

Rozpoznávání typu odpadu dle recyklačního symbolu

Bc. Jan Horák

Diplomová práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Jan Horák
Osobní číslo: A23311
Studijní program: N0613A140022 Informační technologie
Specializace: Softwarové inženýrství
Forma studia: Prezenční
Téma práce: Rozpoznávání typu odpadu dle recyklačního symbolu
Téma práce anglicky: Waste Type Recognition Based on the Recycling Symbol

Zásady pro vypracování

- Zpracujte rešerši algoritmů hlubokého učení pro zpracování obrazu.
- Nastudujte možnosti implementace těchto algoritmů na embedded zařízení.
- Vytvořte nebo upravte vhodný dataset recyklačních symbolů.
- Aplikujte vhodnou architekturu hluboké sítě pro detekci klasifikačních symbolů.
- Vyhodnoťte vlastnosti zvoleného řešení.

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

Seznam doporučené literatury:

- [1] GOODFELLOW, Ian; BENGIO, Yoshua a COURVILLE, Aaron. *Deep learning*. MIT Press, 2016. ISBN 0262035618.
- [2] HALFACREE, Gareth. *The Official Raspberry Pi Beginner's Guide*. 4th edition. Raspberry Pi Trading, 2019. ISBN 978-1-912047-62-8.
- [3] JHA, Ashish Ranjan, *Mastering Pytorch*, Second edition, Packt, 2024. 978-1-80107-430-8.
- [4] ROSENBROCK, Adrian. *Deep learning for computer vision with Python*. Místo není známo: PyimageSearch, 2017. ISBN 978-1986538138.
- [5] GULLI, Antonio; KAPOOR, Amita a PAL, Sujit. *Deep learning with TensorFlow 2 and Keras: Regression, ConvNets, GANs, RNNs, NLP, and more with TensorFlow 2 and the Keras API*. Second edition. Expert insight. Birmingham: Packt, 2019. ISBN 978-1-83882-341-2.

Vedoucí diplomové práce: **Ing. Jakub Novák, Ph.D.**
Ústav automatizace a řídicí techniky

Datum zadání diplomové práce: **27. října 2024**
Termín odevzdání diplomové práce: **2. června 2025**



L.S.

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

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

Ve Zlině dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- 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;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- 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(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....
podpis autora

ABSTRAKT

Tato diplomová práce se zabývá problematikou nasazení detekčních neuronových sítí na embedded zařízení s omezeným výpočetním výkonem, konkrétně na platformu Raspberry Pi. Hlavním cílem bylo navrhnout, implementovat a vyhodnotit systémy pro detekci recyklačních symbolů pomocí moderních metod strojového vidění a hlubokého učení. V rámci řešení byly porovnány tři modely: SSD MobileNet V2 FPNLite, EfficientDet D0 (oba v TensorFlow) a SSDLite320 MobileNet V3 (v PyTorch). Práce analyzuje výhody a nevýhody jednotlivých přístupů z hlediska přesnosti detekce (mAP, AR), výpočetní efektivity (FPS) a vhodnosti pro embedded nasazení. Nejlepších výsledků z hlediska přesnosti dosáhl model EfficientDet D0, zatímco model SSD MobileNet V2 FPNLite se ukázal jako nejefektivnější z hlediska rychlosti a jednoduchosti nasazení. PyTorch model potvrdil svou funkčnost, avšak s omezeným výkonem. Výsledky ukazují klasický kompromis mezi přesností a výpočetní náročností a poskytují doporučení pro volbu vhodného modelu dle specifických požadavků embedded aplikací.

Klíčová slova: strojové vidění, hluboké učení, detekce objektů, embedded zařízení, Raspberry Pi, TensorFlow, PyTorch

ABSTRACT

This thesis focuses on the deployment of object detection neural networks on embedded devices with limited computational power, specifically on the Raspberry Pi platform. The main objective was to design, implement, and evaluate systems for detecting recycling symbols using modern computer vision and deep learning techniques. Three models were compared: SSD MobileNet V2 FPNLite and EfficientDet D0 (both in TensorFlow), and SSDLite320 MobileNet V3 (in PyTorch). The work analyzes the advantages and disadvantages of each approach in terms of detection accuracy (mAP, AR), computational efficiency (FPS), and suitability for embedded deployment. EfficientDet D0 achieved the highest accuracy, while SSD MobileNet V2 FPNLite proved to be the most efficient in terms of speed and ease of deployment. The PyTorch model confirmed its functionality but showed limited performance. The results highlight the classic trade-off between accuracy and computational demand and provide recommendations for selecting a suitable model according to specific embedded application requirements.

Keywords: computer vision, deep learning, object detection, embedded devices, Raspberry Pi, TensorFlow, PyTorch

OBSAH

ÚVOD.....	13
I TEORETICKÁ ČÁST	14
1 STROJOVÉ VIDĚNÍ.....	15
1.1 DEFINICE STROJOVÉHO VIDĚNÍ	15
1.2 SPOJENÍ STROJOVÉHO VIDĚNÍ S HLUBOKÝM UČENÍM	16
1.3 ÚLOHY STROJOVÉHO VIDĚNÍ	17
1.3.1 Klasifikace obrazu (Image Classification)	17
1.3.2 Detekce objektů (Object detection).....	18
1.3.3 Semantická segmentace (Semantic Segmentation)	20
1.3.4 Instance segmentace (Instance Segmentation).....	20
2 HLUBOKÉ UČENÍ.....	22
2.1 NEURONOVÉ SÍTĚ	22
2.1.1 Základní principy neuronových sítí	23
2.1.2 Architektura neuronových sítí.....	25
2.1.3 Typy neuronových sítí.....	26
2.1.4 Výhody a omezení neuronových sítí.....	28
2.2 TRÉNOVÁNÍ MODELŮ.....	30
2.2.1 Datové množiny	30
2.2.2 Ztrátové funkce	31
2.2.3 Backpropagation a optimalizace	32
2.2.4 Přenos učení (Transfer learning).....	33
2.2.5 Přeučení a generalizace	34
2.2.6 Epochy, batch size a iterace	35
2.2.7 Trénovací smyčka	36
2.2.8 Monitorování a ladění	37
2.3 REGULARIZACE	38
2.3.1 Penalizační metody: L1 a L2 regularizace	38
2.3.2 Dropout	39
2.3.3 Další techniky regularizace	40
2.4 OPTIMALIZACE	41
2.4.1 Gradient descent a jeho varianty	42
2.4.2 Pokročilé optimalizační algoritmy	43
2.4.3 Učení s adaptivní rychlostí (learning rate scheduling).....	44
3 KLASIFIKACE A DETEKCE OBJEKTŮ	46
3.1 PŘEHLED KLASIFIKACE A DETEKCE V POČÍTAČOVÉM VIDĚNÍ.....	46
3.2 VÝVOJ METOD DETEKCE OBJEKTŮ	47
3.2.1 Tradiční přístupy	47
3.2.2 Přechod k hlubokému učení	47
3.2.3 Nástup real-time detekce	47
3.2.4 Současné trendy	48

3.3	ARCHITEKTURY DETEKTORŮ	48
3.3.1	Jednostupňové modely	49
3.3.2	Dvoustupňové modely	50
3.3.3	Transformerové detektory	51
3.4	MODELÝ VHODNÉ PRO EMBEDDED ZAŘÍZENÍ	51
4	METRIKY HODNOCENÍ MODELU	53
4.1	INTERSECTION OVER UNION (IoU)	53
4.2	MEAN AVERAGE PRECISION (MAP)	54
4.3	RECALL	55
4.4	FRAMES PER SECOND (FPS)	55
II	PRAKTICKÁ ČÁST	56
5	FRAMEWORKY PRO HLUBOKÉ UČENÍ	57
5.1	TENSORFLOW	57
5.2	PYTORCH	57
5.3	SROVNÁNÍ TENSORFLOW A PYTORCH	58
6	IMPLEMENTACE HLUBOKÝCH SÍTÍ NA EMBEDDED ZAŘÍZENÍ	60
6.1	VÝZVY PŘI NASAZENÍ NA RASPBERRY PI	60
6.2	OPTIMALIZACE MODELŮ	61
6.3	VYUŽITÝ HARDWARE	61
7	DATASET	63
7.1	DOSTUPNÉ DATASETY	63
7.2	STRUKTURA DATASETU	64
7.3	ANOTACE DATASETU	64
8	DETEKČNÍ MODELÝ A JEJICH IMPLEMENTACE	66
8.1	SSDLITE320 MOBILENET V3	66
8.1.1	Architektura a charakteristika modelu	66
8.1.2	Implementace v PyTorch	67
8.2	SSD MOBILENET V2 FPNLITE	69
8.2.1	Architektura a charakteristika modelu	69
8.2.2	Implementace v TensorFlow Object Detection API	70
8.3	EFFICIENTDET D0	73
8.3.1	Architektura a charakteristika modelu	73
8.3.2	Implementace v TensorFlow Object Detection API	74
9	NASAZENÍ NA RASPBERRY PI	76
9.1	NASAZENÍ MODELŮ TENSORFLOW LITE	76
9.1.1	Převod modelu SSD MobileNet V2 FPNLite	76
9.1.2	Převod modelu EfficientDet D0	77

9.1.3	Spuštění inference na Raspberry Pi.....	77
9.2	NASAZENÍ MODELU PYTORCH.....	79
10	VYHODNOCENÍ MODELU	82
10.1	PŘEHLED VÝSLEDKŮ	82
10.2	PRŮBĚH TRÉNINKU	83
10.3	POROVNÁNÍ VÝKONU	86
ZÁVĚR		89
SEZNAM POUŽITÉ LITERATURY.....		90
SEZNAM OBRÁZKŮ		96
SEZNAM TABULEK.....		98
SEZNAM PŘÍLOH.....		99

ÚVOD

Rozvoj metod strojového vidění a hlubokého učení přináší v posledních letech nové možnosti v oblasti automatizovaného zpracování obrazových dat. Tyto technologie nacházejí využití v široké škále aplikací, mimo jiné i při třídění odpadu, kde je jedním z požadavků schopnost spolehlivě identifikovat recyklační symboly na obalech výrobků. Informace obsažené v těchto symbolech jsou důležité pro následné správné zařazení odpadu do příslušné kategorie zpracování.

Cílem této diplomové práce je navrhnout a implementovat systém umožňující detekci recyklačních symbolů pomocí metod hlubokého učení a ověřit jeho funkčnost na zařízení s omezeným výpočetním výkonem, konkrétně na platformě Raspberry Pi. V rámci řešení byly vybrány a porovnány tři modely neuronových sítí: SSD MobileNet V2 FPNLite, EfficientDet D0 a SSDLite320 MobileNet V3. Tyto modely byly vyhodnoceny z hlediska přesnosti detekce, výpočetní náročnosti a vhodnosti pro použití v prostředí embedded zařízení.

Výsledky práce přispívají k rozvoji praktických aplikací hlubokého učení v oblasti recyklace a mohou sloužit jako základ pro vývoj nízkonákladových inteligentních třídících systémů.

I TEORETICKÁ ČÁST

1 STROJOVÉ VIDĚNÍ

Strojové vidění je technologie, která umožňuje počítačům zpracovávat a analyzovat obrazová data s cílem získat relevantní informace o objektech nebo scénách. Na rozdíl od lidského vidění není strojové vidění omezeno subjektivním vnímáním a dokáže provádět přesné a konzistentní analýzy v reálném čase. Tato technologie nachází široké uplatnění v průmyslu, zdravotnictví, dopravě a ekologii, kde se využívá například pro kontrolu kvality, autonomní navigaci, rozpoznávání objektů a třídění odpadu.

Moderní systémy strojového vidění využívají kamerové senzory pro získávání obrazových dat, která jsou následně zpracována pomocí algoritmů zpracování obrazu a strojového učení. Tyto algoritmy umožňují nejen detekci objektů, ale také jejich klasifikaci, identifikaci a sledování v čase.

Významnou roli v rozvoji strojového vidění sehrálo hluboké učení, které přineslo revoluční změny v analýze obrazových dat. Díky konvolučním neuronovým sítím (CNN) jsou dnešní systémy schopny automaticky extrahovat příznaky z obrazu a na jejich základě provádět pokročilé úlohy, jako je detekce objektů, segmentace obrazu a rozpoznávání scén [1].

Spojení strojového vidění s hlubokým učením výrazně zvyšuje přesnost a flexibilitu systémů, což umožňuje jejich nasazení v náročných prostředích s vysokými požadavky na spolehlivost.

1.1 Definice strojového vidění

Strojové vidění je interdisciplinární obor, který kombinuje zpracování obrazu, počítačové vidění a umělou inteligenci, aby umožnil počítačům analyzovat a interpretovat vizuální informace. Cílem strojového vidění je automatizovat procesy vnímání a rozhodování, které by jinak vyžadovaly lidský zrak. Tento obor zahrnuje jak získávání obrazových dat pomocí kamerových senzorů, tak jejich následné zpracování a analýzu pomocí algoritmů zpracování obrazu a neuronových sítí.

Na rozdíl od tradičního počítačového vidění, které se zaměřuje na matematické a geometrické metody analýzy obrazu, strojové vidění klade důraz na praktické aplikace v průmyslu a automatizaci. Typické aplikace zahrnují kontrolu kvality výrobků, detekci vad, navigaci robotů a třídění odpadu podle vizuálních znaků [2].

Strojové vidění se vyznačuje schopností pracovat v reálném čase a analyzovat velké množství obrazových dat s vysokou přesností. To umožňuje jeho nasazení v prostředích, kde je vyžadována vysoká úroveň spolehlivosti a konzistence. Díky pokroku v oblasti hlubokého učení se strojové vidění stalo klíčovou technologií pro inteligentní systémy, které dokážou autonomně rozhodovat na základě vizuální zpětné vazby [1].

1.2 Spojení strojového vidění s hlubokým učením

Spojení strojového vidění s hlubokým učením představuje revoluční přístup v analýze obrazových dat, který výrazně zvyšuje přesnost a flexibilitu těchto systémů. Tradiční metody strojového vidění využívaly ručně navržené příznaky (features) a klasifikační algoritmy, které však měly omezenou schopnost generalizace a nedokázaly efektivně pracovat se složitými vizuálními scénami.

Hluboké učení, zejména konvoluční neuronové sítě (CNN), přineslo zásadní změnu v přístupu ke zpracování obrazu. CNN dokážou automaticky extrahovat příznaky z obrazových dat na různých úrovních abstrakce, což umožňuje lépe rozpoznávat složité vzory a zvyšuje přesnost klasifikace a detekce objektů [1].

CNN se během trénování samy učí extrahovat příznaky, čímž odpadá potřeba jejich ručního navrhování, typická pro klasické přístupy. Tato vlastnost umožňuje systémům přizpůsobit se různým podmínkám, jako jsou změny osvětlení, pozice objektů nebo různé úhly pohledu. Spojení strojového vidění s hlubokým učením tak poskytuje větší robustnost a adaptabilitu[1].

Díky této flexibilitě je možné využít předtrénované modely, které byly natrénovány na rozsáhlých obrazových databázích a následně je přizpůsobit na specifický dataset, například na rozpoznávání recyklačních symbolů. Tento přístup zkracuje dobu trénování a zlepšuje generalizaci modelu na nové vizuální vzory [3].

Kombinace strojového vidění a hlubokého učení přinesla zásadní pokrok také v detekci objektů a segmentaci obrazu. Moderní architektury, jako jsou YOLO (You Only Look Once) nebo EfficientDet, využívají hluboké neuronové sítě pro jednostupňovou detekci objektů, což umožňuje rychlé a přesné rozpoznávání v reálném čase [4].

1.3 Úlohy strojového vidění

Strojové vidění zahrnuje různé úlohy, které umožňují počítačovým systémům analyzovat a interpretovat vizuální informace z digitálních obrazů. Tyto úlohy se uplatňují v celé řadě aplikací, od průmyslové automatizace přes zdravotnictví až po autonomní systémy.

Mezi základní úlohy strojového vidění patří klasifikace obrazu, detekce objektů, segmentace a instance segmentace. Každá z těchto úloh se zaměřuje na jinou úroveň porozumění obrazu:

- Klasifikace obrazu přiřazuje celému vstupnímu obrazu jednu třídu z předem definovaného množství kategorií.
- Detekce objektů rozpoznává výskyt konkrétních objektů a určuje jejich polohu v obraze pomocí ohraničujících boxů.
- Segmentace obrazu rozděluje obraz na oblasti podle vizuálních vlastností a přiřazuje každému pixelu třídu.
- Instance segmentace jde ještě dál a rozlišuje mezi jednotlivými instancemi objektů, i když patří do stejné kategorie.

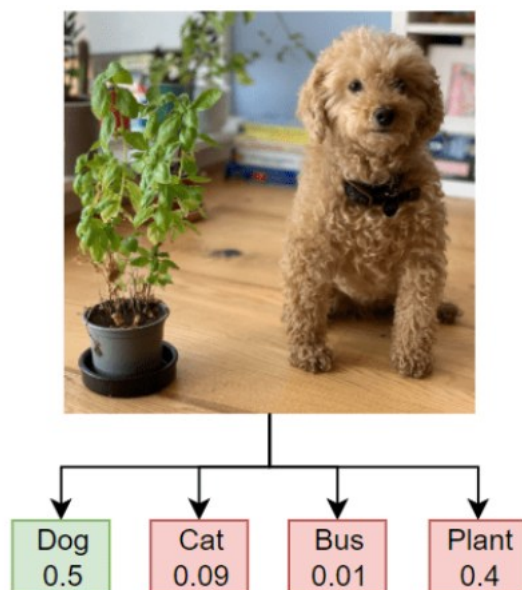
Současné přístupy často kombinují více těchto úloh v jednom systému, což umožňuje komplexní analýzu vizuálních scén. Díky pokroku v oblasti hlubokého učení, zejména konvolučních neuronových sítí (CNN) a transformerových architektur, dosahují modely stále vyšší přesnosti a adaptability. Architektury jako Vision Transformers (ViT) nebo R-CNN poskytují nástroje pro efektivní a škálovatelné zpracování obrazových dat [6].

V následujících podkapitolách jsou jednotlivé úlohy podrobněji popsány z hlediska jejich principů, metod a využití.

1.3.1 Klasifikace obrazu (Image Classification)

Klasifikace obrazu je základní úloha počítačového vidění, jejímž cílem je určit, do které z předem definovaných tříd daný obraz jako celek patří. Systém tedy analyzuje celý vstupní obrázek a rozhodne, kterou kategorii nejlépe reprezentuje – například „auto“, „pták“ nebo „budova“. Výsledkem je přiřazení jediné třídy k celému obrazu, nikoliv k jednotlivým objektům uvnitř něj.

Formálně se jedná o problém vícetřídové klasifikace, kde každý obraz náleží právě jedné třídě z konečného množství. Výstupem modelu bývá vektor pravděpodobností, z něhož je jako finální predikce vybrána třída s nejvyšší hodnotou.



Obrázek 1: Ukázka klasifikace [37]

V minulosti se klasifikace řešila kombinací ručně navržených příznaků (např. hran, textur nebo tvarů) a klasifikačních algoritmů, jako je Support Vector Machine (SVM) – metoda, která hledá optimální hranici mezi třídami v prostoru příznaků. Tyto přístupy však byly citlivé na změny v obraze, například na jiné pozadí nebo úhel pohledu.

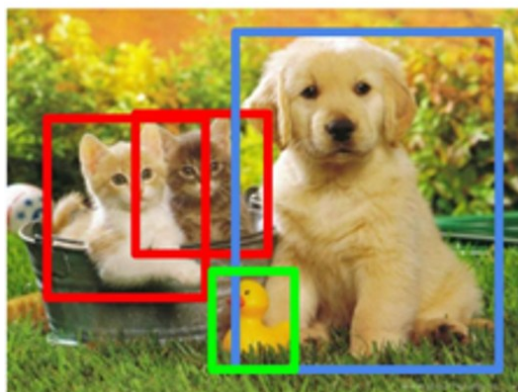
V současnosti je klasifikace obrazů běžně realizována pomocí modelů, které si potřebné příznaky dokáží samy odvodit z dat a naučit se je využívat pro rozhodování. Tento přístup zajišťuje vyšší přesnost i odolnost vůči variabilitě vstupních obrázků, a stal se standardem v moderním počítačovém vidění [1], [3].

Klasifikace obrazu se využívá v celé řadě aplikací, od rozpoznávání objektů ve fotografiích až po analýzu lékařských snímků nebo kontrolu kvality ve výrobě.

1.3.2 Detekce objektů (Object detection)

Detekce objektů (object detection) je úloha počítačového vidění, která umožňuje určit, zda se v obraze nacházejí objekty určitého typu, a zároveň stanovit jejich polohu pomocí ohraničujících boxů (bounding boxes). Tato schopnost je nezbytná v mnoha oblastech, jako jsou autonomní systémy, analýza videa, biometrie nebo automatizované řízení výroby [10].

Object Detection



CAT, DOG, DUCK

Obrázek 2: Ukázka detekce objektů [38]

Tradiční přístupy k detekci objektů využívaly metodu sliding window, při níž se klasifikátor opakovaně aplikoval na různé části obrazu v různých měřítkách. Tento postup byl velmi náročný na výpočetní výkon, protože bylo nutné zpracovat tisíce oblastí. Později se prosadily tzv. region proposal metody, které generují pouze omezený počet kandidátních oblastí pro podrobnější analýzu.

V té době se používaly i modely založené na Haar cascade klasifikátorech, které pracovaly se vzorci jasových změn v obraze a byly známy především v kontextu detekce obličejů. Další metodou byla Histogram of Oriented Gradients (HOG), která analyzuje směr a velikost hran. Ve spojení s Support Vector Machines (SVM) vznikla účinná řešení pro detekci objektů, zejména v prostředí s nízkými výpočetními nároky [11].

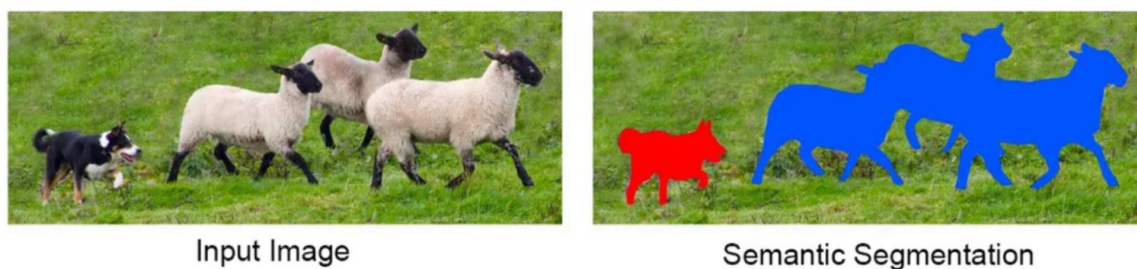
S nástupem konvolučních neuronových sítí (CNN) vznikly modely, které zvládají přesnější detekci při vyšší robustnosti vůči změnám v obraze. V rámci těchto přístupů se rozlišují dvě hlavní skupiny:

- Dvoustupňové modely (two-stage detectors), jako Faster R-CNN, nejprve generují návrhy oblastí, ve kterých by se mohly nacházet objekty, a následně tyto oblasti klasifikují [12].
- Jednostupňové modely (one-stage detectors), jako YOLO (You Only Look Once) a SSD (Single Shot MultiBox Detector), detekují objekty přímo v jednom průchodu sítí, což zajišťuje rychlejší zpracování obrazu [4].

Novější přístupy, jako EfficientDet nebo DETR, využívají optimalizované architektury v kombinaci s mechanismy pozornosti (attention), což přináší vyšší přesnost a lepší zvládnání složitějších scén [13].

1.3.3 Semantická segmentace (Semantic Segmentation)

Semantická segmentace je úloha strojového vidění, jejímž cílem je přiřadit každému pixelu v obraze třídu podle toho, k jakému typu objektu náleží. Na rozdíl od detekce, která pracuje s ohraničujícími boxy, semantická segmentace vytváří tzv. maska tříd, jež poskytuje podrobnější informaci o tvaru a rozložení objektů.



Obrázek 3: Ukázka semantické segmentace [39]

Tento přístup se využívá v aplikacích, kde je důležité porozumět přesnému rozložení scén – například v lékařském zobrazování (označení nádorových oblastí), v autonomních vozidlech (detekce silnic, chodníků, dopravních značek) nebo v zemědělství (analýza ploch porostu) [14].

Tradiční metody segmentace zahrnovaly jednoduché algoritmy, jako je prahování, detekce hran (např. Cannyho detektor) nebo klastrovací metody (např. K-means). Tyto techniky však často selhávaly ve složitějších scénách s vysokou variabilitou nebo šumem [15].

V současnosti se semantická segmentace realizuje převážně pomocí hlubokých neuronových sítí. Modely jako Fully Convolutional Networks (FCN), U-Net nebo DeepLab využívají konvoluční vrstvy k extrakci příznaků a následné dekódování prostorových map do pixelově přesné klasifikace [16].

1.3.4 Instance segmentace (Instance Segmentation)

Instance segmentace rozšiřuje principy semantické segmentace o schopnost rozlišovat jednotlivé výskyty objektů v obraze, i když patří do stejné třídy. Například dvě auta na silnici

budou rozpoznána jako samostatné instance třídy „auto“ a každé z nich bude mít vlastní masku.



Obrázek 4: Ukázka instance segmentace [39]

Tato úloha kombinuje výhody detekce objektů (určení polohy) se segmentací (přesné ohraničení). Využití nachází například v robotice (manipulace s jednotlivými objekty), v bezpečnostních systémech (sledování osob) nebo v biologii (počítání buněk na mikroskopickém snímku).

Moderní přístupy k instance segmentaci zahrnují modely jako Mask R-CNN, které staví na dvoustupňové detekci objektů a přidávají segmentační hlavu pro predikci masky. Novější metody využívají i transformerové architektury – například Segment Anything Model (SAM) od společnosti Meta AI umožňuje interaktivní segmentaci na základě dotazů nebo ukazatelů [17].

2 HLUBOKÉ UČENÍ

Hluboké učení (deep learning) je podmnožinou strojového učení, která využívá vícevrstvé neuronové sítě k analýze a modelování složitých vzorů v datech. Tradiční algoritmy strojového učení se obvykle opírají o ručně definované vstupní rysy, zatímco hluboké učení se je dokáže naučit přímo z dat bez potřeby lidského zásahu. Tento přístup přinesl zásadní pokroky v oblastech, jako je počítačové vidění, zpracování přirozeného jazyka a řízení autonomních systémů [1].

Díky rostoucímu výpočetnímu výkonu a dostupnosti rozsáhlých datasetů se hluboké učení stalo dominantním přístupem v mnoha aplikacích. Konvoluční neuronové sítě (CNN) umožnily revoluční vylepšení analýzy obrazu, zatímco rekurentní neuronové sítě (RNN) a transformerové modely přinesly významné pokroky v oblasti zpracování sekvenčních dat [18].

Hluboké neuronové sítě jsou trénovány na základě zpětnovazební optimalizace (backpropagation), kde se upravují váhy neuronů s cílem minimalizovat chybu modelu. Proces trénování zahrnuje několik klíčových aspektů: výběr architektury sítě, nastavení hyperparametrů, optimalizaci váhových koeficientů a metody regularizace. Správná konfigurace těchto prvků je zásadní pro dosažení dobré generalizace modelu, tedy schopnosti přesně předpovídat výsledky i na dosud neviděných datech [1].

2.1 Neuronové sítě

Neuronové sítě (artificial neural networks, ANN) jsou jedním z klíčových prvků hlubokého učení a jsou inspirovány strukturou lidského mozku. Jsou tvořeny propojenými umělými neurony, které přijímají vstupy, provádějí matematické operace a předávají výsledky dál. Hlavní síla neuronových sítí spočívá ve schopnosti automaticky se učit reprezentace dat, což umožňuje efektivní analýzu složitých vzorů v obrazech, zvuku nebo textu [1].

Hluboké neuronové sítě (deep neural networks, DNN) se skládají z více vrstev neuronů, což jim umožňuje modelovat složité vztahy mezi vstupy a výstupy. Tento přístup byl úspěšně aplikován v různých oblastech, včetně počítačového vidění, zpracování přirozeného jazyka a robotiky. Zásadní pokrok v této oblasti nastal díky zvýšení výpočetního výkonu (GPU, TPU), dostupnosti velkých datasetů a pokročilejším algoritmům optimalizace [18].

Neuronové sítě se liší svou strukturou a typem propojení neuronů, což ovlivňuje jejich schopnosti a vhodnost pro různé úlohy. V následujících podkapitolách budou podrobněji popsány jejich architektura, základní typy a výhody i omezení, které ovlivňují jejich efektivitu v praxi.

2.1.1 Základní principy neuronových sítí

Neuronové sítě jsou výpočetní modely inspirované strukturou lidského mozku, kde jednotlivé umělé neurony spolupracují při zpracování informací. Každý neuron přijímá vstupní signály, aplikuje na ně váženou sumu, přidá bias a následně použije aktivační funkci, která rozhodne, zda neuron vyšle signál dál [1].

Struktura umělého neuronu

Umělý neuron (perceptron) lze matematicky popsat rovnicí:

$$y = f\left(\sum_{i=1}^n w_i x_i + b\right)$$

kde:

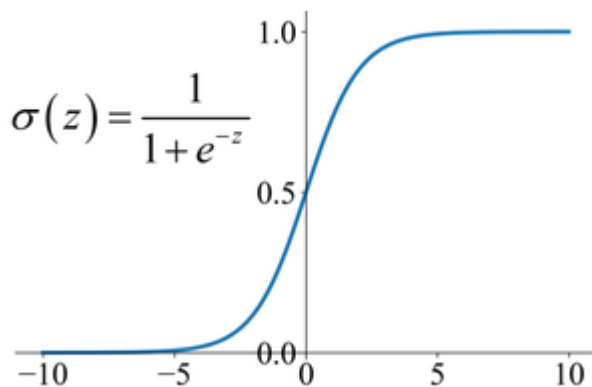
- x_i jsou vstupní hodnoty (např. intenzita pixelu obrazu),
- w_i jsou váhy, které určují důležitost jednotlivých vstupů,
- b je bias, který umožňuje posun aktivační funkce,
- f je aktivační funkce, která zavádí nelinearitu do modelu.

Každý neuron je propojen s ostatními pomocí vážených spojů, které se upravují během trénování modelu.

Aktivační funkce

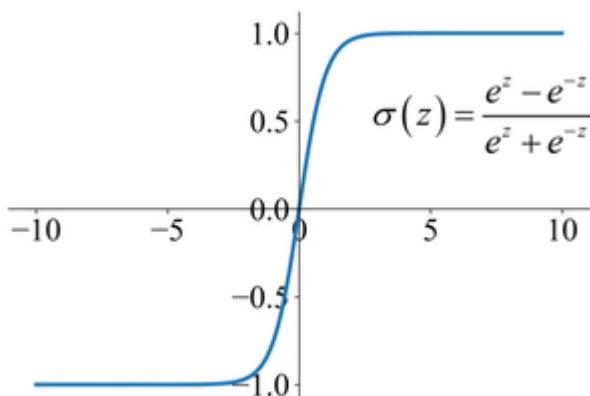
Aby byla neuronová síť schopna modelovat nelineární vztahy v datech, používají se různé aktivační funkce:

- Sigmoid – transformuje výstup do intervalu (0,1), často používaný pro binární klasifikaci.



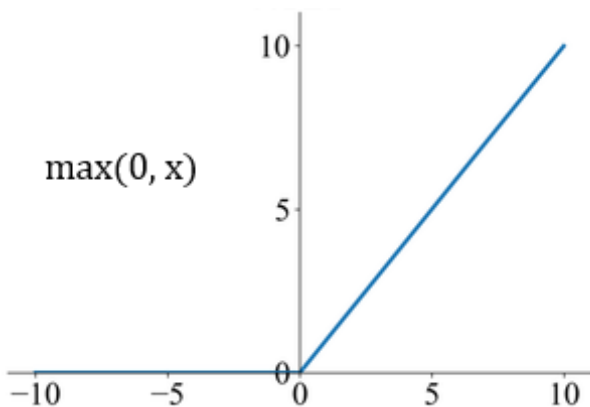
Obrázek 5: Aktivační funkce – Sigmoid [48]

- Tanh – podobná sigmoidní funkci, ale s výstupem v rozsahu (-1,1), lépe se hodí pro data se zápornými hodnotami.



Obrázek 6: Aktivační funkce – Tanh [48]

- ReLU (Rectified Linear Unit) – nejčastější volba v hlubokém učení, definuje výstup jako $\max(0, x)$, což umožňuje rychlejší trénování a lepší gradientový průchod [19].



Obrázek 7: Aktivační funkce – ReLU [48]

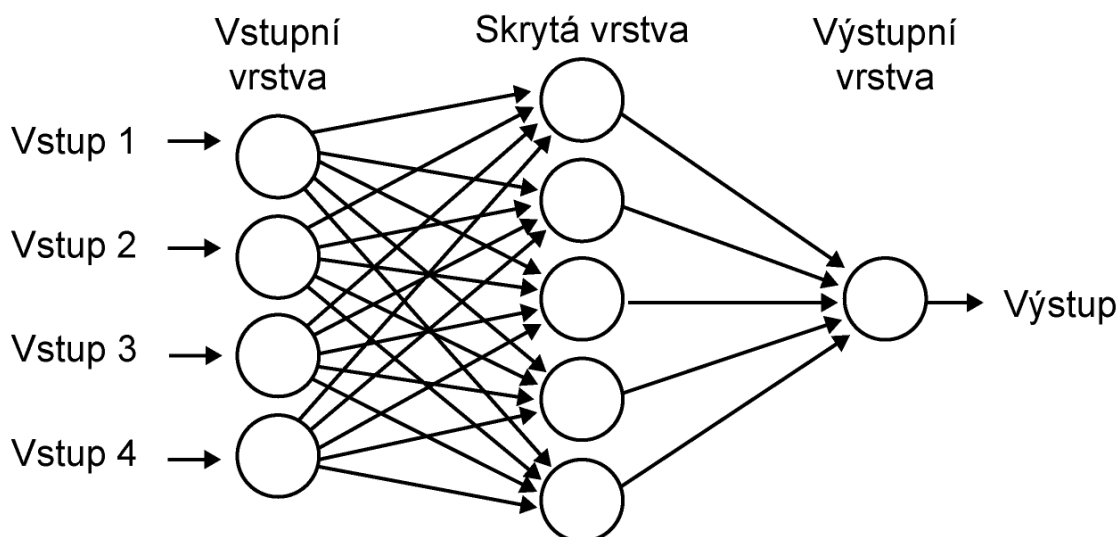
Shrnutí

Neuronové sítě se skládají z umělých neuronů, které spolupracují při zpracování vstupních dat. Jejich schopnost modelovat nelineární vztahy mezi vstupy a výstupy je základem pro řešení složitých úloh strojového učení. V následující kapitole se podrobněji zaměříme na architekturu neuronových sítí, která určuje, jak jsou neurony propojeny a jakým způsobem probíhá zpracování informací v síti.

2.1.2 Architektura neuronových sítí

Architektura neuronové sítě určuje způsob, jakým jsou jednotlivé neurony uspořádány do vrstev a jak jsou mezi sebou propojeny. Tato struktura zásadně ovlivňuje schopnost sítě extrahovat relevantní informace ze vstupních dat a následně je efektivně zpracovávat. V kontextu počítačového vidění a detekce objektů se nejčastěji využívají hluboké neuronové sítě založené na konvolučních vrstvách (CNN), které jsou navrženy tak, aby efektivně pracovaly s prostorovou strukturou obrazových dat [1].

Na nejzákladnější úrovni je neuronová síť tvořena třemi typy vrstev: vstupní, skrytými a výstupní.



Obrázek 8: Neuronové sítě – Vrstvy [49]

Vstupní vrstva přijímá data z okolního prostředí – v případě počítačového vidění se jedná o pixelová data obrazu. Počet neuronů ve vstupní vrstvě odpovídá počtu příznaků, které daný vstup obsahuje (např. počet pixelů v obraze).

Konvoluční vrstvy (convolutional layers) představují klíčový stavební prvek v sítích určených pro zpracování obrazu. Tyto vrstvy využívají sadu filtrů (konvolučních jader),

které se aplikují na vstupní obraz nebo předchozí vrstvu. Výsledkem je tzv. mapování příznaků (feature maps), jež zachycují lokální vzory jako hrany, rohy nebo textury. Použití konvolučních vrstev výrazně snižuje počet parametrů a umožňuje efektivnější učení modelu [3].

Pooling vrstvy, obvykle max-pooling, redukují rozměry map příznaků pomocí subsamplování. Tento proces zajišťuje menší výpočetní náročnost a zároveň zvyšuje robustnost modelu vůči menším posunům nebo deformacím ve vstupu.

Plně propojené vrstvy (fully connected layers) se typicky nacházejí na konci architektury a propojují každý neuron s každým neuronem následující vrstvy. Tím umožňují komplexní kombinaci extrahovaných příznaků z předchozích vrstev a jejich převod na konečný výstup modelu – například pravděpodobnosti příslušnosti ke konkrétním třídám.

Normalizační vrstvy, jako batch normalization, zajišťují stabilitu trénování tím, že normalizují výstupy jednotlivých vrstev. To vede ke zrychlení konvergence a snižuje problémy s vymizením gradientu, které se vyskytují u hlubších architektur.

Dropout vrstvy náhodně vypínají určité neurony během trénování, čímž zabráňují přeučení modelu. Tento jednoduchý, avšak účinný mechanismus zvyšuje schopnost sítě generalizovat na neviděná data [22].

Kombinací těchto typů vrstev lze navrhnout efektivní architektury neuronových sítí, které dosahují vysoké přesnosti při současném zachování výpočetní efektivity.

2.1.3 Typy neuronových sítí

Neuronové sítě se liší svou architekturou a způsobem zpracování dat, což ovlivňuje jejich vhodnost pro různé aplikace. V této kapitole budou popsány základní typy neuronových sítí a jejich využití.

Vícevrstvý perceptron (MLP)

Vícevrstvý perceptron (Multi-Layer Perceptron, MLP) je základní typ neuronové sítě složený z vstupní vrstvy, jedné nebo více skrytých vrstev a výstupní vrstvy. Každý neuron v jedné vrstvě je propojen se všemi neurony v následující vrstvě, čímž vytváří plně propojenou síť.

MLP využívá zpětné šíření chyby (backpropagation) a gradientní sestup pro optimalizaci vah neuronů. Aktivace neuronů je obvykle realizována funkcí ReLU, Sigmoid

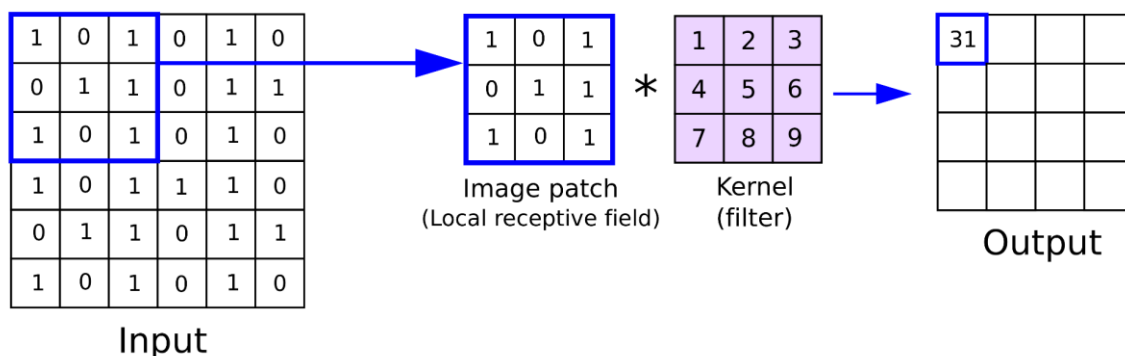
nebo Tanh. Tento typ sítě se používá pro klasifikaci, regresi a základní analýzu dat, ale není vhodný pro zpracování složitějších datových struktur, jako jsou obrazy nebo sekvence [1].

Konvoluční neuronové sítě (CNN)

Konvoluční neuronové sítě (Convolutional Neural Networks, CNN) jsou určeny pro zpracování obrazových dat. Klíčovým prvkem CNN jsou konvoluční vrstvy, které využívají filtry (kernely) k extrakci prostorových rysů z obrazu.

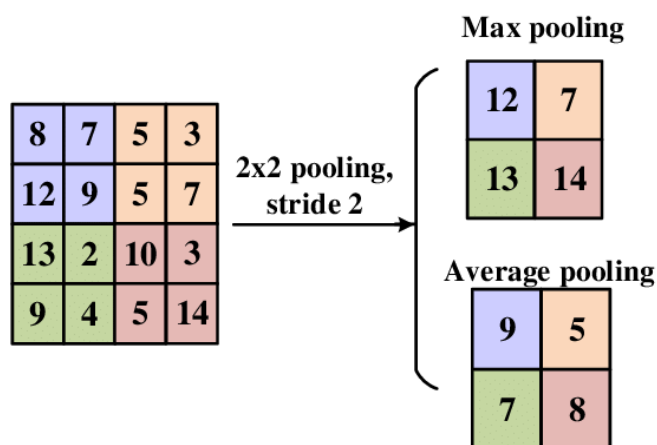
Hlavní výhody CNN spočívají v redukovaném počtu parametrů (díky sdílení vah mezi filtry) a schopnosti zachovat prostorové vztahy mezi pixely. CNN typicky obsahují:

- **Konvoluční vrstvy** – provádějí operace s filtry pro extrakci rysů.



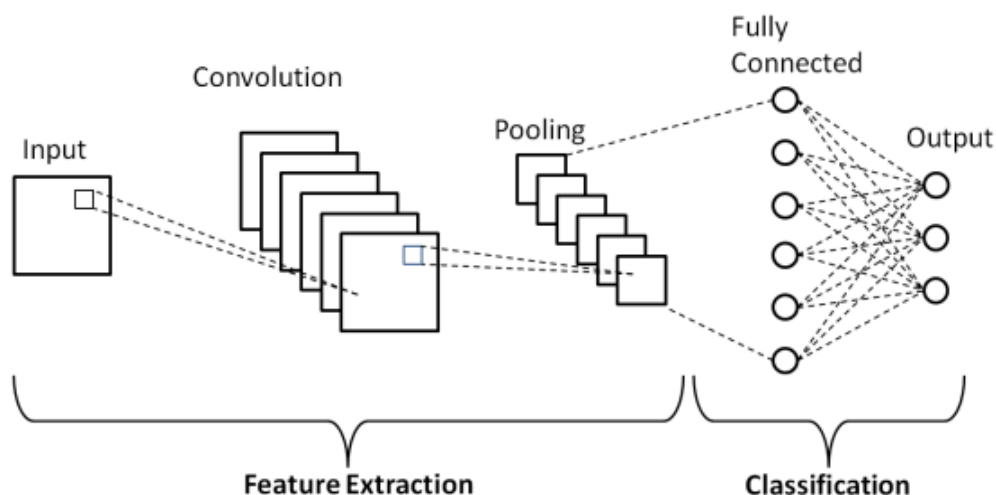
Obrázek 9: Konvoluční vrstva [40]

- **Pooling vrstvy** – redukují rozměry dat a pomáhají minimalizovat výpočetní náročnost.



Obrázek 10: Pooling vrstva [41]

- **Plně propojené vrstvy** – využívají extrahované rysy k finální klasifikaci.



Obrázek 11: Konvoluční neuronová síť [42]

CNN se používají například pro rozpoznávání objektů, detekci symbolů a analýzu lékařských snímků [3].

Srovnání typů neuronových sítí

Tabulka 1: Srovnání typů neuronových sítí

Typ sítě	Charakteristika	Použití	Výhody	Nevýhody
MLP	Plně propojená síť	Klasifikace, regrese	Snadná implementace	Nevhodná pro obrazy a sekvence
CNN	Konvoluční vrstvy	Počítačové vidění	Efektivní zpracování obrazu	Vyžaduje větší dataset

(zdroj: vlastní)

2.1.4 Výhody a omezení neuronových sítí

Neuronové sítě představují jeden z nejefektivnějších nástrojů moderního strojového učení. Jejich schopnost automaticky učit se z dat, modelovat nelineární vztahy a přizpůsobovat se komplexním úlohám je zásadní pro jejich úspěšné nasazení v oblastech, jako je rozpoznávání obrazu, zpracování jazyka či prediktivní analýza.

Výhody neuronových sítí

Jednou z klíčových výhod neuronových sítí je jejich vysoká aproximační schopnost. Již na počátku 90. let bylo teoreticky prokázáno, že vícevrstvé dopředné sítě (MLP) s jednou

skrytou vrstvou a vhodnou aktivační funkcí mohou aproximovat libovolnou spojitou funkci na kompaktní množině [23].

Dále mezi hlavní výhody patří:

- Automatická extrakce příznaků – neuronové sítě eliminují potřebu ručního navrhování vstupních vlastností, což snižuje závislost na doménových znalostech.
- Generalizační schopnosti – při správném trénování dokážou neuronové sítě úspěšně predikovat i na neznámých datech.
- Flexibilita architektur – různé typy sítí (např. CNN, RNN, Transformery) umožňují řešit specifické úlohy s vysokou efektivitou.
- Možnost přenosu znalostí (transfer learning) – využití předtrénovaných modelů výrazně urychluje vývoj aplikací i při malém množství trénovacích dat [18].

Omezení neuronových sítí

Navzdory svým přednostem mají neuronové sítě také významná omezení:

- Výpočetní náročnost – hluboké modely vyžadují značný výpočetní výkon, což omezuje jejich nasazení na embedded zařízení bez speciálních optimalizací.
- Velké množství parametrů – s rostoucí hloubkou sítě roste i počet parametrů, což zvyšuje riziko přeučení (overfitting) a náročnost na data [22].
- Citlivost na kvalitu dat – chybná anotace, šum nebo nevyváženost tříd může výrazně ovlivnit výsledky učení.
- Nízká interpretovatelnost – neuronové sítě jsou často vnímány jako tzv. black-box modely, jejichž rozhodování není snadné vysvětlit. Tato vlastnost komplikuje nasazení sítí v oblastech s vysokými nároky na vysvětlitelnost, například ve zdravotnictví nebo právu [24].
- Komplexita trénování – nalezení optimální architektury a nastavení hyperparametrů je časově náročný proces, často vyžadující experimentální ladění.

Praktické aspekty pro nasazení

V kontextu praktických aplikací, jako je rozpoznávání recyklačních symbolů na embedded zařízeních (např. Raspberry Pi), je nutné zvažovat kompromis mezi výkonem, přesností a

výpočetní efektivitou. Vhodné jsou proto modely navržené pro prostředí s omezenými výpočetními zdroji, jako jsou MobileNet, SSDLite nebo EfficientNet.

Optimalizační techniky jako kvantizace, komprese modelu nebo pruning pak umožňují tyto sítě nasadit i na zařízeních s omezenými prostředky bez výrazného snížení přesnosti.

2.2 Trénování modelů

Trénování modelu hlubokého učení je klíčovým procesem, při němž se neuronová síť učí na základě vstupních dat vytvářet predikce, které odpovídají očekávaným výstupům. Během tohoto procesu dochází k postupné úpravě vah a biasů neuronů s cílem minimalizovat chybu mezi predikovaným a skutečným výstupem. To je umožněno pomocí ztrátové funkce, která kvantifikuje tuto chybu, a následného využití optimalizačních algoritmů, které upravují parametry sítě na základě vypočtených gradientů.

Zásadním prvkem trénování neuronových sítí je také metoda zpětného šíření chyby (backpropagation), která umožňuje efektivní výpočet parciálních derivací ztrátové funkce vůči všem trénovatelným parametrům sítě. Tento mechanismus spolu s vhodně zvoleným optimalizačním algoritmem, jako je například Stochastic Gradient Descent (SGD) nebo Adam, umožňuje sítě postupně zlepšovat své predikční schopnosti.

Trénování však nespočívá pouze v úpravě vah. Kvalita výsledného modelu závisí také na dalších faktorech, jako je volba ztrátové funkce, velikost a kvalita datasetu a délka trénování. Zvláštní pozornost je třeba věnovat problému přeučení (overfittingu), kdy se model příliš přizpůsobí trénovacím datům na úkor jeho výkonu na nových datech. K prevenci přeučení se využívají metody jako dropout, regularizace, nebo časné zastavení trénování (early stopping).

V současné době se běžně využívá také tzv. přenos učení (transfer learning), kdy se model natrénovaný na rozsáhlém datasetu dále přizpůsobí nové úloze s menším množstvím dat.

2.2.1 Datové množiny

Pro efektivní trénování modelů hlubokého učení je důležité správné rozdělení dostupných dat do tří základních skupin: trénovací, validační a testovací množiny. Každá z těchto částí slouží jinému účelu a dohromady tvoří základní rámec pro učení, ladění a ověření schopnosti modelu generalizovat.

- Trénovací data (training set) představují hlavní množinu, na které se model učí. Slouží k optimalizaci vah neuronové sítě tak, aby minimalizovala ztrátovou funkci. Tato data model během trénování opakovaně zpracovává, a právě na nich se učí rozpoznávat vzory a struktury potřebné pro správné predikce.
- Validační data (validation set) se nepoužívají k samotnému učení modelu, ale k jeho průběžnému vyhodnocování během trénování. Na základě výkonu na validační množině lze sledovat vývoj přesnosti, detekovat přeučení a provádět ladění hyperparametrů. Validační data pomáhají určit optimální bod, kdy trénování ukončit, aby model generalizoval co nejlépe.
- Testovací data (test set) slouží až k finálnímu hodnocení modelu po ukončení trénování. Model s těmito daty během učení nepřijde do kontaktu, což zajišťuje objektivní posouzení jeho schopnosti predikovat na zcela nových, neznámých vzorcích. Výsledky na testovací množině obvykle reprezentují očekávaný výkon modelu v reálném nasazení.

Typické rozdělení datasetu bývá například 70 % pro trénování, 15 % pro validaci a 15 % pro testování, nicméně konkrétní poměry mohou být upraveny v závislosti na velikosti a povaze dat. Při nedostatku dat se často využívají techniky jako křížová validace nebo přenos učení.

Správné rozdělení dat je nezbytné pro dosažení kvalitního modelu s dobrou schopností generalizace a minimalizací rizika přeučení.

2.2.2 Ztrátové funkce

Ztrátová funkce (loss function) je klíčovým prvkem při trénování neuronových sítí. Jejím účelem je kvantifikovat chybu modelu, tedy rozdíl mezi skutečným výstupem a predikcí, kterou model na základě svých aktuálních parametrů vytvoří. Hodnota ztrátové funkce slouží jako zpětná vazba pro proces optimalizace, který upravuje váhy neuronové sítě tak, aby výsledná predikce byla co nejblíže očekávané hodnotě [20].

Výběr vhodné ztrátové funkce je závislý na typu řešené úlohy. Pro různé problémy existují specifické typy ztrátových funkcí, které lépe reflektují požadavky dané oblasti:

- Klasifikační úlohy, kde je cílem zařadit vstup do jedné ze tříd, obvykle využívají křížovou entropii (cross-entropy loss). Tato funkce penalizuje rozdíl mezi

pravděpodobnostním výstupem modelu a skutečnou třídou, a je vhodná jak pro binární, tak vícetřídovou klasifikaci.

- Regresní úlohy, v nichž model predikuje spojitou hodnotu, nejčastěji využívají střední kvadratickou chybu (mean squared error, MSE) nebo střední absolutní chybu (mean absolute error, MAE). MSE je vhodná v případech, kdy je žádoucí více penalizovat větší odchylky, zatímco MAE je robustnější vůči odlehlým hodnotám.
- Detekce objektů a podobné úlohy kombinují více ztrát. Například modely typu SSD nebo Faster R-CNN využívají současně klasifikační ztrátu (např. softmax cross-entropy) pro určení třídy objektu a regresní ztrátu (např. smooth L1 loss) pro určení přesné polohy ohraničujícího boxu [10, 12].

Ztrátová funkce tedy představuje centrální prvek učení, který spojuje chování modelu s požadavky uživatele či dané aplikace. Správně zvolená ztráta nejen určuje, co se model učí, ale i jak rychle a stabilně konverguje v průběhu trénování.

Moderní frameworky jako PyTorch a TensorFlow poskytují širokou škálu implementovaných ztrátových funkcí, které lze přímo využít nebo přizpůsobit specifickým potřebám dané úlohy.

2.2.3 Backpropagation a optimalizace

Úprava parametrů neuronové sítě během trénování je umožněna díky algoritmu zpětného šíření chyby (backpropagation). Tento algoritmus efektivně vypočítává gradient ztrátové funkce vůči všem váhám a biasům v síti, což umožňuje jejich následnou aktualizaci tak, aby se minimalizovala chyba na výstupu.

Myšlenka backpropagation byla zásadní pro rozvoj hlubokého učení, neboť umožnila trénovat vícevrstvé neuronové sítě efektivním způsobem [20]. Algoritmus využívá pravidla řetězového derivování a postupuje směrem od výstupní vrstvy zpět ke vstupní, přičemž pro každou vrstvu spočítá příspěvek ke globální chybě modelu.

Výpočet gradientů by ale nebyl prakticky využitelný bez vhodného optimalizačního algoritmu, který podle těchto gradientů aktualizuje parametry. Cílem optimalizace je najít takové hodnoty vah, které minimalizují ztrátovou funkci, a tím zlepšují výkon sítě na trénovacích datech.

Mezi nejčastěji používané optimalizační metody patří:

- Stochastic Gradient Descent (SGD) – jeden z nejjednodušších a nejdéle používaných algoritmů. Aktualizace parametrů probíhá po každém trénovacím vzorku nebo minibatchi. I když je pomalejší a citlivý na volbu rychlosti učení (learning rate), často vede ke stabilní konvergenci.
- Adam (Adaptive Moment Estimation) – pokročilejší metoda kombinující výhody algoritmů SGD a RMSProp. Automaticky upravuje rychlost učení pro každý parametr zvlášť, čímž urychluje konvergenci zejména v hlubokých architekturách [25].

Další metody jako RMSProp, Adagrad nebo AdaDelta nacházejí využití v závislosti na charakteru dat nebo architektuře sítě, nicméně Adam bývá v mnoha aplikacích výchozí volbou díky své stabilitě a rychlosti.

Volba optimalizační metody má zásadní dopad nejen na rychlost trénování, ale také na kvalitu výsledného modelu. Spolu s vhodnou ztrátovou funkcí tvoří základní stavební kámen procesu učení neuronové sítě.

2.2.4 Přenos učení (Transfer learning)

Přenos učení (transfer learning) je technika, která umožňuje využít znalosti získané modelem při řešení jedné úlohy na úlohu jinou, obvykle příbuznou. Ve strojovém učení se tento přístup stal široce používaným zejména v případech, kdy je k dispozici pouze omezené množství trénovacích dat, nebo když není možné trénovat rozsáhlý model od začátku kvůli výpočetní náročnosti.

Typickým případem přenosu učení je využití předtrénovaného modelu na velkém datasetu, který obsahuje přes 14 milionů anotovaných obrazů a tisíce tříd [3]. Modely, které byly na tomto datasetu natrénovány (např. ResNet, VGG, EfficientNet), již obsahují obecné vizuální reprezentace, které lze dále přizpůsobit specifickému problému, jako je například rozpoznávání recyklačních symbolů.

Existují dvě hlavní strategie přenosu učení:

- Feature extraction – využití předtrénovaného modelu jako extraktoru příznaků. Vrstvy zůstávají neměnné a nad jejich výstupy je natrénován nový klasifikátor.
- Fine-tuning – částečné nebo úplné dotrénování modelu, včetně některých nebo všech vrstev, na novém datasetu. Tento přístup je náročnější, ale umožňuje lepší přizpůsobení.

Výhodou přenosu učení je možnost zkrátit dobu trénování, zvýšit přesnost i na malých datasetech a snížit požadavky na množství trénovacích vzorků. To je obzvláště důležité v kontextu nasazení na embedded zařízeních, jako je Raspberry Pi, kde se běžně pracuje s omezenými výpočetními zdroji a specifickými úlohami.

Přenos učení je dnes považován za standardní součást vývoje hlubokých modelů v oblasti počítačového vidění, a je využíván nejen v akademických aplikacích, ale i v průmyslové praxi.

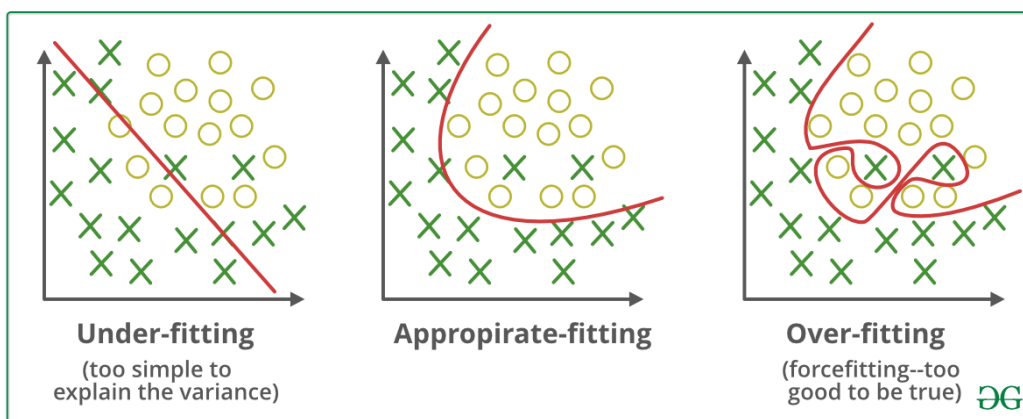
2.2.5 Přeučení a generalizace

Jedním z nejčastějších problémů při trénování neuronových sítí je přeučení (overfitting). Přeučení nastává v situaci, kdy model dosáhne velmi dobrých výsledků na trénovacích datech, ale selhává na datech validačních nebo testovacích, protože se naučil zachycovat šum nebo specifické vzory, které nejsou obecně platné. Výsledkem je nízká schopnost generalizace, tedy schopnost modelu správně reagovat na nová, dosud neviděná data.

K přeučení dochází zpravidla tehdy, když:

- je model příliš složitý (např. hluboká síť s velkým počtem parametrů),
- trénovací data jsou nedostatečná nebo nevyvážená,
- chybí vhodná forma regulace učení.

Naopak underfitting představuje stav, kdy je model příliš jednoduchý a nedokáže zachytit ani základní vztahy v datech.



Obrázek 12: Overfitting & Underfitting [43]

Přeučení lze rozpoznat podle průběhu chyb na trénovací a validační množině. Typicky při přeučení:

- trénovací chyba klesá, zatímco
- validační chyba po určitém bodě roste.

Metody prevence přeučení

Aby model dosahoval dobré generalizace, využívají se různé strategie, které omezují jeho schopnost „naučit se z paměti“:

- Dropout – technika, při které se během trénování náhodně vypínají jednotlivé neurony. Tím se model stává robustnější a méně závislý na konkrétních propojeních [22].
- Early stopping – trénování je ukončeno v okamžiku, kdy se validační chyba přestane snižovat, čímž se předejde dalšímu zhoršování generalizace.
- Regularizace – přidání penalizačního členu do ztrátové funkce (např. L1 nebo L2 normy), který omezuje velikost vah a podporuje jednodušší modely.
- Augmentace dat – umělé rozšíření trénovací množiny pomocí transformací, jako jsou rotace, ořezy, změny kontrastu či přidání šumu. Tím se zvyšuje rozmanitost vstupů a snižuje riziko přeučení [26].

Efektivní trénování modelu by mělo vždy usilovat o dosažení vyváženého kompromisu mezi učením detailních vzorů v datech a schopností zobecnit tyto znalosti na nové příklady. Generalizace je totiž základním předpokladem pro nasazení modelů v reálném prostředí.

2.2.6 Epochy, batch size a iterace

Trénování neuronové sítě je strukturováno do opakujících se kroků, které se odehrávají na různých úrovních – nejčastěji se pracuje s pojmy epocha, batch a iterace.

Epocha představuje jeden kompletní průchod celým trénovacím datasetem. Pokud má síť za úkol naučit se rozpoznávat určité vzory, je třeba, aby každý vzorek v datech byl použit opakovaně – a právě každý takový cyklus odpovídá jedné epoše. Trénování často probíhá v desítkách až stovkách epoch v závislosti na velikosti datasetu, složitosti úlohy a cílové přesnosti.

Batch size (velikost dávky) určuje, kolik trénovacích vzorků je zpracováno současně při jedné aktualizaci vah. Pokud je batch size rovna velikosti celého datasetu, hovoříme o full-batch learningu, který je přesný, ale výpočetně náročný. Mini-batch learning, kdy jsou dávky tvořeny menšími skupinami (např. 32, 64 nebo 128 vzorků), je nejběžnějším

přístupem, protože nabízí dobrý kompromis mezi rychlostí a stabilitou učení. Nejmenší možná velikost dávky je jedna – v tomto případě jde o tzv. stochastic (online) learning, který může být užitečný pro velmi dynamická nebo proudová data [27].

Jedna iterace pak odpovídá jednomu kroku učení, při němž se zpracuje jedna dávka. Počet iterací v jedné epoše tedy závisí na velikosti datasetu a zvoleném batch size. Například pokud má dataset 10 000 vzorků a batch size je 100, pak jedna epocha obsahuje 100 iterací.

Volba těchto parametrů – zejména batch size a počet epoch – má přímý vliv na:

- rychlost trénování,
- stabilitu konvergence,
- schopnost generalizace.

Jejich vhodné nastavení je výsledkem kombinace teoretických znalostí, praktické zkušenosti a ladění na konkrétní úloze.

2.2.7 Trénovací smyčka

Trénovací smyčka představuje jádro procesu učení neuronové sítě. V rámci každé iterace dochází k postupnému zpracování dávky vstupních dat, výpočtu výstupu modelu, vyhodnocení chyby pomocí ztrátové funkce, výpočtu gradientů a následné aktualizaci vah sítě. Tento cyklus se opakuje v průběhu všech iterací v jedné epoše a je klíčový pro to, aby se model na základě zkušeností postupně zlepšoval.

Během jedné iterace probíhají zpravidla následující kroky:

- Forward pass – vstupní data procházejí sítí a na jejím výstupu vzniká predikce.
- Výpočet ztráty – výstup modelu je porovnán s cílovou hodnotou prostřednictvím ztrátové funkce.
- Backward pass (zpětné šíření chyby) – pomocí algoritmu backpropagation jsou spočítány gradienty ztráty vzhledem k vahám.
- Optimalizace (aktualizace vah) – na základě vypočtených gradientů a zvoleného optimalizačního algoritmu (např. SGD nebo Adam) jsou váhy sítě upraveny.

Součástí trénovací smyčky je často také výpočet doplňkových metrik, jako je přesnost (accuracy), recall, nebo F1 skóre, které poskytují hlubší vhled do toho, jak model skutečně funguje v daném kontextu.

Po zpracování všech batchů v jedné epoše je obvykle provedeno vyhodnocení modelu na validační množině. Pokud je validační výkon stabilní nebo se zlepšuje, trénování pokračuje do další epochy. V opačném případě může být spuštěn mechanismus early stopping, případně úprava hyperparametrů.

Trénovací smyčku lze implementovat ručně, například v PyTorchu nebo TensorFlow, nebo využít vyšších abstrakčních nástrojů (např. `fit()` metodu v Kerasu či knihovny jako PyTorch Lightning)[27].

2.2.8 Monitorování a ladění

Během trénování neuronové sítě je důležité nejen definovat architekturu a optimalizační strategii, ale také průběžně sledovat chování modelu, vyhodnocovat jeho výkonnost a včas reagovat na případné problémy. Právě monitorování a ladění představují klíčovou část efektivního trénovacího procesu, která pomáhá předcházet přeučení, zajišťuje stabilitu učení a podporuje dosažení co nejlepších výsledků.

Základním nástrojem pro dohled nad učením jsou trénovací a validační metriky. Mezi nejčastěji sledované hodnoty patří:

- Ztrátová funkce (loss) – jak na trénovací, tak na validační množině.
- Klasifikační metriky, např. přesnost (accuracy), recall, precision, F1 skóre.
- Metriky lokalizace a detekce, např. Intersection over Union (IoU) nebo mean Average Precision (mAP).

Sledování vývoje těchto metrik v průběhu trénování umožňuje identifikovat trendy a anomálie, jako je například začínající přeučení, zpomalení konvergence nebo nestabilita gradientů. Na základě těchto pozorování lze upravit hyperparametry (např. learning rate), aktivovat regularizační techniky nebo rozhodnout o zastavení trénování.

Pro zajištění replikovatelnosti a efektivity trénování se využívá checkpointing, tedy pravidelné ukládání stavu modelu (váh a konfigurace) během učení. Uložené checkpointy lze později využít pro obnovu modelu, pokračování v učení nebo testování ve specifických bodech průběhu trénování.

Monitorování lze provádět ručně pomocí výpisů a grafů, nebo automatizovaně pomocí specializovaných nástrojů, jako jsou:

- TensorBoard (pro TensorFlow),

- Weights & Biases nebo MLflow (pro více frameworků),
- knihovny pro vizualizaci v Pythonu, např. Matplotlib nebo Seaborn [28].

Efektivní ladění modelu vyžaduje kombinaci technického dohledu, interpretace metrik a zkušenosti. Vhodné nastavení monitoringu a ukládání výsledků je tak nedílnou součástí robustního vývojového procesu v oblasti hlubokého učení.

2.3 Regularizace

Jedním z hlavních cílů trénování neuronové sítě je dosáhnout takového stavu, kdy model nebude pouze kopírovat trénovací množinu, ale bude schopen generalizovat své znalosti na nová, neznámá vstupní data. Tento požadavek naráží na riziko přeučení (overfittingu), při němž se síť příliš přizpůsobí specifikům trénovací množiny, a tím ztrácí schopnost predikovat kvalitně v reálném nasazení.

Regularizace představuje soubor technik, jejichž cílem je omezit kapacitu modelu, případně řízeným způsobem narušit proces učení tak, aby síť vytvářela obecnější rozhodovací hranice. Jinými slovy – regularizace pomáhá modelu učit se jednodušší, méně „přesné“, ale o to lépe zobecnitelné funkce.

V oblasti hlubokého učení existuje několik typických přístupů k regularizaci:

- Penalizace vah (např. L1 nebo L2 regularizace), která přímo ovlivňuje velikost parametrů modelu [23].
- Dropout, tedy technika náhodného vypínání neuronů během trénování [22].
- Early stopping, které sleduje vývoj validační chyby a včas zastavuje trénování.
- Data augmentation, které zvyšuje rozmanitost vstupních dat.

Regularizace se stala nepostradatelnou součástí trénování hlubokých neuronových sítí a nachází uplatnění napříč různými typy modelů i aplikací. V následujících podkapitolách budou podrobněji představeny jednotlivé techniky, včetně jejich principu, výhod a případů vhodného použití.

2.3.1 Penalizační metody: L1 a L2 regularizace

Jedním z nejčastěji používaných způsobů regularizace neuronových sítí je penalizace vah modelu prostřednictvím modifikované ztrátové funkce. Cílem těchto metod je omezit růst hodnot vah během trénování tak, aby model preferoval jednodušší a stabilnější řešení s

lepší schopností generalizace. Penalizační termín se přidává k původní ztrátové funkci a při trénování ovlivňuje optimalizační proces.

L2 regularizace (ridge penalty)

L2 regularizace zavádí penalizační člen ve tvaru součtu druhých mocnin vah:

$$L = L_o + \lambda \sum_1 w_i^2$$

kde L_o je původní ztrátová funkce, w_i jsou váhy modelu a λ je regularizační parametr, který určuje sílu penalizace.

Tento přístup vede ke snižování absolutních hodnot vah, aniž by je zcela nuloval. L2 regularizace je často využívána pro své stabilizační účinky – podporuje hladké rozhodovací hranice a eliminuje extrémní hodnoty parametrů [23].

L1 regularizace (lasso penalty)

Oproti tomu L1 regularizace používá součet absolutních hodnot vah:

$$L = L_o + \lambda \sum_i |w_i|$$

Tato forma penalizace má tendenci některé váhy nastavit přímo na nulu, čímž vytváří řidší modely s menším počtem aktivních parametrů. L1 regularizace je tedy vhodná zejména tehdy, když je žádoucí snížit složitost modelu nebo identifikovat nejdůležitější vstupní proměnné.

Kombinované metody

V některých případech se využívá kombinace obou přístupů, označovaná jako Elastic Net, která kombinuje výhody L1 i L2 regularizace.

Volba mezi L1 a L2 závisí na charakteru dat a požadované interpretovatelnosti modelu. V hlubokém učení se častěji používá L2 regularizace, protože je stabilnější a lépe se chová při trénování pomocí gradientních metod.

2.3.2 Dropout

Dropout je jedna z nejúčinnějších a nejpoužívanějších regularizačních technik v hlubokém učení. Její hlavní myšlenkou je náhodně vypínat (deaktivovat) neurony během trénování sítě, a tím zabránit jejich nadměrné závislosti na sobě navzájem. Výsledkem je model, který je robustnější, lépe generalizuje a méně se přeučuje.

Při použití dropoutu se v každé trénovací iteraci každý neuron s určitou pravděpodobností p dočasně vyřadí z výpočtu. V následující iteraci může být aktivní opět jiná podmnožina neuronů. Tímto způsobem dochází k trénování celé množiny různě „osekaných“ modelů, jejichž predikce se při testování průměrují do jednoho finálního modelu. Tento mechanismus má efekt podobný baggingu, tedy kombinaci mnoha slabších modelů za účelem zvýšení stability výstupu [22].

Dropout se obvykle používá ve skrytých vrstvách neuronových sítí, především mezi plně propojenými (dense) vrstvami. Typická hodnota pravděpodobnosti vypnutí neuronů se pohybuje mezi 0.2 až 0.5. U výstupních vrstev a v konvolučních vrstvách bývá dropout používán méně často nebo s menší intenzitou, aby nebyla narušena prostorová struktura dat.

Použití dropout techniky přináší několik výhod:

- Redukuje riziko přeučení, zejména u hlubokých modelů a menších datasetů.
- Zvyšuje robustnost modelu, protože síť se učí poskytovat správné predikce i při absenci části informací.
- Funguje bez nutnosti přidávat nové hyperparametry, kromě samotné dropout pravděpodobnosti.

Při testování modelu se dropout již nepoužívá – místo toho se výstupy neuronů škálují, aby odpovídaly trénovací fázi. Například v PyTorchu se během inference automaticky přepne model do režimu eval(), kde je dropout deaktivován a všechny neurony jsou aktivní.

2.3.3 Další techniky regularizace

Kromě penalizačních metod a dropout techniky existuje řada dalších postupů, které přispívají ke zlepšení schopnosti modelu generalizovat. Tyto techniky se často používají samostatně i v kombinaci a zaměřují se buď na proces trénování, nebo na vstupní data.

Early stopping

Early stopping je metoda, při které se trénování ukončí ve chvíli, kdy se výkon modelu na validační množině přestane zlepšovat, a začne se zhoršovat. Tento postup brání modelu, aby pokračoval v učení na specifika trénovací sady, která nejsou přítomna v nových datech. Early stopping je snadno implementovatelný a bývá standardní součástí trénovacích frameworků – umožňuje tak automaticky vybrat optimální bod zastavení trénování a zároveň šetřit výpočetní čas [1].

Data augmentation

Augmentace dat je technika, při níž se uměle rozšiřuje trénovací dataset pomocí transformací původních vzorků. Cílem je zvýšit variabilitu dat bez nutnosti jejich manuálního rozšiřování.

Typické techniky zahrnují:

- geometrické transformace (rotace, posuny, ořezy),
- změny v barevném prostoru (jas, kontrast, sytost),
- přidání šumu nebo deformací.

Tyto úpravy pomáhají modelu stát se robustnějším vůči změnám ve vstupních datech a snižují riziko přeučení. Augmentace se běžně používá zejména v oblasti počítačového vidění [26].

Batch normalization

Batch normalizace zajišťuje stabilitu trénování tím, že normalizuje výstupy jednotlivých vrstev v průběhu učení. Ačkoli jejím primárním cílem není regularizace, přináší vedlejší efekt zvýšení robustnosti modelu, což v důsledku pomáhá i jeho schopnosti generalizace. V kombinaci s dropoutem bývá běžnou součástí moderních architektur.

Přidání šumu a noise injection

Další možností je přidání náhodného šumu do vstupních dat nebo do aktivačních výstupů neuronů. Tento přístup nutí model vytvářet robustnější reprezentace, které nejsou závislé na přesné hodnotě jednotlivých vstupů.

2.4 Optimalizace

Cílem trénování neuronové sítě je nalezení takových parametrů (váh a biasů), které minimalizují hodnotu ztrátové funkce. Tento proces, známý jako optimalizace, je klíčovým prvkem hlubokého učení. Během optimalizace se model učí, jak nejlépe přizpůsobit své vnitřní reprezentace vstupním datům, tak aby vytvářel co nejpřesnější predikce.

Ve své základní podobě je optimalizace založena na iterativní aktualizaci parametrů sítě pomocí gradientu ztrátové funkce. Tento gradient určuje směr, kterým se mají parametry upravit, aby došlo ke snížení chyby. Rychlost této úpravy je řízena tzv. learning rate – jedním z nejdůležitějších hyperparametrů celého modelu.

V průběhu vývoje hlubokého učení vzniklo množství optimalizačních algoritmů, které rozšiřují základní přístup známý jako Gradient Descent. Tyto algoritmy se snaží řešit některé z problémů tradičního gradientního sestupu, jako je pomalá konvergence, kolísání gradientů nebo uvíznutí v lokálních minimech.

Dnes se běžně používají pokročilé metody jako Momentum, RMSProp nebo Adam, které přinášejí adaptivní řízení kroků učení a stabilnější průběh optimalizace [25]. Důležitou součástí trénování je také plánování rychlosti učení v čase (learning rate scheduling), které umožňuje modelu rychleji konvergovat nebo lépe vycházet z komplikovaných oblastí ztrátové funkce [1].

V následujících podkapitolách budou podrobněji představeny jak základní principy optimalizace, tak i pokročilé algoritmy a strategie řízení tempa učení. Tyto techniky hrají zásadní roli v tom, zda se model naučí efektivně, stabilně a s dobrými výsledky.

2.4.1 Gradient descent a jeho varianty

Základem většiny optimalizačních algoritmů v hlubokém učení je metoda gradientního sestupu (Gradient Descent). Jedná se o jednoduchý, avšak velmi účinný způsob, jak iterativně upravovat parametry modelu tak, aby se minimalizovala hodnota ztrátové funkce. V každém kroku se vypočítá gradient ztráty vzhledem k parametrům a model se „posune“ ve směru, který vede ke snížení chyby.

Matematicky lze základní aktualizaci parametrů zapsat jako:

$$\theta = \theta - \eta \cdot \nabla_{\theta} L(\theta)$$

kde θ představuje parametry modelu, η je learning rate a $\nabla_{\theta} L(\theta)$ je gradient ztráty vzhledem k parametrům.

V praxi se využívají tři hlavní varianty gradientního sestupu, lišící se tím, kolik dat se použije při výpočtu gradientu v jednom kroku:

Batch Gradient Descent

Používá celý trénovací dataset pro výpočet gradientu. Je přesný, ale paměťově a výpočetně náročný, zejména u rozsáhlých datasetů. Má stabilní konvergenci, ale pomalou aktualizaci.

Stochastic Gradient Descent (SGD)

V každém kroku se aktualizace provádí na základě jednoho náhodně vybraného vzorku. Tato metoda je rychlá a často umožňuje dostat se z lokálních minim, ale je velmi nestabilní – ztráta může výrazně kolísat mezi iteracemi [1].

Mini-Batch Gradient Descent

Představuje kompromis mezi předchozími dvěma přístupy. V každé iteraci se vypočítá gradient na základě malé podmnožiny dat (typicky 32 až 256 vzorků). Tato metoda je dnes nejpoužívanější, protože nabízí dobrý poměr mezi stabilitou a rychlostí konvergence.

Volba správné varianty gradientního sestupu a velikosti dávky (batch size) má zásadní vliv na efektivitu trénování modelu. V dalším textu budou představeny pokročilé algoritmy, které na základní principy gradientního sestupu navazují a řeší některé jeho praktické nedostatky [25].

2.4.2 Pokročilé optimalizační algoritmy

Zatímco základní varianty gradientního sestupu poskytují jednoduchý rámec pro aktualizaci parametrů modelu, v praxi často čelí problémům, jako je pomalá konvergence, kolísání gradientů nebo uvíznutí v lokálních minimech. Proto byly vyvinuty pokročilejší optimalizační algoritmy, které tyto nedostatky adresují a výrazně zvyšují efektivitu a stabilitu trénování hlubokých sítí.

Momentum

Metoda Momentum zavádí do aktualizace vah tzv. setrvačnost. Místo toho, aby se parametry aktualizovaly pouze na základě aktuálního gradientu, přihlíží se také k předchozím krokům. To umožňuje zrychlit konvergenci zejména v údolích ztrátové funkce a omezit oscilace v oblastech s kolísavým gradientem. Aktualizace má podobu:

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_{\theta} L(\theta)$$

$$\theta = \theta - v_t$$

kde v je rychlost, γ je parametr momenta (obvykle 0.9), a η je learning rate.

RMSProp

RMSProp (Root Mean Square Propagation) řeší problém nestabilních kroků učením různou rychlostí pro různé parametry. Sleduje průměrnou velikost nedávných gradientů a na jejím základě upravuje velikost kroku:

$$\theta = \theta - \frac{\eta}{\sqrt{E|g^2|_t + \epsilon}} \cdot g_t$$

Tato metoda výrazně zlepšuje konvergenci v nerovnoměrně zakřivených oblastech funkce ztráty, a je vhodná zejména pro rekurentní sítě nebo velmi hluboké architektury.

Adam

Optimalizační algoritmus Adam (Adaptive Moment Estimation) kombinuje výhody Momentum a RMSProp. Udržuje odhad první (průměr gradientu) i druhé momentové složky (průměr druhé mocniny gradientu) a adaptivně upravuje learning rate pro každý parametr zvlášť. Jeho efektivita, robustnost a minimální potřeba ladění z něj učinily jednu z nejčastěji používaných metod v současném hlubokém učení [25].

Adam je v praxi vhodný pro většinu úloh, a to i bez důkladného nastavování parametrů. Typické výchozí hodnoty jsou $\beta_1 = 0,9$, $\beta_2 = 0,999$ a $\epsilon = 10^{-8}$.

2.4.3 Učení s adaptivní rychlostí (learning rate scheduling)

Jedním z nejdůležitějších hyperparametrů při trénování neuronových sítí je learning rate, tedy velikost kroku, kterým se model posouvá po gradientu ztráty směrem k minimu. Nevhodně zvolená hodnota může mít zásadní dopad na úspěšnost učení: příliš vysoká vede k oscilacím nebo divergence, zatímco příliš nízká zpomaluje konvergenci nebo vede k uvíznutí v suboptimálním minimu.

Proto se v praxi často využívá learning rate scheduling, tedy řízení hodnoty learning rate v čase. Místo konstantní hodnoty se tento parametr dynamicky mění během trénování podle předem zvoleného plánu. Mezi běžně používané strategie patří:

Step decay

Learning rate se sníží o pevný faktor po určitém počtu epoch. Například po každých 10 epochách se sníží o 50 %. Tato metoda je jednoduchá, ale může působit skokově a neohebně.

Exponential decay

Hodnota learning rate se snižuje exponenciálně v čase podle rovnice:

$$\eta_t = \eta_0 \cdot e^{-kt}$$

kde η_0 je počáteční learning rate, k je konstanta a t je epocha. Tato metoda je plynulejší než step decay.

Cosine annealing

Hodnota learning rate se mění podle kosinové křivky a periodicky klesá téměř k nule, čímž modelu umožňuje efektivní průchod přes složitý prostor parametrů. Může být kombinována s tzv. warm restarts – opětovným navýšením hodnoty po určité fázi.

Cyclical learning rate

Namísto monotónního klesání se learning rate mění v rámci intervalu mezi minimální a maximální hodnotou. Tato strategie umožňuje modelu lépe procházet různými oblastmi ztrátové funkce a často vede k rychlejší konvergenci.

One Cycle Policy

Tato moderní metoda spojuje prvky několika výše uvedených technik. Learning rate se zpočátku zvyšuje a následně prudce klesá, zatímco zároveň klesá i velikost momenta. Výsledkem bývá rychlé a stabilní dosažení minima ztrátové funkce [29].

Správně zvolený plán změny learning rate má zásadní vliv na kvalitu výsledného modelu, rychlost trénování a stabilitu konvergence. Většina moderních frameworků (např. PyTorch, TensorFlow) poskytuje širokou podporu pro plánování učení, a umožňuje jejich snadné použití v trénovacích smyčkách [1].

3 KLASIFIKACE A DETEKCE OBJEKTŮ

Klasifikace a detekce objektů jsou dvě základní úlohy počítačového vidění, které hrají zásadní roli při automatické interpretaci obrazových dat. Ačkoli obě úlohy vycházejí ze společných principů hlubokého učení, liší se v typu výstupu, výpočetní náročnosti i rozsahu praktického použití.

Zatímco klasifikace se zaměřuje na určení přítomnosti určitého typu objektu na celém snímku, detekce umožňuje současné rozpoznání více objektů v obraze a určení jejich přesné pozice. Detekce je proto složitější úlohou, která vyžaduje schopnost lokalizace a škálování s ohledem na velikost i rozložení objektů.

Moderní metody detekce objektů, jako jsou Faster R-CNN, YOLO či EfficientDet, představují vrchol současného vývoje v oblasti strojového vidění. Liší se přitom nejen svou architekturou, ale také nároky na výpočetní prostředky, čímž se některé z nich stávají vhodnými kandidáty pro použití na embedded zařízeních s omezeným výkonem.

3.1 Přehled klasifikace a detekce v počítačovém vidění

Počítačové vidění jako disciplína usiluje o to, aby počítače dokázaly interpretovat vizuální informace obdobně jako člověk. Mezi základní úlohy tohoto oboru patří klasifikace a detekce objektů, které tvoří základní stavební kámen většiny pokročilých vizuálních systémů – od medicínské diagnostiky až po rozpoznávání dopravních značek či symbolů na obalech.

Klasifikace obrazu spočívá v přiřazení jedné nebo více tříd celému snímku. Výsledkem je informace o tom, „co“ se na obrázku nachází, nikoli „kde“. Tento přístup se osvědčuje například při identifikaci typu materiálu, značky výrobku nebo typu prostředí. Modely pro klasifikaci jsou často méně výpočetně náročné a poskytují vysokou přesnost v jednodušších úlohách, kde se předpokládá jeden dominantní objekt [1].

Detekce objektů oproti tomu přináší schopnost nejen určit typ objektu, ale také jeho konkrétní polohu v obraze, často ve formě ohraničujícího rámečku (bounding box). Tato úloha je výrazně komplexnější, protože musí řešit vícenásobnou přítomnost objektů, jejich různorodé velikosti, polohy i překryvy. Detekce je klíčová v aplikacích, kde není předem znám počet ani pozice objektů – například při automatickém třídění odpadu podle recyklačních symbolů [30].

Klasifikace i detekce jsou dnes běžně řešeny pomocí konvolučních neuronových sítí (CNN), které umožňují automatické extrahování relevantních příznaků z obrazových dat bez

nutnosti ručního návrhu deskriptorů. Zatímco klasifikace obvykle využívá architektury jako VGG, ResNet či EfficientNet, detekce vyžaduje složitější modely schopné generovat lokalizační informace – například Faster R-CNN nebo YOLO [31].

Porozumění rozdílům mezi těmito úlohami je zásadní pro výběr vhodného přístupu při návrhu vizuálního systému. V této práci, která se zaměřuje na detekci různých recyklačních symbolů, se klasifikace ukazuje jako nedostatečná – zejména kvůli tomu, že neumožňuje lokalizaci více objektů současně. Detekce je proto zvolena jako hlavní přístup, protože umožňuje jak identifikaci, tak lokalizaci symbolů, bez ohledu na jejich počet, velikost či umístění.

3.2 Vývoj metod detekce objektů

Detekce objektů patří mezi nejnáročnější úlohy počítačového vidění. Její vývoj prošel během posledních dvou desetiletí zásadní proměnou – od tradičních algoritmů založených na ručně navržených deskriptorech až po dnešní architektury využívající hluboké neuronové sítě. Tento vývoj významně ovlivnil přesnost, rychlost a robustnost detekčních systémů [30, 31].

3.2.1 Tradiční přístupy

Tradiční systémy strojového vidění využívaly ručně navržené příznaky a klasifikátory, což však omezovalo jejich schopnost generalizace. Hluboké učení tento přístup zásadně změnilo díky automatickému učení reprezentací přímo z dat.

3.2.2 Přejít k hlubokému učení

Zásadní zlom přinesl rok 2012, kdy síť AlexNet výrazně překonala dosavadní přístupy v soutěži ImageNet [3]. Brzy poté se objevily první detekční systémy postavené na CNN. Prvním zásadním krokem byla architektura R-CNN (Region-based CNN), která kombinovala selektivní vyhledávání regionů s klasifikací pomocí CNN. Vývoj pak pokračoval přes Fast R-CNN a Faster R-CNN, který jako první integroval návrh oblastí přímo do sítě [10, 12].

3.2.3 Nástup real-time detekce

Vedle dvoustupňových systémů vznikly také jednostupňové detektory, které dokážou v rámci jediného průchodu sítě předpovědět pozici i třídu objektů. K těm nejznámějším patří YOLO (You Only Look Once) a SSD (Single Shot Detector). Tyto přístupy přinesly významné zrychlení a umožnily detekci v reálném čase i na slabších zařízeních [4, 32].

3.2.4 Současné trendy

Moderní metody detekce dále zvyšují přesnost i efektivitu. Architektura EfficientDet spojuje síť pro extrakci příznaků s optimalizovanou hlavou pro detekci. Naopak přístupy jako DETR (Detection Transformer) využívají transformerové jádro a zcela opouštějí tradiční návrhy oblastí [13]. Tyto modely přinášejí lepší generalizaci, ale jsou výpočetně náročnější.

Vývoj metod detekce tak ukazuje snahu o nalezení rovnováhy mezi přesností a rychlostí – což je pro embedded aplikace klíčový faktor. V následujících částech se proto zaměříme na strukturu nejvýznamnějších typů detektorů a jejich vhodnost pro nasazení v praxi.

3.3 Architektury detektorů

Architektura detekčního modelu určuje způsob, jakým síť analyzuje obrazová data a jakým postupem předpovídá výskyt a polohu objektů. V posledních letech se ustálila základní typologie na dvoustupňové modely, jednostupňové modely a moderní transformerové přístupy, přičemž každý z těchto přístupů přináší vlastní výhody a kompromisy.

Dvoustupňové detektory využívají oddělený krok pro generování návrhů objektových oblastí (tzv. region proposals), které jsou následně klasifikovány a lokalizovány. Typickým představitelem této kategorie je Faster R-CNN, který vyniká vysokou přesností, ale je náročnější na výpočetní výkon a méně vhodný pro embedded zařízení [12].

Jednostupňové detektory jako YOLO nebo SSD naopak provádějí predikci tříd i polohy přímo v jednom kroku. Díky tomu dosahují výrazně vyšší rychlosti, a proto se staly preferovanou volbou v reálném čase i v omezených výpočetních prostředích [4, 32].

S rozvojem attention mechanismů se objevily i transformerové architektury, z nichž nejvýznamnější je DETR (Detection Transformer). Tento přístup opouští klasické principy návrhových oblastí a staví na globální pozornosti, čímž přináší zcela novou perspektivu do detekce objektů [13]. Přestože tyto modely nabízejí nové možnosti v oblasti generalizace, jejich využití v embedded prostředí je zatím omezené zejména z důvodu výpočetní náročnosti.

Každý z těchto přístupů je vhodný pro jiný typ aplikace. Při výběru modelu pro rozpoznávání recyklačních symbolů na embedded zařízení, jako je Raspberry Pi, je nutné

zvážit nejen přesnost, ale také rychlost inference, nároky na paměť a možnost kvantizace. Následující části se podrobněji věnují jednotlivým kategoriím detektorů a jejich vlastnostem.

3.3.1 Jednostupňové modely

Jednostupňové modely představují přístup, který výrazně zjednodušuje proces detekce objektů. Namísto oddělené fáze návrhů oblastí a následné klasifikace, jak je tomu u dvoustupňových modelů, predikují přítomnost objektu, jeho třídu i lokalizaci v rámci jednoho výpočetního průchodu neuronovou sítí. Tímto způsobem dosahují vyšší rychlosti a jsou vhodnější pro aplikace v reálném čase.

Nejznámějšími zástupci této kategorie jsou YOLO (You Only Look Once) a SSD (Single Shot MultiBox Detector). Oba tyto modely přinášejí výborný kompromis mezi přesností a výpočetní efektivitou, což z nich činí preferovanou volbu při detekci objektů na mobilních nebo embedded zařízeních [4, 32].

YOLO (You Only Look Once) rozděluje vstupní obraz na síť mřížky a pro každou její buňku predikuje ohraničující rámečky (bounding boxes) a pravděpodobnosti tříd. Díky tomuto přístupu dokáže zpracovávat obraz v reálném čase, i když rané verze měly tendenci trpět horší přesností při detekci menších objektů. Postupný vývoj přinesl výrazná zlepšení, zejména ve verzích YOLOv4 a YOLOv5, které zvýšily jak detekční přesnost, tak výkonnost. Novější verze, jako YOLOv8 až YOLOv12, přinášejí výrazná zlepšení v detekční přesnosti, rychlosti i optimalizaci pro nasazení na zařízení s omezeným výkonem. Díky tomu je YOLO vhodné pro široké spektrum aplikací včetně embedded systémů a průmyslového nasazení [4].

SSD se odlišuje tím, že používá více úrovní rozlišení pro detekci objektů různých velikostí. To znamená, že menší objekty se detekují ve vrstvách s vyšším rozlišením, zatímco větší objekty ve vrstvách hlubších. Díky tomuto principu dosahuje SSD velmi dobré rovnováhy mezi rychlostí a přesností, zejména ve scénářích s rozmanitou velikostí objektů [32].

Jednostupňové modely jsou ve srovnání s dvoustupňovými detektory méně přesné při lokalizaci v případech se silným překryvem objektů nebo v komplikovaných scénách. Přesto se jejich výkon i přesnost v posledních letech výrazně zlepšily a pro většinu praktických aplikací – včetně detekce recyklačních symbolů – představují optimální volbu [30].

Právě díky své efektivitě a jednoduchosti jsou tyto modely často základem systémů nasazovaných na výkonnostně omezených platformách, jako jsou mikropočítače nebo mobilní zařízení. V dalších kapitolách budou podrobněji analyzovány konkrétní modely z této kategorie, vhodné pro použití na Raspberry Pi.

3.3.2 Dvoustupňové modely

Dvoustupňové modely pro detekci objektů patří mezi první hluboké architektury, které výrazně překonaly tradiční přístupy založené na ručně navržených příznacích. Jejich hlavní výhodou je vysoká přesnost lokalizace a klasifikace, což je dáno odděleným zpracováním návrhu objektových oblastí a následné klasifikace.

Tento typ modelů funguje ve dvou krocích: nejprve se pomocí samostatného mechanismu vytvoří návrhy oblastí (region proposals), ve kterých by se mohly nacházet objekty. V druhém kroku je každá z těchto oblastí klasifikována a její přesné ohraničení upraveno pomocí regrese souřadnic. Tento přístup umožňuje síti soustředit se pouze na relevantní části obrazu, což vede k vyšší přesnosti oproti jednostupňovým metodám [10].

Jedním z prvních úspěšných modelů této kategorie byl R-CNN (Regions with CNN features), který kombinoval selektivní vyhledávání s hlubokou konvoluční sítí. Tento model však trpěl nízkou výpočetní efektivitou, neboť musel znovu extrahovat příznaky pro každou oblast zvlášť. Jeho nástupce Fast R-CNN tento problém vyřešil sdílením výpočtů, ale stále se spoléhal na externí generátor návrhů [10].

Skutečný průlom přinesla architektura Faster R-CNN, která do sítě integrovala tzv. Region Proposal Network (RPN) – samostatnou konvoluční vrstvu generující návrhy oblastí přímo během průchodu obrazem. Tento přístup výrazně zrychlil celý proces a stal se základem pro mnoho dalších pokročilých detekčních systémů [12].

Dvoustupňové modely nadále zůstávají standardem v oblastech, kde je klíčová přesnost – například v medicínské diagnostice, autonomních systémech nebo inspekčních úlohách. Jejich hlavní nevýhodou je však vyšší výpočetní náročnost a latence, což omezuje jejich nasazení v reálném čase a na embedded zařízeních s omezeným výkonem [30].

Z tohoto důvodu nejsou dvoustupňové modely zpravidla vhodné pro úlohy, jako je detekce recyklačních symbolů na zařízeních typu Raspberry Pi, kde je důležitá rychlost inference a nízká paměťová stopa. V těchto případech dávají větší smysl efektivnější jednostupňové přístupy.

3.3.3 Transformerové detektory

S rozvojem architektur založených na mechanismu pozornosti (attention) se i v oblasti detekce objektů začaly objevovat modely, které využívají principy známé z přirozeného jazykového zpracování. Transformerové detektory přinášejí nový pohled na strukturu detekčních modelů, neboť opouštějí tradiční postupy, jako je návrh objektových oblastí či použití pevných mřížek pro detekci.

Prvním výrazným zástupcem této třídy je model DETR (DEtection TRansformer), který kombinuje konvoluční backbone pro extrakci příznaků s transformerovým dekodérem pro predikci objektů. DETR zpracovává obraz jako posloupnost objektových dotazů (object queries) a pomocí globální pozornosti určuje jejich pozice a třídy [13].

Hlavní výhodou tohoto přístupu je, že model nemusí generovat návrhy oblastí, což značně zjednodušuje architekturu a eliminuje mnoho ručně nastavených parametrů. Transformerové detektory navíc lépe zvládají globální kontext a komplikované scény, protože každý vstupní bod má přístup k celému obrazu díky mechanismu self-attention [6, 18].

Na druhé straně je však nutné zmínit, že transformery jsou výpočetně velmi náročné, zejména při zpracování vysokého rozlišení obrazu. V původní podobě byl DETR výrazně pomalejší při konvergenci a méně efektivní než optimalizované CNN-based modely. Tyto nedostatky vedly k vývoji různých vylepšených verzí, například Deformable DETR, které kombinují výhody transformátorů s efektivnějším výběrem pozornosti.

Vzhledem k jejich výpočetním nárokům zatím transformerové detektory nepředstavují ideální volbu pro embedded zařízení. Přesto se jedná o významný vývojový směr s potenciálem do budoucna, zejména v kontextu systémů s dostatečně silným hardwarovým zázemím nebo akcelerací.

3.4 Modely vhodné pro embedded zařízení

Nasazení modelů hlubokého učení na embedded zařízení, jako je Raspberry Pi, představuje specifickou výzvu, která vyžaduje pečlivou rovnováhu mezi přesností, výpočetní efektivitou a velikostí modelu. V prostředí s omezenou pamětí, bez dedikovaných akceleratorů a s omezeným výkonem CPU a GPU je nezbytné volit architektury, které byly navrženy nebo optimalizovány s ohledem na tuto realitu.

Mezi nejčastěji využívané architektury pro embedded systémy patří SSD MobileNet a EfficientDet-Lite, případně jejich variace přizpůsobené pro frameworky jako TensorFlow Lite či PyTorch Mobile [25, 32]. Tyto modely jsou navrženy tak, aby minimalizovaly počet parametrů a výpočetních operací, přičemž si zachovávají přijatelnou úroveň přesnosti.

SSD MobileNet V1 a V2 využívají lehkou konvoluční architekturu MobileNet jako základní síť pro extrakci příznaků. Díky použití hloubkově separabilních konvolucí dochází k dramatickému snížení výpočetní náročnosti. Ve spojení s detekční hlavou SSD (Single Shot Detector) vzniká model, který je vhodný i pro inference na zařízeních bez GPU [32,33].

Podobným směrem se ubírá i rodina modelů EfficientDet, konkrétně varianty jako EfficientDet-Lite0 až Lite4, které byly navrženy přímo pro použití s TensorFlow Lite. Tyto modely kombinují výpočetně efektivní základ (EfficientNet) s optimalizovanou hlavou pro detekci, čímž dosahují vysokého výkonu i na mobilních a edge zařízeních [34].

Další zajímavou možností je použití modelu `ssdlite320_mobilenet_v3_large` dostupného v PyTorch, který rovněž nabízí vynikající poměr mezi rychlostí a přesností a je předtrénovaný na běžných detekčních datasetech (např. COCO). Tento model je podporován také pro export do TorchScript či ONNX, což usnadňuje jeho nasazení mimo desktopové prostředí [25].

4 METRIKY HODNOCENÍ MODELU

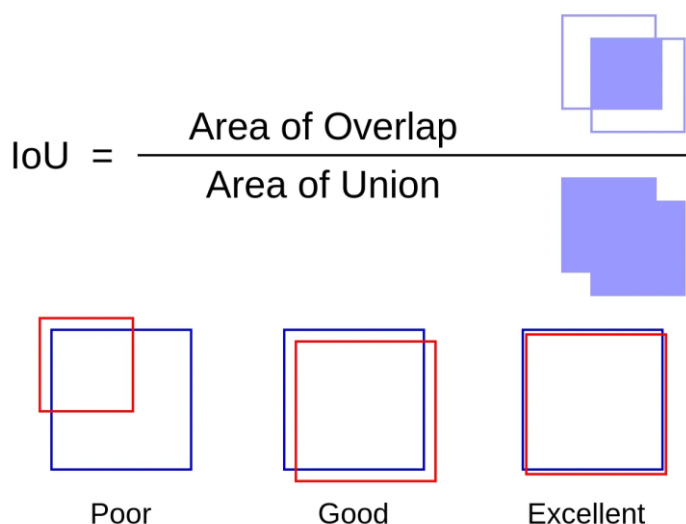
Ve vizuálních úlohách, jako je detekce objektů, se používá několik standardizovaných metrik, které kvantifikují přesnost, úplnost nebo efektivitu predikcí. Každá metrika reflektuje jiný aspekt kvality modelu, a proto je důležité jejich správné porozumění i vhodná interpretace.

Ve vizuální detekci se obvykle sleduje kvalita lokalizace objektů v obraze, úspěšnost zachycení všech relevantních instancí, ale také rychlost, s jakou je model schopen výsledky produkovat. Výběr a správná kombinace těchto metrik umožňuje smysluplně porovnat různé modely či konfigurace a posoudit jejich vhodnost pro konkrétní scénář použití.

4.1 Intersection over Union (IoU)

Intersection over Union (IoU) je základní metrika používaná pro hodnocení kvality lokalizace objektů v úlohách detekce. Její princip spočívá ve výpočtu poměru průniku a sjednocení dvou oblastí: predikované oblasti objektu a skutečné (ground truth) oblasti. Tento poměr je definován jako:

$$IoU = \frac{\text{Oblast průniku}}{\text{Oblast sjednocení}} = \frac{A \cap B}{A \cup B}$$



Obrázek 13: Intersection over Union

kde A představuje predikovanou bounding box oblast B odpovídající ground truth oblast. Výsledná hodnota IoU se pohybuje v intervalu $[0,1]$, přičemž 1 označuje naprosté překrytí a 0 žádné překrytí.

IoU se běžně používá jako rozhodovací kritérium pro klasifikaci predikce jako správné – například v případě, kdy je IoU větší než určitou prahovou hodnotu, obvykle 0,5 nebo 0,75. Tato prahová hodnota pak ovlivňuje výpočty dalších metrik, jako je Precision nebo Average Precision (AP) [31].

Tato metrika má své limity – například pokud jsou bounding boxy tvarově velmi rozdílné, IoU může podhodnocovat jinak vizuálně správné detekce. Přesto zůstává nepostradatelným nástrojem pro hodnocení přesnosti lokalizačních úloh a je základem pro pokročilejší metriky jako mAP (Mean Average Precision) [31].

4.2 Mean Average Precision (mAP)

Mean Average Precision (mAP) je jednou z nejdůležitějších metrik při hodnocení modelů pro detekci objektů, protože poskytuje ucelený přehled o jejich celkové přesnosti napříč třídami i prahovými hodnotami. Metrika mAP vychází ze základního principu výpočtu Average Precision (AP), což je plocha pod křivkou precision-recall. Tento přístup odměňuje modely, které nejen přesně lokalizují objekty, ale zároveň minimalizují počet falešně pozitivních i negativních predikcí.

Pro každou třídu se určí, jak přesně model predikoval daný objekt v různých prahových hodnotách IoU, a tyto hodnoty se následně zprůměrují. Konečný výsledek vyjadřuje střední přesnost modelu napříč všemi třídami. Běžně používaným standardem je například mAP@0.5, tedy průměrná přesnost pro detekce s $\text{IoU} \geq 0.5$. V pokročilejších hodnoceních, jako je COCO benchmark, se používá mAP@[.5:.95], tedy průměr přesností pro různé prahy od 0.5 do 0.95 v krocích po 0.05 [31].

Výhodou mAP je jeho schopnost vyváženě zohlednit jak přesnost (precision), tak úplnost (recall), což jej činí robustní metrikou pro porovnávání různých architektur modelů. Na druhou stranu, interpretace výsledků může být složitější, zejména pro laiky, a výpočet je náročnější než u jednodušších metrik, jako je pouze přesnost nebo recall.

4.3 Recall

Recall (česky úplnost) vyjadřuje schopnost modelu najít všechny relevantní objekty ve vstupním obraze. Je definován jako poměr mezi počtem správně detekovaných objektů a celkovým počtem skutečných (ground truth) objektů dané třídy:

$$Recall = \frac{TP}{TP + FN}$$

kde TP (true positives) označuje počet správně detekovaných objektů a FN (false negatives) počet objektů, které model nedetekoval.

Recall je klíčovou metrikou v situacích, kde je důležité minimalizovat počet zcela opomenutých objektů – například v aplikacích pro bezpečnost, medicínu nebo ekologii. V kontextu detekce recyklačních symbolů je vysoký recall důležitý pro to, aby žádný symbol nebyl přehlédnut, což by mohlo vést k nesprávnému třídění odpadu.

Jedním z důvodů, proč je recall často uváděn společně s metrikou precision, je skutečnost, že tyto dvě veličiny stojí v určitém napětí. Zvýšení recallu často vede ke snížení precision, protože více detekovaných objektů zahrnuje i falešně pozitivní výsledky. K pochopení celkové výkonnosti modelu je proto vhodné tyto dvě metriky kombinovat – například pomocí křivky precision-recall nebo výpočtem metriky mAP, jak bylo uvedeno dříve [31].

4.4 Frames Per Second (FPS)

Frames Per Second (FPS) je metrika, která udává, kolik snímků za sekundu je model schopen zpracovat při detekci objektů. Na rozdíl od metrik jako IoU, mAP či recall, které hodnotí přesnost detekce, FPS se zaměřuje na rychlost a výpočetní efektivitu modelu.

Vysoká hodnota FPS znamená, že model zvládá rychlé zpracování datového toku, což je klíčové pro aplikace s omezenými výpočetními zdroji nebo tam, kde je vyžadována okamžitá odezva – například v autonomních systémech nebo při vizuálním třídění odpadu na pásu. Naopak příliš nízká hodnota FPS může způsobit prodlevy v rozhodování nebo přetížení systému.

Je důležité zdůraznit, že mezi rychlostí a přesností často existuje kompromis. Modely s vyšší přesností bývají výpočetně náročnější a jejich FPS je nižší. Proto se při volbě modelu pro embedded prostředí obvykle hledá optimální rovnováha mezi rychlostí a kvalitou detekce, která odpovídá požadavkům konkrétní aplikace [33].

II PRAKTICKÁ ČÁST

5 FRAMEWORKY PRO HLUBOKÉ UČENÍ

Pro návrh detekčního systému recyklačních symbolů byly zvažovány dva hlavní frameworky pro hluboké učení – TensorFlow a PyTorch. Oba patří mezi nejpoužívanější nástroje v oblasti strojového učení a nabízejí širokou podporu pro trénování i nasazení neuronových sítí. V této práci byly oba frameworky využity v praxi – TensorFlow ve spojení s Object Detection API a PyTorch prostřednictvím modelu dostupného v knihovně TorchVision.

5.1 TensorFlow

TensorFlow je open-source framework vyvíjený společností Google, který patří mezi nejrozšířenější nástroje pro hluboké učení. Pro detekci objektů nabízí specializované rozšíření zvané Object Detection API, které výrazně zjednodušuje celý vývojový proces – od přípravy dat přes trénování až po export optimalizovaných modelů.

Součástí tohoto API je tzv. Model Zoo, tedy kolekce předtrénovaných detekčních modelů, které jsou připraveny pro použití s vlastními daty pomocí techniky transfer learningu. Pro aplikace na embedded zařízení jsou obzvláště vhodné lehké modely jako:

- EfficientDet-D0
- SSD MobileNet V2 FPNLite

Tyto modely nabízejí dobrý kompromis mezi rychlostí inference, přesností detekce a velikostí modelu.

Jednou z hlavních výhod TensorFlow Object Detection API je možnost snadného exportu trénovaného modelu do formátu TensorFlow Lite (TFLite), který umožňuje efektivní inference na zařízeních s omezeným výpočetním výkonem. Díky široké podpoře a ucelenému workflow se API ukázalo jako velmi vhodné řešení pro embedded nasazení v této práci.

5.2 PyTorch

PyTorch je open-source framework vyvíjený společností Meta AI (dříve Facebook), který si získal oblibu zejména ve výzkumné komunitě díky své flexibilitě, jednoduché syntaxi a dynamickému vykonávání výpočetního grafu. Nabízí široké možnosti pro návrh a trénování hlubokých neuronových sítí, přičemž jeho architektura výborně podporuje experimentální vývoj i vlastní přizpůsobení modelů.

V oblasti detekce objektů je PyTorch obvykle používán spolu s knihovnou TorchVision, která obsahuje několik předtrénovaných modelů, například Faster R-CNN, RetinaNet, SSD a SSDLite. Právě posledně jmenovaný model byl v rámci této práce využit, a to zejména kvůli své nízké výpočetní náročnosti a kompaktní architektuře vhodné pro embedded systémy.

Na rozdíl od TensorFlow však PyTorch neposkytuje jednotný framework pro celý pipeline detekce – od anotace dat přes konfiguraci až po export optimalizovaného modelu. Práce s TorchVision tak vyžaduje více manuálního nastavování, například při definici trénovacích smyček, zpracování vstupních dat či evaluaci výsledků. Export modelů pro embedded inference je možný prostřednictvím převodu do formátu ONNX, případně pomocí vlastních skriptů, což je technicky náročnější.

I přes tyto nevýhody je PyTorch v praxi velmi silným nástrojem, který dává uživateli velkou míru kontroly. V rámci této práce bylo jeho použití přínosné pro srovnání rozdílů v přístupu k detekci objektů mezi dvěma hlavními frameworky.

5.3 Srovnání TensorFlow a PyTorch

TensorFlow i PyTorch představují silné nástroje pro návrh a nasazení hlubokých neuronových sítí. Z hlediska detekce objektů a využití na embedded zařízeních se však jejich přístupy v několika ohledech liší. Tabulka níže shrnuje hlavní rozdíly v kontextu této práce:

Tabulka 2: Porovnání Tensorflow a PyTorch

Kritérium	TensorFlow (Object Detection API)	PyTorch (TorchVision)
Dostupnost detekčních modelů	Široká nabídka včetně EfficientDet, SSD, R-CNN	Omezenější, např. SSD, SSDLite, Faster R-CNN
Podpora workflow	Ucelený pipeline pro trénink, evaluaci i export	Vyžaduje ruční skriptování a konfiguraci
Export pro embedded inference	Podpora TFLite, EdgeTPU	ONNX nebo vlastní řešení, méně pohodlné

Vhodnost pro začátečníky	Vysoká díky hotovým šablonám a dokumentaci	Vyšší nároky na znalosti a úpravy
Flexibilita architektury	Standardní konfigurace v rámci API	Vyšší flexibilita díky ručnímu přístupu

(zdroj: vlastní)

Na základě těchto rozdílů byl pro hlavní část práce zvolen framework TensorFlow pro jeho přehlednost, dostupnost předtrénovaných modelů a jednoduchý export na embedded zařízení. PyTorch byl využit doplňkově pro porovnání přístupu a jako alternativa s jinou architekturou detekčního modelu.

6 IMPLEMENTACE HLUBOKÝCH SÍTÍ NA EMBEDDED ZAŘÍZENÍ

Tato kapitola se zabývá možnostmi nasazení detekčních neuronových sítí na embedded zařízení s omezeným výpočetním výkonem. Cílem je ověřit, jaké modely a techniky lze použít k zajištění efektivní detekce objektů přímo na zařízení bez nutnosti cloudového zpracování. Důraz je kladen na volbu vhodného hardwaru, optimalizaci modelů a přizpůsobení softwarového řešení pro konkrétní platformu.

6.1 Výzvy při nasazení na Raspberry Pi

Embedded zařízení, jako je Raspberry Pi, jsou pro nasazení hlubokých neuronových sítí atraktivní zejména díky své nízké ceně, kompaktním rozměrům a dostupnosti. Jejich využití však přináší řadu technických omezení, která je třeba při návrhu systému zohlednit.

Jedním z hlavních omezení je nízký výpočetní výkon ve srovnání s běžnými desktopovými nebo serverovými systémy. Raspberry Pi 5 sice přináší výrazné zlepšení oproti předchozím verzím (vyšší takt, podpora PCIe), ale stále nedisponuje dedikovaným GPU pro akceleraci výpočtů hlubokých sítí. To znamená, že většina výpočtů probíhá na CPU, což omezuje rychlost inference.

Dalším faktorem je omezená kapacita operační paměti a absence výkonného chlazení. Při delší zátěži může docházet k přehřívání a tzv. throttlingu procesoru, tedy automatickému snižování jeho výkonu s cílem zabránit přehřátí. Tento jev má za následek snížení frekvence procesoru, což negativně ovlivňuje plynulost běhu systému. Z tohoto důvodu je nutné pracovat s kompaktními modely, případně doplnit zařízení o pasivní nebo aktivní chlazení.

Pro reálné nasazení bylo rovněž důležité zajistit dostatečnou kvalitu vstupních dat. Raspberry Pi 5 bylo v této práci využito spolu s oficiálním HQ kamerovým modulem, který poskytuje vysoké rozlišení a lepší kontrolu expozice.

Výše uvedené faktory vedly k výběru modelů, které jsou navrženy s důrazem na nízkou výpočetní náročnost, a současně k aplikaci optimalizačních technik pro zajištění použitelného výkonu přímo na embedded zařízení.

6.2 Optimalizace modelů

Aby bylo možné nasadit detekční modely na zařízení s omezeným výpočetním výkonem, je nutné provést jejich optimalizaci. Ta zahrnuje úpravy modelů tak, aby měly menší paměťovou a výpočetní náročnost, a byly schopné inference v reálném čase nebo s minimálním zpožděním.

V případě modelů trénovaných v TensorFlow byl využit nástroj TensorFlow Lite (TFLite), který umožňuje export modelu do kompaktního formátu vhodného pro nasazení na embedded platformy, jako je Raspberry Pi. Součástí tohoto procesu byla také kvantizace, tedy převod vah a operací z 32bitového formátu (FP32) na menší reprezentaci (např. INT8 nebo FP16), čímž došlo ke snížení velikosti modelu a zrychlení výpočtů.

Pro model vytvořený v PyTorchu nebylo možné využít přímý export do optimalizovaného formátu jako v případě TFLite. Namísto toho byl použit export do formátu ONNX (Open Neural Network Exchange), který slouží jako meziformát pro přenos modelů mezi různými frameworky. Výsledný ONNX model byl následně nasazen pomocí knihovny ONNX Runtime, která umožňuje spouštění inference bez potřeby plnohodnotného PyTorch prostředí. Tento postup však vyžaduje více manuálních kroků, například úpravu vstupních dimenzí nebo kontrolu kompatibility jednotlivých vrstev.

Optimalizace modelů tak hrála klíčovou roli v tom, aby bylo možné realizovat inference na zařízení typu Raspberry Pi bez externí výpočetní podpory a při zachování dostatečné rychlosti detekce.

6.3 Využitý hardware

Pro nasazení detekčního systému bylo využito embedded zařízení Raspberry Pi 5, které oproti předchozím verzím nabízí vyšší výpočetní výkon, větší paměťovou kapacitu a modernizované rozhraní pro připojení periférií. Díky těmto vlastnostem se jedná o vhodnou platformu pro experimenty s inference hlubokých neuronových sítí v reálném čase.

7 DATASET

Datová sada představuje základní komponentu každého úkolu v oblasti počítačového vidění. Kvalita, rozmanitost a přesnost anotací datasetu mají přímý vliv na výkonnost a generalizační schopnost modelu. V rámci této diplomové práce bylo nutné pracovat s daty, která pokrývají širokou škálu recyklačních symbolů napříč různými materiálovými kategoriemi.

7.1 Dostupné datasety

Pro trénování detekčního modelu byl zvolen veřejně dostupný dataset GreenUP, který je dostupný na platformě Roboflow Universe. Tento dataset obsahuje obrázky s anotacemi různých recyklačních symbolů používaných k označování materiálového složení obalů.



Obrázek 16: Dataset [47]

Výběr tohoto datasetu byl motivován absencí alternativ, které by pokrývaly srovnatelně široké spektrum tříd. Většina volně dostupných datasetů se zaměřuje pouze na vybrané skupiny symbolů (například jen plastové obaly) nebo obsahuje jen obecné ikony bez podrobného rozlišení podle kódu materiálu. Dataset GreenUP oproti tomu nabízí rozsáhlý soubor tříd, včetně méně běžných označení pro sklo, textil, kovové obaly či kompozitní materiály.

Dataset byl dále manuálně upraven a doplněn (viz podkapitola 7.3), aby odpovídal potřebám této práce a zajišťoval dostatečnou kvalitu anotací.

7.2 Struktura datasetu

Dataset použitý v této práci je rozdělen do tří částí:

- Trénovací množina: 2 877 obrázků
- Validací množina: 596 obrázků
- Testovací množina: 267 obrázků

Celkem tedy dataset obsahuje 3 740 anotovaných obrázků, přičemž každý z nich může obsahovat jeden nebo více recyklačních symbolů. Obrázky jsou ve formátu JPEG a reprezentují symboly v různých podmínkách — například s různým osvětlením, perspektivou nebo částečným zakrytím.

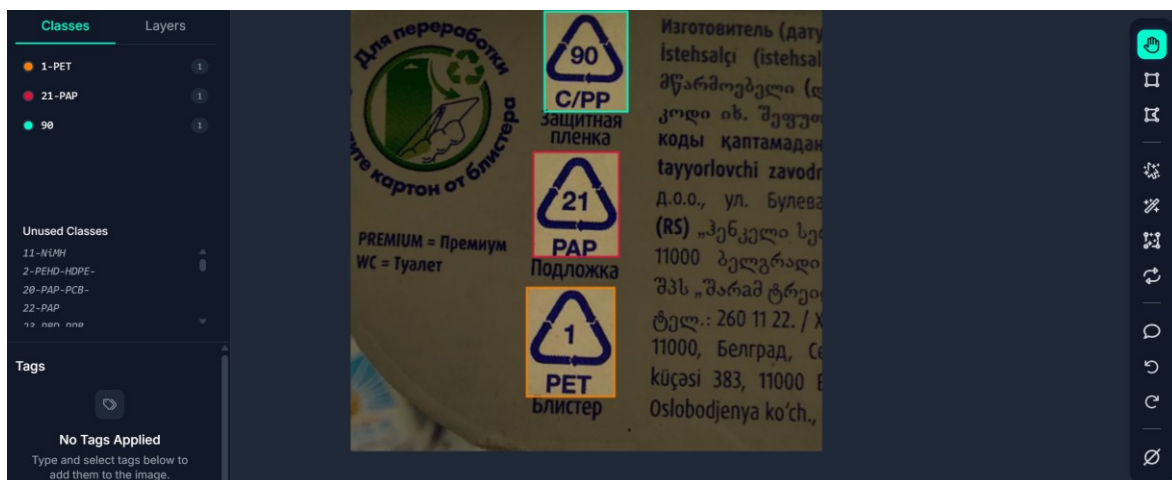
Dataset obsahuje celkem 46 tříd, které odpovídají recyklačním značkám používaným k označení materiálového složení obalů. Mezi třídy patří například:

"1-PET", "21-PAP", "41-ALU", "7-O-OTHER-", "C-ALU", "unclassified"

Některé třídy zahrnují i kombinované materiály nebo skupiny symbolů, jako např. 42-49, C-HDPE nebo 9-19-Alkaline.

7.3 Anotace datasetu

Přestože byl dataset RecycleGreenUP k dispozici veřejně, nebyly všechny anotace zcela přesné nebo kompletní. V rámci přípravy dat pro trénování modelu bylo proto nutné provést ruční úpravy a doplnění. Tyto úpravy probíhaly přímo ve webovém rozhraní Roboflow, které umožňuje pohodlnou správu datasetu i editaci bounding boxů.



Obrázek 17: Editační prostředí (zdroj: vlastní)

Konkrétní úpravy zahrnovaly:

- korekci pozice a velikosti ohraničujících boxů, které byly nepřesně umístěné,
- doplnění chybějících anotací v obrázcích, kde symboly zjevně chyběly,
- odstranění případných chyb

Roboflow ukládá anotace v interním JSON formátu, kde jsou souřadnice objektů vyjádřeny v absolutních hodnotách vzhledem k velikosti obrázku (pixels). Při exportu lze ale vybrat výstupní formát podle potřeby — například YOLO, COCO JSON, Pascal VOC nebo TFRecord.

8 DETEKČNÍ MODELÝ A JEJICH IMPLEMENTACE

V této kapitole jsou popsány konkrétní modely použité pro rozpoznávání recyklačních symbolů na obalech. Cílem bylo navrhnout a otestovat takové architektury, které budou schopné spolehlivě detekovat malé objekty v různorodém prostředí, a současně budou vhodné pro nasazení na zařízení s omezeným výpočetním výkonem.

Pro potřeby této práce byly zvoleny následující tři modely:

- SSDLite320 MobileNet V3 – extrémně kompaktní model zaměřený na inference v reálném čase, implementován v prostředí PyTorch.
- SSD MobileNet V2 FPNLite – lehká síť s využitím feature pyramid network pro lepší detekci menších objektů, implementována pomocí TensorFlow Object Detection API.
- EfficientDet D0–D7 – škálovatelná architektura s důrazem na vyváženost mezi přesností a výpočetní náročností, také implementována v TensorFlow.

Každý z těchto modelů byl dále upraven pro daný dataset a přizpůsoben konkrétnímu počtu tříd. Použitá implementační prostředí a knihovny odrážejí běžné vývojové workflow: zatímco TensorFlow Object Detection API poskytuje komplexní nástroje pro vývoj a trénování modelů, PyTorch umožňuje větší flexibilitu při přímé manipulaci s architekturou modelu.

8.1 SSDLite320 MobileNet V3

Model SSDLite320 MobileNet V3 patří mezi lehké detektory objektů určené pro nasazení v prostředí s omezeným výpočetním výkonem. V této části je popsán způsob jeho nasazení v rámci úlohy rozpoznávání recyklačních symbolů. Dále jsou rozvedeny důvody jeho výběru, základní architektonické vlastnosti a způsob implementace pomocí knihovny PyTorch.

8.1.1 Architektura a charakteristika modelu

Model SSDLite320 MobileNet V3 využívá detekční architekturu SSDLite, která je zjednodušenou a výpočetně úspornou verzí klasického SSD (Single Shot Detector). Jedná se o jednostupňový model, který lokalizuje objekty a zároveň je klasifikuje v rámci jednoho průchodu neuronovou sítí. To umožňuje rychlou inferenci bez potřeby generování návrhových oblastí (region proposals), jak je tomu například u dvoustupňových modelů typu Faster R-CNN.

Základem sítě je páteří model MobileNet V3 Large, navržený speciálně pro mobilní a embedded zařízení. Tato architektura využívá tzv. hloubkově separabilní konvoluce (depthwise separable convolutions), které rozdělují klasickou konvoluci na dvě jednodušší operace: nejprve zpracují každý kanál vstupu samostatně (depthwise konvoluce) a následně mezi nimi provádějí kombinaci pomocí 1×1 filtrů (pointwise konvoluce). Tento přístup výrazně snižuje výpočetní náročnost oproti běžné konvoluci, při zachování schopnosti extrahovat prostorové i kanálové informace [36].

Dalším prvkem architektury jsou tzv. squeeze-and-excitation bloky, které umožňují dynamicky upravit důležitost jednotlivých kanálů na základě jejich globální aktivity. Tyto bloky zlepšují reprezentaci obrazu tím, že umožňují síti zaměřit se na nejvýznamnější rysy vstupu.

Aktivační funkcí použitou v MobileNet V3 je hard-swish, což je výpočetně efektivní aproximace funkce swish. Na rozdíl od tradičních funkcí jako ReLU či sigmoid přináší hard-swish lepší gradientový tok a vede k vyšší přesnosti bez výrazného navýšení náročnosti.

Výstupní detekční část modelu (tzv. detekční hlava) generuje predikce na několika úrovních rozlišení, přičemž pro každou výstupní mapu jsou vypočítány ohraničující boxy (bounding boxes) a odpovídající skóre pro jednotlivé třídy. Na rozdíl od složitějších architektur, které využívají mechanismy pro zpracování více měřítek (například Feature Pyramid Network), spoléhá SSDLite na pečlivě optimalizované vrstvy s nižší výpočetní zátěží.

Celkově je architektura navržena tak, aby umožňovala použití i v podmínkách, kde není k dispozici dedikované GPU. Díky tomu je možné ji efektivně provozovat i na zařízení typu Raspberry Pi bez výrazného omezení přesnosti detekce menších objektů.

8.1.2 Implementace v PyTorch

Model SSDLite320 MobileNet V3 byl implementován v prostředí PyTorch pomocí modulu `torchvision.models.detection`. Byla využita předtrénovaná varianta modelu `ssdlite320_mobilenet_v3_large`, kterou bylo možné dále upravit podle specifik datasetu.

Architektonická úprava spočívala v nahrazení výstupní klasifikační hlavy, a to tak, aby odpovídala počtu tříd obsažených v datasetu. Pro úpravu výstupních vrstev byl použit následující úsek kódu:

```

num_classes = len(train_dataset.coco.cats) + 1 # +1 pro pozadí
old_head = model.head.classification_head
in_channels = [layer[0][0].in_channels for layer in old_head.module_list]
num_anchors = [old_head.module_list[0][1].out_channels // old_head.num_columns] * len(in_channels)
model.head.classification_head = SSDLiteClassificationHead(in_channels, num_anchors, num_classes, None)

```

Obrázek 18: Načtení a úprava modelu SSDLite (zdroj: vlastní)

Datová sada byla ve formátu COCO a načítána pomocí CocoDetection. Pro správné zpracování anotací bylo nutné definovat vlastní funkci `collate_fn`, která převáděla anotace z formátu `[x, y, width, height]` do formátu `[x_min, y_min, x_max, y_max]` a doplňovala je o informace o třídách:

```

def collate_fn(batch):
    images, targets = [], []
    for img, ann in batch:
        boxes = [[x, y, x + w, y + h] for (x, y, w, h) in [obj['bbox'] for obj in ann]]
        labels = [obj['category_id'] + 1 for obj in ann]
        images.append(img)
        targets.append({
            "boxes": torch.as_tensor(boxes, dtype=torch.float32),
            "labels": torch.as_tensor(labels, dtype=torch.int64),
            "image_id": torch.tensor([ann[0]['image_id']]) if ann else -1
        })
    return images, targets

```

Obrázek 19: Úprava formátu pro COCO (zdroj: vlastní)

Model byl trénován po dobu 100 epoch s využitím optimalizátoru SGD a plánovače rychlosti učení založeného na kombinaci zahřívací fáze (warmup) a kosinového rozpadu:

```

def warmup_cosine_decay(epoch, warmup_epochs=10, total_epochs=100, base_lr=0.4, warmup_lr=0.13333):
    if epoch < warmup_epochs:
        return warmup_lr + (base_lr - warmup_lr) * (epoch / warmup_epochs)
    return base_lr * (0.5 * (1 + math.cos(math.pi * (epoch - warmup_epochs) / (total_epochs - warmup_epochs))))

```

Obrázek 20: Implementace warmup cosine decay (zdroj: vlastní)

Celý trénovací proces probíhal v klasické smyčce přes jednotlivé epochy. V každé epoše byl model nejprve trénován (`model.train()`), přičemž pro každou dávku dat byl spočítán výstup modelu, následně ztrátová funkce a proveden zpětný průchod:

```
optimizer.zero_grad()  
loss_dict = model(images, targets)  
loss = sum(loss for loss in loss_dict.values())  
loss.backward()  
optimizer.step()
```

Obrázek 21: Výpočet a zpětný průchod (zdroj: vlastní)

Po dokončení trénování na trénovací množině následovala fáze vyhodnocení (`model.eval()`), ve které byly predikce modelu porovnány s validačními anotacemi pomocí knihovny `pycocotools`. Vyhodnocení zahrnovalo výpočet standardních metrik (`AP@[0.50:0.95]`, `AP@0.50`, `AP@0.75`, `AR`) pomocí `COCOeval`:

```
coco_eval = COCOeval(coco_gt, coco_dt, "bbox")  
coco_eval.evaluate()  
coco_eval.accumulate()  
coco_eval.summarize()
```

Obrázek 22: Vyhodnocení SSDLite (zdroj: vlastní)

Po každé epoše byly metriky zaznamenány pomocí `TensorBoardu` a v případě zlepšení hlavní metriky `AP` byl model uložen jako „nejlepší“. Po dokončení všech epoch byl navíc exportován finální model ve formátu `ssdlite320_mobilenet_v3_final.pth`, určený pro další nasazení.

8.2 SSD MobileNet V2 FPNLite

Model SSD MobileNet V2 FPNLite představuje jednu z dalších testovaných architektur pro detekci recyklačních symbolů. V této části bude popsána jeho struktura, způsob implementace a přizpůsobení na vlastní dataset pomocí knihovny `TensorFlow Object Detection API`.

8.2.1 Architektura a charakteristika modelu

SSD MobileNet V2 FPNLite je detekční model navržený pro použití v prostředí s omezeným výpočetním výkonem. Kombinuje efektivní páteří síť `MobileNet V2` s

detektorem objektů SSD (Single Shot Detector), který je rozšířen o modul FPN – Feature Pyramid Network.

MobileNet V2 je síť určená pro mobilní a embedded zařízení. Využívá tzv. inverted residual bloky, ve kterých se nejprve rozšíří počet kanálů (feature maps), poté se aplikují výpočetně úsporné hloubkově separabilní konvoluce, a nakonec se výstup znovu zúží a propojí s původním vstupem pomocí přímého spojení (skip connection). Tento návrh snižuje počet parametrů a zároveň zajišťuje, že síť neztrácí důležité informace ze vstupu.

Feature Pyramid Network (FPN) je mechanismus, který umožňuje síti lépe zachytit objekty různých velikostí. Využívá výstupy z různých úrovní hloubky neuronové sítě a kombinuje je tak, aby výsledná detekce pokrývala jak malé, tak větší objekty v obraze. To je zvláště užitečné v případech, kdy se v obraze nacházejí drobné symboly, jako jsou recyklační piktogramy.

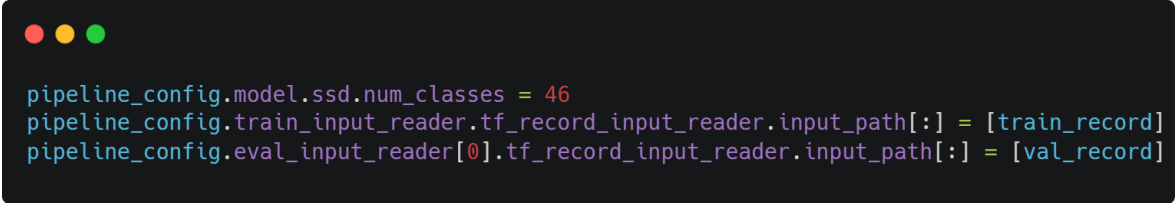
Výstupní část modelu SSD generuje pro každou oblast v obraze množinu kandidátních objektů (tzv. anchor boxy) a následně pro každý z nich predikuje pozici ohraničujícího boxu a pravděpodobnost příslušnosti ke konkrétní třídě.

Model pracuje s pevně nastaveným rozlišením vstupního obrazu 320×320 pixelů. Díky tomu a použitým architektonickým zjednodušením je vhodný pro běh i na méně výkonném hardwaru bez výrazné ztráty přesnosti.

8.2.2 Implementace v TensorFlow Object Detection API

Model SSD MobileNet V2 FPNLite byl implementován pomocí knihovny TensorFlow Object Detection API. Jako výchozí byl použit model `ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8`, který obsahuje připravenou architekturu, konfigurační soubory a předtrénované váhy.

Základní konfigurace trénování byla definována v souboru `pipeline.config`. Část nejčastěji upravovaných parametrů (například počet tříd, cesty k datasetům nebo velikost dávky) byla změněna automaticky pomocí Python skriptu využívajícího `pipeline_pb2`. Tento skript sloužil pro rychlé přenastavení opakujících se hodnot:



```
pipeline_config.model.ssd.num_classes = 46
pipeline_config.train_input_reader.tf_record_input_reader.input_path[:] = [train_record]
pipeline_config.eval_input_reader[0].tf_record_input_reader.input_path[:] = [val_record]
```

Obrázek 23: Úprava pipeline (*zdroj: vlastní*)

Pro podrobnější a rozsáhlejší konfiguraci – například nastavení typu optimalizátoru, schématu učení, typu augmentací, konfigurace FPN nebo generování anchor boxů – mohou být provedeny úpravou přímo v souboru `pipeline.config`. Tento přístup umožňuje využít kompletní možnosti, které TensorFlow Object Detection API nabízí.

Soubor `pipeline.config` obsahuje několik hlavních částí:

- `model`: definuje typ detektoru (SSD), páteřní síť (MobileNet V2), zapnutí FPN, počet tříd a další architektonické prvky.
- `train_config`: určuje parametry trénování, jako počet kroků, velikost dávky, optimalizátor a metody augmentace.
- `train_input_reader` / `eval_input_reader`: obsahují cesty k trénovacím a validačním datům ve formátu TFRecord a k label mapě.
- `eval_config`: nastavuje metriky vyhodnocení, v tomto případě `coco_detection_metrics`.

Optimalizace byla nastavena pomocí optimalizátoru se setrvačností `momentum = 0.9`, přičemž rychlost učení byla řízena funkcí cosine decay doplněnou o počáteční zahřívací fázi (`warmup`). Toto nastavení bylo přímo součástí konfiguračního souboru:

```
optimizer {  
  momentum_optimizer {  
    learning_rate {  
      cosine_decay_learning_rate {  
        learning_rate_base: 0.08  
        total_steps: 50000  
        warmup_learning_rate: 0.026666  
        warmup_steps: 1000  
      }  
    }  
    momentum_optimizer_value: 0.9  
  }  
}
```

Obrázek 24: Optimizer SSD MobileNet V2 FPNLite (zdroj: vlastní)

Během trénování byly aplikovány základní metody augmentace obrazu – horizontální převrácení a náhodné ořezávání. Tyto operace byly specifikovány přímo v konfiguračním souboru:

```
data_augmentation_options {  
  random_horizontal_flip {  
  }  
}  
data_augmentation_options {  
  random_crop_image {  
    min_object_covered: 0.0  
    min_aspect_ratio: 0.75  
    max_aspect_ratio: 3.0  
    min_area: 0.75  
    max_area: 1.0  
    overlap_thresh: 0.0  
  }  
}
```

Obrázek 25: Augmentace vstupních dat SSD MobileNet V2 FPNLite (zdroj: vlastní)

Při trénování modelu byl spuštěn samostatný proces vyhodnocení (evaluace) pomocí následujícího příkazu:



```
CUDA_VISIBLE_DEVICES=-1 python models/research/object_detection/model_main_tf2.py \
--model_dir=<path_to_model> \
--pipeline_config_path=<path_to_pipeline> \
--checkpoint_dir=<path_to_checkpoint_dir> \
--alsologtostderr
```

Obrázek 26: Evaluace Tensorflow Object Detection API (*zdroj: vlastní*)

Evaluace byla spuštěna bez využití GPU (`CUDA_VISIBLE_DEVICES=-1`) a probíhala nad validační množinou. Použity byly metriky COCO (`coco_detection_metrics`), které poskytují přesnost napříč různými prahy překrytí (IoU), včetně hlavní metriky `AP@[0.50:0.95]`.

Po dokončení tréninku a vyhodnocení byl model uložen ve formě checkpointů a připraven k následnému exportu a nasazení.

8.3 EfficientDet D0

EfficientDet-D0 je kompaktní detekční model navržený pro efektivní detekci objektů při nízké výpočetní náročnosti. Vychází z architektury EfficientDet, která kombinuje páteřní síť EfficientNet s rozšířením výstupních vrstev pomocí BiFPN (Bidirectional Feature Pyramid Network).

8.3.1 Architektura a charakteristika modelu

EfficientDet-D0 je detekční model navržený s důrazem na vyvážení přesnosti a efektivity. Využívá páteřní síť EfficientNet-B0, která je optimalizována pro výpočetně úsporné zpracování obrazu pomocí principu compound scaling – tedy souběžného škálování šířky sítě, hloubky a rozlišení vstupu. Na rozdíl od klasických CNN, kde jsou tyto parametry laděny nezávisle, EfficientNet kombinuje všechny tři v rámci jednoho vzorce a dosahuje tak lepší efektivity při menším počtu parametrů.

Detekční část modelu je založena na architektuře SSD (Single Shot Detector), která umožňuje detekci objektů v jediném průchodu sítí bez potřeby generování návrhových oblastí. Výstupy z různých úrovní páteřní sítě jsou dále zpracovávány pomocí BiFPN (Bidirectional Feature Pyramid Network) – rozšířené verze klasické FPN, která kombinuje informace z různých měřítek v obou směrech (shora dolů i zdola nahoru). Díky tomu je síť schopna lépe zachytit i menší objekty a jejich kontext.

Jako aktivační funkce je v EfficientDet použita Swish, která na rozdíl od běžně používané ReLU umožňuje plynulejší průchod gradientu a podporuje lepší konvergenci. Kromě toho jsou všechny konvoluční vrstvy normalizovány pomocí batch normalization, což pomáhá stabilizovat trénink.

Výstupní detekční hlava generuje pro každý vstupní obrázek sadu návrhových boxů (anchor boxes), ke kterým predikuje pozice a skóre jednotlivých tříd. Vstupní rozměr modelu byl v této konfiguraci nastaven na 512×512 pixelů, přičemž poměr stran obrazu je zachován a nedostatečně zaplněné oblasti jsou doplněny pomocí paddingu.

Díky svému návrhu je EfficientDet-D0 vhodný pro běh i na méně výkonném hardwaru, a přitom si zachovává konkurenceschopnou přesnost detekce.

8.3.2 Implementace v TensorFlow Object Detection API

Implementace modelu EfficientDet-D0 probíhala obdobně jako u předchozích modelů s využitím knihovny TensorFlow Object Detection API. Výchozím modelem byl `efficientdet_d0_coco17_tpu-32`, který byl stažen a upraven pomocí automatizačního skriptu. Skript prováděl základní úpravy v konfiguračním souboru – nastavení počtu tříd, cest k datům, velikosti dávky a načtení checkpointu.

Hlavní rozdíly oproti dříve použitým modelům spočívaly v rozsahu architektury a datové přípravě. Vstupní velikost obrazu byla nastavena na 512×512 pixelů s doplněním pomocí paddingu. Model využíval páteřní síť EfficientNet-B0 a vícenásobné propojení vrstev pomocí tří iterací BiFPN.

Optimalizace byla nastavena pomocí momentového optimalizátoru s cosine decay a warmupem. Doba trénování byla výrazně delší – celkem 300 000 kroků s dávkou o velikosti 8:

```
optimizer {  
  momentum_optimizer {  
    learning_rate {  
      cosine_decay_learning_rate {  
        learning_rate_base: 0.08  
        total_steps: 300000  
        warmup_learning_rate: 0.001  
        warmup_steps: 2500  
      }  
    }  
    momentum_optimizer_value: 0.9  
  }  
}
```

Obrázek 27: Optimizer EfficientDet-D0 (zdroj: vlastní)

Datová augmentace kombinovala horizontální převrácení s pokročilejší transformací `random_scale_crop_and_pad_to_square`, která umožňuje náhodné škálování objektů v obraze a jejich zarovnání na výstupní velikost:

```
data_augmentation_options {  
  random_horizontal_flip {  
  }  
}  
data_augmentation_options {  
  random_scale_crop_and_pad_to_square {  
    output_size: 512  
    scale_min: 0.1  
    scale_max: 2.0  
  }  
}
```

Obrázek 28: Augmentace vstupních dat EfficientDet-D0 (zdroj: vlastní)

Evaluaace modelu probíhala souběžně během trénování na CPU, stejně jako v předchozích případech. Pro měření výkonu byly použity metriky `coco_detection_metrics`.

Po dokončení tréninku byl model uložen jako sada checkpointů a připraven pro export do formátu vhodného pro nasazení.

9 NAsazení na RASPBERRY PI

Tato kapitola se zaměřuje na proces nasazení detekčních modelů na cílové zařízení. V návaznosti na předchozí fáze vývoje, kdy došlo k tréninku a implementaci modelů, bylo nutné připravit a spustit jednotlivé varianty v prostředí s omezenými výpočetními prostředky.

Popisovány jsou konkrétní kroky potřebné pro převod modelů do vhodného formátu, jejich úprava pro běh na zařízení a následná implementace skriptů zajišťujících inferenci v reálném čase.

9.1 Nasazení modelů TensorFlow Lite

Pro nasazení modelů vytvořených v prostředí TensorFlow byla využita platforma TensorFlow Lite, která je určena pro inference na zařízeních s omezeným výpočetním výkonem. Modely byly převedeny do formátu .tflite a následně spuštěny přímo na zařízení.

9.1.1 Převod modelu SSD MobileNet V2 FPNLite

Model `ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8` bylo možné převést do formátu TensorFlow Lite bez jakýchkoli úprav. Nejprve byl pomocí nástroje `export_tflite_graph_tf2.py`, který je součástí balíčku TensorFlow Object Detection API, vytvořen adresář se strukturovaným modelem ve formátu SavedModel:



```
python object_detection/export_tflite_graph_tf2.py \
  --pipeline_config_path <path_to_pipeline> \
  --trained_checkpoint_dir <path_to_checkpoints> \
  --output_directory <path_to_export_dir>
```

Obrázek 29: Příkaz na exportování modelu (*zdroj: vlastní*)

Následně byl model převeden do formátu .tflite pomocí následujícího kódu v Pythonu:

```
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model("exported-model/saved_model")
converter.target_spec.supported_ops = [
    tf.lite.OpsSet.TFLITE_BUILTINS,
    tf.lite.OpsSet.SELECT_TF_OPS
]
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

with open("model.tflite", "wb") as f:
    f.write(tflite_model)
```

Obrázek 30: Převod na TFLite (zdroj: vlastní)

Tento skript nejprve načte model ve formátu SavedModel, nastaví sadu podporovaných operací včetně rozšířených TensorFlow operací a aktivuje výchozí optimalizační režim. Výsledkem je převedený model ve formátu .tflite, který je uložen do souboru připraveného k nasazení.

9.1.2 Převod modelu EfficientDet D0

U modelu `efficientdet_d0_coco17_tpu-32` bylo nutné upravit výchozí konfiguraci `pipeline.config`, která obsahovala prvek `keep_aspect_ratio_resizer`. Tento způsob změny velikosti vstupního obrazu není kompatibilní s TensorFlow Lite. Pro úspěšný převod byl nahrazen resizerem `fixed_shape_resizer`.

Po této úpravě bylo možné model exportovat i převést obdobným způsobem jako v případě SSD MobileNet V2.

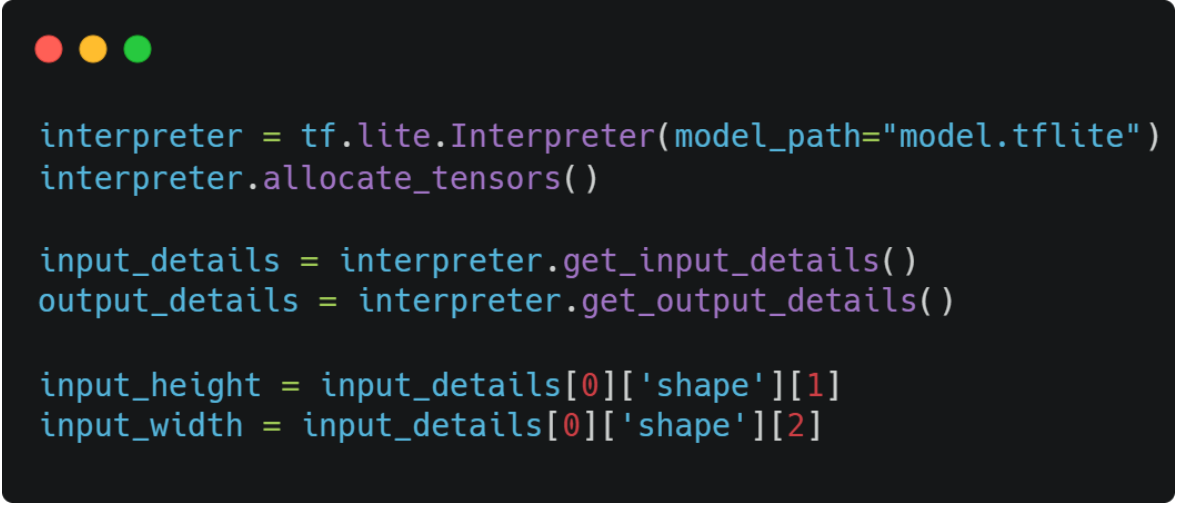
9.1.3 Spuštění inference na Raspberry Pi

Na Raspberry Pi byly použity samostatné inferenční skripty pro každý z modelů, implementované v jazyce Python. Skript využívá tyto knihovny:

- `tensorflow / tf.lite_runtime` – načítání a spouštění modelu ve formátu .tflite,
- `opencv (cv2)` – zpracování a zobrazení obrazu, převod barev, vykreslování detekcí a FPS,
- `numpy` – práce s poli a příprava vstupních dat (např. normalizace a rozměry),

- picamera2 – získávání aktuálního obrazu z kamery Raspberry Pi v požadovaném rozlišení.

Po načtení potřebných knihoven došlo k inicializaci modelu a interpreta:



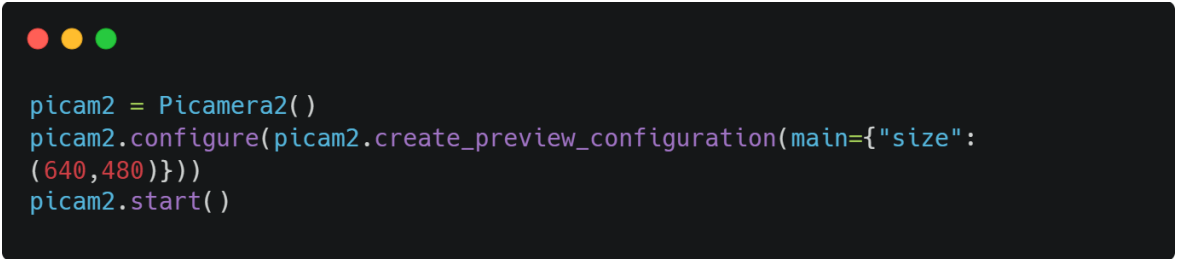
```
interpreter = tf.lite.Interpreter(model_path="model.tflite")
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

input_height = input_details[0]['shape'][1]
input_width = input_details[0]['shape'][2]
```

Obrázek 31: Načtení TFLite modelu (zdroj: vlastní)

Model je načten z uloženého .tflite souboru, inicializuje se a interpreter zjistí vstupní rozměry, které očekává síť (např. 320×320 pixelů). Následně se připraví kamera:



```
picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration(main={"size":
(640,480)}))
picam2.start()
```

Obrázek 32: Načtení kamery (zdroj: vlastní)

Kamera je nastavena na rozlišení 640×480 pixelů a spuštěna. Po této přípravě se spustí hlavní smyčka běžící v reálném čase. V každém kroku se nejprve získá aktuální snímek, který je dále upraven pro vstup do modelu:



```
frame = picam2.capture_array()
image = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
image_resized = cv2.resize(image, (input_width, input_height))
input_data = np.expand_dims(image_resized, axis=0).astype(np.float32) / 255.0
```

Obrázek 33: Načtení a úprava vstupu (zdroj: vlastní)

Obraz z kamery je převeden do barevného prostoru RGB, upraven na požadovanou velikost a normalizován na hodnoty 0–1. Poté je vložen do modelu a spuštěna inference:

```
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()

scores = interpreter.get_tensor(output_details[0]['index'])[0]
boxes = interpreter.get_tensor(output_details[1]['index'])[0]
classes = interpreter.get_tensor(output_details[3]['index'])
[0]
```

Obrázek 34: Předání vstupu do modelu (zdroj: vlastní)

Model vrátí detekční skóre, souřadnice boxů a indexy detekovaných tříd. Následuje zpracování výsledků, kde se pro každou detekci s dostatečnou jistotou vykreslí do obrazu rámeček a textový popis, který reprezentuje typ odpadu:

```
for i in range(len(scores)):
    if scores[i] >= 0.5:
        ymin, xmin, ymax, xmax = boxes[i]
        x1, y1 = int(xmin * 640), int(ymin * 480)
        x2, y2 = int(xmax * 640), int(ymax * 480)
        label = "plastic" # zjednodušený příklad
        cv2.rectangle(frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
        cv2.putText(frame, f"{label}: {scores[i]:.2f}", (x1, y1 - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
```

Obrázek 35: Vykreslení detekce (zdroj: vlastní)

Tento skript představuje kompletní implementaci inference v reálném čase, od snímání až po vizuální výstup. Díky využití TensorFlow Lite a přímé optimalizaci modelu byl systém schopen běžet samostatně na Raspberry Pi bez nutnosti dalšího hardwaru nebo externí výpočetní podpory.

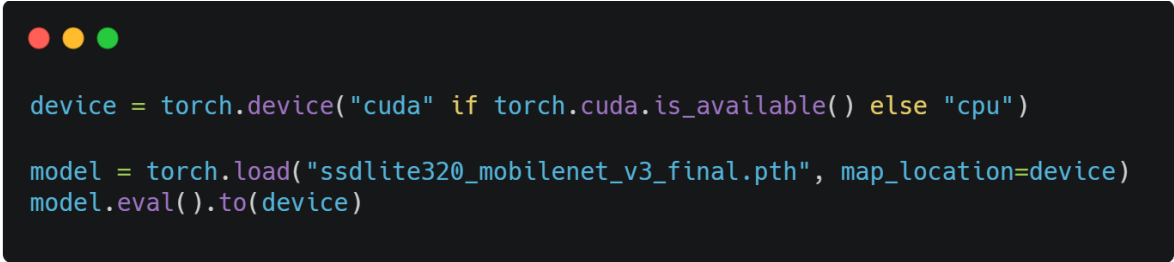
9.2 Nasazení modelu PyTorch.

Pro nasazení modelu vytvořeného v prostředí PyTorch byl použit plně trénovaný detektor založený na architektuře SSDLite320 MobileNet V3. Model byl uložen ve formátu .pth a

načítán přímo v prostředí Raspberry Pi bez převodu do mobilní podoby, protože export do formátu PyTorch Mobile nebyl úspěšný z důvodu nepodporovaných struktur.

Inferenční skript využívá knihovny torch a torchvision pro práci s modelem, numpy a opencv pro manipulaci s obrazem a vykreslování a picamera2 pro přímý přístup ke kameře.

Po načtení knihoven dochází k inicializaci modelu:



```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = torch.load("ssdlite320_mobilenet_v3_final.pth", map_location=device)
model.eval().to(device)
```

Obrázek 36: Načtení PyTorch modelu (*zdroj: vlastní*)

Model je načten do paměti a přepnut do evaluačního režimu. Vzhledem k tomu, že Raspberry Pi nepodporuje GPU akceleraci pro PyTorch inference, běží celý proces na CPU.

Následuje nastavení rozlišení kamery a spuštění:



```
picam2 = Picamera2()
picam2.configure(picam2.create_preview_configuration(main={"size": (640, 480)}))
picam2.start()
```

Obrázek 37: Spuštění kamery (*zdroj: vlastní*)

V hlavní smyčce se z kamery získává aktuální snímek, který je převeden na RGB, změněn na vstupní velikost a normalizován podle specifikace trénovaného modelu:



```
frame = picam2.capture_array()
image_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
image_resized = cv2.resize(image_rgb, (320, 320))
tensor = F.to_tensor(image_resized).to(device)
tensor = F.normalize(tensor, [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
```

Obrázek 38: Načtení a úprava vstupu pro PyTorch (*zdroj: vlastní*)

Vstupní obraz je převeden na tenzor, který je následně předán do modelu k inferenci:



```
with torch.no_grad():
    outputs = model([tensor])[0]
```

Obrázek 39: Předání do modelu (*zdroj: vlastní*)

Výstupem modelu je slovník obsahující pole s detekovanými boxy, skóre a příslušnými třídami. Každý výstup s dostatečně vysokým skóre je následně vykreslen do obrazu:



```
for box, score, label in zip(outputs["boxes"], outputs["scores"], outputs["labels"]):
    if score >= 0.7:
        x1, y1, x2, y2 = map(int, box.tolist())
        class_id = label.item() - 1
        label_text = "plastic" # příklad z mapování
        cv2.rectangle(frame, (x1 * 2, y1 * 2), (x2 * 2, y2 * 2), (0, 255, 0), 2)
        cv2.putText(frame, f"{label_text}: {score:.2f}", (x1 * 2, y1 * 2 - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)
```

Obrázek 40: Vykreslení detekce PyTorch (*zdroj: vlastní*)

Na rozdíl od TensorFlow Lite nebylo možné model převést do mobilního formátu, protože PyTorch Mobile nepodporuje některé interní struktury použitých modelů z torchvision, konkrétně třídu ImageList. Z tohoto důvodu byl pro inference použit plný PyTorch model, což vedlo k vyšší náročnosti na paměť i výpočetní výkon. Přesto byl systém funkční a dokázal detekovat objekty přímo na zařízení.

10 VYHODNOCENÍ MODELU

V této kapitole jsou prezentovány a porovnávány výsledky dosažené jednotlivými detekčními modely vytvořenými v rámci této práce. Cílem je podat přehled o jejich výkonnosti na základě zvolených kvantitativních metrik a následného chování při nasazení v cílovém prostředí.

Hodnocení zahrnuje modely SSD MobileNet V2 FPNLite a EfficientDet D0, vytvořené a trénované v prostředí TensorFlow, a dále model SSDLite320 MobileNet V3, realizovaný v PyTorch. Pro každý z těchto modelů byly během tréninku sledovány metriky přesnosti a citlivosti detekce. Dále byla u všech variant provedena praktická inference na zařízení Raspberry Pi, kde byl sledován reálný výkon v podobě snímkové frekvence (FPS).

10.1 Přehled výsledků

V následujícím přehledu jsou uvedeny hodnoty tří hlavních metrik, které byly sledovány u všech modelů:

- $mAP@[0.50:0.95]$ – průměrná přesnost napříč různými prahy IoU,
- $AR@100$ – průměrný recall při maximálně 100 detekcích na obrázek,
- FPS – skutečný počet snímků za sekundu při běhu inference na Raspberry Pi.

Tabulka níže shrnuje dosažené výsledky po ukončení tréninku a implementaci:

Tabulka 3: Výsledky trénování

Model	$mAP@[0.50:0.95]$	$AR@100$	FPS na Raspberry Pi
SSD MobileNet V2 FPNLite	0,62	0,85	~11
EfficientDet D0	0,73	0,85	~2,5
SSDLite320 MobileNet V3 (PyTorch)	0,53	0,63	~4,5

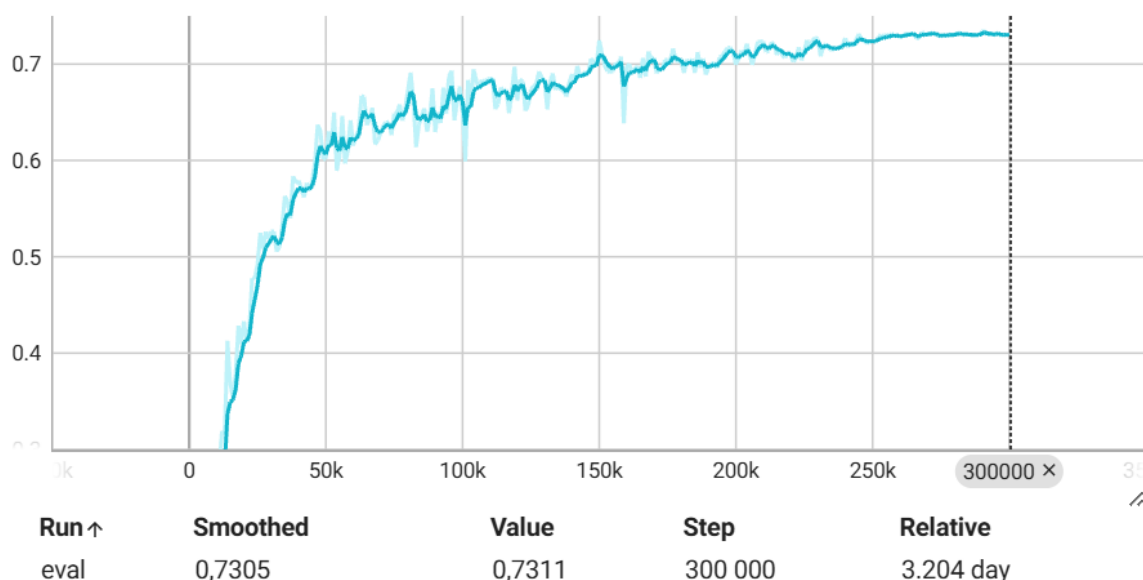
(zdroj: vlastní)

Model EfficientDet D0 dosahuje nejvyšší hodnoty přesnosti i recallu. SSD MobileNet V2 se přibližuje jeho citlivosti, a to i přes architektonicky jednodušší strukturu, a výrazně překonává ostatní modely ve snímkové frekvenci. Naproti tomu model SSDLite320

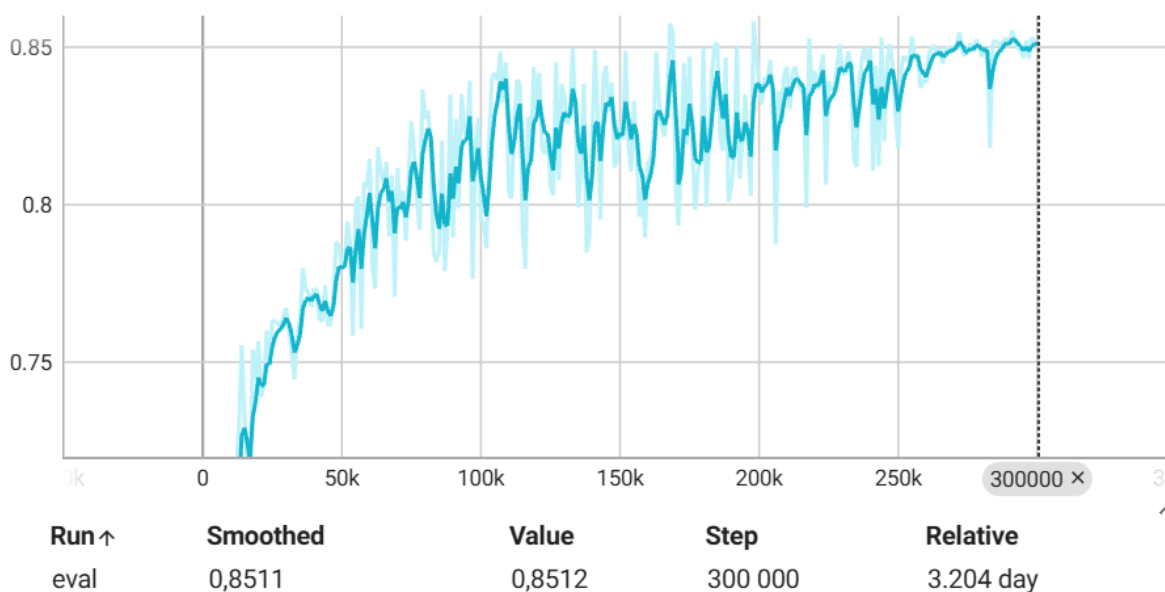
MobileNet V3 vykazuje výrazně nižší přesnost i citlivost, a jeho výsledky jsou tak na hranici praktické použitelnosti – při přibližně poloviční úspěšnosti správné klasifikace může být vhodný pouze pro méně náročné scénáře nebo jako vývojové řešení v prostředí PyTorch.

10.2 Průběh tréninku

U modelu EfficientDet D0 byla metrika $mAP@[0.50:0.95]$ zaznamenávána průběžně až do 300 000 trénovacích kroků. Hodnota mAP se postupně zvyšovala a kolem 200 000 kroků došlo ke zdatelnému zpomalení růstu a stabilizaci výsledků. Finální hodnota mAP dosáhla přibližně 0,73, přičemž i metrika $AR@100$ sledovala podobný vývoj a ustálila se na 0,85. Průběh tréninku byl plynulý a konzistentní, bez výraznějších výkyvů nebo nestability.

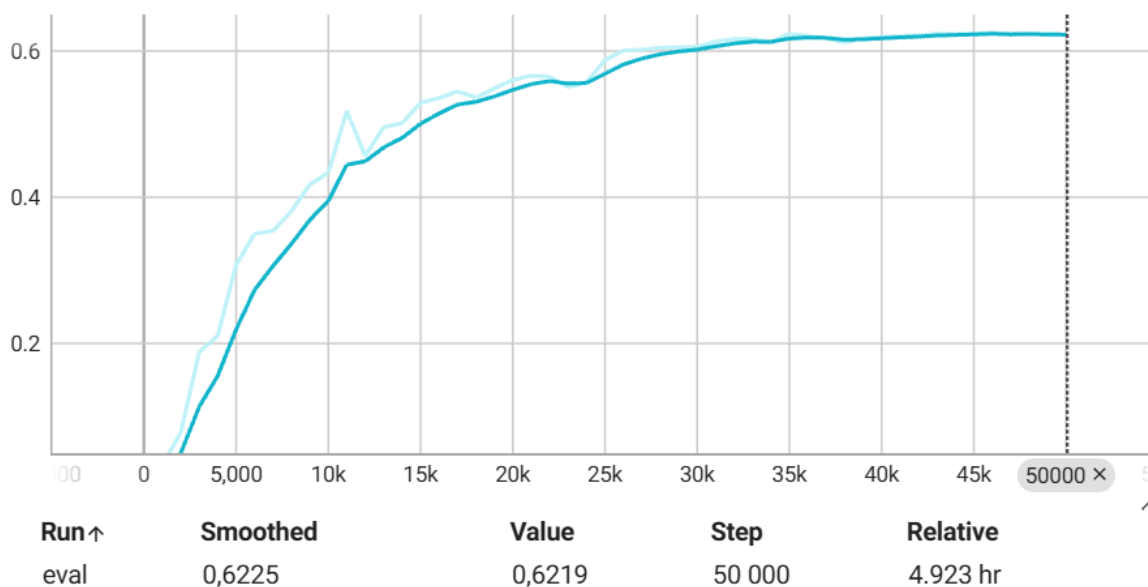


Obrázek 41: EfficientDet D0 - $mAP@[0.50:0.95]$ (zdroj: vlastní)

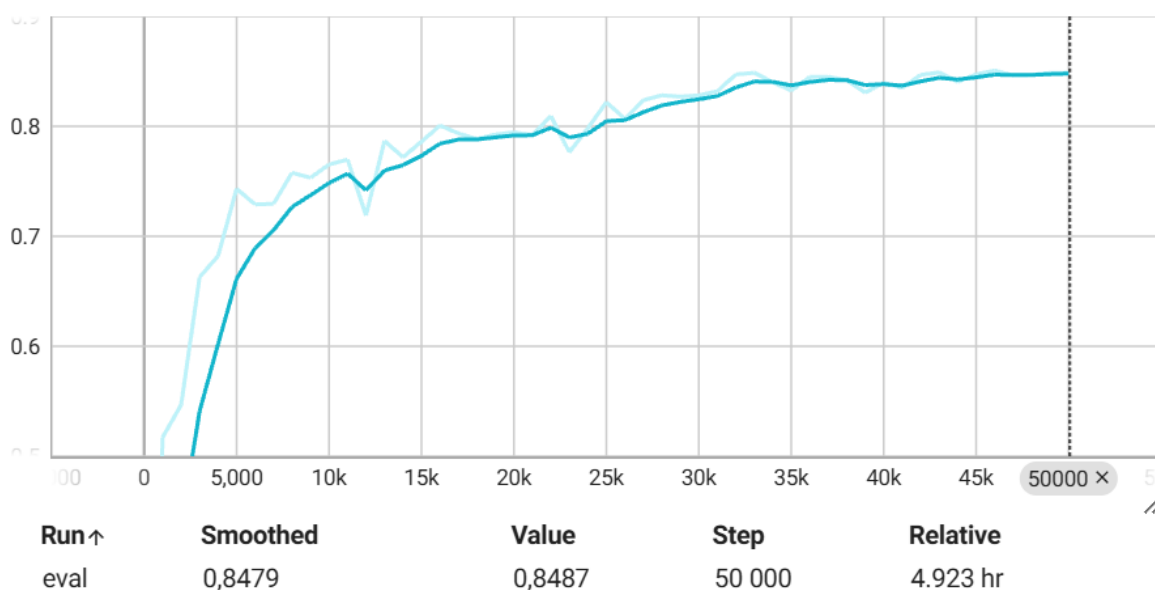


Obrázek 42: EfficientDet D0 - AR@100 (zdroj: vlastní)

Model SSD MobileNet V2 FPNLite byl trénován po dobu 50 000 kroků. mAP se zde zvyšoval pozvolna a ustálil se již kolem 35 000 kroků na hodnotě 0,62. Vývoj metriky AR@100 probíhal stabilněji, výsledná hodnota činila 0,85, tedy stejně jako u složitějšího EfficientDet. Vzhledem k menšímu počtu parametrů a jednodušší architektuře model konvergoval poměrně rychle, a to bez známek přetučení.

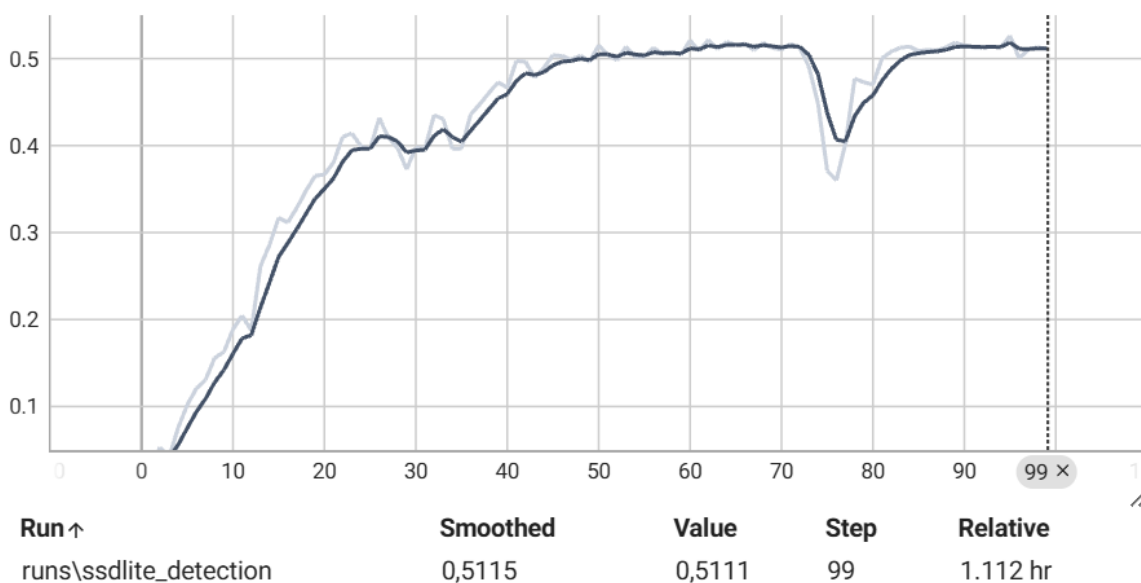


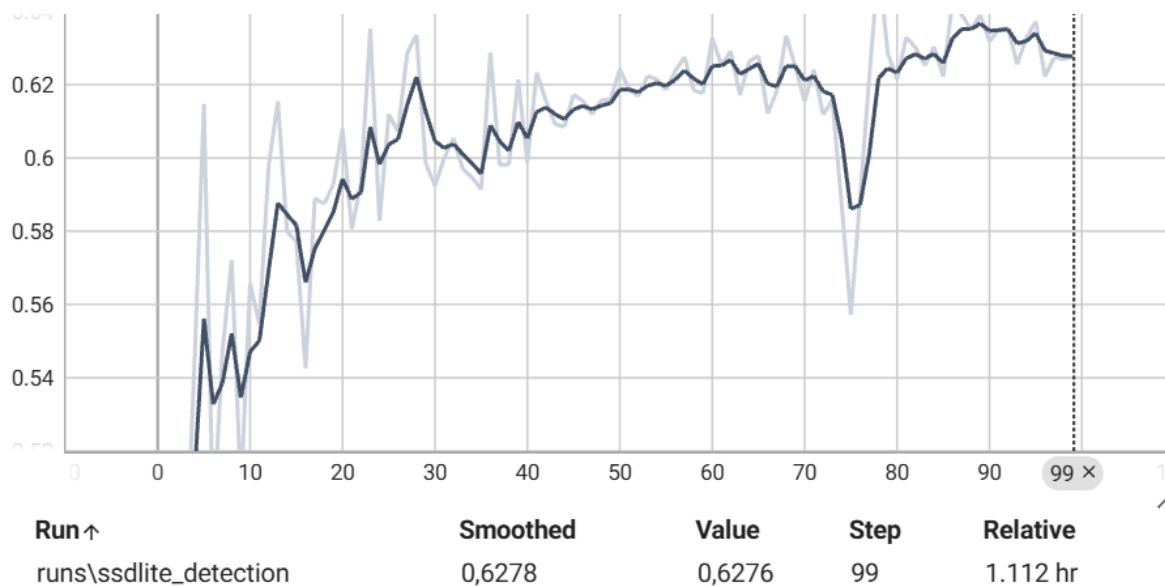
Obrázek 43: SSD MobileNet V2 FPNLite - mAP@[0.50:0.95] (zdroj: vlastní)



Obrázek 44: SSD MobileNet V2 FPNLite - AR@100 (zdroj: vlastní)

U modelu SSDLite320 MobileNet V3 trénovaného v prostředí PyTorch byl vývoj metrik méně stabilní. $mAP@[0.50:0.95]$ kolísal a jeho růst byl pomalejší. Až v pozdější fázi tréninku se hodnota ustálila na úrovni 0,53, přičemž i zde bylo patrné určité zpomalení nárůstu v závěru. Získaná metrika AR@100 dosáhla 0,63, což je znatelně nižší než u ostatních modelů. Trénink byl výrazně náročnější na ladění a výsledky ukazují na nižší schopnost modelu generalizovat na složitější nebo méně jednoznačné příklady.

Obrázek 45: SSDLite320 MobileNet V3 - $mAP@[0.50:0.95]$ (zdroj: vlastní)

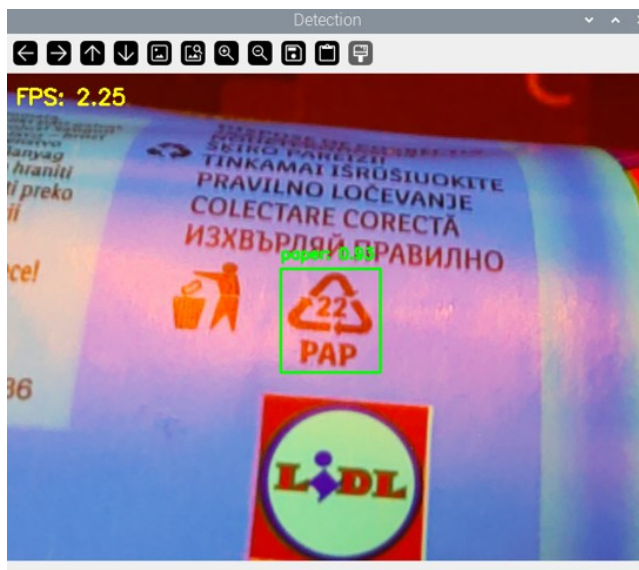


Obrázek 46: SSDLite320 MobileNet V3 - AR@100 (zdroj: vlastní)

10.3 Porovnání výkonu

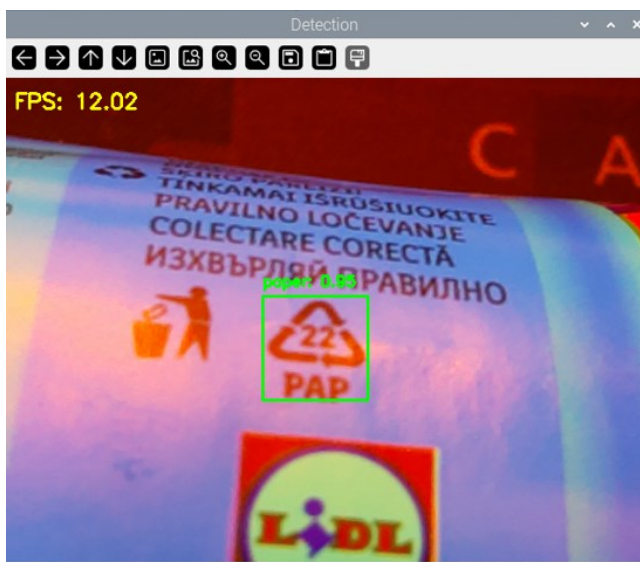
Porovnání výkonu jednotlivých modelů ukazuje na výrazné rozdíly mezi architekturami, a to nejen ve smyslu dosažené přesnosti, ale i z hlediska praktické použitelnosti na cílové platformě.

Model EfficientDet D0 dosáhl nejlepších výsledků v přesnosti ($mAP = 0,73$) i citlivosti ($AR@100 = 0,85$). Jeho výhodou je vysoká schopnost generalizace, konzistentní průběh tréninku a výstupy s minimem falešných detekcí. Ačkoli inference na Raspberry Pi byla s výkonem přibližně 2,5 FPS nejpomalejší, kvalita detekce jednoznačně převyšovala ostatní modely. Převod do formátu TensorFlow Lite byl možný po jednoduché úpravě konfiguračního souboru, což potvrzuje i jeho praktickou použitelnost při zachování vysoké přesnosti.



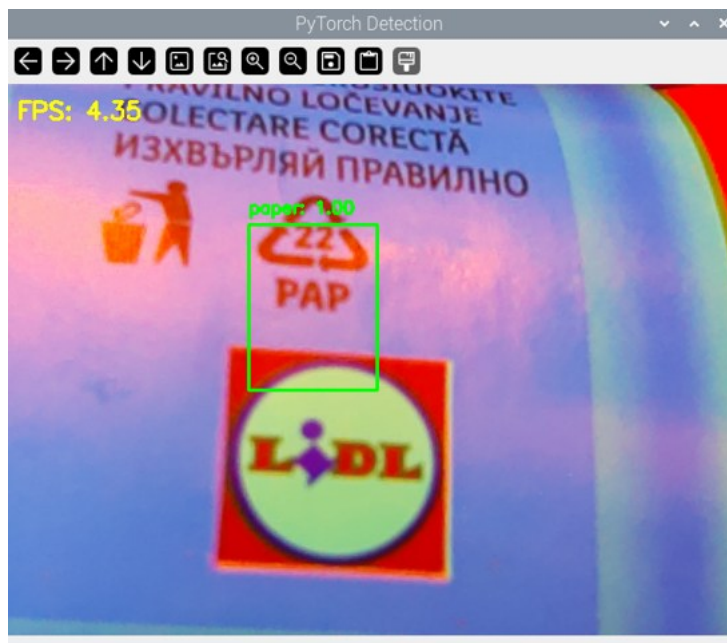
Obrázek 47: EfficientDet D0 - FPS

Model SSD MobileNet V2 FPNLite nabídl nejlepší poměr mezi přesností ($mAP = 0,62$), vysokou citlivostí ($AR@100 = 0,85$) a výkonností inference, která probíhala rychlostí cca 11 FPS. Díky bezproblémovému převodu do TensorFlow Lite formátu a stabilnímu výstupu byl model nejefektivněji nasaditelný v reálných podmínkách.



Obrázek 48: SSD MobileNet V2 FPNLite - FPS

Model SSDLite320 MobileNet V3, implementovaný v PyTorch, vykázal nižší hodnoty přesnosti ($mAP = 0,53$) i citlivosti ($AR@100 = 0,63$), přičemž inference probíhala rychlostí okolo 4,5 FPS. Významným omezením byla nemožnost převodu do mobilního formátu, což znamená vyšší nároky na běhové prostředí. Přesto model úspěšně fungoval i na Raspberry Pi, což potvrzuje funkčnost základní PyTorch inference i bez optimalizace.



Obrázek 49: SSDLite320 MobileNet V3 - FPS

Výsledky tak ilustrují klasický kompromis mezi přesností detekce a výpočetní náročností. EfficientDet D0 se ukazuje jako nejlepší z hlediska detekční kvality, zatímco SSD MobileNet V2 exceluje v efektivitě a jednoduchosti nasazení. PyTorch model představuje rozumnou volbu v prostředí bez podpory TensorFlow Lite, ovšem s omezenou detekční spolehlivostí.

ZÁVĚR

Tato diplomová práce se zabývala návrhem, implementací a ověřením systému pro detekci recyklačních symbolů s využitím metod hlubokého učení na embedded zařízení Raspberry Pi. Byly porovnány tři moderní detekční modely – SSD MobileNet V2 FPNLite, EfficientDet D0 a SSDLite320 MobileNet V3 – z hlediska přesnosti, výpočetní náročnosti a vhodnosti pro nasazení v omezeném hardwarovém prostředí.

Model EfficientDet D0 dosáhl nejvyšší přesnosti detekce ($mAP@0.50:0.95$ až 0,73), zatímco SSD MobileNet V2 FPNLite se ukázal jako nejefektivnější z hlediska rychlosti a jednoduchosti integrace. PyTorch model (SSDLite320 MobileNet V3) byl funkční, ale méně vhodný pro embedded nasazení kvůli horší podpoře optimalizace a exportu.

Výsledky potvrdily klasický kompromis mezi přesností modelu a jeho nároky na výpočetní výkon. Pro embedded systémy, kde je klíčová rychlost a energetická nenáročnost, je výhodnější volit lehčí architektury, ačkoliv za cenu mírně nižší přesnosti. Ačkoliv dosažená přesnost umožňuje spolehlivou detekci recyklačních symbolů, reálná rychlost zpracování na Raspberry Pi ukazuje, že systém není vhodný pro nasazení v aplikacích vyžadujících skutečně real-time odezvu bez dalších optimalizací.

V budoucnu by bylo možné zvýšit výkonnost celého systému několika směry. Prvním z nich je rozšíření datasetu o více obrazových vzorků, což by mohlo zlepšit generalizační schopnosti modelu. Další možností je použití výkonnějšího hardwaru nebo rozšíření Raspberry Pi o akcelerátory, jako je Google Coral TPU nebo Intel Neural Compute Stick, které umožňují nasazení složitějších a přesnějších modelů bez výrazného navýšení spotřeby energie. Rovněž by stálo za zvážení využití novějších architektur, např. transformerových detektorů optimalizovaných pro edge zařízení.

Tato práce tak poskytuje nejen funkční implementaci systému detekce recyklačních symbolů, ale i podklad pro další vývoj nízkonákladových inteligentních třídících zařízení s využitím metod hlubokého učení.

SEZNAM POUŽITÉ LITERATURY

- [1] GOODFELLOW, Ian; BENGIO, Yoshua a COURVILLE, Aaron. *Deep learning. Adaptive computation and machine learning series*. Cambridge, Massachusetts: The MIT Press, [2016]. ISBN 978-0-262-03561-3.
- [2] DAVIES, E. R. *Computer vision: principles, algorithms, applications, learning*. Fifth edition. London, England: Academic Press, 2018. ISBN 978-0-12-809575-1.
- [3] RUSSAKOVSKY, Olga, DENG, Jia, SU, Hao, KRAUSE, Jonathan, SATHEESH, Sanjeev, MA, Sean, HUANG, Zhiheng, KARPATY, Andrej, KHOSLA, Aditya, BERNSTEIN, Michael, BERG, Alexander C. a FEI-FEI, Li. *ImageNet Large Scale Visual Recognition Challenge*. Online. 2015. Dostupné z: <https://arxiv.org/abs/1409.0575>. [cit. 2024-05-24].
- [4] REDMON, Joseph a FARHADI, Ali. *YOLOv3: An Incremental Improvement*. Online. 2018. Dostupné z: <https://arxiv.org/abs/1804.02767>. [cit. 2024-05-24].
- [5] PRINCE, Simon J. D. *Computer vision: models, learning, and inference*. New York: Cambridge University Press, 2012. ISBN 978-1-107-01179-3.
- [6] DOSOVITSKIY, Alexey, Lucas BEYER, Alexander KOLESNIKOV, Dirk WEISSENBORN, Xiaohua ZHAI, Thomas UNTERTHINER, Mostafa DEGHANI, Matthias MINDERER, Georg HEIGOLD, Sylvain GELLY, Jakob USZKOREIT a Neil HOULSBY. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. Online. 2021. Dostupné z: <https://arxiv.org/abs/2010.11929>. [cit. 2024-05-24].
- [7] LOWE, D. G. *Distinctive Image Features from Scale-Invariant Keypoints* [Online]. International Journal of Computer Vision, 2004, vol. 60, no. 2, pp. 91–110. Dostupné z: <https://doi.org/10.1023/B:VISI.0000029664.99615.94>. [cit. 2024-05-24].
- [8] TURK, Matthew a PENTLAND, Alex. *Eigenfaces for Recognition*. Online. Journal of Cognitive Neuroscience, 1991, vol. 3, no. 1, pp. 71–86. Dostupné z: <https://doi.org/10.1162/jocn.1991.3.1.71>. [cit. 2024-05-24].
- [9] SCHROFF, Florian, KALENICHENKO, Dmitry a PHILBIN, James. *FaceNet: A unified embedding for face recognition and clustering*. Online. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2015, pp. 815–823. Dostupné z: <https://doi.org/10.1109/CVPR.2015.7298682>. [cit. 2024-05-24].

- [10] GIRSHICK, Ross, DONAHUE, Jeff, DARRELL, Trevor a MALIK, Jitendra. *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation*. Online. 2014. Dostupné z: <https://arxiv.org/abs/1311.2524>. [cit. 2024-05-24].
- [11] DALAL, N. a TRIGGS, B. *Histograms of oriented gradients for human detection*. Online. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, San Diego, CA, USA, 2005, vol. 1, pp. 886–893. Dostupné z: <https://doi.org/10.1109/CVPR.2005.177>. [cit. 2024-05-24].
- [12] REN, Shaoqing, HE, Kaiming, GIRSHICK, Ross a SUN, Jian. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. Online. 2016. Dostupné z: <https://arxiv.org/abs/1506.01497>. [cit. 2024-05-24].
- [13] CARION, Nicolas, MASSA, Francisco, SYNNAEVE, Gabriel, USUNIER, Nicolas, KIRILLOV, Alexander a ZAGORUYKO, Sergey. *End-to-End Object Detection with Transformers*. Online. 2020. Dostupné z: <https://arxiv.org/abs/2005.12872>. [cit. 2024-05-24].
- [14] LONG, Jonathan, SELHAMER, Evan a DARRELL, Trevor. *Fully Convolutional Networks for Semantic Segmentation*. Online. 2015. Dostupné z: <https://arxiv.org/abs/1411.4038>. [cit. 2024-05-24].
- [15] GONZALEZ, Rafael C. a WOODS, Richard E. *Digital image processing*. Fourth edition. New York: Pearson, [2018]. ISBN 978-0-13-335672-4.
- [16] RONNEBERGER, Olaf, FISCHER, Philipp a BROX, Thomas. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. Online. 2015. Dostupné z: <https://arxiv.org/abs/1505.04597>. [cit. 2024-05-24].
- [17] KIRILLOV, Alexander, MINTUN, Eric, RAVI, Nikhila, MAO, Hanzi, ROLLAND, Chloe, GUSTAFSON, Laura, XIAO, Tete, WHITEHEAD, Spencer, BERG, Alexander C., LO, Wan-Yen, DOLLÁR, Piotr a GIRSHICK, Ross. *Segment Anything*. Online. 2023. Dostupné z: <https://arxiv.org/abs/2304.02643>. [cit. 2024-05-24].
- [18] VASWANI, Ashish, SHAZEER, Noam, PARMAR, Niki, USZKOREIT, Jakob, JONES, Llion, GOMEZ, Aidan N., KAISER, Lukasz a POLOSUKHIN, Illia. *Attention Is All You Need*. Online. 2023. Dostupné z: <https://arxiv.org/abs/1706.03762>. [cit. 2024-05-24].

- [19] NAIR, Vinod a HINTON, Geoffrey E. *Rectified Linear Units Improve Restricted Boltzmann Machines*. Online. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010. Dostupné z: <https://icml.cc/Conferences/2010/papers/432.pdf>. [cit. 2024-05-24].
- [20] RUMELHART, David E., HINTON, Geoffrey E. a WILLIAMS, Ronald J. *Learning Representations by Back-Propagating Errors*. Online. *Nature*, 1986, vol. 323, no. 6088, pp. 533–536. Dostupné z: <https://doi.org/10.1038/323533a0>. [cit. 2024-05-24].
- [21] HOCHREITER, Sepp a SCHMIDHUBER, Jürgen. *Long Short-Term Memory*. Online. *Neural Computation*, 1997, vol. 9, no. 8, pp. 1735–1780. Dostupné z: <https://doi.org/10.1162/neco.1997.9.8.1735>. [cit. 2024-05-24].
- [22] SRIVASTAVA, Nitish, HINTON, Geoffrey, KRIZHEVSKY, Alex, SUTSKEVER, Ilya a SALAKHUTDINOV, Ruslan. *Dropout: a simple way to prevent neural networks from overfitting*. Online. *Journal of Machine Learning Research*, 2014, vol. 15, no. 1, pp. 1929–1958. Dostupné z: <https://dl.acm.org/doi/10.5555/2627435.2670313>. [cit. 2024-05-24].
- [23] HORNIK, Kurt. *Approximation capabilities of multilayer feedforward networks*. Online. *Neural Networks*, 1991, vol. 4, no. 2, pp. 251–257. ISSN 0893-6080. Dostupné z: [https://doi.org/10.1016/0893-6080\(91\)90009-T](https://doi.org/10.1016/0893-6080(91)90009-T). [cit. 2024-05-24].
- [24] GILPIN, Leilani H., BAU, David, YUAN, Ben Z., BAJWA, Ayesha, SPECTER, Michael a KAGAL, Lalana. *Explaining Explanations: An Overview of Interpretability of Machine Learning*. Online. 2019. Dostupné z: <https://arxiv.org/abs/1806.00069>. [cit. 2024-05-24].
- [25] KINGMA, Diederik P. a BA, Jimmy. *Adam: A Method for Stochastic Optimization*. Online. 2017. Dostupné z: <https://arxiv.org/abs/1412.6980>. [cit. 2024-05-24].
- [26] SHORTEN, Connor a KHOSHGOFTAAR, Taghi M. *A survey on Image Data Augmentation for Deep Learning*. Online. *Journal of Big Data*, 2019, vol. 6, no. 1, pp. 60. ISSN 2196-1115. Dostupné z: <https://doi.org/10.1186/s40537-019-0197-0>. [cit. 2024-05-24].
- [27] JHA, Ranjan Ashish. *Mastering PyTorch: create and deploy deep learning models from CNNs to multimodal models, LLMs, and beyond*. Second edition. Expert insight. Birmingham: Packt, 2024. ISBN 978-1-80107-430-8.

- [28] GULLI, Antonio; KAPOOR, Amita a PAL, Sujit. *Deep learning with TensorFlow 2 and Keras: Regression, ConvNets, GANs, RNNs, NLP, and more with TensorFlow 2 and the Keras API*. Second edition. Expert insight. Birmingham: Packt, 2019. ISBN 978-1-83882-341-2.
- [29] SMITH, Leslie N. *Cyclical Learning Rates for Training Neural Networks*. Online. 2017. Dostupné z: <https://arxiv.org/abs/1506.01186>. [cit. 2024-05-24].
- [30] JIAO, Licheng, ZHANG, Fan, LIU, Fang, YANG, Shuyuan, LI, Lingling, FENG, Zhixi a QU, Rong. *A Survey of Deep Learning-Based Object Detection*. Online. 2019. Dostupné z: <https://arxiv.org/pdf/1907.09408.pdf>. [cit. 2024-05-24].
- [31] ZOU, Zhengxia, CHEN, Keyan, SHI, Zhenwei, GUO, Yuhong a YE, Jieping. *Object Detection in 20 Years: A Survey*. Online. 2023. Dostupné z: <https://arxiv.org/abs/1905.05055>. [cit. 2024-05-24].
- [32] LIU, Wei, ANGUELOV, Dragomir, ERHAN, Dumitru, SZEGEDY, Christian, REED, Scott, FU, Cheng-Yang a BERG, Alexander C. *SSD: Single Shot MultiBox Detector*. Online. 2016. Dostupné z: <https://arxiv.org/pdf/1512.02325>. [cit. 2024-05-24].
- [33] HOWARD, Andrew G., ZHU, Menglong, CHEN, Bo, KALENICHENKO, Dmitry, WANG, Weijun, WEYAND, Tobias, ANDREETTO, Marco a ADAM, Hartwig. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. Online. 2017. Dostupné z: <https://arxiv.org/abs/1704.04861>. [cit. 2024-05-24].
- [34] TAN, Mingxing, PANG, Ruoming a LE, Quoc V. *EfficientDet: Scalable and Efficient Object Detection*. Online. 2020. Dostupné z: <https://arxiv.org/abs/1911.09070>. [cit. 2024-05-24].
- [35] SIMONYAN, Karen a ZISSERMAN, Andrew. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. Online. 2015. Dostupné z: <https://arxiv.org/abs/1409.1556>. [cit. 2024-05-24].
- [36] HOWARD, Andrew, SANDLER, Mark, CHU, Grace, CHEN, Liang-Chieh, TAN, Mingxing, WANG, Bo, LE, Quoc V. a ADAM, Hartwig. *Searching for MobileNetV3*. Online. 2019. Dostupné z: <https://arxiv.org/abs/1905.02244>. [cit. 2024-05-24].
- [37] *Multilabel Image Classification Using Deep Learning*. Online, obrázek. In: The MathWorks, Inc. 2025. Dostupné

- z: <https://www.mathworks.com/help/deeplearning/ug/multilabel-image-classification-using-deep-learning.html>. [cit. 2025-05-26].
- [38] *Image classification vs. object detection*. Online, obrázek. In: GeeksforGeeks. 2024. Dostupné z: https://www.researchgate.net/publication/373293605_Indoor_Scene_Understanding_for_the_Visually_Impaired_Based_on_Semantic_Segmentation. [cit. 2025-05-26].
- [39] *Semantic Segmentation & Instance Segmentation*. Online, obrázek. In: ResearchGate. 2025. Dostupné z: https://www.researchgate.net/publication/373293605_Indoor_Scene_Understanding_for_the_Visually_Impaired_Based_on_Semantic_Segmentation. [cit. 2025-05-26].
- [40] *Convolutional layer*. Online, obrázek. In: Anh H. Reynolds. 2025. Dostupné z: <https://anhreynolds.com/blogs/cnn.html>. [cit. 2025-05-26].
- [41] *Pooling layer*. Online, obrázek. In: ResearchGate. 2025. Dostupné z: https://www.researchgate.net/figure/Pooling-layer-operation-oproaches-1-Pooling-layers-For-the-function-of-decreasing-the_fig4_340812216. [cit. 2025-05-26].
- [42] *Basic structure of Convolutional Neural Network*. Online, obrázek. In: ResearchGate. 2025. Dostupné z: https://www.researchgate.net/figure/Basic-structure-of-Convolutional-Neural-Network-CNN-23-Artificial-neural-network_fig1_353049916. [cit. 2025-05-26].
- [43] *Underfitting and overfitting*. Online, obrázek. In: GeeksforGeeks. 2025. Dostupné z: <https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/>. [cit. 2025-05-26].
- [44] *Intersection over Union*. Online, obrázek. In: Idiot Developer. 2023. Dostupné z: <https://idiotdeveloper.com/what-is-intersection-over-union-iou/>. [cit. 2025-05-26].
- [45] *Raspberry Pi*. Online, obrázek. In: RPishop.cz. 2025. Dostupné z: <https://rpishop.cz/raspberry-pi-5/6498-raspberry-pi-5-8gb-ram.html>. [cit. 2025-05-26].
- [46] *Raspberry Pi HQ Camera*. Online, obrázek. In: Mouser Electronics, Inc. 2025. Dostupné z: <https://cz.mouser.com/ProductDetail/Raspberry-Pi/SC0818?qs=T%252BzbugeAwjiDj10FRM2%2FWA%3D%3D>. [cit. 2025-05-26].

- [47] *Dataset*. Online, obrázek. In: Robotflow, Inc. 2025. Dostupné z: <https://universe.roboflow.com/recyclegreenup/greenup/browse?queryText=&pageSize=50&startingIndex=0&browseQuery=true>. [cit. 2025-05-26].
- [48] *Activation functions*. Online, obrázek. In: ResearchGate. 2025. Dostupné z: https://www.researchgate.net/figure/Commonly-used-activation-functions-a-Sigmoid-b-Tanh-c-ReLU-and-d-LReLU_fig3_335845675. [cit. 2025-05-26].
- [49] Struktura neuronové sítě. Online, obrázek. In: Masarykova Univerzita. 2025. Dostupné z: <https://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-intelligence--neuronove-site-jednotlivy-neuron--uvod-do-neuronovych-siti--koncept-umele-neuronove-site>. [cit. 2025-05-26].

SEZNAM OBRÁZKŮ

Obrázek 1: Ukázka klasifikace [37].....	18
Obrázek 2: Ukázka detekce objektů [38].....	19
Obrázek 3: Ukázka semantické segmentace [39]	20
Obrázek 4: Ukázka instance segmentace [39]	21
Obrázek 5: Aktivační funkce – Sigmoid [48]	24
Obrázek 6: Aktivační funkce – Tanh [48]	24
Obrázek 7: Aktivační funkce – ReLU [48].....	24
Obrázek 8: Neuronové sítě – Vrstvy [49]	25
Obrázek 9: Konvoluční vrstva [40].....	27
Obrázek 10: Pooling vrstva [41]	27
Obrázek 11: Konvoluční neuronová síť [42]	28
Obrázek 12: Overfitting & Underfitting [43].....	34
Obrázek 13: Intersection over Union	53
Obrázek 14: Raspberry Pi 5 [45].....	62
Obrázek 15: HQ Kamera [46].....	62
Obrázek 16: Dataset [47]	63
Obrázek 17: Editační prostředí (<i>zdroj: vlastní</i>).....	64
Obrázek 18: Načtení a úprava modelu SSDLite (<i>zdroj: vlastní</i>)	68
Obrázek 19: Úprava formátu pro COCO (<i>zdroj: vlastní</i>)	68
Obrázek 20: Implementace warmup cosine decay (<i>zdroj: vlastní</i>).....	68
Obrázek 21: Výpočet a zpětný průchod (<i>zdroj: vlastní</i>)	69
Obrázek 22: Vyhodnocení SSDLite (<i>zdroj: vlastní</i>).....	69
Obrázek 23: Úprava pipeline (<i>zdroj: vlastní</i>)	71
Obrázek 24: Optimizer SSD MobileNet V2 FPNLite (<i>zdroj: vlastní</i>).....	72
Obrázek 25: Augmentace vstupních dat SSD MobileNet V2 FPNLite (<i>zdroj: vlastní</i>)	72
Obrázek 26: Evaluace Tensorflow Object Detection API (<i>zdroj: vlastní</i>)	73
Obrázek 27: Optimizer EfficientDet-D0 (<i>zdroj: vlastní</i>)	75
Obrázek 28: Augmentace vstupních dat EfficientDet-D0 (<i>zdroj: vlastní</i>)	75
Obrázek 29: Příkaz na exportování modelu (<i>zdroj: vlastní</i>)	76
Obrázek 30: Převod na TFLite (<i>zdroj: vlastní</i>)	77
Obrázek 31: Načtení TFLite modelu (<i>zdroj: vlastní</i>).....	78
Obrázek 32: Načtení kamery (<i>zdroj: vlastní</i>).....	78
Obrázek 33: Načtení a úprava vstupu (<i>zdroj: vlastní</i>)	78
Obrázek 34: Předání vstupu do modelu (<i>zdroj: vlastní</i>)	79

Obrázek 35: Vykreslení detekce (<i>zdroj: vlastní</i>)	79
Obrázek 36: Načtení PyTorch modelu (<i>zdroj: vlastní</i>)	80
Obrázek 37: Spuštění kamery (<i>zdroj: vlastní</i>)	80
Obrázek 38: Načtení a úprava vstupu pro PyTorch (<i>zdroj: vlastní</i>)	80
Obrázek 39: Předání do modelu (<i>zdroj: vlastní</i>)	81
Obrázek 40: Vykreslení detekce PyTorch (<i>zdroj: vlastní</i>)	81
Obrázek 41: EfficientDet D0 - mAP@[0.50:0.95] (<i>zdroj: vlastní</i>)	83
Obrázek 42: EfficientDet D0 - AR@100 (<i>zdroj: vlastní</i>)	84
Obrázek 43: SSD MobileNet V2 FPNLite - mAP@[0.50:0.95] (<i>zdroj: vlastní</i>)	84
Obrázek 44: SSD MobileNet V2 FPNLite - AR@100 (<i>zdroj: vlastní</i>)	85
Obrázek 45: SSDLite320 MobileNet V3 - mAP@[0.50:0.95] (<i>zdroj: vlastní</i>)	85
Obrázek 46: SSDLite320 MobileNet V3 - AR@100 (<i>zdroj: vlastní</i>)	86
Obrázek 47: EfficientDet D0 - FPS	87
Obrázek 48: SSD MobileNet V2 FPNLite - FPS	87
Obrázek 49: SSDLite320 MobileNet V3 - FPS	88

SEZNAM TABULEK

Tabulka 1: Srovnání typů neuronových sítí	28
Tabulka 2: Porovnání Tensorflow a PyTorch	58
Tabulka 3: Výsledky trénování	82

SEZNAM PŘÍLOH

Příloha P I: ZIP soubor

PŘÍLOHA P I: STRUKTURA ODEVZDANÝCH DAT (PRILOHA.ZIP)

Přiložený zip soubor obsahuje:

- Adresář **detect_scripts**
 - Detekční scripty určené pro Raspberry Pi
- Adresář **trained_models**
 - Natrénované modely
- Adresář **training_efficient_det**
 - Potřebné soubory k natrénování EfficientDet D0
- Adresář **training_pytorch**
 - Potřebné soubory k natrénování SSDLite320 MobileNet V3
- Adresář **training_ssd_mobilenet_v2**
 - Potřebné soubory k natrénování SSD MobileNet V2 FPNLite