,89	917,55	638,68	2752,65
200x200	32	0.0001	209,64	914,22	628,91	2742,65	212,65	917,24	637,95	2751,71

Tab. 8.7: Tabuľka doby trvania predikcie jednotlivých dát na *HP Elite Book*

Prvé dve tabuľky, porovnávajúce výpočty na notebook-och, neukazujú veľké rozdiely.

Raspberry Pi 3B+										
			Bez predspracovávania vstupných dát				S predspracovávaním vstupných dát			
Počiatočné nastavenia modelov			Priemerný čas [ms]		Celkový čas [s]		Priemerný čas [ms]		Celkový čas [s]	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16	Sequential	VGG16
100x100	16	0.001	673,34	3043,80	2020,01	9131,41	688,73	3059,75	2066,19	9179,24
100x100	16	0.0001	662,38	3043,15	1987,14	9129,44	677,76	3058,66	2033,27	9175,99
100x100	32	0.001	666,57	3042,55	1999,71	9127,66	682,04	3058,06	2046,12	9174,19
100x100	32	0.0001	674,38	3045,57	2023,15	9136,71	690,38	3061,17	2071,13	9183,50
200x200	16	0.001	2499,34	11197,33	7498,03	33591,99	2517,84	11215,74	7553,52	33647,22
200x200	16	0.0001	2499,29	10992,86	7497,88	32978,57	2517,72	11011,02	7553,16	33033,05
200x200	32	0.001	2494,96	11326,21	7484,88	33978,63	2513,03	11344,78	7539,09	34034,34
200x200	32	0.0001	2504,65	11013,88	7513,95	33041,63	2523,34	11032,16	7570,03	33096,49

Tab. 8.8: Tabuľka doby trvania predikcie jednotlivých dát na *Raspberry Pi 3B+*

Podobnú situáciu som aj očakával. Pri meraní doby trvania procesu na CPU, alebo inej procesorovej jednotke, záleží na viacerých okolnostiach. Ako aj na operačnom systéme, na ktorom daná aplikácia beží, tak aj na aktuálne spustených procesoch v dobe merania. Keďže obe platformy, na ktorých testy prebiehali sú pre tento účel pomerne výkonné, nieje vo výsledkoch až taký razantný rozdiel. Čas potrebný na predspracovanie dát pre testovanie je minimálny. Pre obe platformy to sú približne *3 ms*, čo je oproti celkovej dobe zanedbateľný rozdiel. Pri *Raspberry Pi* je to však iné. Toto meranie naplnilo predpoklady. Rozdiel v počte jadier, ich frekvencii, architektúre a veľkosti pamäte RAM je značný. Dáta sú približne *12* až *13-krát* pomalšie vyhodnocované. Rovnako tak aj predspracovávanie dát trvá približne *5-krát* dlhšie. Toto zariadenie je teda vhodný adept na využitie výpočetnej sily Movidius-u. V tabuľke 8.9 je ešte porovnanie dĺžky testovania modelu *InceptionV3* pre všetky 3 vyššie spomenuté platformy.

	Inception V3			
	Bez predspracovávania vstupných dát		S predspracovávaním vstupných dát	
	Priemerný čas [ms]	Celkový čas [s]	Priemerný čas [ms]	Celkový čas [s]
MacBoo Pro	101,92	305,75	105,91	317,73
HP Elite Book	81,86	245,57	84,93	254,80
Raspberry Pi 3B+	922,35	2767,06	937,17	2811,51

Tab. 8.9: Tabuľka doby trvania predikcie jednotlivých dát pre *InceptionV3* architektúru

Pri porovnaní času potrebného na otestovanie dát pre konfiguráciu s najvyššou úspešnosťou na platforme *MacBook Pro* a dátami z tabuľky 8.9, pre *InceptionV3* model je doba spracovania kratšia. Konkrétne pre *Sequential* model s počiatočným nastavením - rozmery vstupných dát **200x200 px**, *batch size* **32** a *learning rate* **0,0001** to je *143,42 ms*, oproti *101,92 ms* pre *InceptionV3*. Rozdiel približne *43%*. Ale v prípade, že sa porovná model s druhou najvyššou úspešnosťou, ktorá je len o *0,14%* nižšia, tak je doba spracovania o vyše polovicu nižšia, ako pri *InceptionV3*.

Posledným, a pre potreby tejto práce najzaujímavejším porovnaním je čas potrebný na testovanie jednotlivých konfigurácií *VGG16* modelu pri použití *Intel Movidius NCS*. Keďže iba túto architektúru bolo možné implementovať na Movidius, porovnanie je len

pre *VGG16* medzi dvoma platformami. Tabuľka 8.10 odpovedá nameraným hodnotám na HW *HP Elite Book s použitím Intel Movidius* a tabuľka 8.11 *Raspberry Pi 3B+ s použitím Intel Movidius.*

HP Elite Book + Intel Movidius						
			Graph súbory modelu VGG16 architektúry			
Počiatočné nastavenia modelov			Bez predspracovávania vstupných dát		S predspracovávaním vstupných dát	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Priemerný čas [ms]	Celkový čas [s]	Priemerný čas [ms]	Celkový čas [s]
100x100	16	0.001	3,13	9,38	9,14	27,41
100x100	16	0.0001	3,11	9,33	8,88	26,64
100x100	32	0.001	3,11	9,34	9,15	27,45
100x100	32	0.0001	3,07	9,21	8,61	25,82
200x200	16	0.001	4,41	13,24	10,55	31,64
200x200	16	0.0001	4,13	12,38	13,04	39,11
200x200	32	0.001	4,02	12,05	9,57	28,72
200x200	32	0.0001	4,03	12,09	9,69	29,06

Tab. 8.10: Tabuľka doby trvania predikcie jednotlivých dát na *HP Elite Book s použitím Intel Movidius*

Raspberry Pi 3B+ + Intel Movidius						
			Graph súbory modelu VGG16 architektúry			
Počiatočné nastavenia modelov			Bez predspracovávania vstupných dát		S predspracovávaním vstupných dát	
Rozmery vstupných dát [px]	batch size [-]	learning rate [-]	Priemerný čas [ms]	Celkový čas [s]	Priemerný čas [ms]	Celkový čas [s]
100x100	16	0.001	7,70	23,10	35,16	105,47
100x100	16	0.0001	7,49	22,48	34,78	104,35
100x100	32	0.001	7,56	22,68	34,98	104,94
100x100	32	0.0001	7,67	23,02	35,21	105,62
200x200	16	0.001	15,51	46,52	47,64	142,92
200x200	16	0.0001	15,56	46,69	47,59	142,77
200x200	32	0.001	15,50	46,50	47,67	143,01
200x200	32	0.0001	15,46	46,39	47,39	142,16

Tab. 8.11: Tabuľka doby trvania predikcie jednotlivých dát na *Raspberry Pi 3B+ s použitím Intel Movidius*

Testovanie dát pri nastavení modelu s rozmerom vstupných dát **200x200 px** trvalo na *MacBook Pro* pre model *VGG16* približne *3-krát* dlhšie ako nastavenie s rozmerami **100x100 px**. Pre *HP Elite Book* a *Raspberry Pi* bol rozdiel ešte o niečo väčší. Použitím Movidius-u sa však rozdiely výrazne znížili. Pre *Raspberri Pi* sa doba testovania pre väčší rozmer vstupných dát nepredlžuje *4-krát*, ako tomu bolo bez použitia Movidius-u, ale iba *2-krát*. Pri zapojení Movidius-u do *HP Elite Book* je toto porovnanie ešte oveľa výraznejšie. Nárast doby testovania bol iba približne *30%*, čo je *13-krát* menej ako bez použitia Movidius-u.

Pri celkovom porovnaní doby testovania medzi odpovedajúcimi si platformami môžeme vidieť veľmi výraznú časovú úsporu pri použití Movidius-u. Porovnajme dve rôzne nastavenia *VGG16* modelu. Prvé, označme ho ako *číslo 1*, je nastavenie pre počiatočný

rozmer dát **100x100 px**, *batch size* **32** a *learning rate* **0,001**. Toto nastavenie vykazovalo najvyššiu úspešnosť spomedzi všetkých ostatných nastavení o rozmere dát **100x100 px** a to konkrétne *99,23%*. Druhé nastavenie, *číslo 2*, je zasa najúspešnejšou konfiguráciou, medzi rozmerami vstupných dát **200x200 px**. *Batch size* je **16** a *learning rate* **0,001** s testovacou úspešnosťou *99,47%*.

Pre *HP Elite Book* a nastavenie *číslo 1*, je vyhodnotenie jedného obrázku približne *76-krát* rýchlejšie ako bez použitia Movidius-u. Pôvodná doba trvania predikcie bola *236,31 ms*, za použitia akcelerátoru to bolo však iba *3,11 ms*. Pri nastavení *číslo 2* je táto úspora ešte razantnejšia, a to približne *208-krát*. Pôvodná doba bola *916,41 ms*, za použitia akcelerátoru *4,41 ms*. Pri *Raspberry Pi* sú tieto rozdiely ešte výraznejšie. Pri menších rozmeroch vstupných dát - nastavení *číslo 1*, sa jeden obrázok klasifikuje za *7,56 ms* pri použití akcelerátoru, čo je približne *402-krát* kratšia doba spracovania oproti testovaniu bez Movidius-u, kde sa jeden obrázok vyhodnotil za *3042,55 ms*. Pri nastavení *číslo 2*, predikcia jedného obrázku trvá približne *15,51 ms* s použitím Movidius-u. To je oproti dobe vyhodnotenia *11197,33 ms* takmer *721-krát* väčšia časová úspora. V čom je ale Movidius o niečo pomalší, sú všetky ostatné operácie, potrebné pre klasifikáciu dát. Pomerne časovo náročné je iteratívne vytváranie komunikácie a prenosu dát cez USB zbernicu z hlavného HW do Movidius-u. Tieto porovnania je možné vidieť v oboch vyššie spomenutých tabuľkách. Pre *HP Elite Book* je čas potrebný na predspracovanie a poslanie dát do Movidius-u takmer *3-krát* vyšší pre nastavenie *číslo 1*, pre nastavenie *číslo 2* to je *2,4-krát* vyššia doba. Pre *Raspberry Pi* a nastavenie *číslo 1* to je *4,6-krát* vyššia doba vyhodnotenia, pre nastavenie *číslo 2* je to o niečo menší pomer - *3,1-krát* dlhšia doba vyhodnocovania. Ale aj pri tomto zohľadnení je doba, za ktorú *Raspberry Pi s použitím Movidius-u* predikuje jeden testovací obrázok pre nastavenie *číslo 1 87-krát*, pre nastavenie *číslo 2* až *235-krát* nižšia.

Záver

Cieľom tejto bakalárskej práce bolo navrhnúť model neurónovej siete, ktorý by bol vhodný na klasifikáciu vstupných dát do šiestich kategórií. Model následne skompilovať pomocou *NCSDK* a *mvnc* API, a na predikciu výsledkov použiť výpočetnú jednotku - *Intel Movidius Neural Compute Stick*. Po úspešnej implementácii skompilovaného modelu, ďalej porovnať výpočetnú náročnosť na rôznych platformách *s* a *bez* použitia Intel Movidius-u.

Pre túto prácu boli vytvorené celkovo *2* modely konvolučných neurónových sietí. Každý z modelov bol natrénovaný *8-krát* na 8 rôznych nastavení. Parametre, ktoré boli menené a porovnávané medzi jednotlivými konfiguráciami sú - rozmery vstupných dát, batch size a počiatočná learning rate. Všetky nastavenia vykazovali po natrénovaní úspešnosť na testovacích dátach okolo *99%*. V kapitole 8.2 sú znázornené tabuľky, obsahujúce jednotlivé úspešnosti každej konfigurácie. Meranie doby predikcie dát bolo uskutočnené na *piatich* rôznych zariadeniach. Dva z nich obsahovali Intel Movidius ako akcelerátor pre predikciu neurónovej siete. Okrem celkovej doby a doby jednej predikcie, bol zaznamenávaný aj čas, potrebný na predspracovanie jednotlivých dát. Pri porovnávaní doby spracovávania na platformách, do ktorých nebol zapojený Intel Movidius nám vyšlo, že najpomalším HW je *Raspberry Pi*. *MacBook Pro* a *HP Elite Book* mali pomerne porovnateľné výsledky, čo sa týka doby predikcie dát. Pri meraniach, uskutočnených na platformách s pripojeným Intel Movidius-om, bola predikcia dát uskutočnená *76* až 208-krát rýchlejšie pre *HP Elite Book*. Pre *Raspberry Pi* s pripojeným Movidius-om boli tieto hodnoty medzi *402* až *721-krát* nižšie. V týchto prípadoch však pomerne výraznú úlohu zohrávajú *Input/Output* operácie medzi Intel Movidius-om a HW, ku ktorému je pripojený, ktoré samotný proces spomaľujú. Pri *Raspberry Pi* sme sa aj napriek tomu dostali na kratšiu dobu spracovania ako na oveľa výkonnejších notebook-och, spomenutých vyššie. Oba vytvorené modely v rámci tejto práce vykazovali vyššiu úspešnosť na testovacích dátach, ako dodaný model *InceptionV3*.

Použitie Intel Movidius-u prekázateľne zrýchlilo a zefektívnilo proces predikcie dát neurónovými sieťami. V kombinácii s *Raspberry Pi* to môže byť zaujímavá voľba pre užívateľov, ktorý majú buď obmedzený finančný rozpočet, alebo potrebujú zabrať čo najmenej miesta týmto HW. Pripojiť Movidius k notebook-u už nedáva až taký veľký zmysel, z dvoch dôvodov. Prvým je, že inštalácia všetkých potrebných balíčkov je celkom obmedzená na určité typy operačných systémov. Druhým dôvodom, ktorý nadväzuje na ten prvý je, že v tejto dobe je pohodlnejšie si zaplatiť VM na *cloud*-e, kde by sa výpočty spracovávali. Tým sa vyriešia aj problémy s nekompatibilitou.

Literatúra

[1] SMITH, Ch., University of Washington: *The History of Artificial Intelligence* [online]. Posledná aktualizácia 12. 2006 [cit. 18. 05. 2020]. Dostupné z URL: <https://courses.cs.washington.edu/courses/csep590/06au/projects/history-ai.pdf>.

[2] Full Scale: *Machine Learning and computer vision* [online]. Posledná aktualizácia 08. 05. 2019 [cit. 11. 04. 2020]. Dostupné z URL: <https://www.fullscale.io/machine-learning-computer-vision/>.

[3] ASHISH, P.: *Machine Learning Algorithm Overview* [online]. Posledná aktualizácia 21. 07. 2018 [cit. 30. 04. 2020]. Dostupné z URL: <https://medium.com/ml-research-lab/machine-learning-algorithm-overview-5816a2e6303>.

[4] HEIDENREICH, H.: *What are the types of machine learning* [online]. Posledná aktualizácia 04. 12. 2018 [cit. 11. 04. 2020]. Dostupné z URL: <https://www.towardsdatascience.com/what-are-the-types-of-machine-learning-e2b9e5d1756f>.

[5] The MathWorks, Inc.: *Unsupervised learning* [online]. [cit. 11. 04. 2020]. Dostupné z URL: <https://www.mathworks.com/discovery/unsupervised-learning.html/>.

[6] OSINSKI, B., KONRAD, B.: *What is reinforcement learning* [online]. Posledná aktualizácia 08. 07. 2018 [cit. 11. 04. 2020]. Dostupné z URL: <https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/>.

[7] NIGAM, V.: *Understanding Neural Networks. From neuron to RNN, CNN, and Deep Learning* [online]. Posledná aktualizácia 11. 09. 2018 [cit. 30. 04. 2020]. Dostupné z URL: <https://towardsdatascience.com/understanding-neural-networks-from-neuron-to-rnn-cnn-and-deep-learning-cd88e90e0a90>.

[8] BROWNLEE, J.: *What is Computer Vision* [online]. Posledná aktualizácia 05. 07. 2019 [cit. 12. 04. 2020]. Dostupné z URL: <https://machinelearningmastery.com/what-is-computer-vision/>.

[9] SOLEM, J., E.: *Programming Computer Vision with Python: Tools And Algorithms For Analyzing Images.* Unites States of America: O'Reilly Media, Inc., 1. vyd., 2012, s. 9, IBAN-13: 978-14493165549.

[10] Exxact Corporation: *Applications of Computer Vision For Deep Learning* [online]. Posledná aktualizácia 30. 07. 2019 [cit. 12. 04. 2020]. Dostupné z URL:

<https://blog.exxactcorp.com/applications-of-computer-vision-for-deep-learning/>.

[11] BROWNLEE, J.: *A Gentle Introduction to Object Recognition With Deep Learning* [online]. Posledná aktualizácia 22. 06. 2019 [cit. 13. 04. 2020]. Dostupné z URL: <https://machinelearningmastery.com/object-recognition-with-deep-learning/>.

[12] *Part2: Fast R-CNN (Object Detection)* [online]. Posledná aktualizácia 18. 07. 2019 [cit. 13. 04. 2020]. Dostupné z URL: <https://mc.ai/part-2-fast-r-cnn-object-detection/>.

[13] ELLIOTT, T.: *The State Of the Octoverse Machine Learning* [online]. Posledná aktualizácia 24. 01. 2019 [cit. 13. 04. 2020]. Dostupné z URL: <https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/>.

[14] Python Software Foundation: *What Is Python* [online]. [cit. 13. 04. 2020]. Dostupné z URL: <https://www.python.org/doc/essays/blurb/>.

[15] Keras.io: *Keras: The Python Deep Learning Library* [online]. [cit. 13. 04. 2020]. Dostupné z URL: <https://keras.io/#support>.

[16] Codete.com: *Mechine Learning Libraries Overview: Top 10 Libraries and Frameworks You Should Know* [online]. Posledná aktualizácia 20. 08. 2019 [cit. 01. 05. 2020]. Dostupné z URL: <https://codete.com/blog/machine-learning-libraries-overview-top-10-libraries-and-frameworks-you-should-know/>.

[17] Official TensorFlow documentation: *Keras: The Python Deep Learning Library* [online]. Posledná aktualizácia 02. 04. 2020 [cit. 15. 04. 2020]. Dostupné z URL: <https://www.tensorflow.org/probability/api_docs/python/tfp/math/ode/Solver>.

[18] UNRUH, A.: *Estimators - An Easy Way To Work With Tensorflow* [online]. Posledná aktualizácia 09. 11. 2017 [cit. 15. 04. 2020]. Dostupné z URL: <https://opensource.com/article/17/11/intro-tensorflow>.

[19] SODIMANA, K.: *What Is the TensorFlow Machine Intelligence Platform?* [online]. Posledná aktualizácia 31. 08. 2018 [cit. 19. 04. 2020]. Dostupné z URL: <http://keshans.com/Tensorflow-estimators/>.

[20] NVIDIA Corporation: *CUDA Overview* [online]. [cit. 19. 04. 2020]. Dostupné z URL: <https://developer.nvidia.com/cuda-zone>.

[21] BAHRAMPOUR, S., RAMAKRISHNAN, N., SCHOTT, L., SHAH, M.: *Comparative Study of Caffe, Neon, Theano And Torch For Deep Learning* [online]. Posledná aktualizácia 17. 02. 2016 [cit. 19. 04. 2020]. Dostupné z URL: <https://openreview.net/pdf?id=q7kEN7WoXU8LEkD3t7BQ>.

[22] OmniSci Inc.: *Hardware Acceleration* [online]. [cit. 19. 04. 2020]. Dostupné z URL: <https://www.omnisci.com/technical-glossary/hardware-acceleration>.

[23] Intel Corporation: *Intel Movidius Neural Compute Stick* [online]. Posledná aktualizácia 22. 08. 2018 [cit. 20. 04. 2020]. Dostupné z URL: <https://software.intel.com/en-us/articles/intel-movidius-neural-compute-stick>.

[24] Janakiram MSV: *A Closer Look at Intel Movidius Neural Compute Stick* [online]. Posledná aktualizácia 08. 03. 2019 [cit. 20. 04. 2020]. Dostupné z URL: <https://thenewstack.io/a-closer-look-at-intel-movidius-neural-compute-stick/>.

[25] Github: *Software Development Kit for the Neural Compute Stick* [online]. Posledná aktualizácia 15. 08. 2019 [cit. 21. 04. 2020]. Dostupné z URL: <https://github.com/movidius/ncsdk>.

[26] Intel Corporation: *Neural Compute Stick Documentation* [online]. [cit. 21. 04. 2020]. Dostupné z URL: <https://movidius.github.io/ncsdk/>.

[27] Google Cloud: *Official Google Cloud documentation* [online]. [cit. 01. 05. 2020]. Dostupné z URL: <https://cloud.google.com/ai-platform/training/docs/using-gpus>.

[28] Raspberry Pi Foundation: *Raspbrry Pi 4 Model B Technical Specifications* [online]. [cit. 21. 04. 2020]. Dostupné z URL: <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/>.

[29] VIJAYAKUMAR, A.: *Run NCS Applications on Raspberry Pi* [online]. Posledná aktualizácia 25. 10. 2017 [cit. 21. 04. 2020]. Dostupné z URL: <https://movidius.github.io/blog/ncs-apps-on-rpi/>.

[30] Tyco Electronics Czech s.r.o.: *Zameranie spoločnosti TE Kuřim* [online]. [cit. 22. 04. 2020]. Dostupné z URL: <https://te.jobs.cz/kurim/kontakty/>.

[31] *BMP file format* [online]. [cit. 22. 04. 2020]. Dostupné z URL: <https://techterms.com/definition/bmp>.

[32] RUIZENDAAL, R.: *More on CNNs And Handling Overfitting* [online]. Posledná aktualizácia 12. 05. 2017 [cit. 22. 04. 2020]. Dostupné z URL: <https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d>.

[33] Scikit-learn official documentation: *Sklearn model_selection train_test_split* [online]. [cit. 22. 04. 2020]. Dostupné z URL: <https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html>.

[34] Intel Corporation, NCSDK Documentation: *Installation and Configuration with a Virtual Machine* [online]. [cit. 04. 05. 2020]. Dostupné z URL: <https://movidius.github.io/ncsdk/vm_config.html>.

[35] Intel Corporation, NCSDK Documentation: *Installation and Configuration with virtualenv* [online]. [cit. 04. 05. 2020]. Dostupné z URL: <https://movidius.github.io/ncsdk/virtualenv.html>.

[36] Intel Corporation, NCSDK Documentation: *Installation and Configuration with Docker* [online]. [cit. 04. 05. 2020]. Dostupné z URL: <https://movidius.github.io/ncsdk/docker.html>.

[37] HowChoo: *Download Raspbian Stretch* [online]. Posledná aktualizácia 09. 09. 2019 [cit. 03. 05. 2020]. Dostupné z URL: <https://howchoo.com/g/nzc0yjzjy2u/raspbian-stretch-download>.

[38] Dask project on GitHub: *Dask project on GitHub* [online]. [cit. 03. 05. 2020]. Dostupné z URL: <https://github.com/dask/dask/archive/2.0.0.zip>.

[39] Caffe: *Caffe Installation* [online]. [cit. 03. 05. 2020]. Dostupné z URL: <https://caffe.berkeleyvision.org/installation.html>.

[40] Brigham Young University, Office of Research Computing: *I keep getting Illegal Instruction errors. What do I do?* [online]. Posledná aktualizácia 11. 07. 2011 [cit. 03. 05. 2020]. Dostupné z URL: <https://rc.byu.edu/wiki/index.php?page=I+keep+getting+Illegal+Instruction+errors.+What+do+I+do%3F>.

[41] Intel Corporation, NCSDK Documentation: *mvNCCheck* [online]. [cit. 04. 05. 2020]. Dostupné z URL: <https://movidius.github.io/ncsdk/tools/check.html>.

[42] Intel Corporation, NCSDK Documentation: *mvNCProfile* [online]. [cit. 04. 05. 2020]. Dostupné z URL: <https://movidius.github.io/ncsdk/tools/profile.html>.

[43] Intel Corporation, NCSDK Documentation: *mvNCCompile* [online]. [cit. 04. 05. 2020]. Dostupné z URL:
<https://movidius.github.io/ncsdk/tools/compile.html>.

[44] Intel Corporation, NCSDK Documentation: *Chybové kódy a ich význam pri kompilácii* [online]. [cit. 04. 05. 2020]. Dostupné z URL:
<https://movidius.github.io/ncsdk/compiler_error.html>.

[45] Numpy: *Broadcasting* [online]. [cit. 04. 05. 2020]. Dostupné z URL:
<https://numpy.org/doc/stable/user/basics.broadcasting.html?highlight=broadcasting#module-numpy.doc.broadcasting>.

[46] MILTON-BARKER, A., Intel Corporation: *Inception V3 Deep Convolutional Architecture* [online]. Posledná aktualizácia 17. 02. 2019 [cit. 04. 05. 2020]. Dostupné z URL:
<https://software.intel.com/content/www/us/en/develop/articles/inception-v3-deep-convolutional-architecture-for-classifying-acute-myeloidlymphoblastic.html>.

[47] HASSAN, M.,: *VGG16 – Convolutional Network for Classification and Detection* [online]. Posledná aktualizácia 20. 11. 2018 [cit. 04. 05. 2020]. Dostupné z URL:
<https://neurohive.io/en/popular-networks/vgg16/>.

[48] CHOLLET, F., Keras Official Documentation: *Complete guide to the Sequential model.* [online]. Posledná aktualizácia 12. 04. 2020 [cit. 14. 05. 2020]. Dostupné z URL:
<https://keras.io/guides/sequential_model/>.

[49] Keras Official Documentation: *Conv2D layer* [online]. [cit. 14. 05. 2020]. Dostupné z URL:
<https://keras.io/api/layers/convolution_layers/convolution2d/>.

[50] ANH, V.: *Deep Learning – Computer Vision and Convolutional Neural Networks* [online]. Posledná aktualizácia 02. 01. 2018 [cit. 14. 05. 2020]. Dostupné z URL:
<https://anhvnn.wordpress.com/2018/02/01/deep-learning-computer-vision-and-convolutional-neural-networks/>.

[51] Keras Official Documentation: *BatchNormalization class* [online]. [cit. 14. 05. 2020]. Dostupné z URL:
<https://keras.io/api/layers/normalization_layers/batch_normalization/>.

[52] Keras Official Documentation: *MaxPooling2D class* [online]. [cit. 14. 05. 2020]. Dostupné z URL:
<https://keras.io/api/layers/pooling_layers/max_pooling2d/>.

[53] Adventuresinmachinelearning.com: *Convolutional Neural Networks Tutorial in TensorFlow* [online]. [cit. 14. 05. 2020]. Dostupné z URL:

<http://adventuresinmachinelearning.com/convolutional-neural-networks-tutorial-tensorflow/>.

[54] Keras Official Documentation: *Layer activation functions* [online]. [cit. 14. 05. 2020]. Dostupné z URL: <https://keras.io/api/layers/activations/>.

[55] MAKLIN, C.: *Dropout Neural Network Layer In Keras Explained.* [online]. Posledná aktualizácia 03. 06. 2019 [cit. 14. 05. 2020]. Dostupné z URL: <https://towardsdatascience.com/machine-learning-part-20dropout-keras-layers-explained-8c9f6dc4c9ab>.

[56] Keras Official Documentation: *Flatten class.* [online]. [cit. 14. 05. 2020]. Dostupné z URL: <https://keras.io/api/layers/reshaping_layers/flatten/>.

[57] Keras Official Documentation: *Dense class* [online]. [cit. 14. 05. 2020]. Dostupné z URL: <https://keras.io/api/layers/core_layers/dense/>.

[58] DAITYARI, S.: *Dense Layer Example* [online]. Posledná aktualizácia 18. 10. 2019 [cit. 14. 05. 2020]. Dostupné z URL: <https://www.sitepoint.com/keras-digit-recognition-tutorial/>.

[59] YANG, J.: *ReLU and Softmax Activation Functions* [online]. Posledná aktualizácia 11. 2. 2017 [cit. 14. 05. 2020]. Dostupné z URL: <https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions>.

[60] ENDRYCH, David. *Analýza rozložení stran textových dokumentů pomocí hlubokých neuronových sítí.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Oldřich Kodym [cit. 14. 05. 2020].

[61] BROWNLEE, J.: *Understand the Impact of Learning Rate on Neural Network Performance* [online]. Posledná aktualizácia 25. 01. 2019 [cit. 14. 05. 2020]. Dostupné z URL: <https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/>.

[62] Intel Corporation, Official Intel Forum: *mvNCCheck error* [online]. Posledná aktualizácia 03. 04. 2019 [cit. 18. 05. 2020]. Dostupné z URL: <https://forums.intel.com/s/question/0D50P00004NM0hdSAD/mvnccheck-error?language=en_US&fbclid=IwAR0GuXVaFrVRdpBMEuxSMNQhdSuG4iIbiL39wO%2DR0F2v-uzSZDVW8s43_kTk>.

[63] Intel Corporation, NCSDK Documentation: *Intel Movidius Neural Compute SDK Python API v2* [online]. [cit. 18. 05. 2020]. Dostupné z URL: <https://movidius.github.io/ncsdk/ncapi/ncapi2/py_api/readme.html>.

[64] The Python Software Foundation, Official Documentation: *Time access and conversions* [online]. Posledná aktualizácia 01. 06. 2020 [cit. 02. 06. 2020]. Dostupné z URL: <https://docs.python.org/3/library/time.html>.

Zoznam symbolov, veličín a skratiek

ML	Strojové učenie – Machine Learning
AI	Umelá inteligencia – Artificial Intelligence
CV	Počítačové videnie – Computer Vision
NN	Neurónové siete – Neural Networks
CNN	Convolučné neurónové siete – Convolutional Neural Networks
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
ResNet	Residual Neural Network
MS COCO	Microsoft Common Objects in Context
R-CNN	Region-Based Convolutional Neural Network
YOLO	You Only Look Once
RNN	Rekurentné neurónové siete – Recurrent Neural Network
API	Application Programming Interface
CPU	centrálna procesorová jednotka – Central Processing Unit
GPU	grafická procesorová jednotka – Graphical Processing Unit
ONEIROS	Open-ended Neuro-Electronic Intelligent Robot Operating System
ASIC	Zákaznícky integrovaný obvod – Application-Specific Integrated Circuit
FPGA	programovatelné hradlové pole – Field Programmable Gate Array
VPU	Vision Processing Unit
NCS	Movidius Neural Compute Stick
SHAVE	Streaming Hybrid Architecture Vector Engine
NCSDK	Neural Compute SDK
BMP	bitmap image file
VM	Virtual Machine