|  |  |
| --- | --- |
|  | |
| **Mobilní aplikace pro rozpoznávání zvířat v zoologických zahradách** | |
|  | |
| Jiří Daberger | |
|  | |
|  |  |
| 2023 | Popis: fai_logo_cz |
|  |  |

\*\*\*Do tištěné verze zde vložte oficiální zadání práce, **do PDF verze, která se nahrává do IS/STAG vložte zadání bez podpisů!**\*\*\*

**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.

Ve Zlíně, dne …………………….

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 frameworkem 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 ComposePoděkování, motto a čestné prohlášení, že odevzdaná verze bakalářské práce a verze elektronická, nahraná do IS/STAG jsou totožné ve znění:

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

Potřebné poděkování MetaCentru, nezapomenout aktualizovat, kdyby to změnili.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA CZ LM2018140 ) supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Hledat TODO pro doplnění

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

OBSAH

[Úvod 9](#_Toc130402680)

1. [TEORETICKÁ ČÁST 11](#_Toc130402681)

[1 Počítačové vidění 12](#_Toc130402682)

[1.1 Klasické úkoly počítačového vidění 12](#_Toc130402683)

[1.1.1 Klasifikace obrazu 12](#_Toc130402684)

[1.1.2 Detekce objektů 12](#_Toc130402685)

[1.1.3 Segmentace obrazu 13](#_Toc130402686)

[1.1.4 OCR 13](#_Toc130402687)

[1.2 Využití CV 14](#_Toc130402688)

[2 konvoluční Neuronové sítě 15](#_Toc130402689)

[2.1 Vrstvy CNN 15](#_Toc130402690)

[2.1.1 Konvoluční vrstva 15](#_Toc130402691)

[2.1.2 Pooling vrstva 16](#_Toc130402692)

[2.1.3 Plně propojená a Aktivační vrstva 16](#_Toc130402693)

[2.2 Učení neuronových sítí 17](#_Toc130402694)

[2.2.1 TensorFlow 18](#_Toc130402695)

[2.2.1.1 TensorFlow Lite 18](#_Toc130402696)

[2.2.2 Problémy při učení 19](#_Toc130402697)

[2.2.2.1 Trénovací data 19](#_Toc130402698)

[2.2.2.2 Overfitting 20](#_Toc130402699)

[2.2.2.3 Underfitting 20](#_Toc130402700)

[2.2.3 True vs False a Positive vs Negative 21](#_Toc130402701)

[2.3 Technika Non-Max-Suppression 21](#_Toc130402702)

[2.4 Detektory 22](#_Toc130402703)

[2.4.1 Dvoustupňové detektory 22](#_Toc130402704)

[2.4.2 Jednostupňové detektory 22](#_Toc130402705)

[2.4.2.1 YOLO 23](#_Toc130402706)

[2.4.2.2 SSD MobileNet V2 24](#_Toc130402707)

[3 Vývoj Mobilních aplikací 25](#_Toc130402708)

[3.1 Druhy vývoje mobilních aplikací 25](#_Toc130402709)

[3.1.1 Platformě závislé 26](#_Toc130402710)

[3.1.1.1 Nativní vývoj 26](#_Toc130402711)

[3.1.2 Platformě nezávislé – multiplatformní 26](#_Toc130402712)

[3.1.2.1 Webový vývoj 26](#_Toc130402713)

[3.1.2.2 Hybridní vývoj 27](#_Toc130402714)

[3.2 Architektonické vzory 27](#_Toc130402715)

[3.2.1 Model-View-Controller 27](#_Toc130402716)

[3.2.2 Model-View-Presenter 28](#_Toc130402717)

[3.2.3 Model-View-ViewModel 29](#_Toc130402718)

[3.3 Android 30](#_Toc130402719)

[3.3.1 Úrovně API 30](#_Toc130402720)

[3.3.2 Android Studio 32](#_Toc130402721)

[3.4 Framework Jetpack Compose 33](#_Toc130402722)

[4 Seznámení s podobnými projekty 34](#_Toc130402723)

[4.1 Google Lens 34](#_Toc130402724)

[4.2 Seek by iNaturalist 34](#_Toc130402725)

1. [Praktická část 35](#_Toc130402726)

[5 návrh mobilní aplikace 36](#_Toc130402727)

[5.1 Funkcionální požadavky 36](#_Toc130402728)

[5.2 Nefunkcionální požadavky 37](#_Toc130402729)

[5.2.1 Kompatibilita 37](#_Toc130402730)

[5.2.2 Forma ukládání dat 37](#_Toc130402731)

[5.3 Wireframe aplikace 38](#_Toc130402732)

[5.4 Data aplikace 39](#_Toc130402733)

[5.4.1 Resources Values 39](#_Toc130402734)

[5.4.2 Resources Drawable 39](#_Toc130402735)

[5.4.3 Úložiště SharedPreferences 39](#_Toc130402736)

[5.4.4 Ostatní data 40](#_Toc130402737)

[6 Vývoj mobilní aplikace 41](#_Toc130402738)

[6.1 Zvolené technologie 41](#_Toc130402739)

[6.2 Začátek vývoje aplikace 41](#_Toc130402740)

[6.3 Náhled kamery 42](#_Toc130402741)

[6.4 Data zvířat 43](#_Toc130402742)

[6.4.1 Data zvířat ve values podobě 45](#_Toc130402743)

[6.5 Náhled detailu zvířete 46](#_Toc130402744)

[6.5.1 Mapa zvířat 46](#_Toc130402745)

[6.5.2 Taxonomie zvířat 47](#_Toc130402746)

[6.6 Manifest 47](#_Toc130402747)

[7 Tvorba datasetu 48](#_Toc130402748)

[7.1 Image Classification dataset 48](#_Toc130402749)

[7.1.1 Použité techniky 49](#_Toc130402750)

[7.1.1.1 Flickr a jeho API 49](#_Toc130402751)

[7.1.1.2 Image augmentation 49](#_Toc130402752)

[7.1.2 Finální zhodnocení a informace 50](#_Toc130402753)

[7.2 Object Detection dataset 51](#_Toc130402754)

[7.2.1 Použité nástroje 51](#_Toc130402755)

[7.2.1.1 Program LabelImg 52](#_Toc130402756)

[7.2.1.2 Volně dostupné datasety 53](#_Toc130402757)

[7.2.1.3 Skript pro kontrolu vadných obrázků 53](#_Toc130402758)

[7.2.1.4 Skript pro přejmenování souborů 53](#_Toc130402759)

[7.2.1.5 Skript pro editaci XML souborů 54](#_Toc130402760)

[7.2.1.6 Skript pro odstranění malých obrázků 54](#_Toc130402761)

[7.2.1.7 Skript pro tvorbu „background“ obrázků 54](#_Toc130402762)

[7.2.1.8 Skript pro dokončení datasetu 54](#_Toc130402763)

[7.2.1.9 Skript pro generování TFRecord souboru 55](#_Toc130402764)

[7.2.2 Finální zhodnocení a informace 55](#_Toc130402765)

[8 trénování modelu 57](#_Toc130402766)

[8.1 API pro vytvoření vlastních modelů 57](#_Toc130402767)

[8.1.1 TensorFlow lite model maker 58](#_Toc130402768)

[8.1.2 TensorFlow 2 Object Detection API 59](#_Toc130402769)

[8.1.2.1 Instalace 59](#_Toc130402770)

[8.1.2.2 Příprava TF Object Detection API 59](#_Toc130402771)

[8.1.2.3 Trénování s TF Object Detection API 60](#_Toc130402772)

[8.1.2.4 Vytvoření tflite modelu 61](#_Toc130402773)

[8.2 Model pro klasifikaci 62](#_Toc130402774)

[8.3 Model pro detekci 62](#_Toc130402775)

[8.4 MetaCentrum 63](#_Toc130402776)

[8.4.1 Seznámení a prvotní nastavení 63](#_Toc130402777)

[8.4.1.1 Konfigurace nástroje PuTTy 64](#_Toc130402778)

[8.4.1.2 Instalace potřebných knihoven 65](#_Toc130402779)

[9 Testování 67](#_Toc130402780)

[9.1 Srovnání s dostupným řešením 67](#_Toc130402781)

[Závěr 68](#_Toc130402782)

[Seznam použitých symbolů a zkratek 73](#_Toc130402783)

[Seznam obrázků 74](#_Toc130402784)

[Seznam tabulek 75](#_Toc130402785)

[Seznam Příloh 76](#_Toc130402786)

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 náš 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 pomoct 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 detekovat tvář uživatele a zaměnit ji za něco jiného, například za hlavu zvířete, popřípadě tvář digitálně zkrášlit.

Výsledná aplikace je vytvořena za pomoci 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 frameworkem Jetpack Compose. Teoretická část je zakončena seznámením s 2 aplikacemi (4), ke kterým se vytvořená aplikace v této práci podobá.

Praktická část již obsahuje samotný proces vývoje aplikace a jejího modelu pro rozpoznávání zvířat. Popisuje jednotlivé problémy a jejich řešení, se kterými se autor během vývoje setkal. Obsahuje detailní popis tvorby dvou datasetů a jejich následné použití pro naučení modelu neuronové sítě, ke kterému dopomohla organizace MetaCentrum zrychlením času trénování. Další část je již zaměřena na vývoj nativní mobilní aplikace programovacím jazykem Kotlin s využitím frameworku Jetpack Compose. Ve finále jsou testováním zhodnoceny výsledky a porovnání s již existujícími řešeními, které řeší podobný problém, jako tato práce.

|  |  |
| --- | --- |
|  | TEORETICKÁ ČÁST |

# 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]

## 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 tisíci snímcích v relativně krátkém čase, není šance, aby se o to postaral člověk.

### Klasifikace obrazu

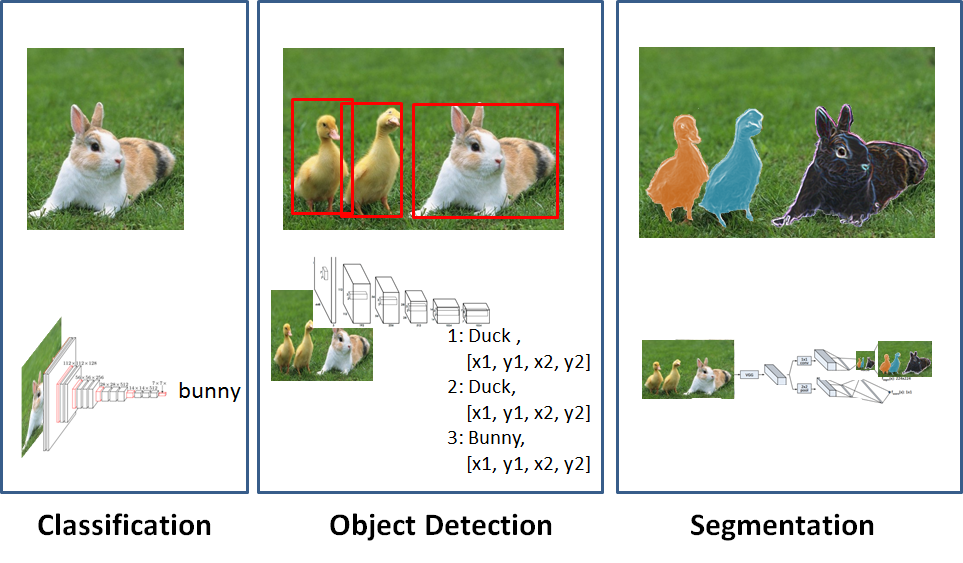
Jedná se o oblíbenou úlohu počítačového vidění, která se stará o klasifikaci obrazu do jedné předem definované třídy, podle toho, co se v obrazu vyskytuje. Výstupem je buď číselná hodnota udávající index, jež se spojuje s názvem detekované třídy objektu nebo prázdná hodnota, značící, že klasifikační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

### Detekce objektů

Jedná se o rozšíření úlohy „klasifikace obrazu“ o možnost detekovat **všechny** objekty v obraze 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 ohraničení 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]

### Segmentace obrazu

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



Obrázek 1. Kategorie počítačového vidění. [3]

### 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]

Obsah obrázku text, bílá tabule

Popis byl vytvořen automaticky

Obrázek 2. Ukázka fungování OCR. [5]

## 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 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.
* **Bezpečností systémy** – mezi které můžeme zařadit například bezpečností 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 zdlouhavě potvrzovat 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.

# konvoluční Neuronové sítě

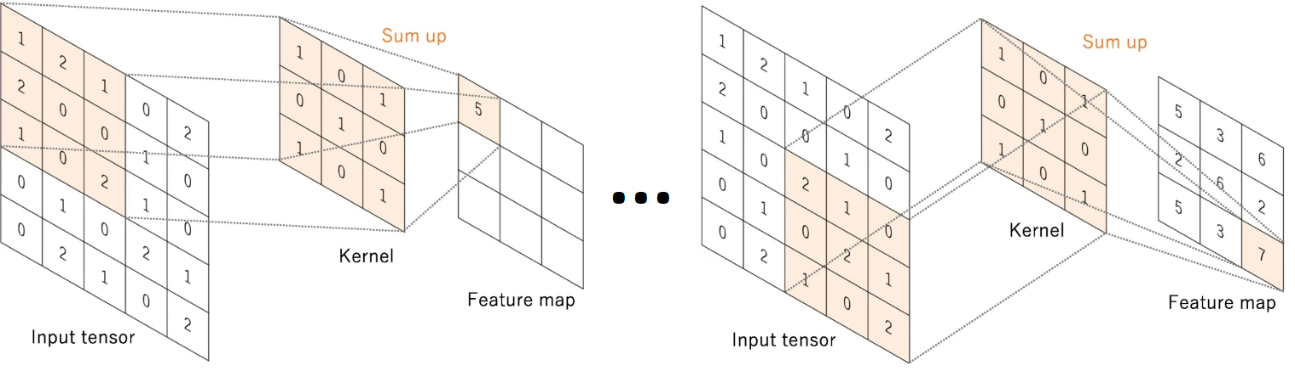
Konvoluční neuronové sítě (anglicky Convolutional Neural Network, **CNN**) jsou nejpoužívanějším typem neuronových sítí starající se o detekci či rozpoznávání objektu v obraze nebo zvuku. Jedná se o rozšířenou variantu klasických neuronových sítí o speciální vrstvu, která dopomáhá elegantně eliminovat 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]

## 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.

### Konvoluční vrstva

Jedná se o vrstvu, která v určitém směru prochází obraz pomocí několika filtrů, které zachycují různé informace obrázku, jako jsou hrany, světlost, popř. barva a jiné. Tento filtr si můžeme představit jako matici N×N (obvykle 3×3 nebo 5×5) navíc s doplněním barevného kanálu obrázku (×3 pro barevný RGB režim nebo ×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]](#footnote-2). [7]



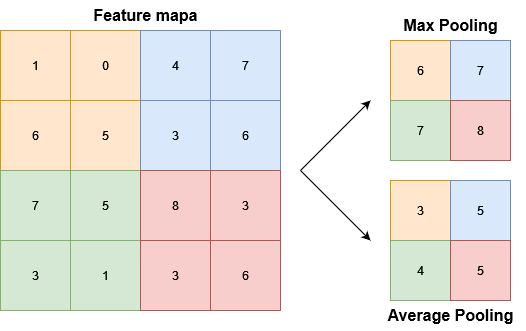
Obrázek 3. Aplikace filtru na vstupní obrázek v konvoluční vrstvě. [7]

### 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 Featrue mapy vždy se stejným nastavením.

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



Obrázek 4. Ukázka použití filtrů v Pooling vrstvě.

Na obrázku (Obr. 4) je patrné, že se mapa velikostně zredukuje o ¼ díky odstranění redundantních pixelů. To zapříčiní snížení celkového rozlišení obrázku, ale také snížení potřebného výkonu pro další potřebné výpočty a samotné trénování sítě.

### Plně propojená a Aktivační vrstva

Doposud se vždy pracovalo s 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 rozpoznat.

V této poslední 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]

## 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 iterací 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 obrázků, na kterých se bude síť učit v jeden moment. Závisí na velikosti grafické paměti a obvyklá hodnota nastavení je 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***.

|  |  |
| --- | --- |
|  | (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.

|  |  |
| --- | --- |
|  | (2) |

Každá iterace je tedy složena z **dopředné propagace**, která přijímá trénovací data na vstupu a pomocí jednotlivých vah 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 iterací. Tato úprava je závislá na chybě, která vznikne mezi očekávaným výstupem a skutečným výstupem. [8]

### 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 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 je takto vytvořené grafy možné exportovat na různá zařízení jiných systémů i architektur. [10]

#### 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 Linuxu. [11]

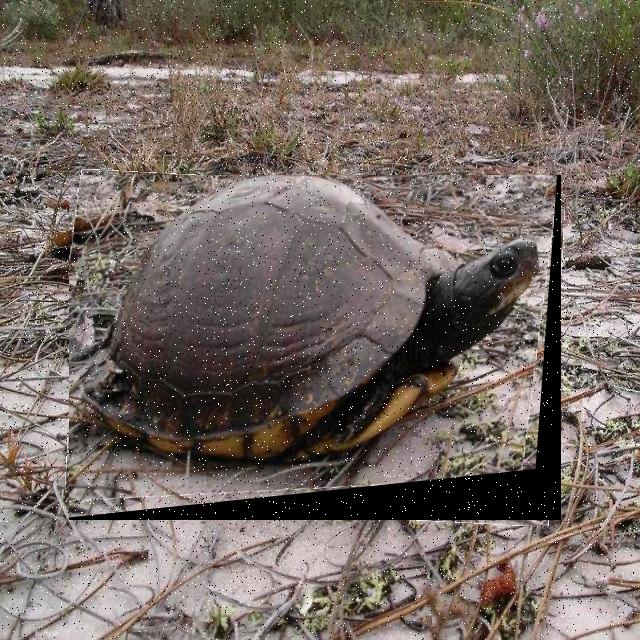
### Problémy při učení

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.

#### 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. Obecným pravidlem je alespoň 1000 obrázků do každé třídy, kterou se má síť naučit. [12]

Pro zajištění dostatečného 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.

#### Overfitting

Problém zvaný overfitting můžeme chápat 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** regulace. Jejím úkolem je ukončení spojení mezi náhodně vybranými aktivačními neurony v každé iteraci učení, čímž zvyšuje robustnost celé sítě. [13] [14]

#### Underfitting

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

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

### 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.

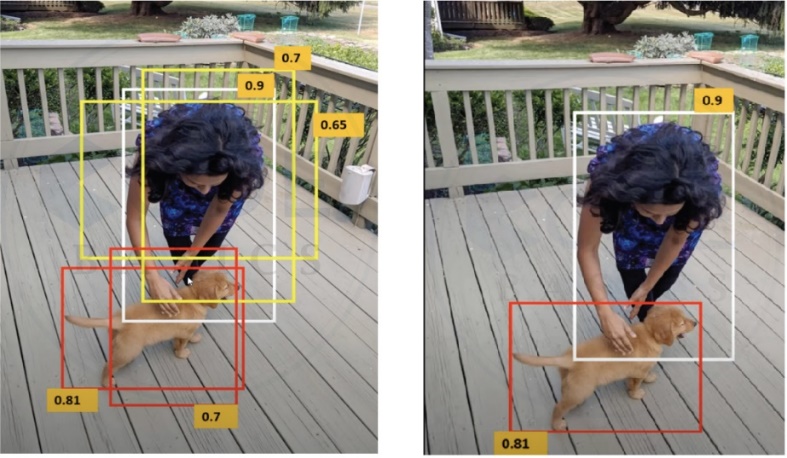
Tabulka 1. Pojmy při klasifikování dat modelem.

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

## Technika Non-Max-Suppression

Jedná se o techniku aplikovanou v poslední části detekci objektů v obraze. Její úlohou je odstranit redundantní výskyty detekcí stejného objektua 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]

## Detektory

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.

### Dvoustupňové detektory

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

Významným pokrokem v detekci objektů byl detektor **R-CNN**, který na vstupním obraze generuje přibližně 2000 regionů použitím „region proposal“ algoritmu **selektivního vyhledávání**[[2]](#footnote-3), 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 **R**egion **P**roposal **N**etwork, která generuje oblasti zájmů velice rychle. Tímto se dvoustupňový detektor Faster R-CNN přiblížil k rychlosti jednostupňových detektorů, ale stále nedosahuje dostatečnou rychlost 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]

### 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 detekuje 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.

#### YOLO

**Y**ou **O**nly **L**ook **O**nce 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. 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:

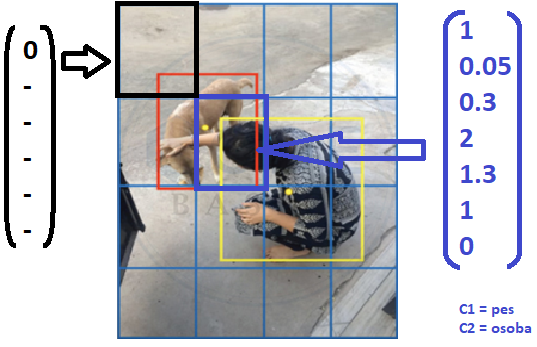
* [Pc, Bx, By, Bw, Bh, C1, C2, C...], kde **Pc** značí pozitivitu (1) nálezu nebo ne (0). Pokud je nález pozitivní (1), na dalších 4 místech je určena poloha objektu v dané buňce skrze souřadnice X a Y a velikosti šířky a výšky. Hodnoty **Ci** udávají samotné třídy sítě a aktivována (1) je pouze ta, která je právě predikovaná. Pokud je nález negativní (0), další hodnoty ve vektoru jsou irelevantní.

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).

|  |  |
| --- | --- |
|  | (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. [18]



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

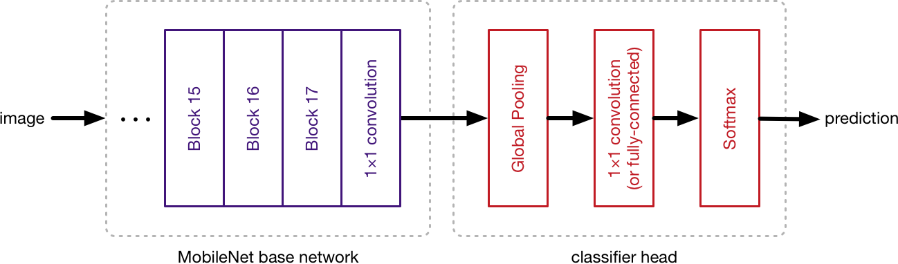
Celá klasifikace i lokalizace objektu v obraze proběhne téměř v jeden okamžik a celkový počet výsledných lokalizací jednoho objektu může být vysoký. Z tohoto důvod se na konci YOLO detektoru aplikuje již zmíněná technika **Non-Max-Suppression** (2.3). [15]

#### SSD MobileNet V2

Jedná se o spojení dvou samostatných sítí **S**ingle **S**hot **D**etector 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 okamžik 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. [19]

**MobileNet** je konvoluční neuronová síť přizpůsobena pro výpočty na mobilních zařízení. Pomocí konvolučních vrstev se stará o extrahování klíčových vlastností z obrazu a tím redukuje celkový počet parametrů potřebných k naučení sítě. [20]



Obrázek 8. Spojení MobileNet a SSD. [21]

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. [22]

# 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 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. [22]

## 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 platforem, musí se zvážit i tato možnost. Odpovědí 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. [23]

### 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 vyvíjeného zařízení. Takto vytvořené aplikace v sobě nesou menší bezpečnostní riziko, jelikož jsou přesně přizpůsobeny dané platformě a jejímu standardu.

#### Nativní vývoj

Nativní vývoj aplikací v praxi znamená, že vyvíjená 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. [23]

### 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. [23]

#### Webový vývoj

Přesným opakem nativního vývoje mobilních aplikací je webový vývoj. Jedná se o responzivní 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. [24]

#### 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. [24]

## 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.

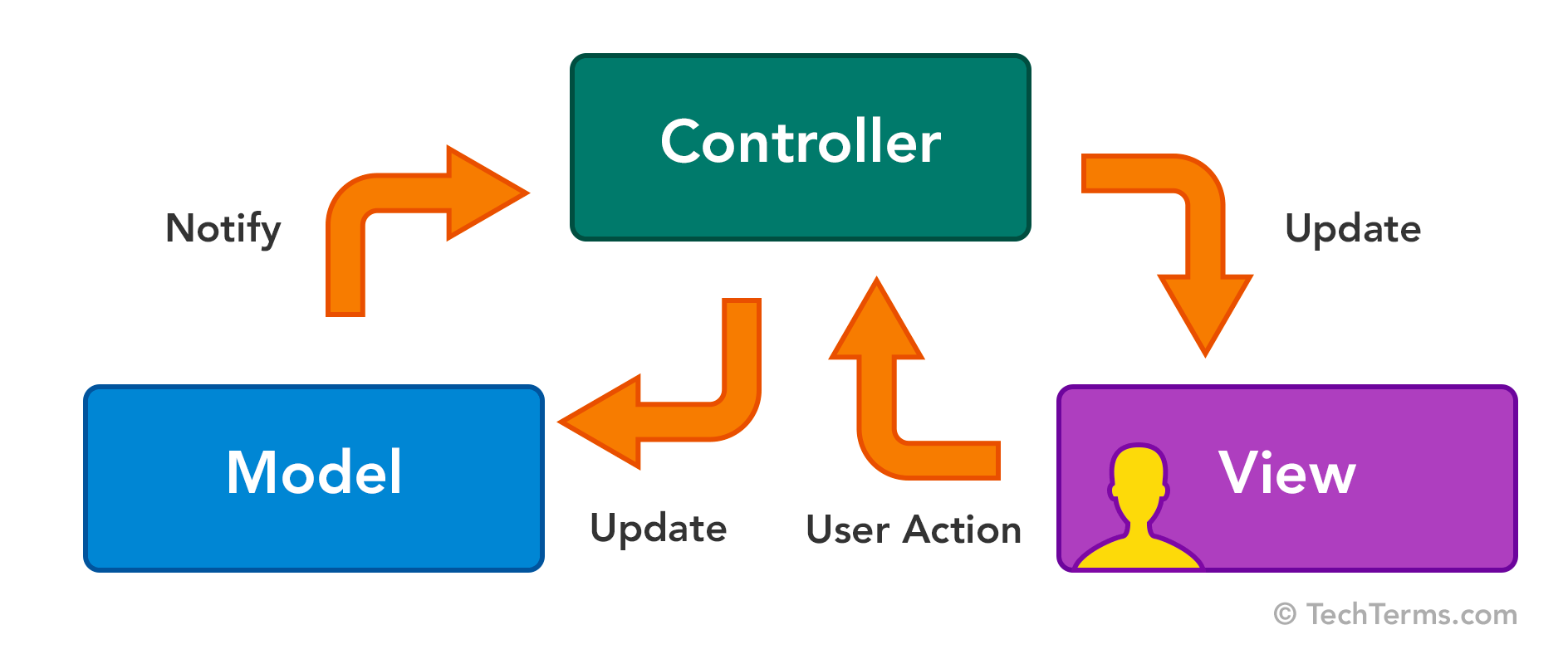
Následující příklady vzorů spadají pod tzv. třívrstvou architekturu, která se již podle názvu vyznačuje třemi základními vrstvami:

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.

### Model-View-Controller

**M**odel-**V**iew-**C**ontroller neboli **MVC**, je jeden z nejčastějších architektonických vzorů při tvorbě webových aplikací. Jeho rozdělí do tří vrstev je následující: [25]

* 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.



Obrázek 9. MVC diagram. [26]

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.

### 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.

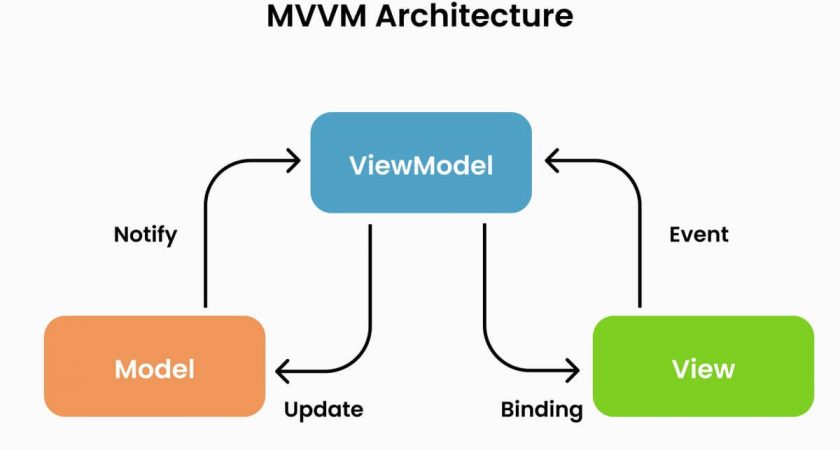
Změna od MVC přichází u samotné komunikaci mezi View a Modelem, které oproti MVC nejsou nyní napřímo propojeny, a tudíž musí Presenter předat data z Modelu do View sám   
a tím samotný View naformátovat. To umožňuje větší flexibilitu při automatizovaném testování. [27]



Obrázek 10. MVP diagram. [27]

### Model-View-ViewModel

**M**odel-**V**iew-**V**iew**M**odel (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 přeposílání na View. To znamená, že z UI je zcela odstraněn kód aplikace a data jsou pevně vázána právě na třídu ve ViewModelu, kde si neustále uchovávají aktuální stav. Tento architektonický vzor je použitý při vývoji mobilní aplikace popsané v praktické části této práce.



Obrázek 11. MVVM diagram. [28]

* **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. [29]

## Android

Jedná se o open-source projekt založený na Linuxovém jádru vyvíjeným společností Google od roku 2008. [30] Dá se považovat za krále mobilních operačních systémů, jelikož se jedná o nejvíce používaný operační systém v mobilních zařízení a před druhým nejčastějším operačním systémem iOS si každoročně udržuje odstup přibližně 43 % ve využívaných zařízení. [22]

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 jakékoliv aplikace. Tyto SDK nástroje jsou běžnou součástí oficiálního vývojového prostředí Android Studio a spadá mezi ně například nástroj **ADB**, jenž se stará   
o spuštění vyvíjených aplikací v android emulátoru nebo na připojeném fyzickém zařízením.

### Úrovně API

**A**pplication **P**rogram **I**nterface (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 mohli 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 můžou tyto informace nastavit [31]:

* **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 funkcionalitu aplikace, která by se v jiných API levelech mohla chovat odlišně, například 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 označení. **Kódové označení** je důležité zejména při vývoji aplikaci, 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 modře označeného řádku obsahující Android verzi 5.0 s API úrovní 21. Jedná se o minimální podporovanou verzi pro framework Jetpack Compose, který hraje značnou roli této práce.

Tabulka 2. Přehled API verzí s verzí Android. [31]

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

### Android Studio

Android Studio je oficiální vývojové prostředí pro tvorbu aplikací podporující operační systém Android založené na silných nástrojích od IntelliJ IDEA. 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. [32]

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

Mezi tyto 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.

## 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. [33]

Posláním Jetpack Compose je nahradit dlouhou dobu používaný návrh UI pomocí XML, jež 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 navrhnout 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**. [34] 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.

# Seznámení s podobnými projekty

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

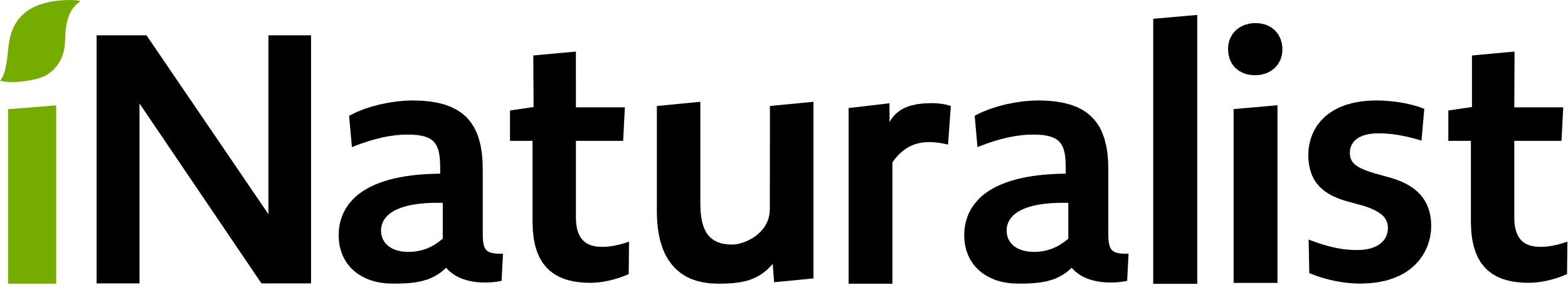
## Google Lens

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. [35]

## Seek by iNaturalist

Tato aplikace opět využívá mobilní kameru k rozpoznávání okolních objektů. Tyto objekty jsou ale specificky zaměřeny přímo na živočišné a rostlinné druhy. Aplikace umožňuje 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. [36]



Obrázek 12. Logo iNaturalist. [37]

|  |  |
| --- | --- |
|  | Praktická část |

# 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.

## Funkcionální požadavky

Mezi funkcionální požadavky můžeme zařadit to, co se od aplikace jako takové očekává. Většina požadavků byla určena při prvotním 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 je schopen rozpoznat 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 času od vyfocení
* Systém aplikace je schopen rozpoznat 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 možností filtrování Vše/Objevené/Neobjevené a jejich status objevení
* 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 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

## 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 používáním.

### 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 % [31], takže se nejednalo o závažný problém. Tím bylo zajištěné široké spektrum kompatibilních zařízení, na kterých může být aplikace spuštěna.

### 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 aplikací.

## 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 kdekoliv 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.

## 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.**

### 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 plurály[[3]](#footnote-4). 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 důvod – všechno to jsou taky jen texty.

### 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.

### 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 tato obrazovka neukáže. To je způsobeno právě uložením správné hodnoty do tohoto úložiště: key=“welcome“, value=true.

### Ostatní data

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

## 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.

# Vývoj mobilní aplikace

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

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

## 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.

## 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 změň 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 fungovaní 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 jakékoliv další obrazovky aplikace.

## Náhled kamery

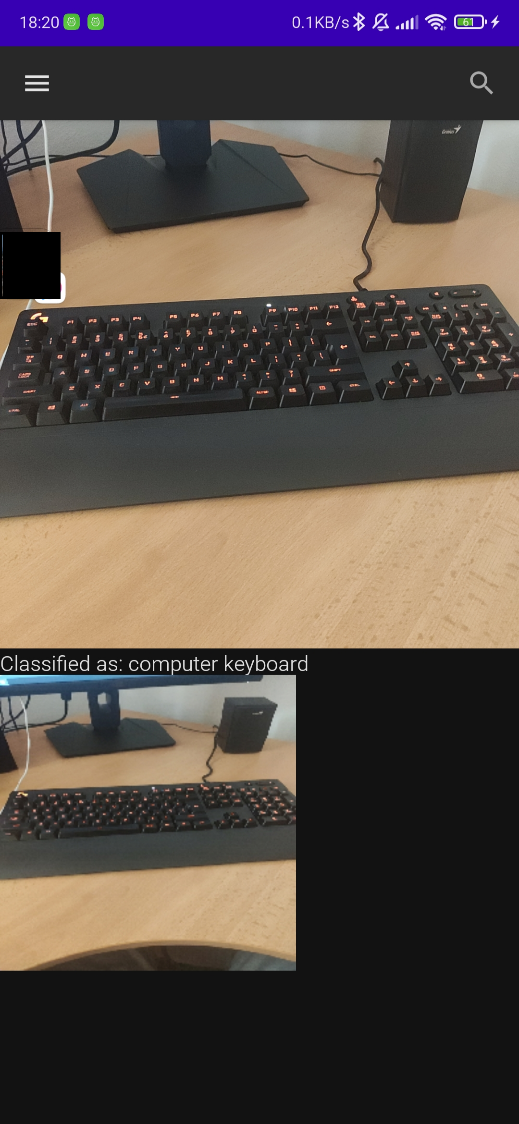
Jelikož je aplikace zaměřená na rozpoznávání zvířat za pomocí mobilní kamery, tak se jedná o jednu z nejdůležitějších obrazovek v aplikaci.

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 čemu dopomohlo volně přístupné řešení na internetu a následně zmenšený podle parametru modelu.

Ovšem aby byla kamera vůbec uživateli přístupná, musí ji před jejím prvním zpuštění 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 (6.4 Manifest).

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 náčrt obrazovky, 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.

## 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 **singleton** tříd, díky kterým je možné tuto třídu (objekt) jednou inicializovat a poté k datům kdykoliv a kdekoliv 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:

Obsah obrázku text, plaketa, snímek obrazovky

Popis byl vytvořen automaticky

Obrázek 15. 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 vstup 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.

### Data zvířat ve values podobě

K výše uvedenému obrázku (Obr. 15) a vytvoření objektu Animal nám tedy patří hodnoty zvířat v jejich podobě values hodnot. Ty mají podle obrázku níže (Obr. 16) 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 v následující kapitole.



Obrázek 16. Data zvířat v podobě values hodnot.

## 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 verzí aplikace.

### 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 tvořily 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 velikosti.

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) 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.

Obsah obrázku text

Popis byl vytvořen automaticky

Obrázek 17. Ukázka enum třídy obsahující data souřadnic mapy.

### 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 16.

### 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 zahradě v České republice, pokud těmito informacemi disponují.

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.

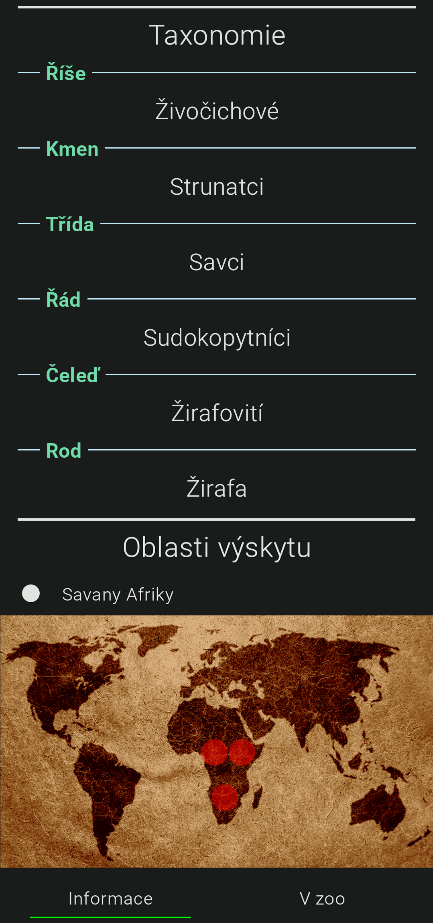
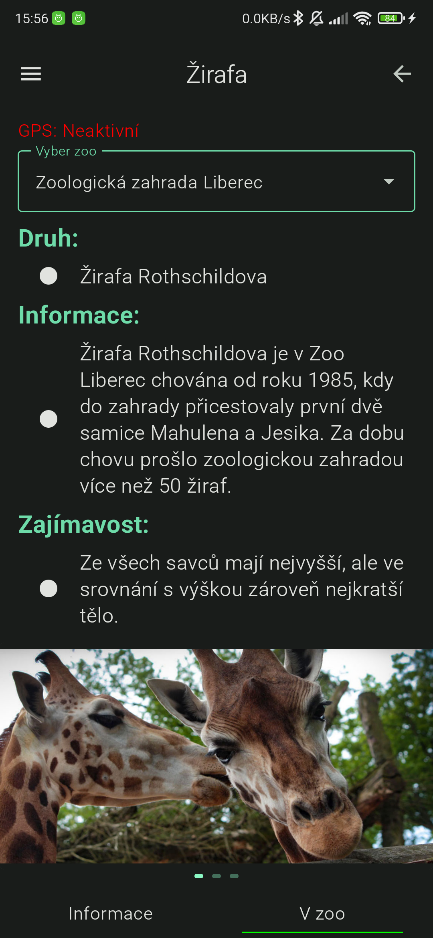
### Ukázka

Na následujícím obrázku (Obr. 18) 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 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 detekovat.

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 taktéž 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 . Detail žirafy v aplikaci.

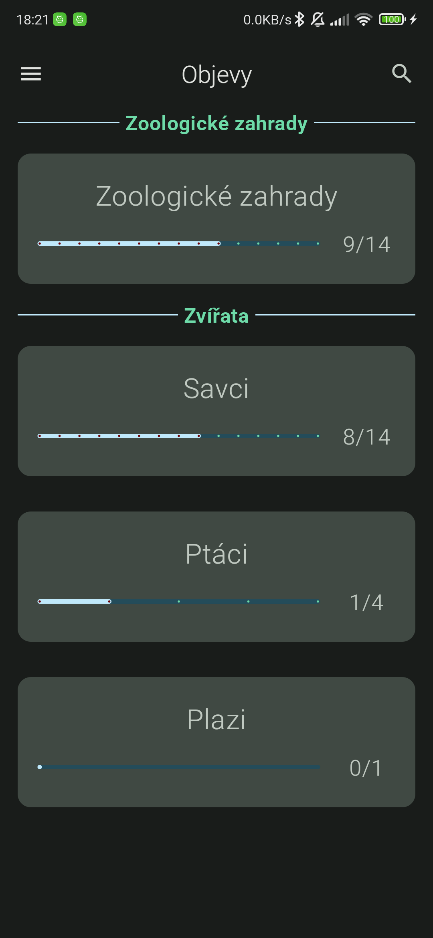
Když se přesuneme na záložku „V zoo“, dostaneme výsledek první dostupné zoologické zahrady v seznamu, která dané zvíře chová. Pokud máme připojení GPS, automaticky se zobrazí nejbližší zoologická zahrada, ale jen v případě, že dané zvíře chová. Můžeme vidět, že na stránce máme informace ohledně druhu 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.

## 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 funkcionalitu 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 zahrady a zvířata, která jsou navíc rozřazena na kategorie (Savci, Ptáci, Plazi). Každá z těchto položek má svůj vlastní počet objektů (zahrady 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“.

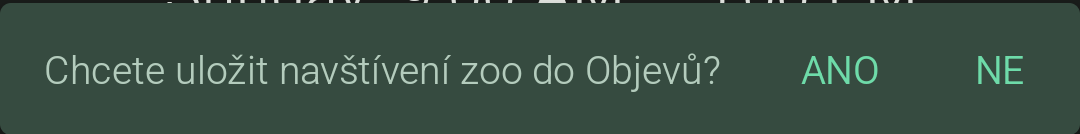
Obsah obrázku text

Popis byl vytvořen automaticky 

Obrázek . Detail navigačního menu a obrazovky Objevy.

### 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 kterým můžeme zahradu uložit do objevů (zleva doprava) nebo ji z objevů odstranit (zprava doleva). Další možnost uložení je navigování do detailu položky pomocí prokliku a zde ve spodní části skrze vyskakovací **snackbar** položku uložit. Tento snackbar je zobrazen pouze tehdy, když není položka objevena.



Obrázek . Vyskakovací snackbar na spodku aplikace.

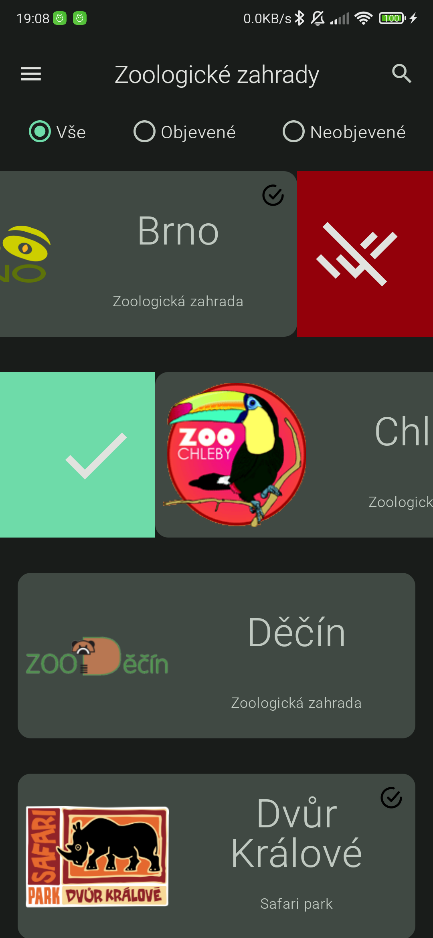
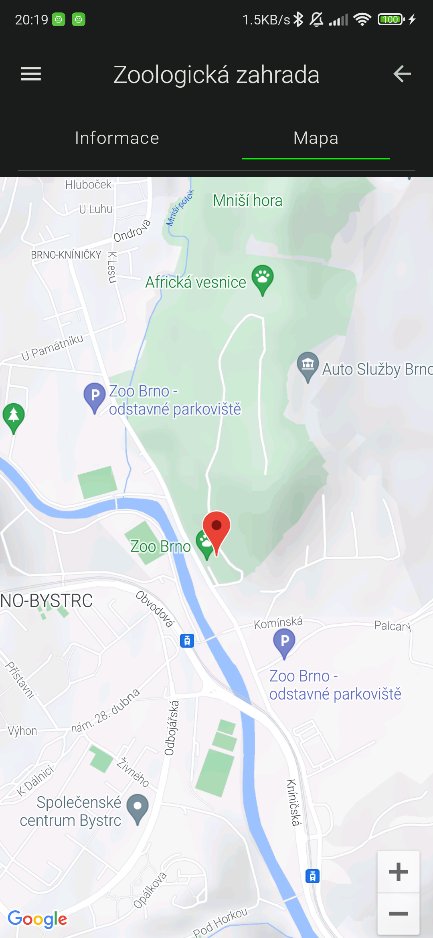
Seznam zoologických zahrad je se swipe gesty vyobrazen na obrázku níže (Obr. 21) na jeho levé straně.

### 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 [developers.google](https://developers.google.com/) stránce propojený s kreditní kartou, jelikož může být jejich použití při mnoha dotazech zpoplatněno.

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

## Manifest

Až na konci

# 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 roztříděné 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.

## 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 Klasifikace obrazu), 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 roztříděné na třídy v přibližně rovnoměrném rozdělení, k čemu dopomohou různé nástroje.

### 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.

#### 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 zkontroloval 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 opakované 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.

#### 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.

### 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 dolaďovat 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 dozvíme níže, ne všechny druhy zvířat byly v konečném trénování zahrnuty (8.1.1).

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 |

## Object Detection dataset

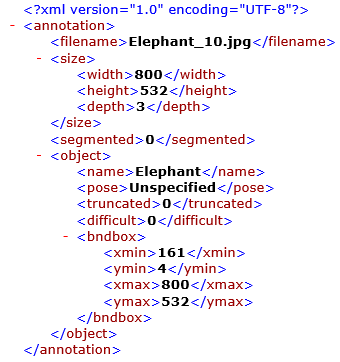
Tato sekce obsahuje detailní popis tvorby datasetu využívající CNN s technikou object detection. Vytvoření tohoto typu datasetu vyžaduje mnohem více úsilí a času než u předchozího typu. Je to díky tomu, že aby se CNN mohla lépe učit nalézt objekt v obraze, musíme jí tento objekt v trénovacích datech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

### 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 můžete vidět na obrázku níže (Obr. 15). 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.



Obrázek 22. Anotace ve formátu VOC Pascal XML.

#### 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*.

Obsah obrázku text, strom, snímek obrazovky

Popis byl vytvořen automaticky

Obrázek 23. 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.

#### 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](#Příloha_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. Takovéto 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.

TODO, jednotlivé skripty asi s odkazem do příloh

#### 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 pouze 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 zapříčiňovali jeho pád.

#### Skript pro přejmenování souborů

Během úprav datasetu bylo vždy několik obrázků staženo nebo odstraněno, což značilo nekonzistentnost 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.

#### 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.

#### 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.

#### Skript pro tvorbu „background“ obrázků

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

#### Skript pro dokončení datasetu

Tento skript je rozdělen na 2 samostatné funkce.

**První** funkce s názvem ***split*** se stará o rozdělení 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*.

#### 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.

### 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 56\_403 obrázků zabírajících 6,31 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 | 11 450 | 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 |
| Tučňák | 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 |

# 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 síť naučit, bylo potřebné nasbírat a vytřídit 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. Rychlost trénování modelů je vždy závislá 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** (8.3), které díky jejich výkonu dovolili plné trénování modelů v relativně krátkém čase.

## 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.

### TensorFlow lite model maker

Jedná se o knihovnu, která sjednocuje jednoduché trénování modelu spolu s **transfer learning**[[4]](#footnote-5), 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. [38]

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

Obsah obrázku text, dopis

Popis byl vytvořen automaticky

Obrázek 24. 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.

Obsah obrázku text, dopis

Popis byl vytvořen automaticky

Obrázek 25. Příklad kódu pro trénování klasifikačního modelu.

Z výše uvedených obrázků (Obr. 18, Obr. 19) 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.

### 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.

#### 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**. Takto 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.

#### Příprava TF Object Detection API

Trénování pomoci 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 [22] vyplývá, že jako jediné možné řešení je použití některého z modelů architektury SSD.

Během prvních testů byl stažen model [SSD MobileNet V2 FPNLite 320x320](http://download.tensorflow.org/models/object_detection/tf2/20200711/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.tar.gz). 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 vytvořená 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

│ │ └── checkpoint

│ ├── 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.

#### 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 --pipline\_cofig\_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.

#### 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/pipeline.config --trained\_checkpoint\_dir=models/my\_ssd\_lite\_320 --output\_directory=models/my\_ssd\_lite\_320/tfliteexport*

*tflite\_convert --saved\_model\_dir=models/my\_ssd\_lite\_320/tfliteexport/saved\_model --output\_file=models/my\_ssd\_lite\_320/tfliteexport/saved\_model/ssd\_320.tflite --input\_shape=1,320,320,3 --input\_arrays=normalized\_input\_image\_tensor --output\_arrays='TFLite\_Detection\_PostProcess','TFLite\_Detection\_PostProcess:1','TFLite\_Detection\_PostProcess:2','TFLite\_Detection\_PostProcess:3' --inference\_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/tfliteexport/saved\_model/ssd\_320.tflite -l ../workspace/ train\_1/models/my\_ssd\_lite\_320/classes.txt -o ../workspace/animals/models/my\_ssd\_lite\_320/tfliteexport/saved\_model/ssd\_320\_metadata.tflite*

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

## 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 dostatečné veliké.

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

## 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](http://download.tensorflow.org/models/object_detection/tf2/20200711/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.tar.gz), 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, tučňák a kapybara. Zbytek zvířat se bohužel buď kvůli malému počtu obrázků nebo jejich kvalitě do finálního modelu nedostalo.

## MetaCentrum

MetaCentrum (**MetaVO**) je virtuální organizace České republiky zabývající se poskytováním vysoce výkonného hardwaru ve formě Gridových výpočtů[[5]](#footnote-6), studentům a akademickým pracovníkům ve členském sdružení CESNET zdarma. [39]



Obrázek 26. Mapa propojených míst spadajících do MetaCentrum. [39]

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.

### Seznámení a prvotní nastavení

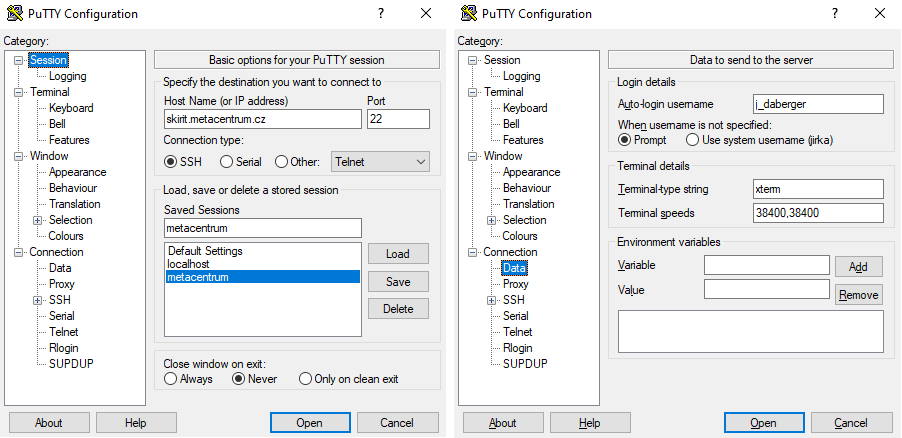
Samotná registrace probíhá klasicky jako na každé jiné stránce, ale s využitím školního emailu studenta, který je spravován federací **eduID.cz** spadající pod již zmíněné sdružení CESNET.

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 (<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.

#### 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 vše nemuseli dělat manuálně vždy, když se chceme na síť MetaVO přihlásit.



Obrázek 27. Nastavení přihlášení v PuTTy.

Na obrázku (Obr. 16) lze na levé straně v sekci „Session“ vidět, že celá konfigurace je uložená pod názvem „metacentum“. 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 znovu. Nakonec jen zadá své heslo a přihlášení do sítě je hotové.

Obsah obrázku text

Popis byl vytvořen automaticky

Obrázek 28. 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 stoje 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.

#### 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\_daberger/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 normálním 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 níž největší byla právě proměnná s názvem ***XLA\_FLAGS***.

# Testování

85.89 % score, 75.07 % iou, 2535965 ms, model c.19

76.56 % score, 71.75 % iou, 2556583 ms, model c.16

84.56 % score, 73.90 % iou, 2592627 ms, model c.18

86.07 % score, 76.20 % iou, 2616447 ms, model c.20

12.10 % score, 11.82 % iou, 2586918 ms, model c.13 – no capybara

## Srovnání s dostupným řešením

Závěr

Text

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 kateogrie počítačového vidění. In: *ML Fundamentals* [online]. [cit. 2023-02-10]. Dostupné z: https://ataspinar.com/wp-content/uploads/2017/11/deeplearing\_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] | Přeučení 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] | 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-framewor-python/ |
| [19] | 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/ |
| [20] | ALSAADI, Elham Mohammed Thabit A. a Nidhal K. El 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 |
| [21] | HOLLEMANS, Matthijs. *SSD MobileNet V2* [online]. 2018 [cit. 2023-02-04]. Dostupné z: https://machinethink.net/blog/mobilenet-v2/ |
| [22] | 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 |
| [23] | Vývoj mobilních aplikací. *Rascasone* [online]. 2021 [cit. 2023-03-15]. Dostupné z: https://www.rascasone.com/cs/blog/typy-mobilnich-aplikaci |
| [24] | VALA, Radek. *Programování mobilních aplikací: Metody vývoje mobilních aplikací* [Prezentace]. Fakulta aplikované informatiky - Univerzita Tomáše Bati ve Zlíně, 2022. |
| [25] | 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 |
| [26] | MVC diagram. In: *TechTerms* [online]. [cit. 2022-12-18]. Dostupné z: https://techterms.com/definition/mvc |
| [27] | 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 |
| [28] | MVVM diagram. In: *Bach Khoa-npower* [online]. [cit. 2022-12-19]. Dostupné z: http://bachkhoa-npower.vn/mvvm-la-gi/ |
| [29] | 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/ |
| [30] | Android. In: *Wikipedia* [online]. [cit. 2022-12-10]. Dostupné z: https://cs.wikipedia.org/wiki/Android\_(opera%C4%8Dn%C3%AD\_syst%C3%A9m) |
| [31] | Android API levely. In: *Android API Levels* [online]. [cit. 2023-02-04]. Dostupné z: https://apilevels.com/ |
| [32] | Android Studio. In: *Android Developers* [online]. [cit. 2023-02-09]. Dostupné z: https://developer.android.com/studio/intro |
| [33] | Informace o Jetpack Compose. In: *Android Developers* [online]. [cit. 2023-02-11]. Dostupné z: https://developer.android.com/guide |
| [34] | Rekompozice v Jetpack Compose. In: *Android Developers* [online]. [cit. 2023-02-11]. Dostupné z: https://developer.android.com/jetpack/compose/state#introducing-state |
| [35] | *Google Lens* [online]. [cit. 2023-02-17]. Dostupné z: https://lens.google/howlensworks/ |
| [36] | *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 |
| [37] | Logo iNaturalist. In: *INaturalist* [online]. [cit. 2023-02-17]. Dostupné z: https://www.inaturalist.org/ |
| [38] | 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 |
| [39] | *MetaCentrum* [online]. [cit. 2022-12-30]. Dostupné z: https://metavo.metacentrum.cz/cs/ |
| [40] | UIJLINGS, J. R. R., K. E. A. 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

SDK Software Development Kit

IDE Integrated Development Environment

API Application Programming Interface

UI User Interface

OS Operating System

iOS iPhone Operating System

NFC Near Field Communication

GPS Global Positioning System

SMS Short Message Service

HTML Hypertext Markup Language

Tzv tak zvaný

Popř popřípadě

TODO

Seznam obrázků

[Obrázek 1. Kategorie počítačového vidění. [3] 12](#_Toc129262259)

[Obrázek 2. Ukázka fungování OCR. [5] 12](#_Toc129262260)

[Obrázek 3. Aplikace filtru na vstupní obrázek v konvoluční vrstvě. [7] 14](#_Toc129262261)

[Obrázek 4. Ukázka použití filtrů v Pooling vrstvě. 15](#_Toc129262262)

[Obrázek 5. Příklad augmentovaného obrázku. 18](#_Toc129262263)

[Obrázek 6. Obrázek znázorňující použití **NMS** techniky. [15] 20](#_Toc129262264)

[Obrázek 7. Obrázek s mřížkou 4×4 obsahující objekty se středovými body a vektory predikce. [15] 22](#_Toc129262265)

[Obrázek 8. Spojení MobileNet a SSD. [21] 23](#_Toc129262266)

[Obrázek 9. MVC diagram. [25] 26](#_Toc129262267)

[Obrázek 10. MVP diagram. [26] 27](#_Toc129262268)

[Obrázek 11. MVVM diagram. [27] 28](#_Toc129262269)

[Obrázek 12. Logo iNaturalist. [36] 32](#_Toc129262270)

[Obrázek 13. Návrh oken aplikace. 37](#_Toc129262271)

[Obrázek 14. Anotace ve formátu VOC Pascal XML. 42](#_Toc129262272)

[Obrázek 15. Ukázka prostředí programu LabelImg. 43](#_Toc129262273)

[Obrázek 16. Mapa propojených míst spadajících do MetaCentrum. [37] 48](#_Toc129262274)

[Obrázek 17. Nastavení přihlášení v PuTTy. 49](#_Toc129262275)

[Obrázek 18. Úspěšné přihlášení do sítě programem PuTTy. 49](#_Toc129262276)

Seznam tabulek

[Tabulka 1. Pojmy při klasifikování dat modelem. 20](#_Toc127894117)

[Tabulka 2. Přehled API verzí s verzí Android. [29] 29](#_Toc127894118)

Seznam Příloh

Příloha P I: Stažené datasety zvířat

Kapybara

* https://github.com/freds0/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#](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

1. Matici **Kernel** na obrázku chápejme jako **filtr**. [↑](#footnote-ref-2)
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. [40] [↑](#footnote-ref-3)
3. Plurál = odvození množného čísla v texu (1 zebr**a**, 2 zebr**y**) [↑](#footnote-ref-4)
4. Jedná se o techniku strojového učení, kdy se naučené vlastnosti nějakého problému aplikují na jinou/novou úlohu. [↑](#footnote-ref-5)
5. Gridové výpočty znamenají propojení několika PC z různých lokací pro dosažení co největšího výpočetního výkonu. [↑](#footnote-ref-6)