UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Aplikace pro smart detekci výrobních vad

Bc. Ondřej Beneš

Diplomová práce
2018

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2017/2018**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Ondřej Beneš**
Osobní číslo:	**I15195**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Aplikace pro smart detekci výrobních vad**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Postup: Cílem práce je vytvoření software pro automatickou detekci vadných výrobků založeného na technologii konvolučních sítí. Software bude umožňovat efektivní import velkého množství vizuálních dat pro vytvoření trénovací množiny, možnost hromadné úpravy dat ve smyslu oříznutí, normalizace kontrastu a identifikace klíčových částí obrázku, možnost trénování konvoluční sítě včetně vizualizace průběhu a možnost exportu výsledné sítě jako samostatné aplikace s jednoznačným rozhraním. Teoretická část: Stručná rešerše problematiky konvolučních sítí, vývoj konvolučních sítí, rešerše nejznámějších použití. Praktická část: Tvorba aplikace, ověření aplikace, testování různých scénářů, statistické vyhodnocení, uživatelská příručka k aplikaci, diskuse.

Rozsah grafických prací: **10**

Rozsah pracovní zprávy: **60**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

BUDUMA, N. Fundamentals of Deep Learning. Sebastopol, CA : O?Reilly Media, Inc, 2017. 304 s. ISBN 978-1491925614
HAYKIN, S. Neural Networks. New Jersey : Prentice Hall, 1999. 845 s. ISBN 0-13-273350-1

Vedoucí diplomové práce: **doc. Ing. Petr Doležel, Ph.D.**
Katedra řízení procesů

Datum zadání diplomové práce: **30. října 2017**

Termín odevzdání diplomové práce: **18. května 2018**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2017

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 10. 5. 2018.

Ondřej Beneš

Poděkování

Mé díky patří vedoucímu práce, doc. Petru Doleželovi, za jeho vždy přátelský a profesionální přístup. Děkuji také rodině a blízkým za podporu během studia, obzvlášť v době, kdy jsem tvořil tuto práci. Velice si vážím jejich zpětné vazby.

Anotace

Práce se po teoretické i praktické stránce zabývá využitím metod zpracování obrazu a umělé inteligence při vývoji aplikací pro detekci výrobních vad. Textová část práce se zaměřuje zejména na implementaci aplikací, které byly vyvinuty jako součást práce. Tyto aplikace vznikly v rámci společného projektu Univerzity Pardubice a firmy ze soukromé sféry. Očekává se, že budou dále prakticky využívány.

Klíčovou roli v aplikacích hrají konvoluční neuronové sítě. Tento typ umělých neuronových sítí, využívaný zejména pro klasifikaci obrázků, roste v poslední době v oblibě. A to nejen v akademickém prostředí, ale také v praxi. Je úspěšně využíván IT giganty jako Google nebo Facebook.

Ačkoliv vyvinuté aplikace slouží primárně pro detekci výrobních vad konkrétního dílu, jsou dostatečně univerzální na to, aby byly použity pro klasifikaci libovolných obrázků za předpokladu, že uživatel poskytne dostatečně velkou množinu dat pro učení sítě.

Klíčová slova

Umělá inteligence, konvoluční neuronová síť, strojové učení, zpracování obrazu

Title

Smart Detection of Manufacturing Defects

Annotation

This thesis covers the use of computer vision and artificial intelligence methods in development of software used for detection of manufacturing defects. The printed part of the thesis is mostly focused on the description of the implementation of applications that comprise the practical part of the thesis. These applications were developed as part of a project with a privately owned company and are expected to be used in production.

A key role in the applications is played by convolutional neural networks. This kind of artificial neural networks, used mostly for image classification, has recently been growing in popularity, both in academic and practical fields. It is being used by IT giants such as Google or Facebook.

Even though the applications are primarily used for detecting defects of a specific part, they are univarsal enough to be used for classification of any images, provided that the user supplies a large enough dataset to train the network.

Key words

Artificial intelligence, convolutional neural network, machine learning, computer vision

Obsah

Úvod **11**
Cíle práce 11
Popis kapitol 11
Použité konvence 12
1 Úvod do umělých neuronových sítí **13**
1.1 Umělý neuron 13
1.2 Neuronová síť 13
1.3 Konvoluční neuronová síť 14
1.3.1 Historie 15
1.3.2 Princip 15
2 Použité sw technologie **19**
2.1 Python 19
2.2 OpenCV 19
2.3 Tensorflow 20
2.4 Další technologie 26
2.4.1 Kivy 26
2.4.2 NumPy 26
3 Metodika vývoje **27**
3.1 První prototyp 27
3.2 Druhý prototyp 27
3.3 Třetí prototyp 28
3.4 Čtvrtý prototyp 28
3.5 Finální verze 29
3.6 Programy používané během vývoje 29
4 Předzpracování obrázových dat **30**
4.1 Popis fotografií 30
4.2 První fáze – příprava obrázku 32
4.3 Druhá fáze - detekce svarů 32
4.3.1 Před prahováním 33
4.3.2 Výpočet souřadnic 33
4.3.3 Optimalizace 38
4.4 Třetí fáze – finální úpravy 40
4.5 Dávkové zpracování 41
4.6 Logování 42
4.7 Vizualizace 43
4.8 Shrnutí 44
5 Implementace konvoluční sítě **45**
5.1 Inicializace 45
5.2 Definice architektury 45
5.2.1 Výchozí architektura 46
5.2.2 Vlastní architektura 46
5.2.3 Získání informací o vrstvách 47

5.3 Učení ... 47
5.3.1 Proměnné učení ... 47
5.3.2 DataProvider ... 48
5.3.3 Offline učení ... 49
5.3.4 Online učení ... 50
5.4 Klasifikace ... 50
5.5 Ukládání a načítání ... 51
6 Grafické uživatelské rozhraní ... 52
6.1 Struktura ... 52
6.1.1 Definice sítě ... 52
6.1.2 Učení sítě ... 53
6.1.3 Použití sítě ... 54
7 Konzolová aplikace ... 55
7.1 Implementace ... 55
7.2 Použití ... 56
8 Testování ... 57
8.1 Unit testy ... 57
8.2 Testovací scénáře ... 57
9 Výběr vhodné architektury ... 58
9.1 Vyhodnocované architektury ... 58
Závěr ... 61
Literatura ... 63
Příloha A: Vysokoúrovňový pohled na předzpracování (diagram) ... I
Příloha B: Diagram procesů Příprava obrázku a Finální úpravy ... II
Příloha C: Diagram procesu Detekce svárů ... III
Příloha D: UML diagram tříd ... IV

Seznam obrázků

Obrázek 1 - Jednoduchá dopředná neuronová síť 13
Obrázek 2 - Průchod obrázku konvoluční sítí 17
Obrázek 3 - Vizualizace výpočetního grafu pomocí Tensorflow 19
Obrázek 4 - Vizualizace funkce Rastrigin pro n=2 24
Obrázek 5 - Aproximace funkce Rastrigin pomocí neuronové sítě 24
Obrázek 6 - Typický vzhled GUI v Kivy a TkInter 25
Obrázek 7 - Tooltip 28
Obrázek 8 - Předzpracovaný obrázek 29
Obrázek 9 - Originální fotografie dílu 30
Obrázek 10 - Fotografie dílu s vyznačenými tvary a svary 30
Obrázek 11 - Metody prahování 33
Obrázek 12 - Obrysy všech detekovaných tvarů pro jeden z prahů 33
Obrázek 13 - Výřez okolo středu svaru 39
Obrázek 14 - Vliv počtu vláken na rychlost předzpracování 40
Obrázek 15 - Příklad vizualizace 1 42
Obrázek 16 - Příklad vizualizace 2 43
Obrázek 17 - Záložka Definice sítě 51
Obrázek 18 - Dialogové okno pro vytvoření sítě 52
Obrázek 19 - Záložka Učení sítě 52
Obrázek 20 - Záložka Použití sítě 53
Obrázek 21 - Konzolová aplikace 54

Seznam tabulek

Tabulka 1 - Příklady nelineárních funkci 15
Tabulka 2 - Funkce pro výpočet souřadnic svarů 35
Tabulka 3 - Chyby detekce před optimalizací 38
Tabulka 4 - Chyby detekce po optimalizaci 38
Tabulka 5 - Testovací scénáře 56
Tabulka 6 - Architektury sítí s nejlepšími výsledky testování 58
Tabulka 7 - Výsledky testování dvou nejlepších sítí 59

Seznam zkratek a značek

API	Application Programming Interface	Aplikační programovací rozhraní
CNN	Convolutional Neural Network	Konvoluční neuronová síť
ELU	Exponential Linear Unit	Exponenciální lineární jednotka
IDE	Integrated Development Environment	Integrované vývojové prostředí
GUI	Graphical User Interface	Grafické uživatelské rozhraní
ReLU	Rectified Linear Unit	Usměrněná lineární jednotka

Úvod

Cíle práce

Hlavním cílem práce bylo vytvoření aplikace s grafickým uživatelským rozhraním (GUI), která uživateli umožní definovat konvoluční neuronovou síť (CNN), provést učení této sítě a následně ji využívat pro klasifikaci obrazových dat. Předpokládá se, že klasifikovány budou fotografie svařovaných dílů, a že bude rozhodováno, zda jsou svary v pořádku nebo zda jsou vadné. Součástí zadání práce byl datový set, obsahující cca 30 000 fotografií svařovaných dílů. Tento set byl rozdělen na kategorie OK a Vadný a dále do skupin podle dne, kdy byla fotografie pořízena.

Souvisejícím cílem bylo vytvoření doprovodné konzolové aplikace, která slouží výhradně pro klasifikaci obrázků. Obrázky mohou být klasifikovány jednotlivě nebo v dávkách. Aplikace využívá CNN definované a vytrénované v GUI aplikaci. Zvyšuje efektivitu práce tím, že klasifikace může probíhat na pozadí a výsledky mohou být snadno uloženy např. do souboru.

Dílčím cílem práce bylo vytvoření algoritmu, který na fotografii dílu detekuje svary, provede okolo nich výřezy a tyto výřezy spojí do jednoho obrázku. Tím budou získána kvalitní data pro učení sítě. Umožní to také využívat aplikace ke klasifikaci fotografií, které v době vytváření práce ještě nebyly pořízeny nebo autorovi nebyly poskytnuty. Algoritmus musí být dostatečně robustní, protože pozice svarů na fotografiích se měnila, zejména v závislosti na tom, ze které podmnožiny datového setu fotografie pocházela.

Výše zmíněné cíle tvoří praktickou část práce. Cílem teoretické části bylo čtenáře stručně uvést do problematiky umělých neuronových sítí. Důraz je kladen na konvoluční sítě, konkrétně jejich historii, princip fungování a popis možného použití.

Popis kapitol

První kapitola představuje teoretickou část práce. Je zde provedena rešerše problematiky umělých neuronů a umělých neuronových sítí (zejména konvolučních).

V druhé kapitole jsou popsány softwarové technologie použité při implementaci. Ve stručnosti je představen programovací jazyk Python. Jsou zmíněny výhody a nevýhody plynoucí z použití tohoto jazyka. Představena je také softwarová knihovna OpenCV, kterou využívá zejména algoritmus pro detekci svarů. Větší pozornost je věnována knihovně Tensorflow. Tato knihovna je využívána pro implementaci CNN. Použití Tensorflow je demonstrováno na příkladech. Zmíněno je také využití knihoven Kivy a NumPy.

Třetí kapitola se zabývá použitou metodikou vývoje aplikací. Je zde popsáno, jak byly aplikace iterovány přes několik prototypů až po finální verze. Stručně jsou také zmíněny programy používané během vývoje.

Ve čtvrté kapitole je popsán algoritmus pro předzpracování obrazových dat. Pozornost je věnována popisu datové množiny. Následně je algoritmus popsán z vysokoúrovňového pohledu (příprava obrázku a operace prováděné po detekci svarů) a z nízkoúrovňového pohledu (samotná detekce svarů). Zmíněno je také to, jak probíhalo ladění a optimalizace algoritmu.

Pátá kapitola je věnována popisu implementace CNN. Je popsána Python třída `CNN`, včetně metod pro inicializaci, definici architektury, učení, klasifikaci obrázku, ukládání a načítání. Zmíněny jsou také Python třídy představující jednotlivé vrstvy sítě a třída `DataProvider`, sloužící k poskytování dávek obrázků při učení sítě.

Šestá kapitola se zabývá tím, jak bylo implementováno GUI. Součástí kapitoly jsou obrázky ilustrující rozložení GUI elementů. V sedmé kapitole je popsána implementace konzolové aplikace. Její možné využití je demonstrováno na příkladech. Osmá kapitola je věnována testování aplikací. Jsou zmíněny jak automatizované Unit testy, tak testovací scénáře vyžadující aktivitu ze strany testera.

V poslední kapitole je popsán proces hledání vhodné architektury sítě. Jsou zmíněna omezení, která bylo nutné brát v úvahu při návrhu architektur. Také je zde uvedeno, jak se postupně měnily parametry učení a na základě jaké metriky byla vybrána síť, která slouží jako výchozí.

Použité konvence

Názvy tříd, funkcí, metod apod. jsou psány písmem `Courier New`. Ukázky kódu používají výchozí styl integrovaného vývojového prostředí (IDE) PyCharm. Klíčová slova jsou psána **tučně modře,** zabudované funkce tmavě modře, jména parametrů fialově, číselné konstanty světle modře a textové konstanty **`zeleně.`**

1 Úvod do umělých neuronových sítí

Klíčovou úlohu v aplikacích pro smart detekci výrobních vad tvoří konvoluční neuronová síť (CNN), proto je první kapitola věnována stručnému úvodu do problematiky umělých neuronových sítí, se zaměřením právě na CNN.

Problematika umělých neuronových sítí spadá pod obor umělá inteligence, konkrétně pod strojové učení. Neuronové sítě jsou výpočetní modely, které se používají mj. pro predikci, aproximaci a rozpoznávání vzorů [1].

1.1 Umělý neuron

Umělý neuron je matematický model, který představuje základní prvek umělých neuronových sítí. Inspirací pro umělé neurony jsou biologické neurony, což jsou „základní stavební funkční prvky nervové soustavy“ [2].

Do umělého neuronu vstupují signály, které jsou agregovány. Agregace je obvykle prováděna jako suma součinů vstupu a jeho váhy. K výsledku agregace je přičten práh (bias). Takto získaná hodnota je označována jako vnitřní potenciál neuronu a je použita jako argument tzv. aktivační funkce. Výsledek aktivační funkce tvoří výstup neuronu. Matematicky je výše popsané zapsané v Rovnici 1 [2]:

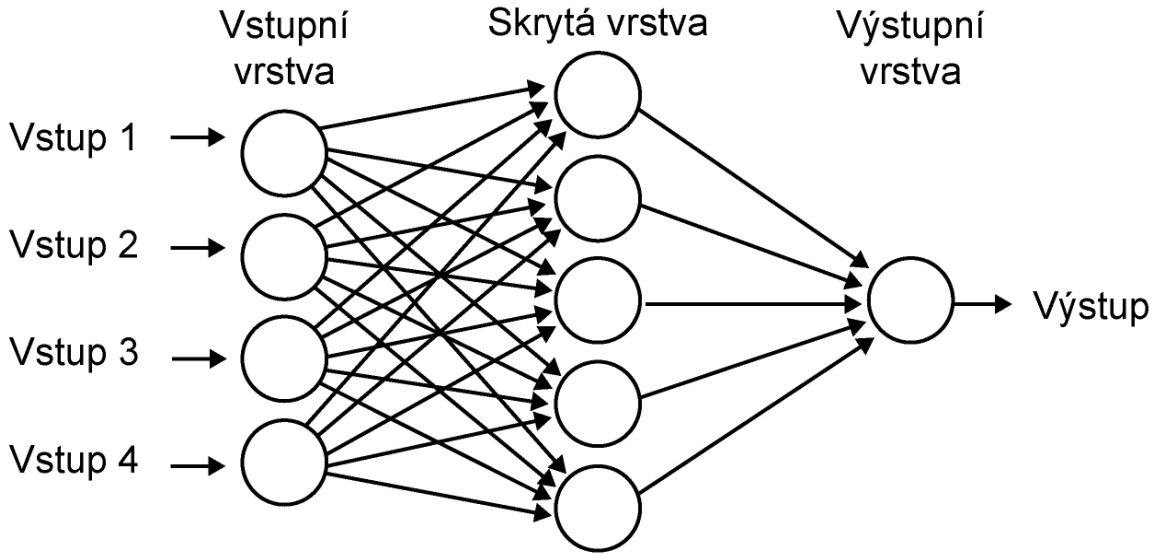
$$y = f\left(\sum_{i=1}^{n}(x_i w_i) + b\right) \tag{1}$$

kde y je výstup neuronu, f je aktivační funkce, n je počet vstupů, x jsou vstupy, w jsou váhy a b je práh. Při implementaci umělého neuronu jsou vstupy obvykle řádkové matice a váhy sloupcové matice (či naopak). Vynásobení těchto matic poté odpovídá agregační funkci (výsledek bude skalár).

1.2 Neuronová síť

Umělý neuron lze použít pro simulaci jednoduchých funkcí, jako například logické AND. Obecně se jedná o funkce, jejichž obor hodnot je možné rozdělit na dvě lineárně separovatelné množiny. V případě, kdy jeden neuron slouží jako výpočetní model, hovoříme o perceptronu [2].

Pro komplexnější úlohy je již nutné použít neuronů více, čímž vzniká neuronová síť. Výstup z jednoho neuronu tvoří jeden ze vstupů neuronů následující vrstvy. Běžným typem zapojení je vícevrstvé dopředné zapojení, ve kterém nedochází k tomu, že by neurony byly propojeny cyklicky. Dopředné neuronové sítě mají jednu vstupní vrstvu, libovolný počet skrytých (vnitřních) vrstev a jednu výstupní vrstvu (viz Obrázek 1). Počet neuronů v jednotlivých vrstvách a způsob jejich propojení se označuje pojmem architektura sítě [2].



Obrázek 1 - Jednoduchá dopředná neuronová síť
zdroj: http://bit.ly/2vVGmdo

Klíčovým procesem je učení sítě. To spočívá v úpravě vah a prahů neuronů tak, aby skutečný výstup ze sítě odpovídal očekávanému výstupu. Dostatečná přesnost výpočtu závisí na typu úlohy. Při učení s učitelem je vypočítána chyba výstupu, podle které jsou upraveny váhy (obvykle pomocí algoritmu Zpětného šíření chyby nebo jeho variace). Alternativou je učení bez učitele, ve kterém nejsou výstupy předem známé. Tento typ učení se používá např. pro klasifikaci vstupů [2].

1.3 Konvoluční neuronová síť

Konvoluční sítě jsou zvláštním typem umělých neuronových sítí, které se používají zejména pro zpracování 2D dat, tedy takových dat, pro která je relevantní nejen hodnota v jednom bodě, ale i v jeho okolí. Typickým zástupcem takových dat jsou obrázky. Důležitou vlastností CNN je schopnost provádět konvoluci mezi filtrem a částí dat. Na základě výsledku konvoluce lze říci, jak velká je podobnost filtru a analyzované části dat. Postupnou aplikací více filtrů je možné provádět detekci objektů nacházejících se na obrázku [31].

Konvoluční sítě se využívají zejména pro následující úlohy [4]:

- klasifikace obrázku – obrázku je přiřazena třída, která jej nejlépe vystihuje (např. pes, auto, obličej),
- lokalizace objektu – je provedena klasifikace obrázku a objekt specifický pro danou třídu je umístěn do ohraničujícího obdélníku,
- detekce objektu – každý detekovaný objekt na obrázku je klasifikován a je umístěn do ohraničujícího obdélníku,
- segmentace objektu – každý detekovaný objekt je klasifikován a je kolem něj vytvořen obrys.

1.3.1 Historie

Využitím neuronových sítí pro rozpoznávání znaků se zabýval už koncem 50. let 20. století Frank Rossenblatt, který spolu s týmem odborníků sestrojil počítač Mark I Perceptron. Tento neuropočítač byl schopný klasifikovat znaky na základě intenzity obrazových bodů [2].

Důležitou roli v oblasti detekce vzorů hrála umělá neuronová síť Neocognitron, kterou představil v 80. letech Kunihiko Fukushima. V této síti byly za sebou řazeny vrstvy „S buněk“ a „C buněk“. Úkolem S buněk bylo extrahovat příznaky (např. hrany). C buňky poté zaručovaly, že absolutní umístění příznaku nebude bráno v potaz [5].

Dalším milníkem byl rok 2012, kdy Alex Krizhevsky vyhrál soutěž ImageNet za použití konvoluční neuronové sítě AlexNet. V této soutěži mají soutěžící za úkol vytvořit program, který dokáže klasifikovat obrázky s co nejmenší chybou. Datová množina obsahovala přes 10 000 000 obrázků a více než 10 000 kategorií. Vítězná síť úspěšně klasifikovala 85 % obrázků [6].

Tento výsledek byl o 11 % lepší než v minulém ročníku, což byl jeden z důvodů zvýšeného zájem o konvoluční sítě. V následujících letech byla chyba dále snižována. Týmu z firmy Microsoft se podařilo vytvořit síť s chybou 4,94 %, což je lepší výsledek, než jakého je schopný dosáhnout člověk (5,1 %) [7].

Konvoluční sítě se dnes těší relativně velké oblibě, a to i u významných IT firem. Např. vyhledávání obrázků ve vyhledávači Google využívá CNN. Facebook používá CNN pro detekci obličejů na fotografiích. Velký zájem pravděpodobně také souvisí s dobře dostupným výkonným hardwarem (zejména s grafickými kartami) a s kvalitní softwarovou podporou (knihovny jako cuDNN, Caffe nebo Tensorflow). Firma Google dokonce vyrábí speciální zařízení pro strojové učení zvané Tensor Processing Unit [4][8].

1.3.2 Princip

Vstupem do CNN je obrázek ve formě matice (používá se také označení tensor). Prvky matice tvoří intenzity jasu v jednotlivých pixelech. U obrázků v odstínech šedi jednomu pixelu obvykle odpovídá jeden prvek matice, u barevných obrázků je jeden pixel reprezentován více prvky (obvykle jeden pro každý barevný kanál). Z tensorů jsou na základě filtrů extrahovány příznaky. V nižších vrstvách mohou být příznaky například horizontální nebo diagonální čáry. Ve vyšších vrstvách už jsou detekovány komplexnější tvary. Pokud by například byla síť použita pro detekci obličejů, mohl by takovým příznakem být nos, oko, nebo celá tvář.

Idea rozpoznávání jednoduchých tvarů a z nich skládání tvarů komplexnějších vychází z toho, jak funguje zrak u živých organismů. Již v 60. letech 20. století dokázali David H. Hubel a Torsten Wiesel, že to, jaké neurony se aktivují, závisí na tom, jak orientovaná hrana byla spatřena [4].

Proces aplikace filtrů na tensor se nazývá konvoluce a probíhá v konvolučních vrstvách. Filtr má dané rozměry – výška h_f a šířka w_f odpovídá rozměrům okolí, které je analyzováno, a hloubka c_f odpovídá počtu kanálů barev. Filtr je neuron, jehož váhy jsou stejně jako u jiných sítí upravovány během učení. Před zahájením jsou váhy obvykle inicializovány na malé náhodné hodnoty. Není tedy nutné (ani obvyklé) specifikovat, co má který filtr detekovat [4].

Filtr se postupně aplikuje na každý prvek tensoru a jeho okolí (které je stejně velké jako filtr). Pro každý pixel se provede suma násobků prvků filtru a okolí pixelu. Pokud je hodnota této sumy relativně vysoká, značí to, že nejspíše došlo k detekci tvaru, na který je filtr specializován. Výsledek je uložen do tensoru, který tvoří výstup konvoluce. Následně je filtr posunut na další prvek tensoru (krok posunutí je obvykle 1), dokud není analyzován celý tensor [4].

Konvoluční vrstvy obsahují více filtrů, díky čemuž mohou být detekovány různé příznaky. Výstupem konvoluční vrstvy je tedy tensor s rozměry w_i, h_i, fc . d_i., kde w_i je šířka vstupu, h_i je výška vstupu, fc je počet filtrů a d_i je hloubka vstupu. Tento výstup bývá nazýván aktivační mapa nebo mapa příznaků. Vstupem může být původní obrázek nebo tensor z předchozí vrstvy sítě. Aby šířka výstupu odpovídala šířce vstupu, je nutné vstup „obalit“ hodnotami 0 tak, aby se filtr dal aplikovat i na krajní hodnoty původního vstupu. Alternativně je možné filtr aplikovat jen na pixely, jejichž vzdálenost od okraje je větší nebo rovna danému rozměru filtru. Potom bude šířka výstupu rovna w_i - w_f – 1 a výška výstupu bude h_i - h_f – 1 (za předpokladu, že krok filtru je 1) [4][31].

Je důležité poznamenat, že při konvoluci probíhá velké množství relativně jednoduchých výpočtů. Tyto výpočty mohou být prováděny paralelně na grafické kartě, která má řádově tisíce jader, což vede k výraznému zrychlení oproti tomu, kdyby byly výpočty prováděny na procesoru.

Dalším typem vrstev v konvolučních sítích jsou ReLU (Rectified Linear Unit – Usměrněná lineární jednotka) vrstvy. V těchto vrstvách je na každý prvek vstupního tensoru aplikována funkce ReLU. Použitím této nelineární funkce je do modelu zavedena nelinearita, což je důležité proto, že vstupní data jsou nelineární, a tak by je nebylo možné modelovat pouze pomocí lineárních funkcí (operace prováděné při konvoluci jsou lineární). Jako alternativu k ReLU lze například použít ELU (Exponential Linear Unit – Exponenciální lineární jednotka) nebo Leaky ReLU (Prosakující usměrněná lineární jednotka). Předpisy funkcí jsou uvedeny v Tabulce 1 [4][9][10].

Tabulka 1 - Příklady nelineárních funkci

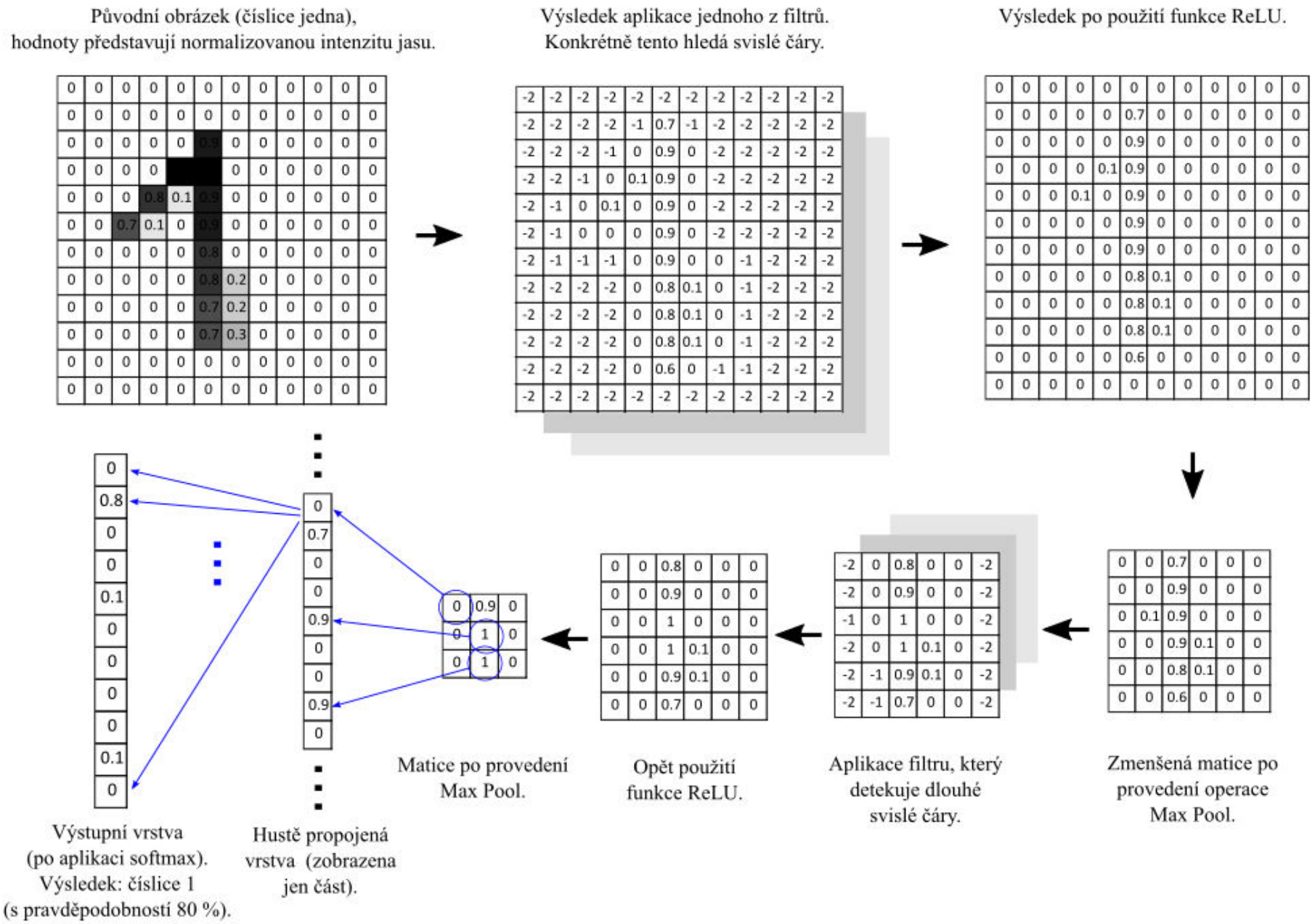
Funkce	Tvar
ReLU	$f(x) = \max(0, x)$
ELU	$f(x) = \begin{cases} \propto (e^x - 1) & \text{pro } x < 0 \\ x & \text{pro } x \geq 0 \end{cases}$
Leaky ReLU	$f(x) = \begin{cases} 0{,}01x & \text{pro } x < 0 \\ x & \text{pro } x \geq 0 \end{cases}$

Posledním typickým zástupcem vrstev CNN jsou tzv. pooling vrstvy. V těchto vrstvách dochází k podvzorkování vstupu. Stejně jako u konvoluce, i zde je procházen tensor pohyblivým oknem, které na prvky v okně aplikuje funkci a její výsledek uloží do nového tensoru, který se stane výstupem této vrstvy. Běžně se jedná o funkci max, tzn., že je vybrána největší hodnota z oblasti pokryté oknem. Alternativně může být např. vypočtena průměrná hodnota. Krok okna se volí obvykle 2 a okno obvykle bývá čtverec s délkou strany 2. V takovém případě je délka i šířka tensoru zmenšena na poloviční. Důvodem pro použití podvzorkování je zejména to, že důležitější než absolutní pozice příznaku je jeho relativní pozice k ostatním příznakům [4].

Je běžné vytvářet bloky vrstev, které se skládají z konvoluční, ReLU a Max Pool vrstvy. Za tyto bloky se umísťují hustě propojené vrstvy. Tyto vrstvy odpovídají klasickým skrytým vrstvám dopředných neuronových sítí. Počet výstupů poslední vrstvy odpovídá počtu klasifikačních tříd. Na výstup (označme jej Y) se obvykle aplikuje funkce softmax. Ta upraví vektor tak, že součet jeho prvků bude roven jedné. Lze pak tvrdit, že index třídy obrázku i je roven argmax(softmax(Y)) s pravděpodobností softmax(Y)$_i$. Funkce argmax vrací index největšího prvku vektoru. Pro ilustraci průchodu obrázku CNN byl vytvořen Obrázek 2 [4].

Před výstup je ještě obvykle řazena dropout vrstva, která má za úkol zabránit přetrénování sítě. Přetrénování je stav, kdy má síť vysokou přesnost klasifikace obrázků z trénovací množiny, ale nízkou přesnost klasifikace obrázků z validační a testovací množiny. Dropout vrstva s určitou pravděpodobností změní aktivaci neuronů na nulu, což vede k tomu, že jeho váhy nejsou upraveny během zpětného šíření chyby. Tím je docíleno snížení přetrénování [4].

Učení CNN probíhá podobně, jako u klasických dopředných neuronových sítí, tedy na základě (modifikovaného) algoritmu Zpětného šíření chyby. Vstupy do sítě tvoří obrázky z trénovací množiny. Očekávanými výstupy jsou tzv. one-hot vektory, což jsou vektory, které mají právě jeden prvek roven jedné a zbylé prvky rovny nule. Index hodnoty 1 odpovídá indexu třídy obrázku. Po průchodu trénovací dávky sítí je spočítána chyba, například jako suma čtverců rozdílů skutečných a očekávaných hodnot. Váhy jsou poté upravovány tak, aby byla chyba co nejmenší [4].



Obrázek 2 - Průchod obrázku konvoluční sítí

2 Použité SW technologie

Pro praktickou část práce byly využity zejména softwarové technologie, které jsou popsány v této kapitole.

2.1 Python

Naprostá většina praktické části práce byla napsána v programovacím jazyce Python[1] (verze 3.6). Jedná se o multiparadigmatický jazyk. V práci bylo využito jednak objektově orientované paradigma (třídy, metody, dědičnost apod.), jednak funkcionální paradigma.

Mezi hlavní důvody pro výběr tohoto jazyka patří urychlení vývoje díky dynamické typové kontrole a snadné instalaci balíčků pomocí nástroje pip. Dalším důvodem je velice dobrá čitelnost kódu ve srovnání například s jazyky C++ nebo Java, a to mj. díky tomu, že signatury funkcí a metod nevyžadují specifikaci datového typu, že není nutné používat za výrazy středníky, že se místo složených závorek používá odsazení, atd.

Horší výkonnost Pythonu oproti kompilovaným jazykům (např. C++) nepředstavuje významný problém, protože nejvíce náročné algoritmy (učení sítě a klasifikace obrázku) jsou prováděny na grafické kartě a jsou implementovány ve výkonnějších jazycích. V Pythonu je pak napsáno aplikační programovací rozhraní (API) [11].

2.2 OpenCV

Knihovna OpenCV[2] (Open Source Computer Vision Library) se v aplikacích používá pro úpravu a analýzu obrázků. Zejména se jedná o rozmazání obrázku, převod obrázku na černobílý, ekvalizaci histogramu a především detekci a analýzu tvarů (kontur).

Důvodem pro výběr OpenCV bylo to, že se jedná o knihovnu s dlouhou tradicí (od roku 1999), která poskytuje všechny funkcionality, které byly požadovány (viz odstavec výše). Knihovna je implementována v jazycích C a C++, což spolu s kvalitní optimalizací zaručuje vysokou výkonnost. Poskytuje také API v jazyce Python (a dalších). OpenCV také obsahuje moduly pro provádění operací na grafické kartě (pomocí knihovny CUDA). Postup pro zpřístupnění těchto operací z prostředí Pythonu je ale poměrně složitý a v práci nebyl použit. Jedná se ale o zajímavou možnost pro případ, že by aplikace byla dále optimalizována [12][13].

Mezi alternativy k OpenCV patří například knihovny SimpleCV nebo scikit-image. Vzhledem k tomu, že se OpenCV zcela osvědčilo, nebyly tyto knihovny hlouběji zkoumány [14].

Během vývoje bylo zjištěno, že OpenCV nedokáže načíst obrázek z disku, pokud cesta k obrázku obsahuje českou diakritiku. Problém byl vyřešen pomocí funkce `load_image` v modulu `utils`, která obrázek zakódovaný ve formátu JPG načte pomocí zabudované

[1] https://www.python.org/
[2] http://opencv.org/

funkce `open` a převede jej na pole. Toto pole je následně dekódováno na pole pixelů (pomocí OpenCV). Při ukládání obrázku je použit reverzní postup.

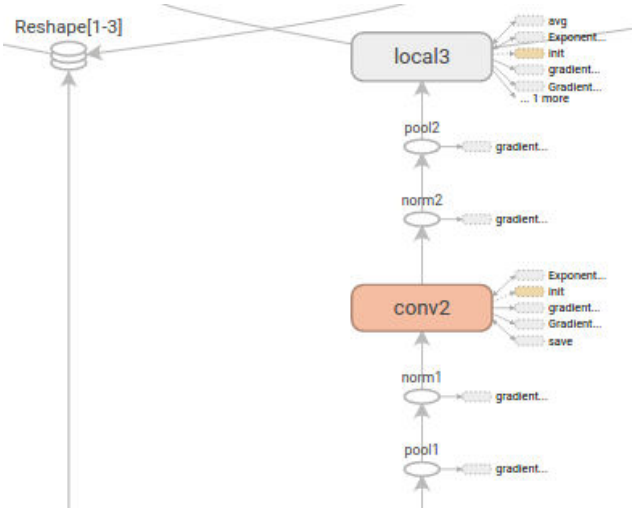
2.3 Tensorflow

„Tensorflow[3] je open-source knihovna pro provádění numerických výpočtů pomocí grafů. Vrcholy grafu reprezentují matematické operace, zatímco hrany grafu reprezentují multidimenzionální datová pole (tensory), které komunikují mezi vrcholy.“ [15]. Knihovna byla původně vyvinuta zaměstnanci firmy Google za účelem výzkumu strojového učení. Je ale natolik univerzální, že může být využita k řešení většiny úloh, které se dají definovat na základě matematických operací [15].

V aplikacích se knihovna Tensorflow používá pro definici, učení a použití konvoluční neuronové sítě. Byla vybrána proto, že se jedná o knihovnu, za kterou stojí jedna z nejvýznamnějších IT firem (Google), je stále aktivně vyvíjena a poskytuje velice dobrý výkon. Mezi alternativy patří mj. scikit-learn, Theano či Caffe [16][17].

Využití Tensorflow vyžaduje změnu úhlu pohledu na tvorbu kódu oproti klasickému programování. To může zejména v raných fázích vývoje představovat výzvu, což platilo i v případě autora této práce. K překonání této výzvy pomáhá kvalitně zpracovaná dokumentace a tutoriály [18].

Je důležité mít na paměti, že veškeré výpočty probíhají v rámci sezení (`Session`). Před spuštěním Session se provede definice výpočetního grafu. Definují se zejména matematické operace a tensory – proměnné, konstanty a dočasné hodnoty. Až po spuštění sezení jsou do tensorů dosazeny skutečné hodnoty, na které jsou aplikovány operace tak, jak jsou definovány v grafu. Průběh výpočtu je možné vizualizovat pomocí nástroje TensorBoard, viz Obrázek 3 [19].



Obrázek 3 - Vizualizace výpočetního grafu pomocí Tensorflow
zdroj: https://www.tensorflow.org/images/colorby_structure.png, upraveno

[3] https://www.tensorflow.org/

Vzhledem k tomu, že Tensorflow je relativně nová knihovna a v české literatuře se téměř nevyskytuje (jako hodnotný zdroj informací byla nalezena pouze přednáška Petra Zadražila), jsou uvedeny následující tři příklady použití [20].

V prvním příkladu jsou demonstrovány základy použití Tensorflow.

Příklad 1: Tensory a operace

```
import tensorflow as tf

# deklarace konstant
a = tf.constant(2.)
b = tf.constant(3.)

# deklarace dočasné proměnné typu float32, skutečné hodnoty budou vloženy po
  spuštění sezení (Session)
input = tf.placeholder(dtype=tf.float32)

# deklarace operace, v tomto případě lineární funkce y = 2x + 3
y = tf.add(tf.multiply(a, input), b)

# důležité - vše do této chvíle byly jen deklarace, zatím ještě neproběhly žádné
  výpočty!

# vytvoření sezení
with tf.Session() as sess:
    # výpočty se provádějí v metodě Session.run
    print(a)                                    # výpis informací o konstantě a
    print(sess.run(a))                          # výpis hodnoty konstanty a
    print(sess.run(tf.add(a, b)))               # výpis součtu konstant a a b
    print(sess.run(a + b))                      # zkrácený zápis

    x = np.linspace(-1, 1, 10)                  # vytvoření vstupů
    feed_dict = {input: x}                      # nahrazení dočasné hodnoty
                                                  skutečnou
    print(sess.run(y, feed_dict=feed_dict))     # výpočet a výpis výsledku
```

Výstup programu vypadá následovně:

```
Tensor("Const:0", shape=(), dtype=float32)
2.0
5.0
5.0
[ 1.        1.44  1.89  … 4.11  4.56  5.      ]
```

Ve druhém příkladu je již vytvořena a vytrénována nejjednodušší možná neuronová síť.

Příklad 2: Perceptron, simulující logickou funkci AND

```
import tensorflow as tf

x = [[1., 1.], [1., 0], [0, 1.], [0, 0]]     # vstupy
y = [[1.], [0], [0], [0]]                     # očekávané výstupy
epochs = 10000                                # počet epoch trénování

w = tf.Variable(tf.random_normal([2, 1]))     # váhy perceptronu
b = tf.Variable(tf.random_normal([1]))        # bias perceptronu
y_ = tf.nn.sigmoid(tf.matmul(x, w) + b)       # potenciál a aktivační funkce

# chyba je rovna sumě čtverců rozdílu očekávané a skutečné hodnoty
error = tf.reduce_sum(tf.square(y - y_))
```

```
# krok trénovaní - minimalizace chyby, rychlost učení 0.5
train = tf.train.GradientDescentOptimizer(0.5).minimize(error)

with tf.Session() as sess:
    # Inicializace proměnných
    sess.run(tf.global_variables_initializer())

    print('Výsledek před zahájením učení:\n', sess.run(y_))

    for i in range(epochs):
        sess.run(train)     # provedení jedné epochy trénování
                            # optimizátor upraví váhy tak, aby se snížila chyba

    print('Výsledek po ukončení učení:\n', sess.run(y_))
```

Výstup programu vypadá následovně:

Výsledek před zahájením učení:
[[0.60574317]
[0.65000296]
[0.63088709]
[0.6738441]]
Výsledek po ukončení učení:
[[9.80597794e-01]
[1.63715668e-02]
[1.63715668e-02]
[5.48121852e-06]]

Jako aktivační funkci v Příkladu 2 by bylo vhodnější použít skokovou funkci místo sigmoidy. Tu by ale bylo nejdříve nutné implementovat, což by tento příklad zbytečně znepřehlednilo.

V posledním příkladu je vytvořena vícevrstvá neuronová síť pro aproximaci funkce Rastrigin. Jedná se o funkci, která se využívá pro testování optimalizačních algoritmů a podle [21] je definována následovně (Rovnice 2):

$$y = A + \sum_{i=1}^{n} [x_i^2 - A\cos(2\pi x_i)] \tag{2}$$

kde n je počet dimenzí a A je konstanta, obvykle velikosti 10. Definiční obor je obvykle volen [-5,12; 5,12] pro každé x_i.

Příklad 3: Aproximace funkce Rastrigin

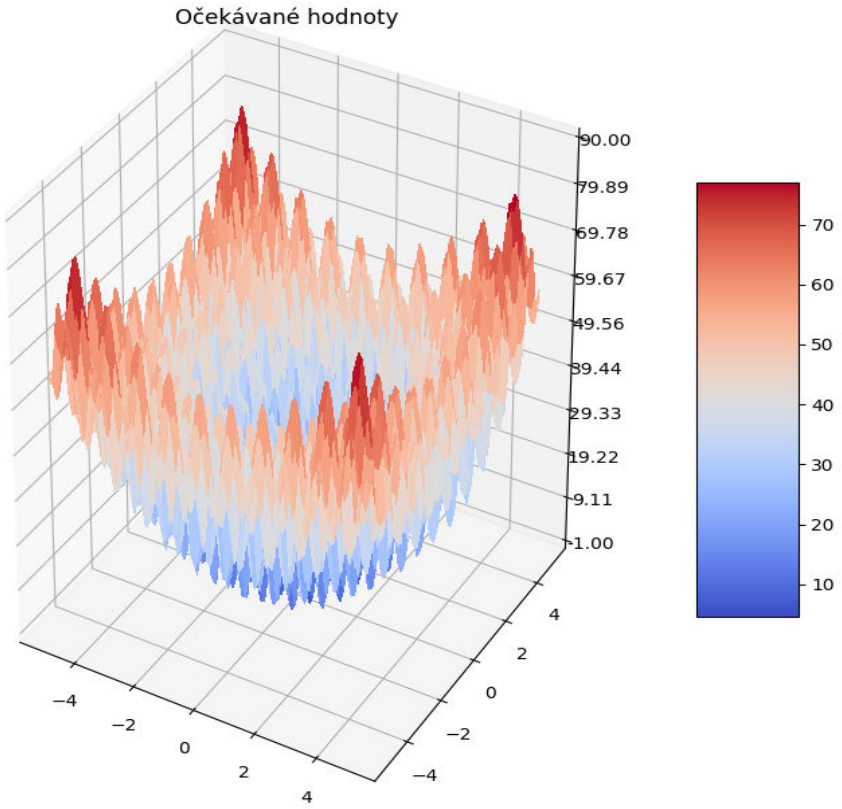
```
from math import cos, pi
import numpy as np
import tensorflow as tf


def rastrigin(x):
    """
    Funkce Rastrigin.
    :param x: Řádkový vektor
    :return: Hodnota funkce Rastrigin pro x
    """
    A = 10
    n = len(x)
    s = sum([xi**2 - A*cos(2*pi*xi) for xi in x])
    y = A*n+s

    return y



def add_layer(input, in_size, out_size, activation_fcn = None):
    """
    Funkce pro přidání vrstvy sítě.
    :param input: Vstup, může se jednat o vstupní data nebo výstup
                  z předchozí vrstvy
    :param in_size: Velikost vstupu
    :param out_size: Velikost výstupu (= počet neuronů v síti)
    :param activation_fcn: Aktivační funkce neuronů
    :return: výstup z vrstvy
    """
    # váhy, inicializovány na náhodnou hodnotu
    w = tf.Variable(tf.random_normal([in_size, out_size]))

    # bias, inicializován na náhodnou hodnotu
    b = tf.Variable(tf.random_normal([1, out_size]))

    # přidání aktivační funkce, pokud byla zadána
    if activation_fcn is not None:
        out = activation_fcn(tf.matmul(input, w) + b)
    else:
        out = tf.matmul(input, w) + b

    return out


def rastrigin_aproximator():
    """
    Funkce, která vytrénuje dopřednou neuronovou síť pro aproximaci
    funkce Rastrigin.
    """

    # vytvoření trénovacích dat
    linspace = np.linspace(-5.12, stop=5.12, num=100)
    x = [[x1i, x2i] for x1i in linspace for x2i in linspace]  # vstupy
    expected_out = [[rastrigin(_x)] for _x in x]              # očekávané výstupy

    input_size = 2  # počet vstupů
    output_size = 1 # počet výstupů
    l1_size = 512   # počet neuronů v první vrstvě
    l2_size = 1024  # počet neuronů v druhé vrstvě
    l3_size = 1024  # počet neuronů v třetí vrstvě
    l4_size = 1024  # počet neuronů ve čtvrté vrstvě
    l5_size = 512   # počet neuronů v páté vrstvě
```

```
    epochs = 50000  # počet epoch učení

    # dočasný vstup, None značí, že velikost není specifikována
    input = tf.placeholder(tf.float32, [None, input_size])
    l1_out = add_layer(input, input_size, l1_size, tf.nn.tanh)  # 1. skrytá v.
    l2_out = add_layer(l1_out, l1_size, l2_size, tf.nn.tanh)    # 2. skrytá v.
    l3_out = add_layer(l2_out, l2_size, l3_size, tf.nn.tanh)    # 3. skrytá v.
    l4_out = add_layer(l3_out, l3_size, l4_size, tf.nn.tanh)    # 4. skrytá v.
    l5_out = add_layer(l4_out, l4_size, l5_size, tf.nn.tanh)    # 5. skrytá v.
    out = add_layer(l5_out, l5_size, output_size)               # výstupní v.

    # Výpočet chyby jako střední hodnota ze sumy čtverců
    err = tf.reduce_mean(tf.reduce_sum(tf.square(expected_out - out), axis=[1]))

    # Operace pro učení
    train = tf.train.GradientDescentOptimizer(0.001).minimize(err)

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())

        for i in range(epochs):
            sess.run(train, feed_dict={input: x})

            if i % 1000 == 0:
                print(i, '=', sess.run(err, feed_dict={input: x}))

        # ... vizualizace

rastrigin_aproximator()
```

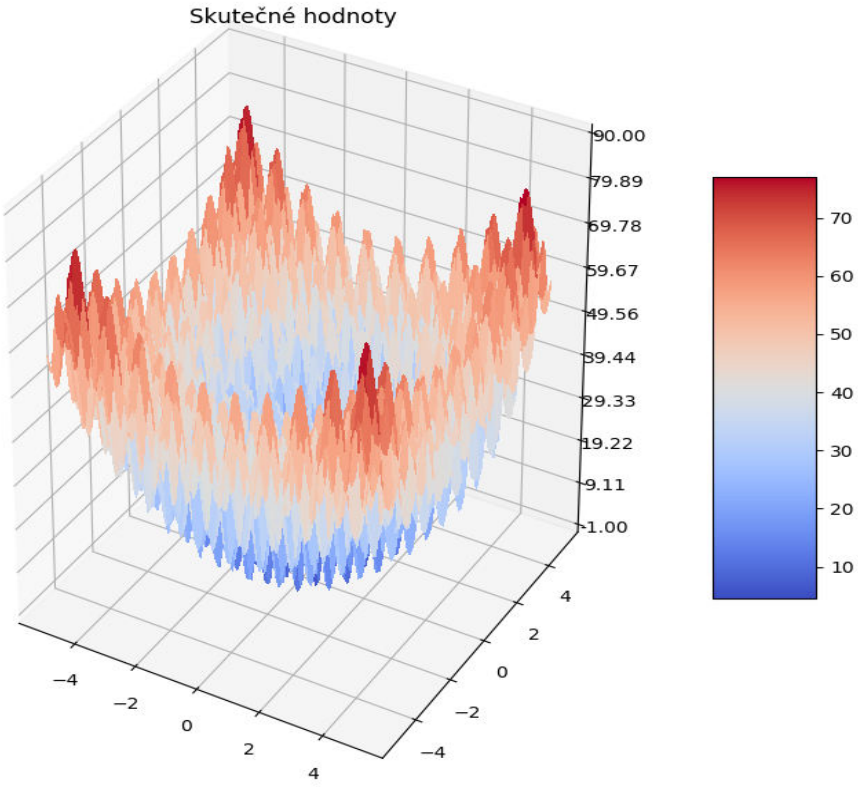
Výstup programu může vypadat následovně:

```
0 = 1555.47
1000 = 136.627
2000 = 78.4571
...
22000 = 0.067576
23000 = 0.847499
24000 = 0.0264567
...
47000 = 0.000475504
48000 = 0.000484442
49000 = 0.000424193
```

Po srovnání Obrázku 4 a Obrázku 5 je možné konstatovat, že jsou grafy téměř totožné, což znamená, že síť je dobrým aproximátorem funkce Rastrigin (na daném definičním oboru). V Příkladu 3 nebyla uvedena část kódu provádějící vizualizaci, pomocí které byl získán Obrázek 4 a Obrázek 5, protože není důležitá pro demonstraci toho, jak je možné pomocí Tensorflow vytvořit hlubokou neuronovou síť. Kompletní zdrojový kód pro všechny příklady je možné najít na přiloženém DVD ve složce source/examples.



Obrázek 4 - Vizualizace funkce Rastrigin pro n=2

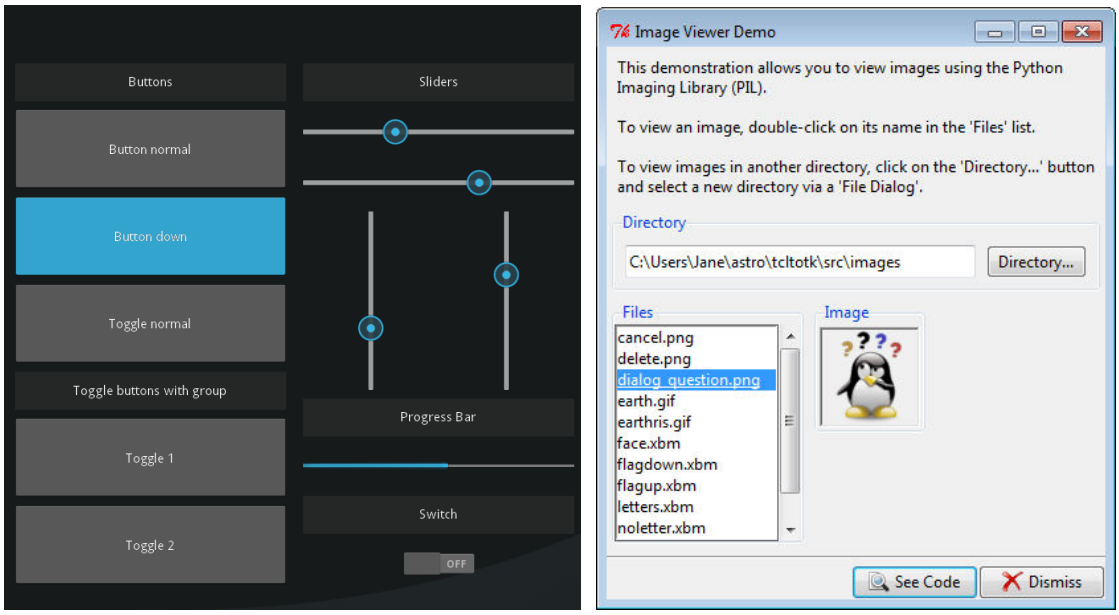


Obrázek 5 - Aproximace funkce Rastrigin pomocí neuronové sítě

2.4 Další technologie

2.4.1 Kivy

Pro tvorbu grafického uživatelského rozhraní byla použita knihovna Kivy[4]. Důvodem pro výběr této knihovny byl zejména intuitivní styl psaní kódu (separace vzhledu a chování, definice vlastních GUI elementů pomocí dědičnosti apod.). Mezi další důvody patří moderní vzhled aplikací a kvalitně zpracovaná dokumentace.



Obrázek 6 - Typický vzhled GUI v Kivy (vlevo) a TkInter (vpravo)
zdroje: Kivy (upraveno), TkInter

Na Obrázku 6 je pro ilustraci uvedeno srovnání typického vzhledu aplikace při použití Kivy a TkInter. TkInter je nejčastěji používaná knihovna pro tvorbu GUI v Pythonu (zároveň je také součástí instalace Pythonu) [22].

2.4.2 NumPy

Důležitou podpůrnou roli plní knihovna NumPy[5]. Používá se zejména pro reprezentaci obrázku ve formě datového pole pixelů. Nad těmito poli jsou prováděny operace jako ořez, spojení či transpozice. NumPy je také využíváno knihovnou OpenCV. Výhodou NumPy je vysoká výkonnost díky tomu, že je na pozadí implementováno v jazyce C [23].

[4] https://kivy.org
[5] http://www.numpy.org/

3 Metodika vývoje

Pro vývoj aplikací byl zvolen prototypový přístup. Důvodem pro výběr tohoto přístupu bylo především to, že autor práce byl s problematikou strojového učení a zpracování obrazu před zahájením vývoje seznámen jen povrchně, a tak by výstupy fází analýza a návrh (ve smyslu metodiky Unified Process) pravděpodobně nebyly při implementaci příliš dobře použitelné.

Pro každý prototyp byly stanoveny cíle, po jejichž dosažení byl prototyp konzultován s vedoucím práce. To připomíná situaci z praxe, kdy jsou prototypy konzultovány se zákazníkem.

3.1 První prototyp

První prototyp sloužil jako ověření proveditelnosti (Proof of Concept). V této fázi vývoje ještě nebylo jisté, zda je pro problém klasifikace svarů možné efektivně použít konvoluční neuronovou síť.

Předzpracování obrazových dat bylo vyřešeno pouze elementárně. Z malého počtu obrázků byly zjištěny koordináty svarů, tyto hodnoty byly zprůměrovány a následně se pro všechny obrázky provedlo vyříznutí oblastí svarů a jejich následné spojení do jednoho obrázku. Rovněž byla provedena ekvalizace histogramu. Zatím ještě nebylo možné specifikovat počet kanálů barev a velikost obrázku po předzpracování (tyto hodnoty byly zadány jako konstanty), ale předzpracování už probíhalo paralelně.

Pro klasifikaci byla použita konvoluční síť z oficiálního tutoriálu [24], upravená tak, aby ji bylo možné použít s poskytnutými daty (zejm. velikost vstupů a počet tříd pro klasifikaci). Definice a učení CNN probíhalo uvnitř jedné funkce. Součástí prvního prototypu byly funkce pro rozdělení dat na trénovací, validační a testovací množinu. Množina dat pro učení byla tvořena jen částí celkové množiny (byly použity fotografie jen z jednoho dne focení, tedy cca 3 000 fotografií).

Přesnost testování se pohybovala mezi 85 a 90 %, což bylo považováno jako dostatečné pro ověření proveditelnosti.

3.2 Druhý prototyp

Druhý prototyp již obsahoval jednoduché GUI. Uživatel měl možnost načíst obrázek pro klasifikaci. Po načtení se provedlo předzpracování a klasifikace. Následně byl zobrazen předzpracovaný obrázek a uživatel byl informován o výsledku klasifikace (zatím jen třída obrázku – OK nebo Vadný).

Předzpracování obrazových dat již bylo oproti prvnímu prototypu sofistikovanější. Na základě souřadnic horizontálních odlesků a vertikálního odlesku (více viz kap. 4) byly vypočítány souřadnice pro výřezy okolo svarů. Tyto výřezy byly následně spojeny do jednoho obrázku. Tato metoda byla vyzkoušena na části fotografií z každého dne focení

a poskytovala již celkem uspokojující výsledky. Bylo také implementováno logování a vizualizace detekovaných tvarů.

Byla vytvořena Python třída představující CNN s metodami pro vytvoření, učení, klasifikaci, ukládání a načítání. Tím došlo k značnému zpřehlednění kódu a především zvětšení jeho univerzálnosti.

Vzhledem k tomu, že se v této fázi projekt již poměrně rozrostl, byl vytvořen lokální git repositář. Ten se v pozdějších fázích vývoje ověřil zejména v situacích, kdy bylo nutné přejít zpět k fungujícímu kódu.

3.3 Třetí prototyp

Třetí prototyp již měl GUI silně připomínající finální verzi. Uživatel měl možnost definovat novou síť, zatím jen ale s výchozí architekturou. Také mohl zjišťovat informace o vrstvách sítě (např. počet neuronů hustě propojené vrstvy). Dále bylo možné zadat parametry učení, spustit jej a průběh sledovat na grafech. Byl také přidán výpis pravděpodobnosti, že je klasifikace správná. Poslední důležitou změnou v GUI bylo přidání možnosti online učení (pro případy, kdy klasifikace nebyla správná).

Co se předzpracování obrázků týče, tak bylo přidáno detekování kruhového a obdélníkového výřezu (viz Obrázek 10). Vzhledem k relativně vysokému počtu falešných pozitiv bylo nutné implementovat funkce pro filtrování detekovaných tvarů. Byly také vytvořeny funkce pro výpočet souřadnic svarů na základě toho, jaké tvary byly detekovány. Tyto funkce byly následně optimalizovány.

Třída `CNN` se v tomto prototypu příliš nezměnila. Byla přidána metoda pro online učení. Dále byla vytvořena třída `DataProvider` pro načítání obrázků z disku, rozdělení obrázků do množin (trénovací, validační a testovací) a poskytování dávek obrázků během učení.

3.4 Čtvrtý prototyp

Ve čtvrtém prototypu bylo GUI doplněno o možnost definovat vlastní architekturu sítě. Byla také implementována kontrola vstupních parametrů, aby aplikace nebyla neočekávaně ukončena v případě, že uživatel zadá špatný vstup (například řetězec místo čísla).

Do třídy `CNN` byla přidána podpora více tříd pro klasifikaci a podpora barevných obrázků. Bylo také nutné upravit to, jak se síť ukládá na disk, protože již není možné předpokládat, že síť bude mít výchozí architekturu. Kromě základních parametrů sítě byly tedy ukládány i informace o jednotlivých vrstvách.

Podpora barevných obrázků byla přidána také do algoritmu pro předzpracování obrazových dat. Uživatel také nyní může zadat, jaká funkce má být použita pro předzpracování, což zvyšuje univerzálnost aplikace.

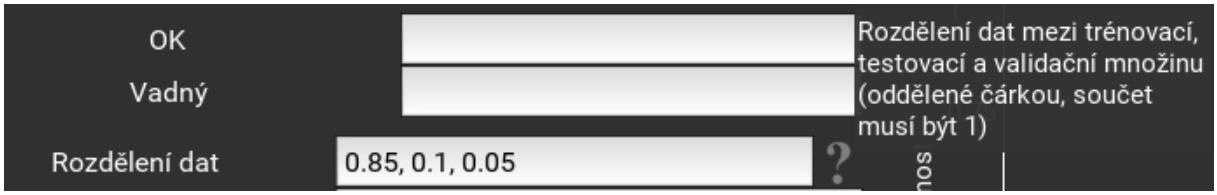
Za účelem uživatelsky pohodlnější a rychlejší klasifikace zejména většího množství obrázků byla vytvořena konzolová aplikace, která je více popsána v kapitole 7. Dále byly vytvořeny Unit testy a testovací scénáře, kterým je věnována kapitola 8. Součástí prototypu také byla Uživatelská příručka a Návod pro instalaci (nacházející se na přiloženém DVD).

3.5 Finální verze

Poté, co bylo ověřeno, že čtvrtý prototyp splňuje všechny požadavky, bylo přistoupeno k refaktorování, jehož cílem bylo vytvoření čitelného, dobře zdokumentovaného kódu. Zejména docházelo k dělení funkcí a metod na menší, vytváření nových tříd a odstraňování duplicit. Byly také odstraňovány chyby v kódu, zjištěné ať už při testování, nebo při procházení kódu.

Textové řetězce, jako např. chybové hlášky nebo popisky GUI elementů, byly přesunuty do jednoho souboru. Tím jednak došlo k větší separaci kódu a GUI a také tím bude usnadněn případný překlad do jiného jazyka. Podobně byly přesunuty „magické konstanty“ do jednoho konfiguračního souboru.

V GUI již nebylo provedeno mnoho změn. Pozornost byla kladena spíše na design a s tím související zlepšení zkušenosti uživatele s aplikací. Bylo tedy upraveno rozložení GUI elementů tak, aby působilo co nejintuitivněji. Pro zvýšení uživatelského komfortu byly do aplikace přidány tooltipy. Po najetí kurzorem na otazník dojde k zobrazení okna s nápovědou, viz Obrázek 7.



Obrázek 7 - Tooltip

Do projektu byl přidán powershellový skript pro kontrolu požadavků aplikace a pro instalaci Python balíčků. Mezi požadavky patří Python interpret, CUDA, apod. Rovněž byl přidán dávkový soubor pro spuštění GUI aplikace.

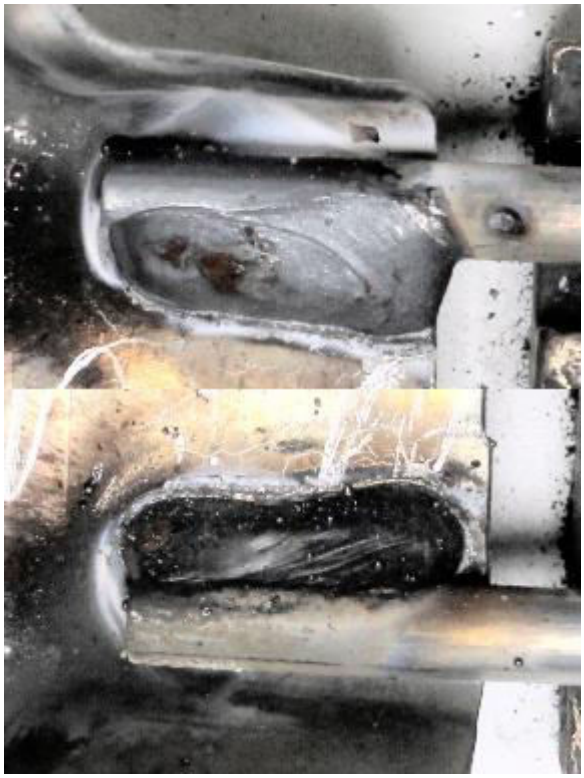
3.6 Programy používané během vývoje

Projekt byl vyvíjen v integrovaném vývojovém prostředí PyCharm Community Edition 2017.1.4[6]. Aplikace je distribuována právě jako projekt tohoto IDE. Vývojové a UML diagramy, které se nacházejí v přílohách, byly vytvořeny v programu Enterprise Architect[7].

[6] https://www.jetbrains.com/pycharm/specials/pycharm/pycharm.html
[7] http://www.sparxsystems.com/products/ea/

4 Předzpracování obrazových dat

Klíčovou součástí práce bylo vytvoření algoritmu, který z fotografie dílu používaného v automobilovém průmyslu provede výřezy okolo svarů a tyto výřezy spojí do jednoho obrázku. Takto předzpracované obrázky se použijí při učení sítě. Algoritmus se také použije na vstupní obrázek při klasifikaci. Výsledek předzpracování je ilustrován na Obrázku 8. Konkrétně tento výstup byl získán předzpracováním Obrázku 9.



Obrázek 8 - Předzpracovaný obrázek

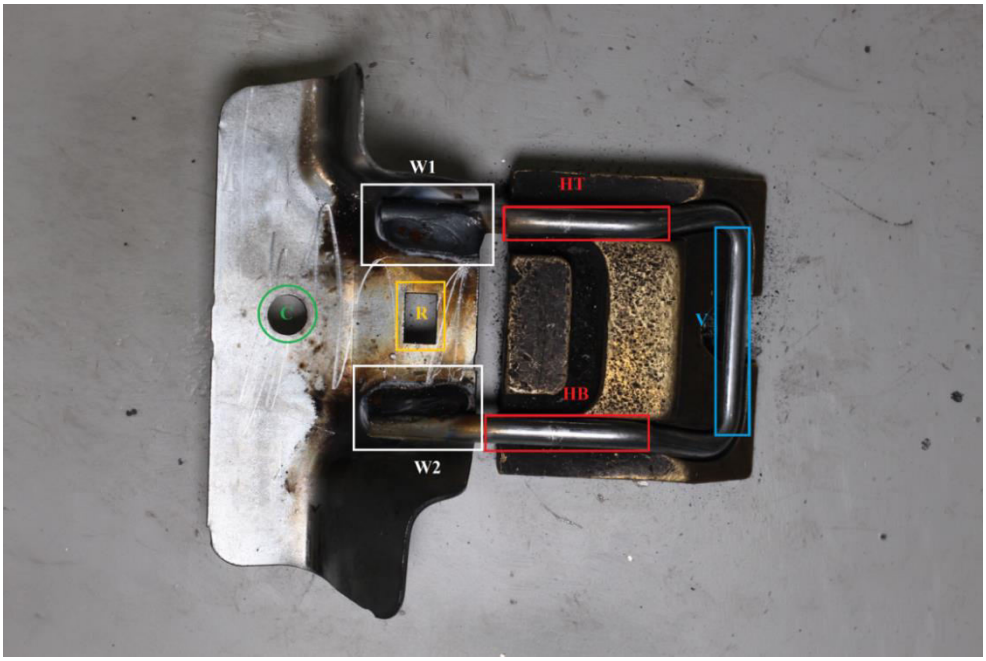
Kód pro předzpracování obrazových dat se nachází ve zdrojových souborech umístěných v modulu `preprocess`. Vývojový diagram, ve kterém je zachyceno předzpracování z vysokoúrovňového pohledu, se nachází v Příloze A.

4.1 Popis fotografií

Množina dat je tvořena 30 332 barevnými obrázky ve formátu JPG v rozlišení 3 456 x 2 304 px. Fotografie se liší zejména pozicí dílu na fotografii a natočením dílu. Svary jsou vyznačeny na Obrázku 10 s označením **W1 a W2.** Bohužel není známo, který ze dvou svarů je vadný (případně zda nejsou vadné oba). Kdyby tato informace byla známa, tak by nebylo nutné svary po oříznutí spojovat do jednoho obrázku. Především by ale síť pravděpodobně byla efektivnější, protože by pouze rozhodovala, zda je svar vadný, či ne. Takto ale síť musí vyhodnotit dvojici svarů, ze které jeden svar může být vadný a druhý ne.

Obrázek 9 - Originální fotografie dílu
zdroj: zadávací dokumentace

Původním záměrem bylo souřadnice svarů detekovat na základě kruhového a obdélníkového výřezu (na Obrázku 10 jsou označeny jako **C**, resp. **R**). Tento postup se ale příliš neosvědčil, protože se tyto výřezy často nepodařilo detekovat, pokud jejich hrany splývaly s okolím. Bylo ale zjištěno, že jsou téměř vždy detekovány odlesky (na Obrázku 10 jsou označeny jako **HT**, **HB** a **V**), na základě kterých je možné získat souřadnice svarů.



Obrázek 10 - Fotografie dílu s vyznačenými tvary a svary
zdroj: zadávací dokumentace, upraveno

V případě, že se podaří detekovat výřezy, tak i jejich pozice je zohledněna do výpočtu souřadnic svarů. To znamená, že algoritmus bude fungovat i v případě, že například kvůli změně světelných podmínek se již na obrázcích nebudou vyskytovat odlesky. Je ale zřejmé, že úspěšnost detekce bude nižší.

4.2 První fáze – příprava obrázku

Algoritmus předzpracování lze rozdělit na tři fáze. První z nich je příprava obrázku, druhou je detekce svarů a poslední je fáze finálních úprav. Celý proces je zachycen na diagramu v Příloze B.

První a třetí fázi provádějí funkce definované v Python modulu `img_preprocess`. Prvním krokem je načtení obrázku. Zda bude obrázek načten v odstínech šedi nebo ve formátu RGB záleží na parametru funkce `proprecess_single`. Pokud načtení selže, tak je předzpracování ukončeno. Důvodem může být např. poškozený soubor nebo jeho nenalezení.

Následně se zkontroluje orientace obrázku. Pokud je obrázek orientován na výšku, je otočen o 90 ° doleva (tj. proti směru hodinových ručiček). Zde se pracuje s jedním z mála předpokladů o fotografiích, a to tím, že vodorovná osa dílu je rovnoběžná s delší stranou obrázku (resp. že spolu svírají velice ostrý úhel). K otočení se používá funkce `cv2.transpose`, která transponuje datové pole (matici), ve kterém jsou uloženy pixely obrázku.

Součástí přípravy obrázku je také ověření, že jeho rozměry nejsou menší než velikost výřezu. Pokud by rozměry byly menší, tak by se výřez provedl mimo obrázek, což je neplatná operace.

4.3 Druhá fáze - detekce svarů

Funkce pro detekci svarů jsou implementovány v Python modulu `get_welds_coords`. Proces detekce svarů je znázorněn ve formě vývojového diagramu v Příloze C.

Zjednodušeně lze říci, že se provede příprava obrázku na analýzu. Poté se obrázek opakovaně převede do odstínů šedi (s různými hodnotami prahů). Na takto upravených obrázcích jsou detekovány a filtrovány tvary popsané v kapitole 4.1 a vyznačené na Obrázku 10. Na základě pozice detekovaných tvarů jsou vypočítány souřadnice svarů. Tyto hodnoty jsou poté zprůměrovány, čímž jsou získány konečné souřadnice svarů.

Souřadnice svaru se obecně vypočítají přičtením nebo odečtením empiricky zjištěných hodnot od středu detekovaného tvaru. Při implementaci bylo myšleno na to, že rozlišení obrázků by se mohlo v budoucnu změnit. Proto se obecně nepracuje s konstantními hodnotami, ale použije se jeden z rozměrů detekovaného tvaru, vynásobený empiricky zjištěnou hodnotou.

V ojedinělých případech se sice pracuje s konstantními hodnotami, ty jsou ale vynásobeny poměrem očekávané a skutečné velikosti obrázku (viz dále), čímž je opět zaručeno to, že

detekce bude fungovat i pro obrázky s jiným rozlišením, než jaké měly ty z původní datové množiny.

I když je díl na většině fotografií orientován tak, jako např. na Obrázku 9, dokáže algoritmus detekovat středy i v případě, že je díl otočen o 180 °. Zjištění orientace dílu se provádí ve funkci `get_offset_sign`. Tato funkce vrací hodnotu -1 nebo 1, kdy -1 značí obvyklou orientaci. Orientace se zjistí porovnáním x-ových souřadnic detekovaných tvarů. Pokud byl např. detekován kruhový výřez a vertikální odlesk a x-ová souřadnice kruhového výřezu je větší než x-ová souřadnice vertikálního odlesku, jedná se o neobvyklou orientaci a funkce vrátí hodnotu 1.

4.3.1 Před prahováním

Než se přistoupí k prahování, tak je obrázek upraven tak, aby se zlepšila úspěšnost detekce. Jsou provedeny úpravy, které vychází z doporučení v [25] a [26]. Konkrétně dojde k ekvalizaci histogramu a rozmazání obrázku pomocí funkce `cv2.GaussianBlur`. Rozmazání se provádí proto, aby byl z obrázku (alespoň částečně) odstraněn šum.

Rovněž dojde k inicializaci seznamu prahů, a to na základě počáteční hodnoty, koncové hodnoty a kroku. Tyto hodnoty jsou načteny z konfiguračního souboru. Obvykle dojde k vytvoření seznamu v podobě 30, 35, … 195, 200.

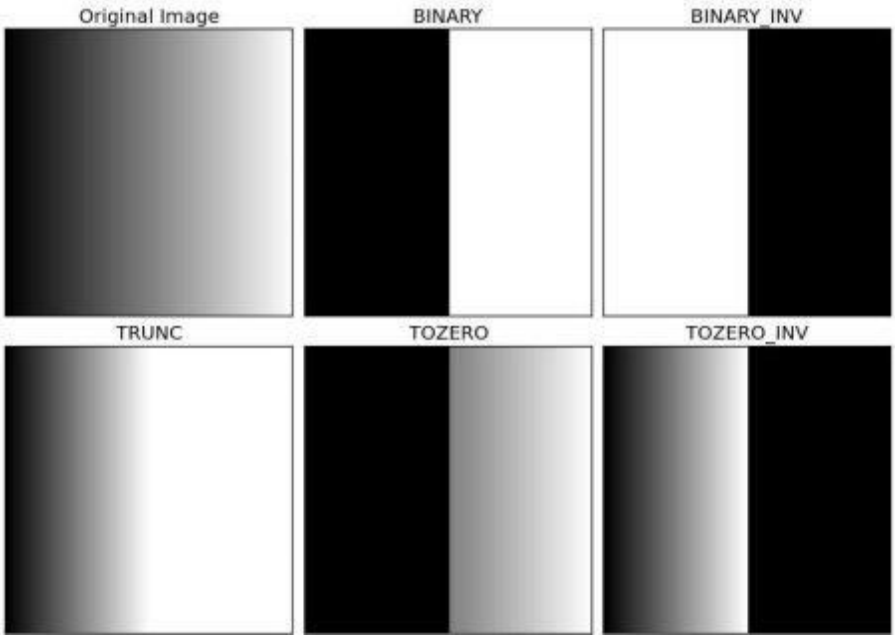
4.3.2 Výpočet souřadnic

Výpočet souřadnic se provádí pro každý práh. Výpočet může probíhat paralelně. Při paralelním výpočtu došlo k přibližně dvojnásobnému zrychlení celého předzpracování (na PC s CPU, který podporuje 8 paralelně běžících vláken).

4.3.2.1 Prahování

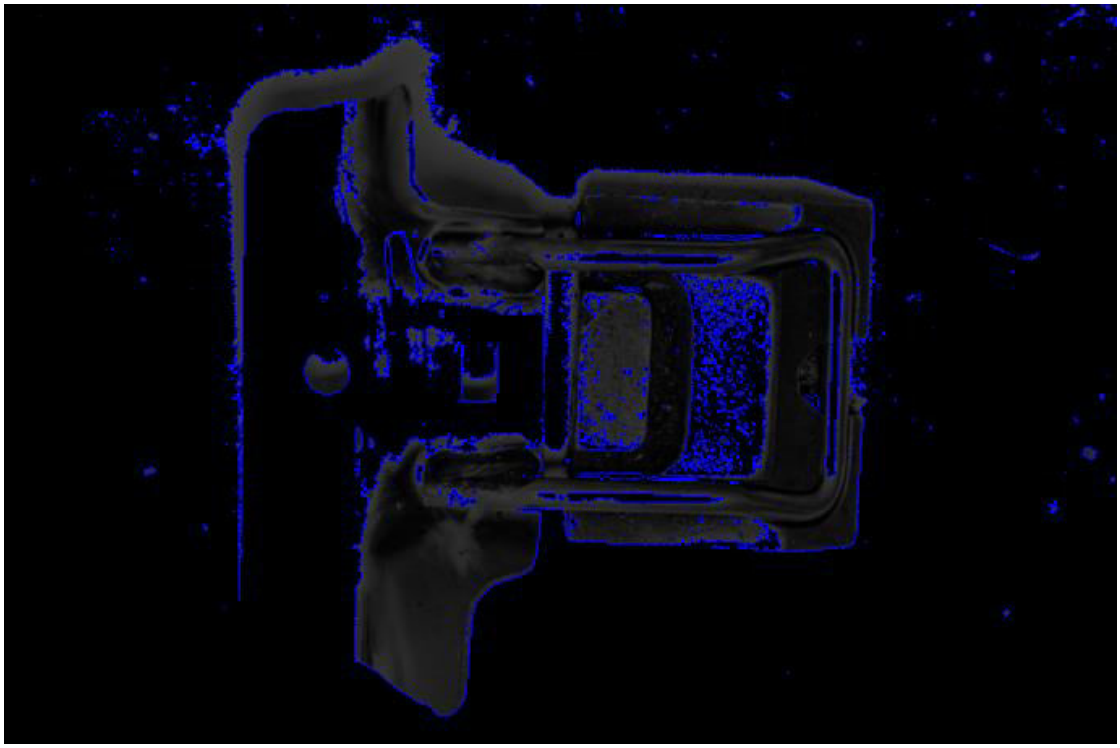
Prahováním se rozumí taková úprava obrázku, kdy hodnotám menší nebo rovným prahu je přiřazena barva na základě jedné funkce, a hodnotám větším než práh na základě jiné funkce. OpenCV nabízí několik metod prahování (viz Obrázek 11). Nejlépe se osvědčila metoda `cv2.THRESH_TOZERO_INV`.

Bylo zjištěno, že z důvodu různých nečistot na povrchu dílu je nutné detekci tvarů provádět opakovaně pro různé hodnoty prahů. Zatímco například pro jeden obrázek by všechny tvary byly detekovány pro práh 80, pro jiný obrázek by pro stejný práh nemusel být detekován tvar žádný.



Obrázek 11 - Metody prahování, práh = 127
zdroj: http://docs.opencv.org/trunk/threshold.jpg

Po prahování je překročeno k detekci tvarů pomocí funkce `cv2.findContours`. V tuto chvíli je z důvodu velkého počtu hran v obrázku detekováno velké množství tvarů (viz Obrázek 12). Je tedy nutné provést klasifikaci a filtraci.



Obrázek 12 - Obrysy všech detekovaných tvarů pro jeden z prahů

Nejdříve jsou odstraněny tvary s příliš malým či velkým obsahem. Obsah tvaru je možné zjistit pomocí funkce `cv2.contourArea`. Empiricky zjištěné minimální a maximální hodnoty jsou načteny z konfiguračního souboru a vynásobeny poměrem skutečné a očekávané velikosti obrázku.

4.3.2.2 Klasifikace tvarů

Následně je provedena klasifikace. Pomocí funkce `cv2.approxPolyDP` je zjištěn počet vrcholů tvaru. Bylo zjištěno, že odlesky obvykle mají 4 nebo 5 vrcholů. Pokud má tvar více vrcholů, pravděpodobně se jedná o výřez. Tato úvaha vychází z [27].

V případě odlesků je kolem tvaru vytvořen ohraničující obdélník (pomocí `cv2.minAreaRect`). Tento obdélník je převeden na instanci třídy `Rectangle`. Je vypočítán poměr stran tohoto obdélníku jako šířka / výška. Pokud je poměr v intervalu [4; 40], je tvar považován za horizontální odlesk. Pokud je poměr v intervalu [0,01; 0,1], je tvar považován za vertikální odlesk.

Tvar je klasifikován jako obdélníkový výřez, pokud je poměr stran jeho ohraničujícího obdélníku v intervalu [0,8; 1,9] a pokud obsah tvaru je alespoň poloviční jako obsah ohraničujícího obdélníku.

A konečně tvar je klasifikován jako kruhový výřez, pokud obsah tvaru zaplní alespoň 60 % plochy ohraničujícího kruhu. Ohraničující kruh je instance třídy `Circle`, vytvořená z hodnot získaných pomocí funkce `cv2.minEnclosingCircle`. Hranice intervalů a potřebné zaplnění ohraničujících tvarů byly v průběhu optimalizace algoritmu upravovány.

4.3.2.3 Filtrování tvarů

V tuto chvíli ještě množina detekovaných tvarů obsahuje velké množství falešných pozitiv. Častým jevem je, že se tvar klasifikuje jako kruhový a zároveň obdélníkový výřez. Také dochází k tomu, že jsou detekovány tvary podobné těm, které hledáme, ale na nesprávných místech.

Rovněž se stává, že jsou detekovány dva odlesky blízko u sebe. Není možné rozhodnout, který tvar je skutečný odlesk a který je falešné pozitivum, a tak jsou tyto odlesky spojeny do jednoho tak, že jsou zprůměrovány jejich souřadnice, rozměry a úhel otočení. Toto spojení je implementováno ve statické metodě `merge_rects` třídy `Rectangle`.

Následně jsou odstraňována falešná pozitiva výřezů na základě porovnání souřadnic výřezů a odlesků (samozřejmě za předpokladu, že odlesky byly detekovány). V případě vertikálního odlesku je také brán v potaz rozdíl y-ových souřadnic výřezu a vertikálního odlesku. Tento rozdíl by měl být minimální, protože výřezy i vertikální odlesk leží na (přibližně) vodorovné ose. Pokud byly detekovány oba horizontální odlesky, pak je další y-ová hodnota pro porovnání získána zprůměrováním y-ových souřadnic horizontálních odlesků.

Algoritmus je schopný detekovat falešná pozitiva výřezů, pokud byl detekován alespoň jeden odlesk. Ideální ale je, pokud byly detekovány všechny odlesky. Obecně filtrování funguje tak, že rozdíl souřadnic musí být větší než *m*-násobek šířky (či výšky) odlesku a menší než *n*-násobek šířky (či výšky) odlesku, kde *m* a *n* jsou empiricky zjištěné hodnoty.

Dále se provede srovnání vzdáleností všech výřezů, a pokud jsou nalezeny dva výřezy, mezi kterými je příliš malá vzdálenost, tak jsou oba brány jako falešná pozitiva (není již možné rozhodnout, který z nich skutečně odpovídá výřezu). Za příliš malou se považuje vzdálenost 100 px * *p*, kde *p* je poměr skutečné a očekávané velikosti obrázku.

Posledním krokem je odstranění nejasných detekcí. Výstupem filtrování musí být nejvýše jeden kruhový výřez, nejvýše jeden obdélníkový výřez, nejvýše dva horizontální odlesky a nejvýše jeden vertikální odlesk. Pokud v některé z kategorií bylo detekováno více tvarů a nebyly odstraněny filtrováním, jsou všechny tvary dané kategorie považovány za falešná pozitiva a jsou odfiltrovány.

4.3.2.4 Výpočet souřadnic

Podle toho, jaké tvary nebyly vyfiltrovány, se zvolí jedna z funkcí pro výpočet souřadnic, uvedených v Tabulce 2.

Tabulka 2 – Funkce pro výpočet souřadnic svarů

Funkce	Počet detekovaných tvarů				Ohodnocení
	Kruhový výřez	Středový výřez	Horizontální odlesky	Vertikální odlesk	
c1_mr1_h2_v1	1	1	2	1	9
c1_mr1_h2_v0	1	1	2	0	8
c1_mr1_h1_v1	1	1	1	1	8
c1_mr1_h1_v0	1	1	1	0	7
c1_mr1_h0_v1	1	1	0	1	7
c1_mr1_h0_v0	1	1	0	0	6
c1_mr0_h2_v1	1	0	2	1	6
c1_mr0_h2_v0	1	0	2	0	5
c1_mr0_h1_v1	1	0	1	1	5
c1_mr0_h1_v0	1	0	1	0	4
c1_mr0_h0_v1	1	0	0	1	4
c0_mr1_h2_v1	0	1	2	1	6
c0_mr1_h2_v0	0	1	2	0	5
c0_mr1_h1_v1	0	1	1	1	5
c0_mr1_h1_v0	0	1	1	0	4
c0_mr1_h0_v1	0	1	0	1	4
c0_mr0_h2_v1	0	0	2	1	3
c0_mr0_h1_v1	0	0	1	1	2

Kombinace tvarů, které nejsou uvedené v Tabulce 2, se nepoužívají, a to z důvodu příliš nízké spolehlivosti. Většina nízkoúrovňových funkcí z Tabulky 2 využívá následující vysokoúrovňové funkce:

- `get_coords_from_circle_and_mid_rect`
  - vypočítá souřadnice podle pozice kruhového a obdélníkového výřezu,
- `get_coords_from_rects`
  - vypočítá souřadnice podle pozic odlesků,
- `get_coords_from_circle_and_vertical`
  - vypočítá souřadnice podle pozice kruhového výřezu a vertikálního odlesku,
- `get_coords_from_middle_rect_and_vertical`
  - vypočítá souřadnice podle pozice obdélníkového výřezu a vertikálního odlesku.

Souřadnice svarů se vypočítávají na základě souřadnic tvarů, ke kterým jsou přičítány či odečítány hodnoty odpovídající vzdálenostem mezi tvary, případně rozměrům tvarů. Tyto hodnoty jsou vynásobeny o empiricky zjištěné konstanty. Tyto konstanty byly postupně optimalizovány tak, aby chyba detekce byla co nejmenší, viz kapitola 4.3.3.

Ukázka kódu: Funkce get_coords_from_circle_and_mid_rect

```
def get_coords_from_circle_and_mid_rect(circle, mid_rect):
    if circle is None or mid_rect is None:
        return None

    dx = abs(circle.x - mid_rect.x)

    w1x = mid_rect.x
    w1y = mid_rect.y - 0.8 * dx
    w2x = w1x
    w2y = mid_rect.y + 0.8 * dx

    return w1x, w1y, w2x, w2y
```

Hodnoty vrácené vysokoúrovňovými funkcemi mohou být v nízkoúrovňových kombinovány. Např. funkce `c1_mr1_h2_v1` vypočte souřadnice jako průměr souřadnic získaných z funkcí `get_coords_from_circle_and_mid_rect` a `get_coords_from_rects`.

Každé nízkoúrovňové funkci bylo přiřazeno ohodnocení *o*, které vychází především z počtu a typu tvarů použitých pro výpočet souřadnic. Vypočtené souřadnice jsou *o*-krát vloženy do seznamu všech souřadnic. Z tohoto seznamu jsou po dokončení výpočtu souřadnic pro všechny prahy zprůměrováním spočítány finální souřadnice. Jak již bylo zmíněno, prahování může probíhat paralelně, a tak je nutné přidávání do seznamu všech souřadnic ošetřit pomocí mutexu.

V případě, že pro daný práh nebyl detekován dostatek tvarů pro výpočet souřadnic, tak do seznamu žádné souřadnice vloženy nejsou. Pokud svary nebyly detekovány ani pro jeden práh, použijí se výchozí souřadnice (parametr funkce `preprocess_single`), pokud byly zadány.

4.3.3 Optimalizace

Algoritmus detekce svarů byl optimalizován ve dvou oblastech. Nejdříve byly optimalizovány hodnoty používané ve fázi před prahováním. Primárním cílem bylo snížit počet případů, kdy detekce selže. Sekundárním cílem bylo zmenšit chyby výpočtu. Optimalizováno bylo následující:

- zda použít ekvalizaci histogramu či ne.
  - Bylo zjištěno, že ekvalizace histogramu zvyšuje úspěšnost algoritmu.
- Počáteční práh, konečný práh a krok mezi prahy.
  - Jako nejvhodnější se ukázaly být hodnoty již zmíněné v kapitole 4.3.1.
- Velikost jádra pro funkci Gaussian Blur.
  - Nejlepších výsledků bylo dosaženo s jádrem o velikost 11 x 11 px.
- Jakou použít metodu prahování.
  - Nejlepších výsledků bylo dosaženo s metodou `cv2.THRESH_TOZERO_INV`.

Za účelem sběru dat pro optimalizaci bylo prováděno předzpracování 2 944 obrázků, což odpovídá 128 náhodně vybraným obrázkům z každé podmnožiny dat. Postupnou optimalizací se podařilo snížit počet případů selhání detekce ze 44 na 7.

Druhou oblastí optimalizace byly nízkoúrovňové funkce pro výpočet souřadnic svarů. Cílem bylo optimalizovat empiricky zjištěné hodnoty multiplikátorů tak, aby se pro každou funkci snížily chyby e_i a celková chyba E.

Pokud jsou očekávané souřadnice EC vektor (w1x, w1y, w2x, w2y) a skutečné souřadnice vektor AC (w1x, w1y, w2x, w2y), potom jsou chyby e_i rovny $EC_\mathrm{i} - AC_\mathrm{i}$. Celková chyba E je poté vypočtena podle Rovnice 3:

$$E = \sum_{i=1}^{4} e_i^2 \tag{3}$$

Podle velikosti e_i byly hodnoty multiplikátorů buďto zvětšeny, nebo zmenšeny. Celkem proběhlo šest iterací optimalizace. Průměrné hodnoty chyb před optimalizací pro každou nízkoúrovňovou funkci jsou zachycené v Tabulce 3. Hodnoty po optimalizaci jsou v Tabulce 4. Pokud je např. chyba e_1 kladné číslo, znamená to, že vypočtená x-ová souřadnice horního svaru je menší než očekávaná. Je tedy nutné upravit výpočet této souřadnice tak, aby se souřadnice posunula doprava.

Tabulka 3 - Chyby detekce před optimalizací

Funkce	E	e_1	e_2	e_3	e_4
c0_mr0_h1_v1	59829.5	89.8	4.2	98.7	22.1
c0_mr0_h2_v1	48001.7	113.6	3.9	93.5	20.5
c0_mr1_h0_v1	58550.3	4.6	60.1	20.4	50.8
c0_mr1_h1_v0	406100.9	-181.7	-179.2	-181.5	200.2
c0_mr1_h1_v1	68323.1	68.7	29.9	82.5	38.7
c0_mr1_h2_v0	327999.8	165.9	-18.7	139.2	-18.5
c0_mr1_h2_v1	61529.3	69.3	30.1	57.9	35.9
c1_mr0_h0_v1	19463.5	-5.3	-3.9	-7.7	38.7
c1_mr0_h1_v0	282014.2	68.8	64.3	69.4	8.5
c1_mr0_h1_v1	18138.2	21.6	-7.8	16.4	4.1
c1_mr0_h2_v0	255894.9	238.8	-21.6	211.4	9.8
c1_mr0_h2_v1	18404.7	41.8	3.1	29.4	28.4
c1_mr1_h0_v0	572869.1	-380.3	221.5	-382.2	-324.6
c1_mr1_h0_v1	274824.4	342.4	31.9	358.0	35.8
c1_mr1_h1_v0	261437.7	147.7	-11.7	54.3	118.4
c1_mr1_h1_v1	82235.8	52.6	16.0	62.0	31.2
c1_mr1_h2_v0	552659.9	289.5	-141.4	273.6	121.5
c1_mr1_h2_v1	98729.7	69.6	7.7	53.9	37.4
Celkem	57364.5	67.5	7.5	59.4	23.3

Tabulka 4 - Chyby detekce po optimalizaci

Funkce	E	e_1	e_2	e_3	e_4
c0_mr0_h1_v1	46699.1	-8.2	-6.7	0.4	6.4
c0_mr0_h2_v1	33767.5	23.8	-5.7	6.8	4.1
c0_mr1_h0_v1	57427.8	8.2	21.4	26.9	11.8
c0_mr1_h1_v0	98640.6	-19.3	3.0	-16.4	45.6
c0_mr1_h1_v1	49552.6	16.7	-4.6	29.4	3.2
c0_mr1_h2_v0	65024.4	-12.5	-0.8	-60.4	5.2
c0_mr1_h2_v1	61804.3	27.5	-6.8	15.9	-2.4
c1_mr0_h0_v1	11270.2	-10.7	28.9	-16.7	-4.8
c1_mr0_h1_v0	33273.1	-62.3	52.8	-75.9	73.5
c1_mr0_h1_v1	17554.3	-29.6	11.1	-32.7	-14.5
c1_mr0_h2_v0	30449.2	-34.0	-9.6	-47.9	5.5
c1_mr0_h2_v1	16406.2	0.9	13.1	-6.7	-4.4
c1_mr1_h0_v0	151272.4	-10.9	4.1	-44.7	68.3
c1_mr1_h0_v1	48730.9	38.3	-3.6	31.0	47.7
c1_mr1_h1_v0	24231.6	-19.8	-19.9	-61.1	44.8
c1_mr1_h1_v1	75264.3	4.1	-32.5	-10.3	30.7
c1_mr1_h2_v0	20013.3	10.9	-23.6	-35.6	20.5
c1_mr1_h2_v1	114451.5	36.0	-47.3	-5.8	33.8
Celkem	38513.9	5.9	-1.1	-3.2	3.6

Při ruční kontrole předzpracovaných obrázků bylo zjištěno, že v případě jedné podmnožiny algoritmus produkuje nepoužitelné obrázky. Důvodem bylo, že zatímco ve většině případů je díl na fotografii natočen tak, že je buďto vodorovný nebo že vodorovná osa dílu klesá zleva doprava, tak v případě této podmnožiny osa zleva doprava stoupala. Ve funkci `get_coords_from_rects`, která způsobovala největší podíl chyb, byl do výpočtu zohledněn úhel natočení odlesků (který přibližně odpovídá celkovému úhlu natočení dílu), čímž došlo k nápravě.

4.4 Třetí fáze – finální úpravy

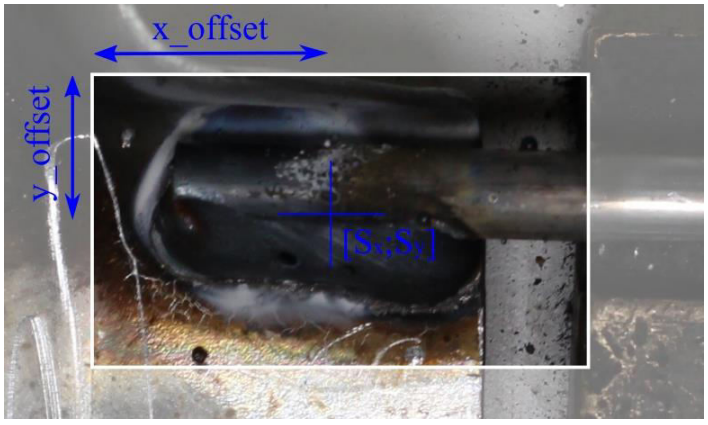
V případě, že detekce nebyla úspěšná, se použijí výchozí souřadnice svarů, pokud byly zadány. Jestliže nebyly zadány, tak předzpracování končí neúspěchem. Výchozí souřadnice byly využívány při přípravě dat pro učení sítě, jak již bylo zmíněno v kapitole 4.1. Při běžném používání sítě výchozí souřadnice zadávány nejsou.

Dalším krokem je kontrola, že se souřadnice středů svarů nacházejí dostatečně daleko od okrajů obrázku, aby mohl být proveden výřez. V případě, že tomu tak není, jsou souřadnice upraveny následovně:

Ukázka kódu: Úprava souřadnic středů svarů

```
w1x = np.clip(coords[0], x_offset, width - x_offset)
w1y = np.clip(coords[1], y_offset, height - y_offset)
w2x = np.clip(coords[2], x_offset, width - x_offset)
w2y = np.clip(coords[3], y_offset, height - y_offset)
```

Funkce `np.clip` zajistí, že x-ové souřadnice budou větší nebo rovny vzdálenosti `x_offset` a menší nebo rovny vzdálenosti `width - x_offset`. `x_offset` je roven rozdílu x-ové souřadnice levého horního rohu výřezu a x-ové souřadnice středu svaru. `width` je šířka originálního obrázku. Ořez y-ových souřadnic funguje obdobně, použita je ale vzdálenost `y_offset`, která je rovna rozdílu y-ové souřadnice levého horního rohu výřezu a y-ové souřadnice středu svaru, a hodnota `height`, což je výška originálního obrázku. Střed výřezu a vzdálenosti `x_offset` a `y_offset` jsou vyznačeny na Obrázku 13.



Obrázek 13 - Výřez okolo středu svaru

Následně je provedeno samotné vyříznutí oblastí okolo svarů, kdy se používají již zmíněné vzdálenosti `x_offset` a `y_offset` pro výpočet levého horního rohu ořezu. Z matice, ve které jsou uloženy pixely obrázku, jsou vybrány jen ty hodnoty, které leží uvnitř oblastí. Tyto oblasti jsou poté spojeny do jednoho pole pomocí funkce `np.append`.

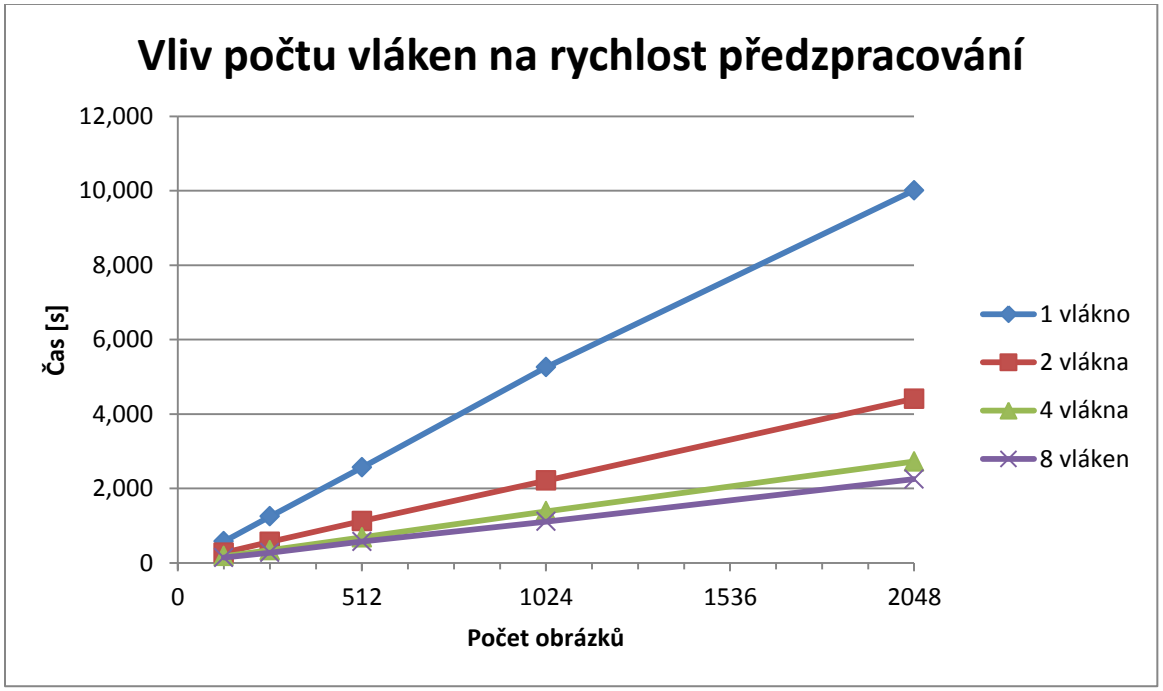
Dále je provedena ekvalizace histogramu, což je jedna z heuristik pro zvýšení přesnosti učení. Ekvalizace se liší podle toho, zda má výstupem být obrázek v odstínech šedi či barevný obrázek. V případě obrázku v odstínech šedi je přímo provedena ekvalizace obrázku pomocí funkce `cv2.equalizeHist`. Pokud má být výstupem barevný obrázek, jsou nejdříve výřezy převedeny z formátu RGB na formát YUV, poté je ekvalizována jasová složka Y a nakonec je obrázek převeden zpět do formátu RGB.

Posledním krokem je úprava velikosti obrázku na velikost zadanou jako argument funkce `preprocess_single`. Pro úpravu velikosti se používá funkce `cv2.resize`.

4.5 Dávkové zpracování

Již v počáteční fázi implementace algoritmu bylo jasné, že vzhledem k velkému počtu a vysokému rozlišení obrázků bude žádané předzpracování provádět paralelně.

Byla tedy vytvořena funkce `preprocess_all`, která jako parametr přijímá mj. to, kolik vláken bude paralelně provádět předzpracování. Následně se načtou cesty ke všem obrázkům na dané cestě a tyto cesty se rozdělí do (přibližně) stejně velkých skupin pomocí funkce `np.array_split`. Pro každou skupinu je vytvořeno jedno vlákno, které provede předzpracování skupiny obrázků ve funkci `preprocess_batch` postupným voláním funkce `preprocess_single` pro každý obrázek ze skupiny.



Obrázek 14 - Vliv počtu vláken na rychlost předzpracování

Zrychlení předzpracování díky paralelnímu výpočtu je možné sledovat na Obrázku 14. Při měření bylo vypnuto logování a vizualizace.

4.6 Logování

Pro usnadnění ladění kódu bylo implementováno logování. Jeho podstatou je vytvoření záznamu o tom, jak probíhalo předzpracování. Ve výčtovém typu `LogLevel` byly definovány tři úrovně logování:

- `None` – logování je vypnuto,
- `Basic` – základní úroveň logování. Loguje se čas zahájení a ukončení předzpracování, finální souřadnice, celkové ohodnocení a případné neobvyklé situace, jako např. selhání detekce, nutnost otočit obrázek apod. Celkové ohodnocení je suma ohodnocení funkcí použitých pro jednotlivé prahy a vypovídá o kvalitě detekce. Pro prahy se vypíšou zjištěné souřadnice a jméno použité funkce.
- `Full` – Kompletní logování. Zapisuje se vše, jako při úrovni `Basic`, navíc se ještě pro jednotlivé prahy zapisuje, jaké tvary byly detekovány a jak probíhalo filtrování.

Pro každý obrázek se vytvoří textový soubor s logem. Nepředpokládá se, že by se logování používalo při klasifikaci obrázků, proto je ve výchozím stavu vypnuto. V případě základní úrovně logování může log vypadat následovně:

```
Předzpracování zahájeno v 2017-09-22 10:33:10.285324
Otáčím obrázek
Hledám souřadnice pro práh 30
Nalezené souřadnice: None. Hodnocení: 0. Metoda: None.
…
Hledám souřadnice pro práh 80
Nalezené souřadnice: [ 1939.37 835.44 1939.37 1473.56]. Hodnocení: 5.
Metoda: c1_mr0_h1_v1.
Hledám souřadnice pro práh 85
Nalezené souřadnice: [ 1938.32   803.07  1938.33  1504.93]. Hodnocení: 8.
Metoda: c1_mr1_h1_v1.
…
Hledám souřadnice pro práh 200
Nalezené souřadnice: [ 1921.51   822.19  1921.51  1470.61]. Hodnocení: 5.
Metoda: c1_mr0_h1_v1.

Iterování přes hodnoty prahování bylo dokončeno. Souřadnice svarů po aplikaci váženého
průměru: (1943, 829, 1922, 1478). Celkové ohodnocení: 182
Předzpracování ukončeno v 2017-09-22 10:33:13.405328
```

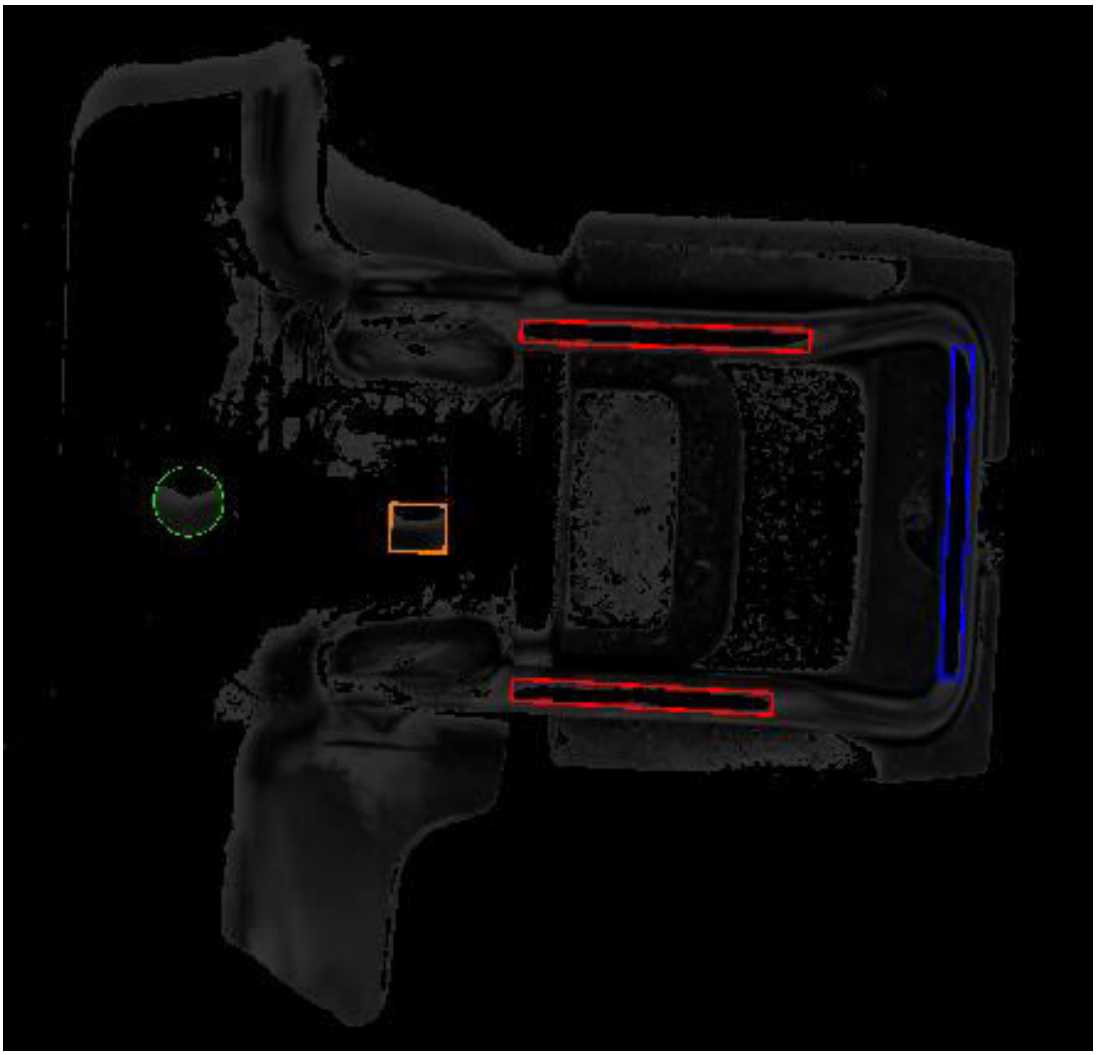
4.7 Vizualizace

Cílem vizualizace bylo ověření, zda algoritmus detekuje jen ty tvary, které by měl detekovat, a že jsou tvary detekovány na správných místech. Vizualizace je postupně prováděna pro každý práh. Možné výstupy vizualizace jsou ilustrovány na Obrázku 15 a Obrázku 16.

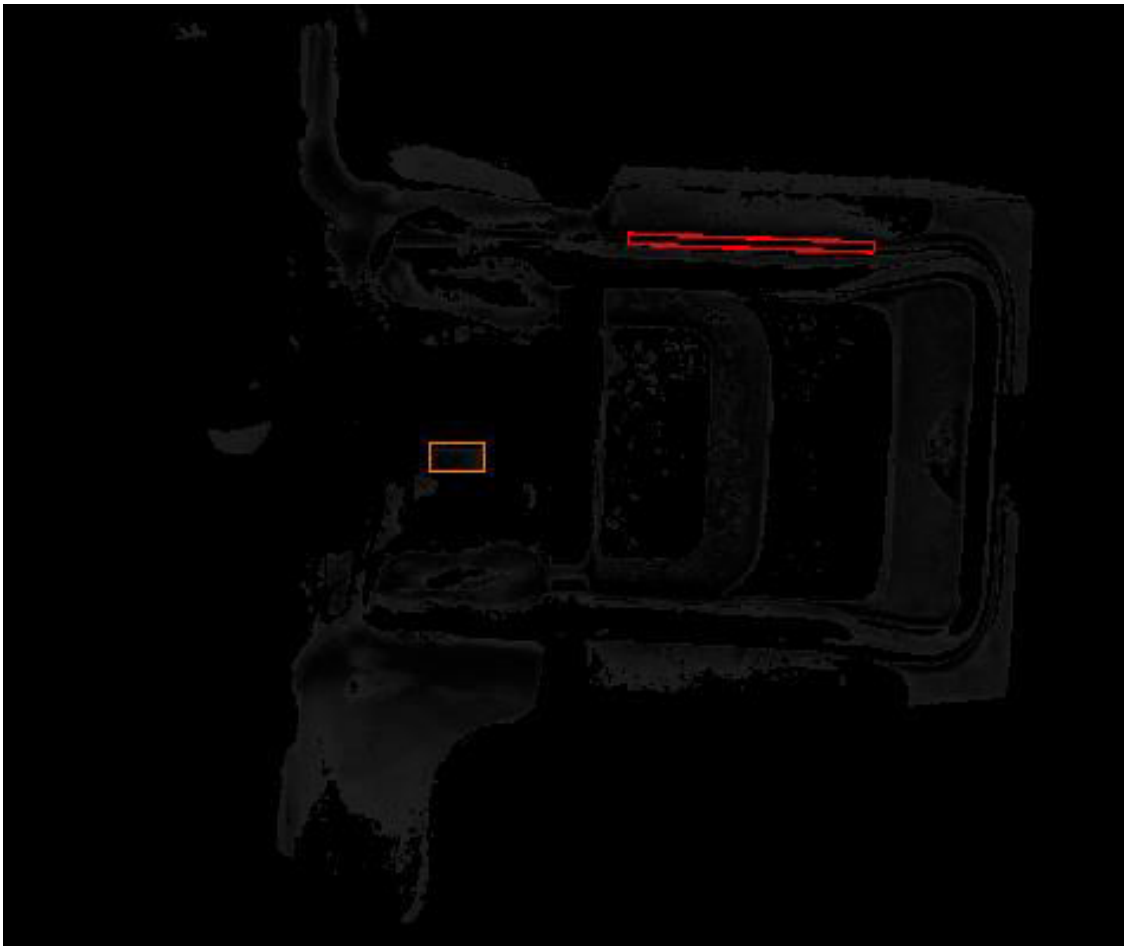
Pro přehlednější vizualizaci bylo použito následující barevné kódování:

- zelená – kruhový výřez,
- oranžová – obdélníkový výřez,
- červená – horizontální odlesky,
- modrá – vertikální odlesk.

Na základě vizualizace byly upravovány některé empiricky zjištěné hodnoty, zejm. ty, které se používají pro klasifikaci tvarů.



Obrázek 15 - Příklad vizualizace, detekovány byly všechny tvary



Obrázek 16 - Příklad vizualizace, detekován byl jen obdélníkový výřez a horní horizontální odlesk

Kresleno je na kopie originálního obrázku. Vizualizace je implementována ve funkci `visualize_contours`. Pro kreslení jsou používány funkce `cv2.drawContour` (odlesky a obdélníkový výřez) a `cv2.circle` (kruhový výřez).

Stejně jako logování je i vizualizace ve výchozím stavu vypnutá.

4.8 Shrnutí

Když bylo dosaženo dostatečné optimalizace, bylo provedeno předzpracování celého datového setu. Z 30 332 obrázků detekce selhala v 29 případech, z toho 5 fotografií bylo nekvalitních (příliš rozmazané, díl byl překryt cizím předmětem nebo na fotografii vůbec nebyl). Tyto nekvalitní fotografie byly z datového setu odstraněny. Při následné ruční kontrole byly 443 obrázky vyhodnoceny jako nepoužitelné. Důvodem bylo buď to, že se svary na obrázku nenacházely vůbec, nebo na něm byly jen zčásti. Úspěšnost algoritmu tedy byla 98,44 %, což bylo považováno za dostatečné.

5 Implementace konvoluční sítě

Konvoluční neuronová síť je v aplikaci reprezentována třídou `CNN`. Tato třída je implementována v Python modulu `cnn`. UML diagram této a souvisejících tříd je zachycen v Příloze D.

5.1 Inicializace

Inicializace je prováděna v konstruktoru (v Pythonu se jedná o metodu `__init__`). Při inicializaci instance třídy `CNN` je nutné zadat:

- seznam jmen tříd, na které budou obrázky klasifikovány,
- šířka a výška obrázku.

Nepovinnými parametry jsou:

- počet kanálů barev obrázku:
  - 1 nebo 3, výchozí je 1, tj. odstíny šedé,
- jméno funkce pro předzpracování:
  - musí se jednat o kompletní jméno funkce (včetně modulů), aby na jeho základě bylo možné funkci importovat a volat.
  - Výchozí hodnota je `None` – v tom případě se použije výchozí funkce pro předzpracování, která pouze načte obrázek (v odstínech šedi nebo barevný, podle toho, co bylo zadáno při vytváření `CNN`) a upraví jeho velikost na takovou, jaká byla zadaná při inicializaci `CNN`.

Kromě zpracování parametrů dojde také k resetu výchozího Tensorflow grafu. Důvodem pro reset je konflikt jmen, která Tensorflow přiřazuje tensorům a operacím. Konflikt jmen vznikal při načítání a vytváření sítě a způsoboval nedefinované chování.

Dojde také k vytvoření instance třídy `Session`, která se používá pro průchod tensorů výpočetním grafem při učení a klasifikaci.

V konstruktoru jsou také inicializovány dočasné hodnoty grafu. Jedná se o tensor vstupů (`_input`) a hodnoty používané při učení – očekávaný výstup (`_expected_output`), pravděpodobnost úpravy vah neuronu při zpětném šíření chyby (`_keep_prob`) a rychlost učení (`_optimizer_learning_rate`). Skutečné hodnoty, které nahradí tyto dočasné, jsou dodány před zahájením epochy učení, resp. při klasifikaci obrázku.

5.2 Definice architektury

Třída `CNN` poskytuje dva způsoby pro definici architektury, a to výchozí nebo vlastní. Pro vytvoření tensorů představujících váhy a biasy se používají statické metody převzaté z oficiálního tutoriálu [24].

5.2.1 Výchozí architektura

Pro vytvoření výchozí architektury slouží metoda `build_default_net`. Tato metoda přidá do sítě vrstvy v takovém pořadí a s takovými parametry, že bude vytvořena architektura odpovídající té, která se ukázala být nejlepší při vyhodnocování architektur (viz kap. 9). V této metodě je také provedena inicializace proměnných učení (viz kap. 5.3.1).

5.2.2 Vlastní architektura

Vlastní architektura se vytváří postupným voláním následujících metod:

- `add_input_layer`
  - Přidá vstupní vrstvu, ve které dojde k převedení obrázku z vektoru na matici a k jeho normalizaci na interval [0; 1].
  - Vstupní vrstva musí být první vrstvou v síti.
- `add_conv_layer`
  - Přidá konvoluční vrstvu. Povinnými parametry jsou počet a velikost filtrů (výška a šířka v pixelech), nepovinnými krok okna konvoluce a metoda pro padding.
  - Konvoluční vrstva nesmí být první vrstvou v síti a nesmí být zařazena za dropout, výstupní nebo hustě propojenou vrstvu. Výstupem těchto vrstev je jednorozměrný tensor, zatímco konvoluční vrstva jako vstup očekává tensor vícerozměrný.
- `add_relu_layer`
  - Bezparametrická metoda, která na výstup z předchozí vrstvy aplikuje funkci ReLU.
  - Kontrolováno je totéž jako u konvolučení vrstvy.
- `add_max_pool_layer`
  - Metoda pro provedení operace Max pool. Povinným parametrem je velikost kroku okna, nepovinným metoda pro padding. Okno je čtverec s délkou strany stejnou, jako je velikost kroku.
  - Kontrolováno je totéž jako u konvolučení vrstvy.
- `add_densely_connected_layer`
  - Přidá hustě propojenou vrstvu. Povinnými parametry jsou počet neuronů vrstvy a aktivační funkce. Pokud byl výstup z předchozí vrstvy vícerozměrný tensor, tak nejdříve dojde k jeho zploštění (tj. převedení na vektor) voláním metody `_adjust_input_shape`.
  - Hustě propojená vrstva nesmí být první vrstvou v síti a nesmí být zařazena za dropout nebo výstupní vrstvu.
- `add_dropout_layer`
  - S pravděpodobností p nastaví hodnotu aktivace neuronu na 0, takže nebudou upraveny jeho váhy a bias při zpětném síření chyby, čímž se sníží riziko přetrénování sítě. Jak již bylo zmíněno, pravděpodobnost p je dočaná

hodnota, konkrétní hodnota je dosazena při učení (např. 0.5 pro trénovací množinu, 1 pro testovací a validační), resp. při klasifikaci. V případě klasifikace se používá hodnota 1, protože není žádané, aby byly váhy neuronu deaktivovány.
    - Kontrolováno je to totéž jako u hustě propojené vrstvy.
- `add_readout_layer`
    - Vrstva, jejíž počet neuronů odpovídá počtu tříd pro klasifikaci. Výstup z jednotlivých neuronů značí míru jistoty, že obrázek patří do dané třídy.
    - Kontrolováno je totéž jako u hustě propojené vrstvy.

Tyto metody obecně pracují tak, že se nejdříve provede kontrola vstupních argumentů a stavu sítě. Pokud kontrola selže, je vyvolána výjimka, jinak se na výstup z předchozí vrstvy aplikuje Tensorflow operace, jejíž výsledek se stane výstupem vrstvy. Dojde také k vytvoření instance třídy `Layer`, která je přidána do seznamu vrstev sítě.

Po přidání výstupní vrstvy je ještě nutné zavolat metodu `init_train_variables`.

5.2.3 Získání informací o vrstvách

K získání informací o vrstvách sítě slouží metoda `get_layers_info`. Tato metoda prochází seznam vrstev (instance třídy `Layer`) a vytváří nový seznam instancí třídy `LayerInfo`, což je třída podobná třídě `Layer`, obsahuje ale pouze informace o jménu, typu, popisu a parametrech vrstvy (tzn., že oproti `Layer` neobsahuje tensory a operace).

5.3 Učení

Podstatou učení sítě je upravit váhy a biasy neuronů tak, aby byla chyba klasifikace co nejmenší. Za tímto účelem se používá algoritmus zvaný Zpětné šíření chyby.

5.3.1 Proměnné učení

Předtím, než bude možné spustit učení sítě, je nutné provést inicializaci proměnných učení. Inicializaci je ale nutné provádět až poté, co je definována architektura sítě, protože proměnné pracují s definicí výstupu z poslední vrstvy.

5.3.1.1 Výpočet chyby

Pro výpočet chyby se používá Tensorflow funkce `softmax_cross_entropy_with_logits`. Tato funkce vrací tensor, jehož prvky odpovídají chybě klasifikace pro každý obrázek dávky. Následně je vypočtena střední hodnota této matice pomocí Tensorflow funkce `reduce_mean`, čímž je získána konečná hodnota chyby.

5.3.1.2 Optimizátor

Tensorflow nabízí řadu optimizátorů (Gradient Descent, Momentum apod.). Stejně jako v oficiálním tutoriálu [24], byl i ve vytvořené aplikaci použit optimizátor `AdamOptimizer`. Úkolem optimizátoru je snížit chybu (viz předchozí odstavec) pomocí

algoritmu Adam. Epocha trénování odpovídá volání metody `optimize` třídy `AdamOptimizer` [28][29][30].

5.3.1.3 Výpočet přesnosti

Přesnost klasifikace se vypočte následovně:

1) mějme tensor `correct_predicion`, jehož velikost odpovídá počtu obrázků v dávce.
2) Pro každý obrázek dávky se porovná index největšího prvku výstupu sítě s indexem hodnoty 1 v one-hot vektoru daného obrázku. Pokud si jsou hodnoty rovny, bude prvku tensoru `correct_predicion` přiřazena hodnota `True`, jinak `False`.
3) Prvky tensoru `correct_predicion` jsou převedeny na desetinná čísla (1.0 pro `True`, 0.0 pro `False`).
4) Výsledná přesnost se vypočítá jako střední hodnota tensoru `correct_predicion`.

Je vhodné poznamenat, že i když byl obrázek klasifikován správně, tak je velice pravděpodobně, že na výstupu byla chyba. Chyba by nenastala jen v případě, že by výstupní vektor měl jen jeden (navíc správný) prvek roven jedné a zbylé prvky by měl rovny nule.

5.3.2 DataProvider

Třída `DataProvider`, implementovaná v Python modulu `data_provider`, slouží pro poskytování dávek obrázků, které se využívají pro trénování, validaci a testování sítě. Nutnost rozdělit obrázky na dávky vychází z omezených hardwarových zdrojů. V ideálním případě by se pro epochu trénování použila celá trénovací množina.

Konstruktor třídy má tyto parametry:

- `paths` – slovník, ve kterém je pro každou třídu klasifikace uložena cesta k obrázkům dané třídy.
- `classes` – seznam tříd obrázků, odpovídající seznamu tříd v `CNN`.
- `channels` – počet kanálů barev. Na základě tohoto parametru se obrázky budou načítat v odstínech šedi nebo ve formátu RGB.
- `image_shape` – na jaké rozlišení má být obrázek upraven, pokud toto rozlišení po načtení nemá.
- `sets_division` – `tuple`, která určuje, jak velká část množiny se má použít pro trénovací, validační a testovací množinu.

V metodě `_load_all` se načtou cesty ke všem obrázkům pro každou třídu. Na základě indexu třídy v seznamu tříd `classes` je pro každý obrázek vytvořen one-hot vektor, který je spárován s cestou k obrázku. Pokud např. obrázek patří do třídy s indexem 2, bude mu přidělen vektor [0;0;1;…;0]. Tyto páry jsou vloženy do seznamu společného pro

všechny třídy obrázků. Tento seznam je v metodě `_split_into_sets` rozdělen na dílčí seznamy na základě parametru `sets_division`. Jsou tedy vytvořeny seznamy pro trénovací, validační a testovací množinu.

Pro každý dílčí seznam je ve slovníku `_indexes` uložen aktuální index. Při vyžádání nové dávky metodou `next_train_batch`, `next_validation_batch` nebo `next_test_batch` dojde k načtení *n* obrázků z dané množiny a posunutí aktuálního indexu o *n*. Seznamy jsou procházeny cyklicky. Při návratu na začátek seznamu je seznam náhodně zamíchán. Pro vytváření dávek slouží metoda `_next_batch`.

Načítání obrázků se provádí v metodě `_load_img_as_vector`. Obrázek je načten, a pokud je to nutné, je upravena jeho velikost. Následně je převeden z matice pixelů na vektor. Seznam párů těchto vektorů a one-hot vektorů tvoří dávky, které jsou vraceny metodami `next_..._batch`, a následně využity pro učení sítě. V ideálním případě by se každý obrázek načetl jen jednou a poté by byl udržován v operační paměti. Z důvodu omezené velikosti paměti je ale nutné obrázky načítat při každém vytváření dávky.

5.3.3 Offline učení

Offline učení je implementováno v metodě `train` třídy `CNN`. Tato třída vrací generátor, jehož jedna iterace odpovídá provedení *n* epoch trénování, kde *n* odpovídá hodnotě parametru `validate_every` metody `train`. Po provedení *n* epoch trénování se provede validace. Pokud byla přesnost validace nedostatečná nebo pokud nebyl proveden minimální počet epoch trénování (parametry `min_validation_accuracy`, resp. `min_train_steps`), vrátí generátor instanci třídy `ValidationResult`. Jinak je učení ukončeno, je provedeno testování a generátor vrátí instanci třídy `TestResult`. Učení je také ukončeno, pokud je počet epoch roven parametru `max_train_steps`.

`ValidationResult` a `TestResult` jsou jednoduché třídy, které pouze obsahují proměnné informující o tom, kolikátá epocha byla dokončena a jaká byla přesnost trénovací a testovací množiny (v případě `TestResult`), resp. validační množiny (v případě `ValidationResult`).

V metodě `train` je používána instance třídy `DataProvider`. Argumenty `paths` a `sets_division` metody `train` jsou předány konstruktoru třídy `DataProvider`. To znamená, že při každém zahájení trénování dojde k různému rozdělení obrázků mezi trénovací, validační a testovací množinu. Velikost dávky `batch_size`, což je další parametr metody `train`, se používá pro získání dávky dané velikosti pomocí jedné z metod `next_..._batch` třídy `DataProvider`.

Dalšími parametry metody `train` jsou `learning_rate` (rychlost učení) a `keep_prob` (pravděpodobnost úpravy vah a biasu neuronu při učení). Tyto hodnoty budou použity k dosazení skutečných hodnot na místo dočasných, viz kapitola 6.1. Vyšší rychlost učení může snížit čas potřebný k učení sítě, může ale také zapříčinit to, že optimálnější nastavení vah a biasů bude „přeskočeno“. `keep_prob` snižuje riziko přetrénování sítě, při zbytečně

nízké hodnotě ale může prodloužit dobu trénování. Je tedy zřejmé, že s oběma parametry je nutné experimentovat. Jako nejlepší se pro problematiku, které se věnuje tato práce, ukázaly být hodnoty 10^{-5} pro rychlost učení a 0.5 pro `keep_prob`.

Parametr `validation_batches` určuje, kolik dávek se má použít pro výpočet přesnosti validace. Pokud by např. byla `batch_size` 50 a `validation_batches` 10, tak se přesnost vypočítá na základě klasifikace 500 obrázků z validační množiny.

Přesnost testování se počítá na základě správnosti klasifikace pro všechny obrázky z testovací množiny. Při testování i validaci se používá metoda `_get_accuracy_from_multiple _batches`. Tato metoda má jako parametry počet obrázků, které mají být klasifikovány (`to_run`), velikost dávky (`batch_size`) a funkci, která se využívá pro získání dávky (`next_batch_op`). Je implementována tak, že počítá přesnost klasifikace pro tolik dávek, dokud není počet klasifikovaných obrázků roven argumentu `to_run`. Celková přesnost je vypočítána jako průměr přesností dávek.

Posledním dosud nezmíněným parametrem metody `train` je parametr `reset_variables`. Pokud je jeho hodnota `False`, tak nedojde k resetování vah a biasů, čímž by bylo možné navázat na předchozí trénování. Výsledky ale budou zavádějící, protože obrázky, které byly původně v trénovací množině, by nyní mohly být v testovací nebo validační množině. Síť by je pravděpodobně klasifikovala správně, a tak by přesnost testování a validace byla vyšší, než přesnost při praktickém používání sítě po ukončení procesu učení.

5.3.4 Online učení

Zatímco při offline učení se váhy a biasy upravují po klasifikaci dávky obrázků, při online učení se upraví po klasifikaci jednoho obrázku.

Online učení je implementováno v metodě `train_single`. Tato metoda očekává jako parametr obrázek (ve formě matice), jméno třídy obrázku a rychlost učení. Obdobně jako v `DataProvider` se i v této metodě provede převedení obrázku na vektor a vytvoření one-hot vektoru na základě indexu třídy v seznamu tříd.

Poté se provede jedna epocha trénování, kde vstup je tvořen seznamem, jehož jediným prvkem je obrázek převedený na vektor, a očekávaným výstupem je seznam, jehož jediným prvkem je zmíněný one-hot vektor.

5.4 Klasifikace

Pro klasifikaci obrázku slouží metoda `classify`. Tato metoda má jediný parametr, a to obrázek v podobě matice, který má být klasifikován. Tento obrázek je následně převeden na vektor, který se použije jako vstup sítě. Obrázek poté projde sítí. Index třídy odpovídá indexu největšího prvku výstupu. K zjištění indexu největšího prvku slouží Tensorflow funkce `argmax`. Jméno třídy se pak jednoduše získá ze seznamu jmen tříd (index seznamu odpovídá výsledku funkce `argmax`).

Pomocí Tensorflow funkce `softmax` se upraví výstup tak, že součet všech jeho prvků je roven jedné. Pak je možné použít hodnotu `vystup[index]` jako pravděpodobnost, že obrázek patří do dané třídy.

Funkce `classify` vrací `tuple`, jejímž prvním prvkem je jméno třídy a druhým prvkem pravděpodobnost, že obrázek skutečně patří do této třídy.

5.5 Ukládání a načítání

Ukládání CNN je implementováno v metodě `save`. Tato metoda očekává jako parametr jméno sítě a adresář, do kterého budou soubory sítě uloženy.

Ukládání probíhá ve dvou krocích. Nejdříve jsou do textového souboru s příponou cnn zapsána metadata o síti (rozlišení obrázku, počet kanálu barev, jména tříd pro klasifikaci a jméno funkce pro klasifikaci) a o vrstvách sítě (typ vrstvy a případně další parametry). Druhým krokem je uložení sezení (`Session`) pomocí Tensorflow třídy `Saver`.

Pro načítání slouží statická metoda `load`. Stejně jako metoda `save`, i `load` očekává jako parametr jméno sítě a adresář, ve kterém je síť uložena.

Nejdříve dojde k vytvoření instance třídy `CNN` na základě metadat načtených ze souboru cnn. Poté jsou ve statické metodě `_load_layers` do sítě přidávány vrstvy na základě metadat o vrstvách, získaných opět ze souboru cnn. Jako poslední je načtena `Session`. V případě, že načtení proběhlo v pořádku, je vrácena instance třídy `CNN`.

Vzhledem k tomu, že třída `Saver` nepodporuje ukládání a načítání, pokud cesta k souboru obsahuje českou diakritiku, je v metodě `_check_paths_encoding` kontrolováno, že cesta obsahuje pouze ASCII znaky. Kdyby cesta obsahovala znaky nepatřící do ASCII, tak metoda vyvolá výjimku `ValueError` a ukládání či načítání selže.

6 Grafické uživatelské rozhraní

Moduly, ve kterých je definováno rozložení GUI elementů a ve kterých je implementováno chování GUI, se nacházejí v balíčku `gui`. Ve stejnojmenném modulu je definována třída `MainTabbedPanelApp`, odvozená od Kivy třídy `App`. V modulu `main` je vytvořena instance této třídy a následně zavolána její metoda `run`, čímž dojde ke spuštění aplikace.

6.1 Struktura

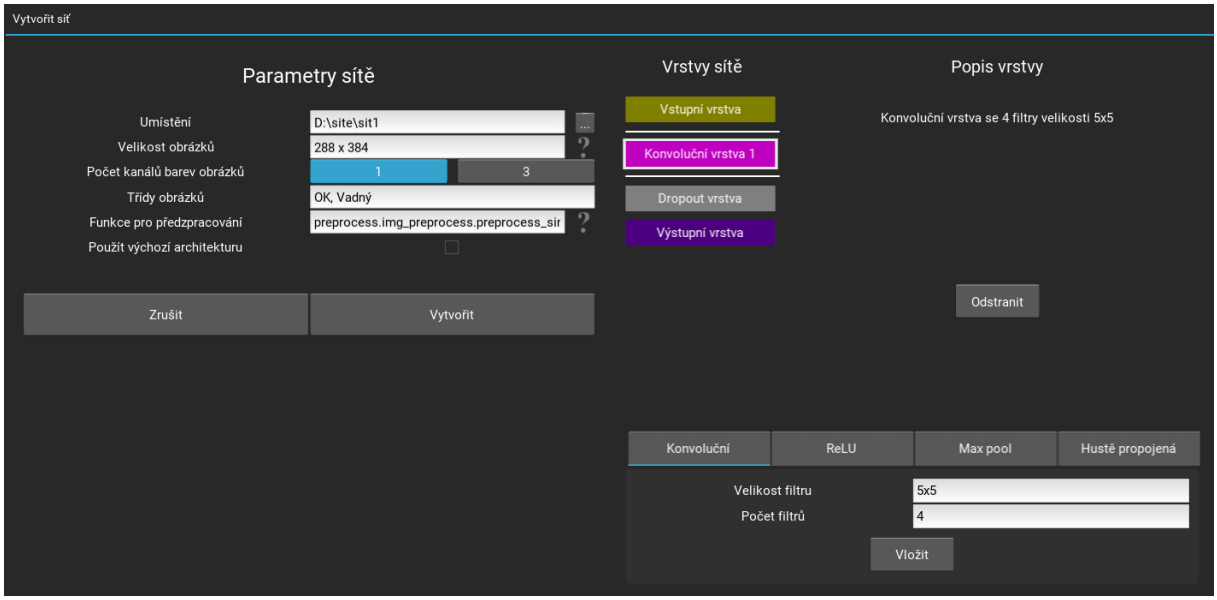
GUI je členěno do tří záložek – Definice sítě, Učení sítě a Použití sítě. Tyto záložky jsou součástí kořenového elementu GUI. Tento kořenový element je instancí třídy `MainTabbedPanel`. Instance třídy `MainTabbedPanel` si udržuje referenci na aktuálně používanou síť (tj. instanci třídy `CNN`). V této třídě jsou také definovány metody pro zobrazení dialogových oken (načtení, uložení a varovná hláška).

6.1.1 Definice sítě

V záložce Definice sítě (viz Obrázek 17) je možné získat informace o aktuálně používané síti a jejích vrstvách. Dále je zde možné provést uložení, načtení nebo vytvoření nové sítě.

Obrázek 17 - Záložka Definice sítě

Rozložení GUI elementů je definováno v souboru NetInfo.kv a chování je implementováno ve třídě `NetInfo`. Součástí této záložky je instance třídy `LayersInfo`, což je GUI element sloužící pro výpis vrstev sítě ve formě seznamu tlačítek. Kliknutím na tlačítko se zobrazí popis dané vrstvy. Stejný typ elementu je použit i v dialogovém okně pro vytváření sítě, pokud uživatel zvolil, že nechce využít výchozí architekturu (viz Obrázek 18).

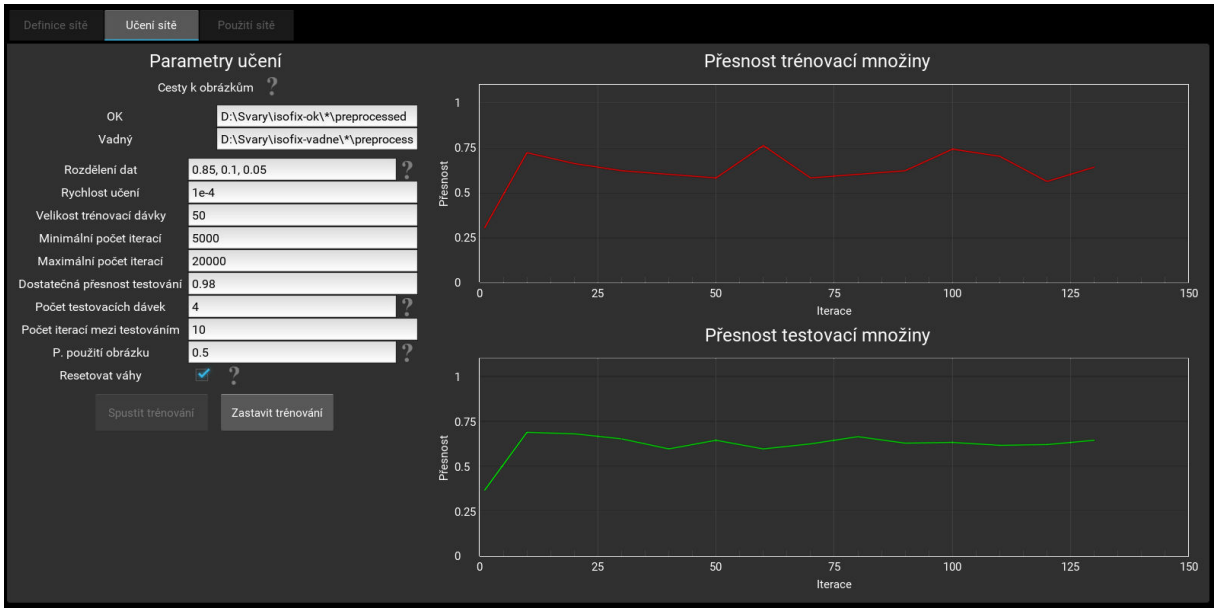


Obrázek 18 - Dialogové okno pro vytvoření sítě

Po vytvoření sítě se nově vytvořená síť nastaví jako aktuální (v kořenovém GUI elementu). Je tedy možné ihned zahájit učení.

6.1.2 Učení sítě

Záložka učení sítě (viz Obrázek 19) je instancí třídy `NetTrain`. Rozložení elementů je definováno v souboru NetTrain.kv.



Obrázek 19 - Záložka Učení sítě

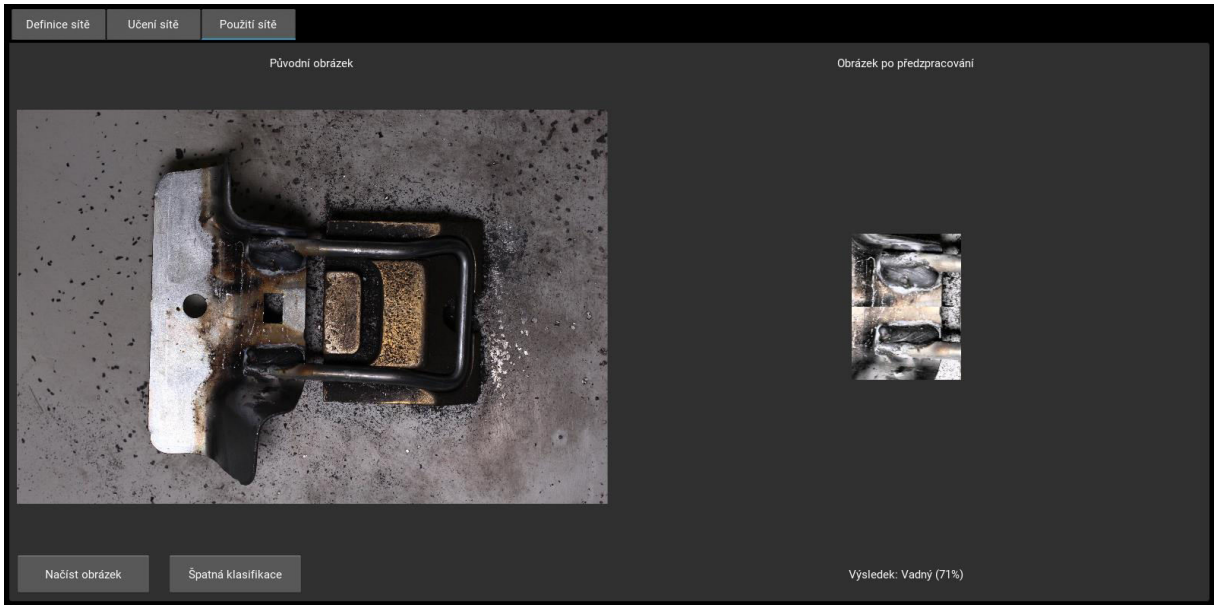
Uživatel zadá cestu k obrázkům pro každou třídu. Může při tom použít zástupné symboly „?“ (libovolný znak) nebo „*“ (libovolná skupina znaků). Tímto je mj. možné načíst obrázky ze sesterských složek.

Pro zbylé parametry učení (které byly blíže popsány v kapitole 5.3) jsou definovány výchozí hodnoty, nicméně uživatel má možnost zadat hodnoty vlastní. Po spuštění učení se začne na grafy v pravé části záložky vykreslovat přesnost klasifikace jak pro trénovací, tak pro validační množinu. Během učení jsou upravovány osy grafů tak, aby bylo vždy možné sledovat celý průběh učení.

Po dokončení učení se zobrazí dialogové okno informující o počtu provedených epoch trénování a o přesnosti testování.

6.1.3 Použití sítě

Záložka Použití sítě (viz Obrázek 20) slouží ke klasifikaci obrázků. Její design je specifikován v souboru NetUse.kv a implementace chování je provedena ve třídě `NetUse`.



Obrázek 20 - Záložka Použití sítě

Uživatel má kromě provedení klasifikace obrázku také možnost označit klasifikaci za chybnou. Po kliknutí na tlačítko Špatná klasifikace se zobrazí dialogové okno, ve kterém uživatel vybere ze seznamu správnou třídu a zadá rychlost učení. Po potvrzení dojde k online učení sítě (více viz kap. 5.3.4).

Nečekaně problémovým se ukázalo být zobrazení předzpracovaného obrázku v GUI. Osy datového pole s pixely musely být navzájem zaměněny, čímž vznikl obrázek otočený o 90 ° ve směru hodinových ručiček, převrácený zrcadlově podle vertikální osy. Takto upravený obrázek byl otočen o 90 ° proti směru hodinových ručiček. Z tohoto obrázku již bylo možné vytvořit Kivy texturu, která vypadá stejně jako předzpracovaný obrázek a kterou je možné v GUI zobrazit.

7 Konzolová aplikace

K hlavní GUI aplikaci byla vytvořena doprovodná konzolová aplikace. Zatímco v GUI aplikaci je možné provádět i definování a učení sítě, konzolová aplikace slouží pouze pro klasifikaci obrázků. Na Obrázek 21 je ilustrováno použití aplikace v prostředí Powershell.

Obrázek 21 - Konzolová aplikace

Konzolová aplikace klade větší nároky na uživatele, co se týče znalostí ovládání PC (uživatel musí umět používat konzoli), nabízí ale zvýšení pracovní výkonnosti při klasifikaci zejména většího počtu obrázků najednou.

Aplikace také umožňuje automatizaci klasifikace. Např. by mohlo být možné pořizovat fotografie dílů na výrobním páse, tyto fotografie ihned klasifikovat a v případě, že díl bude označen jako vadný, jej stáhnout z výroby.

7.1 Implementace

Konzolová aplikace je implementována jako Python skript `source/terminal.py`. Skript obsahuje následující funkce:

- `main` – provede parsování uživatelem zadaných argumentů (pomocí třídy `ArgumentParser`) a volá hlavní metodu `run`,
- `run` – importuje funkci pro předzpracování obrazových dat a podle zvoleného režimu volá funkci pro klasifikaci jednoho obrázku nebo dávky obrázků,
- `classify_single` – provede předzpracování obrázku a tento obrázek použije jako vstup do CNN. Vrátí výstup metody `classify` třídy `CNN`,
- `classify_batch` – načte všechny obrázky ve formátu JPG na zadané cestě, postupně je předzpracovává a klasifikuje (voláním metody `classify_single`) a na konzoli postupně vypisuje výsledky klasifikace,
- `load_net` – načte síť (vytvořenou v GUI aplikaci) a vrátí ji jako instanci třídy `CNN`. Tato instance je používána metodami `classify_single` a `classify_batch`.

V případě, že došlo k chybě, je vypsána chybová hláška. Jsou definovány tři chybové stavy:

- `CNN_LOAD_ERROR` – chyba při načítání sítě (neplatná cesta či poškozený soubor),
- `IMG_LOAD_ERROR` – chyba při načítání obrázku (nepodporovaný formát, neplatná cesta apod.),
- `CLASSIFICATION_ERROR` – chyba při klasifikaci (selhání předzpracování nebo neočekávaná výjimka při průchodu tensoru výpočetním grafem).

7.2 Použití

V případě, že uživatel chce klasifikovat pouze jeden obrázek, vypadá syntax příkazu následovně:

python <cesta_k_projektu>\terminal.py <cesta_k_obrazku> <cesta_k_siti>

Např. po zadání příkazu *python terminal.py D:\Svary\terminal_test\ok\IMG_0011.JPG D:\site\default\default.cnn* může být vypsáno *OK (99.15 %).*

Pro klasifikaci dávky obrázků se místo cesty k jednomu obrázku zadá cesta k adresáři s obrázky (či k adresářům, pokud se použije zástupný symbol „?“ nebo „*“) a použije se přepínač --batch:

python <cesta_k_projektu>\terminal.py --batch <cesta_k_obrazkum> <cesta_k_siti>

V případě, že by uživatel chtěl provést klasifikaci všech obrázků v podadresářích adresáře D:\Svary\terminal_test, by vypadal příkaz takto:

python terminal.py --batch D:\Svary\terminal_test\ D:\site\default\default.cnn*

A na konzoli by mohlo být vypsáno např.:

```
Načítám síť ...
D:\Svary\terminal_test\ok\IMG_0011.JPG:    OK (87.30%)
D:\Svary\terminal_test\ok\IMG_0100.JPG:    OK (99.99%)
D:\Svary\terminal_test\ok\IMG_2420.JPG:    OK (99.82%)
D:\Svary\terminal_test\vadne\IMG_0193.JPG:        Vadný (99.89%)
D:\Svary\terminal_test\vadne\IMG_0480.JPG:        Vadný (100.00%)
D:\Svary\terminal_test\vadne\IMG_0779.JPG:        Vadný (99.57%)
```

Třída ArgumentParser přidává možnost vypsat nápovědu, a to použitím přepínače -h. Zadáním příkazu *python terminal.py _ _ -h* dojde k výpisu následujícího:

```
usage: terminal.py [-h] [--batch] cesta sit
Klasifikace obrázku (nebo dávky obrázků) pomocí CNN přes terminál
positional arguments:
      cesta    Cesta k obrázku nebo k adresáři s obrázky. Je možné použít
               zástupné symboly ? a *
      sit      Cesta k síti (soubor .cnn)
optional arguments:
  -h, --help  show this help message and exit
  --batch     Provést klasifikaci pro dávku obrázků (pokud tento přepínač není
              zadán, je parametr cesta interpretován jako cesta k jednomu
              obrázku, pro který se provede klasifikace)
```

8 Testování

Aplikace byly testovány dvěma způsoby. Prvním z nich byly Unit testy a druhým byly testovací scénáře.

8.1 Unit testy

Unit testy jsou implementovány v modulech umístěných v balíčku `test`. Celkem bylo vytvořeno 31 testů, které byly rozděleny do pěti kategorií. Každé kategorii odpovídá jeden modul:

- `cnn_tests` – testy metod třídy `CNN`. Zahrnují vytvoření instance, použití výchozí architektury, vytvoření vlastní architektury, online a offline učení, ukládání a načítání.
- `data_provider_tests` – testy třídy `DataProvider`. Testována je inicializace a správnost rozdělení obrázků na trénovací, validační a testovací množinu.
- `preprocess_tests` – testy funkcí, které se podílejí na předzpracování obrazových dat. Je testováno filtrování tvarů, výpočet souřadnic pro různé kombinace detekovaných tvarů, průměrování souřadnic, úprava souřadnic a dávkové zpracování.
- `shapes_tests` – testy tříd `Circle` a `Rectangle`, tj. tříd, které reprezentují kruhový, resp. obdélníkový výřez. Mj. je testováno spojování obdélníků.
- `terminal_tests` – testy konzolové aplikace. Je testováno, že v případě zadání správných argumentů bude provedena klasifikace, a že v případě špatných argumentů bude uživatel informován o tom, kde nastala chyba.

8.2 Testovací scénáře

Testovací scénáře mají podobu sešitu aplikace Microsoft Excel. Jeden test poté odpovídá jednomu listu sešitu. Zamýšlené použití je takové, že tester vyplní datum testu a svoje jméno do předpřipravených polí. Poté bude provádět jednotlivé kroky testu a pro každý krok zapíše, zda skutečný výsledek odpovídal očekávanému, případně jaká nastala chyba. Seznam testů je uveden v Tabulce 5.

Tabulka 5 - Testovací scénáře

Kód	Popis
TS01	Ukládání a načítaní Konvoluční neuronové sítě
TS02	Vytvoření, trénování a použití CNN s výchozí architekturou
TS03	Vytvoření, trénování a použití CNN s uživatelem definovanou architekturou
TS04	Komplexní test, pokrývající veškeré očekávané využití aplikace

Testovací scénáře jsou součástí přiloženého DVD.

9 Výběr vhodné architektury

Poté, co byla dokončena implementace aplikací, bylo překročeno k hledání vhodné architektury sítě. Cílem bylo najít co nejlepší architekturu pro výchozí síť, tedy síť, která bude dodávaná spolu s aplikacemi.

Při volbě architektury bylo nutné brát v potaz jak omezené HW prostředky (zejm. 2 GB paměti na grafické kartě), tak časovou náročnost učení (u běžně hlubokých sítí[8] byla rychlost učení cca 2 500 iterací za hodinu).

Vyhodnocování probíhalo nejdříve přímo v GUI aplikaci, později byl vytvořen skript `arch_eval.py`, čímž došlo k částečné automatizaci. Celkem bylo vyhodnoceno 43 architektur. Výsledky se nachází v sešitě MS Excel, který je součástí přiloženého DVD.

9.1 Vyhodnocované architektury

Při vyhodnocování architektur byly z počátku použity následující parametry učení:

- rozdělení dat: 85 % trénovací, 10 % validační, 5 % testovací,
- velikost dávky: 50,
- minimální počet iterací: 5 000,
- maximální počet iterací: 20 000,
- dostatečné přesnost testování: 98 %,
- počet testovacích dávek: 4,
- počet iterací mezi testováním: 10,
- pravděpodobnost použití obrázku: 50 %.

Architektury sítí se lišily v následujícím:

- rozměry obrázku (288 x 384 px, 144 x 192 px nebo 72 x 96 px),
- počet kanálů barev obrázku (jeden nebo tři),
- velikost a počet filtrů v konvolučních vrstvách,
- velikost kroku v max pool vrstvách,
- velikost a aktivační funkce hustě propojených vrstev,
- počet a typ vrstev.

Po vyhodnocení osmi architektur bylo rozhodnuto, že je učení ukončeno příliš brzy. Byl tedy zvýšen minimální počet iterací na 10 000, počet testovacích dávek na 12 a dostatečná přesnost testování na 99,5 %. Ale ani poté nebyla nalezena jednoznačně nejlepší architektura.

Po vyhodnocení 43 sítí bylo rozhodnuto hledání z časových důvodů ukončit. Úspěšnost testování se pohybovala mezi 90 až 96 %. Pouze dvěma sítím se podařilo překročit

[8] Např. sítě v Tabulce 6

úspěšnost testování 96 %. Architektura těchto sítí je uvedena v Tabulce 6. Sítě se liší pouze v tom, jaké aktivační funkce jsou použity v hustě propojených vrstvách.

Tabulka 6 - Architektury sítí s nejlepšími výsledky testování

Jméno sítě	Default_grey_fc_relu	Default_grey_fc_sigmoid
Velikost obrázků:	288 * 384	288 * 384
Rozdělení dat:	0.85, 0.1, 0.05	0.85, 0.1, 0.05
Rychlost učení:	0.0001	0.0001
Velikost dávky:	50	50
Min iterací:	5 000	5 000
Max iterací:	20 000	20 000
Dostatečná přesnost:	0.995	0.995
Počet validačních dávek:	12	12
Validate every:	10	10
Keep prob:	0.5	0.5
Počet kanálů barev	1	1
Vrstvy	Konv(4x4, 4)	Konv(4x4, 4)
	ReLU	ReLU
	MaxPool(2)	MaxPool(2)
	Konv(4x4, 16)	Konv(4x4, 16)
	ReLU	ReLU
	MaxPool(2)	MaxPool(2)
	Konv(4x4, 64)	Konv(4x4, 64)
	ReLU	ReLU
	MaxPool(6)	MaxPool(6)
	FC(1024, ReLU)	FC(1024, Sigmoid)
	FC(1024, ReLU)	FC(1024, Sigmoid)

Pro každou kombinaci kvalita svaru – den pořízení fotografie bylo vybráno deset obrázků (celkem 230), pomocí kterých se testovaly nejúspěšnější sítě. Výsledky se nachází v Tabulce 7. Jedná se o spojení dvou kontingenčních tabulek (jedna pro každou síť). Klíčová metrika, na základě které byla vybrána nejlepší síť, je informovanost *J* (angl. informedness). Jedná se o přibližnou pravděpodobnost, že síť provede správnou klasifikaci. Vypočítá se na základě Rovnice 4 [32]:

$$SE = \frac{\text{skutečná pozitiva}}{\text{skutečná pozitiva} + \text{falešná negativa}}$$

$$SP = \frac{\text{skutečná negativa}}{\text{skutečná negativa} + \text{falešná pozitiva}}$$

$$J = SE + SP - 1 \tag{4}$$

kde *SE* je senzitivita a *SP* je specificita [32].

Tabulka 7 – Výsledky testování dvou nejlepších sítí

		Skutečné hodnoty			
		Default_grey_fc_relu		**Default_grey_fc_sigmoid**	
		OK	Vadný	OK	Vadný
Očekávané hodnoty	OK	99	3	99	3
	Vadný	1	129	3	127
	Senzitivita	0.9706		0.9706	
	Specificita	0.9923		0.9769	
	Informovanost	**0.9629**		**0.9475**	

Vyšší informovanost má síť Default_grey_fc_relu, a proto byla tato síť zvolena jako výchozí. Za povšimnutí stojí fakt, že obě sítě měly stejnou senzitivitu, ale různou specificitu.

Závěr

Cíle práce se podařilo splnit. Byly vytvořeny dvě aplikace pro detekci výrobních vad, z nichž jedna disponuje grafickým uživatelským rozhraním a druhá je pouze konzolová. Je možné je snadno využít ve výrobním procesu (zejména kvůli konzolové aplikaci). Aplikace provádějí rychlou a spolehlivou detekci výrobních vad. Spolehlivost vychází z kvalitně implementovaného procesu učení a z robustního algoritmu pro předzpracování obrazových dat.

GUI aplikace umožňuje uživateli definovat vlastní architekturu sítě a další parametry, jako například třídy obrázků nebo počet kanálů barev. Díky tomu jsou aplikace univerzálně použitelné. Významnou výhodou oproti manuální kontrole výrobků je, že odstraňují chyby způsobené lidským faktorem a že jsou konstantní v čase.

V teoretické části byla stručně popsána relevantní témata z oblasti umělé inteligence. Po jejím prostudování bude i čtenář dříve neseznámený s touto problematikou schopen porozumět kapitolám z praktické části. Seznámení čtenáře s problematikou konvolučních neuronových sítí bylo dílčím cílem této práce.

Vybrané softwarové technologie a zvolená metodika vývoje se osvědčily. Psaní i čtení kódu v jazyce Python je (alespoň v případě autora práce) snadnější než například v jazycích založených na jazyku C. Knihovny OpenCV a Tensorflow poskytly veškeré potřebné funkce pro zpracování obrazu, respektive pro implementaci CNN. Pomocí knihovny Kivy bylo vytvořeno vizuálně atraktivní GUI. Díky prototypování bylo možné do aplikací přidávat nové funkcionality a přitom mít kontrolu nad tím, že jsou postupně splňovány požadavky a že nedochází k odklonění od cesty k cíli.

Na základě algoritmu pro detekci svarů bylo úspěšně předzpracováno více než 98 % obrázků z celé datové množiny. Díky paralelnímu zpracovávání obrázků je algoritmus poměrně výkonný. Je také dostatečně robustní na to, aby svary detekoval bez ohledu na pozici dílu na fotografii. Díl může být také mírně natočen. Algoritmus dokáže předzpracovat obrázky s větším či menším rozlišením, než jaké měly ty z původní datové množiny.

Implementace třídy pro CNN se obešla bez větších potíží. Třída obsahuje funkcionality potřebné pro definici, učení a použití konvoluční neuronové sítě. Zajímavě je řešena metoda pro učení sítě. Vzhledem k tomu, že tato metoda vrací generátor, jehož produkty odpovídají výsledkům validace, resp. testování, lze snadno provést vizualizaci učení. Pokud by metoda pracovala klasicky, tj. ukládala by výsledky do datové struktury, ze které by byly čteny z GUI, byla by implementace značně složitější, zejména co se týče synchronizace vláken.

Testování aplikací (prováděné jak pomocí Unit testů, tak testovacích scénářů) bylo velice prospěšné, jednak proto, že během tvorby testů byly objeveny chyby v kódu, a také proto, že po provedení změn v kódu bylo možné ověřit, že aplikace stále fungují tak, jak by měly.

Hledání vhodné architektury byl časově velice náročný proces. Bylo by zajímavé provést učení sítě na PC s výkonnější grafickou kartou a proces ještě více zautomatizovat. Přesto se podařilo vytrénovat síť s vysokou přesností (96.62 %) a informovaností (0.9629), které slouží jako výchozí síť.

Aplikace by bylo vhodné v budoucnu dále rozvíjet. Efektivnější zapojení do procesu výroby by mohlo být dosaženo migrací z PC do embedded zařízení, založeného například na NVIDIA Jetson[9]. Algoritmus pro předzpracování obrazových dat by mohl být zrychlen tím, že by se v něm využívaly funkce knihovny Tensorflow. Lepších výsledků při učení by mohlo být dosaženo augmentací dat. Ta obvykle spočívá v zrcadlovém otočení obrázku, použití pouze výřezu z obrázku, otočení obrázku či úpravě intenzity pixelů přičtením náhodné složky. Ale vzhledem k tomu, jak relativně velká byla datová množina, bylo rozhodnuto tyto úpravy neprovádět.

[9] http://www.nvidia.com/object/embedded-systems.html

Literatura

[1] CHALUPNÍK, Vitalij. Biologické algoritmy (4) - Neuronové sítě. *Root.cz* [online]. Praha, 2018, 25. 4. 2012 [cit. 2017-09-16]. Dostupné z: https://www.root.cz/clanky/biologicke-algoritmy-4-neuronove-site/

[2] VOLNÁ, Eva. Neuronové sítě 1. *Ostravská univerzita* [online]. Ostrava: Ostravská univerzita v Ostravě, 2008, 2008 [cit. 2017-09-16]. Dostupné z: http://www1.osu.cz/~volna/Neuronove_site_skripta.pdf

[3] Deep Learning v prostředí MATLAB. *Sciencemag* [online]. Praha: Nitemedia, © 2018, 26. 2. 2017 [cit. 2017-09-16]. Dostupné z: http://sciencemag.cz/deep-learning-v-prostredi-matlab/

[4] DESHPANDE, Adit. A Beginner's Guide To Understanding Convolutional Neural Networks. *GitHub Pages* [online]. San Francisco: GitHub, © 2018, 20. 7. 2016 [cit. 2017-09-17]. Dostupné z: https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/

[5] FUKUSHIMA, Kunihiko. Neocognitron. *Scholarpedia* [online]. Scholarpedia, 2005, 19. 1. 2007 [cit. 2017-09-18]. Dostupné z: https://tinyurl.com/lmvmyg5

[6] RUSSAKOVSKY., Olga. et al. Large Scale Visual Recognition Challenge 2012. *ImageNet* [online]. ©2012 [cit. 2017-09-18]. Dostupné z: http://www.image-net.org/challenges/LSVRC/2012/

[7] HE, Kaiming, Xiangyu ZHANG, Shaoqing REN a Jian SUN. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *2015 IEEE International Conference on Computer Vision (ICCV)*. IEEE, 2015, 2015, 1026-1034. DOI: 10.1109/ICCV.2015.123. ISBN 978-1-4673-8391-2. Dostupné také z: http://ieeexplore.ieee.org/document/7410480/

[8] Cloud TPUs - ML accelerators for Tensorflow. *Google Cloud Platform* [online]. 2017, [2017] [cit. 2017-09-20]. Dostupné z: https://cloud.google.com/tpu/

[9] Y'BARBO, Doug. Why must a nonlinear activation function be used in a backpropagation neural network? *Stack Overflow* [online]. New York: Stack Exchange, © 2018, 20. 3. 2012 [cit. 2017-09-21]. Dostupné z: https://stackoverflow.com/questions/9782071/why-must-a-nonlinear-activation-function-be-used-in-a-backpropagation-neural-net/9783865#9783865

[10] 14 Design Patterns To Improve Your Convolutional Neural Networks. *Topbots* [online]. Medium, 2017, 22. 3. [2017] [cit. 2017-09-21]. Dostupné z: https://tinyurl.com/ya3b6wu7

[11] GOUY, Isaac. Python 3 programs versus C++ g++. *The Computer Language Benchmarks Game* [online]. [2017] [cit. 2017-09-22]. Dostupné z: http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=python3&lang2=gpp

[12] MORDVINTSEV, Alexander a Abid RAHMAN K. Introduction to OpenCV-Python Tutorials. *OpenCV library* [online]. OpenCV, © 2018, 2013 [cit. 2017-09-23]. Dostupné z: https://docs.opencv.org/3.3.0/d0/de3/tutorial_py_intro.html

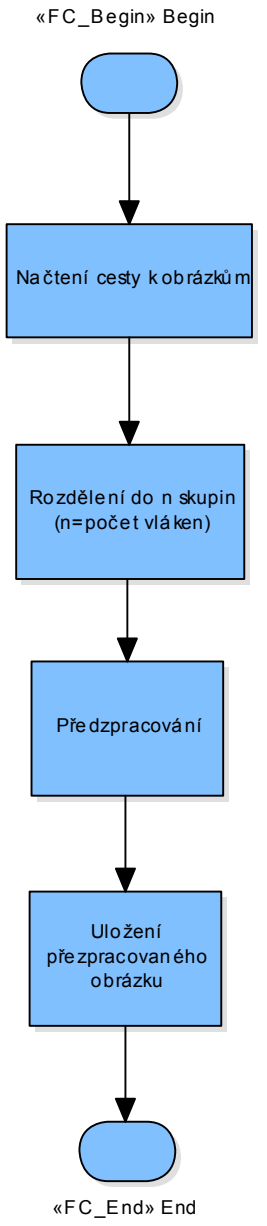
[13] TRAN, Andrew. Accessing OpenCV CUDA Functions from Python (No PyCUDA). *Stack Overflow* [online]. New York: Stack Exchange, © 2018, 9. 2. 2017 [cit. 2017-09-23]. Dostupné z: https://tinyurl.com/y9j9smdv

[14] ROSEBROCK, Adrian. My Top 9 Favorite Python Libraries for Building Image Search Engines. *Pyimagesearch* [online]. © 2018 [cit. 2017-09-24]. Dostupné z: https://www.pyimagesearch.com/2014/01/12/my-top-9-favorite-python-libraries-for-building-image-search-engines/

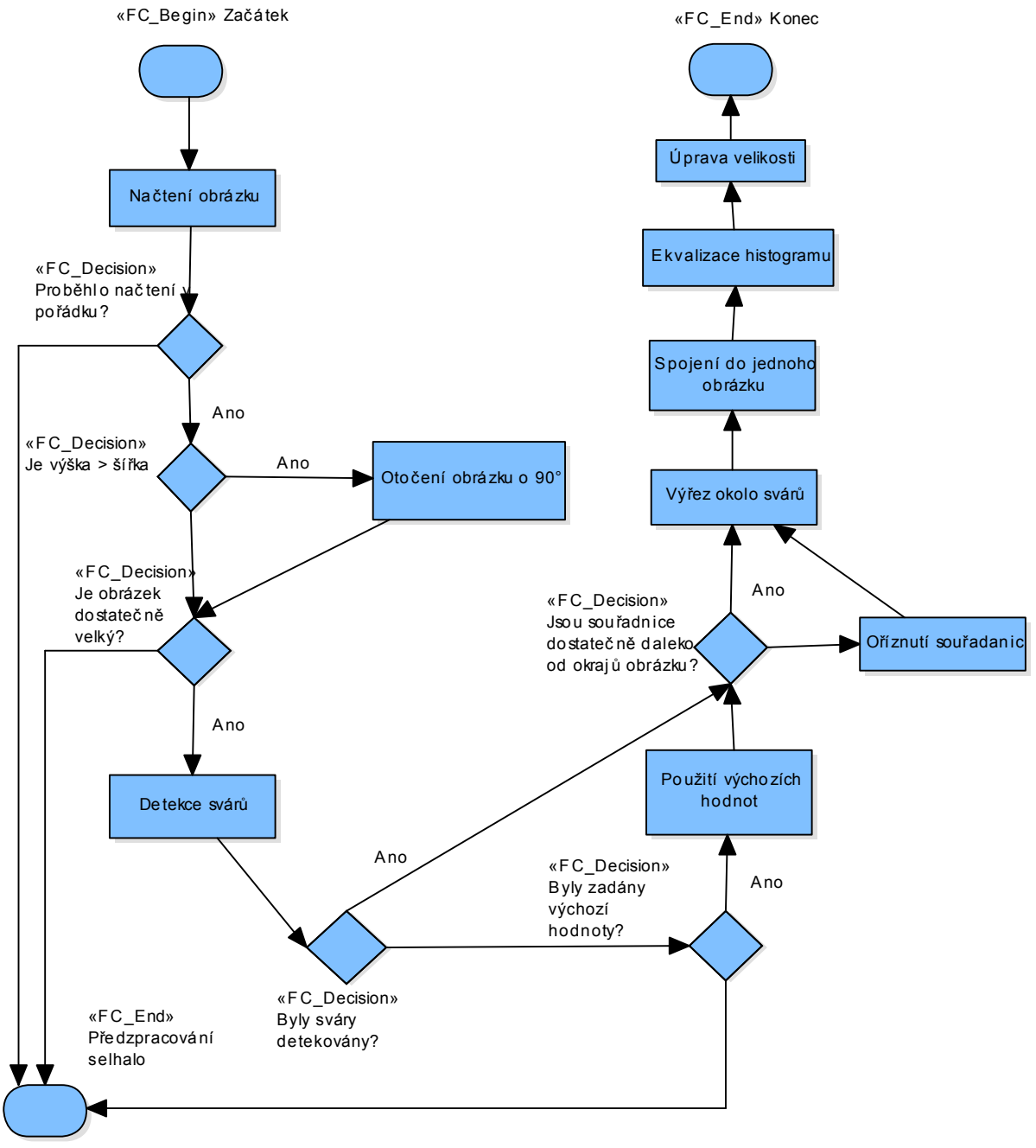
[15] TensorFlow: An open-source machine learning framework for everyone. *TensorFlow* [online]. [2017] [cit. 2017-09-25]. Dostupné z: https://www.tensorflow.org/
[16] CHINTALA, Soumith. Convnet benchmarks. *GitHub* [online]. San Francisco: GitHub, © 2018, 2014 [cit. 2017-09-25]. Dostupné z: https://github.com/soumith/convnet-benchmarks
[17] ROBINSON, Scott. The Best Machine Learning Libraries in Python. *Stack Abuse* [online]. © 2018, 10. 11. 2015 [cit. 2017-09-25]. Dostupné z: http://stackabuse.com/the-best-machine-learning-libraries-in-python/
[18] Getting Started with TensorFlow. *TensorFlow* [online]. [2017] [cit. 2017-09-27]. Dostupné z: https://www.tensorflow.org/get_started/get_started
[19] TensorBoard: Graph Visualization. *TensorFlow* [online]. [2017] [cit. 2017-09-28]. Dostupné z: https://www.tensorflow.org/get_started/graph_viz
[20] ZADRAŽIL, Petr. MDevTalk #1: TensorFlow v mobilních aplikacích. *YouTube* [online]. mDevTalk, 22. 1. 2016 [vid. 2017-09-29]. Dostupné z: https://youtu.be/7jhDYKZ8wqc
[21] BINGHAM, Derek a Sonja SURJANOVIC. Rastrigin Function. *Test Functions and Datasets* [online]. Burnaby: Simon Fraser University, © 2013, Srpen 2017 [cit. 2017-09-30]. Dostupné z: https://www.sfu.ca/~ssurjano/rastr.html
[22] TkInter - Python Wiki. *The Python Wiki* [online]. [2017], [2017] [cit. 2017-09-30]. Dostupné z: https://wiki.python.org/moin/TkInter
[23] Numpy main repository. *GitHub* [online]. San Francisco: GitHub, © 2018, [2011] [cit. 2017-10-01]. Dostupné z: https://github.com/numpy/numpy
[24] A Guide to TF Layers: Building a Convolutional Neural Network. *TensorFlow* [online]. [2017] [cit. 2017-10-03]. Dostupné z: https://www.tensorflow.org/get_started/mnist/pros
[25] Contours : Getting Started. *OpenCV* [online]. OpenCV, © 2018, 18. 12. 2015 [cit. 2017-10-05]. Dostupné z: https://tinyurl.com/ydghhspf
[26] Contour Features. *OpenCV* [online]. OpenCV, © 2018, 18. 12. 2015 [cit. 2017-10-05]. Dostupné z: https://tinyurl.com/y8s3mlzv
[27] RAHMAN K, Abid. How to detect simple geometric shapes using OpenCV? *Stack Overflow* [online]. New York: Stack Exchange, © 2018, 20. 3. 2012 [cit. 2017-10-06]. Dostupné z: https://stackoverflow.com/a/11427501
[28] Training: Optimizer. *TensorFlow* [online]. [2017] [cit. 2017-10-07]. Dostupné z: https://www.tensorflow.org/api_guides/python/train#Optimizers
[29] Training: AdamOptimizer. *TensorFlow* [online]. [2017] [cit. 2017-10-08]. Dostupné z: https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer
[30] KINGMA, Diederik P. a Jimmy Lei BA. Adam: A Method for Stochastic Optimization: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. *ArXiv* [online]. 22. 12. 2014 [cit. 2017-10-10]. Dostupné z: http://www.arxiv.org/abs/1412.6980
[31] BUDUMA, Nikhil a Nicholas LOCASCIO. *Fundamentals of deep learning: designing next-generation machine intelligence algorithms*. Sebastopol, CA: O'Reilly Media, 2017. ISBN 978-1491925614.
[32] POWERS, David Martin Ward. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation. *Journal of Machine Learning Technologies*. 2011, **2**(1), 37-63. DOI: 10.9735/2229-3981. ISSN 22293981.

Přílohy

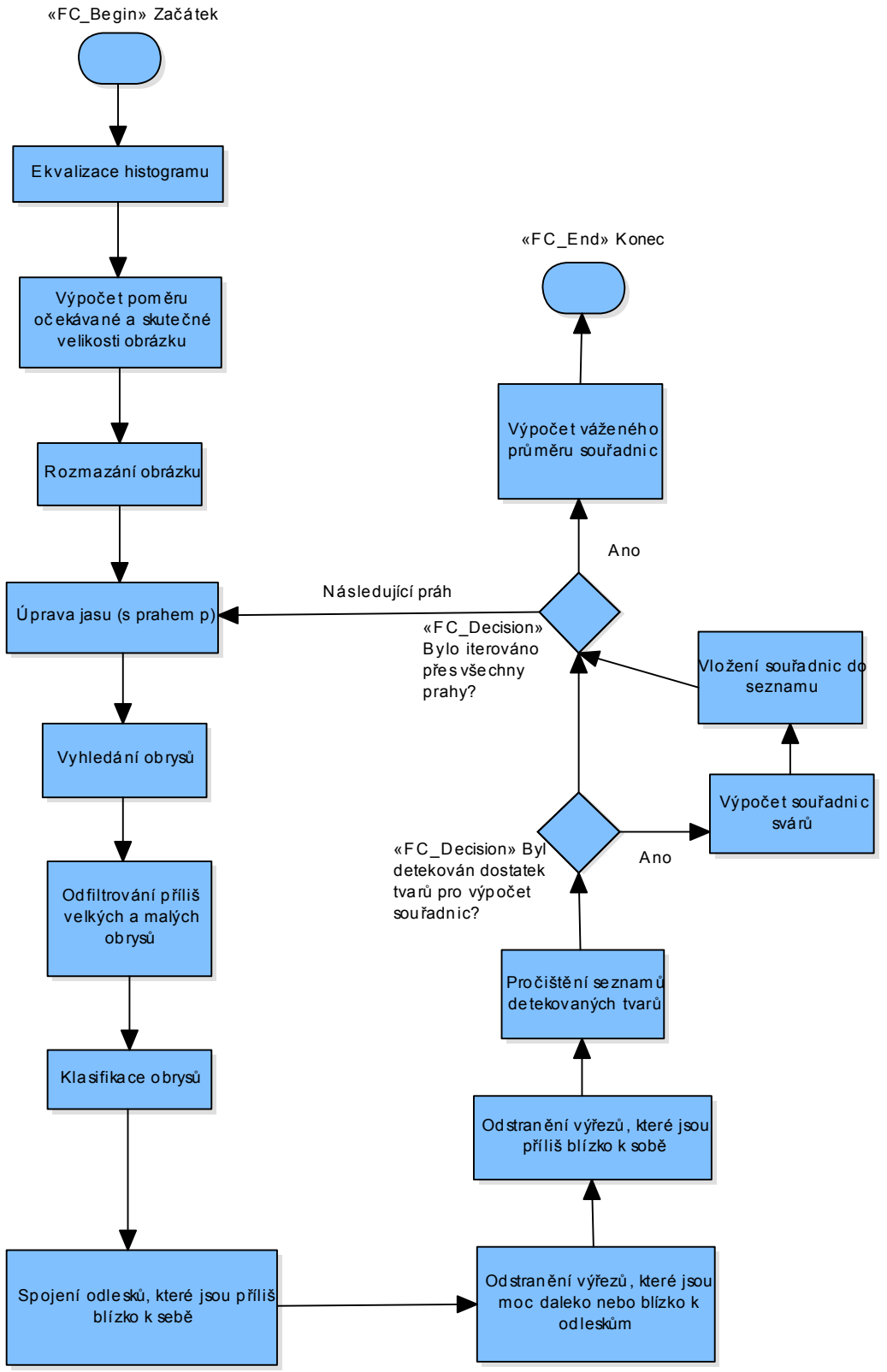
Příloha A: Vysokoúrovňový pohled na předzpracování obrázků (diagram)



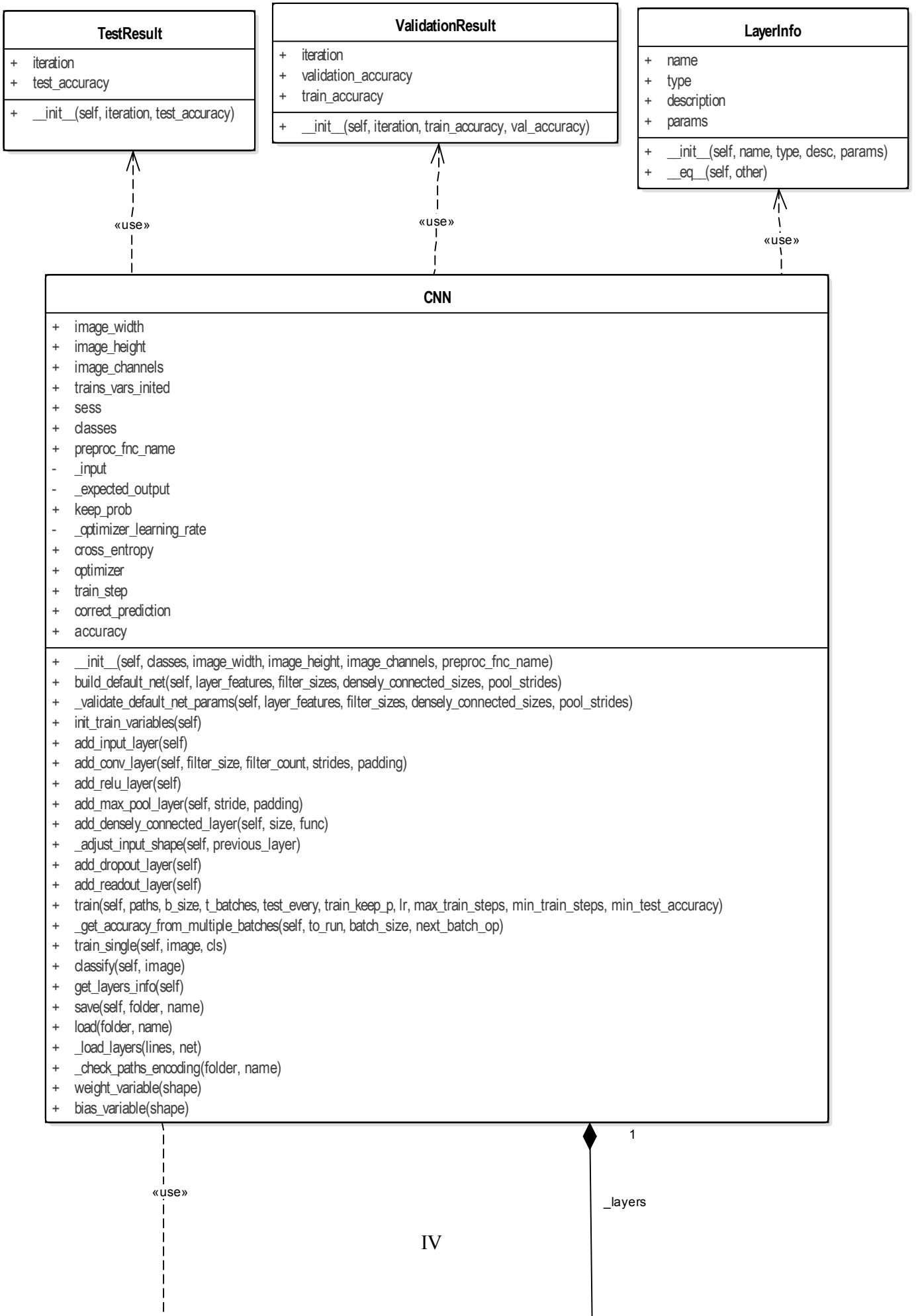
Příloha B: Diagram procesů Příprava obrázku a Finální úpravy

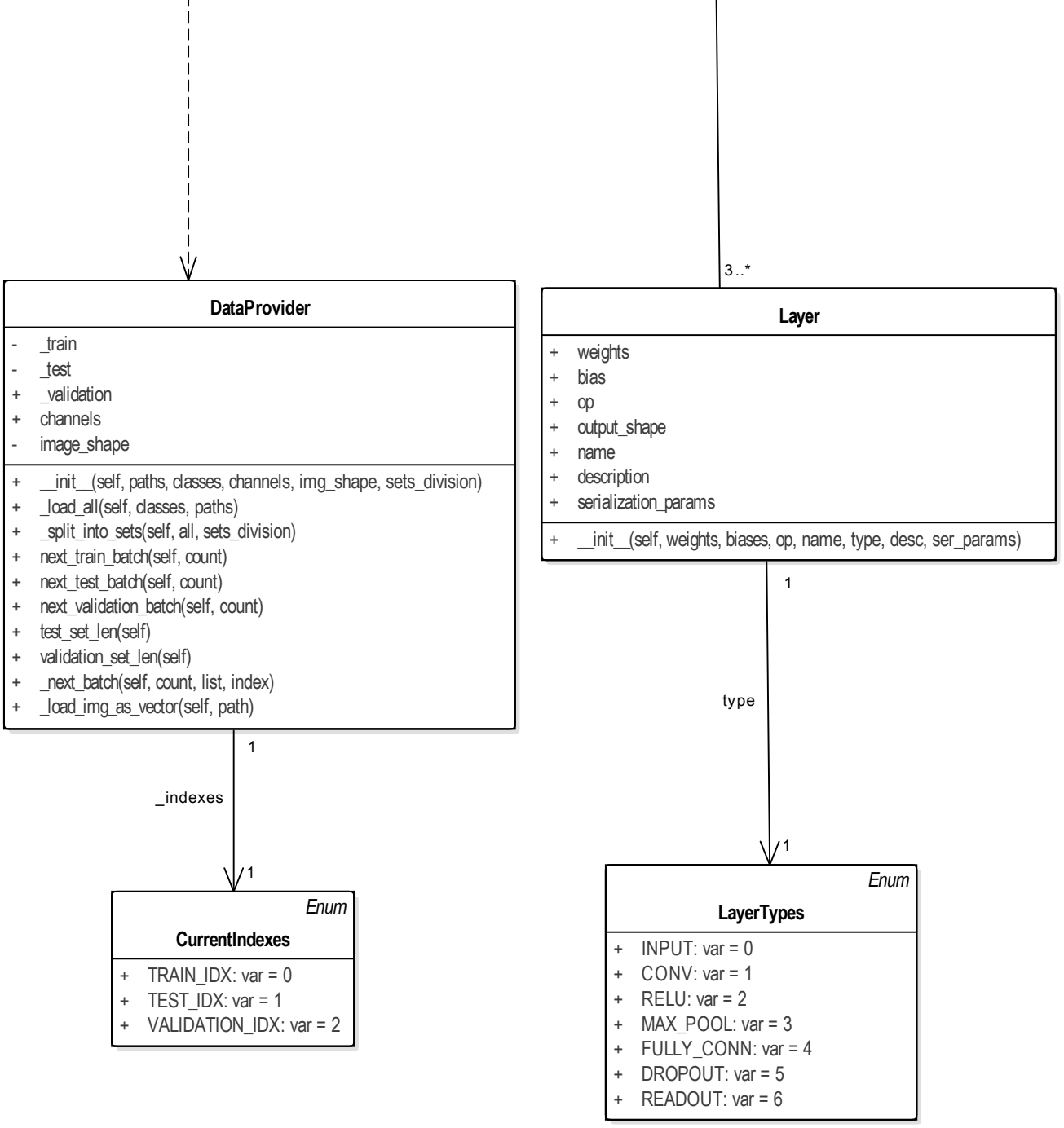


Příloha C: Diagram procesu Detekce svárů



Příloha D: UML diagram tříd




DataProvider
- _train
- _test
+ _validation
+ channels
- image_shape
+ __init__(self, paths, classes, channels, img_shape, sets_division)
+ _load_all(self, classes, paths)
+ _split_into_sets(self, all, sets_division)
+ next_train_batch(self, count)
+ next_test_batch(self, count)
+ next_validation_batch(self, count)
+ test_set_len(self)
+ validation_set_len(self)
+ _next_batch(self, count, list, index)
+ _load_img_as_vector(self, path)
1
_indexes
1
Enum
CurrentIndexes
+ TRAIN_IDX: var = 0
+ TEST_IDX: var = 1
+ VALIDATION_IDX: var = 2
3..*
Layer
+ weights
+ bias
+ op
+ output_shape
+ name
+ description
+ serialization_params
+ __init__(self, weights, biases, op, name, type, desc, ser_params)
1
type
1
Enum
LayerTypes
+ INPUT: var = 0
+ CONV: var = 1
+ RELU: var = 2
+ MAX_POOL: var = 3
+ FULLY_CONN: var = 4
+ DROPOUT: var = 5
+ READOUT: var = 6