UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2024 **Bc. Tomáš Vladyka**

UNIVERZITA PARDUBICE

Fakulta elektrotechniky a informatiky

ROBOTICKÉ VIDĚNÍ S VYUŽITÍM UMĚLÉ INTELIGENCE

Bc. Tomáš Vladyka

Diplomová práce

2024

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Tomáš Vladyka**
Osobní číslo:	**I22201**
Studijní program:	**N0714A150005 Automatické řízení**
Téma práce:	**Robotické vidění s využitím umělé inteligence**
Zadávající katedra:	**Katedra řízení procesů**

Zásady pro vypracování

Cíl práce: Cílem práce je vytvořit vlastní systém robotického vidění s prvky umělé inteligence založený na konvolučních neuronových sítích. Získané informace budou předávány robotickému ramenu pomocí standardního protokolu.
Obsah teoretické části: Student provede rešerši problematiky zpracování obrazu, konvolučních neuronových sítí, lokalizace a identifikace objektů.
Obsah praktické části: Student navrhne a vytvoří konvoluční neuronovou síť pro lokalizaci a identifikaci objektů ukázkové aplikace. V rámci testování student posoudí kvalitu lokalizace a identifikace v závislosti na úpravě obrazu prostřednictvím použitých filtrů, rozměrech neuronové sítě, či volbě vhodných aktivačních funkcí perceptronů. Systém zajistí transformaci obrazových souřadnic do souřadného systému robotu a komunikaci s robotickým ramenem standardním protokolem.

Rozsah pracovní zprávy: **60**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

SZELISKI, Richard. Computer Vision: Algorithms and Applications. Texts in computer science. London: Springer, 2010. ISBN 978-1-84882-934-3.
AGGARWAL, Charu C. Neural networks and deep learning: a textbook. Cham, Switzerland: Springer, 2018. ISBN 978-3-319-94462-3.
RASCHKA, Sebastian a MIRJALILI, Vahid. Python machine learning: machine learning and deep learning with Python, scikit-learn, and TensorFlow. Second edition. Expert insight. Birmingham: Packt, 2017. ISBN 978-1-78712-593-3.

Vedoucí diplomové práce: **Ing. Daniel Honc, Ph.D.**
Katedra řízení procesů

Datum zadání diplomové práce: **8. listopadu 2023**
Termín odevzdání diplomové práce: **17. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Daniel Honc, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 14. listopadu 2023

Prohlášení

Prohlašuji:

Práci s názvem Robotické vidění s využitím umělé inteligence jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne

Tomáš Vladyka

Poděkování

Rád bych poděkoval Ing. Danielu Honcovi, Ph.D. za jeho vedení a vstřícnost při zpracovávání této diplomové práce.

ANOTACE

Cílem práce je vytvořit vlastní systém robotického vidění s prvky umělé inteligence založený na konvolučních neuronových sítích. Práce se zabývá ověřením principů a posouzením kvality lokalizace a kategorizace rozpoznávaných objektů. Výsledná data z konvoluční sítě jsou předána robotickému systému prostřednictvím standardního protokolu.

KLÍČOVÁ SLOVA

umělá inteligence, konvoluční neuronová síť, Python, lokalizace, kategorizace.

TITLE

ROBOTIC VISION USING ARTIFICIAL INTELLIGENCE

ANNOTATION

The aim of the work is to create a robotic vision system with artificial intelligence elements based on convolutional neural networks. The work deals with the verification of principles and quality assessment of localization and categorization of recognized objects. The resulting data from the convolutional network is transmitted to the robotic system via a standard protocol.

KEYWORDS

Artificial intelligence, convolutional neural network, Python, localization, categorization.

OBSAH

Seznam zkratek a značek 8

Seznam symbolů veličin a funkcí 9

Seznam ilustrací 10

Úvod 13

1 Teoretická část 14

1.1 Systémy strojového vidění 15

1.1.1 Vývoj 17

1.1.2 Typy aplikací 18

1.2 Konvoluční neuronové sítě 20

1.2.1 Princip funkčnosti 20

1.2.2 Druhy konvolučních neuronových sítí 25

1.3 Průmyslové systémy strojového vidění 28

1.3.1 Systémy 2D robotického vidění 29

1.3.2 Systémy 3D robotického vidění 32

2 Praktická část 34

2.1 Matematické funkce 35

2.1.1 Funkce random 35

2.1.2 Standardní maticové operace 37

2.1.3 Aktivační funkce a výpočet odchylky 43

2.2 Předzpracování obrazu 53

2.2.1 Nalezení objektu 54

2.2.2 Prohledávání obrazu 55

2.2.3 Určení oblasti zájmu 57

2.2.4 Vytvoření datasetu 59

2.3 Vytvoření konvoluční neuronové sítě 63

2.3.1 Konvoluční vrstva 63

2.3.2 Vyhlazovací vrstva 65
2.3.3 Plně-propojená vrstva 65
2.3.4 Komunikace s robotickým systémem 67
2.3.5 Sériová komunikace 68
2.3.6 Protokol modbus 70
2.4 Testovací aplikace 72
2.4.1 Detekce dvou tříd objektů 73
2.4.2 Detekce dvou tříd podobných objektů 74
2.4.3 Detekce tří tříd objektů 76
3 Závěr 78
Použitá literatura 79

SEZNAM ZKRATEK A ZNAČEK

2D	dvourozměrný
3D	třírozměrný
AGV	samo-naváděné vozíky
AR	rozšířená realita
BCE	binární křížová entropie
BFS	prohledávání do šířky
CNN	konvoluční neuronová síť
CTRL	kontrola
FBD	diagram funkčních bloků
ID	identifikační číslo
IP	internetový protokol
len	délka
MSE	střední kvadratická chyba
PC	osobní počítač
PLC	programovatelný automat
RAM	operační paměť
RCNN	rekurzivní konvoluční neuronová síť
ReLU	usměrněná lineární jednotka
ROI	oblast zájmu
RX	přijímač
ST	strukturovaný text
TX	vysílač
VR	virtuální realita

SEZNAM SYMBOLŮ VELIČIN A FUNKCÍ

A, B	matice
a, b	členy matice (vektoru)
i, j, k	indexy pozice
m, n, p	rozměry matice (vektoru)
R	transformační matice rotace
S	transformační matice střihu
$Tanh$	hyperbolická tangenciální funkce
x, y	proměnná
θ	úhel otočení
σ	funkce sigmoid
n	počet kroků

SEZNAM ILUSTRACÍ

Obrázek 1.1 – Označení naučených objektů (Singh, 2022) 15
Obrázek 1.2 – Predikce trajektorie dopravy (Cao, 2021) 16
Obrázek 1.3 – Reprezentace obrazu (Roberts, 1963) 17
Obrázek 1.4 – Typy aplikací strojového vidění (Ramos, 2021) 19
Obrázek 1.5 – Počet vstupních hodnot v závislosti na rozlišení obrazu 21
Obrázek 1.6 – Princip konvoluce nad vstupním obrazem (Baskin, 2017) 22
Obrázek 1.7 – Výsledek aplikace filtrů na vstupní obraz 22
Obrázek 1.8 – Principy sdružovacích metod (Yani, 2019) 24
Obrázek 1.9 – Model perceptronu (Zanetti, 2008) 25
Obrázek 1.10 – Obecná architektura CNN (Aggarwal, 2018) 25
Obrázek 1.11 – Princip architektury sítě R-CNN (Ghosh, 2019) 27
Obrázek 1.12 – Princip architektury sítě Fast R-CNN (Ghosh, 2019) 27
Obrázek 1.13 – Metody osvětlení objektu (Štursa, 2024) 28
Obrázek 1.14 – Zjišťování defektů (DobotVisionStudio, 2024) 29
Obrázek 1.15 – Rozpoznávání objektů (DobotVisionStudio, 2024) 30
Obrázek 1.16 – OCR (DobotVisionStudio, 2024) 31
Obrázek 1.17 – Programování funkčních bloků (DobotVisionStudio, 2024) 31
Obrázek 1.18 – Principy 3D snímání (Iwamoto, 2018) 32
Obrázek 1.19 – Rekonstrukce objektu pomocí projekce (Iwamoto, 2018) 32
Obrázek 1.20 – Příklad aplikace bin-picking (Accupick, 2024) 33
Obrázek 2.1 – Rozložení hodnot, pokud *mod* = "Gauss" 36
Obrázek 2.2 – Lineární rozložení hodnot 36
Obrázek 2.3 – Přepsání vybraného souboru 37
Obrázek 2.4 – Vrácení požadovaných pseudo-náhodných hodnot 37
Obrázek 2.5 – Funkce pro vracení prvků v poslední dimenzi pole 38
Obrázek 2.6 - Součet matic 39

Obrázek 2.7 – Rozdíl matic 40
Obrázek 2.8 – Funkce pro zjištění rozměrů pole 40
Obrázek 2.9 – Funkce pro násobení matic 42
Obrázek 2.10 – Transpozice matice 43
Obrázek 2.11 – Funkce Sigmoid 44
Obrázek 2.12 - Výpočet funkce Sigmoid 44
Obrázek 2.13 – Derivace funkce Sigmoid 45
Obrázek 2.14 – Funkce Tanh 46
Obrázek 2.15 – Výpočet funkce Tanh 46
Obrázek 2.16 – Derivace funkce Tanh 47
Obrázek 2.17 – Funkce ReLU 48
Obrázek 2.18 – Výpočet aktivační funkce ReLU 49
Obrázek 2.19 – Funkce pro výpočet středí kvadratické odchylky 51
Obrázek 2.20 – Funkce pro výpočet binární křížové entropie 52
Obrázek 2.21 – Obraz pořízený kamerou 52
Obrázek 2.22 – Ztráta informace zmenšením obrazu 54
Obrázek 2.23 – Ztráta informace zmenšením po nalezení objektu 55
Obrázek 2.24 – Inicializace funkce Převod obrazu 56
Obrázek 2.25 – Nalezení objektu na vstupním obrazu 56
Obrázek 2.26 – Oříznutí a uložení obrazu 57
Obrázek 2.27 – Inicializace funkce 58
Obrázek 2.28 – Prohledávání do šířky 58
Obrázek 2.29 – Nalezení obdélníkové hranice hledaného objektu 59
Obrázek 2.30 – Efekt Aliasingu 60
Obrázek 2.31 – Postupné otáčení obrazu metodou tří střihů 61
Obrázek 2.32 – Funkce pro rotaci původního obrazu 62
Obrázek 2.33 – Výsledný dataset z jednoho vstupního obrazu 62

Obrázek 2.34 – Vytvoření neuronové sítě 63
Obrázek 2.35 – Inicializace konvoluční vrstvy 64
Obrázek 2.36 – Dopředné šíření konvoluční vrstvy 64
Obrázek 2.37 – Konvoluční vrstva – zpětné šíření 64
Obrázek 2.38 – Změna tvaru matice 65
Obrázek 2.39 – Inicializace plně-propojené vrstvy 67
Obrázek 2.40 – Dopředné šíření predikce plně-propojené vrstvy 67
Obrázek 2.41 – Zpětné šíření chyby plně-propojené vrstvy 67
Obrázek 2.42 – Roboty MG400 a CR3 67
Obrázek 2.43 – Reprezentace příkazu GetPose (Johnston, 2019) 68
Obrázek 2.44 – Odpověď robotu na příkaz pro určení polohy (Johnston, 2019) 69
Obrázek 2.45 – Zahájení komunikace po sériové lince 69
Obrázek 2.46 – Funkce pro pohyb robotu pomocí sériové komunikace 70
Obrázek 2.47 – Program pro komunikaci přes Modbus 71
Obrázek 2.48 – Dataset pro detekci dvou tříd objektů 73
Obrázek 2.49 – Průběh učení detekce dvou tříd objektů 74
Obrázek 2.50 – Dataset dvou tříd podobných objektů 74
Obrázek 2.51 – Průběh učení detekce dvou tříd podobných objektů 75
Obrázek 2.52 – Podobnost vstupních obrazů 76
Obrázek 2.53 – Dataset pro detekci tří tříd objektů 76
Obrázek 2.54 – Průběh učení detekce tří tříd objektů 77

ÚVOD

Systémy robotického vidění (někdy označovány jako vidění kamerové či strojové) umožňují strojům rozpoznávat, analyzovat a interpretovat obrazová data z okolního prostředí. Tato schopnost je často založena na využití různých technik z oblasti umělé inteligence, jako jsou konvoluční neuronové sítě, které jsou schopny automaticky identifikovat relevantní informace z obrazových dat a díky nim provádět různé úkony, jako je lokalizace objektů nebo jejich kategorizace.

V současném technologickém prostředí se robotické vidění stává často využívaným prvkem pro mnoho aplikací od průmyslové automatizace po autonomní vozidla a domácí roboty. Kvalitu robotického vidění lze posoudit dle několika kritických parametrů, jako je přesnost rozpoznávání objektů v různých prostředích, rychlost zpracování dat či náročnost a účinnost učení nových objektů.

Cílem této diplomové práce je vytvořit vlastní systém robotického vidění s prvky umělé inteligence, který bude schopen lokalizovat a kategorizovat předem naučené objekty. Architektura neuronové sítě bude vystavěna na principech konvolučních neuronových sítí, které jsou v současnosti jednou z nejefektivnějších metod pro zpracování obrazových dat. Výsledná data z konvoluční sítě budou předávána do existujícího robotického systému prostřednictvím standardního protokolu.

1 TEORETICKÁ ČÁST

Teoretická část práce je věnována obecnému popisu strojového vidění. V první části jsou popsány principy fungování systémů strojového vidění, kde je vysvětlen způsob, jakým tyto systémy interpretují vizuální data. Je zde uveden vývoj strojového vidění od jeho počátků v 60. letech 20. století až do současnosti. Dále jsou v práci vysvětleny a popsány různé typy aplikací strojového vidění, které využívají konvoluční neuronové sítě. Kapitola je zaměřena na specifické aplikace, které demonstrují schopnost těchto sítí rozpoznávat objekty s vysokou přesností.

Ve druhé kapitole je uveden popis základních principů konvolučních neuronových sítí. V této části jsou popsány typy vrstev, které se používají při programování konvolučních neuronových sítí, jako jsou konvoluční vrstvy, sdružovací vrstvy a plně propojené vrstvy, které společně definují jejich architekturu. Dále jsou v kapitole uvedeny specifické druhy konvolučních neuronových sítí, které jsou běžně využívány, konkrétně CNN (Convolutional Neural Networks), R-CNN (Region-based Convolutional Neural Networks) a Fast R-CNN. Tyto modely jsou analyzovány s ohledem na jejich aplikace ve zpracování obrazu a detekci objektů.

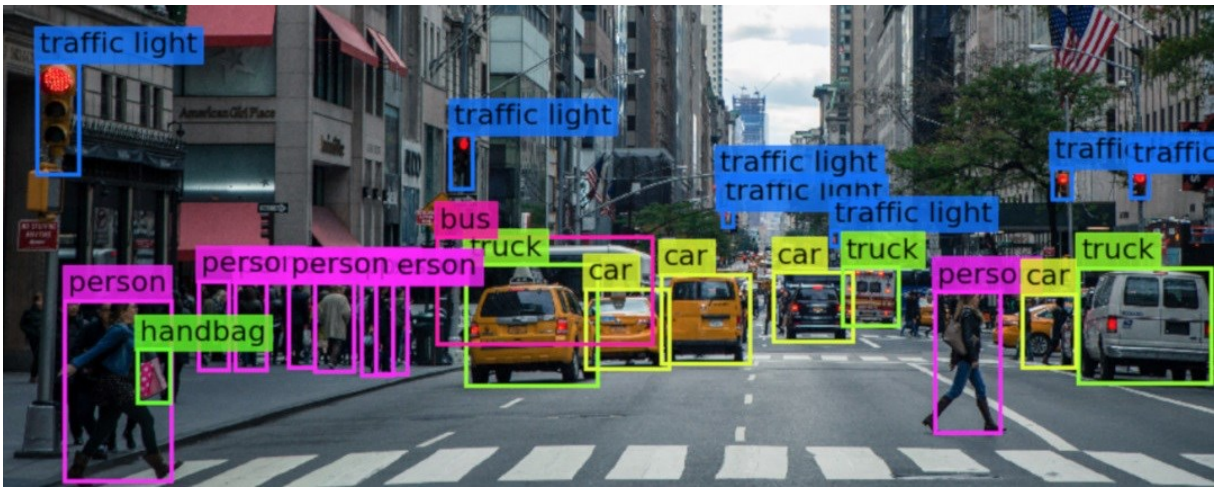
Třetí kapitola se zabývá specifickými průmyslovými aplikacemi strojového vidění, zejména ve výrobě a kontrole kvality. Je zde uvedeno, jak strojové vidění umožňuje automatizaci složitých výrobních procesů, zvyšuje produktivitu a zlepšuje kvalitu výrobků. Kapitola popisuje jak 2D, tak 3D vidění a jejich aplikace v průmyslu, od jednoduchého sledování a třídění objektů po komplexní navigaci a manipulaci s objekty v trojrozměrném prostoru.

1.1 SYSTÉMY STROJOVÉHO VIDĚNÍ

Strojové vidění je obor umělé inteligence, který umožňuje získávat smysluplné informace z digitálních obrazů, videí a dalších vizuálních vstupů a na základě těchto informací jednat. Strojové vidění se v dnešní době používá v řadě různých odvětvích, r zdravotnictví, automobilového průmyslu, výroby a bezpečnosti. Hlavním účelem vývoje systémů strojového vidění je zautomatizovat analýzu a interpretaci vizuálních dat.

Základem strojového vidění je zpracování obrazu. Tento proces často začíná technikami předběžného zpracování (někdy označováno jako předzpracování či preprocess), které zlepšují kvalitu obrazu a snižují šum. Díky předběžnému zpracování je možné zaměřit se pouze na tu část obrazu, která je pro danou aplikaci důležitá, zatím co nepotřebné části jsou zanedbány. Následně jsou z obrazu extrahovány rysy, kterými mohou být barva, textura, tvar nebo jiné charakteristické vlastnosti. Tyto rysy lze poté využít k trénování modelů strojového učení, což jim umožňuje provádět předpovědi nebo rozhodnutí na základě nových, nezpracovaných snímků.

Jedním ze základních úkolů systémů strojového vidění je rozpoznávání objektů. Algoritmus tak dokáže identifikovat a klasifikovat objekty na snímcích nebo videích. Tato schopnost je klíčová v mnoha aplikacích, od systémů rozpoznávání obličejů a automatizované navigace vozidel až po robotickou chirurgii nebo aplikace rozšířené reality (AR). Pro nasazení strojového vidění do aplikací existuje celá řada algoritmů. V dnešní době jsou pro aplikace často nasazovány architektury umělé inteligence, jako jsou konvoluční neuronové sítě (CNN). Využití konvolučních neuronových sítí má zásadní význam pro dosažení vysoké přesnosti v úlohách rozpoznávání objektů a simuluje způsob, jakým lidský mozek pracuje při rozpoznávání vizuálních vzorů. (Aggarwal, 2018)



Obrázek 1.1 – Označení naučených objektů (Singh, 2022)

Kromě detekce a klasifikace objektů na statických obrazech je významnou oblastí strojového vidění i analýza pohybu, která zahrnuje porozumění změny polohy objektů v sekvenci snímků nebo videu. Takové aplikace zahrnují sledování dráhy pohybujících se objektů, odhad jejich rychlosti a předpovídání budoucí polohy. Analýza pohybu má zásadní význam v oblasti samonaváděných vozíků (také označovány jako AGV), kde pomáhá sledovat situaci na pracovišti a predikovat možné kolize, či v oblasti sportovní analýzy, kde pomáhá zvyšovat výkony pomocí přesné analýzy pohybu. (Cao, 2021)

Obrázek 1.2 – Predikce trajektorie dopravy (Cao, 2021)

V současné době se objevuje nový typ aplikace pro strojové vidění – rekonstrukce prostředí. Jedná se o vytvoření trojrozměrného modelu objektu či okolí ze souboru několika stovek dvourozměrných snímků. Tato technologie se často využívá ve spojení se systémy rozšířené (AR) i virtuální reality (VR), kde zlepšuje interakci uživatele s digitálním prostředím. V robotice rekonstrukce scény umožňuje robotickým systémům efektivnější navigaci a interakci s prostředím.

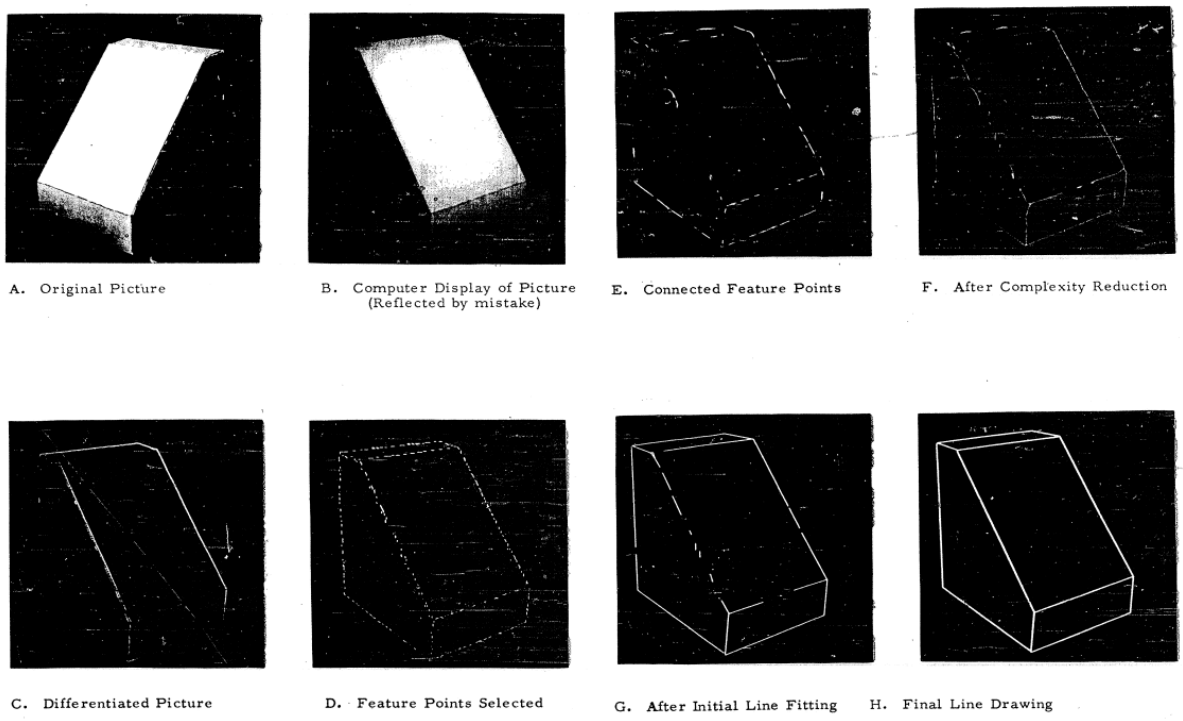
Při vývoji a nasazování systémů strojového vidění je zapotřebí brát ohled na poměrně velké množství faktorů, které mohou mít vliv na kvalitu zpracování obrazu. Často je nutné zaměřit se na řešení různých světelných podmínek, úhlů pohledu, clony (kdy je objekt zájmu částečně zakrytý) či změny vzhledu objektu. V praxi u algoritmů deep learningu obecně platí pravidlo – čím složitější aplikace (tedy s jak velkým množství proměnných okolních vlivů je

nutné počítat), tím je zapotřebí větší velikost datasetu, na kterém je systém naučen. (Raschka, 2017)

1.1.1 VÝVOJ

Za vznik strojového vidění lze považovat výzkum amerického psychologa Franka Rosenblatta v 50. a počátcích 60. let 20. století. Ten byl jedním z prvních, kdo se rozhodl experimentovat s principy fungování lidského mozku a který v roce 1957 představil koncept perceptronu – první jednoduché neuronové sítě pro počítače. Tato síť prokázala, že stroje dokáží rozpoznávat jednoduché vzory naučené vstupními daty. (Rosenblatt, 1961)

Koncept počítačového vidění byl poprvé představen v 60. letech 20. století, kdy na MIT pracoval Lawrence Roberts. Jeho disertační práce "Strojové vnímání trojrozměrných těles" položila základy pro extrakci trojrozměrných tvarů z dvojrozměrných obrazů, což je v dnešní době jeden ze základních principů strojového vidění. V průběhu 70. let 20. století umožnil pokrok v oblasti počítačového hardwaru a technik zpracování obrazu další zkoumání počítačového vidění. Byly vyvinuty algoritmy, které dokázaly detekovat hrany, rozpoznávat textury a interpretovat tvary z obrazů. V této době byly také založeny první specializované laboratoře pro studium počítačového vidění, jako například MIT Computer Science & Artificial Intelligence Laboratory, která hrála klíčovou roli v dalším rozvoji. (Roberts, 1963)



Obrázek 1.3 – Reprezentace obrazu (Roberts, 1963)

V 90. letech 20. století došlo k prudkému nárůstu praktických aplikací počítačového vidění, který byl způsoben rostoucím výpočetním výkonem a nástupem sofistikovanějších zobrazovacích technologií. V roce 2010 bylo možné integrovat počítačové vidění s rostoucím vývojem hlubokého učení, zejména díky vývoji konvolučních neuronových sítí (CNN). Konvoluční neuronové sítě, zpopularizované výzkumníky jako je Yann LeCun, se staly dnes již běžným způsobem, jak dosáhnout vysoké přesnosti v úlohách, jako je klasifikace obrazu, detekce objektů či sémantická segmentace. (LeCun, 2010)

1.1.2 TYPY APLIKACÍ

Strojové vidění, podobor umělé inteligence, má širokou škálu aplikací. Čtyři nejčastější úlohy počítačového vidění jsou aplikace klasifikace, lokalizace, detekce a segmentace.

Klasifikace

Klasifikace je jednou z nejzákladnějších, přesto však zásadních úloh počítačového vidění. Zahrnuje kategorizaci celých obrazů do jedné z mnoha předem definovaných tříd. Tento proces je využíván v mnoha aplikacích, od třídění objektů, rozpoznávání obličejů, až po diagnostiku nemocí v lékařském prostředí. Nástup hlubokého učení výrazně zvýšil přesnost a rychlost úloh klasifikace obrazu a umožnil tak využít procesů klasifikace u řady složitých aplikací.

Lokalizace

Lokalizace označuje proces nejen identifikace objektů v obrazu, ale také určení jejich konkrétní polohy. U těchto aplikací jsou obvykle výstupem souřadnice ohraničujícího rámečku, který obklopuje každý objekt. Lokalizace je klíčová pro úlohy, kde je nezbytná znalost přesné polohy objektu, například v autonomní navigaci vozidel, kde je pro bezpečnou navigaci rozhodující pochopení přesného umístění překážek., či v systémech robotického vidění pro aplikace typu pick-and-place.

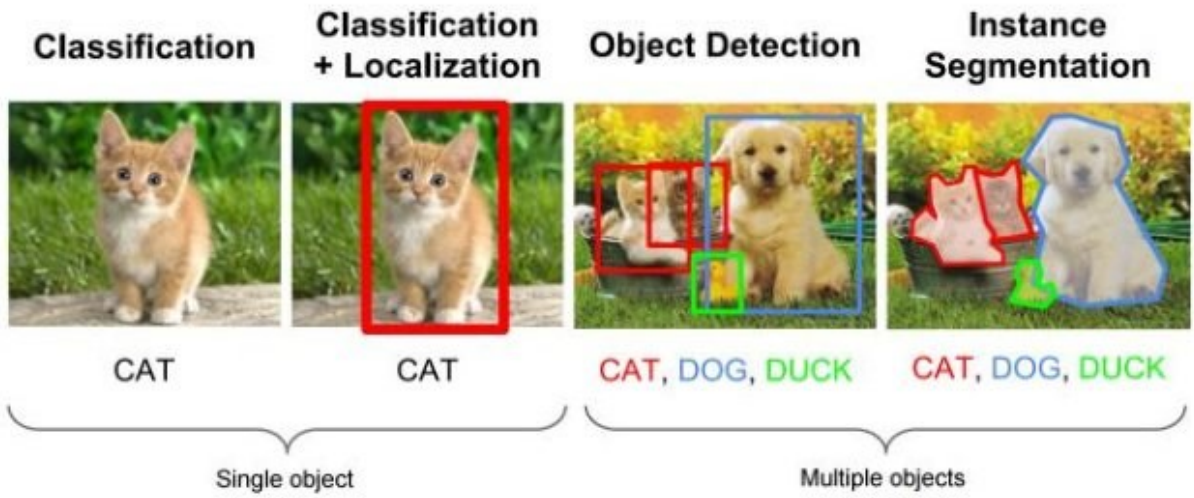
Detekce

Detekce kombinuje prvky klasifikace a lokalizace. Tato úloha zahrnuje identifikaci všech výskytů předem naučených objektů v obraze a určení jejich přesné polohy. Detekce je složitější než klasifikace nebo lokalizace, protože vyžaduje, aby systém byl přesný nejen při

rozpoznávání objektů, ale také při určování jejich hranic v různých scénářích a na různých pozadích. Aplikace detekce jsou velmi rozšířené, jejich aplikace najdou uplatnění v průmyslové výrobě pro detekci defektů výrobků či ve zdravotnictví pro identifikaci nálezů v lékařských snímcích.

Segmentace

je proces rozdělení obrazu na více segmentů nebo oblastí, z nichž každá odpovídá jinému objektu nebo části objektu. Segmentace obrazu je obvykle používaná k přesnější lokalizaci objektů a hranic – na rozdíl od detekce, která poskytuje ohraničující rámeček, segmentace poskytuje přesný obrys objektu. Existují dva hlavní typy segmentace: sémantická segmentace a segmentace instanční. Sémantická segmentace klasifikuje každý pixel obrazu do odpovídající třídy objektů, čímž segmentuje obraz na úrovni pixelů, což je užitečné pro aplikace, jako jsou systémy samonaváděných vozíků, kde je pro rozhodování nutné komplexní pochopení okolí. Instanční segmentace na rozdíl od segmentace sémantické identifikuje každý pixel objektu a zároveň rozlišuje mezi různými instancemi téhož objektu. Tato segmentace je obvykle používaná v aplikacích, kdy se v těsné blízkosti vyskytuje více podobných objektů, například při určování počtu lidí v přeplněném prostoru.



Obrázek 1.4 – Typy aplikací strojového vidění (Ramos, 2021)

1.2 KONVOLUČNÍ NEURONOVÉ SÍTĚ

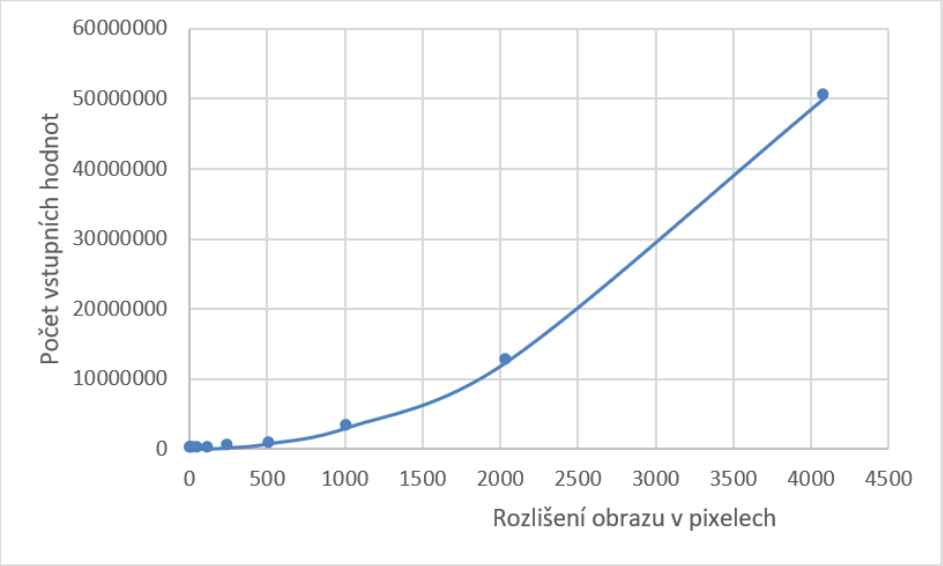
Konvoluční neuronové sítě (CNN) jednou z možných architektur neuronových sítí. Oproti ostatním architekturám jsou konvoluční neuronové sítě určené pro úlohy zahrnující rozpoznávání a zpracování obrazu. Předchůdce CNN lze vysledovat až k Neocognitronu, který představil Kunihiko Fukushima na konci 80. let 20. století. Neocognitron je typ neuronové sítě, která se skládá z několika hierarchicky uspořádaných vrstev buněk, simulujících principy zpracování obrazu lidského oka. Každá vrstva v Neocognitronu je navržena tak, aby identifikovala stále složitější rysy ve vstupních obrazech, počínaje jednoduchými hranami ve spodních vrstvách sítě až po složité vzory, jako jsou obličeje, ve vyšších vrstvách. (Fukushima, 1980)

Do obecného povědomí se konvoluční neuronové sítě dostaly poté, co v soutěži ImageNet Large Scale Visual Recognition Challenge v roce 2012 vyhrál první místo tým Alexe Krizhevskeho. Toto vítězství dokázalo potenciál CNN k dosažení vysoké přesnosti v úlohách vizuálního rozpoznávání. Základní složkou této architektury sítě je konvoluční vrstva, která na vstupní obraz aplikuje řadu naučitelných filtrů. Termín "konvoluční" se týká matematické operace konvoluce – specializovaného druhu lineární operace. CNN využívají konvoluci místo obecného násobení matic alespoň v jedné ze svých vrstev. Tato operace umožňuje zpracovávat data s topologií podobnou mřížce, jako je obrázek, což je ve své podstatě pole pixelů. Díky použití stejného filtru na celý vstupní obraz si konvoluční sítě zachovávají nižší složitost než plně propojené sítě, které posuzují každý pixel zvlášť. To v jednoduchosti znamená, že jakmile se jednou síť naučí příznak v jedné části obrazu, lze jeho výskyt rozpoznat kdekoli na obraze. V typických případech se často postupně skládá několik konvolučních vrstev za sebou, což zvyšuje schopnost sítě zpracovávat velké a složité soubory dat. (Krizhevsky, 2012)

1.2.1 PRINCIP FUNKČNOSTI

Standardní neuronové sítě využívají ve svých architekturách plně-propojené vrstvy. S rostoucím rozlišením obrazu u těchto vrstev však může velice rychle dojít k neúměrně vysokému počtu potřebných vstupních parametrů (viz obrázek 1.5). Tedy v případě rozlišení obrazu 16 x 16 je výsledný počet vstupních parametrů 16 x 16 x 3 (počet barevných kanálů RGB) = 768. U rozlišení obrazu 32 x 32 je výsledný počet vstupních parametrů roven 3072, tedy při dvojnásobné hodnotě rozlišení je počet parametrů vstupujících do neuronové sítě čtyřnásobně větší. V navazujících, skrytých, vrstvách by takováto síť obsahovala tolik parametrů, že by se celá aplikace stal výpočetně velice náročnou. Konvoluční neuronové sítě

využívají takovou strukturu, kde vstupní parametry jsou uspořádány do trojrozměrného vektoru obsahující výšku a šířku obrazu spolu s hloubkou představující počet barevných kanálů obrazu či počet filtrů na obraz použitých.



Obrázek 1.5 – Počet vstupních hodnot v závislosti na rozlišení obrazu

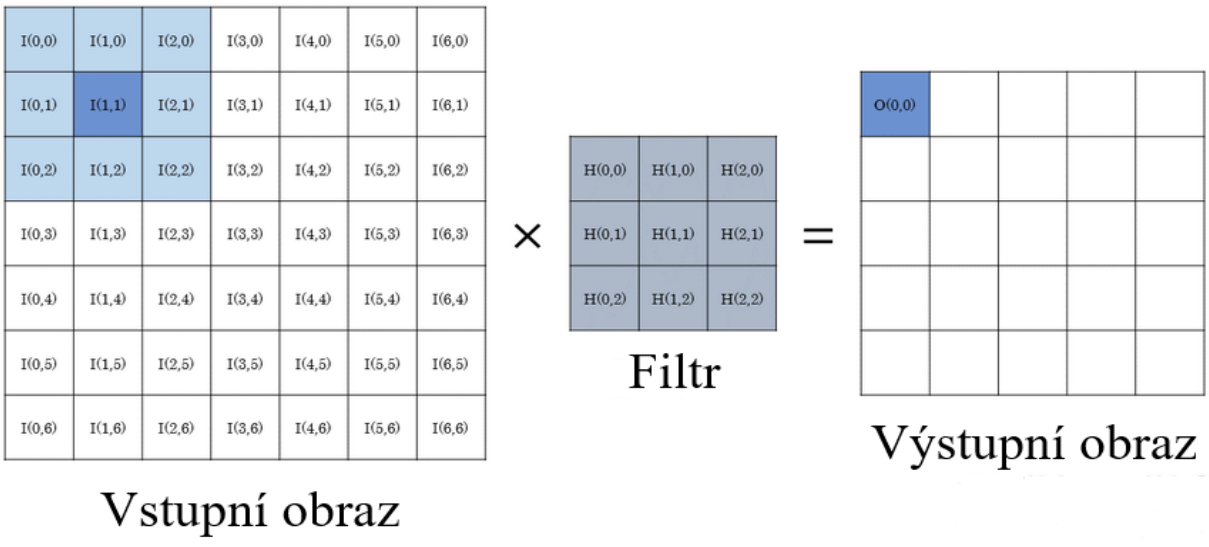
Typická architektura konvolučních neuronových sítí se obecně skládá z následujících typů vrstev:

- konvoluční,
- sdružovací (někdy označováno jako poolingová),
- plně-propojené.

Konvoluční vrstva

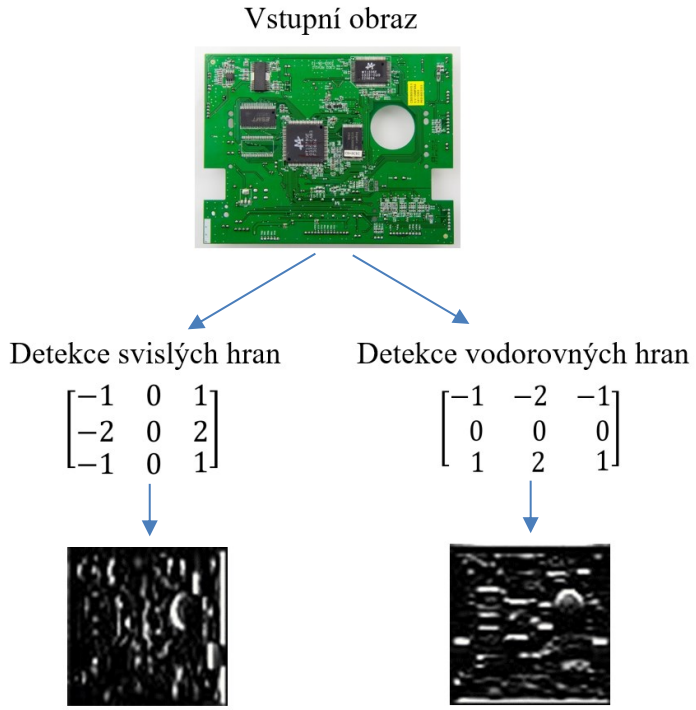
Konvoluční vrstva je základní součástí konvolučních neuronových sítí. Tato vrstva hraje zásadní roli při detekci příznaků. V konvoluční vrstvě se na vstupní data aplikuje řada filtrů (někdy označovány jako kernely). Každý filtr v konvoluční vrstvě je určen k detekci specifických rysů, jako jsou hrany, rohy nebo textury objektů na vstupních obrazech. Aplikací filtru na data se provede maticový součin určité oblasti dat, které filtr „pokrývá“. Poté dojde k součtu všech prvků ve výsledné matici a tato hodnota je uložena (viz obrázek 1.6).

Následně je oblast filtru posunuta o určitý počet kroků a opět proveden maticový součin s danými daty. Výsledkem aplikace filtru na celý obraz je mapa příznaků. Tato mapa ukazuje,



Obrázek 1.6 – Princip konvoluce nad vstupním obrazem (Baskin, 2017)

kde a v jaké míře se na vstupu nacházejí určité prvky, jako jsou horizontální a vertikální hrany. Použitím více filtrů přizpůsobených různým rysům může konvoluční vrstva vytvořit komplexní reprezentaci vstupu a zachytit tak detaily, které nemusí být na originálním obraze patrné (viz obrázek 1.7).



Obrázek 1.7 – Výsledek aplikace filtrů na vstupní obraz

Při výpočtu konvoluce filtru a vstupního obrazu je zapotřebí brát v potaz dva parametry – krok (stride) a výplň (padding). Krok určuje, jak moc se filtr pohybuje po vstupu.

Menší krok vede k hustšímu vzorkování vstupu, a tím k vytváření větších map příznaků. Standardně se hodnota kroku volí v rozmezí 1 až N, kde N je rozměr filtru.

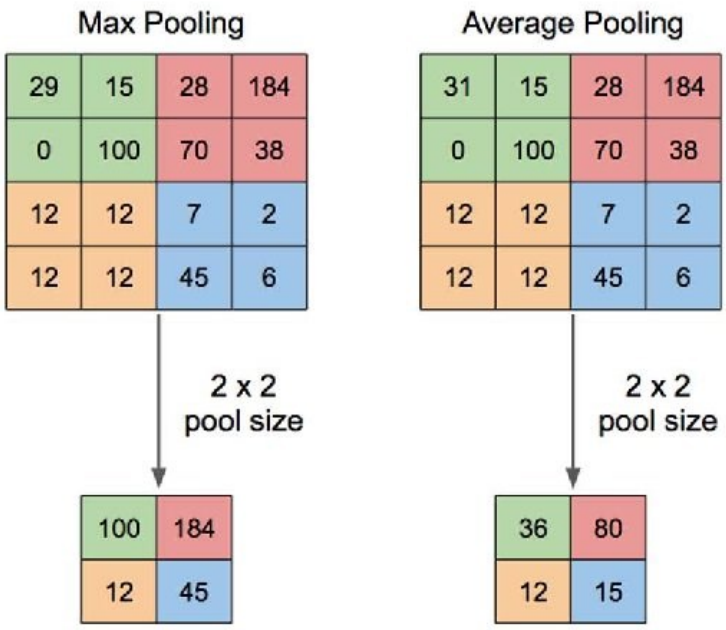
Jak je patrné z obrázku 1.6, může při konvoluci obrazu docházet ke ztrátě informace na jeho okrajích. Pokud aplikujeme na obraz o M x M rozměrech filtr o rozměrech N x N, bude mít výsledný obraz rozměry zmenšené o N-1, tedy M-(N-1). V některých případech tak může dojít k velkému zkreslení objektů nacházejících se na okrajích obrazu. Výplň zahrnuje přidání příslušného počtu řádků a sloupců k okrajům vstupu. Hodnoty obsažené ve výplni záleží na typu aplikace, nejčastěji se však používá buď „prázdná“ výplň, kde všechny hodnoty výplně jsou rovny 0, nebo je hodnotám výplně přiřazena hodnota rovnající se hodnotě nejbližšího pixelu vstupního obrazu. Tím je zajištěno, že rozměry výstupního obrazu nejsou změněny, tedy nedochází ke ztrátě informace na jeho okrajích.

Sdružovací vrstva

Sdružovací vrstva v konvolučních neuronových sítích slouží pro snížení rozměrů (někdy označováno jako dimenzionalita) obrazu a zároveň zachování jeho základních informací. Tato vrstva se obvykle používá ke snížení výpočetní složitosti a zmírnění nadměrného přeučování sítě. Principem sdružovací vrstvy je sjednocení informace v rámci malých lokálních oblastí vstupního obrazu. Sdružovací vrstvy obvykle pracují s malou velikostí filtru, například 2x2 nebo 3x3, a často používají krok 2, což znamená, že dochází k posunu filtru o dva pixely. Velikost filtru a krok lze upravit na základě konkrétních požadavků sítě a zpracovávaných dat. V současné době jsou nejčastěji používané dva principy sdružování, označované jako max pooling a average pooling.

Sdružování metodou max pooling vybírá maximální hodnotu z každé lokální oblasti vstupní mapy. Pro sdružovací filtr o rozměrech 2×2 je na originální obrazu vybrána čtvercová oblast čtyř pixelů. Následně je do nového obrazu uložen pixel s nejvyšší hodnotou mezi touto čtveřicí. Princip sdružování pomocí max poolingu je znázorněn na obrázku 1.8

Metoda average pooling naproti tomu počítá průměr hodnot pixelů v rámci každé oblasti. Tento postup mírní ztrátu informace tím, že na výsledný obraz mají vliv všechny rysy v jednotlivých oblastech, místo aby se zaměřoval jen na ten nejvýraznější. Sdružování metodou average poolingu je znázorněno na obrázku 1.8.



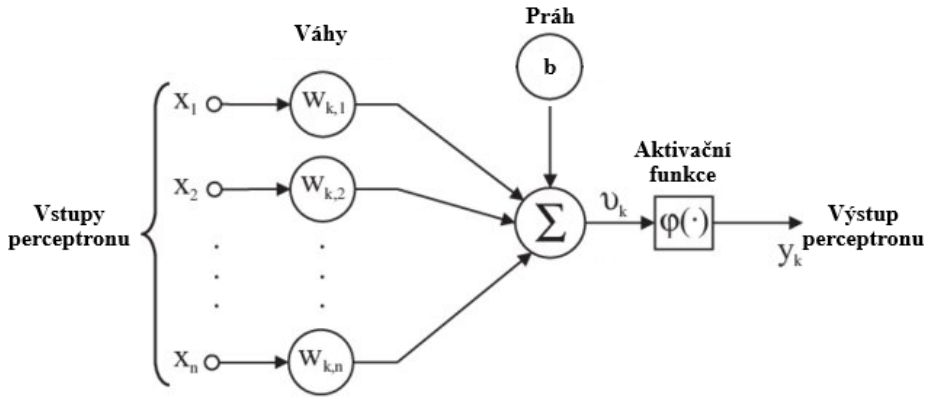
Obrázek 1.8 – Principy sdružovacích metod (Yani, 2019)

Sdružování snižuje rozměry vstupního obrazu, což nejen snižuje výpočetní nároky, ale také snižuje pravděpodobnost přeučení sítě. Díky sdružování jsou do jisté míry (v závislosti na velikosti použitého filtru) vyrušeny vlivy posunutí objektu. Výstup sítě se tak výrazně nezmění, pokud dojde k posunu objektu na vstupním obrazu. Při použití této vrstvy v neuronové síti za vrstvou konvoluční dochází zobecňování naučených funkcí. Metoda max poolingu zachovává nejvýraznější rysy na vstupním obrazu. Často se jedná o identifikační prvky objektu, jako jsou hrany nebo textury. Průměrné sdružování naopak všechny příznaky prvků vyhlazuje a zachovává obecné vzory.

Plně-propojená vrstva

Základní jednotkou plně propojené vrstvy je perceptron, někdy také označovaný jako neuron. Perceptron v neuronové síti je v podstatě matematická funkce, která přijímá několik vstupních hodnot, zpracovává je a vytváří výstup. Skládá z následujících složek:

- Vstup: Hodnoty, které pocházejí ze vstupních dat nebo z výstupů předchozích vrstev.
- Váhy: Každý vstup má přiřazenou váhu. Váhy určují důležitost každého vstupu pro výstup neuronu.
- Práh: Práh je dodatečný parametr, který určuje důležitost daného neuronu v síti.
- Aktivační funkce: Aktivační funkce aplikuje nelineární transformaci na výstup.
- Výstup: Výsledná hodnota, dále předávána jako vstup dalším neuronům nebo jako konečný výstup sítě.



Obrázek 1.9 – Model perceptronu (Zanetti, 2008)

Během trénování se perceptrony „učí“ pomocí úpravy hodnot vah a prahů tak, aby minimalizovaly rozdíl mezi predikcí sítě a skutečným výstupem. Tento proces zahrnuje techniky, jako je zpětné šíření chyby a gradientní sestup.

Využitím jediného perceptronu není možné řešit nelineární problémy. Proto se perceptrony řadí do sítě, které zahrnují více vrstev neuronů. Taková síť se nazývá plně – propojená někdy označovaná také jako hustá. Tato síť je základní součástí neuronových sítí, užitečná zejména v modelech hlubokého učení, jako jsou konvoluční neuronové sítě. V plně propojené vrstvě je každý neuron propojen s každým neuronem v předchozí vrstvě. Hlavní úlohou plně propojené vrstvy je provádět závěrečné fáze učení a odvozování v neuronových sítích.

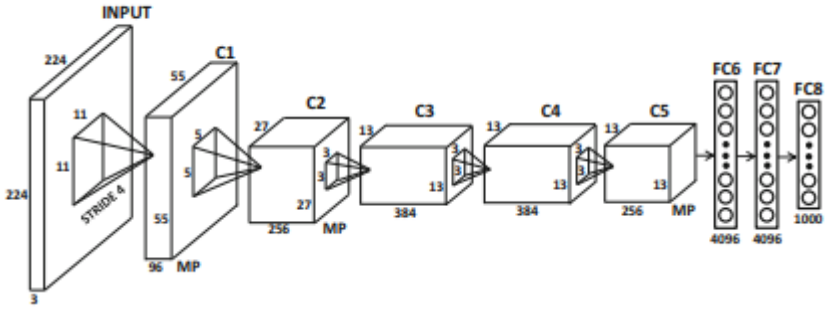
1.2.2 DRUHY KONVOLUČNÍCH NEURONOVÝCH SÍTÍ

Potřeba různých typů konvolučních neuronových sítí často vyplývá ze specifických požadavků aplikací, pro které jsou navrženy. Klasifikace obrazu, detekce objektů a sémantická segmentace představují různé problémy a vyvinout jednu obecnou síť, která by řešila vše je jak výpočetně, tak technicky náročné. Zatímco CNN určená pro klasifikaci obrazu se zaměřuje na identifikaci globálních vzorů, síť vytvořená pro sémantickou segmentaci musí klasifikovat každý pixel zvlášť. Tyto úlohy vyžadují odlišné architektonické návrhy, což vedlo k vývoji různých typů CNN.

Při utváření architektur CNN hrají významnou roli také charakteristiky dat. Povaha vstupních dat, jako je rozlišení, barevné kanály a úroveň šumu, výrazně ovlivňuje návrh sítě. Například zpracování satelitních snímků s vysokým rozlišením může vyžadovat robustnější architektury pro zachycení jemných detailů, zatímco jednodušší snímky ve stupních šedi lze zpracovat pomocí menších sítí.

Konvoluční neuronová síť (CNN)

Typická architektura CNN se skládá ze vstupní vrstvy, několika skrytých vrstev (včetně konvolučních, aktivačních a sdružovacích vrstev) a výstupní vrstvy. Konvoluční vrstvy aplikují na vstup filtry a zachycují prostorové povahy rysů. Aktivační vrstvy, obvykle ReLU (Rectified Linear Units), zavádějí nelinearity, které pomáhají učit síť složité vzory. Spojovací vrstvy snižují velikost a výpočetní náročnost a plně propojené vrstvy klasifikují vstup do kategorií na základě příznaků vytvořených konvolučními vrstvami. Díky své architektuře jsou CNN vysoce efektivní pro úlohy rozpoznávání obrazu, protože jsou schopny zachytit lokální závislosti objektů a prostředí. Konvoluční vrstvy snižují počet parametrů zapojených do sítě, čímž snižují výpočetní zátěž. CNN je mimořádně efektivní při učení příznaků. Na druhou stranu vyžadují značné množství dat, aby bylo možné je efektivně trénovat a vyhnout se přeučení sítě. Ze své podstaty jsou omezeny na zpracování vstupů pevných velikostí a jsou výpočetně náročné, zejména u hlubších sítí. (Aggarwal, 2018)

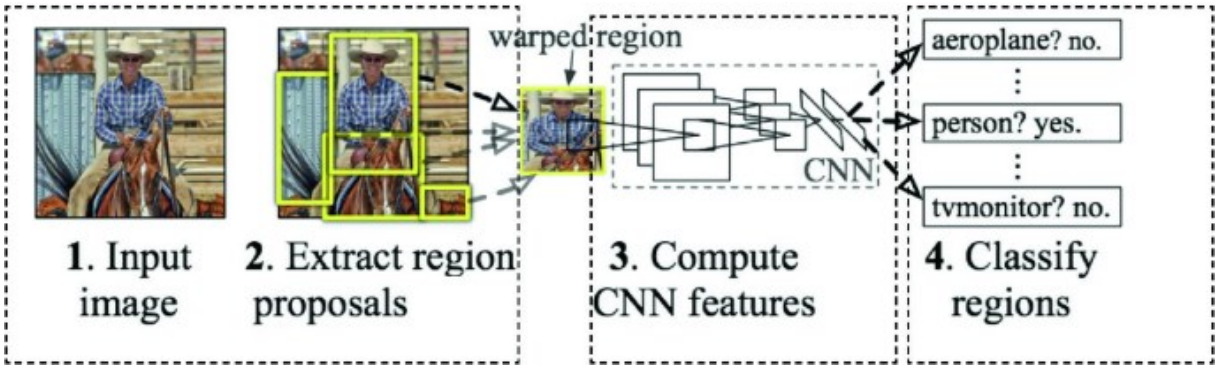


Obrázek 1.10 – Obecná architektura CNN (Aggarwal, 2018)

Regionální konvoluční síť (R-CNN)

R-CNN jsou rozšířením klasické CNN, které do procesu přidávají krok návrhu regionu, aby zvládly úlohu detekce objektů. R-CNN nejprve vytvoří potenciální ohraničující boxy v obraze (návrhy regionů) a poté na těchto regionech aplikují algoritmy pro detekci objektů. Každá oblast je deformována do čtverce, aby byla zajištěna pevná velikost vstupu pro CNN, a poté je vedena přes řadu konvolučních a plně propojených vrstev, aby byla oblast klasifikována. R-CNN výrazně zlepšují přesnost modelů detekce objektů, protože kombinují silné stránky vysokokapacitních modelů s návrhy oblastí. Tato metoda je vysoce účinná v aplikacích, kde je zapotřebí zajistit přesnou lokalizaci objektů. Hlavní nevýhodou R-CNN je jejich rychlost.

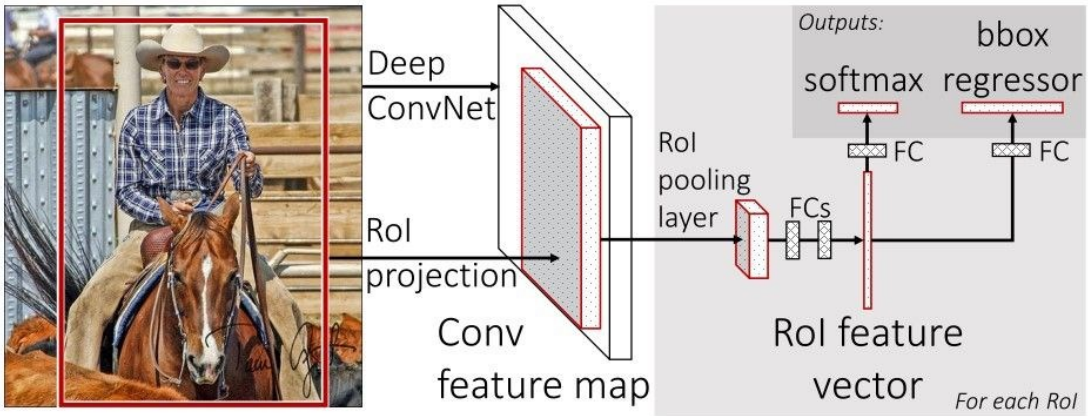
Každý návrh oblasti musí být zpracován samostatně, což může být výpočetně náročné a pomalé. Proto jsou R-CNN méně vhodné pro aplikace v reálném čase.



Obrázek 1.11 – Princip architektury sítě R-CNN (Ghosh, 2019)

Rychlá regionální konvoluční síť (FAST R-CNN)

Rychlá R-CNN vylepšuje původní návrh sítě. Namísto samostatného použití CNN pro každou oblast používá rychlá R-CNN sdílenou konvoluční síť pro celý obraz. Vrstva pro sdružování oblastí zájmu (Region of Interest - RoI) vytvoří z příznaků pro každou oblast vektor rysů s pevnou velikostí. Tyto příznakové vektory jsou pak přivedeny do řady plně propojených vrstev, které určí predikci třídy. Rychlá R-CNN, jak už název vypovídá, je výrazně rychlejší než R-CNN, protože zjednodušuje proces trénování tím, že umožňuje použití jediné konvoluční neuronové sítě pro celý obraz. Tím se snižuje redundance a zvyšuje rychlost trénování. Návrh rychlé R-CNN sice nabízí zlepšení oproti původní R-CNN, ale stále se spoléhá na algoritmus selektivního vyhledávání při navrhování oblastí, což může vysoký dopad na rychlost a přizpůsobivost učení. Navíc stále vyžaduje značné množství výpočtů, což nemusí být optimální pro systémy pracující v reálném čase. (Ghosh, 2019)



Obrázek 1.12 – Princip architektury sítě Fast R-CNN (Ghosh, 2019)

1.3 PRŮMYSLOVÉ SYSTÉMY STROJOVÉHO VIDĚNÍ

Systémy strojového vidění jsou automatizované systémy určené k analýze vizuálních dat, především pro kontrolu kvality a automatizaci ve výrobě a průmyslu. Tyto systémy jsou důležitou součástí moderní průmyslové automatizace a hrají roli při zvyšování produktivity a zajišťování kvality výrobků. Základním prvkem je soustava jedné nebo více kamer, které snímají danou oblast či objekt. V závislosti na aplikaci se tyto kamery mohou lišit svými specifikacemi, jako je rozlišení, snímková frekvence a citlivost. Pro kvalitu a konzistenci obrazu je zásadní správné osvětlení. Osvětlovací techniky jsou často přizpůsobeny konkrétním požadavkům systému vidění a pomáhají zvýraznit příslušné prvky nebo minimalizovat odlesky a stíny.

Obrázek 1.13 – Metody osvětlení objektu (Štursa, 2024)

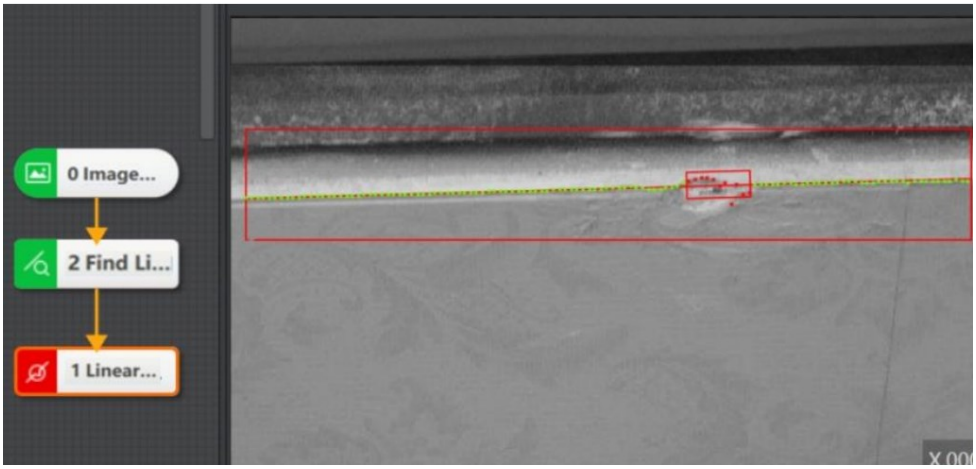
Systémy průmyslového vidění často sdělují své výsledky ostatním systémům, jako jsou robotická ramena či řídící systémy. Tato komunikace často probíhá pomocí standardních protokolů, jako je Modbus, Ethernet IP či sériová komunikace.

Systémy průmyslového vidění lze obecně rozdělit na základě jejich funkcí a použití na systémy 2D vidění a systémy 3D vidění. Systémy 2D vidění zachycují ploché, dvourozměrné obrazy. Běžně se používají pro aplikace, jako je kontrola povrchových vad, čtení čárových kódů a navádění jednoduchých robotických pohybů. Systémy 3D vidění zachycují prostorové informace o objektu, což umožňuje podrobnější analýzu. Jsou užitečné pro aplikace, jako je komplexní navádění robotů, objemová měření a přesná kontrola trojrozměrných struktur.

1.3.1 SYSTÉMY 2D ROBOTICKÉHO VIDĚNÍ

Systémy 2D robotického vidění se v současné době stávají často používaným nástrojem v moderním průmyslu a automatizaci. Tyto systémy ve standardně pracují s jednou kamerou a zdrojem světla, což z nich činí relativně jednoduché a cenově dostupné řešení pro širokou škálu aplikací. Hlavní výhodou těchto systémů je jejich schopnost zpracovávat obrazové informace a analyzovat "ploché" vlastnosti, jako jsou barva, tvar a velikost snímaného objektu. Kromě toho mohou systémy identifikovat defekty na povrchu objektu, počítat počet objektů na obraze, číst čárové a QR kódy nebo rozpoznávat text. Tato technologie je zásadní pro automatizované výrobní procesy, kde umožňuje kontrolu kvality a třídění produktů. Systémy dvourozměrného robotického vidění poskytují vysokou míru přesnosti a spolehlivosti, což je důležité zejména v odvětvích, kde jsou kladeny požadavky na kvalitu a rychlost řešení. Důležitou vlastností je také jejich flexibilita, díky programování je lze snadno přizpůsobit různé aplikace a scénáře.

Jednou z nejčastějších aplikací robotických systémů vidění je kategorizace a porovnávání objektů s předem naučenými vzorovými předměty. Tyto funkce umožňují robotům rozpoznat a analyzovat objekty na základě předem definovaných nebo naučených vzorů. V praxi je toto porovnávání objektů využíváno například k identifikaci konkrétních součástek na montážní lince nebo ke kontrole kvality výrobků. Systém srovnává aktuální obraz s referenčním vzorem a zjišťuje, zda objekt odpovídá požadovaným kritériím. Tato metoda je často používaná pro detekci chyb nebo vad, které mohou být obtížně zjistitelné jinými způsoby. Kategorizace pak umožňuje systémům rozdělit objekty do různých tříd na základě jejich podobnosti. To je užitečné například v logistice, kde mohou být produkty automaticky tříděny podle typu, velikosti nebo jiných charakteristik. Tento proces je často prováděn na základě vizuálních rysů, které jsou extrahovány z obrazu a následně porovnány s předem definovanými kategoriemi.

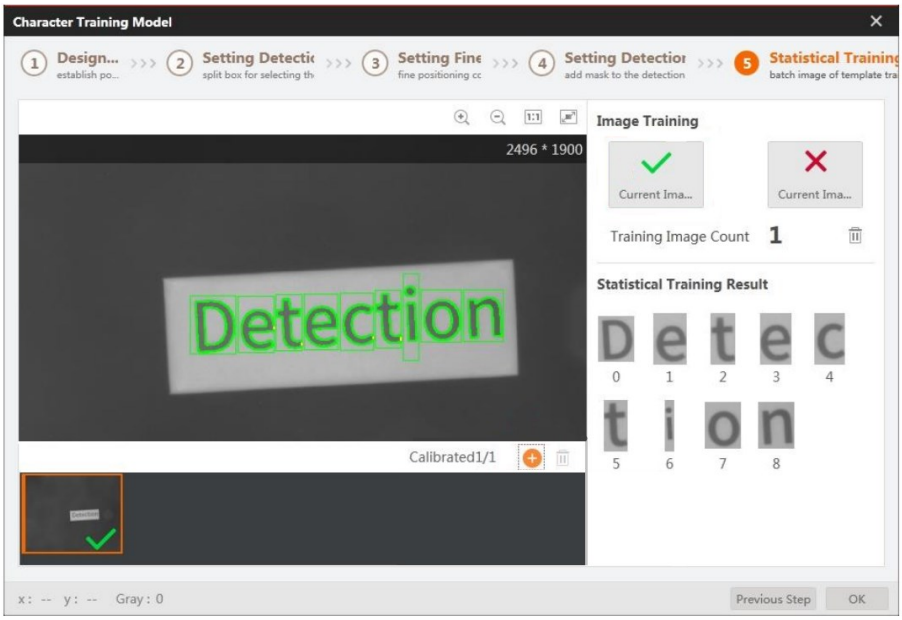


Obrázek 1.14 – Zjišťování defektů (DobotVisionStudio, 2024)

V některých aplikacích je zapotřebí rozlišit mezi různými kategoriemi objektů. Identifikace objektů je používána například při montáži a balení produktů, kde je důležité, aby systém rozpoznal konkrétní díly. Systémy robotického vidění mohou díky identifikaci objektů také kontrolovat, zda jsou součásti většího celku na správných pozicích a zda jsou splněny všechny požadavky na kvalitu výrobku. Identifikace objektů je také nezbytná pro logistické aplikace, kde umožňuje systémům sledovat a třídit zboží. V takových aplikací často dochází k rozpoznávání identifikačních značek, jako jsou čárové kódy nebo QR kódy.

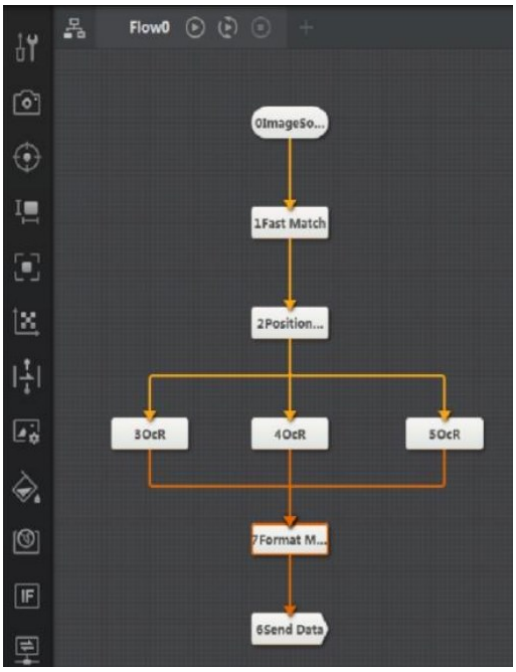
Obrázek 1.15 – Rozpoznávání objektů (DobotVisionStudio, 2024)

Optické rozpoznávání znaků (Optical Character Recognition, OCR) je dalším příkladem aplikace pro dvourozměrné systémy robotického vidění. OCR umožňuje počítačovým systémům extrahovat a interpretovat text z obrazů. Tato technologie je široce využívána k automatizaci procesů, které vyžadují zpracování velkého množství textových informací, a tím zvyšuje efektivitu a přesnost. OCR se často používá při zpracování dokumentů, kde umožňuje převádět tištěný nebo ručně psaný text do digitální podoby. V průmyslových aplikacích se OCR často používá ke sledování a identifikaci produktů podle tištěných nebo vyražených označení. Často je OCR využíváno pro sledování a kontrolu výrobních čísel, dat výroby nebo jiných identifikačních informací.



Obrázek 1.16 – OCR (DobotVisionStudio, 2024)

Aby byly systémy vidění co možná nejflexibilnější a dokázaly se přizpůsobit velké škále aplikací, je často nutné zajistit vývoj programu. Uživatel má ne výběr buď obdržet systém s již naprogramovanou úlohou své aplikace. Toto řešení je ovšem často cenově méně efektivní než vlastní vývoj. Proto je ve většině případů součástí dodávky systému robotického vidění i vývojový software, díky kterému si uživatel může svépomocí naprogramovat vlastní aplikaci tak, aby přesně splňovala požadavky. Dříve bylo standardem programování formou psaného kódu, také označováno jako Structured Text (ST). V dnešní době mnoho výrobců přechází na formu programování pomocí funkčních bloků kódu (FBD). Tato forma programování má řadu

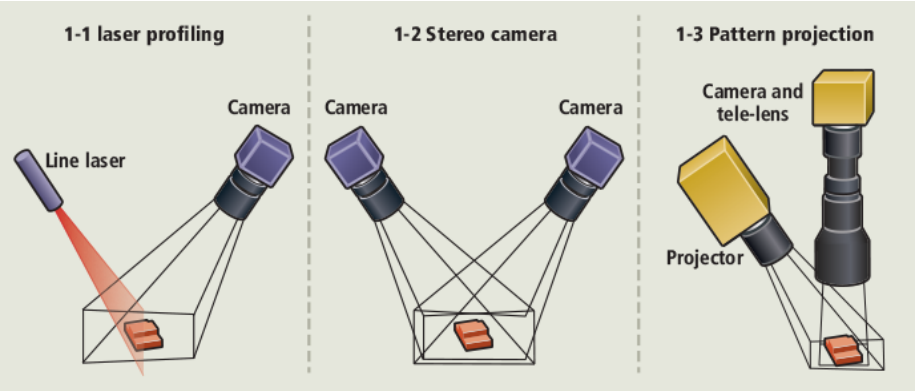


Obrázek 1.17 – Programování funkčních bloků (DobotVisionStudio, 2024)

výhod oproti klasickému způsobu. Hlavní výhodou spočívá v jeho jednoduchosti, není zapotřebí se učit syntaxi žádného programovacího jazyku. Tento způsob je tak přístupnější většímu množství uživatelů.

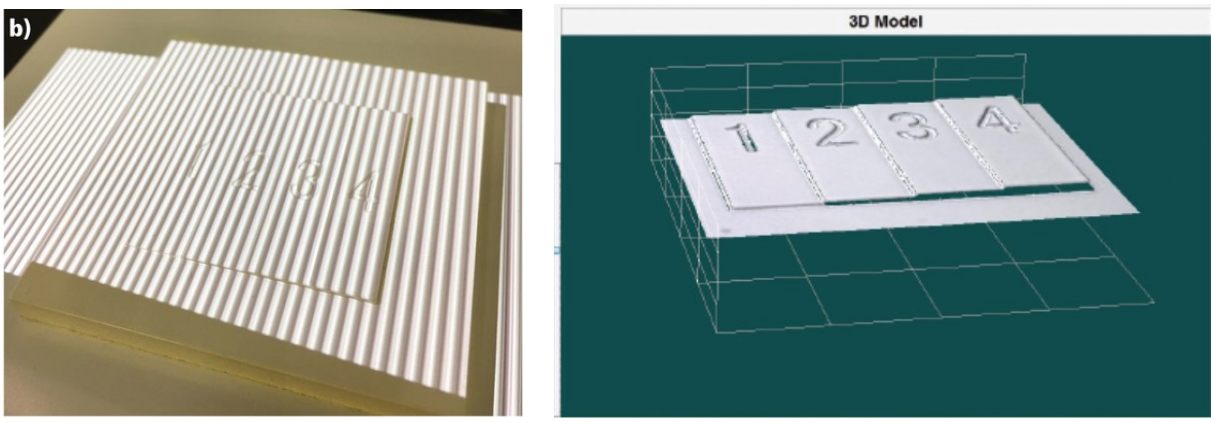
1.3.2 SYSTÉMY 3D ROBOTICKÉHO VIDĚNÍ

Pro pokročilejší aplikace, které vyžadují kromě klasických dvou rozměrů x a y ještě rozměr hloubky (z), je možné využít systémů 3D robotického vidění. Základem 3D vidění je schopnost snímat hloubku, což umožňuje robotům vnímat svět v trojrozměrné perspektivě. Existuje několik způsobů, jak mohou systémy získávat tyto informace.



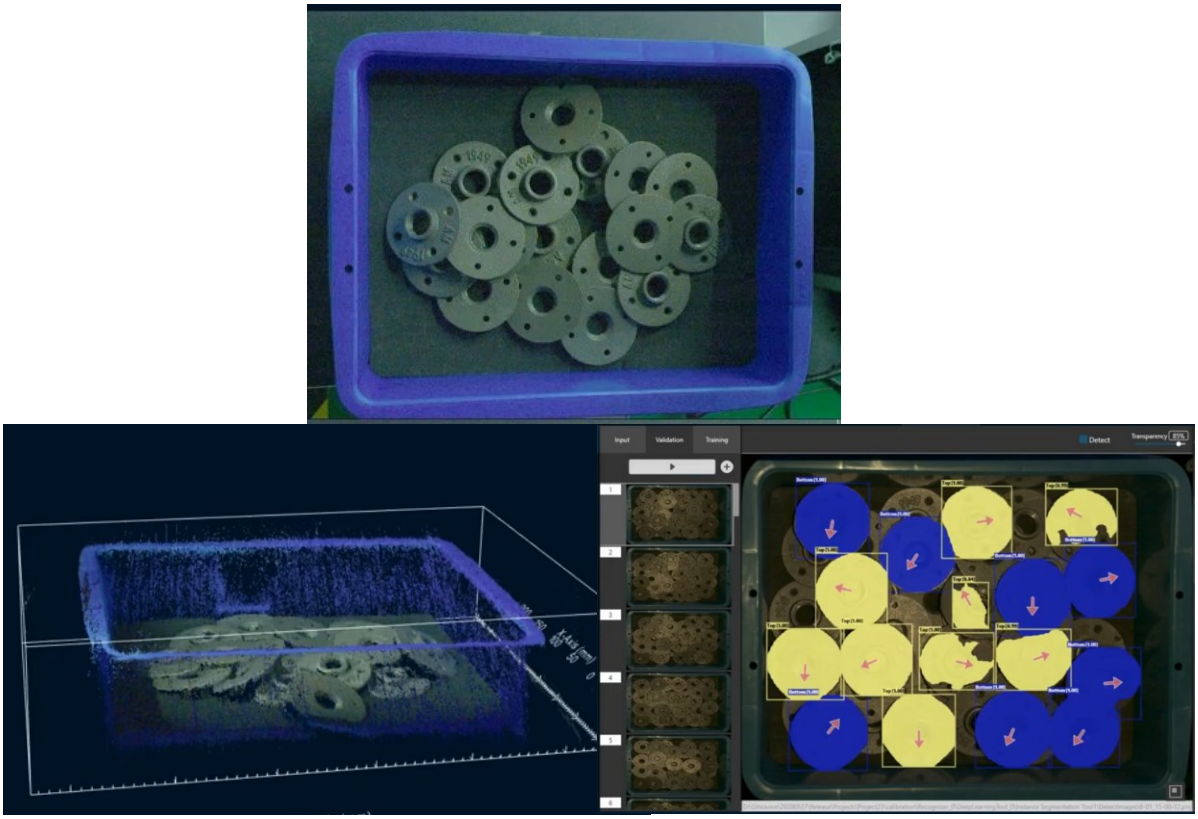
Obrázek 1.18 – Principy 3D snímání (Iwamoto, 2018)

Jednou z technik je stereoskopické vidění, které využívá dvě kamery k zachycení snímků z různých úhlů, podobně jako lidské oči. Srovnáním těchto snímků mohou systémy odhadnout vzdálenost a prostorové vztahy mezi objekty. Další metodou je strukturované světlo, kde se na scénu promítá vzor a kamera zaznamenává jeho deformace způsobené tvary objektů. Tímto způsobem lze vytvořit detailní 3D mapu povrchů. Alternativně lze použít metodu "time-of-flight", kde kamera měří dobu, kterou potřebuje světlo k tomu, aby se odrazilo od objektů a vrátilo se zpět, což umožňuje přesné měření vzdáleností. (Iwamoto, 2018)



Obrázek 1.19 – Rekonstrukce objektu pomocí projekce (Iwamoto, 2018)

Trojrozměrné vidění umožňuje robotům provádět řadu úkolů, které nejsou možné s 2D viděním. Jednou z hlavních aplikací je přesná manipulace s objekty. Díky schopnosti vnímat hloubku mohou roboty přesně uchopit předměty, které nemají stejnou orientaci, což je zásadní v průmyslové automatizaci, například při aplikacích typu bin-picking, montáži nebo kompletaci balení. Další aplikací, kde lze 3D vidění využít, je navigace v systému v prostoru. Řídicí systémy tak mohou analyzovat trojrozměrnou strukturu svého okolí a vyhnout se překážkám, což je klíčové pro autonomní roboty a vozidla. Navíc systémy robotického 3D vidění umožňují provádět všechny druhy aplikací, kterých lze docílit se systémy 2D (popsané v kapitole 3.1).



Obrázek 1.20 – Příklad aplikace bin-picking (Accupick, 2024)

Programování těchto systémů je komplikovanější, než tomu bylo u systémů 2D vidění. I zde je opět trend programování pomocí funkčních bloků. Jsou zde však kladeny větší nároky na učení rozpoznávaných objektů. Zatímco u 2D systémů stačí obrazová data s různou rotací a posunutím objektů, kvůli přidání třetího rozměru snímání je nutné u 3D systémů pořídit daleko větší množství obrazů. Mnoho výrobců tak přidává možnost nahrání trojrozměrného modelu, s podporou různých modelovacích programů, přímo do vývojového softwaru. Systém je pak schopen se sám naučit, jak daný objekt vypadá. Tuto metodu však nelze využít tehdy, není-li možné zaručit podobnost rozpoznávaných objektů.

2 PRAKTICKÁ ČÁST

Tato část práce se zabývá vývojem a aplikací vlastních algoritmů konvoluční neuronové sítě. Standardně používané knihovny, které jazyk Python nabízí, jako jsou NumPy, Keras či TensorFlow, mohou na běžného uživatele působit jako takzvané „černé skříňky“, kdy nemusí být zcela zřejmé, jakým způsobem fungují. Proto byly pro potřeby této práce vyvinuty vlastní funkce, které mají za účel nahradit výše zmíněné knihovny. Výjimkou je využití několika funkcí z knihovny Pillow pro otevírání obrazových souborů.

Samotná struktura programu byla rozdělena do čtyř funkčních bloků – tříd. Každá třída se zabývá jen určitou částí problematiky aplikací robotického vidění. Třída matematických funkcí obsahuje maticové operace, aktivační funkce a výpočty odchylek predikcí sítě. Účelem třídy pro předzpracování obrazu je úprava vstupních obrazových dat do formátu požadovaného vstupní vrstvou neuronové sítě. Dále se zde nacházejí funkce pro generování učícího datasetu. Třída pro samotnou konvoluční neuronovou síť obsahuje deklarace konvoluční, vyhlazovací a plně-propojené vrstvy. Každá vrstva je vnořenou třídou celého souboru a obsahuje funkce pro dopředné šíření predikce i zpětné šíření chyby a gradientů učení. Poslední třída se zabývá komunikací výsledků predikce neuronové sítě s robotickým systémem. Jsou zde popsány dva standardní komunikační protokoly – sériová komunikace a protokol Modbus TCP/IP.

Navržené řešení stavby neuronové konvoluční sítě je otestováno na třech typech aplikací. První aplikace se zabývá rozpoznáváním dvou relativně odlišných tříd objektů. Tento test má za účel zhodnotit obecnou funkčnost vytvořeného algoritmu pro standardní využití konvolučních neuronových sítí. V druhé aplikaci jsou rozpoznávány dvě třídy podobně vypadajících objektů, čímž je otestována schopnost sítě učit se na vstupních obrazových datech detailní příznaky. Poslední aplikace testuje chování sítě při rozpoznávání většího množství tříd objektů, než tomu bylo v předchozích zadáních.

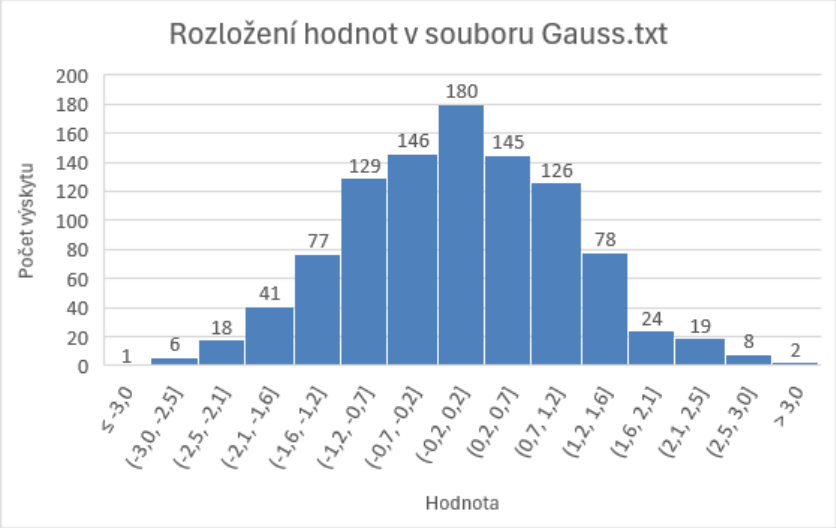
2.1 MATEMATICKÉ FUNKCE

V oblasti strojového učení výběr výpočetních nástrojů a metod může výrazně ovlivnit rychlost a kvalitu učení neuronových sítí. Tato kapitola se zabývá programováním vlastních matematických funkcí přizpůsobených pro výpočetní operace konvolučních neuronových sítí. Knihovny, jako je například NumPy, nabízejí řadu optimalizovaných matematických funkcí, které jsou univerzální a efektivní. Tyto knihovny však často připomínají "černé skříňky" - zatímco poskytují vysoký výkon, principy a použité matematické metody nemusí být pro běžného koncového uživatele zcela zřejmé. Ačkoliv je možné dohledat popisy funkcí v instrukčním manuálu přímo od tvůrců této knihovny, struktura datového pole je zcela odlišná od standardního pole používaného v jazyce Python. Tato nejistota může být překážkou v pochopení detailů matematických operací a jejich implementace v neuronových sítích. V této práci jsou proto základní principy a matematické funkce, jimiž se konvoluční neuronové sítě řídí, implementovány prostřednictvím programování vlastních funkcí. Vlastní sada matematických funkcí zahrnuje základní operace s maticemi, jako je jejich sčítání či násobení, konvoluční operace nebo transformace otočení obrazu.

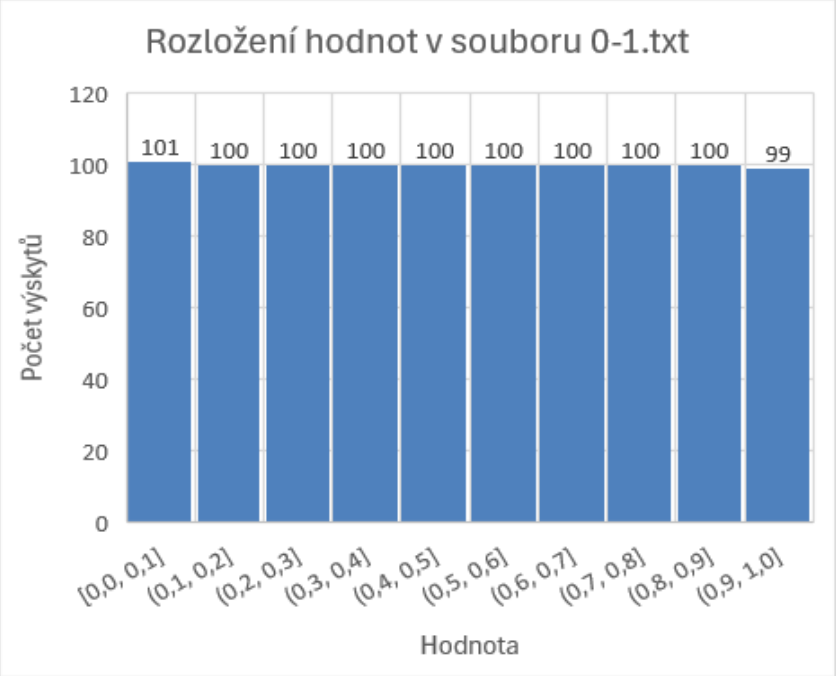
2.1.1 FUNKCE RANDOM

Funkce Random je určena ke generování seznamu pseudo-náhodných čísel z předem definovaných datových sad na základě vygenerovaného seznamu. Do funkce vstupují dva parametry – počet požadovaných náhodných hodnot ‚n‘ a jméno souboru, ze kterého má funkce čísla přečíst. Díky tomu je možné využít tuto funkci pro generování náhodných čísel s určitým statistickým rozložením. V této práci jsou využity dva seznamy, se kterými tato funkce při vracení náhodných čísel pracuje – lineární rozložení na intervalu <0; 1> a Gaussovo rozložení.

Funkce začíná určením datové sady, která má být použita, na základě zadaného souboru. Pokud je vybrán režim "Gauss", načte se ze souboru, který pravděpodobně obsahuje čísla s Gaussovým rozdělením (viz obr. 4.1). V opačném případě se využijí hodnoty ze souboru, který obsahuje rovnoměrně rozdělená čísla mezi 0 a 1 (viz obr. 4.2). Oba soubory obsahují tisíc náhodně vygenerovaných hodnot, ze kterých program následně čísla vrací. Tento počet se pro tuto aplikaci ukázal jako dostatečný. Pokud by bylo zapotřebí generovat hodnoty z jiného statistického rozložení dat, je možné jednoduše přidat nový předpřipravený seznam do stávajícího programu. Pro vygenerování seznamů s lineárním a Gaussovým rozložením byla použita funkce random ze stejnojmenné knihovny.



Obrázek 2.1 – Rozložení hodnot, pokud *mod* = "Gauss"



Obrázek 2.2 – Lineární rozložení hodnot

Po výběru vhodného souboru funkce načte všechna čísla z tohoto souboru do nově vytvořeného pole. Následně přepíše vybraný soubor tak, aby právě generované hodnoty byly zapsány na konci tohoto souboru. Díky tomu nedochází k vracení stejných hodnot při následných voláních této funkce. Funkce nejdříve zapíše hodnoty od řádku s posledním vraceným číslem (tedy pokud funkce vrací pět čísel, začne zápis od řádku 6) až do konce seznamu hodnot. Po dosažení konce seznamu pokračuje od začátku až do výchozího bodu. Tento způsob přeskupování čísel slouží k cyklickému využívání dat tak, aby nedocházelo ke zbytečnému opakování stejných hodnot.

```
with open(soubor, "w") as f:
    for i in range(n%len(obsah), len(obsah)):
        f.write(obsah[i])
    for i in range(n%len(obsah)):
        f.write(obsah[i])
```

Obrázek 2.3 – Přepsání vybraného souboru

Generování výstupu závisí na tom, zda počet požadovaných náhodných hodnot převyšuje množství dat načtených ze souboru. Pokud je potřeba vracet čísel méně, než je v souboru k dispozici, použije se k sestavení konečného výstupu pouze dané množství hodnot, přičemž se každý řetězec převede na datový typ float, aby se usnadnily další výpočty. Pokud je požadováno více hodnot, než je v souboru, seznam se opakovaně připojuje k sobě samému, aby bylo splněno požadované množství, a v případě potřeby se doplní případný zbytek stejným způsobem, jako v předchozím případu.

```
vystup = []
if(n>len(obsah)):
    for i in range(len(obsah)):
        obsah[i] = float(obsah[i].replace('\n',''))
    for i in range(n//len(obsah)):
        vystup.extend(obsah)
    for i in range(n%len(obsah)):
        vystup.append(obsah[i])
else:
    for i in range(n%len(obsah)):
        vystup.append(obsah[i])
    for i in range(len(vystup)):
        vystup[i] = float(vystup[i].replace('\n',''))
return vystup
```

Obrázek 2.4 – Vrácení požadovaných pseudo-náhodných hodnot

2.1.2 STANDARDNÍ MATICOVÉ OPERACE

Při počítání matematických funkcí při učení a testování konvoluční neuronové sítě nemusí mít vždy pole vah a prahů shodný rozměr se vstupními poli dat. V takovém případě by nebylo možné korektně provádět mezi těmito poli matematické funkce. Metoda

prvky_v_posledni_dimensi() řeší tento problém tak, že vrací dvourozměrné pole s prvky, které se nachází v poslední dimenzi vnořeného pole.

```
def prvky_v_posledni_dimensi(pole):
    if isinstance(pole, list):
        if isinstance(pole[0], list):
            # Pokud prvek je seznam, rekurzivně voláme funkci na každém prvku
            return [elem for sub_pole in pole for elem in MyMath.prvky_v_posledni_dimensi(sub_pole)]
        else:
            # Pokud prvek není seznamem, jsme na konci dimenze, vrátíme prvek
            return [pole]
    else:
        raise ValueError("Input není ve správném formátu, očekává se seznam.")
```

Obrázek 2.5 – Funkce pro vracení prvků v poslední dimenzi pole

Funkce nejprve zkontroluje, zda je vstupní proměnná pole. Pokud není, vyvolá chybu ValueError, která uživatele upozorní, že vstup není v očekávaném formátu. Je-li vstupní proměnná skutečně pole, funkce zkontroluje, zda je její první prvek také pole. V případě, že první prvek polem je, znamená to, že se jedná o vnořený seznam. Funkce rekurzivně zavolá sama sebe pro každý podseznam (sub_pole) nalezený ve vstupní proměnné. Tato rekurze probíhá, dokud nedosáhne poslední úrovně, kde se již žádné další podseznamy nevyskytují. Na této poslední úrovni se nachází užitečné hodnoty pro další výpočty. V případě, že vstupní proměnná již neobsahuje vnořený seznam, znamená to, že funkce dosáhl posledního rozměru vstupní proměnné. Funkce proto vrátí seznam obsahující aktuální prvky. Tato metoda efektivně zploští všechny prvky poslední dimenze do jediného seznamu bez ohledu na to, jak vnořený byl původní seznam.

Maticový součet po prvcích

Pro součet dvou matic byla vytvořena funkce sum. Tato funkce provádí součet po prvcích, tedy každý prvek matice A je sečten s prvkem na korespondující pozici v matici B. Tato operace je označena jako $\oplus$:

$$A \oplus B = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \cdots & a_{1m} + b_{1m} \\ a_{21} + b_{21} & a_{22} + b_{22} & \cdots & a_{2m} + b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} + b_{n1} & a_{n2} + b_{n2} & \cdots & a_{nm} + b_{nm} \end{pmatrix} \tag{2.1}$$

$$c_{ij} = a_{ij} + b_{ij} \tag{2.2}$$

kde: A je vstupní matice,

B je vstupní matice, která má být přičtena k matici A,

a_{ij} je prvek matice A na pozici i, j,

b_{ij} je prvek matice B na pozici i, j,

c_{ij} je prvek výsledné matice, kterou funkce vrací.

Jak je z rovnice 2.1 zřejmé, je potřeba zajistit, aby obě matice měly shodné rozměry. Funkce nejprve použije výše popsanou metodu prvky_v_posledni_dimensi k uložení prvků z posledních dimenzí obou matic *a*, *b* do proměnných *pe_a* a *pe_b* (poslední element matice *A* respektive matice *B*). Dále je zkontrolováno, zda jsou rozměry obou matic shodné. V případě že jsou obě matice stejně dlouhé, funkce iteruje přes všechny indexy prvků seznamů *pe_a* a *pe_b*. Pro každý index i přidá prvek z *pe_b* k odpovídajícímu prvku v *pe_a*. Nakonec funkce vrátí upravený seznam *a*.

```
def sum(a, b):
    pe_a = MyMath.prvky_v_posledni_dimensi(a)
    pe_b = MyMath.prvky_v_posledni_dimensi(b)
    if(len(le_a) == len(le_b)):
        for i in range(len(pe_a[0])):
            pe_a[0][i] += pe_b[0][i]
    return a
```

Obrázek 2.6 - Součet matic

Pro rozdíl dvou matic byla vytvořena funkce sub. Podobně jako u maticového součtu v kapitole tato funkce provádí rozdíl po prvcích, tedy každý prvek matice A je odečten s prvkem na korespondující pozici v matici B:

$$A \ominus B = \begin{pmatrix} a_{11} - b_{11} & a_{12} - b_{12} & \cdots & a_{1m} - b_{1m} \\ a_{21} - b_{21} & a_{22} - b_{22} & \cdots & a_{2m} - b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} - b_{n1} & a_{n2} - b_{n2} & \cdots & a_{nm} - b_{nm} \end{pmatrix} \tag{2.3}$$

$$c_{ij} = a_{ij} - b_{ij} \tag{2.4}$$

kde: A je vstupní matice,

B je vstupní matice, která má být přičtena k matici A,

a_{ij} je prvek matice A na pozici i, j,

b_{ij} je prvek matice B na pozici i, j,

c_{ij} je prvek výsledné matice, kterou funkce vrací.

Funkce pro maticový rozdíl po prvcích je totožná s fungováním funkce sum popsané výše.

```
def sub(a, b):
    pe_a = MyMath.prvky_v_posledni_dimensi(a)
    pe_b = MyMath.prvky_v_posledni_dimensi(b)
    if(len(le_a) == len(le_b)):
        for i in range(len(pe_a[0])):
            pe_a[0][i] -= pe_b[0][i]
    return a
```

Obrázek 2.7 – Rozdíl matic

Maticový součin

Maticový součin je rozdělen do několika případů, podle rozměrů vstupních proměnných. Funkce může provést součin v různých datových strukturách, jako jsou skaláry, vektory a matice, a dynamicky se přizpůsobuje rozměrům vstupů. Pokud jsou obě vstupní proměnné skaláry (tedy mají nulový rozměr, jedná se o číslo), jedná se o klasické násobení dvou čísel, kde výsledek je opět jen číslo.

Pokud je jedna proměnná skalár a druhá pole (její rozměr je roven jedné), je skalární proměnnou vynásoben každý prvek v poli. Je-li tedy proměnná a skalár a proměnná b pole, je výsledný vektor c roven

$$c = b \cdot a = \begin{bmatrix} b_1 \cdot a \\ b_2 \cdot a \\ \vdots \\ b_3 \cdot a \end{bmatrix} \tag{2.5}$$

$$c_i = b_i \cdot a$$

kde: a je vstupní skalární hodnota,

b je vstupní sloupcový, respektive řádkový vektor,

c je výsledný sloupcový, respektive řádkový vektor.

Rozměr výsledného vektoru c je stejný, jako rozměr vektoru b. V programu jsou nejdříve zjištěny rozměry obou vstupních proměnných. Funkce *rozmery_pole* má jeden vstupní parametr – proměnnou, u které se zjišťuje její rozměr. Do proměnné *dimenze* se následně ukládá rozměr vnořeného pole. Je-li ve vstupní proměnné více vrstev vnořených polí, funkce iterativně projede všechna pole a jejich rozměry zapíše do této proměnné. Ta je následně vrácena zpět do programu.

```
def rozmery_pole(pole):
    dimenze = []
    while isinstance(pole, list):
        dimenze.append(len(pole))
        pole = pole[0] if pole else None
    return dimenze
```

Obrázek 2.8 – Funkce pro zjištění rozměrů pole

Pokud jsou obě vstupní proměnné vektory a mají stejnou délku, je výsledkem součinu součet součinů jejich příslušných složek.

$$c = a \times b = \begin{bmatrix} a_1 \cdot b_1 \\ a_2 \cdot b_2 \\ \vdots \\ a_n \cdot b_n \end{bmatrix} \qquad (2.6)$$
$$c_i = a_i \cdot b_i$$

V případě, že jednou vstupní proměnnou je matice a druhou vektor, je výstupem nový vektor, jehož hodnoty jsou výsledkem součinu řádků matice a vektoru.

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix}$$
$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} \qquad (2.7)$$
$$c = \begin{bmatrix} a_{11} \cdot b_1 + \cdots + a_{1m} \cdot b_1 \\ \vdots \\ a_{n1} \cdot b_n + \cdots + a_{nm} \cdot b_n \end{bmatrix}$$
$$c_i = a_{i1} \cdot b_i + a_{i2} \cdot b_i + \cdots + a_{im} \cdot b_i = \sum_{j=1}^{m} a_{ij} \cdot b_i$$

Poslední možností násobení, které tato funkce umožňuje, je násobení dvou matic. Zde je funkce omezena pouze na násobení dvourozměrných matic, což je pro potřeby této práce dostatečné. Další podmínkou, která u násobení matic vyplývá, je nutnost dodržení kompatibilních rozměrů. Pokud má matice *A* rozměry *m*×*n*, musí mít matice *B* rozměry *n*×*p*. Tedy u násobení matic musí být počet sloupců první matice roven počtu řádků druhé matice.

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$
$$B = \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{bmatrix} \qquad (2.8)$$
$$C = \begin{bmatrix} a_{11} \cdot b_{11} + \cdots + a_{1n} \cdot b_{n1} & \cdots & a_{11} \cdot b_{1p} + \cdots + a_{1n} + b_{np} \\ \vdots & \ddots & \vdots \\ a_{m1} \cdot b_{11} + \cdots + a_{mn} \cdot b_{n1} & \cdots & a_{m1} \cdot b_{1p} + \cdots + a_{mn} \cdot b_{np} \end{bmatrix}$$
$$c_{ik} = a_{i1} \cdot b_{1k} + a_{i2} \cdot b_{2k} + \cdots + a_{in} \cdot b_{nk} = \sum_{j=1}^{n} a_{ij} \cdot b_{jk}$$

```python
def dot(a, b):
    dim_a = MyMath.rozmery_pole(a)
    dim_b = MyMath.rozmery_pole(b)

    if(dim_a == [] and dim_b == []):
        return a*b
    if(dim_a == []):
        return [x * a for x in b]
    if(dim_a == [1]):
        return [x * a[0] for x in b]
    if(dim_b == []):
        return [x * b for x in a]
    if(dim_b == [] or dim_b == [1]):
        return [x[0] * b[0] for x in MyMath.prvky_v_posledni_dimensi(a)]

    if(len(dim_a)==1 and len(dim_b)==1 and dim_a[0] == dim_b[0]):
        return sum(x * y for x, y in zip(a, b))

    if(len(dim_a)==2 and len(dim_b)==2 and dim_a[1] == dim_b[0]):
        return [[sum(x * y for x, y in zip(row_a, col_b)) for col_b in zip(*b)] for row_a in a]

    if(len(dim_a)==2 and len(dim_b)==1):
        return [sum(x * y for x, y in zip(row_a, b)) for row_a in a]

    raise Exception(f"Nekompatibilní rozměry: {dim_a} a {dim_b}")
```

Obrázek 2.9 – Funkce pro násobení matic

Transpozice matice

Transpozice matice je základní operací v matematice a informatice, která ve stávající matic, respektive vektoru, zamění hodnoty ve sloupcích a řádcích. Jedná se tedy o operaci, která v podstatě otočí matici o 90°. Pokud je použita transpozice na matici o rozměrech $m \times n$, má výsledná nová matice rozměry $n \times m$.

Funkce pro transpozici matice či vektoru je označena jako *T(a)*. Tato funkce umí pracovat s jednorozměrnými i dvourozměrnými poli. Na prvním řádku funkce zkontroluje, zda první prvek pole *a* (*a[0]*) má atribut ‚*length*‘. Tento parametr dokáže určit, zda je první prvek v poli skalární číslo, v tom případě vrací funkce *hasattr* logickou hodnotu *false*. Pokud prvek obsahuje další pole (jedná se tedy o vnořené pole), vrací funkce *hasattr* hodnotu *true*. Díky tomu dokáže funkce určit, zda se jedná o jednorozměrné pole (vektor), nebo o pole dvourozměrné (matice). Následně funkce projde všechny prvky ve vstupním poli a vytvoří nové pole *aT*, které obsahuje transponovanou matici *a*. V případě, že vstupní proměnná *a* je dvourozměrné pole, funkce vytvoří proměnnou *aT*. Počet řádků v *aT* se bude rovnat počtu sloupců v a (tj. len(a[0])) a počet sloupců se bude rovnat počtu řádků v a (tj. len(a)). Každý prvek je zpočátku nastaven na nulu. Následně se přes vnořené smyčky iterují řádky a sloupce původní matice *a*. Pro každou dvojici indexů (i, j) je prvek na pozici [j][i] v původní matici a přiřazen pozici [i][j] v nové matici *aT*.

Pokud je vstupní proměnná *a* jednodimenzionální seznam (tj. nemá atribut *__len__* ve svém prvním prvku), funkce vytvoří nové pole *aT*. Výsledkem je dvoudimenzionální pole, kde se každý původní prvek pole *a* stane řádkem v novém poli *aT*, čímž se původní řádkový vektor změní na vektor sloupcový.

```
def T(a):
    if(hasattr(a[0], "__len__")):
        aT = a

        aT = [[0]*len(a) for i in range(len(a[0]))]
        for i in range(len(a[0])):
            for j in range(len(a)):
                aT[i][j] = a[j][i]

    else:
        aT = [[a[i]] for i in range(len(a))]
    return aT
```

Obrázek 2.10 – Transpozice matice

2.1.3 AKTIVAČNÍ FUNKCE A VÝPOČET ODCHYLKY

Aktivační funkce v neuronových sítích mají zásadní význam pro přidání nelineárních vlastností do modelu. Bez aktivačních funkcí by se neuronová síť, bez ohledu na počet použitých skrytých vrstev, chovala stejně jako lineární regresní model. Takový model je schopen pochopit pouze jednoduché vztahy v datech, které lze lineárně rozdělit. Přidání nelineárních aktivačních funkcí umožňuje těmto sítím učit se složitější vzorce, které jsou nezbytné pro složitější úlohy, jako je rozpoznávání obrazu.

Volba aktivační funkce významně ovlivňuje, jak efektivně se síť dokáže učit a zda dokáže konvergovat k řešení v rozumném čase. Aktivační funkce navíc ovlivňují i to, jak se během trénování šíří gradienty zpět sítí; mohou ovlivnit rychlost konvergence a pravděpodobnost, že proces trénování uvízne v lokálních minimech. Pro účely této práce byly použity tři typy aktivačních funkcí: Sigmoid, Tanh a ReLU.

Aktivační funkce Sigmoid

Aktivační funkce Sigmoid, někdy také označovaná jako logistická funkce, je jednou ze základních aktivačních funkcí v oblasti neuronových sítí, zejména v sítích určených pro binární klasifikaci. Historicky byly sigmoidní a hyperbolické funkce často využívané, protože jsou hladké, diferencovatelné a omezené. V dnešní době se tyto funkce však v mnoha aplikacích hlubokého učení přestaly uplatňovat. Příčinou jsou jejich další vlastnosti, které se s vývojem neuronových sítí (obzvláště u složitých, vícevrstvých sítí) začaly projevovat. Největším problémem těchto funkcí je mizení gradientů, kdy se parametry pro učení sítě stávají příliš

malými pro efektivní učení, když se zpětně šíří sítí. Tento problém je zvláště výrazný v komplexnějších sítích, kde hloubka zvětšuje útlum gradientů, což efektivně zastavuje proces učení během zpětného šíření.

Aktivační funkci Sigmoid lze matematicky vyjádřit pomocí rovnice 2.9,

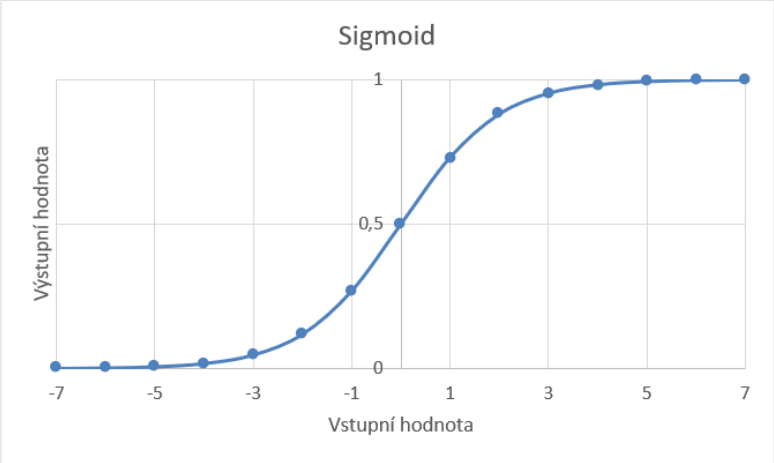
$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

kde: σ je funkce Sigmoid,

e je Eulerovo číslo,

x je vstupní proměnná.

Výstupní rozsah funkce Sigmoid je mezi hodnotami 0 a 1. Díky tomu je často užitečná pro modely, kde je zapotřebí interpretovat výstup jako pravděpodobnost, například v úlohách binární klasifikace. Funkce je spojitá na celém svém definičním oboru (viz obrázek 2.11).



Obrázek 2.11 – Funkce Sigmoid

Funkce nejdříve zjistí, zda je vstupní proměnná *a* skalární hodnota, nebo vektor. V případě že se jedná o skalární hodnotu, je proveden standardní výpočet dle rovnice 2.9 a výsledná hodnota je vrácena zpět do programu. Pokud je vstupní proměnná *a* polem, provede se výpočet pro každý prvek tohoto pole.

```
def Sigmoid(a):
    if(type(a) == list):
        for i in range(len(a)):
            a[i] = 1/(1+CONST_E**(-a[i]))
    else:
        a = 1/(1+CONST_E**(-a))
    return a
```

Obrázek 2.12 - Výpočet funkce Sigmoid

Základem trénování neuronových sítí je zpětné šíření chyby, které je se používá doladění vah sítě na základě chybovosti predikce výstupu v porovnání s očekávaným výsledkem. Odchylka se šíří zpět sítí pomocí výpočtu gradientu ztrátové funkce vzhledem ke každé váze. Pro funkci Sigmoid je derivace (která udává, jak se mění výstup funkce v závislosti na změnách jejího vstupu) dána vztahem:

$$\begin{aligned}\frac{d}{dx}\sigma &= \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{d}{dx}((1+e^{-x})^{-1}) = (1+e^{-x})^{-2}\cdot(-e^{-x})\cdot(-1)\\ &= \frac{-e^{-x}}{-(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\cdot\frac{e^{-x}}{1+e^{-x}}\\ &= \frac{1}{1+e^{-x}}\cdot\frac{(1+e^{-x})-1}{1+e^{-x}} = \frac{1}{1+e^{-x}}\cdot\left(1-\frac{1}{1+e^{-x}}\right)\\ &= \sigma\cdot(1-\sigma)\end{aligned} \tag{2.10}$$

$$\frac{d}{dx}\sigma = \sigma\cdot(1-\sigma) = \frac{1}{1+e^{-x}}\cdot\left(1-\frac{1}{1+e^{-x}}\right)$$

Funkce pro výpočet zpětného šíření Sigmoidu je podobná, jako funkce pro standardní výpočet Sigmoid. Funkce nejdříve zjistí, zda je vstupní proměnná *a* skalár nebo pole. Následně provede výpočet podle rovnice 2.10 pro všechny prvky, které proměnná *a* obsahuje.

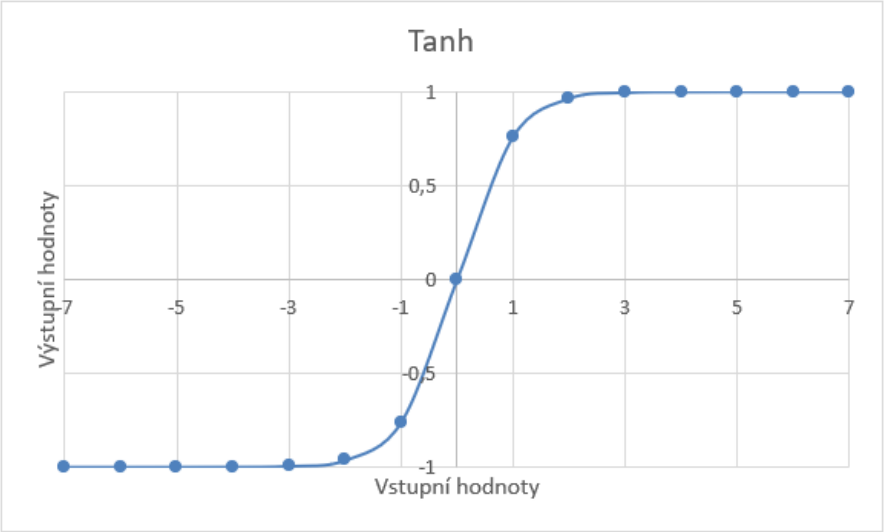
```
def Sigmoid_Derivace(a):
    if(type(a) == list):
        for i in range(len(a)):
            a[i] = (1/(1+CONST_E**(-a[i]))) * (1-(1/(1+CONST_E**(-a[i]))))
    else:
        a = (1/(1+CONST_E**(-a))) * (1-(1/(1+CONST_E**(-a))))
    return a
```

Obrázek 2.13 – Derivace funkce Sigmoid

Aktivační funkce Tanh

Hyperbolická tangenciální funkce, běžně označovaná jako Tanh, je aktivační funkce používaná v neuronových sítích, která mapuje reálná čísla na rozsah [-1,1]. Oproti funkci Sigmoid, aktivační funkce Tanh poskytuje škálovatelný a posunutý výstup. Díky tomu nachází v moderních neuronových sítích častější uplatnění než právě zmiňovaná funkce Sigmoid. Tato aktivační funkce je nejčastěji používána ve skrytých vrstvách neuronových sítí, kde je požadována normalizace dat nebo symetrie kolem nuly. Matematická reprezentace funkce Tanh je dána vztahem:

$$Tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.11}$$



Obrázek 2.14 – Funkce Tanh

Tato funkce je v podstatě škálovaná sigmoidní funkce, která rozšiřuje svůj výstupní rozsah na [-1,1] místo [0,1], jak lze vidět u funkce Sigmoid (viz obrázek 2.14). Toto škálování poskytuje silnější gradient pro většinu svého vstupního rozsahu, což je zásadní výhoda při procesu trénování neuronových sítí. Na rozdíl od funkce Sigmoid je Tanh centrovaná kolem nuly. Tato vlastnost pomáhá při normalizaci dat a usnadňuje proces učení další vrstvy. Zabraňuje zkreslení gradientů během zpětného šíření, což může urychlit konvergenci sítě k predikovanému řešení. Aktivační funkce Tanh je častou volbou pro určité typy neuronových sítí, zejména pro rekurentní neuronové sítě (RNN), kde je výhodné zachovat symetrii dat a silné gradientní vlastnosti.

Ve funkci Tanh probíhá výpočet hodnoty podle rovnice 2.11. Opět je nejdříve zkontrolováno, zda je vstupní proměnná *a* řádkový vektor nebo matice a následně se provede výpočet pro každý prvek, který proměnná *a* obsahuje.

```
def Tanh(a):
    if(type(a) == list):
        for i in range(len(a)):
            a[i] = (CONST_E**a[i]-CONST_E**-a[i])/(CONST_E**a[i]+CONST_E**-a[i])
    else:
        a = (CONST_E**a-CONST_E**-a)/(CONST_E**a+CONST_E**-a)
    return a
```

Obrázek 2.15 – Výpočet funkce Tanh

Pro zpětné šíření chyby je spočítána derivace funkce Tanh (viz rovnice 2.12). Rozsah derivace je mezi 0 a 1, což znamená, že gradienty nejsou ani zesíleny, ani obráceny, ale zmenšeny, což může pomoci při stabilizaci procesu učení.

$$\begin{aligned}\frac{d}{dx}Tanh(x) &= \frac{d}{dx}\left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right) = \left|\left(\frac{f}{g}\right)' = \right| \\ &= \frac{(e^x + e^{-x}) \cdot (e^x + e^{-x}) - (e^x - e^{-x}) \cdot (e^x - e^{-x})}{(e^x + e^{-x})^2} \\ &= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\ &= 1 - Tanh(x)^2\end{aligned} \tag{2.12}$$

$$\frac{d}{dx}Tanh(x) = 1 - Tanh(x)^2 = 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2}$$

```
def Tanh_Derivace(a):
    if(type(a) == list):
        for i in range(len(a)):
            a[i] = 1-((CONST_E**a[i]-CONST_E**-a[i])**2/(CONST_E**a[i]+CONST_E**-a[i])**2)
    else:
        a = 1-((CONST_E**a-CONST_E**-a)**2/(CONST_E**a+CONST_E**-a)**2)
    return a
```

Obrázek 2.16 – Derivace funkce Tanh

Aktivační funkce ReLU

Rectified Linear Unit (ReLU) se stala jednou z hlavních aktivačních funkcí v oblasti neuronových sítí, zejména v kontextu modelů hlubokého učení. Její zjednodušená a zároveň vysoce efektivní matematická funkce jí umožňuje v mnoha ohledech překonávat starší, tradičně používané aktivační funkce, jako jsou sigmoid a tanh, zejména z hlediska výpočetní efektivity a rychlosti konvergence k řešení během trénování.

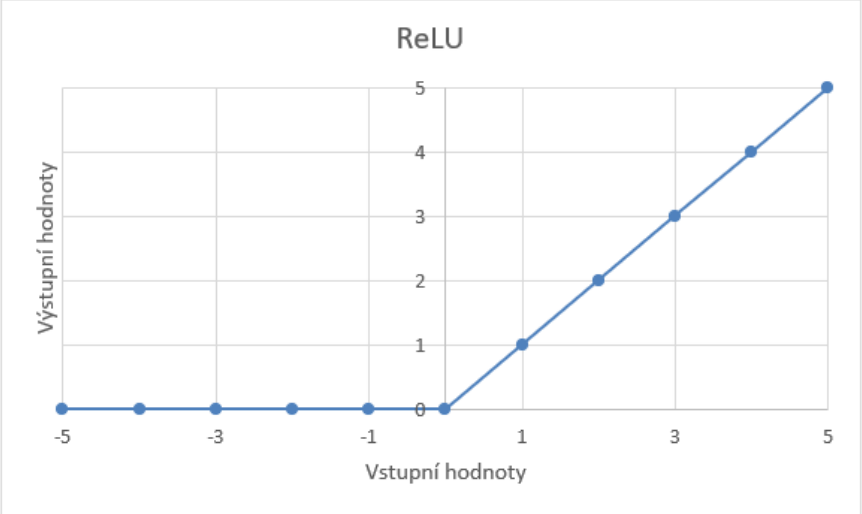
Matematicky je ReLU vyjádřena následovně:

$$ReLU(x) = \begin{cases} x \; pro \; x > 0, \\ 0 \; pro \; x \leq 0 \end{cases} \tag{2.13}$$

Pokud má vstupní proměnná nenulovou kladnou hodnotu, tato funkce vrací stejnou hodnotu zpět. V opačném případě je výstupní hodnota rovna nulu. Jednoduchost funkce ReLU spočívá v této lineární, shora neomezené formě, která jí umožňuje poskytovat rozsah

aktivačních potenciálů od nuly do nekonečna. Ve své podstatě aktivační funkce ReLU nahrazuje zápornou část svého vstupu za nulu.

Na rozdíl od tanh nebo sigmoidy nevyžaduje ReLU složité matematické operace, jako jsou exponenciály, což ji činí výpočetně efektivní a výrazně urychluje proces trénování. Ačkoliv ReLU vypadá lineárně, do modelu vnáší nelinearitu a právě tato nelineární vlastnost umožňuje hlubokým neuronovým sítím mapovat netriviální problémy. V praxi tato funkce vytváří takzvaný „řídký“ aktivační model, což znamená, že v každém okamžiku je aktivní pouze určitá podmnožina neuronů (ostatní neurony mají váhu vstupu rovnu nule). Díky tomu je reprezentace vstupních dat efektivnější a výsledná neuronová síť je méně náchylná k nadměrnému přeučení.



Obrázek 2.17 – Funkce ReLU

Pro derivaci funkce ReLU je nutné rozdělit její funkční interval na dvě části a řešit tedy dvě triviální derivace pro různé hodnoty vstupní proměnné. U standardního řešení není tato funkce diferencovatelná při nule, což může při optimalizaci pomocí sestupu po gradientu způsobit problémy. V této práci je derivace funkce ReLU v bodě nula rovna nule.

$$ReLU(x)' = \begin{cases} x > 0: \frac{d}{dx}x = 1 \\ x \leq 0: \frac{d}{dx}0 = 0 \end{cases} \tag{2.14}$$

Jednou z hlavních výhod zpětné propagace chyby při použití funkce ReLU je její schopnost zmírnit problém mizejícího gradientu, který je běžný u funkcí Sigmoid a Tanh. Protože gradient pro kladné vstupy je vždy roven 1, ReLU zajišťuje, že během zpětného šíření nedochází k nasycení gradientu, a tedy ani k problému s jeho klesáním. Díky lineárnímu tvaru mohou trénovací modely při použití ReLU konvergovat mnohem rychleji než modely využívající jiné aktivační funkce. To se nejvíce projevuje zejména u sítí, které jsou určené pro řešení komplexních úloh a jejichž trénování vyžaduje značný výpočetní výkon.

```
def ReLU(a):
    if(type(a) == list):
        for i in range(len(a)):
            a[i] = max(0, a[i])
    else:
            a = max(0, a)
    return a

def ReLU_Derivace(a):
    if(type(a) == list):
            for i in range(len(a)):
                a[i] = 1 if a[i]>0 else 0
    else:
        a = 1 if a>0 else 0
    return a
```

Obrázek 2.18 – Výpočet aktivační funkce ReLU

Výpočet odchylky MSE

Střední kvadratická chyba (MSE) je široce používanou metrikou výkonnosti v oblasti strojového učení a statistiky, zejména v regresní analýze a při trénování neuronových sítí. MSE poskytuje jasnou a kvantifikovatelnou míru chyby mezi předpovídanými hodnotami a hodnotami skutečnými. Je to jedna z nejjednodušších a nejefektivnějších metod hodnocení přesnosti modelu, která zajišťuje efektivní řízení optimalizačních technik při trénování. Matematicky se střední kvadratická chyba vypočítá jako průměr rozdílů na druhou mezi předpovídanými a skutečnými hodnotami.

$$MSE = \frac{1}{n} \cdot \left(\hat{Y} - Y\right)^2 = \frac{1}{n} \cdot \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \tag{2.15}$$

kde: MSE je funkce střední kvadratické chyby,

n je počet hodnot,

$\hat{Y}$ je vektor odhadovaných hodnot,

Y je vektor skutečných hodnot,

$\hat{y}_i$ je i-tý prvek vektoru $\hat{Y}$,

y_i je *i*-tý prvek vektoru *Y*.

Jednou z klíčových vlastností MSE je její citlivost na velké chyby. Vzhledem k tomu, že chyby jsou před zprůměrováním kvadratizovány, s rostoucí hodnotou chyby roste výsledná hodnota kvadraticky. Díky své diferencovatelné povaze je MSE ideální pro optimalizační algoritmy, které se spoléhají na zpětné šíření chyby. To je výhodné zejména u neuronových sítí, kde je třeba během trénování minimalizovat ztrátovou funkci. Jednoduchost vzorce MSE usnadňuje jeho implementaci. Poskytuje jasné měřítko výkonnosti neuronové sítě ve smyslu toho, jak blízko jsou předpovězené hodnoty k hodnotám skutečným.

Pro derivaci této funkce byl nejdříve výsledný vektor rozepsán jako derivace jednotlivých prvků, to je znázorněno v rovnici 2.16. Následně byla spočtena derivace prvního prvku tohoto vektoru. Po rozepsání sumy vzorce MSE (rovnice 2.17) je patrné, že všechny členy krom členu, který obsahuje právě derivovanou proměnnou (v případě rovnice 2.17 je to člen obsahující y_1) jsou při derivování rovny nule. Díky tomu je možné celý vzorec zjednodušit jen na potřebný člen. Výsledek derivace lze následně zobecnit pro každý člen vektoru a tedy i na celý vektor výsledných, respektive odhadovaných hodnot, jak je popsáno rovnicí 2.18.

$$\frac{\partial MSE}{\partial Y} = \begin{bmatrix} \frac{\partial MSE}{\partial y_1} \\ \vdots \\ \frac{\partial MSE}{\partial y_i} \end{bmatrix} \tag{2.16}$$

$$\frac{\partial MSE}{\partial y_1} = \frac{\partial}{\partial y_1} \cdot \frac{1}{n} \cdot [(\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \cdots (\hat{y}_n - y_n)^2] \tag{2.17}$$

$$\frac{\partial MSE}{\partial y_1} = \frac{\partial}{\partial y_1} \cdot \frac{1}{n} \cdot (\hat{y}_1 - y_1)^2 = \frac{2}{n}(y_1 - \hat{y}_1)$$

$$\frac{\partial MSE}{\partial y_i} = \frac{2}{n}(y_i - \hat{y}_i) \tag{2.18}$$

$$\frac{\partial MSE}{\partial Y} = \frac{2}{n}\left(Y_i - \hat{Y}_i\right)$$

Rovnice pro výpočet MSE a její derivace (popsány rovnicemi 2.15 a 2.18) jsou vytvořeny v programu ve stejně pojmenovaných funkcích. Jelikož pro potřeby této práce byl výsledkem neuronové sítě vektor pouze s jednou hodnotou, udávající třídu objektu, jsou funkce upraveny pouze pro skalární výpočty.

```python
def mse(y_true, y_pred):
    result = (y_true[0] - y_pred[0])**2 / len(y_true)
    return result

def mse_p(y_true, y_pred):
    result = 2 * (y_pred[0] - y_true[0]) / len(y_true)
    return result
```

Obrázek 2.19 – Funkce pro výpočet středí kvadratické odchylky

Výpočet odchylky BCE

Binární křížová entropie (BCE – Binary Cross Entropy), známá také jako logaritmická ztráta, je ztrátová funkce používaná především v binárních klasifikačních modelech. Měří chybovost naučeného modelu neuronové sítě, jehož výstupem je hodnota pravděpodobnosti mezi 0 a 1. Binární křížová entropie určuje velikost rozdílu mezi skutečnými hodnotami a hodnotami předpovídanými. Díky tomu poskytuje metriku, kterou lze minimalizovat během procesu trénování neuronových sítí. Binární křížová entropie je matematicky definována rovnicí 2.19.

$$BCE = -\frac{1}{n} \cdot \sum_{i=1}^{n} [y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)] \tag{2.19}$$

kde: BCE je funkce binární křížové entropie,

n je počet hodnot,

$\hat{y}_i$ je i-tý prvek vektoru $\hat{Y}$,

y_i je i-tý prvek vektoru Y.

Jak už z názvu vyplívá, je funkce BCE používána v úlohách, při kterých dochází k binární klasifikaci (tedy ke klasifikaci dvou stavů). Tím pádem je velmi účinná při rozlišování mezi pravděpodobnostmi blízkými hodnotám 1 nebo 0, které jsou jisté, a pravděpodobnostmi blízkými hodnotám 0,5, které představují nejistotu. Binární křížová entropie je díky své nelinearitě vhodná pro trénování regresních modelů a neuronových sítí, kde jsou sigmoidní funkce použity jako aktivační funkce poslední vrstvy pro predikci. Tato ztrátová funkce je spojitá a diferencovatelná, s výjimkou bodů, kde se odhad rovná jedné nebo nule. Protože binární křížová entropie pracuje s logaritmickými funkcemi, mohou předpovědi, které jsou extrémně blízké 0 nebo 1, vést k velmi vysokým hodnotám ztrát, pokud jsou chybné, což může destabilizovat proces trénování.

Na rozdíl od jiných ztrátových funkcí, které mohou mít při správné predikci nulový gradient pro úpravu vah a prahů neuronů, binární křížová entropie stále poskytuje nenulový gradient. To může vést k aktualizacím vah modelu, i když model funguje optimálně, což může způsobit přeučení sítě.

Pro derivaci funkce byl použit obdobný postup, jako u funkce MSE. Derivace je nejdříve rozepsána jako vektor dílčích derivací jednotlivých prvků (rovnice 2.20). Následně byla spočtena derivace prvního prvku (rovnice 2.21), která byla dále zobecněna pro každý prvek vektoru (rovnice 2.22).

$$\frac{\partial BCE}{\partial Y} = \begin{bmatrix} \frac{\partial BCE}{\partial y_1} \\ \vdots \\ \frac{\partial BCE}{\partial y_i} \end{bmatrix} \tag{2.20}$$

$$\begin{aligned} \frac{\partial BCE}{\partial y_1} &= \frac{\partial}{\partial y_1} \cdot \left(-\frac{1}{n} \cdot [y_1 \cdot \log(\hat{y}_1) + (1 - y_1) \cdot \log(1 - \hat{y}_1)] \right) \\ &= -\frac{1}{n} \cdot \left(\frac{\hat{y}_1}{y_1} - \frac{1 - \hat{y}_1}{1 - y_1} \right) = \frac{1}{n} \cdot \left(\frac{1 - \hat{y}_1}{1 - y_1} - \frac{\hat{y}_1}{y_1} \right) \end{aligned} \tag{2.21}$$

$$\frac{\partial BCE}{\partial y_i} = \frac{1}{n} \cdot \left(\frac{1 - \hat{y}_i}{1 - y_i} - \frac{\hat{y}_i}{y_i} \right) \tag{2.22}$$

V programu byly vytvořeny funkce dle rovnic 2.19 a 2.22. Pro funkci logaritmu byla využita funkce log z knihovny NumPy, jelikož jazyk Python standardně logaritmické funkce nenabízí.

```
def bce(y_true, y_pred):
    return -(1/len(y_true))*(y_true*np.log(y_pred)+(1-y_true)*np.log(1-y_pred))

def bce_derivace(y_true, y_pred):
    return (1/len(y_true))*((1-y_true)/(1-y_pred)-y_true/y_pred)
```

Obrázek 2.20 – Funkce pro výpočet binární křížové entropie

2.2 PŘEDZPRACOVÁNÍ OBRAZU

Předzpracování obrazů je ve valné většině případů prvním krokem v procesu učení a nasazení konvolučních neuronových sítí, zejména při práci s vizuálními daty, jako jsou obrazy. Účinnost CNN závisí na kvalitě a formátu vstupních dat. Předzpracování obrazu pomáhá transformovat surová vstupní obrazová data do vhodné podoby, která maximalizuje schopnost sítě naučit se významné rysy, aniž by byla silněji ovlivněna chybami a anomáliemi ve vstupních datech.

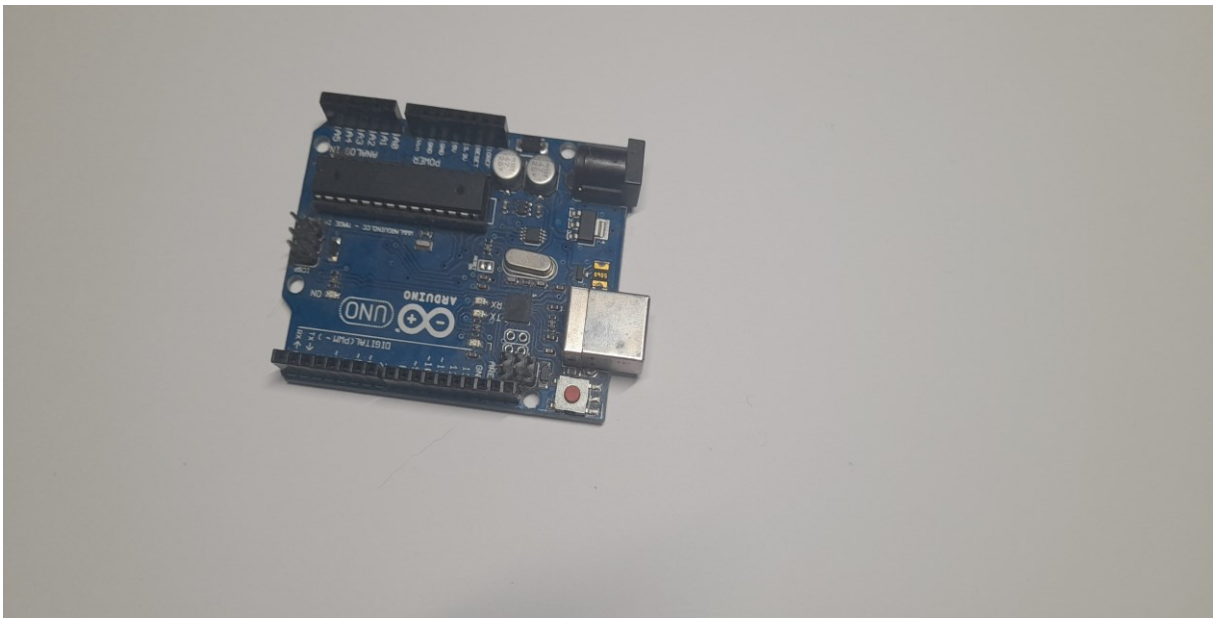
Hlavním cílem předzpracování obrazu je učinit obrazy vhodnějšími pro analýzu a zpracování neuronovou sítí. To zahrnuje několik metod, z nichž každá je určena k řešení specifických aspektů obrazových dat, které by mohly ovlivnit proces učení. Jedním ze základních kroků je změna velikosti a normalizace obrazu. Sítě CNN vyžadují pevnou velikost vstupních dat, a proto je třeba změnit velikost obrazů, které často přicházejí v různých velikostech a rozlišeních, na jednotný tvar. Normalizace navíc zahrnuje škálování hodnot pixelů do společného rozsahu. Hodnota barvy jednoho pixelu v obrazu je standardně v rozmezí [255; 0]. Pro použití těchto dat se v neuronových sítích normalizují hodnoty pixelů obvykle na hodnoty v rozmezí [1; 0], což pomáhá urychlit konvergenci tím, že poskytuje konzistentní měřítko vstupních rysů.

Pro účely této práce byly obrazy rozpoznávaných objektů pořízeny průmyslovou kamerou MV-CE050-30UC od výrobce Hikrobot. Tato kamera obsahuje snímač typu CMOS s výstupním rozlišením 2592×1944 pixelů. Kamera dokáže snímat obraz rychlostí 31 snímků za sekundu, což je pro potřeby této práce dostatečné.

Dataset byl vytvořen z pěti různých druhů předmětů, každý předmět je reprezentován 110 různými obrazy vytvořených z deseti vstupních obrazů. Předměty použité pro účely této práce jsou následující: Arduino Uno, plošný spoj rezistorového pole, baterie typu CR2032, dotykový TFT displej a několik typů šroubů. Tyto objekty byly vybrány pro ověření funkčnosti učení navržené neuronové sítě v případě podobných objektů, lesklých objektů a objektů s malými rozměry.

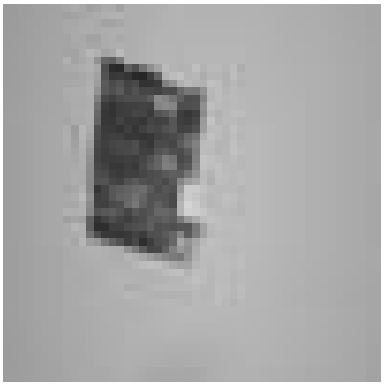
2.2.1 NALEZENÍ OBJEKTU

Výstupní obraz z kamery má rozlišení 2592×1944 pixelů. Toto rozlišení je zbytečně velké pro potřeby rozpoznávání třídy objektů a vyžadovalo by poměrně vysoký výkon jak pro předzpracování obrazu, tak pro následné zpracování a učení obrazu neuronovou sítí. Z těchto důvodů je vstupní obraz nejdříve zmenšený na rozměry 450×300 pixelů. Tento rozměr je již možné programem zpracovávat v rozumném čase, nicméně pro využití jako vstupní data pro neuronovou síť stále obsahuje příliš velké množství informací (pro obraz těchto rozměrů by bylo zapotřebí 135 000 vstupních neuronů). Proto je obraz před vstupem do neuronové sítě zmenšen na rozměry 50×50 pixelů, což se ukázalo pro potřeby této práce jako dostatečné.



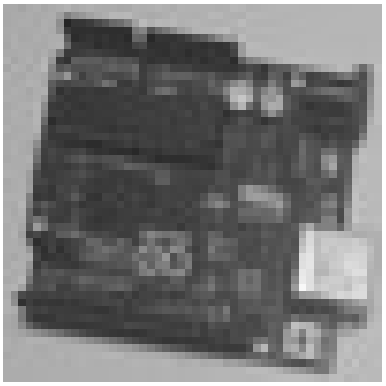
Obrázek 2.21 – Obraz pořízený kamerou

Jak je možné vidět na obrázku 2.21, obsahuje pořízený obraz z výstupu kamery značnou plochu „hluchých“ míst – míst kde se nenachází snímaný objekt. Pokud by došlo ke zmenšení tohoto obrazu na rozměr 50×50 pixelů, může nadměrná ztráta informace snímaného objektu (zobrazena na obrázku 2.22) způsobit velké nepřesnosti v rámci učení sítě.



Obrázek 2.22 – Ztráta informace zmenšením obrazu

Tento problém je řešen nalezením objektu na vstupním obrazu a jeho následné oříznutí. Tím je zaručeno co nejmenší ztráta užitečných informací snímaného objektu. Naopak dochází k minimalizaci prázdných, nepotřebných oblastí pro další zpracování neuronovou sítí.



Obrázek 2.23 – Ztráta informace zmenšením po nalezení objektu

Z principu aplikace robotického vidění často není možné předem určit pozici snímaného objektu. Z toho důvodu nelze v programu nastavit, která část obrazu má být ořezána. Snímaný objekt tedy musí být nalezen pomocí algoritmů umožňujících zjištění jeho pozice a rozměrů. V této práci jsou tyto informace zároveň využity pro získání souřadnic polohy objektu na pracovní ploše a zpracovány do souřadného systému robotu. Předpoklad pro aplikaci robotického vidění, kterou se tato práce dále zabývá, je konstantní bílé pozadí snímané pracovní plochy. Díky tomuto předpokladu je možné nalézt snímaný objekt porovnáváním známé barvy pozadí s pixely obrazu a následná aplikace algoritmů prohledávání pro zjištění jeho polohy a rozměrů.

2.2.2 PROHLEDÁVÁNÍ OBRAZU

K prohledávání obrazu bylo využito algoritmu prohledávání do šířky. Na začátku programu se načte obrazový soubor, který je převeden do formátu odstínů šedi. Cesta k souboru je dána vstupní proměnnou *cesta*. Tento obraz se následně zmenší na rozměry 450 pixelů na šířku a 300 pixelů na výšku. Tím sice dojde k určité ztrátě informace obrazových dat snímaného objektu, což je ovšem pro učení konvolučních neuronových sítí do jisté míry žádoucí (síť se tak nesoustředí na zbytečné detaily objektů, ale na objekt jako celek).

Následně je vytvořeno dvourozměrné pole nazvané *pix*, které uchovává hodnoty pixelů obrazu. Hodnota barvy levého horního pixelu je určena jako barva pozadí, která je uložena do stejnojmenné proměnné. Nastaví se proměnná nazvaná *prah*, která slouží k určení potřebného rozdílu v barvě k odlišení rozpoznávaného objektu od pozadí.

```
def Prevod_Obrazu(cesta, offset, objekt, i):
    obr = Image.open(cesta).convert('L')
    obr = obr.resize((450,300))
    pix = obr.load()

    pozadi = pix[0, 0]
    prah = 25
```

Obrázek 2.24 – Inicializace funkce Převod obrazu

V samotném těle funkce je vytvořena proměnná *vrcholy_obrazu* pro uložení souřadnic bodů rozpoznávaného objektu identifikovaných v obraze. Funkce poté iteračně kontroluje každý pixel v obrazu pomocí dvou smyček *for*. Pokud je hodnota barvy aktuálně kontrolovaného pixelu nižší, než barva pozadí (upravená prahovou hodnotou), program začne pomocí algoritmu prohledávání do šířky (viz kapitola 5.3), která najde všechny body, které jsou vázané relací 8-sousedctví, které toto kritérium splňují. Tato funkce vrací seznam souřadnic představující oblast nebo tvar v obraze, který kontrastuje s pozadím. Smyčka je přerušena, jakmile je nalezen neprázdný seznam významných bodů, což znamená, že funkce ukončí další prohledávání.

```
vrcholy_obrazu = []
for y in range(obr.size[1]):
    for x in range(obr.size[0]):
        if(pix[x, y] < pozadi - prah):
            vrcholy_obrazu = Prohledavani_Do_Sirky(pix, x, y, pozadi - prah, pix, obr)
            break
    if(not(vrcholy_obrazu == [])):
        break
```

Obrázek 2.25 – Nalezení objektu na vstupním obrazu

Funkce dále zpracovává identifikovaný tvar v podobě pole souřadnic, přičemž začíná určením jeho rozměrů a polohy. Funkce *Find_Dim* vypočítá hranice tvaru na základě získaného seznamu souřadnic a vrátí je jako pole čtyř dvojic souřadnic čtvercové oblasti, ve které se objekt nachází. Tyto rozměry se pak použijí k oříznutí původního obrazu s tím, že rozměry oříznutí jsou z každé strany zvětšeny o hodnotu parametru *offset,* aby oříznutá oblast zahrnovala okraj kolem zjištěného tvaru, což pomáhá zachovat hranice objektu v obrazu. Operace oříznutí je zabalena do bloku *try-except*, který řeší případné výjimky, které se mohou vyskytnout během procesu ořezávání obrazu, kdy například rozměry přesahující hranice obrazu.

Po oříznutí následuje smyčka, která se desetkrát iteruje. V každém průchodu smyčkou dojde k vytvoření nového, otočeného, obrazu, díky čemu je z původního jednoho obrazu vytvořeno deset nových (viz kapitola 5.4). Nakonec je obraz zmenšen na menší jednotnou

velikost 50x50 pixelů a uložen pomocí cesty, která obsahuje název objektu (uložen v proměnné *objekt*).

```
dim = Rozmer_Obrazu(vrcholy_obrazu)
try:
    obr = obr.crop((dim[0]-offset, dim[1]-offset, dim[2]+offset, dim[3]+offset))
except:
    pass
for j in range(10):
    Otocit_Obraz(obr, objekt, j+(i+1)*10)

obr = obr.resize((50,50))
obr.save("Dataset\\"+objekt+"\\Preprocess\\"+str(i)+".png")
```

Obrázek 2.26 – Oříznutí a uložení obrazu

2.2.3 URČENÍ OBLASTI ZÁJMU

Pro účely této práce byl pro prohledávání obrazu zvolen algoritmus prohledávání do šířky (BFS - Breadth-First Search). Tato metoda se používá v různých aplikacích, včetně hledání nejkratší cesty v nevážených grafech, analýzy sítí a algoritmů pro vyhledávání, směrování a plánování.

Algoritmus prohledávání do šířky vychází z jednoho počátečního uzlu a prozkoumá všechny jeho sousední uzly v současné hloubce předtím, než přejde k uzlům v další hloubkové úrovni. Algoritmus nejprve zařadí zdrojový uzel do seznamu již navštívených uzlů a poté zařadí jeho sousední uzly do seznamu čekajících na navštívení. Následně uzel, který se nachází v seznamu nenavštívených a přesune jej do seznamu uzlů již navštívených. Zároveň jeho sousedy opět umístí do seznamu nenavštívených uzlů Tento proces se opakuje, dokud není seznam nenavštívených uzlů prázdný, čímž se zajistí, že každý uzel je navštíven přesně jednou a v rostoucí vzdálenosti od uzlu zdrojového.

Pro prohledávání „širokých“ stavových prostorů, jako v případě obrazových dat, může být algoritmus prohledávání do šířky paměťově úspornější než algoritmy prohledávání do hloubky. BFS využívá více paměti, protože ukládá uzly jedné úrovně, ale tyto uzly nemají mnoho potomků. Naopak DFS může potenciálně vyžadovat více paměti, protože musí ukládat všechny uzly v jediné cestě od zdrojového uzlu až po nejhlubší uzel.

Implementace algoritmu prohledávání do šířky je následující. Do funkce vstupuje několik parametrů, včetně proměnné *data*, což je pole pixelů obrazu, souřadnic (x, y) prvního bodu hledaného objektů a barevné hodnoty v proměnné *pozadi* (barevný práh pro porovnání s barvou hledaného objektu).

Na začátku funkce se vytvoří pole *neprohledane_stavy*, do kterého budou ukládány pixely, jež je třeba prozkoumat, a pole *prohledane_stavy*, které slouží k uložení již zpracovaných pixelů, aby se zabránilo jejich opětovnému zpracování. Počáteční pixel je přidán do pole *neprohledane_stavy*, jelikož k němu zatím nebyly prohledány sousední body.

```
def Prohledavani_Do_Sirky(data, x, y, pozadi, pix, obr):

    neprohledane_stavy = []
    prohledane_stavy = []

    neprohledane_stavy.append([x, y])
```

Obrázek 2.27 – Inicializace funkce

Následně funkce vstoupí do smyčky, která se opakuje tak dlouho, dokud není pole *neprohledane_stavy* prázdné. V těle smyčky se odstraní první pixel z pole *neprohledane_stavy* (pole tedy funguje jako fronta) a poté zkontroluje všech jeho 8 okolních sousedů (včetně těch na diagonálách). U každého sousedního pixelu nejprve zkontroluje, zda je barva pixelu menší než prahová hodnota pozadí. Před přidáním sousedního pixelu do pole *neprohledane_stavy* se funkce ujistí, zda již nebyl přidán do tohoto seznamu v předchozích iteracích smyčky nebo do pole *prohledane_stavy*, aby se zabránilo zacyklení v prohledávání. Jakékoli výjimky vzniklé během tohoto procesu (typicky v důsledku přístupu k pixelům mimo hranice obrazu) jsou ignorovány, což umožňuje funkci pokračovat ve zpracování dalších pixelů. Po dokončení prozkoumání funkce vrátí seznam prozkoumaných pixelů, který obsahuje všechny pixely, které jsou součástí oblasti spojené s počátečním objektu.

```
while(len(neprohledane_stavy) > 0):
    pixel = neprohledane_stavy.pop(0)
    prohledane_stavy.append(pixel)

    for i in range(-1, 2):
        for j in range(-1, 2):
            try:
                if(data[pixel[0]+i, pixel[1]+j] < pozadi):
                    duplikace = False

                    for point in neprohledane_stavy:
                        if(point[0] == pixel[0]+i and point[1] == pixel[1]+j):
                            duplikace = True
                    for point in prohledane_stavy:
                        if(point[0] == pixel[0]+i and point[1] == pixel[1]+j):
                            duplikace = True

                    if(duplikace == False):
                        neprohledane_stavy.append([pixel[0]+i, pixel[1]+j])
            except:
                pass

return prohledane_stavy
```

Obrázek 2.28 – Prohledávání do šířky

Funkce *Rozmer_Obrazu* slouží k určení rozměrů oblasti objektu definovaného polem bodů vrácených funkcí *Prohledavani_Do_Sirky*. Funkce hledá minimální a maximální hodnoty souřadni x a y, čímž určí obdélníkové ohraničení objektu. Počáteční hodnoty proměnných *x_min* a *y_min* jsou nastaveny na maximální hodnoty rozměru obrazu (450 a 300). Naopak hodnoty *x_max* a *y_max* jsou inicializovány na hodnotu nula.

Funkce následně kontroluje všechny prvky vstupního pole. Porovnáním hodnot aktuálně kontrolovaného bodu a hodnot uložených ve vytvořených proměnných zajistí, že na konci cyklu budou *x_min* a *y_min* obsahovat nejmenší hodnoty x a y nalezené mezi body a *x_max* a *y_max* budou obsahovat hodnoty největší. Funkce pak vrátí seznam těchto čtyř hodnot, které představují souřadnice levého horního rohu [*x_min*, *y_min*] a pravého dolního rohu [*x_max*, *y_max*] obdélníkové hranice.

```
def Rozmer_Obrazu(points):
    x_min = 450
    x_max = 0
    y_min = 300
    y_max = 0

    for point in points:
        if(point[0] > x_max): x_max = point[0]
        elif(point[0] < x_min): x_min = point[0]

        if(point[1] > y_max): y_max = point[1]
        elif(point[1] < y_min): y_min = point[1]

    return [x_min, y_min, x_max, y_max]
```

Obrázek 2.29 – Nalezení obdélníkové hranice hledaného objektu

2.2.4 VYTVOŘENÍ DATASETU

Pro správné učení neuronových sítí je zapotřebí zajistit dostatečně velký datový soubor (dataset) vstupních dat, na kterých je neuronová síť učena a testována. U příliš malého datasetu dochází k nedostatečnému zobecnění naučených parametrů a síť tak nedokáže správně kategorizovat testovací vstupy, na kterých nebyla učena. V této práci bylo pro dataset pořízeno ke každé třídě objektů deset snímků. Po rozdělení tohoto datasetu na trénovací, validační a testovací data, v poměru 6:2:2, nebylo možné síť natrénovat k dosažení použitelných výsledků. Proto byl pomocí těchto deseti snímků vygenerován větší dataset. Z každého snímku bylo vytvořeno deset snímků nových, které se lišily různým natočením snímku původního. Díky tomu bylo možné vytvořit dataset obsahující 110 snímků ke každé třídě objektů.

Při rotaci obrazu je použita transformační matice na souřadnice každého pixelu v obrazu. Tato rotační matice $R(\theta)$ a její aplikace je definována rovnicí 2.23, respektive 2.24.

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{2.23}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\theta) \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \cdot \cos(\theta) - y \cdot \sin(\theta) \\ x \cdot \sin(\theta) + y \cdot \cos(\theta) \end{bmatrix} \tag{2.24}$$

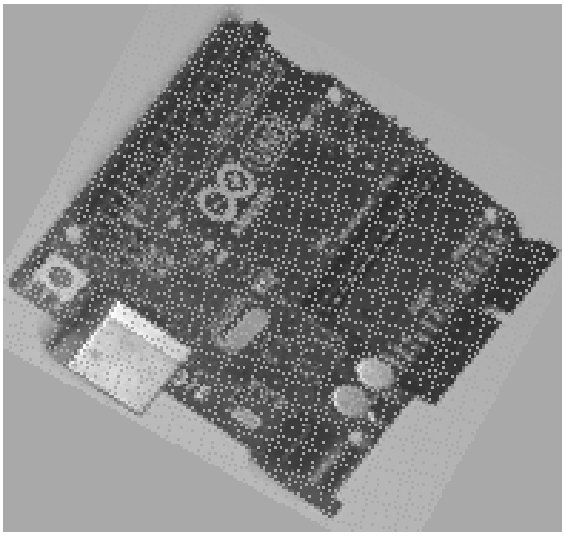
kde: R je transformační matice rotace,

θ je úhel otočení v protisměru hodinových ručiček,

x, y jsou původní souřadnice transformovaného bodu,

x', y' jsou nové souřadnice transformovaného bodu.

Při otáčení obrazu pro každý pixel z původního obrazu zapotřebí vypočítat nové hodnoty souřadnic jeho nového umístění. Protože však transformace může vést k neceločíselným souřadnicím, zejména při rotaci jiné než o ortogonální úhly, dochází k efektu Aliasingu, kdy se ztrácí informace z původního obrazu (viz obrázek 2.30). To může mít negativní vliv na proces učení neuronové sítě a vést tak k neoptimálnímu rozpoznání naučených objektů.



Obrázek 2.30 – Efekt Aliasingu

Aby při rotaci obrazů nedocházelo k tomuto efektu, byla v této práci využita rotace metodou tří střihů (Three Shears). Metoda tří střihů používá k dosažení rotace posloupnost takzvaných „střihových“ transformací. Tato metoda může být při digitálním zpracování obrazu efektivnější a poskytovat kvalitnější výsledky. K rotaci obrazu o úhel θ pomocí metody tří smyků je matice rotace rozložena do tří střihových matic: dvou horizontálních střihů a jednoho vertikálního střihu. Tyto matice a jejich aplikace jsou popsány rovnicí 2.25 respektive 2.26.

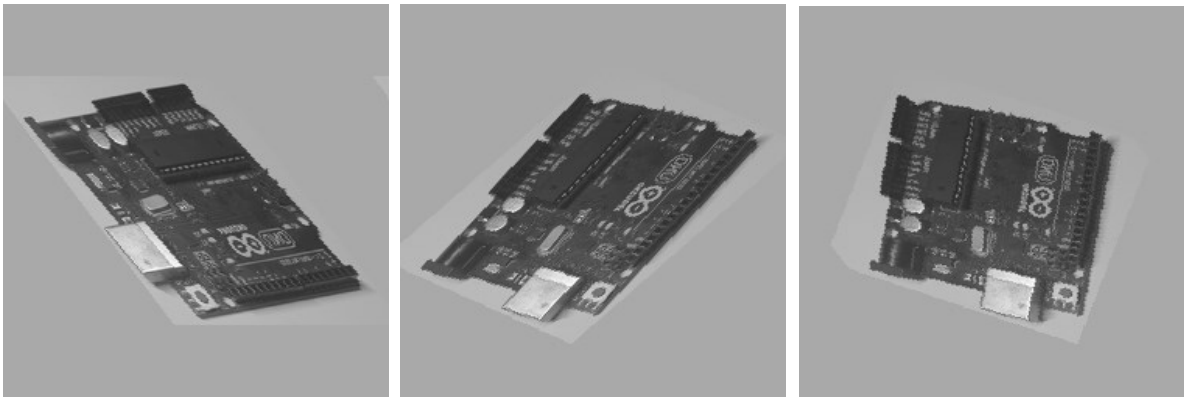
$$S_1(\theta) = \begin{bmatrix} 1 & -\tan\left(\frac{\theta}{2}\right) \\ 0 & 1 \end{bmatrix} \tag{2.25}$$

$$S_2(\theta) = \begin{bmatrix} 1 & 0 \\ sin\left(\frac{\theta}{2}\right) & 1 \end{bmatrix}$$

$$S_3(\theta) = \begin{bmatrix} 1 & -\tan\left(\frac{\theta}{2}\right) \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = S_1(\theta) \cdot S_2(\theta) \cdot S_3(\theta) \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & -\tan\left(\frac{\theta}{2}\right) \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ sin\left(\frac{\theta}{2}\right) & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -\tan\left(\frac{\theta}{2}\right) \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.26}$$

V programu je násobení střihovými maticemi s hodnotami souřadnic prováděno postupně od matice S_3 až po matici S_1. Díky tomu, že v každém kroku násobení je změněna pouze souřadnice v jedné ose, nedochází k efektům Aliasingu, jak lze vidět na obrázku 2.31.



Obrázek 2.31 – Postupné otáčení obrazu metodou tří střihů

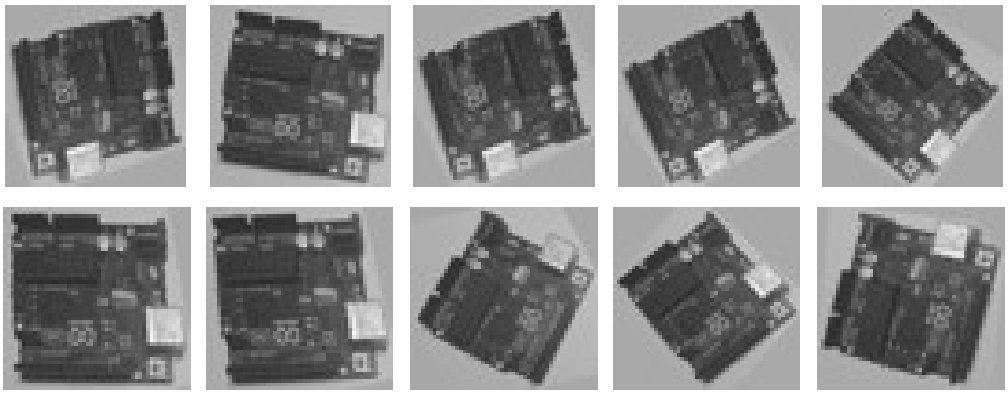
Funkce *Otocit_Obraz* je určena ke změně obrazu použitím rotace, která obsahuje náhodný prvek, takže každá transformace vstupního obrazu je unikátní. Zpočátku funkce načte data ze vstupního obrazu a vytvoří nový, o něco větší obraz. Zvětšení obrazu je provedeno kvůli následující transformaci střihovými maticemi, kdy výsledný transformovaný obraz má větší rozměr, než obraz vstupní. Barva prvního pixelu z původního obrazu se použije k nastavení barvy pozadí nového obrazu, čímž se zachová vizuální konzistence.

Funkce přepočítá hodnoty souřadnic transformovaného obrazu z obrazu původního použitím střihových matic. To zahrnuje úpravu souřadnic každého pixelu v původním obrazu podle úhlu natočení a jeho následné umístění na odpovídající pozici v novém obrazu.

```
def Otocit_Obraz(obr, objekt, i):
    pix = obr.load()

    novy_obr = Image.new(mode="L", size=(obr.size[0]+100, obr.size[1]+100), color=pix[0, 0])
    novy_pix = novy_obr.load()
    n = math.radians(mm.Random()[0] *180-90)

    for y in range(obr.size[1]):
        for x in range(obr.size[0]):
            novy_x = (x-(obr.size[0]/2)) - (y-(obr.size[1]/2))*math.tan(n/2) -novy_obr.size[0]/2
            novy_y = (y-(obr.size[1]/2)) -novy_obr.size[1]/2
            try:
                novy_pix[novy_x,novy_y] = pix[x,y]
            except:
                pass

    novy_obr.save("Dataset\\"+objekt+"\\Preprocess\\"+str(i)+".png")
    obr = Image.open("Dataset\\"+objekt+"\\Preprocess\\"+str(i)+".png").convert('L')
    pix = obr.load()
```

Obrázek 2.32 – Funkce pro rotaci původního obrazu

Následně jsou využity funkce z kapitoly 5.2 pro opětovné nalezení objektu na obrazu a tím pádem oříznutí okrajů, které byly kvůli rotaci na začátku funkce přidány. Všechny transformované obrazy jsou zmenšeny na rozměry 50×50.



Obrázek 2.33 – Výsledný dataset z jednoho vstupního obrazu

2.3 VYTVOŘENÍ KONVOLUČNÍ NEURONOVÉ SÍTĚ

Pro programování jednotlivých výpočetních vrstev (konvoluční, vyhlazovací a plně-propojené) byly využity matematické funkce popsané v kapitole 4. Všechny prvky CNN jsou programovány jako pod-třídy hlavní třídy *Vrstva*. Díky tomu je možné nastavit vlastnosti vrstvy na začátku programu a následně danou vrstvu využívat dále. Tento přístup navíc umožňuje vytvoření vícero vrstev stejného typu, aniž by bylo zapotřebí větších zásahů do již existujícího kódu.

```
sit = [
    Vrstva.Konvoluce((1, 28, 28), 3, 5), Aktivace.Sigmoid(),
    Vrstva.Zmena_Tvaru((5, 26, 26), (5*26*26, 1)),
    Vrstva.Plne_Propojena(5*26*26, 100), Aktivace.Sigmoid(),
    Vrstva.Plne_Propojena(100, 2), Aktivace.Sigmoid()
]
```

Obrázek 2.34 – Vytvoření neuronové sítě

2.3.1 KONVOLUČNÍ VRSTVA

Třída Konvoluce, která představuje konvoluční vrstvu, má tři funkce - *__init__*, *dopredne_siremi_predikce* a *zpetne_sireni_chyby*. Funkce *__init__* inicializuje instanci třídy. Funkce pro své fungování potřebuje znát několik vstupních parametrů, jako je *tvar_vstupu*, *velikost_kernelu* a *hloubka*. Proměnná *tvar_vstupu* je tuple, který definuje tvar (hloubka, výška, šířka) vstupního obrazu, kde hloubka představuje počet barevných složek (1 pro obrazy v odstínech šedi, 3 pro obrazy s barvami RGB), respektive počet výstupů z předchozí vrstvy. Proměnná *velikost_kernelu* je celé číslo určující výšku a šířku jádra čtvercového filtru. Parametry filtru jsou nastaveny v těle funkce. Parametr *hloubka* představuje počet výstupních obrazů, tedy kolik různých filtrů se má na původní vstupní obraz aplikovat.

Uvnitř funkce *__init__* se vypočítá proměnná *tvar_vystupu*, která definuje rozměry výstupních obrazů. Prostorové rozměry jsou zmenšeny na základě velikosti jádra v důsledku operace konvoluce bez výplně (paddingu). Definuje se také *tvar_kernelu*, který udává tvar konvolučních filtrů. Ve výpočtu je zahrnut počet výstupních kanálů, vstupních kanálů a rozměry jádra. Parametry používaných kernelů jsou tvořeny náhodnými hodnotami představujícími váhy konvolučních filtrů. Self.biases je podobně inicializován náhodnými hodnotami a odpovídá tvaru výstupu.

```
class Konvoluce:
    def __init__(self, tvar_vstupu, velikost_kernelu, hloubka):
        hloubka_vstupu, vyska_vstupu, sirka_vstupu = tvar_vstupu
        self.hloubka = hloubka
        self.tvar_vstupu = tvar_vstupu
        self.hloubka_vstupu = hloubka_vstupu
        self.tvar_vystupu = (hloubka, vyska_vstupu-velikost_kernelu+1, sirka_vstupu-velikost_kernelu+1)
        self.tvar_kernelu = (hloubka, hloubka_vstupu, velikost_kernelu, velikost_kernelu)
        self.kernely = np.random.randn(*self.tvar_kernelu)
        self.prahy = np.random.randn(*self.tvar_vystupu)
```

Obrázek 2.35 – Inicializace konvoluční vrstvy

Funkce dopředného průchodu (forward) přijímá jako argument *vstupní_data*, která představují vstupní data pro tuto vrstvu. Funkce pak pro každou oblast na vstupních obrazech, provede 2D křížovou korelaci mezi vstupní daty a odpovídajícím jádrem. Tato operace aplikuje konvoluční jádro na vstupní data a výsledkem je mapa příznaků pro každý výstupní obraz. Výsledek této operace se přičte k počátečním hodnotám prahů neuronů.

```
def Dopredne_Sireni_Predikce(self, vstupni_hodnoty):
    self.vstup = vstupni_hodnoty
    self.vystup = np.copy(self.prahy)
    for i in range(self.hloubka):
        for j in range(self.hloubka_vstupu):
            self.vystup[i] += signal.correlate2d(self.vstup[j], self.kernely[i, j], "valid")
    return self.vystup
```

Obrázek 2.36 – Dopředné šíření konvoluční vrstvy

Pro zpětné šíření je vytvořena funkce *Zpetne_Sireni_Chyby*, která se využívá pro trénování neuronové sítě. Přijímá hodnotu výstupního gradientu z předcházející vrstvy (gradient ztráty vzhledem k výstupu vrstvy) a parametr α (*a*), který určuje rychlost učení. Dále je vytvořeno pole pro uložení gradientů ztrát vzhledem k jádrům a pole pro uložení gradientů ztrát vzhledem ke vstupu. Pro každý výstupní a vstupní kanál je spočítána korelace vstupu s gradientem výstupu. Tím se hodnoty filtrů vrstvy upraví podle toho, jak moc přispěly k chybě výstupu. Rovněž aktualizuje vstupní gradient, čímž se chyba šíří zpět na vstup vrstvy (a tady výstup vrstvy předchozí).

```
def Zpetne_Sireni_Chyby(self, gradient_vystupu, a):
    gradient_kernelu = np.zeros(self.tvar_kernelu)
    gradient_vstupu = np.zeros(self.tvar_vstupu)

    for i in range(self.depth):
        for j in range(self.hloubka_vstupu):
            gradient_kernelu[i, j] = signal.correlate2d(self.vstup[j], gradient_vystupu[i], "valid")
            gradient_vstupu[j] += signal.convolve2d(gradient_vystupu[i], self.kernely[i, j], "full")

    self.kernely -= a * gradient_kernelu
    self.prahy -= a * gradient_vystupu
    return gradient_vstupu
```

Obrázek 2.37 – Konvoluční vrstva – zpětné šíření

2.3.2 VYHLAZOVACÍ VRSTVA

Vyhlazovací vrstva, v programu obecněji pojmenovaná jako *Změna_Tvaru*, se používá k úpravě tvaru dat v neuronové síti, aniž by se změnil jejich obsah. Tato vrstva slouží jako „prostřední“ mezi vrstvou konvoluční, která zpracovává dvourozměrná, respektive třírozměrná pole dat, a vrstvou plně-propojenou, která zpracovává pouze jednorozměrný sloupcový vektor.

Funkce *__init__* slouží k inicializaci dále využívaných proměnných. Přijímá dva vstupní parametry, *tvar_vstupu* a *tvar_vystupu*. Proměnná *tvar_vstupu* udává očekávaný tvar vstupních dat do této vrstvy, *tvar_vystupu* popisuje nový tvar, do kterého mají být vstupní data transformována. Funkce *Dopredne_Sireni_Predikce* zpracovává dopředné šíření přes tuto vrstvu. Přijímá jeden parametr, *vstupni_hodnoty*. Ta představuje vstupní data, která mají být transformována. Rozměr těchto dat musí odpovídat hodnotě uložené v proměnné *tvar_vstupu*.

Metoda *Zpetne_Sireni_Chyby* je určena pro zpětné šíření touto vrstvou při procesu učení. Přijímá dva parametry, *gradient_vystupu* – gradient ztráty vzhledem k výstupu této vrstvy, a proměnnou pro parametr α, který představuje hodnotu rychlosti učení. Tento parametr není ve funkci nijak využit (jelikož tato vrstva nemá žádné trénovatelné parametry), je zahrnut jen z důvodu, aby byla zachována konzistentní struktura metody s ostatními vrstvami, které jej vyžadují. Tato metoda zajišťuje, že gradient může být správně šířen zpět sítí a odpovídá tvaru vstupu vrstvy

```
class Zmena_Tvaru:
    def __init__(self, tvar_vstupu, tvar_vystupu):
        self.tvar_vstupu = tvar_vstupu
        self.tvar_vystupu = tvar_vystupu

    def Dopredne_Sireni_Predikce(self, vstupni_hodnoty):
        return np.reshape(vstupni_hodnoty, self.tvar_vystupu)

    def Zpetne_Sireni_Chyby(self, gradient_vystupu, a):
        return np.reshape(gradient_vystupu, self.tvar_vstupu)
```

Obrázek 2.38 – Změna tvaru matice

.

2.3.3 PLNĚ-PROPOJENÁ VRSTVA

Třída *Plne_Propojena* představuje plně propojenou (někdy také označovaná jako hustá) vrstvu v neuronové síti. Stejně jako konvoluční vrstva (popsaná v kapitole 6.1) má tato třída tři funkce: inicializační, metodu dopředného šíření (*Dopredne_Sireni_Predikce*) a metodu zpětného šíření (*Zpetne_Sireni_Chyby*). Tato vrstva je běžným prvkem většiny neuronových sítí.

Inicializace třídy přijímá dva parametry: *pocet_vstupu* a *pocet*_perceptronu. Dále funkce vytvoří matici vah jako dvourozměrné pole náhodných generovaných hodnot pro všechny perceptrony. Matice vah je vytvořena s Gaussově rozloženými hodnotami a má velikost podle počtu vstupů. Prahy (biasy) neuronů jsou tvořeny obdobným způsobem, jako tomu bylo u matice vah.

```
class Plne_Propojena:
    def __init__(self, pocet_vstupu, pocet_perceptronu):
        self.vahy = [mm.Random(pocet_vstupu, 'Gauss') for i in range(pocet_perceptronu)]
        self.prahy = [mm.Random(1, 'Gauss') for i in range(pocet_perceptronu)]
```

Obrázek 2.39 – Inicializace plně-propojené vrstvy

Funkce *Dopredne_Sireni_Predikce* provádí dopředné šíření pomocí vstupních hodnot *vstupni_hodnoty*. Metoda vypočítá součin váhových matic se vstupními hodnotami, které následně sečte s hodnotami prahů jednotlivých neuronů. Výstupem této funkce je aktivace neuronů, vektor výsledných hodnot lze předat následující vrstvě nebo použít při výstupních predikcích.

```
def Dopredne_Sireni_Predikce(self, vstupni_hodnoty):
    self.vstup = vstupni_hodnoty
    vystup = mm.dot(self.vahy, self.vstup)
    vystup = mm.sum(vystup, self.prahy)
    return vystup
```

Obrázek 2.40 – Dopředné šíření predikce plně-propojené vrstvy

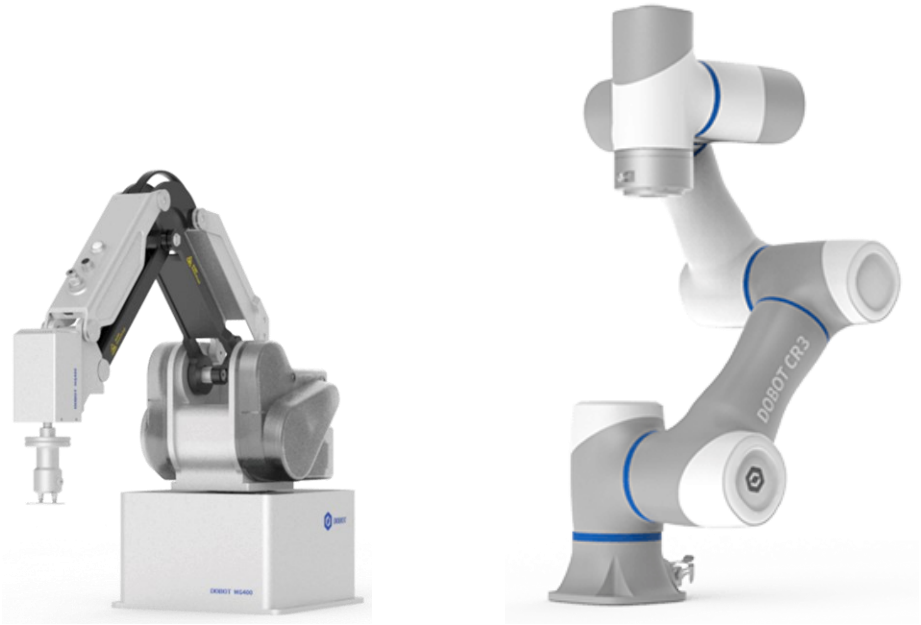
Metoda pro zpětného šíření zprostředkovává trénování neuronové sítě pomocí úprav vah a prahů jednotlivých neuronů. Do metody vstupuje *gradient_vystupu* (gradient ztráty vzhledem k výstupu této vrstvy) a rychlost učení α. V těle metody je spočítán gradient vah jako součin gradientu výstupu a transponovaného vektoru vstupu. Dále je vypočten gradient vstupu jako součin transponované matice vah a gradientu výstupu. Váhy a prahy se následně aktualizují odečtením součinu míry učení a jejich příslušných gradientů. Metoda vrací *gradient_vstupu*, což je gradient ztráty vzhledem ke vstupu vrstvy.

```
def Zpetne_Sireni_Chyby(self, gradient_vystupu, a):
    gradient_vah = mm.dot(gradient_vystupu, mm.T(self.vstup))
    gradient_vstupu = mm.dot(mm.T(self.vahy), gradient_vystupu)
    self.vahy = mm.sub(self.vahy.copy(), mm.dot(a, gradient_vah))
    self.prahy = mm.sub(self.prahy, mm.dot(a,gradient_vystupu))
    return gradient_vstupu
```

Obrázek 2.41 – Zpětné šíření chyby plně-propojené vrstvy

2.3.4 KOMUNIKACE S ROBOTICKÝM SYSTÉMEM

V rámci ověření funkčnosti navrženého řešení robotického vidění byla vytvořena testovací aplikace s využitím průmyslových robotů firmy Dobot. Konkrétně se jedná o roboty MG400 a roboty řady CR. Dobot MG400 je malé stolní průmyslové rameno se čtyřmi stupni volnosti pohybu. S dosahem ramene až 440 mm a nosností do 750 g je vhodným řešením spíše pro jednoduché až středně velké typy aplikací, jako je paletizace či bin-picking. Výrobce udává přesnost a opakovatelnost tohoto ramene ±0,05 mm. Roboty řady CR jsou šestiosé angulární roboty. Ve srovnání s roboty MG400 mají větší dosah i nosnost. Jednotlivé modely této řady se liší svým dosahem a nosností a vyhovují tak různým průmyslovým aplikacím. Nosnost se pohybuje od 5 do 15 kilogramů, dosah může v závislosti na konkrétním modelu dosahovat až 1500 milimetrů.



Obrázek 2.42 – Roboty MG400 a CR3

Pro komunikaci s roboty byly využity dva komunikační protokoly – komunikaci pomocí Sériové linky a prostřednictvím protokolu Modbus. Jedná se o standardní protokoly běžně používané v průmyslovém prostředí. Je tedy možné využít programy popsané v kapitolách 7.1 a 7.2 i pro jiné typy robotických ramen, které podporují alespoň jeden z těchto druhý komunikace, s minimální potřebou úprav.

2.3.5 SÉRIOVÁ KOMUNIKACE

Sériová komunikace je založena na posílání bitů po sobě jdoucí sérii. Existují různé standardy sériové komunikace, přičemž RS-232, RS-485, a UART jsou mezi nejznámějšími. Pro přenos dat jsou využívány dva vodiče – jeden pro přenos (TX) a druhý pro příjem (RX).

Výhody sériové komunikace spočívají v její jednoduchosti a efektivitě. Díky svému jednoduchému zapojení a omezenému počtu vodičů je snadná na implementaci. Méně kabelů znamená také nižší náklady na výrobu a údržbu. Sériová komunikace umožňuje přenos dat na delší vzdálenosti bez ztráty kvality signálu. Přenos dat je sekvenční, což znamená, že musí být přenášen jeden bit za druhým. To může vést ke zpomalení přenosu dat ve srovnání s paralelní komunikací, která umožňuje přenos více bitů najednou. Navíc, při vysokých přenosových rychlostech, může dojít k chybám při špatné kvalitě nebo rušení signálu.

Délka jednotlivých zpráv pro sériovou komunikaci s robotem není pevně stanovena. Příkazové pakety se skládají z hlavičky, hodnoty délky posílané zprávy, samotné zprávy a kontrolního součtu. Pakety jsou tvořeny parametry, a každý parametr se skládá ze dvou hexadecimálních hodnot. Robot okamžitě vykoná a odpoví na přijatý paket – pokud byl správně interpretován. Na obrázku 2.43 je znázorněno, jak vypadá příkaz getPose (příkaz k získání aktuální pozice robotu).

<table>
<tr><td>Hex</td><td>AA</td><td>AA</td><td>02</td><td>0A</td><td>00</td><td>F6</td></tr>
<tr><td>Dec</td><td>170</td><td>170</td><td>2</td><td>10</td><td>0</td><td>246</td></tr>
<tr><td rowspan="2"></td><td colspan="2" rowspan="2">Header</td><td rowspan="2">Len</td><td>ID</td><td>Ctrl</td><td rowspan="2">Checksum</td></tr>
<tr><td colspan="2">Payload</td></tr>
</table>

Obrázek 2.43 – Reprezentace příkazu GetPose (Johnston, 2019)

Význam jednotlivých parametrů je následující:

- Header (hlavička): První dvě hodnoty v každém příkazu jsou vždy 'AA AA', což znamená začátek nového paketu.
- Len (délka): počet parametrů ve zprávě (tyto parametry jsou na obrázku 2.43 označeny oranžovou barvou)
- Payload (zpráva): Strukturu zprávy každého instrukčního příkazu lze nalézt v dokumentu komunikačního protokolu a liší se u každého příkazu, ale vždy začíná podpaketem ID a Ctrl. Desítková hodnota ID odpovídá číslu ID spojenému s každým příkazem. Podpaket Ctrl ukazuje, zda je příkaz pro čtení nebo zápis informací (RW) a zda je příkaz zařazen do fronty (isQueue).
- Checksum (kontrolní součet): Hodnota kontrolního součtu se počítá pouze ze zprávy příkazu a slouží k označení konce instrukce a k poskytnutí jistoty, že data byla přenesena správně.

Pro čtení přijatých zpráv z robotu je princip obdobný. Na obrázku 2.44 je zobrazena reprezentace přijaté odpovědi na příkaz pro získání polohy (obrázek 2.43). Každý podpaket pozice v rámci vrácené části nákladu je 32bitové číslo typu float. Po převodu všech paketů ve zprávě lze zjistit souřadnice aktuální polohy ramene jak v kartézském souřadném systému, tak v úhlových souřadnicích natočení jednotlivých kloubů.

Hex	AA	AA	22	0A	00	83	35	16	43	A4	93	8F	BF	10	67	A0	41	48	0F	DB	BE	00	31	A3	BE	DD	B8	FB	BE	F0	C8	4D	42	00	00	00	00	F1
Dec	170	170	34	10	0	131	53	22	67	164	147	143	191	16	103	160	65	72	15	219	190	0	49	163	190	221	184	251	190	240	200	77	66	0	0	0	0	241
	Header		Len	ID	Ctrl	X				Y				Z				r				Base angle				Rear arm angle				Forearm angle				EndEffector angle				Checksum
				Payload																																			

Obrázek 2.44 – Odpověď robotu na příkaz pro určení polohy (Johnston, 2019)

Každý podpaket obsahující hodnotu souřadnice v rámci vrácené části zprávy je 32bitové číslo typu float. Tuto hodnotu je nutné nejdříve převést z bajtové reprezentace (pro hodnotu souřadnice X je bajtová hodnota reprezentována v šestnáctkové soustavě 83 35 16 43) na hodnotu datového typu float. V případě příkazů pro pohyb ramene je nutné stejným principem převádět hodnoty souřadnic datového typu float na jednotlivé skupiny bajtů.

Pro komunikace s roboty firmy Dobot byla definována třída *dobot_serial*. Pro inicializaci třídy (*__init__*) je nejdříve zapotřebí nastavit COMport, který specifikuje sériový port pro připojení k robotu. Dále se nastaví parametry sériové komunikace, jako baudrate 115200 baudů, osm datových bitů a jeden stop bit. Při neúspěšném pokusu o připojení je vypsáno upozornění. Následně funkce vyzkouší, zda je možné s k robotu připojit. Funkce *Zahajit_Komunikaci* otevře komunikaci přes sériový port s parametry nastavenými v inicializaci třídy. Pokud není možné se k robotu připojit, vypíše funkce chybovou hlášku.

```
class dobot_serial:
    def __init__(self, COMport):
        self.pripojeno = False

        try:
            self.dobot = serial.Serial(port=COMport, baudrate=115200, bytesize=serial.EIGHTBITS,
                stopbits=serial.STOPBITS_ONE, parity=serial.PARITY_NONE, timeout=5)
        except:
            print("Nelze se připojit ke Comportu")
        else:
            print("V pořádku")
            self.dobot.close()

    def Zahajit_Komunikaci(self):
        try:
            self.dobot.open()
        except:
            print("chyba připojení")
            return(False)
        else:
            if(self.dobot.is_open):
                self.pripojeno = True
                print("Připojeno")
                return(True)
```

Obrázek 2.45 – Zahájení komunikace po sériové lince

Funkce *Pohyb* umožňuje odeslání příkazu pro pohyb robota na určitou pozici specifikovanou souřadnicemi x, y, z, r, ale pouze pokud je port otevřený a připojení aktivní. Souřadnice jsou baleny do binárního formátu s kódováním Big Endian a odeslány přes sériový port. Funkce *Poloha* odesílá příkaz pro získání aktuální polohy robota. Odpověď je přijímána jako bajtový řetězec a je tedy nutné pomocí smyčky for převést hodnoty souřadných os zpět na datový typ float.

```
def Pohyb(self, x, y, z, r):
    if(self.pripojeno):
        x = bytearray(struct.pack(">f", x))
        y = bytearray(struct.pack(">f", y))
        z = bytearray(struct.pack(">f", z))
        r = bytearray(struct.pack(">f", r))

        try:
            self.dobot.write(b'\xAA\xAA\x13\x54\x03\x01'+x+y+z+r+'\xAF')
        except:
            print("Chyba řetězce")
            return(False)
        else:
            return(True)

def Poloha(self):
    if(self.pripojeno):
        try:
            self.dobot.write(b'\xAA\xAA\x02\x0A\x00\xF6')
        except:
            print("Chyba řetězce")
            return(False)
        else:
            time.sleep(0.5)
            result = 0
            data = ser.read(38).hex()
            for byte in data:
                result = (result << 8) | byte
            print(result)
```

Obrázek 2.46 – Funkce pro pohyb robotu pomocí sériové

2.3.6 PROTOKOL MODBUS

Modbus je standardní průmyslový protokol, který slouží k výměně dat mezi zařízeními v automatizovaných systémech. Jeho jednoduchost, spolehlivost a podpora různých komunikačních rozhraní ho činí častou volbou pro komunikaci v průmyslovém prostředí. Modbus umožňuje přenos příkazů, stavů a dalších dat mezi různými zařízeními, která mohou být fyzicky oddělena a používat odlišné platformy. Ovládání robotického ramene a jeho monitorování vyžaduje komunikaci s řídicím systémem, kterým může být buď PC nebo PLC. Využití protokolu Modbus pro tuto komunikaci umožňuje řídicímu systému odesílat příkazy a požadavky na robotické rameno MG400, stejně jako získávat stavové informace a data o jeho provozu.

Robot používá ke komunikaci pomocí sběrnice Modbus dva základní typy registrů – Coil register a Holding register. Coil Register slouží k ovládání binárních vstupů robotu, jako

je spuštění, pozastavení či úplné zastavení vykonávání programu. Holding Register slouží k ukládání a udržování hodnot, které mohou být čteny a zapisovány. Může být využit pro uchovávání nastavení, proměnných hodnot nebo jiných datových informací.

V současné době není možné posílat příkazy pro pohyb ramene robotu přímo. Stejného efektu však lze docílit připravením pomocného programu v prostředí DobotStudio Pro a zapisováním hodnot do Holding registru robotu. Program běžící v robotu pomocí těchto hodnot provede pohyb na pozici, která je zadána řídicím systémem. V prostředí Python uživatel zadá požadované souřadnice, kam je potřeba rameno robotu přesunout.

Na začátku funkce byly naimportovány dvě funkce z knihovny PyModbus – funkce ModbusTcpClient pro připojení se k robotu a zahájení komunikace a BinaryPayloadBuilder pro převod hodnot souřadnic z datového typu float na jednotlivé bajty. V dalším kroku funkce se iteruje proměnná *pole_souradnice*, přičemž každá souřadnice je převedena do binární formy jako 32bitové float číslo a přidána do konstruktoru. Jelikož je potřeba zapsat horní a dolní slovo 32bitového float čísla do dvou 16bitových registrů, provádí se zápis do robotického systému po dvojicích registrů pro každou souřadnici (x, y, z). Na specifických adresách registrů 3096 až 3101 jsou zapsány jednotlivé souřadnice. Do registru 3102 je zapsán identifikátor objektu, který rozpoznal systém robotického vidění. Na závěr je komunikace s robotem ukončena zavoláním funkce close().

```
# naimportování potřebných knihoven
from pymodbus.client import ModbusTcpClient
from pymodbus.payload import BinaryPayloadBuilder, Endian

def Posun_Robotu(ip, pole_souradnice, objekt)
# Připojení klienta
    klient = ModbusTcpClient(ip)
    print(klient.connect())

    # Vytvoření kostruktoru
    konstruktor = BinaryPayloadBuilder(byteorder=Endian.Big, wordorder=Endian.Big)
    konstruktor.reset()

    # Zpracování souřadnic do potřebného tvaru
    for souradnice in pole_souradnic:
    konstruktor.add_32bit_float(souradnice)
    pole_souradnic = konstruktor.to_registers()

    # Zapsání souřadnice x do robotu
    klient.write_register(3096, pole_souradnic[1])
    klient.write_register(3097, pole_souradnic[0])

    # Zapsání souřadnice y do robotu
    klient.write_register(3098, pole_souradnic[3])
    klient.write_register(3099, pole_souradnic[2])

    # Zapsání souřadnice z do robotu
    klient.write_register(3100, pole_souradnic[5])
    klient.write_register(3101, pole_souradnic[4])

    # Určení rozpoznávaného objektu
    klient.write_register(3102, objekt)

    # ukončení komunikace
    klient.close()
```

Obrázek 2.47 – Program pro komunikaci přes Modbus

2.4 TESTOVACÍ APLIKACE

Algoritmy popsané v předchozích kapitolách byly otestovány prostřednictvím tří různých aplikací. První z nich se zaměřuje na identifikaci dvou navzájem odlišných tříd objektů, což slouží k ověření vyvinutého algoritmu v běžných aplikacích konvolučních neuronových sítí. Druhá aplikace zkouší schopnost sítě rozpoznávat objekty ve dvou třídách, které jsou si vizuálně podobné, a tím testuje, jak dobře se síť dokáže učit z jemných rozdílů ve vstupních obrazech. Třetí testovací aplikace hodnotí výkon sítě při klasifikaci většího počtu tříd objektů než v předchozích případech. Výsledné grafy jsou tvořeny daty skutečných měření (světlé průběhy) a jejich průměrem (syté průběhy).

Úlohy byly zkoušeny na třech různých hardwarových soustavách kvůli ověření optimalizace výpočtů. Jelikož nebyly vytvořeny ani využity žádné funkce pro paralelní výpočty pomocí grafických karet, jsou brány v úvahu pouze parametry procesoru a operační paměti zařízení. Parametry použitých soustav jsou uvedeny v tabulce 2.1.

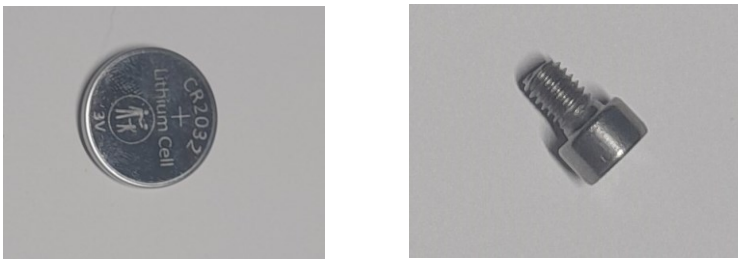
Tabulka 2.1 – Parametry použitých soustav

Model	Procesor [GHz]		RAM [GB]
MSI MEG INFINITE X 11TD-1026AT TWR	Intel® Core™ i7-10700KF	3,8 - 5,1	32
Vlastní sestava	Intel® Core™ i5-9400F	2,9 - 4,1	16
Lenovo IdeaPad Flex 5-14ARE05	AMD Ryzen 5 4500U	2,3 - 4,0	16

Jelikož byla úloha lokalizace řešena odděleně od kategorizace objektů, není zohledněna ve výše popsaných měřeních. Při testování lokalizace pomocí robotického ramene se ukázalo toto řešení jako ne zcela vhodné, přesto však dostatečné. Největší nevýhodou tohoto řešení je, že není brána v potaz orientace snímaného objektu. Pro korektní fungování je tedy zapotřebí použít předměty, u které není jeho orientace rozhodující pro manipulaci, či zaručit neměnnost jejich natočení. Dále je nutné zajistit konstantní světelné podmínky pracoviště. Pokud jsou výše zmíněné podmínky splněny, program dokáže zajistit dostatečně přesnou lokalizaci snímaného objektu a jeho následnou manipulaci.

2.4.1 DETEKCE DVOU TŘÍD OBJEKTŮ

K rozpoznání a klasifikaci dvou specifických typů objektů byl zvolen dataset obsahující obrazy baterie typu CR 2032 a šroubu. Tyto objekty byly vybrány z důvodu jejich značné rozdílnosti tvaru, což umožnilo využití relativně jednoduchého přístupu při návrhu samotné neuronové sítě.



Obrázek 2.48 – Dataset pro detekci dvou tříd objektů

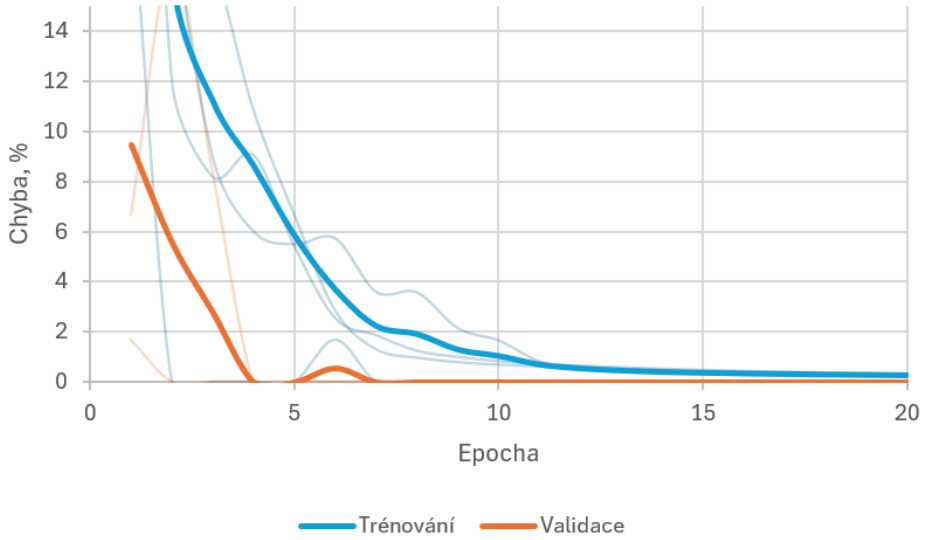
Pro účely klasifikace byla navržena neuronová síť sestávající z několika standaradních vrstev. Hlavní architekturu tvořila jedna konvoluční vrstva, která byla zodpovědná za extrakci příznaků ze vstupních obrazů. Po konvoluční vrstvě následovaly dvě vrstvy plně-propojené, které sloužily k dalšímu zpracování a klasifikaci extrahovaných příznaků na základě naučených vah. Topologie sítě byla následující:

- konvoluční vrstva 5 použitých filtrů,
- vyhlazovací vrstva,
- plně-propojená vrstva 100 neuronů,
- plně-propojená vrstva 2 neurony.

Neuronová síť byla trénována na základě datasetu, který byl rozdělen na trénovací a validační množinu, aby bylo možné monitorovat a optimalizovat výkon sítě a snížit tak případný vliv přeučení. Celkem bylo pro učení použito 70 obrazů každé třídy. Validační množina obsahovala 20 obrazů každé třídy. Trénování bylo prováděno do té doby, dokud nedošlo k stabilizaci výkonu sítě.

Tabulka 2.2 – Časová náročnost učení

Model	Doba učení [h]	Doba predikce [s]	Správnost predikce [%]
MSI MEG INFINITE X 11TD-1026AT TWR	2,1	1,21	99,5
Vlastní sestava	3,4	1,45	99,5
Lenovo IdeaPad Flex 5-14ARE05	3,7	1,47	99,5

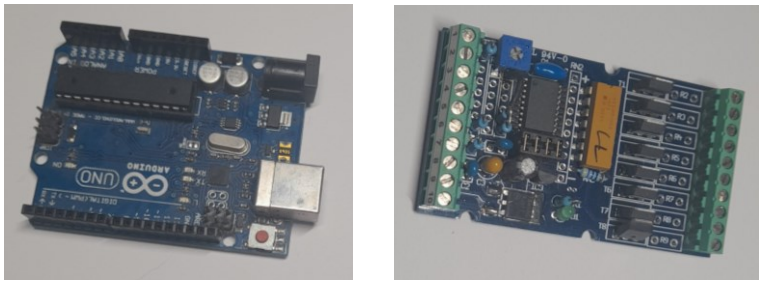


Obrázek 2.49 – Průběh učení detekce dvou tříd objektů

Po dokončení trénovacího procesu byla schopnost sítě rozlišit mezi obrazy baterií a šroubů testována na nezávislé testovací množině. Síť byla testována na 20 obrazech objektů každé třídy. Výsledná pravděpodobnost správné predikce byla v průměru 99,5 %.

2.4.2 DETEKCE DVOU TŘÍD PODOBNÝCH OBJEKTŮ

Pro tuto aplikaci byl vytvořen dataset obsahující obrazy vývojové desky Arduino a plošného spoje rezistorové sítě. Přestože bylo testováno několik různých architektur neuronových sítí, žádná z nich nedosáhla požadované úrovně přesnosti, s maximální hodnotou správné predikce pouze 68 %. Nejedná se tedy o spolehlivé řešení. Pro dosažení lepších výsledků by bylo pravděpodobně zapotřebí pracovat s obrazy, které mají větší rozlišení než 50×50 pixelů a poměrně složitá konvoluční část sítě, což v rámci navrhovaného řešení nebylo realizováno z důvodů vysoké výpočetní i časové náročnosti procesu učení.



Obrázek 2.50 – Dataset dvou tříd podobných objektů

Pro účely klasifikace byla navržena komplexnější neuronová síť, než která byla použita pro rozpoznávání dvou relativně odlišných objektů. Hlavní architekturu sítě tvoří dvě

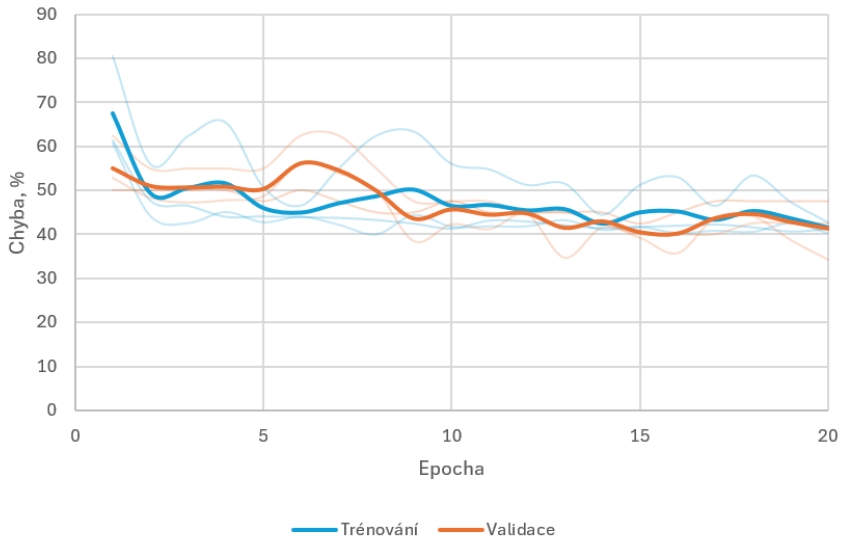
konvoluční vrstvy, každá vybavená pěti filtry. Po konvolučních vrstvách následuje vyhlazovací vrstva, která slouží k redukci prostorové velikosti reprezentace. Závěrečnou část architektury tvoří tři plně-propojené vrstvy. První dvě plně-propojené vrstvy obsahují každá 200 neuronů a zpracovávají příznaky získané z předchozích vrstev, přičemž poslední plně-propojená vrstva obsahuje 2 neurony a slouží k závěrečné klasifikaci objektů. Topologie sítě byla následující:

- konvoluční vrstva 5 použitých filtrů,
- konvoluční vrstva 5 použitých filtrů,
- vyhlazovací vrstva,
- plně-propojená vrstva 200 neuronů,
- plně-propojená vrstva 200 neuronů,
- plně-propojená vrstva 2 neurony.

Trénování neuronové sítě probíhalo na základě předem připraveného datasetu, rozděleného na trénovací a validační množinu obrazů. Pro trénovací proces bylo použito celkem 70 obrazů pro každou třídu rozpoznávaných objektů. Pro validaci výkonu sítě bylo vyčleněno 20 obrazů původního datasetu z každé kategorie.

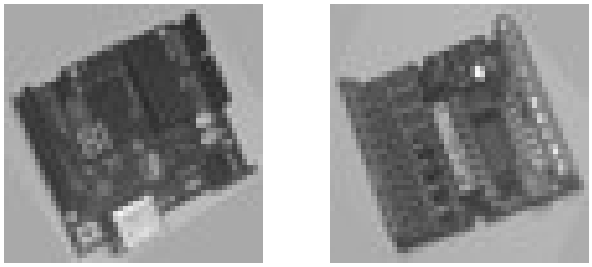
Tabulka 2.3 – Časová náročnost učení

Model	Doba učení [h]	Doba predikce [s]	Správnost predikce [%]
MSI MEG INFINITE X 11TD-1026AT TWR	3,3	1,48	69,48
Vlastní sestava	4,8	1,7	68,34
Lenovo IdeaPad Flex 5-14ARE05	5,1	1,75	66,57



Obrázek 2.51 – Průběh učení detekce dvou tříd podobných objektů

Testování navrženého řešení bylo prováděno na zbývajících dvaceti obrazech datasetu pro každou třídu objektů. I přes poměrně komplexní architekturu neuronové sítě nebylo dosaženo spolehlivé predikce třídy rozpoznávaného objektu. Výsledná přesnost sítě je 68 %, což ukazuje více na náhodný výběr třídy než na učení příznaků obrazu. Pro zlepšení výsledků sítě by bylo zapotřebí pracovat s obrazy větších rozměrů než použitých 50×50 pixelů, jelikož na takto zmenšených obrazech dochází ke ztrátám rozhodujících detailů (viz obrázek 2.52).



Obrázek 2.52 – Podobnost vstupních obrazů

2.4.3 DETEKCE TŘÍ TŘÍD OBJEKTŮ

Pro testování detekce tří různých tříd objektů bylo použito datasetu obrazů vývojové desky Arduino, dotykového TFT displeje ILI9488 a šroubu. Tyto objekty byly zvoleny kvůli své vzájemné odlišnosti. Byla použita komplexnější síť, než bylo u výše uvedených příkladů. V architektuře byly použity dvě konvoluční vrstvy a tři vrstvy plně-propojené.

Obrázek 2.53 – Dataset pro detekci tří tříd objektů

Pro účely této aplikace byla použita podobná síť, jako tomu bylo v kapitole 9.2. Díky poměrně velkým rozdílům mezi rozpoznávanými objekty mohla být architektura sítě mírně zmenšena, což přispělo ke kratší době učení i predikce jednotlivých objektů. Architektura použité sítě je následující:

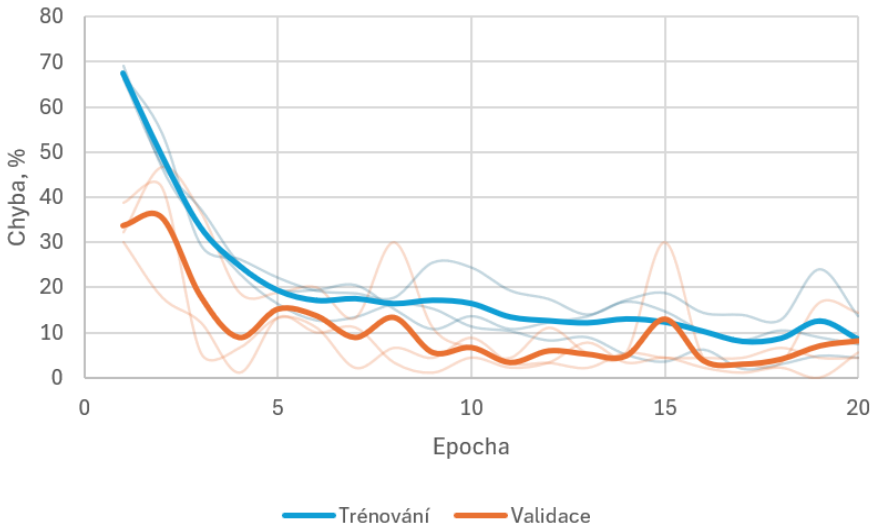
- konvoluční vrstva 5 použitých filtrů,
- konvoluční vrstva 5 použitých filtrů,
- vyhlazovací vrstva,
- plně-propojená vrstva 200 neuronů,

- plně-propojená vrstva 100 neuronů,
- plně-propojená vrstva 3 neurony.

Proces trénování neuronové sítě byl uskutečněn pomocí datasetu, který byl opět rozčleněn na části pro trénování, validaci a testování. V trénovací fázi bylo využito 70 obrazů pro každou klasifikovanou třídu. Validační část datasetu pak obsahovala 20 obrazů z každé třídy, a trénování bylo prováděno až do momentu, kdy byl dosažen stabilní výkon sítě. Celkem tedy bylo použito 210 obrazů pro trénování, 60 pro validaci po každé učící epoše a 60 obrazů datasetu bylo použito pro následné testování sítě.

Tabulka 2.4 – Časová náročnost učení

Model	Doba učení [h]	Doba predikce [s]	Správnost predikce [%]
MSI MEG INFINITE X 11TD-1026AT TWR	3,1	1,36	92,44
Vlastní sestava	4,4	1,59	89,72
Lenovo IdeaPad Flex 5-14ARE05	4,8	1,62	93,15



Obrázek 2.54 – Průběh učení detekce tří tříd objektů

Průměrná hodnota správné predikce byla 91,77 %. Ačkoliv se nejedná o hodnotu správné predikce, která by mohla být využitelná v praxi (přibližně 10 % objektů bylo rozpoznáno nesprávně), dokazuje to schopnost sítě učit se rozpoznávat více tříd objektů zároveň. Pro lepší výsledky by bylo zapotřebí hlubší sítě a přidání konvolučních vrstev, což už je pro tento způsob řešení výpočetně i časově náročné.

3 ZÁVĚR

Cílem této diplomové práce bylo vytvořit vlastní systém robotického vidění s prvky umělé inteligence, schopen lokalizovat a kategorizovat předem naučené objekty. Architektura neuronové sítě byla vystavěna na principech konvolučních neuronových sítí, které jsou v současnosti jednou z nejefektivnějších metod pro zpracování obrazových dat. Výsledná data z konvoluční sítě byla předávána do existujícího robotického systému prostřednictvím standardního protokolu. Navrhnuté řešení bylo otestováno na třech aplikacích – kategorizace objektů dvou tříd, kategorizace podobných objektů dvou tříd a kategorizace objektů tří tříd.

Výsledky praktického ověření funkčnosti řešení na vybraných aplikacích odhalily, že systém dosáhl vysoké úrovně přesnosti v kategorizaci dvou tříd vzájemně odlišných předmětů s průměrnou úspěšností predikce 99,5 %. Tato hodnota dokazuje výkonnost konvoluční neuronové sítě v kontextu správného zpracování a klasifikace obrazových dat, když jsou parametry modelu a datové sady optimální.

V případě aplikace pro detekci dvou tříd vizuálně podobných objektů bylo dosaženo pouze 68% přesnosti, což poukazuje na potenciální omezení navrženého systému pro aplikace rozlišování mezi objekty s minimálními vizuálními rozdíly. Tyto výsledky naznačují, že pro zlepšení přesnosti je zapotřebí větší optimalizace síťové architektury, případně zvýšení rozlišení použitých obrazů, aby byla zlepšena schopnost sítě rozeznávat a učit se relevantní vizuální charakteristiky.

Kategorizace tří tříd objektů dosáhla průměrné správné predikce 91,77 %. Ačkoli tato míra úspěšnosti nemusí být zcela dostatečná pro praktické využití (přibližně 10 % objektů nebylo identifikováno správně) reflektuje tato hodnota schopnost neuronové sítě efektivně se učit rozlišovat mezi více třídami objektů. Pro zvýšení účinnosti predikce by bylo vhodné použít hlubší architekturu neuronové sítě, která je pro toto řešení již časově i výpočetně náročná.

POUŽITÁ LITERATURA

AGGARWAL, C. 2018. Neural networks and deep learning: a textbook. New York: Springer, [cit. 16. 12. 2023]. ISBN 978-3-319-94462-3.

BASKIN, C., LISS, N. a MANDELSON, A. 2017. *Streaming Architecture for Large-Scale Quantized Neural Networks on an FPGA-Based Dataflow Platform*. [online]. ReasearchGate [cit. 12. 3. 2024]. Dostupné z: https://www.researchgate.net/publication/318849314_Streaming_Architecture_for_Large-Scale_Quantized_Neural_Networks_on_an_FPGA-Based_Dataflow_Platform

CAO, Q. a kol. 2021. *Real-Time Vehicle Trajectory Prediction for Traffic Conflict Detection at Unsignalized Intersections*. [online]. Londýn: Hindawi.[cit. 16. 12. 2023] Dostupné z: https://www.hindawi.com/journals/jat/2021/8453726/

FUKUSHIMA, K. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. V: *Biological Cybernetics* vol. 36 (1980), strana 193-202. [online]. Berlín: Springer-Verlag GmbH. [cit. 18. 12. 2023]. ISSN 0340-1200. Dostupné z: https://doi.org/10.1007/BF00344251

GHOSH, A. a kol. 2019. Fundamental Concepts of Convolutional Neural Network. V: *Recent Trends and Advances in Artificial Intelligence and Internet of Things,* vol. 172 (2020), strana 519-567. [online]. Cham: Springer Cham. [cit. 5. 2. 2024]. ISSN 1868-4394. Dostupné z: https://doi.org/10.1007/978-3-030-32644-9_36

CHOLLET, F. 2018. *Deep Learning with Python*. Ney York: Manning. [cit. 16. 1. 2024]. ISBN 978-1-617-29443-3.

IWAMOTO, M. 2018. *Active pattern projection improves AOI 3D measurement accuracy*. V: Vision Systems Design (02. 2018), strana 3. [online]. Sarasota: Vision Systems Design. [cit. 21. 4. 2024].

JHONSTON, J. 2019. *Communicating with the Dobot Magician using raw protocol* [online]. Sydney. [cit. 7. 11. 2023]. Dostupné z: https://jacquesjohnston.wordpress.com/2019/01/29/communicating-with-the-dobot-magician-using-raw-protocol/

KRIZHEVSKY, A.; SUTSKEVER, I. a HINTIN, G. 2017. ImageNet classification with deep convolutional neural networks. V: *Communications of the ACM*, vol 60 (2017), strana 84-90. [online]. New York: Association for Computing Machinery. [cit. 9. 3. 2024]. ISSN 0001-0782. Dostupné z: https://dl.acm.org/doi/10.1145/3065386

LECUN, Y.; KAVUKCUOGLU, K. a FARABET, C. 2010. Convolutional networks and applications in vision. V: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems* (2010), strana 253-256. [online]. Paříž: IEEE. [cit. 21. 2. 2024]. Dostupné z: https://ieeexplore.ieee.org/abstract/document/5537907

PAETH, A. 1986. A fast algorithm for general raster rotation. V: *Proceedings on Graphics Interface '86/Vision Interface '86* (1986), strana 77-81. [online]. Toronto: Canadian Information Processing Society. [cit. 9. 3. 2024]. Dostupné z: https://graphicsinterface.org/wp-content/uploads/gi1986-15.pdf

RASCHKA, S a MIRJALILI, V. 2017. *Python machine learning: machine learning and deep learning with Python, scikit-learn, and TensorFlow*. Birmingham: Pack publishing. [cit. 13. 2. 2024] ISBN 978-1-78712-593-3.

ROBERTS, L. 1937. *Machine perception of three-dimensional solids*. Cambridge: Massachusetts Institute of Technology. [cit. 29. 12. 2023]. Dostupné z: http://hdl.handle.net/1721.1/11589

MALSBURG, C. 1948. Frank Rosenblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. V: *Proceedings of the First Trieste Meeting on Brain Theory* (1984), strana 245-248. [online]. Heidelberg: Springer Berlin. [cit. 26. 2. 2024]. ISSN 978-3-642-70913-5. Dostupné z: https://doi.org/10.1007/978-3-642-70911-1_20

ROTATING IMAGES, 2013. DataGenetics [online]. Seattle [cit. 3. 3. 2024]. Dostupné z: http://datagenetics.com/blog/august32013/index.html

SZELISKI, R. 2010. *Computer Vision: Algorithms and Applications*. Londýn: Springer. [cit. 18. 12. 2023]. ISBN 978-1-84882-934-3.

ŠTURSA, D. 2023. *NSTVI – Strojové vidění* (2023) Pardubice: Univerzita Pardubice. [cit. 16. 4. 2024]

YANI, M.; IRAWAN, B. a SEIANINGSIH, C. 2019. Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail. V: *International Conference on Electronics Representation and Algorithm* (2019). [online]. Bristol: IOPscience. [cit. 4. 4. 2024]. Dostupné z: https://doi.org/10.1088/1742-6596/1201/1/012052

ZANETTI, S. a kol. 2008. Reference evapotranspiration estimate in Rio de Janeiro State using artificial neural networks. V: *Revista Brasileira de Engenharia Agrícola e Ambiental*, vol. 12 (2008), strana 174-180. [online]. Campina Grande: Unidade Acadêmica de Engenharia Agrícola. [cit. 16. 3. 2024]. ISSN 1415-4366. Dostupné z: https://doi.org/10.1590/S1415-43662008000200010