UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Mobilní aplikace pro detekci objektů pomocí knihovny TensorFlow Lite

Jan Hendrych

Bakalářská práce
2024

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jan Hendrych**
Osobní číslo:	**I20094**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce:	**Mobilní aplikace pro detekci objektů pomocí knihovny TensorFlow Lite.**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem bakalářské práce bude vytvoření stabilní mobilní aplikace, která bude detekovat jednotlivé objekty a přiřazovat jim určité vlastnosti. Bakalářská práce bude obsahovat detailní popis postupu vývoje aplikace a následný popis aplikace. Bakalářská práce bude také obsahovat popis samotné TensorFlow knihovny a vysvětlení pojmů strojové učení a umělá inteligence.

Rozsah pracovní zprávy: **min. 30 stran**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

BAYLISS, Darryl, Tom BLANKENSHIP, Fuad KAMAL a Namrata BANDEKAR. *Android apprentice: beginning android developement with Kotlin*. Second edition. [McGaheysville]: Razeware, [2019]. ISBN 978-1-942878-77-3.

CHOLLET, François. *Deep learning v jazyku Python: knihovny Keras, Tensorflow*. Přeložil Rudolf PECINOVSKÝ. Praha: Grada Publishing, 2019. Knihovna programátora (Grada). ISBN 978-80-247-3100-1.

Vedoucí bakalářské práce: **Ing. Jan Panuš, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **16. prosince 2022**
Termín odevzdání bakalářské práce: **12. května 2023**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2023

Prohlášení autora

Prohlašuji:
Práci s názvem mobilní aplikace pro detekci objektů pomocí knihovny TensorFlow Lite jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.
Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 10. 05. 2024 Jan Hendrych

Poděkování

Rád bych vyjádřil svou vděčnost všem, kteří mi pomohli při tvorbě této bakalářské práce.

Nejprve bych chtěl poděkovat mému vedoucímu práce, Ing. Janu Panušovi, Ph.D. za jeho trpělivost, vedení a cenné rady během celého procesu tvorby této práce. Jeho odborné znalosti a podpora byly neocenitelné.

Dále bych rád poděkoval své rodině a přátelům za jejich neustálou podporu, povzbuzení a porozumění, které mi poskytovali během mého celé doby mého studia.

Vaše podpora byla pro mě neocenitelná.

Jan Hendrych.

Anotace

Cílem této práce je seznámit čtenáře s termíny umělé inteligence a strojového učení. Tato práce ve své první části popisuje umělou inteligenci a její historii. V této části se také čtenář dozví informace potřebné k pochopení fungování modelů strojového učení, určených k detekci objektů na obrázcích. V další části práce je čtenáři popsán vývoj mobilní aplikace využívající modely od společnosti TensorFlow. Tyto modely jsou určené k detekci a klasifikaci objektů na obrázcích. V poslední části práce je čtenáři představen postup vývoje vlastního modelu pro detekci objektů.

Po přečtení této práce je čtenář schopný vytvořit vlastní mobilní aplikaci a implementovat do ní TensorFlow modely.

Klíčová slova

Umělá inteligence, strojové učení, hluboké učení, Android, Kotlin, TensorFlow, klasifikace objektů, detekce objektů.

Title

A mobile application for object detection using the TensorFlow Lite library.

Annotation

The aim of this thesis is to introduce the reader to the terms artificial intelligence and machine learning. In its first part, this thesis describes artificial intelligence and its history. In this part, the reader will also learn the information needed to understand the operation of machine learning models designed to detect objects in images. The next part of the thesis describes to the reader the development of a mobile application using models from TensorFlow. These models are designed to detect and classify objects in images. In the last part of the thesis, the reader is introduced to the process of developing a custom model for object detection.

After reading this thesis, the reader is able to create a custom mobile application and implement TensorFlow models in it.

Keywords

Artificial Intelligence, Machine Learning, Deep Learning, Android, Kotlin, TensorFlow, Object Classification, Object Detection.

Obsah

Seznam zkratek

AI	Artificial intelligence
ACM	Association for Computing Machinery
MIT	Massachusetts Institute of Technology
IBM	International Business Machines Inc.
IEEE	Institute of Electrical and Electronics Engineers
AAAI	Association for the Advancement of Artificial Intelligence
DARPA	Defense Advanced Research Projects Agency
GPT	Generative Pre-trained Transformer
R. U. R.	Rossumovi univerzální roboti
MVC	Model View Controller
UI	User interface
SDK	Software development kit
XML	Extensible Markup Language
KT	Kotlin
SQLite	Structured Query Language Lite
BLOB	Binary Large Object
SQL	Structured Query Language
JPG	Joint Photographic Experts Group
GB	Gigabajt
T4	Název grafické karty v aplikace Google Colab
PBTXT	Protocol Buffer text file
TFRECORD	TensorFlow Record
TAR	Tape archive
GZ	gzip
mAP	Mean Average Precision

Seznam obrázků

Seznam tabulek

Úvod

V dnešní digitální době hraje strojové učení a umělá inteligence důležitou roli při vývoji nových technologií a aplikací. Systémů a aplikací využívajících umělou inteligenci neustále přibývá. Tato práce seznámí čtenáře se základními koncepty umělé inteligence, strojového učení a hlubokého učení.

Čtenář této práce se seznámí se základními principy strojového učení. Tato práce popisuje základní algoritmy učení používané pro vývoj modelů umělé inteligence. Čtenáři této práce jsou také představeny základy neuronové sítě. Práce také popisuje historii umělé inteligence, od počátečních myšlenek a konceptů až po moderní využití umělé inteligence v různých systémech a aplikacích. Tento pohled do historie představí čtenáři klíčové milníky a události v historii vývoje umělé inteligence.

Následně se práce zaměří na praktický popis vývoje mobilní aplikace pro operační systém Android. Tato aplikace slouží jako demonstrace modelů počítačového vidění. Aplikace využívá konkrétně modely pro klasifikaci objektů a detekci objektů na obrázcích. Čtenáři jsou představeny postupy a algoritmy k vytvoření spustitelné aplikace, využití modelů pro detekci objektů a zobrazení korektních výsledků z modelů.

Po představení implementace již existujících modelů do aplikace je v práci popsán postup vývoje vlastního modelu umělé inteligence. Tento postup zahrnuje detailní popis všech částí vývoje modelu. Tento model bude vytvořen pomocí knihovny TensorFlow. Proces vývoje modelu zahrnuje sběr a přípravu dostatečného množství dat. Po sběru dat následuje výběr, konfigurace a trénování modelu. Proces vývoje modelu končí testováním a optimalizací vytvořeného modelu.

Po přečtení této práce bude mít čtenář přehled o základních principech a historii umělé inteligence. Čtenář bude schopný vytvořit vlastní mobilní aplikaci a implementovat do ní již existující, či vlastní modely umělé inteligence.

1 Umělá inteligence, strojové učení a hluboké učení

Umělá inteligence a obory s ní spojené se stávají jedním z nejdůležitějších témat, které se probírají na konferencích největších technologických institutů. Umělá inteligence se stala hlavním tématem masových médií s rostoucí popularitou aplikací jednotlivých modelů. Modely umělé inteligence umožnily vývoj inteligentních aplikací, které pomáhají uživatelům s různými úkoly. Příklady těchto inteligentních aplikací jsou například aplikace pro analýzu psaného textu, či mluveného slova, aplikace pro překlad textů a aplikace pro generování obrázků [3, 29, 30].

Termíny umělá inteligence, strojové učení a hluboké učení se stávají základními stavebními kameny většiny aplikací. Každý tento termín má specifické vlastnosti a odlišnosti. V této kapitole jsou tyto základní termíny v oblasti umělé inteligence popsány.

1.1 Umělá inteligence

Umělá inteligence je ze tří zmiňovaných termínů nejstarší a nejobsáhlejší. Termíny strojové učení a hluboké učení bychom mohli označit jako podmnožinu umělé inteligence. Obecně, když hovoříme o umělé inteligenci, hovoříme o simulaci lidské inteligence prováděné stroji [29].

Umělou inteligenci bychom mohli rozdělit na slabou a silnou umělou inteligenci. Slabá umělá inteligence je vytvořena a natrénována pouze na úzce vymezené množině dat. Tato slabá umělá inteligence byla vytvořena pro práci s úzce vymezenými úlohami. Tato umělá inteligence většinou nepracuje bez lidské interakce. Mezi příklady této inteligence bychom mohli uvést hlasové asistenty jako je Google Assistant, nebo Siri od společnosti Apple. Silná umělá inteligence umí řešit několik složitých úloh najednou a nalézt řešení bez lidské pomoci a intervence. Silná umělá inteligence bývá vytvořena pro řešení komplexních úloh [3, 47]. Příkladem silné umělé inteligence je například samořízené auto. Aplikace umožňující autu samo se řídit musí zaznamenávat pomocí kamer a senzorů své okolí. Následně by tato aplikace musela vyhodnocovat možné hrozby a upravovat průběžně řízení vzhledem k hrozbám a dopravní situaci. Samořízená auta jsou zatím pouze ve fázi vývoje [29, 30].

1.1.1 Historie umělé inteligence

Umělá inteligence, také označována jako AI (z anglického „Artificial intelligence“), se zrodila v padesátých letech minulého století. Jedním z prvních a nejznámějších informatiků zabývajících se umělou inteligencí byl britský matematik, logik a kryptoanalytik Alan Turing. Alan Turing je veřejně známý díky svým zásluhám během 2. světové války, kdy pomohl dešifrovat nacistický tajný kód Enigma. Pro dešifrování německého kódu Enigma sestrojil Turing elektromagnetický stroj, který dokázal kód dešifrovat. Dešifrováním kódu zachránil Turing spoustu lidských životů. V kontextu umělé inteligence je ale nejdůležitější pokus zvaný Turingův test. Alan Turing představil tento test v roce 1950. Turing se při konstrukci svého testu snažil zodpovědět na otázku „mohou stroje myslet?“. Namísto úvah, co přesně znamená termín „myslet“ navrhl Turing zkoušku založenou na imitaci lidské konverzace. Podle Turingova testu se může stroj označit za myslící, pokud nedokáže nestranná osoba určit rozdíl v chování mezi strojem imitujícím člověka a člověkem samotným. Test spočíval v komunikaci mezi člověkem, který hodnotí test a dalším člověkem, nebo počítačovým programem. Test probíhal prostřednictvím textové konverzační aplikace a hodnotitel se snažil rozlišit, zda komunikuje se strojem, nebo s

lidskou bytostí. Pokud počítačový program přesvědčil hodnotitele o své inteligenci a lidské podstatě, byl test považován za úspěšný. Na počest Alana Turinga byla vytvořena v roce 1966 Turingova cena. Jedná se o každoroční ocenění udělované asociací výpočetní techniky (zkratka ACM) za technický přínos pro obor informatiky. Turingova cena bývá označována jako „Nobelova cena informatiky“ [3, 4, 49].

Další důležitý milník pro umělou inteligenci je rok 1956. V létě tohoto roku se uskutečnila letní konference ve výzkumném a vzdělávacím institutu Dartmouth. Dartmouth se nachází ve městě Hanover ve státě New Hampshire ve Spojených státech amerických. Na této konferenci představili John McCarthy Marvin Minsky, Nathaniel Rochester a Claude Shannon projekt, který se oficiálně nazýval „The Darthmouth Summer Research Project on Artificial Intelligence“. Projekt měl za úkol zkoumat možnosti, jak by mohli počítače a inteligentní systémy nabývat inteligence. Bohužel samotný projekt nesplnil očekávání v oblasti spolupráce a ustálení pravidel pro definici počítačové umělé inteligence. John McCarthy je uznáván jako první člověk, který použil termín umělá inteligence. Za svůj přínos v oboru umělé inteligence získal John McCarthy v roce 1971 Turingovu cenu. John McCarthy je také tvůrce programovacího jazyka Lisp Výzkumný projekt v Darthmouth je označován jako událost, kdy byla umělá inteligence uznána jako vědecký obor [5].

Na přelomu 50. a 60. let 20. století byl v institutu MIT zahájen vývoj programovacího jazyka Lisp. Jeho autorem byl John McCarthy. První verze jazyka byla pojmenována Lisp 1.5 a byla zpřístupněna veřejnosti v 60. letech 20. století. Význam zkratky Lisp není jednoznačně známý, ale diskutuje se mezi dvěma názory. První názor tvrdí, že Lisp je zkratkou pro „List Programming“, kdežto druhý názor tvrdí, že se jedná o zkratku pro „List Processor“. Je ovšem jednoznačné, že v základu pracuje Lisp převážně s datovou strukturou „List“. List (česky seznam) je abstraktní datová struktura, která představuje konečný počet uspořádaných hodnot. Lisp poskytuje programátorovi prostředky pro kvalitní a pohodlnou práci s listy. Díky této schopnosti zpracovávání listů, byl Lisp využíván v systémech pro dokazování teorémů, v systémech pro manipulaci s matematickými výrazy a v systémech pro vývoj umělé inteligence včetně expertních systémů. Expertní systém je specializovaný typ umělé inteligence, který využívá znalosti odborníků v oboru k řešení konkrétních problémů. Postupem času byl jazyk Lisp neustále rozšiřován o nové verze, nadstavby a dialekty. Mezi tyto nové verze patří například MacLisp, InterLisp a OpenLisp [6].

Další známé jméno v historii umělé inteligence je Arthur Lee Samuel. Arthur Lee Samuel dokončil studium na institutu MIT v roce 1926. Po několika pracovních zkušenostech, například v Bellových laboratořích, se stal v roce 1946 profesorem na univerzitě v Illinois. Zde dostal svůj nápad na vytvoření programu, který by dokázal porazit i světové šampióny ve hře dáma. Vytvořením tohoto programu chtěl demonstrovat sílu a schopnosti elektronických počítačů. V roce 1949 se stal Arthur Lee Samuel součástí IBM laboratoře ve městě Poughkeepsie, ve státě New York. V této laboratoři vytvořil na počítači IBM 701 svůj první program pro hraní hry dáma. Později se ukázalo, že se jednalo o první samoučící se program na světě. Program byl demonstrací na svou dobu revolučního hardwaru a kvalitního programování. Při demonstrování programu vzrostly akcie firmy IBM za jedinou noc. Program pro hraní hry dámy byl velmi raným příkladem metod, které se nyní používají ve strojovém učení. Při tvorbě programu využil Arthur Samuel záznamy komentovaných profesionálních her, kde byli rozlišeny dobré a špatné tahy. Pomocí tohoto rozlišení tahů naučil svůj program vybírat ty nejlepší možné tahy v dané situaci. V roce 1961 vyzval Samuel šampióna ze státu Connecticut, aby si zahrál proti jeho programu. Samuelův program vyhrál. Arthur Samuel se i ve stáří nadále zabýval programováním. Samuel je také

označován za jednoho z prvních a největších popularizátorů strojového učení. Za své zásluhy získal v roce 1987 od společnosti IEEE cenu „Computer Pioneer Award“ (česky „počítačový průkopník“) [7].

Mezi lety 1964 až 1966 byl německo-americkým profesorem Josephem Weizenbaumem vytvořen první program, který dokázal komunikovat s uživatelem pomocí textových zpráv. Program dostal jméno Eliza. Hlavním cílem programu Eliza bylo chovat se jako rogersovský psychoterapeut. Rogersovská psychoterapie spočívá ve vciťování se do pacienta a v empatickém porozumění. Program převážně opakoval uživatelovi výroky a kladl jednoduché a krátké otázky sestavené na základě klíčových slov z předešlých zpráv uživatele. Program Eliza spíše simuloval inteligenci, místo snahy opravdu porozumět textu od uživatele. Ovšem navzdory své jednoduchosti dokázal program Eliza zapojit uživatele do smysluplné konverzace a stal se předlohou pro novější podobné programy. Díky tomu, že se s programem komunikovalo pomocí psaných zpráv se celé kategorii podobných programů začalo říkat konverzační boti (v ančličtině „chatterbot“, nebo „chatbot“) [8, 9].

Mezi lety 1966 až 1972 byl ve výzkumném centru Stanford vyvíjen jeden z prvních robotů, kteří dokázali plánovat a vykonávat komplexní úkoly a práce. Robot dostal název Shakey, protože byl relativně vysoký a při pohybu se často třásl (anglicky „shake“ znamená „otřásat“). Projekt, který měl za úkol sestavit tohoto robota kombinoval prvky robotiky, počítačového vidění a umělé inteligence. Zatímco ostatní roboti museli mít instrukce pro svůj každý jednotlivý krok, Shakey dokázal analyzovat složitější příkazy a sám si je rozdělit na menší instrukce. Robot Shakey byl vybaven sonarovými dálkoměry, kamerou, radiovou anténou pro příjem signálu, motorem, otočnými kolečky, hlavní řídící jednotkou a senzory pro detekci kolizí. Díky tomuto vybavení dokázal určit svoji polohu, vyhýbat se překážkám a přesouvat předměty. Robot byl převážně naprogramován pomocí programovacího jazyka Lisp [10].

V roce 1979 byla založena organizace s názvem „American Association for Artificial Intelligence“ (zkráceně AAAI). Jedná se o mezinárodní vědeckou organizaci působící dodnes. Tato organizace má za úkol zvýšit povědomí o umělé inteligenci, zlepšit školení odborníků a financování současného vývoje umělé inteligence. Tato organizace také sponzoruje a provádí konference na téma umělé inteligence a vydává vlastní časopis, kde informuje o novinkách z oboru umělé inteligence a strojového učení. Společnosti od svého založení předsedali významní technologičtí vědci a profesoři, například John McCarthy, Marvin Minsky a Allen Newell. V roce 2007 se organizace přejmenovala na „Association for the Advancement of Artificial Intelligence“ [6, 11].

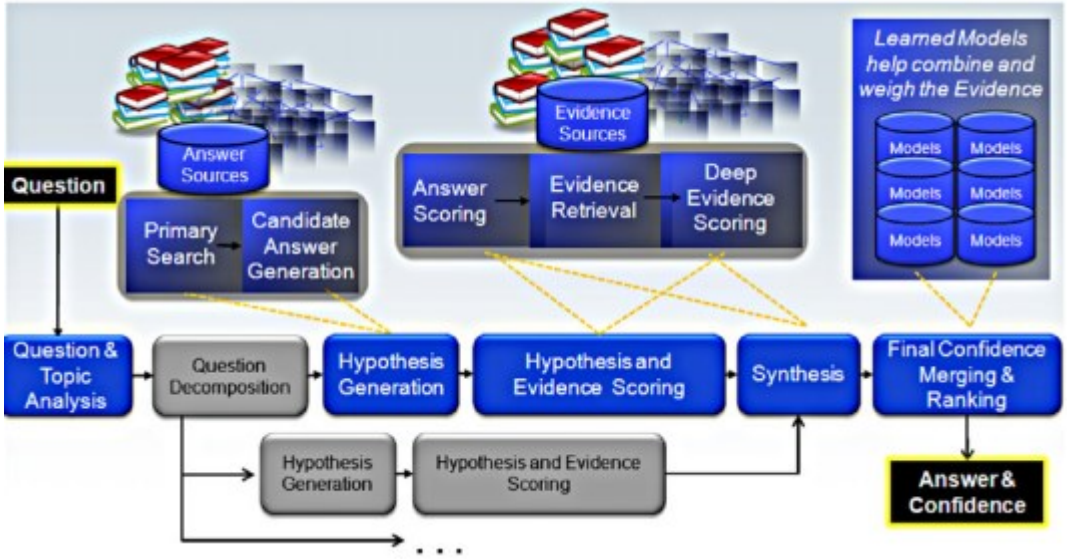
V 80. letech 20. století v Německu vytvořil tým odborníků, který vedl Ernst Dickmanns, dodávku se zabudovanými senzory a kamerami. Dodávka obsahovala také řídící jednotku a pomocí již zmíněných kamer dokázala zpracovávat sekvence obrázků z výhledu řidiče v reálném čase. Dodávka dokázala reagovat na světla semaforů, silniční značky a překážky v cestě. Testování samořídící autonomní dodávky probíhalo na uzavřené silnici bez provozu. Dodávka se v roce 1986 dokázala již kompletně sama řídit. Tento jeden z prvních autonomních dopravních prostředků dostal název „VaMoRs“ [50].

Ačkoliv se umělá inteligence zdála být velice úspěšná a byl o ní velký zájem, přišel ke konci 20. století velký úpadek ve vývoji a provozu umělé inteligence. Jednalo se o období neúspěchů, skepse a ztráty víry v umělou inteligenci, které se dnes nazývá „AI Winter“ (česky „zima umělé inteligence“, nebo také „zamrznutí umělé inteligence“). Jeden z velkých problémů pro umělou inteligenci během tohoto období byl článek Jamese Lighthilla pro

Britskou vědecko-výzkumnou radu. Tato zpráva byla zveřejněna v roce 1973. Zpráva přinesla velmi pesimistickou prognózu, říkajíc, že „v žádné části oboru dosavadní objevy nevyvolaly takový velký dopad, jaký byl tehdy slibován“. Této zprávě se začalo říkat „Lighthill report“ a byla základem pro rozhodnutí britské vlády ukončit podporu vývoje a výzkumu umělé inteligence na britských univerzitách. Další drtivá rána pro vývoj umělé inteligence byla zrušení financování od agentury s názvem „Defense Advanced Research Projects Agency“ (zkráceně DARPA). DARPA je americká agentura ministerstva obrany pro pokročilé výzkumné projekty [51, 52, 53].

Zájem o umělou inteligenci v očích veřejnosti opět vzplanul po poražení Garryho Kasparova umělou inteligencí ve hře šachy. Tento šachový systém se jmenoval „Deep Blue“ a byl vyvíjen v polovině 90. let firmou IBM. Systém Deep Blue měl několik verzí. První verze systému prohrála s Garrym Kasparovem v roce 1996, ale druhá dokonalejší verze ho o rok později porazila. Druhá verze Deep Blue byla trénována na velké databázi her. Databáze obsahovala přes 700 000 profesionálních her. Deep Blue také dokázal rychle vyhodnocovat komplexní funkce a zjišťoval, které tahy jsou nejefektivnější (například jak důležitá je pozice figurky). Systém Deep Blue by se dal označit za masivně paralelní superpočítač s několika úrovněmi paralelismu a silným důrazem na vyhledávací algoritmy. Vyhledávání probíhalo pomocí několika procesorů. Tyto procesory prohledávali komplexní a rozsáhlý herní šachový strom. Tento strom obsahuje všechny možné tahy v dané hře šachů. Tento strom se postupně rozvětvuje v závislosti na oponentových i hráčových tazích. Šachový algoritmus musel prohledat všechny uzle a listy tohoto stromu a vyhodnotit nejlepší tah. Deep Blue obsahoval jeden hlavní procesor, který prohledával první uzly stromu a následně předával další uzly stromu ostatním procesorům pro prohledávání. Rychlost prohledávání se měnila v závislosti na složitosti a komplexnosti tahů, ale průměrně dokázal Deep Blue zhodnotit 100 milionů pozic za sekundu [12, 18].

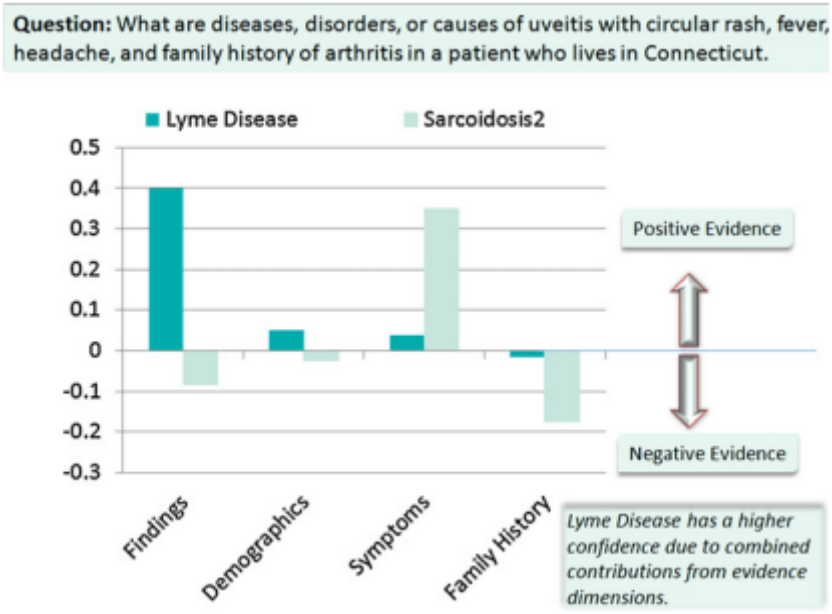
V roce 2007 si dala IBM za úkol sestavit další inteligentní systém, který by dokázal vyhrát nad šampióny v další hře. Tentokrát chtěli vytvořit inteligentní systém, který by dokázal odpovídat na položené otázky v přirozeném jazyce. Počítači dali jméno Watson. Watson měl za úkol porazit nejlepší hráče ve hře „Jeopardy!“. Jeopardy je název americké televizní soutěže, kde soutěžící dostávají otázky z různých okruhů a musí na ně správně odpovědět. V soutěži se vyskytují například tyto otázky: „Slovo, které má 4 písmena a označuje vyhlídku, nebo víru se nazývá?“ (správná odpověď je „view“, česky pohled, nebo názor). Watson byl vyvinut na architektuře DeepQA. DeepQA představuje softwarovou architekturu určenou pro hloubkovou analýzu obsahu a pro myšlení na základě důkazů. Tato architektura zahrnuje vyspělé techniky zpracování přirozeného jazyka, sémantické analýzy, vyhledávání informací, automatického uvažování a strojového učení, což jí dává silnou analytickou kapacitu [13].



Obrázek 1 - DeepQA architektura

Zdroj: Watson: beyond jeopardy!. Artificial Intelligence [13]

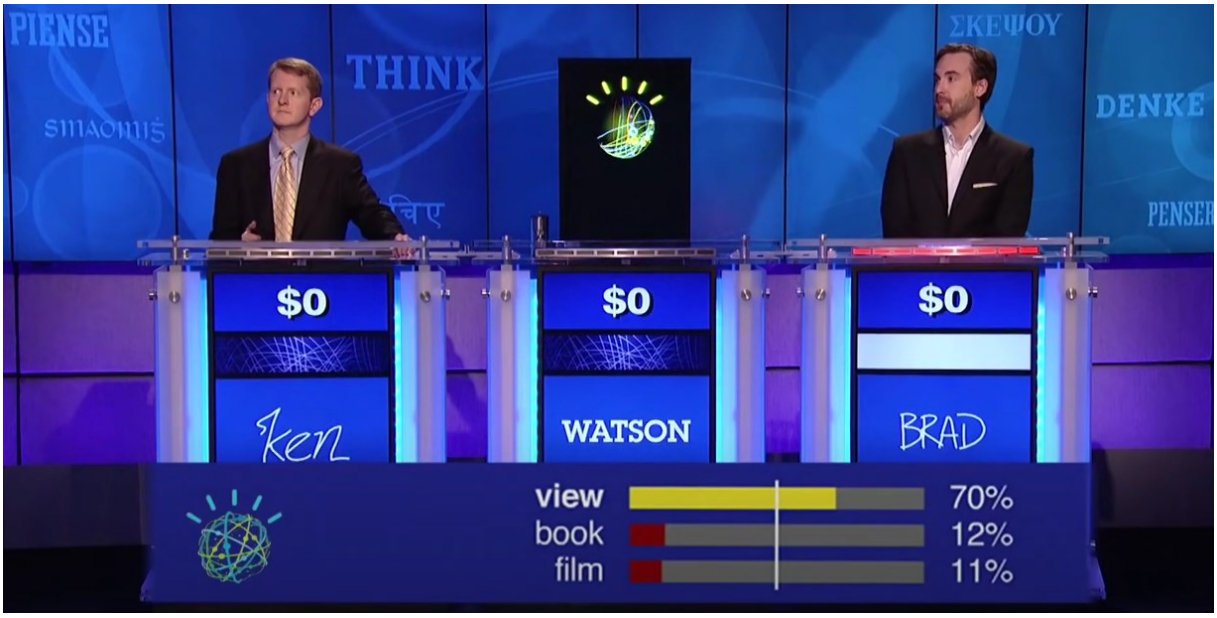
Architektura DeepQA obsahovala velký zdroj odpovědí a důkazů, ve kterých hledala nejlepší odpověď na otázku. DeepQA architektura nejdříve analyzovala vstup, aby přesně určila, na co se jí uživatel ptá a vygenerovala mnoho možných částečných odpovědí. Pro každou tuto částečnou odpověď byla vytvořena hypotéza na základě kontextu originální otázky a tématu. Každou hypotézu obsluhuje samostatné vlákno, které se jí snaží dokázat. Každé hypotéze je přiřazené skóre, které označuje, jak pravděpodobné je, že je hypotéza pravdivá. DeepQA přiřazuje každé hypotéze skóre na základě velkého zdroje důkazních materiálů, které obsahují záznamy z hlediska, času, geografie, popularity, spolehlivosti a další. Výsledkem tohoto algoritmu je seznam ohodnocených a důvěryhodných částečných odpovědí. Později se architektura DeepQA začala testovat i ve zdravotnictví.



Obrázek 2 - Nalezení správné nemoci

Zdroj: Watson: beyond Jeopardy!. Artificial Inteligence [13]

V roce 2011 vyhrál počítač Watson, postavený na architektuře DeepQA, soutěž Jeopardy proti dvojici lidských hráčů [13].



Obrázek 3 - Ukázka průběhu soutěže Jeopardy!
Zdroj: Watson and the Jeopardy! Challenge [Youtube] [14]

V roce 2011 se také na trhu objevil první populární hlasový asistent Siri od společnosti Apple. Siri byla nejdříve aplikace v operačním systému iOS, ale 4. října 2011 byla implementována jako asistent na mobilní zařízení iPhone 4S. Od této doby jí společnost Apple neustále vyvíjí a implementuje do všech svých zařízení, která pracují s operačním systémem iOS. Siri pracuje se vstupem v podobě hlasových dotazů. Následně dokáže Siri provést zadané úkony či vyhledat informace. Siri dokáže upravovat nastavení telefonu (například měnit jas, nebo hlasitost), nebo například volat kontaktům, či vyhledávat v internetovém prohlížeči, nebo otevírat aplikace.

V posledních letech se vývoj umělé inteligence rapidně zrychlil. Širokou veřejnost oslnilo spousty firem, které se zabývají vývojem umělé inteligence, strojového učení, nebo také výrobou robotů. Například firma Boston Dynamics je známá pro svoje konstrukční řešení několika robotických systému. Tyto systémy mají humanoidní dvounohé, nebo čtyřnohé zástupce, kteří zatím slouží k přenosu materiálu a břemen. Další známou robotickou společností je Hanson robotics, která je nejvíce známá díky svému vývoji humanoidního robota se jménem Sophia. Sophia byla aktivována 19. dubna 2015. Jedná se o humanoidního robota, který má tvář a postavu dospělé ženy. Pro návrh tváře Sophie posloužila britská modelka a herečka Audrey Hepburn. Mezi známé a průlomové vlastnosti robota Sophie patří schopnost gestikulovat a předvádět mimiku obličeje. Sophie je vybavena několika kamerami díky kterým dokáže rozpoznat lidské tváře, a ukládat si jejich tváře a jména do paměti. Díky tomu dokáže každého z databáze oslovit, nebo pozdravit jeho jménem. Sophie je také schopna lidské interakce a hovoru o specifických tématech. Sophie funguje jako textový chatbot, ovšem komunikuje prostřednictvím hlasových dotazů a odpovídá vlastním hlasem. Díky své nadčasovosti a podobnosti s člověkem získala Sophie jako úplně první robot lidské občanství v Saudské Arábii [15].

V únoru roku 2016 oznámil Demis Hassabis, jeden ze zakladatelů Google DeepMind, soutěž mezi systémem AlphaGo a světovým šampionem v deskové hře Go Lee Sedolem. Tento turnaj se měl odehrát v březnu téhož roku ve městě hlavní město Jižní Koreje Soulu. Tento turnaj mezi inteligentním systémem a profesionálním hráčem vyhrál systém AlphaGo v poměru výher 4:1. Hra Go, je desková hra s původem v Číně. Hra se skládá z hrací desky

a černých a bílých žetonů. Hrací deska se skládá z 19 svislých a 19 vodorovných čar. Hráči se střídají a pokládají žetony na průsečíky čar na hrací ploše. Cílem hráče je obklíčit svými žetony protivníkovi žetony, čímž je „zajme“, vyjme je z hrací desky a dá si je na svoji stranu stolu. Tyto zajaté žetony slouží jako bodování hry. Hráčovo skóre tvoří také získané území na desce ohraničené hráčovými žetony. Kvůli velké herní desce, která obsahuje celkem 361 políček a velkému počtu možných strategických umístění žetonu je velice komplikované najít nejlepší umístění pro konkrétní žeton. Systém AlphaGo pro rychlé vyhodnocování situace na hrací desce kombinoval metodu Monte Carlo pro prohledávání stromové struktury dat a neuronové sítě, které byli trénovány pomocí dvou metod. První metoda se jmenovala „učení po dohledem“ a byla vytrénována za pomoci databáze her od profesionálních hráčů. Druhá metoda se jmenovala „učení posilováním“ a pomáhala systému AlphaGo se zdokonalovat na základě svých odehraných her. Princip systému AlphaGo se skládal ze tří hlavních částí. První část systému se starala o predikování tahů, na základě již zmíněních neuronových sítí, které byli trénovány na již odehraných hrách. Další část se starala o predikování tahů, podobně jako první část systému, ale mnohem rychleji. Tato druhá část simulovala lidskou intuici. Poslední část vyhodnocovala pravděpodobnost výhry na základě aktuální situace na herní desce. Při trénování systému byli programu předávány na vstupu pozice žetonů ve formě obrázků. Systém poté využil neuronové sítě ke snížení potřebné hloubky a šířky hledání ve stromové struktuře možných tahů [16, 17, 18].

V posledních letech ovšem širokou veřejností rezonuje společnost OpenAI a její produkt GPT-3. GPT-3 je již třetí generací modelu umělé inteligence pro generování přirozeného jazyka. Společnost OpenAI byla založena v roce 2015. Společnost se začala objevovat v prvních článcích již v roce 2019 díky zveřejnění modelu GPT-2 (zkratka pro „Generative Pre-trained Transformer“ česky „Generativní předtrénovaný transformátor“). Tato druhá generace modelu byla trénována na textových datech o velikosti 40 GB. Architektura GPT modelů je postavena na principu strojového učení a neuronových sítí. Uživatel zadá vstup v textové formě, GPT model analyzuje jazyk a pomocí prediktoru vytvoří nejpravděpodobnější a nejpřesnější výstup. Generování správných výstupů je docíleno díky velké množině textových dat, na kterých jsou modely GPT trénované. Kvůli možnosti zneužití technologie GPT společnost OpenAI pozdržela spuštění této technologie pro veřejnost. Po vlně kritiky, ze strany médií, byl ovšem model vypuštěn do světa. Nejznámějším produktem firmy OpenAI je aplikace ChatGPT. Jedná se o chatbot spuštěný v roce 2022. Tento chatbot byl vytvořen na základě GPT modelů a slouží výhradně ke generování textového výstupu. Nejnovější čtvrtá generace GPT modelu, ovšem dokáže místo psaní textu také vytvářet obrázky, básně, zdrojové kódy, testovací data, seznamy, tabulky a mnoho dalšího. Ačkoliv nejnovější model GPT dokáže velice věrně simulovat inteligentní chování a poskytovat užitečné informace, ne vždy jsou informace úplné a správné [19].

Z tohoto oddílu je patrné, že umělá inteligence má za sebou bohatou, ale krátkou historii. Historie umělé inteligence dokazuje, že i ze začátku jednoduchá myšlenka dokáže časem vyrůst do složité vědní disciplíny. Můžeme předpokládat, že se toto odvětví informatiky bude rychle vyvíjet a posouvat.

1.1.2 Kultura a umělá inteligence

S termínem umělá inteligence se můžeme setkat i v historii České republiky. Byl to Karel Čapek, který společně se svým bratrem Josefem vymysleli a jako první použili slovo robot v Karlově divadelní hře R.U.R. (zkratka pro „Rossum’s Universal Robots“, česky

„Rossumovi univerzální roboti“). Divadelní hra R.U.R. vznikla roku 1920 a slovo robot bylo použito pro označení robotických strojů, kteří svým chováním a zjevem připomínali člověka. Pro slovo robot bylo bratrům Čapkovým inspirací slovo robota. Robota označovala těžkou, nucenou práci, mnohdy bez jakékoliv odměny [20].

1.1.3 Technologická singularita

Technologická singularita je pojem, který označuje vznik technologických entit, které se inteligencí vyrovnají člověku, či ho inteligenčně předčí. Jedná se o budoucí teoretický scénář, ve kterém umělá inteligence překoná inteligenci a schopnosti člověka. Taková inteligence by následně mohla vytvářet lepší verzi sebe sama. Čím kvalitnější by tato inteligence byla, tím by se vyvíjela rychleji. Křivka zdokonalování by tedy byla exponenciální [21]. Taková umělá inteligence by dokázala předčit a plně zastoupit člověka.

1.2 Strojové učení

Strojové učení je podoblast vědy zabývající se umělou inteligencí. Program založený na strojovém učení je schopný se sám učit a zlepšovat na základě získaných zkušeností. Schopnost učit se znamená ve strojovém učení schopnost změnit a zefektivnit algoritmus aplikace, který se dokáže naučit vzory a vztahy mezi vstupními daty. Z informací získaných z těchto vstupních dat dokáže umělá inteligence provést určitá rozhodnutí a také předvídat budoucí vstupní data. Strojové učení je přesnější při použití většího počtu vstupních dat. Přesnost se také zvyšuje s opakovaným spuštěním trénovacího programu implementujícího strojové učení [27]. Strojové učení se spoléhá na přirozenou dovednost člověka učit se ze zkušeností [1].

1.3 Základní rozdělení strojového učení

Existuje několik typů algoritmů strojového učení, z nichž každý má své výhody i nevýhody. V této kapitole se čtenář seznámí se čtyřmi základními typy strojového učení. Jedná se o učení s učitelem, učení bez učitele, samořízené učení a posilované učení. Pochopením a porozuměním rozdílů, mezi těmito typy strojového učení nám pomůže lépe porozumět jejich aplikaci v praktických úlohách [1].

1.3.1 Učení s učitelem

V metodě učení pod dohledem se algoritmus učí z označených dat. Těmto označeným datům je poskytnuta sada korektních výstupních hodnot. Algoritmus se tedy učí dosáhnout požadované korektnosti.

V současnosti se jedná o nejrozšířenější případ strojového učení. Mezi příklady aplikací využívající učení s učitelem patří aplikace pro rozpoznávání znaků, rozpoznávání řeči a překladu jazyka. Pokročilejší příklady učení s učitelem jsou i aplikace, které umí detekovat, či segmentovat objekty na obrázku. Chceme-li vyvinout aplikaci pro detekci objektů, musíme připravit vstupní sadu obrázků. Obrázky musí obsahovat ohraničující obdélník u každého detekovatelného objektu na obrázku. Chceme-li vyvinout aplikaci pro segmentaci objektů, musíme nakreslit přesnou masku kolem objektů na vstupních obrázcích.

Metoda učení pod dohledem také umožňuje klasifikovat data. Metoda se učí předpovědět a určit kategorii, nebo třídu pro daná vstupní data. Například jsou-li vstupní data obrázky zvířat může být cílem klasifikovat každý obrázek do jedné z několika kategorií, jako jsou

například "ovce", "kráva" nebo "koza". Ke správnému určení kategorie se musí aplikace dostatečně natrénovat z kontrolních dat [1, 27, 38].

1.3.2 Učení bez učitele

V metodě učení bez dohledu se algoritmus učí z neoznačených dat. Algoritmu tedy není poskytnut korektní výstup a snaží se najít vztahy mezi daty sám. Algoritmus využívá především dříve nabyté zkušenosti ke správnému rozpoznání tříd. Tento algoritmus se většinou využívá k nalezení skrytých vzorců a podobností mezi prvky velké množiny vstupních dat. Nejčastěji se algoritmus učení bez učitele využívá ke shlukování a vizualizaci dat [1, 27, 38].

1.3.3 Samořízené učení

V metodě samořízeného učení se algoritmus učí z označených dat podobně jako u učení s učitelem. Ovšem na rozdíl od učení s učitelem nejsou data označena člověkem. Algoritmus si data na vstupní množině označuje sám.

Příklad algoritmu samořízeného učení jsou například aplikace, které dokážou předvídat následující snímek videa, na základě předchozích snímků. Dalším příkladem samořízeného učení jsou aplikace, které dokážou předvídat následující slovo ve větě, na základě předchozích slov [1, 27,38].

1.3.4 Posilované učení

Při posilovaném učení získává algoritmus informace o svém okolí a snaží se volit akce, které maximalizují nějakou odměnu. Místo kontroly správnosti jako v metodě učení s učitelem, je v posilovaném učení implementován systém odměn a trestů. Příkladem posilovaného učení může být aplikace hrající hru. Tato aplikace se každým svým tahem snaží docílit co nejlepšího skóre. Dalším příkladem může být algoritmus hrající šachy, který provádí nejideálnější tah [1, 27,38].

1.4 Hluboké učení

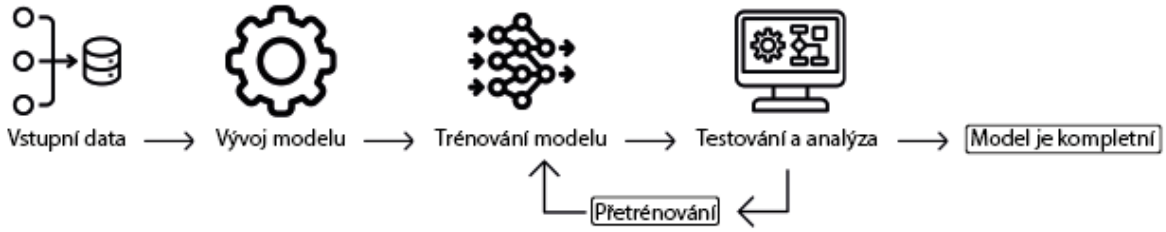
Hluboké učení je specifickou podmnožinou strojového učení. Jedná se o způsob učení se z reprezentací dat. Tyto data jsou ovšem členěny do několika vrstev. Počet těchto vrstev lze označit jako hloubka modelu. Tyto vrstvy postupně extrahují z dat stále složitější a abstraktnější rysy. Díky tomuto chování jsou schopny modely hlubokého učení pracovat efektivně s velkým množstvím dat a efektivně se přizpůsobovat složitým vzorům. Modely hlubokého učení se tyto vrstvené reprezentace dat snaží naučit pomocí neuronových sítí.

Převážně díky vrstvám a neuronovým sítím je dnes hluboké učení jednou z nejvíce se rozvíjejících oblastí umělé inteligence [27, 39].

1.5 Princip vytvoření aplikace založené na strojovém učení

Pro správné vytvoření aplikace založené na strojovém učení je potřeba získat vstupní data. Je důležité, aby byl počet dat adekvátní a dostatečně velký. Data musí být následně zkontrolována, zda neobsahují chyby, které by ovlivnili výpočet. Následně je vybrán vhodný model pro analýzu dat. Pomocí správně zvoleného modelu budou data zpracována. Na řadu přichází trénování modelu. Při trénování upravuje model své parametry a vlastnosti, aby poskytl správné a přesné výsledky. Na konci každého trénování modelu jsou data

zanalyzována a porovnána s testovací sadou dat. Testuje se přesnost, nebo například rozptyl. Pokud výstupní data modelu neodpovídají kontrolním datům na požadované přesnosti je proces trénování modelu zopakován. Pokud jsou výsledná data modelu dostatečně přesná je model dostačující a může být implementován do aplikace [48].



Obrázek 4 - Proces strojového učení
Zdroj: Vlastní zpracování

2 Neuronové sítě

Neuronová síť připomíná svojí podobou strukturu grafu, ve kterém je několik vrcholů spojeno hranami. Každý vrchol označuje v neuronové síti vrstvu, nebo uzel a hranu lze chápat jako spojení mezi vrstvami. Touto grafovou strukturou se neuronové sítě občas popisují jako simulace lidského mozku. V této simulaci mozku by uzly sítě představovali biologické neurony [23].

Základní stavební strukturou v neuronové síti je vrstva. Vrstva reprezentuje modul zpracování dat, který na vstupu přijímá jeden, nebo více tenzorů a vrací jeden nebo více tenzorů na výstupu. Tenzory se dají zjednodušeně popsat jako vícerozměrné pole hodnot. Vrstvy jsou mezi sebou navzájem propojené. Tato propojení mají svoji váhu neboli vliv, jaký mají na další vrstvu [27]. Váhy se během trénování neuronové sítě učí. Během trénování se váhy mění tak, aby minimalizovali ztrátu. Ztráta udává rozdíl mezi předpovídanou a skutečnou hodnotu. Cílem učení je najít optimální hodnoty vah, které umožní síti optimální učení [22, 23, 28].

Velmi často používaný algoritmus v neuronových sítích je algoritmus zpětného šíření chyby. Tento algoritmus se využívá ke zpětné úpravě vah u jednotlivých vrstev. Hodnota vah se mění s cílem nalezení optimální hodnoty, která zaručí co největší přesnost a správnost výsledné neuronové sítě [40].

3 TensorFlow

K vývoji aplikace byla využita knihovna TensorFlow. Tato knihovna poskytuje užitečné nástroje pro práci s modely umělé inteligence. Knihovna je v projektu využita k demonstraci klasifikace a detekce objektů.

3.1 Popis TensorFlow Lite

TensorFlow Lite je menší verzí knihovny TensorFlow, která se dá lépe implementovat na omezenějších přístrojích jako jsou mobilní telefony a tablety. Modely v TensorFlow Lite jsou jednoduše implementované, což umožňuje rychlé vytvoření široké škály aplikací. Aplikace implementující knihovnu TensorFlow Lite mohou běžet lokálně a nevyžadují připojení ke cloudu, kde normálně probíhá složitý výpočet [2, 48].

3.2 Počítačové vidění

Počítačové vidění lze chápat jako analýzu fotografií, videí a zvukových vjemů počítačem. V počítačovém vidění hraje hlavní roli umělá inteligence, která získává z fotografií, videí a zvuků důležité informace. Implementaci počítačového vidění lze nalézt například v robotice, kde roboti kamerami a senzory skenují okolí [41]. K rozpoznání objektů na pořízených snímcích se využívají metody klasifikace, detekce a segmentace objektů. Klasifikaci a detekci objektů si demonstrujeme později při vytváření Android aplikace.

3.2.1 Klasifikace objektů

Klasifikace objektů spočívá v přiřazení klasifikovatelné třídy objektu na obrázku na základě jeho vizuálního vzhledu. Často se s ní můžeme setkat v jednoduchých úlohách demonstrujících počítačové vidění. Tyto úlohy se například snaží rozlišit, zda se na obrázku vyskytuje kočka, či pes. Klasifikace objektů se většinou využívá, je-li na snímku přítomný pouze jeden objekt. Modely umělé inteligence, které klasifikují objekty na obrázcích rozpoznají objekt na obrázku s určitou mírou pravděpodobnosti správného určení objektu [24, 41]. Pokud je tedy vstupem modelu například obrázek s kočkou, výstupem modelu může být správný název objektu a číslo, udávající pravděpodobnost správnosti klasifikace. Klasifikaci lze chápat také jako rozpoznání objektu. Klasifikace objektů bude podrobněji demonstrována při tvorbě Android aplikace.

3.2.2 Detekce objektů

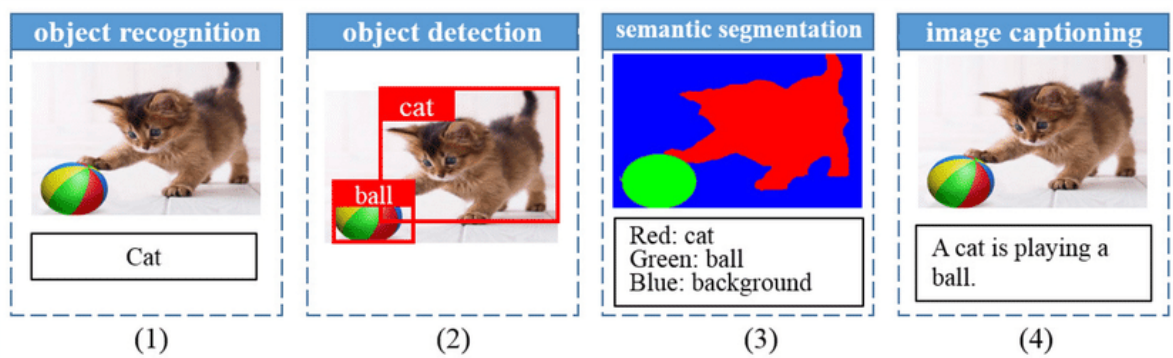
Detekce objektů je již pokročilejší příklad počítačového vidění. Detekci objektů je možné efektivně využít i na snímcích obsahující několik různých objektů. Každý objekt na snímku je ohraničen obdélníkem. Poblíž ohraničujícího obdélníku se umísťuje název detekovaného objektu [41]. K názvu objektu lze přidat také pravděpodobnost správnosti detekce objektu. Podrobněji bude detekce objektů demonstrována také při vytváření Android aplikace.

3.2.3 Segmentace objektů

Mezi další příklad počítačového vidění patří segmentace objektů na obrázku. Segmentace objektů je z již zmiňovaných příkladů počítačového vidění nejnáročnější na implementaci. Při trénování modelu, který dokáže segmentovat objekty je totiž nutné shromáždit dostatečně velkou trénovací množinu obrázků. Navíc musí každý obrázek obsahovat masku s přesně

vyznačeným objektem na obrázku. Výsledný model ovšem dokáže přesně určit souřadnice celého objektu na obrázku [41].

Existuje spousta dalších příkladů počítačového vidění. Dalším příkladem může být titulkování obrázků, kdy model umělé inteligence přesně popíše obrázek a objekty na něm. Odvětví počítačového vidění se ovšem neustále vyvíjí a příklady počítačového vidění jsou neustále zdokonalovány.



Obrázek 5 - Příklady počítačového vidění
Zdroj: Logic could be learned from images [42]

4 Aplikace

Projekt byl vytvořen pomocí vývojového prostředí Android Studio. Android studio je vývojové prostředí založené na prostředí IntelliJ. Vývojové prostředí IntelliJ poskytuje sadu výkonných nástrojů pro usnadnění vývoje aplikací. Prostředí IntelliJ bylo vyvinuto společností JetBrains [26].

V projektu si demonstrujeme práci s již vyvinutými modely umělé inteligence poskytnutým pro vzdělávací účely společností TensorFlow. Demonstrujeme si klasifikaci a detekci objektů. Dále bude čtenář také seznámen s postupem trénování vlastního TensorFlow modelu pro detekci objektů.

4.1 Kotlin

Aplikace je vytvořena pomocí programovacího jazyka Kotlin. Kotlin je programovací jazyk vyvinut v roce 2011. Nejvíce populární se stal Kotlin v roce 2017, když společnost Google označila Kotlin za ideální programovací jazyk pro vývoj Android aplikací. Díky interoperabilitě s programovacím jazykem Java může Kotlin využívat služby a pokročilé funkce Javy [43].

Jako v každém programovacím jazyku, jsou základní stavební prvky v programovacím jazyku Kotlin proměnné. Proměnná slouží k uchování hodnoty, nebo odkazu na celý objekt. V Kotlinu vytvoříme proměnnou pomocí klíčového slova var. Kotlin je typovaný jazyk, a proto má každá proměnná přiřazený svůj datový typ, který udává, jaké hodnoty lze do proměnné uložit. Tento datový typ je ale většinou doplněn kompilátorem. V kotlinu lze vytvořit i konstanty, tedy proměnné, jejichž hodnotu již nelze změnit, pomocí klíčového slova val [43].

```
val pi: Double = 3.14159
var celeCislo: Int = 150
var jmeno = "Petr"
val student = Person(name = jmeno)
```

K rozhodování lze v kotlinu využít podmínkové výrazy. Nejzákladnější podmínkový výraz je výraz s názvem if. Tento výraz dovolí programu provést určitý kód pouze za předpokladu, že je splněna nějaká podmínka. Další výraz využívaný k určitému rozhodování je výraz s názvem when. Výraz when se využívá očekáváme-li více možných rozhodnutí [43].

```
if (animal == "Cat" || animal == "Dog") {
    println("Animal is a house pet.")
} else {
    println("Animal is not a house pet.")
}

val numberName = when (number) {
  2 -> "two"
  4 -> "four"
  6 -> "six"
  8 -> "eight"
  10 -> "ten"
}
```

Chceme-li uchovat větší množství hodnot v jedné proměnné, využijeme pole. Pole lze chápat jako kontejnery, které umožňují uložení několika hodnot pohromadě. Pole uchovávají v paměti prvky daného typu hodnot, který do pole ukládáme. Prvky v poli jsou indexované. To znamená že každý prvek v poli má přiřazený speciální identifikátor, pomocí kterého lze získat a pracovat s určitým prvkem v poli na jakékoliv pozici [43].

Chceme-li v Kotlinu provést kód několikrát, využijeme cykly. Pomocí cyklu sdělíme kompilátoru, aby provedl následující blok kódu vícekrát. Cykly dělíme na cykly s pevným počtem opakování a na cykly s volným počtem opakování. Cyklus s pevným počtem opakování se v Kotlinu zapíše pomocí klíčového slova for. Tento cyklus využijeme například chceme-li naplnit vytvořenou matici čísly, nebo přečíst každý prvek v poli hodnot. Pokud nevíme kolikrát přesně se má daný bloku kód opakovat využijeme cyklus while. Tento cyklus lze využít například při čtení prvků ze souboru, u kterého neznáme jeho velikost [43].

```
var sum = 0
for (i in 1..10) {
    sum += i
}
for (item in array) {
    println(item)
}

var line = file.readLine()
while (line != EOF) {
    line = file.readLine()
}
```

Dalším důležitým prvkem jazyka Kotlin jsou třídy. Třídy jsou základní stavební kameny objektově orientovaného programování. Objektově orientované programování je styl programování, kde objekty uchovávají data a provádějí určité akce. Pomocí tříd tyto objekty vytváříme. Ve třídách představují data proměnné neboli vlastnosti třídy, a akce se provádějí pomocí metod třídy. Metody jsou bloky kódu, které vykonávají nějakou činnost. Obvykle jsou uvnitř tříd a lze je volat jejich jménem. Metody mohou obsahovat vstupní parametry, které se vloží do těla metody, kde tyto parametry může kód v metodě upravit. Třídy lze

v Kotlinu vytvořit pomocí klíčového slova class. Navíc Kotlin poskytuje i speciální klíčové slovo data, které lze zapsat před class. Pomocí tohoto slova vytvoří vývojové prostředí vlastnosti třídy a konstruktor automaticky za nás. Proměnné neboli vlastnosti třídy zapíšeme do závorky za název třídy. Pomocí této závorky se vytvoří také konstruktor, pomocí kterého lze vytvořit instanci třídy [43].

```
data class Car(var brand: String, var model: String, var year: Int) {
    fun drive() {
        println("Wrooom!")
    }
    fun changeYear(var year: Int) {
        this.year = year
    }
}

fun main() {
    val c1 = Car("Ford", "Mustang", 1969)

    c1.drive()
    c1.changeYear(2005)
}
```

Speciálním typem třídy je třída s názvem enum. Tato třída slouží k definici sady konstantních hodnot. Tyto hodnoty lze považovat za konstanty. V Kotlinu lze v enum třídě uchovávat i proměnnou pro každou konstantní hodnotu. V kódu lze následně zobrazit tuto proměnnou u určité konstanty třídy enum [43].

```
enum class DetectionTypes(val type: String) {
    PHOTO_DETECTION("photoDetection"),
    REAL_TIME_DETECTION("realTimeDetection")
}
```

V Android studiu se aplikace vytváří pomocí skupiny několika obrazovek, či pláten. Tyto obrazovky se nazývají aktivity. Při vytvoření nové aktivity v Android Studiu vývojové prostředí vygeneruje připojený soubor typu XML. V tomto souboru vývojář definuje ovládací prvky zobrazené v okně aplikace. Aktivita je třída implementující metody z životního cyklu aktivit. Nejdůležitější je překrytá metoda s názvem onCreate(), která se zavolá při vytvoření a spuštění aktivity. Do těla třídy aktivity lze následně implementovat kód a algoritmus aplikace [26].

Chceme-li vytvořit aplikaci využívající asynchronní programování musíme znát také mechanismus korutin. Korutiny jsou speciální mechanismy v jazyce Kotlin. Jsou vytvořeny tak, aby umožňovaly efektivní a jednoduché vytváření vláken a paralelních úloh. Pomocí vláken docílíme toho, že naše aplikace nebude blokována žádným voláním, či zdlouhavým čtením například souborů. Chceme-li spustit úlohu v kontextu celé aplikace, využijeme proto globální oblast. Tato oblast se nazývá GlobalScope a umožňuje běh vlákna, dokud není uzavřena celá aplikace. Můžeme se setkat například i s názvem CoroutineScope. Tento název umožňuje vytvoření korutiny, která je ukončena při uzavření zobrazované aktivity.

Načítáme-li například data ze souboru v jiné aktivitě a vrátíme se zpátky na hlavní aktivitu, je načítání ve vedlejší aktivitě ukončeno [43].

```
GlobalScope.launch {
    val data = loadDataFromFile(fileName)
    println("Data ze souboru: $data")
}

println("Čekání na načtení dat ze souboru...")
```

4.2 Struktura projektu

Náš projekt bude vytvořen pomocí návrhového vzoru Model-View-Controller. Kvůli přehlednosti si rozdělíme soubory obsahující zdrojový kód aplikace do tří složek s názvy Model, View a Controller. Mimo tyto složky umístíme hlavní aktivitu, která se spustí při otevření aplikace. Z této hlavní aktivity bude mít uživatel možnost otevírat nové aktivity. Tyto aktivity budou umístěny ve složce Views. Kvůli oddělení složitějších algoritmů vytvoříme aktivitám vlastní třídu typu Controller. Tato třída bude zprostředkovávat složitější algoritmy, například algoritmus detekce objektů na obrázku. Z Controllerů i aktivit lze využívat a odkazovat se na třídy ve složce Model.

Kromě hlavní složky se zdrojovým kódem aplikace obsahuje struktura projektu i další složky. Jedná se například o složky s konfiguračními soubory build.gradle, nebo AndroidManifest.xml. Další důležitá složka ve struktuře projektu je složka s názvem res. Tato složka obsahuje například soubory s použitými barvami a texty v aplikaci, nebo vygenerované XML soubory, které definují komponenty vykreslené do aktivit. Vytvoříme si také složku, do které budeme ukládat TensorFlow modely a textové soubory obsahující názvy tříd. Tyto třídy u TensorFlow modelů označují názvy detekovatelných objektů na obrázcích.

```
app/
├─ manifests/
│   ├─ AndroidManifest.xml
├─ java/
│   ├─ com.example.objectclassification/
│   │   ├─ Controller/
│   │   │   ├─ DatabaseScreenController.kt
│   │   │   ├─ MainScreenController.kt
│   │   │   ├─ MoneyDetectionController.kt
│   │   │   ├─ PhotoDetectionController.kt
│   │   │   ├─ RealTimeDetectionController.kt
│   │   ├─ Model/
│   │   │   ├─ DetectionTypes.kt
│   │   │   ├─ DrawableObject.kt
│   │   │   ├─ ImageDatabase.kt
│   │   │   ├─ ImageDatabaseObject.kt
│   │   │   ├─ ImageDatabaseObjectDAO.kt
│   │   │   ├─ imageDatabaseObjectSingleton.kt
│   │   │   ├─ ImageObjectAdapter.kt
│   │   │   ├─ imageSingleton.kt
│   │   │   ├─ IPhoto.kt
│   │   │   ├─ Photo.kt
│   │   ├─ View/
│   │   │   ├─ DatabaseScreen.kt
│   │   │   ├─ ItemDetailsScreen.kt
│   │   │   ├─ MoneyDetectionScreen.kt
│   │   │   ├─ OverlayView.kt
│   │   │   ├─ PhotoDetectionScreen.kt
│   │   │   ├─ RealTimeDetectionScreen.kt
│   │   │   ├─ SaveToDatabaseScreen.kt
│   │   ├─ MainActivity.kt
├─ res/
│   ├─ values/
│   │   ├─ colors.xml
│   │   ├─ strings.xml
│   ├─ layout/
│   │   ├─ activity_main.xml
├─ ml/
│   ├─ DetectMoney.tflite
│   ├─ moneyLabels.txt
│   ├─ PhotoModel.tflite
│   ├─ photoLabels.txt
│   ├─ RealTimeModel.tflite
│   ├─ realTimeLabels.txt
Gradle Scripts/
├─ build.gradle
```

4.2.1 MVC

Pro vytvoření aplikace pro detekci objektů byl využit návrhový vzor Model-View-Controller. Tento návrhový vzor se využívá převážně pro vývoj softwarových aplikací s grafickým uživatelským rozhraním. Hlavním cílem tohoto modelu je oddělit logiku aplikace od zobrazovaných dat.

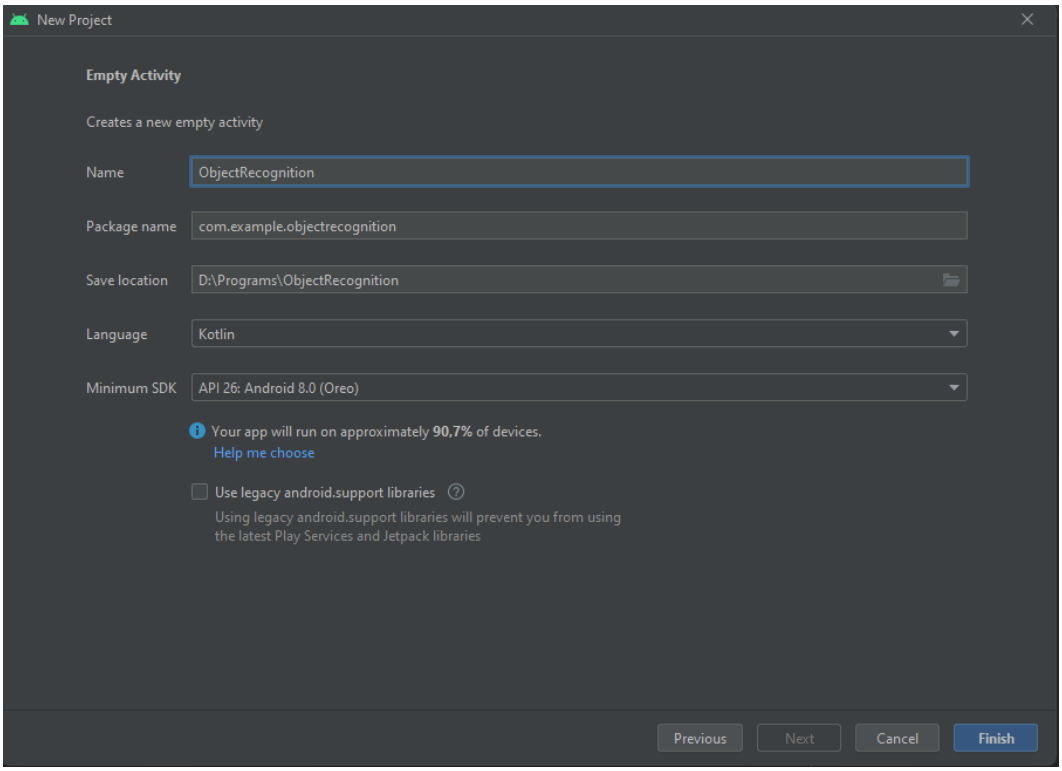
Tento návrhový vzor rozděluje většinou data do tří částí, pojmenovaných Model, View a Controller. Část s názvem model obsahuje třídy, které specifikují reprezentované informace a data, s nimiž aplikace pracuje. Třídy v části View jsou většinou grafické

a nazývají se pohledy. Zobrazují tedy určitý výstup uživateli. Tyto pohledy převádí data reprezentována v modelu do vhodné podoby k vizualizaci v okně aplikace. V projektu budeme považovat aktivity jako pohledy ve skupině View. Poslední část Controller je označována také jako řadič. Třídy typu Controller reagují na události, manipulují s pohledy v části View a zajišťují změny v třídách Modelu [25].

4.3 Vytvoření projektu

Jako první si v aplikaci Android Studio vytvoříme nový projekt. Zvolíme projekt s prázdnou aktivitou. Aktivity v Android Studiu označují komponentu, se kterou interaguje uživatel aplikace. Aktivity zpracovávají vstupy od uživatele, zobrazují UI (user interface) komponenty (například tlačítka) a řídí svůj životní cyklus. Aktivity jsou většinou celoplošná okna zobrazená uživateli, ale existují i menší fragmenty, které jsou menší a na jedné obrazovce jich můžeme nalézt větší množství.

Před vytvořením projektu zvolíme název naší aplikace, umístění souborů na našem počítači a programovací jazyk. Programovací jazyk určuje, který jazyk bude použit při kompilaci a sestavení aplikace. V základu Android Studio nabízí jazyk Kotlin a jazyk Java. Zvolíme programovací jazyk Kotlin a také minimální verzi operačního systému Android, na kterém poběží naše aplikace. Zvolíme verzi systému Android 8.0 s názvem Oreo a vytvoříme projekt.



Obrázek 6 - Vytváření projektu
Zdroj: Vlastní zpracování

Následně otevřeme konfigurační soubor s názvem build.gradle. Tento soubor definuje základní aspekty pro vytvoření aplikace. Nalezneme zde pluginy využívané v našem projektu, název aplikace, základní SDK konfiguraci a závislosti (anglicky „dependencies“).

Zkratka SDK označuje sadu vývojových nástrojů (anglicky „Software development kit“), která umožňuje programátorovy vytvářet aplikace pro určité softwarové balíčky, frameworky, počítačové systémy a jiné platformy. Otevřeme soubor s názvem build.gradle a do bloku s názvem dependencies přidáme potřebné závislosti, které využijeme v tomto projektu. Do bloku přidáme balíčky pro podporu Tensorflow knihovny, která nám umožní pracovat s Tensorflow modely. Přidáme také balíčky pro práci s Room databází, která bude v aplikaci ukládat detekované obrázky a objekty. Nakonec přidáme implementaci RecyclerView komponenty, která nám umožní zobrazovat záznamy z databáze v aktivitě.

```
dependencies {
    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
    implementation 'org.tensorflow:tensorflow-lite-support:0.1.0'
    implementation 'org.tensorflow:tensorflow-lite-metadata:0.1.0'
    implementation 'org.tensorflow:tensorflow-lite-gpu:2.3.0'
    implementation 'androidx.recyclerview:recyclerview:1.3.2'
    implementation 'com.google.android.material:material:1.11.0'
    kapt 'androidx.room:room-compiler:2.4.3'
    implementation 'androidx.room:room-runtime:2.4.3'
    implementation 'androidx.room:room-ktx:2.4.3'
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.5.1'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.5'
    androidTestImplementation 'androidx.test.espresso:espresso-
core:3.5.1'
}
```

Po změně konfiguračního souboru musíme také upravit soubor s názvem „AndroidManifest.xml“, který se nachází ve složce s názvem manifests. Tento soubor poskytuje základní informace o aplikaci systému Android. Soubor obsahuje například název aplikace, ikonu aplikace, která se zobrazí v systému Android, nebo seznam komponent a oprávnění. Do tohoto souboru vložíme bloky s názvem uses-feature a uses-permission, které nám umožní práci s hardwarem mobilního zařízení. Konkrétně bude v projektu potřeba umožnit přístup ke kameře mobilního zařízení za účelem pořizování fotografií objektů pro následnou detekci.

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

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

4.4 Hlavní aktivita

Hlavní aktivita se spouští při zkompilování, nainstalování a spuštění aplikace. V projektu bude sloužit jako hlavní okno aplikace a rozcestník mezi dalšími okny. V souboru typu XML (zkratka pro „Extensible Markup Language“), který se vygeneroval společně s hlavní aktivitou, můžeme do okna přidávat ovládací prvky a komponenty. Jako první vytvoříme komponentu s názvem LinearLayout (česky „lineární rozložení“), které nastavíme vlastnost s názvem orientation na hodnotu vertical. Tato komponenta nám zaručí vertikální orientaci prvků uvnitř lineárního rozložení. Do okna lze přidat také vodící linky, které nám po

navázání dalších komponent na ně zaručí odsazení komponent od okraje zobrazovaného okna. Do vertikálního rozložení vložíme další komponentu s názvem TableLayout (česky „tabulkové rozložení“). Tabulkové rozložení nám umožní na každý TableRow (česky „řádek tabulky“) vložit komponentu zarovnanou z hlediska všech řádků tabulky. V případě naší aplikace nám zaručí tabulka stejnou velikost tlačítek nezávisle na textu v tlačítkách. Do tohoto tabulkového rozložení vložíme čtyři tlačítka, které v našem projektu budou sloužit jako rozcestníky do dalších aktivit aplikace. V těchto dalších aktivitách budou představeny různě možnosti implementace modelů pro klasifikaci a detekci objektů.

```
<LinearLayout>
    <TextView
        android:text="@string/app_title"/>
    <Space/>
    <TableLayout>
        <TableRow>
            <Button
                android:id="@+id/btnTakePicture"
                android:text="@string/btn_photo_detection" />
        </TableRow>

        <!--
            Další tři tlačítka
        -->

    </TableLayout>
</LinearLayout>
```

Každá komponenta, která interaguje s uživatelem musí mít nastavený identifikátor (zkráceně „id“), aby byla komponenta viditelná pro programovací jazyk Kotlin. Je nutné tedy v projektu nastavit tlačítkům jedinečné identifikátory, na které se bude v kódu aplikace odkazovat. Pro udržení konzistence Android Studio doporučuje vytvoření souborů typu XML pro manipulaci s hodnotami barev, textu a velikosti komponent. Tlačítku tedy nastavíme vlastnost „text“ na hodnotu „string/btn_photo_detection“, která odkazuje na záznam v souboru string.xml.

Po dokončení editace souboru XML, který definuje komponenty v hlavní aktivitě je potřeba deklarovat a implementovat logiku aplikace. K tomuto účelu slouží soubor s koncovkou KT (typ souboru obsahující kód v programovacím jazyce Kotlin), který nese název hlavní aktivity. Soubor se může například jmenovat „MainActivity.kt“. Pokud tento soubor otevřeme zjistíme, že se jedná o třídu, která rozšiřuje třídu AppCompatActivity. Naše aktivita je tedy dědičnou třídou AppCompatActivity, která je též dědičnou třídou základní třídy s názvem Activity.

Při vytvoření třídy Android Studio automaticky překryje (anglicky „override“) metodu s názvem „onCreate()“ z dědičné třídy AppCompatActivity. Jedná se o jednu z několika metod implementujících životní cyklus aplikace. Třída MainActivity v této překryté metodě volá metodu „onCreate()“ nadřazené rodičovské třídy a předává parametr s názvem „savedInstanceState“, který ukládá data a stav aktivity. Následně je nutné nastavit pomocí metody „setContentView()“ správný XML soubor s ovládacími prvky a komponenty.

```
package com.example.objectclassification.View

import androidx.appcompat.app.AppCompatActivity
```

```
import android.os.Bundle
import com.example.objectclassification.R

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

V překryté metodě „onCreate()“ následně inicializujeme tlačítka, která byla definována v přiřazeném XML souboru. Inicializaci provedeme pomocí metody s názvem „findViewById<>()“. Jedná se o metodu, které lze přiřadit typový parametr, v našem případě „Button“. To znamená, že komponenta nalezená touto metodou musí být typu Button. V parametru této metody nalezneme díky výrazu „R.id.*id_komponenty*“ správný identifikační název požadovaného tlačítka. „R“ ve výrazu značí zdroje aplikace (anglicky „resources“). Po inicializaci všech tlačítek v kódu aplikace vytvoříme proměnou „intent“. Intent je objekt, který obsahuje popis operace, která má být vytvořena. Intent slouží k otevírání nových aktivit a k přenosu primitivních dat mezi aktivitami. Nakonec nastavíme každému tlačítku vlastní posluchač události na kliknutí (anglicky „click listener“) pomocí „setOnClickListener“. „SetOnClickListener“ je rozhraní (anglicky „interface“), které definuje metodu „OnClick()“, která je vyvolána po kliknutí na tlačítko. Po kliknutí na dané tlačítko se vytvoří nový intent, kterému se pomocí dvou vstupních parametrů přiřadí kontext právě běžící aktivity a odkaz na třídu. Pomocí .java je do parametru Intentu vložena instance Java třídy, která odpovídá KClass (třída Kotlin). Následně pomocí metody „startActivity(intent)“ vytvoříme a otevřeme novou aktivitu. Každému tlačítku nastavíme takový intent, který bude odpovídat otevření správného okna aplikace. Všechny další aktivity budou uloženy ve složce s názvem View, pro zachování MVC architektury.

```
val btnPhotoDetection = findViewById<Button>(R.id. btnPhotoDetection)
val btnRealTimeDetection
=findViewById<Button>(R.id.btnRealTimeDetection)
val btnTrainedModel = findViewById<Button>(R.id.btnTrainedModel)
val btnDatabase = findViewById<Button>(R.id.btnShowDatabase)

var intent = Intent()

btnPhotoDetection.setOnClickListener(){
    intent = Intent(this,PhotoDetectionScreen::class.java)
    startActivity(intent)
}
/*
Nastavení dalších tří tlačítek
*/
```

Po nastavení událostí na kliknutí u všech tlačítek máme hotové hlavní okno aplikace. Hlavní okno obsahuje čtyři tlačítka umožňující uživatelskou interakci s aplikací. Toto hlavní okno nyní slouží jako rozcestí mezi aktivitami demonstrující použití různých TensorFlow modelů. Každé tlačítko spouští příslušnou aktivitu. Z každé aktivity je možný návrat zpátky do hlavního okna aplikace pomocí tlačítka s názvem cancel, nebo zpět. Nyní se seznámíme s implementací aktivit demonstrující práci s TensorFlow modely pro detekci a klasifikaci objektů.

Obrázek 7 - Ukázka hlavního okna aplikace
Zdroj: Vlastní zpracování

4.5 Klasifikace objektu z fotky

Pro znázornění správného fungování klasifikace objektů si stáhneme již vytrénovaný model s názvem mobilenet_v1_1.0_224_quant_and_labels z oficiálních stránek TensorFlow (dostupné z: https://www.tensorflow.org/lite/examples/image_classification/overview[45]). Tento model dokáže klasifikovat 1000 objektů. Vstupem modelu je bajtový buffer. Buffer je označení pro dočasné paměťové úložiště. Bajtový buffer obsahuje data v podobě sekvence bajtů. V našem konkrétním případě je vstupem bajtový buffer, který obsahuje zkoumaný obrázek převedený na sekvenci bajtů. Výstupem je pole hodnot s identifikátorem třídy a pravděpodobností výskytu třídy na obrázku. Pro implementaci modelu importujeme model a textový soubor s názvy tříd do projektu. Modely a textové soubory můžeme ukládat do samostatné složky ve zdrojových souborech modelu.

První tlačítko na hlavní obrazovce zobrazí novou aktivitu, jejímž úkolem je klasifikace objektu na vyfocené fotografii. K vytvoření nové aktivity klikneme pravým tlačítkem na adresář v projektu, kam chceme novou aktivitu uložit, a ze seznamu zvolíme možnost „new“. V rozbalené nabídce zvolíme další možnost Activity a poté Empty Activity. Novou aktivitu si následně pojmenujeme. Jméno aktivity je použito v tlačítku v hlavní aktivitě a musí být proto stejné, abychom mohli aktivitu z hlavního okna otevřít po stisknutí tlačítka. Aktivitě také necháme Android Studiem vygenerovat XML layout soubor pro vložení ovládacích komponent. Do XML souboru vložíme komponenty uživatelského rozhraní pro klasifikaci objektů. Důležité komponenty jsou TextView, ImageView a tlačítka. TextView využijeme pro zobrazení výsledné předpovědi klasifikace. ImageView slouží k zobrazení zkoumané fotky a tlačítka zprostředkují uživatelskou interakci s aplikací. Tyto prvky můžeme do XML souboru přidat například opět ve vertikálně zarovnaném rozložení. Jako první vložíme ImageView, které zobrazí vyfocený obrázek z fotoaparátu mobilního zařízení. Hned pod ImageView vložíme TextView, které zobrazí výstup klasifikačního modelu. Následně vložíme do tabulkového rozložení tři tlačítka. První tlačítko obstará otevření aplikace

fotoaparátu a vyfocení fotografie. Druhé tlačítko slouží k uložení obrázku a predikce do databáze a třetí tlačítko uzavře okno pro klasifikaci obrázku a program se vrátí zpět na hlavní aktivitu aplikace.

```
<LinearLayout>
    <ImageView
        android:id="@+id/imgViewBitmap"/>
    <Space/>
    <TextView
        android:id="@+id/txtPredictions"/>
    <Space/>
    <TableLayout
        <TableRow
            <Button
                android:id="@+id/btnTakePicture"/>
        </TableRow>
        <TableRow>
            <Button
                android:id="@+id/btnSaveToDB"/>
        </TableRow>
        <TableRow>
            <Button
                android:id="@+id/btnCancel"/>
        </TableRow>
    </TableLayout>
</LinearLayout>
```

Vytvořená aktivita obsluhující část aplikace pro klasifikaci obrazu má podobnou strukturu jako aktivita hlavní. Nová aktivita, stejně jako aktivita hlavní, je potomkem třídy AppCompatActivity a překrývá onCreate metodu. Na rozdíl od hlavní aktivity bude tato nová aktivita obsahovat důležité vlastnosti ve formě privátních proměnných. Tyto privátní proměnné slouží k volání té stejné inicializované hodnoty, či reference v různých částech a metodách třídy. Jako první si inicializujeme konstantu pro požadovaný počet fotografií vyfocených aplikací fotoaparát na hodnotu 1. Následně si pomocí klíčového slova lateinit deklarujeme proměnné pro manipulaci s komponenty v aktivitě. Klíčové slovo lateinit umožňuje vyhnout se inicializaci vlastnosti či proměnné při vytváření objektu. Inicializace lateinit proměnných proběhne až později v kódu. V případě proměnných pro práci s komponenty v aktivitě se může klíčové slovo lateinit použít například pokud programátor nechce komponentu vyhledávat pokaždé když s ní pracuje. Poslední lateinit proměnná bude sloužit k pozdější inicializaci controlleru.

```
class PhotoDetectionScreen() : AppCompatActivity() {
    private val REQUEST_IMAGE_CAPTURE: Int = 1
    private lateinit var imgView: ImageView
    private lateinit var txtPredictions: TextView
    private lateinit var photoDetectionController:
                                            PhotoDetectionController

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_photo_detection_screen)
    }
}
```

V metodě onCreate inicializujeme komponenty ImageView a TextView. Proměnné pro tyto komponenty jsme deklarovali v těle třídy této aktivity mimo metodu onCreate. Pro každé tlačítko deklarujeme vlastní proměnnou. Proměnné pro tlačítko inicializujeme správnou komponentu pomocí metody findViewById. Inicializujeme si také proměnnou pro controller. Controller bude sloužit k reagování na požadavek o detekci. Následně nastavíme tlačítku obstarávající pořízení snímku posluchač na kliknutí. Po stisknutí tlačítka se zavolá patřičná metoda pro pořízení fotografie z kamery mobilního zařízení.

```
imgView = findViewById<ImageView>(R.id.imgViewBitmap)
val btnTakePicture = findViewById<Button>(R.id.btnTakePicture)
val btnSaveToDB = findViewById<Button>(R.id.btnSaveToDB)
val btnCancel = findViewById<Button>(R.id.btnCancelPhotoDetection)
txtPredictions = findViewById<TextView>(R.id.txtPredictions)

photoDetectionController = PhotoDetectionController()

btnTakePicture.setOnClickListener(){
    takeAPicture()
}
```

Nyní implementujeme základní logiku pro snímání fotografií z kamery mobilního zařízení. Nejdříve v metodě takeAPicture, kterou spouští tlačítko při kliknutí uživatele, zkontrolujeme oprávnění pro kameru. Nejprve se zkontroluje, zda má aplikace povolení k přístupu do fotoaparátu. Pokud povolení není již uděleno, provede se žádost o povolení od uživatele pomocí volání requestPermissions. Pokud bylo oprávnění pro využívání aplikace fotoaparátu od uživatele uděleno pokračuje program vytvořením nového intentu, který otevře nové okno s aplikací fotoaparátu. Následně spustíme intent, pomocí metody startActivityForResult, který otevře novou aktivitu, která po svém ukončení vrátí výsledek ve formě požadovaných dat. Tato nová aktivita obsahuje aplikaci fotoaparát, která umožní pořídit fotografii a vrátit jí zpět do projektu jako výsledek aktivity.

```
fun takeAPicture() {
    if (checkPermisions()){
        Intent(MediaStore.ACTION_IMAGE_CAPTURE).also {
takePictureIntent ->
            takePictureIntent.resolveActivity(packageManager)?.also {
                startActivityForResult(takePictureIntent,
                    REQUEST_IMAGE_CAPTURE)
            }
        }
    }
}

private fun checkPermisions(): Boolean {
    if(ContextCompat.checkSelfPermission(this,
               android.Manifest.permission.CAMERA) !=
               PackageManager.PERMISSION_GRANTED){
                   requestPermissions(
                       arrayOf(android.Manifest.permission.CAMERA),101)
                   return false
    }
    return true
}
```

Následně překryjeme metodu onActivityResult ve které implementujeme vlastní logiku pro zpracování fotografie získané z aplikace fotoaparátu mobilního zařízení. Pokud proběhlo získání fotografie úspěšně, zobrazíme fotografii v komponentě ImageView. Následně vytvoříme instanci modelu pro klasifikaci objektů na fotografii. Do inicializovaného controlleru vložíme jako parametr metody zkoumaný obrázek a Tensorflow model. V controlleru implementujeme algoritmus pro klasifikaci objektů na obrázku a model v aktivitě ukončíme zavoláním metody close. Nakonec zobrazíme výstup z controlleru uživateli v komponentě TextView.

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode ==
RESULT_OK) {
        var imageBitmap = data?.extras?.get("data") as Bitmap
        imgView.setImageBitmap(imageBitmap)

        val model = PhotoModel.newInstance(this)
        photoDetectionController.classifyImage(imageBitmap,model)

        model.close()
        updatePredictions()
    }
}
private fun updatePredictions() {
    txtPredictions.setText(
        photoDetectionController.predictions(
            assets.open("photoLabels.txt")))
}
```

Pro implementaci algoritmu pro klasifikaci objektů na obrázku si ve složce Controller vytvoříme třídu controlleru pro obsluhu požadované aktivity. Ve třídě controlleru implementujeme metodu pro klasifikaci objektů s parametry typu ImageBitmap a Model. V metodě inicializujeme třídu Photo, která bude uchovávat samotný zkoumaný obrázek a výsledné klasifikace tříd na obrázku. Následně změníme obrázku velikost na požadovanou výšku a šířku. Vytvoříme instanci třídy TensorBuffer, kterou využijeme jako vstupní data pro Tensorflow model. TensorBuffer je dočasné paměťové úložiště sloužící jako vstup a výstup Tensorflow modelů a uchovává data ve formě tenzorů. Tenzor je primární datová struktura v TensorFlow programech. Jedná se o n-dimenzionální datovou strukturu pro uchování dat [2]. Do TensorBufferu vložíme zkoumaný obrázek ve formě bufferu, ve kterém je uložen obrázek ve formě sekvence bajtů. Pomocí metody s názvem process v proměnné Tensorflow modelu spustíme klasifikaci. Po klasifikování objektu na obrázku uložíme výstup do proměnné. Výstup převedeme na TensorBuffer a následně na pole obsahující desetinná čísla. Tento výsledek klasifikace uložíme do třídy Photo. Výslednou klasifikaci ze třídy Photo lze v hlavní aktivitě volat z controlleru pomocí metody s názvem predictions.

```
lateinit var photo: Photo

fun classifyImage(imageBitmap: Bitmap, model: PhotoModel) {
    photo = Photo(imageBitmap)
```

```
    var resized = imageBitmap.copy(Bitmap.Config.ARGB_8888, true)
    resized = Bitmap.createScaledBitmap(resized, 224, 224, true)

    val model = model

    val tensorBuffer = TensorBuffer.createFixedSize(
                        intArrayOf(1,224,224,3), DataType.UINT8)
    var tImage = TensorImage(DataType.UINT8)
    tImage.load(resized)
    var byteBuffer = tImage.buffer
    tensorBuffer.loadBuffer(byteBuffer)

    val outputs = model.process(tensorBuffer)
    val outputBuffer = outputs.outputFeature0AsTensorBuffer

    photo.predictionsImage = outputBuffer.floatArray
}

fun predictions(labels: InputStream): String {
    return photo.getStyledPredictions(labels)
}
```

Ve třídě Photo implementujeme algoritmus pro získání výsledných klasifikací objektů na obrázku. Nejdříve přečteme soubor se všemi třídami, které umí TensorFlow model klasifikovat. Všechny třídy si uložíme do proměnné typu pole, které bude obsahovat textové hodnoty s názvy klasifikovatelných tříd. Třída Photo má uložené výsledky ze zpracování detekce obrázku v proměnné predictionsImage. Tyto výsledky si převedeme na zřetězený seznam. Pro získání čtyř nejpřesnějších výsledků si vytvoříme cyklus s pevným počtem opakování. V těle cyklu vytvoříme proměnné pro uchování nejvyšší nalezené přesnosti a uchování pozice třídy s nejvyšší pravděpodobností výskytu. Tyto proměnné inicializujeme ze začátku na hodnotu 0. Následně musíme vytvořit nový cyklus, který prohlédne všechny prvky v proměnné výsledků z klasifikace. Cyklus průběžně porovnává navštívený prvek z proměnné výsledků klasifikace s dříve vytvořenou proměnnou udávající nejvyšší nalezenou klasifikaci. Pokud má prvek v poli výsledků vyšší hodnotu než proměnná uchovávající nejvyšší nalezenou klasifikaci, vloží se nově nalezená nejvyšší hodnota klasifikace do proměnné. Jakmile cyklus navštíví všechny prvky v proměnné s výsledky obsahují dříve definované proměnné desetinné číslo udávající maximální pravděpodobnost výskytu třídy na obrázku a pozici třídy ve zkoumaném seznamu výsledků. Na konci metody vrátíme pomocí klíčového slova return výsledek. Výsledek je ve formátu čtyř záznamů, z nichž každý je na novém řádku. Záznam obsahuje název klasifikovatelné třídy, který lze získat z pole názvů klasifikovatelných tříd na pozici třídy s nejvyšší pravděpodobností výskytu. Záznam také obsahuje pravděpodobnost správného určení třídy v procentech.

```
interface IPhoto {
    var image: Bitmap
    var predictionsImage: FloatArray
    fun getStyledPredictions(labels: InputStream): String
}
```

```
override fun getStyledPredictions(labels: InputStream): String {
    var result: String = ""
    val inputStream = labels
    var listOfLabels = ArrayList<String>()
```

```
    inputStream.bufferedReader().use { reader ->
        reader.readLines().forEach() { line ->
            listOfLabels.add(line)
        }
    }
    var arrayOfConfidences = predictionsImage.toMutableList()
    for (x in 0..3) {
        var maxConfidence = 0.0f
        var maxConfidenceIndex = 0

        for ((index, value ) in arrayOfConfidences.withIndex()){
            if (value > maxConfidence){
                maxConfidence = value
                maxConfidenceIndex = index
            }
        }
        var resultConfidence = maxConfidence
        if (resultConfidence > 100.0f) {
            resultConfidence = 100.0f
        }
        result += listOfLabels.get(maxConfidenceIndex) + " : " +
                  resultConfidence + "%\n"
        arrayOfConfidences[maxConfidenceIndex] = 0f
    }
    return result
}
```

Nyní lze v okně, sloužícím k demonstraci klasifikace objektů, zobrazit výsledky z controlleru pomocí metody s názvem updatePredictions.

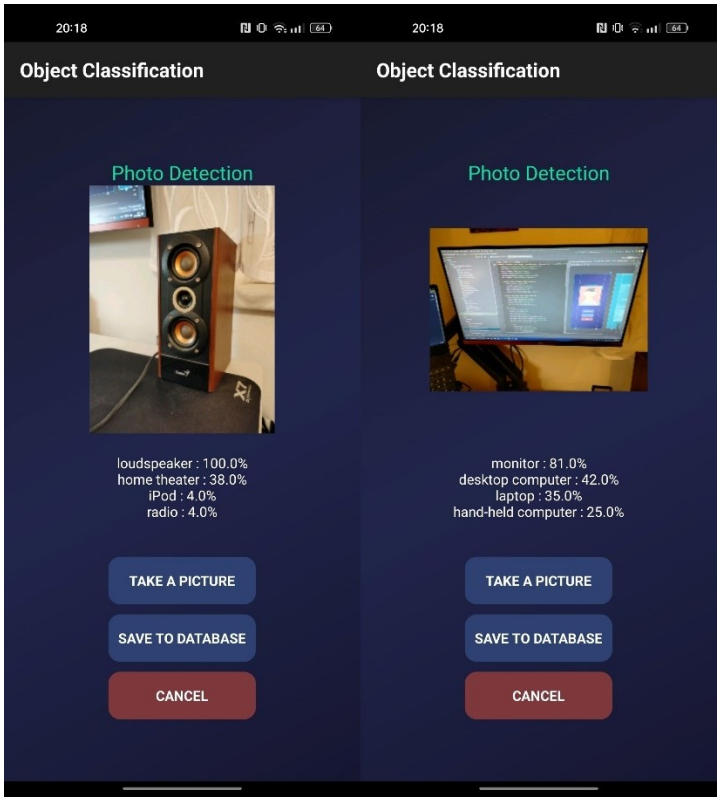
```
private fun updatePredictions() {
    txtPredictions.setText(
        photoDetectionController.predictions(
            assets.open("photoLabels.txt")))
}
```

Aplikace nyní zobrazuje výsledné klasifikace objektů v textové komponentě po pořízení fotografie. Nyní si definujeme ukládání do vytvořené databáze. Tlačítku pro uložení do databáze přidáme naslouchač na událost kliknutí. Po kliknutí na tlačítko nejdříve zkontrolujeme, zda byla pořízena nějaká fotografie a byla již tedy provedena klasifikace objektů. Pokud byla již klasifikace provedena zjistíme název objektu s nejvyšší pravděpodobností výskytu na obrázku. Název objektu zjistíme pomocí stejného výpisu z controlleru, který poskytl čtyři nejpravděpodobnější klasifikace. Z tohoto výpisu získáme první název pomocí metody split, která vrátí první název a odřízne zbytek textu. K otevření okna sloužícího k uložení záznamu do databáze inicializujeme nový intent, který odkazuje na patřičnou třídu. Do intentu vložíme pod identifikátor imageName také název nejpravděpodobnější klasifikace. Do intentu také vložíme správný typ detekce. Tento typ umožní budoucí filtrování záznamů v databázi. Nakonec uložíme obrázek do globální proměnné jedináčka a spustíme aktivitu pomocí vytvořeného intentu. V nové aktivitě již uživatel pomocí formuláře uloží nový záznam do databáze. V této aktivitě se také zobrazí obrázek z globální proměnné.

Poslední tlačítko v okně pro klasifikaci objektů slouží k ukončení aktivního okna a návratu na hlavní okno aplikace. Nastavíme tedy tlačítku naslouchač na událost kliknutí, ve které zavoláme metodu finish, která ukončí aktivitu a zavře okno.

```
btnSaveToDB.setOnClickListener(){
    if (!photoDetectionController.isPhotoInitialized()) {
        Toast.makeText(this, "Žádný obrázek k
                              uložení!",Toast.LENGTH_LONG).show()
    } else {
        val predictions =
                photoDetectionController.photo.getStyledPredictions(
                assets.open("photoLabels.txt"))
        val maxPrediction = predictions.split(" :").get(0)

        val saveDBIntent = Intent(this,
SaveToDatabaseScreen::class.java)
        saveDBIntent.putExtra("imageName", maxPrediction)
        saveDBIntent.putExtra("imageType",
                DetectionTypes.PHOTO_DETECTION.type)

        imageSingleton.image = photoDetectionController.photo.image

        startActivity(saveDBIntent)
    }
}

btnCancel.setOnClickListener(){
    finish()
}
```

Nyní je aktivita demonstrující klasifikaci objektů plně funkční. Aktivitu můžeme otestovat stisknutím tlačítka pro pořízení snímku z fotoaparátu mobilního zařízení. Po pořízení snímku se fotografie zobrazí v aktivitě v komponentě ImageView. Pod obrázkem se vypíší čtyři nejpravděpodobnější klasifikace vyfoceného objektu. Tyto klasifikace byli vygenerovány modelem v controlleru obsluhujícím tuto aktivitu. Jsme-li s predikcí a obrázkem spokojeni můžeme obrázek uložit do databáze stisknutím tlačítka pro uložení. Tlačítko otevře nové okno aplikace, které se stará o potvrzení uložení a následné uložení záznamu do databáze aplikace. Návrat na hlavní aktivitu aplikace obstarává poslední tlačítko, které ukončuje běh této aktivity demonstrující klasifikaci objektů.



Obrázek 8 - Ukázka aktivity demonstrující klasifikaci objektů
Zdroj: Vlastní zpracování

4.6 Databáze

V projektu je využita knihovna s názvem Room. Jedná se o vrstvu abstrakce nad SQLite, což umožňuje lokální ukládání do paměti mobilního zařízení. Mobilní zařízení tedy nemusí být připojeno k internetu, aby databáze fungovala. Room poskytuje nástroje pro vytváření a práci s relační databází.

Pro práci s Room databází je potřeba vytvořit databázovou tabulku. Pro vytvoření této tabulky vytvoříme novou třídu s anotací @Entity. Tato anotace nám umožní zadat jméno tabulky. Následně definujeme sloupce tabulky. Tabulka bude obsahovat sloupce s identifikátorem záznamu, jménem objektu s největší pravděpodobností výskytu, typem detekce a obrázkem. Sloupec, ve kterém bude uložený obrázek, je typu BLOB a uchovává obrázek převedený na sekvenci bajtů [31].

```
@Entity (
    tableName = "imageTable",
    indices = arrayOf(
        Index(value = arrayOf ("name"), unique = false),
        Index(value = arrayOf("name", "type")))
)
data class ImageDatabaseObject(
    @PrimaryKey(autoGenerate = true) val id : Int = 0,
    var name:String,
    val type:String,
    @ColumnInfo(name = "imageData", typeAffinity = ColumnInfo.BLOB)
    val imageData : ByteArray):java.io.Serializable
```

Následně vytvoříme rozhraní s anotací @Dao. Tato anotace označuje objekt sloužící jako přístup k datům databáze. Rozhraní s touto anotací obsahuje metody, které představují provádění databázových dotazů. Vytvoříme metody pro ukládání nových záznamů do tabulky, pro editaci záznamu v tabulce a pro odstranění záznamu z tabulky. Implementujeme také metody pro získání všech záznamů pomocí anotace @Query a následného SQL příkazu pro získání všech záznamů z tabulky. Implementujeme také metodu, která vrátí záznamy, které splňují požadované kritérium. V našem případě se jedná o zobrazení obrázků, které jsou určitého typu [31].

```
@Dao
interface ImageDatabaseObjectDAO {
    @Insert(onConflict = ABORT)
    fun addImage (imageObj: ImageDatabaseObject)
    @Update(onConflict = ABORT)
    fun updateImage (imageObj: ImageDatabaseObject) : Int
    @Delete
    fun deleteImage (imageObj: ImageDatabaseObject) : Int
    @Query("SELECT * FROM imageTable")
    fun getAllImages(): Flow<List<ImageDatabaseObject>>
    @Query ("SELECT * FROM imageTable where type = :detectionType")
    fun getImagesByTypes (detectionType : String) :
                    Flow<List<ImageDatabaseObject>>
}
```

Nyní pomocí anotace @Database vytvoříme třídu, která bude představovat samotnou databázi. V anotaci třídy určíme typ entity v databázi. Tento typ entity nastavíme na dříve vytvořenou třídu s anotací @Entity. Pomocí této třídy budeme přistupovat k instanci databáze [31].

```
@Database(
    entities = [ImageDatabaseObject::class],
    version = 1
)
abstract class ImageDatabase: RoomDatabase() {
    abstract fun imageDatabaseObjectDao(): ImageDatabaseObjectDAO
    companion object {
        //Implementace vytvoření databáze a získání instance databáze
    }
}
```

Nyní vytvoříme novou aktivitu, kterou budeme používat pokaždé, když budeme chtít uložit nový záznam do tabulky databáze. V XML souboru definujeme komponenty, které se budou zobrazovat v aktivitě aplikace. Nejdůležitější komponenta je ImageView, ve které bude zobrazen náhled ukládajícího se obrázku. Následuje komponenta s názvem EditText, která umožňuje uživateli editovat zobrazovaný text. Zobrazovaný text v komponentě EditText bude udávat název objektu na obrázku. Nakonec vložíme dvě ovládací tlačítka pro potvrzení uložení obrázku do tabulky databáze a zrušení ukládání.

```
<LinearLayout>
    <ImageView
        android:id="@+id/imgViewBitmap"/>
```

```
    <EditText
        android:id="@+id/editImageName"/>
    <TableLayout>
        <TableRow>
            <Button
                android:id="@+id/btnSaveToDB"
                android:text="@string/btn_save_to_db"/>
        </TableRow>
        <TableRow>
            <Button
                android:id="@+id/btnShowDatabase"
                android:text="@string/btn_cancel"/>
        </TableRow>
    </TableLayout>
</LinearLayout>
```

Následně provedeme implementaci algoritmu pro uložení konkrétního záznamu do databáze. Aktivita spoléhá, že jakmile je zavolána intentem při požadavku na uložení, tak získá z intentu název detekovaného objektu na obrázku a typ obrázku, který slouží k filtrování záznamů. Následně aktivita zobrazí z globální proměnné ukládaný obrázek v komponentě ImageView. Následně nastavíme naslouchač pro každé tlačítko v aktivitě. Tlačítko pro zrušení zavolá metodu s názvem finish a ukončí běžící aktivitu bez uložení záznamu do tabulky databáze. Tlačítko pro potvrzení uložení do databáze zkomprimuje obrázek a převede ho na sekvenci bajtů.

Pomocí rozhraní s názvem GlobalScope spustíme metodu addImage v proměnné s názvem imageDao. Tato proměnná obsahuje rozhraní sloužící pro přístup k datům databáze. Rozhraní bylo získané z instance databáze. Pomocí metody addImage vytvoříme nový záznam z textu v komponentě EditText, typu obrázku získaného z intentu a sekvence bajtů z obrázku. Jelikož jsme metodu ukládající data do databáze zavolali pomocí rozhraní s názvem GlobalScope, nebude po uzavření aktivity ukládání do databáze přerušeno. Ukládání bude běžet na pozadí aplikace, dokud není dokončeno.

```
class SaveToDatabaseScreen : AppCompatActivity() {
    private val imageDao by lazy {
        ImageDatabase.getDatabase(this).imageDatabaseObjectDao()
}
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_save_to_database_screen)
        val name = intent.getStringExtra("imageName") as String
        val type = intent.getStringExtra("imageType") as String
        val bitmap = imageSingleton.image
        val imageView = findViewById<ImageView>(R.id.imgViewBitmap)
        val txtView = findViewById<EditText>(R.id.editImageName)
        imageView.setImageBitmap(bitmap)
        txtView.text = name
        val btnSaveToDB = findViewById<Button>(R.id.btnSaveToDB)
        val btnCancel = findViewById<Button>(R.id.btnShowDatabase)
        btnSaveToDB.setOnClickListener(){
            val stream = ByteArrayOutputStream()
            bitmap.compress(Bitmap.CompressFormat.PNG, 70, stream)
            val image = stream.toByteArray()
            GlobalScope.launch {
                imageDao.addImage(
```

```
                    ImageDatabaseObject(0,
                    txtView.text.toString(), type, image))
            }
            finish()
        }
        btnCancel.setOnClickListener(){
            finish()
        }
    }
}
```

Poslední aktivita, která je spojená s funkčností databáze je aktivita, která má za úkol zobrazovat uložené záznamy z tabulky databáze. Vytvoříme novou aktivitu a do XML souboru pro zadanou aktivitu vložíme komponenty Spinner, RecyclerView a FloatingActionButton. Komponenta Spinner umožňuje uživateli vybírat hodnoty z rozbalovacího seznamu. Komponenta RecyclerView slouží k zobrazení velkého počtu dat v seznamu. Pomocí posouvání zobrazení lze v RecyclerView zobrazit mnoho dat, které by se za normálních okolností nevešli na obrazovku mobilního zařízení. FloatingActionButton je komponenta podobná běžnému tlačítku, ale je pevně umístěna na obrazovce. Toto tlačítko leží nad obsahem aktivity aplikace a je proto snadno dosažitelné uživatelem.

```
<LinearLayout>
    <Spinner
        android:id="@+id/spinnerDetectionTypes"/>
    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/imageObjectRecyclerView"/>
</LinearLayout>

<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/btnBack"/>
```

Nyní si vytvoříme třídy typu ListAdapter a ViewHolder. Třída ListAdapter slouží k zobrazení seznamu položek v komponentě RecyclerView a třída ViewHolder slouží k efektivnímu přístupu k položkám v seznamu. Důležitá je pro nás hlavně metoda bind ve třídě ViewHolder, ve které definujeme proměnné získané z databáze. Uložíme si do proměnné aktivity jméno detekovaného objektu, typ obrázku, a nakonec převedeme obrázek ve formě sekvence bajtů na obrázek typu Bitmap, který lze zobrazit například v komponentě ImageView.

```
fun bind(imageObject: ImageDatabaseObject) {
    binding.name.text = imageObject.name
    binding.type.text = imageObject.type
    var imgByteArray = imageObject.imageData
    var bitmap = BitmapFactory.decodeByteArray(
        imgByteArray, 0, imgByteArray.size);
    binding.imgDB.setImageBitmap(bitmap)
}
```

V aktivitě, ve které budeme zobrazovat záznamy z databáze, definujeme opět proměnnou s názvem imageDao, která je instancí rozhraní pro přístup k datům získané z instance databáze. Definujeme také pole s typy možných detekcí. Do pole vložíme na první pozici

prvek, který po svém zvolení zobrazí všechny záznamy z databáze. Toto pole vložíme do adaptéru komponenty Spinner, která toto pole zobrazí v aktivitě jako rozbalovací seznam.

Následně komponentě Spinner implementujeme naslouchač na výběr prvku ze seznamu. V naslouchači překryjeme dvě metody s názvem onItemSelected a onNothingSelected. V překryté metodě onItemSelected zkontrolujeme pomocí podmínky, zda jsme vybrali první prvek v seznamu. Pokud ano, zavoláme metodu getAllImages v proměnné s názvem imageDao. Vrátí se seznam prvků z databáze, které pošleme Adapteru komponenty RecyclerView, která tyto prvky zobrazí v aktivitě. Celý tento algoritmus zobrazení prvků z databáze se vykonává v korutině životního cyklu aktivity. Tato korutina běží asynchronně na pozadí aplikace. Načítání velkých dat z databáze tedy nezpomaluje aplikaci.

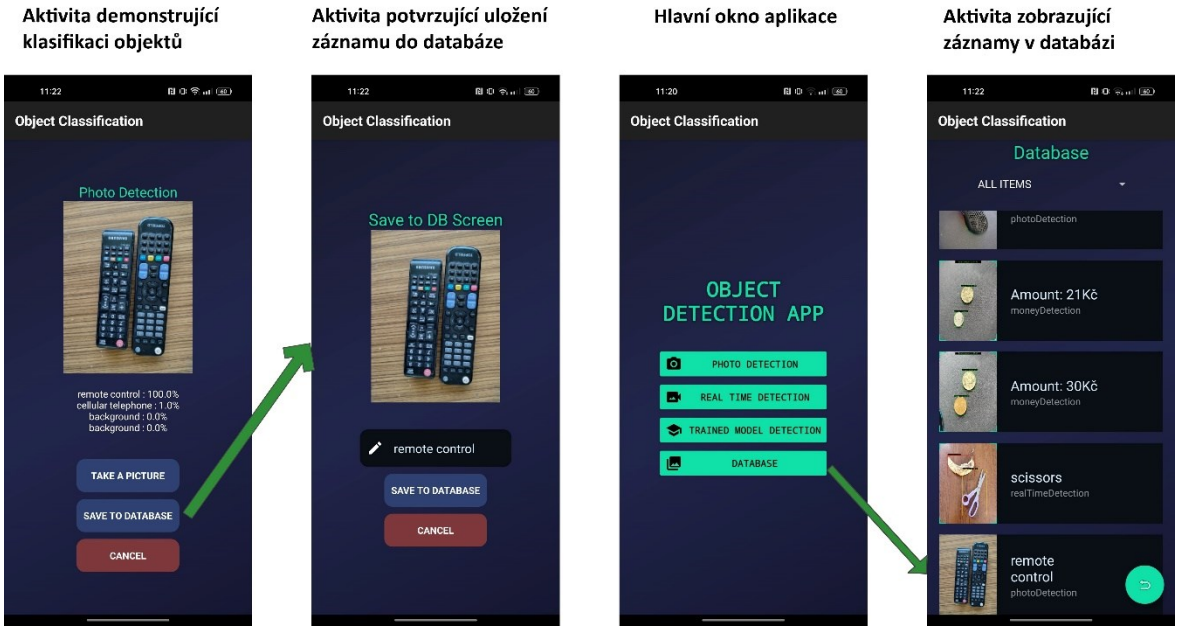
Pokud byl vybrán v komponentě Spinner jiný prvek než první, zavoláme v proměnné s názvem imageDao metodu getImagesByTypes. Do této metody vložíme jako parametr zvolený typ obrázku z komponenty Spinner. Z databáze se zobrazí pouze ty záznamy, které mají stejný typ.

```
val imageDao by lazy {
    ImageDatabase.getDatabase(this).imageDatabaseObjectDao()
}
val detectionTypes = arrayOf("ALL ITEMS",
    DetectionTypes.PHOTO_DETECTION,
    DetectionTypes.REAL_TIME_DETECTION,
    DetectionTypes.MONEY_DETECTION)
val adapter = ArrayAdapter(this,
        android.R.layout.simple_list_item_1, detectionTypes)
spinner.adapter = adapter

spinner.onItemSelectedListener = object :
AdapterView.OnItemSelectedListener {
    override fun onItemSelected(parent: AdapterView<*>, view: View,
            position: Int, id: Long) {
        if (position == 0) {
            lifecycle.coroutineScope.launch {
                imageDao.getAllImages().collect() {
                    imageAdapter.submitList(it)
                }
            }
        }else {
            lifecycle.coroutineScope.launch {
                imageDao.getImagesByTypes((detectionTypes[position] as
                DetectionTypes).type).collect() {
                    imageAdapter.submitList(it)
                }
            }
        }
    }
    override fun onNothingSelected(parent: AdapterView<*>) {
        lifecycle.coroutineScope.launch {
            imageDao.getAllImages().collect() {
                imageAdapter.submitList(it)
            }
        }
    }
}
```

Nyní aplikace dokáže pracovat s knihovnou Room a databází. Je možné aplikaci libovolně rozšířit, například přidáním další aktivity, která se zobrazí po kliknutí na prvek v RecyclerView. V této aktivitě by byli zobrazeny detaily a informace zvoleného prvku a bylo by možné tento prvek upravit, nebo smazat z tabulky databáze.

4.6.1 Příklad použití databáze v aplikaci

Na obrázku níže (Obrázek 9) vidíme příklad uložení záznamu do databáze. Obrázek byl pořízen v aktivitě demonstrující klasifikaci objektů. Po stisknutí na tlačítko pro uložení do databáze se otevřela nová aktivita, ve které se vyplnil název objektu s nejvyšší pravděpodobností výskytu na fotografii. Po potvrzení uživatele byl tento záznam uložen do databáze. Z hlavního okna aplikace následně můžeme otevřít aktivitu, která má za úkol zobrazování záznamů z databáze. V této aktivitě může uživatel zkontrolovat správné uložení záznamu do databáze aplikace.



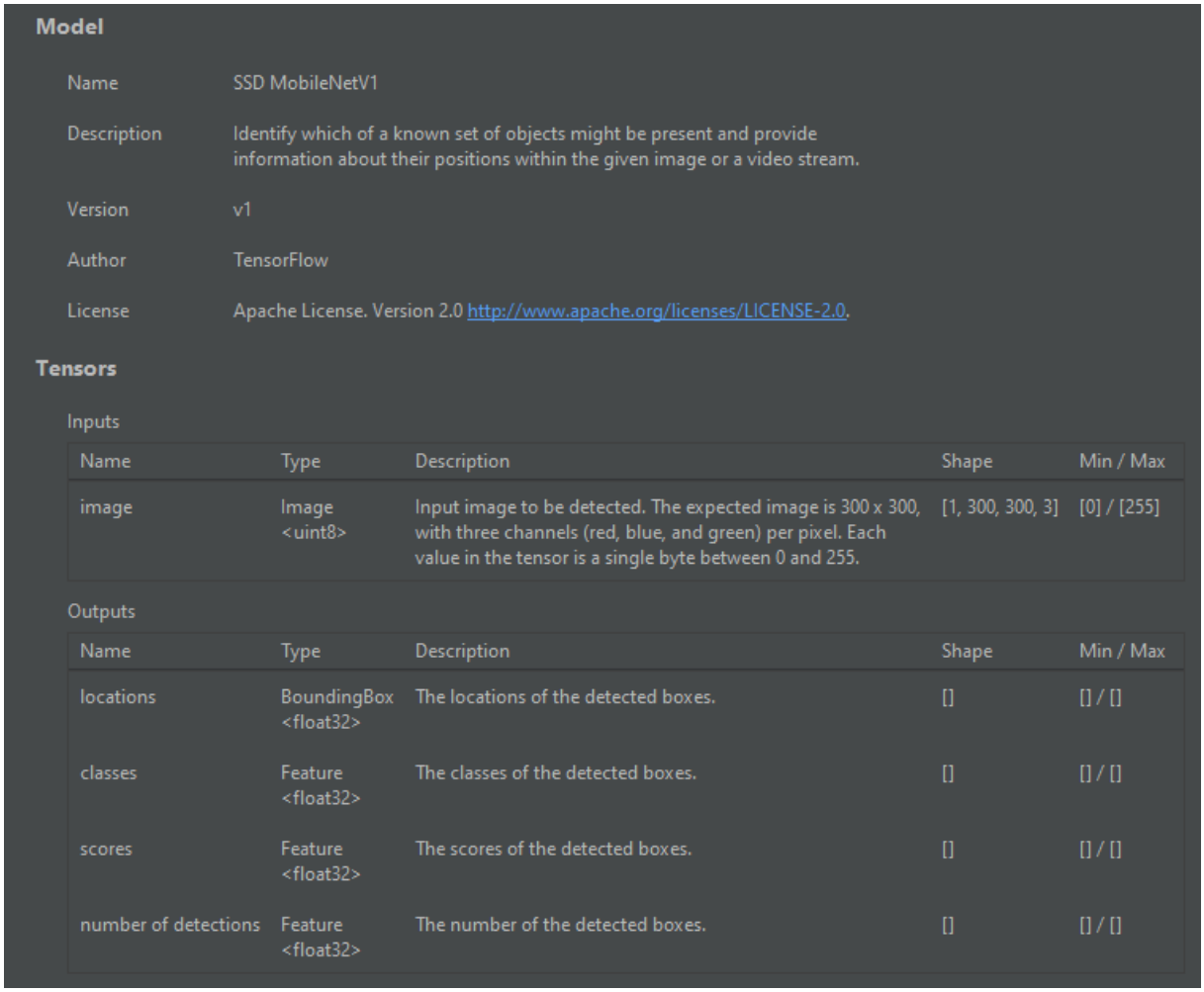
Obrázek 9 - Ukázka uložení záznamu do databáze
Zdroj: Vlastní zpracování

4.7 Detekce objektů v reálném čase

Pro demonstraci detekce objektů na fotografii využijeme již natrénovaný TensorFlow model. Jedná se o model určený pro studium a nalezneme ho na oficiálních stránkách TensorFlow (dostupné z: https://www.tensorflow.org/lite/examples/object_detection/overview [44]). Jedná se o menší model, který dokáže rozpoznat 90 tříd. Mezi příklady detekovatelných tříd je například monitor, klávesnice, kniha a televize. Vstupní tenzor modelu nese informace o zkoumaném obrázku. Tenzor se skládá z vícerozměrného pole bajtů. Model vyžaduje, aby vstupní obrázek měl určitou výšku a šířku. Výstupem modelu po detekci objektů na zkoumaném obrázku jsou čtyři tenzory. Tyto tenzory lze definovat jako bajtové buffery a následně je převést na proměnné typu pole. Výsledné tenzory nesou informace o pozici

ohraničení objektu na obrázku, o názvu detekovaných tříd, o pravděpodobnosti správného určení třídy a o počtu detekovaných objektů na obrázku.

Model a textový soubor s názvy detekovatelných tříd objektů importujeme do projektu.



Model

Name	SSD MobileNetV1
Description	Identify which of a known set of objects might be present and provide information about their positions within the given image or a video stream.
Version	v1
Author	TensorFlow
License	Apache License. Version 2.0 http://www.apache.org/licenses/LICENSE-2.0.

Tensors

Inputs

Name	Type	Description	Shape	Min / Max
image	Image <uint8>	Input image to be detected. The expected image is 300 x 300, with three channels (red, blue, and green) per pixel. Each value in the tensor is a single byte between 0 and 255.	[1, 300, 300, 3]	[0] / [255]

Outputs

Name	Type	Description	Shape	Min / Max
locations	BoundingBox <float32>	The locations of the detected boxes.	[]	[] / []
classes	Feature <float32>	The classes of the detected boxes.	[]	[] / []
scores	Feature <float32>	The scores of the detected boxes.	[]	[] / []
number of detections	Feature <float32>	The number of the detected boxes.	[]	[] / []

Obrázek 10 - Metadata modelu pro detekci objektů
Zdroj: Vlastní zpracování

Vytvoříme XML soubor pro aktivitu demonstrující detekci objektů. V tomto XML souboru jsou uvedeny komponenty, na které se bude v aktivitě odkazovat. Do XML souboru vložíme komponentu s názvem TextureView ve které budeme zobrazovat náhled z kamery mobilního zařízení. Následně vložíme tabulku, do které vložíme tlačítka pro uložení obrázku do databáze a zavření aktivity. Tlačítka mají stejnou implementaci jako v aktivitě pro klasifikaci objektů. Tlačítko pro ukládání do databáze vytváří nový intent, který spouští aktivitu s názvem SaveToDatabaseScreen a pomocí intentu přenáší do aktivity název detekovaného objektu a typ. Tlačítko pro uzavření aktivity volá metodu s názvem finish pro ukončení běžící aktivity.

```
<TextureView
    android:id="@+id/textureView"/>
<LinearLayout>
    <com.example.objectclassification.View.OverlayView
        android:id="@+id/overlayView"
    <TableLayout>
        <TableRow>
```

```
            <Button
                android:id="@+id/btnSaveToDB"
                android:text="@string/btn_save_to_db"/>
        </TableRow>
        <TableRow>
            <Button
                android:id="@+id/btnCancel"
                android:text="@string/btn_cancel"/>
        </TableRow>
    </TableLayout>
</LinearLayout>
```

Tlačítko, v hlavním okně aplikace, sloužící k otevření okna demonstrující detekci objektů spustí novou aktivitu. Ve třídě aktivity vytvoříme proměnné pomocí klíčového slova lateinit, které umožní pozdější inicializaci proměnné. Vytvoříme proměnné pro uchování názvů detekovatelných objektů, zkoumaného obrázku, použitého TensorFlow modelu a komponenty TextureView. Komponenta TextureView slouží k zobrazování videí, animací, grafických efektů a fotografií v aktivitě. Následně si připravíme i proměnné typu CameraManager, Handler, CameraDevice a OverlayView. Vytvoříme také proměnnou pro controller obsluhující spuštěnou aktivitu. Tyto proměnné inicializujeme později v kódu aplikace. Třída CameraManager poskytuje rozhraní pro interakci s fotoaparáty mobilního zařízení. Třída typu Handler umožňuje pracovat s více vlákny a dokáže zpracovávat zprávy a úkoly v různých vláknech. K reprezentaci fyzického fotoaparátu na mobilním zařízení slouží proměnná typu CameraDevice. Poslední OverlayView je třída implementující třídu View a slouží k zobrazení určitého výstupu do okna aktivity.

V metodě aktivity s názvem onCreate inicializujeme dříve vytvořené proměnné. Inicializujeme proměnnou controlleru pomocí konstruktoru a proměnnou typu CameraManager pomocí metody getSystemService(CAMERA_SERVICE), která zprostředkovává přístup ke službě fotoaparátu mobilního zařízení. Následně vytvoříme nové vlákno aplikace pomocí konstruktoru třídy HandlerThread. Toto vlákno zprostředkovává asynchronní zpracování zpráv. Vlákno je následně potřeba spustit pomocí metody start. Smyčku zpráv tohoto běžícího vlákna uložíme do dříve definované proměnné typu Handler. Tímto způsobem lze asynchronně zpracovat požadavky v novém vlákně. Následně inicializujeme proměnnou typu OverlayView. OverlayView je komponentou v XML souboru a pro její inicializaci musíme najít odkaz na definovaný prvek pomocí metody findViewById. K vytvoření instance TensorFlow modelu využijeme metodu modelu s názvem newInstance. Nakonec přečteme soubor s názvy tříd k detekování a zkontrolujeme oprávnění pro otevření kamery stejným způsobem jako v aktivitě sloužící ke klasifikaci objektů. Pokud uživatel povolil využívání fotoaparátu na mobilním zařízení spustí se metoda s názvem UISetup, která nastaví a připraví požadované komponenty ke snímání fotografií.

```
lateinit var labels:List<String>
lateinit var bitmap: Bitmap
lateinit var textureView: TextureView
lateinit var cameraManager: CameraManager
lateinit var handler: Handler
lateinit var cameraDevice: CameraDevice
lateinit var model: RealTimeModel
lateinit var realTimeDetectionController: RealTimeDetectionController
lateinit var overlayView: OverlayView
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_real_time_detection_screen)

    realTimeDetectionController = RealTimeDetectionController()
    cameraManager = getSystemService(CAMERA_SERVICE) as CameraManager

    val handlerThread = HandlerThread("cameraThread")
    handlerThread.start()
    handler = Handler(handlerThread.looper)

    overlayView = findViewById(R.id.overlayView)

    model = RealTimeModel.newInstance(this)

    labels = FileUtil.loadLabels(this, "realTimeLabels.txt")

    if (checkPermisions()){
        UISetup()
    }
}
```

V metodě UISetup definujeme proměnnou typu TextureView, kterou nalezneme jako komponentu v XML souboru pomocí metody findViewById. Definované komponentě TextureView implementujeme naslouchač pro SurfaceTexture. SurfaceTexture je třída, která umožňuje manipulaci s texturami v TextureView. V Textuře lze zobrazovat obraz z kamery fotoaparátu. Tento naslouchač překrývá metody spojené pro práci s texturou v TextureView. V projektu využijeme převážně metodu s názvem onSurfaceTextureAvailable, která se spustí při vytvoření komponenty. Využijeme také metodu s názvem onSurfaceTextureUpdated, která je vyvolána s každou změnou na textuře TextureView.

V metodě s názvem onSurfaceTextureAvailable v TextureView naslouchači zavoláme metodu cameraSetup, ve které využijeme dříve definovanou proměnnou typu CameraManager. Proměnná typu CameraManager otevře kameru fotoaparátu a začne naslouchat událostem spojeným s aktuálním stavem fotoaparátu. Naslouchání je implementováno pomocí překrytí obslužných metod onOpened, onDisconnected a onError. V metodě s názvem onOpened inicializujeme proměnnou typu CameraDevice, která slouží jako reprezentace kamery fotoaparátu. Následně vytvoříme odkaz na povrch v TextureView, do kterého budeme ukládat fotografie z fotoaparátu. Nyní vytvoříme požadavek na pořízení snímku z kamery fotoaparátu. Výsledný snímek z požadavku uložíme do povrchu v TextureView.

Po nastavení požadavku na pořizování snímků vytvoříme relaci snímání pomocí metody s názvem createCaptureSession. Tuto metodu poskytuje reprezentace kamery v proměnné typu CameraDevice. V překryté metodě s názvem onConfigured následně nastavíme relaci, která bude opakovaně pořizovat nové snímky, dokud není relace ukončena. Tuto relaci vytvoříme pomocí metody session.setRepeatingRequest, která v parametru přijme požadavek na opakující se snímání.

Následně již jenom ošetříme možné odpojení kamery, či chybu kamery. Pomocí překrytých metod s názvem onDisconnected a onError uvolníme zdroje spojené s fotoaparátem a zastavíme všechny aktivní relace fotoaparátu.

Nyní aplikace zobrazuje v TextureView sekvenci snímků pořízených kamerou mobilního zařízení. Abychom docílili požadované detekce objektů vrátíme se do metody s názvem UISetup a následně do překryté metody s názvem onSurfaceTextureUpdated. Metoda onSurfaceTextureUpdated je zavolána při každé změně v textuře komponenty TextureView. Díky opakujícímu se požadavku na snímání docílíme neustálého zobrazování nového snímku z fotoaparátu. Z povrchu komponenty TextureView získáme zobrazovaný obrázek typu Bitmap. Tento obrázek vložíme společně s modelem a seznamem detekovatelných jmen jako parametr do metody controlleru, která detekuje objekty na obrázku. Controller vrátí seznam detekovaných objektů na obrázku a vloží je do komponenty OverlayView, která překrývá komponentu TextureView a díky tomu může překreslit zobrazovaný obrázek vykreslovaný v textuře komponenty TextureView.

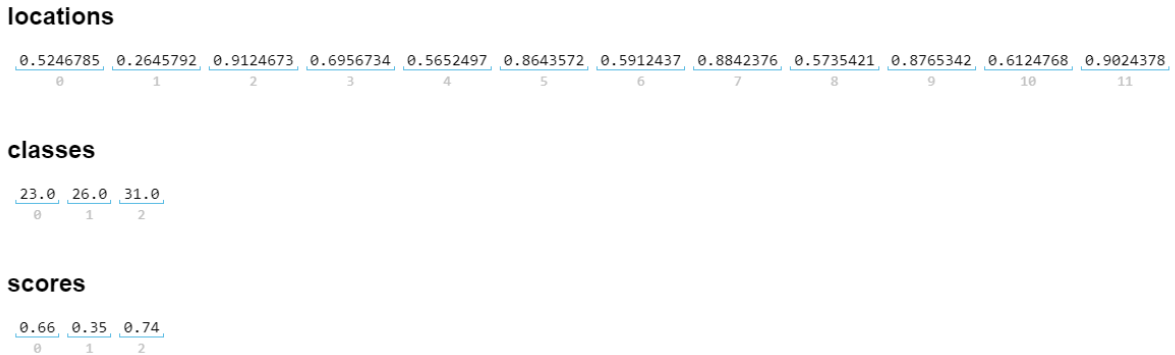
```
private fun UISetup() {
    textureView = findViewById(R.id.textureView)
    textureView.surfaceTextureListener =
                            object:TextureView.SurfaceTextureListener{
        override fun onSurfaceTextureAvailable(
            surface: SurfaceTexture,
            width: Int,
            height: Int
        ) {
            cameraSetup()
        }
        override fun onSurfaceTextureSizeChanged(
            surface: SurfaceTexture,
            width: Int,
            height: Int
        ) {}
        override fun onSurfaceTextureDestroyed(surface: SurfaceTexture):
Boolean {
            return false
        }
        override fun onSurfaceTextureUpdated(surface: SurfaceTexture) {
            bitmap = textureView.bitmap!!

            var objectsToDraw =
                        realTimeDetectionController.detectObjects(
                                labels,bitmap,model)
            overlayView.setRects(objectsToDraw)
        }
    }
}
fun cameraSetup() {
    cameraManager.openCamera(cameraManager.cameraIdList[0],
object:CameraDevice.StateCallback(){
        override fun onOpened(camera: CameraDevice) {
            cameraDevice = camera

            var surfaceTexture = textureView.surfaceTexture
            var surface = Surface(surfaceTexture)

            var captureRequest = cameraDevice.createCaptureRequest(
                        CameraDevice.TEMPLATE_PREVIEW)
            captureRequest.addTarget(surface)

            cameraDevice.createCaptureSession(listOf(surface),
                    object:CameraCaptureSession.StateCallback(){
                override fun onConfigured(session: CameraCaptureSession)
                {
                    session.setRepeatingRequest(captureRequest.build(),
                    null, null)
                }
                override fun onConfigureFailed(session:
                    CameraCaptureSession) {
                    camera.close()
                }

            }, handler)
        }
        override fun onDisconnected(camera: CameraDevice) {
```

```
            camera.close()
        }
        override fun onError(camera: CameraDevice, error: Int) {
            camera.close()
        }
    }, handler)
}
```

Ve složce s názvem Controller vytvoříme třídu controller pro obsluhu aktivity demonstrující detekci objektů. V controlleru vytvoříme metodu, kterou voláme z aktivity aplikace. Metoda implementuje algoritmus pro detekci objektů na fotografii. Nejdříve si vytvoříme rozhraní typu ImageProcessor, které změní výšku a šířku obrázku na požadované hodnoty. Po změně velikosti obrázku ho vložíme do TensorFlow modelu jako třídu TensorImage a provedeme detekci objektů. Po detekování objektů si uložíme výsledně tenzory do proměnné. Z této proměnné získáme buffery, které lze převézt na pole, jehož hodnoty jsou desetinná čísla. Tyto pole obsahují informace o pozicích objektu, názvu, do které třídy detekovaný objekt spadá a pravděpodobnosti správného přiřazení objektu do třídy. Pole obsahující hodnoty pozice objektu je větší než ostatní výsledná pole, protože obsahuje dvě souřadnice pro horizontální osu X a dvě souřadnice pro vertikální osu Y.

```
val imageProcessor = ImageProcessor.Builder().add(
    ResizeOp(
    300,
    300,
    ResizeOp.ResizeMethod.BILINEAR)
    ).build()

var image = TensorImage.fromBitmap(imageToProcess)
image = imageProcessor.process(image)

val outputs = realTimeModel.process(image)

val locations = outputs.locationsAsTensorBuffer.floatArray
val classes = outputs.classesAsTensorBuffer.floatArray
val scores = outputs.scoresAsTensorBuffer.floatArray
```

Výsledné proměnné typu pole obsahují desetinná čísla. Proměnná s názvem Locations označuje souřadnice detekovaného objektu na obrázku. Locations obsahuje čtyři hodnoty pro každou detekovanou třídu. Proměnná s názvem Classes obsahuje identifikátor detekované třídy. Pomocí tohoto identifikátoru lze nalézt název třídy v souboru, který obsahuje seznam jmen detekovatelných objektů. Proměnná s názvem Scores udává pravděpodobnost správného detekování objektu. Pole Scores má stejnou velikost jako pole Classes. Pro každý prvek v Classes existuje právě jeden prvek v poli s názvem Scores.



Obrázek 11 - Příklad výsledných proměnných po zpracování obrázku TF modelem

Zdroj: Vlastní zpracování

Po získání výsledků z TensorFlow modelu implementujeme algoritmus pro vykreslení informací do komponenty OverlayView, která překrývá snímky v TextureView komponentě. Deklarujeme proměnnou typu seznam, ve které bude algoritmus uchovávat prvky typu DrawableObject. Třída DrawableObject obsahuje proměnné uchovávající obdélník, který slouží jako ohraničení detekovaného objektu, název detekované třídy, pravděpodobnost správnosti detekce a pozici pro vykreslení názvu objektu. Následně deklarujeme proměnnou, která nám usnadní práci s proměnnou, která uchovává souřadnice obdélníku, označující objekt na obrázku. Tuto proměnnou, v příkladě pojmenovaná X, inicializujeme na hodnotu 0. Následně vytvoříme cyklus, který iteruje přes pole s výslednými pravděpodobnostmi správné detekce. Do proměnné X vložíme aktuální index a následně hodnotu X vynásobíme číslem čtyři. Toto chování umožní získávat pozice správného obdélníku z pole s názvem Locations. Následně vytvoříme podmínku, že chceme vykreslit pouze objekty, jejichž pravděpodobnost správného detekování je vyšší než 50 %. Pokud má nějaký objekt větší pravděpodobnost správného detekování vytvoří se objekt typu RectF. Objekt typu RectF lze chápat jako obdélník, do kterého uložíme souřadnice objektu z pole s názvem Locations. Pomocí definované proměnné X získáme souřadnice obdélníku označující objekt na obrázku. Hodnoty pozic musíme před uložením upravit, aby odpovídali velikosti zkoumaného obrázku. Dvě souřadnice pro horizontální osu X proto vynásobíme šířkou obrázku a dvě souřadnice pro vertikální osu Y vynásobíme výškou obrázku. Nyní vytvoříme dvě proměnné, do kterých uložíme souřadnice levého horního rohu detekovaného objektu. Na tyto souřadnice budeme v OverlayView vypisovat název detekovaného objektu a pravděpodobnost správného určení objektu. Tyto dvě proměnné a objekt typu RectF uložíme do nového objektu typu DrawableObject. Do tohoto nového objektu typu DrawableObject vložíme také název detekovaného objektu ze souboru se jmény detekovatelných tříd a pravděpodobnost správného určení objektu. Tento nový objekt vložíme do proměnné typu seznam, kterou jsme deklarovali na začátku metody. Po ukončení všech iterací cyklu uložíme do proměnné controlleru název detekované třídy s největší pravděpodobností správného určení. Tuto proměnnou controlleru využijeme při ukládání záznamu do databáze. Z metody vrátíme pomocí klíčového slova return proměnnou typu seznam, která obsahuje objekty, které se vykreslí v OverlayView.

```
var objectsToDraw: ArrayList<DrawableObject> =
                            arrayListOf<DrawableObject>()
var x = 0
scores.forEachIndexed { index, confidence ->
    x = index
    x *= 4
    if(confidence > 0.5){
        if(confidence > maxConfidence){
            maxConfidence = confidence
            maxConfidenceIndex = x
        }
        val rectF : RectF = RectF(
            locations.get(x+1)*weight,
            locations.get(x)*height,
            locations.get(x+3)*weight,
            locations.get(x+2)*height)
        val posX = locations.get(x + 1) * weight
        val posY = locations.get(x) * height - 1
        objectsToDraw.add(
            DrawableObject(rectF,
            labels.get(classes.get(index).toInt()),
            confidence,
            posX.toInt(),
            posY.toInt()))
    }
}
maxPredictionName = labels.get(classes.get(maxConfidenceIndex).toInt())
return objectsToDraw
```

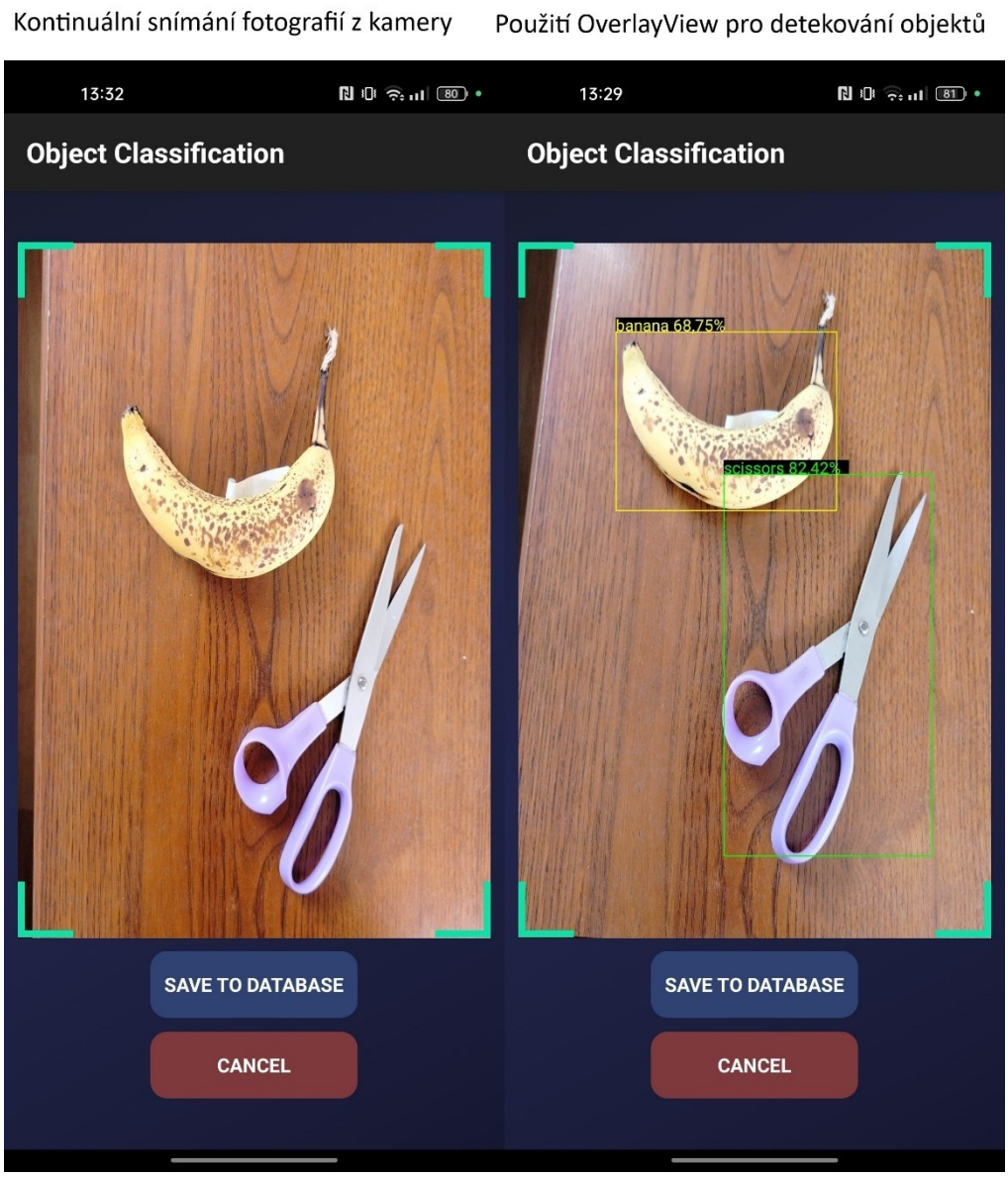
```
data class DrawableObject(
    val rectF: RectF,
    val objectName: String,
    val confidence: Float,
    val posX: Int,
    val posY: Int
) {}
```

Pro dokončení funkčnosti detekování objektů již pouze implementujeme vykreslování prvků do komponenty OverlayView. Třída OverlayView implementuje třídu View. Díky tomuto lze překrýt metodu s názvem Draw, pomocí které lze kreslit objekty do plátna. Když v aktivitě na komponentě typu OverlayView zavoláme metodu s názvem setRects, přepíšeme proměnnou typu seznam novými prvky k vykreslení. Poté v OverlayView vyvoláme metodu s názvem invalidate, která zavolá metodu draw. Do metody s názvem draw implementujeme algortismus vykreslování všech prvků typu DrawableObject ze seznamu. Pomocí třídy Paint lze nastavit vlastnosti vykreslovaných objektů, jako například velikost textu, šířka tahu, nebo barvu. V metodě draw vytvoříme cyklus, který postupně vykreslí všechny prvky ze seznamu. Volitelně lze také měnit barvu vykreslovaného textu a obdélníku, například pokud je pravděpodobnost výskytu menší. Pomocí metody s názvem drawRect ve třídě s názvem canvas vykreslíme do komponenty OverlayView obdélník označující pozici detekovaného objektu. Následně vynásobíme pravděpodobnost výskytu číslem 100. Díky tomu lze zobrazit výslednou pravděpodobnost pomocí procent. K číselné hodnotě pravděpodobnosti přidáme název detekovaného objektu a symbol procenta. Tento text vykreslíme pomocí metody s názvem

drawText ve třídě canvas. Text lze zdůraznit pomocí vykreslení dalšího obdélníku za vypisovaný text.

```
class OverlayView(attrs): View(attrs) {
    private var objectsToDraw = arrayListOf<DrawableObject>()
    fun setRects(objectsList: ArrayList<DrawableObject>) {
        this.objectsToDraw = objectsList
        invalidate()
    }
    private var rectPaint: Paint = Paint().apply {
        color = Color.GREEN
        style = Paint.Style.STROKE
        strokeWidth = 2.5f
    }
    private var textPaint: Paint = Paint().apply {
        color = Color.GREEN
        style = Paint.Style.FILL
        textSize = 35f
    }
    override fun draw(canvas: Canvas) {
        super.draw(canvas)
        textPaint.color = Color.GREEN
        rectPaint.color = Color.GREEN
        for (objToDraw in objectsToDraw) {
            var confidence = objToDraw.confidence
            if (confidence < 0.8) {
                textPaint.color = Color.YELLOW
                rectPaint.color = Color.YELLOW
            }
            if (confidence < 0.6) {
                textPaint.color = Color.RED
                rectPaint.color = Color.RED
            }
            objToDraw.rectF?.let { canvas?.drawRect(it, rectPaint) }
            var resultConfidence = confidence * 100
            var objName = objToDraw.objectName + " " +
                          "%.2f".format(resultConfidence) + "%"
            canvas.drawText(
                objName,
                objToDraw.posX.toFloat(),
                objToDraw.posY.toFloat(),
                textPaint)
        }
    }
}
```

Aplikace nyní v reálném čase zobrazuje detekci objektů a detekované objekty zvýrazňuje pomocí obdélníku. Následující obrázek (Obrázek 12) demonstruje příklad detekce objektů pomocí předpřipraveného TensorFlow modelu.



Obrázek 12 - Ukázka detekce objektů

Zdroj: Vlastní zpracování

Po úspěšné implementaci algoritmu pro detekci objektů přidáme do této aktivity již pouze obsluhu pro uložení záznamu do databáze a ukončení aktivity. Ukončení aktivity je provedeno pomocí metody finish, stejně jako v aktivitě demonstrující klasifikaci objektů. Uložení záznamu do databáze se také provádí stejně jako v aktivitě pro klasifikaci objektů. Vytvoří se stejný intent, do kterého vložíme název třídy s největší pravděpodobností detekce, kterou získáme z controlleru. Následně do intentu vložíme také typ obrázku. V této aktivitě je typem obrázku hodnota s názvem REAL_TIME_DETECTION. Jediný rozdíl je ten, že obrázek nezískáme z komponenty ImageView, ale z komponenty TextureView. Textura komponenty TextureView ale nezobrazuje překreslující prvky v OverlayView. Tyto prvky musíme do obrázku dodatečně vložit pomocí třídy Canvas.

```
btnSaveToDB.setOnClickListener(){
    val imageToSave: Bitmap = Bitmap.createBitmap(textureView.bitmap!!)
    val canvas = Canvas(imageToSave)
    overlayView.draw(canvas)

    val saveDBIntent = Intent(this, SaveToDatabaseScreen::class.java)
    saveDBIntent.putExtra("imageName",
                realTimeDetectionController.maxPredictionName)
    saveDBIntent.putExtra("imageType",
                DetectionTypes.REAL_TIME_DETECTION.type)

    imageSingleton.image = imageToSave

    startActivity(saveDBIntent)
}
```

5 Vlastní model

Po demonstraci implementace již existujících trénovaných modelů si ukážeme postup trénování vlastního modelu pro detekci objektů. Proces vývoje modelu obsahuje sběr požadovaného počtu dat, přípravu a konfiguraci trénování modelu, trénování samotné a následnou analýzu výsledného modelu [54].

Vývoj kvalitního a přesného modelu je zdlouhavý, nákladný a také náročný proces. Proto je následující model pouze demonstrací procesu vývoje.

Na začátku trénování je potřeba shromáždit co největší množinu dat. Na této množině bude model následně trénovaný a analyzovaný. Při trénování modelů pracujících s obrázky a fotografiemi platí pravidlo, že čím více fotografií obsahuje trénovací množina dat, tím je model lepší a přesnější. Kvalitní a profesionální modely se tedy mohou běžně trénovat na množině dat o velikosti tisíců, milionů až několik desítek milionů fotografií a obrázků.

V této kapitole bude demonstrován postup vývoje TensorFlow modelu, který dokáže detekovat České bankovky a mince. Bylo pořízeno 444 fotografií mincí i bankovek. Fotografie byli pořízeny na odlišném pozadí. Bylo použito jednobarevné pozadí, ale i pozadí s rušivým vzorem, jako například dřevěná podlaha, nebo pestrý ubrus na stole. Fotografie obsahují různé kombinace objektů. Některé fotografie obsahují samostatný objekt, například minci o hodnotě 5 Kč, jiné fotografie obsahují několik různých objektů. Pro větší rozmanitost trénovacích dat bylo použito také 20 obrázků s bankovkami a mincemi z internetu. Obrázky byli získány z aplikace Google Images. Obrázky byli a budou využity pouze ke studijním účelům. Celkově bylo pro trénování vlastního modelu využito 464 fotografií a obrázků mincí a bankovek.

5.1 Instalace a práce v aplikaci LabelImg

Po získání požadovaného množství obrázků je potřeba na každém obrázku zvlášť označit zkoumané objekty. K označování objektů na obrázcích bylo vyvinuto několik aplikací, v našem konkrétním příkladě byla využita aplikace s názvem LabelImg jejímž autorem je TzuTa Lin. Aplikace umožňuje snadno a relativně rychle označit objekty na fotografiích a obrázcích [32].

Pro spuštění aplikace potřebujeme python nainstalovaný v systému. Pokud nemáme v systému python nainstalovaný stáhneme si nejnovější distribuci z oficiálních stránek pythonu (dostupné z https://www.python.org/ [46]). Po stažení nejnovější distribuce python nainstalujeme spuštěním instalačního programu.

Následně otevřeme adresář, do kterého chceme umístit soubory aplikace LabelImg. Z tohoto adresáře otevřeme terminál, nebo příkazový řádek operačního systému. Pomocí příkazu pip nainstalujeme modul s názvem setuptools. Tento modul je vylepšení a rozšíření funkcí standardní knihovny Python distutils.

```
pip install setuptools
```

Následně nainstalujeme moduly PyQt5 a lxml.

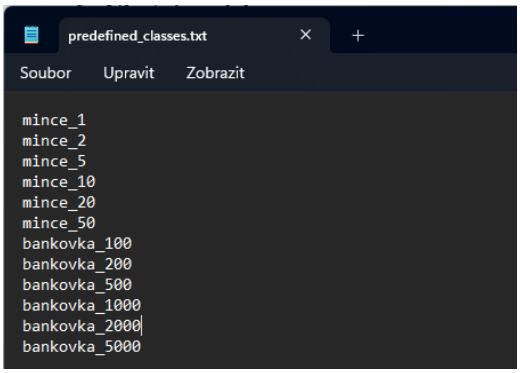
```
pip install PyQt5
pip install lxml
```

Nyní stáhneme soubory aplikace (dostupné z https:// github.com/tzutalin/labelImg [32]) a vložíme je do otevřené složky. Je možné také stáhnout aplikaci pomocí příkazu pip install labelImg.

```
pip install labelImg
```

Pokud máme soubory aplikace již ve složce můžeme aplikaci spustit kliknutím na soubor labelImg.py nebo pomocí příkazu v terminálu. Před spuštěním lze definovat seznam předdefinovaných tříd. Pro upravení seznamu otevřeme složku data a následně soubor s názvem predefined_classes.txt. Do tohoto soubor vložíme seznam tříd, které chceme, aby náš budoucí TensorFlow model uměl detekovat.

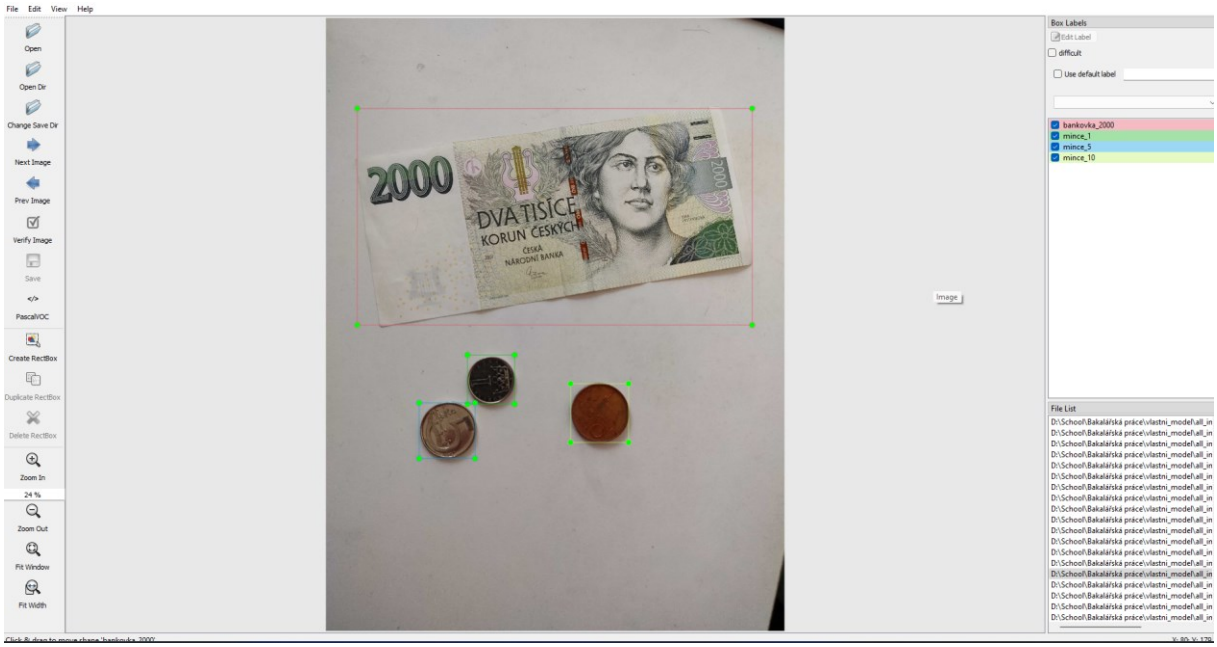
```
python labelImg.py
```



Obrázek 13- Třídy vlastního modelu pro detekci objektů

Zdroj: Vlastní zpracování

Po spuštění aplikace si můžeme všimnout hlavního okna a ovládacích tlačítek. Pomocí tlačítka otevřeme celou složku s fotografiemi a pomocí obdélníků začneme na každé fotografii označovat objekty pro detekování. Při označování objektů je důležité označit celý objekt. Aplikaci lze nastavit automatické ukládání při přepnutí na další obrázek ve složce.



Obrázek 14 - Ukázka označení objektů na obrázku pomocí aplikace LabelImg

Zdroj: Vlastní zpracování

Výsledkem označení fotografie je XML soubor, který nese informace o označených objektech na obrázku. Každý obrázek má právě jeden anotační soubor typu XML. Struktura souboru je vyobrazena níže. Tato struktura obsahuje název složky, souboru i kompletní cesty k souboru typu JPG. Následně obsahuje XML soubor také informace o velikosti obrázku neboli souboru typu JPG. Po těchto informacích obsahuje anotační soubor seznam objektů, které byli pomocí programu LabelImg označeny na obrázku. Každý objekt se skládá z názvu třídy a ze čtyř souřadnic obdélníku, který ohraničuje označený objekt.

```
<annotation>
    <folder>all_images</folder>
    <filename>img349.jpg</filename>
    <path>D:\School\all_images\img349.jpg</path>
    <size>
        <width>3000</width>
        <height>4000</height>
    </size>
    <object>
        <name>mince_5</name>
        <bndbox>
            <xmin>901</xmin>
            <ymin>1491</ymin>
            <xmax>1455</xmax>
            <ymax>2066</ymax>
        </bndbox>
    </object>
    <object>
        <name>mince_10</name>
        <bndbox>
            <xmin>1451</xmin>
            <ymin>1925</ymin>
```

```
            <xmax>2043</xmax>
            <ymax>2516</ymax>
        </bndbox>
    </object>
</annotation>
```

5.2 Trénování modelu

Model byl trénován pomocí cloudové služby Google Colaboratory. Tato služba je také označována jako Google Colab. Jedná se o online cloudový Jupyter Notebook. Jupyter Notebok je označení pro službu zpřístupňující programování pomocí webového rozhraní. Název Jupyter vznikl spojením programovacích jazyků Julia, Python a R. Google Colab navíc poskytuje bezplatný přístup k výpočetním zdrojům, jako například grafickému procesoru, nebo operační paměti. Díky tomu poskytuje Colab výkonný nástroj pro strojové učení a vzdělávání [55].

Práci zahájíme vytvořením nového sešitu a připojením zdrojů. Model byl trénován s připojenými 12,7 GB operační paměti RAM, 15 GB paměti grafické karty T4 a diskem o velikosti 78,2 GB. Po připojení výpočetních zdrojů a založení nového sešitu si naklonujeme do Colab projektu oficiální TensorFlow adresář s různými modely připravenými na trénování. Adresář naklonujeme pomocí příkazu git clone. Po naklonování adresáře spustíme instalaci TensorFlow nástrojů a požadovaných balíčků z adresáře models/research. Tento adresář obsahuje programy a scripty usnadňující práci s vývojem TensorFlow modelu [33].

```
!git clone --depth 1 https://github.com/tensorflow/models
!pip install /content/models/research/
!pip install tensorflow
```

Po instalaci všech potřebných balíčků nahrajeme složku s obrázky a anotačními soubory do Colab sešitu. Obrázky rozdělíme do dvou složek, které pojmenujeme train a validation. Do složky train vložíme většinu obrázků a anotací. Obrázky v této složce budou sloužit samotnému trénování modelu. V každém opakování trénování se z této složky vybere několik obrázků, které se vloží do neuronové sítě. Tato síť při trénování modelu určuje názvy a pozice objektů na obrázcích. Tréninkový algoritmus při trénování vypočítává průběžně ztrátu. Ztráta udává, zda byla detekce chybná. Následně algoritmus upraví váhy sítě pomocí algoritmu zpětného šíření chyby. Obrázky a anotace ve složce validation jsou trénovacím algoritmem využity ke kontrole průběhu trénování a úpravě rychlosti učení.

Při trénování modelu bylo do složky train vloženo 368 obrázků a do složky validation 96 obrázků. Nyní vytvoříme soubor typu PBTXT, který obsahuje identifikátor a název detekovatelné třídy. Celý použitý soubor typu PBTXT je detailně popsán v příloze A.

```
item {
  id: 1
  name: 'mince_1'
}
// Další záznamy
```

Po vytvoření souboru typu PBTXT musíme vytvořit dva soubory typu TFRECORD pro každou složku s obrázky a anotacemi. [34]. Soubor TFRECORD obsahuje sadu záznamů, které jsou uloženy jako sekvence binárních řetězců. Každý záznam může obsahovat různé

druhy dat, jako jsou obrazy, texty, čísla a jakékoliv další typy dat. Pro vytvoření dvou souborů typu TFRECORD využijeme z importovaného TensorFlow adresáře soubor s názvem create_pascal_tf_record.py. Tento soubor se nachází ve složce s názvem research a následně v podsložce s názvem object_detection [33]. Tato složka s názvem object_detection poskytuje potřebné scripty pro trénování vlastního TensorFlow modelu. Soubor pro generování TFRECORD souborů se nachází v podsložce s názvem dataset_tools. Tento soubor obsahuje algoritmus pro vytvoření souboru TFRECORD z anotačních XML souborů, které jsme nahráli dříve do složek train a validation. Pomocí následujících příkazů spustíme v Colab sešitě scripty a vytvoříme dva soubory typu TFRECORD.

```
python
models/research/object_detection/dataset_tools/create_pascal_tf_record.py
\
    --data_dir=/train \
    --set=train \
    --year=merged \
    --output_path=/train.tfrecord \
    --label_map_path=/label_map.pbtxt

python
models/research/object_detection/dataset_tools/create_pascal_tf_record.py
\
    --data_dir=/validation \
    --set=val \
    --year=merged \
    --output_path=/validation.tfrecord \
    --label_map_path=/label_map.pbtxt
```

Nyní vybereme TensorFlow lite model, který natrénujeme na naší množině dat. TensorFlow poskytuje mnoho použitelných modelů. Některé modely jsou výkonnější a některé jsou určeny pro specifické množiny dat. Pro demonstraci trénování modelu byl zvolen model s názvem ssd-mobilenet-v2-fpnlite-320. Pro spuštění trénování budeme potřebovat konfigurační soubor a samotný model s příponou .tar.gz. Konfigurační soubor modelu lze najít opět ve složce s názvem object_detection [33]. Konkrétně v podsložce s názvem configs a následně v podsložce tf2. Konfigurační soubor pro konkrétní vybraný model nese název ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.config. Konfigurační soubor můžeme zkopírovat do hlavního adresáře a upravit, nebo upravit rovnou v této složce. Pokud ale upravujeme soubor v config adresáři musíme při spuštění trénování zadat správně celou adresu ke konfiguračnímu souboru. V konfiguračním souboru modelu změníme hodnotu num_classes na počet tříd, které bude náš model umět detekovat. V našem konkrétním příkladě se jedná o 12 tříd. Následně nastavíme hodnotu pro parametr s názvem fine_tune_checkpoint. Do tohoto parametru vložíme adresu k prvnímu kontrolnímu bodu vybraného TensorFlow modelu. Změníme hodnotu pro parametry batch_size a num_steps. Batch_size udává počet obrázků, které se využijí při jedné iteraci trénování modelu. Hodnota s názvem num_steps udává počet iterací, či kroků, trénování modelu. Obecně platí, že čím více kroků model při trénování udělá, tím je model přesnější.

Následně v části konfiguračního souboru s názvem train_input_reader nastavíme parametr určující cestu k souboru typu PBTXT, který přiřazuje identifikátory jednotlivým

detekovatelným třídám. Následně nastavíme také parametr určující cestu k souboru typu TFRECORD uchovávající data pro obrázky a anotace ve složce train. Nakonec nastavíme v části s názvem eval_input_reader stejný parametr určující cestu k souboru typu PBTXT. Nastavíme také obdobně jako v předchozí části parametr určující cestu k souboru TFRECORD, ale tentokrát vybereme soubor uchovávající data pro obrázky a anotace ve složce validation.

```
model {
  ssd {
    num_classes: 12
 }
}
// Konfigurační soubor
train_config: {
  fine_tune_checkpoint:
"/content/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-
8/checkpoint/ckpt-0"
  batch_size: 16
  num_steps: 1500
}
// Konfigurační soubor
train_input_reader: {
  label_map_path: "/content/labelmap.pbtxt"
  tf_record_input_reader {
    input_path: "/content/train.tfrecord"
  }
}
eval_input_reader: {
  label_map_path: "/content/labelmap.pbtxt"
  tf_record_input_reader {
    input_path: "/content/validation.tfrecord"
  }
}
```

Celý konfigurační soubor je k dispozici v příloze B.

Před spuštěním trénování nám již stačí jen implementovat vybraný TensorFlow model. Model můžeme stáhnout z oficiálních stránek TensorFlow, nebo pomocí příkazu wget [35].

```
!wget
http://download.tensorflow.org/models/object_detection/tf2/20200711/ssd_m
obilenet_v2_fpnlite_320x320_coco17_tpu-8.tar.gz
```

Nyní po úpravě konfiguračního souboru a stažení modelu můžeme spustit trénování modelu. K trénování modelu využijeme script ze složky object_detection s názvem model_main_tf2.py [33]. Do prvního parametru scriptu s názvem pipeline_config_path vložíme cestu k našemu upravenému konfiguračnímu souboru. V dalším parametru nastavíme scriptu cestu do adresáře se staženým TensorFlow modelem. Parametr s názvem alsologtostderr nastaví zaznamenávání a vypisování výstupu během trénování modelu. Další parametr s názvem num_train_steps udává počet kroků trénování modelu. Tato hodnota se musí shodovat s hodnotou v konfiguračním souboru. Poslední parametr s názvem

sample_1_of_n_eval_examples udává kolik obrázků se má využít pro evaluaci modelu během jedné iterace trénování. Po dosazení všech hodnot lze spustit trénování modelu.

```
!python /content/models/research/object_detection/model_main_tf2.py \
    --pipeline_config_path={config_file} \
    --model_dir={model_folder} \
    --alsologtostderr \
    --num_train_steps={1500} \
    --sample_1_of_n_eval_examples=1
```

Během trénování lze průběžně sledovat výsledky pomocí nástroje TensorBoard [2]. TensorBoard poskytuje vizualizaci dat během trénování TensorFlow modelů. Rozšíření pro TensorBoard načteme pomocí příkazu %load_ext. Nástroj TensorBoard následně spustíme pomocí příkazu %tensorboard a zadání parametru, který udává složku, do které model zaznamenává průběžné hodnoty při trénování. Tyto hodnoty udávají například přesnost detekce, či ztrátu [36].

```
%load_ext tensorboard
%tensorboard --logdir 'content/training'
```

5.2.1 Verze modelu

Pro znázornění důležitosti a rozdílů v počtu provedených kroků trénování, bylo vytvořeno několik verzí modelu pro detekci mincí a bankovek. Změněna byla vždy hodnota počtu kroků trénování modelu. Pro správnou změnu počtu kroků pro trénování modelu je nutné změnit tuto hodnotu v konfiguračním souboru a při spuštění scriptu pro trénování.

Z následující tabulky (Tabulka 1) si můžeme všimnout, že po určitém počtu opakování se celková mAP ustálila kolem 70 %. Zkratka mAP značí průměrnou přesnost detekce objektů na obrázku pro každou třídu. Celkovou přesnost detekce získáme zprůměrováním mAP pro každou detekovatelnou třídu. Jedná se o metriku vyhodnocení výkonu detekčního modelu [37].

Výpočet průměrné přesnosti se provádí pomocí rovnice:

$$precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn}$$

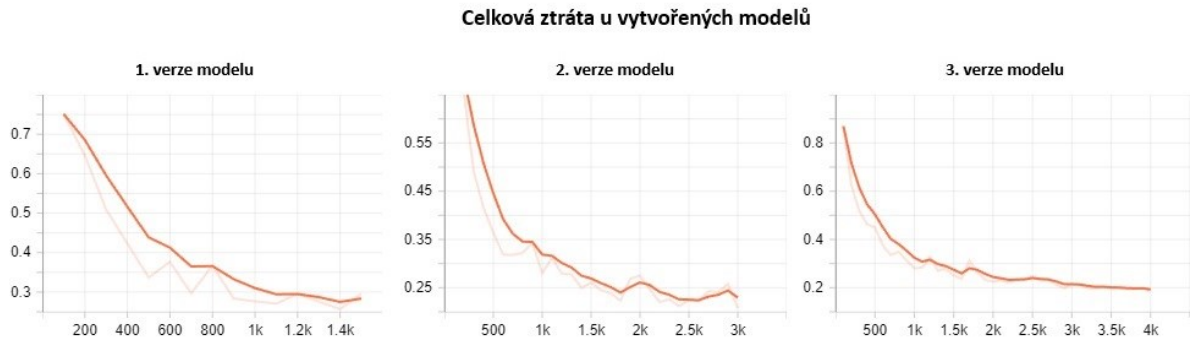
Kde tp značí správnou identifikaci objektů v obraze,
fp značí chybu, kdy model detekoval objekt, ale objekt se na obrázku nevyskytuje
fn značí chybu, kdy model nedetekoval objekt, který se vyskytuje na obrázku

Z proměnných s názvem precision a recall zjistíme průměr. Zjistíme-li tento průměr u všech detekovatelných tříd získáme mAP [24].

Tabulka 1 - Porovnání vytvořených verzí TFlitemodelu

Verze modelu	Doba provedení jedné iterace	Velikost fotografií	Počet kroků trénování	Celková doba trénování modelu	Celkové mAP
1. Verze modelu	5-6 s	400-500 KB	1500	2 hod. 13 min.	43,67 %
2. Verze modelu	5-6 s	400-500 KB	3000	4 hod. 09 min.	72,52 %
3. Verze modelu	5-6 s	400-500 KB	4000	5 hod. 39 min.	70,62 %

Pomocí nástroje TensorBoard si můžeme zobrazit a porovnat výsledné grafy vytvořených modelů. Grafy na ose X zobrazují počet iterací trénování a na ose Y udávají celkovou hodnotu ztráty při trénování. Obecně platí, že čím je ztráta menší, tím je model přesnější. Z grafů na obrázku (Obrázek 15) si můžeme všimnout, že se ztráta po určitém počtu opakování ustálí na hodnotě mezi 0,2 až 0,3. Ztráta udává pravděpodobnost chybné detekce objektu na obrázku.



Obrázek 15 - Celková ztráta u vytvořených modelů

Zdroj: Vlastní zpracování

Po dokončení trénování musíme model exportovat do kompatibilní podoby pro TensorFlow Lite knihovnu. K exportu využijeme opět script ze složky object_detection s názvem export_tflite_graph_tf2.py [33]. Script spustíme příkazem python a zadáním správných parametrů. Jako první parametr vložíme adresu posledního kontrolního bodu, který se vytvořil při trénování modelu. Následně nastavíme parametr s názvem output_directory, který udává kam se výsledný model uloží. Nakonec musíme do scriptu vložit konfigurační soubor, který jsme použili při trénování modelu.

```
!python
/content/models/research/object_detection/export_tflite_graph_tf2.py \
    --trained_checkpoint_dir {last_model_checkpoint} \
    --output_directory {output_folder} \
    --pipeline_config_path {config_file}
```

Po spuštění scriptu se vytvoří model typu TFLITE. Tento model si následně z Google Colab můžeme stáhnout příkazem files.download a implementovat do naší Android aplikace.

```
From google.colab import files
Files.donwload ('/content/model.tflite)
```

5.3 Implementace vlastního modelu do aplikace

Implementace vytvořeného modelu do aplikace je obdobná, jako implementace aktivity pro demonstraci detekce objektů. Vytvoříme novou aktivitu s XML souborem, který je totožný jako XML soubor aktivity pro detekci objektů. Hlavní aktivita obsahuje stejný algoritmus pro sekvenční snímání snímků a vykreslování objektů do překrývající komponenty OverlayView. Jediná změna je změna typu detekce, která se vkládá do nového intentu při ukládání záznamu do databáze.

Další změna nastává v Controlleru obsluhující aktivitu s naším vlastním modelem. Je nutné změnit algoritmus pro získání výsledků z modelu. Náš model totiž vyžaduje na vstupu, stejně jako model pro klasifikaci objektů, obrázek ve formě bajtového bufferu. Náš model také požaduje na vstupu obrázek s pevně stanovenou výškou a šířkou. Výška i šířka obrázku musí být 320 pixelů. Nejdříve je nutné tedy obrázku změnit velikost a následně vytvořit bajtový buffer se stejnou velikostí. Do bufferu vložíme obrázek převedený na sekvenci bajtů a následně buffer vložíme do modelu, který provede detekování objektů. Výsledné proměnné jsou pole desetinných čísel stejně jako v příkladu pro detekci objektů. Výsledné proměnné nesou informace o pravděpodobnosti správné detekce objektů, souřadnicích objektů a identifikátoru označujícího název objektu. Název objektu získáme také ze souboru, stejně jako v předchozích příkladech demonstrujících klasifikaci a detekci objektů.

```
var resizedImage = imageToProcces.copy(Bitmap.Config.ARGB_8888,true)
resizedImage = Bitmap.createScaledBitmap(resizedImage,320,320,true)

val model = moneyDetectionModel
val tensorBuffer = TensorBuffer.createFixedSize(intArrayOf(1, 320, 320,
3), DataType.UINT8)

var tImage = TensorImage(DataType.UINT8)
tImage.load(resizedImage)
var byteBuffer = tImage.buffer
tensorBuffer.loadBuffer(byteBuffer)

val outputs = model.process(tensorBuffer)
val scores = outputs.outputFeature0AsTensorBuffer.floatArray
val boxes = outputs.outputFeature1AsTensorBuffer.floatArray
val classes = outputs.outputFeature3AsTensorBuffer.floatArray
```

Program můžeme rozšířit implementací algoritmu, který spočítá celkovou hodnotu mincí a bankovek na obrázku. Pro toto rozšíření vložíme do Controlleru proměnnou, která bude průběžně uchovávat celkovou hodnotu sečtených peněz. Číselnou hodnotu z názvu třídy získáme pomocí metody filter. Do této metody vložíme podmínku, která vrátí pouze číselné hodnoty z názvu detekovatelného objektu.

```
var totalAmount:Int = 0
fun detectObjects() {
    scores.forEachIndexed { index, confidence ->
        if(confidence > 0.5 && confidence < 1.0){

            /*
                Stejný algoritmus jako v Controller pro detekci objektů
            */
            val objectName = labels.get(classes.get(index).toInt())
            moneyCount += (objectName.filter { it.isDigit() }).toInt()
        }
    }
    totalAmount = moneyCount
}
```

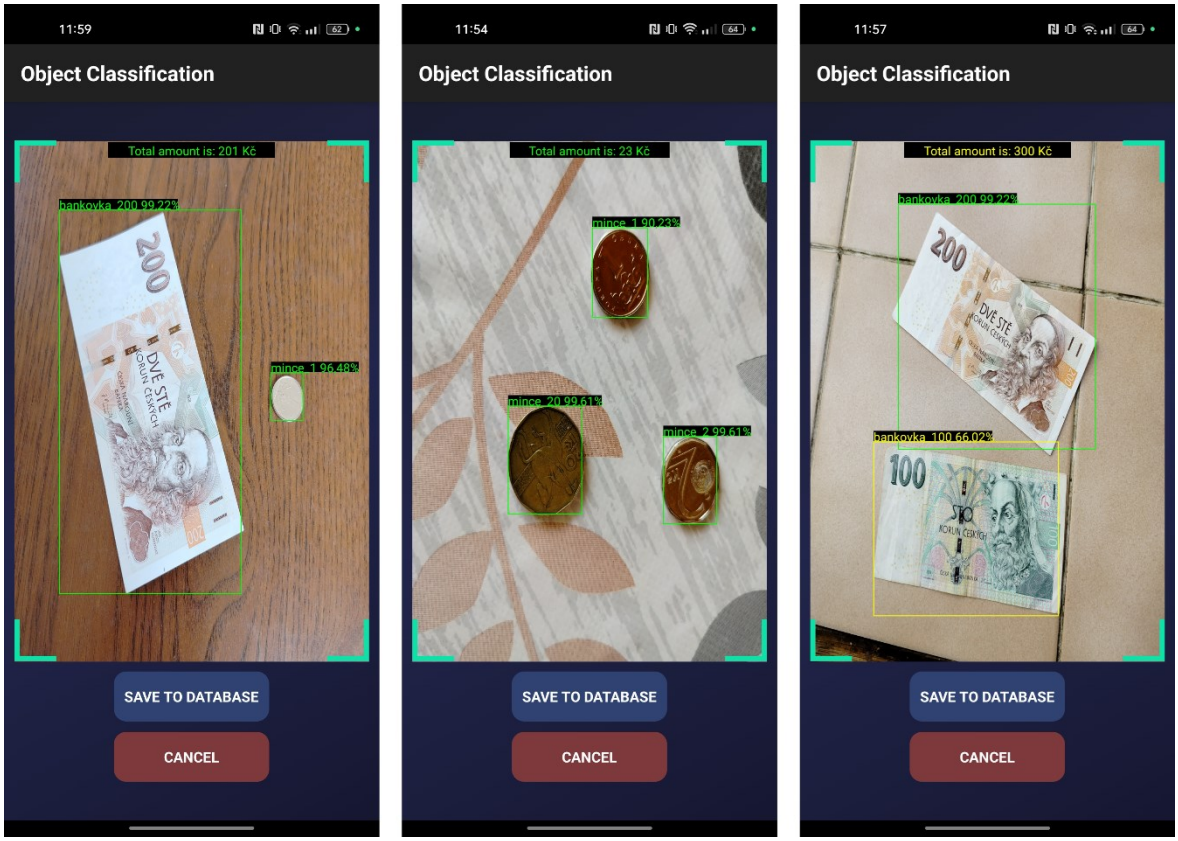
Pro korektní zobrazování celkové hodnoty mincí a bankovek musíme upravit ještě algoritmus v komponentě OverlayView. Jedná se o stejnou komponentu OverlayView, která byla použita při demonstraci detekce objektů. Kód pro vykreslování objektů je tedy stejný. Do OvelayView přidáme tedy pouze proměnnou s názvem headline, která bude uchovávat celkovou hodnotu mincí a bankovek. Z aktivity se hodnota této proměnné nastaví zavoláním metody s názvem drawHeadline. Tato metoda vloží svůj parametr do hodnoty této proměnné a spustí se překreslení. Překreslení OverlayView proběhne v překryté metodě s názvem draw, do které přidáme podmínku, která kontroluje, zda je proměnná headline prázdná. Pokud proměnná prázdná není, a tudíž obsahuje celkovou hodnotu mincí a bankovek vypíše se do komponenty text z této proměnné.

```
private var headline:String = ""
override fun draw(canvas: Canvas) {
    super.draw(canvas)

    /*
        Algoritmus vykreslování objektů představený dříve
    */

    if(!headline.isNullOrEmpty()){
        canvas.drawText(
            headline,
            (canvas.width/2)-(headline.length*8f),
            42.5f,
            textPaint)
    }
}
fun drawHeadline(showTotalAmount: String) {
    headline = showTotalAmount
    invalidate()
}
```

5.4 Demonstrace funkčnosti modelu

Po úspěšné implementaci modelu do naší aplikace může být model otestován. Testování provedeme pomocí opakovaného detekování různých objektů. V našem případě představují

objekty mince a bankovky, na kterých byl model natrénován. Správné testování by mělo být prováděno na velkém množství testovacích případů. Je dobré testovat detekci s malým i velkým počtem detekovatelných objektů. Objekty na obrázcích by měli být různorodé a focené na odlišném podkladu. Podklad může být rušivý, nebo prostý. Rušivým podkladem může být například pestrobarevný ubrus, nebo dřevěná podlaha s různou kresbou dřeva. Na demonstrujícím obrázku níže (Obrázek 16) si můžeme prohlédnout příklady testovaných případů.



Obrázek 16 - Demonstrace funkčnosti vlastního modelu v aplikaci
Zdroj: Vlastní zpracování

6 Závěr

Tato práce prezentovala čtenáři základní teoretické znalosti o strojovém učení a umělé inteligenci. Tyto znalosti lze nadále rozšířit podrobnějším studiem umělé inteligence. V části popisující historii umělé inteligence bylo představeno, že i ze začátku jednoduchá myšlenka dokáže časem vyrůst do složité vědní disciplíny.

Čtenáři byli v této práci představeny základní praktiky tvorby mobilní aplikace pomocí programovacího jazyka Kotlin. V části zabývající se tvorbou aplikace byl čtenář seznámen se základními principy programovacího jazyka Kotlin. Následně byla demonstrována implementace několika modelů umělé inteligence do mobilní aplikace. Tato mobilní aplikace využívala kameru mobilního zařízení k pořízení fotografií s detekovatelnými objekty.

Po demonstraci implementace již vytvořených TensorFlow modelů, byl čtenáři popsán postup vývoje vlastního modelu umělé inteligence. Tento model dokázal detekovat mince a bankovky na obrázku a určit jejich celkovou hodnotu. Model byl vytrénován pomocí 464 obrázků. Tyto obrázky sloužili jako trénovací a validační data vyvíjeného modelu.

Práce slouží jako vhodný zdroj základních informací pro čtenáře, které zajímá obor umělé inteligence. Přínosem této práce je podrobný popis vývoje mobilní aplikace a vývoje vlastního modelu pro detekci objektů na obrázcích.

7 Literatura

1. ALZUBI, Jafar; NAYYAR, Anand a KUMAR, Akshi. Machine Learning from Theory to Algorithms: An Overview. Online. Journal of Physics: Conference Series. 2018, roč. 1142. ISSN 1742-6588. Dostupné z: http://dx.doi.org/10.1088/1742-6596/1142/1/012012. [citováno 2024-03-20].
2. Google for Developers. Machine Learning Glossary. Online. 2024. Dostupné z: https://developers.google.com/machine-learning/glossary/tensorflow. [citováno 2024-02-07].
3. Machine Learning College. Umělá inteligence pro každého. Online. Dostupné z: https://www.mlcollege.com/umela-inteligence-pro-kazdeho/. [citováno 2024-03-21].
4. Harvard Kenneth C. Griffin. The History of Artificial Intelligence. Online. 2017. Dostupné z: https://sitn.hms.harvard.edu/flash/2017/history-artificial-intelligence/. [citováno 2024-03-22].
5. MOOR, J., "The Dartmouth College Artificial Intelligence Conference: The Next Fifty years", AI Magazine, č. 27.4, s. 87–89, 2006. Dostupné z: https://doi.org/10.1609/aimag.v27i4.1911. [citováno 2024-02-17].
6. STEELE, Guy L. An overview of COMMON LISP. Online. In: Proceedings of the 1982 ACM symposium on LISP and functional programming – LFP '82. New York, New York, USA: ACM Press, 1982, s. 98-107. ISBN 0897910826. Dostupné z: https://doi.org/10.1145/800068.802140. [citováno 2024-02-16].
7. MCCARTHY, John; FEIGENBAUM, Edward A. In Memoriam: Arthur Samuel: Pioneer in Machine Learning. AI Magazine, 1990, č. 11.3, s. 10-11. Dostupné z: https://doi.org/10.1609/aimag.v11i3.840. [citováno 2024-02-18].
8. SHUM, Heung-Yeung; HE, Xiao-dong; LI, Di. From Eliza to XiaoIce: challenges and opportunities with social chatbots. Frontiers of Information Technology & Electronic Engineering, 2018, č. 19, s. 10-26. Dostupné z: https://doi.org/10.48550/arXiv.1801.01957. [citováno 2024-02-18].
9. BERRY, David M. The Limits of Computation: Joseph Weizenbaum and the ELIZA Chatbot. Weizenbaum Journal of the Digital Society, 2023, č. 3.3. Dostupné z: https://doi.org/10.34669/WI.WJDS/3.3.2. [citováno 2024-02-18].
10. KUIPERS, Benjamin; FEIGENBAUM, Edward A.; HART, Peter E. a NILSSON, Nils J. Shakey: From Conception to History. Online. AI Magazine. 2017, roč. 38, č.1, s. 88-103. ISSN 0738-4602. Dostupné z: https://doi.org/10.1609/aimag.v38i1.2716. [citováno 2024-02-18].
11. Association for the Advancement of Artificial Intelligence. Online. Dostupné z: https://aaai.org/. [citováno 2024-02-07].
12. CAMPBELL, Murray; HOANE, A.Joseph a HSU, Feng-hsiung. Deep Blue. Online. Artificial Intelligence. 2002, roč. 134, č. 1-2, s. 57-83. ISSN 00043702. Dostupné z: https://doi.org/10.1016/S0004-3702(01)00129-1. [citováno 2024-02-19].
13. FERRUCCI, David; LEVAS, Anthony; BAGCHI, Sugato; GONDEK, David a MUELLER, Erik T. Watson: Beyond Jeopardy!. Online. Artificial Intelligence. 2013, roč. 199-200, s. 93-105. ISSN 00043702. Dostupné z: https://doi.org/10.1016/j.artint.2012.06.009. [citováno. 2024-02-19].
14. Watson and the Jeopardy! Challenge. Na: Youtube. Online. 06.11.2013. Dostupné z: https://www.youtube.com/watch?v=P18EdAKuC1U. Kanál uživatele IBM Research. [citováno 2024-02-19].

15. RETTO, Jesús. Sophia, first citizen robot of the world. ResearchGate. Online. 2017. Dostupné z: https://www.researchgate.net/publication/321319964_SOPHIA_FIRST_CITIZEN_ROBOT_OF_THE_WORLD. [citováno 2024-02-19].
16. CHEN, Jim X. The Evolution of Computing: AlphaGo. Online. Computing in Science & Engineering. 2016, roč. 18, č. 4, s. 4-7. ISSN 1521-9615. Dostupné z: https://doi.org/10.1109/MCSE.2016.74. [citováno 2024-02-19].
17. CHANG, Hyeong Soo, et al. Google DeepMind's AlphaGo: operations research's unheralded role in the path-breaking achievement. Online. 2019. Dostupné z: https://doi.org/10.1287/orms.2016.05.10. [citováno 2024-02-19].
18. YU, Haofeng. From Deep Blue to DeepMind: What AlphaGo Tells Us. Predictive Analytics and Futurism, 2016, č. 13, s. 42-45. Dostupné z: https://www.soa.org/globalassets/assets/Library/Newsletters/Predictive-Analytics-and-Futurism/2016/july/paf-2016-iss13-yu.pdf. [citováno 2024-02-19].
19. DALE, Robert. GPT-3: What's it good for? Online. Natural Language Engineering. 2021, roč. 27, č. 1, s. 113-118. ISSN 1351-3249. Dostupné z: https://doi.org/10.1017/S1351324920000601. [citováno 2024-02-20].
20. SZABOLCSI, Róbert. The birth of the term robot. Advances in Military Technology. 2014, č. 9.1, s. 117-128. ISSN 1802-2308 Dostupné z: https://aimt.cz/index.php/aimt/article/view/1021. [citováno 2024-02-20].
21. VINGE, Vernor. Technological singularity. VISION-21 Symposium sponsored by NASA Lewis Research Center and the Ohio Aerospace Institute. Online. 1993. Dostupné z: https://edoras.sdsu.edu/~vinge/misc/singularity.html. [citováno 2024-02-22].
22. ISLAM, Mohaiminul; CHEN, Guorong a JIN, Shangzhu. An Overview of Neural Network. Online. American Journal of Neural Networks and Applications. 2019, roč. 5, č. 1. ISSN 2469-7400. Dostupné z: https://doi.org/10.11648/j.ajnna.20190501.12. [citováno 2024-03-12].
23. DONGARE, A. D., et al. Introduction to artificial neural network. International Journal of Engineering and Innovative Technology (IJEIT), 2012, č. 2.1, s. 189-194. ISSN 2277-3754. Dostupné z: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=04d0b6952a4f0c7203577afc9476c2fcab2cba06. [citováno 2024-03-13].
24. DAI, Junyan. Real-time and accurate object detection on edge device with TensorFlow Lite. Online. Journal of Physics: Conference Series. 2020, roč. 1651, č. 1. ISSN 1742-6588. Dostupné z: https://doi.org/10.1088/1742-6596/1651/1/012114. [citováno 2024-03-20].
25. DEACON, John. Model-view-controller (mvc) architecture. Online. 2009. Dostupné z: https://www.academia.edu/30077059/Model_View_Controller_MVC_Architecture [citováno 2024-03-21].
26. BAYLISS, Darryl, Tom BLANKENSHIP, Fuad KAMAL a Namrata BANDEKAR. Android apprentice: beginning android developement with Kotlin. Second edition. [McGaheysville]: Razeware, [2019]. ISBN 978-1-942878-77-3. [citováno 2024-03-22].
27. CHOLLET, François. Deep learning v jazyku Python: knihovny Keras, Tensorflow. Přeložil Rudolf PECINOVSKÝ. Praha: Grada Publishing, 2019. Knihovna programátora (Grada). ISBN 978-80-247-3100-1. [citováno 2024-03-23].

28. KŘIVAN, Miloš. Umělé neuronové sítě. [s.l.]: Nakladatelství Oeconomica, Vysoká škola ekonomická v Praze 77 s. Dostupné online. ISBN 978-80-245-2420-7. Dostupné z: https://oeconomica.vse.cz/wp-content/uploads/publication/21284/Krivan_2021_Umele-neuronove-site.pdf. [citováno 2024-03-22].
29. GARNHAM, Adam. Artificial Intelligence: An Introduction. Velká Británie: Taylor & Francis. 2017. ISBN 9781351337861. [citováno 2024-03-23].
30. KARTHIKEYAN, J., TING, SU-HIE, Jin, Dr, ADGAONKAR, Shubham. Artificial Intelligence. 2022. ISBN 978-93-92995-15-6.
31. Android Developers. Save data in a local database using Room. Online. 2024 Dostupné z: https://developer.android.com/training/data-storage/room. [citováno 2024-04-09]
32. TZUTALIN: LabelImg Online. Git code. 2015. Dostupné z: https://github.com/tzutalin/labelImg. [citováno 2024-04-09]
33. Google Inc.: Tensorflow Object Detection API. Online. 2020. Dostupné z: https://github.com/tensorflow/models/tree/master/research/object_detection. [citováno 2024-04-09].
34. Google Inc.: Tensorflow TFRecord and tf.train.Example [online]. [citováno 2024-04-09]. Dostupné z: https://www.tensorflow.org/tutorials/load_data/tfrecord.
35. Google Inc.: Tensorflow. Online. Dostupné z: https://www.tensorflow.org/.
36. Google Inc.: Get started with TensorBoard. Online. Dostupné z: https://www.tensorflow.org/tensorboard/get_started. [citováno 2024-04-09]
37. HENDERSON, Paul a FERRARI, Vittorio. End-to-End Training of Object Class Detectors for Mean Average Precision. [online]. v: LAI, Shang-Hong; LEPETIT, Vincent; NISHINO, Ko a SATO, Yoichi (ed.). Computer Vision – ACCV 2016. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, s. 198-213. ISBN 978-3-319-54192-1. Dostupné z: https://doi.org/10.1007/978-3-319-54193-8_13. [citováno 2024-04-09].
38. GAIN, Ayan. A Survey on Machine Learning Algorithms. Online. International Journal for Research in Applied Science and Engineering Technology. 2021, roč. 9, č. 9, s. 357-366. ISSN 23219653. Dostupné z: https://doi.org/10.22214/ijraset.2021.37969. [citováno 2024-04-09].
39. SCHMIDHUBER, Juergen. Deep Learning. Scholarpedia. Online. 2015, roč. 10, č. 11. ISSN 1941-6016. Dostupné z: https://doi.org/10.4249/scholarpedia.32832. [citováno 2024-03-09]
40. SHIHAB, Khalil. A Backpropagation Neural Network for Computer Network Security. Online. Journal of Computer Science. 2006, roč. 2, č. 9, s. 710-715. ISSN 15493636. Dostupné z: https://doi.org/10.3844/jcssp.2006.710.715. [citováno 2024-03-26].
41. KHAN, Abdullah; LAGHARI, Asif a AWAN, Shafique. Machine Learning in Computer Vision: A Review. Online. ICST Transactions on Scalable Information Systems. 2021, s. 10.4108/eai.21-4-2021.169418. ISSN 2032-9407. Dostupné z: https://doi.org/10.4108/eai.21-4-2021.169418. [citováno 2024-04-7].
42. GUO, Qian; QIAN, Yuhua; LIANG, Xinyan; SHE, Yanhong; LI, Deyu et al. Logic could be learned from images. Online. International Journal of Machine Learning and Cybernetics. 2021, roč. 12, č. 12, s. 3397-3414. ISSN 1868-8071. Dostupné z: https://doi.org/10.1007/s13042-021-01366-w. [citováno 2024-04-11].

43. TEAM, Raywenderlich Com, GALATA, Irina and HOWARD, Joe. Kotlin Apprentice: Beginning Programming with Kotlin. B.m.: Razeware LLC. 2018. ISBN 978-1942878506. [citováno 2024-04-08].
44. Google Inc.: Tensorflow Object detection. Online. Dostupné z: https://www.tensorflow.org/lite/examples/object_detection/overview. [citováno 2024-04-08].
45. Google Inc.: Tensorflow Image classification Online. Dostupné z: https://www.tensorflow.org/lite/examples/image_classification/overview. [citováno 2024-04-09].
46. Python Software Foundation: Python. Online. Dostupné z: https://www.python.org/. [citováno 2024-04-10].
47. IBM. What is strong AI?. Online. Dostupné z: https://www.ibm.com/topics/strong-ai. [citováno 2024-04-11].
48. TensorFLow. Introduction to TensorFlow. Online. Dostupné z: https://www.tensorflow.org/learn. [citováno 2024-04-11].
49. IWM. How Alan Turing Cracked The Enigma Code. Online. Dostupné z: https://www.iwm.org.uk/history/how-alan-turing-cracked-the-enigma-code. [citováno 2024-04-11].
50. DICKMANNS, E.D. a ZAPP, A. Autonomous High Speed Road Vehicle Guidance by Computer Vision 1. Online. IFAC Proceedings Volumes. 1987, roč. 20, č. 5, s. 221-226. ISSN 14746670. Dostupné z: https://doi.org/10.1016/S1474-6670(17)55320-3. [citováno 2024-04-11].
51. Chilton Computing. A general survey by Sir James Lighthill. Online. Dostupné z: https://www.chilton-computing.org.uk/inf/literature/reports/lighthill_report/p001.htm. [citováno 2024-04-11].
52. Turing Post. The Story of AI Winters and What it Teaches Us Today. Online. 2023. Dostupné z: https://www.turingpost.com/p/aiwinters. [citováno 2024-04-11].
53. TechTarget. AI winter. Online. Dostupné z: https://www.techtarget.com/searchenterpriseai/definition/AI-winter. [citováno 2024-04-11].
54. EVAN Juras, EJ Technology Consultants. TensorFlow Lite Object Detection API in Colab. Online. 2023. Dostupné z: https://colab.research.google.com/github/EdjeElectronics/TensorFlow-Lite-Object-Detection-on-Android-and-Raspberry-Pi/blob/master/Train_TFLite2_Object_Detction_Model.ipynb. [citováno 2024-04-12].
55. Google Inc.: Google Colaboratory. Online. 2024. Dostupné z: https://colab.google/. [citováno 2024-04-12].

8 Přílohy

8.1 Příloha A – soubor PBTXT

Soubor mapující názvy detekovatelných tříd na číselné identifikátory.

```
item {
  id: 1
  name: 'mince_1'
}
item {
  id: 2
  name: 'mince_2'
}
item {
  id: 3
  name: 'mince_5'
}
item {
  id: 4
  name: 'mince_10'
}
item {
  id: 5
  name: 'mince_20'
}
item {
  id: 6
  name: 'mince_50'
}
item {
  id: 7
  name: 'bankovka_100'
}
item {
  id: 8
  name: 'bankovka_200'
}
item {
  id: 9
  name: 'bankovka_500'
}
item {
  id: 10
  name: 'bankovka_1000'
}
item {
  id: 11
  name: 'bankovka_2000'
}
```

```
item {
  id: 12
  name: 'bankovka_5000'
}
```

8.2 Příloha B – konfigurační soubor trénovaného modelu

Konfigurační soubor získaný z TensorFlow adresáře určeného pro trénování modelů.

```
# SSD with Mobilenet v2 FPN-lite (go/fpn-lite) feature extractor,
shared box
# predictor and focal loss (a mobile version of Retinanet).
# Retinanet: see Lin et al, https://arxiv.org/abs/1708.02002
# Trained on COCO, initialized from Imagenet classification checkpoint
# Train on TPU-8
#
# Achieves 22.2 mAP on COCO17 Val

model {
  ssd {
    inplace_batchnorm_update: true
    freeze_batchnorm: false
    num_classes: 12
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
      }
    }
    matcher {
      argmax_matcher {
        matched_threshold: 0.5
        unmatched_threshold: 0.5
        ignore_thresholds: false
        negatives_lower_than_unmatched: true
        force_match_for_each_row: true
        use_matmul_gather: true
      }
    }
    similarity_calculator {
      iou_similarity {
      }
    }
    encode_background_as_zeros: true
    anchor_generator {
      multiscale_anchor_generator {
        min_level: 3
        max_level: 7
```

```
      anchor_scale: 4.0
      aspect_ratios: [1.0, 2.0, 0.5]
      scales_per_octave: 2
    }
  }
  image_resizer {
    fixed_shape_resizer {
      height: 320
      width: 320
    }
  }
  box_predictor {
    weight_shared_convolutional_box_predictor {
      depth: 128
      class_prediction_bias_init: -4.6
      conv_hyperparams {
        activation: RELU_6,
        regularizer {
          l2_regularizer {
            weight: 0.00004
          }
        }
        initializer {
          random_normal_initializer {
            stddev: 0.01
            mean: 0.0
          }
        }
        batch_norm {
          scale: true,
          decay: 0.997,
          epsilon: 0.001,
        }
      }
      num_layers_before_predictor: 4
      share_prediction_tower: true
      use_depthwise: true
      kernel_size: 3
    }
  }
  feature_extractor {
    type: 'ssd_mobilenet_v2_fpn_keras'
    use_depthwise: true
    fpn {
      min_level: 3
      max_level: 7
      additional_layer_depth: 128
    }
    min_depth: 16
    depth_multiplier: 1.0
    conv_hyperparams {
```

```
        activation: RELU_6,
        regularizer {
          l2_regularizer {
            weight: 0.00004
          }
        }
        initializer {
          random_normal_initializer {
            stddev: 0.01
            mean: 0.0
          }
        }
        batch_norm {
          scale: true,
          decay: 0.997,
          epsilon: 0.001,
        }
      }
      override_base_feature_extractor_hyperparams: true
    }
    loss {
      classification_loss {
        weighted_sigmoid_focal {
          alpha: 0.25
          gamma: 2.0
        }
      }
      localization_loss {
        weighted_smooth_l1 {
        }
      }
      classification_weight: 1.0
      localization_weight: 1.0
    }
    normalize_loss_by_num_matches: true
    normalize_loc_loss_by_codesize: true
    post_processing {
      batch_non_max_suppression {
        score_threshold: 1e-8
        iou_threshold: 0.6
        max_detections_per_class: 100
        max_total_detections: 100
      }
      score_converter: SIGMOID
    }
  }
}

train_config: {
  fine_tune_checkpoint_version: V2
```

```
  fine_tune_checkpoint:
"/content/models/mymodel/ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-
8/checkpoint/ckpt-0"
  fine_tune_checkpoint_type: "detection"
  batch_size: 16
  sync_replicas: true
  startup_delay_steps: 0
  replicas_to_aggregate: 8
  num_steps: 4000
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
    random_crop_image {
      min_object_covered: 0.0
      min_aspect_ratio: 0.75
      max_aspect_ratio: 3.0
      min_area: 0.75
      max_area: 1.0
      overlap_thresh: 0.0
    }
  }
  optimizer {
    momentum_optimizer: {
      learning_rate: {
        cosine_decay_learning_rate {
          learning_rate_base: .08
          total_steps: 50000
          warmup_learning_rate: .026666
          warmup_steps: 1000
        }
      }
      momentum_optimizer_value: 0.9
    }
    use_moving_average: false
  }
  max_number_of_boxes: 100
  unpad_groundtruth_tensors: false
}

train_input_reader: {
  label_map_path: "/content/labelmap.pbtxt"
  tf_record_input_reader {
    input_path: "/content/train.tfrecord"
  }
}

eval_config: {
  metrics_set: "coco_detection_metrics"
  use_moving_averages: false
```

```
}

eval_input_reader: {
  label_map_path: "/content/labelmap.pbtxt"
  shuffle: false
  num_epochs: 1
  tf_record_input_reader {
    input_path: "/content/val.tfrecord"
  }
}
```