Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Experimentovanie s konvolučnými sieťami na datasete vygenerovanom z malého počtu parametrov

Bakalárska práca

2023
Adam Sršeň

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Experimentovanie s konvolučnými sieťami na datasete vygenerovanom z malého počtu parametrov

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra aplikovanej informatiky
Školiteľ:	RNDr. Andrej Lúčny, PhD.

Bratislava, 2023
Adam Sršeň

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Adam Sršeň
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Experimentovanie s konvolučnými sieťami na datasete vygenerovanom z malého počtu parametrov
Experiments with the convolutional neural networks and datasets generated from few parameters

Anotácia: Implementačná práca v oblasti počítačového videnia a umelej inteligencie.

Cieľ: Cieľom práce je navrhnúť a realizovať niekoľko experimentov s konvolučnými neurónovými sieťami pri ktorých sa pracuje s jednoduchými vygenerovanými datasetmi. Neurónovou sieťou sa riešia tak jednoduché úlohy, že ich riešenie vieme navrhnúť pomocou filtrov a skúma sa, či neurónová sieť získa trénovaní analogické kernely. Pracuje sa s datasetmi objektov s malým počtom parametrov, takže možno skúmať, či ich v nejakej podobe nájdeme v príznakovom vektore natrénovaného klasifikátora. Natrénované modely analyzujeme pomocou receptívnych polí a ďalších metód navrhnutých v tejto práci.

Literatúra: Chollet, F.: Deep learning v jazyku Python, Grada, 2019
Learning OpenCV 3, Computer Vision in C++ with the OpenCV Library By Gary Bradski, Adrian Kaehler, O'Reilly Media, 2016
learnopencv.com
opencv.org
https://www.agentspace.org/andy/lucnyitat-ver2.pdf

Poznámka: Platforma: Python, knižnice Keras alebo Pytorch, OpenCV alebo PIL

Kľúčové slová: kernely, konvolučné neurónové siete, hlboké učenie, počítačové videnie

Vedúci: RNDr. Andrej Lúčny, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 08.10.2022

Dátum schválenia: 10.10.2022

doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Abstrakt

Táto práca sa zaoberá konvolučnými neurónovými sieťami použitými na spracovanie obrázkov. Porovnávame viacero typov modelov na rôznych problémoch počítačového videnia. Pomocou už vytvorených knižníc jednotlivé modely konvolučných neurónových sietí vytvárame a následne učíme pre potreby našich experimentov. Venujeme sa hlavne učeniu modelov konvolučných neurónových sietí, ktoré si sami vytvárame, a aj inicializujeme. Využívame viacero typov modelov, ako sú konvolučné neurónové siete, klasifikátory a autoenkodéry.

Kľúčové slová: neurónová sieť, konvolučná vrstva, autoenkóder

Abstract

This works studies convolutional neural networks used on images. We are comparing several types of models on different problems of computer vision. With the help of already-created libraries, we will build models of convolutional neural networks and train them for our experiments. Our work mainly studies models of convolutional neural networks, which we are creating and initializing. We are using several kinds of models, for example, convolutional neural networks, classification networks, and autoencoders.

Keywords: neural network, convolutional layer, autoencoder

Obsah

Zoznam obrázkov

Úvod

Táto bakalárska práca sa zaoberá oblasťou strojového učenia nazývaného neurónové siete, respektíve ešte špecifickejšie budeme sa venovať konvolučným neurónovým sieťam.

Táto oblasť sa v posledných rokoch dosť rozvíja a väčšina prác sa zaoberá veľkými modelmi. Vylepšovaním už existujúcich modelov, prípadne preučenie určitých modelov neurónových sietí na nové problémy, ale pomerne málo výskumu ide do základných stavebných prvkov týchto neurónových sietí.

Preto sme sa rozhodli skúmať menšie modely a porovnávať na nich, aký vplyv na ne majú jednotlivé parametre. Budeme prevádzať experimenty na viacerých typoch datasetov. Datasety budeme vytvárať pomocou algoritmov počítačového videnia a generátorov obrázkov s malým počtom parametrov.

Cieľom práce bude porovnanie modelov na týchto datasetoch a pozrieme sa aj na váhy niektorých jednoduchších modelov a porovnáme tieto váhy z hodnotami, ktoré by sme nastavili manuálne a robili by rovnakú úlohu.

Rozoberme si ešte stručne jednotlivé kapitoly. V prvej kapitole 1 si zavedieme pojmy, ktoré budeme ďalej v práci používať. Povieme si niečo o ich histórii a základných princípoch ich fungovania.

Ďalšiu kapitolu 2 sme venovali nástrojom, s ktorými budeme pracovať. Väčšina z nich sú knižnice. Pomôžu nám jednoducho pracovať s neurónovými sieťami, so spracovaním dát a vizualizáciou výsledkov.

V kapitole 3 robíme experimenty na datasetoch vygenerovaných s algoritmov počítačového videnia. Navrhneme modely blízko napodobujúc tieto algoritmy a následne ich budeme cvičiť.

Kapitola 4 pokračuje z ďalšími experimentami. Datasety si budeme generovať programami, ktoré nám z malého počtu parametrov vygenerujú obrázok. Menením týchto parametrov dostaneme rôzne obrázky, z ktorých sa bude skladať dataset. Na týchto datasetoch budeme skúšať rôzne typy konvolučných neurónových sietí.

Kapitola 1

Konvolučné neurónové siete

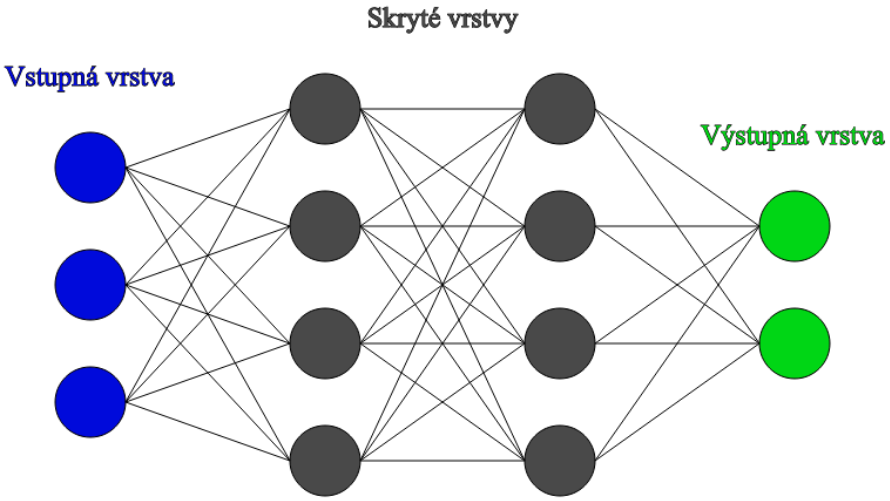
V tejto kapitole si zadefinujeme základné pojmy, ktoré budem v tejto práci využívať. Povieme niečo o histórii a základnom princípe fungovania konvolučných neurónových sietí, ktoré sa najčastejšie používajú na spracovanie obrázkových vstupov. Problémy plne prepojených neurónových sietí, ktoré viedli ku vymysleniu nového typu neurónových sietí. Jak tieto neurónové siete fungujú a čím sa autori inšpirovali pri vytváraní tohto výpočtového modelu.

1.1 Neurónové siete

Začnime tým, že si vysvetlíme čo sú to neurónové siete pred tým ako sa pustíme do vysvetľovania konvolučných neurónových sietí, ktorými sa bude primárne zaoberať táto práca. Neurónová sieť je výpočtový model vytvorený na základe neurónov v našom mozgu [7]. Používajú sa v oblasti umelej inteligencie a strojového učenia.

Základný typ neurónových sietí, ktorý sa používa nazývame takzvane plne prepojené neurónové siete a ako základnú jednotku využívajú výpočtový prvok s názvom perceptrón. Tento typ používa neurónové vrstvy. V každej vrstve sa nachádza vopred zadefinovaný počet perceptrónov. Rozdeľujeme ich na tri typy. Skryté vrstvy, ktoré majú prepojené perceptróny z každým perceptrónom z predchádzajúcej a nasledujúcej vrstvy. Tieto prepojenia si v sebe uchovávajú váhu medzi perceptrónmi. V neurónových sieťach sa nachádzajú ešte dve špeciálne vrstvy a to vstupná a výstupná. Tieto vrstvy sú spojené len s jednou ďalšou vrstvou. Vstupná vrstva slúži na zadanie dát, ktoré majú byť spracované neurónovou sieťou. Na druhej strane výstupná vrstva má úlohu výsledku neurónovej siete. Takýto výsledok často reprezentuje napríklad klasifikáciu, ohodnotenie vstupu alebo predpovedanie pokračovania daného vstupu.

Poznáme dva typy učenia neurónových sietí, ktoré označujeme z učiteľom a bez učiteľa. Neurónové siete, keď sú v procese učenia s učiteľom tak na začiatku sa inicializujú s nejakou heuristikou, na prázdno alebo náhodne. Následne dávame na vstup



Obr. 1.1: Vizualizácia plne prepojenej neurónovej siete s tromi vstupmi, dvomi skrytými vrstvami a dvomi výstupmi.

dáta z datasetu a porovnávame ich výsledok z výsledkami priradenými daním vstupom v datasete. Z tohto porovnania dostaneme chybu pomocou, ktorej následne zmeníme váhy v neurónovej sieti, aby v ďalšej epoche mali nižšiu chybu ako v predchádzajúcej. Medzi typy učenia bez učiteľa patria napríklad evolučné učenie [2] alebo Q-learning [9].

Ako sa neskôr dokázalo tak na všeobecnú aproximáciu perceptrónových neurónových sietí nám stačia len dve skryté vrstvy [4]. Na obrázku 1.1 môžme vidieť vizualizáciu malej plne prepojenej neurónovej siete s dvomi skrytími vrstvami.

1.2 História konvolučných neurónových sietí

Z dôvodu potreby spracovávania vizuálnych dát, nám tradičné plne prepojené neurónové siete nestačili. Na to aby sme dosiahli nejakú zmysluplnú presnosť by tieto neurónové siete museli byť obrovské a na ich natrénovanie by sme potrebovali priveľké množstvo dát na trénovanie, čo by aj výrazne predĺžilo čas učenia neurónových sietí. Tento problém by sa len evidentne zhoršoval s čím ďalej detailnejšími obrázkami. Kvôli tomuto faktu sa vedecká komunita snažila nájsť vyspelejšiu architektúru neurónových sietí, ktoré by sa vedeli jednoduchšie učiť na tomto type dát.

Prvá takáto práca, ktorá sa zaoberala zlepšením neurónových sietí na vizuálne rozpoznávanie vzorov bola práca pod názvom Neocognitron [1]. Neocognitron bola predchodcom moderných konvolučných neurónových sieti, ktoré poznáme dnes. Táto práca

navrhla výpočtový model rozpoznávajúc vizuálne vzory na základe ich geometrickej podobnosti, bez akéhokoľvek ovplyvnenia pozíciou daného vzoru na vstupe. Autori sa inšpirovali neurónovým systémom stavovcov, ktorý je schopný rozpoznať rôzne vzory. Navrhli dva typy vrstiev, ktoré sa striedajú a nazvali ich „S-bunky“ (jednoduché bunky alebo hyperkomplexné bunky nižšej úrovni) a „C-bunky“ (zložité bunky alebo hyperkomplexné bunky vyššej úrovni). „S-bunky“ boli niečo ako dnešné konvolučné vrstvy a „C-bunky“ tvorili vrstvu, ktorú dnes poznáme ako pooling-ová vrstva.

Názov konvolučná neurónová sieť vznikla až s dizajnom neurónovej siete LeNet-5 v práci [5]. V dizajne použili 5x5 konvolučné filtre a 2x2 pooling-ové vrstvy. Jednotlivé vrstvy fungovali presne takým istým spôsobom ako dnešné konvolučné a pooling-ové vrstvy. Vývoj tejto siete bol na rozpoznávanie ručne písaných číslic.

Nové architektúry konvolučných neurónových sietí môžme vďačiť ImageNet-u. Čo je súbor dát, ktorý sa použil vo výzve „ImageNet large scale visual recognition challenge (ILSVRC)“. ImageNet sa využíva na porovnanie rôznych typov algoritmov na klasifikácii obrázkov od vzniku v roku 2010.

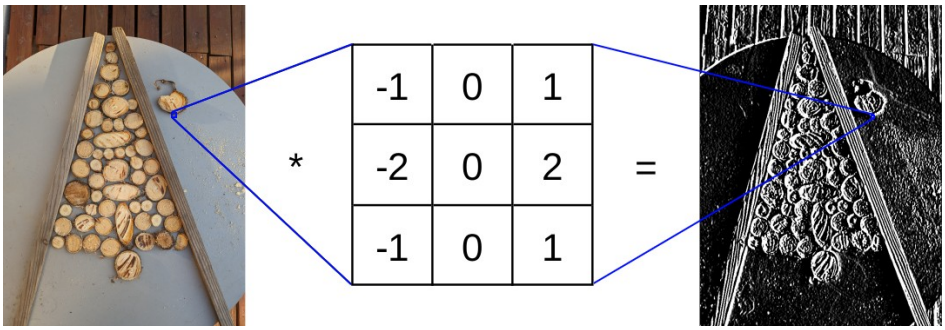
Prvýkrát sa podarilo konvolučnej neurónovej sieti poraziť ostatné algoritmy v roku 2012, kedy Alex Krizhevsky a jeho kolegovia navrhli architektúru konvolučnej neurónovej siete známu ako AlexNet [3] a znížili chybovosť doteraz najlepšieho algoritmu z 25,8% na 16,4%. Od roku 2012 každý rok vyhrávajú túto výzvu výhradne konvolučné neurónové siete.

V roku 2014 bola v tejto výzve predstavená ďalšia významná architektúra konvolučných neurónových sietí s pomenovaním VGGNet [8]. Tento názov dostala podľa svojich tvorcov na Oxfordskej univerzite s názvom „Visual Geometry Group“. Prišli s ideou, že hlbšie konvolučné neurónové siete by mohli byť lepšie na riešenie problémov a poraziť ostatné doteraz používané architektúry konvolučných neurónových sietí na súbore dát ImageNet.

1.3 Fungovanie konvolučných neurónových sietí

Na úvod tejto časti si zadefinujme aký typ neurónových sietí pod pojmom konvolučné neurónové siete myslíme. Pod pojmom konvolučná neurónová sieť myslíme architektúru, ktorá používa konvolučné vrstvy v svojej architektúre. Podobne pod pojmom plne konvolučná neurónová sieť myslíme architektúru využívajúcu iba konvolučné alebo pooling-ové vrstvy v svojej architektúre a tým pádom nepotrebuje mať konštantnú veľkosť vstupu ani výstupu.

Spomínali sme v predchádzajúcej podkapitole 1.2 konvolučné vrstvy. Poďme si teda vysvetliť, čo je táto vrstva a čím je výnimočná. Ako sme už písali v predchádzajúcej podkapitole 1.2, tak konvolučné vrstvy boli vytvorené na základe problému jednoduch-



Obr. 1.2: Ukážka Sobelovho algoritmu na detekciu vertikálnych hrán implementovaného pomocou konvolučného filtra na obrázku.

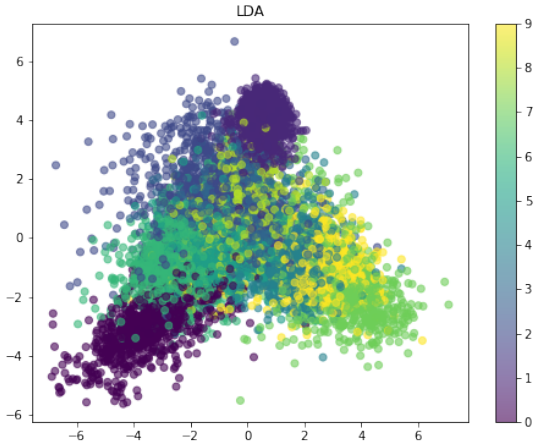
šie a univerzálnejšie detektovať vizuálne vzory na obrázku nezávislo od ich pozície na ňom. Z tohto problému sa nám vyvinulo riešenie nazývané konvolučná vrstva. Takáto vrstva sa skladá z vopred zadefinovaného počtu konvolučných filtrov s rovnakou šírkou a výškou. Tento počet konvolučných filtrov tiež nazývame hĺbka konvolučnej vrstvy.

Konvolučný filter je matica s preddefinovanou výškou a šírkou, ktorá sa aplikuje na všetky súvislé podmnožiny vstupu veľkosti filtra ako je to znázornené na obrázku 1.2. Čo znamená, že zo vstupu si zoberie časť, ktorá má rovnakú veľkosť ako filter a vynásobí túto časť so sebou. Ako výstup dá jednu hodnotu reprezentujúcu túto operáciu, ktorá ešte prejde aktivačnou funkciou. K tým známejším patria napríklad ReLU. Pre svoje kladné hodnoty sa správa ako lineárna funkcia a pre záporné hodnoty vracia nulu, obyčajná lineárna funkcia alebo inverzná ktorá vynásobí hodnotu -1.

Ďalšou podstatnou vrstvou v konvolučných neurónových sieťach je takzvaná pooling-ová vrstva. Táto vrstva zoberie zo vstupu dáta vo veľkosti výšky a šírky tejto vrstvy a na základe určitého pravidla ich zredukuje na jednu hodnotu. Na rozdiel od konvolučnej vrstvy, pooling-ová vrstva sa aplikuje na unikátne dáta, čo znamená, že pooling-ové filtre sa nemôžu navzájom prekrývať na vstupe. Najčastejšie používané pooling-ové filtre zredukujú dáta na základe maximálnej alebo priemernej hodnoty vo filtri. Používa sa na zjednodušenie jej vstupu a extrakciu najpodstatnejších vlastností pre ľahšie spracovanie v ďalších konvolučných vrstvách.

1.4 Autoenkodér

Povedzme si ešte niečo o špeciálnom type konvolučných neurónových sietí nazývanom autoenkodér. Autoenkodéry sa môžu zdať na prvý pohľad, že sú celkom zbytočné. Ich úlohou je na výstupe vyprodukovať to isté, čo mú dáme na vstup a pri tom využívajú enormné množstvo výpočtovej sily. Prečo by sme len nenačítali vstup na priamo, keď autoenkodér nám len pridáva chybu do vstupu. Opak je však pravdou, autoenkodéry sú v skutočnosti veľmi dôležité. To akým spôsobom autoenkodér funguje je veľmi



Obr. 1.3: Reprezentácia latentného priestoru na datasete MNIST pomocou latentnej Dirichletovej alokácie.

zaujímavé a má dosť praktické využitie, pri trénovaní konvolučných neurónových sietí.

Jedným zo zaujímavých vlastností autoenkodérov je ich vnútorné fungovanie. Autoenkodér sa skladá z dvoch základných stavebných prvkov. Nazývame ich kodér a dekodér. Tieto dva komponenty sú spojené takzvaným latentným priestorom alebo inak povedané vektorom vlastností.

Latentný priestor je generovaný kodér, ktorý ho následne posunie do dekodéra. Ten sa z neho pokúsi zrekonštruovať pôvodný vstup. Zaujímavou schopnosťou tohto vektora vlastností je, že zvyčajne býva podstatne menší ako vstup a obsahuje dôležité vlastnosti o vstupe. Ďalšia jeho pekná vlastnosť súvisí s podobnosťou vstupov nachádzajúcich sa blízko pri sebe v latentnom priestore a odlišné sú od seba ďaleko. Na obrázku 1.3 môžme vidieť vizualizáciu latentného priestoru a body s rovnakou farbou reprezentujú obrázky zobrazujúce rovnaké číslo.

Kodér a dekodér vieme po natrénovaní autonekodéra oddeliť a každý z nich vieme použiť na odlišný typ práce. Kodér sa dá použiť na natrénovanie klasifikátora spôsobom, že za vektor vlastností pridáme pár perceptrónových vrstiev. Následne budeme učiť, len túto časť neurónovej siete. Predpokladáme, že v predchádzajúcom kroku sme natrénovali dobrý autoenkodér a latentný priestor už obsahuje podstatné informácie o vstupe. Prípadne sa dá tiež použiť na detekciu objektov alebo segmentáciu vstupu. Na druhej strane dekodér vieme použiť na generovanie nových variantov vstupu.

Autoenkodéry sami o sebe sa dajú tiež použiť na určité typy úloh. Napríklad na stratovú kompresiu dát, ktorá pri niektorých typoch dát, kde už teraz používame algoritmy na stratovú kompresiu. Takýto autoenkodér môže ponúknuť zlepšenie v kompresii alebo zvýšenie kvality. Pri miernej modifikácii autoenkodéra je možné ho využiť pre zvýšenie rozlíšenia alebo odstránenie určitých artefaktov vytvorených kompresnými

algoritmami.

Kapitola 2

Nástroje použité na experimenty

V tejto kapitole si zadefinujeme nástroje, ktoré budeme používať v tejto práci. Budeme programovať v jazyku Python, pretože ho už veľmi dobre poznám a sú preňho vytvorené knižnice na spracovanie obrázkov a prácu s neurónovými sieťami. Na spracovanie obrázkov použijeme knižnicu OpenCV. Táto knižnica má implementované mnohé algoritmy počítačového videnia, ktoré budeme neskôr využívať. TensorFlow nám zasa poslúži na prácu s neurónovými sieťami.

2.1 Python

Python je programovací jazyk vytvorený Guido van Rossumom, ktorý na projekte pracoval ako hlavný vývojár od decembra 1989 až do 12 júla 2018, kedy odstúpil z tejto pozície. Následne Python Software Foundation - nadácia, ktorá sa do dnes stará o vývoj tohto jazyka. Táto nadácia vymenovala riadiacu radu, ktorá nahradila Guido van Rossumové miesto na projekte.

V čase písania tejto práce najnovšia verzia Pythonu je 3.11 a najstaršia podporovaná verzia je 3.7, ktorej končí podpora koncom tohto júna. Počas svojho vývinu Python prešiel tromi veľkými verziami. Prvá verzia bola vydaná v roku 1991 a neobsahovala kopec funkcii, ktoré môžme využívať v dnešnej verzii. Na druhej strane položil základy syntaxu a filozofiu Pythonu. Ako dobrú čitateľnosť kódu v anglickom jazyku s veľkým dôrazom na odsadenie kódu namiesto zátvoriek. Python 2.0 bol veľký skok s podporou množstva veľkých funkcii ako sú počítanie referencii, cyklus detekujúci zberač odpadu a podpora pre Unicode. Táto verzia bola vydaná v roku 2000. Python 3.0 priniesol zo sebou tiež kopec nových funkcii a je stále vyvíjaný. Niektoré z týchto funkcii boli spätne implementované do Python 2.0. Okrem nových funkcii Python 3.0 sa spravili zmeny v syntaxe, kde veľké množstvo predtým vstavaných výrazov sa pretransformovali na klasické funkcie.

Python používa dynamické určovanie typov a dynamické rozlišovanie názvov pre-

menných, čo znamená, že nemusíme explicitne priraďovať typy premenných. Kontrola typov spolu s názvami premenných a funkcii sa vykonáva za behu programu. Filozofiou Pythonu je vytváranie pekného čitatelného jazyka na úkor jeho celkovej rýchlosti. Programátorovi sa snaží dať jednoduché rozhranie, ktoré sa píše podobne ako vety v anglickom jazyku. Presne preto sa stal veľmi populárnym jazykom na rýchlu tvorbu prototypov a v kruhoch dátových vedcov. Táto skutočnosť viedla aj k vývoju viacerých knižníc zameraných na pracovanie s dátami a neúrónovými sieťami ako sú napríklad NumPy, TensorFlow alebo PyTorch. PyTorch sa používa na strojové učenie podobne ako TensorFlow, ale nebudeme s ním pracovať v tejto práci. Python je populárny aj medzi ostatnými programátormi a dá sa použiť na rôzne typy aplikácii, od konzolových nástrojov, cez grafické programy, až vývoj webových serverov. Jeho nedostatok v rýchlosti sa dá čiastočne získať späť pomocou písania častí kódu, ktorý potrebuje byť rýchly v iných jazykoch ako je napríklad C alebo využitia nástrojov ako je Cython. Cython slúži ako prekladač Python kódu, do jazyka C, pričom využíva priamo C volania do Python interpretera.

2.2 OpenCV

OpenCV je knižnica vyvinutá hlavne pre účely počítačového videnia v reálnom čase. Poskytuje veľké množstvo funkcii na spracovanie obrázkov, ktoré budeme využívať v tejto práci. OpenCV bolo vyvíjané vo svojom životnom cykle viacerými skupinami, na začiatku to vzniklo ako projekt v Intele, neskôr do nej začal prispievať Willow Garage, ktorý je tiež známy vývinom projektu s otvoreným zdrojom na prácu v robotike nazývaný Robot Operating System alebo v skratke ROS. V dnešnej dobe Intel pokračuje jeho vývinom OpenCV.

OpenCV podporuje veľké množstvo operačných systémov, hardweru a má schopnosť bežať na rôznych typoch akcelerátoroch ako sú napríklad grafické karty, pomocou modulov ako CUDA alebo OpenCL. Taktiež poskytuje rozhranie pre viaceré populárne jazyky, nás však bude zaujímať iba rozhranie pre jazyk Python, ktorý sme sa rozhodli používať v tejto práci.

My túto knižnicu použijeme na spracovávanie obrázkov, konkrétne na úpravu jasu a kontrastu v kapitole 3.1 a Sobelov detektor rohov v kapitole 3.2. Táto knižnica je omnoho schopnejšia a vie splniť omnoho komplexnejšie úlohy. Ako napríklad detekcia objektov, klasifikácia obrázkov a názorne si môžme ukázať na obrázku 2.1 detekciu tváre pomocou tejto knižnice.

Obr. 2.1: Ukážka detekcie tvárí implementovanej v knižnici OpenCV na obrázku.

2.3 NumPy

NumPy vznikol ako pokračovanie dvoch dovtedy používaných knižníc a to Numeric a Numarray. Každá z týchto knižníc mala svoje silné a slabé stránky. Vo všeobecnosti sa považovalo, že Numeric je lepší na prácu s menším počtom dát a naopak Numarray sa zas pýšil vyšším výkonom pri spracovávaní väčšieho množstva dát. NumPy sa považuje viac ako následník Numericu, ale vypožičal si aj funkcie a optimalizácie Numarray. Obe tieto knižnice sú už zastarané a nikto sa už nepodieľa na ich vývoji. Takže sa NumPy stal náhradou za obe s výhodami z oboch knižníc.

V dnešnej dobe mnohé knižnice v Pythone využívajú NumPy alebo implementujú NumPy rozhranie optimalizované pre beh na špecializovanom hardware ako sú grafické karty alebo iné viac špecifické akcelerátory. Slúži na efektívnu prácu s viacrozmernými poliami a maticami s konštantnou veľkosťou. Je schopná aj upravovať veľkosť polí, ale nepridáva pri tom veľké zlepšenia oproti vstavanej podpore polí v Pythone. Implementuje nad nimi mnohé štandardné algebraické operácie a ďalšie užitočné funkcie.

2.4 TensorFlow

TensorFlow slúži na číselné výpočty využívajúc grafovú štruktúru na dosiahnutie výsledku. Je vyvíjaný spoločnosťou Google. Zo začiatku na vnútorné použitie pre vlastné experimenty. Neskôr ju však zverejnili ako knižnicu s otvoreným zdrojovým kódom,

Kód 2.1: Ukážka použitia knižnice NumPy.

```
import numpy as np

# vytvorime 2D-pole velkosti 10x3 z hodnotami od 0 do 29
x = np.arange(30, dtype=np.int64).reshape(3,10)
# vynasobime v kazdom tretom stlpci
# hodnotu -2-krat
x[:, 2::3] *= -2
# teraz zistime najvacsiu hodnotu v kazdom stlpci
x.max(axis=0)
```

aby ju mohli používať výskumníci všade po svete pri svojich pokusoch. V dnešnej dobe je využívaná masami ľudí v podobe produktov od Googlu a iných spoločností.

Modely v TensorFlowe sú popísané grafmi. Vrcholy grafu zväčša reprezentujú operácie matematických výpočtov, ale môžu obsadzovať aj iné úlohy ako načítanie vstupu alebo vrátenie výstupu.

Tensorflow bol a je vyvíjaný ako knižnica na vytváranie neurónových sieti, ale nie je na to obmedzený a môžme pomocou neho namodelovať systém na riešenie diferenciálnych rovníc.

V TensorFlowe sa základná jednotka volá tensor, je to štruktúra podobná poliam z knižnice NumPy, ktorú sme si spomínali v kapitole 2.3. V tensore sa nachádzajú hodnoty, ktoré budeme spracovávať rovnakou operáciou, teda môžeme povedať, že tensor je vrchol grafu. Spojitosť s ostatnými tensormi nám zabezpečujú operácie a aktivačné funkcie, tieto spojenia predstavujú orientované hrany grafu.

TensorFlow ponúka programátorom dva typy rozhraní. Sekvenčné - jednoduchšie na používanie, ale zároveň aj viac obmedzené pri vytváraní komplexnejších modelov. Druhé rozhranie je funkčné a umožňuje prepájať jednotlivé tensori s viac než jednou operáciou a aktivačnou funkciou, pričom vytvárame modely viac podobné grafom. Okrem týchto rozhraní implementuje rozhranie knižnice Keras, ktorú spomenieme v kapitole 2.5. Toto rozhranie sa používa na jednoduchšie vytváranie modelov v TensorFlowe a dá sa využiť ako aj pri sekvenčnom tak aj pri funkčnom rozhraní TensorFlowu.

2.5 Keras

Knižnica Keras bola pôvodne vyvinutá na prácu s knižnicou Theano, ale neskôr bola pridaná podpora aj pre knižnicu TensorFlow, o ktorej sme sa bavili v kapitole 2.4. V budúcnosti môže byť pridaná podpora aj pre nové lepšie knižnice, ktoré budú implementovať operácie na maticiach potrebné pre túto knižnicu.

Knižnica Keras slúži na tvorbu neurónových sietí. Obsahuje viaceré často používané typy vrstiev a aktivačné funkcie, ktoré sa využívajú na tvorbu neurónových sietí. Keras bol dizajnovaný modulárne, aby sa do neho dalo ľahko pridávať nové vrstvy, operácie a prvky. Tieto vlastnosti umožňujú jednoduché a rýchle prototypovanie rôznych typov neurónových sietí. Keras podporuje dva typy rozhraní na vytváranie modelov ako sú sekvenčné a grafové rozhrania. Tieto rozhrania sú dosť analogické k tým, ktoré nám poskytuje TensorFlow. Z toho aj vyplýva podpora týchto rozhraní v TensorFlowe, ktoré pochádzajú z knižnice Keras.

Keras umožňuje tvorbu klasických plne prepojených neurónových sietí, konvolučných neurónových sietí a rekurzívnych neurónových sietí. Na to nám slúžia implementované typy vrstiev ako napríklad plne prepojené, konvolučné, pooling-ové, aktivačné vrstvy a vrstvy upravujúce rozmery na sploštenie viac rozmerných polí na jedno rozmerné. Keras nám tiež poskytuje rôzne nastavenia týchto vrstiev ako napríklad ich veľkosť a spôsob inicializovania ich váh.

2.6 Matplotlib

Poslednú knižnicu, ktorú si spomenieme je Matplotlib. Je to knižnica na vizualizáciu dát pomocou rôznych typov grafov pre programovací jazyk Python. Autorom tejto knižnice bol John Hunter a prvú verziu zverejnil v roku 2003. Po jeho smrti bol v 2012 nominovaný Michael Droettboom na hlavného vývojára. Dodnes je táto knižnica vyvíjaná komunitou a podporovaná neziskovou organizáciou NumFocus.

Matplotlib má podporu pre mnohé populárne Python knižnice, ako napríklad knižnica NumPy, ktorú sme spomínaly v kapitole 2.3. Zároveň tiež umožňuje ľahkú integráciu do rôznych grafických prostredí kde vie interaktívne vizualizovať dáta alebo len základné uloženie grafov do obrázku.

V našej práci využijeme túto knižnicu na jednoduchú vizualizáciu dát z učenia konvolučných neurónových sieti.

Kód 2.2: Použitie sekvenčného rozhrania TensorFlowu a stavebných prvkov Kerasu na natrénovanie neurónovej siete na datasete MNIST.

```
import tensorflow as tf
# nacitanie datasetu MNIST
mnist = tf.keras.datasets.mnist

# rozdelenie datasetu na data na trenovanie a data
# na validaciu
(in_train, out_train),(in_test, out_test) = mnist.load_data()
# normalizacia dat aby hodnoty boli medzi 0 a 1
in_train, in_test = in_train / 255.0, in_test / 255.0

# zadefinovanie modelu pomocou sekvencneho rozhrania
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax'),
])

# nastavenie optimalizatora a chybovej funkcie na ucenie
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'],
)

# natrenovanie a validacia modelu
model.fit(in_train, out_train, epochs=5)
model.evaluate(in_test, out_test)
```

Kapitola 3

Algoritmy počítačového videnia

Táto kapitola sa bude zaoberať učením konvolučných neurónových sietí na algoritmoch počítačového videnia. Zistíme, či vieme konvolučnú neurónovú sieť naučiť tieto algoritmy. Či na to budeme potrebovať väčšiu konvolučnú sieť ako bol pôvodný výpočtový model. Budeme v konvolučných vrstvách vidieť filtre použité v originálnom algoritme. Skúsime aj zmenšiť konvolučnú neurónovú sieť a uvidíme, či ju budeme schopný naučiť dostatočne dobre aproximovať pôvodný algoritmus.

3.1 Jas a kontrast

Zmenu jasu a kontrastu na obrázku je pomerne jednoduchý problém spracovania obrázkov. Čo z neho robí dobrý základ na učenie práce s knižnicami z kapitoly 2, ktoré budeme používať pre potreby tejto práce.

V tomto experimente si odfotografujeme dva obrázky. Mali mať dostatočnú variáciu v pixloch. Ale nemusíme sa tým veľmi trápiť stačí, že budeme mať na fotkách dostatočné zastúpenie základných farieb ako je červená, modrá a zelená. Prípadne si na trénovanie môžme vygenerovať vlastný obrázok pomocou gradientov medzi červenou, zelenou a modrou farbou. Takýto obrázok bude ideálny na učenie tohto problému. Jeden z nich použijeme na trénovanie a druhý na validáciu. Následne si zvolíme jas a kontrast. Následne ich budeme chcieť aplikovať na obrázky a pomocou knižnice OpenCV tieto hodnoty na ne aplikujeme. Upravené obrázky použijeme ako výstup, ktorý by sme chceli dostať z konvolučnej neurónovej siete. Môžme zmeny jasu a kontrastu vidieť na obrázku 3.1 na ktorom sme použili +15 jasu a 160% kontrast.

Na natrénovanie tohto algoritmu použijeme architektúru z tabuľky 3.1 - využíva tri konvolučné filtre veľkosti jedna krát jedna. Obsahuje dvanásť parametrov na trénovanie a to jeden bias a tri parametre pre jednotlivé farebné zložky obrázku pre každý konvolučný filter. To sú štyri parametre na konvolučný filter a máme tri konvolučné filtre lebo chceme na výsledku dostať farebný obrázok s tromi zložkami červená, zelená

Obr. 3.1: Pôvodný obrázok a obrázok, na ktorý sme aplikovali jas a kontrast.

Tabuľka 3.1: Sumarizácia architektúry neurónovej siete použitej na aplikovanie jasu a kontrastu.

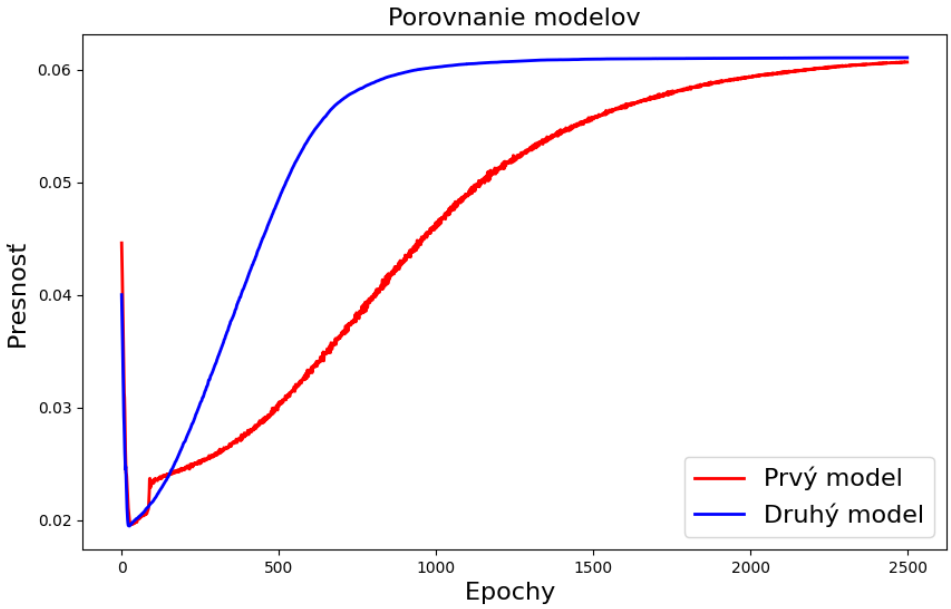
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, None, None, 3)]	0
conv2d (Conv2D)	(None, None, None, 3)	12
Total params: 12		
Trainable params: 12		
Non-trainable params: 0		

a modrá. Čo nám dáva dokopy dvanásť parametrov.

Na zostrojenie tohto modelu konvolučnej neurónovej siete použijeme sekvenčné rozhranie knižnice TensorFlow. Je to ideálne na začiatok, pretože je jednoduchšie na pochopenie základných princípov konvolučných neurónových sietí a náš model nie je na toľko zložitý, že by nás toto rozhranie v niečom obmedzovalo.

Na obrázku 3.1 si ukážeme ako na zlom vstupe sa nám naša konvolučná neurónová sieť môže preučiť na špecifickom vstupe. Môžeme si všimnúť na tomto obrázku chýbajú výraznejšie modré a zelené farby.

Keď potom takúto konvolučnú neurónovú sieť otestujeme na obrázku, ktorý má dobré zastúpenie všetkých farieb tak si všimneme, že výsledok sa celkom líši od toho čo sme chceli dosiahnuť. Následne ak sa pozrieme na váhy konvolučnej vrstvy tak na diagonále by sme očakávali kontrast a v biase jas, ktoré sme použili pri úprave obrázku. Jas sa nám podarí nacvičiť aj na tomto neoptimálnom obrázku. Na druhej strane kontrast nevyzeral vôbec ako by sme si ho predstavovali. Presnosť natrénovaných konvolučných neurónových sietí na tomto obrázku je pri testovaní na iných obrázkoch optimálnejších na trénovanie okolo 60% až 70%.



Obr. 3.2: Graf presnosti jednotlivých modelov počas trénovania Sobelovho detektora hrán.

Pri použití dobrého obrázku na trénovanie sa nám podarí jednoducho naučiť nás model na aplikovanie špecifického kontrastu a jasu. Takto natrénovaná konvolučná neurónová sieť je potom dobrá na rôznych typoch obrázkov. Aj na takých čo na trénovanie neboli úplne vyhovujúce. Zároveň tiež môžme vidieť vo váhach konvolučnej neurónovej siete, že sú veľmi podobné tomu čo by sme očakávali, keby sme nastavili tieto váhy manuálne.

3.2 Sobelov detektor hrán

V nasledujúcom celku budeme učiť konvolučnú neurónovú sieť Sobelov detektor hrán. Začneme tým, že si znova pripravíme obrázky na trénovanie, na ktorých zdetegujeme hrany pomocou Sobelovho algoritmu implementovaného v knižnici OpenCV. Môžme kľudne aj použiť obrázky z minulého experimentu, len si treba dávať pozor aby ten obrázok mal dostatok hrán. Čo znamená, že by sme sa vyhli obrázku gradienta vygenerovaného v predchádzajúcej časti.

Z jeho definície by nám mohlo napadnúť použiť dva konvolučné filtre o veľkosti tri krát tri a s absolútnou funkciou ako aktivačnou. Bohužiaľ nám TensorFlow neponúka absolútnu funkciu ako jednu z aktivačných, ale vieme ju zostrojiť z už existujúcich funkcii v TensorFlowe. Máme dve možnosti jak ju vieme implementovať do modelu. Na obidve použijeme funkcionálne rozhranie TensorFlowu, s ktorým sa naučíme pracovať pre flexibilnejšiu manipuláciu z neurónovými sieťami.

Tabuľka 3.2: Sumarizácia neurónovej siete použitej na detekciu hrán pomocou prvej metódy.

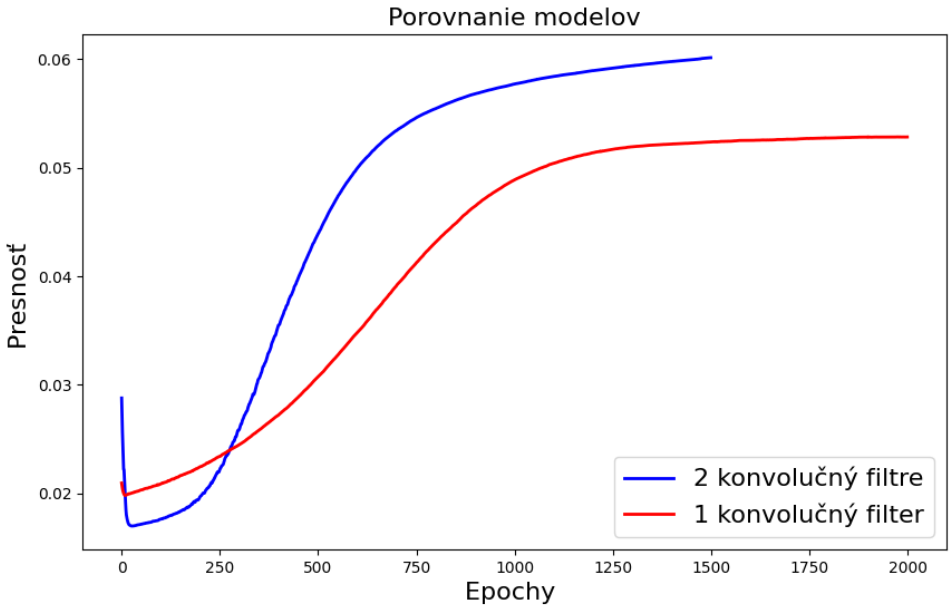
Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, None, None, 0, 1)]	0	[]
conv2d (Conv2D)	(None, None, None, 10, 1)	10	['input_1[0][0]']
conv2d_1 (Conv2D)	(None, None, None, 10, 1)	10	['input_1[0][0]']
conv2d_2 (Conv2D)	(None, None, None, 10, 1)	10	['input_1[0][0]']
conv2d_3 (Conv2D)	(None, None, None, 10, 1)	10	['input_1[0][0]']
add (Add)	(None, None, None, 0, 1)	0	['conv2d[0][0]', 'conv2d_1[0][0]', 'conv2d_2[0][0]', 'conv2d_3[0][0]']
Total params: 40			
Trainable params: 40			
Non-trainable params: 0			

Pri prvej by sme použili namiesto dvoch konvolučných filtrov štyri s aktivačnou funkciou ReLU. Základný princíp tejto metódy by bol mať dva horizontálne a dva vertikálne filtre, pričom každý filter by detegoval hranu z iného smeru (z hora, z prava, z dola, z ľava) a vertikálne alebo horizontálne filtre by boli invertom toho druhého.

V druhej možnosti by sme zanechali dva konvolučné filtre, ale na ich výstup by sme pripojili viacero aktivačných funkcii. Po sčítaní týchto funkcií by nám dávali absolútnu hodnotu výsledku jednotlivých filtrov. Použijeme na to dve ReLU aktivačné funkcie a jednu inverznú funkciu.

Poďme teda porovnať tieto metódy medzi sebou. Už len zo základnej definície môžeme vidieť, že prvá metóda má dvakrát toľko parametrov na trénovanie ako druhá a toto si vieme aj potvrdiť, keď si implementuje oba modely v TensorFlowe a dáme si vypísať ich sumarizáciu viď tabuľky 3.2 a 3.3.

Na obrázku 3.2 vidíme graf učenia oboch metód. Prvá vec, ktorá nás dosť prekvapila je veľmi nízka presnosť modelov. V oboch prípadoch sme neboli schopný nacvičiť konvolučnú neurónovú sieť s presnosťou vyššou ako 6%. Keď sme si dali vizualizovať výstup neurónovej siete na obrázkoch z trénovacieho setu alebo aj z obrázkov, ktoré náš model predtým nevidel. Výsledok bol veľmi podobný tomu čo sme dostali zo Sobelovho detektora hrán implementovaného knižnicou OpenCV - používali sme ju na vygenerovanie výsledkov v datasete. Túto malú presnosť počítanú TensorFlowom sa nám nepodarilo žiadnym spôsobom zvýšiť. Tým pádom budeme ďalej predpokladať, že presnosť,



Obr. 3.3: Graf presnosti jednotlivých modelov počas trénovania Sobelovho detektora hrán.

ktorú sme dosiahli bola nejakou chybou pri výpočte alebo zlým nastavením siete aj keď toto správanie sme nespozorovali pri iných experimentoch. Ďalšou zaujímavou vecou je počet potrebných epóch na učenie Sobelovho detektora hrán. Používame konvolučné neurónové siete s malým počtom parametrov na cvičenie. V ostatných experimentoch nám stačilo násobne menej epóch na naučenie aj násobne väčších modelov.

Vráťme sa späť k porovnaniu našich dvoch modelov. Z grafu na obrázku 3.2 vidíme rýchlejšie učenie nášho menšieho modelu zvýrazneného modrou farbou. Model bol už skoro naučený po 1000 epochách a po 1500. epoche sme my už nevideli žiadne ďalšie učenie modelu. Väčší model sa učil o dosť pomalšie a podobnú presnosť dosiahol až pri epoche 2500 a ďalšie trénovanie modelu nám už presnosť ďalej nezvyšovalo. Ak by sme chceli porovnať ich presnosť po natrénovaní tak obidva modely nám dávali výsledky s veľmi podobnou presnosťou. Čiže na základe presnosti môžme použiť hociktorý model a dosiahneme rovnaký výsledok. Tak sa pozrime na parametre a rozhodnime, ktorý model je lepší.

Ďalší parameter na porovnanie je ako dlho trvá jedna epocha pri učení, a ktorý model je výpočtovo náročnejší po natrénovaní. Prvý väčší model bohužiaľ prehráva aj v tejto kategórii, okrem väčšieho počtu epóch na natrénovanie modelu. Každá epocha trvala o 25% dlhšie na vykonanie a po natrénovaní spustenie prvého modelu na obrázku trvalo v priemere o 60% dlhšie v porovnaní s druhým modelom.

Na základe týchto faktov by sme povedali, že druhý menší model je vo všetkom aspoň tak dobrý ako prvý väčší model. Čo z neho robí jednoznačného víťaza. Keď

Tabuľka 3.3: Sumarizácia neurónovej siete použitej na detekciu hrán pomocou druhej metódy.

Layer (type)	Output Shape	Param #	Connected to
input_1	[(None, None,	0	[]
(InputLayer)	None, 0, 1)]		
conv2d	(None, None,	10	['input_1[0][0]']
(Conv2D)	None, 10, 1)		
conv2d_1	(None, None,	10	['input_1[0][0]']
(Conv2D)	None, 10, 1)		
tf.nn.relu	(None, None,	0	['conv2d[0][0]']
(TFOpLambda)	None, 0, 1)		
tf.nn.relu_1	(None, None,	0	['conv2d_1[0][0]']
(TFOpLambda)	None, 0, 1)		
tf.math.negative	(None, None,	0	['conv2d[0][0]']
(TFOpLambda)	None, 0, 1)		
tf.math.negative_1	(None, None,	0	['conv2d_1[0][0]']
(TFOpLambda)	None, 0, 1)		
add (Add)	(None, None,	0	['tf.nn.relu[0][0]',
	None, 0, 1)		'tf.nn.relu[0][0]',
			'tf.nn.relu_1[0][0]',
			'tf.nn.relu_1[0][0]',
			'tf.math.negative[0][0]',
			'tf.math.negative_1[0][0]']
Total params: 20			
Trainable params: 20			
Non-trainable params: 0			

Tabuľka 3.4: Sumarizácia neurónovej siete použitej na detekciu hrán pomocou len jedného konvolučného filtra.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, None, None, 0, 1)]	0	[]
conv2d (Conv2D)	(None, None, None, 10, 1)	10	['input_1[0][0]']
tf.nn.relu (TFOpLambda)	(None, None, None, 0, 1)	0	['conv2d[0][0]']
tf.math.negative (TFOpLambda)	(None, None, None, 0, 1)	0	['conv2d[0][0]']
add (Add)	(None, None, None, 0, 1)	0	['tf.nn.relu[0][0]', 'tf.nn.relu[0][0]', 'tf.math.negative[0][0]']
Total params: 10			
Trainable params: 10			
Non-trainable params: 0			

sme zistili, že vytváranie väčšieho modelu na tento problém nie je potrebné, čo keby sme skúsili použiť koncept s druhého modelu a odstrániť z neho jeden konvolučný filter. Bude takáto konvolučná neurónová sieť schopná nám dať porovnateľnú presnosť, poďme to zistiť.

Graf na obrázku 3.3 nám ukazuje nižšiu presnosť o 14% pri použití jedného konvolučného filtra oproti dvom konvolučným filtrom, ktoré sa používajú aj v pôvodnom Sobelovom detektore hrán. Keď si zobrazíme výstup obidvoch modelov do obrázku, tak si všimneme, že model iba z jedným filtrom deteguje niektoré hrany s menšou intenzitou a nie je tak dobre vidieť typické Sobelovksé štvorcové hrany v prípade, keď jeden pixel je odlišný od všetkých jeho susedov.

Na druhej strane je model iba s jedným konvolučným filtrom o 30% rýchlejší na spustenie po natrénovaní. Na základe toho by trebalo zvážiť na čom tento model bude bežať a podľa toho vybrať jeden z nich. Ak chceme čo najväčšiu presnosť je vhodné použiť dva konvolučné filtre.

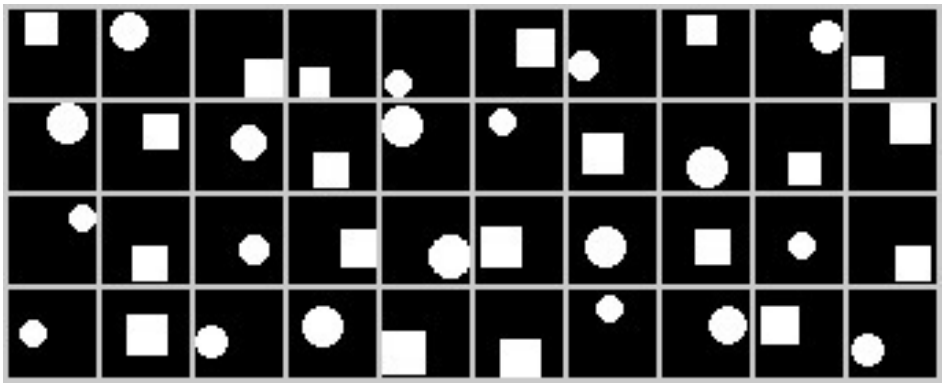
Kapitola 4

Dataset z malého počtu parametrov

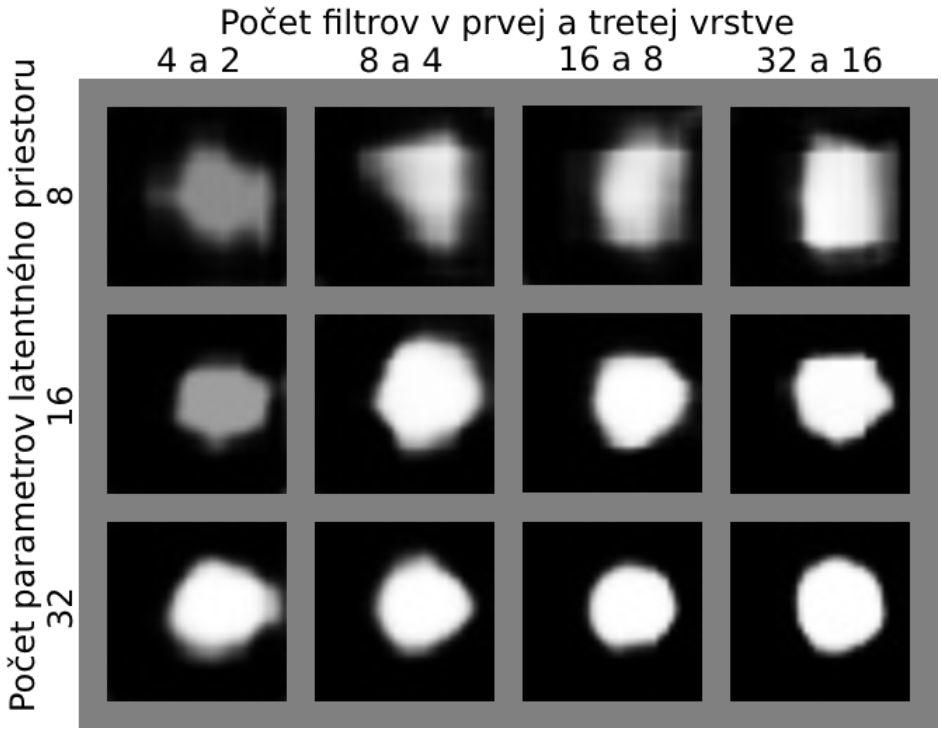
V poslednej kapitole budeme generovať rôzne typy datasetov z malého počtu parametrov a následne na nich budeme učiť konvolučné neurónové siete rozpoznávať parametre, z ktorých sme dataset vygenerovali. Budeme robiť pokusy na rôznych generátoroch a budeme učiť konvolučné neurónové siete rozpoznať rôzny počet parametrov a tiež učiť autoenkodéry z malým latentným priestorom a pozerať či sme schopný vidieť nejaké parametre v ňom.

4.1 Generovanie jednoduchých tvarov

Naprogramovali sme si generátor datasetu, ktorý generuje štvorce a kruhy na rôznych pozíciach na obrázku, rôznou veľkosťou a farbou. Tieto všetky parametre a ďalšie ako veľkosť obrázkov alebo veľkosť datasetu vieme nastaviť generátoru na robenie viacerých typov experimentov.



Obr. 4.1: Ukážka datasetu vygenerovaného pomocou generátora jednoduchých tvarov s rôznou veľkosťou a pozíciou.



Obr. 4.2: Porovnanie výstupov jednotlivých autoenkodérov na rovnakom vstupe.

4.1.1 Porovnanie autoenkodérov

Na náš prvý experiment použijeme čierno biele obrázky s veľkosťou 32x32 pixlov a budeme tvary generovať na rôznych pozíciach a s variáciou vo veľkosti. Malú ukážku typu generovaných obrázkov môžme vidieť na obrázku 4.1.

V tomto pokuse sa zameriame na presnosť autoenkodéra v závislosti od počtu konvolučných filtrov a veľkosti latentného priestoru. Nastavíme 200 epoch na učenie modelu. Rozdelíme si autoenkodér na kodér a dekodér. Budú si dosť podobné a počet filtrov sa bude navzájom zrkadliť. Plánujeme používať konvolučné filtre iba s veľkosťou tri krát tri. Prvá vrstva je konvolučná s dvakrát viac konvolučnými filtrami ako tretia vrstva, ktorá bude tiež konvolučná. Pomocou týchto dvoch vrstiev následne kontrolujeme veľkosť konvolučnej neurónovej siete. Druhá a štvrtá vrstva je max pooling-ová s veľkosťou dva krát dva. Piata vrstva je konvolučná a pomocou nej budeme kontrolovať veľkosť latentného priestoru. Nakoniec v kodéry budeme mať max pooling-ovú vrstvu s veľkosťou štyri krát štyri na výrazne obmedzenie veľkosti latentného priestoru. Dekodér má analogické vrstvy, ktoré nám obrázok obnovia a zrkadlia vrstvy kodéra. Na koniec pridáme ešte jednu konvolučnú vrstvu s jedným konvolučným filtrom. Táto vrstva nám poslúži ako čierno biely výstup. Ešte nastavíme všetkým vrstvám výplň (*padding*) na rovnakú. Toto robíme z dôvodu výrazne lepšej presnosti autoenkodéra oproti validnej výplni, čo sme si všimli pri robení experimentov. V tabuľke 4.1 si ukážeme ako vyzerá najmenší model, ktorý v tomto pokuse použijeme.

Tabuľka 4.1: Sumarizácia najmenšieho autoenkodéra použitého na datasete vygenerovanom generátorom jednoduchých tvarov.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 4)	40
max_pooling2d (MaxPooling2D)	(None, 16, 16, 4)	0
conv2d_1 (Conv2D)	(None, 16, 16, 2)	74
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 2)	0
conv2d_2 (Conv2D)	(None, 8, 8, 2)	38
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 2)	0
up_sampling2d (UpSampling2D)	(None, 8, 8, 2)	0
conv2d_transpose (Conv2DTranspose)	(None, 8, 8, 2)	38
up_sampling2d_1 (UpSampling2D)	(None, 16, 16, 2)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 16, 16, 2)	38
up_sampling2d_2 (UpSampling2D)	(None, 32, 32, 2)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 32, 32, 4)	76
conv2d_transpose_3 (Conv2DTranspose)	(None, 32, 32, 1)	37
Total params: 341		
Trainable params: 341		
Non-trainable params: 0		

Tabuľka 4.2: Porovnanie presnosti neurónových sietí, podľa počtu konvolučných filtrov a veľkosti latentného priestoru na datasete vygenerovanom pomocou generátora jednoduchých tvarov.

	4 a 2	8 a 4	16 a 8	32 a 16
8	92,97%	92,63%	93,66%	94,38%
16	95,48%	95,90%	96,68%	97,35%
32	96,53%	97,08%	97,93%	98,58%

Na porovnanie použijeme konvolučné neurónové siete s štyrmi, ôsmimi, šestnástimi a tridsiatimi dvoma konvolučnými filtrami v prvej vrstve nášho modelu, respektíve dva, štyri, osem a šestnásť konvolučných filtrov v tretej vrstve. Modely rozdelíme ešte podľa počtu parametrov v latentnom priestore na osem, šestnásť a tridsaťdva.

V tabuľke 4.2 vidíme porovnanie presnosti jednotlivých konvolučných neurónových sietí. Je vidno, že zo zväčšujúcim modelom aj z väčším počtom parametrov v latentnom priestore sa nám presnosť zvyšovala. Čisto podľa presnosti všetky modely vyzerajú byť celkom dobré.

Toto však nie je celý obraz a ak sa pozrieme na výstupy autoenkodérov všimneme si dosť veľké rozdiely. Na obrázku 4.2 sme porovnali výstup jednotlivých konvolučných neurónových sietí na tom istom vstupe, ktorý nebol súčasťou učenia. Je jednoznačne vidieť, že kvalita sa výrazne zvyšuje vo väčších modeloch. Pričom malé konvolučné neurónové siete neboli schopné ani správne obnoviť bielu farbu tvarov. Na druhej strane veľké modely boli schopné celkom dobre zrekonštruovať aj tvar na vstupe. Vlastnosti ako pozícia a veľkosť jednotlivých tvarov boli všetky autoenkodéry schopné sa naučiť.

4.1.2 Klasifikácia tvarov

Keďže sme si všimli v minulom experimente, že autoenkodéry mali problémy z rekonštrukciou jednotlivých tvarov, tak nám prišlo zaujímavé rozpoznávať presne tento parameter z generovaných obrázkov. Takže v tomto experimente budeme rozpoznávať jednotlivé tvary na obrázkoch generovaných generátorom jednoduchých tvarov.

Bolo celkom náročné nájsť architektúru modelu vhodnú na tento problém. Pri príliš malých konvolučných neurónových sieťach sme mali problém dostať vyššiu ako 50% presnosť, čo vôbec nie je dobré. Podobnú pravdepodobnosť nám dá aj generátor náhodných výsledkov. Na druhej strane, keď sme model príliš zväčšili sa príliš ľahko preučil na trénovacom datasete, že nebol schopný skoro vôbec generalizovať na vstupy mimo trénovacieho datasetu.

Nakoniec sme zvolili model sumarizovaný v tabuľke 4.3, v ktorom sme použili rovnakú výplň ako v predchádzajúcom experimente a tiež nám to pomohlo pri trénovaní modelu. Na učenie sme použili 500 epoch. Ani tento model nie je úplne ideálny a dostávame aj preučené modely. Zároveň aj modely, ktoré sú tak dobré ako hod mincou. Aký model dostaneme záleží od inicializácie váh na začiatku. Tento model je dostatočne malý na to aby sme nacvičili viacero modelov a vybrali si z nich jeden, ktorý bude mať pre nás dostatočne dobrú presnosť.

Keďže sme mali problémy z cvičením modelu v tomto pokuse. Tak sme sa rozhodli preskúmať rôzne typy inicializácii váh v jednotlivých vrstvách. K jedným z veľmi často používaných inicializátorov patrí Xavierov inicializátor, ktorý mnohý považujú za lepšiu ako čisto náhodnú inicializáciu. Tiež je často používaná pri cvičení úplne nových

Tabuľka 4.3: Sumarizácia neurónovej siete použitej na klasifikáciu tvarov generovaných generátorom jednoduchých tvarov.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 4)	40
max_pooling2d (MaxPooling2D)	(None, 16, 16, 4)	0
conv2d_1 (Conv2D)	(None, 16, 16, 2)	74
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 2)	0
conv2d_2 (Conv2D)	(None, 8, 8, 1)	19
max_pooling2d_2 (MaxPooling2D)	(None, 2, 2, 1)	0
flatten (Flatten)	(None, 4)	0
dense (Dense)	(None, 16)	80
dense_1 (Dense)	(None, 2)	34
Total params: 247		
Trainable params: 247		
Non-trainable params: 0		

modelov.

Z nášho testovania sme nenašli jednoznačného víťaza medzi nimi. Keď sme skúšali rozpoznávať tvary, ktoré mali rovnakú veľkosť tak sme mali vyššiu úspešnosť natrénovať vo všeobecnosti lepší model nám dávala náhodná inicializácia na našej konvolučnej neurónovej sieti. Keď sme pridali variáciu veľkosti jednotlivých tvarov tak naopak Xavierová inicializácia nám dávala lepšie výsledky.

Na základe nášho limitovaného pozorovania máme hypotézu, že Xavierová inicializácia je lepšia na cvičenie zložitejších konceptov a pri jednoduchých konceptoch nevydáva tak dobré výsledky. Túto našu hypotézu by však bolo treba lepšie preskúmať v nejakej ďalšej práci. Aby sme vedeli povedať nejaké závery. Toto správanie tiež mohol byť len čistá zhoda náhod, na ktorú sme narazili.

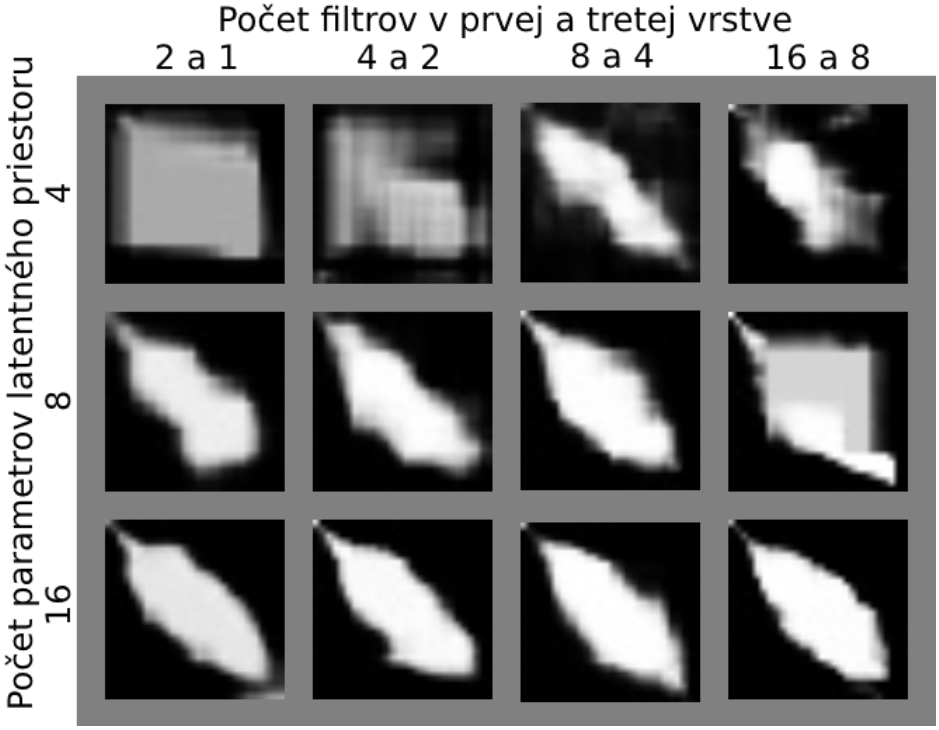
4.2 Lindenmayerov systém

Posledný experiment v tejto práci bude trénovanie autoenkodérov z rôznou veľkosťou ako v predchádzajúcej časti, keď sme trénovali autoenkodéry na datasete z jednoduchých tvarov. Až na to, že na tento experiment si požičiame generátor ružových lístkov z práce [6] Andreja Lúčneho.

Použijeme na to veľmi podobnú architektúru ako v predchádzajúcom autoenkodéry, len ho upravíme aby fungoval s obrázkami veľkosti 28x28 pixlov a budeme učiť celkovo menšie konvoloučné neurónové siete.

Tabuľka 4.4: Porovnanie presnosti neurónových sietí, podľa počtu konvolučných filtrov a veľkosti latentného priestoru na datasete ružových lístkov.

	2 a 1	4 a 2	8 a 4	16 a 8
4	93,02%	92,72%	94,81%	95,33%
8	96,31%	96,03%	97,53%	96,97%
16	97,14%	97,99%	98,02%	98,57%



Obr. 4.3: Porovnanie výstupov jednotlivých autoenkodérov na obrázku lístka.

Z obrázku 4.3 vidíme, že na tomto datasete sme videli väčšie zlepšenia v kvalite výstupu zväčšovaním latentného priestoru. Pridávanie filtrov malo omnoho menší dopad na kvalitu výstupov ako v predchádzajúcom experimente.

Záver

Cieľom našej bakalárskej práce bolo preštudovanie problematiky neurónových sieti. Pre nás bola najviac podstatná oblasť konvolučných neurónových sietí. Naučili sme sa tiež pracovať s nástrojmi užitočnými pri vytváraní, učení a následnom vyhodnocovaní neurónových sieti. O týchto veciach sme sa vo väčších detailoch písali v kapitolách 1 a 2.

V nasledujúcich kapitolách 3 a 4 sme sa venovali experimentom, ktoré sme robili. Výsledky experimentov si môžme znova pripomenúť. Ukázali sme, že je jednoduché natrénovať konvolučnú neurónovú sieť, aplikovať kontrast a jas na obrázok. Pri bližšej inšpekcii váh konvolučného filtra sme v nich videli jas a kontrast, ktorý bol aplikovaný na obrázok.

V ďalšom našom experimente sme učili konvolučnú neurónovú sieť Sobelov detektor hrán. Mali sme dve potencionálne implementácie Sobelovho detektora hrán pomocou konvolučných neurónových sieti. Tieto dva modely sme porovnali a vyšlo nám, že lepšia implementácia je pomocou spájania viacerých aktivačných funkcií. Pozerali sme sa aj na váhy modelu, ale nevyzerali tak ako by sme ich očakávali. Aj napriek tomu tento model sa naučil detegovať hrany aj na obrázkoch mimo datasetu na trénovanie.

Následne sme robili pokusy na datasetoch vygenerovaných z malého počtu parametrov. Na generovanie datasetov sme používali dva typy generátor jeden náš a druhý z práce [6] Andreja Lúčneho.

Na našom generátore jednoduchých tvarov sme učili autoenkodér a klasifikátor. Nacvičili sme viacero autoenkodérov s rôznou veľkosťou a porovnávali sme ich. Pozorovali sme lepšie výsledky pri väčších autoenkodéroch. Menšie autoenkodéry sa snažili optimalizovať gól, ale pre ich obmedzenú veľkosť sa naučili generovať obrázky, ktoré neboli tak výrazné a neboli schopné rekonštruovať jednotlivé tvary iba ich polohu a veľkosť.

Pri trénovaní klasifikátora sme si všimli, že na rôznych datasetoch boli rôzne inicializátori lepšie. Toto je vec, ktorá by sa dala ďalej skúmať v ďalších prácach.

Na generátore ružových lístkov sme cvičili autoenkóder. Všimli sme si väčšie zväčšienie kvality pri zväčšovaní latentného priestoru a na druhej strane veľmi malé zvyšovanie kvality pri pridávaní ďalších konvolučných filtrov.

Literatúra

[1] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[2] Faustino J Gomez, Risto Miikkulainen, et al. Solving non-markovian control tasks with neuroevolution. In *IJCAI*, volume 99, pages 1356–1361. Citeseer, 1999.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[4] Věra Krková. Kolmogorov's theorem and multilayer neural networks. *Neural networks*, 5(3):501–506, 1992.

[5] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[6] Andrej Lucny. On lindenmayer systems and autoencoders. In *ITAT*, pages 217–222, 2020.

[7] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[9] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Príloha: obsah elektronickej prílohy

V elektronickej prílohe priloženej k práci sa nachádza zdrojový kód použitý na experimenty. Zdrojový kód je zverejnený aj na stránke `https://github.com/adamsrsen/bakalarka/`.