

Mobilní aplikace pro rozpoznávání zvířat v zoologických zahradách

Jiří Dabberger

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: **Jiří Dabberger**

Osobní číslo: **A20429**

Studijní program: **B0613A140020 Softwarové inženýrství**

Forma studia: **Prezenční**

Téma práce: **Mobilní aplikace pro rozpoznávání zvířat v zoologických zahradách**

Téma práce anglicky: **Mobile Application for Animal Recognition in Zoos**

Zásady pro vypracování

1. Nastudujte a popište problematiku spojenou s detekcí objektů v obraze.
2. Zvolte vhodné technologie a prostředky k implementaci aplikace.
3. Navrhněte mobilní aplikaci pro rozpoznávání vybraných zvířat pomocí fotoaparátu na platformě Android.
4. Zvolte vhodná zvířata a vytvořte jejich dataset pro rozpoznání v obraze.
5. Implementujte vámi navrženou aplikaci.
6. Výslednou implementaci vhodně otestujte a popište výsledky.



Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

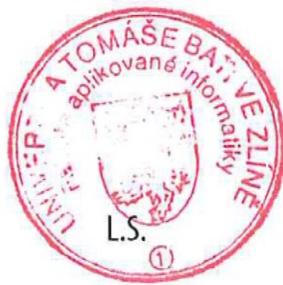
1. TensorFlow Lite. TensorFlow [online]. [cit. 2022-09-18]. Dostupné z: <https://www.tensorflow.org/lite/android>.
2. Jetpack Compose. Android Developers [online]. [cit. 2022-09-18]. Dostupné z: <https://developer.android.com/jetpack/compose>.
3. Klasifikace obrazu. TensorFlow [online]. [cit. 2022-09-18]. Dostupné z: <https://www.tensorflow.org/tutorials/images/classification>.
4. Augmentace dat. Nanonets [online]. [cit. 2022-09-18]. Dostupné z: <https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/>.
5. Konvoluční neuronové sítě. IBM [online]. [cit. 2022-09-18]. Dostupné z: <https://www.ibm.com/cloud/learn/convolutional-neural-networks>.

Vedoucí bakalářské práce: **Ing. Petr Žáček, Ph.D.**

Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **2. prosince 2022**

Termín odevzdání bakalářské práce: **26. května 2023**



doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Jiří Dabberger, v.r.

Ve Zlíně, dne 20.5. 2023

.....

podpis studenta

ABSTRAKT

Tato bakalářská práce popisuje vývoj mobilní aplikace určené jako nástroj pro rozpoznávání vybraných druhů zoologických zvířat. Práce také obsahuje porovnání existujících nebo podobných řešení. Cílem bylo vytvořit aplikaci pro mobilní platformu Android, která skloubí návštěvu zoologické zahrady a poslouží zároveň i ke vzdělávání uživatelů. Jádro aplikace je postaveno na modelu konvoluční neuronové sítě, která se stará o vyhodnocení obrazu z kamery mobilního telefonu a identifikuje zvíře z přednastaveného seznamu. K implementaci aplikace byl zvolen programovací jazyk Kotlin spolu s novým frameworkm Jetpack Compose a pro natrénování modelu sítě se využily knihovny TensorFlow. Výsledná práce také klade důraz na otestování kvality rozpoznání, včetně celkové úspěšnosti a přesnosti rozpoznávání zvířat.

Klíčová slova: Neuronová síť, mobilní aplikace, rozpoznávání zvířat, TensorFlow, Jetpack Compose

ABSTRACT

This bachelor thesis describes development of a mobile application designed as a tool for recognizing several species of zoological animals. This work also contains comparison existing or similar solutions. The goal was to create an Android application to make zoo visits more pleasant and educative for its users. The main pillar of the application is the convolutional neural network model, which takes care of evaluating the image from the mobile phone camera and finds out whether any of the pre-defined animals are presented in the image. Programming language Kotlin has been used to create the mobile app together with a new Jetpack Compose framework and TensorFlow libraries have been used for the model training. The result of this work deals with testing and overall success in the accuracy of the animal recognition.

Keywords: Neural network, mobile application, animals recognition, TensorFlow, Jetpack Compose

Tímto bych rád poděkoval váženému panu Ing. Petru Žáčkovi, Ph.D, který se ujmul vedení mé práce a po celou dobu mi byl nápomocen a vždy ochotně pomohl nebo navedl na správnou cestu. Dále bych chtěl poděkovat svým kamarádům za podporu, obzvlášť Mgr. Jakubu Fišerovi, který mi toto téma navrhnul a rodině za pochopení a poskytnutí času na vypracování této práce.

Poděkování patří i organizaci MetaCentrum, která mi poskytla přístup k výkonnému HW, ve znění: Computational resources were provided by the e-INFRA CZ project (ID:90140), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Prohlašuji, že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD.....	9
1 TEORETICKÁ ČÁST.....	11
1 POČÍTAČOVÉ VIDĚNÍ	12
1.1 KLASICKÉ ÚKOLY POČÍTAČOVÉHO VIDĚNÍ.....	12
1.1.1 Klasifikace obrazu.....	12
1.1.2 Detekce objektů.....	12
1.1.3 Segmentace obrazu.....	13
1.1.4 OCR.....	13
1.2 VYUŽITÍ CV	14
2 KONVOLUČNÍ NEURONOVÉ SÍTĚ	15
2.1 VRSTVY CNN	15
2.1.1 Konvoluční vrstva	15
2.1.2 Pooling vrstva.....	16
2.1.3 Plně propojená a Aktivační vrstva	16
2.2 UČENÍ NEURONOVÝCH SÍTÍ.....	17
2.2.1 TensorFlow.....	18
2.2.1.1 TensorFlow Lite.....	18
2.2.2 Problémy při učení	19
2.2.2.1 Trénovací data.....	19
2.2.2.2 Overfitting.....	20
2.2.2.3 Underfitting.....	20
2.2.3 True vs False a Positive vs Negative.....	21
2.3 TECHNIKA NON-MAX-SUPPRESSION.....	21
2.4 DETEKTORY	22
2.4.1 Dvoustupňové detektory	22
2.4.2 Jednostupňové detektory	22
2.4.2.1 YOLO	23
2.4.2.2 SSD MobileNet V2	24
3 VÝVOJ MOBILNÍCH APLIKACÍ	25
3.1 DRUHY VÝVOJE MOBILNÍCH APLIKACÍ.....	25
3.1.1 Platformě závislé	26
3.1.1.1 Nativní vývoj	26
3.1.2 Platformě nezávislé – multiplatformní	26
3.1.2.1 Webový vývoj.....	26
3.1.2.2 Hybridní vývoj	27
3.2 ARCHITEKTONICKÉ VZORY	27
3.2.1 Model-View-Controller.....	27
3.2.2 Model-View-Presenter	28
3.2.3 Model-View-ViewModel	29
3.3 ANDROID.....	30
3.3.1 Úrovně API	30

3.3.2	Android Studio	32
3.4	FRAMEWORK JETPACK COMPOSE.....	33
4	SEZNÁMENÍ S PODOBNÝMI PROJEKTY	34
4.1	GOOGLE LENS	34
4.2	SEEK BY iNATURALIST	34
II	PRAKTICKÁ ČÁST	35
5	NÁVRH MOBILNÍ APLIKACE	36
5.1	FUNKCIONÁLNÍ POŽADAVKY	36
5.2	NEFUNKCIONÁLNÍ POŽADAVKY.....	37
5.2.1	Kompatibilita.....	37
5.2.2	Forma ukládání dat.....	38
5.3	WIREFRAME APLIKACE.....	38
5.4	DATA APLIKACE	39
5.4.1	Resources Values	39
5.4.2	Resources Drawable.....	40
5.4.3	Úložiště SharedPreferences.....	40
5.4.4	Ostatní data.....	40
5.5	DESIGN APLIKACE	40
6	VÝVOJ MOBILNÍ APLIKACE	41
6.1	ZVOLENÉ TECHNOLOGIE.....	41
6.2	ZAČÁTEK VÝVOJE APLIKACE	41
6.3	NÁHLED KAMERY	42
6.3.1	Finální verze náhledu kamery	43
6.3.1.1	Informace o detekovaném zvířeti	44
6.3.1.2	Aktualizace UI	44
6.4	DATA ZVÍŘAT	45
6.4.1	Data zvířat ve Values podobě.....	47
6.5	NÁHLED DETAILU ZVÍŘETE	47
6.5.1	Mapa zvířat.....	48
6.5.2	Taxonomie zvířat.....	49
6.5.3	Detail zvířete v zoologické zahradě	49
6.5.4	Ukázka.....	49
6.6	OBJEVY	50
6.6.1	Objevy zoologických zahrad.....	51
6.6.2	Detail zoologické zahrady	52
6.7	DALŠÍ FUNKCE.....	53
6.8	MANIFEST	54
7	TVORBA DATASETU	55
7.1	IMAGE CLASSIFICATION DATASET	55
7.1.1	Použité techniky	56
7.1.1.1	Flickr a jeho API.....	56

7.1.1.2	Image augmentation.....	56
7.1.2	Finální zhodnocení a informace	57
7.2	OBJECT DETECTION DATASET	58
7.2.1	Použité nástroje	58
7.2.1.1	Program LabelImg	59
7.2.1.2	Volně dostupné datasety	60
7.2.1.3	Skript pro kontrolu vadných obrázků	60
7.2.1.4	Skript pro přejmenování souborů	60
7.2.1.5	Skript pro editaci XML souborů.....	61
7.2.1.6	Skript pro odstranění malých obrázků	61
7.2.1.7	Skript pro tvorbu „background“ obrázků.....	61
7.2.1.8	Skript pro dokončení datasetu.....	61
7.2.1.9	Skript pro generování TFRecord souboru	62
7.2.2	Finální zhodnocení a informace	62
8	TRÉNOVÁNÍ MODELU.....	64
8.1	API PRO VYTVOŘENÍ VLASTNÍCH MODELŮ	64
8.1.1	TensorFlow lite model maker	65
8.1.2	TensorFlow 2 Object Detection API.....	66
8.1.2.1	Instalace	66
8.1.2.2	Příprava TF Object Detection API.....	66
8.1.2.3	Trénování s TF Object Detection API	68
8.1.2.4	Vytvoření tflite modelu.....	68
8.2	MODEL PRO KLASIFIKACI	69
8.3	MODEL PRO DETEKCI.....	69
8.4	METACENTRUM	70
8.4.1	Seznámení a prvotní nastavení	70
8.4.1.1	Konfigurace nástroje PuTTy.....	71
8.4.1.2	Instalace potřebných knihoven	72
9	TESTOVÁNÍ MODELU	74
9.1	TÉMĚŘ REÁLNÉ TESTOVÁNÍ V APLIKACI.....	75
9.2	ZÁVĚREČNÉ TESTOVÁNÍ V ZOOLOGICKÉ ZAHRADĚ ZLÍN LEŠNÁ.....	77
ZÁVĚR	79	
SEZNAM POUŽITÉ LITERATURY.....	81	
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....	85	
SEZNAM OBRÁZKŮ	86	
SEZNAM TABULEK	88	
SEZNAM PŘÍLOH.....	89	

ÚVOD

Chytré mobilní telefony a jejich aplikace jsou trendem dnešní společnosti a většina z nás, si život bez chytrého mobilního telefonu nedokáže ani představit. Dnešní telefony již neslouží pouze jako prostředek ke komunikaci s našimi přáteli pomocí telefonování a SMS, ale zejména k využívání chytrých aplikací, které mohou udělat nás život jednodušší.

Mobilní telefony dnes obsahují širokou škálu hardwaru, jako je GPS sloužící pro zjištění aktuální polohy, čip NFC, který mimo jiné umožňuje provádět bezkontaktní platby, kvalitní fotoaparát pro zachycení kvalitních fotografií a v neposlední řadě výkonný procesor, který se o všechny operace dokáže postarat. Výše uvedené možnosti ale vyžadují jednu důležitou věc, a tou jsou mobilní aplikace, díky kterým je možné využít mobilní telefon prakticky k čemukoliv. Takových aplikací existuje celá řada, některé slouží pro zábavu ve volném čase, některé mohou posloužit jako zdravotní pomůcka nemocným lidem, jiné zase jako pomocník při cestování nebo nakupování a v neposlední řadě ty, které se snaží člověka něco přiučit, naučit nebo mu pomoci vyhledat si další informace. Mezi poslední množinu z výčtu aplikací se může zařadit i aplikace popsaná v této bakalářské práci, jelikož se snaží uživateli usnadnit návštěvu zoologické zahrady a udělat ji více interaktivní za pomocí mobilního fotoaparátu a moderních technologií dnešního světa. Aplikací využívající mobilní fotoaparát je nespočet, převážně se ale jedná o aplikace sloužící jako komunikační prostředek s možností sdílení fotografií, jako jsou sociální sítě Instagram nebo Snapchat, popřípadě aplikace, které dokážou pomocí umělé inteligence detektovat tvář uživatele a zaměnit ji za něco jiného, například za hlavu zvířete, popřípadě tvář digitálně zkrášlit.

Výsledná aplikace je vytvořena za pomocí umělých neuronových sítí, které jsou díky své schopnosti „trénování“, vhodná pro řešení komplikovaných úloh v oblastech, jako je například klasifikace obrazových dat.

Důležitou částí práce byl vývoj mobilní aplikace, jakožto prostředek pro detekci zvířat s využitím naučeného modelu konvoluční neuronové sítě za použití vlastního trénovacího datasetu.

V úvodu teoretické části je představeno odvětví počítačového vidění (1), do kterého spadá nejen základní problematika této práce. Z počítačového vidění přejdeme k pojmu konvoluční neuronové sítě (2), kde si popíšeme jejich architekturu a samotný proces učení. Seznámíme se s knihovnami TensorFlow a zjistíme reálné možnosti existujících řešení pro rozpoznávání objektů v obraze. Před praktickou částí se ještě seznámíme s vývojem mobilních aplikací (3), operačním systémem Android a jeho novým frameworkm Jetpack Compose.

Teoretická část je zakončena seznámením s 2 aplikacemi (4), ke kterým se vytvořená aplikace v této práci podobá.

Praktická část začíná návrhem mobilní aplikace (5) a jejími funkcionálními a nefunkcionálními požadavky. Z návrhu přejdeme k samotnému vývoji aplikace (6), kde se seznámíme s použitými technologiemi. Zde nás dále čeká proces vývoje náhledu kamery v aplikaci, ukládání a načítání dat zvířat na jejich obrazovce detailu. Dozvíme se, k čemu můžeme v takové aplikaci využít polohovací technologii GPS a co čeká uživatele při objevování zvířat a zoologických zahrad. Od vývoje aplikace přejdeme k vývoji datasetu (7), bez kterého by aplikace nemohla ani vzniknout. Seznámíme se se dvěma druhy datasetu a jaké techniky a python skripty pomohli k jejím vytvořením. Po jejich zhodnocení se podíváme na samotnou tvorbu a učení modelu konvoluční neuronové sítě (8) z předchozích datasetů. Zde se seznámíme se dvěma API nástroji, které udělají trénování modelu „jednoduchou“ záležitostí. Zmíníme organizaci MetaCentrum (8.4), díky které bylo možné modely natrénovat za pomocí jejich výkonných HW strojů. Před samotným závěrem otestujeme přesnost klasifikace zvířat v aplikaci pomocí natrénovaných modelů a výsledky porovnáme s jinými aplikacemi (9).

I. TEORETICKÁ ČÁST

1 POČÍTAČOVÉ VIDĚNÍ

Počítačové vidění neboli **Computer Vision (CV)** je trendem dnešního světa. Jedná se o oblast umělé inteligence, která pomocí různých algoritmů a systémů umožňuje počítačům porozumět okolí, kolem kterého se nachází. Cílem počítačového vidění je porozumět vstupnímu obrazu počítačem, podobně jako tomu je v lidském těle pomocí očí a mozku. Sofistikované algoritmy a neuronové sítě spolu se strojovým učením jsou hlavními pilíři této rozsáhlé technické oblasti, která lidstvu dopomohla v mnoha komplexních případech světa. [1]

1.1 Klasické úkoly počítačového vidění

Především se jedná o procesy detekce nebo identifikace obrazu. Taková úloha se může z pohledu člověka zdát velice triviální, ovšem pokud jsou tyto procesy potřeba provádět na tisících snímcích v relativně krátkém čase, není šance, aby se o ně postaral člověk.

1.1.1 Klasifikace obrazu

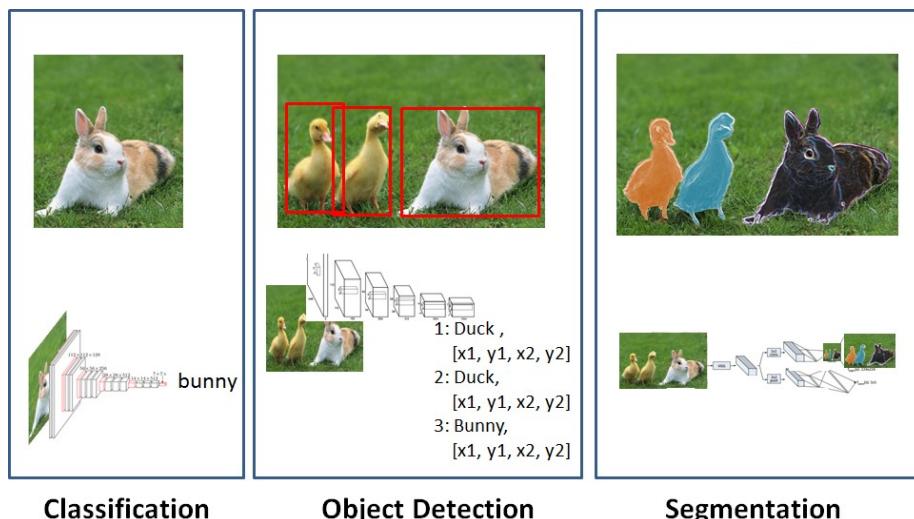
Jedná se o oblíbenou úlohu počítačového vidění, která se stará o klasifikaci obrazu do jedné předem definované třídy, podle toho, co se v obrazu vyskytuje. Výstupem je buď číselná hodnota udávající index, jež se spojuje s názvem detekované třídy objektu nebo prázdná hodnota, značící, že klasifikační model nenašel v obraze žádnou naučenou třídu. Díky této technice je tedy možné zjistit **jednu** konkrétní třídu objektu nacházející se ve vstupním obrazu. [2]

1.1.2 Detekce objektů

Jedná se o rozšíření úlohy „klasifikace obrazu“ o možnost detekovat **všechny** objekty v obraze navíc s vyznačením jejich polohy. Stejně jako pro všechny ostatní úkoly je pro provádění této činnosti potřeba zařízení schopné zpracovat vstupní obraz a předat ho do předem naučeného modelu, který se o detekci postará. Výsledný výstup obvykle obsahuje pole souřadnic s přidruženým indexem třídy detekovaného objektu. Tyto informace se dále většinou využijí pro vizuální zobrazení detekovaných objektů vykreslením ohrazení ze získaných souřadnic a k nim doplnění názvu objektu z pole tříd. Toto vykreslení může probíhat na jednotlivém obrázku, ale i na snímcích videa, a to všechno v reálném čase při použití nejnovějších modelů a dostatečně výkonného hardwaru. [2]

1.1.3 Segmentace obrazu

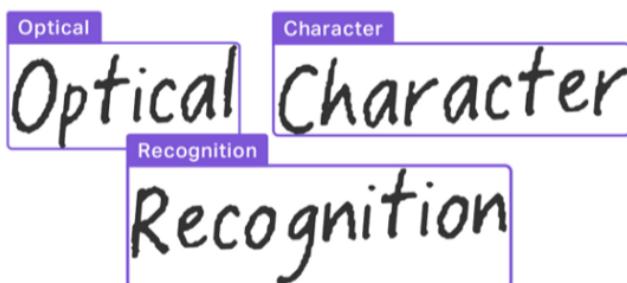
Segmentace obrazu je další důležitou úlohou počítačového vidění zabývající se „rozdělením“ obrazu na jednotlivé objekty, které jsou v něm obsaženy. Tako nalezené objekty jsou mezi sebou většinou rozdeleny podle jejich hranic, což může taky vést k jednomu z kroků detekce objektů. Pomocí této techniky je výstupem (maska nebo matice hraničních pixelů s třídou) „rozdelený“ obraz na jednotlivé segmenty označené příslušnou třídou. [2]



Obrázek 1. Základní kategorie počítačového vidění. [3]

1.1.4 OCR

Jedná se o proces digitalizace textu neboli převod ručně psaného, popř. tištěného textu do podoby, se kterou lze s textem pracovat digitálně například na počítači. V oblasti počítačového vidění spadá OCR pod nejčastěji se vyskytující odvětví. V praxi funguje velice podobně jako detekce objektů, tudíž se hledá oblast v obrázku, která obsahuje větu, slovo nebo znak a jakmile se tato oblast nalezne, je aplikována klasifikace pro rozpoznání konkrétního znaku. [4]



Obrázek 2. Ukázka fungování OCR. [5]

1.2 Využití CV

Jak vyplývá z běžných úkolů CV, jeho využití je možné uplatnit v široké škále mnoha odvětví a oblastí. Pro samotné využití je ale potřeba několik zásadních komponent, které dopomohou k vytvoření aplikace počítačového vidění:

1. Hardware: Výkonný počítač s procesorem, grafickou kartou a pamětí pro zajištění stabilního zpracování velkého počtu dat a složitých výpočtů.
2. Software: Pro vývoj CV je vhodné využít několik dostupných algoritmů a knihoven, jako je například OpenCV nebo TensorFlow pro operace nad obrazem.
3. Datová sada: Je nutné zajistit velký počet dat, na kterých se může síť učit rozpozнат objekty pro následné vytvoření aplikace počítačového vidění.

S pomocí vytvořeného modelu schopného rozumět vstupnímu obrazu přichází na řadu jeho využití. Mezi hlavní odvětví můžeme zařadit právě ty, jejichž existence závisí na nějakém druhu automatizace, jako jsou například:

- **Autonomní vozidla** – jejich vývoj jde neustále dopředu, a to hlavně díky CV. Bez možnosti automatizované detekce objektů, jako je v tomto případě řada překážek na silnici, chodci, všechny druhy vozidel, dopravní značení nebo samotná cesta by nebylo možné autonomní vozidla vyvinout.
- **Medicína** – v posledních několika letech se můžeme s CV setkat i v lékařském prostředí, kde dopomáhá určovat diagnózy pacientů z RTG a jiných snímků.
- **Bezpečnostní systémy** – mezi které můžeme zařadit například bezpečnostní kamery na pracovištích a parkovištích. Ty automaticky snímají jejich okolí, detekují procházející osoby a v případě problémů nebo zločinů je lze snadno vystopovat.
- **Identifikace** – rychlou a bezpečnou formou moderních pracovišť je rozhodně možnost identifikace pracovníků podle jejich obličeje. Na základě správného rozpoznání obličeje má osoba povolený vstup do areálu nebo místnosti, aniž by se musela zdůrazňovat například heslem.
- **Robotika** – robotická ramena nebo vozíky využívající řadu senzorů a kamer pro plynulé pohybování v prostoru s interakcí na okolní vlivy podobně jako autonomní vozidla.

2 KONVOLUČNÍ NEURONOVÉ SÍTĚ

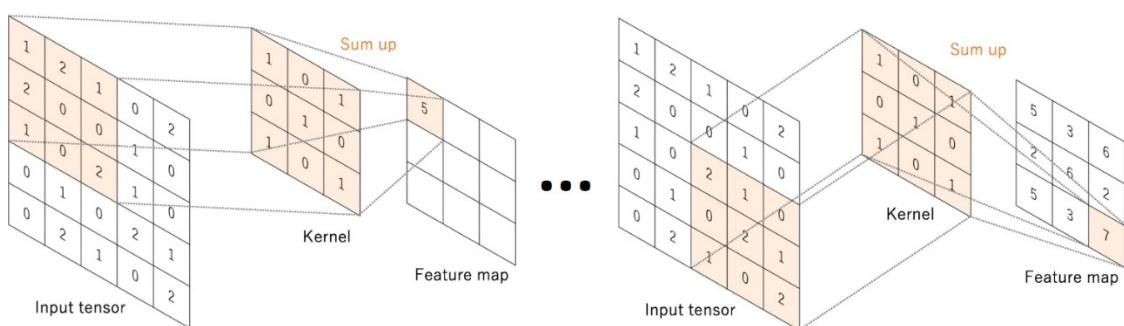
Konvoluční neuronové sítě (anglicky Convolutional Neural Network, CNN) jsou nejpoužívanějším typem neuronových sítí starající se o detekci či rozpoznávání objektu v obrazu nebo zvuku. Jedná se o rozšířenou variantu klasických neuronových sítí o speciální vrstvu, která dopomáhá elegantně eliminovat problém s velkými daty, jako jsou právě obrázky. Tímto elegantním řešením se zabývá tzv. konvoluční vrstva (od toho poté název Konvoluční neuronové sítě), která značně redukuje vstupní parametry obrazu. [6]

2.1 Vrstvy CNN

Jak již bylo naznačeno, architekturou konvoluční neuronové sítě jsou navzájem propojené vrstvy, které mají každá svá specifika. Tyto vrstvy jsou mezi sebou několikrát střídány, což s různou kombinací zapříčiňuje rozlišné trénování sítě. Proto je návrh po-sobě jdoucích vrstev často nejdůležitější přípravou při trénování sítě.

2.1.1 Konvoluční vrstva

Jedná se o vrstvu, která v určitém směru prochází obraz pomocí několika filtrov, které zachycují různé informace obrázku, jako jsou hrany, světlost, popř. barva a jiné. Tento filtr si můžeme představit jako matici $N \times N$ (obvykle 3×3 nebo 5×5) navíc s doplněním barevného kanálu obrázku ($\times 3$ pro barevný RGB režim nebo $\times 1$ pro režim šedi). Při procházení, které má navíc nastavení udávající krok (obvykle 1), se generuje tzv. **Feature mapa** obsahující součet „aktivovaných“ pixelů ze vstupního obrázku a filtru, tak, jak můžeme vidět na obrázku níže (Obr. 3)¹. [7]



Obrázek 3. Aplikace filtru na vstupní obrázek v konvoluční vrstvě. [7]

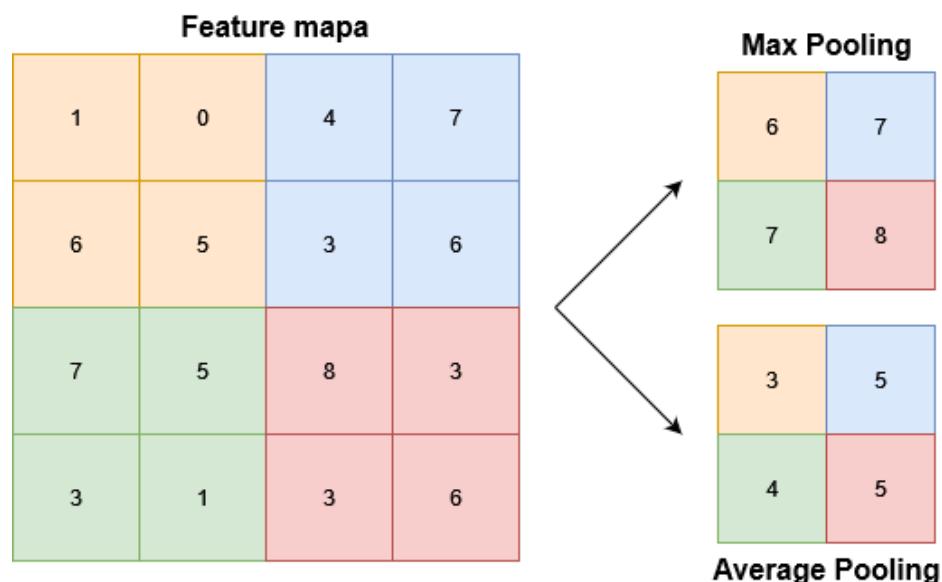
¹ Matici **Kernel** na obrázku chápejme jako **filtr**.

2.1.2 Pooling vrstva

Pooling vrstva neboli vrstva sdružující, se stará o další zredukování vstupních parametrů pomocí jednoho z existujících filtrů. Tato operace se podobá operacím v konvoluční vrstvě, s výjimkou, že vstupní maticí v Pooling vrstvě je výstup z konvoluční vrstvy, tzn. Feature mapa.

Nejčastějším filtrem je operace nazvaná **Max Pooling**, kterou je obvykle matice 2×2 s krokem 2, aby se operace nepřekrývali a tím dosáhli největšího zredukování vstupních parametrů. Tento filtr má za následek to, že se z Feature mapy přečte pole o velikosti 2×2 a do výstupu se z daného pole uloží **maximum**. Tato operace se opakuje až do konce Feature mapy vždy se stejným nastavením.

Druhým nejčastějším filtrem je **Average Pooling**, který do výstupu vkládá průměrnou hodnotu z hodnot vstupní matici. [7]



Obrázek 4. Ukázka použití filtrov v Pooling vrstvě.

Na obrázku (Obr. 4) je patrné, že se mapa velikostně zredukuje o $\frac{1}{4}$ díky odstranění redundantních pixelů. To zapříčiní snížení celkového rozlišení obrázku, ale také snížení potřebného výkonu pro další potřebné výpočty a samotné trénování sítě.

2.1.3 Plně propojená a Aktivační vrstva

Doposud se pokaždé pracovalo s několika 2-D maticemi, které jsou ale pro plné propojení potřeba vektorizovat neboli převést do 1-D vektoru. Tím se propojí každá vstupní hodnota s výstupní a vznikne **plně propojená** vrstva (někdy také nazvaná jako **Dense** vrstva).

Typickým počtem výstupních hodnot je počet vstupních tříd, které má daná neuronová síť umět rozpozнат.

V poslední (Aktivační) vrstvě se získává pravděpodobnost každé třídy ze vstupního obrazu a je to tedy vrstva starající se o samotnou klasifikaci. Každá hodnota je v rozsahu 0–1, kde jejich společný součet musí vrátit hodnotu 1. Čím větší číslo (pravděpodobnost) u třídy, tím si je síť více jistá, že se na obrázku nachází daná třída. [7]

2.2 Učení neuronových sítí

Jak již celý koncept umělých neuronových sítí vychází z neuronů lidského těla, ani jejich samotné učení není výjimkou. Jako i my lidé se učíme od jiných lidí, stejně tak se učí i umělé neuronové sítě, kdy se pomocí zpětné vazby dotazují své úspěšnosti trénování.

O trénování sítě se starají dvě hlavní části, které se nazývají **dopředná** a **zpětná** propagace. Každá z těchto „funkcí“ se stará o vyhodnocování učení a jeho následnou optimalizaci v dalších iteracích učení. Kolikrát se budou iterace opakovat je obsaženo v hodnotě zvané **epoch**. Počet iterací v jedné epoše závisí na 2 hlavních bodech:

1. Počet trénovacích dat.
2. Velikost **batch** – jedná se o počet dat, na kterých se bude síť učit v jeden moment. Závisí na velikosti grafické paměti a obvyklá hodnota nastavení je mocnina 2 (např. 8, 16, 32 nebo 64).

Z následujících dat lze vytvořit rovnici (1), která spočítá počet iterací *i* mezi podílem počtu trénovacích dat **dataSize** a velikostí **batch**.

$$i = \frac{\text{dataSize}}{\text{batch}} \quad (1)$$

1 epocha tedy udává jednu celou iteraci učení přes všechna trénovací data. Z toho lze určit druhá rovnice (2), která udává celkový počet všech iterací **sumI** učení sítě, než bude trénování dokončeno.

$$\text{sumI} = i * \text{epoch} \quad (2)$$

Každá iterace je tedy složena z **dopředné propagace**, která přijímá trénovací data na vstupu a pomocí jednotlivých váh neuronu a očekávaného výstupu rozhoduje o jejich aktivaci. Po této části následuje **zpětná propagace**, ve které se váhy neuronů mírně upravují pro zajištění lepší přesnosti sítě v dalších iteracích. Tato úprava je závislá na chybě, která vznikne mezi očekávaným výstupem a skutečným výstupem. [8]

2.2.1 TensorFlow

Jedná se o známou knihovnu používanou pro projekty strojového učení, jako je například analýza dat, klasifikace obrazu nebo překlad textu. Byla vytvořena týmem Google Brain v roce 2015 a volně poskytnuta jako open-source. [9]

TensorFlow funguje na principu toku dat skrze graf. Data jsou v tomto případě reprezentovány N-dimenzionálními strukturami, které jsou nazvané **Tensory**, z čehož vyplývá i samotný název knihovny. Struktura uložených dat je buď vektor nebo matice. **Graf** obsahuje propojené uzly, které jsou využity pro aplikování různých operací na datech (Tensorech). K dosažení své vysoké výpočetní rychlosti bylo TensorFlow vytvořeno za pomocí programovacího jazyka C++, ale důležitou roli hraje nadstavba Python API pro zajištění snadnější dostupnosti knihovny širokému okolí uživatelů. [9]

Výpočetní graf je datovou strukturou, jejíž hlavní výhoda je možné uložení, znovu rozběhnutí nebo například i její vizualizace. Z této flexibility těží celá knihovna TensorFlow, jelikož takto vytvořené grafy je možné exportovat na různá zařízení jiných systémů i architektur. [10]

2.2.1.1 TensorFlow Lite

Jelikož se tato práce zabývá vývojem na mobilní platformu, je potřeba zajistit kompatibilní model strojového učení na mobilní zařízení. Jak je známo, mobilní zařízení mají omezené množství výkonu i paměti, díky čemuž by na nich nebylo možné klasický model spustit.

TensorFlow proto přišel s nástrojem **TensorFlow Lite**, který je speciálně navržen pro mobilní a embedded zařízení, jenž eliminuje základní problémy strojového učení na mobilních zařízeních.

- **Prodleva** – není potřeba server pro uložení modelu, jelikož je model uložen na samotném zařízení
- **Soukromí** – data nejsou ze zařízení odesílána
- **Konektivita** – internetové připojení není potřebné
- **Velikost** – velikost modelu je redukována na potřebné minimum
- **Spotřeba** – efektivní rozpoznávání, nízká velikost a nepotřebné připojení k síti vede ke snížení spotřeby energie

K efektivnímu redukování velikosti a zrychlení **inference** (rozpoznávání/klasifikace) modelu přispívá knihovna **FlatBuffer**.

Tento typ modelu lze natrénovat s pomocí nástroje **TensorFlow Lite Model Maker** nebo konvertovat klasický TensorFlow model nástrojem **TensorFlow Lite Converter**. Vytvořený model, který je reprezentován koncovkou **tflite**, pak můžeme využít nejen v operačních systémech Android a iOS, ale také v mikropočítačích založených na operačním systému Linux. [11]

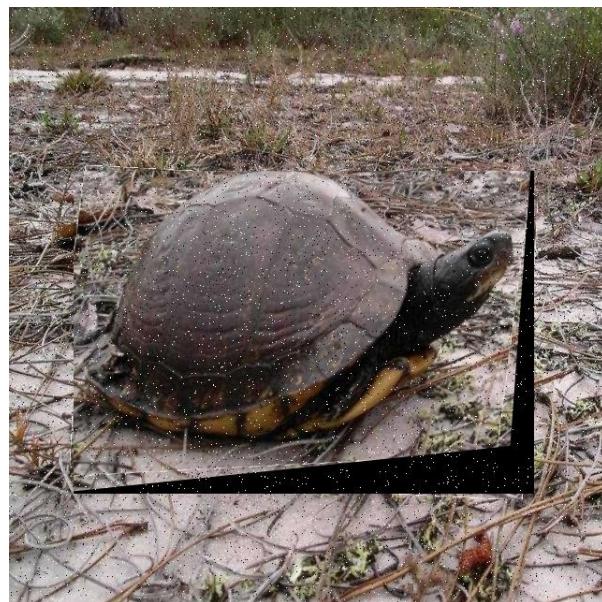
2.2.2 Problémy při učení

Tak jako i my lidé se při učení občas dostáváme do problémů a nepochopení látky, kterou se zrovna učíme, je tomu stejně tak i u učení CNN.

2.2.2.1 Trénovací data

Jedním z příkladů klasického problému při učení sítí může být nedostatek dat nebo jejich kvalita. Počet potřebných dat vždy závisí na robustnosti sítě a daném problému, kterému má síť porozumět a naučit se jej řešit. Obecným pravidlem je alespoň 1000 obrázků do každé třídy, kterou se má síť naučit. [12]

Pro zajištění dostačného počtu trénovacích dat lze využít technika **augmentace dat**, díky které můžeme trénovací data přetransformovat na nová data se změněnou podobou. Díky augmentaci vytvoříme nová data, která můžou mít jiný barevný kontrast, rotaci, měřítko anebo například nanesenou masku či šum.



Obrázek 5. Příklad augmentovaného obrázku.

2.2.2.2 *Overfitting*

Problém zvaný overfitting můžeme chápát jako **přeučení** sítě. Je charakteristické tím, že se síť příliš přizpůsobila trénovacím datům a nebude schopná určovat přesné výsledky klasifikace na nových datech, které ještě „neviděla“.

K monitorování tohoto jevu se při trénování využívá část datasetu zvaná **testovací** nebo **validační**. Jedná se o menší výseč z datasetu, která obsahuje data, která nesmí být využita při samotném trénování sítě, ale právě naopak se využívají pro validaci a posouzení správného učení. Během trénování se vždy po daných krocích (většinou jedné iteraci) tyto validační data aplikují na doposud naučenou síť a vrátí hodnocení kvality sítě v podobě metriky **loss**. Hodnota **loss** by měla být co **nejmenší** a udává chybovost, v jakou síť porozuměla validačním datům. [13]

Při tomto monitorování můžeme jako jedno z možných řešení zamezení overfittingu využít tzv. **early stopping**. Předčasné ukončení trénování sítě provedeme v případě, když se hodnota loss udržuje v podobných hodnotách nebo začíná stoupat. V tomto momentě můžeme trénování zastavit a tím zamezit dalšímu učení, které by mohlo mít za následek špatnou generalizaci na nových datech. [13] Mějme však na paměti, že vynucení předčasného zastavení můžeme provést pouze tehdy, pokud při učení generujeme tzv. záchytné soubory obsahující stav doposud naučené sítě.

Dalším možným řešením je zajištění většího počtu trénovacích dat, například již zmíněnou augmentací, na kterých se síť může učit. To zapříčiní větší počet parametrů, které se musí síť naučit, a tudíž redukuje její možné přeučení.

Mezi často používanou techniku zaměřující se na zabránění přeučení spadá i **dropout** regulačce. Jejím úkolem je ukončení spojení mezi náhodně vybranými aktivacními neurony v každé iteraci učení, čímž zvyšuje robustnost celé sítě. [13] [14]

2.2.2.3 *Underfitting*

Underfitting je téměř opakem overfittingu. Jedná se o problém, který je specifický nedostatečným časem trénování nebo příliš jednoduchou architekturou sítě, což opět způsobuje špatnou generalizaci na nových datech. [14]

Z tohoto lze chápát, že pokud se snažíme zabránit problému přeučení, můžeme se snadno dostat do opačného problému „nedoučení“ a zase naopak. Proto se při trénování sítě snažíme nalézt „zlatou střední cestu“ pro zajištění kvalitních výsledků finálního modelu sítě.

2.2.3 True vs False a Positive vs Negative

Jedná se o 4 základní pojmy při klasifikování vstupních dat na naučeném modelu sítě, využívající se pro finální zhodnocení přesnosti modelu.

Pro lepší pochopení následujících významů v tabulce (Tab. 1) je potřeba zvážit, že je naše síť naučena rozpoznávat 2 druhy zvířat: slon a zebra.

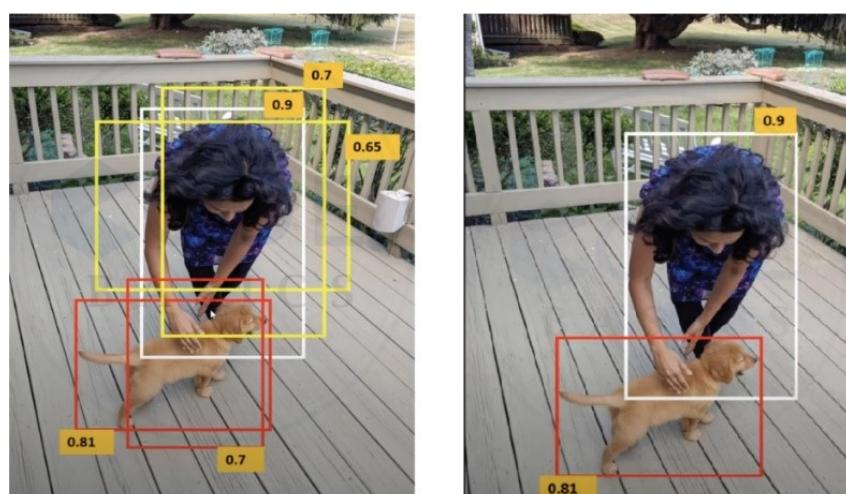
Tabulka 1. Pojmy při klasifikování dat modelem.

<i>Název pojmu</i>	<i>Vstupní obrázek</i>	<i>Klasifikace modelem</i>	<i>Vyhodnocení</i>
<i>True-Positive</i>	Slon	Slon	SPRÁVNÉ
<i>True-Negative</i>	Jelen	Nerozpoznáno	SPRÁVNÉ
<i>False-Positive</i>	Nosorožec	Slon/Zebra	CHYBNÉ
<i>False-Negative</i>	Zebra	Nerozpoznáno	CHYBNÉ

2.3 Technika Non-Max-Suppression

Jedná se o techniku aplikovanou v poslední části detekci objektů v obrazu. Její úlohou je odstranit redundantní výskytu detekcí stejného objektu a zajistit tak pouze jeho jednu nejlepší lokalizaci.

Její funkčnost je prostá a jednoduchá. Začíná tím, že si získá lokalizaci s nejvyšší udanou přesností nějakého objektu a nad touto oblastí provede **průnik** všech ostatních lokalizací nad **daným** objektem. Pokud je hodnota průniku větší než nastavený práh, je daná lokalizace odstraněna. Tímto prahem se ujišťujeme, že se jedná o lokalizace nad stejným objektem, a ne nad stejnou třídou v jiné části obrazu. Tato operace se opakuje tak dlouho, dokud se průniky vyskytují. [15]



Obrázek 6. Obrázek znázorňující použití NMS techniky. [15]

2.4 Detektory

Různé druhy populárních algoritmů neboli **detektorů** se běžně používají pro trénování vlastních modelů sítě. Tyto detektory se dělí na tzv. **one-stage** a **two-stage** detektory, jejichž hlavní rozdíl je v tom, jak prochází vstupní obraz a detekují v něm objekty.

2.4.1 Dvoustupňové detektory

Dvoustupňové detektory (**two-stage**) se skládají ze dvou na sobě závislých operací. V první operaci se metodou „region proposal“ generují tzv. **oblasti zájmů** obsahující návrhy regionů možných objektů v obrazu. V druhém kroku se tyto oblasti aplikují na konvoluční neuronovou síť, která dané oblasti klasifikuje do tříd a po zdokonalení se vytvoří ohraňující rámeček kolem nalezeného objektu. [16] V praxi to znamená **výrazně pomalejší** detekci, ale vyšší přesnost, než jakou mají jednostupňové detektory.

Významným pokrokem v detekci objektů byl detektor **R-CNN**, který na vstupním obrazu generuje přibližně 2000 regionů použitím „region proposal“ algoritmu **selektivního vyhledávání**², což ovšem způsobuje zdlouhavé trénování i testování. Jako náhrada za R-CNN vznikl detektor Fast R-CNN a **Faster R-CNN**, který již nevyužívá pomalé selektivní vyhledávání, ale obsahuje malou konvoluční síť zvanou **Region Proposal Network**, která generuje oblasti zájmů velice rychle. Tímto se dvoustupňový detektor Faster R-CNN přiblížil k rychlosti jednostupňových detektorů, ale stále nedosahuje dostatečnou rychlosť a snížení potřebného výkonu jako detektory jednostupňové a z tohoto důvodu je jejich použití v mobilních zařízeních kvůli omezenému výkonu takřka nemožné a nepraktické. [17]

2.4.2 Jednostupňové detektory

Oproti two-stage detektorům se ve **one-stage** detektorech aplikuje pouze jedna operace, která zajistí region a třídu detekovaného objektu. Jedná se o plně propojenou konvoluční vrstvu, která přeskakuje zdlouhavé generování návrhů možných objektů, ale přímo je detektuje v jednom kroku. Díky tomu je mnohem rychlejší, ale v několika případech na two-stage detektorech ztrácí v přesnosti. [16] Mezi hojně využívané one-stage detektory i v oblasti mobilního rozpoznávání patří YOLO, SSD nebo EfficientDet.

² Rozdělí obraz na malé regiony s podobnými vlastnostmi (barva, textura, ...), které by mohli tvořit jeden objekt a ty pak seskupuje do větších regionů, na které je aplikována klasifikace. [43]

2.4.2.1 YOLO

You Only Look Once je ve své oblasti velice populárním detektorem. Jeho první verze byla publikovaná v roce 2015 a jeho nejnovější verze, která má další „pod verze“, nese název **YOLOv5**. YOLO využívá několik konvolučních vrstev pro extrakci vlastností obrázku a několik Dense vrstev pro klasifikaci objektu. [18] Hlavní myšlenkou tohoto detektoru je pomyslné rozdělení vstupního obrazu do mřížky, kde se v každé její buňce predikuje možný výskyt objektu. Tato predikce je udávána vektorem, který při pozitivním nálezu obsahuje třídu a lokalizaci detekovaného objektu. Tento vektor může být reprezentován následovně:

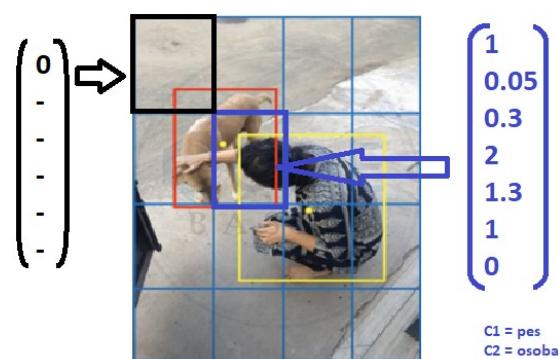
- $[P_c, B_x, B_y, B_w, B_h, C_1, C_2, C_{...}]$, kde P_c značí pozitivitu (1) nálezu nebo ne (0). Pokud je nález pozitivní (1), na dalších 4 místech je určena poloha objektu v dané buňce skrze souřadnice X a Y a velikosti šířky a výšky. Hodnoty C_i udávají samotné třídy sítě a aktivována (1) je pouze ta, která je právě predikovaná. Pokud je nález negativní (0), další hodnoty ve vektoru jsou irrelevantní nebo prázdné.

Z ukázky vektoru si lze povšimnout, že vektor musí mít předem stanovenou velikost, aby mohl poskytovat predikce více než jedné třídě a popřípadě i určit více tříd v jedné buňce mřížky. Pro takový výpočet se používá následující rovnice (3).

$$V_{size} = B * (5 + C) \quad (3)$$

Hodnota B udává možný počet výskytů objektů v jedné buňce, konstanta **5** značí prvních 5 hodnot vektoru (pozitivita nálezu a souřadnice) a C je počet existujících tříd. [15]

Obvyklá velikost mřížky je 19×19 . Významem každé buňky není pouze vytvoření daného vektoru obsahující informace o predikci, ale tato buňka je dále použita jako středový bod výsledného objektu v obraze. [19]



Obrázek 7. Obrázek s mřížkou 4×4 obsahující objekty se středovými body a vektory predikce. [15]

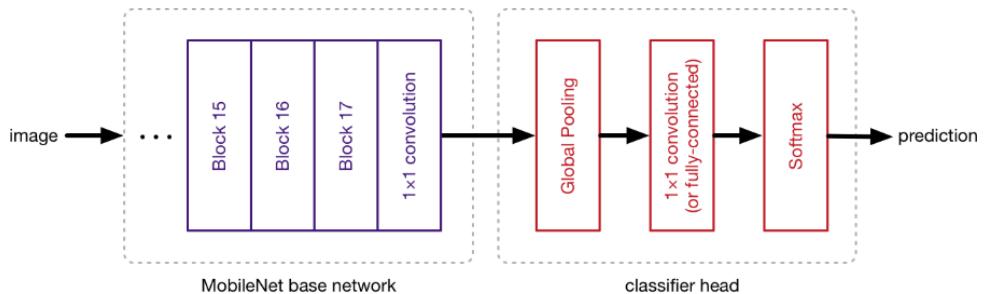
Celá klasifikace i lokalizace objektu v obrazu proběhne téměř v jeden okamžík a celkový počet výsledných lokalizací jednoho objektu může být vysoký. Z tohoto důvodu se na konci YOLO detektoru aplikuje již zmíněná technika **Non-Max-Suppression** (2.3). [15]

2.4.2.2 SSD MobileNet V2

Jedná se o spojení dvou samostatných sítí Single Shot Detector a MobileNet, které spolu tvoří silnou kombinaci pro rychlé a účinné detekování objektů v reálném čase, jehož použití je přizpůsobeno zařízením s omezeným výkonem.

SSD je specifický tím, že k detekování objektů využívá několik konvolučních vrstev lišících se ve velikosti, které může výsledný objekt mít. Díky tomu je schopný v jeden okamžík generovat několik ohraničení různých velikostí pro libovolný objekt. Jedná se o klíčový prvek tohoto detektoru, jehož hlavním úkolem je predikce ohraničení a jejich následná klasifikace do tříd. [20]

MobileNet je konvoluční neuronová síť přizpůsobena pro výpočty na mobilních zařízeních. Pomocí konvolučních vrstev se stará o extrahování klíčových vlastností z obrazu a tím redukuje celkový počet parametrů potřebných k naučení sítě. [21]



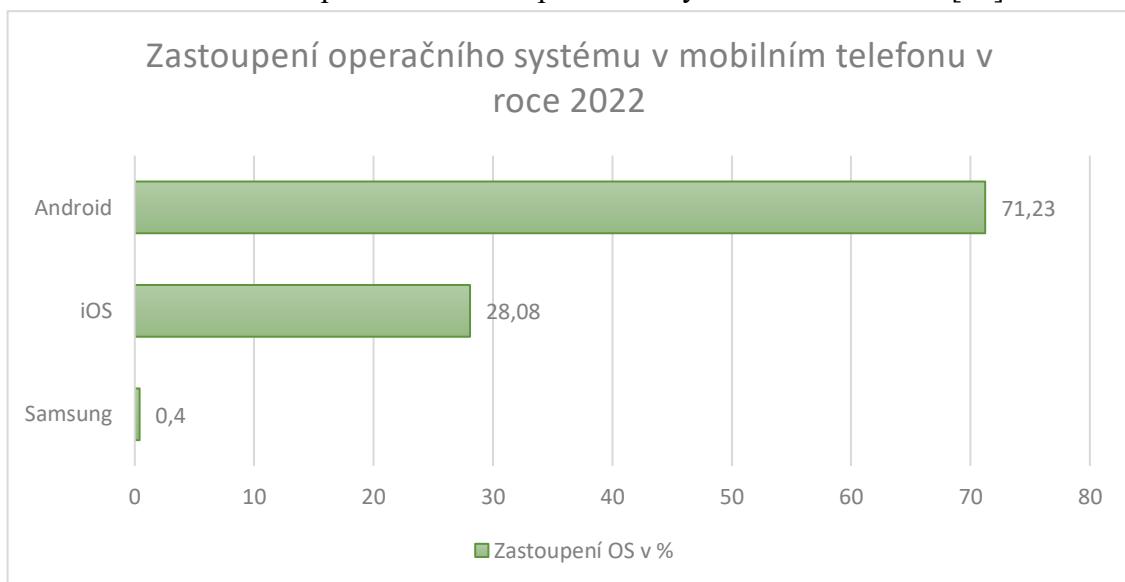
Obrázek 8. Spojení MobileNet a SSD. [22]

Momentálně (4.2.2023) se jedná o jediný detektor podporující konverzi do formátu **tflite** použitelného pro mobilní aplikace Android skrze *TensorFlow Object Detection API*, a proto byl tento detektor využitý v praktické části této práce. [23]

3 VÝVOJ MOBILNÍCH APLIKACÍ

V dnešní době rozlišujeme 2 hlavní hráče na poli operačních systémů, pro které je možné vyvíjet mobilní aplikace. Jedná se o zařízení s operačním systémem iOS nebo Android. Z grafu (Graf 1) lze vidět, že ve světě převládá operační systém Android a jelikož je tato práce zaměřena na vývoj mobilní aplikace na platformě Android, bude velká část této kapitoly věnovaná právě tomuto operačnímu systému.

Graf 1. Zastoupení mobilních operačních systémů za rok 2022. [24]



3.1 Druhy vývoje mobilních aplikací

Vývoj mobilních aplikací je velkým tématem této doby, díky čemuž přišlo na trh několik možností, jak jej zrealizovat. Hlavní otázkou před započetím vývoje je výběr platformy, pro kterou bude aplikace vytvořena. Pokud by si zákazník navíc přál podporu pro více platform, musí se zvážit i tato možnost. Odpověď na tuto otázku je důkladná analýza cílů aplikace a jejich požadavků.

Jak tedy bylo řečeno, existuje možnost vytvořit aplikaci závislou pouze na jednu platformu použitím specifického programovacího jazyka dané platformy, ale taky možnost vytvoření aplikace, jejíž stejný kód bude díky různým nástrojům spustitelný i na odlišných platformách.

Jednotlivý popis těchto druhů vývoje mobilních aplikací je rozepsán v kapitolách níže. [25]

3.1.1 Platformě závislé

Platformě závislý vývoj můžeme dělit pouze na jeden možný druh zvaný **Nativní vývoj**. Jedná se o nejstarší variantu vývoje aplikací závislou na konkrétní platformě vyvýjeného zařízení. Takto vytvořené aplikace v sobě nesou menší bezpečnostní riziko, jelikož jsou přesně přizpůsobeny dané platformě a jejímu standardu.

3.1.1.1 *Nativní vývoj*

Nativní vývoj aplikací v praxi znamená, že vyvýjená aplikace bude spustitelná pouze na příslušné platformě operačního systému, pro který je aplikace naprogramovaná. Tento způsob vývoje převládá u takových aplikací, které se neobejdou bez veškerého výkonu a hardwaru mobilního telefonu, jako je například fotoaparát, čipy NFC nebo GPS, popřípadě přímá práce s paměťovým adresářem.

Při takovém vývoji je ale nutné vytvořit zvlášť aplikaci pro platformu Android a zvlášť aplikaci pro platformu iOS a díky tomu je celkový vývoj mnohdy časově i finančně náročnější. [25]

3.1.2 Platformě nezávislé – multiplatformní

Pod multiplatformní vývoj aplikací spadá právě takový vývoj, jehož výstup je možný zprostředkovat na kterékoliv dostupné platformy bez nutnosti použití nativního přístupu. Dělí se na 2 druhy, které si jsou velmi blízké. Oproti nativnímu vývoji jsou ale takto vytvořené aplikace náchylnější na bezpečnost a nevyužití celého potenciálu zařízení. Vývoj je ovšem snadnější, rychlejší a levnější. [25]

3.1.2.1 *Webový vývoj*

Přesným opakem nativního vývoje mobilních aplikací je webový vývoj. Jedná se o responsivní webovou stránku využívající webové technologie, jako je standard HTML 5 a JavaScriptové frameworky. Tyto aplikace většinou nemají přímý přístup k hardwaru mobilního telefonu, s výjimkou GPS nebo základní funkčnosti kamery. Jelikož takto vyrobené aplikace využívají pro svůj běh pouze internetový prohlížeč, není možné je samostatně nainstalovat do zařízení, ale podporují uložení na plochu zařízení. [26]

3.1.2.2 Hybridní vývoj

Hybridní vývoj je dosti podobný k vývoji webovému. Opět se jedná o aplikaci využívající webové rozhraní internetového prohlížeče, které se nazývá WebView. Toto rozhraní je „zabaleno“ v nativní části aplikace, díky které je možné aplikaci nainstalovat na libovolné zařízení. K výše zmíněnému „zabalení“ se využívá tzv. wrapper technologie, například Ionic Capacitor, která se postará, aby webová aplikace dokázala komunikovat s hardwarem mobilního telefonu a tím využívat jeho funkce. [26]

3.2 Architektonické vzory

Slouží pro nastolení přístupu a pravidel při vývoji mobilní aplikace. Díky dodržení všech pravidel některého z vybraného vzoru, má aplikace pevně danou strukturu, ve které se dá snadno orientovat díky rozdělení do určitých celků a v budoucnosti rozšiřovat o novou implementaci.

Následující příklady vzorů spadají pod tzv. třívrstvou architekturu, která se již podle názvu vyznačuje třemi základními vrstvami: [27]

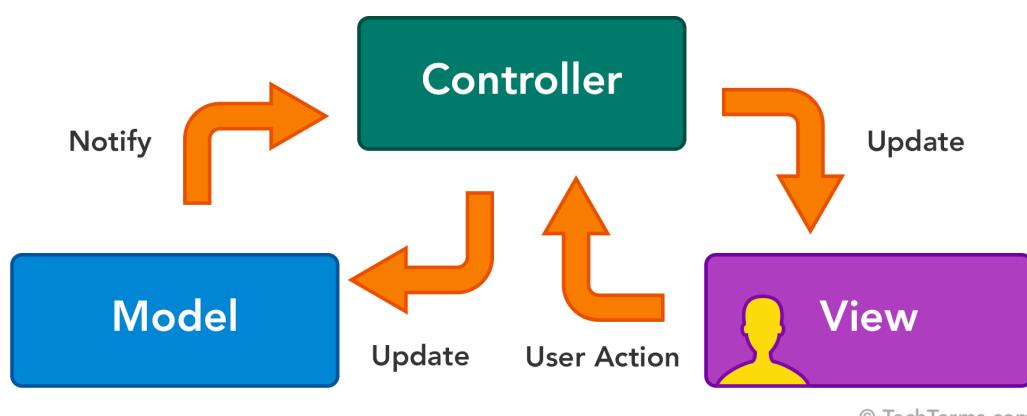
1. **Prezentační vrstva** se stará o prezentování/zobrazení uživatelského rozhraní uživateli dané aplikace a umožňuje mu tím aplikaci ovládat. Jedná se o platformě závislou vrstvu – webová, desktopová, mobilní.
2. **Aplikační vrstva** se stará o veškeré operace a výpočty v běhu aplikace. Zprostředkovává komunikaci mezi Datovou a Prezentační vrstvou. Zpracovává vstupy od uživatele, nebo naopak posílá data do Prezentační vrstvy pro zobrazení uživateli.
3. **Datová vrstva** je vrstva, která se stará o ukládání a práci s daty aplikace. Může se jednat o lokální nebo serverové úložiště nebo databázi. Tato data jsou vždy dostupná pro Aplikační vrstvu.

3.2.1 Model-View-Controller

Model-View-Controller neboli **MVC**, je jeden z nejčastějších architektonických vzorů při tvorbě webových aplikací. Jeho rozdělí do tří vrstev je následující: [28]

- Model – Tato část představuje logiku a práci s daty aplikace. Stará se o načítání, ukládání a zpracování dat z databáze a při tom nemá tušení o tom, jak jsou data na aplikaci závislá a zobrazená uživateli.

- View – View můžeme z angličtiny přeložit jako *pohled*, a to je přesně tím, čím je. Prakticky se jedná o grafické zobrazení aplikace s daty, které View čerpá z Modelu. Tato vrstva by nikdy neměla být schopná měnit data aplikace a vykonávat jinou logiku, než je zobrazování šablony s UI.
- Controller – Kontrolér většinou reaguje na události uživatele nebo samotného systému. Komunikuje s Modelem a na základě vstupu od uživatele rozhoduje, které View aplikace se načte, popř. s jakými daty.



© TechTerms.com

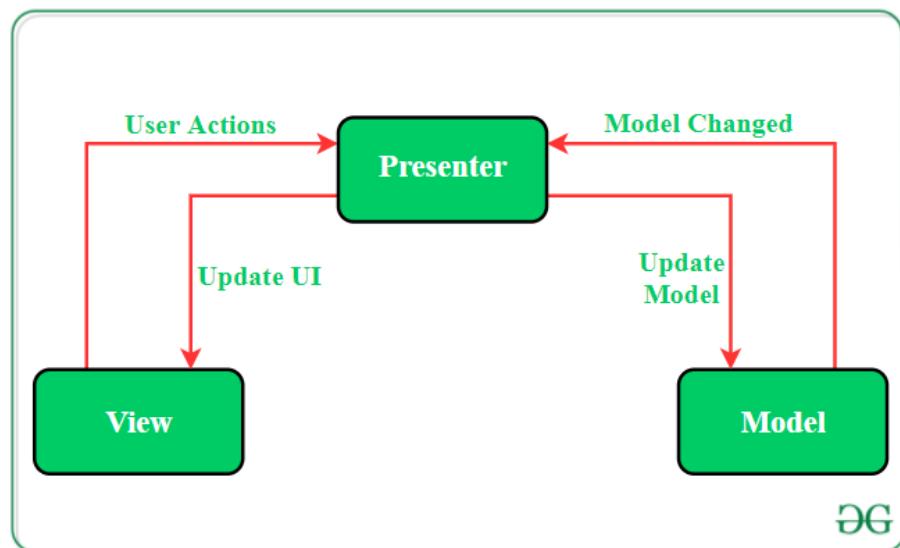
Obrázek 9. MVC diagram. [29]

Z tohoto rozdělení je patrné, že všechny vrstvy jsou na sobě nezávislé a rozdělené na bloky, které souvisí pouze spolu. Díky tomu je aplikaci možné snadno škálovat a modifikovat bez obav porušení některé z existující funkcionality aplikace.

3.2.2 Model-View-Presenter

Architektonický vzor MVP vychází ze vzoru MVC a díky tomu mají téměř identické chování. Model opět reprezentuje datovou vrstvu, View zobrazuje data uživateli a také reaguje na požadavky uživatele, které posílá na svůj Presenter, který se navíc stará o komunikaci mezi vrstvami Model a View.

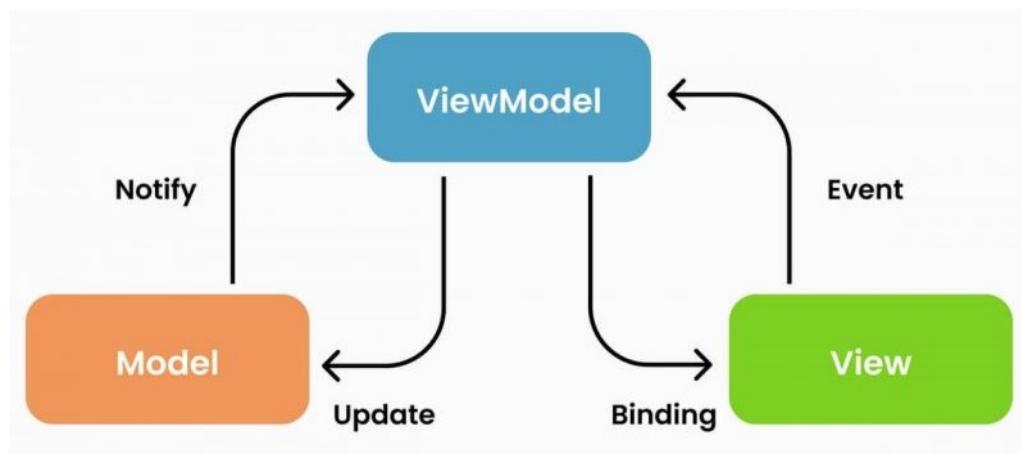
Změna od MVC přichází u samotné komunikaci mezi View a Modelem, které oproti MVC nejsou nyní napřímo propojeny, a tudíž musí Presenter předat data z Modelu do View sám a tím samotný View naformátovat. To umožňuje větší flexibilitu při automatizovaném testování. [30]



Obrázek 10. MVP diagram. [30]

3.2.3 Model-View-ViewModel

Model-View-ViewModel (MVVM) je silným architektonickým vzorem pro tvorbu rozsáhlých, nejen, mobilních aplikací. Spočívá ve vytvoření vazeb mezi daty a uživatelským rozhraním. Tato vazba probíhá za pomocí binding třídy ve ViewModelu a stará se o držení dat, jejich aktualizování a přenosu na View. To znamená, že z UI je zcela odstraněn kód aplikace a data jsou pevně vázány právě na třídu ve ViewModelu, kde si neustále uchovávají aktuální stav. Tento architektonický vzor je použitý při vývoji mobilní aplikace popsáné v praktické části této práce.



Obrázek 11. MVVM diagram. [31]

- **Model** představuje identickou funkci jako u vzorů MVC a MVP, to je, že se stará o získávání dat z databáze.

- **View** popisuje uživatelské rozhraní, stará se o komponenty, které jsou ovlivněné vstupy od uživatele. Nemělo by obsahovat aplikační logiku.
- **ViewModel** – nejdůležitější část tohoto arch. vzoru. Obsahuje veškerou aplikační logiku, stará se o udržování stavu a aktualizování dat podle potřeb View.

Jelikož ViewModel nemusí být pevně vázaný na konkrétní View, může být tak automaticky použito ve více případech a tím značně redukovat celý kód aplikace, kterou je pak možné snadno testovat a rozšiřovat. [32]

3.3 Android

Jedná se o open-source projekt založený na Linuxovém jádru vyvíjeným společností Google od roku 2008. [33] Dá se považovat za krále mobilních operačních systémů, jelikož se jedná o nejvíce používaný operační systém v mobilních zařízeních a před druhým nejčastějším operačním systémem iOS si každoročně udržuje odstup přibližně 43 % ve využívaných zařízeních. [24]

Aplikace vyvíjené pro tento systém se programují v jazyku Java nebo v moderním jazyce Kotlin. Vývojářům jsou dostupné různé nástroje v podobě SDK balíčků, které hrají důležitou roli při vývoji jakékoli aplikace. Tyto SDK nástroje jsou běžnou součástí oficiálního vývojového prostředí Android Studio a spadají mezi ně například nástroj **ADB**, jenž se stará o spuštění vyvíjených aplikací v android emulátoru nebo na připojeném fyzickém zařízení.

3.3.1 Úrovně API

Application Program Interface (API) je číselný kód označující podporovanou funkcionalitu všech vydaných verzí operačního systému Android. Každá verze operačního systému Android má tedy svou vlastní API úroveň udávající, jaké funkce a možnosti jsou dostupné pro vývojáře mobilní aplikace. Starší operační systémy s nižší úrovní API nemusí být podporovány aplikacemi, které jsou naprogramované pro vyšší API úrovně, jelikož by mohly v dané aplikaci chybět důležité funkce a z tohoto důvodu musí vývojáři aplikací zvážit, kterou API úroveň zvolí pro dosažení co nejvyššího počtu podporovaných zařízení.

Při vývoji mají programátoři dostupné všechny potřebné informace a pomocí následujících základních značek a parametrů si mohou tyto informace nastavit [34]:

- **Build.VERSION.SDK.INT**: Jedná se o značku dostupnou programátorovi aplikace přímo v kódu. Obsahuje informaci o aktuálním API levelu na daném mobilním

zařízení. S pomocí této značky může upravovat funkciálnitu aplikace, která by se v jiných API levelech mohla chovat odlišně, např. čtení a zápis v mobilním adresáři.

- **minSdk**: Jedná se o důležitý parametr při vytváření aplikace. Uchovává informaci o **minimálním** levelu API, na kterém může aplikace fungovat. Pokud bude mít tedy mobilní telefon API level 19 (Android 4.4) a aplikace bude obsahovat **minSdk 21** (Android 5.0), nebude tuto aplikaci možné na daném zařízení najít, popř. spustit.
- **targetSdk a compileSdk**: Mělo by se jednat vždy o stejně nastavenou úroveň API. Udává verzi operačního systému, pro kterou je aplikace určena.

Toto a mnohé další je součástí důležitého souboru **Gradle**. Tento soubor obsahuje veškeré potřebné nastavení pro správné „sestavení“ aplikace a také tzv. *dependencies*, odkazy na používané knihovny ze strany jiných vývojářů.

V tabulce níže (Tab. 2) můžeme vidět přehled všech aktuálně existujících verzí operačního systému Android s nejen jejich příslušnou API úrovní a kódovým označením. **Kódové označení** je důležité zejména při vývoji aplikace, kdy je nutné v zařízení zjistit verzi systému a podle toho se rozhodnout, jaká část kódu bude použita (většinou z důvodu odlišného chování určitých funkcí mezi verzemi). V kódu (Kotlin) jej lze spolu s API levelem porovnat následovně: `if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q){}`.

Zároveň si můžeme všimnout modrého označeného řádku obsahující Android verzi 5.0 s API úrovní 21. Jedná se o minimální podporovanou API verzi pro framework Jetpack Compose, který hraje značnou roli této práce.

Tabulka 2. Přehled API verzí s verzí Android. [34]

ANDROID VERZE	KÓD VERZE	API LEVEL	ROK VYDÁNÍ
1.0	Base	1	2008
1.1	Base_1_1	2	2009
1.5	Cupcake	3	2009
1.6	Donut	4	2009
2.0	Eclair	5	2009
2.0.1	Eclair_0_1	6	2009
2.1	Eclair_MR1	7	2010
2.2	Froyo	8	2010
2.3.0 – 2.3.2	Gingerbread	9	2010
2.3.3 – 2.3.7	Gingerbread_MR1	10	2011
3.0	Honeycomb	11	2011
3.1	Honeycomb_MR1	12	2011

3.2	Honeycomb_MR2	13	2011
4.0.1 – 4.0.2	Ice_Cream_Sandwitch	14	2011
4.0.3 – 4.0.4	Ice_Cream_Sandwitch_MR1	15	2011
4.1	Jelly_Bean	16	2012
4.2	Jelly_Bean_MR1	17	2012
4.3	Jelly_Bean_MR2	18	2013
4.4	Kitkat	19	2013
4.4W	Kitkat_Watch	20	2014
5.0	Lollipop	21	2014
5.1	Lollipop_MR1	22	2015
6.0	M	23	2015
7.0	N	24	2016
7.1	N_MR1	25	2016
8.0	O	26	2017
8.1	O_MR1	27	2017
9	P	28	2018
10	Q	29	2019
11	R	30	2020
12	S	31	2021
12L	S_V2	32	2022
13	Tiramisu	33	2022

3.3.2 Android Studio

Android Studio je oficiální vývojové prostředí pro tvorbu aplikací podporující operační systém Android založené na silných nástrojích softwaru IntelliJ IDEA od firmy JetBrains. Jedná se o neustále vyvíjející se IDE s novými technologiemi. Díky multiplatformní podpoře s ním lze pracovat na operačních systémech Windows, Linux, MacOS i Chrome OS. [35]

Android Studio obsahuje širokou škálu výkonných nástrojů, intuitivní ovládání a příjemný vzhled k zajištění co největší produktivity během vývoje aplikace.

Mezi jeho nástroje můžeme zařadit:

- inteligentní našeptávač kódu, který intuitivně doplňuje části kódu nebo importuje potřebné balíčky funkcí
- nástroj pro ladění aplikace, díky kterému lze snadno identifikovat problémy v běhu aplikace
- možnost náhledu uživatelského prostředí či jeho vizuální tvorba
- manažer virtuálních zařízení, který umožňuje vytvořit a spravovat virtuální (mobilní) zařízení, na kterém se může vyvíjená aplikace spouštět a testovat

- možnost zvolit si programovací jazyk Java nebo Kotlin
- zabudovaná správa verzí skrze Git systém.

3.4 Framework Jetpack Compose

Jetpack Compose je nový moderní framework pro vývoj uživatelského rozhraní pro aplikace založené na platformě Android. Trhu byl představen v roce 2021 firmou Google a od té doby prošel velkým počtem aktualizací přinášející mnoho změn a novinek. Funkčnost tohoto frameworku závisí v použití programovacího jazyka Kotlin a minimálního SDK na levelu 21. Přínosem tohoto frameworku byla i knihovna **CameraX**, díky které lze velice jednoduše pracovat s kamerou mobilního telefonu. [36]

Posláním Jetpack Compose je nahradit dlouhou dobu používaný návrh UI pomocí XML, jenž funguje stále poměrně bezchybně, ale jeho vývoj je zdlouhavý a mnohdy náročný. Návrh UI pomocí Jetpack Compose se provádí vytvářením samostatných funkcí označených anotací `@Compose`. V těle této funkce již definujeme UI pomocí existujících komponent **Material Designu** a s využitím programovacího jazyka Kotlin můžeme jejich chování naprogramovat. Díky tomu, že se jedná o „obyčejné“ funkce, lze je klidně vnořovat do sebe a rozrůstat tak celkový návrh UI s minimálním množstvím nově napsaného kódu při zachování jeho čitelnosti.

Během vývoje UI má programátor možnost zobrazit si kteroukoliv `@Composable` funkci pomocí anotace `@Preview` přímo v IDE Android Studio, a tak ji kompletně navrhnut bez samotného spuštění aplikace.

Aby mohl uživatel aplikaci plnohodnotně používat, je potřeba, aby se v aplikaci aktualizovali data vždy, kdy je potřeba. V Jetpack Compose se ke změně stavům využívá tzv. **rekompozice**. Rekompozice UI se v Jetpack Compose volá v momentě, kdy se změní stav libovolné proměnné k tomu určené – obvykle označeny jako **remember** s typem udávající **stav**. [37] Tyto data je taky možné uchovávat ve ViewModelu, který může být specificky vytvořen pro různé funkce.

Samotný framework se nám dále stará o veškeré aktivity a životní cykly, které se v aplikaci nacházejí a tím její vývoj velice usnadňuje.

4 SEZNÁMENÍ S PODOBNÝMI PROJEKTY

Během zjišťování informací na dané téma této práce autor narazil na 2 zajímavé projekty jiných autorů (firem), které mají několik společných rysů. Jedna z hlavních podob je samozřejmě schopnost rozpoznávat různé objekty reálného světa pouze za pomocí mobilní aplikace a fotoaparátu.

4.1 Google Lens

Prvním kandidátem je aplikace Google Lens vyvinutá a představená v roce 2017 společností Google. Tato aplikace je jednou z nejvyspělejších technologií, která dokáže nejenom rozpoznat nespočet objektů reálného světa a vyhledat k nim relevantní informace, ale zároveň dokáže i vyhledávat a překládat texty v cizích jazycích nebo skenovat čárové kódy a mnohé další. Jedná se tedy o velice chytrého pomocníka do kapsy, který dokáže s vysokou přesností určit nebo napovědět, co se na snímku nachází. [38]

4.2 Seek by iNaturalist

Tato aplikace opět využívá mobilní kamery k rozpoznávání okolních objektů. Tyto objekty jsou ale specificky zaměřeny přímo na živočišné a rostlinné druhy. Aplikace umožnuje uživatelům vypracovávat různé denní výzvy nebo získávat úspěchy rozpoznáváním zvířat a rostlin.

Model vytvořený sdružením iNaturalist dokáže identifikovat přes 10 000 různých kategorií zvířat a rostlin. Výhodou této organizace je, že samotní uživatelé aplikace přispívají sběrem velkého množství dat a tím zlepšují kvalitu výsledného detekčního modelu. V poslední dostupné aktualizaci bylo trénování modelu spuštěno na neuvěřitelných 30 milionu obrázků s přibližně 68 000 druhů organismů. Trénování takového modelu trvá kolem 4 měsíců. [39]



Obrázek 12. Logo iNaturalist. [40]

II. PRAKTICKÁ ČÁST

5 NÁVRH MOBILNÍ APLIKACE

V začátku vývoje mobilní aplikace bylo zapotřebí vytvořit základní návrh celkové aplikace s otázkou, jaké funkcionální požadavky bude obsahovat a jak bude jejich funkčnost provedena. Hlavní myšlenkou bylo vytvořit mobilní aplikaci, která by obsahovala seznam několika zvířat a možnost některé z nich identifikovat za pomocí mobilní kamery přímo v dané aplikaci s důrazem na pohodlné a intuitivní ovládání.

Postupem času se aplikace rozrůstala o nové funkce přidávající uživateli větší přehled a bližší seznámení se zvířaty a zoologickými zahradami v České republice.

5.1 Funkcionální požadavky

Mezi funkcionální požadavky můžeme zařadit to, co se od aplikace jako takové očekává. Většina požadavků byla určena při prvním návrhu aplikace, ale podstatná část nových požadavků byla doplněna později již během samotného vývoje aplikace.

- Systém aplikace umožňuje zobrazit seznam zvířat rozdělených do následujících kategorií: Savci, Ptáci, Plazi
- Systém aplikace umožňuje zobrazení informací o jednotlivém zvířeti
- Systém aplikace umožňuje povolení kamery mobilního zařízení pro možnost zobrazení jejího náhledu
- Systém aplikace umožňuje uživateli přiblížovat a oddalovat náhled kamery pomocí gesta na obrazovce nebo posuvníkem
- Systém aplikace je schopen rozpozнат několik druhů zvířat z náhledu mobilní kamery a případně rozpoznávání i pozastavit
- Systém aplikace je schopen během náhledu fotoaparátu vytvořit snímek a uložit jej do mobilního zařízení s možností tento snímek odstranit do určitého časového intervalu od vyfocení
- Systém aplikace je schopen rozpozнат několik druhů zvířat z vybrané fotografie uložené v mobilním zařízení
- Systém aplikace umožňuje vyhledat zvířata na základně jejich jména
- Systém aplikace umožňuje zobrazit zoologické zahrady v sekci „Objevy“ s jejich statusem objevení a s možností filtrování Vše/Objevené/Neobjevené
- Systém aplikace umožňuje uložit „objevení“ zoologické zahrady do úložiště v aplikaci

- Systém aplikace umožňuje načtení informací ohledně časové dostupnosti zoologické zahrady pomocí internetu a Places API a zobrazení její polohy na mapě pomocí Maps API od společnosti Google
- Systém aplikace umožňuje podobně jako u zoologických zahrad filtrovat a ukládat „viděné“ zvířata v sekci „Objevy“ aplikace
- Systém aplikace umožňuje odstranit zoologickou zahradu nebo zvíře z „objevů“
- Systém aplikace umožňuje zobrazení informací o aplikaci obsahující text a seznam odkazů na stránky, které poskytly aplikaci cenná data během jejího vývoje
- Systém aplikace umožňuje zobrazit různé informace o zvířatech z konkrétních zoologických zahrad v detailu zvířete v sekci „V zoo“
- Systém aplikace umožňuje po povolení lokace získat pomocí internetu a GPS aktuální pozici uživatele, a tak nastavit nejbližší zoologickou zahradu v sekci „V zoo“ u informací zvířete
- Systém aplikace umožňuje během prvotního zapnutí aplikace zobrazit uvítací obrazovku s několika informacemi na použití aplikace

5.2 Nefunkcionální požadavky

Do nefunkcionálních požadavků můžeme zařadit spíše technické specifikace a vlastnosti aplikace, které nejsou spojeny přímo s uživatelem aplikace, ale právě s vývojem aplikace, výběrem technologií, určení kompatibilních zařízení, stabilitou a například i bezpečností při jejím používání.

5.2.1 Kompatibilita

Jedním z hlavních nefunkcionálních požadavků je minimální verze systému Android, na kterém je aplikace podporována. Jelikož byl pro programování vybrán jazyk Kotlin spolu s frameworkem Jetpack Compose, nezbývalo nic jiného, než začít na nejnižším podporovaném SDK levelu Jetpack Compose 21. Postupně ale byla potřeba minSdk zvýšit na level 24, jehož využití ale ve světě bezproblémově přesahuje 96 % [34], takže se nejednalo o závažný problém. Tím bylo zajištěné široké spektrum kompatibilních zařízení, na kterých může být aplikace spuštěna. UI design je optimalizován spíše na mobilní zařízení, ale díky responzivitě frameworku Jetpack Compose je možné aplikaci ovládat i na tablettech s větší velikostí displeje bez nutnosti vytvoření nového designu.

5.2.2 Forma ukládání dat

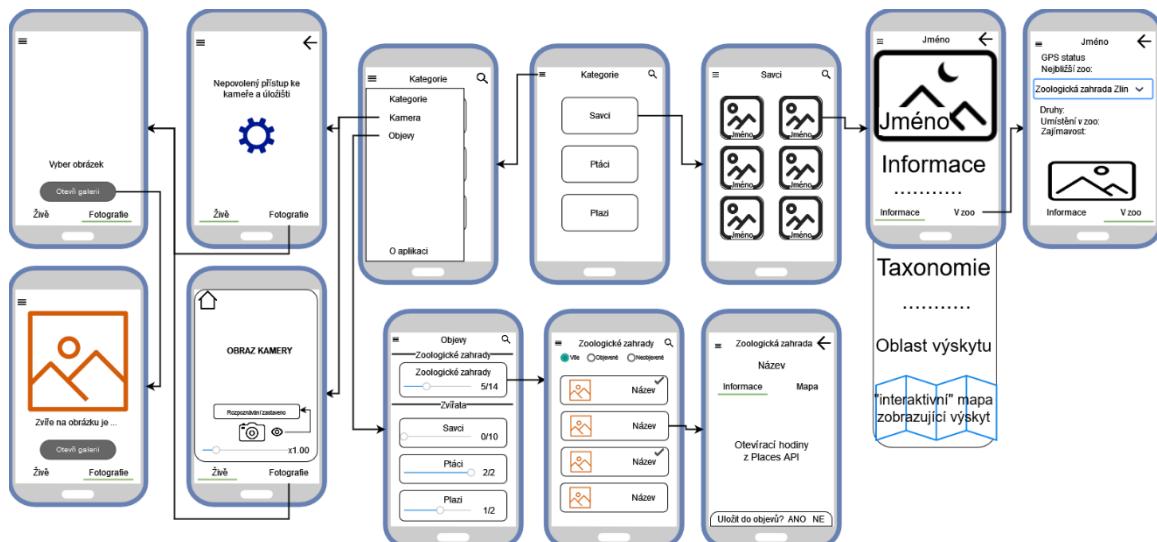
Jedním z dalších konkrétních nefunkcionálních požadavků při vývoji aplikaci bylo zajistit, aby veškeré texty aplikace byly uloženy prostřednictvím Android **Resources Values** a tím vytvořit ucelenou strukturu všech textů v jednotném jazyce. Díky tomu je možné jednoduše vytvořit překlady, a tak aplikaci nabídnout celému světu.

Pro ukládání dat, jako je například status objevení zoologické zahrady nebo zvířete, je v aplikaci využité jednoduchého API **SharedPreferences**. Toto API dovoluje uložení tzv. *key-value* párových hodnot do úložiště aplikace, které je zabezpečené před dostupností v jiných aplikacích.

5.3 Wireframe aplikace

Celá aplikace obsahuje několik oken, které jsou mezi sebou propojeny navigačním panelem, který je umístěn pod tlačítkem v levém horním rohu, popřípadě je dostupný táhnutím prstu zleva doprava kdekoli na obrazovce. Rozvržení a základní podoba oken aplikace byla vytvořena webovým nástrojem draw.io dostupným na následující adrese: <https://app.diagrams.net/>.

Tento návrh hrál důležitou roli během reálného vývoje aplikace, jelikož autorovi poskytoval potřebnou předlohu pro naprogramování UI.



Obrázek 13. Návrh drátového modelu aplikace.

Na obrázku výše (Obr. 13) můžeme vidět poslední verzi drátového modelu obsahující téměř všechny dostupná okna v reálné aplikaci, která byla na základě tohoto vytvořena. Startovací obrazovkou je obrazovka označena názvem „Kategorie“ uprostřed horní řady. Tato

obrazovka je spuštěna po startu aplikace a uživatel se má možnost pomocí navigace dostat na kteroukoliv jinou obrazovku v aplikaci.

Tento wireframe neobsahuje uvítací obrazovku, jelikož ta byla vytvořena téměř v samotném závěru programování aplikace a jelikož je její UI stavba poměrně nenáročná, obešla se bez dodatečného návrhu.

5.4 Data aplikace

Veškerá data celé aplikace jsou uložena 2 hlavními způsoby, které jsou mezi sebou velice odlišné. Databáze jako taková v aplikaci chybí – byla nahrazena položkami **Values**.

5.4.1 Resources Values

Jak je zmíněno výše (5.2), veškerá textová data jsou uložena ve speciálních XML souborech ve formátu android Resources **Values**. Tento formát dovoluje ukládat řetězce, pole a čísla, popřípadě jejich plurály³. Každá položka má svůj klíč, díky kterému se dá v kódu zavolat a použít tak její hodnotu.

K zavolání je dostupná jednoduchá notace: R.*type*.*key*, kde

- *Type* označuje uloženou hodnotu:
 - string – samostatný text
 - array – pole hodnot
 - ...
- *Key* znamená jméno dané položky uložené pod parametrem „name“.

Tímto způsobem jsou v aplikaci uložena nejen všechna textová data, ale také veškeré informace o zvířatech a zoologických zahradách, a to z jednoho prostého důvodu – všechno to jsou taky jen texty.

Nesmírnou výhodou ukládání textů aplikace touto formou je následná možnost vytvoření jazykových mutací pro celou aplikaci bez nutnosti měnit cokoliv uvnitř kódu. Stačí pouze vytvořit nový lokalizovaný soubor obsahující stejně klíče hodnot, kde bude jejich text

³ Plurál = odvození množného čísla v texu (1 zebra, 2 zebry)

přeložen. Aplikace pak sama vybere vhodný lokalizační soubor textů podle nastaveného jazyka v mobilním telefonu.

5.4.2 Resources Drawable

Tohle „úložiště“ poskytuje ukládání veškerých obrázků aplikace. Prakticky se jedná pouze o složku pojmenovanou „drawable“, na kterou se v kódu odkazuje podobným způsobem jako na položky Values – R.drawable.*nazev_obrazku*. Stejně jako u hodnot, i zde vrací tato anotace jeden unikátní index na daný obrázek/hodnotu, kterým pomocí dalších funkcí tato data zobrazíme uživateli na obrazovce.

5.4.3 Úložiště SharedPreferences

Pro ukládání dynamicky se měnících dat bylo použité API SharedPreferences. Díky němu lze rychlým a jednoduchým způsobem ukládat požadavky uživatele při jeho používání aplikace. Mezi již zmíněný status objevení zoologických zahrad a zvířat je toto úložiště použito i pro ukládání potvrzení prvotního otevření aplikace, kdy se uživatel musí „proklikat“ uvítací obrazovkou, která se mu již po dalším otevření aplikace neukáže. To je způsobeno právě uložením správné hodnoty do tohoto úložiště: key=“welcome“, value=true.

5.4.4 Ostatní data

Samozřejmostí je, že aplikace obsahuje i další statická data, které jsou ale uloženy přímo v podobě kódu například ve speciálně navržených třídách.

5.5 Design aplikace

K zajištění příjemného uživatelského prostředí i jeho ovládání se muselo při návrhu aplikace dbát na správnost a dodržení základních charakteristik designování aplikací. Pro pohodlí uživatele muselo být zároveň dovoleno aplikaci spouštět v tzv. **dark módu**, který zajišťuje vhodné přizpůsobení barev do tmavého tónu, a tak například šetřit oči uživatele.

6 VÝVOJ MOBILNÍ APLIKACE

Jak bylo řečeno v předchozí kapitole, samotný vývoj aplikace může začít až po jejím, ale spoň základním, návrhu, zvolením platformy a výběru architektury spolu s programovacím jazykem.

Celý vývoj a testování aplikace probíhalo na fyzickém zařízení Xiaomi Mi 9T s operačním systémem Android ve verzi 11 a rozlišením obrazovky 1080x2340.

6.1 Zvolené technologie

Mobilní aplikace tedy vznikla pro mobilní platformu Android použitím programovacího jazyka Kotlin ve verzi 1.7.10. Pro tvorbu UI byl použitý framework Jetpack Compose, který byl vždy postupně aktualizován, jelikož je stále ve vývoji. Na samotném začátku byl jako grafický standard použitý Material Design 2, který byl po uvedení nové verze nahrazen za Material Design 3. S přechodem vzniklo několik, ať už méně nebo více závažných problémů, které ale byly postupem času odstraněny. Pro veškeré naprogramování aplikace bylo také využito vývojové prostředí Android Studio.

Jelikož se jednalo o vůbec první seznámení těchto technologií s autorem této práce, byl začátek vývoje aplikace poměrně chaotický. Postupem času se ale vývoj uklidnil a zvolený architektonický model MVVM (3.2.3) začínal do aplikace vnášet pořádek, a právě díky němu bylo možné v aplikaci správě uchovávat data a jejich stav při běhu aplikace.

6.2 Začátek vývoje aplikace

Prvním krokem bylo vytvoření prázdného projektu zaměřeného na vývoj aplikace pomocí Jetpack Compose. Android Studio tímto způsobem vygeneruje předpřipravený projekt, který je ihned připravený k použití. Dále následovalo vytvoření GitHub repozitáře pro uchovávání změn v projektu a jeho případnou zálohu pro případ nouze.

Mezi první naprogramovanou obrazovku patří samozřejmě úvodní obrazovka „Kategorie“ obsahující 3 tlačítka pro výběr kategorie zvířat, ve které si může uživatel zvířata procházet. K přechodu mezi obrazovkami bylo potřeba vytvoření navigace, která se o veškeré přechody postará. K tomuto účelu byla vytvořena funkce **Navigation** obsahující komponentu NavHost od Jetpack Compose, která zajišťuje veškeré změny v navigaci na základě cesty. Tyto cesty jsou deklarovány pomocí objektů v sealed třídě Routes.

Potom, co bylo dosaženo fungování navigace, započala práce na horním menu, které je reprezentované 3 vodorovnými čárkami. V Jetpack Compose se jedná o tzv. TopAppBar komponentu, která reaguje buď na tlačítko nebo právě tahem obrazovky zleva doprava. Díky jedné z těchto interakcí dojde k vysunutí navigačního menu na levé straně aplikace a uživatel má možnost se takto dostat na další obrazovky. Výhodou tohoto menu je nejen jeho okamžitá dostupnost, ale hlavně úspora místa na obrazovce.

Po dosažení těchto 2 základních met (fungující navigace a navigační menu) bylo možné pokračovat ve vývoji jakékoli další obrazovky aplikace.

6.3 Náhled kamery

Jelikož je aplikace zaměřená na rozpoznávání zvířat za pomocí mobilní kamery, tak se jedná o jednu z nejdůležitějších obrazovek v aplikaci.

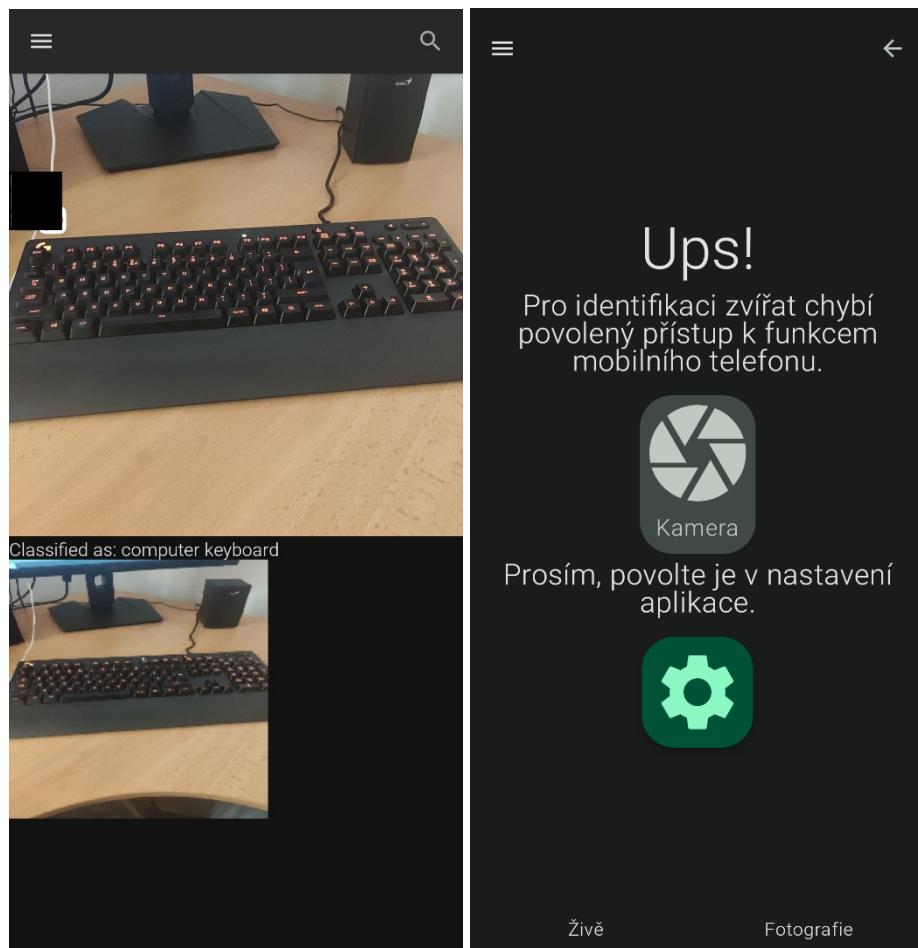
První verze kódu obsahovala funkci o velikosti 323 řádků. Jednalo se o poměrně neoptimální řešení bez využití MVVM vzoru s minimální optimalizací. Ovšem díky tomuto prvotnímu řešení byl autor schopný testovat modely CNN naživo v aplikaci a ověřit si tak jejich fungování.

K práci s kamerou byla využita knihovna **CameraX**, která vývoj velice usnadnila. Díky jedné funkci **setAnalyzer** bylo možné získat předem definovaný snímek z kamery ve formátu YUV_420_888. Tento snímek ale musel být ještě před vstupem do modelu převeden do bitmapy, k čemuž dopomohlo volně přístupné řešení na internetu, a následně zmenšený podle parametru modelu.

Ovšem aby byla kamera vůbec uživateli přístupná, musí ji před jejím prvním zpuštěním povolit. Po prvním odmítnutí má uživatel ještě jednu možnost, potom musí kameru povolit přímo v systémovém nastavení aplikace. Pokud je aplikaci přístup ke kameře odmítnut, je zobrazena stránka s příslušným popisem, viz obrázek níže (Obr. 14). Přístup k povolení je inicializován v souboru `AndroidManifest.xml` (6.8).

Jakmile byl naprogramován přístup ke kameře, začalo testování její funkčnosti s použitým před-trénovaným modelem, který byl schopný určit až 1000 kategorií. Tento model je dostupný na adrese: <https://tfhub.dev/tensorflow/lite-model/efficientnet/lite3/int8/2>. Zprvu model nefungoval, tak jak by měl a přišlo na řadu několik hodin hledání problému, který by to způsoboval. Během testování si autor všiml, že snímky z kamery jsou otočeny o 90°.

Dříve vytvořenou bitmapu tedy pomocí třídy **Matrix** otočil a vše začalo fungovat tak, jak mělo.



Obrázek 14. Na levé straně je vůbec první funkční zobrazení kamery.

Pravá strana obsahuje obrazovku, když chybí povolení ke kameře.

Po zajištění fungování náhledu kamery a ověření správného nastavení jeho obrazu pro vstup do modelu CNN se mohl autor zaměřit na vzhled, nové funkce a potřebnou optimalizaci.

6.3.1 Finální verze náhledu kamery

Po množství optimalizace i přidání nových funkcí byl náhled kamery téměř hotový. Funkce jako taková se ze 323 řádků zmenšila na přibližně 153 řádků, a přitom výsledek obsahuje mnoho nových funkcí. Hlavní optimalizační část byla docílena díky ViewModelu, který nyní uchovává veškerá data kamery, její nastavení, stav i funkci na samotné detekování. Pro zpracování obrazu byla vytvořena třída **ImageClassifier**, která obsahuje funkci **detect** pro načtení obrazu z kamery, jeho transformaci a vložení do modelu. Pokud model vyhodnotí, že se v obrazu nachází nějaké zvíře s jistotou přesahující 50 %, pak vrátí jeho název, v opačném případě vrací prázdný řetězec. Pomocí tohoto výstupu se bud' získá objekt **Animal**

obsahující informace o detekovaném zvířeti, nebo prázdná hodnota **null**. Tato informace je dostupná ve ViewModelu v položce **classifiedAnimal.value**.

6.3.1.1 Informace o detekovaném zvířeti

Z textu předchozí kapitoly již víme, jak se zvíře detekuje, teď zbývá tuto informaci zobrazit uživateli aplikace. Veškerá „logika“ se děje v hlavní **composable** funkci pro zobrazení kamery. Zde se v pravidelných intervalech 800 ms kontroluje hodnota **classifiedAnimal** a pokud je po tomto čase stejná, můžeme s jistotou říct, že pozorujeme stejný objekt (zvíře nebo nic). Pokud je tedy tato hodnota zvíře, zobrazíme ji uživateli tak, že se z pod stránky vysune tzv. **ModalBottomSheet** neboli „modální“ okno, jehož charakteristikou je vyobrazení na nejvyšší úrovni – UI prvky pod ním nejsou aktivní. Jakmile dojde k tomuto okamžiku, aplikace dále pozastaví detekování a v modálním okně zobrazí obrázek detekovaného zvířete s tlačítkem, kterým se uživatel může dostat do jeho detailu. Jakmile uživatel okno zavře (kliknutím mimo okno nebo jeho stažení dolů), popř. se vrátí z detailu zvířete, aplikace pokračuje v detekování.

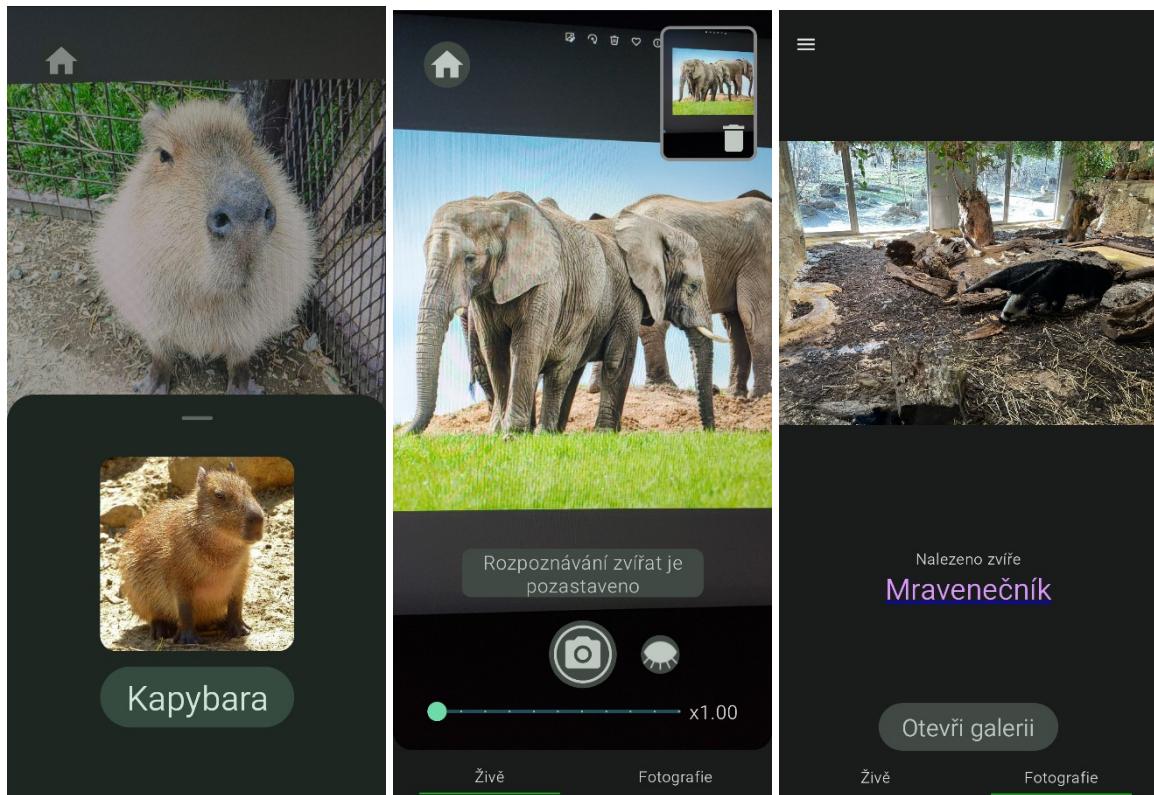
6.3.1.2 Aktualizace UI

Je jasné, že design náhledu kamery musel být patřičně upraven. Nyní obraz z kamery zabírá téměř celou plochu stránky. Opakem je místo ve spodní části, které obsahuje rozdělení detekování zvířat na záložky „Živě“ a „Fotografie“.

Záložka **Živě** je automaticky spuštěna při vstupu do této stránky aplikace a značí zobrazení „živého“ obrazu z kamery, a naopak v záložce **Fotografie** má uživatel možnost vybrat fotografií z úložiště mobilního telefonu, na které si přeje detektovat zvíře.

Na hlavní stránce (živě) navíc přibyly funkce pro **pozastavení detekce, vytvoření snímku** a **zoom** kamery. Horní menu aplikace bylo odstraněno a pro umožnění uživateli tuto stránku opustit bylo v levém horním rohu doplněné tlačítko se symbolem domečku, kterým se po stisku dostane na prvotní stránku aplikace – Kategorie.

Pokud uživatel vyfotí fotografií, její zmenšenina se objeví v pravém horním rohu. Zde ji má ještě po dobu 3 vteřin možnost odstranit. Po uplynutí této doby se zmenšenina fotky vytratí a snímek zůstane uložený přímo v globálním úložišti, kde si jej může uživatel zobrazit.



Obrázek 15. Na levém obrázku můžeme vidět vysunovací okno obsahující detekované zvíře. Prostřední obrázek obsahuje design zobrazení kamery spolu s pořízeným snímkem. Obrázek vpravo ukazuje záložku „Fotografie“ s detekovaným mravenečníkem v obrazu.

6.4 Data zvířat

Z kapitoly výše (5.4.1) vyplývá, že veškerá data zvířat jsou uložena v textové podobě `Values`. Abychom ale k těmto datům mohli jako uživatelé přistoupit v samotné aplikaci, musíme si je prvně nějakým způsobem inicializovat do potřebných proměnných. Způsobů, jak toho docílit bylo hned několik, ale zvítězit mohl pouze jeden.

Pro tuto potřebu tedy byla vytvořena třída, nebo přesněji **objekt AnimalData** v souboru `AnimalData.kt`. Typ **objekt** se v programovacím jazyku Kotlin používá pro vytvoření **singletón** tříd, díky kterým je možné tuto třídu (objekt) jednou inicializovat a poté k datům kdykoliv a kdekoli přistupovat. To znamená velikou výhodu během načítání velkého množství dat, kdy je tímto řešením stačí načíst pouze jednou, a to při zapnutí aplikace. Problém, s kterým se autor potýkal bylo pak samotné načítání dat z hodnot `Values` do proměnných kódu kotlin. Již víme, že každá hodnota má unikátní klíč, tento klíč je ale nutné zapsat přímo v kódu nebo jej získat pomocí funkce. Druhé řešení má za výsledek krátký a jednoduchý kód, který automaticky získá všechny potřebné klíče zvířat ovšem za cenu snížení rychlosti

načítání. Proto bylo během vývoje využito první řešení, kde bylo zapotřebí ručně sepsat list všech klíčů s použitím zmíněné notace R.array.zvire + ke každému zvířeti i jejich obrázky. Toto řešení má asi jako jedinou výhodu v rychlosti načítání dat (několik milisekund).

Vytvořený objekt uchovává instanci dat pro savce, ptáky i plaze zároveň pomocí proměnné nazvané *allAnimalsInstance*, která je typu *mutableListOf<MutableMap<String, Animal>>*. Můžeme si povšimnout třídy **Animal**. Jedná se o **data třídu** indikující způsob uložení informací konkrétního zvířete ve tvaru:

```
@Parcelize
data class Animal(
    val name: String,
    val category: String,
    val info: Map<String, String>,
    val taxonomyMain: Map<String, String>,
    val taxonomyOrder: Map<String, String>,
    val taxonomyFamily: Map<String, String>,
    val taxonomyGenus: Map<String, List<String>>,
    val description: String,
    val previewImage: Int,
    val mainImage: Int,
    var seen: Boolean = false,
    val appearance: List<List<Float>>,
    var canDetect: Boolean = false,
): Parcelable
```

Obrázek 16. Data třída Animal.

Tato třída navíc využívá anotaci *Parcelize* potřebnou pro přenášení vytvořených objektů Animal mezi obrazovkami v navigaci.

Hlavní část v objektu AnimalData je funkce *animalsToList()*. Ta se na základně vstupu stará o vygenerování výstupu do listu *allAnimalsInstance*. Mezi vstupy patří hlavně list klíčů hodnot obsahující textové informace zvířat a klíče k jejím obrázkům. Postupem času navíc i souřadnice do mapy, která je k nalezení u informacích každého zvířete a ve které je možné získat přehled, kde se dané zvíře ve světě nachází. Díky těmto informacím se vytvoří objekt Animal, který se vkládá do mapy (typ **map** v Kotlinu) s příslušným klíčem zvířete (anglický název). Objekt dále obsahuje funkce pro uložení a odstranění zvířete z objevů v aplikaci a také funkci na aktualizování jejich stavu, které je potřebné při startu aplikace. Taktéž

obsahuje funkci, která vrací hodnotu true nebo false (pravda/nepravda), jež udává, zda bylo zvíře objevené.

Podobným duchem se nese i **objekt ZooData** v souboru ZooData.kt. Zde se ale pracuje s daty zoologických zahrad.

6.4.1 Data zvířat ve Values podobě

K výše uvedenému obrázku (Obr. 16) a vytvoření objektu Animal nám tedy patří hodnoty zvířat v jejich podobě Values hodnot. Ty mají podle obrázku níže (Obr. 17) následující formát. Můžeme si povšimnout, že data zvířete jsou shodná s prvními osmi potřebnými daty v data třídě Animal. Zobrazení těchto dat v aplikaci uvidíme na konci následující kapitoly.

```

<string-array name="animalInfo">
    <item>Váha</item>
    <item>Délka těla</item>
    <item>Výška těla</item>
    <item>Délka ocasu/Rozpětí křidel</item>
    <item>Délka života</item>
    <item>Výskyt</item>
    <item>Březost samic</item>
</string-array>

<string name="taxonomy">Taxonomie</string>
<string-array name="taxonomy">
    <item>Říše</item>
    <item>Kmen</item>
    <item>Řád</item>
    <item>Čeleď</item>
    <item>Rod</item>
</string-array>

<array name="Elephant">
    <item>Slon</item>
    <item>Savec</item>
    <item>2.7–6.3 t</item>
    <item>5.5–6.5 m</item>
    <item>2.5–3.5 m</item>
    <item>None</item>
    <item>70–75 let</item>
    <item>Africká savana, polopouště</item>
    <item>1 mládě po 20 měsících</item>
    <item>
        Slon (Loxodonta) je rod slonů z řádu chovavců. 
        \n\nSloni jsou největší suchozemští živočichové na světě. 
        \n\nSloni jsou býložravci a jejich stravou jsou listy, kořeny a květy různých rostlin.
    </item>
</array>
<array name="taxononyElephant">
    <item>Chobotnatci</item>
    <item>Slonovití</item>
    <item>Slon</item>
</array>

<array name="mammalsTaxonomy">
    <item>Živočichové</item>
    <item>Strunatci</item>
    <item>Savci</item>
</array>

```

Obrázek 17. Data zvířat v podobě values hodnot.

6.5 Náhled detailu zvířete

V momentě, kdy fungovalo načítání dat o zvířatech a bylo se k nim tak možné pomocí navigace dostat, začala se práce na samotném programování stránky, která zvířecí data zobrazí uživateli. Z drátového modelu aplikace si lze povšimnout velkého obrázku zvířete zabírajícího téměř půl strany. Ve spodní části tohoto obrázku je pak název daného zvířete. Pod obrázkem už jsou v přehledné formě zobrazeny základní informace a dále delší informační text, který je rolovatelný pro šetření místa. Tímto způsobem byl detail zvířete zobrazen v prvních verzích aplikace.

6.5.1 Mapa zvířat

Během vývoje vznikalo nespočet nápadů na funkcionality, které ovšem nebylo možné vždy zcela zrealizovat. Jedním z realizovaných nápadů ovšem bylo vytvoření mapy, na které by se zobrazovaly tečky v místech, kde se na Zemi vyskytují dané zvířecí druhy. Aby bylo možné tečky vykreslovat v přesných místech obrázku mapy při různých velikostech, bylo potřebné získat její měřítko v závislosti na velikosti reálné fotky a zobrazené v mobilním telefonu. Tato část byla nejsložitější, ale nakonec byla zprovozněna. Následné body v mapě se tvorily na základě originálního obrázku a souřadnic zobrazených nástrojem Malování a kurzorem myši. Tímto způsobem je možné transformovat originální body z obrázku do zmenšeného obrázku mapy zobrazené v mobilní aplikaci a v těchto bodech vykreslit požadované tečky, které mají na sobě například aplikovanou animaci měnící se velikost.

Data těchto bodových souřadnic jsou vytvořena pomocí **enum** hodnot, které mají v Kotlinu tu výhodu, že k nim mohou být přidělené funkce. Tímto způsobem byla vytvořena třída **enum class AnimalAppearance(){};**, která má jako své hodnoty názvy kontinentů, popř. jejich části a abstraktní funkci **getCoords()**, kterou každá hodnota (kontinent) implementací přepisuje a vrací tak své body na mapě v podobě **listu**. Každý list obsahuje 3 hodnoty, kde první dvě z nich udávají polohu v originálním obrázku **X** a **Y** a třetí hodnota udává měřítko zobrazení tohoto bodu v podobě tečky. Ukázkou zápisu této třídy můžeme vidět níže (Obr. 18).

```
enum class AnimalAppearance(){
    EUROPE {
        override fun getCoords(): List<List<Float>> {
            return listOf(
                listOf(1350f, 340f, 2.5f)
            )
        }
    },
    EUROPE_SOUTH {
        override fun getCoords(): List<List<Float>> {
            return listOf(
                listOf(1150f, 470f, 0.7f),
                listOf(1330f, 450f, 0.7f)
            )
        }
    },
}
```

Obrázek 18. Enum třída obsahující data souřadnic mapy.

6.5.2 Taxonomie zvířat

Dalším rozšířením detailu zvířat bylo přidání jejich 6 základních informací taxonomické kategorie – říše, kmen, třída, řád, čeleď, rod. Tyto informace museli být vhodným způsobem doplněny k jednotlivým zvířatům opět v podobě **Values** uvnitř xml souboru, viz Obrázek 17.

6.5.3 Detail zvířete v zoologické zahradě

Tato část byla vytvořena téměř v samotném závěru aplikace a kombinuje několik zajímavých technik. Jedná se o obrazovku, ve které si může uživatel zobrazit informace o konkrétním druhu zvířete z téměř kterékoliv zoologické zahrady v České republice, pokud těmito informacemi disponuje.

Zajímavější částí této obrazovky však je, že pokud je uživatel připojený k internetu a má zapnuté polohovací služby GPS, aplikace mu sama automaticky vybere nejbližší zoologickou zahradu a zobrazí data právě této zahrady. Ke zjištění polohy je samozřejmostí opět přítomný dotaz uživateli o její povolení.

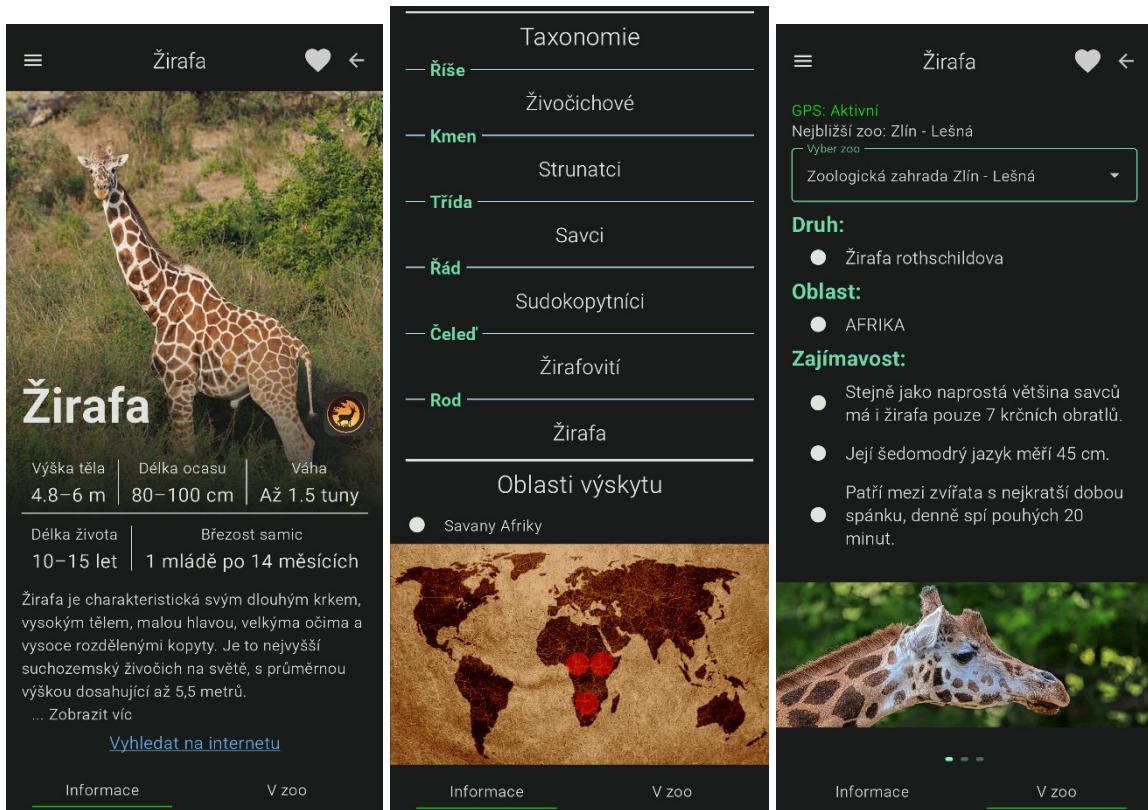
6.5.4 Ukázka

Na následujícím obrázku (Obr. 19) můžeme vidět obě části detailu zvířete, které v aplikaci jsou. Téměř veškeré komponenty jsou při vstupu do detailu pro příjemnější vzhled animovány a stejně tak je animován i přechod mezi záložkami „Informace“ a „V zoo“.

Název zvířete má aplikovanou vertikální animaci, kdy text sjede shora dolů a řádky „tabulky“ mají animaci horizontální k sobě opačným směrem – horní řada textu přechází zleva doprava a spodní řádek zprava doleva. Dále si můžeme povšimnout ikonky na pravé straně od názvu zvířete, tato ikonka udává, že je aplikace schopná toto zvíře pomocí kamery detektovat.

Informační text je v základu limitován na zobrazení 6 řádků, na které když uživatel klikne (kdekoliv v textu), tak se stránka „natáhne“ a zobrazí veškerý dostupný text. Toto zobrazení je také animováno plynulým přechodem a lehkým poskočením.

Položky Taxonomie mají opět horizontální animaci vysunutí zleva doprava a jak již bylo řečeno, tak i tečky na mapě mění svou velikost.



Obrázek 19. Detail žirafy v aplikaci.

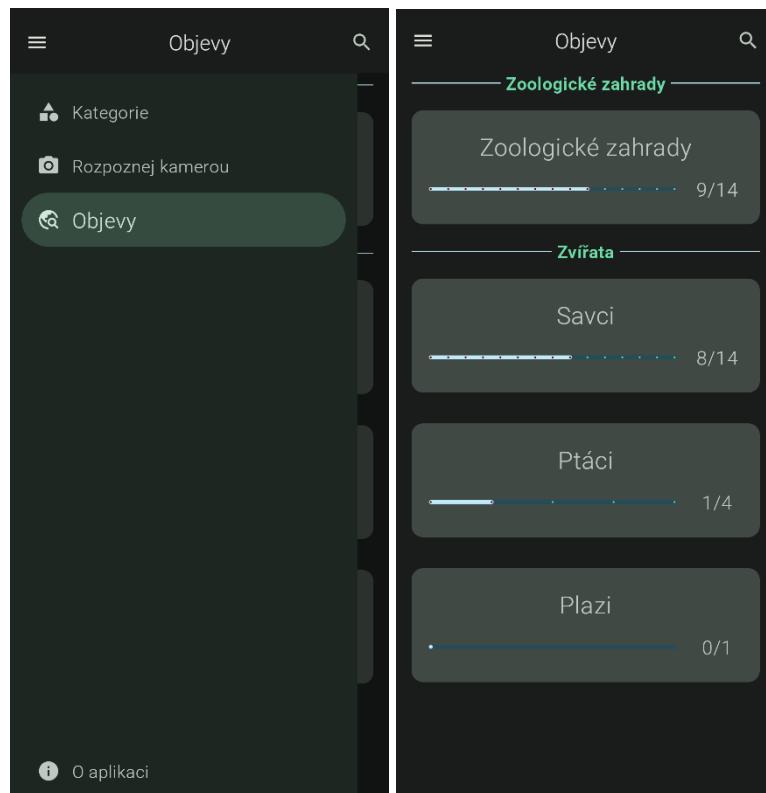
Když se přesuneme na záložku „V zoo“, dostaneme výsledek první dostupné zoologické zahrady v seznamu, která dané zvíře chová. Pokud máme připojení GPS, automaticky se zobrazí nejbližší zoologická zahrada, ale jen v případě, že dané zvíře chová. Můžeme vidět, že na stránce máme informace ohledně druhu chovaných zvířat daného rodu, informace zoologické zahrady o jejím chovu a popřípadě zajímavost. Veškeré texty jsou pořízeny ze stránek každé ze zoologických zahrad. Ve spodní části si můžeme zobrazit až 3 obrázky (pokud jsou na jejich stránkách dostupné) zvířete z dané zahrady. Po kliknutí na obrázek se jeho podoba zvětší na celou stránku, opětovným kliknutím se zmenší do základní podoby.

6.6 Objevy

Objevy zvířat i zoologických zahrad byly v aplikaci plánovány již od samotného začátku. V aktuální verzi je dostupné pouze jejich uložení, popř. zobrazení detailu zoologické zahrady, ale v budoucnu je možnost tuto funkci rozšířit například o odznaky nebo ukládání informací spolu s fotografiemi uživatele, kde a kdy dané zvíře či zahradu navštívil.

Jak tedy z textu vychází, objevy jsou rozděleny na zoologické zahradы a zvířata, která jsou navíc rozřazena na kategorie (Savci, Ptáci, Plazi). Každá z těchto položek má svůj vlastní počet objektů (zahradы a zvířata), které uživatel může „objevit“. Aktuální stav spolu

s celkovým počtem objevených objektů lze vidět u každé položky zvlášť. K těmto informacím se dostaneme skrze vyjížděcí navigační menu a jeho položky „Objevy“.



Obrázek 20. Detail navigačního menu a obrazovky Objevy.

6.6.1 Objevy zoologických zahrad

V detailu u objevů zoologických zahrad nalezneme seznam obsahující 14 zoologických zahrad České republiky a filtr pro jejich zobrazení – vše, objevené, neobjevené. Každá položka seznamu obsahuje logo zahrady, její typ, název / město a status, zda byla objevena. K objevení můžeme využít **swipe gesta**, díky kterém můžeme zahradu uložit do objevů (zleva doprava) nebo ji z objevů odstranit (zprava doleva). Další možnost uložení **bylo** navigování do detailu položky pomocí prokliku a zde ve spodní části skrze vyskakovací **Snackbar** položku uložit. Tento Snackbar byl zobrazen pouze tehdy, když nebyla položka ještě objevena.



Obrázek 21. Vyskakovací Snackbar na spodku aplikace.

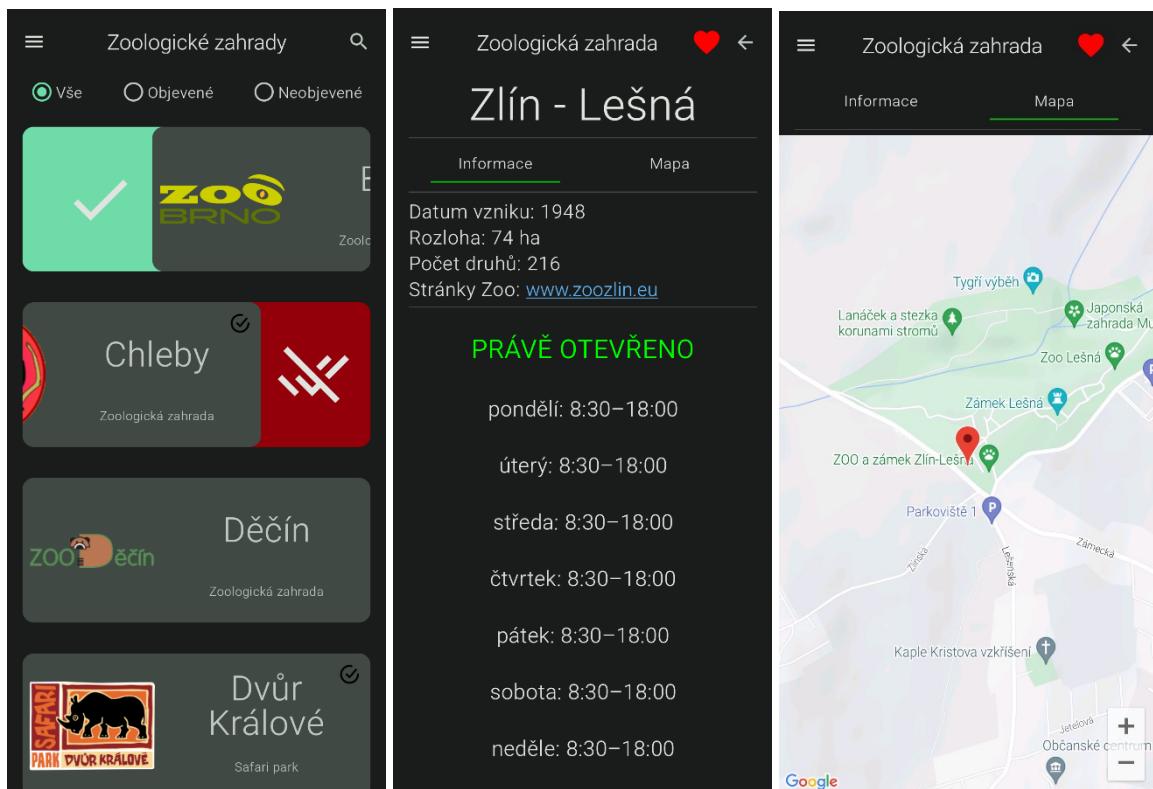
Tento Snackbar byl ale později **nahrazen ikonou srdička** ve vrchní části detailu položek (zoologická zahrada a zvířata), jak můžeme spolu se seznamem zoologických zahrad s vyobrazenými swipe gesty vidět na obrázku níže (Obr. 22).

6.6.2 Detail zoologické zahrady

Při prokliku do zoologické zahrady se nám načte záložka „Informace“ s velkým nápisem obsahující název zahrady a 4 základní informace: datum vzniku, rozloha, počet druhů zvířat a odkaz na webové stránky. Pokud má uživatel přístup k internetu, aplikace mu dále načte pomocí **Google Places API** informace o otevíracích hodinách na celý týden a dále poskytne informaci, zda je zahrada aktuálně otevřena. Pro tuto potřebu byl naprogramován ViewModel, který se o načtení informací postará a zároveň si jejich stav uchovává. Je nutné podotknout, že ukládání těchto informací není na základě podmínek použití povoleno (<https://developers.google.com/maps/documentation/places/web-service/policies>).

Ve druhé záložce „Mapa“ se téměř přes celou obrazovku vykreslí Google mapa s nastavenou polohou dané zoo. Mapa je vykreslena opět pouze s dostupným internetovým připojením a o vykreslení se stará **Google Maps API**.

Pro získání API klíče k těmto dvěma API nástrojům je nutné vlastnit **fakturační účet** na stránce [developers.google](https://developers.google.com) propojený s kreditní kartou, jelikož může být jejich použití při mnoha dotazech zpoplatněno.



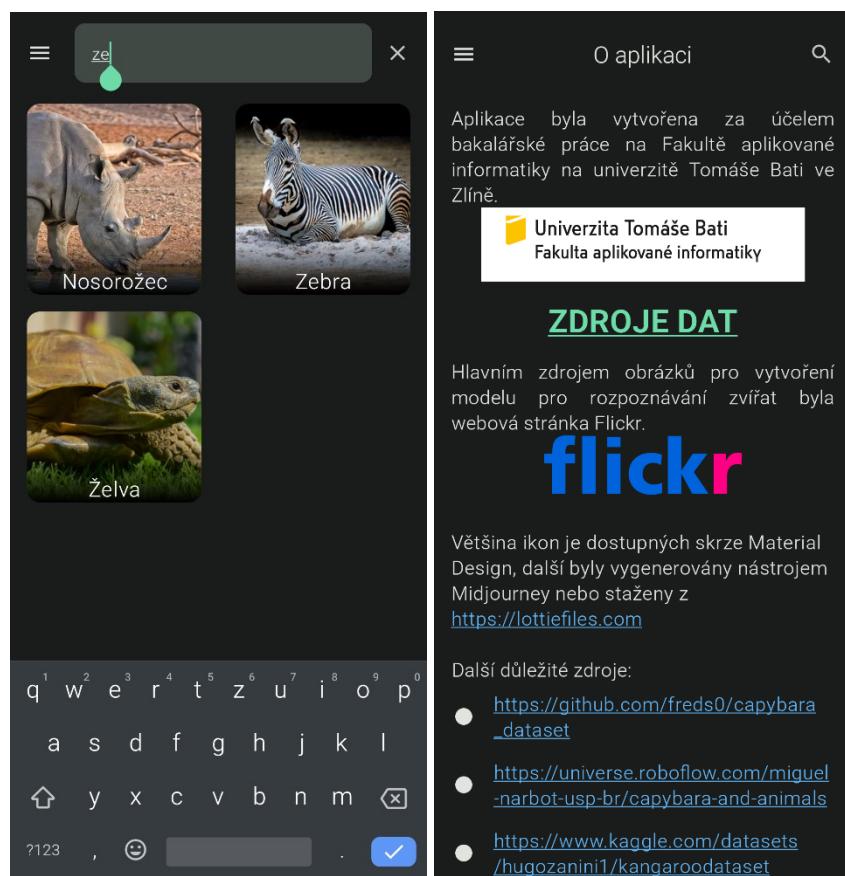
Obrázek 22. Pohled na obrazovku zoologických zahrad v Objevech a detail obrazovky obsahující informace a mapu konkrétní zoologické zahrady Brno.

Výhodou Google Places API je například její jazyková mutace, díky které může být její výstup na základě nastaveného jazyka v mobilním telefonu přeložen i do českého jazyka, a tak kooperovat s jazykovými mutacemi celé aplikace podobně jako **resources values** (5.4.1).

6.7 Další funkce

Mezi další funkci aplikace můžeme zařadit například vyhledávání zvířat podle jejich jména. Ikonku lupy můžeme naleznout téměř na každé obrazovce v pravém horním rohu. Po jejím stisku se vysune klávesnice a textové pole, kde uživatel může zadat text. Pokud je počet zadaných znaků ≥ 2 , aplikace začne vyhledávat zvířata nehledě na diakritiku, mezery a pozici znaků v názvu zvířat. Výstup vyhledávání je stejný jako zobrazení konkrétní kategorie zvířat, viz obrázek níže (Obr. 23).

Asi poslední existující, mimo uvítací, obrazovkou aplikace je obrazovka „O aplikaci“. Ta obsahuje veškeré informace o vzniku této aplikace s popisem, odkud autor čerpal data a za jakým účelem byla aplikace vůbec vytvořena.



6.8 Manifest

Manifest je srdcem mobilní aplikace. V našem případě deklaruje požadavky pro povolení:

- Internet
- GPS služby
- Kamera

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.CAMERA" />

<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />

<uses-feature android:name="android.hardware.camera" />
<uses-feature android:name="android.hardware.camera.autofocus" />
```

Obrázek 24. Manifest s požadavky na povolení.

Dále manifest obsahuje nezbytné informace jako například téma **splash** obrazovky, ikonku a název aplikace, nastavené hlavní aktivity nebo **Google API** klíč převzatý ze souboru *local.properties*.

```
<application
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher_foreground"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_foreground"
    android:supportsRtl="true"
    android:theme="@style/Theme.Splash"
    tools:targetApi="31">
    <activity
        android:name=".MainActivity"
        android:exported="true"
        android:theme="@style/Theme.Splash">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <meta-data android:name="com.google.android.geo.API_KEY"
        android:value="${MAPS_API_KEY}" />
</application>
```

Obrázek 25. Manifest se zbytkem informací o aplikaci.

7 TVORBA DATASETU

Dataset pro CNN se dá chápat jako obrovská sada obrázkových dat, na kterých se trénuje neuronová síť. Tyto obrázky se musí setřídit do společných tříd (složek), které spolu souvisí a nevznikala tak pouze nepojmenovaná hromada dat, pro kterou by nebylo využití. Z tohoto jasně vyplývá, že se jedná o poměrně těžkou manuální práci, která vyžaduje píli, přesnost a preciznost, aby nevznikaly chyby, které by zapříčinily nesprávnému trénování sítě.

K natrénování aktuálních modelů byly vytvořeny prakticky 2 datasety.

- První z nich je zaměřený na již zmíněný druh strojového učení s využitím CNN pro **Image Classification**. Tento typ datasetu vyžaduje roztríděné obrázky na třídy, které se v nich vyskytují. Tyto třídy jsou určeny pojmenovanou složkou, do nichž se dané obrázky umístí a tím vznikne následující hierarchie:
 - Slon
 - obrázek1.jpg, obrázek2.jpg, ...
 - Zebra
 - obrázek1.jpg, obrázek2.jpg, ...
 - ...
- Druhý typ datasetu je pro CNN určenou na **Object Detection**. Tento druh datasetu je typický tím, že ke každému obrázku se musí vyskytovat speciální soubor, který obsahuje souřadnice hledaných objektů z daného obrázku. Díky tomu je celý proces tvorby tohoto datasetu mnohem obtížnější a časově náročnější než u prvního.

Dataset tvoří jádro celé práce, jelikož bez něj by se nedala natrénovat neuronová síť, která by byla schopná zvířata rozpoznat. Pro jeho přípravu bylo využito několik technik a nástrojů, které pomohli velkou část jeho tvorby urychlit automatizací.

7.1 Image Classification dataset

V této sekci se zaměříme především na vznik a vývoj datasetu pro typ CNN zaměřené na klasifikaci obrazu. Jak již víme z textu výše (1.1.1), jedná se o klasifikaci obrázku, který síť klasifikuje jako jeho celek do třídy, kterou je naučena a schopna identifikovat.

Tento druh datasetu vyžaduje poměrně velký počet trénovacích dat, které musí být roztríděné na třídy v přibližně rovnoměrném rozdělení, k čemuž dopomohou různé nástroje.

7.1.1 Použité techniky

Využitých nástrojů a technik při tvorbě datasetu pro image classification bylo několik. Důležitým prostředníkem hráli naprogramované skripty v jazyce Python a aplikace *Flickr* a jeho API, se kterou se seznámíme níže. Díky tomu bylo alespoň částečně možné automatizovat složitý a časově vyčerpávající proces, který by manuálně snad nešel ani dokončit.

7.1.1.1 *Flickr a jeho API*

Hlavním zdrojem obrázků pro tuto práci byla webová aplikace <https://www.flickr.com/>, která obsahuje miliony obrázků označeny tagy, které dopomáhají jejich přesnému vyhledání a možnosti stažení. Ruční stahování tisíce obrázků je poměrně nemyslitelná záležitost, na kterou v aplikaci mysleli a pro vývojáře vytvořili šikovnou API. K jejímu získání se stačí na stránce zaregistrovat a nechat si zdarma vygenerovat její klíč, který je potřebný ve scriptu sloužící právě pro stahování obrázků.

Script byl napsaný v programovacím jazyce Python 3.7.9 a stará se o stahování obrázků zvířat, které si programátor zvolí. V první verzi script pouze stahoval obrázky a nic jiného neřešil, což při následné ruční filtraci dělalo potíže, jelikož bylo zjištěno, že některé obrázky jsou staženy například 8krát, protože je tam lidé několikrát nahráli. Proto byl script upraven tak, aby po stažení obrázku zkontoval jeho binární hash s ostatními staženými obrázky a pokud by se již daný hash někde vyskytoval, právě stažený obrázek vymaže. Navíc se díky tomu redukoval celkový počet originálních obrázků, které bylo možné stáhnout a hrozilo v zacyklení „stažení – vymazání“ a proto byla přidána další kontrola pro opakování vymazání staženého obrázku, kdy se po 60 cyklech stahování přeruší a popřípadě se přejde na další druh zvířete.

Stažené obrázky se museli ještě ručně vyfiltrovat, jelikož se v sadě vyskytovali takové, které nemají s daným zvířetem nic společného (například automobil značky Jaguar), nebo se v obrázku nacházelo v minimálním měřítku. Po vyfiltrováním bylo u každého druhu zvířete kolem 2000–3000 obrázků, což bylo pro klasifikaci tohoto rozměru poměrně málo.

7.1.1.2 *Image augmentation*

Pro vyřešení problému s nízkým počtem obrázků existuje technika nazvaná augmentace dat. Díky ní je možné z malé sady obrázků vytvořit sadu větší, a to s využitím transformací originálních obrázků.

Využité transformace:

- Zrcadlení obrázku horizontálně.
- Zrcadlení obrázku vertikálně.
- Posun obrázku nahoru nebo dolů.
- Otočení obrázku v rozmezí -30° až 30° .
- Změna měřítka obrázku v rozmezí 75 % až 130 % originální velikosti.

S použitím této techniky bylo vygenerováno přibližně 6000 obrázků do každé kategorie pro zajištění kolem 8000 až 9000 obrázků každého zvířete, což už je slušný počet pro trénování CNN zaměřené na klasifikaci.

7.1.2 Finální zhodnocení a informace

Dataset se neustále měnil a přetvářel, přibývaly jiné druhy zvířat, popřípadě byly odstraněny ty, které neobsahovaly správná data pro dostatečně kvalitní trénování modelu. Tato fáze tvoření probíhala zcela paralelně se samotnou tvorbou modelu a jeho učením, kdy bylo potřeba doladovat různé detaily datasetu pro zajištění co největší přesnosti naučeného modelu.

Finální podoba datasetu obsahuje 20 druhů zvířat s celkovým počtem 78 208 obrázků přesahující 11,4 GB dat. Tento počet dat byl při trénování přibližně dvojnásobný, jelikož byla na obrázky aplikovaná augmentace (7.1.1.2). Jak se dozvímme níže (8.2), ne všechny druhy zvířat byly v konečném trénování zahrnuty.

Tabulka 3. Přehled dat v klasifikačním datasetu.

Jméno zvířete		Počet obrázků	Velikost dat	Aplikovaná augmentace
Česky	Anglicky			
Mravenečník	Anteater	2 909	487 MB	3x
Velbloud	Camel	3 284	483 MB	3x
Kapybara	Capybara	3 315	493 MB	3x
Kočka	Cat	4 098	471 MB	2x
Pes	Dog	4 214	518 MB	2x
Kachna	Duck	4 287	591 MB	2x
Slon	Elephant	3 221	587 MB	3x
Plameňák	Flamigno	4 292	568 MB	2x
Žirafa	Giraffe	4 269	651 MB	2x
Gorila	Gorilla	4 102	671 MB	2x
Jaguár	Jaguar	3 932	684 MB	2x
Klokan	Kangaroo	4 136	711 MB	2x
Lev	Lion	4 245	701 MB	2x

Papoušek	Parrot	4 236	503 MB	2x
Tučňák	Penguin	4 189	556 MB	2x
Nosorožec	Rhino	2 652	449 MB	3x
Lachtan	Sealion	4 266	598 MB	2x
Tygr	Tiger	4 146	721 MB	2x
Želva	Turtle	4 179	648 MB	2x
Zebra	Zebra	4 236	771 MB	2x

7.2 Object Detection dataset

Tato sekce obsahuje detailní popis tvorby datasetu využívající CNN s technikou object detection. Vytvoření tohoto typu datasetu vyžaduje mnohem více úsilí a času než u předchozího typu. Je to díky tomu, že aby se CNN mohla lépe učit nalézt objekt v obraze, musíme jí tento objekt v trénovacích datech vyznačit, což již značí spoustu ruční práce, kterou nelze jednoduše automatizovat.

Celý dataset je rozdělený na 2 samostatné části.

- **Train** obsahuje 88 % z celkového počtu dat a slouží pro trénování sítě.
- **Test** obsahuje zbylých 12 % dat a je využita pouze pro validaci během trénování. Na těchto datech se síť netrénuje.

7.2.1 Použité nástroje

I při tvorbě tohoto druhu datasetu bylo využito několik nástrojů, převážně v podobě Python skriptů, které práci mnohonásobně urychlili nebo pomohli opravit chyby v desítkách tisíců souborů. Veškeré vytvořené anotace jsou ve speciálním formátu zvaném VOC Pascal, který je potřebný knihovnou TensorFlow použitou pro trénování. Data tohoto formátu mají ucelenou strukturu, jsou uložena v souborech s koncovkou XML a jejich formát můžeme vidět na obrázku níže (Obr. 26). Z těchto dat (obrázek a k němu přidružený XML soubor obsahující anotaci) se v konečné fázi vytváří soubor **tfrecord**, který je vstupním prvkem trénování sítě.

```

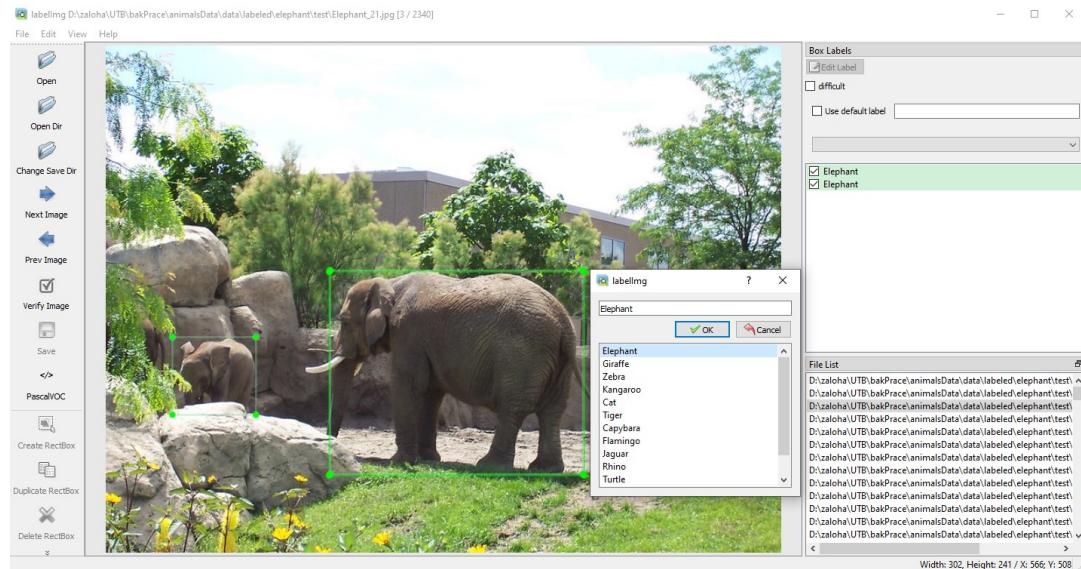
<?xml version="1.0" encoding="UTF-8"?>
- <annotation>
  <filename>Elephant_10.jpg</filename>
  - <size>
    <width>800</width>
    <height>532</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  - <object>
    <name>Elephant</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    - <bndbox>
      <xmin>161</xmin>
      <ymin>4</ymin>
      <xmax>800</xmax>
      <ymax>532</ymax>
    </bndbox>
  </object>
</annotation>

```

Obrázek 26. Anotace ve formátu VOC Pascal XML.

7.2.1.1 Program LabelImg

Prvním důležitým nástrojem je program LabelImg. Jedná se o grafický nástroj pro anotaci a označování objektů v obrázku. Je napsán v programovacím jazyce Python a pro všechny zcela zdarma dostupný pod instalačním příkazem *pip install labelImg*.



Obrázek 27. Ukázka prostředí programu LabelImg.

V tomto programu bylo ručně vytvořeno pár tisíc anotací obrázků několika málo druhů zvířat, jejichž obrázky byly převzaty z již dříve staženého datasetu pro image classification.

Tato část tvorby si vyžádala nejvíce času, a i přes to nebylo možné bez získání většího počtu dat pokračovat, a proto přišlo na řadu stahování volně dostupných datasetů se zvířaty.

7.2.1.2 Volně dostupné datasety

Pro zajištění dostatečně velkého počtu obrázků bylo zapotřebí najít a stáhnout volně dostupná řešení jiných vývojářů. Mezi známé portály obsahující datasety se řadí především stránka <https://www.kaggle.com/> a <https://roboflow.com/>. Na nich je možné zdarma stáhnout vytvořené datasety od jiných uživatelů a použít je tak ve vlastním zájmu.

V této práci to nebylo výjimkou a několik datasetů bylo staženo, viz Příloha P I. Některé stažené datasety nebyly využity vůbec kvůli nízkému počtu obrázků.

Dohromady bylo staženo kolem 55_000 obrázků (7,40 GB dat), které bylo potřeba vyfiltrovat, jelikož spoustu z nich bylo vadných, a nakonec je nebylo možné použít. Takové obrázky byly většinou příliš zmenšené, obsahovaly chybné nebo dokonce žádné označení zvířat anebo byly záběry pořízené z fotopastí, ve kterých bylo zvíře zachycené ze špatných úhlů.

7.2.1.3 Skript pro kontrolu vadných obrázků

V této práci byl využitý Python skript, který zkontroloval všechny obrázky v datasetu, pro zajištění jejich správného formátu JPEG. Tento skript byl použit poté, co se během trénování objevila chyba označující, že dataset obsahuje poškozený obrázek. Bohužel ale tato chyba neobsahovala konkrétní obrázek a vyhledat jej ručně bylo v sadě desítek tisíců obrázků prakticky nemožné. Skript je výtvorem Yasooba Khalida a lze jej najít například na tomto odkaze: <https://stackoverflow.com/a/65367773>.

Skript byl lehce upraven, aby místo odstranění vadného obrázku vypsal jeho jméno a následně byl obrázek s jeho příslušným XML souborem smazán ručně. Díky tomuto skriptu bylo nalezeno celkově asi 12 poškozených obrázků, které během trénování modelu zapříčňovali jeho pád.

7.2.1.4 Skript pro přejmenování souborů

Během úprav datasetu bylo vždy několik obrázků staženo nebo odstraněno, což značilo ne-konzistentnost v názvů souborů. Také názvy souborů ze stažených datasetů obsahovaly pouze směsici písmen a čísel, na které nebyla radost pohledět. Z tohoto důvodu byl vytvořený skript, který přejmenoval všechny soubory podle představ autora, a to ve tvaru pro vlastní obrázky: *zvire_cislo.jpg* a ve tvaru pro stažené obrázky: *zvire_download_cislo.jpg*.

Tento skript dále musí brát v potaz taky XML soubory příslušných obrázků, jejichž jména musí být až na koncovku totožná.

7.2.1.5 Skript pro editaci XML souborů

Tento skript byl vytvořen za účelem aktualizování tagu *filename* po aplikování skriptu se změnou názvů souborů. Postupem času byl aktualizován na vyhledávání chybných anotací ze stažených datasetů, jelikož mnohé z nich obsahovali anotované objekty o velikosti menší než 33 pixelů na šířku nebo výšku, což je pro TensorFlow hraniční hodnota minimální velikosti objektu v obrázku. Pokud se počet objektů v XML souboru rovnal 0, byl tento soubor spolu s příslušným obrázkem přesunut do složky, která obsahuje všechny data ve vadném formátu pro TensorFlow.

Dále se ve stažených datasetech objevoval problém v tom, že byl objekt anotován o 1 pixel až za hranice samotného obrázku, tudíž byl skript doplněný o opravu i těchto hranic.

Skript taky odstraňuje nepotřebné tagy *path*, *folder*, *source*, *occuled*, *polygon* a tag *pose* nastavuje na *Unspecified*.

Pokud soubor obsahuje anotovaný objekt, který uživatel nepotřebuje, lze jej tímto skriptem odstranit.

7.2.1.6 Skript pro odstranění malých obrázků

Mezi staženými datasety se často objevovali obrázky, jejichž velikost byla příliš malá. Tento skript se stará o přesunutí takových obrázků do již zmíněné složky, která obsahuje vadná data. Prahová hodnota obrázků je nastavena na 331 pixelů do šířky i výšky a všechny menší obrázky byly přesunuty.

7.2.1.7 Skript pro tvorbu „background“ obrázků

Díky tomuto skriptu bylo vygenerováno 8000 XML souborů k obrázkům, které **neobsahují** živé, jehož třídu by měla neuronová síť umět rozpoznat. Díky tomu se může síť přiučit nová data a redukovat tak poměrně časté **false-positive** výskytu. Výsledný XML soubor tak ob-sahuje pouze základní informace o obrázku **bez** tagu *object*.

7.2.1.8 Skript pro dokončení datasetu

Tento skript je rozdělen na 2 samostatné funkce.

První funkce s názvem *split* se stará o rozdelení obrázků a příslušných anotací jednotlivých zvířat do jejich složek *train* a *test*. Počet obrázků v testovací sadě je 12 % z celkového počtu obrázků a na těchto obrázcích se aplikuje validace během trénování.

Druhá funkce *moveToTestTrainFolders* je použita pro překopírování všech dat ze složek *train* a *test* jednotlivých zvířat do jedné společné složky *train* a *test*. Tyto složky jsou finální částí dokončením datasetu, jelikož obsahují všechna trénovací a validační data spolu na „jednom“ místě a už z nich zbývá jen vytvořit soubory *tfrecords*.

7.2.1.9 Skript pro generování TFRecord souboru

Jedná se o originální skript od TensorFlow dostupný pod tímto odkazem: <https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/training.html#create-tensorflow-records>, který byl lehce upraven pro možnost doplnění background obrázků do datasetu s pomocí <https://github.com/tensorflow/models/issues/3365#issuecomment-661326737>.

Z předešlých složek *train* a *test* jsou vygenerovány TFRecord soubory obsahující binární informace o jednotlivé sadě dat. Díky těmto souborům je možné efektivněji manipulovat s velkým množstvím dat, jelikož jsou tímto způsobem uložena pohromadě a tvoří tak jeden velký list, nad kterým lze rychle operovat.

7.2.2 Finální zhodnocení a informace

Podobně jako u klasifikačního datasetu se i tento dataset budoval paralelně během vývoje celé aplikace a trénování jejího modelu. Jednalo se o časově náročnou práci a veškeré jeho úpravy by nebylo ani možné bez dodatečných skriptů provést. Důvodem vytvoření tohoto datasetu a následně i modelu CNN byla naděje pro získání lepších výsledků během rozpoznávání zvířat.

Stejně jako u předchozího datasetu má i tento nějakou finální podobu. Obsahuje celkem 19 druhů zvířat, kdy bohužel nebylo možné všechny použít. Celkem se jedná o 53_302 obrázků zabírajících 6,19 GB dat na disku (spolu se soubory obsahující anotace). Zároveň můžeme připočítat 1,18 GB dat již zmíněného „background“ datasetu s 8_001 obrázky.

Tabulka 4. Přehled dat v detekčním datasetu.

Jméno zvířete		Počet obrázků	Celková velikost dat
Česky	Anglicky		
Buvol	Buffalo	286	174 MB
Kapybara	Capybara	728	109 MB
Kočka	Cat	11 094	1,06 GB
Kráva	Cow	4 016	134 MB
Jelen	Deer	3 447	186 MB
Pes	Dog	11 455	1,16 GB
Slon	Elephant	2 340	244 MB
Plameňák	Flamigno	100	12,80 MB
Žirafa	Giraffe	1 086	281 MB
Jaguár	Jaguar	1 128	160 MB
Klokan	Kangaroo	1 498	265 MB
Lev	Lion	1 416	70,50 MB
Papoušek	Parrot	2 056	77,40 MB
Tučnák	Penguin	2 856	144 MB
Nosorožec	Rhino	1 370	178 MB
Ovce	Sheep	2 966	213 MB
Tygr	Tiger	2 199	1,28 GB
Želva	Turtle	949	98,30 MB
Zebra	Zebra	2 312	385 MB

Jakmile byl dataset vytvořen, autor se rozhodl jej nahrát na známý server kaggle.com, který obsahuje spoustu datasetů jiných autorů. Tento dataset je volně dostupný na následujícím odkazu: <https://www.kaggle.com/datasets/jirkadaberger/zoo-animals>.

8 TRÉNOVÁNÍ MODELU

Tato část popisuje celkový vývoj modelu, který je v aplikaci využitý pro rozpoznávání zvířat. Model konvoluční neuronové sítě hraje nejdůležitější roli této aplikace. Abychom mohli sítě naučit, bylo potřebné nasbírat a vytrídit velký počet obrázků, na kterých se síť učí. Popis této části je uveden výše v kapitole Tvorba datasetu (7).

Trénování navíc od datasetu vyžaduje i vysoký hardwarový výkon a spoustu času. První vytvořené modely byly trénovány na osobním notebooku **MSI GE62 VR 7RF Apache Pro** s 16 GB pamětí RAM, grafickou kartou NVIDIA GTX1060 s 3 GB pamětí a CPU i7-7700HQ. Rychlosť trénování modelů obvykle závisí na dostupném HW, složitosti modelu a počtu trénovacích dat. Každé provedené trénování (viz níže) na osobním notebooku trvalo přibližně od 10 h po 16 h. Později byla možnost použití grafických karet virtuální organizace **MetaCentrum** (8.4), které díky jejich výkonu dovolili plné trénování modelů v relativně krátkém čase.

8.1 API pro vytvoření vlastních modelů

Jelikož je téměř nemožné pro jednotlivce s omezenými zdroji vytvořit kompletní model neuronové sítě schopné detekovat komplexní objekty, kterými jsou právě zvířata, bylo pro trénování využito několika API.

Další obrovskou limitací během trénování byl požadavek, aby mohl výsledný model fungovat na mobilních zařízeních. Takový model vyžaduje formát tflite (2.2.1.1) se speciálně upravenými parametry, které bohužel nejsou dostupné kdekoliv a jejich export (vytvoření) z „klasického“ modelu není jednoduchý.

Během vývoje bylo využito několik odlišných API, kdy ale ne všechny podporovaly právě export modelu do **tflite** formátu obsahujícím **metadata**, které udávají informace o vstupu a výstupu modelu pro jednodušší použití v Android Studiu.

8.1.1 TensorFlow lite model maker

Jedná se o knihovnu, která sjednocuje jednoduché trénování modelu spolu s **transfer learning**⁴, které dopomůže v rychlosti trénování. S použitím této knihovny je možné natrénovat model optimalizovaný na mobilní zařízení v pár řádcích kódu bez nutnosti velkých vědomostí.

Abychom toto API mohli použít, stačí jej nainstalovat konzolovým příkazem **pip install -q tflite-model-maker** (berme v potaz předem nainstalovaný a nastavený python). Tento příkaz stáhne a nainstaluje všechny potřebné balíčky a připraví knihovnu k jejímu použití. [41]

K následnému použití stačí vytvořit python skript a na jeho začátek importovat potřebné balíčky:

```
import glob
from struct import unpack

import tensorflow as tf
from tqdm import tqdm

assert tf.__version__.startswith('2')

from tflite_model_maker import image_classifier
from tflite_model_maker.image_classifier import DataLoader
```

Obrázek 28. Import knihoven pro TFlite model maker.

Dále nám již jenom stačí načíst dataset, vybrat verzi modelu a spustit jeho trénování funkcí *image_classifier.create()*. V závěru můžeme model exportovat do formátu tflite, který již zároveň bude obsahovat metadata.

```
tf.config.experimental.set_memory_growth(tf.config.list_physical_devices('GPU')[0], True)

images_path = "animalsData\\images"
data = DataLoader.from_folder(images_path)
trainData, restData = data.split(0.8)
validationData, testData = restData.split(0.5)

modelSpec = 'efficientnet_lite3'
model = image_classifier.create(
    train_data=trainData,
    validation_data=validationData,
    model_spec=modelSpec,
    epochs=5
)

loss, accuracy = model.evaluate(testData)
model.export(export_dir='.'
```

Obrázek 29. Příklad kódu pro trénování klasifikačního modelu.

⁴ Jedná se o techniku strojového učení, kdy se naučené vlastnosti nějakého problému aplikují na jinou/novou úlohu.

Z výše uvedených obrázků (Obr. 28, Obr. 29) je patrné, že s použitím této knihovny je vytvoření modelu opravdu jednoduchou záležitostí.

Knihovna dále podporuje i trénování modelu pro detekci objektů. Jelikož se ale jedná o složitější a výkonově náročnější proces, nebylo možné trénování spustit na osobním notebooku ani po několika pokusech – trénování vždy spadlo na nedostatek grafické paměti. Později proto byla knihovna nainstalována na stroje **MetaCentra**, kde ji ale bohužel nebylo možné spustit v důsledku chybějících systémových ovladačů, které knihovna potřebuje a autor je nemohl bez patřičných práv nainstalovat.

8.1.2 TensorFlow 2 Object Detection API

Jelikož selhal pokus trénování modelu pro detekci objektů pomocí [TensorFlow lite model maker](#), bylo potřebné najít nějaké jiné řešení. Jedním z těchto řešení bylo použití „bratrské“ API knihovny zaměřené přímo na detekci objektů: **TensorFlow 2 Object Detection API**. Díky tomuto API je možné trénování modelu „od nuly“ nebo skrze **transfer learning** použít již existující řešení, které nám ušetří spoustu času.

8.1.2.1 Instalace

Instalace této knihovny je poněkud složitější, a ne vždy všechno fungovalo i když byl dodržen doporučený postup. Problém byl většinou v novějších verzích, kdy některé jejich části nespolupracují s knihovnami jiných stran, které jsou ale taky potřebné. Pro zajištění funkčnosti tohoto API i po instalaci jiných knihoven bylo vytvořeno čisté python prostředí pomocí nástroje **Anaconda**. Tako vytvořené prostředí se izoluje od hlavního systémového prostředí a tím vznikne jedno unikátní a konzistentní prostředí pro různé potřeby, v tomto případě se jednalo o nainstalování daného Object Detection API.

Jak lze v doporučeném návodu vidět (<https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/install.html>), pro správné nainstalování a přichystání veškerých funkcionalit knihovny je potřeba dodržet několik přesných kroků.

8.1.2.2 Příprava TF Object Detection API

Trénování pomocí tohoto API většinou funguje na principu **transfer learning**, kdy si uživatelé mohou vybrat z několika před trénovaných modelů sítí a na nich aplikovat svá data pro jejich konkrétní úlohu. Veškeré modely jsou dostupné na následující adrese:

https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md. Výběr je opravdu veliký, ale jak již bylo zmíněno, tato práce je limitovaná na použití modelu, který je možné použít na mobilním zařízení. Z [23] vyplývá, že jako jediné možné řešení je použití některého z modelů architektury SSD.

Během prvních testů byl stažen model SSD MobileNet V2 FPNLite 320x320. Před spuštěním trénování bylo vždy nutné upravit konfigurační soubor **pipeline.config**, který je dostupný se staženým modelem. Bylo nutné změnit počet kategorií, nastavit počet trénovacích kroků a cesty k vytvořeným **tfrecords** souborů, které obsahují veškerá data datasetu. Z návodu byla vytvořena následující hierarchie pro snadnou a rychlou orientaci mezi složkami a soubory:

```
tensorflow
├── models
├── protoc
└── scripts
    ├── metadata_creator.py
    └── generate_tfrecords.py
└── workspace
    └── train_1
        └── annotations
            ├── label_map.pbtxt
            ├── test.record
            └── train.record
        └── models
            └── my_ssd_lite_320
                ├── classes.txt
                └── pipeline.config
        └── pre-trained-models
            └── ssd_lite_320
                ├── checkpoint
                │   ├── ckpt-0.data-00000-of-00001
                │   └── ckpt-0.index
                └── saved_model
                    └── pipeline.config
        └── export_tflite_graph_tf2.py
        └── exporter_main_v2.py
        └── model_main_tf2.py
    └── notes.txt
```

Poslední 3 ***.py** soubory jsou skripty samotného API, které můžeme naleznout v tensorflow/models/research/object_detection.

8.1.2.3 Trénování s TF Object Detection API

Pokud máme vše připravené, stačí už jenom použít jeden příkaz a trénování modelu začne:

```
python model_main_tf2.py --model_dir=models/my_ssd_lite_320 --  
pipeline_config_path=models/my_ssd_lite_320/pipeline.config --num_workers=2
```

Během trénování se do složky *my_ssd_lite_320* ukládají **checkpoint** soubory uchovávající poslední stav sítě během trénování. Z těchto souborů jde možné buď pokračovat v trénování, nebo z nich vytvořit finální podobu modelu a jeho případný export do jiných formátů.

8.1.2.4 Vytvoření tflite modelu

Z posledního checkpointu můžeme kdykoliv vytvořit finální model sítě, jež potom můžeme využít k němu přidružené aplikaci. Abychom mohli model využít v naší mobilní aplikaci, potřebujeme model v podobě tflite formátu obsahující metadata naučené sítě. K tomu můžeme využít již nachystané skripty, opět s použitím příkazového řádku:

```
python export_tflite_graph_tf2.py --pipeline_config_path=models/my_ssd_lite_320/pipe-  
line.config --trained_checkpoint_dir=models/my_ssd_lite_320 --output_directory=mo-  
dels/my_ssd_lite_320/tfliteexport  
  
tflite_convert --saved_model_dir=models/my_ssd_lite_320/tfliteexport/saved_model --out-  
put_file=models/my_ssd_lite_320/tfliteexport/saved_model/ssd_320.tflite --  
input_shape=1,320,320,3 --input_arrays=normalized_input_image_tensor --out-  
put_arrays='TFLite_Detection_PostProcess','TFLite_Detection_PostPro-  
cess:1','TFLite_Detection_PostProcess:2','TFLite_Detection_PostProcess:3' --infe-  
rence_type=UINT8 --allow_custom_ops
```

Poslední částí je zápis metadat do vytvořeného tflite souboru. K tomu byl použitý skript dostupný na adrese: https://www.tensorflow.org/lite/models/convert/metadata_writer_tutorial#object_detectors. Vstupem do tohoto skriptu je cesta k existujícímu tflite souboru, textovému souboru obsahující všechny třídy ve správném pořadí a cesta s názvem vygenerovaného tflite souboru s metadaty. Příkaz může vypadat následovně:

```
python metadata_creator.py -t ../workspace/train_1/models/my_ssd_lite_320/tfliteex-  
port/saved_model/ssd_320.tflite -l ../workspace/train_1/models/my_ssd_lite_320/clas-  
ses.txt -o ../workspace/animals/models/my_ssd_lite_320/tfliteexport/saved_mo-  
del/ssd_320_metadata.tflite
```

Tímto způsobem můžeme v pár jednoduchých krocích získat model, který lze importovat do mobilní aplikace a konečně ho použít.

8.2 Model pro klasifikaci

Vůbec prvním natrénovaným modelem byl model pro klasifikaci obrazu (1.1.1), jelikož se jednalo o jednodušší variantu datasetu (7) a jak bylo zjištěno, tak i trénování.

Trénování proběhlo na osobním notebooku autora s pomocí [TensorFlow lite model maker knihovny](#). Celkem byly natrénovány asi 3 modely, které se mezi sebou lišily pouze počtem obrázků a kategorií.

Finální klasifikační model obsahuje celkem 18 kategorií zvířat: mravenečník, kočka, pes, kachna, slon, plameňák, žirafa, gorila, jaguár, klokan, lev, papoušek, tučňák, nosorožec, tuleň, tygr, želva a zebra. Z tabulky (Tab. 3) tedy vyplývá, že kapybara a velbloud byli z finální podoby klasifikačního modelu vynecháni. Velbloud byl vyřazen z důvodu nevhodných obrázků a kapybara chybí z důvodu pozdního vytvoření jejího datasetu.

Na tomto modelu probíhalo první reálné testování aplikace v zoologické zahradě Zlín – Lešná a s překvapením autora fungoval poměrně dobře – bylo ovšem nutné používat zoom, aby bylo zvíře na obrazu dostatečně veliké. Díky výsledku testování, které můžeme vidět níže (9.1), byl tento model zařazen do vytvořené mobilní aplikace ve funkci rozpoznávání zvířat z fotografie.

Od té doby byl model ponechán tak jak je a autor se začal věnovat modelu pro detekci, který mohl přinést přesnější výsledky.

8.3 Model pro detekci

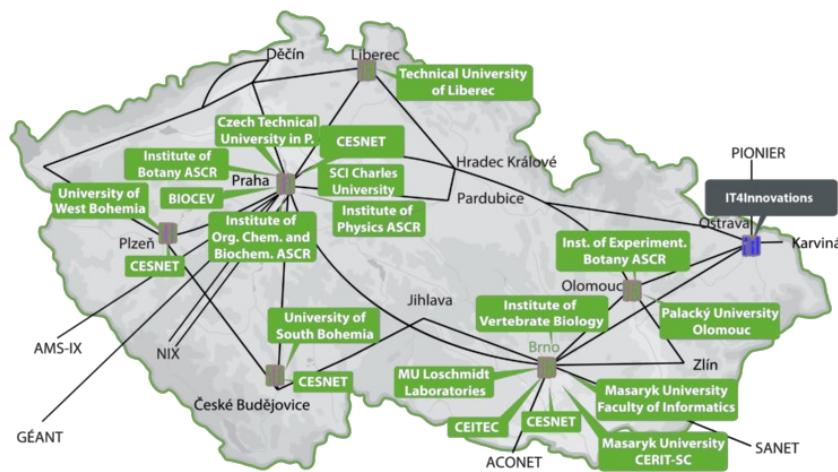
Z výše uvedených příčin byl model pro detekci objektů (zvířat) natrénovaný pomocí TensorFlow Object Detection API. Během vývoje byl natrénovaný nespočet modelů, jejichž charakteristiky se měnili v důsledku změn datasetu a taky nastavení pipeline.config souboru. Navíc trénování nejdřív probíhalo na detektoru SSD MobileNet V2 FPNLite 320x320, který byl později nahrazen za jeho druhou variantu s vyšším rozlišením obrazu **640x640**. Tento detektor má o 6 % (28.2 %) lepší průměrnou přesnost na velkém datasetu COCO.

Z již zmíněného datasetu pro object detection (7.2.2) bylo k vytvoření finálního modelu využito 29_017 obrázků pro trénování a 3_933 pro validaci, celkem tedy 32_950 obrázků. Z tohoto počtu obrázků bylo díky anotování získáno **celkem 46_739 objektů zvířat**.

Mezi naučené zvířata finálního modelu se dostalo těchto 13 tříd: slon, žirafa, zebra, klokan, tygr, jaguár, nosorožec, želva, kočka, pes, papoušek, tučnák a kapybara. Zbytek zvířat se bohužel buď kvůli malému počtu obrázků nebo jejich kvalitě do finálního modelu nedostalo.

8.4 MetaCentrum

MetaCentrum (**MetaVO**) je virtuální organizace České republiky zabývající se poskytováním vysoko výkonného hardwaru ve formě Gridových výpočtů⁵, studentům a akademickým pracovníkům ve členském sdružení CESNET zdarma. [42]



Obrázek 30. Mapa propojených míst spadajících do MetaCentrum. [42]

Pro finální trénování použitého modelu ve vytvořené aplikaci byla použita právě tato služba, která výrazně dopomohla snížení času trénování, a hlavně vytíženosti osobního počítače autora této práce.

8.4.1 Seznámení a prvotní nastavení

Samotná registrace probíhá klasicky jako na každé jiné stránce, ale s využitím školního emailu studenta, který je spravován federací **eduID.cz** spadající pod již zmíněné sdružení CESNET.

K plnému využití grafických karet při trénování modelu jsou zapotřebí knihovny CUDA, které jsou dostupné na vybraných grafických kartách společnosti NVIDIA. Abychom je ovšem mohli využít, je potřeba zaregistrovat se jako developer na stránkách NVIDIA

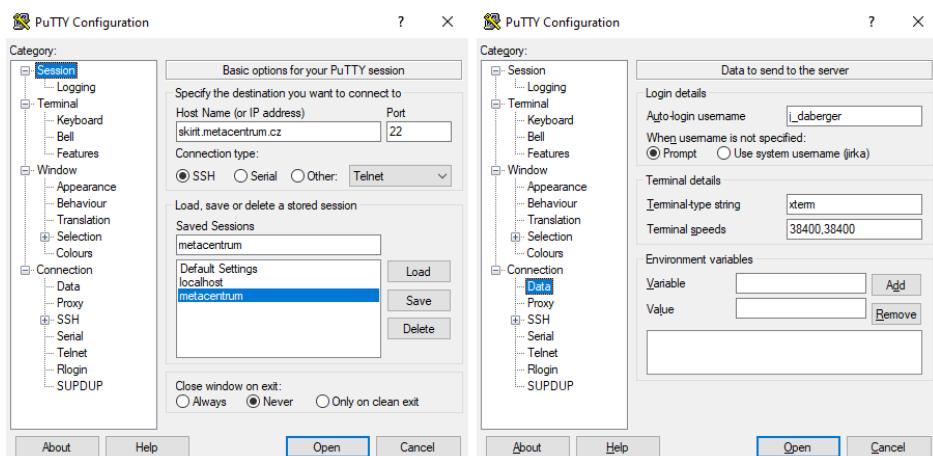
⁵ Gridové výpočty znamenají propojení několika PC z různých lokací pro dosažení co největšího výpočetního výkonu.

(<https://developer.nvidia.com/>) a souhlasit s jejich licencí. V MetaVO jsou pro nás tyto a mnohé další licence připraveny a stačí s nimi souhlasit.

8.4.1.1 Konfigurace nástroje PuTTy

Po registraci a přihlášení již můžete dostupné výpočetní zdroje využívat pro své potřeby. Pro přístup na samotná zařízení se dá na počítačích s operačním systémem Windows použít volně dostupná aplikace **PuTTy**, která přes okno terminálu obstará připojení na vzdálenou síť MetaVO, která využívá operační systém Linux.

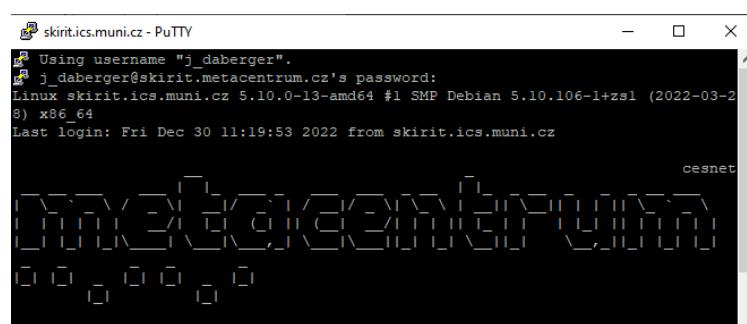
Pro pohodlné a rychlé přihlášení lze uložit nastavení přihlašovací konfigurace, abychom po každé nemuseli dělat vše manuálně, když se chceme na síť MetaVO přihlásit.



Obrázek 31. Nastavení přihlášení v PuTTy.

Na obrázku (Obr. 31) lze na levé straně v sekci „Session“ vidět, že celá konfigurace je uložená pod názvem „metacentrum“. Adresa, na kterou se uživatel připojí je „skirit.metacentrum.cz“ s portem 22 a typ připojení je zabezpečený kanál SSH. V sekci „Connection/Data“ lze taky nastavit login uživatele, který se chce přihlásit.

Díky tomuto nastavení se uživatel může kdykoliv rychle přihlásit bez nutnosti zadávat vše znova. Nakonec jen zadá své heslo a přihlášení do sítě je hotové.



Obrázek 32. Úspěšné přihlášení do sítě programem PuTTy.

Toto terminálové okno je určeno jako hlavní a taky jediná možnost pro práci s výpočetními stroji. Uživatel je zprvu přihlášen do tzv. **čelního uzlu**, který slouží jako jakýsi rozcestník a k základní manipulaci v uživatelském úložišti, a proto je na něm zakázáno provádět výkonově náročné operace.

Takové operace se již provádějí na výpočetních strojích k tomu určených. Pro připojení na stojane je potřeba vytvoření úlohy z čelního uzlu.

- **Interaktivní** úloha slouží pro samotné přichystání prostředí, ve kterém bude uživatel dále pracovat. Touto úlohou se uživateli přidělí požadovaný stroj a uživatel může manuálně provádět operace potřebné k nachystání svého prostředí.
- **Dávková** úloha se stará o přímé zpracování konkrétního scriptu, který si uživatel vytvoří předem. Po přidělení požadovaného stroje se začne úloha provádět sama bez dalšího vstupu od uživatele.

Příklady použití obou úloh nalezneme níže v textu.

8.4.1.2 *Instalace potřebných knihoven*

První kroky v MetaVO byly poměrně chaotické, a i přes poměrně rozsáhlou dokumentaci ve formě wiki stránek (https://wiki.metacentrum.cz/wiki/Hlavn%C3%AD_strana) se nedařilo knihovny TensorFlow zprovoznit. Z těchto důvodů byla kontaktována uživatelská podpora, která velice ochotně poradila a pomohla zprovoznit základní instalaci.

Pro tuto instalaci již byla potřebná **interaktivní** úloha, která byla spuštěna následujícím příkazem:

```
qsub -l walltime=4:0:0 -q default@meta-pbs.metacentrum.cz -l select=1:ncpus=1:mem=8gb:scratch_local=10gb -I
```

Tímto příkazem se požaduje stroj s jedním CPU, 8 GB RAM a 10 GB lokálního úložiště, který bude uživateli dostupný po dobu 4 hodin.

```
MISTO_INSTALACE=/storage/brno2/home/j_daberg/obj_detect
cd $MISTO_INSTALACE
export TMPDIR=$SCRATCHDIR
curl -Ls https://micro.mamba.pm/api/micromamba/linux-64/latest | tar -xvj bin/micromamba
eval "$(. ./bin/micromamba shell hook -s bash)"

MAMBA_EXE=bin/micromamba micromamba create -p $(pwd)/tf2 -c conda-forge python==3.8 pycocotools libprotobuf==3.19.4 cudatoolkit=11.2 cudnn=8.1.0
MAMBA_EXE=bin/micromamba micromamba activate $(pwd)/tf2
git clone https://github.com/tensorflow/models TensorFlow/models
cd TensorFlow/models/research/
cp object_detection/packages/tf2/setup.py .
pip install .

protoc object_detection/protos/*.proto --python_out=$(pip show pycocotools|awk '/^Location:/ {print \$2; }')
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CONDA_PREFIX/lib
export XLA_FLAGS="--xla_gpu_cuda_data_dir=$MISTO_INSTALACE/tf2/lib"
export TF_XLA_FLAGS="--tf_xla_auto_jit=2"
clean_scratch
```

Výše uvedené linuxové příkazy mají za úkol vytvořit virtuální prostředí **micromamba** nazvané „tf2“ v domovském adresáři uživatele. Virtuální prostředí umožňuje instalaci různých balíků a knihoven, aniž by „udělalo nepořádek“ v globálním prostředí uživatele, což by mohlo mít za následek nekompatibilitu některých knihoven a tím jejich nefunkčnost. Do vytvořeného prostředí byly dále nainstalovány potřebné knihovny pro práci s frameworkem **TensorFlow 2 Object Detection API** i s podporou GPU pro rychlejší trénování. Z oficiálního GitHubu byl naklonován a nainstalován repozitář tohoto frameworku. Pro finální zprovoznění je zapotřebí vytvořit speciální **proměnné** prostředí, které obsahují cesty k důležitým souborům, které jsou frameworkem požadovány pro jeho fungování.

Toto, v konečném výsledku poměrně jednoduché, nastavování nakonec zabralo téměř 3 dny aktivního zkoumání, jelikož se objevovala celá řada problémů, z nichž největší byla právě proměnná s názvem **XLA_FLAGS**.

Jakmile bylo všechno nastavené a funkční, byla vytvořena **dávková** úloha starající se o aktivaci vytvořeného prostředí a v něm spuštění trénování na připravených datech. Tato část je popsána v kapitole výše (8.1.2.2).

9 TESTOVÁNÍ MODELU

Pro otestování exportovaného **tflite** modelu byl vytvořen skript, který jej prověří na testovacích datech. Tento skript byl převzatý z GitHubu (<https://github.com/ChrystleMyrna-Lobo/tflite-object-detection/blob/master/inference.py>) a následně upraven pro potřeby autora. Tímto způsobem byly porovnány 4 poslední vytvořené modely, které se mezi sebou lišili zejména způsobem trénování a lehkými změnami datasetu. Dále byl do testování zařazen v pořadí 13. model, jež byl posledním modelem neobsahující background obrázky v datasetu.

Tabulka 5. Porovnání natrénovaných modelů.

Číslo modelu	Přesnost klasifikace	Přesnost průniku detekce	Doba měření
13	16,10 %	15,72 %	32 minut
16	76,56 %	71,75 %	42 minut
18	84,56 %	73,90 %	43 minut
19	85,89 %	75,07 %	42 minut
20	86,07 %	76,20 %	43 minut

Z tabulky (Tab. 5) můžeme vyčíst různé hodnoty měření každého modelu i přes použití stejného testovacího datasetu (pro modely č. 16–20). Samozřejmě je to díky tomu, jak byly jednotlivé modely natrénovány. Tyto modely oproti modelu č. 13 obsahují ve svém trénovacím datasetu background obrázky, díky čemu můžeme vidět extrémní rozdíl v přesnosti modelů. Skript ovšem nefiltruje přesnost každé detekce, tudíž se do celkového průměru počítají i velmi špatné detekce například pod 50 %. Stejně tak to je ale i u zbylých modelů, které dosahují mnohem lepší výsledek, což je zajímavé sledovat.

Modely číslo **16 a 18** jsou trénovány na shodném datasetu se shodně nastavenými parametry konfiguračního souboru. Jediný rozdíl je ten, že na **modelu 16** byl aplikován experiment **trénování od nuly**. To znamená, že byl učen pouze na autorem vytvořeném datasetu a nebylo tedy využito techniky *transfer learning*. Učení **modelu 18** naopak vycházelo s předučeného modelu, a tak můžeme vidět rozdíl přesně 8 % v přesnosti klasifikace na nových datech trénovacího datasetu.

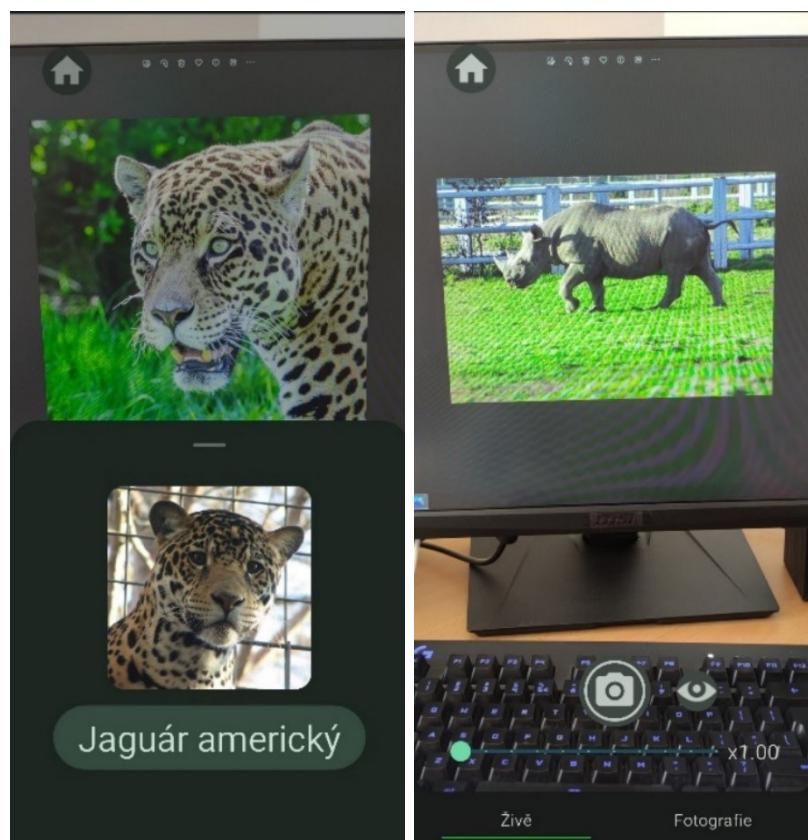
Model č. 19 je téměř shodný jako model 16/18. Jedinou změnou jsou nové obrázky datasetu kočky a psa pro trénování. Následné trénování probíhalo totožně jako model 18, tedy s využitím *transfer learning* metody. Díky tomu můžeme pozorovat přes 1 % zlepšení v přesnosti klasifikace i lokalizace.

Dalším **experimentem** byl model číslo **20**. Trénování probíhalo na totožném datasetu jako u modelu 19, ale změnou prošel konfigurační soubor trénování, což podle našeho skriptu přináší ještě o něco málo lepší výsledek.

9.1 Téměř reálné testování v aplikaci

Doposud probíhalo testování modelu na přímých fotografiích zvířat, bylo tedy nutné toto testování provést skrze vytvořenou mobilní aplikaci s využitím kamery mobilního telefonu.

Jelikož by testování s použitím stejného množství obrázků, jako u předchozího druhu testování, trvalo velmi dlouho, bylo toto testování provedeno pouze na **3 obrázcích** každé kategorie použitím **modelu č. 19 a 20 + model klasifikační**. Obrázky se mezi sebou liší v obtížnosti – vzdálenost zvířete, kvalita nebo zakryté části zvířete. Zároveň bylo využito i již zmíněných aplikací **Google Lens** a **Seek** pro srovnání přesnosti mezi jednotlivými aplikacemi.



Obrázek 33. Ukázka testování pomocí obrázků.

Pro spravedlivé otestování byl mobilní telefon uložen do vyvýšeného pouzdra na stole tak, aby byly zajištěny stejné podmínky (vzdálenost od monitoru, náklon) pro testování každé aplikace na všech obrázcích. Ukázka této formy testování je vidět na obrázku výše (Obr. 33) a zahrnuje správně klasifikovaný první obrázek jaguára a neúspěšné klasifikování prvního obrázku nosorožce při použití **modelu č. 20**.

Všech 57 použitých obrázků je uloženo v adresáři: prilohy/DabbergerJiri_bp_2023/datasets/final_tests/, který je dostupný jako příloha.

Tabulka 6. Porovnání klasifikování zvířat z obrázků mezi vytvořenými modely a aplikacemi Seek a Google Lens.

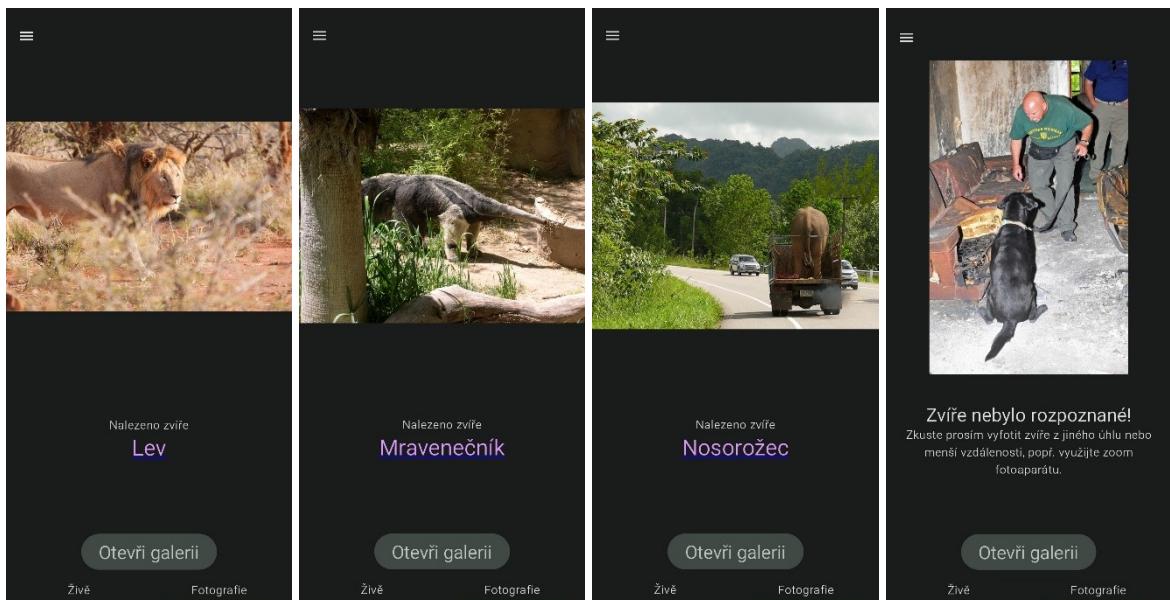
Zvíře	Model č. 19	Model č. 20	Klas. mo-del	Seek	Google Lens
Gorila	-	-	✗✗✗	✓✓✗	✓✓✓
Jaguár	✓✗✗	✓✗✗	✓✗✗	✓✓✗	✓✗✓
Kachna	-	-	✗✗✗	✓✓✓	✓✓✓
Kapybara	✗✓✗	✗✗✗	-	✗✗✗	✓✓✗
Klokan	✓✗✗	✓✗✗	✗✗✗	✓✗✗	✓✓✓
Kočka	✓✗✓	✓✗✓	✗✗✗	✗✗✗	✓✓✓
Lachtan	-	-	✗✗✗	✗✓✗	✓✗✗
Lev	-	-	✗✗✗	✗✗✗	✗✓✓
Mravenečník	-	-	✓✗✗	✓✗✗	✓✗✓
Nosorožec	✗✗✗	✗✗✗	✗✗✗	✗✗✗	✗✓✓
Papoušek	✗✗✓	✗✓✓	✗✗✓	✗✓✓	✓✓✓
Pes	✓✗✗	✓✓✗	✓✗✗	✓✗✗	✓✓✗
Plameňák	-	-	✗✗✗	✓✓✗	✓✓✓
Slon	✓✗✗	✗✗✗	✗✗✗	✗✗✗	✓✓✓
Tučňák	✗✓✗	✗✓✗	✗✗✗	✗✓✗	✗✓✗
Tygr	✓✗✓	✓✗✓	✗✗✗	✓✓✓	✓✓✓
Zebra	✓✓✓	✓✓✓	✗✗✗	✓✓✓	✓✓✓
Želva	✗✗✗	✗✓✓	✗✗✗	✗✓✓	✗✗✓
Žirafa	✓✓✗	✓✓✗	✗✗✗	✓✓✗	✓✓✗
Úspěšnost	16/39	18/39	4/54	25/57	44/57

V tabulce výše (Tab. 6) můžeme vidět řadu ikonek, které udávají výsledek klasifikace každé aplikace, popř. modelu, na jednotlivých **třech** obrázcích daného druhu zvířete. Symbol **zelené fajfky** značí úspěšnou klasifikaci, a naopak **červený křízek** značí neúspěšnou klasifikaci.

Z těchto údajů vyplývá poměrně velké selhání v **klasifikačním modelu** (pouze 4 správě klasifikované obrázky z počtu 54). Naopak aplikace **Google Lens** byla s pomocí (výběr objektu v obrázku) schopná rozpozнат a klasifikovat téměř 100 % obrázků, bez pomoci jich bylo 44 z 57.

Model číslo 20. má v celkovém počtu úspěšných detekcí o 2 více než model 19. Aplikace **Seek** byla po většinu času schopná určit, zda se například jedná o savce nebo ptáka, přesně klasifikovat zvíře byla ale schopná ve 25 případech. Pokud bychom ale brali pouze shodné kategorie naučených zvířat jako na modelu 20, vítězem by se oproti Seek stal model č. 20, který má o 2 správně klasifikované obrázky více.

Musíme však podotknout, že pohled na obrázky v kamere mobilního telefonu byl poměrně vzdálený, a tak byla plocha reprezentována zvířetem v malém měřítku, což detekci ztěžovalo a tím více i klasifikaci pouze s použitím klasifikačního modelu. Obrázky byly proto nahrány do mobilního telefonu, kde ve vytvořené aplikaci proběhl test **klasifikačního** modelu pomocí funkce klasifikování fotografie, kde byl model schopný **správně** určit **51 obrázků** (2 nedokázal klasifikovat a třetí obrázek slona klasifikoval jako nosorožce).

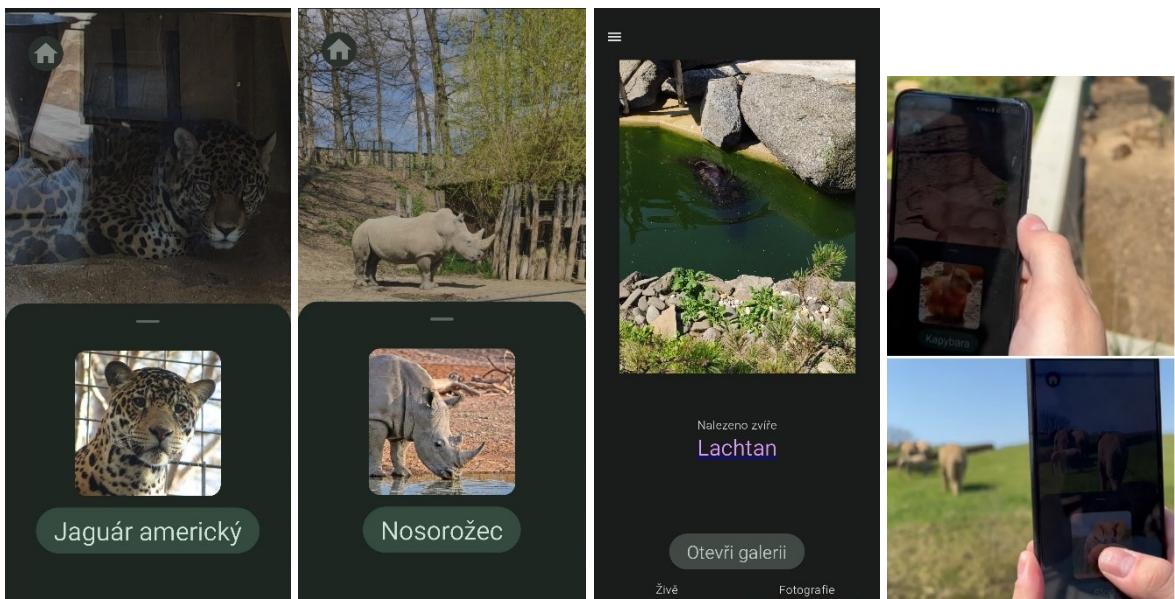


Obrázek 34. Testování klasifikačního modelu skrze aplikaci.

9.2 Závěrečné testování v zoologické zahradě Zlín Lešná

Poslední zařazené testování proběhlo v reálných podmírkách zoologické zahrady Zlín. U většiny velkých savců (slon, nosorožec, žirafa, tygr, zebra) neměla aplikace problém a bezproblémově je rozpoznala i s nízkým přiblížením kamery, problémy ale nastaly při

pokusu o rozpoznávání malých zvířat, a ještě k tomu z velké vzdálenosti nebo špatných pozorovacích úhlů či pozorovacích možností (skleněné vitríny, voda, síť, vzdálenost). Největší problém snad dělaly odrazy ve sklech, s kterými si aplikace neporadila a zvířata za ním nebyla téměř schopná rozpoznat. Podobný problém spojený se vzdáleností byl u sítí, které vymezují prostor ptáků – papoušků. Tyto problémy ovšem vyřešil druhý model ve funkci klasifikace vyfoceného obrázku, s jehož použitím bylo i tyto zvířata možné rozpoznat. V otevřených prostorách byla aplikace pomocí funkce přiblížení kamery schopná rozpoznat i menší zvířata – kapybary, želvy nebo např. malé klokany.



Obrázek 35. Příklady klasifikovaných zvířat v zoo Zlín.

Ve výše uvedených obrázcích (Obr. 35) můžeme vidět některé z úspěšně klasifikovaných zvířat v zoologické zahradě Zlín. Další příklady lze shlédnout ve videu, které je opět v příloze této práce: prilohy/DabbergerJiri_bp_2023/text/data/app_testing_zlin.mp4.

ZÁVĚR

Cílem této práce bylo vytvořit kompletní mobilní aplikaci na operační systém Android, která by byla schopná rozpoznat vybrané druhy zoologických zvířat. Pro tento účel bylo potřeba vytvořit a natrénovat model konvoluční neuronové sítě. K natrénování takového modelu bylo navíc potřebné vytvořit kvalitní dataset obsahující tisíce obrázků s danými zvířaty. Aby aplikace nebyla chudá na funkcionalitách, bylo v aplikaci za cíl vytvořit i seznam těchto zvířat spolu s jejich detailním náhledem a umožnění uživateli uložení těchto zvířat, a navíc i zoologických zahrad, do tzv. objevů.

Nejprve se bylo nutné seznámit s fungováním neuronových sítí, jejich stavbou a potřebami z hlediska dat, na kterých se musí učit. Během vývoje bylo vytvořeno několik odlišných verzí jak datasetu, tak i natrénovaných modelů pro rozpoznávání vybraných zvířat. Tato část si vyžádala nejvíce času i úsilí, jelikož bez dostatečně kvalitního datasetu nemohl vzniknout funkční model, který by přesně rozpoznával zvířata. Během občas i několika-denního trénování modelu probíhala práce na samotné mobilní aplikaci. Ta byla naprogramovaná jazykem Kotlin doplněným o framework Jetpack Compose k tvorbě uživatelského prostředí. Po návrhu několika funkcionalit byla mezi prvními vytvořenými právě možnost zobrazení náhledu kamery v aplikaci. Následovalo vytvoření řešení starající se o ukládání dat zvířat a zahrad. Poté přišly na řadu funkce spojené se zobrazením těchto dat uživateli na konkrétních obrazovkách, na které se může dostat skrze navigační systém aplikace.

Během přibližně 10 měsíců vznikla mobilní aplikace schopná rozpoznat 13 druhů zoologických zvířat pomocí obrazu mobilní kamery. Dalších 6 druhů zvířat, celkově tedy 19, je aplikace schopna rozpoznat pomocí odlišně naučeného modelu na vybrané fotografií z galerie mobilního zařízení. Aplikace také dokáže pozastavit detekci zvířat, vyfotit snímek nebo zoomovat kamerou. Mezi její další funkce patří vyhledání a zobrazení detailních informací zvířete, vyhledání nejbližší zoologické zahrady pomocí GPS nebo uložení navštívení zoologických zahrad a zvířat do statistik nazvaných v aplikaci jako Objevy. Ve finále byla aplikace řádně otestována a porovnaná s dvěma populárními aplikacemi Seek a Google Lens.

Aplikaci je možné využít na jakémkoliv Android zařízení ve verzi 7.0 a vyšší. Může sloužit jako učební pomůcka dětem nebo jako interaktivní průvodce zoologickou zahradou. Díky vyhledávání zvířat může sloužit i jako rychlá encyklopédie nebo přehledný nástroj sloužící k zobrazení [v budoucnu] všech zoologických zahrad minimálně v České republice.

V rámci diplomové práce může být v plánu další rozšíření aplikace. Aplikaci by prospělo přidání nových zvířat i zoologických zahrad. Zasloužila by si hezčí uživatelské prostředí a volbu nastavení, ve kterém by si uživatel mohl nastavit například domovskou obrazovku, jazyk aplikace nebo funkce mobilní kamery. Mezi nové funkcionality by **bylo možné** zařadit rozpoznávání zvířat i podle zvuku, možnost zvíře zobrazit pomocí **rozšířené reality**, nebo možnost ukládání více informací o objevu či nastavit jejich automatické objevení při samotném rozpoznání. V neposlední řadě spolupráce přímo se Zoologickou zahradou Lešná.

SEZNAM POUŽITÉ LITERATURY

- [1] WILEY, Victor a Thomas LUCAS. *Počítačové vidění* [online]. 2018 [cit. 2023-02-10].
<https://doi.org/10.29099/ijair.v2i1.42>.
- [2] SHARMA, Pulkit. Image Classification vs. Object Detection vs. Image Segmentation. *Medium* [online]. 2019 [cit. 2023-03-15]. Dostupné z: <https://medium.com/analytics-vidhya/image-classification-vs-object-detection-vs-image-segmentation-f36db85fe81>
- [3] 3 kategorie počítačového vidění. In: *ML Fundamentals* [online]. [cit. 2023-02-10]. Dostupné z: https://ataspinar.com/wp-content/uploads/2017/11/deeplearning_types.png
- [4] Co je OCR. In: *Nanonets* [online]. [cit. 2023-01-12]. Dostupné z: <https://nanonets.com/blog/what-is-optical-character-recognition>
- [5] COUMAR, Nanda. OCR. In: *Medium* [online]. [cit. 2023-01-12]. Dostupné z: <https://medium.com/@nandacoumar/optical-character-recognition-ocr-image-opencv-pytesseract-and-easyocr-62603ca4357>
- [6] *Pokročilé techniky neuronových sítí*. [online]. In: . [cit. 2022-12-20]. Dostupné z: <https://course.elementsofai.com/cs/5/3>
- [7] YAMASHITA, R., M. NISHIO a R.K.G. DO. Konvoluční neuronové sítě. In: *Convolutional neural networks: an overview and application in radiology* [online]. 2018, s. 611-629 [cit. 2022-12-20]. Dostupné z: doi:<https://doi.org/10.1007/s13244-018-0639-9>
- [8] MU, Li, Zachary LIPTON, Zhang ASTON a Alexander SMOLA. Dopředná a zpětná propagace. In: *Dive into Deep Learning* [online]. 2021, s. 224-227 [cit. 2023-01-13]. Dostupné z: doi:<https://arxiv.org/abs/2106.11342v3>
- [9] JOHNSON, Daniel. Informace o TensorFlow knihovně. In: *GURU99* [online]. 2022 [cit. 2023-01-15]. Dostupné z: <https://www.guru99.com/what-is-tensorflow.html>
- [10] TensorFlow graph. In: *TensorFlow* [online]. [cit. 2023-01-15]. Dostupné z: https://www.tensorflow.org/guide/intro_to_graphs
- [11] TensorFlow Lite. In: *TensorFlow* [online]. [cit. 2023-01-23]. Dostupné z: <https://www.tensorflow.org/lite/guide>

- [12] DERNONCOURT, Franc. Počet trénovacích dat. In: *StackExchange* [online]. 2016 [cit. 2023-01-17]. Dostupné z: <https://stats.stackexchange.com/questions/226672/how-few-training-examples-is-too-few-when-training-a-neural-network/226693#226693>
- [13] Přeúčení sítí. In: *IBM* [online]. [cit. 2023-01-17]. Dostupné z: <https://www.ibm.com/topics/overfitting>
- [14] RUIZENDAAL, Rutger. Problémy s učením sítě. *Towards Data Science* [online]. [cit. 2023-01-17]. Dostupné z: <https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d>
- [15] PULLURU, Likhitha. Vektor v YOLO. In: *Code Basics* [online]. [cit. 2023-01-27]. Dostupné z: <https://codebasics.io/blog/understanding-of-yolo>
- [16] CARRANZA-GARCÍA, Manuel, Jesús TORRES-MATEO, Pedro LARA-BENÍTEZ a Jorge GARCÍA-GUTIÉRREZ. Detektory. In: *On the Performance of One-Stage and Two-Stage Object Detectors in Autonomous Vehicles Using Camera Data* [online]. 2020, 3., 23 s. [cit. 2023-01-23]. Dostupné z: doi:<https://doi.org/10.3390/rs13010089>
- [17] Two/One-stage detektory. *Guide to Object Detection using Deep Learning: Faster R-CNN, YOLO, SSD* [online]. [cit. 2023-01-23]. Dostupné z: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e>
- [18] KUNDU, Rohit. Historie YOLO. In: *V7labs* [online]. 2023 [cit. 2023-04-27]. Dostupné z: <https://www.v7labs.com/blog/yolo-object-detection>
- [19] SHARMA, Pulkit. YOLO. In: *Analytics Vidhya* [online]. 2021 [cit. 2023-01-27]. Dostupné z: <https://www.analyticsvidhya.com/blog/2018/12/practical-guide-object-detection-yolo-framework-python/>
- [20] SINGH CHOUDHARY, ANURAG. *Mobilenet SSD* [online]. In: . 2022 [cit. 2023-03-08]. Dostupné z: <https://www.analyticsvidhya.com/blog/2022/09/object-detection-using-yolo-and-mobilenet-ssd/>
- [21] ALSAAIDI, Elham a Nidhal ABBADI. SSD-MobileNet. In: *An Automated Mammals Detection Based on SSD-Mobile Net* [online]. 2021 [cit. 2023-02-03]. Dostupné z: doi:<https://dx.doi.org/10.1088/1742-6596/1879/2/022086>

- [22] HOLLEMANS, Matthijs. *SSD MobileNet V2* [online]. 2018 [cit. 2023-02-04]. Dostupné z: <https://machinethink.net/blog/mobilenet-v2/>
- [23] Running TF2 Detection API Models on mobile. In: *GitHub* [online]. 2021 [cit. 2023-03-20]. Dostupné z: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/running_on_mobile_tf2.md
- [24] Zastoupení mobilních operačních systémů. In: *Statcounter* [online]. [cit. 2022-12-18]. Dostupné z: <https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-202111-202211-bar>
- [25] Vývoj mobilních aplikací. *Rascasone* [online]. 2021 [cit. 2023-03-15]. Dostupné z: <https://www.rascasone.com/cs/blog/typy-mobilnich-aplikaci>
- [26] VALA, Radek. *Programování mobilních aplikací: Metody vývoje mobilních aplikací* [Prezentace]. Fakulta aplikované informatiky - Univerzita Tomáše Bati ve Zlíně, 2022.
- [27] Třívrstvá architektura. In: *Managementmania* [online]. 2015 [cit. 2023-04-27]. Dostupné z: <https://managementmania.com/cs/trivrstva-architektura-three-tier-architecture>
- [28] POP, Dragos-Paul a Adam ALTAR. Vzor MVC. In: *Designing an MVC Model for Rapid Web Application Development* [online]. 2014, s. 1172-1179 [cit. 2022-12-18]. ISSN 1877-7058. Dostupné z: [doi:https://doi.org/10.1016/j.proeng.2014.03.106](https://doi.org/10.1016/j.proeng.2014.03.106)
- [29] MVC diagram. In: *TechTerms* [online]. [cit. 2022-12-18]. Dostupné z: <https://techterms.com/definition/mvc>
- [30] MISHRA, Rishu. Difference Between MVC and MVP Architecture Pattern in Android. In: *GeeksForGeeks* [online]. 2020 [cit. 2022-12-18]. Dostupné z: <https://www.geeksforgeeks.org/difference-between-mvc-and-mvp-architecture-pattern-in-android/?ref=gcs>
- [31] MVVM diagram. In: *Bach Khoa-npower* [online]. [cit. 2022-12-19]. Dostupné z: <http://bachkhoa-npower.vn/mvvm-la-gi/>
- [32] MISHRA, Rishu. MVVM (Model View ViewModel) Architecture Pattern in Android. In: *GeeksForGeeks* [online]. 2022 [cit. 2022-12-19]. Dostupné z:

- <https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/>
- [33] Android. In: *Wikipedia* [online]. [cit. 2022-12-10]. Dostupné z: [https://cs.wikipedia.org/wiki/Android_\(opera%C4%8Dn%C3%AD_syst%C3%A9m\)](https://cs.wikipedia.org/wiki/Android_(opera%C4%8Dn%C3%AD_syst%C3%A9m))
- [34] Android API levels. In: *Android API Levels* [online]. [cit. 2023-02-04]. Dostupné z: <https://apilevels.com/>
- [35] Android Studio. In: *Android Developers* [online]. [cit. 2023-02-09]. Dostupné z: <https://developer.android.com/studio/intro>
- [36] Informace o Jetpack Compose. In: *Android Developers* [online]. [cit. 2023-02-11]. Dostupné z: <https://developer.android.com/guide>
- [37] Rekompozice v Jetpack Compose. In: *Android Developers* [online]. [cit. 2023-02-11]. Dostupné z: <https://developer.android.com/jetpack/compose/state#introducing-state>
- [38] *Google Lens* [online]. [cit. 2023-02-17]. Dostupné z: <https://lens.google/howlensworks/>
- [39] Seek by iNaturalist [online]. [cit. 2023-02-17]. Dostupné z: https://www.inaturalist.org/posts/search?utf8=%E2%9C%93&q=vision+model&post%5Bparent_type%5D=Site&post%5Bparent_id%5D=1&commit=Search
- [40] Logo iNaturalist. In: *iNaturalist* [online]. [cit. 2023-02-17]. Dostupné z: <https://www.inaturalist.org/>
- [41] Image classification with TensorFlow Lite Model Maker. In: *TensorFlow* [online]. 2022 [cit. 2023-03-20]. Dostupné z: https://www.tensorflow.org/lite/models/modify/model_maker/image_classification
- [42] MetaCentrum [online]. [cit. 2022-12-30]. Dostupné z: <https://metavo.metacentrum.cz/cs/>
- [43] UIJLINGS, J., K. VAN DE SANDE a T. GEVERS. Selektivní vyhledávání. In: *Selective Search for Object Recognition* [online]. 2013 [cit. 2023-01-23]. Dostupné z: doi:<https://doi.org/10.1007/s11263-013-0620-5>

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

- ADB Android Debug Bridge
- API Application Programming Interface
- GB GigaByte
- GPS Global Positioning System
- HTML Hypertext Markup Language
- HW hardware
- IDE Integrated Development Environment
- iOS iPhone Operating System
- MB MegaByte
- NFC Near Field Communication
- OCR Optical Character Recognition
- OS Operating System
- Popř. popřípadě
- .py Typ souboru značící Python skript
- RAM Random Access Memory
- RGB Red Green Blue barevný formát
- SDK Software Development Kit
- SMS Short Message Service
- Tzv. tak zvaný
- UI User Interface

SEZNAM OBRÁZKŮ

Obrázek 1. Základní kategorie počítačového vidění. [3].....	13
Obrázek 2. Ukázka fungování OCR. [5].....	13
Obrázek 3. Aplikace filtru na vstupní obrázek v konvoluční vrstvě. [7].....	15
Obrázek 4. Ukázka použití filtrů v Pooling vrstvě.	16
Obrázek 5. Příklad augmentovaného obrázku.	19
Obrázek 6. Obrázek znázorňující použití NMS techniky. [15]	21
Obrázek 7. Obrázek s mřížkou 4×4 obsahující objekty se středovými body a vektory predikce. [15].....	23
Obrázek 8. Spojení MobileNet a SSD. [22].....	24
Obrázek 9. MVC diagram. [29]	28
Obrázek 10. MVP diagram. [30]	29
Obrázek 11. MVVM diagram. [31]	29
Obrázek 12. Logo iNaturalist. [40].....	34
Obrázek 13. Návrh drátového modelu aplikace.....	38
Obrázek 14. Na levé straně je vůbec první funkční zobrazení kamery. Pravá strana obsahuje obrazovku, když chybí povolení ke kameře.....	43
Obrázek 15. Na levém obrázku můžeme vidět vysunovací okno obsahující detekované zvíře. Prostřední obrázek obsahuje design zobrazení kamery spolu s pořízeným snímkem. Obrázek vpravo ukazuje záložku „Fotografie“ s detekovaným mravenečníkem v obrazu.....	45
Obrázek 16. Data třída Animal.....	46
Obrázek 17. Data zvířat v podobě values hodnot.....	47
Obrázek 18. Enum třída obsahující data souřadnic mapy.....	48
Obrázek 19. Detail žirafy v aplikaci.	50
Obrázek 20. Detail navigačního menu a obrazovky Objevy.	51
Obrázek 21. Vyskakovací snackbars na spodku aplikace.	51
Obrázek 22. Pohled na obrazovku zoologických zahrad v Objevech a detail obrazovky obsahující informace a mapu konkrétní zoologické zahrady Brno.	52
Obrázek 23. Vyhledávač a „O aplikaci“.....	53
Obrázek 24. Manifest s požadavky na povolení.	54
Obrázek 25. Manifest se zbytkem informací o aplikaci.	54
Obrázek 26. Anotace ve formátu VOC Pascal XML.....	59

Obrázek 27. Ukázka prostředí programu LabelImg.	59
Obrázek 28. Import knihoven pro TFlite model maker.	65
Obrázek 29. Příklad kódu pro trénování klasifikačního modelu.	65
Obrázek 30. Mapa propojených míst spadajících do MetaCentrum. [42]	70
Obrázek 31. Nastavení přihlášení v PuTTy.	71
Obrázek 32. Úspěšné přihlášení do sítě programem PuTTy.....	71
Obrázek 33. Ukázka testování pomocí obrázků.	75
Obrázek 34. Testování klasifikačního modelu skrze aplikaci.	77
Obrázek 35. Příklady klasifikovaných zvířat v zoo Zlín.	78

SEZNAM TABULEK

Tabulka 1. Pojmy při klasifikování dat modelem.....	21
Tabulka 2. Přehled API verzí s verzí Android. [34]	31
Tabulka 3. Přehled dat v klasifikačním datasetu.	57
Tabulka 4. Přehled dat v detekčním datasetu.	63
Tabulka 5. Porovnání natrénovaných modelů.	74
Tabulka 6. Porovnání klasifikování zvířat z obrázků mezi vytvořenými modely a aplikacemi Seek a Google Lens.	76

SEZNAM PŘÍLOH

Příloha P I: Stažené datasety zvířat

Příloha P II: Struktura dat na přiložené SD kartě

PŘÍLOHA P I: STAŽENÉ DATASETY ZVÍŘAT

Kapybara

- https://github.com/fredso/capybara_dataset
- <https://universe.roboflow.com/miguel-narbot-usp-br/capybara-and-animals/dataset/1>

Klokan

- <https://www.kaggle.com/datasets/hugozanini1/kangaroodataset?resource=download>
- <https://github.com/experiencor/kangaroo>
- <https://universe.roboflow.com/z-jeans-pig/kangaroo-epscj/dataset/1>

Tygr

- <https://cvwc2019.github.io/challenge.html#>

Savana

- <https://www.kaggle.com/datasets/biancaferreira/african-wildlife>

Slon

- <https://universe.roboflow.com/new-workspace-5kofa/elephant-dataset/dataset/6>

Jaguár

- <https://universe.roboflow.com/nathanael-hutama-harsono/large-cat/dataset/1/images/?split=train>

Žirafa

- <https://universe.roboflow.com/giraffs-and-cows/giraffes-and-cows/dataset/1>

Želva

- <https://universe.roboflow.com/turtledetector/turtledetector/dataset/2>
- <https://www.kaggle.com/datasets/smaranjitghose/sea-turtle-face-detection>

Zebra

- <https://universe.roboflow.com/fadilyounes-me-gmail-com/zebra---savanna/dataset/1>
- <https://universe.roboflow.com/test-qeryf/yolov5-9snhq>

- <https://universe.roboflow.com/or-the-king/two-zebras>
- <https://universe.roboflow.com/wild-animals-datasets/zebra-images/dataset/2>
- <https://universe.roboflow.com/zebras/zebras/dataset/2>
- https://universe.roboflow.com/v2-rabotaem-xkxra/zebras_v2/dataset/5

Nosorožec

- https://universe.roboflow.com/vijay-vikas-mangena/animal_od_test1/dataset/1
- https://universe.roboflow.com/bdoma13-gmail-com/rhino_horn/dataset/7

Kočka a pes

- <https://universe.roboflow.com/rudtkd134-naver-com/finalproject2/dataset/2>
- <https://universe.roboflow.com/the-super-nekita/cats-brofl/dataset/2>

Tučňák

- <https://universe.roboflow.com/lihi-gur-arie/pinguin-object-detection/dataset/2>
- <https://universe.roboflow.com/utas-377cc/penguindataset-4dujc/dataset/10>
- <https://universe.roboflow.com/new-workspace-tdyir/penguin-clfnj/dataset/1>
- https://universe.roboflow.com/utas-wd4sd/kit315_assignment/dataset/7

Jelen

- <https://universe.roboflow.com/jeonjuuniv/deer-hqp4i/dataset/1>

Ovce

- <https://universe.roboflow.com/new-workspace-hqowp/sheeps/dataset/1>
- <https://universe.roboflow.com/ali-eren-altindag/sheepstest2/dataset/1>
- <https://universe.roboflow.com/yaser/sheep-0gudu/dataset/3>
- https://universe.roboflow.com/ali-eren-altindag/mixed_sheep/dataset/1

Kráva

- <https://universe.roboflow.com/pkm-kc-2022/sapi-birahi/dataset/2>
- https://universe.roboflow.com/ghostikgh/team1_cows/dataset/5

Lev

- <https://universe.roboflow.com/ml-dlq4x/liontrain/dataset/2>
- <https://universe.roboflow.com/animals/lionnew/dataset/2>

Papoušek

- https://universe.roboflow.com/parrottrening/parrot_trening/dataset/1
- <https://universe.roboflow.com/uet-hi8bg/parrots-r4tfl/dataset/1>
- https://universe.roboflow.com/superweight/parrot_poop/dataset/5

Kočka, pes, opice, člověk

- <https://www.kaggle.com/datasets/tarunbisht11/intruder-detection>

PŘÍLOHA II: STRUKTURA DAT NA PŘILOŽENÉ SD KARTĚ

```
└── android_application/ - mobilní aplikace ve standardním složkovém formátu obsahující veškerá data
    ├── app/
    ├── build/
    ├── gradle/
    └── ...
        └── local.properties – konfigurační soubor pro vložení API klíče ke Google Places/Maps API
    └── datasets/
        ├── classification/ - dataset pro trénování klasifikačního modelu
        ├── detection/ - dataset pro trénování detekčního modelu
        ├── final_tests/ - obrázky použité v závěrečném testování aplikace
        └── scripts/ - python skripty využité při práci s datasety
    └── text/
        ├── data/ - složka obsahující obrázky a diagramy *drawio obsažené v textu a prezentaci práce
        ├── zadani.pdf - zadání bakalářské práce
        └── fulltext.pdf - finální verze odevzdané práce do systému STAG
    └── training/
        ├── classification/
            ├── train_classification.py - TensorFlow Model Maker API skript pro trénování klasifikačního modelu
            └── 18animals.tflite – vytvořený klasifikační model ve formátu *tflite s metadaty
                └── labels.txt - zvířata, které je model schopný klasifikovat
        ├── detection/
            └── tf_api_training/ - složka obsahující data využité během trénování detekčního modelu
                ├── models/ - složka obsahující data z instalace TensorFlow Object Detection API
                    ├── protoc/ - kompilační soubory pro práci TF API
                    └── scripts/
                        ├── generate_tfrecords.py - skript pro generování souborů využitých při trénování
                        └── metadata_creator.py - skript pro vložení metadat do natrénovaného modelu
                └── model_testing.py - skript využitý pro otestování vytvořených *tflite modelů
        └── workspace/
            ├── last_training(20)/ - složka obsahující přípravu u posledního trénování
            └── models_trained_on_metacentrum/ - složka obsahující několik natrénovaných modelů pomocí Metacentra
                └── ...
                    └── notes.txt - textový soubor s poznámkami k jednotlivým modelům
```

*Vytvořeno webovým nástrojem <https://tree.nathanfriend.io>.

Stejná data jsou zároveň dostupná na adrese: <https://1url.cz/TrmgR>