Mobilná aplikácia pre rozpoznávanie štvorlístkov v rámci fotografií

Bc. Matúš Giertl

Diplomová práce

2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

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

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Matúš Giertl**
Osobní číslo:	**A20195**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Mobilní aplikace pro rozpoznávání čtyřlístků v rámci fotografií**
Téma práce anglicky:	**Mobile Application for Recognizing Four-Leaf Clovers within Photos**

Zásady pro vypracování

1. Vypracujte literární rešerši zabývající se modely pro detekci, klasifikaci, případně segmentaci objektů v obraze.
2. Připravte a proveďte anotaci datasetu obsahující čtyřlístky a trojlístky jetele.
3. Vyberte několik modelů hlubokého učení vhodných pro detekci, případně segmentaci čtyřlístků.
4. Natrénujte vybrané modely na připraveném a anotovaném datasetu a proveďte kvantitativní a kvalitativní srovnání výsledků realizovaných modelů.
5. Finální model implementujte do mobilní aplikace.

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

Seznam doporučené literatury:

1. MARTÍNEZ Jesús. TensorFlow 2.0 Computer Vision Cookbook. Birmingham: Pack publishing, 2021. ISBN 9781838829131.
2. PLANCHE, Benjamin a Eliot ANDRES. Hands-On Computer Vision with TensorFlow 2. Birmingham: Pack publishing, 2019. ISBN 9781788830645.
3. ROSEBROCK, Adrian. Starter Bundle. In: Deep Learning for Computer Vision with Python. PyimageSearch.com. 2017.
4. ROSEBROCK, Adrian. Practitioner Bundle. In: Deep Learning for Computer Vision with Python. PyimageSearch.com. 2017.
5. KAR, Krishnendu. Mastering Computer Vision with TensorFlow 2.x. Packt Publishing. 2020. ISBN 978-1-83882-706-9.
6. RASCHKA, Sebastian a Vahid MIRJALILI. Python machine learning: machine learning and deep learning with Python, scikit-learn, and TensorFlow . Second edition. Birmingham: Packt, 2017, xviii, 595 s. ISBN 978-1-78712-593-3.
7. CHOLLET, François. Deep learning v jazyku Python: knihovny Keras, Tensorflow . Praha: Grada Publishing, 2019, 328 s. Knihovna programátora. ISBN 978-80-247-3100-1.

Vedoucí diplomové práce: **Ing. Petr Žáček, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **3. prosince 2021**
Termín odevzdání diplomové práce: **23. května 2022**

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

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

Ve Zlíně dne 24. ledna 2022

Jméno, příjmení: Bc. Matúš Giertl

Název diplomové práce: Mobilní aplikace pro rozpoznávání čtyřlístků v rámci fotografií

Prohlašuji, že

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

Prohlašuji,

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

Ve Zlíně, dne

Bc. Matúš Giertl, v.r.
podpis studenta

ABSTRAKT

Práca sa zaoberá analýzou a implementáciou machine learning modelu pre object detection a jeho následnou integráciou do mobilnej aplikácie. Cieľom bolo vytvorenie aplikácie pre real-time on-device detekciu štvorlístkov v tráve. Samotnej implementácii predchádzal výskum ako oblasti computer vision, tak i sveta mobilného vývoja a výsledný model, ako i aplikácia boli trénované a vyvíjané pomocou Apple nástrojov Create ML a SwiftUI.

Klíčová slova: Machine Learning, On-device machine learning, Computer Vision, Object Detection, Object Localization, Apple, Core ML, Create ML, Swift, SwiftUI, UIKit, mobilný vývoj, natívny vývoj, mobilné aplikácie, TestFlight, App Store Connect, GitHub

ABSTRACT

The thesis deals with the analysis and implementation of a machine learning model for object detection and its subsequent integration into a mobile application. The goal was to create an application for real-time on-device detection of four-leaf clover in patches of grass. The implementation itself was preceded by research in both the field of computer vision and the world of mobile app development. The resulting model as well as the application were trained and developed using Apple's Create ML and SwiftUI tools.

Keywords: Machine Learning, On-device machine learning, Computer Vision, Object Detection, Object Localization, Apple, Core ML, Create ML, Swift, SwiftUI, UIKit, mobile development, native development, mobile apps, TestFlight, App Store Connect, GitHub

Poďakovanie patrí najmä vedúcemu mojej diplomovej práce Ing. Petrovi Žáčkovi, Ph.D za podnetné rady pri jej tvorbe. Taktiež by som sa rád poďakoval doc. Ing. Zuzane Komínkovej Oplatkovej, Ph.D za cenné konzultácie pri tvorbe výsledného modelu.

V neposlednom rade patrí poďakovanie mojej rodine a priateľke za neochvejnú podporu a motiváciu počas celého štúdia.

Prehlasujem, že odovzdaná verzia diplomovej práce a verzia elektronická nahraná do IS/STAG sú totožné.

OBSAH

ÚVOD

Machine learning v posledných rokoch zažíva nevýdanú mieru pozornosti najmä vďaka jeho možnostiam a schopnostiam vo viacerých odvetviach, či už priemyslu, alebo bežného života. Mnohé odvetvia sa naň spoliehajú pre kontrolu kvality ich výrobkov, analýzu dát v precíznejšej miere, v akej toho človek nemusí byť fyzicky schopný, čím sa znižujú náklady a zároveň zvyšuje efektivita daného subjektu.

Tieto možnosti však nie sú doménou iba data science alebo veľkých firiem. Existuje niekoľko faktorov, ktoré sprístupnili machine learning techniky masám. S narastajúcim výpočetným výkonom, množstvom dát, neustálym vylepšovaním algoritmov a vznikom zrozumiteľných nástrojov pre trénovanie vlastných modelov je dnes machine learning a jeho jednotlivé odvetvia dostupnejší, ako kedykoľvek predtým.

Jednou zo spoločností poskytujúcich kompletný ML ekosystém je i firma Apple, ktorá uviedla na trh svoje nástroje Core ML a Create ML, umožňujúce vývojárom nielen mobilných, ale i desktopových a tabletových aplikácii vytrénovať model, ktorý je pripravený pre následnú implementáciu v rôznych typoch aplikácii.

Cieľom práce je teda poskytnúť náhľad do tohto ekosystému, pomocou poskytnutého vlastnoručne anotovaného datasetu vytrénovať optimálny model priamo na zariadení a následne ho implementovať pomocou Core ML frameworku do mobilnej aplikácie, ktorá bude tak priamo v zariadení, bez nutnosti odosielania dát na server a v réalnom čase schopná detekovať ako štvorlístky, tak i trojlístky v tráve.

I. TEORETICKÁ ČASŤ

1 COMPUTER VISION

Tento pojem spadá pod časť umelej inteligencie (AI), ktorá umožňuje počítačom a systémom získavať zmysluplné dáta z digitálnych obrázkov, videí a ostatných vizuálnych vstupov a na základe týchto informácii vykonávať určité akcie, prípadne odporučenia. Ak „klasická“ umelá inteligencia umožňuje počítačom premýšľať, počítačové videnie im umožňuje pozorovať, porozumieť a v prenesenom zmysle slova vidieť.

Principiálne funguje podobne ako ľudské videnie, s tým, že ľudia majú navrch v podobe bohatého kontextu. Vedia nielen ako rozoznať rozdiely v jednotlivých objektoch medzi sebou, no zároveň aj ako ďaleko od seba sú, či sú v pohybe alebo či s nimi niečo nie je v poriadku.

Počítačové videnie trénuje systém, aby dokázali vykonávať všetky vyššie spomenuté funkcie, no za výrazne kratší čas za pomoci kamier a dát, než pomocou ľudskej zrakovej sústavy. Keďže takto trénovaný systém dokáže sledovať a analyzovať tisíce produktov, môže tak prekonať ľudské schopnosti a vzniká značná výhoda v kvalitatívnej fázi, napríklad výrobného procesu. Okrem výroby sa computer vision používa v rôznych priemyselných oblastiach, počnúc energetikou, manufaktúrou a končiac automobilovým priemyslom. [1]

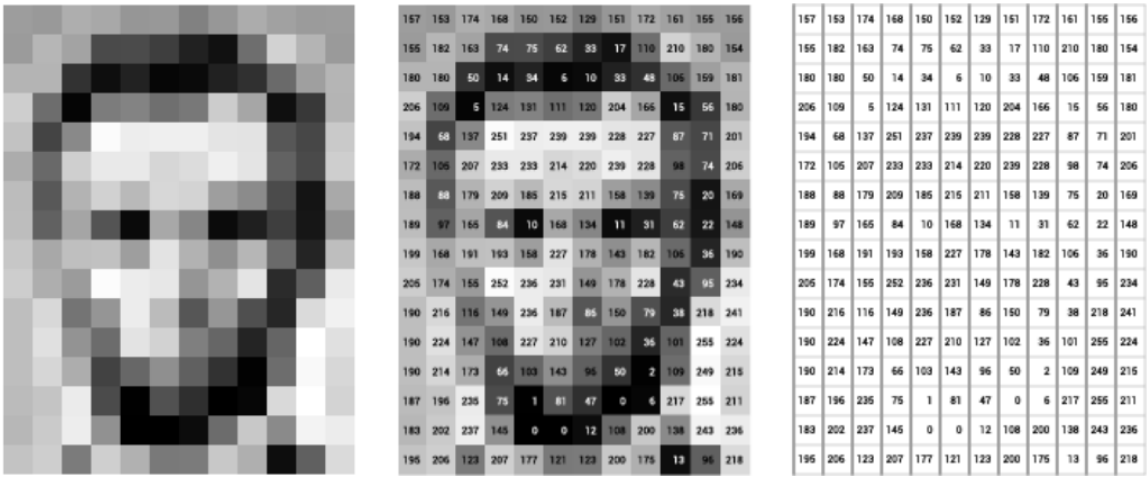
1.1 Princíp Computer Vision

Zjednodušene sa jedná o tri kroky:

1. *Získavanie obrazových dát:* Obrázky, dokonca veľké sety je možné získať v real-time pomocou videa, fotiek, alebo 3D technológie za účelom analýzy.
2. *Sprácovávanie:* Učenie modelu prebieha na získaných obrázkoch, ktoré musia byť anotované
3. *Porozumenie:* Finálny iteratívny krok, kde je objekt identifikovaný alebo klasifikovaný

Tieto zdanlivo jednoduché kroky však v sebe skýtajú mnoho nástrah. Obrázky pozostávajú z pixelov, ako najmenších kvanitfikovatelných jednotiek. Počítače ich spracúvajú ako pole, kde každý pixel má určitú sadu hodnôt reprezentujúcich prítomnosť a intenzitu troch primárnych farieb: červenej, zelenej a modrej. Ich kombináciou vzniká digitálny obrázok. Je teda možné tvrdiť, že každý obrázok je vlastne matica a CV teda pracuje priamo s nimi. Zatiaľ, čo najjednoduchšie algoritmy používajú lineárnu algebru pre manipuláciu s týmito

maticami, komplexné aplikácie zahŕňajú operácie ako konvolúcie a downsampling. Obrázok nižšie slúži ako typický príklad spôsobu, akým počítač „vidí“ obrázok.



Obrázok 1 Príklad computer vision [5]

Hodnoty na obrázku 1 reprezentujú pixelové hodnoty v daných koordinátach obrázka, kde hodnota 255 reprezentuje biely bod a 0 reprezentuje čierny bod. Čím väčší obrázok, tým samozrejme väčšia matica. Pre človeka je jednoduché od pohľadu rozpoznať jednotlivé črty tváre, no počítač musí vykonávať komplexné kalkulácie na týchto maticiach, aby bol schopný replikovať túto prirodzenú ľudskú schopnosť a určiť, či sa na obrázku skutočne nachádza ľudská tvár. [5]

Základom každého odvetvia strojového učenia je veľké množstvo dát. Vybraný algoritmus cyklicky iteruje nad dátami, pokiaľ nie je schopný rozoznávať rozdiely v jednotlivých častiach obrazu a v konečnej fáze rozpoznávať celé obrázky. Ak by bola požiadavka natrénovať počítač tak, aby bol schopný rozoznávať napríklad pneumatiky, bolo by nutné poskytnúť značné množstvo obrázkov pneumatík rôznych tvarov a veľkostí, aby bol v konečnom dôsledku schopný rozoznať defekty a iné žiaduce (alebo nežiaduce) vlastnosti konkrétnej pneumatiky.

K tomu, aby sme dosiahli takýto žiadúci výsledok sú bežne používané dve techniky:

- Deep learning
- Konvolučná neurónová sieť

1.1.1 Deep learning

Deep learning je podmnožinou machine learningu inšpirovaná činnosťou a štruktúrou ľudského mozgu. Tieto algoritmy sa snažia dôjsť k podobným záverom, ako ľudia kontinuálnou analýzou s danou logickou štruktúrou. Aby toho bol deep learning schopný dosiahnuť, používa viacvrstvovú štruktúru algoritmov, zvanú *neurónové siete.*

Ich návrh je založený na štruktúre ľudského mozgu. Rovnako, ako naše mozgy určujú vzory a klasifikujú rôzne typy informácii, neurónové siete sú schopné vykonávať rovnaké úlohy na dátach. Individuálne vrstvy neurónových sietí môžeme považovať za určitý druh filtra. Podobne pracuje náš mozog. Vždy, keď získame novú informáciu, snažíme sa ju porovnať so známymi objektami. Rovnaký koncept používajú i neurónové siete.

Umožňujú vykonávať základné úkony, ako clustering, klasifikácia a regresia, vďaka ktorým môžeme zoskupiť alebo zoradiť neoznačené dáta na základe podobností jednotlivých vzoriek, alebo v prípade klasifikácie vytrénovať sieť na označenom datasete za účelom klasifikácie vzoriek do rôznych kategórii. Vo všeobecnosti môžme tvrdiť, že neurónové siete dokážu vykonávať rovnáke úkony ako klasické machine learning algoritmy, avšak nie naopak.

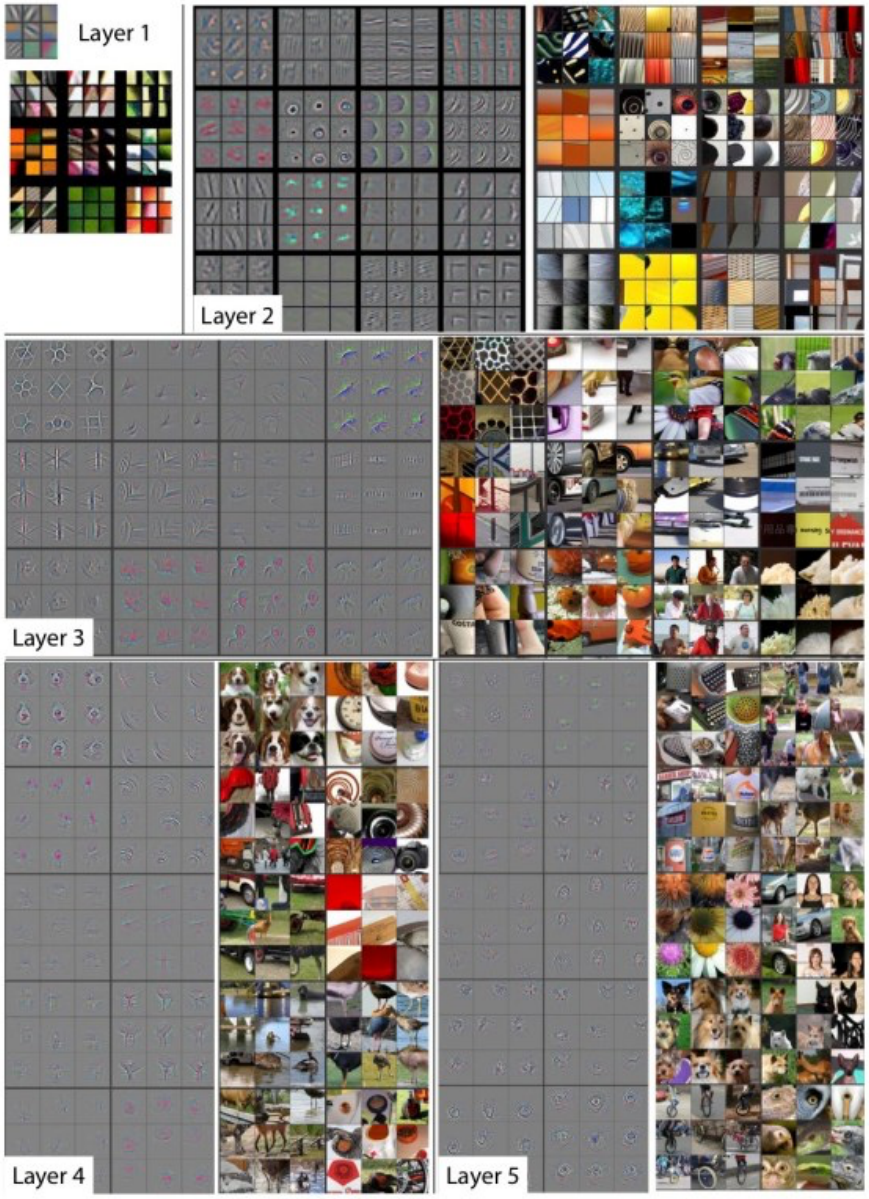
Mnohým pokrokom na poli umelej inteligencie môžme vďačiť práve deep learningu, bez ktorého by neboli možné technológie použité v inteligentných asistentoch Alexa a Siri, no napríklad ani Google Translate by nedosahoval dnešnej úrovne presnosti prekladu bez kontextu. [2]

1.1.2 Konvolučné neurónové siete (CNN)

Je to taký algoritmus, ktorý dostáva vstup v podobe obrázka, automaticky priradí dôležitosť (váhy a bias) rôznym aspektom alebo objektom na obrázku a zároveň je schopný ich rozlišovať. Predspracovanie vyžadované pre použitie CNN je výrazne nižšie v porovnaní s ostatnými klasifikačnými algoritmami. Zatiaľ čo v primitívnejších metódach sú filtre generované ručne, s dostatkom tréningu si dokážu CNN tieto filtre vytvoriť samé.

Architektonicky sú analogické so spojmi v mozgu – neurónmi. Jednotlivé neuróny odpovedajú na stimuly iba v určitých regiónoch vizuálneho poľa zvaného receptívne pole. [3]

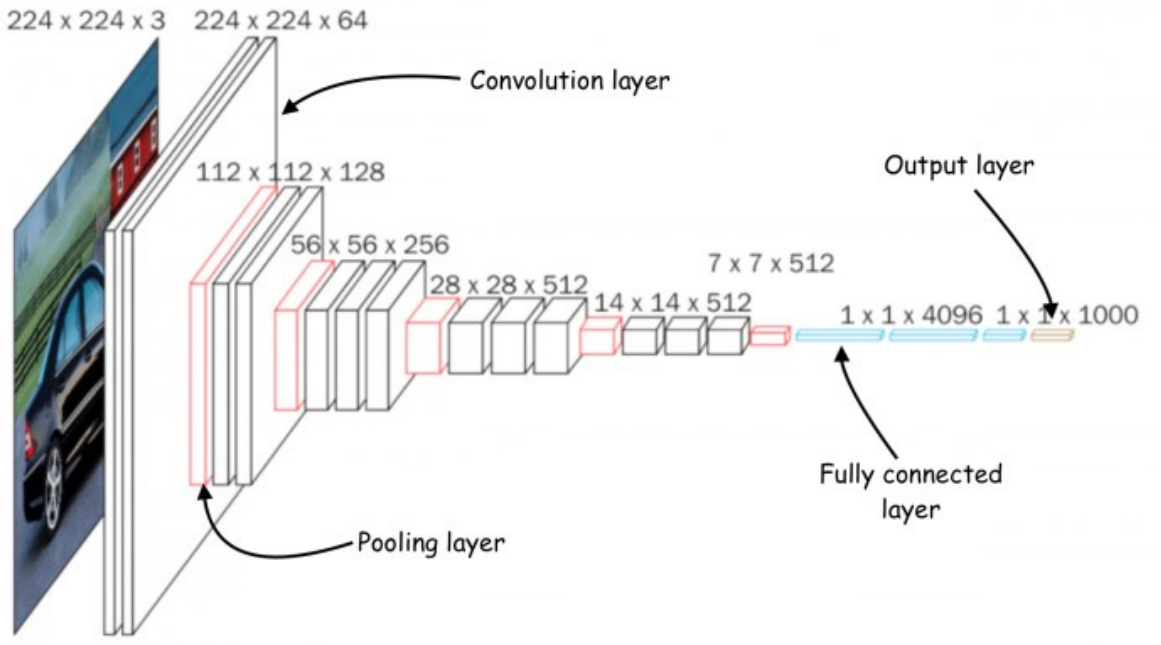
Každá CNN sa skladá z jednej alebo viac *konvolučných vrstiev*, teda softvérového komponentu extrahujúceho zmysluplné hodnoty z obrázka. Skladá sa z niekoľkých filtrov (v tvare štvorcových matíc), ktoré prechádzajú obrázkom a registrujú vážený súčet pixelových hodnôt v rôznych lokáciách. Každý filter má rozdielne hodnoty a extrahuje rozličné vlastnosti z obrázka. Výstupom konvolučnej vrstvy je sada „feature máp“. Položené na seba dokážu detekovať hierarchiu vizuálnych vzorov. Napríklad, nižšie vrstvy budú produkovať feature mapy pre vertikálne a horizontálne okraje a ostatné jednoduché vzory. Ďalšie vrstvy môžu detekovať komplexnejšie vzory, ako kruhy až po konkrétne objekty ako autá, domy, ľudí atď. [7]



Obrázok 2 Ukážka jednotlivých konvolučných vrstiev [7]

Väčšina CNN používa taktiež tzv. *Pooling vrstvy*, aby graduálne zredukovali veľkosť ich feature máp a zanechali len najvýraznejšie časti. Existuje niekoľko typov poolingu, medzi nimi najrozšírenejší *Max-pooling*, ktorý uchováva maximálnu hodnotu v určitom zhluku pixelov. Napríklad, ak sa použije poolingová vrstva o veľkosti 2, vezme zhluky pixelov o veľkosti 2x2 z existujúcich feature máp a zanecháva iba najvyššiu pixelovú hodnotu. Táto oprácia znižuje veľkosť máp o polovicu a zanecháva najrelevantnejšie vlastnosti. Umožňujú teda generalizovať ich možnosti. [7]

Výstup konvolučných vrstiev je sploštený do jednodimenzionálnej matice, ktorá je numerickou reprezentáciou vlastností obsiahnutých v obrázku. Tá potom slúži ako vstup pre tzv. *fully-connected layers*, umelých neurónov, ktoré mapujú jednotlivé vlastnosti na druh výstupu očakávanom od neurónovej siete. [7]



Obrázok 3 Prehľad štruktúry typickej konvolučnej siete

1.2 Praktické využitie computer vision

Existuje mnoho aplikácii demonštrujúce užitočnosť v mnohých priemyselných odvetviach, ako je biznis, zábavný priemysel, logistika, zdravotníctvo, ale aj bežný život. Kľúčom k úspechu týchto aplikácii je pomyselná záplava vizuálnych informácii z rôznych zdrojov, ako sú smartfóny a bezpečnostné systémy. [4]

1.2.1 Computer vision v rámci manufaktúry

Aplikované najmä pri inšpekciách, kontrole kvality, vzdialenom monitoringu a systémovej automatizácii. Často sa sleduje dopad na zmenu na pracovisku, napríklad akým spôsobom zamestnanci trávia svoj čas a spotrebúvajú zdroje alebo implementujú rôzne nástroje. Na základe výsledných dát je možné ďalej optimalizovať time management, kolaboráciu na pracovisku a v neposlednom rade produktivitu zamenstnancov.

Mimo zlepšenie efektivity na pracovisku sú manufaktúry zamerané aj na bezpečnosť, v rámci ktorej môže computer vision pomôcť pri kontrole vybavenia zamestnancov, ako sú helmy a reflexné vesty a teda pomáhať pri dohľade nad dodržiavaním bezpečnostných protokolov.

Chytré kamerové aplikácie poskytujú škálovateľnú metódu implementácie automatizovanej vizuálnej inšpekcie a kontroly kvality produkčných procesov výrobných liniek za použitia real-time objektovej detekcie. Poskytujú tak častokrát presnejšie výsledky než manuálna inšpekcia. [4]

1.2.2 Computer vision v zdravotníctve

Strojové učenie je využívané v zdravotníckom priemysle za účelom detekcie ochorení, ako napr. rakoviny prsníka a kože. Image recognition umožňuje vedcom detekovať nepatrné rozdiely v jednotlivých obrázkoch, čím vo významnej miere napomáha ku včasnej a presnej diagnóze, ktorá je pri liečbe rakoviny kritická.

Mimo diagnózy rakoviny je možné diagnostikovať i COVID-19. Existuje niekoľko deep learning modelov založených na rontgenovej diagnóze. Medzi ďalšie praktické aplikácie patrí:

- *Analýza pohybu:* Neurologické a svalové ochorenia, ako mŕtvica, problémy s rovnováhou pri chôdzi môžu byť detekovatelné pomocou deep learning modelov a computer vision aj bez doktorskej analýzy. Aplikácie špecializujúce sa na odhad pózy človeka taktiež pomáhajú doktorom pri diagnóze.

- *Detekcia zakrytia horných dýchacích ciest***:** Základným nástrojom v boji s virulentným ochorením je ochrana horných dýchacích ciest v podobe respirátora, či rúšky, ktoré významnou mierou znižujú riziko nákazy. Za týmto účelom firmy ako *Uber* implementovali do svojich mobilných aplikácii detekciu ochranných masiek, zvyšujúc tak ochranu svojich vodičov pred potenciálnym nakazením od zákazníkov, s ktorými sa nachádzajú v bezprostrednom kontakte. [4]

1.2.3 Computer vision v poľnohospodárstve

Monitorovanie zvierat je kľúčovou stratégiou chytrého farmárčenia. Strojové učenie využíva živý kamerový záznam pre monitorovanie zdravia dobytku. Účelom je analýza zvieracieho chovania pre zvýšenie produktivity, zdravia a celkového životbytia a v konečnom dôsledku teda zvýšenie zárobku.

Obrázok 4 Príklad využitia computer vision v poľnohospodárstve [4]

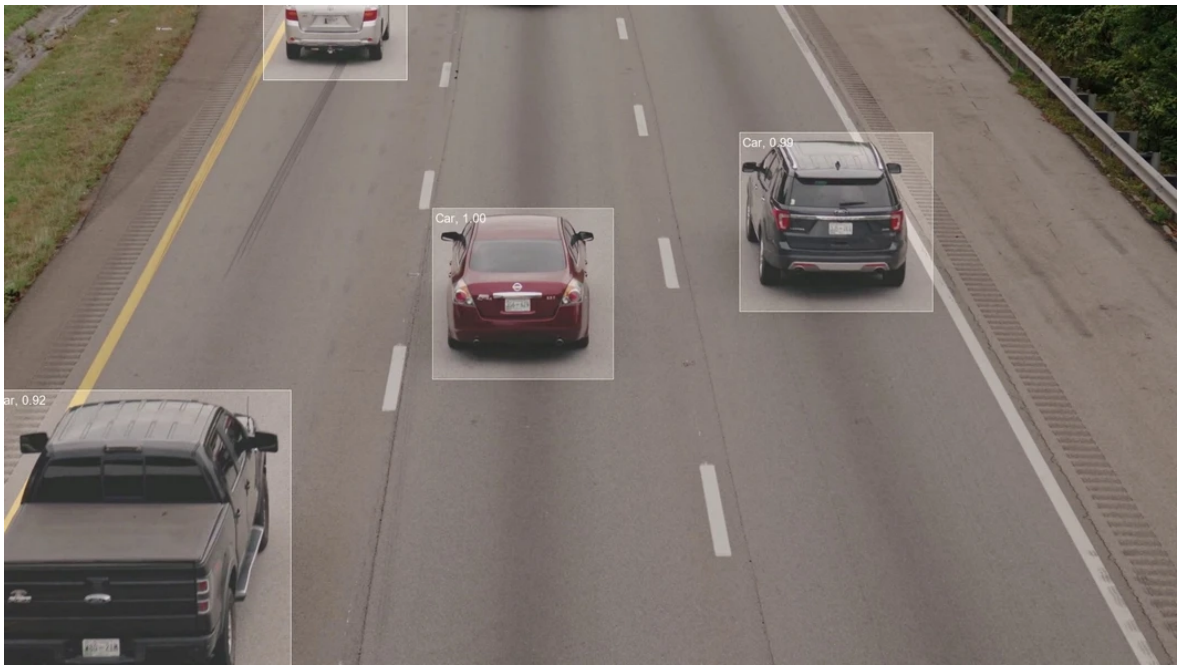
Computer vision v tomto odvetví sa však nezaoberá iba zvieratami, no i poľnohospodárskymi plodinami. Typicky je monitorovanie kvality úrody vykonávané manuálne a spolieha sa na subjektívny odhad človeka, ktorý nemusí byť presný. Computer vision umožňuje kontinuálne a nedeštruktívne monitorovanie rastu obilnín a odporučiť tak vhodné živiny pre ich optimálny rast. V porovnaní s manuálnou kontrolou dokáže rea-time monitoring detekovať nedostatok živín oveľa skôr, vďaka malým nejasnostiam a rozdielom.

Rast a veľkosť úrody však v poľnohospodárstve nie je jedinou výzvou, ktorej farmári musia čeliť. Početné vrásky spôsobuje aj napadnutie škodcami. Rýchle a presné rozpoznávanie druhu a počtu lietajúcich škodcov je významné pre udržanie zdravia celej plantáže.

Mimo škodcov napádajú rastliny i choroby. Ich skoré rozpoznanie je kritické pri odhadovaní závažnosti choroby a potenciálnej straty úrody. Vďaka konvolučným sieťam bolo vyvinutých niekoľko aplikácii, napríklad pre identifikáciu hniloby jabĺk v dôsledku hubovej infekcie (black rot). [4]

1.2.4 Computer vision v doprave

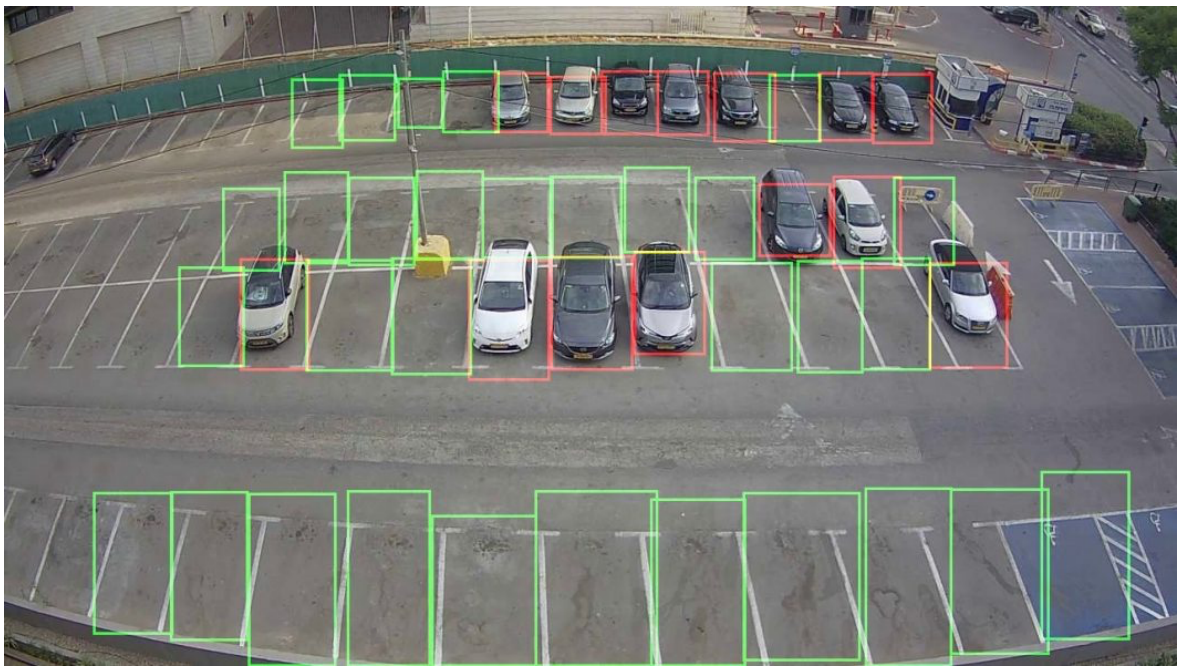
Jedná sa asi o najznámejšie priemyselné využitie computer vision s dlhou históriou. Technológie pre automatizovanú klasifikáciu počtu vozidiel boli vyvíjané po dekády. Deep learning metódy umožňujú implementovať vysoko profilovú analýzu dopravy na bežných a lacných bezpečnostných kamerách. S rapídnym nárastom jednoducho dostupných senzorov ako „closed-circuit television“ kamery, LIDAR a dokonca pre termálne zobrazovanie je možné detekovať, sledovať a kategorizovať vozidlá simultánne v rôznych dopravných pásoch. Kombináciou týchto senzorov je možné dosiahnuť zlepšenie klasifikácie.



Obrázok 5 computer vision v doprave [4]

Detekcia dopravných priestupkov: orgány činné v trestnom konaní nasadzujú kamerové systémy schopné rozpoznávať typ priestupku vodiča, za účelom zníženia nebezpečného chovania na cestách. Pravdepodobne najkritickejšou aplikáciou je detekcia zastavených vozidiel v nebezpečných oblastiach, ako sú dialnice. Za zmienku taktiež stoja iniciatívy chytrých miest, pri ktorých je zahrnutá automatizovaná detekcia typu priestupku, ako zvýšená rýchlosť, jazda na červenú, jazda v protismere alebo nepovolené zatáčky.

Detekcia obsadenosti parkovacích miest: Opäť témou chytrých miest, computer vision poháňa decentralizované a efektívne riešenia pre vizuálnu obsadenosť parkovacích miest založenú na CNN. Existuje niekoľko datasetov pre tento typ detekcie, medzi najznámejšie môžeme radiť PKLot a CNRPark-EXT. Medzi výhody oproti tradičným riešeniam patrí jednoduchá škálovateľnosť, lacná údržba a inštalácia, najmä z hľadiska možnosti znovu použitia starších bezpečnostných kamier.



Obrázok 6 computer vision pri detekcii parkovacích miest [4]

Detekcia pri autonómnych autách: Aplikácie computer vision sú hojne využívané pri detekovaní dopravných značiek a chodcov, predchádzaní kolízii, monitorovaní podmienok premávky a mnohých ďalších.
Z hľadiska bezpečnosti je detekcia chodcov najkritickejším bodom, keďže sa potenciálne jedná o ľudský život. Zahŕňa v sebe škálu senzorov, od tradičných CCTV alebo IP kamier, infračervených zariadení a RGB kamier. Algoritmus detekcie osôb je možné založiť na infračervených stopách, špecifických tvaroch, gradientoch, alebo charakteristík pohybu. [4]

Detekcia pozornosti vodiča: Aplikácii detekcie je viac, počnúc mikrospánkom, cez používanie mobilného zariadenia až po jednoduché neudržiavanie pozornosti. Toto všetko sú faktory, ktoré významne vplývajú na celosvetovú mieru nehodovosti. Umelá inteligencia je používaná práve pre sledovanie očí vodiča a na základe naučeného modelu je schopná

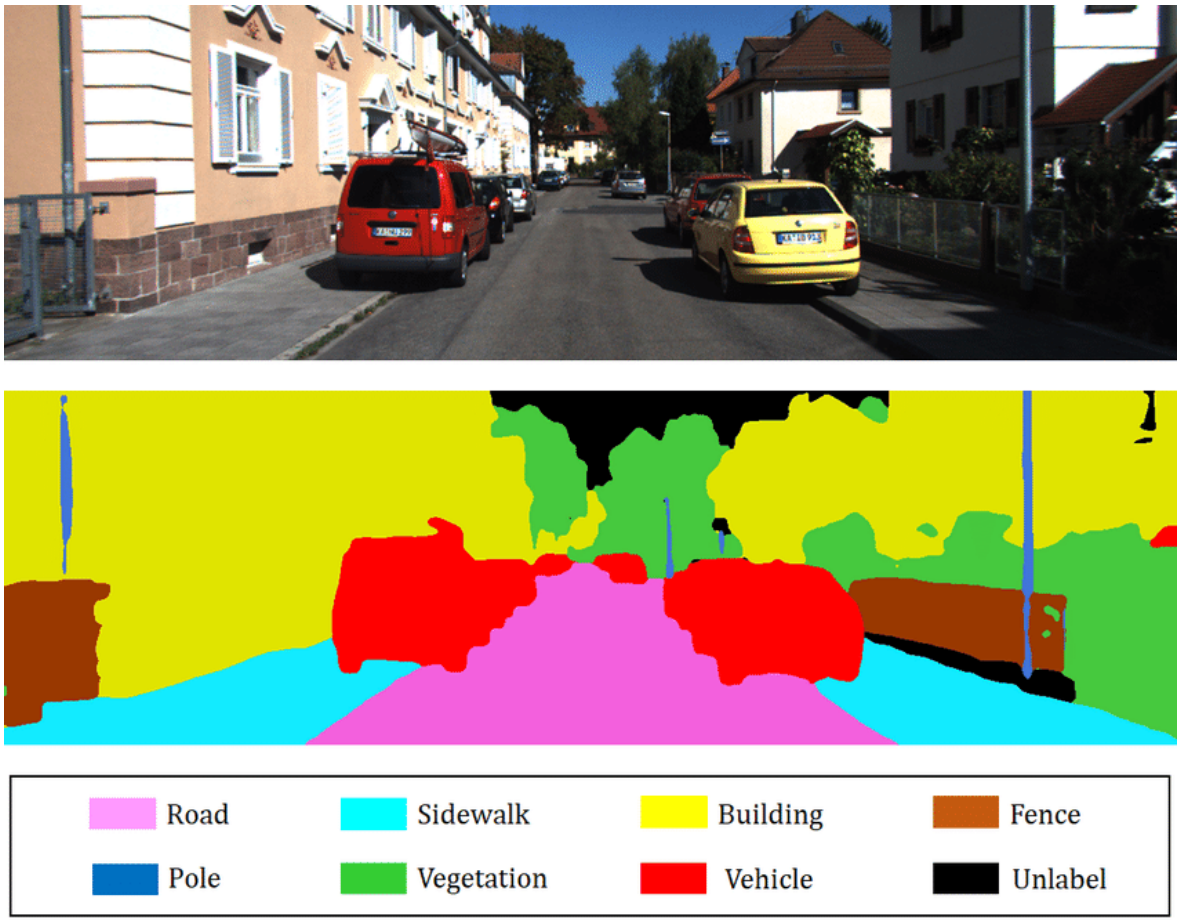
predpovedať mieru rizika v danom momente a pomocou inteligentných systémov v aute dokáže vodiča na túto skutočnosti upozorniť, či už zvukovými alebo vibračnými podnetmi. [4]

1.3 Kategórie computer vision

Computer vision sa v posledných rokoch rozšírilo do mnohých odvetví. Tieto aplikácie však nie je možné generalizovať. Existuje niekoľko kategórii computer vision, ktoré ujasňujú účel danej aplikácie. Niektoré z kategórii sú:

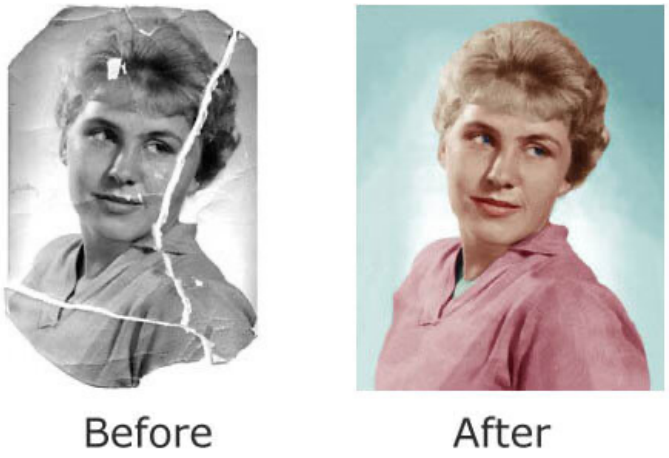
- *Image classification:* Najzákladnejšia úloha computer vision, algoritmus je schopný "vidieť" obrázok a klasifikovať ho. Je na obrázku pes? Mačka, alebo jablko? Presnejšie, je schopný s určitou presnosťou predpovedať, že daný obrázok patrí pod určitú triedu. Napríklad pri sociálnych sieťach je kritické vedieť automaticky identifikovať a segregovať obsah nahraný užívateľmi, prinášajúc so sebou vyšší engagement od užívateľa, no zároveň aj zahladenie bezpečnosntých rizík. [1]
- *Object detection:* Úzko prepojené s image classification pre identifikáciu určitej triedy a následné detekovanie jej pozície v danom obrázku alebo videu. Medzi príklady patrí detekcia nepodarkov na výrobnej linke, prípadne detekcia strojov potrebujúcich údržbu. [1]
- *Object tracking:* Sleduje detekovaný objekt. Táto úloha je často vykonávaná so sekvenčne zachytenými obrázkami alebo videom. Pri autonómnych vozidlách je nutné klasifikovať nielen objekty ako chodci, dopravné značky a stav vozovky, ale i sledovať ich v pohybe, aby boli schopné aktívne sa vyhýbať kolíziám a dodržovať dopravné zákony. [1]
- *Content-based image retrieval:* Používa computer vision pre prehliadanie, vyhľadávanie a získavanie obrázkov z rozsiahlych dátových úložísk na základe obsahu obrázkov než metadát, ktoré sú s nimi asociované. Táto úloha v sebe inkorporuje automatickú anotáciu obrázkov, ktorá nahrádza manuálne označovanie. Táto kategória úloh sa hojne využíva pri systémoch pre správu digitálnych zdrojov a môžu zvýšiť presnosť ich získavania a vyhľadávania. [1]
- *Image segmentation:* Rozdelenie obrázku do podčastí alebo podobjektov, za účelom demonštrácie schopnosti stroja rozlíšiť objekt od pozadia, prípadne objekt od ďalšieho objektu v rámci toho istého obrázka. Segment obrázka reprezentuje

špecifickú triedu objektu, ktorú neurónová sieť dokázala na obrázku identifikovať. [5]



Obrázok 7 Segmentácia obrazu [5]

- *Image restoration:* Používaná k obnovovaniu alebo rekonštrukcii zašlého, alebo starého obrázku a starých fyzických obrázkov, ktoré boli nevhodne uskladňované, čo malo za výsledok stratu kvality. Typický image restoration proces zahŕňa redukciu aditívneho šumu pomocou matematických nástrojov. Samotná rekonštrukcia môže vyžadovať výrazný zásah do pôvodného obrázka, tzv. *„image inpainting“,* v ktorom sú poškodené časti obrázka vyplnené pomocou generatívnych modelov, ktoré odhadujú, čo sa snaží obrázok zachytiť. Obnovovací proces je často nasledovaný kolorizáciou daného subjektu na obrázku v prípade, že je obrázok čiernobiely. [5]



Obrázok 8 Výsledok image restoration [5]

- *Pose estimation:* Spôsob, akým je možné odhadnúť pozíciu a orientáciu kĺbov ľudského tela. Použitie tejto techniky je najmä v kombinácii s AR/VR aplikáciami, hrami, športe, no i v módnom priemysle. [6]

2 OBJECT DETECTION PRINCÍPY A WORKFLOW

Object detection (OD) je dôsledná technika CV sústreďujúca sa na identifikáciu a označovanie objektov v obrázkoch a videách, no i v rea-time zázname. Object detection modely sú trénované veľkým množstvom anotovaných dát, aby mohli jednotlivé objekty účinne detekovať. Kľúčovým komponentom je tzv. „*bounding box*“ identifikujúci okraje rozpoznaného objektu. Typicky má tvar štvorca alebo obdĺžnika. [9]

2.1 Datasety

Sieť určená pre image classification dokáže určiť, či sa na obrázku nachádza určitý objekt, alebo nie. Nepovie však, kde sa daný objekt na obrázku nachádza. OD siete poskytujú teda nielen triedu objektov obsiahnutých v obrázku, ale aj ich bounding box poskytujúci súradnice objektu. OD siete sú veľmi podobné sieťam pre image classification a používajú konvolučné vrstvy pre detekciu vizuálnych vlastností. Vo svojej podstate väčšina takýchto sietí používa konvolučnú sieť pre image classification a poupraví ich pre object detection.

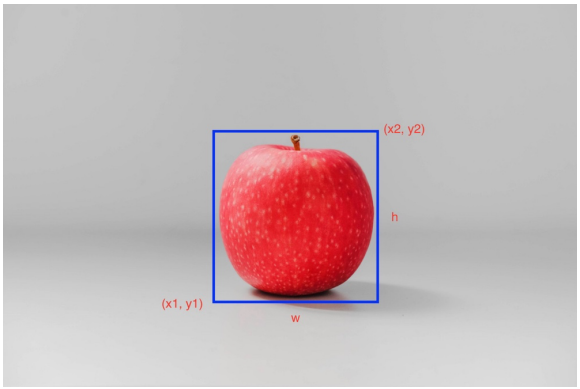
Keďže sa tradične jedná o učenie s učiteľom, je nutné modely trénovať na označených príkladoch. Každý obrázok v trénovacom datasete musí obsahovať sprievodný súbor, obsahujúci ohraničenia a triedy konkrétnych objektov, ktoré chceme detekovať. Takýto model je potom trénovaný dovtedy, kým nie je schopný nájsť regióny v obrázku spadajúce pod každý druh požadovaného objektu. [10]

Nevýhodou je skutočnosť, že sa jedná o manuálny, časovo náročný proces, preto sa odporúča použiť open source dataset. [11]

2.1.1 Typy anotácii

Obrázky používané pre trénovanie, validáciu a testovanie CV algoritmov majú významný dopad na úspech AI projektu. Každý obrázok v datasete musí byť pečlivo a presne označený, aby bol systém schopný rozoznávať objekty podobným spôsobom, ako človek. Čím vyššia kvalita anotácii, tým je výsledný model kvalitnejší a teda presnejší. Objem a rôznorodosť dát, ktoré je nutné označiť podľa špecifík dokáže byť výzvou, ktorá spomaľuje projekt a teda aj rýchlosť, akou dokážeme potenciálny produkt dostať na trh. Existuje viacero spôsobov, ako anotovať dáta, medzni nimi napríklad: [12]

- *Bounding boxes:* sú najbežnejším typom anotácie v CV. Majú štvorcový tvar používaný k definovaniu lokácie cieľového objektu. Sú určené *x* a *y* koordinátmi na dvojdimenzionálnej osi a sú všeobecne používané pri detekčných a lokalizačných úlohách. [11]

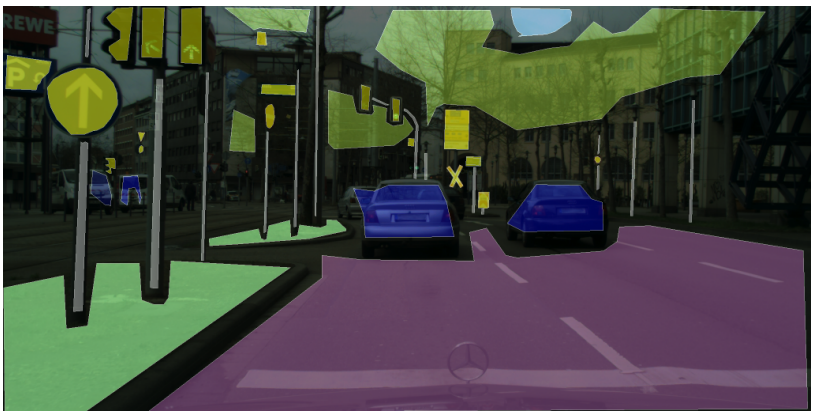


Obrázok 9 Klasické bounding boxy [11]

- *Polygonálna segmentácia:* Používaná pre objekty, ktoré nie sú vždy štvorcového tvaru. Berúc túto myšlienku v úvahu je polygonálna segmentácia Ďalším typom anotovania dát, kde sú použité komplexné polygóny namiesto štvorcov pre definovanie tvaru a lokácie objektov oveľa precíznejším spôsobom. [11]

Obrázok 10 Polygonálna segmentácia [11]

- *Sémantická segmentácia:* Jedná sa o anotáciu jednotlivých pixelov, kde každému z nich je priradená trieda. Tieto triedy môžu byť napríklad chodec, auto, autobus, cesta, prechod atď. Každý pixel v sebe teda nesie sémantický význam. Primárne je používaná v prípadoch, kedy je dôležitý environmentálny kontext. Tento spôsob sa hojne využíva pri učení autonómnych áut. [11]



Obrázok 11 Sémantická segmentácia [11]

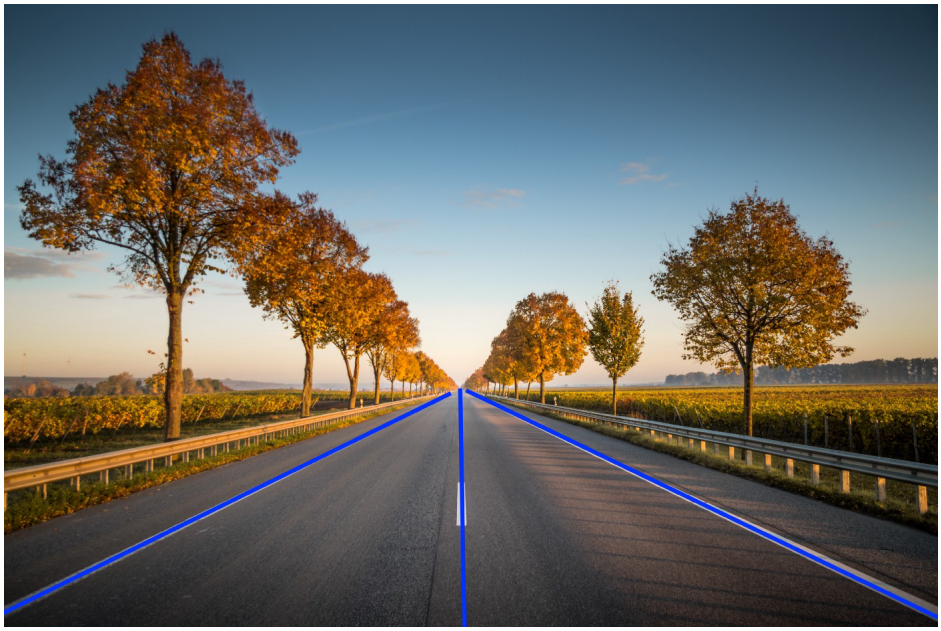
- *3D kuboidy:* Podobné bounding boxom, no s pridanou informáciou o hĺbke a teda je možné získať 3D reprezentáciu objektu umožňujúcu systémom rozoznávať vlastnosti ako objem a pozícia v 3D priestore. Použitie je opäť pri autonómnych autách, kde môžu takéto systémy použiť informáciu o hĺbke pre výpočet bezpečnej vzdialenosti od ďalšieho auta. [11]

Obrázok 12 Ohraničenie pomocou 3D kuboidov [11]

- *Key-Point anotácia:* Slúži pre detekciu malých objektov a tvarov na obrázku. Tento typ anotácie je užitočný pre detekciu čŕt tváre, výrazov, emócii, častí ľudského tela a póz. [11]

Obrázok 13 Key-point anotácia [11]

- *Line annotation:* Ako vyplýva z názvu, tento typ anotácie je vytvorený použitím liniek. Bežne používaný pri autonómnych vozidlách pre rozpoznávanie a detekciu jazdných pruhov. [11]



Obrázok 14 Line anotácia [11]

2.1.2 Anotačné formáty

Pri anotovaní môžeme vyberať hneď z niekoľkých formátov, keďže neexistuje jeden štandardizovaný. Medzi tie najrozšírenejšie patrí:

- *COCO:* Obsahujúci päť anotačných typov pre: object detection, keypoint detection, stuff segmentation, panoptic segmentation a image captioning. Anotácie samotné sú uložené vo formáte JSON. Pre object detection používa COCO nasledujúci formát:

```
annotation{
"id" : int,
"image_id": int,
"category_id": int,
"segmentation": RLE or [polygon],
"area": float,
"bbox": [x,y,width,height],
"iscrowd": 0 or 1,
}
categories[{
"id": int,
"name": str,
"supercategory": str,
}]
```

Obrázok 15 COCO anotačný formát [11]

- *Pascal VOC:* Ukladá anotácie v XML súbore [11]

```
<annotation>
  <folder>Train</folder>
  <filename>01.png</filename>
  <path>/path/Train/01.png</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>224</width>
    <height>224</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>36</name>
    <pose>Frontal</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <occluded>0</occluded>
    <bndbox>
      <xmin>90</xmin>
      <xmax>190</xmax>
      <ymin>54</ymin>
      <ymax>70</ymax>
    </bndbox>
  </object>
</annotation>
```

Obrázok 16 Anotačný formát Pascal VOC [11]

- *YOLO:* Pre každý obrázok je vytvorený *.txt* súbor obsahujúci anotácie ako *object class, object coordinates*, výška a šírka. Všeobecný zápis je: <*object-class*> <*x*> <*y*> <*width*> <*height*> [11]

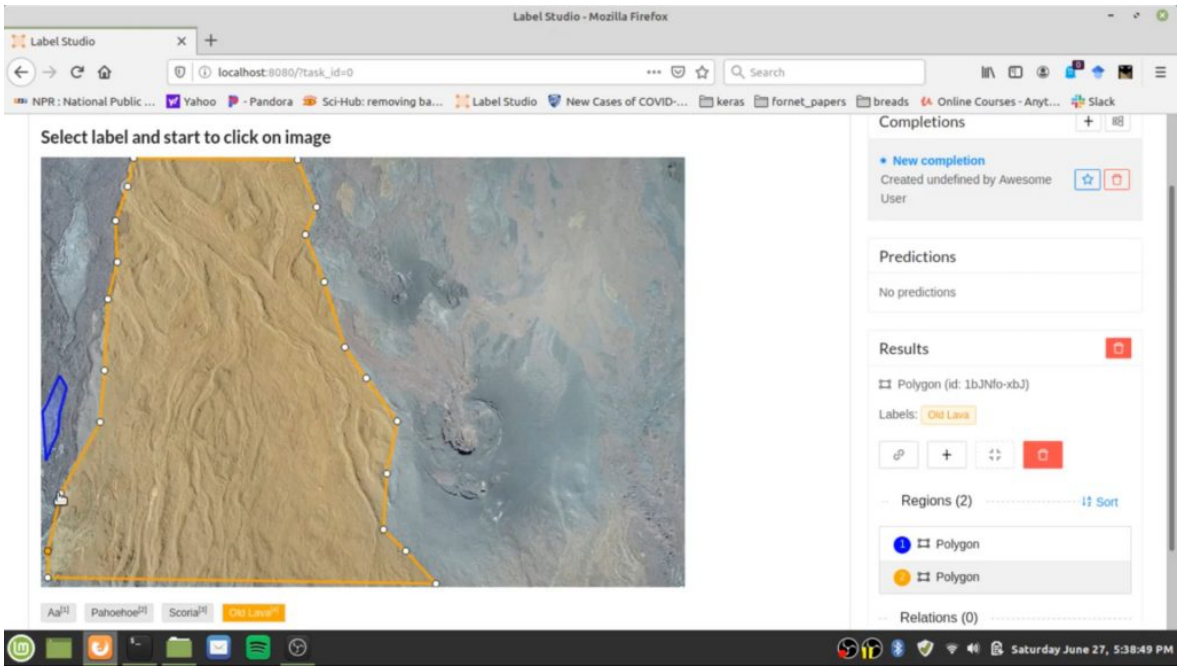
2.1.3 Anotačné nástroje

Nové machine learning platformy sľubujúce inovatívne riešenia, rýchlejší labeling alebo vyššiu presnosť vznikajú každých pár mesiacov a je veľmi jednoduché zmiasť sa už pri výbere. Optimalizácia procesu anotácie dát je však kľúčová pre zabezpečenie vysokej miery spoľahlivosti výsledného modelu. Výber správneho nástroja preto nie je možné vziať na ľahkú váhu. [13]

Obrázky je možné anotovať niekoľkými spôsobmi, od komerčne dostupných, cez open source až po freeware riešenia. Tieto nástroje v sebe kombinujú širokú škálu anotačných možností a exportných formátov opísaných vyššie. [14]

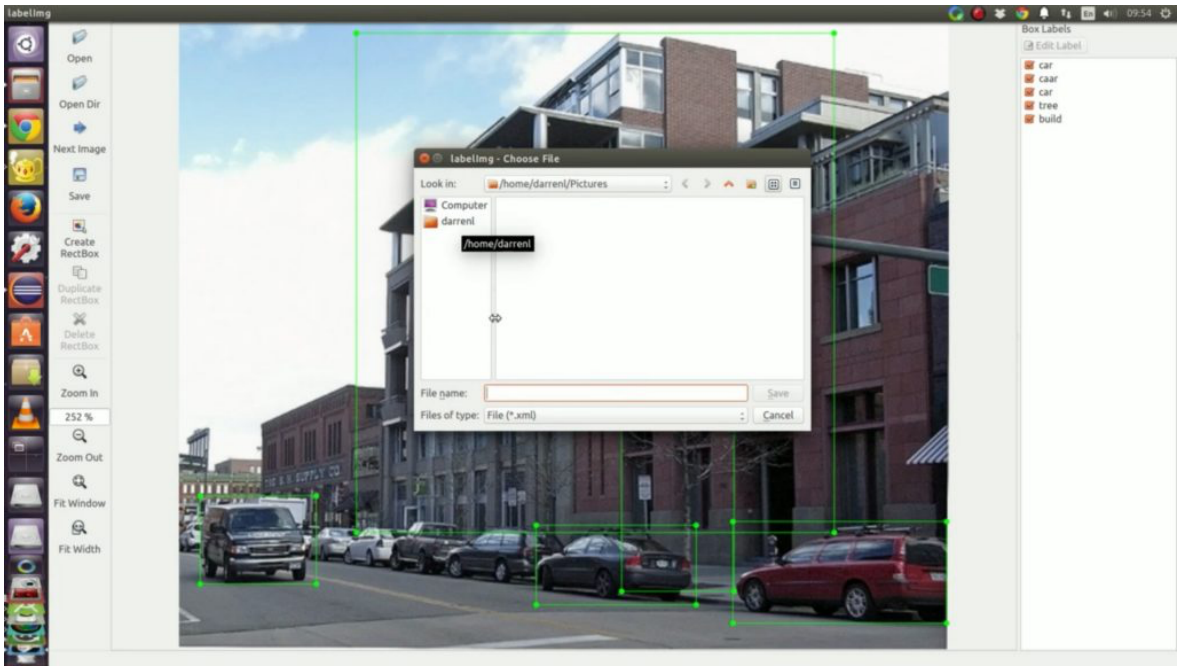
Medzi populárne nástroje patrí:

- *Label Studio:* Open source nástroj od spoločnosti Heartex Inc. Podporuje širokú škálu anotácii vrátane image classification, object detection a semantic segmentation. Je možné anotovať takmer každý typ dát, ako napríklad audio, obrázky, text a dokonca i HTML. Disponuje unikátnou konfiguráciou zvanou "*Labeling Config*", umožňujúcou prispôsobenie si UI na základe požiadaviek užívateľa. Taktiež má širokú škálu algoritmizovaných automatizačných vlastností vrátane *pre-labelingu*, ktorá dokáže anotovať dáta na základe existujúceho machine learning modelu. [15]



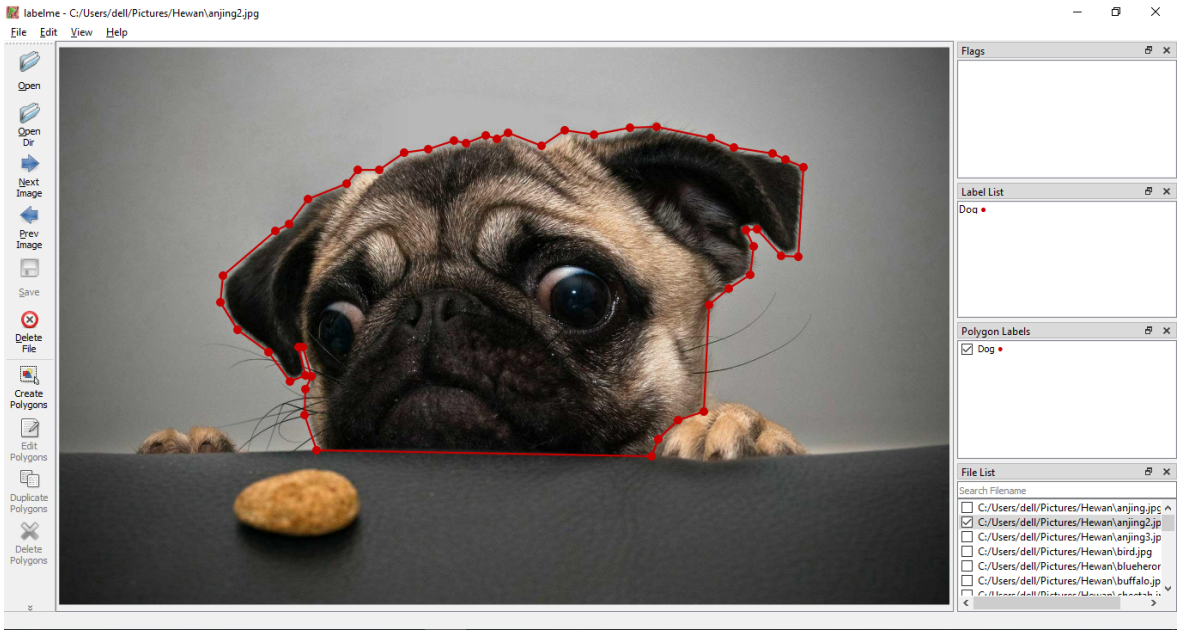
Obrázok 17 UI Label Studia [15]

- *LabelImg:* Je ďalšou populárnou open source možnosťou pre anotovanie obrázkov vďaka jej jednoduchým a intuitívnym UI s možnosťou práce offline. Aplikácia je multiplatformná a teda podporovaná na systémoch Windows, Linux a MacOS a v rámci virtualizácie može byť spustená aj pomocou Anacondy či Dockeru. Nevýhodou však je podpora anotácie len pomocou bounding boxov, ktorá je vhodná len pre jednoduchšie projekty a nemusí stačiť pre rozsiahlejšie a komplexnejšie modely. Export anotácie je podporovaný v Pascal, Yolo a CreateML formátoch. [15]



Obrázok 18 UI LabelImg [15]

- *Labelme:* Pravdepodobne najpoužívanejší nástroj. Podobne ako pri LabelImg je jednoduchá nielen inštalácie ale aj samotné používanie. Na rozdiel však od LabelImg disponuje Labelme funkciami, ktoré z neho robia vhodnejšieho kandidáta pre každodenné použitie. Medzi tieto funkcie patrí napríklad tzv. *„File List“*, zobrazujúci všetky dostupné dáta a stav ich anotácie. V Labelme si môže užívateľ vybrať zo šiestich základných typov počnúc polygonom, obdĺžnikom, štvorcom, kruhom, čiarou, bodom a linkou. Umožňuje tak flexibilitu anotácie, pričom zachováva jednoduchosť používania. Za nevýhodu je možné považovať export, ktorý je možný iba v JSON formáte. [14]

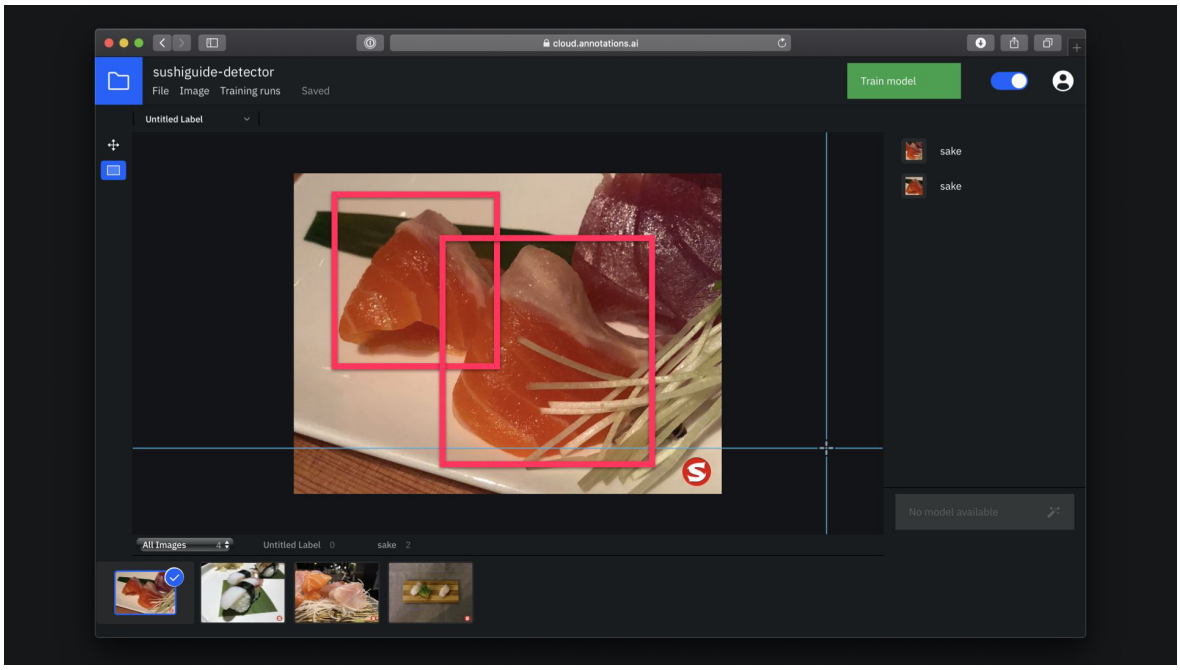


Obrázok 19 UI Labelme [14]

- *COCO Annotator:* Vytvorený pomocou front-end frameworku Vue.js je webová aplikácia sústreďujúca sa na segmentáciu. Pomáha pri vývoji a trénovaní object detection a lokalizačných modelov. Samotnú anotáciu je možné vykonať pomocou kriviek, polygonov a kľúčových bodov, označujúc tak segmenty obrázka. Aplikácie exportuje do COCO formátu. Taktiež podporuje anotovanie obrázkov na základe predtrénovaných modelov. [15]

Obrázok 20 UI Coco annotator [15]

- *IBM Cloud annotations:* Poskytuje rýchle a jednoduché anotácie obrázkov v prehliadači. Taktiež podporuje trénovanie modelov v cloude, ako i export datasetov pre CreateML vo formáte JSON. Anotovanie prebieha za pomoci bounding boxov okolo objektu. [16]



Obrázok 21 UI IBM Cloud annotations [16]

2.2 Object detection metódy a algoritmy

Object detection by bolo nemožné bez modelov navrhnutých špecificky pre tento úkon. Sú trénované stovkami až tisíckami vizuálnych dát pre optimalizáciu detekčnej presnosti na automatickej báze. Trénovanie a vylepšovanie modelov sa vykonáva na základe dostupných datasetov. Existuje niekoľko prominentných typov object detection algoritmov a metód. [8]

2.2.1 R-CNN, Fast R-CNN a Faster R-CNN

Prvou veľmi úspešnou skupinou metód bola R-CNN (Region-Based Convolutional Neural Network), navrhnutá v roku 2014. Svojich predchodcov prekonala extrahovaním iba 2000 regiónov z obrázka, ktoré sa nazývajú *„region proposals“*. Predtým, ako sa začala používať RNN bolo týchto regiónov nutné extrahovať výrazne viac. Princíp je nasledovný: výber obrázka na vstupe, z ktorého sa extrahuje 2000 regiónov. Z každého individuálneho regiónu sa následne vyberú features, ktoré budú klasifikované pod určitou triedou. [17]

R-CNN má však aj niektoré nedostatky:

- Vytrénovať sieť si vyžaduje obrovské množstvo času, keďže na jeden obrázok je nutné klasifikovať 2000 návrhov regiónov. [18]
- Nie je možné ho implementovať v reálnom čase, keďže detekcia trvá zhruba 47 sekúnd pre každý testovací obrázok. [18]

- R-CNN závisí na Selective Search algoritme pre generovanie region proposals. Táto operácia je opäť časovo náročná a samotný algoritmus nie je možné prispôsobiť detekčnému problému. [18]

Z tohto dôvodu vzniklo rýchlejšie *Fast R-CNN*. Nielen proces detekcie objektov s obrovským počtom regiónov je časovo náročný, ale i trénovanie CNN s takým množstvom regiónov. Fast R-CNN výrazne skrátilo čas nutný pre spraovanie vložením obrázkov do vopred natrénovanej CNN, aby vytvorila konvolučnú feature map, čím sa eliminuje proces rozdelenia obrázku na 2000 region proposals. Tie je možné jednoducho identifikovať z vytvorenej feature mapy, ktoré sa posielajú do poolovacej ROI vrstvy, ktorá extrahuje features z daného regiónu. Výstup z predchádzajúcej vrstvy je potom spracovaný vrstvou, kde sa model rozdelí na dva výstupy: Jeden pre predikciu triedy cez softmax vrstvu, a druhý pre predikciu bounding boxu cez lineárny výstup.
Čas potrebný na vytrénovanie pomocou Fast R-CNN oproti R-CNN klesli z ~84 hod. Na 9 hod. a testovací čas klesol z ~50 sekúnd na ~2.5 sekundy. [19]

Tretím a ešte viac vylepšeným modelom bol neskôr uvedený *Faster R-CNN*. Architektúra je podobná Fast R-CNN, no s niekoľkými významnými úpravami. Faster R-CNN nepoužíva Selective Search algoritmus založený na hierarchickom zoskupovaní medzi podobnými regiónmi. Namiesto neho sa používa Region Proposal Network umožňujúca výrazne rýchlejšiu identifikáciu regiónov. Dobu testovania dokázala oproti Faster R-CNN znížiť z ~2.5 sekúnd na ~0.2 sekundy. Vďaka tomuto je vhodnou voľbou pre real-time object detection. [20]

2.2.2 YOLO

Alebo „You Only Look Once“ je populárny algoritmus pre detekovanie objektov v reálnom čase. Proces klasifikácie i predikcie bounding boxov pre detekované objekty zjednocuje pomocou jednej neurónovej siete a ako taký je teda algoritmus výkonnostne optimalizovaný [21]. Yolo je algoritmus založený na bázi regresie. Pri práci s ním nehľadáme regióny obrázka, ktoré môžu potenciálne obsahovať hľadaný objekt. Namiesto toho sa obrázok rozdeľuje na bunky, najčastejšie 19x19. Každá bunka je zodpovedná za predikciu piatich bounding boxov (v prípade, že sa v nej nachádza viac než jeden objekt). Týmto spôsobom sa dostávame k veľkému počtu bounding boxov pre jeden obrázok. Nehľadáme presnú predikciu, ale pravdepodobnosť objektu. Tento spôsob so sebou nesie nižšiu presnosť, no pre real-time object detection je akýmsi nutným zlom. Väčšina buniek však nebude obsahovať žiadny objekt. Preto sa predikuje hodnota *pc*, ktorá slúži pre odstraňovanie boxov ako s nízkou pravdepodobnosťou, tak i s najvyšším zdieľaným priestorom. Tento proces sa nazýva non-max suppression. [22]

Obrázok 22 Odstraňovanie prebytočných predikcii pomocou non-max suppression [23]

Napriek tomu, že sa YOLO zdá byť ideálnym kandidátom pre riešenie real-time object detection problémom, nesie so sebou niekoľko limitácii. Má ťažkosti pri segregácii malých objektov, ktoré sú na obrázkoch zoskupené. Sem môžme zahrnúť napríklad prirodzene sa vyskytujúcu mravčiu kolóniu. Zároveň je z podstaty princípu pomalší ako napríklad Fast R-CNN. [23]

Existuje niekoľko variant tohto algoritmu, ktoré vznikli za posledné roky, ako napríklad:

- *YOLOv2:* Občas nazývaný tiež YOLO9000 bol vyvinutý v roku 2016 pôvodnými tvorcami Josephom Redmonom a Alim Farhadim. Názov 9 000 vznikol zo schopnosti modelu predikovať 9 000 rôznych kategórii objektov v reálnom čase. Nová verzia modelu bola simultánne trénovaná nielen na object detection a klasifikačných datasetoch, ale získala aj Darknet-19 ako nový základný model. YOLOv2 je možné považovať za obrovský úspech a stal sa najmodernejším object detection modelom, čo prirodzene zvýšilo jeho popularitu a vznikalo teda aj prirodzene viac variácii. [22]
- *YOLOv3:* Architektúra sa opäť zmenila na bázu Darknet-53 obsahujúcu 53 konvolučných vrstiev. Predikcie boli vykonávané na troch rôznych škálach, čo vylepšilo predikciu malých objektov. Táto variácia bola opäť vytvorená Josephom Redmonom, ktorý sa však rozhodol ukončiť prácu na ďalších verziách YOLO, aby zabránil svojmu potenciálne negatívnemu dopadu na svet. Dnes sa v3 používa ako základ pre nové object detection architektúry. [22]
- YOLOv4: Štvrtá verzia YOLO algoritmu bola vydaná v roku 2022 Alexeyom Bochkovskiym, Chien-Yao Wangom a Hong-Yuan Mark Liaom. Založená na SPDarknet53 architektúre. Zaujímavá je tým, že uviedla niekoľko nových konceptov ako *Weighted Residual Connections, Cross-Stage-Partial connections, mini-batch normalization* atď. Výsledkom týchto zmien je o 10% vyššia precíznosť modelu oproti v3. [22]

- *YOLOv5:* piata iterácia bola vydaná krátko po v4 Glennom Jocherom. Využívala deep learning framework PyTorch, čím vyvolala pomerne vysokú dávku kontroverzie medzi užívateľmi. V zásade sa jedná o v3 rozšírenú o spomínaný PyTorch. [22]

- *YOLOP (Single Shot Panoptic Driving Perception):* Real-time možnosti YOLO architektúry umožnili vznik odnožiam špecializujúcim sa oblasťou autonómnych vozidiel. Aby mohol systém pomáhať s navigáciou vozidla po vozovke, musí byť percepčný systém schopný extrahovať vizuálne informácie z okolia. Medzi tri najvýznamnejšie požiadavky patrí detekovanie objektov v premávke, segmentácia

oblasti, po ktorej sa môže vozidlo pohybovať a detekcia jazdných pruhov. Ku každej tejto úlohe existujú moderné algoritmy, ako napríklad Mask R-CNN, YOLOR alebo modely ako UNet a PSPNet pre sémantickú segregáciu. Napriek ich individuálnom výkonom však spracovanie jednotllivých úloh trvá dlhú dobu a nie je tak vhodné pre real-time detection. Navyše, embedded zariadenia týchto modelov majú veľmi nízky výpočetný výkon a v dôsledku toho je tento sekvenčný prístup ešte viac nepraktický. Jednotlivé úlohy, ktoré je nutné spracovať majú množstvo súvisiacich informácii. Napríklad jazdné pruhy sú častokrát používané k ohraničeniu oblasti, po ktorej je možné bezpečne jazdiť a väčšina objektov nutných pre detegovanie sa taktiež nachádza v tejto oblasti. YOLOP je schopné vykonávať všetky tieto úlohy paralelne, čím vzniká rýchle a efektívne riešenie. [24]

- *YOLOR:* Je moderný algoritmus pre object detection líšiaci sa od pôvodného YOLOv1 – YOLOv5 nielen autorom, no i architektúrou a modelovou infraštruktúrou. Špecializuje sa čisto na object detection, na rozdiel od iných use casov ako object identification alebo object analysis. To je dané jeho sústredením sa na všeobecné identifikátory, ktoré zaraďujú objekt do jednotlivých kategórii a tried. Ľudia sú schopní učiť sa a porozumieť fyzickému svetu na základe svojich zmyslov (explicitná znalosť), no i na základe predošlej skúsenosti (implicitná znalosť). Vďaka tomu sú schopní efektívne spracovávať úplne nové dáta využitím znalostí, ktoré sa naučili v minulosti uložených v mozgu. YOLOR sa týmto princípom inšpiruje a je navrhnutý ako „unifikovaná sieť, kde je implicitná a explicitná znalosť zakódovaná spolu. [25]

- *YOLOS:* You Only Look at One Sequence je postavený na úplne rozdielnej architektúre a priniesol so sebou vlastnosti ako *ViT transformers* a *DET tokens.* Zatiaľ sa jedná o koncept bez výkonnostných optimalizácii. Cieľom je dokázanie všestrannosti týchto transformerov pri zložitejších object detection úlohách. [26]

2.2.3 Single-shot detector

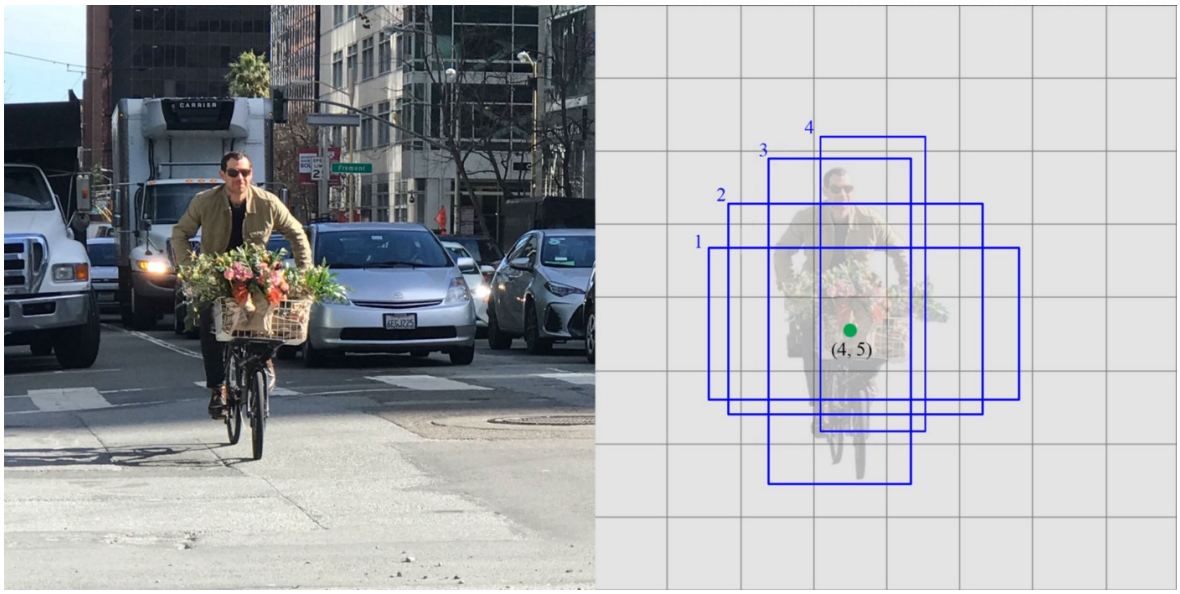
SSD bol navrhnutý pre real-time object detection. Faster R-CNN používa region proposal sieť pre tvorbu boundary boxov, ktoré následne používa pre klasifikáciu objektov. Tento prístup je síce presný, no celý proces prebieha v 7 snímkoch za sekundu, čo je hodnota

neskĺbiteľná s potrebami real-time detekcie. SSD tento proces zrýchľuje eliminovaním potreby region proposal siete. Na obnovenie poklesu presnosti používa SSD niekoľko vylepšení, vrátane multi-scale featues a default boxes. Tieto vylepšenia umožňujú SSD rýchlostne sa vyrovnať presnosti Faster R-CNN využitím obrázkov s nižším rozlíšením, ktoré taktiež vo významnej miere ovplyvňujú rýchlosť. [27]

SSD sa skladá z dvoch častí:

- Extrahovanie feature máp
- Aplikácia konvolučných filtrov pre detekciu objektov.

Pre extrahovanie feature máp využíva architektúru VGG16. Následne deteguje objektu pomocou Conv4_3 vrstvy. Každá predikcia sa skladá z boundary boxu a 21 hodnotení pre každú triedu. Spomedzi týchto hodnotení sa vyberie najvyššia hodnota pre daný objekt. Conv4_3 vykonáva celkovo 38 * 38 * 4 predikcií, tzn. 4 predikcie pre každú bunku. Mnoho predikcií neobsahuje žiadny objekt. Pre tieto predikcie si SSD rezervuje tzv. triedu 0. [27]



Obrázok 23 Extrakcia feature máp [27]

3 VÝVOJ MOBILNÝCH APLIKÁCII

Vývoj mobilných aplikácii sa dá definovať ako sada procesov a procedúr zahrnutých pri tvorbe software pre malé, bezdrôtové zariadenia, ako napr. smartfóny, a iné zariadenia. Podobne ako vývoj webových aplikácii má mobilný vývoj korene v tradičnom software. Kritickým rozdielom je však skutočnosť, že mobilné aplikácie sú tvorené špecificky s cieľom využitia unikátnych vlastností mobilného hardware. Napríklad hra môže využívať akcelerometer zabudovaný vo smartfóne, alebo zdravotná aplikácia môže využiť senzory zabudované v smart hodinkách. [28] Moderné smartfóny sú vybavené technológiami ako Bluetooth, NFC, gyroskopické senzory GPS a kamery. Taktiež môžu byť využité pre vývoj augmented reality aplikácii, skenovanie atď. [29]

Relevantné mobilné platformy v čase písanie diplomovej práce sú dve a to iOS od spoločnosti Apple a Android od spoločnosti Google. iOS vznikol ako proprietárny mobilný operačný system špecificky pre smartfóny Apple iPhone. Android má však širší záber, cieli na zariadenia rôznych výrobcov, ktorí si ho môžu v rámci jeho open source nátury ľubovoľne upravovať. [30]
Nepriek tomu, že máme k dispozícii tieto dve platformy, je nutné si zvoliť tú správnu na základe faktorov, ako je cena vývoja, cieľová skupina užívateľov, súčasný technologický trend, biznis model, čas nutný pre vývoj a ďalšie. Na základe štatistík má Android viac ako 70% podiel na globálnom trhu, zatiaľ čo iOS má približne 30%. Obchod Play Store pre platformu Android má taktiež menej obmedzení, než konkurenčný App Store od Apple. Na druhej strane mobilné aplikácie vyvinuté pre iOS majú oveľa menšiu škálu zariadení, ktoré musia podporovať, čo zjednodušuje optimalizáciu. [30]

Tvorba aplikácie má svoje výhody aj nevýhody. Výberom prístupu, ktorý zodpovedá zvolenej stratégii je možné dosiahnuť lepší užívateľský zážitok, ušetriť výpočtové zdroje a taktiež vytvárať natívne funkcie potrebné pre výslednú aplikáciu. Medzi dnes rozšírené prístupy patria [31]:

- Natívne aplikácie
- Multiplatformné a hybridné aplikácie
- Progresívne webové aplikácie

3.1 Natívny vývoj

Natívne aplikácie sú vyvíjané exkluzívne pre jednu platformu za pomocou špecifického programovacieho jazyka. Užívatelia k nim majú prístup z dedikovaných aplikačných obchodov (Google Play pre Android a App Store pre iOS). [32]
Apple i Google poskytujú vývojárom natívne Software Development Kits (SDK) a nástroje. To so sebou prináša značné výhody, najmä:

- *Výkon:* Aplikácie kompilované pomocou natívnych nástrojov sú rýchle a responzívne. Aplikácia je uložená priamo v telefóne, takže je schopná efektívne využívať výpočetné zdroje daného zariadenia. Výsledkom je jej rýchle načítavanie. [33]
- *Bezpečnosť:* Natívne aplikácie sa pre svoju funkcionalitu nespoliehajú na technológie, ktoré nie sú pre danú platformu natíve, ako HTML, CSS a JavaScript a teda je schopná vyhnúť sa známym bezpečnostným rizikám, ktoré so sebou webové technológie v prípade neopatrnosti vývojára prinášajú. [34]
- *Široká škála funkcionalít:* Vývojárom je umožnený prístup ku každému API a nástroju, ktoré daná platforma ponúka, ako sú napr. GPS, kamera, mikrofón a v neposlednom rade natívna podpora push notifikácii. Majú teda priamy prístup k hardware. [32]
- *User experience:* Každá platforma so sebou prináša určité konvencie pre vzhľad. Natívne aplikácie používajú tieto konvencie, či už sú to farby, rozmiestnenie komponent na obrazovke, alebo samotný vzhľad týchto komponent. Vývojár sa tak nemusí starať o dodatočné štýlovanie a odpadá mu tak práca navyše. Zároveň dodá užívateľom konzistentný vzhľad a spôsob používania aplikácie, na ktorý sú zvyknutí. [33]
- *Potenciál menšieho množstva bugov počas vývoja:* Z podstaty veci obsahujú výrazne menej závislostí, pretože nemajú žiadneho prostredníka umožňujúceho spustenie aplikácie na tej, či onej platforme, ako napr. Flutter alebo React Native. Hybridné aplikácie pristupujú k hardware cez tzv. *„bridge“,* ktorý častokrát spomaľuje vývoj. Tento problém je častý najmä pri nových verziách operačných systémov, kde musí byť tento bridge aktualizovaný pre udržanie kompatibility, v prípade novej funkcionality, alebo zmeny v samotných operačných systémoch. Tento proces je

taktiež časovo náročný a nové features nie sú dostupné pre cross-platform vývojárov ihneď. [35]

Natívny vývoj však napriek zjavným výhodám so sebou nesie i významné úskalia a preto nie je vždy jasnou voľbou pre firmy, ktoré chcú vyvinúť svoj mobilný produkt. Preto je nutné vziať do úvahy aspekty ako:

- *Takmer nulová znovupoužitelnosť kódu:* Natívne aplikácie sú tvorené separátne a teda vzniká nutnosť udržiavania dvoch code bases v prípade, že chceme aplikáciu dodať pre Android i iOS. [35]
- *Cena:* Výrazne vyššia oproti iným riešeniam. Táto skutočnosť sa odvíja od nutnej znalosti platformy, pre ktorú je aplikácia vyvíjaná. Nároky na vývojárov sú tým pádom vyššie. [36]
- *Doba trvania vývoja:* Väčšina natívnych vývojárov sa špecializuje iba na jednu platformu. Častokrát vzniká teda nutnosť najať ďalších vývojárov. [36]

3.2 Multiplatformný vývoj

Základnou myšlienkou multiplatformného vývoja je, že výsledný softvérový produkt by mal fungovať vo viac ako jednom digitálnom prostredí. Cieľom je zvyčajne vydať softvér pre niekoľko operačných systémov zároveň. V prípade mobilného vývoja sú to Android a iOS. Medzi základné stratégie pre multiplatformný vývoj patrí kompilácia rôznych verzii toho istého programu pre rôzne operačné systémy. Ďalším možným prístupom je tvorba abstraktnej vrstvy pre určité úrovne programu, ktorá je schopná sa týmto platformám prispôsobiť. O takto napísanom softvéri môžme tvrdiť, že je platformne agnostický a teda neuprednostňuje jednu platformu pred druhou. V prípade potreby sú však vývojárom k dispozícii natívne API pre kontrolu špecifických častí jednotlivých operačných systémov a prípadne hardvéru [37].

V konečnom dôsledku môžeme tvrdiť, že hlavným cieľom multiplatformných aplikácii je pokryť čo najväčší počet zariadení a dostať sa tým k maximálnemu možnému počtu užívateľov. Tento spôsob vývoja však so sebou prináša i ďalšie výrazné výhody, medzi ktoré patrí napríklad:

- *Cena vývoja:* Keďže sa držíme konceptu *"napíš raz, spusti všade",* disponujeme znovupoužiteľným kódom pre obe platformy v rámci jedného repozitára, čo výrazne prispieva k zníženiu častu potrebného pre vývoj aplikácie a teda v konečnom dôsledku aj cenu. [38]
- *Jednoduchšia údržba:* Ďalší vývoj a oprava chýb sa stáva jednoduchšou vďaka jednej codebase z ktorej je možné nasadzovať kód pre obe platformy. [38]
- *Rýchlejší vývoj:* Jednotná code base pre obe platformy pomáha znižovať nároky na vývoj o 50 – 80%. [38]
- *Jednotný dizajn:* Synchronizácia rôznych vývojových projektov je náročná úloha ako po funkcionálnej, tak po vizuálnej stránke. Multiplatformný vývoj umožňuje zachovať jednodtný dizajn naprieč platformami, umocniť tak brand recognition a dodať konzistentný užívateľský zážitok naprieč platformami. [38]

Zo zoznamu vyššie jednoznačne vyplýva, že čas dodania aplikácie na trh je výrazne nižší, než pri natívnom vývoji. Napriek tomu, že táto metrika je veľmi významná, nemala by sa brať vždy v úvahu ako jediná. Multiplatformný prístup so sebou taktiež nesie i úskalia [39]:

- *Nižší výkon:* Multiplatformné aplikácie čelia integračným výzvam operačných systémov, na ktoré cielia. Z veľkej miery je to kvôli nekonzistentnej komunikácie medzi natívnymi a nenatívnymi prvkami, čo má vplyv na konečný výkon aplikácie. [39]
- *Limitovaný zážitok pre užívateľa:* Kód nie je vždy dodatočne optimalizovaný pre najnovšie vlastnosti operačných systémov a teda prístup k novším častiam hardvéru alebo API môže byť obmedzený, nesúc so sebou množstvo integračných problémov. [40]
- *Limitované množstvo nástrojov:* Najnovšie API a možnosti platformy nie sú vždy dostupné hneď po vydaní a vývojári tak msuia čakať na novú verziu multiplatformného frameworku, ktorý častokrát zaostáva za natívnym vývojom. Vývojári tak musia určitý čas pracovať na starších verziách daných platforiem.
- *Bezpečnosť:* Útoky na mobilné aplikácie nie sú zriedkavým javom. Natívne aplikácie sa im dokážu vyhnúť zaplátaním chýb agilnejším spôsobom. [41]

3.3 Progresívne webové aplikácie

Úspech smartfónov a mobilných aplikácii umožnilo okamžitý prístup k ľubovoľnému systému, či už sa jedná o online nakupovanie, platbu účtov, či bookovanie hotelov. Fakt, že smartfóny sú mobilné zariadenia, umožňujú jednoduchý prístup k internetu a nevyžadujú komplexný hardvér spôsobil postupné vytlačovanie tradičných desktop zariadení. Mobilné aplikácie ponúkajú svojim užívateľom jednoduchý prístup k týmto službám a funkcionalitou možnosťami prekonávajú webové aplikácie.
Problémom však je počet stiahnutí. Náklady na vývoj sú taktiež značné a častokrát je ich obhajoba oproti webovým aplikáciám náročná. Mnoho menších firiem si nemôže dovoliť financovanie vývoja natívnej mobilnej aplikácie z viacerých dôvodov, medzi ktoré patrí obmedzený rozpočet, čas potrebný pre vývoj, zdroje atď. Taktiež mnoho užívateľov preferuje webové aplikácie pred inštaláciou dedikovanej mobilnej aplikácie.
Odpoveď na tieto problémy sa snažia priviesť práve Progresívne webové aplikácie – PWA. [42]

PWA sú webové aplikácie, ktoré vyzerajú a fungujú ako mobilné aplikácie. Pristupuje sa k nim pomocou URL, rovnako ako pri konvenčných webových stránkach a aplikáciách. Tento relatívne nový prístup k vývoju aplikácii sa snaží priniesť výhody moderných prehliadačov v kombinácii s mobilným zážitkom. [42]

Medzi tri základné piliere PWA patrí: schopnosť, spoľahlivosť a inštalovateľnosť, vďaka ktorým dokážu poskytnúť užívateľský zážitok podobný tomu natívnemu. [43]

3.3.1 Schopnosť

Trendom vo webových technológiách je ich rapídny vývoj. Napríklad donedávna nebolo možné vytvoriť komplexnú aplikáciu pre video chat pomocou WebRTC, geolokácie, push notifikácii, kamery a Bluetooth. Tieto vlastnosti boli vždy doménou natívnych aplikácii. Aj keď je pravdou, že mnoho natívnych možností je stále mimo dosah, nové webové štandardy sa to snažia zmeniť sprístupnením nových API pre vývojárov. Medzi najnovšie features patrí:

- Ovládanie clipboardu
- Prístup k súborom systémy
- Media controls [43]

Medzi modernými API, WebAssembly a ďalšími pripravovanými features sú dnešné webové aplikácie schopnejšie ako kedykoľvek predtým.

3.3.2 Spoľahlivosť

Spoľahlivá Progresívna webová aplikácia vzbudzuje dojem rýchlosti a je nezávislá od stavu siete. Rýchlosť je kritickým faktorom pre udržanie stabilného počtu užívateľov. V tejto kategórii nie je zahrnutá iba rýchlosť načítania, ale i plynulosť animácii a odozva na interakciu užívateľa s aplikáciou.
Spoľahlivé aplikácie musia správne fungovať nezávisle na stave pripojenia k sieti. Užívatelia očakávajú aspoň nejakú funkcionalitu po spustení aplikácie, či už sa jedná o offline prehrávanie médii, alebo prístup ku vstupenkám.

3.3.3 Inštalovateľnosť

PWA je možné si nainštalovať do svojho zariadenia. Napriek tomu, že sú to webové aplikácie, užívateľ nevidí po nainštalovaní (pridanú na plochu) žiadne prvky prehliadača, keďže aplikácia beží v samostatnom okne. Po spustení sa aplikácia tvári ako natívna, má svoje miesto na home screen medzi ostatnými aplikáciami s vlastným názvom aplikácie a ikonou. [43]

3.3.4 Adopcia PWA na trhu

Vo svojom jadre sú PWA iba webové aplikácie, ktoré použitím techník ako progressive enhancement, kedy sú jednotlivé features aplikácie sprístupnené na základe siete a zairadenia užívateľa, nových možností moderných prehliadačov a rapídnemu rozvoju webových technológii sú schopné konkurovať do určitej miery natívnym aplikáciam. Je možné si ich nainsťalovať na plochu, majú offline funkcie, podporu push notifikácii a prístup k niektorým natívnym API. Stále sú však do značnej miery obmedzené svojím webovým jadrom.
Napriek týmto skutočnostiam sú PWA populárnym mobilným riešením pre jednoduché aplikácie a služby ako e-shopy, prezentačné weby, aplikácie slúžiace k predaniu rôznych informácii a ich adopcia na trhu rôznymi firmami narastá každým rokom. Jednou z týchto firiem je aj Twitter, ktorý vďaka implementácii PWA technológii na sovjom webe získal o 65% vyšší user engagement pri jednej návšteve, o 75% viac príspevkov, pričom zredukovali veľkosť svojej aplikácie o 75%.

PWA tak ponúkajú unikátnu príležitosť dodať plnohodnotný webový zážitok na mobilnej platforme. Tieto aplikácie môžu byť nainštalované *hocikým, hocikde* a na *hocijakom* zariadení a to všetko v rámci jednej codebase. [43]

3.4 Voľba vývojového prístupu mobilnej aplikácie

V snahe dodať užívateľovi čo najkvalitnejšiu aplikáciu sa automaticky ponúka zvoliť prístup natívneho vývoja. V skutočnosti je však pravdou, že neexistuje jeden správny alebo univerzálny spôsob, ktorý by bol najlepší pre každý typ aplikácie. Výber prístupu v najväčšej miere ovplyvňujú samotné požiadavky na aplikáciu, časové okno, v ktorom chceme, aby bola aplikácia vyvinutá a pripravená pre release a v neposlednom rade cena, ktorú je ochotný zákazník alebo firma za aplikáciu zaplatiť.

Presná definícia týchto požiadaviek je častokrát náročná, keďže musia zahŕňať i schopnosť nasadzovať zmeny, ktoré korešpondujú s biznis modelom firmy v dostatočnej rýchlosti. Ďalej je nutné pozastaviť sa nad tým, ako si získať a najmä udržať čo najväčší rozsah užívateľov.

Berúc do úvahy všetky tieto faktory, výhody a nevýhody rôznych prístupov a potreby samotnej aplikácie by malo napomôcť pri odhadovaní časového okna projektu, potrebných finančných a technických zdrojov. Výsledné aplikácie sú potom veľmi rôznorodé. Ak je konečným cieľom produktu vyvinúť dlhodobý projekt a teda aplikáciu, ktorá bude zároveň responzívna, mať kvalitný užívateľský zážitok, výborný výkon a spoľahlivú mieru bezpečnosti, potom sa ponúka natívny vývoj ako žiadúca voľba. Naopak, ak je nižší rozpočet, rýchly vývoj a vyšší potenciálny počet užívateľov primárnym zameraním biznisu, je vhodné zvoliť jeden z multiplatformných alebo webových prístupov k tvorbe mobilnej aplikácie. [44]

Tabuľky nižšie obsahujú súhrn všeobecných nefunkcionálnych požiadaviek z hľadiska biznisu, ktoré môžu byť kritické pri rozhodovaní o výbere prístupu, no i konkrétnych API a natívnych funkcionalít, obohacujúcich výslednú funkcionalitu a dojem aplikácie. [45]

Tabuľka 1 Nefunkcionálne požiadavky pre vývoj

	Natívny prístup	**Cross-platform prístup**	**PWA prístup**
Nutné znalosti a nástroje	iOS – Swift a Xcode, SwiftUI ,UIKit Android – Java alebo Kotlin, Android Studio, Android SDK	HTML, CSS, JavaScript, Multiplatformný framework (React Native alebo Flutter)	HTML, CSS, JavaScript
Distribúcia	Aplikačný obchod	Aplikačný obchod	Web
Rýchlosť vývoja	Pomalá	Stredná	Rýchla
Cena vývoja	Vysoká	Stredná	Nízka
Cena za údržbu	Vysoká	Stredná	Nízka
Grafická výkonnosť	Vysoká	Stredná	Dostačujúca
Výkonnosť aplikácie	Vysoká	Stredná	Dostačujúca

Tabuľka 2 Prístup k natívnym APIs pre jednotlivé prístupy

	Natívny prístup	**Cross-platform prístup**	**PWA prístup**
Kamera	Áno	Áno	Áno
Push notifikácie	Áno	Áno	Obmedzene
Kontakty	Áno	Áno	Nie
Offline prístup	Áno	Áno	Obmedzene
Geolokácia	Áno	Áno	Obmedzene
Upload súborov	Áno	Áno	Áno
Gyroskop	Áno	Áno	Obmedzene
Akcelerometer	Áno	Áno	Obmedzene
Swipe navigácia	Áno	Áno	Áno
Mikrofón	Áno	Áno	Áno
Spracovanie Machine learning modelov	Áno	Obmedzene	Nie
Práca s rozšírenou realitou	Áno	Obmedzene	Nie
Bluetooth	Áno	Áno	Obmedzene
Haptická odozva	Áno	Áno	Áno

4 VÝVOJ PRE PLATFORMU IOS

iOS je dedikovaný mobilný operačný systém spoločnosti Apple poháňajúci proprietárny hardware a teda bežiaci na zariadeniach iPhone a v minulosti i na produktovej rade iPad, kde bol nahradený iPadOS-om v roku 2019. Z hľadiska popularity sa jedná o druhý najrozšírenejší mobilný OS, zaostávajúc v adopcii za OS Android. Tvorí základ pre tri ďalšie operačné systémy, iPadOS, tvOS pre produkty Apple TV a watchOS pre chytré hodinky Apple Watch. Napriek tomu, že to je proprietárny sofware, niektoré časti sú open source pod licenciou *Apple Public Source License*. [46]

4.1 Natívny vývoj v rámci iOS

Pre tvorbu aplikácii na iOS je nutné pracovať na MacOS, iné platformy nie sú podporované. Vývojové nástroje sú však zdarma a aplikáciu je možné spustiť a testovať v zabudovanom simulátori [47]. Primárnym vývojovým prostredím – IDE je XCode, ktorý je možné použiť nielen pre vývoj aplikácii pre iPhone, ale i ostatný hardware v rámci produktovej škály Apple. Obsahuje v sebe iOS SDK, nástroje, kompilátor a frameworky umožňujúce dizajnovanie, vývoj a odladťovanie aplikácii [48].

4.1.1 Programovací jazyk Swift

Swift je viacparadigmový, univerzálny programovací jazyk pre tvorbu iPadOS, macOS, watchOS a iOS aplikácii. Bol vytvorený spoločnosťou Apple v roku 2014. Jazyk bol navrhnutý tak, aby bezpečný, rýchly a expresívny. Vznikol tak ako náhrada za zastaralý Objective-C. Keďže je jazyk open-source, jeho kód je verejne dostupný.
Apple tento jazyk vytvoril berúc ohľad na nových programátorov a je tak jednoduchý na naučenie sa. Je to vysokoúrovňový jazyk a teda disponuje vlastnosťami ako:

- *Štruktúry a triedy***:** Vývojár ich môže definovať v jedinom súbore. Štruktúra alebo trieda je potom dostupná naprieč celým projektom bez nutnosti importovania súboru, v ktorom bola definovaná.
- *Natívny error handling:* Swift ponúka klasické ovládanie error stavov pomocou try catch blokov a propagáciu chýb počas behu programu.
- *Generika:* Umožňuje písanie flexibilných, znovoupoužteľných funkcii a typov.
- *Pamäťová bezpečnosť:* Swift vykonáva správu pamäte automaticky. Pomocou *Automatic Reference Counting* taktiež sleduje a manažuje využívanie pamäte.

- *Package manager:* Swift Package Manager je nástroj umožňujúci buildovať, spustiť a testovať swiftové knižnice. [48]

Swift splňuje ciele, ktoré si autori jazyka nastavili výborne, vďaka jeho jasnej syntaxi, dobrej čitatelnosti a dynamicite. No stále sa jedná o relatívne nový jazyk a preto zdroje k učeniu nemusia byť tak robustné, ako je to v prípade iných jazykov. Taktiež je tu silne cítiť vplyv platformy, pre ktorú bol jazyk primárne navrhnutý. Napriek tomu, že jazyk je cross-platform a teda kompatibilný i s operačnými systémami Windows a Linux, najlepší je stále pre vývoj natívnych iOS aplikácii. Fakt, že sa jedná o mladý jazyk odzrkadľuje aj výrazné množstvo aktualizácii, ktoré jazyk dostáva a mnohé vlastnosti, ktoré by programátori očakávali ako prirodzenou súčasťou jazyka prichádzajú neskôr práve až v rámi týchto aktualizácii. [49]

4.1.2 Framework SwiftUI

SwiftUI je framework pre tvorbu UI s podporou iOS, tvOS, macOS, watchOS a iPadOS. Apple ho uvidedol v roku 2019 a odvtedy sa vyvíjal rapídnym tempom. Na rozdiel od predošlého imperatívneho frameworku UIKit je SwiftUI deklaratívny a umožňuje tak tvorbu UI s čo najmenším objemom kódu, ktorý je prenositeľný naprieč rôznymi OS. [50]
Deklaratívne programovanie popisuje čo program vykonáva bez toho, aby bol špecifikovaný spôsobob, akým toho má dosiahnuť. Naopak imperatívne programovanie popisuje práve ako by mal program vykonať nejakú úlohu tým, že programátor explicitne špecifikuje každú inštrukciu krok za krokom [51]. Pri tvorbe UI je výhodnejší deklaratívny spôsob najmä vďaka rýchlosti vývoja a predchádzaniu chýb. Zmeny v stave aplikácie majú vplyv na zmenu UI. Programátor popisuje, ako má UI vyzerať pre daný stav a framework sa postará o samotné zobrazenie. [52]

```swift
import SwiftUI

struct Content : View {

    @State var model = Themes.listModel

    var body: some View {
        List(model.items, action: model.selectItem) { item in
            Image(item.image)
            VStack(alignment: .leading) {
                Text(item.title)
                Text(item.subtitle)
                    .color(.gray)
            }
        }
    }
}
```

Obrázok 24 Ukážka SwiftUI komponenty [51]

```swift
cluesLabel = UILabel()
cluesLabel.translatesAutoresizingMaskIntoConstraints = false
cluesLabel.font = UIFont.systemFont(ofSize: 24)
cluesLabel.text = "CLUES"
cluesLabel.numberOfLines = 0
view.addSubview(cluesLabel)

answersLabel = UILabel()
answersLabel.translatesAutoresizingMaskIntoConstraints = false
answersLabel.font = UIFont.systemFont(ofSize: 24)
answersLabel.text = "ANSWERS"
answersLabel.numberOfLines = 0
answersLabel.textAlignment = .right
view.addSubview(answersLabel)
```

Obrázok 25 Ukážka UIKit komponenty [51]

Napriek tomu, že je SwiftUI veľmi odlišné od UIKitu, stále sa naň pod kapotou spolieha pre výslednú tvorbu UI. [51]

Jedným zo všeobecne najobtiažnejších aspektov mobilného vývoja je synchronizácia aplikačného stavu a UI. Vždy, keď sa zmení stav aplikácie, UI musí túto zmenu reflektovať. Pri UIKite sa jedná zväčša o manuálnu úlohu. Pre SwiftUI je to však opačný prípad. View už nie je výsledkom sekvencie eventov, ale funkcia stavu samotného. Inými slovami, UI je derivátom stavu aplikácie. Ak sa zmení stav, automaticky sa mení aj UI, čím sa eliminuje celá škála potenciálnych bugov [52].

Správa stavu vo SwiftUI nevyužíva delegáty, dátové zdroje ani iné podobné vzory, ktoré sú bežne používané v UIKite. Využíva zabudované *property wrappers,* umožňujúce deklarovať presný spôsob, akým sú dáta pozorované, zobrazované a menené pomocou jednotlivých komponent nazývaných *views* [53]. Medzi tieto properties patria:

- *@State:* Keďže je SwiftUI primárne UI framework, jeho deklaratívny dizajn nemusí nutne ovplyvňovať celú dátovú vrstvu aplikácie, ale iba stav, ktorý je priamo spojený s rôznymi views. Napríklad, ak by sme chceli implementovať view pre registráciu užívateľa zadaním *emailu* a *username.* Nadefinovať ich môžme pomocou kľúčového slova @State a tým automatický vytvorí prepojenie medzi týmito dvoma hodnotami a samotným view. To znamená, že vždy, keď sa tieto hodnoty zmenia, prekreslia sa aj komponenty, ktoré tento stav využívajú. Prepojiť tieto dáta s UI môžme pomocou tzv. *bindovania*, kde pred názov stavovej premennej pridáme „*$”*. V prípade, že binding predáme do editovateľného poľa, napríklad inputu, môžme hodnotu tohto poľa priamo napojiť na stav. *@State* sa teda používa ako interná reprezentácia view a je odporúčané ich označiť pri deklarácii kľúčovým slovom *private,* aby tieto hodnoty neboli prístupné a meniteľné mimo ich view, čím by potenciálne spôsobili pád celej aplikácie. [53]

```
struct SignupView: View {
    var handler: (User) -> Void

    @State private var username = ""
    @State private var email = ""

    var body: some View {
        VStack {
            TextField("Username", text: $username)
            TextField("Email", text: $email)
            Button(
                action: {
                    self.handler(User(
                        username: self.username,
                        email: self.email
                    ))
                },
                label: { Text("Sign up") }
            )
        }
        .padding()
    }
}
```

Obrázok 26 Použitie @State [53]

- *@ObservedObject:* Na rozdiel od @State, ktorý je používaný pre uchovávanie interného stavu aplikácie, @ObservedObject je spôsob, akým je možné pripojiť sa k externým objektom modelu pre rôzne views. Používa sa v kombinácii s *ObservableObject*. Ak vezmeme v úvahu predchádzajúci príklad a presunieme zodpovednosť správy užívateľských dát z view do samostatného modelu a teda dedikovaného objektu, sme schopní vytvoriť *controller,* ktorý dedí od *ObservableObject* protokolu. [53]

```
class UserModelController: ObservableObject {
    @Published var user: User
    ...
}
```

Obrázok 27 Model použitý pre @ObservedObject [53]

Field v takomto controlleri je nutné označiť dekorátorom *@Published*, ktorým signalizujeme odoslanie eventu v prípade, že sa hodnota vo fielde zmenila.
Takto vytvorený controller môžme implementovať v ľubovoľnom view, kde je nutné vytvoriť *ObservedObject* field (rozdiel oproti @State).

```swift
struct ProfileView: View {
    @ObservedObject var userController: UserModelController
    @State private var isEditingViewShown = false

    var body: some View {
        VStack(alignment: .leading, spacing: 10) {
            Text("Username: ")
                .foregroundColor(.secondary)
                + Text(userController.user.username)
            Text("Email: ")
                .foregroundColor(.secondary)
                + Text(userController.user.email)
            Button(
                action: { self.isEditingViewShown = true },
                label: { Text("Edit") }
            )
        }
        .padding()
        .sheet(isPresented: $isEditingViewShown) {
            VStack {
                ProfileEditingView(user: self.$userController.user)
                Button(
                    action: { self.isEditingViewShown = false },
                    label: { Text("Done") }
                )
            }
        }
    }
}
```

Obrázok 28 Použitie @ObservedObject vo SwiftUI komponente [53]

Podstatným rozdielom je však spôsob inicializácie dát. Takto prepojený model je nutné najprv inicializovať.

```swift
struct ProfileView: View {
    @ObservedObject var userController = UserModelController.load()
    ...
}
```

Obrázok 29 Inicializácia modelu [53]

- *@EnvironmentObject:* Existuje ešte druhý spôsob, ako predávať globálne dáta a tým je *@EnvironmentObject,* ktorý na rozdiel od @ObservedObject dokáže predávať data medzi views, ktoré nie sú hierarchicky prepojené. Napriek tomu, že je jednoduché vytvoriť binding medzi parent a children views, tento spôsob nie je praktický v momente, kedy je nutné tú istú hodnotu predávať v rozsiahlejšej

hierarchii, napríklad v prípade témy pre štýlovanie. *@EnvironmentObject* sa aplikuje pomocou jednoduchého modifikátora na inštancii view. [53]

```
struct RootView: View {
    @ObservedObject var theme: Theme
    @ObservedObject var articleLibrary: ArticleLibrary

    var body: some View {
        ArticleListView(articles: articleLibrary.articles)
            .environmentObject(theme)
    }
}

struct ArticleView: View {
    @EnvironmentObject var theme: Theme
    var article: Article

    var body: some View {
        VStack(alignment: .leading) {
            Text(article.title)
                .foregroundColor(theme.titleTextColor)
            Text(article.body)
                .foregroundColor(theme.bodyTextColor)
        }
    }
}
```

Obrázok 30 Použitie @EnvironmentObject [53]

SwiftUI je však stále relatívne novým frameworkom a neobsahuje v sebe všetky prvky, ktoré sú štandardom v UIKite. Minimálna podporovaná verzia je iOS 13 a XCode 11. Z tohto dôvodu je stále užitočné mať aspoň základnú znalosť UIKitu [54]. SwiftUI však ponúka možnosť preniesť views napísané v UIKite do SwiftUI pomocou protokolu *UIViewRepresentable* [55].

```
struct ActivityIndicator: UIViewRepresentable {

    func makeUIView(context: Context) -> UIActivityIndicatorView {
        let v = UIActivityIndicatorView()

        return v
    }

    func updateUIView(_ activityIndicator: UIActivityIndicatorView, context:
        activityIndicator.startAnimating()
    }
}
```

Obrázok 31 Prevod UIKit komponenty do SwiftUI [55]

Týmto spôsobom obalené view je potom možné použiť ako súčasť ľubovoľného SwiftUI view. [55]

```
struct ContentView : View {

    var body: some View {
        ActivityIndicator()
    }
}
```

Obrázok 32 Použitie konvertovaného UIKit view vo SwiftUI [56]

4.2 Multiplatformový vývoj pre iOS

Na trhu dnes existuje niekoľko hojne využívaných technológii, ktoré dokážu pomocou webových technológii a bridgov s natívnymi platformami dodať rýchlejšie vyvinutú aplikáciu, ktorú je možné nasadiť z jednej codebase. Medzi dnes najpoužívanejšie patrí *Flutter* a *React Native* [56].

4.2.1 Flutter

Flutter je najmladším frameworkom, ktorý spôsobil rozruch vo svete mobilného vývoja. Vytvorený spoločnosťou Google v roku 2017, umožňuje vývojárom tvorbu mobilných aplikácii pre iOS i Android. Pozostáva z SDK a Widgetovej UI knižnice. Tá obsahuje rôzne znovu použiteľné prvky UI ako tlačidlá, slidre a textové polia. Flutter využíva programovací

jazyk Dart, ktorý je syntakticky podobný JavaScriptu. Je to silne typovaný objektovo orientovaný programovací jazyk sústrediaci sa na front-end vývoj.
Napriek tomu, že je Flutter relatívne mladý, mnohé spoločnosti si ho vyberajú oproti konkurentom ako Xamarin, Cordova a dokonca React Native. Medzi tieto spoločnosti patrí eBay, Philips Hue a Xianuy od spoločnosti Alibaba. [57]
Z architektonického hľadiska obsahuje Flutter tri vrstvy, každá z nich pozostávajúca zo sady knižníc.
Prvou vrstvou je **Framework** vrstva, využívaná pre interakciu vývojárov s Flutterom, ktorá je napísaná v jazyku Dart. Obsahuje widget, animácie, rendrovaciu logiku atď. Medzi jeho hlavné komponenty patrí:

- *Foundation:* Je akousi základnou triedou poskytujúcu podporu pre vykreslovanie a animácie.
- *Widgets:* Samotné komponenty
- *Material & Cupertino:* tieto widget sú používané k tvorbe layoutov a podpory dizajnu pre iOS.

Druhou vrstvou je **Engine**. Napísaný primárne v C/C++, poskytuje nízko úrovňové implementácie Flutter jadra a kompilačných nástrojov. Je zodpovedný za kompiláciu Dart kódu na danom zariadení.

Treťou a poslednou vrstvou je **Embedder,** slúžiaci ako akýsi vstupný bod pre aplikáciu. Prijíma vlákno pre UI Flutteru a taktiež inicializuje engine. Použitím embedderu je možné integrovať ód do existujúcej aplikácie ako modul. Napríklad Java a C++ embedder pre Android, Objective-C embedder pre iOS a macOS.
Celkový build pozostáva z troch krokov:

1. Dartový kód je najprv kompilovaný do ARM knižnicového kódu
2. Kompilovaný kód je následne vložené do APK súboru pre Android alebo IPA súboru pre iOS
3. Následne sa spustí kód embedderu a aplikácia sa spúšťa. [58]

Widgety – znovu použiteľné prvky UI vznikajú pomocou moderného frameworku inšpirovaného Reactom. Základnou myšlienkou je, že pomocou nich tvoríme kompletné UI. Popisujú, ako by malo dané view vyzerať na základe určitej konfigurácie alebo stavu. Ak sa zmení stav widgetu, zmení sa aj jeho UI.

```dart
import 'package:flutter/material.dart';

void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
```

Obrázok 33 Jednoduchý Flutter widget [60]

V príklade vyššie pozostáva UI z funkcie *runApp()*. Obsahuje v sebe dva widgety, *Center* a *Text*. Framework donúti root widget roztiahnuť sa po celej obrazovke a výsledkom je teda vycentrovaný textový element s nápisom "Hello World". Pri písaní aplikácii je bežné, že si programátor vytvorí nové widgety, ktorú sú podtriedy buď *StatelessWidget* alebo *StatefulWidget* v závislosti od toho, či aplikácia obsahuje nejaký stav, alebo nie [59].

Flutter má značné výhody na poli performance a tvorby UI pomocou widgetov, nevýhodou je veľkosť výslednej aplikácie, ktorá oproti natívnemu alebo dokonca oproti konkurentom na poli multiplatformového vývoja je vyššia. Keďže sa jedná o nový framework, podpora komunity nedosahuje tak vysokej úrovne ako napr. pri React Native a (zatiaľ) chýba množstvo knižníc tretích strán a vývojár je teda častokrát nútený písať určité komponenty sám. Mnoho vývojárov odrádza aj skutočnosť, že Flutter sa píše v Darte, ktorý sa líši od tradičného jazyka multiplatformových frameworkov – JavaScriptu.

Napriek týmto skutočnostiam je však Flutter výbornou voľbou a jeho popularita každým rokom výrazne narastá. [60]

4.2.2 React Native

Mantrou frameworku je „Learn once, write anywhere“. Vychádza z webového frameworku React a teda je možné zjednodušene tvrdiť, ak vie vývojár písať React pre web, vie písať aj React Native pre mobilné zariadenia. Prvá verzia vyšla v januári 2015 pod taktovkou firmy Meta, momentálne sa jedná o open srouce projekt s obrovskou podporou komunity.

Keďže je framework založený na webových technológiách, môže sa zdať, že sa jedná iba o aplikáciu zabalenú do *WebView,* nie je tomu však tak. React Native využíva plne natívne komponenty či už Androidu alebo iOS a na pozadí spúšťa separátne JavaScript vlákno, ktoré sa stará o logiku a interaktivitu aplikácie. Základom je *React Native Bridge,* napísaný v jazykoch Java a C++ umožňujúci komunikáciu medzi hlavným a JavaScriptovým vláknom. Ak teda napríklad užívateľ stlačí natívne vykreslené tlačidlo, hlavné vlákno tento event zaregistruje a pomocou bridgu odošle správu JavaScriptovému vláknu správu na základe ktorej spravuje logiku aplikácie. Celková výkonnosť aplikácie je dobrá, no nedosahuje výsledkov natívnych aplikácii, pretože bridge nie je najrýchlejším riešením. React Native je však stále skvelou a primárnou voľbou nohých firiem a komerčne využívaných aplikácii. [61]

Jedná sa o deklaratívny framework a teda UI je tvorené pomocou znovu použiteľných komponent, ktoré je možné skladať do väčších celkov v rámci aplikácie. Jednotlivé kategórie sú:

- *Základne komponenty:* Zahŕňajú komponenty pre text, textové polia, view a obrázky
- *UI:* Komponenty, ktoré je možné vyrendrovať na hocijakej plarforme, ako napr. *Button* a *Switch.*
- *List Views:* Na rozdiel od scroll view, listy rendrujú iba tie elementy, ktoré sú zobrazené na obrazovke, vhodné pre dlhé zoznamy dát.
- *Android-specific*
- *iOS-specific*
- *Others:* Komponenty, ktoré nemusia byť nutne použité v každej aplikácii. Jedná sa o indikátory aktivity, Linkovanie pre externú navigáciu, manipuláciu so status barom a modály. [62]

```
import React from 'react';
import { Text, View } from 'react-native';

const HelloWorldApp = () => {
  return (
    <View
      style={{
        flex: 1,
        justifyContent: "center",
        alignItems: "center"
      }}>
      <Text>Hello, world!</Text>
    </View>
  )
}
export default HelloWorldApp;
```

Obrázok 34 Jednoduchá React komponenta [63]

Logická časť komponenty je písaná pomocou JavaScriptu, no pre definíciu UI sa používa syntax podobný HTML zvaný JSX. Jedná sa o rozšírenie JavaScriptu, ktoré produkuje React elementy. Jeho úlohou je oddeliť logiku a UI pomocou syntaxu známeho pre webových vývojárov. Každý JSX element je v skutočnosti JavaScriptová funkcia, ktorá v konečnom dôsledku vykresľuje UI. [63]

Interný stav React Native komponent je možné využiť vďaka tzv. *React hooks,* konkrétne hooku *useState.* Hooky sú špeciálne funkcie využívané pre zmenu chovania React komponent. *useState* vytvára stavovú premennú, ktorú je možné využívať ako v rámci funkcii a logiky komponenty, tak aj v rámci JSX. Hook pozostáva z troch častí:

- Premenná udržiavajúca interný stav
- Funkcia pre zmenu stavu
- Počiatočný stav [64]

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```


```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```


Obrázok 35 Lokálny stav React Native aplikácie [65]

React Native sa približuje k natívnym aplikáciám viac, než mnohé ostatné konkurenčné technológie. Využívanie natívne rendrovaných komponent je inovatívny koncept s vysokým potenciálom pre ďalší rozvoj. Existujú však kompromisy spojené s jeho fungovaním a vo výsledku React Native aplikácie ešte stále zoastávajú za natívnymi, je to však lepší prístup, ako hybridné aplikácie, ktoré nevyužívajú žiadne natívne prvky. Medzi hlavné nevýhody mimo výkonu teda patrí:

- *Veľkosť:* Ako každé multiplatformové riešenie, i React Native trpí väčšou výslednou veľkosť aplikácie. To je spôsobené množstvom kódu, ktorý je nutné pre komunikáciu JavaScriptu s natívnymi prvkami.
- *Miera znovu použitelnosti kódu medzi platformami:* Nedosahuje úrovne 100%. Záleží od konfigurácie aplikácie. Častokrát sa stane, že knižnice potrebujú špecifickú konfiguráciu pre jednotlivú platformu. Miera zdielania kódu však stále dokáže dosiahnuť úrovne až 90%

- *Nevhodnosť pre úlohy náročné na CPU:* Ďalšia zdieľaná vlastnosť s ostatnými multiplatformovými riešeniami, React Native nie je najvhodnejším frameworkom pre aplikácie typu AR a Machine Learning, keďže mnoho zdrojov je obetovaných práve pre správne fungovanie aplikácie ako takej. [65]

5 APPLE MACHINE LEARNING EKOSYSTÉM

Apple výrazne investuje do možností strojového učenia pre svoje vývojové platformy. Akvizície ML startupov umožnili využiť tieto technológie v rámci Apple hardware, operačného systému a software, vďaka čomu vznikli frameworky špecificky navrhnuté pre Apple produkty využívajúc široký záber možností. Vývojári môžu využiť množstvo vstupných dát ako video, audio, text a senzory pre tvorbu svojich ML aplikácii pre rôzne sektory, ako zdravotníctvo a rozšírená realita.

ML model vykonáva predikcie na základe vstupných dát **priamo na zariadení** bez toho, aby odosielal dáta na server a teda výsledky nikdy neopustia zariadenie užívateľa. Táto technika sa nazýva *„On-device inference“*.
Aplikácie, ktoré tento princíp nevyužívajú odosielajú dáta na cloudový server, na ktorom je uložený ML model vykonávajúci predukcie na základe dát, ktoré prijme od aplikácie a naspäť odosiela výslednú predikciu. Nevýhodou je nutnosť internetového pripojenia.

Apple Machine learning ekosystém zahŕňa framework *Core ML* a aplikáciu *CreateML* umožňujúcu tvorbu modelov špecificky pre Core ML. [66]

5.1 Core ML

Core ML je machine learning framework pre integráciu ML modelov do iOS aplikácii. Je plne kompatibilný so všetkými Apple produktami a ponúka vysoký býkon a jednoduchú integráciu modelov.
iPhone aplikácia využíva Core ML API a uložené dáta pre analýzu informácii, detekovanie vzorov. Predikcia ako aj tréning sa odohráva priamo na zariadení.

Obrázok 36 Core ML flow [67]

Core ML model je výsledkom aplikovania ML algoritmu pre trénovanie dát. Ten je možné vytvoriť pomocou Create ML, ktorý je súčasťou IDE XCode. V tomto prípade nie je nutná žiadna konverzia, keďže modely trénované pomocou Core ML už sú v požadovanom formáte. Existujú však Core ML nástroje a knižnice tretích strán, ktoré umožňujú konverziu modelu na požadovaný formát.

Framework samotný je postavený na troch hlavných častiach:

- *CPU (Central Processing Unit):* Zodpovedný za pamäťovo náročné procedúry, ako je napr. *Natural Language Processing (NLP).*
- *GPU (Graphics Processing Unit):* Zodpovedná za výpočetne náročné úkony, ako je napr. práca s obrázkami a object detection.
- *ANE (Apple Neural Engine):* Zodpovedný za urýchlovanie neurónových sietí

Core ML umožňuje prepínanie medzi týmito časťami za behu. Výsledkom je optimalizácia výkonu priamo na zariadení. [67]

5.1.1 Architektúra a dostupné modely

Core ML podporuje štyri tréningové domény definujúce jeho architektúru: *Vision, Natural Language Processing (NLP), speech recognition* a *sound analysis.*

- *Vision* spracováva úlohy pre rospoznávanie tváre, textu, detegovanie objektov a obrazovú segmentáciu. Samotná knižnica využíva kamera pre vstupné dáta, ktoré sú následne spracované pomocou *AVfoundation* - multimediálneho frameworku, ktorý je súčasťou UIkitu.
- *NLP* ponúka *speech recognition a natural language processing* za pomoci funkcionalít ako jazyková identifikácia a zber mien z textu.
- Speech recognition a sound analysis sú schopné sledovať hlasové vzory a trénovať modely určené pre identifikáciu rôznych hlasov.

Navyše existuje I Gameplay Kit, ktorý organizuje hernú logiku a pomáha s rozhodovacími stromami. [67]

5.1.2 Integrácia modelov tretích strán

Modely tretích strán musia byť konvertované do Core ML formátu pomocou pythonového balíka *coremltools*. Zoznam podporovaných modelov narastá každým rokom. V roku 2020

bola pridaná podpora pre populárne knižnice *TensorFlow* a *PyTorch.* V tom istom roku bol na trh uvedený nastroj pre konverziu zvaný *„Model Intermediate Language – MIL,* ktorý bol vytvorený pre zjednotenie procesu konverzie a a zjednodušil podporu pre potenciálne nové frameworky. Poskytuje teda štandardné rozhranie pre zachytávanie informácii z rôznych frameworkov. [67]

5.1.3 Zhrnutie

Apple neustále zlepšuje svoje machine learning technológie a Core ML sa dynamický vyvíja každým rokom, pričom prináša stále viac možností, ako ML integrovať do iOS aplikácii. Medzi jeho silné stránky patrí:

- Súkromie, dáta nikdy neopustia užívateľove zariadenie
- Nízka doba odozvy
- Kompatibilita s väčšinou populárnych ML modelov
- Automatické prepínanie medzi CPU, GPU a ANE
- Offline dostupnosť

Uložením modelu priamo v zariadení sa však zvyšuje celková veľkosť aplikácie a vďaka potrebnému výpočetnému výkonu sa zároveň zvyšuje náročnosť na batériu. [67]

5.2 Create ML

Create ML umožňuje trénovanie Core ML modelov priamo na zariadení s macOS bez nutnosti písania ďalšieho kódu. Výhodou aplikácie je jednoduché a prehľadné užívateľské rozhranie spolu s podporou množstva kategórii modelov vhodných pre trénovanie. Obsahuje ďalšie nástroje pre kontrolu kvality výsledného modelu, ako snapshots a ďalšie nástroje pre vizualizáciu trénovacieho procesu a presnosti. [68]

Obrázok 37 Create ML UI

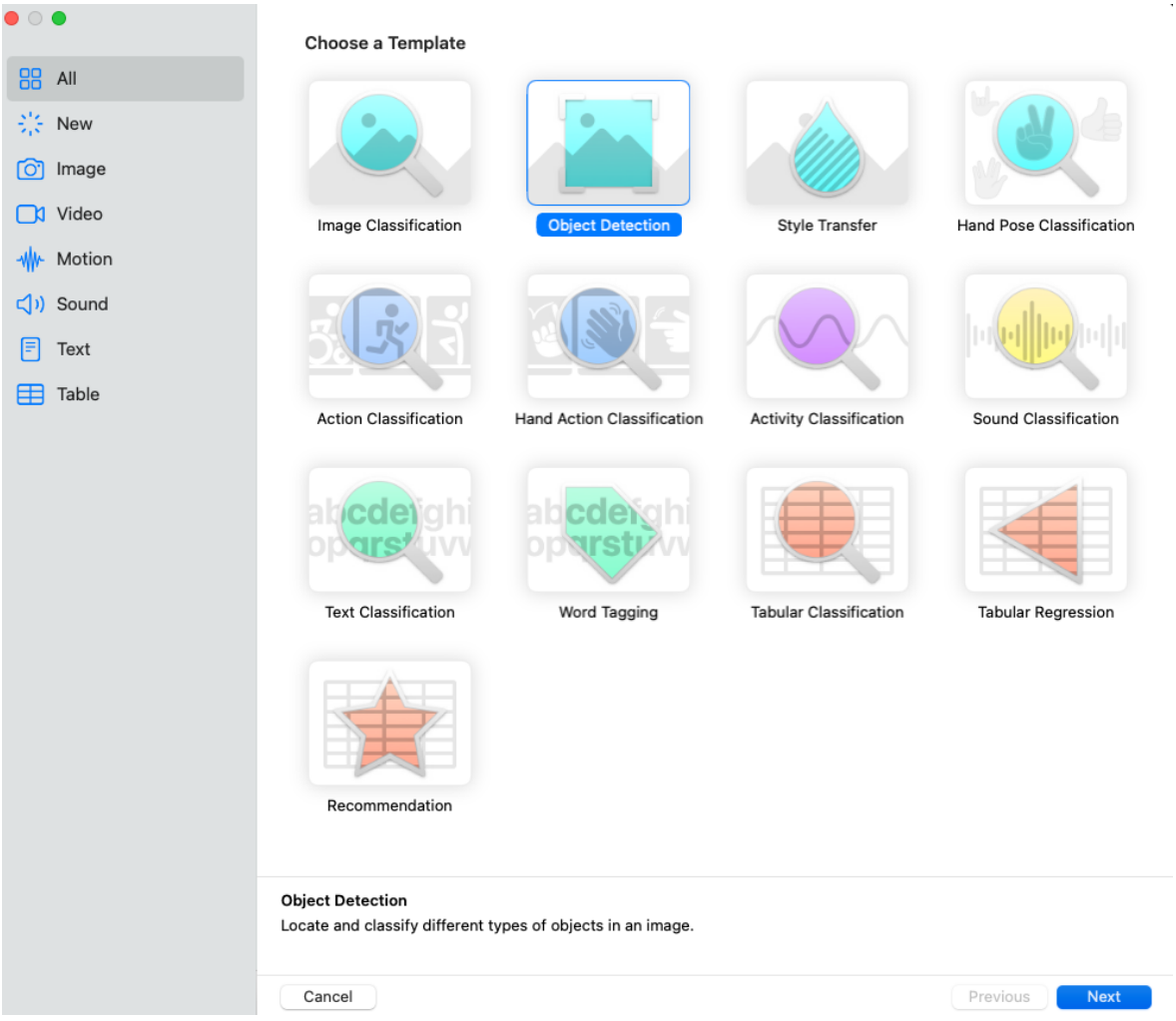
5.2.1 Modelový prehľad

Širokú škálu modelov, ktoré je možné trénovať pomocou Create ML je možné rozdeliť do nasledujúcich kategórii, ktoré majú plnú podporu Core ML:

- *Image:* modely pre image classification a object detection. V rámci manipulácie s obrazom je možné trénovať taktiež *Style Transfer* modely, ktoré sú schopné prevziať štýl referenčného obrazu a aplikovať ho na obraz alebo video. Novinkou je taktiež detekcia pózy ruky.
- *Video:* Medzi špeciálne video modely patrí podobne ako pri obrázkoch *Style Transfer, Action Classification* pre rozpoznávanie pohybu ľudského tela a taktiež

Hand Action Classification určený špecificky pre rozponávanie jednotlivých pohybov ruky.

- *Motion: Activity Classification modely* klasifikujú dominantnú aktivitu zachytenú senzorom.
- *Sound:* Zahŕňa modely pre klasifikáciu zvuku, napríklad plač dieťaťa alebo brechot psa.
- *Text:* Modely tejto kategórie dokážu klasifikovať témy, sentiment, dominantný predmet, prípadne označiť slová na základe požiadaviek,
- *Table: Tabular Classification* predikuje kategorickú hodnotu vlastnosti na základe jej hodnôt. Tieto vlastnosti sú reprezentované ako stĺpce. *Recommendation* modely odporúčajú nové položky na základe predchádzajúcej interakcie. [69]



5.2.2 Trénovacie parametre pre Object Detection

V predvolenom nastavení Create ML trénuje modely pomocou algoritmu YOLO, jedným z najefektívnejších algoritmov na trénovanie modelov detekcie objektov. Create ML to nazýva *„full network“*. [70]

Alternatívnym prístupom k trénovaniu modelu je tzv. „*Transfer Learning“,* ktorý v sebe zahŕňa tvorbu nového modelu na základe už existujúceho. V prenesenom význame môžme tvrdiť, že model „predáva“ jeho znalosť do nového modelu. Apple pre tento účel poskytuje predtrénovaný model trénovaný na objemnom množstve dát, nekonkretizuje však, o aký model sa jedná. Tento prístup prináša rýchlejšie výsledky a výsledný súbor modelu je taktiež nižší. [70]

II. PRAKTICKÁ ČASŤ

6 MODEL DETEKCIE ĎATELINY

Úloha rozpoznávania ďateliny a to konkrétne štvorlístkov v rámci computer vision prináša niekoľko otázok. Z prvotne pomerne jednoduchej myšlienky detekcie sa stáva problém, pri ktorom je nutné vziať do úvahy niekoľko faktorov a to: rôzne tvary štvorlístkov, ich deformácie, napríklad z dôvodu ohryzenia hmyzom, prekrývanie steblami trávy a inou flórou, ktorá môže rásť v okolí. V neposlednom rade je nutné vziať do úvahy samotnú ďatelinu, ktorá sa taktiež môže či už prekrývať, alebo mať rôznu výšku a tým potenciálne ovplyvňovať výsledný odhad. Tieto faktory je možné do významnej miery pozitívne ovplyvniť práve vhodnou anotáciou dát, ktorá je prvým krokom pri tvorbe modelu.

6.1 Anotácia trénovacieho datasetu

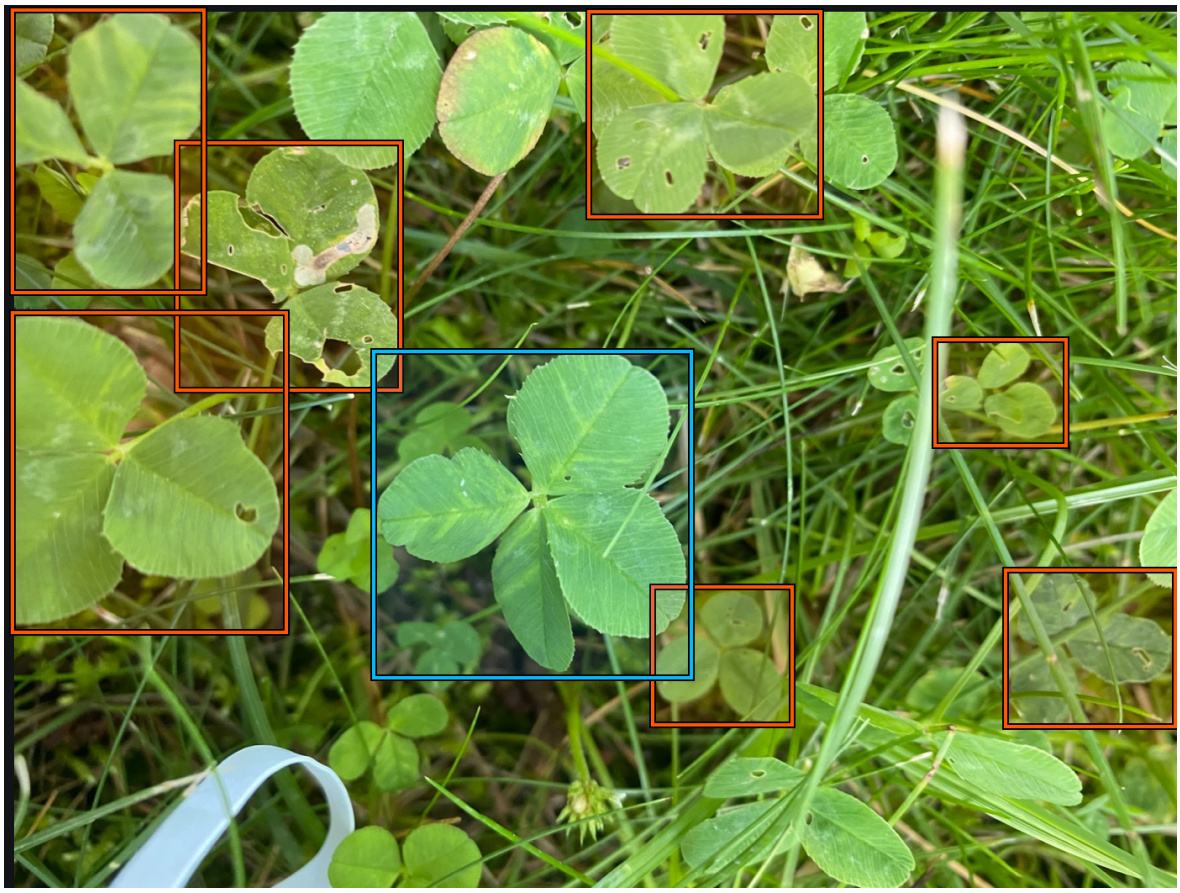
Pôvodný poskytnutý dataset obsahuje 100 fotiek obsahujúcich nielen štvorlístky, ale aj trojlístky v jeho okolí. Výsledný model bude schopný rozpoznávať teda oba druhy ďateliny práve pre vyššiu presnosť detekcie štvorlístku a zmedzeniu potenciálneho vzniku tzv. „false positives“ a teda stavu, kedy model síce rozpozná objekt, no nejde o jeho skutočný výskyt na detekovanom obrázku. V reálnom prípade sa totiž jedná o najväčšie riziko chybnej detekcie. Štvorlístky ako také sú iba jednoduchá mutácia ďateliny a jej výskyt je výrazne menší, ako je to u trojlístkov. Berúc do úvahy všetky vyššie uvedené skutočnosti je prirodzeným ďalším krokom voľba vhodného anotačného nástroja.

Výsledný anotovaný dataset musí podporovať Apple Core ML štandard a teda anotácie musia byť uložené vo formáte JSON. Keďže anotácie budú vykonávané ručne, je taktiež nutné zvoliť taký nástroj, ktorý dokáže priebežne ukladať doposiaľ anotované dáta na internetové úložisko a taktiež umožniť jednoduchý editáciu a pridávanie ďalších dát. Opierajúc sa o rešeršu z teoretickej časti sa zvoleným nástrojom stal IBM Cloud Annotations.

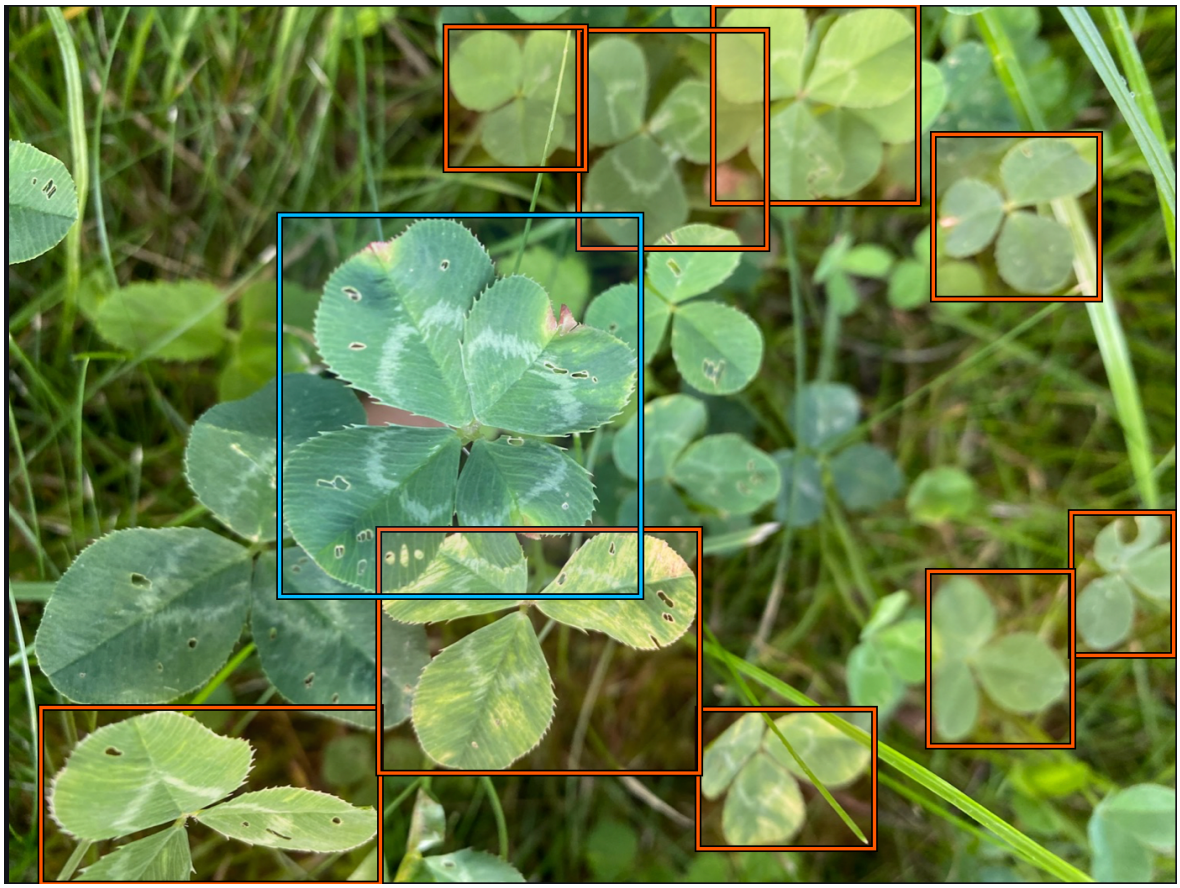
Ukladanie anotovaných dát je v tejto aplikácii vo forme tzv. „Buckets“, ktoré udržiavajú prehľad všetkých pridaných dát, v rámci ktorých sa vytvárajú labels, ktoré slúžia ako unikátny identifikátor jednotliej triedy dát. V prípade tohto modelu sú labels dve a to:

- Three-leaf reprezentujúce trojlístky
- Four-leaf reprezentujúce štvorlístky

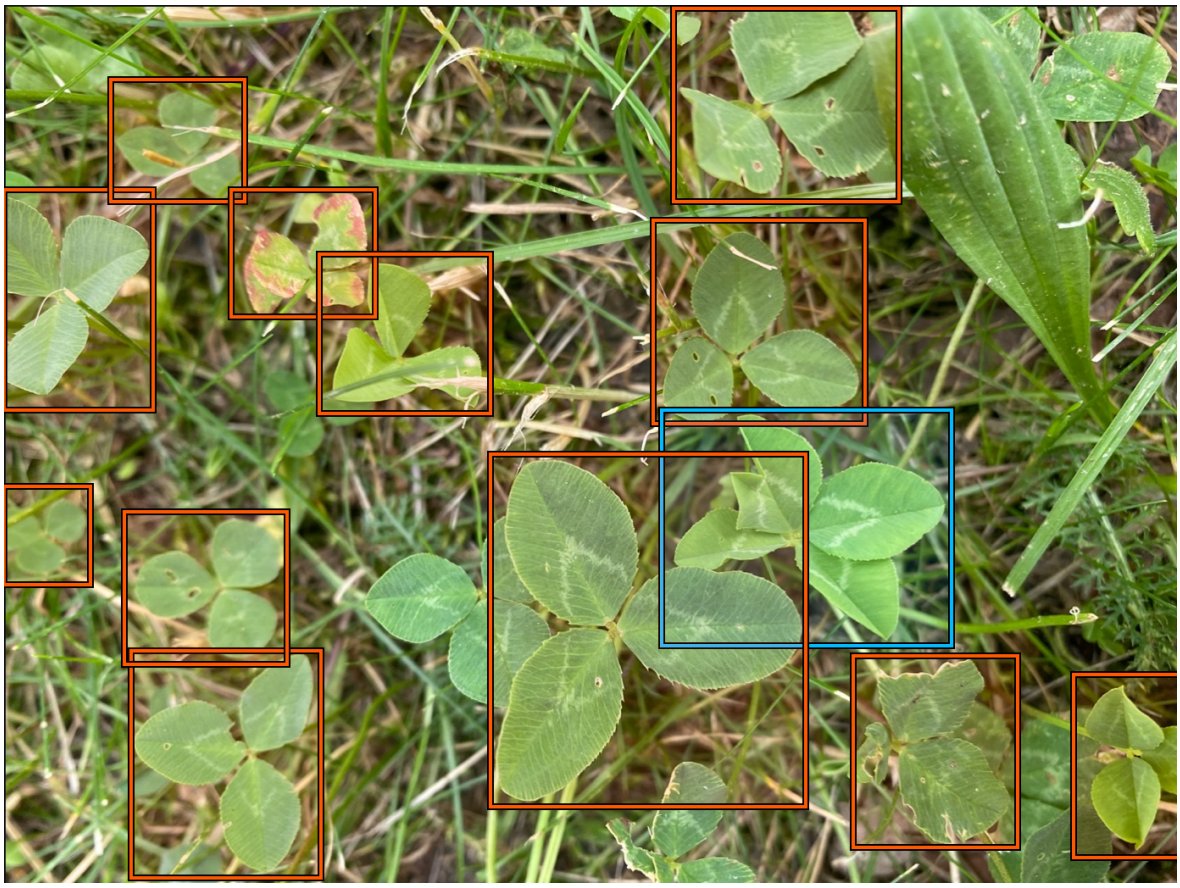
Potenciálnou nevýhodou tohto nástroja je nemožnosť anotovať iným spôsobom, než obdĺžnkovými bounding boxami, no pre potreby súčasnej aplikácie je i tento spôsob viac než dostačujúci. Ukážky jednotlivých anotovaných zdrojových dát, kde štvorlístky sú označené modrým bounding boxom a trojlístky naopak červeným, sú nižšie:



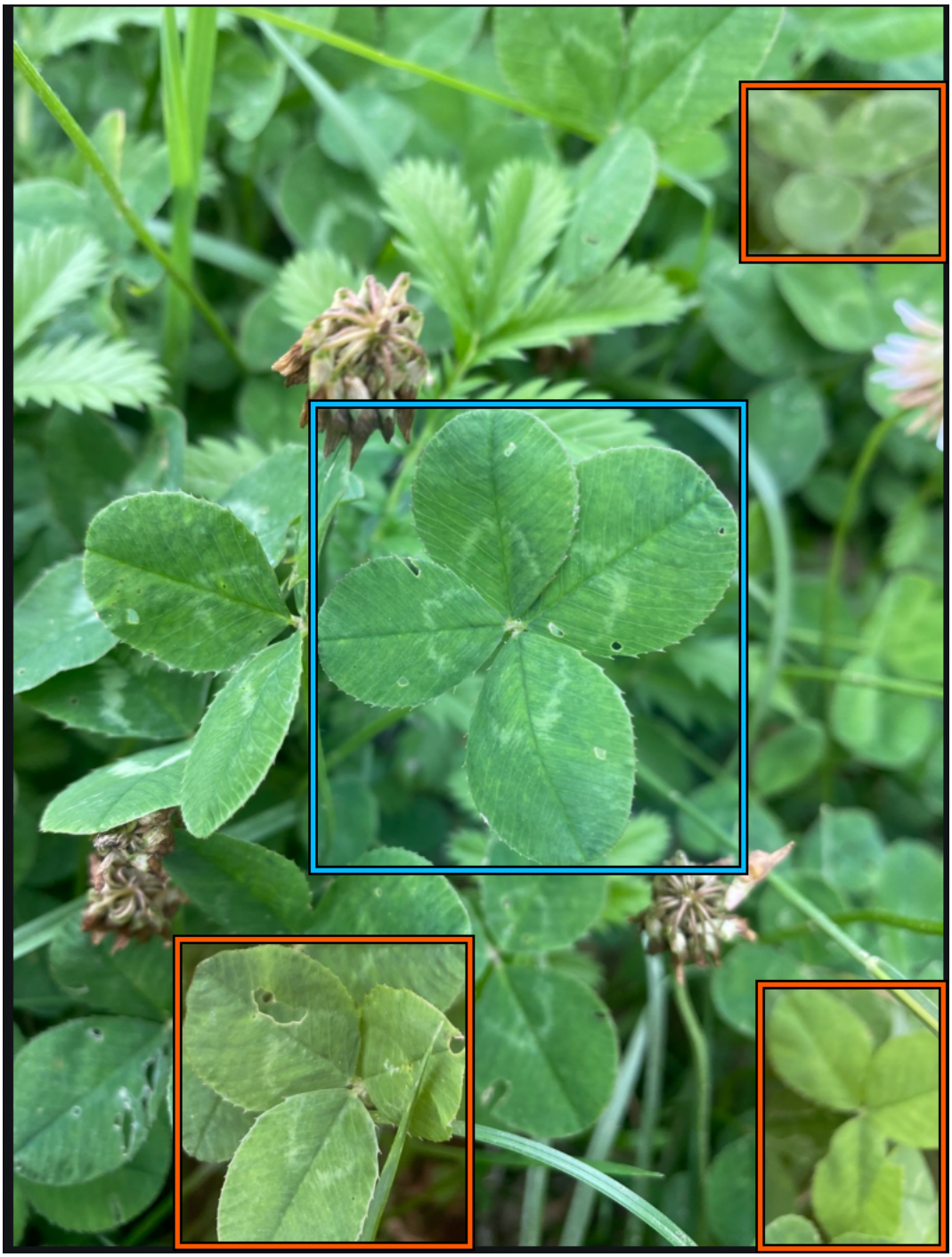
Obrázok 38 Anotovanie štvorlístkov a trojlístkov



Obrázok 39 Príklad prekrývania jednotlivých lístkov



Obrázok 40 Deformovaný štvorlístok



Obrázok 41 Orezané trojlístky ako validný príklad

Na obrázkoch vyššie sú jasne demonštrované ťažkosti pri trénovaní modelu spomenuté začiatkom kapitoly. Cieľom takto tvorenej anotácie je najmä zamedziť vzniku chybných predikcii a maximalizovať tak efektivitu pri reálnom používaní aplikácie.

Výsledný anotovaný dataset je nutné exportovať. Cloud Annotations ponúka viacero možností, medzi nimi YOLO, Pascal VOC, no keďže sa práca zameriava na Apple machine

learning ekosystém, dáta sú exportované v Create ML formáte. Samotný export zahŕňa ako obrázky, tak aj súbor *annotations.json*, ktorého čiastočná štruktúra je zobrazená nižšie.

```
{
  "image": "0a8e6b21-c3f2-4ce2-abd2-af9bd92c9a3a.jpg",
  "annotations": [
    {
      "label": "three-leaf",
      "coordinates": { "x": 216, "y": 1195, "width": 396, "height": 468 }
    },
    {
      "label": "three-leaf",
      "coordinates": { "x": 238, "y": 540, "width": 389, "height": 286 }
    },
    {
      "label": "four-leaf",
      "coordinates": { "x": 555, "y": 873, "width": 376, "height": 410 }
    }
  ]
},
```

Každý objekt v poli reprezentuje jeden obrázok. Medzi základné fields patrí *image*, reprezentujúci názov obrázka a samotné annotations obsahujúce výslednú *label* a jej koordinácie.

Výsledkom anotačného procesu je teda zložka obsahujúca pripravený, manuálne označovaný dataset, ktorý je možné okamžitre použiť pre trénovanie požadovaného modelu.

6.2 Trénovanie Core ML modelu

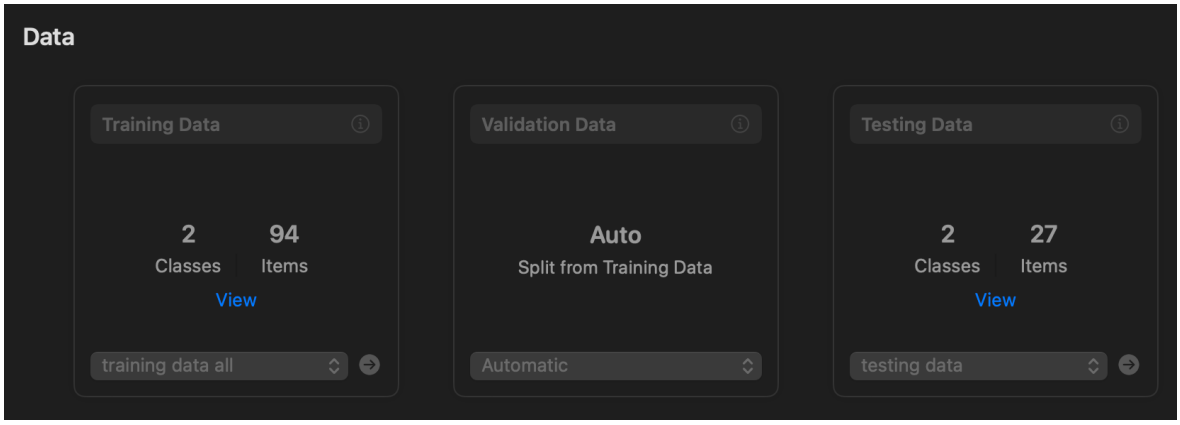
Trénovanie samotné je možné vykonávať pomocou niekoľkých nástrojov, medzi nim napr. známy TensorFlow, ktorého výsledný model je následne možné pomocou *coremltools* prekonvertovať na požadovaný Core ML formát. Apple však poskytuje aplikáciu Create ML, ktorá poskytuje nástroje nielen pre trénovanie ale i testovanie Core ML modelov a teda nutnosť konverzie modelu sa úplne vytráca.

Celkový proces je jednoduchý a intuitívny. Pri tvorbe ML projektu je nutné si zvoliť správny typ trénovaného modelu, ktorým je v tomto prípade *Object Detection* model.
Následné UI poskytuje možnosť pridania troch skupín dát a to:

- Trénovacie

- Validačné
- Testovacie

Pre trénovacie dáta je použitý anotovaný dataset popísaný v predchádzajúcej kapitole. Podstatnou súčasťou kvalitatívnej stránky modelu sú taktiež validačné dáta umožnujúce prvotné testy oproti novým dátam a jedná sa teda o akúsi pomôcku pri ďalšej optimalizácii a majú vplyv na spôsob, akým model predikuje dáta. Create ML uskutočnuje výber automaticky na základe trénovacích dát.
Testovacie dáta sú súčasťou separátneho datasetu anotovaného rovnakým spôsobom, ako trénovacie a jedná sa o náhodné obrázky stiahnuté z internetu a tvoria zhruba tretinu počtu trénovacích dát. Výsledné rozdelenie je nasledovné:



Obrázok 42 Rozloženie datasetu

S takto pripraveným rozdelením dát ostáva na výber algoritmus, ktorý bude použitý pre trénovanie. Create ML poskytuje dve možnosti a to *Transfer Learning* alebo *Full Network.*

Nutné je dodať, že trénovací proces pre oba algoritmy je vykonávaný na zariadení Apple Macbook Pro v nasledujúcej konfigurácii:

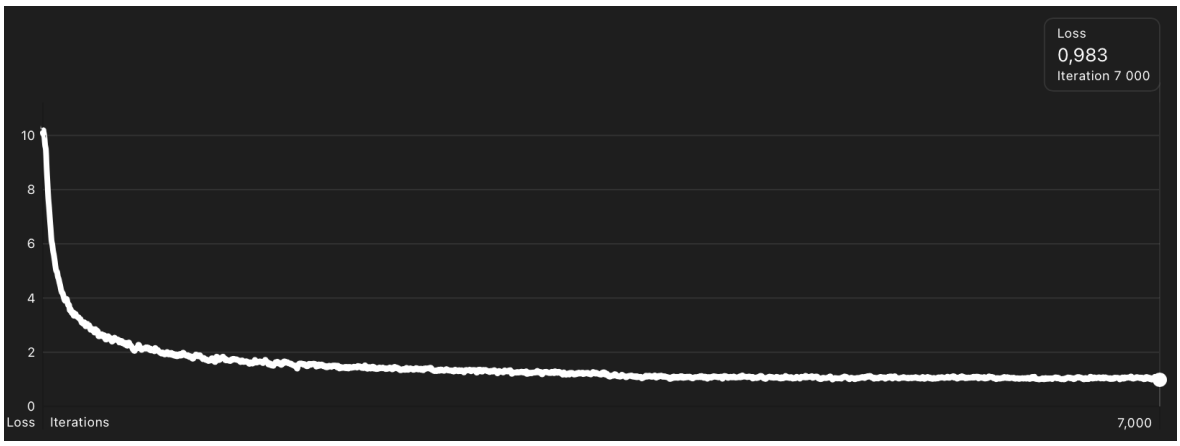
- Procesor Apple M1 Pro 8 jadier 3.22 GHz
- 32 GB RAM
- GPU: Apple M1 Pro 14 jadier 1.28 GHz

6.2.1 Full Network trénovanie

Full Network využíva konvolučnú sieť postavenú na YOLOv2 architektúre, jedným zo štandardných prístupov k object detection. Výhodou takto trénovaného modelu je jeho dostupnosť aj s nižšími verziami iOS, počnúc verziou 12.0, ktorá vyšla v roku 2018 a teda je potenciálne možné podporovať aj výrazne staršie zariadenia.

Ďalšími nastavitelnými parametrami je počet iterácii a batch size. Vzhľadom k veľkosti datasetu nie je nutné nastavovať či už vysoký počet iterácii, alebo batch size, ktorý delí dáta v iterácii na jednotlivé časti. Apple radí, čím vyšší batch size, tým teoreticky kvalitnejší model. Create ML umožňuje automatický výber batch size. Počet iterácii je taktiež volený automaticky a konečnou hodnotou je 7 000 iterácii.

Výsledky trénovania pomocou full network sú nižšie:



Obrázok 43 Priebeh trénovania Full network

Z grafu jednoznačne vyplýva, že nastavený počet iterácii mohol byť skrátený zhruba o polovicu, keďže po iterácii 3500 nenastávajú výraznejšie zmeny presnoti modelu.
Celková doba trénovania dosiahla hodnoty **2 hodiny a 31 minút.**

Hodnota *loss,* ktorú Create ML poskytuje po trénovaní je jedným z meradiel kvality modelu. Čím je hodnota bližšia nule, tým je model kvalitnejší a získavame ju výpočtom na trénovacích a validačných dátach. Konkrétne sa jedná o sumáciu chýb na každom príklade na trénovacích a validačných dátach.

Pre ďalšie zhodnotenie modelu poskytuje Create ML hodnotu *Intersection Over Union*, ktorá porovnáva bounding boxy na anotáciách s boxami tvorené samotným modelom. Create ML udáva túto hodnotu ako I/U > 50% reprezentujúca percentuálnu mieru predikcii, ktorých bounding boxy sa prekrývali s anotovanými boxami väčšou plochou ako 50%. Pre jednotlivé labels model dosiahol tieto hodnoty I/U:

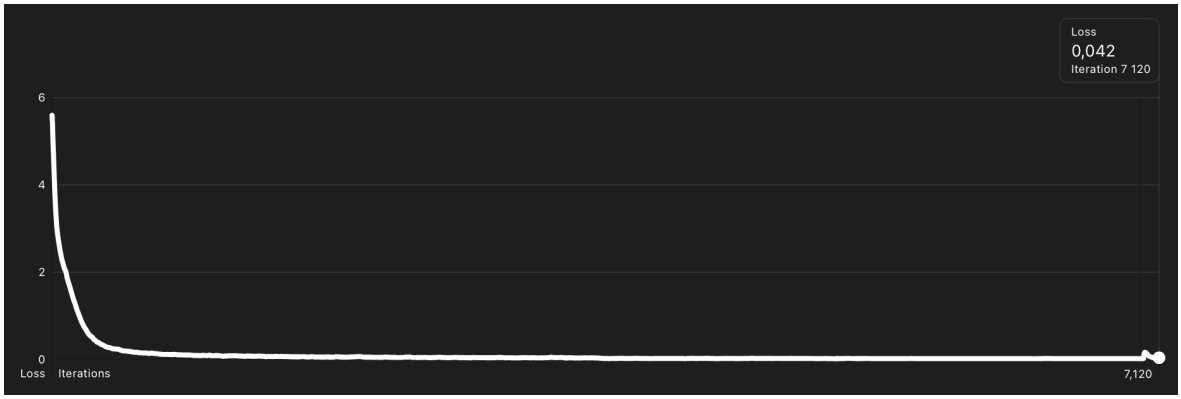
- Three-leaf: 22%
- Four-leaf: 80%

Výsledná veľkosť modelu na disku je 31Mb.

6.2.2 Transfer Learning trénovanie

Táto metóda trénuje detektor za použitím *object vision feature print*. Model sa učí na menej parametroch a berie do úvahy najvýraznejšie features. Častokrát je táto metóda robustnejšia vzhľadom k celkovému počtu trénovacích dát a výsledný model je taktiež menší. Takto trénovaný model je možné použiť na zariadeniach podporujúcich verziu iOS 14 a vyššie.

Nastavenie parametrov ako batch size a počet iterácii je rovnaké, ako v predošlom prípade a teda *auto* a 7000. To isté platí i pre trénovacie, validačné a testovacie dáta. Výsledný priebeh trénovania je nasledovný:



Obrázok 44 Priebeh trénovania Transfer learning metódou

Podobne ako v prípade Full Network i v tomto prípade mohol byť počet iterácii nastavený nižšie. Rozdielom je však bod, kedy graf konverguje k minimu a v tomto prípade je teda

možné tvrdiť, že k vytrénovaniu modelu do výslednej podoby na daných dátach by postačilo 1000 iterácii a teda 3.5x menej iterácii, ako by to bolo v prípade FN.
Celková doba trénovania modelu na rovnakom hardware bola taktiež výrazne kratšia (pri 700 iteráciách) a teda **51 minút.**

Loss vykazuje v porovnaní taktiež lepšie výsledky, konkrétne hodnotu **0,042.**

I/U > 50% pre jednotlivé labels sú nasledovné:

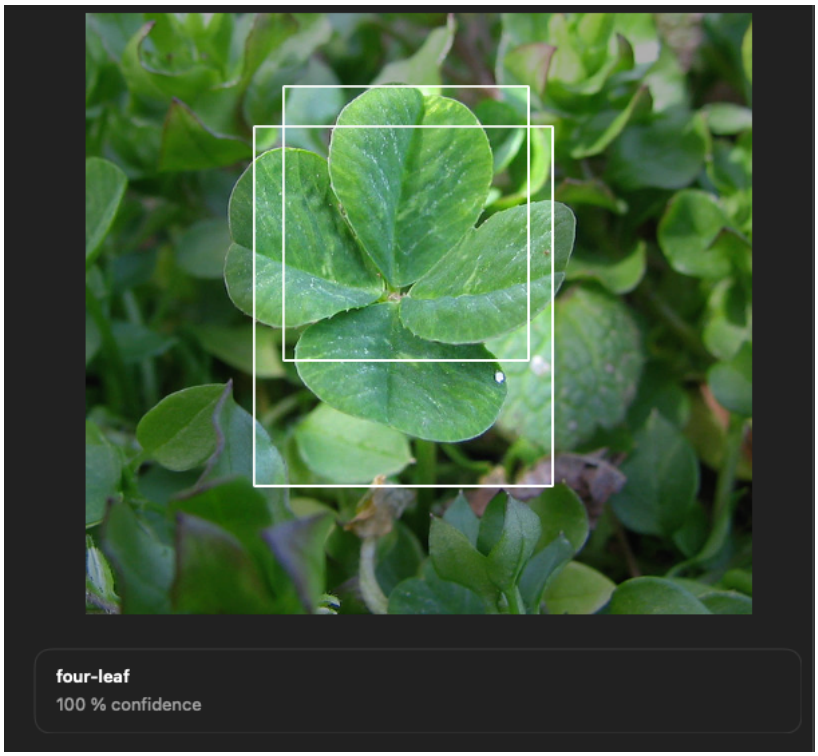
- Three-leaf: 32%
- Four-leaf: 90%

Výsledný model má veľkosť 6,8 Mb a je teda **5x menší**, ako model trénovaný pomocou FN metódy.

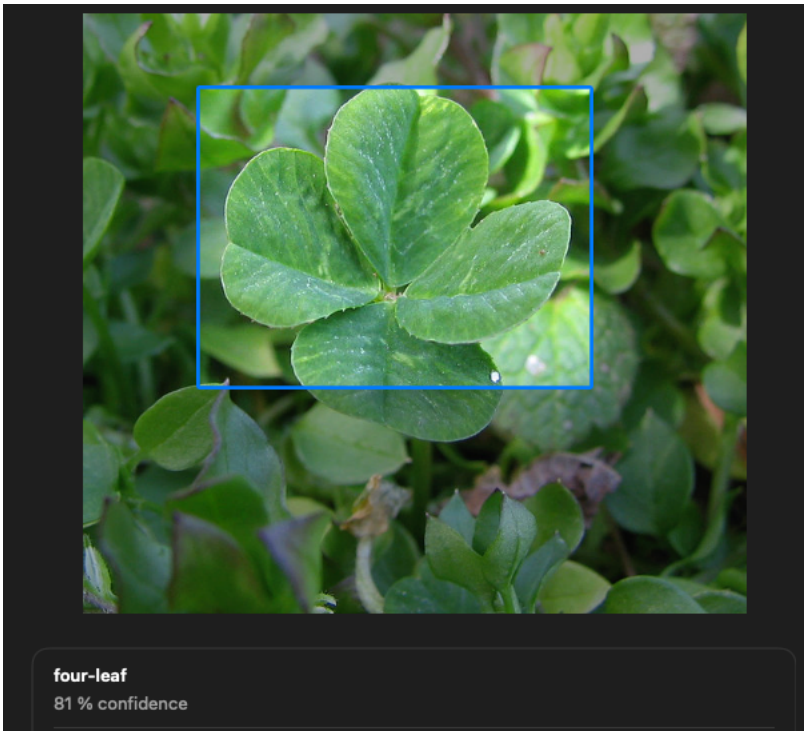
V rámci testovania bolo dodatočne spustených ďalších 120 iterácii, ako je však zjavné z obrázku 44, model vykazoval známky overfittingu.

6.2.3 Vyhodnotenie trénovaných modelov

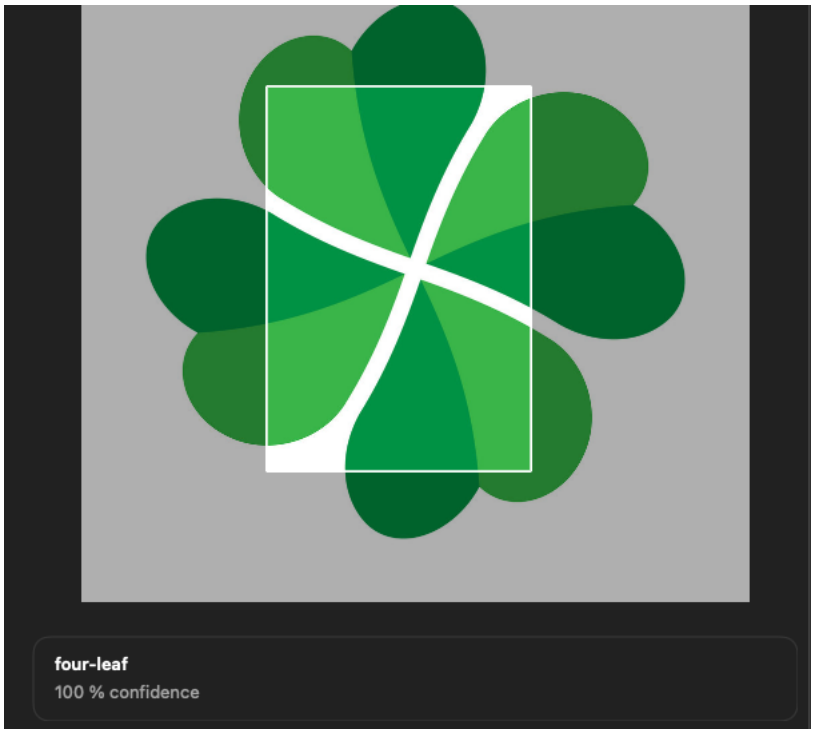
Voľba výsledného modelu podlieha niekoľkým faktorom. Keďže sa jedná o on-device machine learning a teda žiadne dáta nie sú odosielané na server, konečná veľkosť mobilnej aplikácie býva spravidla väčšia a je teda žiadúce minimalizovať objem dát, ktoré budú súčasťou aplikácie. Zároveň je nutné zvoliť taký model, ktorý bude vykonávať čo najpresnejšie predikcie. Z Create ML testovania jednoznačne vyplýva, že tieto dve požiadavky jasne spĺňa model trénovaný pomocou metódy TN, ktorý je nielen objemovo menší, no vykazuje I vyššiu mieru predikcie na testovacích dátach. Oba modely dosahujú nízku hodnotu I/U pre trojlístky, to je však zapríčinené manuálnou anotáciou testovacích dát, na ktorých model identifikoval i trojlístky, ktoré neboli anotované a teda prekonal pôvodné očakávania v rámci predikcii.
Pre manuálnu kontrolu poskytuje Create ML záložku *preview*, kde je možné si overiť predikcie a bounding boxy tvorené modelom. Niektoré z pozorovaní pre oba modely sú uvedené nižšie:



Obrázok 45 Transfer Learning - 100% confidence na reálnom príklade

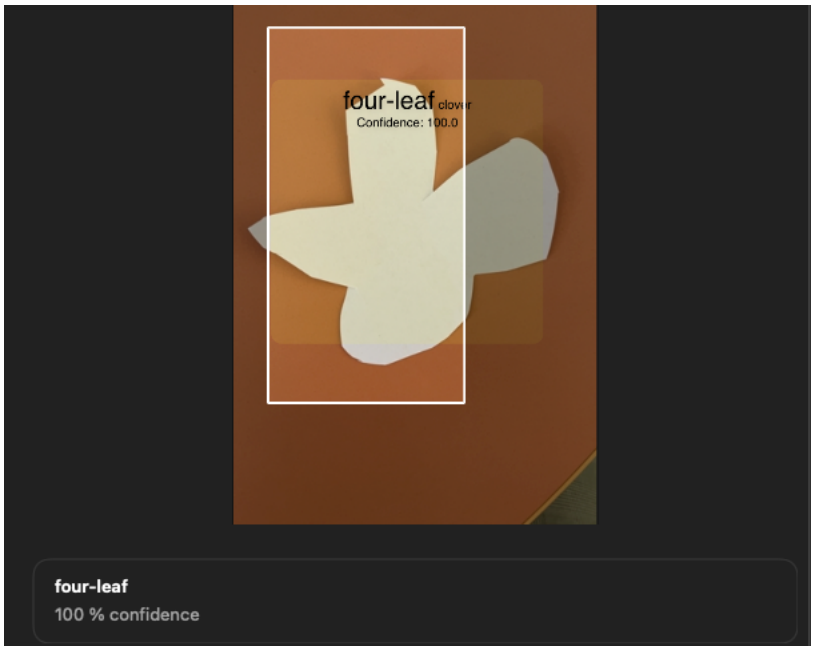


Obrázok 46 Full network - 81% confidence na reálnom príklade

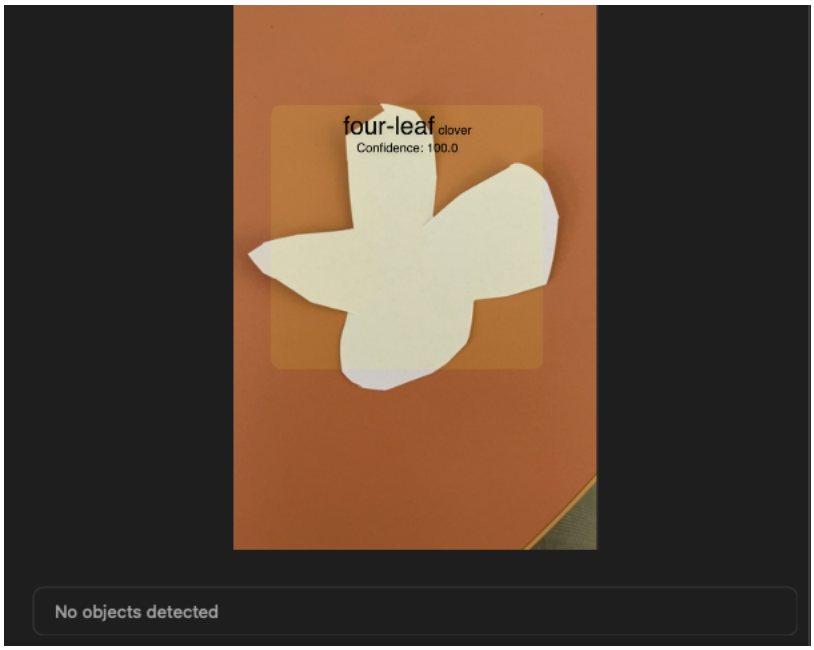


Obrázok 47 Detekcia "false positive" pri Transfer learning

Obrázok 48 Full network nepovažuje reprezentácie štvorlístku za validný objekt



Obrázok 49 Transfer learning ukážka feature printu, model považuje za rozhodujúci tvar, než farbu



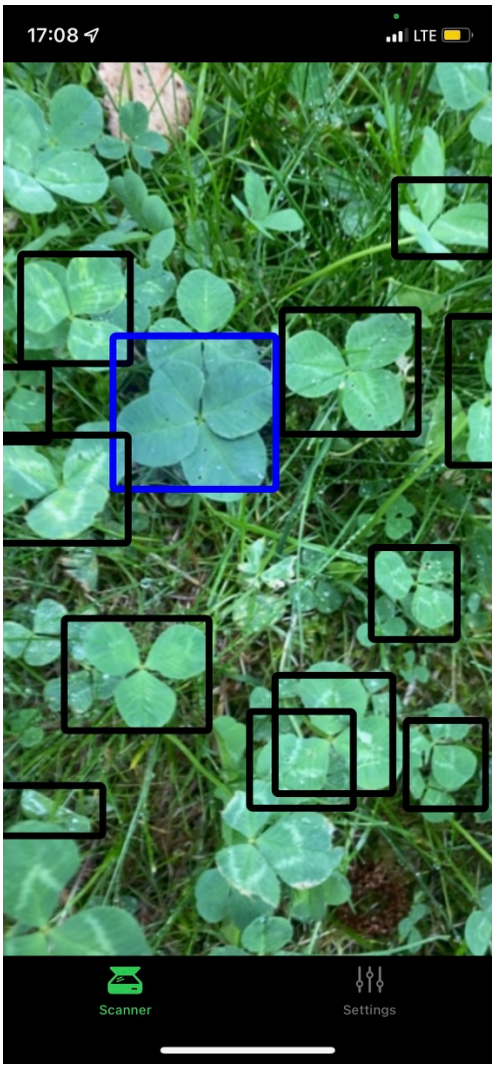
Obrázok 50 Full network opäť nepovažuje vizuálne reprezerntácie za reálne štvorlístok

Pri zadaní reálneho príkladu štvorlístka dokázal TL model s istotou 100% lokalizovať štvorlístok. Pri FN bola táto hodnota nižšia a to 81%. Napriek tomu, že TL model bol v prvotných testoch presnejší, spôsob, akým identifikuje objekt na obrázku sa líši. Primárne berie v úvahu dominantné features ako tvar a krivky bez ohľadu na farbu alebo „reálnosť daného objektu“ a teda určuje ako štvorlístok tzv „false positives“ testovacie dáta vo forme

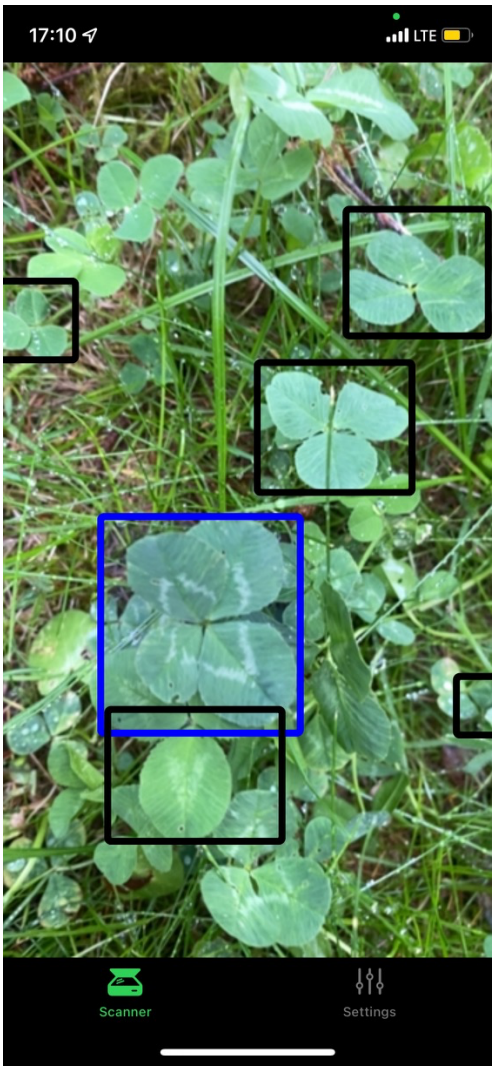
nakresleného štvorlístka a výstrižku papiera, ktorý sa na štvorlístok podobá tvarom. Princíp je podobný, ako keď človek rozoznáva obrazce napr. v oblakoch. Určité tvary sa môžu podobať na objekty reálneho sveta. FN model tento „problém“ nemá a či už papierový ústrižok alebo grafickú reprezentáciu štvorlístka neoznačil ako detekovaný objekt.
Tento fakt napovedá o vyššej vhodnosti FN modelu, avšak s prihliadnutím na využitie aplikácie (offline, v zariadení so snahou nízkeho objemu dát) a skutočnosťou, že ďatelina rastie výhradne na trávnatých plochách a výskyt je teda nemenný, ostáva TL model stále vhodnejším kandidátom pre výslednú implementáciu mobilnej aplikácie, keďže jeho výhody prevyšujú nevýhody vo forme false positives.

Testovanie v reálnom teréne vykazovalo pozitívne výsledky, kde vďaka aplikácii bolo možné nájsť štvorlístok v spleti trojlístkov za použitia Transfer learning modelu. Najlepšie výsledky boli dosiahnuté približne 30cm od zeme, presnosť detekcie sa znižovala so zväčšujúcou sa vzdialenosťou. Jedna z prvotných verzii aplikácie bola schopná úspešne detekovať ako očakávané trojlístky, tak i štvorlístky s vysokou confidence, ktoré dosahovala priemerne > 90%.

Obrázok 51 Reálne výsledky „v teréne“



Obrázok 52 Úspešný príklad detekcie ako trojlístku, tak i štvorlístku - 1



Obrázok 53 Úspešný príklad detekcie ako trojlístku, tak i štvorlístku - 2

7 IMPLEMENTÁCIA MOBILNEJ APLIKÁCIE

Aplikácia je určená pre aktívnych ľudí a hobbistov, ktorí si chcú spestriť či už svoju prechádzku alebo pobyt vonku milou činnosťou hľadaním štvorlístkov. Úlohou je teda poskytnúť možnosť on-device machine learningu bežným užívateľom a vytvoriť tak praktický use-case, mimo najčastejšie vyskytujúceho sa priemyselného využitia. Od toho sa teda odvíja niekoľko funkcionálnych, ale i nefunkcionálnych požiadaviek.

7.1 Požiadavky

Prvotný návrh aplikácie sa odvíja od niekoľkých požiadaviek, ktoré je možné rozdeliť na funkcionálne a nefunkcionálne.

Funkcionálne požiadavky sú popisom funkcionality aplikácie a jej súčastí. Jedná sa o sadu základnej funkcionality alebo želaného chovania.

Nefunkcionálne požiadavky definujú vlastnosti aplikácie, ktoré nie sú jasne definované funkcionality, no dokážu ich výrazne ovplyvniť. Slúžia teda nielen ako technologické nároky na aplikáciu, ale i ako určité obmedzenia.

7.1.1 Funkcionálne požiadavky

R1. Aplikácia musí byť schopná detekovať štvorlístky v reálnom čase

R2. Aplikácia musí byť schopná detekovať trojlístky v reálnom čase

R3. Aplikácia musí byť schopná zobraziť úvodné obrazovky pomáhajúce užívateľovi s prvými krokmi pri používaní aplikácie

R4. Aplikácia musí byť schopná umožniť zmenu jednotlivých nastavení a personalizovať tak detekciu podľa potrieb užívateľa

R5. Aplikácia musí byť schopná uložiť užívateľove preferencie v rámci úložiska na zariadení

7.1.2 Nefunkcionálne požiadavky

NR1. Aplikácia musí fungovať nezávisle od užívateľovho stavu pripojenia

NR2. Aplikácia musí byť jednoducho škálovateľná

NR3. Aplikácia musí byť natívna

7.2 Návrh užívateľského rozhrania

Z požiadaviek vyplýva nutnosť implementácie troch obrazoviek:

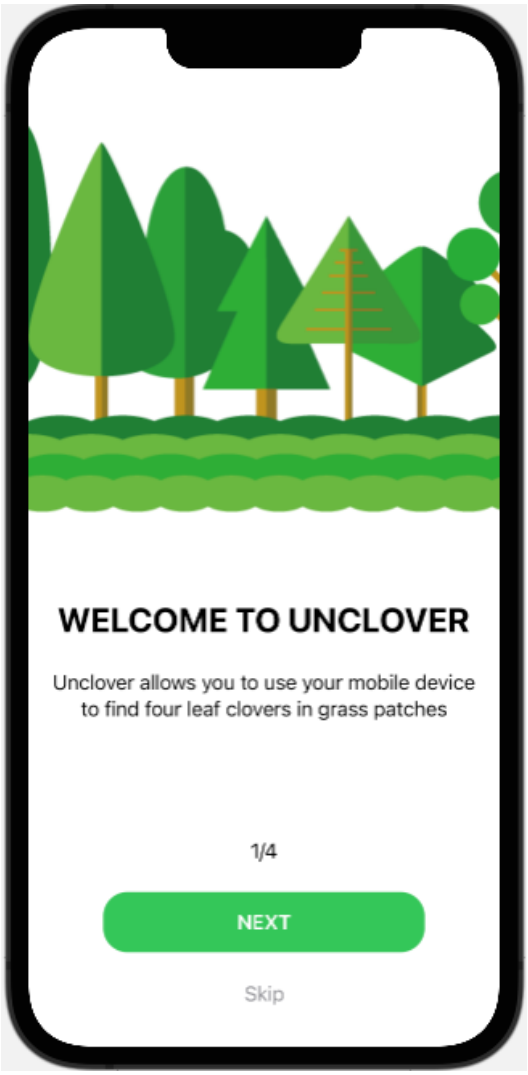
- Onboarding Screen
- Scanner Screen
- Settings Screen

7.2.1 Onboarding Screen

Úlohou obrazovky je zoznámiť užívateľa s aplikáciou a v krátkosti objasniť všetku funkcionalitu obsiahnutú v aplikácii. Typickým spôsobom ako toho docieliť je pomocou formy krátkej prezentácie. Medzi takúto funkcionalitu patrí:

- Privítanie a krátke uvedenie cieľa
- Detekcia na zariadení
- Použitie offline
- Možnosť úpravy aplikácie

Užívateľ môže túto úvodnú obrazovku buď preskočiť alebo prechádzať jednotlivými informácii pomocou dedikovaného tlačidla. V prípade, že užívateľ tento onboarding dokončil, je presmerovaný do jadra aplikácie.



Obrázok 54 Ukážka prvej uvítacej screen

7.2.2 Scanner Screen

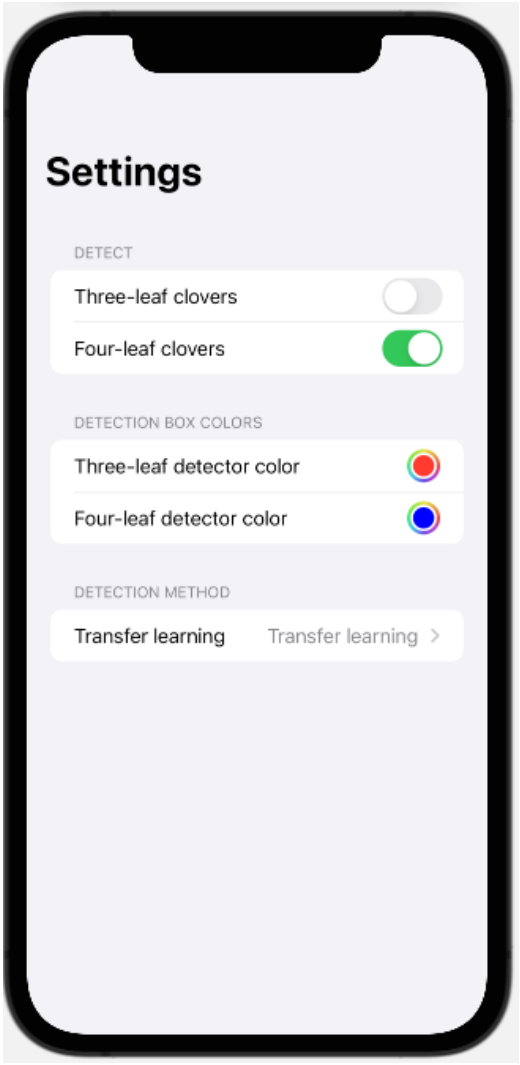
Obrazovka obsahujúca hlavnú funkcionalitu aplikácie a teda detekciu či už troj alebo štvorlístkov. Celú obrazovku musí pokrývať živý camera feed, na ktorom sa v prípade úspešnej predikcie zobrazí bounding box, ktorý ju ohraničuje.

Obrázok 55 Scanner obrazovka pre jeden obrázok z testovacích dát

7.2.3 Settings Screen

Obsahuje nastavenia, vďaka ktorým si užívateľ môže personalizovať, akým spôsobom chce aplikáciu používať. Medzi tieto nastavenia patrí:

- Možnosť zapnút/vypnúť detekciu trojlístkov
- Možnosť zapnút/vypnúť detekciu štvorlístkov
- Možnosť zmeny farby bounding boxu detekovaných trojlístkov
- Možnosť zmeny farby bounding boxu detekovaných štvorlístkov
- Možnosť zmeny modelu používaného pre detekciu



Obrázok 56 Settings obrazovka

7.3 Architektúra aplikácie

Pri tvorbe natívnej aplikácie, ktorá je schopná využívať Core ML framework sa ponúka ako jediná vhodná možnosť programovací jazyk Swift. Z hľadiska UI frameworku sú na výber dve možnosti a to UIKit a SwiftUI. SwiftUI je oproti UIKit moderným spôsobom pre tvorbu UI a poskytuje API ako pre prácu so stavom aplikácie a navigáciou, tak aj deklaratívny spôsob tvorby jednotlivých UI komponent. SwiftUI je zároveň predvolenou možnosťou, ak chcú vývojári vyvíjať nové natívne aplikácie určené pre Apple zariadenia.

7.3.1 Štruktúrálne rozloženie

Projekt je prehľadne štruktúrovaný do zložiek uchovávajúca súbory na základe ich účelu:

- *Models:* Obsahuje Core ML modely používané pre detekciu
- *ViewModels:* Obsahuje *SettingsViewModel*, ktorý uchováva nastavenia aplikácie v jednom objekte
- *Modifiers:* Obsahuje znovu použiteľné funkcie pre štýlovanie UI
- *Types:* Obsahuje vlastné definície tried objektov používaných v aplikácii
- *Controllers:* Obsahuje triedy pre ovládanie kamery a využivanie core ml modelu
- *Assets.*xcassets: Obsahuje obrázky a ikonu aplikácie
- *ContentView.swift:* Vstupný bod aplikácie, kde je vytvárané globálne dátové úložisko, stav onboardingu a jednoduchá navigačná logika

7.3.2 UI komponenty a atomický dizajn

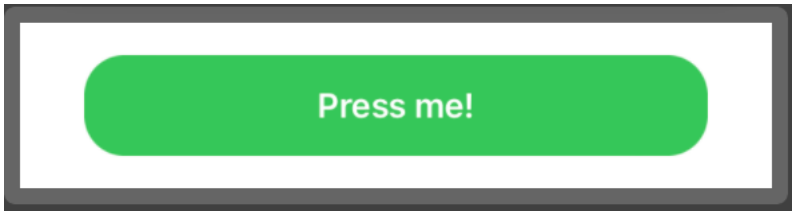
Pre umožnenie dobrej škálovateľnosti sú jednotlivé časti UI rozdelené pomocou atomického dizajnu, ktorý prvky UI radí do nasledujúcich tried:

- *Atoms:* Jedná sa o najmenšie a najzákladnejšie prvky UI ako tlačidlá, inputy, textové prvky a obrázky. Neuchovávajú žiadnu aplikačnú logiku
- *Molecules:* Podobne ako v reálnom svete, aj v atomickom dizajne sú molekuly zložené z jednotlivých atómov a teda kombinujú najzákladnejšie prvky UI, aby vytvorili väčší celok. Spravidla sa môže jednať o input s obrázkom alebo ikonou alebo kombinácia viacerých textových elementov, ktoré tvoria nejaký logický celok.

- *Organisms*: Tvoria väčšie, ucelené prvky aplikácie. Môžu byť zložené ako z atómov, tak aj z molekúl. Spravidla sa jedná napríklad o header a footer v aplikáciách, navigačné bary a iné.
- *Templates*: Reprezentujú podobu stránky. Majú na starosti usporiadanie atómov, molekúl a organizmov v rámci stránky. Neprechovávajú v sebe žiadnu logiku alebo aplikáčné dáta, tie získavajú zo stránok – screens.
- *Screens*: Predstavujú dátový model pre template. Neobsahujú prvky UI, ale predávajú výsledky výpočtov a funkcie ďalej do template.

7.3.2.1 Atómy

SwiftUI poskytuje bohatú ponuku atómov a ak sa chce vývojár držať striktne natívneho vzhľadu aplikácie, nie je do tak vysokej miery nutné ďalšie štýlovanie mimo poziciovania. V rámci aplikácie je teda použitý jediný atóm a to *RoundedButton.*
Jedná sa o naštýlované tlačidlo, ktoré je možné prepoužiť v rámci viacerých častí aplikácie. Je možné mu upraviť ako text, tak aj funkciu, ktorá sa spustí po jeho stlačení.



Obrázok 57 RoudedButton atóm

```
10 struct RoundedButton: View {
11     let label: String
12     let onPress: () -> Void
13 
14     var body: some View {
15         Button(action: onPress) {
16             HStack {
17                 Text(label)
18                     .font(.headline)
19             }
20             .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0, maxHeight: 50)
21             .background(Color.green)
22             .foregroundColor(.white)
23             .cornerRadius(20)
24             .padding(.horizontal)
25         }
26     }
27 }
```

Obrázok 58 Kód atómu RoundedButton

7.3.2.2 Molekuly

Molekuly sú v projekte použité najmä pri úvodnom onboardingu, zoskupujú hlavičku a popis výhod aplikácie. Hlavným reprezentantom je *OnboardingTextItem*:

Obrázok 59 Molekula OnboardingTextItem

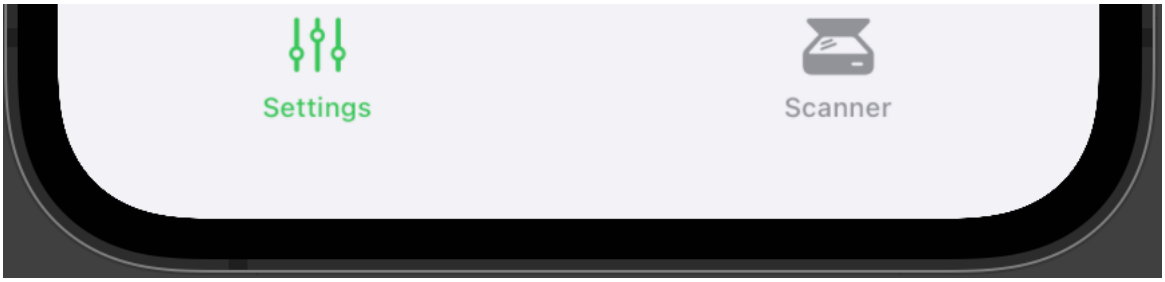
```
10 struct OnboardingTextItem: View {
11     let onboardingItem: OnboardingTextType
12 
13     var body: some View {
14         VStack {
15             Text(onboardingItem.title)
16                 .fontWeight(.bold)
17                 .multilineTextAlignment(.center)
18                 .padding(.vertical)
19                 .font(.title)
20                 .textCase(.uppercase)
21 
22             Text(onboardingItem.label)
23                 .multilineTextAlignment(.center)
24         }
25     }
26 }
```

Obrázok 60 Kód pre OnboardingTextItem

7.3.2.3 Organizmy

Organizmy sú zväčša používané ako väčšie celky aplikácie a spravidla sú i komplexnejšie. V rámci práce sú organizmy nasledovné:

- *Scanner:* Predstavuje modul pre real-time preview a detekciu
- *TabBar*: Spodný navigačný bar aplikácie



Obrázok 61 Navigačný bar

```
10  struct TabBar: View {
11      @ObservedObject var settings: SettingsViewModel
12
13      var body: some View {
14          TabView {
15              ScannerScreen(settings: settings)
16                  .tabItem {
17                      Label("Scanner", systemImage: "scanner")
18                  }
19
20              SettingsScreen(settings: settings)
21                  .tabItem {
22                      Label("Settings", systemImage: "slider.vertical.3")
23                  }
24          }
25          .accentColor(.green)
26      }
27  }
```

Obrázok 62 Kód pre navigačný bar

7.3.2.4 Templates

Templates ako ucelené časti aplikácie majú za úlohu rozmiestniť a poziciovať ostatné prvky UI. Aplikácia používa tieto templates:

- *OnboardingTemplate:* Úvodná slideshow
- *ScannerTemplate:* Slúži ako wrapper pre Scanner organizmus
- *SettingsTemplate:* Obsahuje všetky prvky UI nastavení

```
struct SettingsTemplate: View {
    @ObservedObject var settings: SettingsViewModel

    let learningMethods = ["Transfer learning", "Full network"]

    var body: some View {
        NavigationView {
            Form {
                Section(header: Text("Detect")) {
                    Toggle("Three-leaf clovers", isOn: $settings.detectThreeLeaf)
                    Toggle("Four-leaf clovers", isOn: $settings.detectFourLeaf)
                }
                Section(header: Text("Detection box colors")) {
                    ColorPicker("Three-leaf detector color", selection:
                        $settings.threeLeafBoundingBoxColor)
                    ColorPicker("Four-leaf detector color", selection:
                        $settings.fourLeafBoundingBoxColor)
                }
                Section(header: Text("Detection method")) {
                    Picker(settings.model, selection: $settings.model) {
                        ForEach(learningMethods, id: \.self) { method in
                            Text(method)
                        }
                    }
                }
            }
            .navigationTitle("Settings")
        }
    }
}
```

Obrázok 63 Reprezentácia template

7.3.2.5 Screens

Dátové kontajnery pre jednotlivé templates, riadia dátový flow aplikácie a pracujú s modelom. Nachádzajú sa tu funkcie pre spravovanie interakcie užívateľa s UI. Každá screen si taktiež preberá globálny dátový model v podobe settings, s ktorým ďalej pracuje. Analogicky k templates teda existujú:

- *OnboardingScreen*: Samotná screen sa zobrazí iba raz a obsahuje teda logiku pre mainpuláciu s dokončením zobrazenia slideshow a samotné onboarding texty, ktoré má template zobraziť
- *ScannerScreen*: Používaná v rámci navigácie, preberá globálny dátový model a predáva ho ďalej v hierarchii scanneru
- *SettingsTemplateScreen*: Screen určená pre priamu manipuláciu s globálnym dátovým modelom, ktorého hodnoty predáva ďalej do template a jednotlivých prvkov UI

```
struct OnboardingScreen: View {
    @Binding var hasFinishedOnboarding: Bool

    @State private var onboardingIndex = 0

    let onboardingTexts: [OnboardingTextType] = [
        OnboardingTextType(title: "Welcome to Unclover", label: "Unclover allows you to use your mobile device to find four
            leaf clovers in grass patches"),
        OnboardingTextType(title: "On device Machine Learning", label: "Leverage the power of your iPhone with on-device
            machine learning"),
        OnboardingTextType(title: "Offline Usage", label: "Use Unclover anytime and anywhere you are without the need to think
            about your internet connection"),
        OnboardingTextType(title: "Customize your experience", label: "Don't like the look of your scanner? Change the colors.
            Do you also want to detect three-leaf clovers? The capability is just a tap away in your settings."),
    ]

    func handleSkip() {
        hasFinishedOnboarding = true
    }

    var isLastPage: Bool {
        return onboardingIndex + 1 == onboardingTexts.count
    }

    func handleNextPress() {
        if isLastPage {
            handleSkip()
        } else {
            onboardingIndex += 1
        }
    }

    var body: some View {
        OnboardingTemplate(
            onboardingIndex: $onboardingIndex,
            isLastPage: isLastPage,
            onboardingTexts: onboardingTexts,
            onNextPress: handleNextPress,
            onSkip: handleSkip
        )
    }
}
```

Obrázok 64 Ukážka screen komponenty

7.3.3 Dátová vrstva aplikácie

Dátová vrstva sa delí na dve časti:

- *Detekčný model:* Aplikácia v sebe prechováva pre demonštračné účely ako TN model, tak i FN model
- *SettingsViewModel:* Obsahuje dátový model pre nastavenia, ktorý zároveň slúži ako globálne úložisko dát, ktoré je teda možné použiť v každej časti aplikácie
- *hasFinishedOnboarding:* Jednoduchá boolean hodnota využívajúca úložisko priamo v zariadení, ktorej úlohou je zobrazovanie onboarding obrazoviek

Primárnym zdrojom dát, ktoré ovplyvňujú chovanie aplikácie sú však nastavenia, ktoré si užívateľ môže ľubovoľne upraviť. Nastaviteľné hodnoty sú:

- *detectThreeLeaf*: Boolean, ktorého úlohou je umožňovať alebo zakazovať detekciu trojlístkov
- *detectFourLeaf:* Boolean, ktorého úlohou je umožňovať alebo zakazovať detekciu štvorlístkov

- *model:* String, určujúci výber modelu pre detekciu ďateliny
- *threeLeafBoundingBoxColor:* Color, umožňuje úpravu farby bounding boxov pre trojlístky
- *fourLeafBoundingBoxColor:* Color, umožňuje úpravu farbu bounding boxov pre štvorlístky

Po inicializácii modelu sa dáta ukladajú do *User Defaults*, ktoré prechovávajú stav aplikácie aj v prípade, kedy ju užívateľ zavrie. Dáta sú teda uložené priamo na zariadení a sú prechovávané pri každom spustení aplikácie. Pre predávanie tohto dátového modelu sa používa metóda *ObservableObject.* Model je inicializovaný v *ContentView.swift* a teda vstupnom bode aplikácie a je ďalej predávaný do jednotlivých screens, ktoré s modelom pracujú. Týmto spôsobom je teda zabezpečená globálna povaha úložiska, spolu s offline funkcionalitou a uložením preferencii užívateľa.

```
12 class SettingsViewModel: ObservableObject {
13     @Published var detectThreeLeaf: Bool {
14         didSet {
15             UserDefaults.standard.set(detectThreeLeaf, forKey: "detectThreeLeaf")
16         }
17     }
18     @Published var detectFourLeaf: Bool {
19         didSet {
20             UserDefaults.standard.set(detectFourLeaf, forKey: "detectFourLeaf")
21         }
22     }
23     @Published var model: String {
24         didSet {
25             UserDefaults.standard.set(model, forKey: "model")
26         }
27     }
28     @Published var threeLeafBoundingBoxColor: Color {
29         didSet {
30             UserDefaults.standard.set(threeLeafBoundingBoxColor, forKey: "threeLeafBoundingBoxColor")
31         }
32     }
33     @Published var fourLeafBoundingBoxColor = Color(.sRGB, red: 0, green: 0, blue: 1, opacity: 1)
34 
35     init() {
36         self.detectThreeLeaf = UserDefaults.standard.object(forKey: "detectThreeLeaf") as? Bool ?? false
37         self.detectFourLeaf = UserDefaults.standard.object(forKey: "detectFourLeaf") as? Bool ?? true
38         self.model = UserDefaults.standard.object(forKey: "model") as? String ?? "Transfer learning"
39         self.threeLeafBoundingBoxColor = UserDefaults.standard.object(forKey: "threeLeafBoundingBoxColor")
               as? Color ?? Color.red
40     }
41 }
```

Obrázok 65 Dátový model s uložením do UserDefaults

Poslednou časťou je *hasFinishedOnboarding*, ktorá je taktiež incializovaná v *ContentView.swift,* avšak pomocou špeciálneho dekorátora *@AppStorage*, ktorý slúži ako abstrakcia nad User Defaults bez nutnosti pridávania ďalšej aplikačnej logiky pre ukladanie dát. Tento prístup je vhodný, keďže sa jedná o jednoduchú premennú, využívanú pre jeden špecifický účel.

```
10  struct ContentView: View {
11      @StateObject var settings = SettingsViewModel()
12      @AppStorage("launchCount") var hasFinishedOnboarding = false
13
14
15      var body: some View {
16          if hasFinishedOnboarding {
17              OnboardingScreen(hasFinishedOnboarding: $hasFinishedOnboarding)
18          } else {
19              TabBar(settings: settings)
20          }
21      }
22  }
```

Obrázok 66 Inicializácia dátového modelu a navigácie

7.3.4 Logická vrstva aplikácie

Logická vrstva aplikácie spracováva model predávaný do jednotlivých screen do aplikácie. Okrem klasickej komunikačnej vrsty Screen => Template je nutné implementovať aj komponenty využívajúce UIKit. Konkrétne sa jedná o organizmus *Scanner,* ktorého úlohou je nielen zaistiť real-time camera feed, ale i tento obraz spracovať a následne vykresliť bounding boxy pomocou frameworku *Vision.*

Keďže SwiftUI neobsahuje kompletný zoznam komponent tak, ako UIKit, je nutné niektoré komponenty previesť do SwiftUI. To je možné docieliť vďaka metódam *makeUIViewController* a *updateUIViewController*, ktoré slúžia ako akýsi bridge medzi SwiftUI a UIKitom. Princípom je vytvorenie controlleru, ktorý sa následne exposuje frameworku SwiftUI. Zároveň je to spôsob, ako predávať dáta medzi týmito dvoma frameworkami.

```swift
struct Scanner: UIViewControllerRepresentable {
    let detectFourLeaf: Bool
    let detectThreeLeaf: Bool
    let threeLeafBoundingBoxColor: Color
    let fourLeafBoundingBoxColor: Color
    let model: String

    func makeUIViewController(context: UIViewControllerRepresentableContext<Scanner>) ->
        VisionObjectRecognitionController {
        let controller = VisionObjectRecognitionController(
            detectFourLeaf: detectFourLeaf,
            detectThreeeLeaf: detectThreeLeaf,
            threeLeafBoundingBoxColor: threeLeafBoundingBoxColor,
            fourLeafBoundingBoxColor: fourLeafBoundingBoxColor,
            model: model
        )

        controller.detectThreeLeaf = detectThreeLeaf
        controller.detectFourLeaf = detectFourLeaf
        controller.threeLeafBoundingBoxColor = threeLeafBoundingBoxColor
        controller.fourLeafBoundingBoxColor = fourLeafBoundingBoxColor

        return controller
    }

    func updateUIViewController(_ uiViewController: VisionObjectRecognitionController, context:
        UIViewControllerRepresentableContext<Scanner>) {
        uiViewController.detectThreeLeaf = detectThreeLeaf
        uiViewController.detectFourLeaf = detectFourLeaf
        uiViewController.threeLeafBoundingBoxColor = threeLeafBoundingBoxColor
        uiViewController.fourLeafBoundingBoxColor = fourLeafBoundingBoxColor
        uiViewController.model = model
    }
}

```

Obrázok 67 Komponenta Scanner, ktorej sa predávajú dáta zo SwiftUI vrstvy do UIKit controlleru

Logika pre detekciu je potom obsiahnutá v samostatných controlleroch a to *VisionRecognitionObjectcontroller* a *LivePreviewController.*

LivePreviewController slúži ako sprostredkovávateľ real-time camera feedu pomocou zabudovanej knižnice *AVFoundation*.

Tento controllwe je následne použitý v rámci *VisionRecognitionControlleru,* ktorý pri jeho inicializácii preberá dáta zo SwiftUI vrstvy, na základe ktorých sa tvoria nielen custom bounding boxy, ale volí sa i model pre detekciu. Popis jednotlivých funkcii používaných pre získavanie obrazu a detekcii je popísaný v ukážkach kódu nižšie.

```swift
class VisionObjectRecognitionController: LivePreviewController {
    var model: String
    var detectThreeLeaf: Bool
    var detectFourLeaf: Bool
    var threeLeafBoundingBoxColor: Color
    var fourLeafBoundingBoxColor: Color

    private var detectionOverlay: CALayer! = nil

    // Vision parts
    private var requests = [VNRequest]()

    init(
        detectFourLeaf: Bool,
        detectThreeeLeaf: Bool,
        threeLeafBoundingBoxColor: Color,
        fourLeafBoundingBoxColor: Color,
        model: String
    ) {
        self.detectFourLeaf = detectFourLeaf
        self.detectThreeLeaf = detectThreeeLeaf
        self.threeLeafBoundingBoxColor = threeLeafBoundingBoxColor
        self.fourLeafBoundingBoxColor = fourLeafBoundingBoxColor
        self.model = model

        super.init(nibName: nil, bundle: nil)
    }


    required init?(coder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
```

Obrázok 68 VisionObjectRecognitionController - ukážka dát, ktoré prijíma zo SwiftUI vrstvy a jeho inicializácia

```
@discardableResult
func setupVision() -> NSError? {
    // Setup Vision parts
    let error: NSError! = nil
    let detectionModel = model == "Transfer Learning" ? "UncloverDetector"
        : "FNUncloverDetector"


    guard let modelURL = Bundle.main.url(forResource: detectionModel,
        withExtension: "mlmodelc") else {
        return NSError(domain: "VisionObjectRecognitionViewController",
            code: -1, userInfo: [NSLocalizedDescriptionKey: "Model file is
            missing"])
    }
    do {
        let visionModel = try VNCoreMLModel(for: MLModel(contentsOf:
            modelURL))
        let objectRecognition = VNCoreMLRequest(model: visionModel,
            completionHandler: { (request, error) in
            DispatchQueue.main.async(execute: {
                // perform all the UI updates on the main queue
                if let results = request.results {
                    self.drawVisionRequestResults(results)
                }
            })
        })
        self.requests = [objectRecognition]
    } catch let error as NSError {
        print("Model loading went wrong: \(error)")
    }

    return error
}
```

Obrázok 69 VisionObjectRecognitionController - Voľba ml modelu pre detekciu

```
74      func drawVisionRequestResults(_ results: [Any]) {
75          CATransaction.begin()
76          CATransaction.setValue(kCFBooleanTrue, forKey:
                kCATransactionDisableActions)
77          detectionOverlay.sublayers = nil // remove all the old recognized
                objects
78
79          for observation in results where observation is
                VNRecognizedObjectObservation {
80              guard let objectObservation = observation as?
                    VNRecognizedObjectObservation else {
81                  continue
82              }
83
84
85              // Select only the label with the highest confidence.
86              let topLabelObservation = objectObservation.labels[0]
87
88              if (topLabelObservation.identifier == "three-leaf" &&
                    detectThreeLeaf) || (topLabelObservation.identifier ==
                    "four-leaf" && detectFourLeaf) {
89                  let objectBounds =
                        VNImageRectForNormalizedRect(objectObservation.boundingBox,
                        Int(bufferSize.width), Int(bufferSize.height))
90
91                  let shapeLayer =
                        self.createRoundedRectLayerWithBounds(objectBounds, label:
                        topLabelObservation.identifier)
92
93                  detectionOverlay.addSublayer(shapeLayer)
94              }
95          }
96
97          self.updateLayerGeometry()
98          CATransaction.commit()
99      }
```

Obrázok 70 VisionObjectRecognitionController - Vykresľovanie výsledkov

```
101     override func captureOutput(_ output: AVCaptureOutput, didOutput
            sampleBuffer: CMSampleBuffer, from connection: AVCaptureConnection) {
102         guard let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) else
                {
103             return
104         }
105
106         let exifOrientation = exifOrientationFromDeviceOrientation()
107
108         let imageRequestHandler = VNImageRequestHandler(cvPixelBuffer:
                pixelBuffer, orientation: exifOrientation, options: [:])
109         do {
110             try imageRequestHandler.perform(self.requests)
111         } catch {
112             print(error)
113         }
114     }
```

Obrázok 71 VisionObjectRecognitionController - Pomocné funkcie pre spravovanie realtime videa

```
116     override func setupAVCapture() {
117         super.setupAVCapture()
118
119         // setup Vision parts
120         setupLayers()
121         updateLayerGeometry()
122         setupVision()
123
124         // start the capture
125         startCaptureSession()
126     }
```

Obrázok 72 VisionObjectRecognitionController - Inicializácia funkcii pre vision a video

```
    func setupLayers() {
        detectionOverlay = CALayer() // container layer that has all the
            renderings of the observations
        detectionOverlay.name = "DetectionOverlay"
        detectionOverlay.bounds = CGRect(x: 0.0,
                                         y: 0.0,
                                         width: bufferSize.width,
                                         height: bufferSize.height)
        detectionOverlay.position = CGPoint(x: rootLayer.bounds.midX, y:
            rootLayer.bounds.midY)
        rootLayer.addSublayer(detectionOverlay)
    }

    func updateLayerGeometry() {
        let bounds = rootLayer.bounds
        var scale: CGFloat

        let xScale: CGFloat = bounds.size.width / bufferSize.height
        let yScale: CGFloat = bounds.size.height / bufferSize.width

        scale = fmax(xScale, yScale)
        if scale.isInfinite {
            scale = 1.0
        }
        CATransaction.begin()
        CATransaction.setValue(kCFBooleanTrue, forKey:
            kCATransactionDisableActions)

        // rotate the layer into screen orientation and scale and mirror
        detectionOverlay.setAffineTransform(CGAffineTransform(rotationAngle:
            CGFloat(.pi / 2.0)).scaledBy(x: scale, y: -scale))
        // center the layer
        detectionOverlay.position = CGPoint(x: bounds.midX, y: bounds.midY)

        CATransaction.commit()

    }

    func createRoundedRectLayerWithBounds(_ bounds: CGRect, label: String) ->
        CALayer {
        let borderColor = label == "three-leaf" ? threeLeafBoundingBoxColor :
            fourLeafBoundingBoxColor

        let shapeLayer = CALayer()
        shapeLayer.bounds = bounds
        shapeLayer.position = CGPoint(x: bounds.midX, y: bounds.midY)
        shapeLayer.name = "Found Object"
        shapeLayer.cornerRadius = 5
        shapeLayer.backgroundColor = borderColor.opacity(0.1).cgColor
        shapeLayer.borderWidth = 5
        shapeLayer.borderColor = borderColor.cgColor
        return shapeLayer
    }

}
```

Obrázok 73 VisionObjectRecognitionController - Vykresľovanie bounding boxov a ich finálnej podoby

7.3.5 Verzovanie a deployment

Aplikácia využíva verzovací systém Git, ktorý je štandardným systémom na poli VCS. Ako službu pre uloženie samotného repozitára bol zvolený GitHub, umožňujúci vytváranie separátnych branchí a zároveň poskytuje jednoduchý a zrozumiteľný prehľad všetkých verzii aplikácie.

Testovacie a produkčné prostredie je vytvorené na platforme *App Store Connect*, taktiež Apple nástroj určený pre distribúciu, monetizáciu a prezeranie základných analytík aplikácie. Predtým, ako je možné aplikáciu vydať je nutné prejsť review od Apple trvajúcou 1-2 dní. Aplikácia je v čase písania práce dostupná striktne v testovacom prostredí TestFlight, ktorého výhodou je okamžitá distribúcia testovacích buildov. Celkovo si aplikácia prešla od jej počiatku šiestimi hlavnými verziami.

Aplikácia neobsahuje žiadne CI/CD prvky medzi GitHubom a App Store Connect. Jednotlivé buildy sú vzhľadom na veľkosť vývojového teamu a rozsahu funkcionalít nahrávané manuálne.

ZÁVER

V rámci mojej práce bola vyvinutá aplikácie pre detekciu štvorlístkov a trojlístkov. Práca je rozdelená do dvoch celkov. Prvý celok sa zaoberá oblasťou computer vision v teoretickej časti a následným trénovaním vhodného modelu v praktickej časti. Druhým celkom je rozbor možností dnešného mobilného vývoja a následná implementácia mobilnej aplikácie.

Prvým krokom po literárnej rešerši bola anotácia datasetu, ktorý mi bol poskytnutý vedúcim mojej práce Ing. Petrom Žáčkom, Ph.D. Pomocou IBM Cloud Annotations som vytvoril plne anotovaný dataset, ktorý je následne exportovaný pre trénovanie zvoleného modelu. Keďže sa aplikácia zameriava na Apple machine learning ekosystém, anotovaný dataset bol exportovaný pre formát Core ML.

Samotné trénovanie modelu prebiehalo pomocou aplikácie Create ML, ktorá je schopná na základe trénovacích dát vytvoriť model priamo vhodný pre implementáciu modelu, s ktorým je schopný framework Core ML následne pracovať v mobilnej aplikácii. Trénovacie parametre ako počet iterácii a batch size je aplikácia schopná nastaviť automaticky práve na základe trénovacích dát a odporúča tak predvolené nastavenia pre tréning. Na výber pre Object Detection modely sú dva spôsoby tréningu, ktoré Apple označuje ako „Full Network“ a „Transfer Learning“. Full network model využíva klasickú konvolučnú sieť založenú na rchitektúre YOLOv2 a z výsledkov testov vyplýva výrazne nižšia presnosť oproti Transfer Learning metóde. To je taktiež zapríčinené nízkym počtom trénovacích dát – zhruba 100, avšak presne pre takéto prípady je metóda Transfer Learning vhodná. Použitím predtrénovaného modelu a feature printu dokáže model spoľahlivo detekovať požadované objekty, je tu však zvýšený výskyt false positive prípadov práve kvôli princípu feature printu, kde sa model pozerá na typické črty požadovaného objektu, ako napr. tvar. To však nebráni funkcionalite, keďže výskyt ďateliny je v tráve a toto prostredie je pri reálnom použití nemenné a TL model tak ostáva pre detekciu vhodnejším modelom nielen vďaka schopnosti istejšej detekcie, ale i veľkosti a času potrebného k jeho vytrénovaniu.

Dva vytrénované modely je následne možné implementovať do mobilnej aplikácie. Z technologického hľadiska sa javí ako najvhodnejší prístup k vývoju ten natívny jednak z hľadiska výkonnosti aplikácie a taktiež z hľadiska podpory Core ML modelov, kde nie je potrebné integrovať dodatočné balíčky alebo knižnice, ako by to bolo v prípade multiplatformového vývoja. Natívny vývoj pre iOS poskytuje dva frameworky pre tvorbu aplikácii a to UIKit a SwiftUI. Pre vývoj som zvolil SwiftUI, keďže sa jedná o moderný

a deklaratívny prístup k tvorbe UI a samotný Apple odporúča pre nové aplikácie túto cestu, vďaka ktorej som bol schopný vyvinúť požadovanú aplikáciu vo výrazne kratšom čase, ako by to bolo v prípade UIKitu. Keďže sa však jedná o pomerne mladý framework, niektoré časti museli byť implementované práve v UIKite, ako napríklad skener, kde vznikol technologický problém v podobe predávania dát zo SwiftUI vrstvy do imperatívneho UIKit kódu.

Výsledná aplikácia je vysoko modulárna, užívateľ si môže upraviť takmer každý aspekt od farieb, cez možnosti detekcie až po model, ktorý bude pre samotnú detekciu použitý. Vznikol tak softvér, ktorý je univerzálne použiteľný pre akýkoľvek typ object detection úlohy, kde stačí zameniť model vo formáte Core ML a pár riadkov kódu.

Aj táto diplomová práca je teda dôkazom, že machine learning sa postupne stáva viac a viac dostupným aj mimo sféru data science a i bežným aplikačným vývojárom dokáže pomôcť vyvinúť dielo, ktoré je schopné či už ozvláštniť, alebo uľahčiť život používateľovi, čo je koniec koncov primárnym účelom nás, informatikov.

ZOZNAM POUŽITEJ LITERATÚRY

1. What is computer vision?: Use machine learning and neural networks to teach computers to see defects and issues before they affect operations. *IBM* [online]. [cit. 2022-05-16]. Dostupné z: https://www.ibm.com/topics/computer-vision
2. OPPERMANN, Artem. What is Deep Learning and How does it work?: Learn the most important Basics of Deep Learning and Neural Networks in this detailed Tutorial. *Towards Data Science*[online]. 2019, 12.11.2019 [cit. 2022-05-16]. Dostupné z: https://towardsdatascience.com/what-is-deep-learning-and-how-does-it-work-2ce44bb692ac
3. SAHA, Sumit. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. *Towards Data Science* [online]. 2018, 15.12.2018 [cit. 2022-05-16]. Dostupné z: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
4. MEEL, Vidushi. 87 Most Popular Computer Vision Applications in 2022. *Towards Data Science*[online]. 2018, 15.12.2018 [cit. 2022-05-16]. Dostupné z: https://viso.ai/applications/computer-vision-applications/
5. Computer Vision: Everything You Need to Know: From self driving cars, through defect detection to medical imaging — here's how computer vision is helping modern businesses to solve complex visual tasks. *V7 Labs* [online]. 2018, 16.5.2022 [cit. 2022-05-16]. Dostupné z: https://www.v7labs.com/blog/what-is-computer-vision#how-computer-vision-works
6. WANED, Arnaud. 10 interesting computer vision tasks. *Towards Dev* [online]. 8.12.2021 [cit. 2022-05-16]. Dostupné z: https://towardsdev.com/10-interesting-computer-vision-tasks-3e5437a14668
7. DICKSON, Ben. 10 interesting computer vision task *Towards Dev* [online]. 2021, 21.6.2021 [cit. 2022-05-16]. Dostupné z: https://towardsdev.com/10-interesting-computer-vision-tasks-3e5437a14668
8. Introduction to object detection with deep learning. *Towards Dev* [online]. 2021, 19.10.2021 [cit. 2022-05-16]. Dostupné z: https://blog.superannotate.com/object-detection-with-deep-learning/

9. *Object Detection Guide: Almost everything you need to know about how object detection works*[online]. [cit. 2022-05-16]. Dostupné z: https://www.fritz.ai/object-detection/

10. DICKSON, Ben. An introduction to object detection with deep learning. *Fritz* [online]. 21.6.2021 [cit. 2022-05-16]. Dostupné z: https://bdtechtalks.com/2021/06/21/object-detection-deep-learning/

11. POKHREL, Sabina. Image Data Labelling and Annotation — Everything you need to know: Learn about different types of annotations, annotation formats and annotation tools. *Towards Data Science* [online]. 11.3.2020 [cit. 2022-05-16]. Dostupné z: https://towardsdatascience.com/image-data-labelling-and-annotation-everything-you-need-to-know-86ede6c684b1

12. Image Annotation for Computer Vision: A Guide to Labeling Visual Data for Your Machine Learning Project. *Cloud Factory* [online]. [cit. 2022-05-16]. Dostupné z: https://www.cloudfactory.com/image-annotation-guide

13. 13 Best Image Annotation Tools of 2022 [Reviewed]: Discover the 13 most popular image annotation tools of 2022. Compare their features and pricing, and choose the best data annotation tool for your needs. *Cloud Factory* [online]. 17.5.2022 [cit. 2022-05-16]. Dostupné z: https://www.v7labs.com/blog/best-image-annotation-tools

14. PRATAMA, Andhika. 5 Best Free Image Annotation Tools. *V7 Labs* [online]. 12.1.2021 [cit. 2022-05-16]. Dostupné z: https://medium.com/data-folks-indonesia/5-best-free-image-annotation-tools-80919a4e49a8

15. 10 of the best open-source annotation tools for computer vision 2021. *Humans in the loop*[online]. [cit. 2022-05-16]. Dostupné z: https://humansintheloop.org/10-of-the-best-open-source-annotation-tools-for-computer-vision-2021/

16. PEREIRA, Tiago a Moritz RECKE. Creating annotated data sets with IBM Cloud Annotations. *Create With Swift* [online]. 2021, 16.6.2021 [cit. 2022-05-16]. Dostupné z: https://www.createwithswift.com/creating-annotated-data-sets-with-ibm-cloud-annotations/

17. GANDHI, Rohith. R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms: Understanding object detection algorithms. *Towards Data*

Science [online]. 9.7.2019 [cit. 2022-05-16]. Dostupné z: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

18. GAD, Ahmed. Faster R-CNN Explained for Object Detection Tasks: Understanding object detection algorithms. *Towards Data Science* [online]. 2020 [cit. 2022-05-16]. Dostupné z: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e
19. VERMA, Yugesh. R-CNN vs Fast R-CNN vs Faster R-CNN – A Comparative Guide: R-CNNs (Region-based Convolutional Neural Networks) a family of machine learning models Specially designed for object detection, the original goal of any R-CNN is to detect objects in any input image. *Towards Data Science* [online]. 2020 [cit. 2022-05-16]. Dostupné z: https://analyticsindiamag.com/r-cnn-vs-fast-r-cnn-vs-faster-r-cnn-a-comparative-guide/
20. Faster R-CNN: Using Region Proposal Network for Object Detection. *Analytics India Mag* [online]. 2020 [cit. 2022-05-16]. Dostupné z: https://www.alegion.com/faster-r-cnn
21. AGGARWAL, Ani. YOLO Explained. *Alegion* [online]. 2020, 27.12.2020 [cit. 2022-05-16]. Dostupné z: https://medium.com/analytics-vidhya/yolo-explained-5b6f4564f31
22. ŚWIEŻEWSKI, Jędrzej. YOLO Algorithm and YOLO Object Detection. *Medium* [online]. 2020, 22.5.2020 [cit. 2022-05-16]. Dostupné z: https://appsilon.com/object-detection-yolo-algorithm/
23. BANDYOPADHYAY, Hmrishav. YOLO: Real-Time Object Detection Explained: Learn all about the YOLO algorithm for object detection and start training your own models using personalized datasets. *Appsilon* [online]. 2020, 17.5.2022 [cit. 2022-05-16]. Dostupné z: https://www.v7labs.com/blog/yolo-object-detection#two-stage-detectors
24. SINGH, Aditya. YOLOP: Single Shot Panoptic Driving Perception. *V7 Labs* [online]. 2020, 14.9.2021 [cit. 2022-05-16]. Dostupné z: https://medium.com/augmented-startups/yolop-single-shot-panoptic-driving-perception-47df3e914cf8

25. MEEL, Vidushi. YOLOR – You Only Learn One Representation (What's new, 2022). *Medium*[online]. 2022 [cit. 2022-05-16]. Dostupné z: https://medium.com/augmented-startups/yolop-single-shot-panoptic-driving-perception-47df3e914cf8

26. FANG, Yuixin. You Only Look at One Sequence: Rethinking Transformer in Vision through Object Detection. *Medium* [online]. 2022 [cit. 2022-05-16]. Dostupné z: https://www.arxiv-vanity.com/papers/2106.00666/

27. HUI, Jonathan. SSD object detection: Single Shot MultiBox Detector for real-time processing. *Medium* [online]. 2022 [cit. 2022-05-16]. Dostupné z: https://jonathan-hui.medium.com/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06

28. DAVID, Matthew. Mobile application development. *TechTarget* [online]. [cit. 2022-05-16]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/mobile-application-development

29. PEAK, Sean. What Is Mobile App Development?. *TechTarget* [online]. 13.12.2021 [cit. 2022-05-16]. Dostupné z: https://www.businessnewsdaily.com/5155-mobile-app-development.html

30. Mobile App Development - Step by Step Guide for 2021. *Business News Daily* [online]. [cit. 2022-05-16]. Dostupné z: https://www.openxcell.com/mobile-app-development/

31. PATEL, Vatsal. An Ultimate Guide for Mobile App Development: Everything you should know before getting your hands dirty with mobile app development. *Medium* [online]. [cit. 2022-05-16]. Dostupné z: https://medium.com/nerd-for-tech/an-ultimate-guide-for-mobile-app-development-ff3c17ece87a

32. 5 Key Benefits of Native Mobile App Development. *Medium* [online]. [cit. 2022-05-16]. Dostupné z: https://medium.com/nerd-for-tech/an-ultimate-guide-for-mobile-app-development-ff3c17ece87a

33. What is a Native Mobile App Development?. *Medium* [online]. 12.9.2021 [cit. 2022-05-16]. Dostupné z: https://mdevelopers.com/blog/what-is-a-native-mobile-app-development-

34. Native Apps, Web Apps or Hybrid Apps? What's the Difference?. *Mobiloud* [online]. [cit. 2022-05-16]. Dostupné z: https://www.mobiloud.com/blog/native-web-or-hybrid-apps#5

35. SINGH, Satinder. Native vs Hybrid vs Cross Platform – What to Choose in 2022?. *Netsolutions*[online]. 19.11.2021 [cit. 2022-05-16]. Dostupné z: https://www.netsolutions.com/insights/native-vs-hybrid-vs-cross-platform/

36. ROOMI, Mishal. 5 Advantages and Disadvantages of Native App | Drawbacks & Benefits of Native App. *Hitech Whizz* [online]. 19.4.2021 [cit. 2022-05-16]. Dostupné z: https://www.netsolutions.com/insights/native-vs-hybrid-vs-cross-platform/

37. Cross-Platform Development. *Hitech Whizz* [online]. [cit. 2022-05-16]. Dostupné z: https://www.techopedia.com/definition/30026/cross-platform-development

38. MACHANTA, Amit. The Ultimate Guide to Cross Platform App Development Frameworks in 2022. *Techopedia* [online]. [cit. 2022-05-16]. Dostupné z: https://www.netsolutions.com/insights/cross-platform-app-frameworks-in-2019/

39. Pros and cons of cross platform mobile app development. *Rishabsoft* [online]. [cit. 2022-05-16]. Dostupné z: https://www.rishabhsoft.com/blog/pros-cons-cross-platform-mobile-app-development

40. The Pros and Cons of Cross-Platform Mobile App Development – [2021 Updated]. *Rishabsoft*[online]. 7.1.2021 [cit. 2022-05-16]. Dostupné z: https://www.appschopper.com/blog/pros-cons-cross-platform-mobile-app-development/

41. SHIBAINA, S. Https://www.focaloid.com/the-pros-and-cons-of-cross-platform-apps/. *Focaloid Technologies* [online]. 28.9.2021 [cit. 2022-05-16]. Dostupné z: The Pros and Cons of Cross-Platform Apps

42. What is a PWA — An Intro to Progressive Web Apps. *Focaloid Technologies* [online]. 30.9.2019 [cit. 2022-05-16]. Dostupné z:

https://medium.com/@blockchain_simplified/what-is-a-pwa-an-intro-to-progressive-web-apps-3f280071f909

43. RICHARD, Sam a Pete LEPAGE. What are Progressive Web Apps?. *Medium* [online]. 6.1.2020 [cit. 2022-05-16]. Dostupné z: https://web.dev/what-are-pwas/
44. 5 Key Mobile Development Approaches. *Velvetech* [online]. [cit. 2022-05-16]. Dostupné z: https://www.velvetech.com/blog/5-key-mobile-development-approaches/
45. The Mobile App Comparison Chart: Hybrid vs. Native vs. Mobile Web (2019 Update). *Velvetech*[online]. 14.10.2019 [cit. 2022-05-16].
46. IOS 15. *Velvetech* [online]. [cit. 2022-05-16]. Dostupné z: https://www.apple.com/ios/ios-15/
47. Getting Started with iOS App Development. *Amazon AWS* [online]. [cit. 2022-05-16]. Dostupné z: https://aws.amazon.com/mobile/mobile-application-development/native/ios/
48. MITRA, Mikhail. 10 Reasons To Learn Swift Programming Language. *Antralabs Global* [online]. 25.2.2016 [cit. 2022-05-16]. Dostupné z: https://www.mantralabsglobal.com/blog/10-reasons-to-get-started-with-swift-programming-language/
49. SCHAFFER, Erin. What is Swift? Features, advantages, and syntax basics. *Antralabs Global* [online]. 27.10.2021 [cit. 2022-05-16]. Dostupné z: https://www.educative.io/blog/swift-programming
50. JACOBS, Bart. SwiftUI Fundamentals: What Is SwiftUI. *Educative IO* [online]. [cit. 2022-05-16]. Dostupné z: https://cocoacasts.com/swiftui-fundamentals-what-is-swiftui
51. SHARMA, Manish. SwiftUI Fundamentals: What is SwiftUI. *Cocoacasts* [online]. 3.3.2020 [cit. 2022-05-16]. Dostupné z: https://medium.com/@mansha99/swift-ui-fundamentals-634df221295e
52. STEINBERGER, Peter. SwiftUI Fundamentals: The new shiny. *Medium* [online]. August 2021 [cit. 2022-05-16]. Dostupné z: https://increment.com/mobile/the-shift-to-declarative-ui/

53. SUNDELL, John. Swift by Sundell. *Increment* [online]. 5.7.2020 [cit. 2022-05-16]. Dostupné z: https://www.swiftbysundell.com/articles/swiftui-state-management-guide/

54. HUDSON, Paul. Swift by Sundell. *Swift By Sundell* [online]. 21.5.2022 [cit. 2022-05-16]. Dostupné z: https://www.hackingwithswift.com/quick-start/swiftui/answering-the-big-question-should-you-learn-swiftui-uikit-or-both

55. How to use UIKit in SwiftUI. *Hacking With Swift* [online]. 30.6.2019 [cit. 2022-05-16]. Dostupné z: https://sarunw.com/posts/uikit-in-swiftui/

56. SRIVASTAVA, Sudeep. Top 10 Best Cross-Platform App Development Frameworks. *Sarunw*[online]. 25.4.2022 [cit. 2022-05-16]. Dostupné z: https://appinventiv.com/blog/cross-platform-app-frameworks/

57. What Is the Flutter Framework?. *Appinventiv* [online]. 18.5.2021 [cit. 2022-05-16]. Dostupné z: https://appinventiv.com/blog/cross-platform-app-frameworks/

58. SRIVASTAVA, Naveen. Under The Hood Rendering In Flutter. *Appinventiv* [online]. 19.6.2021 [cit. 2022-05-16]. Dostupné z: https://medium.flutterdevs.com/under-the-hood-rendering-in-flutter-ddc5aadd65ba

59. Introduction to widgets. *Medium* [online]. [cit. 2022-05-16]. Dostupné z: https://docs.flutter.dev/development/ui/widgets-intro

60. BELADIYA, Kiran. What is Flutter App Development? Advantages & Drawbacks of Flutter. *The One Technologies* [online]. [cit. 2022-05-16]. Dostupné z: https://theonetechnologies.com/blog/post/flutter-mobile-application-development

61. What is React Native. *O'Reilly* [online]. [cit. 2022-05-16]. Dostupné z: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html

62. React Native: Learn once, write anywhere. *React Native Dev* [online]. [cit. 2022-05-16]. Dostupné z: https://reactnative.dev

63. Introducing JSX. *React Native Dev* [online]. [cit. 2022-05-16]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html

64. Hooks at a Glance. *React Native Dev* [online]. [cit. 2022-05-16]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html

65. Why You Should (Or Shouldn't) Use React Native. *React Native Dev* [online]. 11.9.2018 [cit. 2022-05-16]. Dostupné z: https://reactjs.org/docs/hooks-overview.html

66. ARNER, Nick. Machine Learning Development on Apple Platforms. *React Native Dev* [online]. 15.10.2019 [cit. 2022-05-16]. Dostupné z: https://medium.com/@narner/machine-learning-development-on-apple-platforms-7b49dc0fa383

67. Apple Core ML: Leveraging the power of machine learning for mobile. *Medium*[online]. [cit. 2022-05-16]. Dostupné z: https://postindustria.com/apple-core-ml-leveraging-the-power-of-machine-learning-for-mobile/

68. Machine Learning: Create intelligent features and enable new experiences for your apps by leveraging powerful on-device machine learning. Learn how to build, train, and deploy machine learning models into your iPhone, iPad, Apple Watch, and Mac apps. *Post Industria* [online]. [cit. 2022-05-16]. Dostupné z: https://developer.apple.com/machine-learning/

69. Create ML: Create machine learning models for use in your app. *Apple* [online]. [cit. 2022-05-16]. Dostupné z: https://developer.apple.com/documentation/createml

70. Object Detection with Create ML. *Apple Developer* [online]. 17.2.2022 [cit. 2022-05-16]. Dostupné z: https://evilmartians.com/chronicles/object-detection-with-create-ml-training-and-demo-app

ZOZNAM POUŽITÝCH SYMBOLOV A SKRATIEK

AR Augmented Reality

CNN Convolutional neural network

CV Computer Vision

FN Full Network

I/U Intersection over Union

IDE Integrated Development Environment

JSON JavaScript Object Notation

OS Operačný systém

RGB Red – Green - Blue

SSD Single-shot Detector

TI Thermal Imaging

TL Transfer Learning

UI User Interface

VCS Version Control System

VR Virtual Reality

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ZOZNAM PRÍLOH

P1 CD s diplomovou prácou a súbory obsahujúce zdrojové kódy a datasety