ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Rozpoznávání matematických a chemických vzorců ve strukturovaných dokumentech

Csaba Filip

Květen 2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Filip** Jméno: **Csaba** Osobní číslo: **466056**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Rozpoznávání matematických a chemických vzorců ve strukturovaných dokumentech

Název bakalářské práce anglicky:

Pokyny pro vypracování:

Cílem práce je navrhnout a implementovad metodu pro detekci jednoduchých matematických (2. třída ZŠ) a chemických vzorců (2. stupeň ZŠ) v tištěných dokumentech, které jsou snímané kamerou.
Postup:
1. Připomeňte si strukturu chemických vzorců [6]. Zvolte vhodné typy dokumentů s aplikačním potenciálem, ve kterých se uvažované matematické a chemické vzorce hojně vyskytují (např.: automatické doplnění/kontrola výsledků v příkladech).
2. Implementujte modul pro automatické generování velké množiny trénovacích dat.
3. Využijte systém YOLO [3] pro detekci vzorců. Pro rozpoznávání jednotlivých znaků využijte dostupné OCR enginy (např. [4]) nebo opět YOLO.
4. Navrhněte rozšíření klasických konečných automatů a/nebo regulárních gramatik [1] pro popis struktury vzorců (viz např. [5]) a implementujte parser vzorců.
5. Analyzujte přesnost a rychlost vyvinutých metod, popište jejich limity.
6. Implementaci doplňte stručnou uživatelskou a programátorskou dokumentací.

Seznam doporučené literatury:

[1] J. E. Hopcroft, R. Motwani, J. D. Ullman: Introduction to Automata Theory, Languages, and Computation (3rd ed.), Pearson, 2013.
[2] R. Sedgewick, K. Wayne, R. Dondero: Introduction to Programming in Python: An Interdisciplinary Approach, Addison-Wesley Professional, 2015.
[3] YOLO: Real-Time Object Detection, https://pjreddie.com/darknet/yolo/
[4] Tesseract Open Source OCR Engine: https://github.com/tesseract-ocr/
[5] K. Nevolová: Online rozpoznávání chemických vzorců a rovnic, diplomová práce, MFF UK, 2016, https://is.cuni.cz/webapps/zzp/detail/115272/
[6] Chemie 8 - Úvod do obecné a anorganické chemie (učebnice), NOVÁ ŠKOLA, s.r.o., 2010.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Daniel Průša, Ph.D., Strojové učení FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **26.01.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

RNDr. Daniel Průša, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.
Datum převzetí zadání

Podpis studenta

Poděkování / Prohlášení

Tímto bych rád poděkoval vedoucímu své bakalářské práce RNDr. Danielu Průšovi, Ph.D., za vstřícný přístup a cenné rady při psaní této práce. Díky patří také mé rodině a přátelům za podporu během celého studia.

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

V Praze dne 23. 5. 2019

.....

Abstrakt / Abstract

Ve své bakalářské práci se budu zabývat problematikou rozpoznávání jednoduchých matematických a chemických rovnic ve strukturovaných dokumentech. K detekci rovnic využiji systém pro detekci objektů YOLO vytrénovaný na automaticky vygenerovaných trénovacích datech. K rozpoznávání jednotlivých znaků použiji OCR engine Tesseract. Pro popis struktury vzorců navrhnu automat a implementuji parser. Nakonec implementuji mobilní aplikaci pro rozpoznávání rovnic v Androidu a otestuji ji.

Klíčová slova: rozpoznávání, YOLO, Tesseract, chemická rovnice, matematická rovnice, chemický vzorec

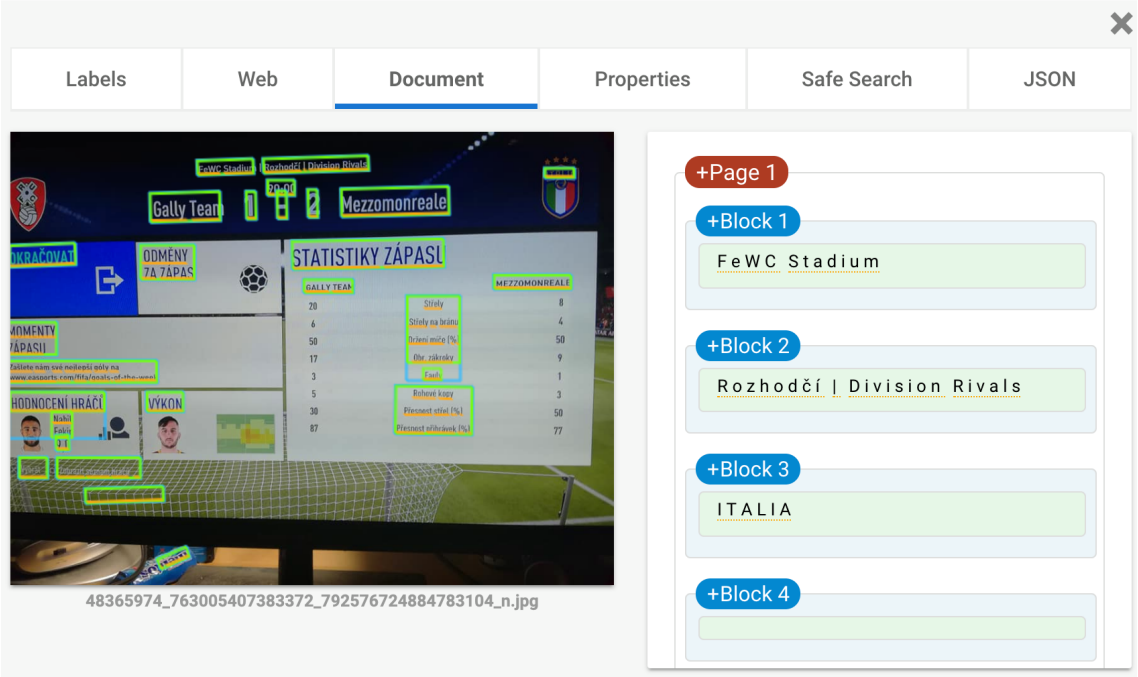
In my bachelor's thesis I will address the problem of recognizing mathematical and chemical equations in structured documents. For the detection of the equations in a scene I use the famous object detection system YOLO. I train my custom model on automatically generated data. For the recognition of individual characters I will use the OCR engine Tesseract. I propose a finite automaton for the description of equations. Lastly I am going to implement and test an Android mobile application capable of detecting and recognizing simple equations.

Keywords: recognition, YOLO, Tesseract, chemical equation, mathematical equation, chemiacl formula

Obsah /

Kapitola 1
Úvod

Lokalizace a rozpoznávání textu na obrázku je jeden z nejrozšířenějších otevřených problémů v oblasti počítačového vidění. Obzvlášť důležitým se tento problém stal s rozmachem mobilních zařízení s kvalitními kamerami. Dobrým příkladem existujícího řešení je například mobilní aplikace Microsoft Translator umožňující detekci, rozpoznání a překlad textu v obrázku ve více než 60 jazycích, nebo (placené) Cloud Vision API od Google či celá řada komerčních produktů společnosti Abbyy.



Obrázek 1.1. Ukázka trial verze Cloud Vision API. I s nekvalitním obrázkem generuje velmi dobré výsledky.

Problém rozpoznávání textu ve scéně lze rozložit na dva podproblémy, a to lokalizace textu a rozpoznávání slov. Cílem lokalizace textu je detekování textu v obrázku a jeho ohraničení obdélníkovým rámečkem. Po úspěšné lokalizaci textu je třeba text rozpoznat, obvykle pomocí nějakého optického rozpoznávače znaků (OCR, *optical character recognition*). Modernější, efektivnější, ale složitější je *end-to-end* přístup k problému, tedy souběžná detekce a rozpoznání textu v jednom frameworku. Příkladem tohoto přístupu je Deep TextSpotter [1] nebo systém navržený C. Leem a S. Osinderem [2].

K problému rozpoznávání chemických a matematických rovnic na obrázku lze přistupovat podobně jako k rozpoznávání obecného textu. Nejprve tedy lokalizovat chemickou rovnici na obrázku (a „ignorovat“ ostatní text) a následně rozpoznat danou rovnici pomocí OCR enginu a syntakticky analyzovat její strukturu.

Některé existující systémy pro rozpoznávání matematických a chemických rovnic jsou *Infty Reader*[3] nebo *MyScript*[1]. Zajímavá je dále soutěž v rozpoznávání ručně psaných matematických výrazů *ICFHR 2016 CROHME* [4]. Uvedené existující systémy byly navrženy pro rozpoznávání rovnic a vzorců se složitou rekurzivní strukturou. Obvykle tyto systémy používají pro popis struktury vzorců dvourozměrné varianty bezkontextových gramatik [5]. Oproti tomu se má práce zabývá rovnicemi s téměř lineární strukturou, pro jejichž rozpoznání lze uplatnit speciální postupy. Namísto 2D gramatik postačuje rozšíření klasických konečných automatů.

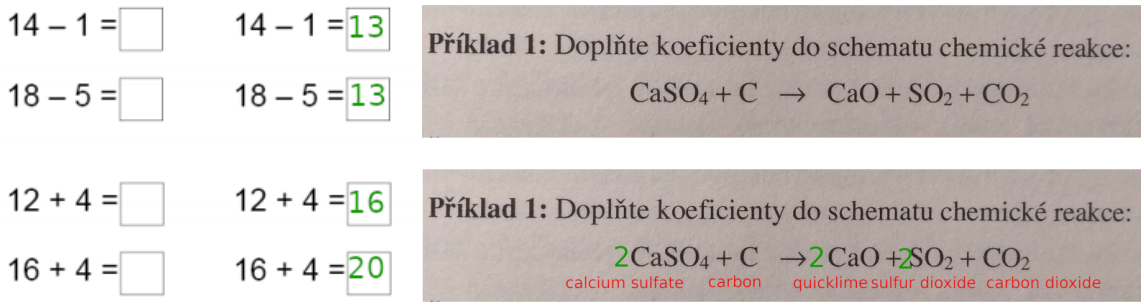
1.1 Návrh řešení a cíl projektu

Pro zjednodušení problému se zaměřím pouze na rozpoznávání tištěných chemických a matematických rovnic. K detekci rovnic použiji systém na detekci objektů YOLO, trénovaný k tomuto účelu. Detekční modul poté propojím s OCR enginem Tesseract 3.05. Na základě znalosti automatu přijímajícího rovnice sestrojím syntaktický analyzátor, který redukuje chybné rozpoznání znaků. Nakonec provedu validaci chemické rovnice a případně její vyčíslení.

Matematické rovnice, kterými se budu zabývat, jsou rovnice se dvěma operandy a operátorem, tedy jednoduché rovnice na sčítání, odčítání, násobení a dělení zhruba odpovídající úrovni 2. třídy ZŠ. Z chemických rovnic to budou ty s téměř lineární strukturou, tedy rovnice s několika sloučeninami na levé a pravé straně. Nebudu se zabývat strukturními vzorci z organické chemie ani oxidačními čísly v reakci.

Cílem projektu je vytvoření kompaktní, uživatelsky přívětivé aplikace na Android, která bude schopna v reálném čase (nebo alespoň co nejrychleji) detekovat rovnice na obrazu z kamery zařízení. Následně aplikace pomocí OCR a analyzátoru rozpozná jednotlivé komponenty rovnice. V případě chemické rovnice vypíše názvy sloučenin a pokusí se rovnici vyčíslit (pokud ještě není vyčíslená). V případě matematické rovnice vypíše výsledek.

Takovouto aplikaci by bylo možné využít například na automatizovanou kontrolu testů. V upravené podobě by také mohla sloužit jako pomocník ke studiu chemie. Kromě výpisu názvů sloučenin by také mohla zobrazovat informace o chemických vlastnostech jednotlivých komponent a celé reakce. Tyto a další data by aplikace mohla čerpat například ze serveru PubChem, který pro tento účel disponuje programovým API.



Obrázek 1.2. Ilustrace fungování zamýšlené aplikace. Matematické rovnice vyřeší, chemické vyčíslí a vypíše triviální názvy sloučenin.

[1] `https://www.myscript.com/`

Kapitola 2
YOLO

YOLO [6] je zkratka pro You Only Look Once (stačí se podívat jednou), která velmi dobře vystihuje, v čem se YOLO liší od většiny ostatních systémů pro detekci objektů. Většina dřívějších systémů využívá klasifikátor pro daný objekt, kterým vyhodnocuje různé oblasti různých velikostí v daném obrázku. Daný obrázek se tedy zpracovává vícekrát za sebou. Oproti tomu YOLO vyhodnocuje obrázek pouze jednou v přímočarém procesu začínajícím pixely obrázku a končícím tensorem zachycujícím oblasti a třídy detekovaných objektů. Díky tomu je celý proces velice rychlý, umožňuje dokonce zpracování obrazu z kamery v reálném čase.

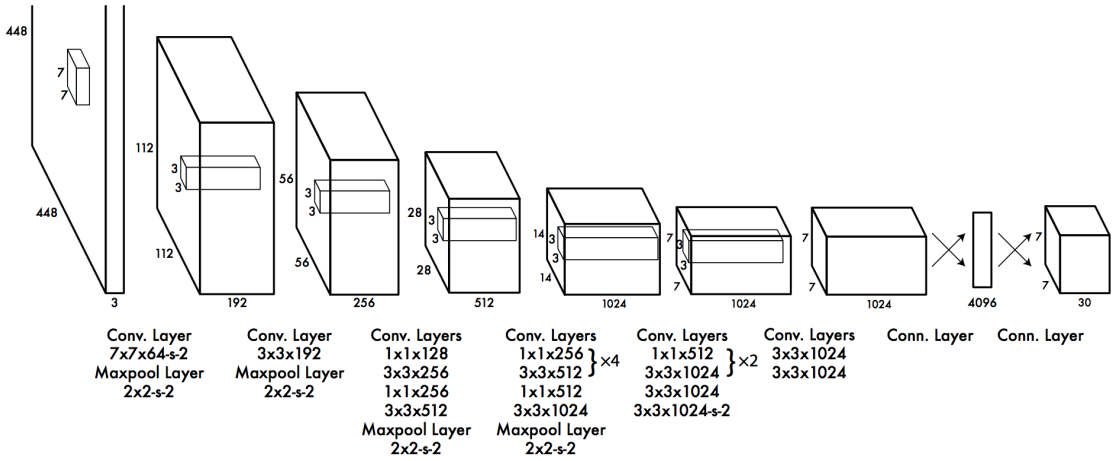
Systém YOLO je konvoluční neuronová síť. Původně byl implementován ve frameworku jednoho svých tvůrců (Joseph Redmon) Darknet. Implementován může být samozřejmě i v jiných frameworcích (Tensorflow, Theano). Pro YOLO v mobilní aplikaci je nejvhodnější Tensorflow Lite podporující Android Neural Network API.

Momentálně je nejnovější verze YOLOv3. Princip systému zůstává stále stejný, obrázky jsou vyhodnocovány pouze jednou. Zatímco YOLOv2 přineslo zlepšení jak v rychlosti, tak i v přesnosti, YOLOv3 zavedením tříúrovňové detekce celý proces malinko zpomaluje, zásadně však zlepšuje rozpoznávání drobných objektů.

2.1 Fungování YOLO

Architektura původního YOLOv1 systému byla inspirována GoogLeNet modelem [7] pro klasifikaci obrázků. Úkolem konvolučních vrstev sítě je převést vstupní obrázek na $S \times S$ mřížku. Spadá-li střed objektu k detekování do některé z buněk mřížky, je úkolem této buňky objekt detekovat. Každá buňka předvídá určitý počet rámečků a *confidence score* (přesvědčení systému), že daný rámeček obsahuje objekt a že jej rámeček ohraničuje přesně. Neobsahuje-li rámeček žádný objekt, mělo by být toto číslo ideálně nula. Kromě *confidence score* sestává každá predikce rámečku z dalších 4 čísel: (x, y) souřadnice prostředku rámečku (tedy odhad prostředku detekovaného objektu) a šířka a výška relativní k velikosti obrázku. Výstupem procesu je tedy $S \times S \times (B * 5 + C)$ tensor, kde $S \times S$ je zmiňovaná velikost mřížky, B je počet rámečků, které každá buňka mřížky predikuje. B je násobeno 5-ti, jelikož se každá predikce (rámečku) skládá z 5-ti údajů.

Každá buňka má tedy na starosti predikovat pravděpodobnosti, že objekt, jehož střed leží v této buňce, je dané třídy. I když neleží střed žádného objektu v dané buňce, predikce musí stejně existovat, ale jejich *confidence scores* by měly být co nejnižší. Buňka také zajišťuje samotnou klasifikaci objektů. Tato klasifikace (jak lze vyčíst z rozměrů tensoru) je u YOLOv1 shodná pro všechny rámečky předpovídané danou buňkou.



Obrázek 2.1. [6] Architektura původního YOLO. Vstupní vrstva s rozlišením 448×448 a výstupní tensor $7 \times 7 \times 30$ odpovídající $S = 7, B = 2, C = 20$ – určený pro Pascal VOC dataset s 20-ti třídami objektů.

YOLOv2 [8] je přímým nástupcem YOLO, který jak v rychlosti, tak v přesnosti překonává původní systém. Ačkoliv princip fungování zůstává stejný, přináší nový systém mnoho vylepšení. Za zmínku určitě stojí tzv. *batch normalization* [9] (způsob normalizace představený v roce 2015 vylepšující výkonnost a stabilitu) a *anchor boxes*. Zatímco v YOLOv1 je klasifikace objektu určena buňkou mřížky a každá predikce dané buňky klasifikaci sdílí, v dalších verzích je již údaj součástí predikce rámečků.

Anchor boxy jsou předem definované velikosti rámečků. Jsou stejné pro každou buňku mřížky a jejich počet musí být shodný s B, počtem rámečků, které buňka predikuje. Po jejich zavedení již buňka nepředpovídá rámečky přímo, místo toho predikuje offsety vzhledem k jednotlivým *anchor boxům*. To umožňuje zvýšení počtů predikcí pro každou buňku bez zvýšení výpočetní náročnosti. S vhodnou volbou boxů budou navíc predikované offsety malé a trénování modelu efektivnější.

Poslední verze, YOLOv3 [10], přináší opět mnoho menších změn. Oproti YOLOv2 je síť hlubší (53 konvolučních vrstev oproti 19) a tedy pomalejší a přesnější. Největší novinkou je, že YOLOv3 předpovídá rámečky na 3 různých úrovních s různým rozlišením mřížky. Díky tomu se velmi zlepšila detekce malých objektů, za kterou je zodpovědná mřížka s nejvyšším rozlišením.

YOLOv3 je pravděpodobně poslední inkarnací tohoto systému. Hlavní autor, Joseph Redmon, nese s velikou nelibostí fakt, že jím představené technologie našli užití především ve velkých společnostech jako jsou Facebook a Google. Zároveň varuje před možným vojenským a hlavně monitorovacím využitím těchto technologií, jako vidíme například v Číně [11].

2.2 Instalace YOLO

Jak jsem již uváděl, YOLO může být implementováno v různých frameworcích pro strojové učení. Nejjednodušší je začít s původní implementací v Darknetu. Na stránkách tvůrce[1] jsou k dispozici všechny potřebné soubory a balíčky i s návodem. Pro operační systém Windows doporučuji upravenou verzi Darknetu[2].

[1] `https://pjreddie.com`
[2] `https://github.com/AlexeyAB`

Většina frameworků umožňuje zrychlení výpočtů použitím GPU. Darknet a Tensorflow využívají k tomuto účelu architekturu CUDA (Compute Unified Device Architecture). Použití je omezeno na grafické procesory společnosti NVIDIA, která technologii vyvinula. Přehled podporovaných grafických procesorů lze nalézt na stránkách výrobce[1]. Samotné podporované GPU však nestačí, je potřeba mít nainstalovaný CUDA Toolkit. Jeho instalace je poměrně komplikovaná. Především na některých distribucích Linuxu, které mají často problémy s grafickými drivery, je instalace téměř nemožná. Dále je vhodné nainstalovat SDK cuDNN (NVIDIA CUDA Deep Neural Network library) pro další zrychlení.

V závislosti na počtu CUDA jader a paměti GPU lze dosáhnout až několika set násobného zrychlení. Možné je dokonce i zapojení více GPU. Využití CPU verze frameworku na trénování modelu (ale i pro samotnou predikci) je krajně nevýhodné. I s nejlepšími dnešními grafickými kartami může zabrat trénování modelu několik dní. S CPU by se pak doba trénování mohla pohybovat v řádu let.

Je vhodné dodat, že grafické karty určené pro koncovou běžnou spotřebu není radno používat na delší trénování.

[1] `https://developer.nvidia.com/cuda-gpus`

Kapitola 3
Tvorba trénovacího datasetu

Pro trénování, testování a především porovnávání výkonů různých modelů existuje několik veřejně dostupných datasetů. Obsahují desítky až stovky tisíc anotovaných (pozice, velikost rámečku a třída objektu) obrázků s několika třídami běžných objektů, jako je například automobil, kniha, člověk nebo pes. Pravděpodobně nejznámější takové datasety jsou Pascal VOC (k roku 2012 měl 20 tříd objektů)[1] a COCO (více než 200000 označkovaných obrázků, 80 tříd objektů)[2].

Pokud vím, tak bohužel neexistuje trénovací dataset s chemickými rovnicemi. Vyfotit několik tisíc rovnic a manuálně je anotovat je časově nesmírně náročné. Je tedy potřeba si vygenerovat syntetický dataset.

Pro tvorbu trénovacího datasetu neexistují žádné konkrétní postupy. Často je nutné postupovat metodou pokus-omyl. Nedetekuje-li například model objekty pod určitým úhlem, je vhodné do trénovacího datasetu přidat více vzorků, kde je objekt zachycen pod tímto úhlem a model opět trénovat. Jedná se často spíše o umění než vědu.

Na začátek je dobré si uvědomit, jaké asi obrázky bude model dostávat jako vstup. V našem případě se bude jednat o obrázky tištěných strukturovaných dokumentů s chemickými rovnicemi. Tyto dokumenty budou často obsahovat kromě chemických rovnic i doprovodný text a budou moci být zachyceny pod různými úhly (ne extrémními) a různým osvětlením. Dále je třeba počítat s různými fonty, pozadími, velikostí písma atd.

3.1 Implementace generátoru datasetu

Tvorbu datasetu implementuji v Pythonu. Pomocí jednoduchého skriptu vytvořím souborovou databázi běžných chemických sloučenin [12] v *TinyDB*. Pomocí těchto dat je již snadné tvořit chemické rovnice, stačí náhodně zvolit několik sloučenin na levou a pravou stranu rovnice. Reakce sice z chemického hlediska nedávají smysl, ale to v této fázi vůbec nevadí. Generátor matematických rovnic s 4 operátory + - × ÷ je zcela triviální.

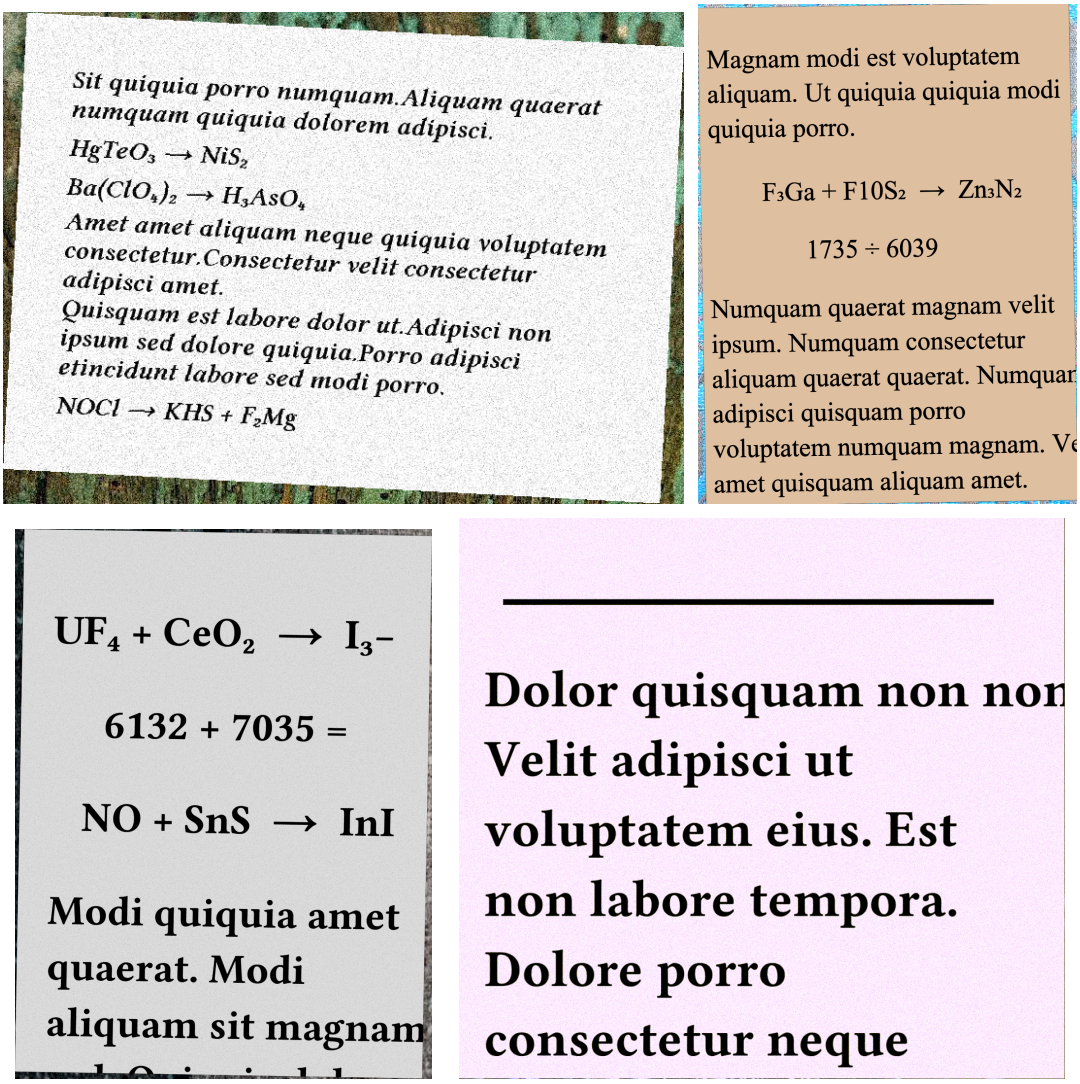
Pro generovaní náhodného textu využiji knihovnu *Lorem*. Zápis textu do obrázků umožňuje knihovna *PIL* (Python Image Library, dříve Pillow). Výplňový text a chemické rovnice pak zvoleným fontem vkresluji do obrázků. Pro zlepšení detekce je vhodné do datasetu přidávat negativní vzorky, tedy obrázky bez výskytu rovnic pouze s textem nebo jen s pozadím.

Pro pozadí vzorků využívám Describable Textures Dataset[3], kolekci několika tisíců texturových obrázků. Rotaci obrázku a přepočet souřadnic rámečků rovnic značně zjednodušuje známá knihovna pro manipulaci s obrazem, OpenCV.

Reálně vyfocené obrázky bývají zatíženy šumem, jsou rozmazané a špatně osvětlené. V poslední řadě tedy na vygenerované vzorky uplatním post-processing, který by

[1] `http://host.robots.ox.ac.uk/pascal/VOC/`
[2] `http://cocodataset.org`
[3] `https://www.robots.ox.ac.uk/~vgg/data/dtd/`

měl zatím bezchybné obrázky více přiblížit reálným. Za tímto účelem využiji knihovnu *imgaug*[1], která poskytuje celou řadu užitečných nástrojů k augmentaci obrázků.



Obrázek 3.1. Ukázky vygenerovaných vzorků

K řízení generátoru potřebuji mnoho globálních proměnných jako například maximální úhel rotace, maximální velikost písma, pravděpodobnost výskytu rovnice, pravděpodobnost výskytu stínu atd. Pro přehlednost jsem tyto proměnné vyextrahoval do konfiguračního souboru. Pomocí něho lze měnit výsledný dataset bez zásahu do kódu.

```
;AUGMENTER params
max_angle       = 4
max_perspective = 0.005
max_noise       = 0.05
max_value       = 50
blur_prob       = 0.35
shadow_prob     = 0.25
max_shadows     = 3
```

Tabulka 3.1. Ukázka několika parametrů konfiguračního souboru

[1] https://github.com/aleju/imgaug

Kontextů, ve kterém se chemické rovnice mohou objevovat, je neomezeně mnoho. Ať již přímo „in line“ v textu, v tabulce, v bodech, s různou barvou pozadí a písma atd. Vytvořit dataset obsahující rovnice ve všech možných kontextech je samozřejmě nemožné. Není to naštěstí ani potřeba, stačí zachytit ty nejběžnější a s trochou štěstí pak YOLO rozpozná rovnice i ve více exotických kontextech. A když ne, je třeba daný kontext zachytit v trénovacím datasetu. Tvorba vhodného datasetu je tedy iterativní proces. Ve fázi testování je potřeba identifikovat kontexty, ve kterých detekce rovnic selhává. Tyto kontexty lze zachytit v trénovacím datasetu a nově trénovaný model by s detekcí již neměl mít problém.

Kapitola 4
Trénování modelu ve frameworku Darknet

V této kapitole se pokusím poskytnut návod a shrnout zkušenosti získané při trénování sítě. Popíšu problémy, se kterými jsem se během trénování setkal a způsoby, kterými jsem je řešil.

Samotný dataset obrázků k trénování nestačí. Model potřebuje zpětnou vazbu, tedy informaci o souřadnicích, velikosti a třídě objektu, který chceme detekovat. Ve frameworku Darknet se tyto data k danému obrázku poskytnou jednoduše ve formě textového souboru (.txt) ve stejné složce a se stejným názvem, jako má obrázek. Pro každý objekt na obrázku obsahuje textový soubor nový řádek s číslem třídy objektu, jeho souřadnicemi a rozměry:

```
<třída> <x-střed> <y-střed> <šířka> <výška>
```

Souřadnice a rozměry jsou relativní, počítají se vzhledem k výšce (resp. šířce) obrázku, jsou to tedy čísla mezi 0 a 1. Názvy tříd jsou v Darknetu definovány zvlášť v souboru s příponou *.names*. Číslo třídy tedy udává řádek s názvem dané třídy ve zmíněném souboru.

Správně anotovat všechny obrázky dle uvedeného předpisu je obtížné a zdlouhavé. Při automatickém generování obrázku je tedy rozumné zároveň rovnou generovat soubor s příslušnou anotací. Pokud obrázky nejsou generované, lze k anotaci využít nástroj Yolo-mark[1] umožňující grafickou anotaci přímo v obrázku.

4.1 Konfigurace modelu

Model YOLO v Darknetu sestává ze 2 souborů, a to ze souboru s váhami v jednotlicýh vrstvách modelu (*.weights*) a ze souboru s konfigurací modelu (*.cfg*). Konfigurační soubor definuje rozměry vstupní vrstvy (a tím i výstupní $S \times S$ mřížku), počet a typy (např.: convolutional, maxpool, upsample) jednotlivých skrytých vrstev a jejich aktivačních funkcí. Dále definuje počet tříd, které chceme detekovat a u YOLOv2 a v3 i *anchor boxy*, které chceme k detekci využít.

Soubor udává i důležité parametry pro trénování modelu. Jedním z nich je parametr *batch*, dávka. Trénování modelu totiž neprobíhá po jednotlivých vzorcích (v tomto případě obrázcích) nýbrž po dávkách. Často je totiž neefektivní aktualizovat vnitřní parametry modelu (váhy) po každém zpracovaném obrázku a je výhodnější toto provádět po dávkách. Při volbě velikosti dávky je potřeba zvážit dostupnou paměť (RAM) GPU. Není dobré zvolit co největší velikost dávky, na kterou paměť GPU vystačí. Jednotlivé dávky se totiž mohou v paměťové náročnosti výrazně lišit a může se stát, že se v několika sté iteraci překročí limit, paměť se nepodaří alokovat a trénování selže na chybě. Takto lze přijít o několik hodin pokroku. Tomu lze předejít buď snížením velikosti dávky, nebo zvýšením velikosti parametru *subdivision*, který určuje, do kolika subdivizí bude dávka rozdělena. GPU pak zpracovává tyto subdivize.

[1] https://github.com/AlexeyAB/Yolo_mark

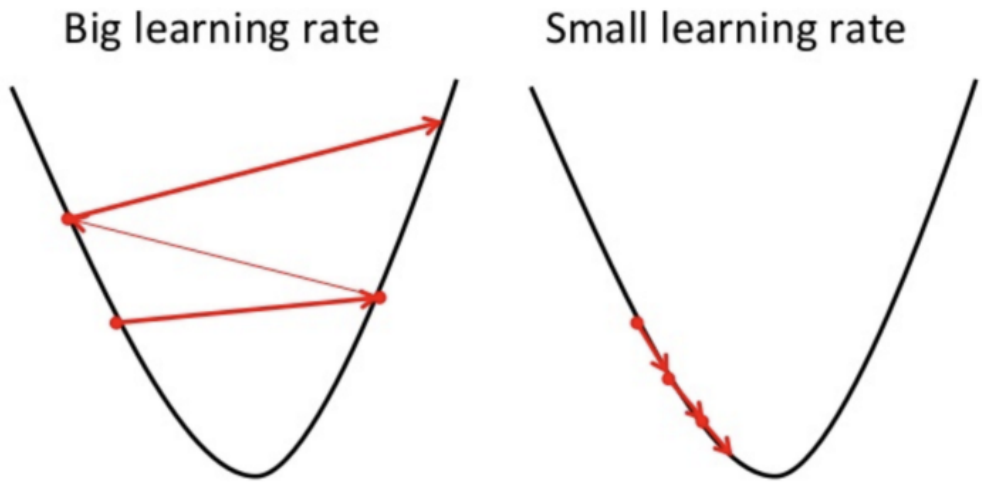
V kontextu neuronových sítí je pojem dávky často nesprávně zaměňován s pojmem epochy. Zatímco dávka definuje počet zpracovaných vzorků, než budou aktualizovány vnitřní parametry modelu, epocha udává počet kompletních průchodů datasetem. Máme-li tedy například dataset 1000 vzorků a velikost dávky 100, pak po 3. epoše bylo zpracováno 30 dávek.

Darknet přímo s epochou jako parametrem nepracuje. Místo toho nabízí parametr `max batches` určující, po kolika dávkách má trénování skončit.

Dalším důležitým parametrem je *learning rate*, tempo učení, udávající, jak moc se mají měnit váhy v modelu vzhledem k vypočtenému gradientu po zpracování dávky.

$$nova\_vaha = stavajici\_vaha - learning\_rate \times gradient$$

Jedná se v podstatě o délku kroku gradientní metody. Je-li *learning rate* moc malý, konvergence může být pomalá a trénování může trvat velmi dlouho a naopak, je-li *learning rate* moc velký, nemusíme se vůbec dočkat fungujícího modelu.



Obrázek 4.1. [13] Vizualizace důležitosti délky kroku při hledání optima. Vpravo příliš velká délka, vlevo malá.

S parametrem learning rate úzce souvisí parametry *steps* a *scales*. *Steps* udává počty dávek, po kterých se má *learning rate* aktualizovat a *scales* definuje, jak se má *learning rate* po těchto dávkách změnit. Pro

```
steps = 100, 500
scales = 10, .1
```

se po zpracování 100 dávek learning rate zvýší desetkrát a po dalších 500 dávkách se learning rate dostane zpět na původní hodnotu.

Volba optimální *learning rate* je věda sama o sobě. Výchozí hodnoty parametrů *learning rate, steps* a *scales* jsou zvoleny tak, aby byly použitelné ve většině případů. Při trénování vlastního modelu jsem tedy využíval převážně tyto výchozí hodnoty.

Za zmínku dále stojí parametry *angle* a *random*. *Angle* definuje maximální úhel, o který mohou být jednotlivé obrázky otočeny. Výchozí hodnota je 0. *Random* může nabývat pouze hodnot 0 a 1. Pokud je *random* 1, při trénování se vždy po několika dávkách změní rozlišení obrázků. Jelikož však můj model pracuje pouze s obrázky s daným rozlišením (mobilní aplikace poskytuje modelu obrázky ke zpracování vždy ve stejném rozlišení) a rotaci obrázků řeším přímo při tvorbě datasetu, tyto parametry nevyužívám.

Výchozí soubory pro trénování *yolo.weights* a *yolo.cfg* jsou dostupné na stránkách autora[1]. Pro mobilní využití je vhodnější odlehčená verze yolov2-tiny, ze které jsem při implementaci aplikace vycházel.

4.2 Zahájení trénování

Pro shrnutí: v tuto chvíli mám nainstalovaný framework Darkent (ideálně GPU verzi), vytvořený a správně anotovaný dataset, stažený předtrénovaný soubor *yolo.weights* a správný konfigurační soubor, především se správným počtem tříd, které chci detekovat. Můj trénovací dataset tvoří 10000 obrázků s několika desítkami tisíc chemických a matematických rovnic. Dále potřebuji výše zmiňovaný soubor *.names* s názvy tříd. Také potřebuji soubor *train.txt* obsahující relativní umístění (vzhledem ke spustitelnému Darknetu) obrázků datasetu na disku, každý na samostatném řádku. Počet řádků tohoto souboru by se tedy měl shodovat s počtem obrázků v datasetu a jednotlivé řádky by měli vypadat takto:

```
relative/path/datasetimg0001.png
```

V neposlední řadě potřebuji soubor *.data*, jehož obsah může vypadat takto:

```
classes = 20
train = data/train.txt
names = data/model.names
backup = backup/
```

Classes odpovídá počtu tříd (stejný jako v konfiguračním souboru modelu). *Train* a *names* udává opět relativní lokaci daných souborů a *backup* definuje umístění složky, do které chci ukládat mezivýsledky v podobě souborů *.weights*. Standardně se mezivýsledky ukládají po 100 dávkách.

Nyní by již mělo být vše připraveno. Trénování je možné zahájit příkazem

```
./darkent detector train data/model.data yolo.cfg yolo.weights
```

Model.data je soubor zmiňovaný výše, jmenovat se samozřejmě může různě. Na systémech s Windows vypadá příkaz stejně, pouze je potřeba místo *./darknet* uvést *darknet.exe*. Chceme-li pokračovat v přerušeném trénování, stačí vzít poslední uložený *.weights* soubor z backup složky a nahradit jím výchozí *.weights* soubor. Po úspěšném načtení modelu by se mělo zobrazit (v závislosti na modelu) něco takového:

```
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16 0.150 BF
    1 max          2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16 0.003 BF
    2 conv     32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32 0.399 BF
    3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32 0.001 BF
    4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64 0.399 BF
    5 max          2 x 2 / 2   104 x 104 x  64   ->    52 x  52 x  64 0.001 BF
    6 conv    128  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x 128 0.399 BF
    7 max          2 x 2 / 2    52 x  52 x 128   ->    26 x  26 x 128 0.000 BF
    8 conv    256  3 x 3 / 1    26 x  26 x 128   ->    26 x  26 x 256 0.399 BF
    9 max          2 x 2 / 2    26 x  26 x 256   ->    13 x  13 x 256 0.000 BF
   10 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512 0.399 BF
   11 max          2 x 2 / 1    13 x  13 x 512   ->    13 x  13 x 512 0.000 BF
   12 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024 1.595 BF
   13 conv   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024 3.190 BF
   14 conv     30  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x  30 0.010 BF
   15 detection
```

Obrázek 4.2. Příklad správného načtení modelu. Jedná se o model yolov2-tiny s 16 vrstvami bez vstupní. Rozlišení první konvoluční vrstvy se shoduje s nastaveným rozlišením.

[1] https://pjreddie.com/darknet/yolo/

Nejčastější chybou, která se v této fázi objevuje je:

```
  13 CUDA Error: out of memory
CUDA Error: out of memory: No error
```

Jedná se o zmiňovaný problém s nedostatkem paměti GPU na zpracování přidělených subdivizí. Ideální řešení je zvýšení parametru *subdivisions* o faktor 2. Alternativně lze snížit velikost dávky.

4.3 Kdy ukončit trénování

Po zpracování každé dávky se zobrazí informativní výpis o pokroku.

```
 3703: 0.064152, 0.044411 avg loss, 0.001000 rate, 0.500952 seconds, 236992 images
Loaded: 3.698349 seconds
Region Avg IOU: 0.738359, Class: 1.000000, Obj: 0.737774, No Obj: 0.002313, Avg Recall: 0.888889,  count: 9
Region Avg IOU: 0.698048, Class: 1.000000, Obj: 0.644349, No Obj: 0.000775, Avg Recall: 1.000000,  count: 1
Region Avg IOU: 0.797191, Class: 1.000000, Obj: 0.757293, No Obj: 0.003580, Avg Recall: 1.000000,  count: 14
Region Avg IOU: 0.808573, Class: 1.000000, Obj: 0.789233, No Obj: 0.005694, Avg Recall: 1.000000,  count: 22
Region Avg IOU: 0.879697, Class: 1.000000, Obj: 0.814925, No Obj: 0.002167, Avg Recall: 1.000000,  count: 7
Region Avg IOU: 0.718929, Class: 1.000000, Obj: 0.736790, No Obj: 0.002413, Avg Recall: 0.900000,  count: 10
Region Avg IOU: 0.813867, Class: 1.000000, Obj: 0.786821, No Obj: 0.005507, Avg Recall: 1.000000,  count: 20
Region Avg IOU: 0.785446, Class: 1.000000, Obj: 0.748328, No Obj: 0.002371, Avg Recall: 1.000000,  count: 9
```

Obrázek 4.3. Výpis po zpracování 3703. dávky.

V příkladu z obrázku pracuji s parametry *batch*=64 a *subdivisions*=8. Jak je vidět, dávka byla opravdu rozdělena na $64 \div 8 = 8$ bloků, každý sestávající z 8-mi obrázků, které byly zpracovány samostatně. Výsledky zpracování těchto subdivizí lze vidět na řádcích začínající slovem Region.

Co ale tento výpis znamená?
IOU neboli *Intersection over Union* udává, jak název napovídá, podíl průniku dvou oblastí ku jejich sjednocení. Jedná se o oblast predikovanou modelem a oblast definovanou anotací v datasetu. *Region Avg IOU* je tedy průměrné *IOU* v dané subdivizi. Čím vyšší je hodnota *IOU*, tím přesněji detekuje model umístění objektu v obrázku.

Přesným výpočtem *Class*, *Obj* a *No Obj* si nejsem zcela jistý. Nicméně *Class* určuje přesnost klasifikace objektů a *Obj* průměrný *confidence score* správně detekovaných objektů v subdivizy. Tyto parametry by měly mít ideálně co nejvyšší hodnotu, maximálně pak hodnotu 1. Naopak *No Obj* určuje průměrný *confidence score* nesprávných detekcí (protože, jak je vysvětleno v sekci fungování YOLO, každá buňka musí povinně predikovat určitý počet rámečků, i když žádný objekt v dané buňce neleží) a jeho hodnota by měla být co nejnižší.

4.3.1 Ztrátová funkce

Nejpodstatnějšími informacemi je určitě *loss* a *avg loss* zobrazené v daném pořadí ihned na první řádku, po číslu dávky. Zatímco *IOU*, *Class*, *Obj*, *No Obj* hodnotí přesnost detekce vždy jen z určitého úhlu pohledu, tak ztrátová funkce (*loss function*) zohledňuje všechny tyto pohledy a spojuje je do jedné složitější, ale elegantní funkce.

$$Loss_{yolo} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2]$$

$$+\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$

$$+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbf{1}_{ij}^{obj}(C_i-\hat{C}_i)^2+\lambda_{noobj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbf{1}_{ij}^{noobj}(C_i-\hat{C}_i)^2$$

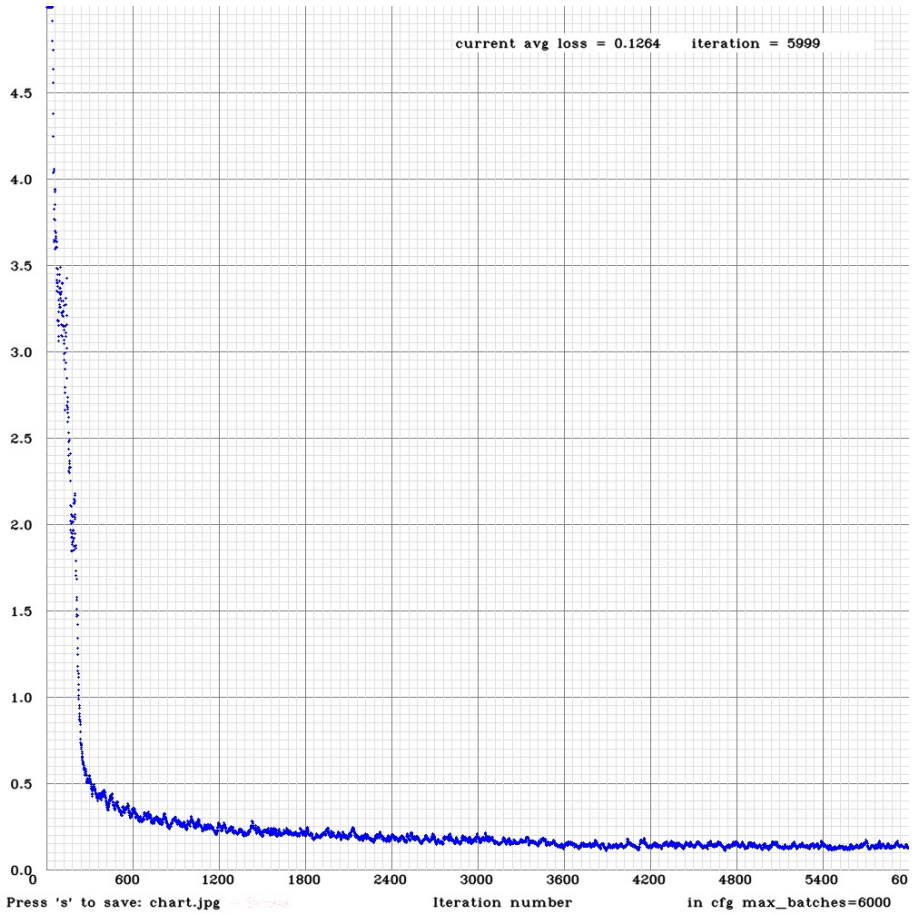
$$+\sum_{i=0}^{S^2}\sum_{c\in classes}(p_i(c)-\hat{p}_i(c))^2$$

S a B (stejně jako v sekci Fungování YOLO) určují po řadě velikost hrany mřížky a počet predikcí jedné buňky. Lambdy jsou pouze koeficienty určující váhu jednotlivých částí funkce a $\mathbf{1}_i^{obj}$ nabývá hodnoty 1, když objekt spadá do buňky j a $\mathbf{1}_{ij}^{obj}$ nabývá hodnoty 1, když rámeček j určený buňkou i je „zodpovědný“ za danou predikci.

První dva řádky ztrátové funkce zohledňují přesnost rámečku ohraničující detekovaný objekt (rozdíl mezi predikovanými souřadnicemi, predikovanými rozměry a souřadnicemi, rozměry určenými v anotaci trénovacích dat), třetí řádek zohledňuje *confidence score* a poslední řádek klasifikaci objektu.

Ztrátová funkce neslouží pouze k hodnocení přesnosti detekce. Pokud na trénování modelu budeme pohlížet jako na optimalizační problém, je účelovou funkcí, kterou se snažíme během trénování minimalizovat.

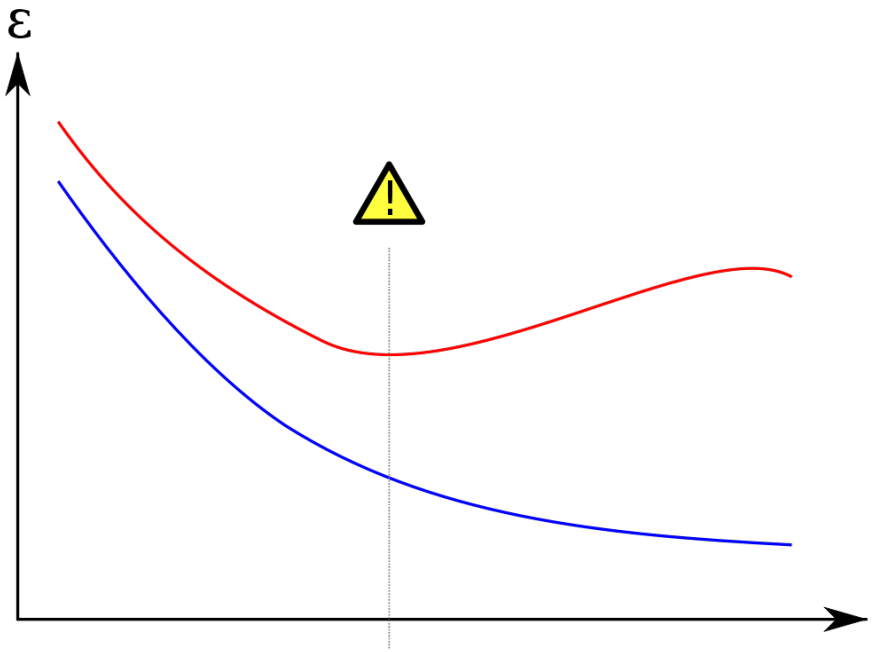
Při zahájení trénování je hodnota *loss* vysoká, v řádu několika stovek. V průběhu trénování by se měla postupně snižovat. Není potřeba se děsit, zvýší-li se hodnota vzhledem k předchozí dávce. Nemělo by se tak ale dít dlouhodobě hodnotě *avg loss*, která se počítá jako průměr *loss* několika posledních dávek.



Obrázek 4.4. Graf zachycující klesající tendenci *loss* v průběhu trénování vlastního modelu. Zde *max batches* = 6000

S trénováním by se mělo skončit, když se *avg loss* po mnoha iteracích dále nesnižuje, automaticky skončí po dosažení *max batches* definovaného v konfiguraci modelu. Minimální hodnota *avg loss* se v závislosti na modelu a složitosti datasetu může pohybovat od 0.01 až po několik jednotek. Ideální je dále mít menší testovací dataset, který není shodný s trénovacím a na kterém lze výsledný model otestovat.

Častým problém spojeným s delším trénováním je tzv. *overfitting* [14]. Může se totiž stát, že ačkoliv hodnota *avg loss* nadále klesá, nově vzniklé modely mají na testovacím datasetu horší výsledky než modely vzniklé o několik set iterací dříve s vyšším *avg loss*. Nový model je totiž příliš přizpůsoben trénovacímu datasetu a začíná ztrácet na obecnosti.



Obrázek 4.5. Znázornění fenoménu *overfitting*. Modrá křivka znázorňuje pokračující pokles *loss* na trénovacím datasetu. Červená křivka reprezentuje *loss* na testovacím datasetu. Lze vidět, že ačkoliv se model na trénovacím datasetu nadále „zlepšuje“, je vhodné trénink ukončit již dříve, u minima testovacího datasetu.[1]

Po ukončení tréninku tedy doporučuji otestovat několik posledních uložených modelů (například 10 modelů z posledních 1000 iterací) a použít ten nejoptimálnější.

4.4 Tipy na zlepšení detekce

Nejčastější příčinou problémů při detekci bývá špatně zkonstruovaný trénovací dataset. Je velmi důležité, aby opravdu každý objekt, který chceme detekovat, byl správně anotován. Je-li v obrázku viditelná pouze část objektu je na zvážení, zda tento vzorek z datasetu vyloučit nebo anotovat viditelnou část, rozhodně však není vhodné jej v datasetu nechat bez anotace.

Velmi důležité jsou negativní vzorky. Chci-li například vycvičit model na detekci pejsků, budu mít v datasetu vedle obrázků pejsků i obrázky lišek, kočiček a dalších čtyřnohých pejskům podobných zvířat. Chci totiž model naučit, že ačkoliv je liška dost podobná pejskovi, pejsek to není.

[1] Obrázek převzat z `https://en.wikipedia.org/wiki/Overfitting`

Jak jsem již zmiňoval v sekci tvorby datasetu, je potřeba pokrýt co nejširší množství situací, ve kterých se objekt může vyskytovat. Můj dataset pejsků by měl obsahovat obrázky různých druhů psů focených pod různým osvětlením, pod různými úhly a z různé vzdálenosti. Zjistím-li pak při testování, že model nedokáže rozpoznat psa zezadu, můj trénovací dataset pravděpodobně neobsahuje dostatek takovýchto vzorků.

Při problémech s přesností detekce či nefunkční detekce drobnějších objektů je vhodným krokem zvýšení rozlišení vstupní vrstvy modelu. Výšku a šířku rozlišení je vždy potřeba měnit o celočíselný násobek 32, jelikož rozlišení výstupní $S \times S$ mřížky je počítáno ze vstupního rozlišení podělením 32. Vstupní rozlišení 416×416 mohu zvýšit na 512×512 (výstupní mřížka pak bude 16×16), nikoliv však na 520×520. Se zvýšením rozlišení se samozřejmě zvyšuje i výpočetní náročnost. V mobilní aplikaci, kde jsou zdroje omezené, zvyšování rozlišení nepřipadá v úvahu.

Další zvýšení přesnosti detekce lze zajistit přepočítáním *anchor boxů* algoritmem *k-means* [15] z datasetu. Velikosti rámečků definované anotací datasetu můžeme pomocí tohoto algoritmu seskupovat do shluků (s YOLOv2 do 5-ti a s YOLOv3 do 9-ti). Těžiště těchto shluků se pak stanou výslednými *anchor boxy*.

Přepočet *anchor boxů* je vhodný zejména, když objekty k detekci mají neobvyklý tvar (jako například chemické a matematické rovnice, které bývají v obrázcích horizontálně protáhlé), nebo při změně vstupního rozlišení modelu, na který se *anchor boxy* vážou. Naštěstí není potřeba algoritmus *k-means* zvlášť programovat. Jeho výpočet je totiž přímo implementovaný v Darknetu. Lze jej pustit příkazem

```
./darknet calc_anchors data/model.data -num_of_clusters 5 -width 416 -height 416
```

Num of clusters je počet shluků a samozřejmě výsledný počet *anchor boxů*. Navíc se musí shodovat s počtem rámečků, které každá buňka předpovídá. *Width* a *height* je vstupní rozlišení modelu a *model.data* je stejný soubor jako při trénování. Vypočtené rozměry se vypíší přímo v terminálu a zároveň uloží do souboru. Stačí je pak zkopírovat do konfiguračního souboru modelu. Při zpracování výstupního tensoru po detekci se ovšem musí počítat s novými hodnotami.

Kapitola 5
Tesseract

Tesseract je open-source OCR engine distribuovaný pod Apache licencí. Původně byl vyvíjen společností Hewlett Packard jako proprietární software. V roce 2005, po několika letech stagnace, byl vypuštěn do světa jako open-source. Od roku 2006 do dnešního dne sponzoruje jeho vývoj společnost Google [16]. Tesseract je dle mého názoru jedním z nejlepších zdarma dostupných enginů. Lze jej využívat přímo, jako program v CLI, nebo nepřímo skrze jednoduché API. Ostatní bezplatné enginy jsou často přímo na Tesseractu založeny (FreeOCR, WeOCR, OCRFeeder) a jsou v podstatě jen grafickým uživatelským rozhraním neposkytující žádnou novou funkcionalitu, nebo jsou omezeny pouze na jednu platformu. Pro mě největší výhodou je pak existence *tess-two*[1], vývojové větve Tesseractu pro android. Jediným mně známým konkurentem tesseractu (obzvlášť na poli mobilních zařízení) je ML Kit OCR, který je vyvíjen Googlem. Při srovnávacích testech [17] sice ML Kit OCR mírně poráží Tesseract 3.05, avšak na rozdíl od Tesseractu jej nelze jednoduše vytrénovat pro vlastní (klidně smyšlený) jazyk, čehož u rozpoznávání rovnic využiji.

Ke dnešnímu dni je nejnovější verze Tesseract 4.0.0 vydaná v říjnu 2018. Tato verze již pracuje s tzv. *Long short-term memory*, speciální architekturou neuronové sítě. Bohužel *tess-two* podporuje zatím pouze verzi 3.05. K účelům této práce by ale měla být dostačující.

5.1 Tesseract API

Pro lepší pochopení fungování tesseractu se pokusím krátce popsat základní funkce jeho API. Na konrétní platformě ani programovacím jazyku moc nezáleží, základ je všude stejný a odlišnosti jsou vesměs jen formální. Příklady budu uvádět na *tess-two* pro android.

Po vytvoření instance TessBaseApi (třída v balíčku Tesseractu poskytující pohodlný přístup k metodám) je potřeba engine inicializovat, tedy nastavit jazyk.

```
tessBaseApi.init(DATAPATH, "lang");
```

K nastavení jazyku musíme mít k dispozici soubor obsahující model daného jazyka: `lang.traineddata`. Na oficiální stránce enginu[2] je k dispozici většina běžných jazyků, včetně češtiny. Není ale problém si Tesseract vytrénovat pro vlastní jazyk (alespoň do verze 3.05, ve verzi 4.0.0 je již trénování složitější). `DATAPATH` je pouze řetězec definující cestu ke složce obsahující potřebná data.

Nyní je možné nastavit metodu segmentace stránky. Segmentace určuje, jaký typ vstupu Tesseract očekává: zda celou stránku textu, nebo homogenní blok textu, samostatný řádek, pouze slovo atd. Vhodná volba by měla zvýšit přesnost rozpoznávání. Dle dokumentace existuje 13 metod segmentace. Nastavit ji lze jednoduše příkazem

[1] `https://github.com/rmtheis/tess-two`
[2] `https://github.com/tesseract-ocr/tessdata`

```
tessBaseAPI.setPageSegMode(TessBaseAPI.PageSegMode.PSM_SINGLE_LINE);
```

Jediným argumentem je typ segmentace. Zde se jedná konkrétně o řádkovou segmentaci. Ta se hodí právě pro chemické rovnice, které jsou vždy zachycené na jediném řádku. Výchozí nastavení enginu je segmentace podle stránky.

Rozpoznávání lze dále zlepšit definicí povolených nebo zakázaných znaků. Například anglický jazykový model obsahuje kromě latinských znaků i některé znaky speciální. Pokud víme, že se v rozpoznávaném textu některé tyto znaky nemohou vyskytovat, je možné je zakázat.

```
tessBaseAPI.setVariable("tessedit_char_blacklist", "&@");
```

Tento příkaz blokuje detekci znaků & a @. Pokud by například engine v obrázku zaměňoval 'a' s '@' (může se stát kvůli šumu), rozpoznávání s takto definovanými nepovolenými znaky by již bylo správné.

Samotný obrázek, na kterém chceme provádět rozpoznávání, se vkládá příkazem

```
tessBaseAPI.setImage(image);
```

Konkrétně v androidu a Java musí být obrázek instancí třídy Bitmap.

Je-li v obrázku více oblastí, které chceme rozpoznávat separátně, není potřeba je vyřezávat a zadávat zvlášť. Efektivnější je zadat obrázek pouze jednou a poté jen definovat tyto (obdélníkové) oblasti.

```
tessBaseAPI.setRectangle(x, y, width, height);
```

Konečně výsledný rozpoznaný text dostaneme z enginu příkazem

```
String result = tessBaseAPI.getUTF8Text();
```

Text ale nemusí být rozpoznán správně. Pro skoro každý znak detekovaný znak navrhuje Tesseract několik možných rozpoznání s různou pravděpodobností (*confidence score* jako u YOLO). `getUTF8Text()` vrací tu celkově nejpravděpodobnější variantu. Při tom je zohledněn slovník, pokud je tedy nějaký k dispozici. Iterátor možností rozpoznání pro celý text lze z Tesseractu extrahovat. Z iterátoru pak lze, například ve while cyklu, získat možnosti pro jednotlivé znaky jako seznam párů znak - *confidence*.

```
ResultIterator ri = tessBaseAPI.getResultIterator();
do {
    List<Pair<String, Double>> characterChoices =
    ri.getChoicesAndConfidence(level);
} while (ri.next(level));
```

Parametr *level* určuje úroveň, na které iterátor pracuje. Teoreticky je možné iterovat přes celá slova nebo věty. V androidu mi ovšem správně fungovala pouze iterace přes jednotlivé znaky. Iterátor využiji při parsování chemických rovnic, kdy jsem při znalosti automatu, který rovnici přijímá, schopen eliminovat mnoho chybných rozpoznání.

5.2 Trénování enginu pro rozpoznání rovnic

Proces trénování Tesseractu je detailně popsán na jeho GitHub repositáři[1]. Cílem je vytvoření souboru `language.trainedata`, jenž slouží jako jakýsi model pro rozpoznávání znaků. Manuální trénování je poměrně složitý a zdlouhavý proces. Pokusím se ho v několika větách shrnout.

[1] https://github.com/tesseract-ocr/tesseract/wiki/Training-Tesseract-3.03%E2%80%933.05

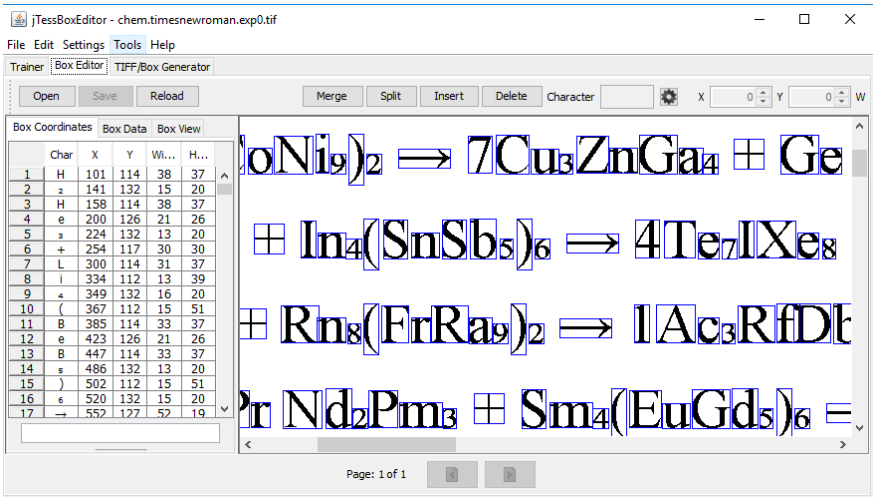
Nejprve musíme vytvořit vzorový text obsahující všechny znaky (ideálně alespoň 5 krát), které chceme detekovat. Musíme se rozhodnout, jaké fonty při trénování použijeme. Ze vzorového textu pak vygenerujeme obrázky s daným fontem a soubory s anotacemi (podobně jako v YOLO, co znak, to anotace). S obrázky a anotacemi můžeme spustit trénink, opět pro každý font zvlášť. Nakonec výsledky pro jednotlivé fonty seshlukujeme do prototypů. Dále potřebujeme soubory s daty o znacích, které detekujeme a o fontech, na kterých bylo trénování prováděno. Pokud chceme, můžeme navíc vytvořit několik typů slovníků pro daný jazyk, například slovník nejfrekventovanějších slov.

Naštěstí si lze práci značně usnadnit. Existuje totiž nástroj jTessBoxEditor[1], který velkou část procesu automatizuje. Trénování pro rozpoznávání chemických rovnic jsem prováděl právě v něm.

Některým částem procesu se ale ani s tímto nástrojem nelze vyhnout. Vzorový text, na kterém bude prováděno trénování, je stále potřeba vytvořit. Dle popisu tréninku v repositáři je pro vzorový text lepší, když znaky, které chceme detekovat, uvádíme organicky, v kontextu, jakém se budou reálně vyskytovat. Místo seskupení všech znaků, čísel, indexů a šipky jsem tedy vytvořil jednoduchý skript v Pythonu, který do souboru zapisoval celé chemické a matematické rovnice a který pokryl celou množina znaků, kterou potřebuji rozpoznávat. Navíc každý znak musel být uveden alespoň 5 krát. Některé znaky latinské abecedy vůbec nemusely být v textu obsaženy, jelikož se nevyskytují ve značce žádného prvku. Po výběru fontu se automaticky vygenerují obrázek s anotací a uloží se do zvolené složky. Tesseract 3.05 lze vytrénovat na maximálně 64 fontech. Jejich správný výběr je důležitý, pro chemické rovnice jsem se snažil volit fonty, ve kterých se nejčastěji tisknou učebnice.

Po vygenerování dostatečného množství trénovacích dat stačí v aplikaci nastavit složku, ve které jsou data, tedy anotované obrázky s textem, uloženy a lze zahájit trénování. Není potřeba vytvářet soubory obsahující informace o rozpoznávaných znacích a trénovacích fontech jako při manuálním trénování.

jTessBoxEditor neusnadňuje vytváření slovníků. Pro rozpoznávání chemických rovnic jsem se rozhodl žádný slovník nepoužít. Bylo by sice možné vytvořit slovník nejčastějších sloučenin (například podle [12]), ty se však stejně většinou v rovnicích vyskytují s číselným prefixem. Navíc budu existenci sloučenin ověřovat při parsování rovnice.



Obrázek 5.1. Vygenerovaný anotovaný obrázek pro font Times New Roman zobrazený v aplikaci jTessBoxEditor

[1] `http://vietocr.sourceforge.net/training.html`

5.3 Tipy na zlepšení rozpoznávání

Ačkoliv Tesseract před samotným rozpoznáváním obrázky předběžně zpracovává (preprocessuje), často to ke kvalitní detekci nestačí. Dobrý návod pro zkvalitnění detekce je k dispozici přímo na repozitáři Tesseractu[1]. V této sekci shrnu základní poznatky.

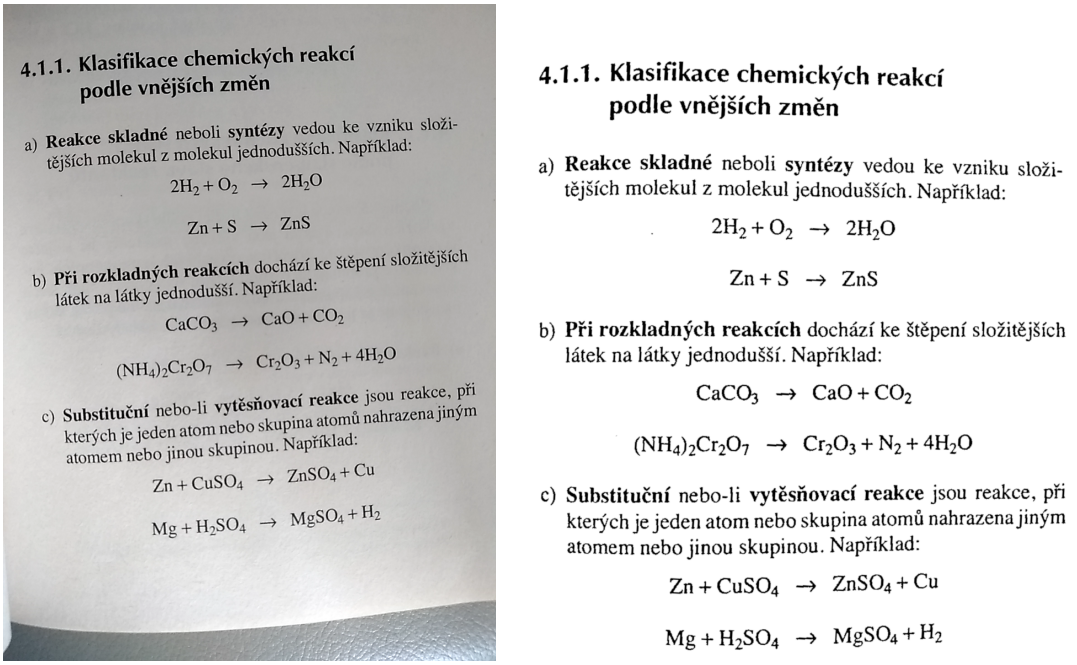
Ze všeho nejdříve by se mělo zajistit, aby vstupní obrázek měl alespoň 300 dpi (*dots per inch*, počet pixelů na palec). S příliš malými obrázky sice engine rozpoznání stále provede, ale na problém nás upozorní varováním. Zajištění vhodné velikosti je tedy prvním krokem ke kvalitnějšímu rozpoznávání.

Rozpoznání může dále velmi negativně ovlivnit rotace textu na vstupním obrázku. Rotace do 3 až 4 stupňů by detekci nijak zásadně ovlivnit neměla, ale s extrémními úhly engine nemusí fungovat vůbec. Problém můžeme vyřešit výpočtem úhlu (pomocí OpenCV), o který je obrázek zkosený a následnou rotací obrázku o daný úhel [18].

Binarizace je převedení obrázku pouze na 2 barvy, černou a bílou. Tesseract interně využívá tzv. *Otsuovu binarizační metodu*. Ta ovšem nemusí být vhodná pro obrázky, ve kterých je pozadí různě tmavé. V takových případech neuškodí vlastní binarizace. Konkrétně pro případy s nerovnoměrným pozadím se hodí *Gaussovská binarizace* implementovaná v OpenCV.

Binarizace často nestačí k odstranění šumu z obrázku. Šum samozřejmě také negativně ovlivňuje rozpoznávání, je tedy dobré jej nějakým způsobem, například opět pomocí OpenCV, odstranit.

V neposlední řadě je dobré obrázek s textem osekat o zbytečné části. Například při skenování dokumentu se do výsledného obrázku často dostanou černé pruhy na místech, kde papír přesně do skeneru nezapadl. Podobné pruhy vznikají z pozadí při binarizaci. Tyto tmavé pruhy mají také neblahý vliv na výslednou kvalitu rozpoznávání, engine v nich často vidí domnělé znaky.



4.1.1. Klasifikace chemických reakcí podle vnějších změn

a) **Reakce skladné** neboli **syntézy** vedou ke vzniku složitějších molekul z molekul jednodušších. Například:

$$2H_2 + O_2 \rightarrow 2H_2O$$

$$Zn + S \rightarrow ZnS$$

b) **Při rozkladných reakcích** dochází ke štěpení složitějších látek na látky jednodušší. Například:

$$CaCO_3 \rightarrow CaO + CO_2$$

$$(NH_4)_2Cr_2O_7 \rightarrow Cr_2O_3 + N_2 + 4H_2O$$

c) **Substituční** nebo-li **vytěsňovací reakce** jsou reakce, při kterých je jeden atom nebo skupina atomů nahrazena jiným atomem nebo jinou skupinou. Například:

$$Zn + CuSO_4 \rightarrow ZnSO_4 + Cu$$

$$Mg + H_2SO_4 \rightarrow MgSO_4 + H_2$$

Obrázek 5.2. Obrázek před a po předběžném zpracování. Původní obrázek byl binarizován, oříznut a nakonec „napřímen“ (*deskewing*).

[1] https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality

Kapitola 6
Syntaktická analýza chemické rovnice

V této kapitole se budu zabývat tvorbou syntaktického analyzátoru, zejména pak teoretickým základem, který je k jeho správné implementaci potřebný. Řešit budu pouze problematiku parsování chemických rovnic. Jelikož je struktura matematických rovnic (na úrovni 2. třídy ZŠ) mnohem jednodušší než těch chemických, nebudu se jimi zabývat. Vše potřebné je možné si odvodit.

Parser bude rozpoznávat pouze správnou strukturu chemické rovnice, nikoliv zda je počet a druh chemických prvků stejný na levé i pravé straně.

6.1 Automat přijímající chemické rovnice

Znalost automatu přijímajícího chemické rovnice a gramatiky generující jazyk je pro správné sestrojení parseru klíčové. Může se sice povést jej sestrojit i bez znalosti automatu a gramatik, je to však mnohem obtížnější a je těžké se vyhnout chybám. Zároveň ke sestrojení automatu svádí tento problém přirozeně.

6.1.1 Vstupní symboly

Před návrhem automatu je dobré si ujasnit množinu vstupních symbolů, abecedu.

V případě automatu přijímajícím chemické rovnice lze některé skupiny symbolů považovat za zcela ekvivalentní. Třeba veškeré prvky lze z hlediska automatu považovat za ekvivalentní symboly. Automat totiž bude sloužit pouze jako syntaktický analyzátor, sémantikou („smysluplností“, zda jsou na levé straně stejné prvky jako na pravé a sloučeniny dávají smysl) se zabývat nebude. Z hlediska syntaxe jsou tedy všechny prvky ekvivalentní, stejně jako všechny dolní indexy a všechna prefixová čísla. Pokud bychom se zabývali gramatikou, prefixy, indexy a prvky by odpovídaly některým neterminálům, zatímco konkrétní znaky by byly terminály a existovala by pravidla, která zmíněné neterminály přepisují na odpovídající terminály. Vzhledem k implementaci analyzátoru pomocí stavového automatu mi ovšem přijde názornější zabývat se pouze automatem.

Například $2(NH_4)_2Cr_2O_7$, dichroman amonný, by ze syntaktického hlediska stačilo reprezentovat jako Prefix(Prvek Prvek_{index})$_{index}$$\text{Prvek}_{index}$$\text{Prvek}_{index}$. Na konkrétních prvcích a číslech nezáleží.

Bohužel situace není tak jednoduchá. Většina značek prvků se skládá ze 2 písmen. Tesseract navíc rozpoznává text po znacích, ne po prvcích. Je tedy potřeba vymyslet způsob, jak reprezentovat prvky po jednotlivých znacích, ale zároveň u toho nebýt moc konkrétní (nemít každý rozpoznatelný znak za vstupní symbol), aby abeceda nebyla zbytečně obsáhlá a automat zbytečně složitý.

Místo jediného hromadného symbolu „Prvek“ zavedu symboly 3. První je velké písmeno, které je samo o sobě prvkem (například H, C, I, ...). Druhé je velké písmeno, které samo o sobě prvkem není (například A, L, M, ...). Třetí je malé písmeno, které spolu s předchozím přečteným písmenem tvoří prvek. Tento poslední hromadný symbol

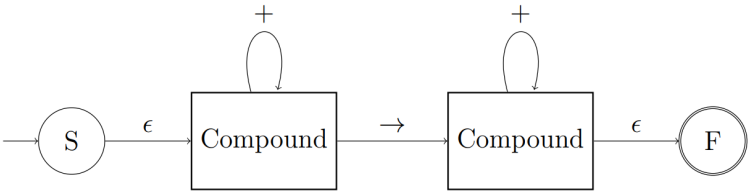
je zvláštní v tom, že kvůli závislosti na předchozím znaku nelze zapsat jako množinu elementárních symbolů. Stejné malé písmeno do něj totiž jednou patřit může a podruhé nemusí. Dále k němu pro jednoduchost budu přistupovat stejně, jako ke zbytku symbolů. V implementaci automatu stačí po přečtení malého písmena zkontrolovat, zda s předchozím symbolem tvoří prvek.

Vstupní symboly, se kterými budu nadále pracovat jsou pro shrnutí:

```
Velké písmeno, které je samo o sobě prvkem (UE, Uppercase Element)
Velké písmeno, které samotné prvkem není (UN, Uppercase Not element)
Malé písmeno, které s předchozím  znakem tvoří prvek (L, Lowercase)
Číslo (N, Number)
Číselný dolní index (I)
Elementární symboly +, (, ) a šipka
```

6.1.2 Návrh automatu

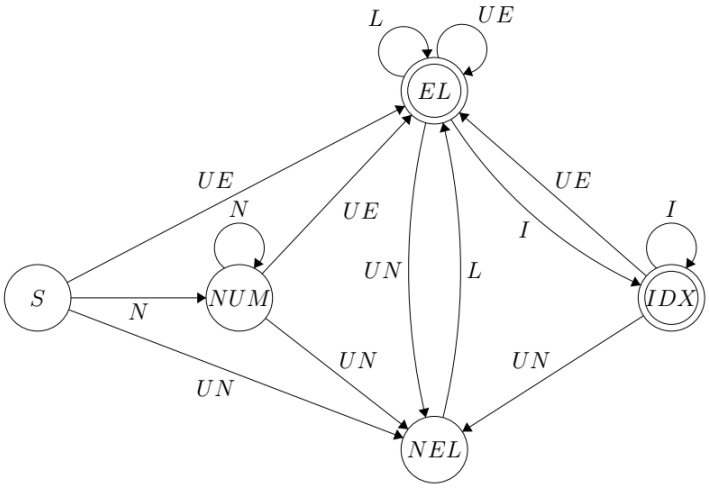
Pro lepší znázornění rozdělím automat na vnější a vnořenou vnitřní část. Vnější bude zpracovávat celou chemickou rovnici. Ke zpracování jednotlivých molekul bude využívat vnořený vnitřní automat.



Obrázek 6.1. Vnější automat využívající vnitřní ke zpracování jednotlivých sloučenin

Z obrázku je zřejmé rozdělení na levou a pravou část rovnice. Na každé straně musí být alespoň jedna sloučenina. Z koncových stavů vnořeného automatu se po přečtení symbolu + dostáváme do vstupního stavu vnořeného automatu. Po přečtení symbolu → se z koncového stavu levého vnořeného automatu dostaneme do počátečního stavu vnořeného automatu na pravé straně rovnice. Nakonec, nachází-li se automat v koncovém stavu vnořeného automatu na pravé straně rovnice a zároveň neexistuje další symbol k přečtení, pak se podařilo úspěšně přijmout rovnici.

Vnořený automat je trošičku složitější, provádí většinu práce.



Obrázek 6.2. Vnitřní automat přijímající sloučeninu

Stav *NUM* reprezentuje číselný prefix sloučeniny. Ten může být pouze na začátku a může mít libovolný počet cifer.

Ve stavu *EL* se automat nachází, jen pokud doposud přečtená sloučenina končí na validní, existující prvek. To je buď po přečtení znaku, který je sám o sobě prvkem, nebo po přečtení malého písmene, které spolu s předchozím znakem tvoří prvek. *EL* je spolu s *IDX* koncovým stavem, validní sloučenina totiž může končit na prvek nebo index (nebo závorku, ale o tom později). V *IDX* je smyčka, jelikož index může být také několika ciferný.

Nakonec stav *NEL* značí situaci, kdy poslední přijatý symbol je velké písmeno, které není prvkem. Aby čtení neskončilo, další znak musí být malé písmeno, které s předchozím písmenem vytvoří znak prvku.

Možná jste si všimli, že automat nepřijímá v žádném stavu závorky. V implementaci samozřejmě se závorkami počítám, do grafu jsem je ale pro přehlednost nekreslil. Po přečtení závorky bychom se totiž dostali do paralelního automatu, který je téměř shodný se zobrazeným. Chybí mu akorát stav *NUM*, protože pokud vím, tak po závorce nemůže následovat číslo. Ekvivalenty jeho koncových stavů by nebyly koncové, vedli by z nich ale hrany označené ')' do stavu *EL*. Do tohoto paralelního závorkového automatu by vedly hrany označené '(' ze stavů *S, NUM, EL* a *IDX*.

6.2 Implementace syntaktického analyzátoru

Se znalostí automatu přijímajícím jazyk by již mělo být jednodušší implementovat parser. Situaci ale komplikuje práce s *ResultIterator*em vráceným tesseractem. V pořadí na každé písmeno rozpoznaného slova máme několik možností znaků. Z nich potřebujeme, pokud možno efektivně, vybrat kombinaci znaků, které automat přijme jako rovnici.

Jako první mě napadl upravený *bruteforce* algoritmus. Oproti běžnému *bruteforce* bych nemusel vytvořit kombinaci, a pak zkoušet, zda-li funguje. Už v průběhu vytváření kombinace by automat kontroloval, zda dosud přečtené znaky dávají `smysl`. Například pokud by bylo výchozí rozpoznání molekuly vody 'H2O', parser by ihned po přečtení znaku '2' rozpoznal, že se někde vyskytla chyba, jelikož po znaku 'H' se nemůže objevit číslo psané velkým písmenem. Další možný znak na druhé pozici by byl $_2$ a parsování by po dočtení skončilo úspěšně.

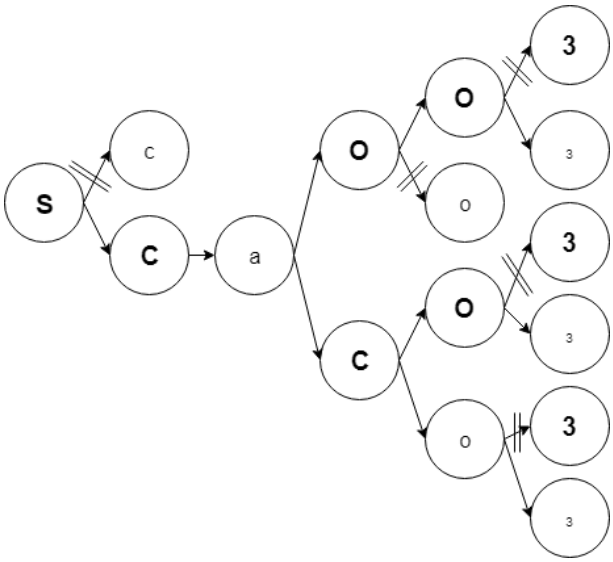
Popisovaný analyzátor jsem implementoval. Fungoval poměrně dobře, měl ale jeden menší nedostatek. Vracel vždy jen první nalezenou kombinaci znaků, které automat přijal. Ta ale nemusí být nejlepší, nemusí mít v součtu nejvyšší *confidence score* (který tesseract ke každému možnému znaku udává). Navíc tesseract si většinou plete znaky, které si jsou podobné a které bychom mohli chybně rozpoznat i my, lidé. Nejčastěji zaměňuje velká čísla za indexy, velké a malé 'O', 'o', velké 'I' a malé 'l' atd. Nebylo by tedy od věci vybrat několik nejlepších (s nejvyšším *confidence score*) syntakticky validních kombinací a zkusit je porovnat proti nějaké databázi nejfrekventovanějších sloučenin, například [12]. Pokud by se některá kombinace shodovala se sloučeninou z databáze, s velikou pravděpodobností by se jednalo o tu správnou.

Algoritmus, kterým získám ohodnocené syntakticky správné možnosti je docela jednoduchý. Jedná se v podstatě jen o procházení stromu, kde díky automatu můžu odsekávat neperspektivní větve. Hledám validní cesty od kořene do listu a zároveň počítám jejich *confidence score*.

Vše se nejlépe předvádí na příkladu. Nechť tedy z *ResultIterator*u obdržíme možnosti pro pozice znaků ve slově reprezentované následující tabulkou.

1	2	3	4	5
c C	a	O C	O o	3 $_3$

Řádky tabulky jsou seřazené podle *confidence score*. Přikazem `getUTF8Text()` bychom z API tesseractu obdrželi řetězec 'caOO_3', jelikož jeho skóre je v součtu nejvyšší. Ten je ze syntaktického hlediska špatný hned kvůli prvního znaku, neboť první znak čtený automatem nemůže být velké písmeno. Hledaná látka je $CaCO_3$, uhličitan draselný. Podívejme se, jak by vypadal strom, který by algoritmus procházel.



Vidíme, že bychom velkou část vrcholů vůbec nemuseli navštívit. Možných cest, tedy syntakticky správných řetězců máme hned 3, a to CaOO_3, $CaCO_3$ a $CaCo_3$. Nejvyšší skóre má první uvedený. Přesnou shodu v databázi nalezneme ale pouze ke druhému řetězci a tudíž jej budeme pokládat za ten správný.

6.2.1 Nedostatky analyzátoru

Bohužel ani s popsaným algoritmem nelze všechny látky poznat správně. Pokud tesseract vůbec nenavrhne správný znak na danou pozici, nemáme šanci sloučeninu najít. Databáze podle [12] navíc neobsahuje veškeré možné sloučeniny, pouze asi 1800 nejčastějších. Nicméně při testování s chemickými učebnicemi byly všechny zkoušené látky v databázi obsaženy. Pokud by však látka v databázi opravdu nebyla, vypíšu syntakticky správný řetězec s nejvyšším skóre.

Další problém je, když tesseract jeden znak rozpozná jako znaky 2. To se stává například u písmene 'O', které je občas rozpoznáno jako '()', 2 závorky. Tento ojedinělý případ by šlo jednoduše řešit podmínkou v implementaci, ale obecné řešení by bylo dost složité. Naštěstí se skutečný znak ve většině případů mezi možnostmi objeví.

6.3 Vyčíslení rovnice

Cílem vyčíslení rovnice je najít nejmenší celočíselné koeficienty takové, že počet prvků na levé straně je stejný jako počet prvků na pravé straně.

Mějme [1]

$$\mathbf{a}Cu + \mathbf{b}HNO_3 \rightarrow \mathbf{c}Cu(NO_3)_2 + \mathbf{d}NO + \mathbf{e}H_2O$$

Příklad lze zapsat jako soustavu matematických rovnic:

$$\begin{aligned} Cu \quad & 1a + 0b - 1c - 0d = 0e \\ H \quad & 0a + 1b - 0c - 0d = -2e. \\ N \quad & 0a + 1b - 2c - 1d = 0e \\ O \quad & 0a + 3b - 6c - 1d = -1e \end{aligned}$$

Zapíšeme koeficienty soustavy do matice:

$$A = (a_{i,j}) = \begin{pmatrix} 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -2 \\ 0 & 1 & -2 & -1 & 0 \\ 0 & 3 & -6 & -1 & -1 \end{pmatrix}$$

Matice je typu (m, n), kde m je počet prvků, které se v chemické rovnici vyskytují a n je počet složek rovnice. $a_{i,j}$ odpovídá počtu prvků i, který obsahuje j-tá složka rovnice. Takovou matici je po správném parsování rovnice sestrojit.

Matici nyní převedeme do redukovaného schodovitého tvaru Gauss-Jordanovou eliminační metodou. To lze velmi jednoduše provést na papíru. V implementaci za tímto účelem využívám algebraickou knihovnu pro manipulaci matic *ejml*[2].

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \frac{3}{4} \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & \frac{3}{4} \\ 0 & 0 & 0 & 1 & \frac{1}{2} \end{pmatrix}$$

Z této matice lze vyextrahovat řešení, stačí zvolit poslední koeficient e tak, abychom se zbavili zlomků. Zde $e = 4$. Vyčíslená rovnice je

$$3Cu + 8HNO_3 \rightarrow 3Cu(NO_3)_2 + 2NO + 4H_2O$$

6.3.1 Nedostatky navržené metody

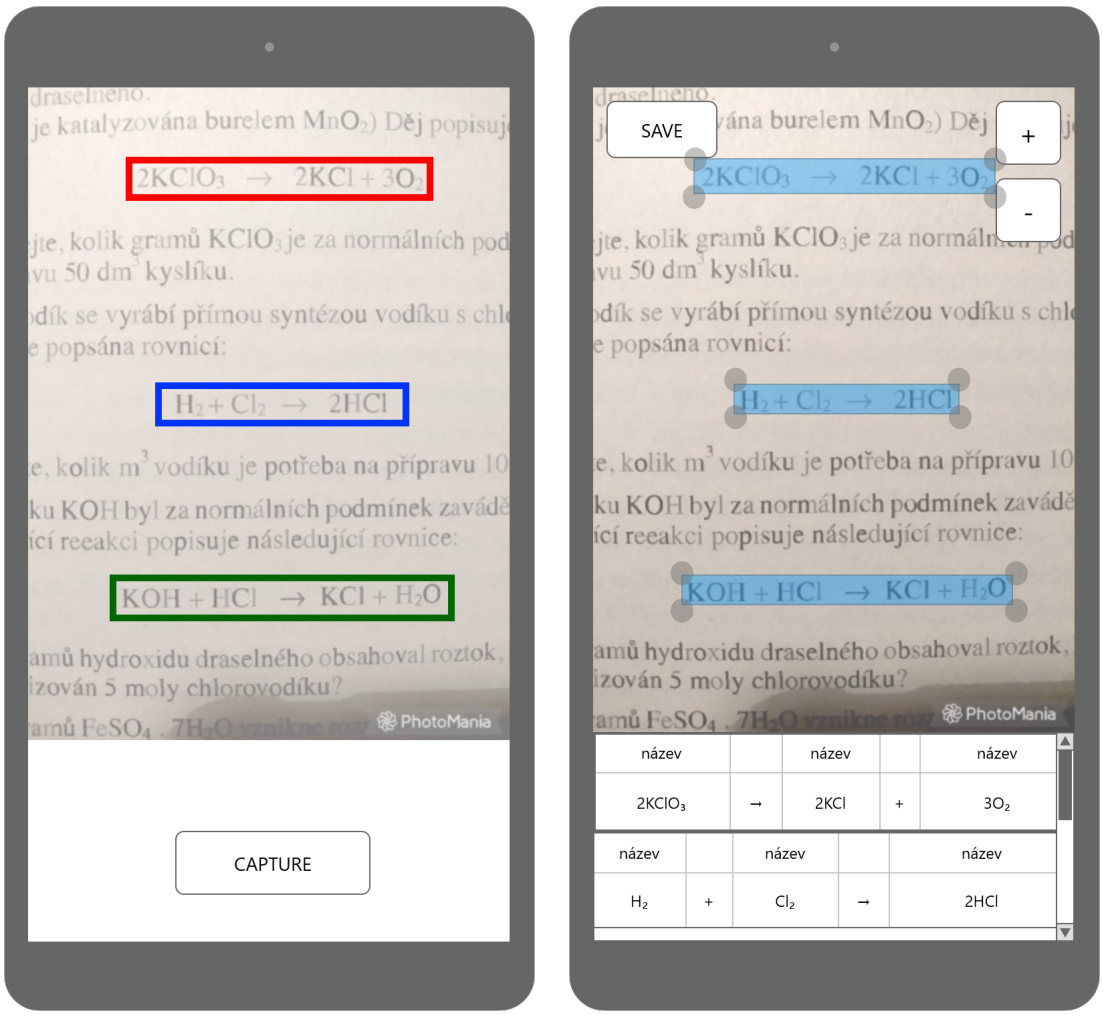
Popsaná metoda nedokáže vyčíslit rovnice, které mají více řešení (závislé na více než jednom parametru, počet neznámých je alespoň o 2 vyšší než počet rovnic). To je poměrně podstatná skupina chemických rovnic. Metoda navržená Lawrence R. Thornem [20], založená na hledání nulového prostoru matice, je schopna řešit i tyto problémové rovnice. Na implementaci je však mnohem složitější. Navíc v příkladech do 2. stupně ZŠ se rovnice s více řešeními neobjevují. Vystačím si tedy s popsaným způsobem vyčíslování.

[1] Příklad z učebnice chemie [19]
[2] Efficient Java Matrix Library, `https://ejml.org/`

Kapitola 7
Návrh mobilní aplikace

Aplikaci budu implementovat v Androidu. Před implementací je vždy dobré si ujasnit grafický vzhled aplikace a potřebné komponenty. Zároveň zde popíšu všechny případy užití aplikace s veškerými interakcemi, které budou v aplikaci implementovány. Kapitola tedy může sloužit i jako uživatelský manuál.

Aplikace bude sestávat ze 2 *aktivit*. *Aktivita* je v prostředí Androidu základním stavebním kamenem aplikace, v podstatě ekvivalent okna na počítači.



CameraActivity DetectorActivity

Obrázek 7.1. Schematický návrh obrazovek vytvořený v Mockplus [1]

První aktivita bude zobrazovat živý záznam z kamery. V co nejkratších časových intervalech, které zařízení umožní, bude obraz snímaný v daném momentu předáván

[1] Prototypovací nástroj. `https://www.mockplus.com/`

jako vstup do sítě YOLO. Zpracované výsledky budou zobrazeny jako rámečky ohraničující rovnice. V závislosti na výkonu zařízení je nutné počítat s prodlevou zobrazování rámečků.

Když je uživatel spokojený s rozpoznáním rovnice, stiskne tlačítko *CAPTURE*. Před otevřením druhé aktivity proběhne syntaktická analýza a případné vyčíslení detekovaných rovnic (a vyřešení rovnic matematických). Pozadí nově otevřené aktivity bude tvořit obrázek snímaný kamerou ve chvíli stisku tlačítka.

I zde budou rovnice ohraničeny rámečky, které ale bude možné v případě potřeby upravit. YOLO totiž ne vždy zcela přesně určí okraje rovnic. V takovém případě je vhodné umožnit uživateli manuální výběr. Rohy rámečků bude tedy možné libovolně posouvat. Při zcela zcestné detekci rovnice bude možné rámeček tlačítkem **-** odstranit. Naopak, nepodaří-li se rovnici vůbec detekovat, je možné tlačítkem **+** rámeček přidat a chybějící rovnici manuálně ohraničit.

Analyzované rovnice se zobrazí tabulkách v dolní části aktivity. Tabulky chemických rovnic budou mít 2 řádky. Pro každou složku rovnice se v dolním řádku zobrazí její značení (s koeficientem vyčíslení) a v horním řádku triviální název složky, pokud nějaký existuje a je dohledatelný v souborové databázi [12]. Tabulky budou barevně indikovat správnost rovnice. Zelená tabulka bude značit správně vyčíslenou rovnici a červená případy, kdy se prvky na levé straně neshodují s prvky na pravé nebo když se rovnici nepodaří správně syntakticky analyzovat.

Pokud složky rovnice nebudou správně rozpoznány, uživatel bude moct celou rovnici upravit. Do editačního módu rovnice se bude možné dostat dvojitým dotekem řádku s rovnicí, kterou je potřeba upravit. Jelikož je na základní klávesnici Androidu obtížné psát dolní indexy, bude stačit psát běžná čísla, z kontextu bude zřejmé, kdy se jedná o indexy. Na místo šipky pak bude možné psát rovnítko. Po úpravě rovnice lze opustit editační mód kliknutím na obrázek.

Nakonec tlačítko *SAVE* slouží k uložení obrázku spolu s anotací ve formátu přijímaným frameworkem *Darknet*. Pomocí aplikace bude tedy možné jednoduše vytvářet ručně anotovaný testovací dataset složený z reálných obrázků rovnic.

Kapitola 8
Implementace mobilní aplikace

V tuto chvíli máme vytrénovaný fungující model YOLO, implementovaný syntaktický analyzátor s OCR a poměrně jasnou představu o designu a případech užití aplikace. Nezbývá než se pustit do práce, propojit existující komponenty a naprogramovat fungující aplikaci.

V této kapitole stručně popíšu architekturu aplikace, hlavní komponenty a komunikaci mezi nimi. Dále se budu věnovat některým problémům, na které jsem během implementace narazil, konkrétně konverzi sítě do frameworku *TFLite*, zpracování výstupu YOLO a nakonec úpravě detekovaných rámečků.

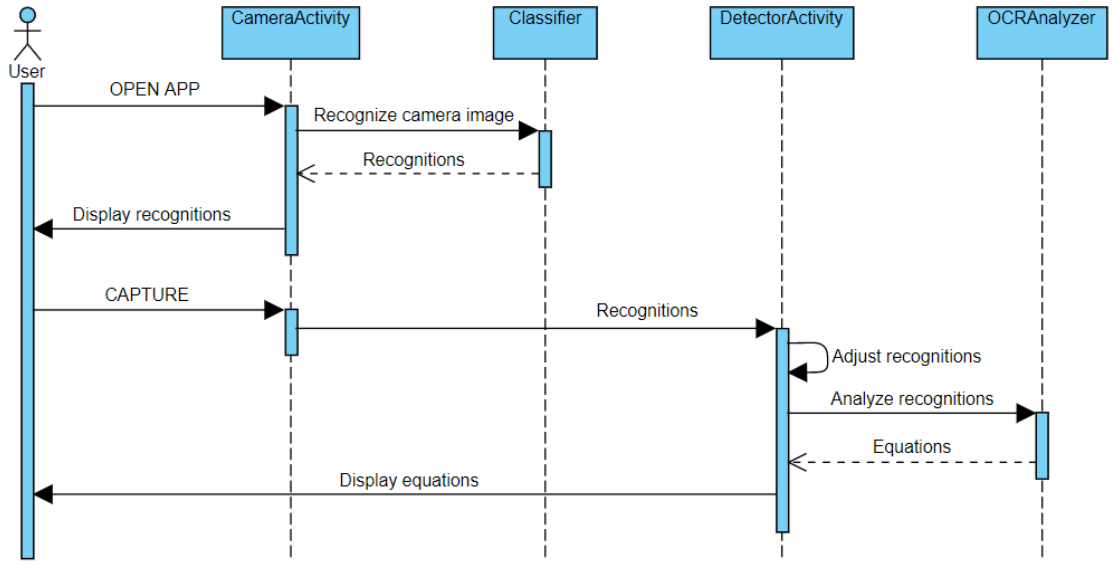
8.1 Architektura

Detailní struktura projektu je popsaná v příloze B. Při implementaci některých částí aplikace jsem vycházel z dema *TFLite*. Tyto části jsou v příloze označeny.

Během implementace jsem se snažil dodržovat oddělení doménové a prezentační vrstvy. Prezentaci a řízení toku událostí mají na starosti samotné Aktivity: *CameraActivity* a *DetectorActivity* viz 7.1. První aktivita konzumuje služby objektu *Classifier*, ve kterém je implementována detekce pomocí YOLO. Druhá aktivita využívá k optické rekognici a analýze objekt *OCRAnalyzer*.

Doménová vrstva se skládá mimo jiné z třídy *Recognition* reprezentující rozpoznaný objekt a třídy *Equation* reprezentující rozpoznanou a analyzovanou matematickou nebo chemickou rovnici.

Základní průchod aplikací, její nejdůležitější zde vyjmenované komponenty a komunikaci mezi nimi nejlépe zachycuje následující sekvenční diagram



Po otevření aplikace se ihned spouští *CameraActivity*. Ta periodicky, v nejkratších intervalech, které zařízení umožňuje, posílá aktuální snímek kamery do *Classifier*. V ob-

jektu *Classifier* probíhá normalizace vstupního obrázku, samotná detekce a zpracování výstupu sítě podle 8.3. Zpracovaný výstup se pak zobrazí v okně *CameraActivity* uživateli. Po stisknutí tlačítka *CAPTURE* se aktuální detekce přepošlou do nově otevřené *DetectorActivity*. Zde proběhne úprava rámečků detekcí podle 8.4. Následně jsou již upravené detekce zpracovány objektem *OCRAnalyzer*. Jak název napovídá, je zde detekce nejprve rozpoznána Tesseractem a následně parsována. Výsledky jsou vráceny již v podobě matematických či chemických rovnic, které jsou v *DetectorActivity* uživateli zobrazeny.

8.2 Konverze sítě do TensorFlow Lite

První závažnou problematikou je spouštění YOLO modelu ve frameworku Darknet na Androidu. Darknet je psaný v jazycích C a CUDA. S akcelerací přes GPU tedy na mobilech počítat nemůžeme. Mobilní čipy navíc bývají architektury ARM s redukovanou instrukční sadou a některé komplexní operace z instrukční sady x86 musí být emulovány, což může framework dále zpomalovat. V mobilní implementaci Darknetu[1], kterou jsem zkoušel, trvala detekce několik minut. Taková prodleva je v navržené aplikaci nemyslitelná.

Řešení nabízí framework TFLite, verze Tensorflow určená pro mobilní využití. Je ale potřeba model určený pro Darknet převést na model pro TFLite. Do TFLite lze navíc modely konvertovat pouze z klasického Tensorflow.

Naštěstí lze konverzi modelu z Darknetu do Tensorflow provést jednoduše, pomoci programu *darkflow*. Stačí do složky s *darkflow* překopírovat soubory *model.cfg* a *model.weights* a pak příkazem

```
python flow --model model.cfg --load model.weights --savepb
```

vytvořit soubor *model.pb* reprezentující síť v Tensorflow.

K vytvoření modelu v TFLite je potřeba mít nainstalovaný Tensorflow. Součástí balíčku je i program *tflite_convert*. Příkaz

```
tflite_convert --graph_def_file=model.pb --output_file=model.tflite
--input_format=TENSORFLOW_GRAPHDEF --output_format=TFLITE
--input_arrays=input --input_shapes=1,416,416,3 --output_arrays=output
```

vytvoří kýžený *model.tflite*.

8.3 Zpracování výstupu YOLO

Aplikace periodicky posílá obraz (jako Bitmapu) snímaný kamerou k detekci. Pokud *Classifier* není zaneprázdněn, snímek přijme. Bitmapa musí být před detekcí upravena na float pole. Tři barevné složky každého pixelu musí být odděleny a normalizovány. Toto zpracování bitmapy je shodné pro Tensorflow i Darknet, lze tedy opět vycházet z dema. Komplikace přicházejí až při zpracování výstupu.

Zatímco Darknet výstup modelu již zpracoval do smysluplné podoby: vyextrahoval z výstupního tensoru souřadnice detekcí, odstranil duplikáty, v TFLite dostaneme jako výstup syrový tensor, který si musíme sami upravit. Naštěstí podle 2.1 přesně víme, jaké má výstupní pole formát. Konkrétně v mém případě je vstupní rozlišení modelu 416×416 a velikost výstupní mřížky tedy bude 13×13. Jelikož pracuji s YOLOv2,

[1] https://github.com/xvolica/DarkNet-for-Android

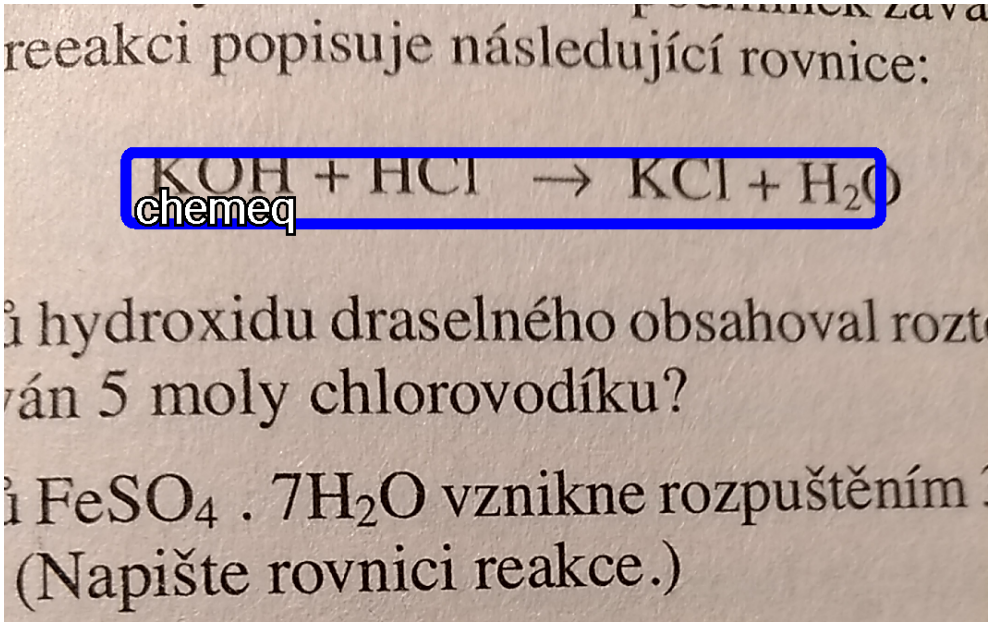
každá buňka mřížky predikuje 5 rámečků. Každá predikce rámečku se skládá ze 7-mi údajů: první 2 udávají souřadnice, další 2 offsety vůči *anchor boxům*, 5. udává *confidence score* a zbylé 2 odpovídají třídám, které rozlišuji - chemické a matematické rovnice (u původního YOLO by se třídy predikovaly pouze jednou pro každou buňku, u vyšších verzí se predikují pro každý rámeček zvlášť). Údaj s vyšší hodnotou odpovídá rozpoznané třídě. Každý predikovaný rámeček se váže na jeden z anchor boxů definovaných v konfiguračním souboru sítě. Proto se musí počet rámečků predikovaných jednou buňkou shodovat s počtem *anchor boxů*.

Dohromady tedy z jediného obrázku dostanu $13 \times 13 \times 5 = 845$ predikcí. Většina z nich bude mít velmi nízké *confidence score*. Musím nadefinovat hranici (například 0.25 jako výchozí hranice u Darkentu), pod kterou budu detekce opomíjet. V úvahu budu brát pouze rámečky s dostatečně vysokým skóre.

Dalším problémem jsou mnohonásobné detekce jediného objektu. Ačkoliv by objekt měl detekovat pouze buňka, do kterého jeho střed spadá, v praxi se tak neděje. Stejnému objektu mohou predikovat rámečky více buněk, navíc jediná buňka může daný objekt rozpoznat vícekrát (jelikož každá buňka predikuje 5 rámečků). V Darknetu je tento problém řešen způsobem nazývaným *non-maximum supression*. Spočívá v tom, že procházíme jednotlivé predikce seřazené podle *confidence score*. Predikci budeme považovat za platnou, pokud její *IoU* nepřesahuje limit 0,5 s žádnou již platnou existující predikcí. Takto budou vícenásobné detekce úspěšně odstraněny.

8.4 Úprava predikovaných rámečků

Predikované rámečky sice bez problému označí rovnici v obrázku, mají však mnohdy problém s přesným ohraničením. Často je rámeček o několik pixelů menší, než by měl být a kvůli tomu je chybně přečtena některá sloučenina rovnice.

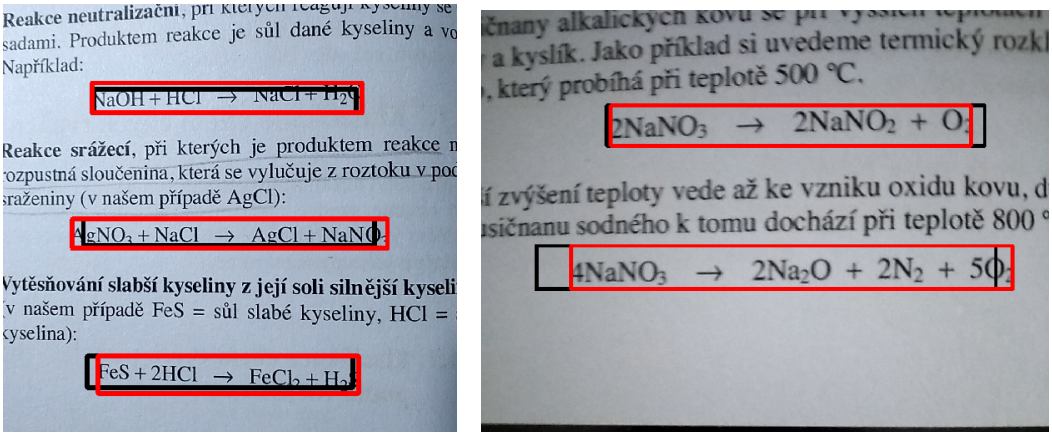


Obrázek 8.1. Obrázek zachycující popsanou situaci. Poslední znak rovnice nespadá celý do rámečku a Tesseract by neměl šanci jej rozpoznat.

V takovéto situaci je možné podle 7.1 upravit manuálně rámeček. Bylo by ale lepší minimalizovat potřebné zásahy uživatele. S pomocí OpenCV a binarizace lze rámečky

přesně přizpůsobit rovnicím. Hrany rámečku totiž v místech, kde zasahují do rovnice, po binarizaci obsahují černé pixely. Stačí tedy danou hranu, levou, pravou, horní nebo dolní, posunout tak, aby neobsahovala žádné černé pixely. Tím se celý rámeček zvětší a bude ohraničovat celou rovnici. Podobně lze rámečky i zmenšovat, jsou-li zbytečně veliké.

Tento přístup bohužel nefunguje ve všech případech. Když není rovnice v obrázku dostatečně oddělena od zbytku textu, přístup může selhat. Pak už opravdu nezbývá nic jiného, než nechat uživatele rámeček manuálně upravit.



Obrázek 8.2. Úprava rámečků v praxi. Černé rámečky jsou výchozí predikce, červené jsou upravené a ohraničují rovnice přesně.

8.5 Nasazení aplikace na cílové zařízení

Aplikace je dostupná pouze jako *debug build*, není a nejspíš ani nebude „releasnutá“. Nejjednodušší způsob instalace aplikace na cílové zařízení je přímo prostřednictvím Android Studia.

Jelikož využívám řadu funkcionalit Javy 8, je potřebné mít na cílovém zařízení novější verzi Androidu. Pro práci s kamerou využívám starší Camera API, protože moje testovací zařízení jiné API nepodporuje. Bohužel si nejsem jistý zpětnou kompatibilitou na novějších zařízení.

Aby se uživatelské rozhraní zobrazilo správně, je potřebné mít display s rozlišením 1080×1920. Dále je potřeba do projektu přidat modul s OpenCV pro Android. Nakonec je nutné překopírovat soubor s jazykovým modelem Tesseractu (*traineddata*) do úložiště zařízení, do složky nazvané *tessdata* v kořenovém adresáři.

Při implementaci aplikace jsem se kompatibilitou napříč zařízeními příliš nezabýval. Aplikaci jsem testoval pouze na svém telefonu, jiné jsem k dispozici neměl. Teoreticky, pokud jsou výše zmiňované podmínky splněny (správné rozlišení, starší Camera API, přidán modul s OpenCV a soubor s jazykovým modelem), neměl by být problém program rozchodit i na jiném mobilu. Nemohu to ovšem zaručit.

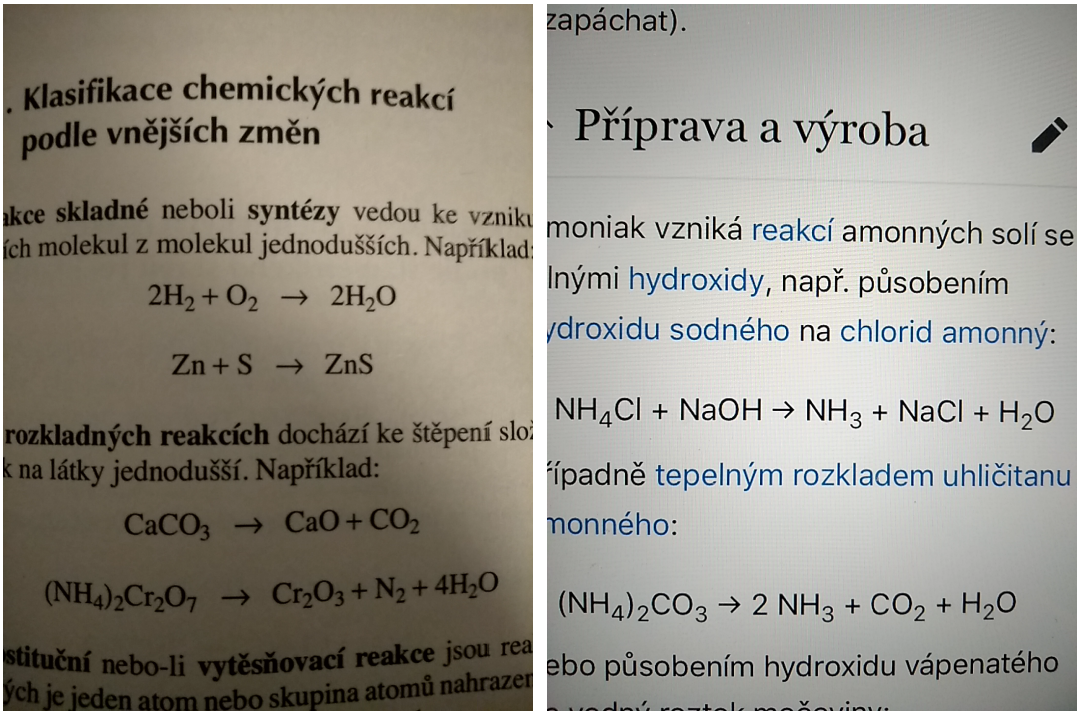
Kapitola 9
Testování aplikace

Cílem testování softwaru je reportování chyb a posuzování kvality produktu. Já se v této kapitole budu zabývat především druhým zmíněným cílem. Budu se snažit objektivně ohodnotit kvalitu a efektivitu rozpoznávání rovnic.

Testovat aplikaci jako celek je obtížné. I s dobře vymyšlenými testovacími scénáři by šlo poměrně těžko zjistit, kde se vyskytuje problém - zda v detekci rovnice, v OCR enginu nebo analyzátoru. Proto nebudu testovat aplikaci jako celek, ale budu testovat její 2 nejpodstatnější moduly: modul s neuronovou sítí pro detekci rovnice a modul s OCR a analyzátorem pro parsování rovnice.

9.1 Testovací data

Pro testování aplikace je nevhodné využít obrázky generované podle 3.1. Lepší by byly reálné fotografie rovnic. K focení mohu vyžít aplikaci. Podle 7.1 jsem implementoval funkci, která umožňuje právě analyzovaný snímek i s (díky posuvným rámečkům upravitelnou) anotací uložit. Takto jsem vytvořil testovací množinu 156 správně anotovaných fotek s rovnicemi. Na fotkách je zachyceno 267 chemických a 26 matematických rovnic. Jako hlavní zdroje k focení jsem využil sérii učebnic [19] a [21], rovnice Wikipedie a několik obrázků vyhledaných na internetu. Chemických rovnic je úmyslně více, protože jsou náročnější na rozpoznání a analýzu.



Obrázek 9.1. Příklady z testovacího datasetu.

Kromě anotace je potřeba přepsat rovnici na obrázku do textové podoby. Testování s větším datasetem fotek by měl lepší vypovídající hodnotu, bohužel je ale jeho tvorba titěrná a časově velmi náročná práce.

Kompletní obrázky budou sloužit jako vstup pro neuronovou síť. Pro modul s OCR a analyzátorem pak poslouží podle anotace vyříznuté obrázky (jako na obrázku 9.2)

9.2 Testování YOLO

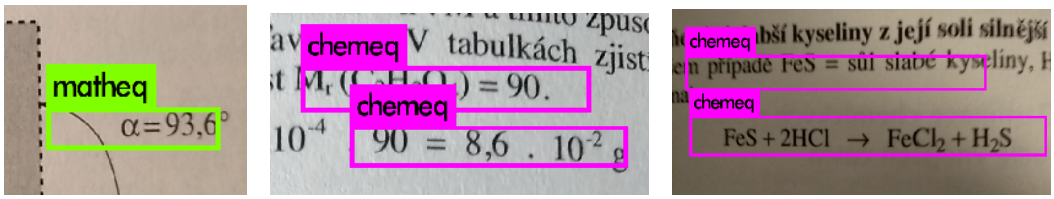
Nejprve otestuji vytrénovaný model přímo ve frameworku Darknet na PC. Výsledky porovnám s testy provedenými na zařízení. Výsledné predikce se sice nebudou lišit, bude mě však zajímat rozdíl v rychlosti detekce. Nakonec otestuji detekci se zapojením úpravy predikovaných rámečků podle 8.4.

Jako hlavní měřítko kvality detekce použiji počty *true positive*, *false positive* a *false negative* detekcí a *IoU*, průnik oblastí detekce a anotace ku jejich sjednocení.

Výsledky z testu spuštěném ve frameworku Darknet zachycuje tabulka:

	čas	TP	FP	FN	průměrné IoU
YOLOv2 Darknet	5980 ms	283	54	10	61,19%

Zpracování jednoho obrázku trvá průměrně necelých 40 ms. To je umožněno zrychlením pomocí GPU (GeForce GTX 1070 Ti). Nepodařilo se detekovat 10 rovnic z 293, což není špatné. Počet FP je poměrně vysoký, asi 16% všech detekcí. Podívejme se na některé z nich.



Vidíme, že FP detekce jsou úseky textu obsahující znaky typické pro rovnice: závorky, čísla, rovnítka a celé chemické značky. To je pravděpodobně důvod, proč jsou detekovány.

Další test implementuji jako *instrumented unit test*. Běží tedy přímo na cílovém zařízení za využití jeho zdrojů. Mohu tedy porovnat délku trvání detekce na zařízení proti výsledkům na PC.

	celkem	průměr
čas	91888 ms	589 ms

Detekce na mobilním zařízení trvá mnohonásobně déle. Navíc při přímém používání aplikace pozoruji ještě delší detekce: pohybují se mezi 600 a 700 ms. To je dle mého názoru způsobeno zaneprázdněným UI threadem, které je při testech volné. *TFLite* totiž využívá dostupná vlákna.

Výsledek se při srovnání s trváním detekce na PC nezdá příliš dobrý. Nicméně ve srovnání s mobilní implementací Darkentu (8.2) se stále jedná o úspěch. Je také potřeba si uvědomit, že mobil Xiaomi Redmi 5 Plus, na kterém testy provádím, patří spíše do nižší třídy. S výkonnějším telefonem by byla detekce pravděpodobně násobně rychlejší.

Bohužel se mi nepodařilo automatizovat testy pro detekci se zapojením úpravy predikovaných rámečků. Vykreslil jsem tedy do obrázků detekované rámečky a rámečky po úpravě různou barvou. Vizuálně jsem pak obrázky kontroloval. Ani v jednom případě nebyl upravený rámeček horší než rámeček výchozí a v mnoha případech (jako v obrázku 8.2) byla detekovaná rovnice díky úpravě dokonale ohraničena.

9.3 Testování OCR analyzátoru

Jak měřit kvalitu OCR analyzátoru? Mohu se například podívat na počet správně rozpoznaných rovnic ze všech. Tento přístup je velmi přísný, stačí jediný špatně rozpoznaný znak a celá rovnice bude považována za chybnou.

Další možný přístup je místo správně rozpoznaných rovnic počítat správně rozpoznané sloučeniny rovnice. Kdyby v celé rovnici se 4-mi složkami byl jediný chybně rozpoznaný znak v jediné sloučenině, rovnice by byla považovaná stále ze $\frac{3}{4}$ za správnou. Testy by byly shovívavější.

Ještě další přístup je brát rovnice jako řetězce znaků a počítat jejich rozdílnost. Tedy kolik znaků by bylo potřeba změnit, přidat nebo odebrat, aby se enginem a analyzátorem rozpoznaný řetězec shodoval s opravdovou rovnicí. To je *Levenštejnova vzdálenost* těchto řetězců.

Každý z těchto přístupů má své výhody. Bez celé správně rozpoznané rovnice nemůžeme vyčíslovat. Nicméně je přece jenom rozdíl, jestli je v rovnici jedna jediná chyba nebo jsou všechny její znaky špatně rozpoznané. V tomhle pohledu mají lepší vypovídající hodnotu počty správně rozpoznaných sloučenin a *Levenštejnova vzdálenost*. Proto zkusím použít všechny 3 popsané metody.

Stejně jako v 9.3 implementuji *instrumented unit test*. Délku rozpoznání a analýzy zachycuje tabulka.

	celkem	průměr
čas	18333 ms	68 ms

Celkové rozpoznání jedné rovnice, tedy průchod Tesseractem a následná analýza, trvá v průměru 68 milisekund. Oproti samotné detekci je tento čas zanedbatelný, není potřeba analyzátor dále zefektivňovat.

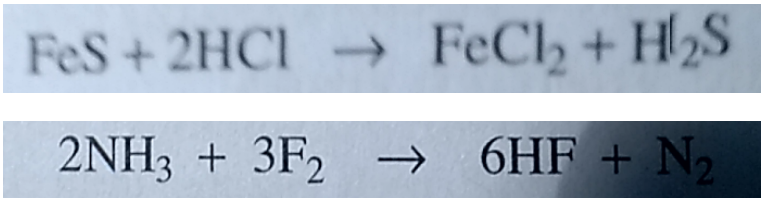
Výsledky testu jsou znázorněny v tabulkách:

	správně	celkem
rozpoznané rovnice	204	267
rozpoznané sloučeniny	883	979
vzdálenost řetězců		182

Vidíme, že asi 75% chemických rovnic je celkově správně rozpoznaných. Celkový počet chybně rozpoznaných sloučenin je asi o polovinu vyšší než počet chybně rozpoznaných rovnic. To znamená, že v chybných rovnicích byl problém často pouze s jedinou sloučeninou. 90% sloučenin je rozpoznáno správně.

Součet *Levenštejnových vzdáleností* rozpoznaných řetězců a těch správných je 182. To znamená, že v 96 chybně rozpoznaných sloučeninách by bylo potřeba změnit, přidat nebo odebrat 182 znaků. Vypadá to, že chybně rozpoznané znaky mají tendenci se seskupovat. To dává z hlediska implementace analyzátoru smysl: po špatně přečteném znaku bude automat v jiném stavu než by měl být, což zvyšuje pravděpodobnost špatného rozpoznání dalšího znaku.

Mezi obrázky, na kterých se rovnice nepodařilo správně rozpoznat, vidím jisté podobnosti. Většina z nich je buď značně rozmazaných, nebo je některá jejich část výrazně tmavší než zbytek obrázku. Další příčinou chyb jsou případy, kdy Tesseract jediný znak rozpozná jako znaky 2 (zmiňováno v 6.2.1). S tím si analyzátor bohužel poradit nedokáže.



Obrázek 9.2. Ukázky obrázků s chybně rozpoznanými rovnicemi. V prvním případě je problém s rozmazanou poslední sloučeninou. V druhém případě je chyba způsobená výrazně tmavším pravým okrajem obrázku.

Zajímavé je srovnání výsledků, když nepoužijeme analyzátor. Tedy když za výslednou detekovanou rovnici budeme považovat výchozí rozpoznání Tesseractu.

	správně	celkem
rozpoznané rovnice	56	267
rozpoznané sloučeniny	412	979
vzdálenost řetězců		993

Vidíme, že analyzátor velmi výrazně zlepšuje rozpoznání. Tesseract sám o sobě by téměř polovinu sloučenin rozpoznal chybně. Se zapojením analyzátoru je to méně než 10%. V takových případech bohužel nezbývá nic jiného, než že uživatel rozpoznání manuálně upraví.

Podívejme se ještě na matematické rovnice:

	správně	celkem
rozpoznané rovnice	26	26
rozpoznané operandy	52	52

Všechny matematické rovnice byly rozpoznány správně. To není překvapivé - struktura matematických rovnic je mnohem jednodušší než těch chemických a množina možných znaků je menší, prostor pro chybování je tedy také menší.

9.4 Limity aplikace

Aplikace je omezena na množství rovnic, které dokáže detekovat v jednom obrázku. V pracovním listu, kde je ve sloupci 15 matematických rovnic, se aplikaci daří detekovat pouze menší část z nich. To je dané malým vstupním rozlišením sítě. Uvědomme si, že YOLO mřížka v aplikaci má velikost 13×13. S 15-ti zarovnanými rovnicemi na zvlášť řádcích by tedy některé buňky mřížky měly být zodpovědné za detekci více objektů. To bohužel od menšího yolov2-tiny vyžadovat nemůžeme. Řešení by bylo využít mohutnější síť s vyšším vstupním rozlišením. To by ale zpracování jediného obrázku trvalo několik vteřin. Podle mého názoru je zde tedy rozumné obětovat přesnost za rychlost a používat menší síť, se kterou je detekce rychlejší.

Kapitola 10
Závěr

Má práce je rozdělena na teoretickou a praktickou část. V teoretické části jsem se snažil představit základy fungování YOLO a popsat proces trénování vlastního modelu. Dále jsem se zabýval API Tesseractu a procesem vytváření vlastního jazykového modelu. Navrhl jsem metodu pro parsování rovnic založenou na konečných automatech, která upravuje výstup Tesseractu. Nakonec jsem vypracoval návrh mobilní aplikace a popsal případy užití.

Hlavními výstupy praktické části práce jsou konfigurovatelný program pro generování trénovacího datasetu v Pythonu a mobilní aplikace pro rozpoznávání rovnic v Androidu. Implementovaná mobilní aplikace by mohla nalézt praktické využití při výuce chemie. Je možné ji použít pro dohledání triviálních názvů složek rovnice, kontrolu validity rovnice a její vyčíslení. Aplikace lze dále využít pro částečně automatizovanou kontrolu jednoduchých matematických rovnic.

Za hlavní přínos práce pokládám to, že se podařilo demonstrovat, že model trénovaný čistě na generovaných syntetických datech dokáže úspěšně detekovat matematické a chemické rovnice v reálných datech. Dalším úspěchem je, že navržená metoda pro parsování rovnic výrazně zlepšuje přesnost rozpoznávání oproti výchozímu výstupu Tesseractu.

Naopak za největší problém považuji časté nepřesné ohraničení rovnice. Rámečky se sice snažím upravovat podle 8.4, ale v některých případech ani to nestačí a je potřeba rovnice manuálně ohraničit.

10.1 Návrhy na možná rozšíření a vylepšení

Práci by bylo možné rozšířit o detekci a rozpoznávání ručně psaných znaků vedle tištěných. Problematické by ale bylo vytváření trénovacího datasetu. Bylo by potřeba nějakým způsobem implementovat generování rovnic, které by byly velmi podobné reálným ručně psaným rovnicím.

Zpracování obrázku pomocí YOLOv2 trvá na mobilním zařízení i přes použití frameworku *TFLite* poměrně dlouho. Detekci by bylo možné zrychlit například kvantizací sítě [22]. Další možná cesta je zapojení Android *Neural Network API*.

Generátor trénovacího datasetu je možné dále rozšiřovat, aby syntetická data byla ještě různorodější a vytrénovaný model lepší. Mohli bychom například rovnice zachycovat v tabulkách nebo jako položky seznamu. Problému s *false positive* detekcemi (viz 9.3) bychom se mohli zbavit přidáním čísel, závorek a řetězců podobných sloučeninám do náhodného textu v obrázcích.

Další vylepšení, které se nabízí, je použití nové verze Tesseractu 4.0. Bohužel v době zahájení implementace aplikace ještě nebyla dostupná tato verze pro Android.

Literatura

[1] Michal Bušta, Lukáš Neumann a Jiří Matas. *Deep TextSpotter: An End-To-End Trainable Scene Text Localization and Recognition Framework.* In: *The IEEE International Conference on Computer Vision (ICCV)*. 2017.

[2] Chen-Yu Lee a Simon Osindero. Recursive Recurrent Nets with Attention Modeling for OCR in the Wild. *CoRR*. 2016, abs/1603.03101

[3] T. Kanahori a M. Suzuki. *Refinement of digitized documents through recognition of mathematical formulae.* In: *Second International Conference on Document Image Analysis for Libraries (DIAL'06)*. 2006. 6 pp.-302.

[4] Harold Mouchere, Christian Viard-Gaudin, Tae Kim, Jin Hyung Kim a Utpal Garain. *CROHME2011: Competition on Recognition of Online Handwritten Mathematical Expressions.* In: 2011. 1497 - 1500.

[5] Francisco Alvaro, Joan-Andreu Sánchez a José-Miguel Benedí. *Recognition of Printed Mathematical Expressions Using Two-Dimensional Stochastic Context-Free Grammars.* In: 2011. 1225-1229.

[6] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick a Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection.* In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. 2016. 779–788.
https://doi.org/10.1109/CVPR.2016.91.

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke a Andrew Rabinovich. *Going Deeper with Convolutions.* In: *Computer Vision and Pattern Recognition (CVPR)*. 2015.
http://arxiv.org/abs/1409.4842.

[8] Joseph Redmon a Ali Farhadi. *YOLO9000: Better, Faster, Stronger.* In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. 2017. 6517–6525.
https://doi.org/10.1109/CVPR.2017.690.

[9] Sergey Ioffe a Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*. 2015, abs/1502.03167

[10] Joseph Redmon a Ali Farhadi. YOLOv3: An Incremental Improvement. *CoRR*. 2018, abs/1804.02767

[11] Paul Mozur. *Inside China's Dystopian Dreams: A.I., Shame and Lots of Cameras*. 2018.
https://www.nytimes.com/2018/07/08/business/china-surveillance-technology.

[12] Wikipedia contributors. *Glossary of chemical formulas*. 2018.
https://en.wikipedia.org/wiki/Glossary_of_chemical_formulas.

[13] Niklas Donges. *Gradient Descent in a Nutshell*. 2018.
https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0.

[14] Igor V. Tetko, David J. Livingstone a Alexander I. Luik. Neural network studies. 1. Comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences*. 1995, 35 (5), 826-833. DOI 10.1021/ci00027a006.

[15] J. A. Hartigan a M. A. Wong. A k-means clustering algorithm. *JSTOR: Applied Statistics*. 1979 , 28 (1), 100–108.

[16] Anthony Kay. *Tesseract: an Open-Source Optical Character Recognition Engine*. 2007.
`https://www.linuxjournal.com/article/9676`.

[17] Zain Sajjad. *Firebase's ML Kit vs Tesseract OCR on Android devices*. 2018.
`https://heartbeat.fritz.ai/comparing-ml-kits-text-recognition-api-on-android-ios-8097aa77b800`.

[18] Adrian Rosebrock. *Text skew correction with OpenCV and Python*. 2017.
`https://www.pyimagesearch.com/2017/02/20/text-skew-correction-opencv-python/`.

[19] Aleš Mareček a Jaroslav Honza. *Chemie pro čtyřletá gymnázia*. Vydáno vlastním nákladem, 2013. ISBN 80-902402-0-8.

[20] Lawrence R. Thorne. *An innovative approach to balancing chemical-reaction equations : a simplified matrix null-space method*. 2009.

[21] Marika Benešová, Erna Pfeiferová a Hana Satrapová. *Odmaturuj z chemie*. DIDAKTIS, 2014. ISBN 978-80-7358-232-6.

[22] Yoni Choukroun, Eli Kravchik a Pavel Kisilev. Low-bit Quantization of Neural Networks for Efficient Inference. *CoRR*. 2019, abs/1902.06822

Příloha A
Obsah přiloženého DVD

- Projekt v Android Studiu ve složce `ChemEq`
- Generátor syntetických obrázků s rovnicemi ve složce `gen`
  - `generateDataset.py` Generování trénovacího datasetu
  - `dataset.ini` Konfigurační soubor řídící generování datasetu
  - `fonts.py` Filtrace systémových fontů, které obsahují potřební znaky pro zachycení chemických a matematických rovnic – dolní indexy, šipku, matematické operátory
  - `drawBBoxes.py` Skript pro vkreslování rámečků do obrázku podle anotace
  - `backgrounds.py` a `backgrounds.pck` Soubory sloužící k vytváření pozadí obrázků
  - `chem_db.json` a `chem_db.py` Soubory ke generování rovnic s existujícími sloučeninami
- Ukázky synteticky generovaných trénovací dat ve složce `traindata`
- Ukázky testovacích obrázku pro YOLO a Tesseract ve složce `testdata`
- `yolov2_darknet.cfg` a `yolov2_darknet.weights` Model pro detekci rovnic v *Darknetu*
- `yolov2_tf.pb` Model pro detekci rovnic v *Tensorflow*
- `yolov2_tflite.tflite` Model pro detekci rovnic v *Tensorflow Lite*
- `chem.traineddata` Jazykový model Tesseractu

Příloha B
Struktura Android projektu

Zde je zachycena struktura zdrojového kódu v balíčcích a krátký popis funkčností jednotlivých tříd. Třídy, při jejichž tvorbě jsem vycházel z demo aplikace *TFLite*[1] jsou označeny `(d)`. Jedná se převážné o třídy zpracovávající obraz z kamery a vkreslující detekce do živého záznamu.

- `CameraActivity (d)` Aktivita otevřená po spuštění aplikace. Zobrazuje detekce.
- `DetectorActivity` Aktivita otevřená po stisknutí tlačítka *CAPTURE*.
- Balíček `model`
  - `Recognition` Třída pro zachycení detekcí.
  - `Compound` Třída reprezentující sloučeninu v chemické rovnici.
  - `Equation` Třída rerezentující matematickou nebo chemickou rovnici.
  - `EqListItem` Položka v seznamu rovnic v `DetectorActivity`.
  - `EqAdapter` Adaptér pro správné zobrazení rovnice.
  - `AdjustableRect` Uživatelem upravitelný rámeček v `DetectorActivity`.
- Balíček `service`
  - `Classifier (d)` Interface pro detekci objektů.
  - `YOLOClassifierImpl` Implementace YOLO detektoru.
  - `OCRAnalyzer` Rozpoznání Tesseractem a následná analýza.
  - `ChemBase` Databáze sloučenin a dohledání triv. názvů.
  - `ObjectTracker (d)` Monitoring detekovaného objektu.
  - `MultiboxTracker (d)` Monitoring detekovaných objektů v `CameraActivity`.
- Balíček `view`
  - `AutofitTV (d)` *TextureView* pro zachycení obrazu z kamery.
  - `CameraFragment (d)` Implementace vykreslování obrazu z kamery.
  - `OverlayView(d)` Debug overlay v `CameraActivity`.
  - `RectDrawView` Overlay pro vykreslování `AdjustableRect`.
- Balíček `util`
  - `BorderedText (d)` Výpis textu k detekcím.
  - `BoundingBox` Pomocná třída při zpracování výstupu YOLO.
  - `ImageUtils (d)` Zpracování a transformace obrazu z kamery.
  - `ThreadUtils (d)` Zjednodušení práce s vlákny.
  - `Logger` Zápis logů.
  - `PassImage` Předávání obrazu mezi aktivitami.

[1] `https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/android/app/src/main/java/org/tensorflow/lite/examples/detection`