Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Knihovna pro práci s hlubokými konvolučními neuronovými sítěmi v jazyce C#

Plzeň 2018 Michal Medek

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 13. května 2018

Michal Medek

Abstract

This thesis deals with the development of a library for convolutional neural networks. The theoretical part describes neural networks' evolution from simple idea to its practical use as a convolutional neural networks that are now a state-of-the-art method for majority of computer vision tasks. The following sections present in detail convolutional neural networks and algorithms used within the learning process and introduce popular convolutional neural networks libraries. The practical part includes a description of the implementation process and performed experiments that prove functionality and stability of the library. The end of this thesis analyses potential future extensions of the library and discusses the achieved results.

Abstrakt

Tato práce se zabývá implementací knihovny pro práci s konvolučními neuronovými sítěmi. Teoretická část stručně popisuje vývoj neuronových sítí od vzniku prvotní myšlenky po zavedení konvolučních neuronových sítí jako špičkové (state-of-the-art) metody pro většinu úloh počítačového vidění. Dále podrobněji rozebírá konvoluční sítě a algoritmy, které se používají v průběhu učicího procesu, například algoritmus zpětného šíření. Další kapitoly pak shrnují běžně používané knihovny pro práci s konvolučními neuronovými sítěmi. Praktickou část tvoří popis implementace knihovny, experimenty dokazující funkčnost i stabilitu knihovny a návrh budoucího rozšíření knihovny včetně zhodnocení výsledků této práce.

Poděkování

Rád bych poděkoval vedoucímu mé diplomové práce, Ing. Kamilu Ekšteinovi, Ph.D., za užitečné konzultace a rady.

Dále bych rád poděkoval Ing. Michalu Kacerovskému za stylistické a gramatické rady, které mi pomohly zvýšit úroveň práce, a v neposlední řadě děkuji své přítelkyni a rodině za podporu v průběhu celého studia.

Obsah

1 Úvod **1**

2 Umělé neuronové sítě **2**
2.1 McCulloch-Pittsův neuron 2
2.2 Perceptron 3
2.3 Období skepse 4
2.4 Neocognitron 5
2.5 Vícevrstvá neuronová síť 6
2.6 Hluboká neuronová síť 8

3 Konvoluční neuronové sítě **9**
3.1 Historie 10
3.2 Obecné principy 11
3.2.1 Algoritmus zpětného šíření 12
3.3 Jednotlivé vrstvy 13
3.3.1 Vstup 13
3.3.2 Konvoluční vrstva 14
3.3.3 Pooling vrstva 18
3.3.4 Lineární vrstva 19
3.3.5 Aktivační vrstva 20
3.4 Chybové funkce 22
3.4.1 MSE 23
3.4.2 Kategoriální cross-entropie 23
3.4.3 Binární cross-entropie 24
3.5 Optimalizační algoritmy 24
3.5.1 Stochastický gradientní sestup 24
3.5.2 Adadelta 25
3.5.3 Adam 26
3.6 Metody inicializace vah 27
3.7 Metody regularizace 28
3.7.1 Dropout 28
3.7.2 L2 regularizace 29
3.7.3 Augmentace dat 29
3.8 Shrnutí základních vlastnosti 30
3.8.1 Pozitivní vlastnosti 30
3.8.2 Negativní vlastnosti 31

3.9 Architektury . . . 32
3.9.1 AlexNet . . . 32
3.9.2 VGG . . . 32
3.9.3 GoogleLeNet . . . 32
3.9.4 ResNet . . . 33

4 Existující knihovny konvolučních neuronových sítí 34
4.1 TensorFlow . . . 34
4.2 Keras . . . 35
4.3 PyTorch . . . 35
4.4 Microsoft Cognitive Toolkit . . . 35
4.5 Caffe . . . 36

5 Formulace úlohy 37
5.1 Cíl práce . . . 37
5.2 Nástin řešení . . . 37

6 Implementace 39
6.1 Výběr implementace jazyka . . . 39
6.2 Navržená knihovna . . . 39
6.2.1 Model . . . 40
6.2.2 Načítání vstupních dat . . . 41
6.2.3 Tvorba architektury sítě . . . 42
6.2.4 Výběr chybové funkce . . . 43
6.2.5 Optimalizační algoritmy . . . 44
6.2.6 Inicializace vah . . . 44
6.2.7 Regularizační metody . . . 45
6.3 Finální realizace načítání dat . . . 45

7 Provedené experimenty 47
7.1 MNIST . . . 47
7.1.1 Vícekategoriální problém . . . 47
7.1.2 Vícekategoriální problém s použitím MSE . . . 52
7.2 Dogs vs Cats . . . 55
7.2.1 Binární problém . . . 55
7.3 Iris . . . 59
7.3.1 Vícekategoriální problém . . . 59
7.4 Jednotkové testy . . . 62
7.5 Čas běhu pro konkrétní vrstvy . . . 62

8 Návrhy na budoucí rozšíření 66

9 Závěr **68**

Literatura **69**

1 Úvod

S aplikacemi založenými na počítačovém vidění se v dnešní době setkáváme každý den. Od automatického označování přátel na sociálních sítích, přes automatizaci podnikových procesů v továrnách, až po autonomní řízení vozidel. Nejčastěji používanými algoritmy umožňujícími vývoj takových aplikací jsou právě konvoluční neuronové sítě. Predikce na následující roky udává stále větší a větší náklady v oblasti umělé inteligence, kam tyto algoritmy řadíme. Od roku 2018 do roku 2022 se předpokládá více než trojnásobný nárůst investic do tohoto odvětví [43]. Lze tak předpokládat i další vývoj umělých neuronových sítí včetně těch konvolučních.

Na počátku však byla pouhá myšlenka pokusit se imitovat proces probíhající v mozku a na základě toho vytvořit algoritmus, jenž bude schopný se učit. Vznikly umělé neuronové sítě a z těch se postupem času vyvinuly nejrůznější druhy sítí jako konvoluční neuronové sítě nebo rekurentní neuronové sítě.

Konvoluční neuronové sítě inspirované také slavným pokusem vědců Davida Hubela a Torstena Wiesela, kteří ukázali reakci buněk v mozku kočky na konkrétně natočené hrany [30], postupně skládají scénu napříč vrstvami od nejméně abstraktních tvarů po tvary velmi blízké konkrétním objektům. Tento proces nám umožňuje modelovat téměř vše, při dostatečném výpočetním výkonu a správné architektuře sítě.

Návrh architektury neuronové sítě je velmi složitá disciplína, která se neustále vyvíjí. Existuje mnoho různých variant a ty nejlepší se mění téměř každý rok, když vezmeme v potaz i chybové funkce, metody incializací vah nebo optimalizační algoritmy. Navíc je naprogramování takové konvoluční neuronové sítě velmi časově náročné, proto vznikají nejrůznější knihovny. Nejznámější z nich jsou například *TensorFlow*, *PyTorch*, *Caffe* nebo *Keras*. Tyto knihovny jsou však vysoce optimalizované, a tak pokud chce vývojář proniknout do tajů konkrétní implementace není to vůbec jednoduché. Pro jazyk C# dokonce implementace přehledné, stabilní a dokumentované knihovny chybí. Existují jen obalovací knihovny na již zmíněné implementace.

Cílem této diplomové práce je vytvoření knihovny pro práci s konvolučními neuronovými sítěmi v jazyce C#. Knihovna by měla sloužit jak k výukovým účelům, tak pro jednoduché experimenty. Hlavním požadavkem na knihovnu je uživatelská přívětivost.

2 Umělé neuronové sítě

Tato část stručně shrnuje historický vývoj *umělých neuronových sítí*, (angl. *artifical neural networks*) od původní myšlenky simulace procesu učení v mozku a vytvoření algoritmu, který bude schopen pokořit *pravidlové znalostní systémy* založené na manuálně sestavovaných pravidlech, až po dnes v praxi hojně využívané metody z oblasti *hlubokého učení* (angl. *deep learning*).

2.1 McCulloch-Pittsův neuron

Počáteční myšlenka, která stojí za zrodem umělých neuronových sítí, byla formulována roku 1943 v práci W. S. McCullocha a W. Pittse [38]. McCulloch a Pitts velmi zjednodušeně popsali, jak v mozku jednotlivé neurony komunikují a formulovali následující matematický model pro jeden neuron.

$$f(\xi) = \sum_{0}^{n} w_n \cdot x_n \tag{2.1}$$

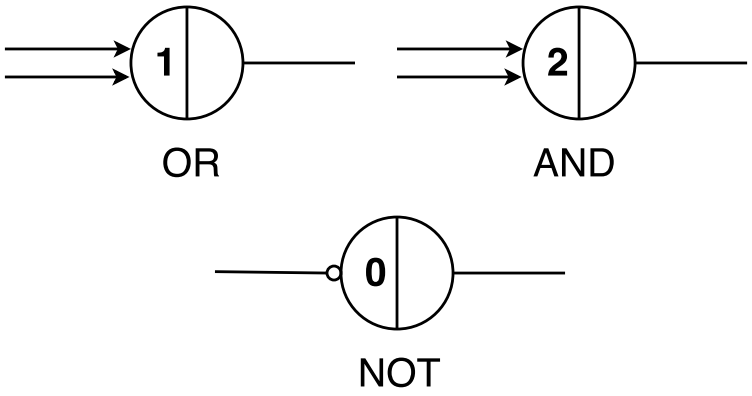
Vstupem modelu je n binárních hodnot, které jsou propojeny s tělem neuronu. V těle neuronu provádíme sumu vstupních hodnot, jež jsou násobené váhou w_n. Sumu následně porovnáváme s definovaným prahem θ.

$$y = \begin{cases} 1 & f(\xi) \geq \theta \\ 0 & f(\xi) < \theta \end{cases} \tag{2.2}$$

McCulloch a Pitts zároveň demonstrovali funkčnost jejich modelu pro standardní logické funkce *OR*, *AND* a *NOT* (obrázek 2.1), jejich práce ovšem byla míněna pouze jako teoretická a nebylo z ní zřejmé reálné využití vzhledem k absenci učicího pravidla.

Roku 1949 Donald Hebb ve své knize popsal proces učení na základě studia lidského mozku [20]. Předpokládal, že pokud bude neuron buzen korektně, spojení zesílí. Naopak pokud ne, spojení zeslábne.

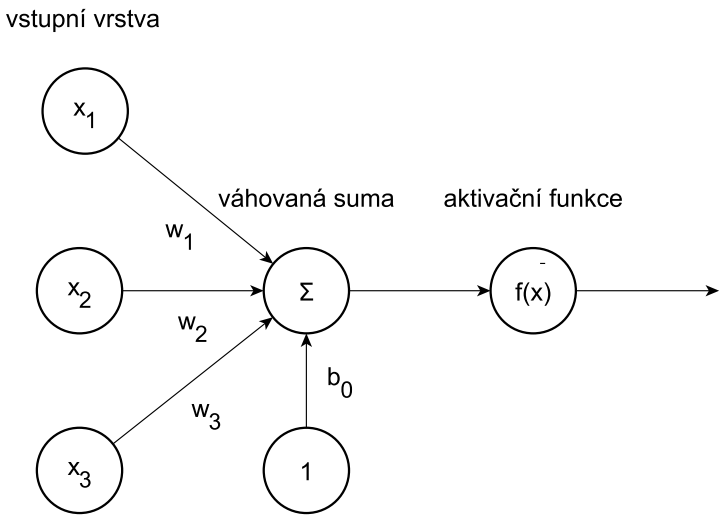
Definice 2.1. Pokud axon buňky A excituje buňku B a tím ji opakovaně nebo trvale aktivuje, dochází k růstovému procesu nebo metabolické změně v jedné nebo obou buňkách v závislosti na účinnosti buňky A. Tento princip označujeme jako *Hebbovo pravidlo.*



Obrázek 2.1: Reprezentace logických funkcí, dle Minského notace [15]

2.2 Perceptron

V návaznosti na práci McCullocha s Pittsem a inspirován Hebbovou prací, sestavil americký psycholog Frank Rosenblatt první prakticky využitelnou umělou neuronou síť, kterou nazval *Perceptron* [44]. Rosenblatt upravil McCulloch-Pittsův model neuronu a následně jej použil pro rozpoznávání jednoduchých znaků o velikosti 20×20 pixelů, které byly promítány na plátno (obrázek 2.2).

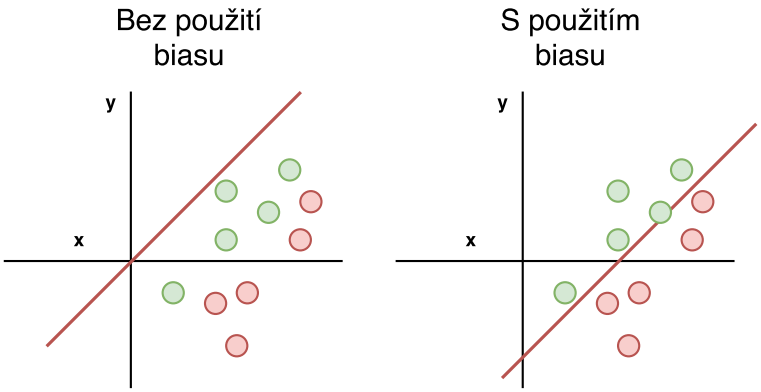


Obrázek 2.2: Matematický model Perceptronu [55]

Vstupem Perceptronu je n vstupních hran ohodnocených konkrétními *váhami*. Jeden vstup vždy tvoří tzv. *bias term*, který používáme pro posun *aktivační funkce* po ose x (obrázek 2.3). Každý vstup vynásobíme jeho vahou a takto ohodnocené je následně sečteme. Pokud je součet po odečtení prahu kladný, bude výstupem Perceptronu 1, jinak 0.

Definice 2.2. Ohodnocení hrany, která se nachází mezi dvěma neurony, označujeme jako *váhu*.

Definice 2.3. Jako *aktivační funkci* (angl. *activation function*) označujeme funkci, která je aplikována na sumu ohodnocení vstupních hran vynásobených s jejich váhami.



Obrázek 2.3: Použití biasu

Rosenblatt ukázku funkčnosti Perceptronu a učicího algoritmu realizoval poskládáním několika Perceptronů do jedné vrstvy. Každý Perceptron přijímal stejný vstup, ale nesl zodpovědnost za jiný výstup. Výstup byl poté odečten na základě maximální sumy každého z Perceptronů, ten s maximální sumou byl roven 1, ostatní byly nastaveny na 0.

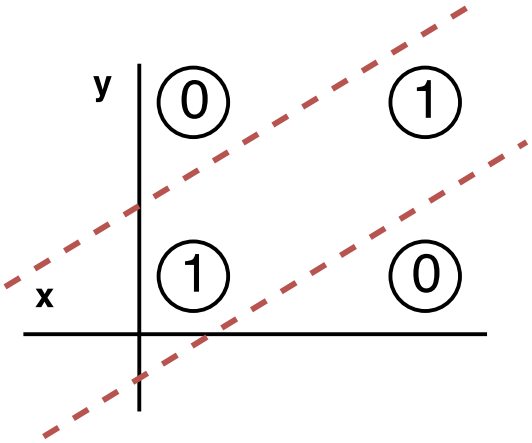
S adaptací Perceptronu přišli roku 1960 Bernard Widrow a Tedd Hoff, kteří odstranili aktivační funkci a jako výstup používali vážený vstup. Ve svém článku mimo jiné ukázali výhody odstranění prahovací aktivační funkce z důvodu její nespojité derivace a následně použili derivaci pro sledování vývoje chyby na základě změny vah. Díky tomu mohli najít lepší nastavení vah [53].

2.3 Období skepse

V této době se akademická obec rozdělila na dva tábory, z nichž jeden byl složen z velkých podporovatelů umělých neuronových sítí, kteří zastávali

názor, že jsou velmi blízko k stvoření umělé inteligence [2]. Druhý tábor tvořili převážně skeptici v oblasti umělých neuronových sítí, kteří zastávali spíše pravidlový přístup k řešení problémů a tuto oblast viděli jako mrtvou větev akademického výzkumu.

A právě dva zastánci druhé skupiny, Marvin Minsky a Seymour Papert, roku 1969 publikovali knihu, která zhodnotila generalizační sílu Perceptronu a zároveň popsala praktické nedostatky tohoto algoritmu [41]. Nejzásadnějším prvkem knihy byl pravděpodobně experiment, který potvrdil, že Perceptron není schopen separovat ani jednoduchý *nelineární problém* typu XOR (obrázek 2.4).



Obrázek 2.4: XOR problém [4]

Definice 2.4. Problém nazýváme *lineárně separovatelným*, pokud jsme schopni najít takovou nadrovinu, která body rozdělí na základě námi definovaných tříd. V opačném případě problém nazýváme *lineárně neseparovatelným*.

Velká část akademické obce se domnívá, že právě tato publikace zapříčinila velký úpadek zájmu o umělé neuronové sítě. Toto období bývá velmi často označováno jako první *zima umělé inteligence* (angl. *AI winter*) – období, kdy po vlně zájmu o umělé neuronové sítě přichází období nezájmu ze strany předních vědeckých časopisů a většiny výzkumníků.

2.4 Neocognitron

Roku 1980 publikoval japonský výzkumník Kunihiko Fukushima koncept zvaný *Neocognitron* [16]. Tento koncept byl opět inspirován mozkem, tentokrát funkcí *vizuálního kortexu*, tedy oblasti, která zajišťuje rozpoznávání

předmětů na základě nervových impulsů ze sítnice. Fukushima vycházel z následující struktury vizuálního kortexu: *spojení optického nervu → jednoduchá buňka → komplexní buňka → nízkoúrovňová hyperkomplexní buňka → vysokoúrovňová hyperkomplexní buňka.* O struktuře lze dále říci, že spojení mezi jednoduchou a komplexní buňkou je podobné jako mezi nízko a vysokoúrovňovou buňkou. To nám umožňuje vycházet z předpokladu, že postupným pronikáním hlouběji do sítě se zvyšuje deskriptivní úroveň jednotlivých vrstev, tedy narůstá úroveň abstrakce.

Původní algoritmus byl algoritmem s učením bez učitele a hloubka sítě závisela na složitosti rozpoznávaných objektů.

2.5 Vícevrstvá neuronová síť

Kniha Minského a Paperta několikrát poukázala na zásadní rozkol mezi původní myšlenkou umělých neuronových sítí a realizací Perceptronu. Perceptron na rozdíl od struktury mozku obsahuje pouze vstupní a výstupní vrstvu, která poté není schopná reprezentovat a řešit pokročilejší (lineárně neseparovatelné) problémy.

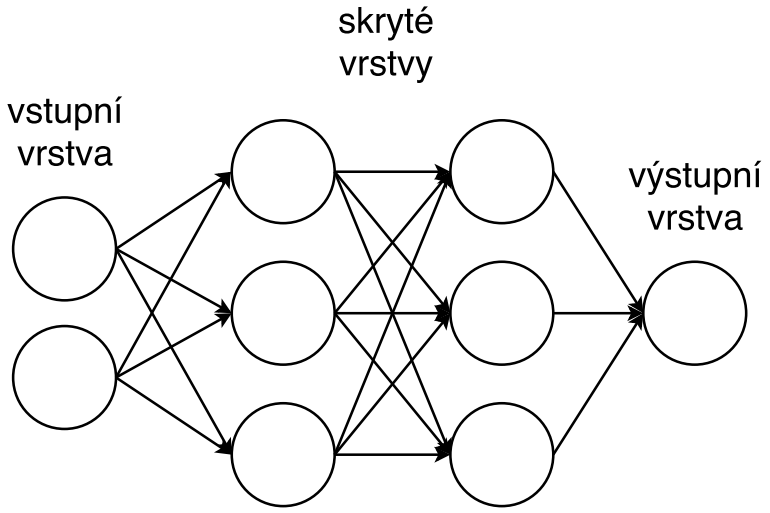
Definice 2.5. První vrstvu umělé neuronové sítě nazýváme *vrstvou vstupní.* Poslední vrstvu *vrstvou výstupní.* Všechny ostatní vrstvy mezi vstupní a výstupní vrstvou nazýváme *vrstvami skrytými* (angl. *hidden layers*).

Zjednodušeně by se dalo říci, že skryté vrstvy umožňují v rámci umělé neuronové sítě reprezentovat různé vzory, které poté značně zjednodušují a zefektivňují proces klasifikace. Důležité je zmínit, že vzory síť najde sama, nemusí být tedy součástí trénovacích dat. K tomu nebylo možné Perceptron použít vzhledem k jeho učicímu algoritmu:

1. Náhodně nastavíme malé váhy Perceptronům, kterých bude stejný počet jako předpokládaný počet výstupních tříd.
2. Pro i-tý vstup z množiny trénovacích dat vypočítáme výstup Perceptronů.
3. Pro každý z Perceptronů přizpůsobíme váhy dle učicího pravidla.
4. Inkrementujeme index i-tého vzorku z trénovacího setu a pokračujeme kroky 2–4 dokud Perceptrony špatně rozpoznávají vstupní data z trénovací množiny.

Na základě tohoto algoritmu přizpůsobujeme váhu pouze podle výstupu, ale nejsme schopni určit změnu vah pro případnou skrytou vrstvu.

Roku 1974 Paul Werbos ve své disertační práci dopodrobna analyzoval a popsal jak použít algoritmus zpětného šíření chyby (angl. *backpropagation*) pro učicí proces vícevrstvých perceptronů [52]. I přes to, že Werbos vyřešil otázku učení vícevrstvých perceptronů (angl. *multilayer perceptron*) s publikací článku o této problematice kvůli nezájmu ze strany akademické obce čekal do roku 1982. Bohužel zůstala jeho práce téměř bez povšimnutí a až roku 1986 Geoffrey Hinton ve spolupráci s Davidem Rumelhartem a Ronaldem Williamsem publikovali článek, který stručně a výstižně popisoval přínos algoritmu zpětného šíření (algoritmus bude detailněji popsán v kapitole 3) ve spojitosti s vícevrstvou neuronovou sítí (obrázek 2.5) [23]. Z této doby také pochází chybné označení vícevrstvé neuronové sítě jako tzv. *vícevrstvého perceptronu*. Označení je chybné z několika důvodů: Perceptron používá jako aktivační funkci prahovací funkci a vícevrstvá síť obvykle sigmoidu (použita v původním článku z roku 1986), hyperbolický tangens nebo jinou diferencovatelnou funkci. Druhým a pádnějším důvodem je učicí pravidlo Perceptronu, které se zásadně liší od algoritmu zpětného šíření [25].



Obrázek 2.5: Vícevrstvá neuronová síť [31]

Ve stejném roce tato trojice publikovala článek [24], kde ověřila řešení problémů, které u Perceptronu formuloval Minsky a Papert. Díky těmto pracím se obnovil velký zájem o oblast umělých neuronových sítí.

2.6 Hluboká neuronová síť

Na základě pokračujícího výzkumu v oblasti umělých neuronových sítí se pozvolna začaly uplatňovat tzv. *hluboké neuronové sítě* (angl. *deep neural networks*). Výhoda těchto sítí spočívala v možnosti popsat komplexnější problémy nejdříve jednoduššími příznaky a pak tyto příznaky kombinovat ve vyšších vrstvách sítě. Pokud poté provedeme kompozici dostatečného množství vrstev, jsme schopní modelovat i velmi složité funkce. Důvodů, proč se hluboké neuronové sítě (vícevrstvé neuronové sítě s více než jednou skrytou vrstvou) nepoužívaly od začátku, je hned několik. Prvním z nich je výpočetní náročnost: čím hlubší a větší síť, tím déle bude trénování trvat. Mezi další důvody patří jevy popisované jako explodující nebo zanikající gradient a optimalizační metody typu gradientní sestup. Těmito problémy se budeme podrobněji zabývat v kapitolách 3.5 a 3.6. Obecně používáme pro oblast strojového učení, kde je kladen důraz na používání hlubokých neuronových sítí a minimální předzpracování dat, termín hluboké učení [26].

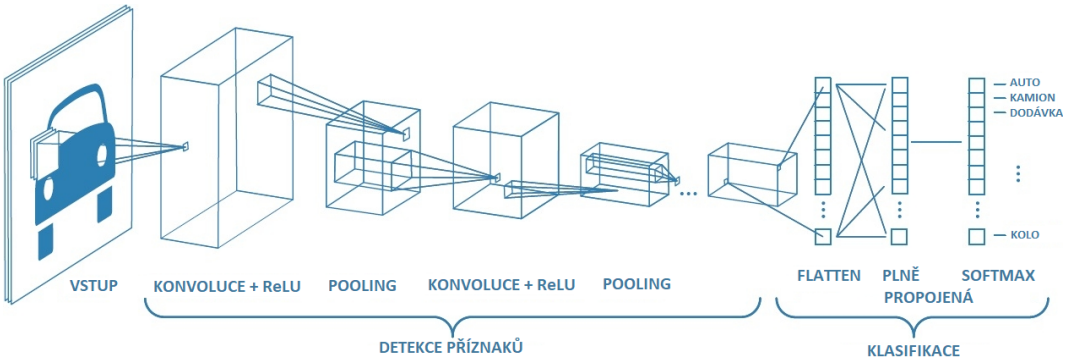
Definice 2.6. Umělou neuronovou síť nazveme *hlubokou neuronovou sítí* v případě, že obsahuje více než jednu skrytou vrstvu.

Definice 2.7. Oblast strojového učení s použitím hlubokých neuronových sítí a minimálním předzpracováním dat budeme označovat pojmem *hluboké učení* (angl. *deep learning*).

3 Konvoluční neuronové sítě

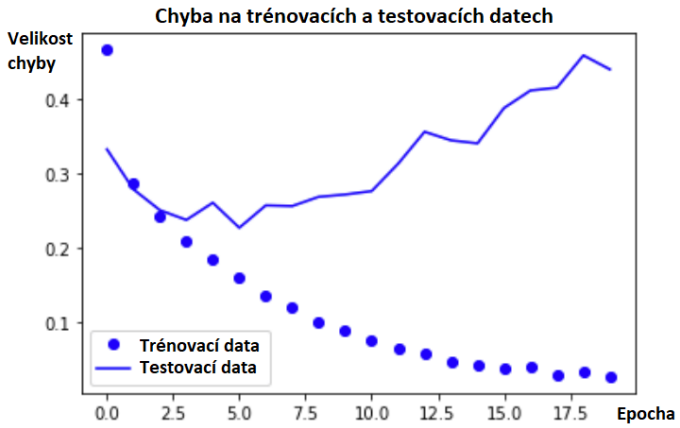
V této kapitole popíšeme myšlenku *konvolučních neuronových sítí* (angl. *convolution neural networks*) a také to, jak tyto sítě fungují. Konkrétně se budeme věnovat funkci a matematické reprezentaci jednotlivých vrstev. V rámci toho zmíníme pozitivní i negativní vlastnosti tohoto druhu sítí. Dále si představíme metody, které nám umožňují předcházet *přeučení sítě* (*přetrénování sítě*) a urychlují *konvergenci učení*. Na závěr si popíšeme aktuálně používané architektury a zmíníme případného nástupce konvolučních neuronových sítí.

Definice 3.1. *Konvoluční neuronová síť*, dále jen CNN, je hluboká neuronová síť, která obsahuje jednu anebo více konvolučních vrstev. Konvoluční neuronová síť se sice primárně používá v oblasti počítačového vidění (angl. *computer vision*), ale lze jejích vlastností využít například i v oblasti zpracování přirozeného jazyka (angl. *natural language processing*). Její hlavní výhodou je, že pro zpracování obrazu nemusíme implementovat složité metody předzpracování (obrázek 3.1).



Obrázek 3.1: Příklad segmentace konvoluční neuronové sítě na část pro předzpracování příznaku a na část, která tyto příznaky kombinuje a na jejich základě provádí klasifikaci [1]

Definice 3.2. *Přeučení sítě* (angl. *overfitting*) je stav, kdy síť vykazuje minimální chybu na trénovacích datech, ale na testovacích datech je chyba výrazně vyšší. Navíc lze obvykle přeučení jednoduše detekovat při vynesení velikosti chyby na testovacích a trénovacích datech v závislosti na čase



Obrázek 3.2: Znázornění vývoje chyby na trénovací a testovací množině dat. Z grafu vidíme, že zhruba u třetí epochy dochází k přetrénování sítě. Chyba se snižuje pouze na trénovací množině a na testovací začíná pozvolna růst [7]

(*počtu epoch*) do grafu (viz obrázek 3.2). Pro snadnější detekci přeučení je také vhodné vynést do grafu i úspěšnost na trénovacích a testovacích datech v závislosti na čase (počtu epoch). Opak přeučení je *nedoučení sítě* (angl. *underfitting*), které charakterizuje obvykle vysoká chyba na trénovací množině, opět ho lze nejlépe vyčíst z grafu.

Definice 3.3. *Epocha* je stav, kdy trénovacím procesem projdou všechna data. Obvykle v průběhu učicího procesu zpracováváme naráz blok dat (batch). Pokud zpracujeme všechny bloky (všechna trénovací data), jedná se o jednu epochu. Naopak o *iteraci* mluvíme v kontextu bloku, po zpracování jednoho bloku jsme tedy provedli jednu iteraci. Budeme-li například 128 vstupních obrázků zpracovávat po blocích o osmi prvcích, provedeme v rámci jedné epochy 16 iterací (128 : 8).

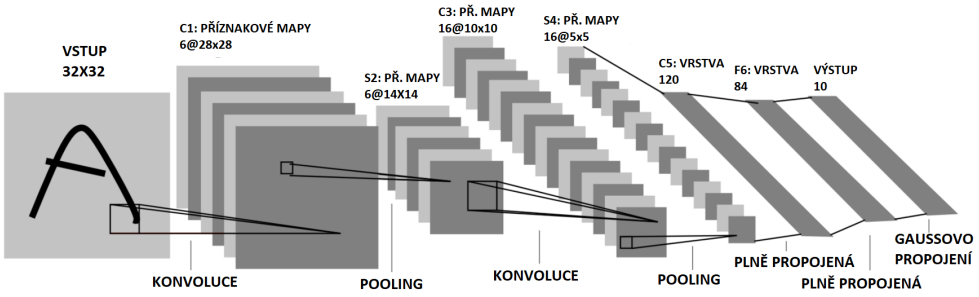
Definice 3.4. O *konvergenci sítě* mluvíme z pohledu chyby na trénovacích datech. Pokud chyba konverguje k nějaké hodnotě, ideálně nízké hodnotě, víme, že učicí proces probíhá v pořádku.

3.1 Historie

Roku 1989 Yann LeCun společně se svými spolupracovníky z AT&T Bell Labs publikoval článek, ve kterém popsal první velmi úspěšné využití hlubokých neuronových sítí v praxi a ověřil užití algoritmu zpětného šíření aplikovaného na velkou sadu dat [35]. Uvedl novou architekturu hluboké neuronové

sítě zvanou konvoluční neuronová síť, jež byla inspirována Neocognitronem Kunihiko Fukushimy ve smyslu spolupráce různých vrstev a jejich společného dekomponování scény od jednodušších příznaků po složitější. Na rozdíl od Neocognitronu ale konvoluční neuronová síť využívala algoritmus zpětného šíření. V rámci publikace využil autor data poštovní společnosti U.S. Postal Service, která obsahovala množství poštovních směrovacích čísel. Ta byla segmentována na jednotlivé číslice použité pro experimenty. Výsledná 5% chybovost na trénovací množině, které se podařilo dosáhnout za pomoci CNN, představovala v té době nejlepší dosažený výsledek v oblasti rozpoznávání číslic.

Použitá síť LeNet-5 se skládá ze vstupní, výstupní a z pěti skrytých vrstev (viz obrázek 3.3). Vstupní obrázek o velikosti 32×32 pixelů nejdříve zpracovává *konvoluční vrstva* následovaná *pooling vrstvou*, tyto dvě vrstvy jsou poté aplikovány ještě jednou (detailnější popis vrstev v kapitole 3.3). Algoritmus následně převede obrázek do 1D vektoru a předá ke zpracování *plně propojené vrstvě* k rozpoznávání čísla. Výstup čteme z *výstupní vrstvy*.



Obrázek 3.3: Ukázka architektury sítě LeNet-5 [5]

Od té doby se architektury konvolučních neuronových sítí výrazně proměnily, ale základní myšlenka použití konvoluční vrstvy zůstala stejná. V následujících kapitolách si CNN popíšeme podrobněji.

3.2 Obecné principy

V této kapitole detailněji popíšeme obecné principy použité v konvolučních neuronových sítích. S těmito principy je nutné se seznámit před podrobným rozborem architektury.

3.2.1 Algoritmus zpětného šíření

Algoritmus zpětného šíření je učicí algoritmus používaný v umělých neuronových sítích. Jedná se o metodu, bez které by pravděpodobně umělé neuronové sítě nikdy nedosáhly dnešní popularity. V následující části se zaměříme na *metodu učení s učitelem* (angl. *supervised learning*). Celý proces učení závisí na existenci adaptabilních vah, jež v umělé neuronové síti spojují jednotlivé neurony. Na základě označkovaných vstupních dat se snažíme síť natrénovat – změnit váhy tak, aby učení konvergovalo. Jedná se o iterační algoritmus, kdy se postupně snažíme nalézt co nejlepší nastavení vah snižováním chyby na trénovacích datech.

Definice 3.5. Algoritmy *učení s učitelem* jsou typem algoritmů, pro jejichž učicí proces máme k dispozici označkovaná vstupní data. Pokud bychom tak chtěli řešení, které od sebe oddělí obrázky, museli bychom mít obrázky jablek a hrušek, u nichž bude uvedené, co zobrazují.

Algoritmus se skládá z následujících kroků:

1. dopředné šíření (dopředný krok),
2. výpočet chyby,
3. zpětné šíření (zpětný krok),
4. aktualizace vah.

Dopředné šíření

Dopředný krok zahájíme vložením dat na vstup umělé neuronové sítě. Data následně projdou napříč celou sítí, vrstvu po vrstvě, a na základě parametrů (vah) jednotlivých vrstev zprostředkuje síť výstup. Váhy na počátku celého učicího procesu inicializujeme metodami rozebranými v kapitole 3.6.

Výpočet chyby

Na základě výstupu sítě po provedení dopředného šíření, vypočítáme chybu, s níž byla klasifikace provedena. Algoritmů pro výpočet chyby existuje několik a jsou závislé na podobě vstupních dat, detailněji je popisuje sekce 3.4.

Zpětné šíření

Zpětné šíření je založené na algoritmu zvaném *řetízkové pravidlo* (angl. *chain rule*), pomocí něhož jsme schopni propagovat chybu, napříč sítí zpět z výstupu na vstup [23].

Aktualizace vah

Po dokončení zpětného šíření aktualizujeme váhy jednotlivých vrstev. Metody na základě, kterých aktualizujeme váhy, nazýváme *optimalizačními metodami* (kapitola 3.5). Cílem tohoto kroku je změnit váhy v jednotlivých vrstvách tak, aby se snížila chyba. Pomocí gradientu vypočteného v rámci zpětného šíření, určíme směr, kterým chybu maximálně snížíme. Následně tímto směrem učiníme krok v závislosti na typu optimalizační metody.

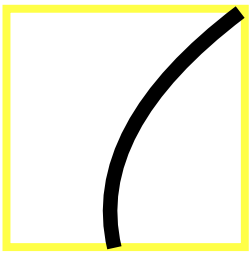
3.3 Jednotlivé vrstvy

V této práci se budeme většinou zabývat pouze *sekvenčními modely* konvolučních neuronových sítí, tedy sítěmi tvořenými skládáním jednotlivých vrstev. Grafové modely zmíníme až v sekci nejznámějších architektur konkrétních sítí. Kapitola se věnuje typům vrstev konvoluční neuronové sítě, a to jak z pohledu jejich významu, tak z pohledu jejich implementace a matematického popisu. Kapitola se zaměří především na aplikace konvoluční neuronové sítě na obrazová data, předpokládáme tedy použití 2D konvoluce.

Definice 3.6. *Sekvenční architektura* konvoluční neuronové sítě je taková architektura, kde existují spojení pouze mezi po sobě jdoucími vrstvami. Jde tedy o sekvenci vrstev, ze kterých se daný model skládá. Opak představuje *grafová architektura*, jež může obsahovat i spojení přes několik vrstev a celkově je její model často výrazně složitější.

3.3.1 Vstup

Výhodou konvolučních neuronových sítí je fakt, že při zpracování obrázků na vstup přivádíme přímo obrázek a není vyžadováno další předzpracování. Vstup charakterizujeme třemi dimenzemi: šířkou, výškou, hloubkou. Hloubka představuje počet barevných kanálů obrázku (např. tři pro RGB model).



0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	30	50	0	0

Obrázek 3.4: Ukázka obrázku ve stupních šedi a číselné reprezentace jednotlivých pixelů [9]

Pracujeme-li s obrázky, budeme obvykle používat 2D konvoluci; pokud pracujeme obecně s nějakým 1D vektorem (například s předzpracovaným textem – v takovém případě už je předzpracování nutné), použijeme 1D konvoluci.

3.3.2 Konvoluční vrstva

Konvoluční vrstva tvoří základní stavební prvek konvolučních neuronových sítí. Tato vrstva je určena k nalezení lokálních příznaků a zároveň se na principu *lokální konektivity* snaží ušetřit co možná nejvíce parametrů (vah), které s přibývajícím množstvím zpomalují proces trénování sítě. Lokální konektivita je realizována pomocí malých *konvolučních filtrů*. Ty se aplikují postupně na celý obrázek (probíhá konvoluce filtru s obrázkem) a na základě aplikace filtru získáme *příznakovou mapu* (angl. *feature map*) pro konkrétní filtr, který je pak s ostatními filtry vstupem do další vrstvy. Následující text detailněji rozebere zmíněný proces.

Definice 3.7. *Konvoluční filtr* je na počátku nejčastěji malá náhodně inicializovaná matice. Obvykle o velikosti $3 \times 3 \times hloubka$, $5 \times 5 \times hloubka$ nebo $7 \times 7 \times hloubka$ v závislosti na velikosti obrázku a typu úlohy. Váhy tohoto filtru jsou postupně přizpůsobovány danému problému pomocí algoritmu zpětného šíření.

Definice 3.8. *Příznaková mapa* vzniká konvolucí obrázku a konvolučního filtru. Jedná se o reprezentaci obrázku pomocí lokálních příznaků jednoho konkretního konvolučního filtru.

Dopředné šíření

V rámci dopředného šíření provedeme konvoluci pomocí všech konvolučních filtrů a vytvoříme příznakové mapy, které budou výstupem této vrstvy. Nejjednodušeji lze dopředný krok vyčíst z obrázku 3.5: Vstupem je obrázek o rozměrech 5×5 pixelů s třemi barevnými kanály. Vidíme, že bylo použito nulové zarovnání a dva filtry o velikosti $3 \times 3 \times 3$, pro každý filtr je uveden bias. Na obrázku vidíme vyznačenou aplikaci druhého filtru na první pixel vstupního obrázku. Výsledné hodnoty pro příznakové mapy lze spočítat takto:

$$feature\_map_{index} = (\sum_{i=-k}^{k} \sum_{j=-k}^{k} f(x-i, y-j) \cdot g(i,j)) + b_{index}, \qquad (3.1)$$

kde provedádíme *Hadamardův součin* [40] matice reprezentující vstupní obrázek f s maticí konvolučního filtru g, dle zpracovávaného bodu $[x, y]$ a o velikosti konvolučního filtru k. Výsledné hodnoty sečteme a na závěr k nim přičteme hodnotu biasu b.

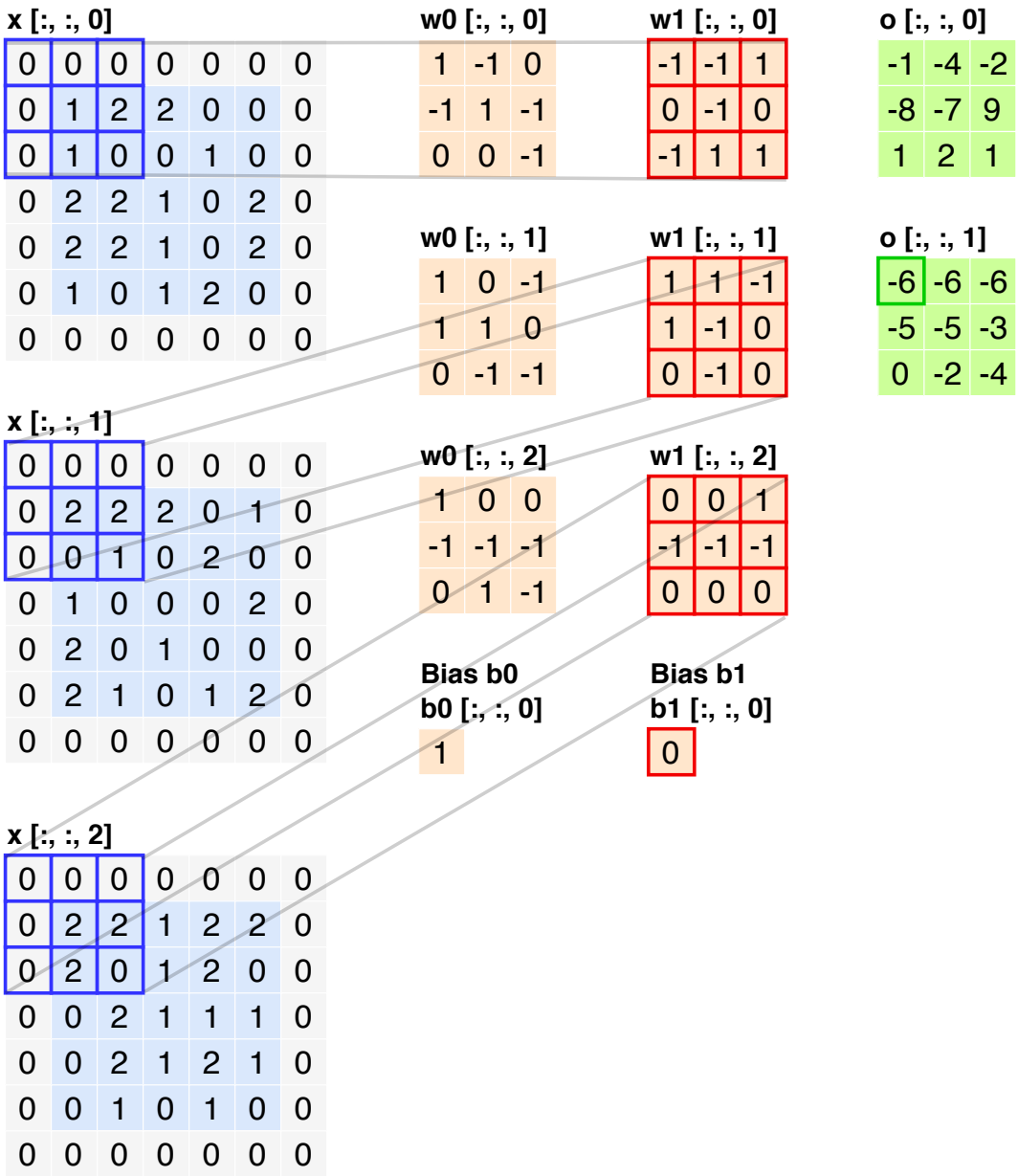
Zpětné šíření

Pro konvoluční vrstvu vypočítáme matici zpětného šíření jako sumu násobků vstupů z předchozí vrstvy (zpětný krok) s konvolučním filtrem. Stejně tak vypočítáme derivaci vah, kdy provedeme sumu součinů z předchozí vrstvy, ale tentokrát s výstupem konvoluční vrstvy, který jsme spočítali v rámci dopředného kroku. Výslednou hodnotu poté normalizujeme počtem prvků, jež obsahuje jednu zpracovávanou skupinu dat. Hodnotu biasu pro každý konvoluční filtr vypočítáme jako sumu gradientů pro každý bias.

Hyperparametry

Jako *hyperparametry* označujeme nastavitelné parametry pro jednotlivé vrstvy. Hyperparametry konvoluční vrstvy ovlivňují velikost výstupních příznakových map a trénování samotné.
Konvoluční vrstva má následující hyperparametry:

- *Velikost konvolučního filtru* – filtr je u 2D konvoluce vždy třídimenzionální, s tím, že hloubka filtru závisí na hloubce obrázku. Velikostí myslíme šířku a výšku filtru, která musí být stejná.
- *Počet konvolučních filtrů* – čím více konvolučních filtrů použijeme, na tím více příznaků jsme schopni scénu dekomponovat, ale zároveň bude

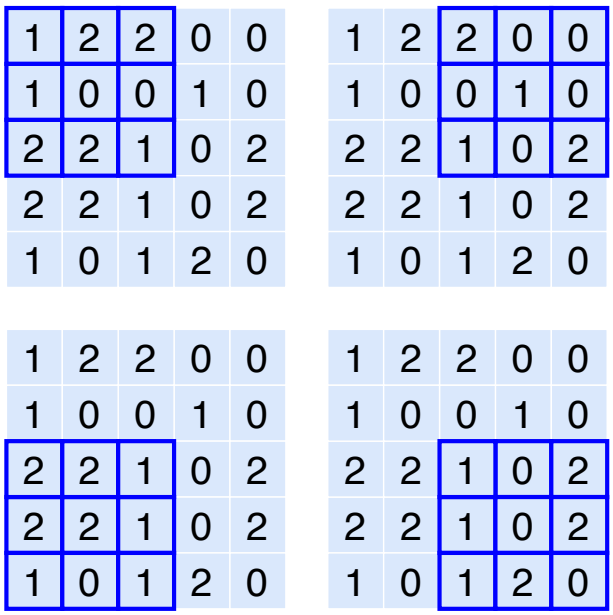


Obrázek 3.5: Ukázka použití konvolučního filtru $3 \times 3 \times 3$ při kroku rovnému dvěma a nastavení nulového zarovnání s notací z programovacího jazyka MATLAB [32]

trénování časově náročnější a může být náchylnější k přetrénování. Obvykle se počet filtrů zvyšuje na základě toho, jak hluboko v síti se nacházíme.

- *Zarovnání nulami* (angl. *zero-padding*) – při aplikaci tohoto hyperparametru počítáme konvoluci i přes pixely na úplném okraji obrázku. To vnitřně zajistíme doplněním vstupního obrázku o řádky a sloupce plné nul, tak abychom mohli posouvat konvoluční filtr po všech pixelech. Díky tomu můžeme například docílit toho, že bude výstupní dimenze příznakové mapy stejná jako dimenze vstupního obrázku (při kroku jedna). Vzorec pro výpočet počtu řádků, o které se musí obrázek rozšířit je následující: $P = 0.5(F-1)$, kde P je výsledná hodnota pro počet řádků a F je velikost konvolučního filtru. Pokud vyjde neceločíselná hodnota pro P, nelze použít zarovnání nulami.

- *Krok* (angl. *stride*) – krok nám říká, jak budeme posouvat konvoluční filtr po obrázku (viz obrázek 3.6). Krok aplikujeme jak horizontálně, tak vertikálně.



Obrázek 3.6: Ukázka posunu konvolučního filtru při kroku rovnému dvěma

V souvislosti s nastavením hyperparametrů se dostáváme k jejich omezení [32], na obrázku 3.5 lze vidět, že při sudé velikosti konvolučního filtru

nejsme schopni určit střed (umístění bodu), který bude konvoluci reprezentovat v příznakové mapě. To je také důvod, proč se v praxi používají výhradně liché velikosti filtru.

Dále vždy musí platit, že výsledkem následujícího vzorce bude celočíselná hodnota, pokud ne, hyperparametry nelze takto nastavit:

$$y = \frac{1}{S}(W - F + 2P) + 1, \tag{3.2}$$

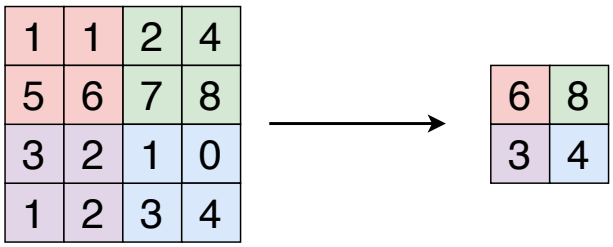
kde W je velikost vstupního obrázku (výška nebo šířka, hodnoty musí být stejné), F je velikost konvolučního filtru (opět výška nebo šířka), P je hodnota pro nulové zarovnání (pokud není použito, bude hodnota rovna nule). Proměnná S značí velikost posunu. Pokud y není celočíselná hodnota, konvoluci nelze provést a hyperparametry musí být nastaveny jinak.

3.3.3 Pooling vrstva

Pooling vrstva slouží především pro redukci parametrů konvoluční neuronové sítě. Tuto vrstvu aplikujeme typicky po konvoluční vrstvě, kdy se snažíme snížit počet parametrů, a tím i výpočetní složitost.

Dopředné šíření

Pro realizaci pooling vrstvy používáme různé algoritmy, tím nejrozšířenějším je algoritmus zvaný *max-pooling*. Mezi další používané algoritmy patří například *L2-pooling* anebo *average pooling*. U tohoto algoritmu používáme posuvné okénko a vybíráme vždy nejvyšší hodnotu ze vstupní matice, která bude okénko reprezentovat.



Obrázek 3.7: Algoritmus max-pooling, kde na základě nastavení posuvného okénka vybíráme jeho maximální prvek jako jeho reprezentaci. V tomto případě je krok 2 a filtr má velikost 2×2 [32]

Zpětné šíření

V pooling vrstvě je šíření závislé na typu metody. Pokud předpokládáme použití max-pooling vrstvy, budeme zpětně šířit chybu jen přes body, pro které byl maximální prvek určen. Měli bychom si tedy pamatovat pozici maximálního prvku z dopředného průchodu a při rekonstrukci matice zpětného šíření šířit chybu jen přes tyto body.

Hyperparametry

Stejně jako konvoluční vrstva má i pooling vrstva své hyperparametry:

- *Velikost posuvného okénka* – filtr je u 2D konvoluce vždy třídimenzionální s tím, že hloubka filtru závisí na hloubce vstupních dat. Velikostí tedy myslíme šířku a výšku filtru, která musí být stejná.
- *Krok* – krok nám říká, jak budeme posouvat konvoluční filtr po obrázku (viz obrázek 3.6). Krok aplikujeme jak horizontálně, tak vertikálně.

U pooling vrstev obvykle nepoužíváme zarovnání nulami.

3.3.4 Lineární vrstva

Lineární vrstva slouží pouze k sečtení váhovaných hran vstupujících do jednotlivých neuronů z vrstvy předchozí. Lineární vrstva, která bývá obvykle následována aktivační vrstvou, je úplně stejná jako ta ve vícevrstvé neuronové síti, kde bývá přímo napojená na aktivační vrstvu – po sečtení váhovaných vstupů je aplikována aktivační funkce. Tuto vrstvu obvykle nazýváme plně propojenou (angl. *dense* nebo *fully connected*), protože všechny neurony předchozí vrstvy jsou propojené se všemi neurony aktuální vrstvy.

Dopředné šíření

Jak již bylo zmíněno, v dopředném kroku pouze sečteme pro každý neuron všechny váhované vstupy (viz následující rovnice):

$$L_x = \sum_i (IN_i \cdot W_i) + b_x, \tag{3.3}$$

kde i je index napříč všemi vstupními neurony a x značí index neuronu v lineární vrstvě a biasu b_x.

Zpětné šíření

V lineární vrstvě nejdříve určíme derivaci vah dW jako násobek matice posledního vstupu do této vrstvy *Last_input*. Následně provedeme transpozici výsledné matice a jako In označíme matici z předchozí vrstvy, vypočtenou pomocí zpětného šíření, n je počet prvků v jednom bloku:

$$d\mathbf{W} = \frac{1}{n}(\mathbf{Last\_input}^T \cdot \mathbf{In}). \tag{3.4}$$

Výstupní matici zpětného šíření vypočítáme jako:

$$\mathbf{B} = \mathbf{In} \cdot \mathbf{W}^T, \tag{3.5}$$

kde W^T je transponovaná matice vah v dané vrstvě. Bias určíme jako průměr ze vstupní matice z předchozí vrstvy.

Hyperparametry

Lineární vrstva má pouze jeden hyperparametr, který je o to zásadnější: Jedná se o *počet neuronů* v této vrstvě. Počet závisí na konkrétní úloze a obvykle je předmětem procesu ladění parametrů (angl. *fine tuning*), kdy testujeme různá nastavení. Obecně platí, že čím více neuronů, tím větší vyjadřovací schopnost, ale také větší sklon k přeučení a delší doba trénování.

3.3.5 Aktivační vrstva

Aktivační vrstvu používáme pro zavedení nelinearity do neuronové sítě. Cílem je vytvořit co nejlepší *rozhodovacích hranic*.

Definice 3.9. Jako *rozhodovací hranici* označujeme nadplochu, kterou se snažíme oddělit data přiváděná na vstup. Například pokud se budeme snažit klasifikovat do správné třídy obrázky koček a psů, rozhodovací hranice v ideálním případě tyto dvě množiny rozdělí.

Dopředné šíření

Aktivační vrstva obsahuje vždy konkrétní aktivační funkci, zde je výčet těch nejznámějších [54]:

- *Sigmoida* – jedná se o historicky první používanou aktivační funkci, v dnešní době se kvůli problému zvanému *zanikající a explodující gradient* (angl. *vanishing and exploding gradient*) a složitějšímu výpočtu používá minimálně (rovnice 3.6).

- *Tanh* – hyperbolický tangens se v běžných sekvenčních modelech konvolučních neuronových sítí také používá minimálně, a to ze stejného důvodu jako sigmoida (rovnice 3.7).

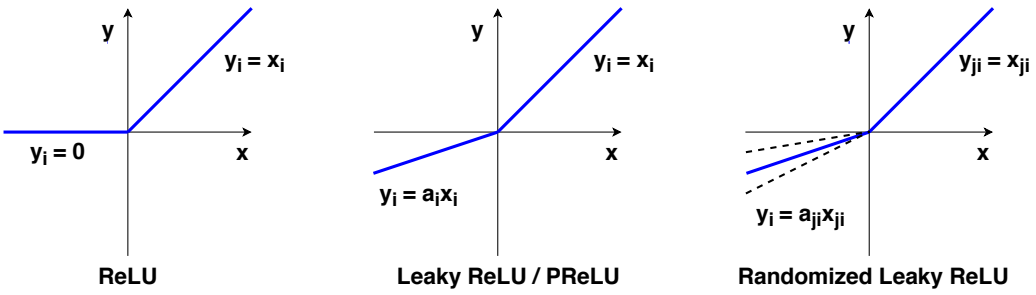
- *ReLU* – Rectified Linear Unit, jedná se o nejpoužívanější funkci kvůli své jednoduchosti a také odolnosti vůči problému zanikajícího gradientu. Navíc tato aktivační funkce zrychluje konvergenci sítě. Na druhou stranu v případě záporného gradientu může nastat *problém umírající ReLU* (angl. *dying ReLU*). Funkce je vyjádřena rovnicí 3.8.

- *Leaky ReLU* – varianta ReLU, která má odstranit problém umírající ReLU. Funkce je vyjádřena v rovnici 3.9, kde a je fixní parametr v rozsahu $(1, +\infty)$.

- *PReLU* – parametrická ReLU, stejně jako Leaky ReLU odstraňuje problém umírající ReLU, ale v tomto případě parametr nevolíme my, ale je volen na základě výsledků algoritmu zpětného šíření. Funkce má stejný předpis jako Leaky ReLU.

- *Randomized ReLU* – jedná se o randomizovanou verzi funkce Leaky ReLU, jejíž hlavní devizou je odolnost vůči přetrénování, protože parametr je náhodný. Předpis funkce je vyjádřen rovnicí 3.10, kde U je rovnoměrné rozdělení. Pro účely testování použijeme průměrné a, které vypočítáme v průběhu trénování.

- *Softmax* – tato funkce se vyskytuje pouze v poslední vrstvě vícevrstvých neuronových sítí, a to proto, že nám umožní určit pravděpodobnost klasifikace každého z n výstupů. Součet jednotlivých pravděpodobností výstupních neuronů bude tedy vždy roven téměř 1 (z důvodu zaokrouhlovací chyby) (rovnice 3.11).



Obrázek 3.8: ReLU aktivační funkce a její adaptace [54]

$$f(x_i) = \frac{1}{1 + e^{-x_i}} \tag{3.6}$$

$$f(x_i) = \tanh(x_i) = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}} \tag{3.7}$$

$$f(x_i) = \max(0, x_i) \tag{3.8}$$

$$f(x_i) = \begin{cases} x_i & x_i \geq 0 \\ ax_i & x_i < 0 \end{cases} \tag{3.9}$$

$$f(x_i) = \begin{cases} x_i & x_i \geq 0 \\ ax_i & x_i < 0 \end{cases}, a \sim U(l, u), l < u; l, u \ \in [0, 1) \tag{3.10}$$

$$f(x_i) = \frac{e^{x_i}}{\sum_{i=0}^{n} e^{x_i}} \tag{3.11}$$

Zpětné šíření

U zpětného šíření, stejně tak jako u ostatních vrstev, nejdříve vypočítáme lokální gradient a pak na základě řetízkového pravidla provedeme Hadamardův součin lokálního gradientu se vstupem z předchozí vrstvy (zpětný krok).

$$\mathbf{B} = \mathbf{In} \odot \mathbf{Out}', \tag{3.12}$$

kde B je matice vypočtená pro zpětné šíření, Out' je derivace aktivační funkce, kam dosadíme aktivace vypočtené v dopředném kroku – lokální gradient, a In je vstup z předchozí vrstvy.

3.4 Chybové funkce

Chybovou funkci (angl. *loss function*) využíváme v rámci algoritmu zpětného šíření pro určení aktuální chyby a možnost její zpětné propagace. Správným zvolením můžeme urychlit konvergenci sítě. Její výběr závisí na typu vstupních dat, respektive na typu tříd vstupních dat. Ty můžeme rozdělit takto:

- *binární problém* – jedná se o problém, kdy vstupní data dělíme do dvou tříd. Třída bude typicky reprezentována hodnotou 0 a 1. Pro tento typ problému používáme chybovou funkci zvanou *binární cross-entropie*.

- *kategoriální problém* – máme několik tříd, do kterých chceme daný objekt klasifikovat. Třídy mohou být popsány dvěma způsoby:
  - *hodnota* – máme například tři třídy reprezentované hodnotami 1, 2, 3; pro tento typ zápisu tříd běžně používáme chybovou funkci *MSE*,
  - *binární reprezentace* – opět máme tři třídy, ale jejich popis bude následující: pro první hodnotu použijeme $[1, 0, 0]$, druhou $[0, 1, 0]$ a pro třetí $[0, 0, 1]$. Typicky použijeme jako chybovou funkci *kategoriální cross-entropii*.

V následující části si popíšeme jednotlivé chybové funkce, které reprezentují každou z popsaných tříd. Nutno však zmínit, že chybových funkcí existuje celá řada, zde zmíníme jen ty základní.

3.4.1 MSE

MSE (angl. *Mean Squared Error*) je chybová funkce převzatá z algoritmu lineární regrese, jednoho z nejstarších algoritmů strojového učení. Klasifikované třídy jsou reprezentované konkrétní číselnou hodnotou a s tím úzce souvisí i počet neuronů ve výstupní vrstvě umělé neuronové sítě. Vzhledem k tomu, že výstup sítě porovnáváme vždy jen s jednou hodnotou, výstupní vrstva bude tvořena pouze jedním neuronem. Tuto techniku v dnešní době používáme zřídka vzhledem k rychlejší konvergenci při použití kategoriální cross-entropie. Předpis pro MSE je následující:

$$E = \frac{\sum_{i=0}^{n}(y_i - \hat{y}_i)^2}{2n}, \tag{3.13}$$

kde n je počet prvků pro jeden batch, y je předpokládaná hodnota na výstupu (dle trénovacích dat) a $\hat{y}$ je skutečná hodnota na výstupu.

3.4.2 Kategoriální cross-entropie

Kategoriální cross-entropie (angl. *categorical cross-entropy*) je nejrozšířenější chybovou funkcí. Používá se především z důvodu rychlosti konvergence sítě, kdy předpokládáme použití aktivační funkce softmax. Ve výstupní vrstvě sítě bude počet neuronů roven počtu rozpoznávaných tříd. Jako vstupní informaci o třídě očekáváme vektor nul o stejném počtu prvků jako rozpoznávaných tříd a na indexu aktuální třídy hodnotu 1. Funkce má následující předpis:

$$E = -\frac{\sum_{i=0}^{n}\sum_{c=0}^{n_c} y_{i,c} * \log(\hat{y}_{i,c})}{n}, \tag{3.14}$$

kde n je počet prvků pro jeden batch, první suma tedy iteruje napříč všemi prvky v daném skupině s indexem i a druhá suma iteruje napříč všemi výstupy pro aktuálně zpracovávaný prvek s indexem c. Význam y a $\hat{y}$ zůstává stejný jako u MSE.

3.4.3 Binární cross-entropie

Binární cross-entropii (angl. *binary cross-entropy*) používáme pro binární problémy, kdy očekáváme třídy s hodnotami 0 nebo 1. Výstupní vrstva sítě je v tomto případě tvořena jedním neuronem. Funkce má následující předpis:

$$E = -\frac{\sum_{i=0}^{n} y_i * \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)}{n}, \tag{3.15}$$

kde je značení stejné jako u předchozích funkcí.

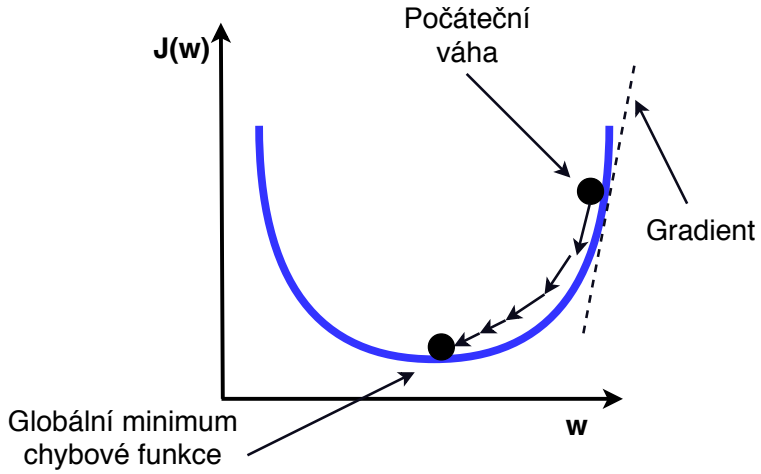
3.5 Optimalizační algoritmy

Po vypočítání chyby a jejím zpětném šíření v rámci algoritmu zpětného šíření chceme modifikovat parametry sítě. K tomuto procesu používáme optimalizační algoritmy, které se snaží chybu minimalizovat (v ideálním případě najít její globální minimum) a upravit parametry sítě na základě nalezeného minima. Optimalizační algoritmus využívá hodnotu gradientu (vektor parciální derivace chybové funkce podle vah) a mění váhy v opačném směru gradientu (pokud bychom použili původní směr chybu bychom naopak maximalizovali).

Následující sekce bude věnována několika základním optimalizačním algoritmům, opět je vhodné zmínit, že těchto algoritmů existuje celá řada. Krom níže uvedených například Momentum a Nestorov Momentum [48], RMSprop [25], Adagrad [12], Adamax [33] nebo Nadam [11].

3.5.1 Stochastický gradientní sestup

Stochastický gradientní sestup (angl. *stochastic gradient descent*) je jeden z původních optimalizačních algoritmů používaných pro učení umělých neuronových sítí. V dnešní době používáme modifikaci tohoto algoritmu zvanou *mini-batch gradientní sestup*, jež se liší od stochastického gradientního sestupu v počtu trénovacích dat, která jsou v jednom okamžiku trénována. Stochastický gradientní sestup v jeden okamžik přizpůsobuje parametry pro



Obrázek 3.9: Minimalizace chybové funkce změnou vah v opačném směru gradientu pomocí optimalizační funkce [3]

jednu položku z trénovací sady, mini-batch gradientní sestup naopak pro skupinu položek z trénovací sady (v takové skupině by měl být vyvážený poměr prvků z jednotlivých tříd). Výhodou mini-batch gradientního sestupu je především rychlost výpočtu. Posledním typem je pak *batch gradientní sestup*, který modifikuje váhy pro všechna trénovací data najednou.

$$w_i = w_{i-1} - \alpha \frac{dE}{g_i}, \tag{3.16}$$

kde w_i jsou parametry vrstvy i, α je koeficient učení (angl. *learning rate*) a g_i je gradient popsaný jako $\frac{dE}{dw_i}$, tedy parciální derivace chybové funkce podle vah této vrstvy.

3.5.2 Adadelta

Adadelta je rozšířením algoritmu zvaného Adagrad, který využívá adaptivní učicí koeficient [56]. Jedná se o jednu z nejpoužívanějších optimalizačních technik společně s metodou Adam. Při pohledu na obrázek 3.9 vidíme, že čím blíže jsme minimu, tím menší kroky bychom měli dělat, abychom minimum neminuli. Problém byl však v tom, že se mohlo stát, že koeficient učení se přiblížil k nule, ale stále se nepodařilo dosáhnout minima. Tento problém řeší právě algoritmus Adadelta. Předpis funkce pro metodu Adadelta je následující:

$$w_i = w_{i-1} - \frac{\alpha}{\sqrt{E[g^2]_i + \epsilon}} g_i, \tag{3.17}$$

kde w_i jsou parametry vrstvy i, α je koeficient učení a g_i je gradient. V rovnici se dále nachází konstanta ϵ a klouzavý průměr (angl. *moving average*) $E[g^2]_i$ z předchozích gradientů, který je vyjádřen takto:

$$E[g^2]_i = \gamma E[g^2]_{i-1} + (1 - \gamma g_i^2), \tag{3.18}$$

kde $E[g^2]_{i-1}$ je předchozí klouzavý průměr a dále je zde konstanta γ. Tato technika je velmi blízká algoritmu zvanému Momentum, který využívá předchozího gradientu – zjednodušeně, pokud odbrzdíme auto na kopci a z kopce ho spustíme, nezastaví se ihned v údolí, ale setrvačností vyjede do stoupající části údolí a poté se začne vracet zpět. Tohoto chování můžeme využít pro rychlejší konvergenci a zároveň únik z lokálních minim. Zároveň však může dojít k překonání globálního minima, tento problém řeší algoritmus Nestorov Momentum.

3.5.3 Adam

Optimalizační algoritmus Adam má navíc adaptivní moment oproti algoritmu Adadelta [33]. Jedná se o nejrozšířenější optimalizační techniku. Pokud bychom Momentum přirovnali k valící se kouli, Adam bude těžká valící se koule s třením, nemělo by tedy docházet k minutí globálního minima [45]. Metoda má následující předpis:

$$w_i = w_{i-1} - \hat{m}_i \frac{\alpha}{\sqrt{\hat{v}_i} + \epsilon}, \tag{3.19}$$

kde proměnné w_i, w_{i-1}, α a ϵ mají stejný význam jako v algoritmu Adadelta. Pro proměnné $\hat{v}_i$ a $\hat{m}_i$ platí předpis:

$$\hat{m}_i = \frac{m_i}{1 - \beta_1^i}, \hat{v}_i = \frac{v_i}{1 - \beta_2^i}, \tag{3.20}$$

kde $\hat{m}_i$ je zanikající průměr předchozích gradientů, m_i s konstantou β_1 vyjádřený rovnicí 3.21, $\hat{v}_i$ je necentrovaný rozptyl (angl. *uncentered variance*) s konstantou β_2 (rovnice 3.22).

$$m_i = \beta_1 m_{i-1} + (1 - \beta_1) g_i \tag{3.21}$$

$$v_i = \beta_2 v_{i-1} + (1 - \beta_2) g_i^2 \tag{3.22}$$

Hodnoty m_i a v_i jsou na počátku inicializovány na 0. Doporučené počáteční hodnoty pro použité konstanty jsou 0.9 pro β_1, 0.999 pro β_2 a 10^{-8} pro ϵ.

3.6 Metody inicializace vah

Inicializace vah hraje výraznou roli v procesu trénování hlubokých neuronových sítí. Důvodem je především stále vzrůstající počet parametrů u nejnovějších architektur. Hledání rozhodovací hranice v průběhu učení je tak stále náročnější. Různé metody inicializace vah nám poskytují další prostředek k urychlení konvergence dané sítě anebo alespoň nějaké konvergence.

Pokud bychom nechali váhy nastavené na 0, síť se nikdy nenaučí, protože aktualizace vah by byla pro každý vstup stejná. Účelem prvotního nastavení je právě narušení symetrie, umožníme tak proces učení. Zároveň by váhy neměly být nikterak velké vzhledem k jejich náhodné inicializaci – při velké výstupní chybě, bude velký i gradient a zároveň změna vah.

Původní metodou používanou pro inicializaci vah je rovnoměrné rozdělení. V článku [35] bylo použito rovnoměrné rozdělení v rozsahu mezi $\frac{-2.4}{F}$ do $\frac{2.4}{F}$, kde F je součet vstupů s výstupy do konkrétní vrstvy pro kterou jsou váhy počítány (rozsah byl použit s ohledem na aktivační funkci sigmoida).

S rostoucí hloubkou jednotlivých sítí se zvýšil zájem o výzkum v oblasti urychlení konvergence hlubokých neuronových sítí. V roce 2010 byl publikován článek s novým přístupem zvaným Xavierova inicializace, který zároveň popsal problém *saturace neuronu* [18]. Tato metoda umožňuje inicializovat váhy takovým způsobem, že k saturaci dochází v omezené míře, a my můžeme trénovat velmi hluboké neuronové sítě.

Definice 3.10. *Saturace neuronu* je stav, jež nastává u aktivačních funkcí, které mají v určité části velmi nízkou hodnotu gradientu, změna vah potom probíhá velmi pomalu a je problematické se z tohoto stavu vůbec dostat. Aktivační funkce náchylné k saturaci neuronů jsou například sigmoida nebo tanh. Funkce jako ReLU je naopak proti saturaci odolná.

V souvislosti s Xavierovou inicializací se později ukázalo, že tento přístup není příliš vhodný pro aktivační funkce, u kterých k saturaci nedochází (především ReLU). V roce 2015 tak byla představena metoda He inicializace, jež tento problém řeší [19].

Velmi používanou metodou je inicializace pomocí metod učení bez učitele [14] nebo využití už natrénované sítě pro část, kde detekujeme příznaky a následně doučíme plně propojené vrstvy pro potřeby klasifikace (angl. *transfer learning*) [37] [17].

3.7 Metody regularizace

Metody regularizace nám pomáhají co nejvíce zamezit přetrénování (přeučení) sítě (definice 3.2). Obvykle použitím regularizačních metod zvýšíme chybu na trénovacích datech, ale zároveň zvýšíme úspěšnost a snížíme chybu na datech testovacích. V této kapitole se budeme věnovat konkrétním regularizačním metodám.

3.7.1 Dropout

Dropout je nejrozšířenější regularizační technikou v hlubokých neuronových sítích [47]. Vychází z jednoduché myšlenky, kdy v průběhu trénování sítě náhodně deaktivujeme některé neurony (nastavíme jejich aktivaci na 0). Výsledkem je robustněji natrénovaná síť (vypnutím neuronů donutíme síť šířit chybu jinou cestou), která je méně náchylná k přetrénování. Při použití dropoutu definujeme hodnotu, jež značí, kolik procent neuronů bude deaktivováno. Na základě toho vytvoříme binární masku (neurony, jež mají být vypnuté, budou mít váhu 0), kterou aplikujeme pomocí Hadamardova součin na vstup v rámci dopředného kroku:

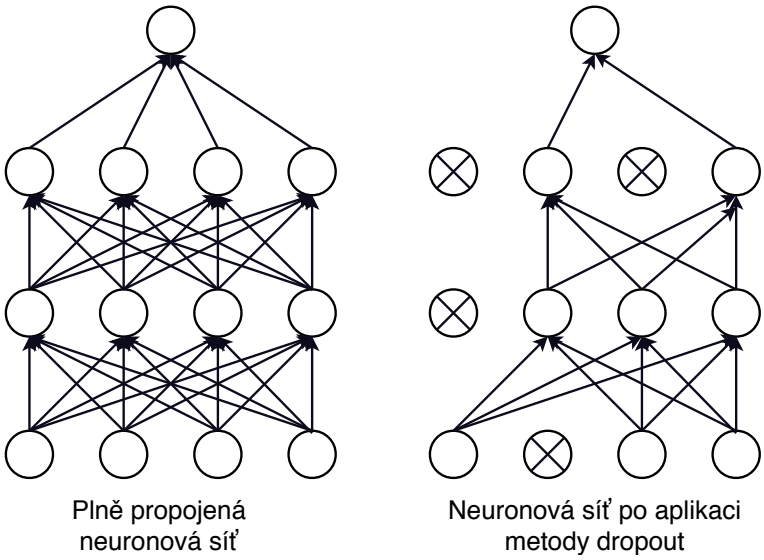
```
U = mask(p, size=hidden_layer.shape) / p
hidden_layer *= U
```

Kde p je pravděpodobnost vypnutí neuronu v dané vrstvě, *mask* je funkce, která vrátí binární masku na základě předané pravděpodobnosti a tvaru skryté vrstvy *hidden_layer.shape*. Dropout používáme pouze při trénování, proto je nutné binární masku škálovat (dělení hodnotou p).

Při zpětném kroku šíříme gradient pouze přes zapnuté neurony, toho jsme schopni docílit opět Hadamardovým součinem s maticí U:

```
derivation_hidden_layer *= U
```

Dropout v této podobě používáme před lineární vrstvou, kde jsou data vždy ve 2D. Před konvoluční vrstvou se dropout využívá jen minimálně. Popisujeme dva typy: buď vypínáme náhodně konkrétní pixely, anebo celé příznakové mapy.



Obrázek 3.10: Vizualizace sítě při použití metody dropout [47]

3.7.2 L2 regularizace

L2 regularizace je metoda založená na přičtení regularizačního členu přímo k chybové funkci. Tato regularizace je aplikována v rámci každé vrstvy zvlášť a na závěr je regularizační člen z každé vrstvy, která L2 regularizaci využívá, přičten přímo k chybové funkci. Obecný vzorec pro přičtení regularizačního členu:

$$E = E_{orig} + \frac{\alpha}{2} \sum w^2, \tag{3.23}$$

kde označujme jako E_{orig} původní chybovou funkci (detailněji kapitola 3.4), jako α značíme regularizační konstantu. Konec vzorce tvoří suma přes všechny parametry sítě.

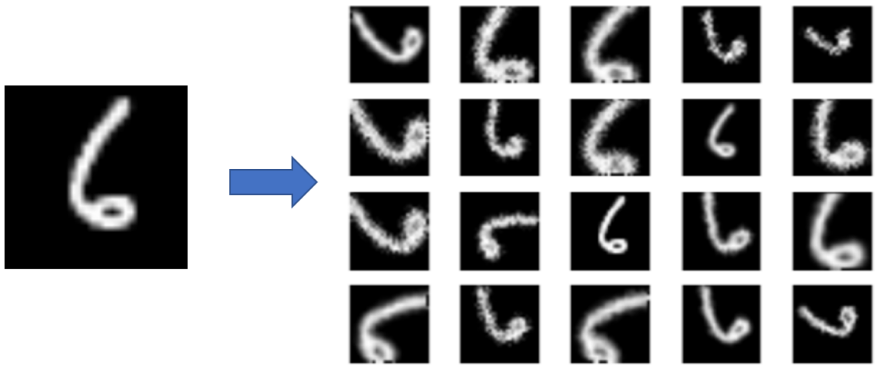
3.7.3 Augmentace dat

Další velmi rozšířenou regularizační technikou je doplnění dat (angl. *data augmentation*). Při použití této metody vytváříme část trénovacích dat uměle. Výhodou je, že tímto způsobem můžeme mnohonásobně zvětšit datovou sadu a omezit tím riziko přetrénování sítě. Umělá trénovací data můžeme vytvářet například pomocí těchto transformací:

- škálování,
- translace,

- rotace,
- přidání šumu,
- rozmazání,
- změna osvětlení.

Augmentaci používáme pouze za předpokladu, že se dané transformace blíží reálným datům. Pokud bychom detekovali součástky průmyslovou kamerou, která bude umístěna stále ve stejné vzdálenosti od snímaných kusů, nemá například škálování význam a mohlo by naopak úroveň detekce zhoršit.



Obrázek 3.11: Ukázka augmentace dat, která byla rotována a škálována pro potřeby zvětšení trénovací sady

3.8 Shrnutí základních vlastnosti

Konvoluční neuronové sítě mají několik charakteristických vlastností, některé jsou pozitivní a některé negativní. V této kapitole si je shrneme a na závěr uvedeme pravděpodobného nástupce tohoto druhu sítí.

3.8.1 Pozitivní vlastnosti

Mezi pozitivní vlastnosti řadíme *sdílené váhy*. To je vlastnost, jež nám umožňuje radikálně snížit počet parametrů sítě. Jinak řečeno, příznakové detektory (filtry) jsou lokální a každý je následně replikován na celý obrázek. Příklad: Vstupem CNN jsou obrázky o velikosti $200 \times 200 \times 3$. Pokud bychom použili vícevrstvou neuronovou síť, první skrytá vrstva by měla 120 000 spojení. Pokud použijeme CNN s konvolučním filtrem o velikosti 5×5 a dvaceti filtry, výsledkem bude $20 \times 5 \times 5 \times 3 + 20 = 1\,520$ spojení (výška filtru $\times$

šířka filtru × hloubka filtru + počet filtrů, tedy bias). Další vlastností je *vzrůstající úroveň abstrakce*, kdy používáme více vrstev pro možnost zachycení narůstající úrovně abstrakce obrázků. S tím souvisí *hloubka sítě*, která je typickým znakem konvolučních neuronových sítí, a umožňuje nám tak modelovat velmi složité problémy.

3.8.2 Negativní vlastnosti

Hlavní negativem konvolučních neuronových sítí v úlohách počítačového vidění je prokládání na sebe navazujících konvolučních vrstev pooling vrstvami. Pooling vrstvy přínášejí následující nevýhody [22]:

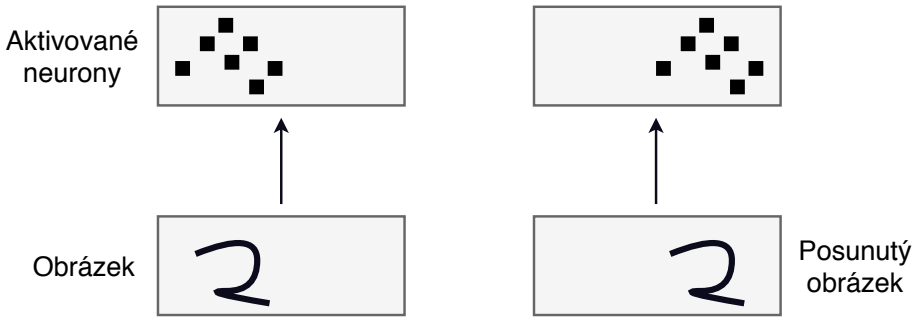
- *Rozpor s lidským vnímání předmětů* – člověk si s největší pravděpodobností konkrétní předmět, který vidí, v mozku vycentruje a zarovná, tak jak ho zná běžně. Abychom poznali písmeno, které je otočené o 90 stupňů mozek ho nejdříve vycentruje a pak teprve rozpozná, o jaké písmeno se jedná.

- *Ekvivariance* – jedná se o pojem, jenž je spojený s aktivovanými neurony CNN, při rozpoznávání stejného předmětu, který je na snímku posunut jinam. V ideálním případě bychom chtěli, aby aktivace neuronů byla vůči translaci invariantní, tedy aby se aktivovaly vždy stejné neurony (obrázek 3.12).

- *Úhel pohledu* – asi největším nedostatkem CNN je fakt, že pokud chceme objekt správně rozpoznat z více stran, musíme mít k dispozici trénovací data z různých úhlů pohledu. To automaticky vede k velkému počtu trénovacích dat, delšímu a složitějšímu trénování.

- *Routování* – jedná se o pojem, který přikládá jednotlivým spojení význam. V dnešních CNN tato vlastnost chybí, respektive ji zajišťuje obvykle max-pooling, který pouze vybere nejvíce aktivovaný neuron, což je ale velmi primitivní přístup. Může se tak stát, že síť rozpozná obličej, který má prohozené pozice očí, nosu a pusy.

V roce 2017 Geoffrey Hinton a jeho tým publikovali článek, ve kterém popsali nový typ neuronových sítí [27]. Tyto sítě pojmenovali jako *kapslové sítě* (angl. *capsule networks*). Tyto sítě se snaží eliminovat negativní vlastnosti CNN. Inspirací pro vznik metody byla jak myšlenka ještě více se přiblížit funkci lidského mozku, tak počítačová grafika. V aplikacích těchto sítí na standardní datové množiny se ukazuje, že jsou schopny překonat i nejlepší modely CNN a mohou tak v budoucnu konvoluční neuronové sítě vytlačit.



Obrázek 3.12: Ukázka aktivace neuronů při translaci objektu – ekvivariance

3.9 Architektury

Konvoluční neuronové sítě lze sestavit nespočetně mnoha způsoby, obvykle však používáme pro základní přístupy ověřené architektury, které se osvědčily na známých datových množinách. První známou CNN LeNet-5 už jsme si popsali v kapitole 3.1, nyní budeme pokračovat modernějšími a dnes běžně používanými architekturami.

3.9.1 AlexNet

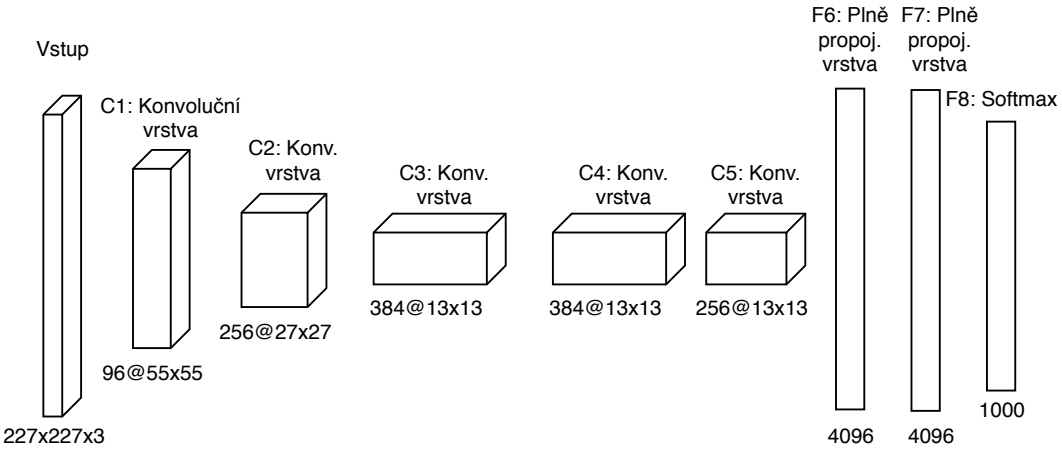
Konvoluční neuronová síť AlexNet velmi zpopularizovala použití CNN, když v roce 2012 výrazně překonala všechny ostatní přístupy v soutěži *ILSVRC challenge* [34]. Jednalo se o síť podobné architektury jako LeNet-5, ale hlubší a s větším počtem filtrů, zároveň zde bylo uspořádáno několik za sebou jdoucích konvolučních vrstev bez pooling vrstvy. V návaznosti na tuto síť zvítězila o rok později síť zvaná *ZF Net*, která byla modifikací AlexNet [57].

3.9.2 VGG

Síť VGG byla první z hlubokých konvolučních neuronových sítí, která používala menší konvoluční filtry, konkrétně o velikosti 3×3, které byly poskládány za sebe [46]. Ukázalo se, že sekvence těchto filtrů může efektivně nahradit filtr větší. Touto architekturou se inspirovaly sítě jako *Inception* anebo *ResNet*.

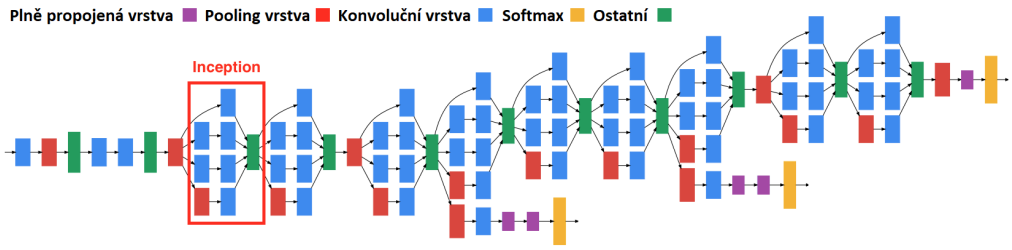
3.9.3 GoogleLeNet

GoogleLeNet je vítězná síť soutěže ILSVRC challenge z roku 2014, která nově obsahovala *inception modul*, jež umožňuje výrazné snížení parametrů sítě a zároveň využití různých konvolučních filtrů a pooling operací paralelně.



Obrázek 3.13: Architektura sítě AlexNet

Následovaných propojovací vrstvou, která jednotlivé příznaky spojila [49]. Na tento model navázaly další sítě s různými úpravami (*Inception*, *Inception-V2* [50], *Xception* [8] a nad ní vystavěná architektura *MobileNets* [28]).



Obrázek 3.14: Architektura sítě GoogleLeNet

3.9.4 ResNet

Architektura sítě ResNet přinesla novou myšlenku *rezidua*, což je spojení, které vede o více než jednu vrstvu kupředu, můžeme tedy spojit například třetí a osmou skrytou vrstvu jednoduchým sečtením výstupů vrstvy [19]. To umožnilo zmírnit problém zanikajícího gradientu a natrénovat tak velmi hluboké konvoluční neuronové sítě. Zároveň byl ResNet inspirací pro vznik nové architektury s názvem *DenseNet*, která provádí zřetězení po sobě jdoucích vrstev místo obvyklé sumy [29].

4 Existující knihovny konvolučních neuronových sítí

V této kapitole popíšeme několik rozšířených *frameworků* pro práci s konvolučními neuronovými sítěmi.

Definice 4.1. *Framework* je software, který je určen k ušetření času programátora a umožňuje mu využívat již implementované funkce a služby, bez toho, aby je sám implementoval. Obvykle se jedná o soubor knihoven, jež poskytují své aplikační rozhraní.

4.1 TensorFlow

Za nejrozšířenější framework pro tvorbu neuronových sítí, od klasické vícevrstvé neuronové sítě až po hluboké konvoluční nebo rekurentní neuronové sítě, lze považovat *TensorFlow*. Framework je velice flexibilní, protože pro definici architektury používá grafové rozhraní. *TensorFlow* podporuje programovací jazyk Python a také jazyk C++ nebo R.
Tento framework nabízí speciální služby jako například:

- *TensorBoard* – vizualizace trénovacího procesu, výsledné úspěšnosti, apod.
- *TensorFlow Serving* – nasazení nejlepšího modelu a údržbu rozsáhlých modelů,
- *TensorFlow Lite* – zjednodušené řešení *TensorFlow* určené pro mobilní zařízení.

Vydavatel	Google Brain Team
Domovská stránka	tensorflow.org
Licence	Apache 2.0
Poslední vydaná verze	1.7.0
Vydána roku	2015

4.2 Keras

Keras obaluje nízkoúrovňové frameworky *TensorFlow*, *CNTK*, *Theano* a *MXNet*. Vzhledem k tomu, že pro začátečníka je vcelku složité začít ihned používat grafové rozhraní a složité konstrukce, vznikl framework *Keras*, který konstrukci sítí značně zjednodušuje a urychluje. Zároveň však umožňuje využití služeb typu *TensorBoard* při použití nízkoúrovňové části od *TensorFlow*. Na několika řádcích kódu v programovacím jazyce Python tak můžeme realizovat hlubokou neuronovou síť. Další velkou výhodou je skvělá dokumentace tohoto frameworku.

Vydavatel	F. Chollet
Domovská stránka	keras.io
Licence	MIT
Poslední vydaná verze	2.1.4
Vydána roku	2015

4.3 PyTorch

PyTorch je implementací frameworku Torch pro programovací jazyk Python. Narozdíl od ostatních frameworků nepodporuje operační systém Windows. PyTorch se hodí především pro experimenty menšího rázu a výzkumné projekty.

Vydavatel	A. Paszke, S. Gross, S. Chintala, G. Chanan
Domovská stránka	pytorch.org
Licence	BSD
Poslední vydaná verze	0.3.1
Vydána roku	2016

4.4 Microsoft Cognitive Toolkit

Microsoft Cognitive Toolkit (CNTK) je vysoce optimalizovaný framework pro hluboké neuronové sítě, který podporuje programovací jazyk Python a C++. Největším problémem tohoto frameworku je nedostatečná dokumentace, jež začátečníkům velmi znesnadňuje práci.

Vydavatel	Microsoft Research
Domovská stránka	microsoft.com/en-us/cognitive-toolkit/
Licence	MIT
Poslední vydaná verze	2.3
Vydána roku	2016

4.5 Caffe

Caffe je jeden z nejstarších frameworků pro hluboké učení. Jedná se o C++ knihovnu s rozhraním pro Python, jež se používá převážně pro konvoluční neuronové sítě. Výhodou je komunita, v rámci které lidé sdílí své datové množiny, a často tak naleznou pomoc a vhodný přístup k danému problému. Zároveň existuje zjednodušená verze *Caffe2*, jež nabízí velkou flexibilitu a rozšiřitelnost.

Vydavatel	Berkeley Vision and Learning Center
Domovská stránka	caffe.berkeleyvision.org
Licence	BSD
Poslední vydaná verze	1.0
Vydána roku	2014

5 Formulace úlohy

Vzhledem k obecnému zadání bude v následující kapitole rozebrán cíl práce a nastíněn zvolený přístup pro implementaci společně s jeho krátkým odůvodněním.

5.1 Cíl práce

Cílem práce je vytvoření knihovny pro práci s hlubokými konvolučními neuronovými sítěmi v jazyce C#. Konkrétní požadavky na tuto knihovnu si nyní upřesníme.

- *Vstupem mohou být 2D anebo 1D data* – CNN obecně používáme v oblasti počítačového vidění. Můžeme ji však použít typicky všude, kde záleží na tom, jak data řadíme. Například tedy při zpracování textu, kdy je na vstup přiveden 1D vektor, jehož složky reprezentují jednotlivá slova.
- *Řešení musí být robustní a lehce rozšiřitelné* – knihovna musí být vhodně navržená, tak abychom mohli pokračovat v jejím rozšiřování.
- *Řešení musí obsahovat prostředky pro realizaci základních úloh učení s učitelem* – pomocí knihovny bychom měli být schopni vyřešit jak problém binární klasifikace, tak klasifikací do více tříd.
- *Řešení musí být řádně zdokumentované* – tak, aby knihovna mohla posloužit výukovým účelům.

5.2 Nástin řešení

Chceme-li dodržet požadavek na rozšiřitelnost a robustnost řešení, připadají pro implementaci v úvahu dva možné přístupy.

- *Definování konvoluční neuronové sítě pomocí sekvenčního modelu* – výhodou je především vysoká srozumitelnost a čitelnost. Tento model je obvykle základním modelem v každé knihovně konvolučních neuronových sítí.

- *Definice pomocí grafového modelu* – hlavní výhodou je možnost vytvořit velmi komplikované architektury konvolučních neuronových sítí, které nám sekvenční model modelovat neumožní.

Z výše zmíněných důvodů, požadavku na čitelnost a možnosti využít vytvořenou knihovnu jako výukový materiál, jsme zvolili implementaci sekvenčního modelu.

6 Implementace

V následující části dokumentu bude objasněn výběr jazyka. Dále bude popsána architektura knihovny a nejdůležitější metody, které umožňují uživateli konstrukci konvolučních neuronových sítí. Podrobnější dokumentace jednotlivých konstrukcí se nachází na webových stránkách projektu [39]. Zdrojové kódy se společně s experimenty a návodem k použití nachází na stránce `https://github.com/ mmedek/convsharp` pod licencí MIT.

6.1 Výběr implementace jazyka

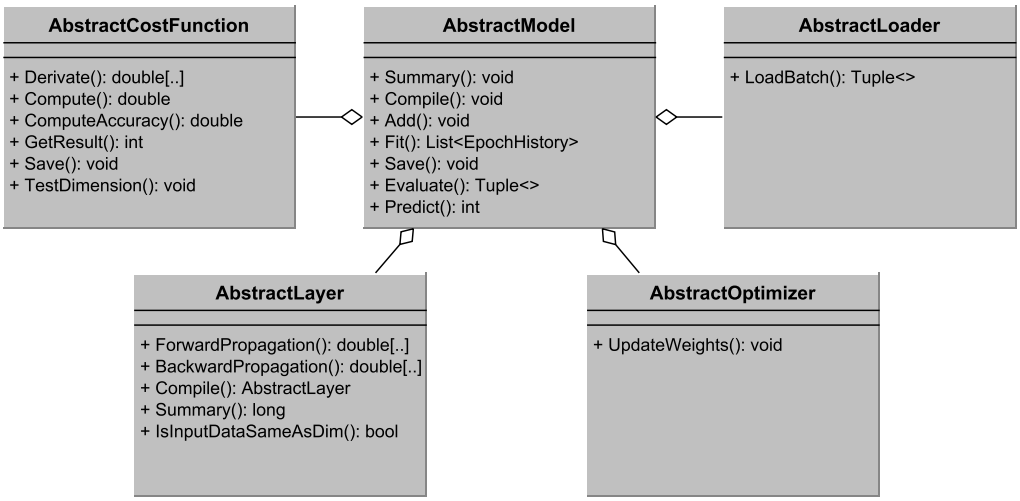
Jak již bylo zmíněno dříve, jako programovací jazyk byl zvolen C#, a to především kvůli absenci nativní knihovny pro tvorbu konvolučních neuronových sítí.

Jako cílová implementace jazyka byl zvolen *.NET CORE 2.0*, z důvodu multiplatformnosti. Na rozdíl od standardního *.NET Frameworku* může být kód vykonáván na strojích s operačním systémem Windows, MacOS nebo Linux. Obě implementace mají hodně společných vlastností a .NET Framework je výrazně větší. Výhodou .NET CORE 2.0 je lepší škálovatelnost a vyšší optimalizace, které vedou k vyšší výkonnosti při správném návrhu. Implementace je podporována společností Microsoft a jedná se o open source projekt pod licencí MIT a Apache 2.

6.2 Navržená knihovna

Návrh knihovny vychází ze sekvenčního modelu knihovny *Keras*, který je velmi intuitivní a na jehož základě můžeme pomocí objektově orientovaného návrhu knihovnu popsat a následně realizovat. Zároveň patří framework *Keras* mezi nejrozšířenější frameworky, a tak lze předpokládat, že pro řadu uživatelů bude práce s námi navrženou knihovnou intuitivní.

Hlavní část knihovny lze nejlépe popsat pomocí UML diagramu tříd zachyceného na obrázku 6.1. Třídy budou detailněji popsány v následující části, kde se budeme věnovat jejich základnímu účelu a obsaženým metodám.



Obrázek 6.1: UML diagram tříd tvořený abstraktními třídami tvořícími kostru knihovny

6.2.1 Model

Nejpodstatnější třídou celé knihovny je abstraktní třída `AbstractModel`, jež obsahuje abstraktní metody, které musí každý konkrétní model implementovat. Potomkem této třídy je třída `SequentialModel`, která je implementací sekvenčního modelu. Každý model slouží k sestavení, natrénování a rozpoznávání pomocí neuronové sítě. Třída `AbstractModel` obsahuje následující metody:

- `Add` – přidání nové vrstvy do modelu,
- `Compile` – kontrola parametrů přidaných vrstev, nastavení optimalizačního algoritmu a chybové funkce,
- `Fit` – trénování neuronové sítě na základě její definice v metodách `Add` a `Compile`,
- `Predict` – vyhodnocení jednoho vzorku pomocí tohoto modelu, vrací třídu rozpoznaného vzorku,
- `Evaluate` – vyhodnocení skupiny trénovacích nebo validačních dat, vrací úspěšnost správně rozpoznaných vzorků a také chybu rozpoznávání vypočítanou na základě chybové funkce,

- `Save` – binární serializace aktuálního stavu modelu do souboru na disku.

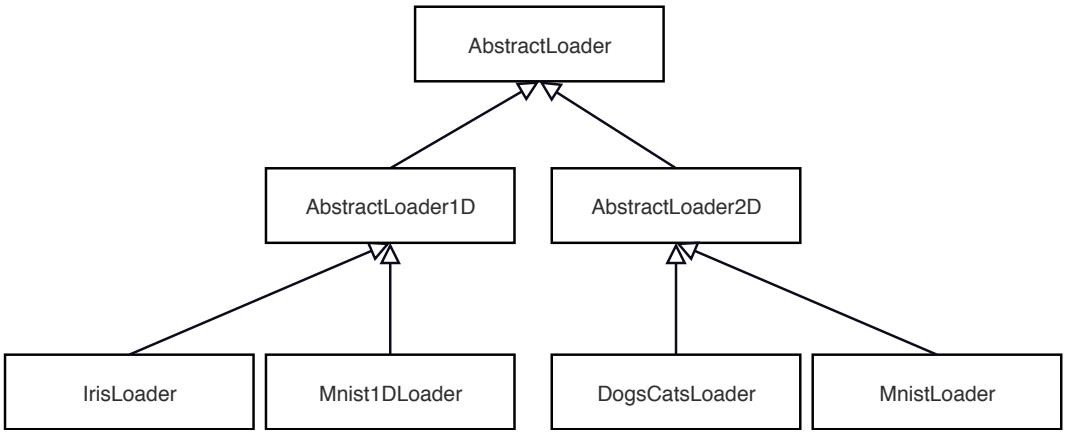
6.2.2 Načítání vstupních dat

Načítání vstupních dat probíhá pomocí třídy `AbstractLoader`, která byla implementována až v pozdní části práce z důvodů paměťové náročnosti větších datových množin. Vzhledem k tomu, že na běžných uživatelských strojích není možné načíst do paměti velké datové množiny jako celek bez zásahu, který by přesahoval rámec práce, načítáme data po menších blocích.

Třída `AbstractLoader` obsahuje pouze jednu abstraktní metodu `Load Batch`, která zajišťuje načtení bloku dat o definované velikosti dle předaného indexu.

Pro každou datovou množinu musíme implementovat jednoduchou třídu, která načte obrázek a převede ho po jednotlivých pixelech do 3D matice anebo 1D matice pro 1D data. Právě typ načítaných dat souvisí s existencí abstraktních potomků `AbstractLoader1D` pro načítání 1D dat a `Abstract Loader2D` pro načítání obrazových dat. Kód obou tříd je velmi podobný s tím rozdílem, že pracujeme s jiným typem dat při přepsání metody `LoadBatch` a definici abstraktní metody `Load`, jež načítá prvky dle jejich indexu.

Dále jsou implementovány třídy `MNISTLoader` jako potomek třídy `Abstract Loader2D` a `IrisLoader`, potomek třídy `AbstractLoader1D`. Tyto třídy slouží jako ukázkové případy využití.



Obrázek 6.2: Diagram dědičnosti části knihovny sloužící pro načítání vstupních dat vygenerovaný nástrojem Doxygen [21]

6.2.3 Tvorba architektury sítě

Architekturu sítě vytváříme přidáváním vrstev do modelu. Základním prvkem je abstraktní třída `AbstractLayer`, od které dědí všechny konkrétní implementace tříd. Třída obsahuje následující metody, z nichž jsou všechny až na `IsInputDataSameAsDim` abstraktní:

- `Compile` – ověření kompatibility vstupní dimenze s hyperparametry vrstvy a nastavení výstupní dimenze,
- `ForwardPropagation` – dopředné šíření,
- `BackwardPropagation` – zpětné šíření,
- `Summary` – vypsání informací o dimenzi vrstvy a počtu parametrů,
- `IsInputDataSameAsDim` – porovnání dimenze vstupních dat s dimenzí nastavenou v průběhu kompilace vrstvy.

Implementace zmíněných metod už je závislá na typu konkrétní vrstvy. Knihovna obsahuje několik druhů vrstev, z nichž každá má své využití a je implementována samostatnou třídou:

- `LinearLayer` – v průběhu dopředného kroku tato vrstva provede váhovanou sumu nad všemi vstupními daty a přičte bias,
- `ActivationLayer` – obsahuje implementaci konkrétních aktivačních funkcí, které aplikuje na data vstupující do této vrstvy a následně derivaci funkce na data v průběhu zpětného šíření,
- `Convolution1DLayer` – konvoluční vrstva vykonávající 1D konvoluci,
- `Convolution2DLayer` – konvoluční vrstva vykonávající 2D konvoluci,
- `MaxPooling1DLayer` – max-pooling vrstva vykonávající 1D max-pooling,
- `MaxPooling2DLayer` – max-pooling vrstva vykonávající 2D max-pooling,
- `AveragePooling2DLayer` – average-pooling vrstva vykonávající 2D average-pooling, pro 1D data nebyla vrstva implementována z důvodů menšího využití,
- `DropoutLayer` – vrstva, která slouží pro aplikaci regularizační techniky zvané Dropout,

- `FlattenLayer` – v rámci dopředného šíření tato vrstva 3D data transformuje na 1D data, v rámci zpětného šíření naopak 1D data transformuje na 3D.

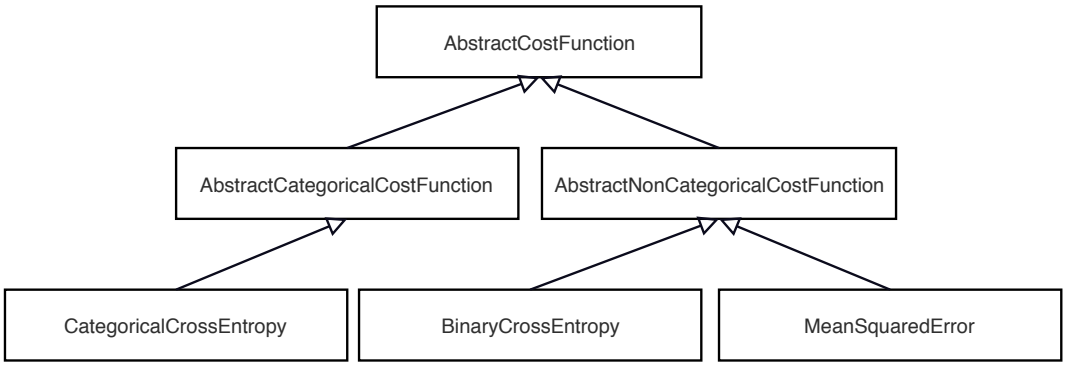
Pro snadnější implementaci učicího procesu bylo zavedeno rozhraní zvané `ILearnable` pro vrstvy, které mají váhy. Rozhraní poskytuje veřejné hodnoty vah a biasu vrstev, které rozhraní implementují. Dále pak lze pomocí tohoto rozhraní nastavit regularizační metodu.

6.2.4 Výběr chybové funkce

Vybranou chybovou funkci modelu nastavíme vytvořením nové instance chybové funkce v metodě `Compile`, kterou každý model implementuje. Každá chybová funkce musí dědit od abstraktní třídy `AbstractCostFunction`, jež má následující metody:

- `Compute` – vrátí velikost chyby na předaném vzorku,
- `Derivate` – zpětný krok v rámci chybové funkce,
- `ComputeAccuracy` – vypočítá úspěšnost na předaném vzorku, tato metoda je zařazena do třídy chybové funkce z důvodu větší modularity řešení,
- `GetResult` – vrátí index rozpoznané třídy dle výstupu neuronové sítě,
- `TestDimension` – otestuje dimenzi vstupních dat a předpokládaných tříd trénovacích vzorků.

Vzhledem k tomu, že výběr chybové funkce závisí na formátu vstupních dat označující jejich třídy, vznikly dvě další abstraktní třídy `AbstractCategoricalCostFunction` a `AbstractNonCategoricalCostFunction`, které řeší dva možné formáty označení vstupních dat: kategoriální označení nebo index třídy (podrobněji v kapitole 3.4).



Obrázek 6.3: Diagram dědičnosti části knihovny sloužící pro výběr chybové funkce vygenerovaný nástrojem Doxygen [21]

6.2.5 Optimalizační algoritmy

Optimalizační algoritmus se pro každý model nastavuje stejným způsobem jako cenová funkce, pomocí metody `Compile` nacházející se v používaném modelu. Každá konkrétní implementace optimalizačního algoritmu musí být potomkem abstraktní třídy `AbstractOptimizer`, která obsahuje proměnnou `learningRate` pro regulaci aktualizace vah v průběhu zpětného šíření. Dále pak obsahuje abstraktní metodu `UpdateWeights`, v jejímž rámci probíhá aktualizace pro jednotlivé vrstvy. Potomky třídy `AbstractOptimizer` jsou třídy:

- `SGD` – implementace mini-batch gradientního sestupu,
- `Adam` – Adam optimalizace.

6.2.6 Inicializace vah

Implementaci inicializačních metod lze rozdělit do dvou kategorií: inicializace vah a inicializace biasu.

Pro potřeby implementace inicializace vah slouží abstraktní třída `Abstract WeightInitializer` s abstraktní metodou `Initialize`, jejíž implementace vrací inicializované pole na základě dimenze předané jako vstupní parametr metody. Potomky této třídy jsou třídy:

- `NormalWeightInitializer` – inicializace pomocí normálního rozdělení,
- `XavierWeightInitializer` – inicializace pomocí Xavierovy metody.

Inicializace biasu je realizována pro všechny konkrétní metody díky abstraktní třídě `AbstractBiasInitializer`, která má také abstraktní metodu `Initialize`. Od této třídy dědí třída `ConstantBiasInitializer`, jež implementuje přiřazení konstantních hodnot jako původní hodnoty biasu.

6.2.7 Regularizační metody

Regularizační metody (kromě dropoutu realizovaného vrstvou) jsou závislé na rozhraní `ILearnable`, protože musí mít přístup k váhám jednotlivých vrstev. Jedinou implementovanou metodou je L2 regularizace, která je potomkem abstraktní třídy `AbstractRegularizer` s následujícími abstraktními metodami:

- `ComputeAdditionToCostFunc` – vypočítá hodnotu, jež reprezentuje zvětšení chyby v důsledku regularizace, a toto číslo vrátí,
- `Regularize` – aplikuje specifickou metodu regularizace.

6.3 Finální realizace načítání dat

Prvotní verze načítání dat počítala s načtením celé datové sady do paměti a jejím následným zpracováním. To se však ukázalo jako neúnosné při zpracování větších datových sad, kdy byly nároky na paměť příliš velké. Například při použití množiny 20 000 obrázků o velikosti 128×128 se třemi barevnými kanály si pouhé načtení do paměti vyžádá cca $7,5$ GiB, a to nebereme v potaz paměťové nároky knihovny při použití hluboké konvoluční sítě.

Závěrečná podoba implementace načítání dat se tak odvíjí od paměťových nároků a rozsahu diplomové práce. Načítání dat ve finální podobě probíhá po jednotlivých blocích dat definovaných uživatelem (batch). Data se načítají pokaždé, když jsou zapotřebí. To s sebou nese režii v průběhu opětovného načítání těchto dat, nedochází však k zahlcení paměti stroje, kde knihovna běží. Problémem je především stav, kdy začne mohutně docházet k výpadkům stránek kvůli odkládání na disk a program se výrazně zpomalí.

Ideálním řešením by byla podpora výpočtu na grafické kartě s ohledem na to, že jde většinou o maticové výpočty, a pomocí grafické karty bychom docílili největšího urychlení. Pak je ale problémem načítání dat na grafickou kartu, protože datové soubory použité pro trénování mohou být velké a často je tvoří velké obrázky (viz experiment s datovým souborem koček a psů). Batch je obvykle příliš malý pro to, aby se přesun dat na grafickou kartu vůbec vyplatil. Bylo by tedy nutné provést sérii pokusů a stanovit

vhodné velikosti skupin dat, které se vyplatí na grafickou kartu přenést. Taková optimalizace značně převyšuje rozsah diplomové práce, a proto nebyla zařazena.

7 Provedené experimenty

Pro ověření funkčnosti knihovny jsme vybrali několik jednoduchých experimentů, které pokryjí co možná nejvíce případů užití knihovny. Datové sady byly zvoleny na základě popularity a vhodnosti pro otestování jednotlivých funkcionalit. Výsledky jednotlivých experimentů budou porovnány s výstupem knihovny *Keras*, za použití *TensorFlow* a se snahou o sestavení co nejpodobnějších modelů. Při experimentech jsme se nesoustředili na porovnávání rychlostí knihovny *Keras* s implementovanou knihovnou, protože optimalizace rychlosti knihovny přesahuje rámec práce. Nutno však zmínit, že knihovna *Keras* je výrazně rychlejší.

7.1 MNIST

K velké části experimentů jsme zvolili datovou sadu MNIST skládající se z obrázků ručně psaných číslic nula až devět ve stupních šedi o rozměru 28×28 pixelů [36]. Trénovací množinu tvoří 60 000 obrázků a testovací množinu 10 000.

7.1.1 Vícekategoriální problém

Nejčastějším přístupem k rozpoznávání ručně psaných číslic z datového souboru MNIST je reprezentace vstupních vzorků pomocí 2D pole o velikosti $28 \times 28 \times 1$ (obrázek je ve stupních šedi, hloubka je tedy rovna 1). Třídy reprezentujeme kategoriálně, pro každý vzorek existuje vektor o velikosti počtu tříd s nastavenou hodnotou na indexu třídy na 1, ostatní hodnoty jsou nulové.

Vzhledem k tomu, že problém není nikterak složitý, použijeme jednoduchou konvoluční síť s následující architekturou:

1. konvoluční 2D vrstva – dvacet filtrů o velikosti 3×3, bez zarovnání nulami s krokem 1,
2. aktivační vrstva – ReLU aktivace,
3. max-pooling vrstva – velikost filtrů 2×2 s krokem 1,
4. flatten vrstva,
5. lineární vrstva – 64 neuronů,

6. aktivační vrstva – ReLU aktivace,
7. lineární vrstva – 10 neuronů,
8. aktivační vrstva – Softmax aktivace.

Jako chybovou funkci použijeme kategoriální cross-entropii a algoritmus Adam jako optimalizační metodu s konstantou učení nastavenou na $0,001$. Inicializace vah je prováděna pomocí Xavierova algoritmu a hodnoty biasu jsou nastaveny na 0. Pro trénování použijeme 1 000 a pro testování 100 vzorků. Batch má velikost 32 a učení bude probíhat v šesti epochách.

```
int batchSize = 32;
int epochCount = 6;

Dimension inputDim = new Dimension(batchSize, 1, 28, 28);

SequentialModel model = new SequentialModel();
model.Add(new Convolution2DLayer(inputDim, filterSize: 3,
    filterCount: 20, zeroPadding: false));
model.Add(new ActivationLayer(new Relu()));
model.Add(new MaxPooling2DLayer());
model.Add(new FlattenLayer());
model.Add(new LinearLayer(numNeurons: 64));
model.Add(new ActivationLayer(new Relu()));
model.Add(new LinearLayer(numNeurons: 10));
model.Add(new ActivationLayer(new Softmax()));

model.Compile(new CategoricalCrossEntropy(), new Adam(0.001d)
    );

MnistLoader loader = new MnistLoader(1000, 100, batchSize:
    batchSize);
List<EpochHistory> history = model.Fit(loader, epochCount:
    epochCount, useValidationSet: true);
```

Ukázka kódu 7.1: Definice architektury a trénování sítě pomocí implementované knihovny

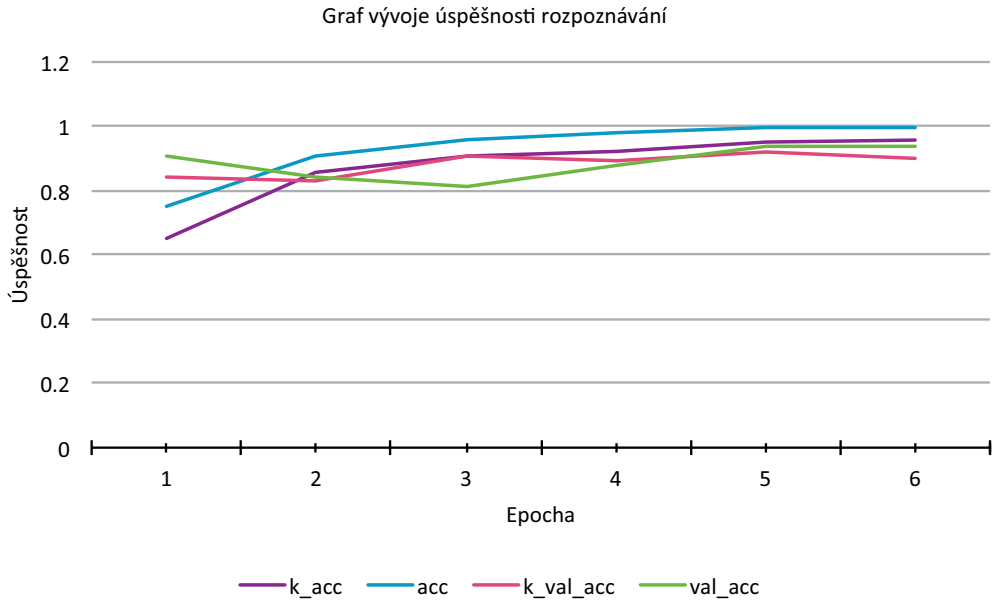
Pokud sestavíme identický model v knihovně *Keras* bude kód vypadat velmi podobně s rozdíly v syntaxi jazyka:

```
epoch_count = 6
curr_batch_size = 32

model = models.Sequential()
model.add(layers.Conv2D(20, (3, 3), input_shape=(28, 28, 1)))
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2, 2)))
```

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='adam', loss='
    categorical_crossentropy', metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=
    epoch_count, batch_size=curr_batch_size, validation_data=(
    test_images, test_labels))
```

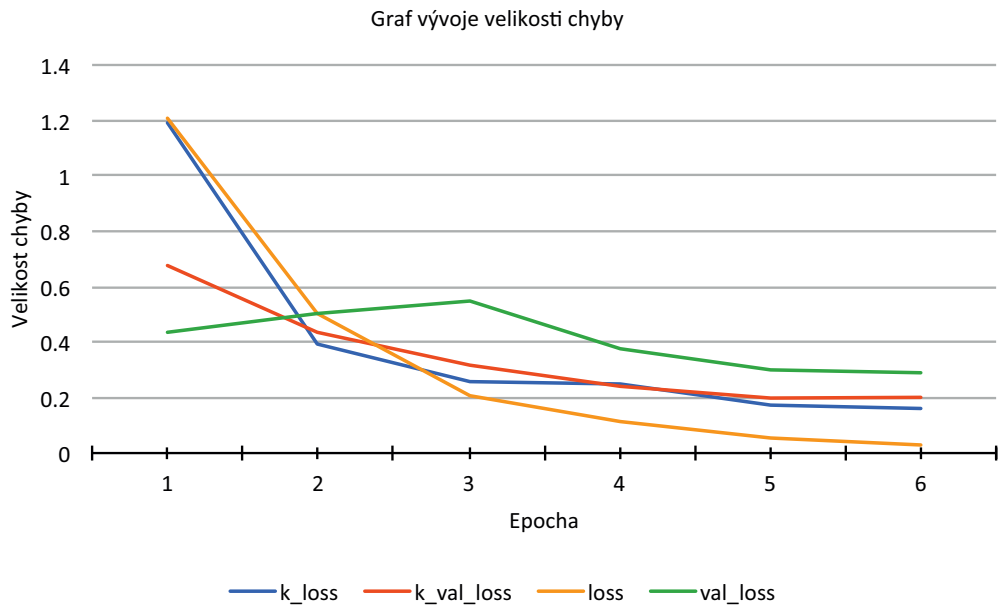
Ukázka kódu 7.2: Definice architektury a trénování sítě pomocí knihovny Keras

Graf na obrázku 7.1 znázorňující vývoj úspěšnosti obsahuje hodnoty z předchozích experimentů jak pro knihovnu *Keras*, tak pro knihovnu implementovanou v rámci této práce. V grafu vidíme vývoj úspěšnosti na trénovacích datech pro *Keras* značených jako `k_acc` a vývoj úspěšnosti na testovacích datech `k_val_acc`. Úspěšnost implementované knihovny na trénovacích datech značíme `acc` a na testovacích datech `val_acc`. Výsledné hodnoty tvoří průměr tří po sobě jdoucích měření. Zmíněné značení používáme i v následujících experimentech. Z grafu vidíme, že implementovaná knihovna dosahuje velmi podobných výsledků jako *Keras* jak na testovacích, tak na trénovacích datech. Oproti *Kerasu* dosahuje dokonce výsledků lepších o několik procent.



Obrázek 7.1: Graf vývoje úspěšnosti na datovém souboru MNIST

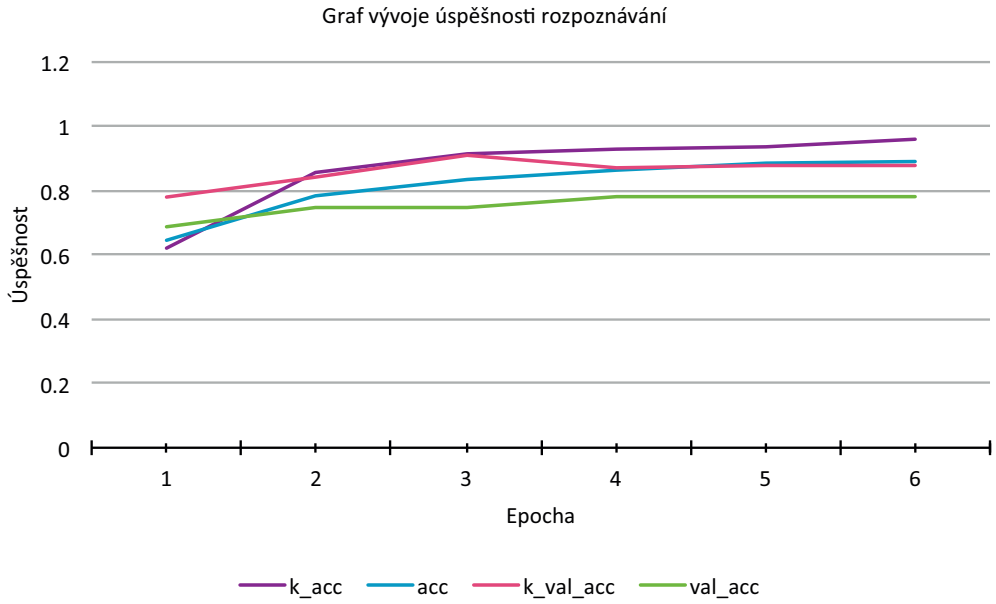
Na grafu vývoje chyby 7.1 vidíme, že chyba knihovny (`loss`) klesá rychleji než u frameworku *Keras* (`k_loss`). Vývoj chyby na trénovacích a testovacích datech je však u *Kerasu* stabilnější. Chyba na testovacích datech pro implementovanou knihovnu `val_loss` je větší než chyba `k_val_loss` naměřená u knihovny *Keras*. Stejnou legendu použijeme v dalších grafech vývoje velikosti chyby.



Obrázek 7.2: Graf vývoje velikosti chyby na datovém souboru MNIST

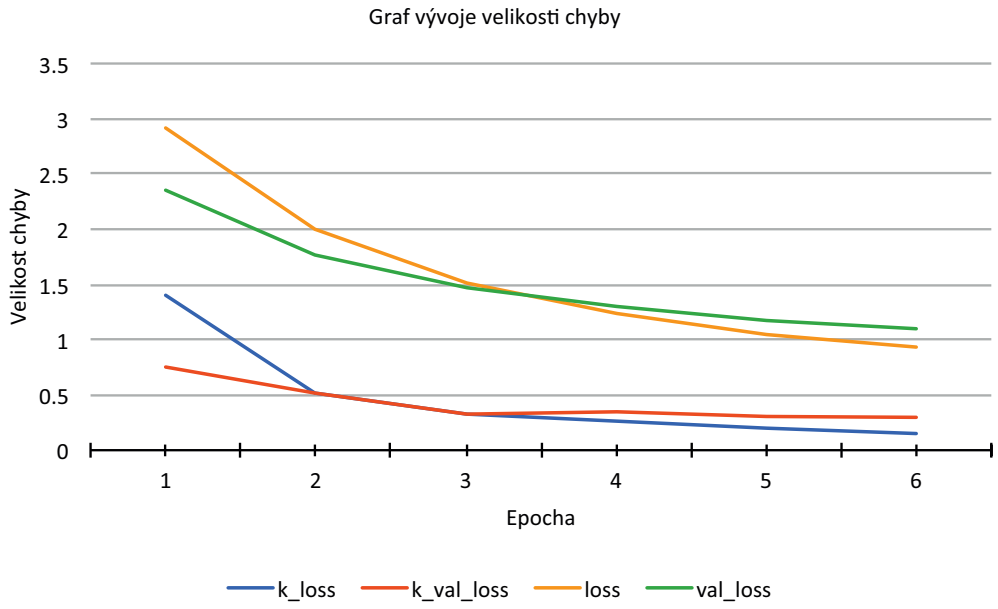
Experiment s podobnými výsledky a téměř identickou architekturou jsme provedli pro 1D vstupní data, kdy jsme jednotlivé pixely z obrázků poskládali za sebe a použili 1D konvoluci a 1D max-pooling pro potvrzení funkčnosti této implementace. Vzhledem k podobnosti experimentu nebude experiment blíže popisován, funkční kód lze najít v ukázkových příkladech na stránkách projektu [39].

Z grafu 7.3, který znázorňuje vývoj úspěšnosti na datovém souboru, kdy jsou na vstup přiváděná data v 1D formátu, lze vyčíst, že úspěšnost implementované knihovny na testovacích datech je v tomto případě nižší přibližně o 10 % a úspěšnost na trénovacích datech se blíží úspěšnosti na testovacích datech u knihovny *Keras*.



Obrázek 7.3: Graf vývoje úspěšnosti na datovém souboru MNIST při vstupních datech v 1D

Popsanému trendu horších výsledků implementované knihovny pak odpovídá i vývoj velikosti chyby v grafu na obrázku 7.4, kdy je chyba na datech naměřených implementovanou knihovnou větší po celou dobu trénování a validace dat. Průběhy jsou tentokrát velmi podobné s výjimkou velikosti chyby.



Obrázek 7.4: Graf vývoje velikosti chyby na datovém souboru MNIST při vstupních datech v 1D

7.1.2 Vícekategoriální problém s použitím MSE

Jednotlivé obrázky reprezentujeme opět jako 3D pole, ale informace o třídách předáváme hodnotou. Třídám reprezentujícím ručně psaná čísla 0 až 9 přiřadíme hodnotu 0 až 9, která bude reprezentovat jejich třídu. Tomu musíme přizpůsobit jak architekturu sítě, tak chybovou funkci:

1. konvoluční 2D vrstva – dvacet filtrů o velikosti 3×3, bez nulového zarovnání s krokem 1,
2. aktivační vrstva – ReLU aktivace,
3. max-pooling vrstva – velikost filtrů 2×2 s krokem 1,
4. flatten vrstva,
5. lineární vrstva – 64 neuronů,
6. aktivační vrstva – ReLU aktivace,
7. lineární vrstva – 1 neuron.

Jako chybovou funkci použijeme metodu Mean Square Error. Počet prvků v jedné zpracovávané skupině je stejný jako u předchozích úloh, tedy 32, a učení bude probíhat po dobu deseti epoch. Použijeme 1 000 trénovacích a 100 testovacích vzorků.

```
int batchSize = 32;
int currEpochCount = 10;

Dimension inputDim = new Dimension(batchSize, 1, 28, 28);

SequentialModel model = new SequentialModel();
model.Add(new Convolution2DLayer(inputDim, filterSize: 3,
    filterCount: 20, zeroPadding: false));
model.Add(new ActivationLayer(new Relu()));
model.Add(new MaxPooling2DLayer());
model.Add(new FlattenLayer());
model.Add(new LinearLayer(numNeurons: 64));
model.Add(new ActivationLayer(new Relu()));
model.Add(new LinearLayer(numNeurons: 1));

model.Compile(new MeanSquaredError(), new Adam(0.001d));

MnistLoader loader = new MnistLoader(1000, 100, batchSize:
    batchSize, categorical: false);
List<EpochHistory> history = model.Fit(loader, epochCount:
    currEpochCount, useValidationSet: true);
```

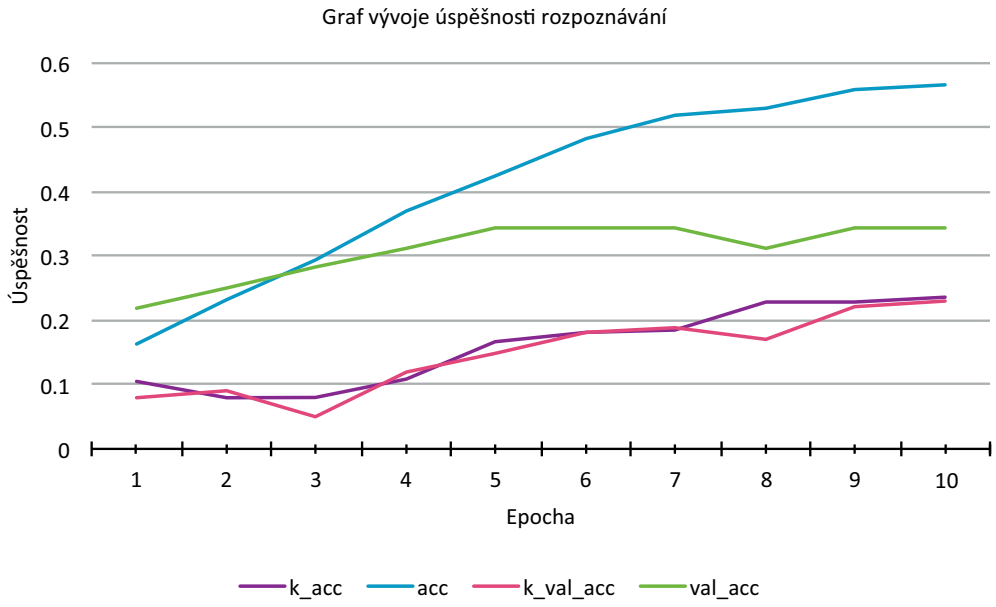
Ukázka kódu 7.3: Definice architektury a trénování sítě pomocí implementované knihovny pro třídy reprezentované hodnotou

Model v knihovně *Keras* bude opět téměř identický:

```
epoch_count = 10
curr_batch_size = 32

model = models.Sequential()
model.add(layers.Conv2D(20, (3, 3), input_shape=(28, 28, 1)))
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(1))
model.compile(optimizer='adam', loss='mse', metrics=['
    accuracy'])

history = model.fit(train_images, train_labels, epochs=
    epoch_count, batch_size=curr_batch_size, validation_data=(
    test_images, test_labels))
```

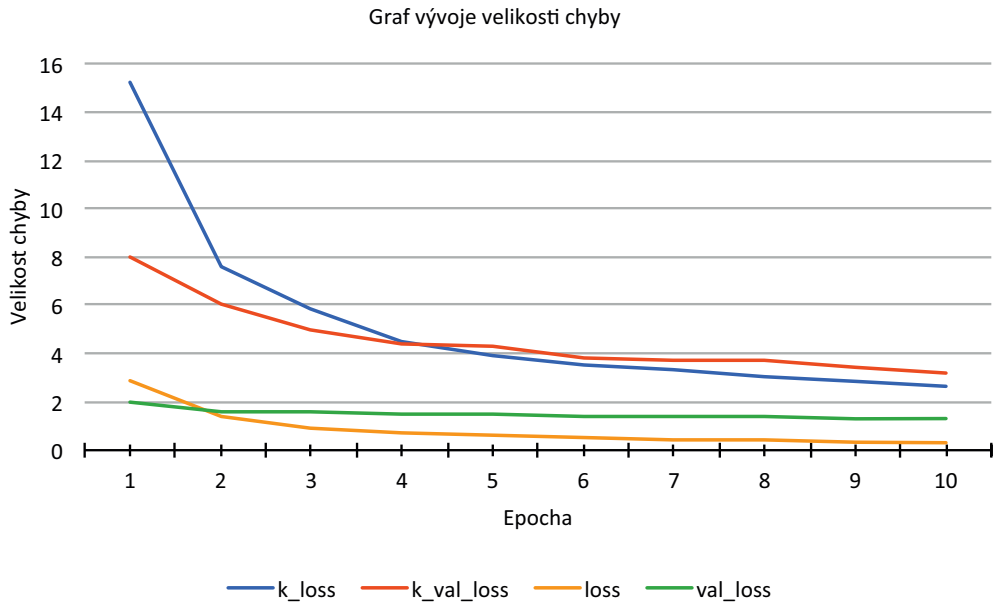
Ukázka kódu 7.4: Definice architektury a trénování sítě pomocí knihovny Keras pro třídy reprezentované hodnotou

Graf 7.10 zobrazuje vývoj úspěšnosti napříč epochami. V tomto případě jsou výsledky implementované knihovny o 10 % lepší na trénovací sadě a více než dvakrát lepší na testovací sadě. Vidíme také, že výsledky na testovací sadě jsou výrazně lepší než na sadě trénovací.



Obrázek 7.5: Graf vývoje velikosti chyby na datovém souboru MNIST při použití MSE

Vyšším hodnotám úspěšnosti implementované knihovny odpovídá i velikost chyby zanesená do grafu 7.10. Chyba na testovacích datech klesá pomaleji, ale po celou dobu. Klesání chybové funkce je plynulejší a stabilnější v případě knihovny *Keras*.



Obrázek 7.6: Graf vývoje velikosti chyby na datovém souboru MNIST při použití MSE

7.2 Dogs vs Cats

Tato datová množina obsahuje obrázky psů a koček. Úkol spočívá v rozpoznání těchto dvou tříd [13]. Jednotlivé obrázky mají rozměr 240×240 pixelů a jsou tvořeny třemi barevnými kanály. Trénovacích obrázků je 25 000 a testovacích 10 000, množina je vyvážená. Tuto množinu jsme použili pro demonstraci funkčnosti knihovny na větších a různorodých datech.

7.2.1 Binární problém

Obrázky reprezentujeme jako 3D pole a jednotlivé třídy značíme hodnotou (například 0 pro snímek kočky a 1 pro snímek psa). Jako chybovou funkci použijeme binární cross-entropii a pro optimalizaci použijeme metodu Adam s učicí konstantou nastavenou na $0,001$. K inicializaci vah je opět použit Xavierův algoritmus a hodnoty biasu jsou nastaveny na 0. Pro potřeby trénování je použito 1 000 a pro testování 100 vzorků. Velikost jednoho batche je 32 a trénování probíhalo 7 epoch. Počet epoch by obzvláště v tomto případě mohl být vyšší, ale pro příklad práce s knihovnou se příliš dlouhý experi-

ment zahrnutý do ukázek kódu nehodí, protože uživatel obvykle nechce čekat několik dní na výsledek prvního testu práce s knihovnou.

Architektura sítě je složitější než v ostatních experimentech, protože data jsou větší a složitější:

1. konvoluční 2D vrstva – 32 filtrů o velikosti 3×3, se zarovnáním nul s krokem 1,
2. aktivační vrstva – ReLU aktivace,
3. max-pooling vrstva – velikost filtrů 2×2 s krokem 1,
4. konvoluční 2D vrstva – 64 filtrů o velikosti 3×3, se zarovnáním nul s krokem 1,
5. aktivační vrstva – ReLU aktivace,
6. max-pooling vrstva – velikost filtrů 2×2 s krokem 1,
7. flatten vrstva,
8. lineární vrstva – 128 neuronů,
9. aktivační vrstva – ReLU aktivace,
10. lineární vrstva – 1 neuron.

Ukázka kódu, která je součástí knihovny vypadá následovně:

```
int batchSize = 32;
int currEpochCount = 7;

Dimension inputDim = new Dimension(batchSize, 3, 128, 128);

SequentialModel model = new SequentialModel();
model.Add(new Convolution2DLayer(inputDim, filterSize: 3,
    filterCount: 32, zeroPadding: true));
model.Add(new ActivationLayer(new Relu()));
model.Add(new MaxPooling2DLayer());
model.Add(new Convolution2DLayer(inputDim, filterSize: 3,
    filterCount: 64, zeroPadding: true));
model.Add(new ActivationLayer(new Relu()));
model.Add(new MaxPooling2DLayer());
model.Add(new FlattenLayer());
model.Add(new LinearLayer(numNeurons: 128));
model.Add(new ActivationLayer(new Relu()));
model.Add(new LinearLayer(numNeurons: 1));
model.Add(new ActivationLayer(new Sigmoid()));
```

```
18
19 model.Compile(new BinaryCrossEntropy(), new Adam(0.001d));
20
21 DogsCatsLoader loader = new DogsCatsLoader(1000, 100,
       batchSize: batchSize, trainPath: trainImagesAbsPath,
       testPath: testImagesAbsPath);
22 List<EpochHistory> history = model.Fit(loader, epochCount:
       currEpochCount, useValidationSet: true);
```

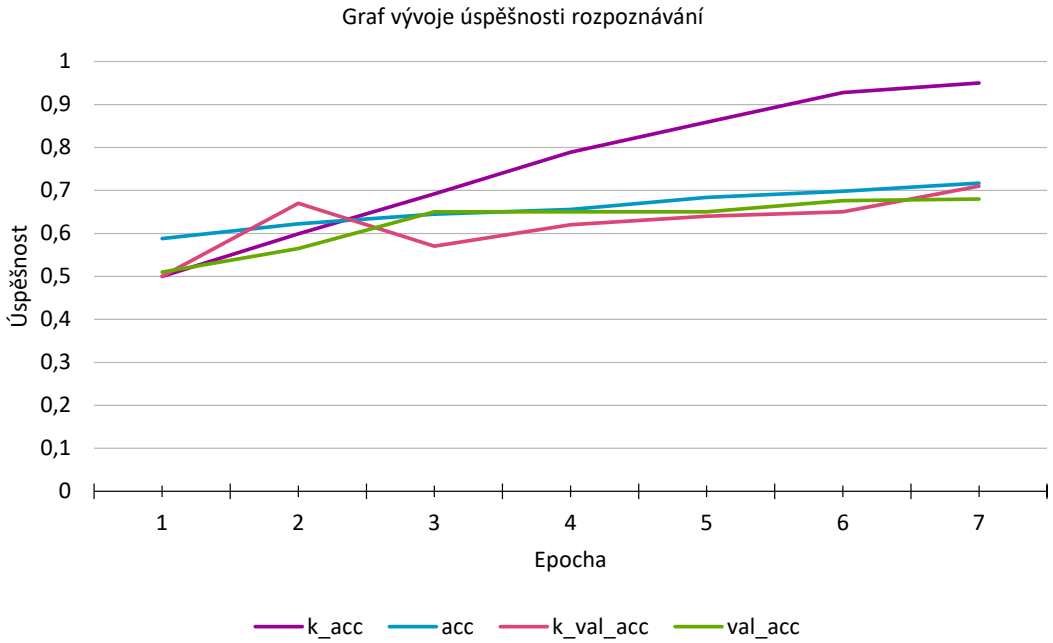
Ukázka kódu 7.5: Definice architektury a trénování sítě pomocí implementované knihovny pro binární problém na datovém setu Dogs vs Cats

Zdrojový kód psaný pomocí knihovny *Keras*:

```
1 epoch_count = 7
2 curr_batch_size = 32
3
4 model = models.Sequential()
5 model.add(layers.Conv2D(32, (3,3), activation='relu', padding
      ='same', input_shape=(128,128,3)))
6 model.add(layers.Activation('relu'))
7 model.add(layers.MaxPooling2D((2,2)))
8 model.add(layers.Conv2D(64, (3,3), activation='relu', padding
      ='same'))
9 model.add(layers.MaxPooling2D((2,2)))
10 model.add(layers.Flatten())
11 model.add(layers.Dense(128, activation='relu'))
12 model.add(layers.Dense(1, activation='sigmoid'))
13
14 model.compile(optimizer='adam', loss='binary_crossentropy',
      metrics=['accuracy'])
15
16 history = model.fit(train_images, train_labels, epochs=
      epoch_count, batch_size=curr_batch_size, validation_data=(
      test_images, test_labels))
```

Ukázka kódu 7.6: Definice architektury a trénování sítě pomocí knihovny Keras pro binární problém na datovém setu Dogs vs Cats

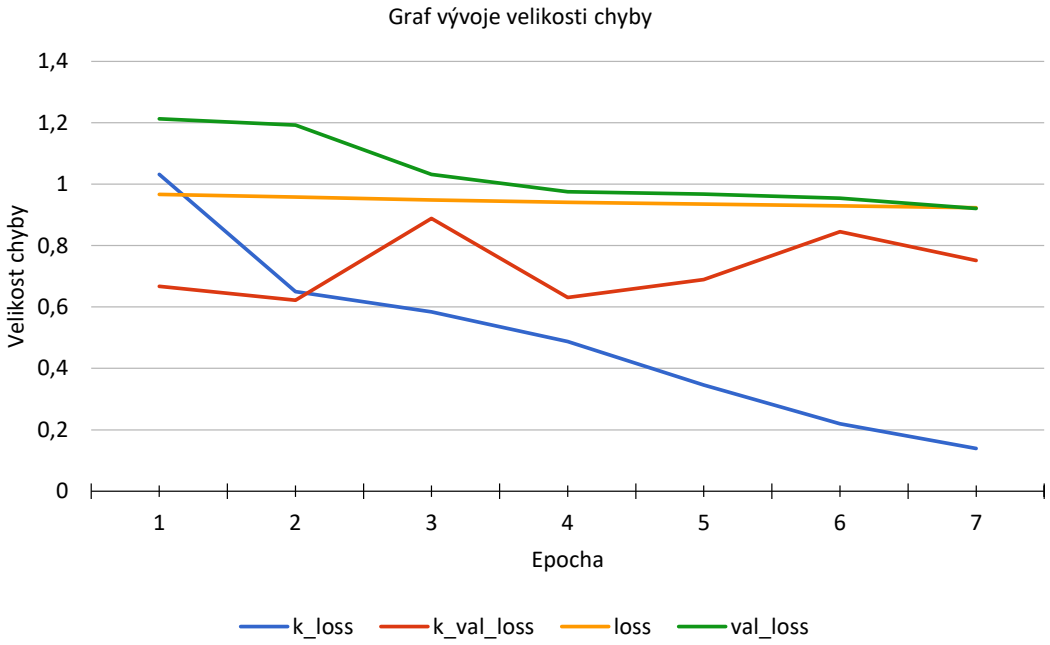
V grafu 7.7 zobrazujícím vývoj úspěšnosti vidíme, že úspěšnost implementované knihovny stoupá pomaleji jak pro trénovací, tak stejně i pro testovací datovou množinu. Vývoj hodnoty úspěšnosti pro knihovnu *Keras* se liší výsledky na trénovací a testovací množině o téměř dvacet procentních bodů.



Obrázek 7.7: Graf vývoje úspěšnosti na datové sadě Dogs vs Cats

Graf vývoje chyby 7.8 koresponduje s grafem úspěšnosti. Vidíme, že chyba implementované knihovny se snižuje pomaleji, ale stabilně. Chyba u knihovny *Keras* se snižuje rychle na trénovacích datech, ale nestabilně a pomaleji na testovacích datech.

V této úloze dochází k největším rozdílům mezi výsledky implementované knihovny a knihovny *Keras*. Doba běhu implementované knihovny zde několikanásobně převyšuje dobou běhu v *Kerasu*. Zajímavé je, že v této úloze se v průběhu učicího procesu chová implementovaná knihovna oproti ostatním experimentům stabilněji.



Obrázek 7.8: Graf vývoje velikosti chyby na datové sadě Dogs vs Cats

7.3 Iris

Jedná se o datovou množinu tvořenou příznaky třech různých druhů kosatců [10]. Každý vzorek je reprezentován délkou a výškou okvětního lístku a délkou a výškou kalichu. Datový soubor sestává ze 150 vzorků kosatce. Iris byl zařazen do experimentů z důvodu předvedení vícevrstvé neuronové sítě, nemusí se tedy nutně jednat o konvoluční neuronovou síť.

7.3.1 Vícekategoriální problém

Oproti ostatním experimentům, kdy byly na vstupu obrázky, používáme jako vstupní data příznaky reprezentující jednotlivé vzorky. Není proto důvod používat konvoluční vrstvu. Tři třídy, do kterých kosatce v datové množině dělíme, popíšeme kategoriálně a na základě toho zvolíme jako chybovou funkci kategoriální cross-entropii. Pro potřeby optimalizace použijeme me-

todu Adam s konstantou učení $0,001$. Pro trénování je použito 120 vzorků a pro testování 30. Jeden batch má velikost 16 a síť byla trénována po dobu dvanácti epoch. Struktura použité vícevrstvé sítě je následující:

1. flatten vrstva – předpokládáme použití knihovny primárně pro konvoluční neuronové sítě a lineární vrstva tak neobsahuje možnost nastavení vstupní dimenze, z tohoto důvodu musí být první vrstvou flatten vrstva,
2. lineární vrstva – 32 neuronů,
3. aktivační vrstva – ReLU aktivace,
4. lineární vrstva – 3 neurony,
5. aktivační vrstva – softmax aktivace.

Kód pro definici a učení sítě:

```
int batchSize = 16;
int currEpochCount = 12;

Dimension inputDim = new Dimension(batchSize, 1, 1, 4);

SequentialModel model = new SequentialModel();
model.Add(new FlattenLayer(inputDim));
model.Add(new LinearLayer(numNeurons: 32));
model.Add(new ActivationLayer(new Relu()));
model.Add(new LinearLayer(numNeurons: 3));
model.Add(new ActivationLayer(new Softmax()));
model.Compile(new CategoricalCrossEntropy(), new Adam(0.001d)
    );

IrisLoader loader = new IrisLoader(120, 30, batchSize:
    batchSize);
List<EpochHistory> history = model.Fit(loader, epochCount:
    currEpochCount, useValidationSet: true);
```

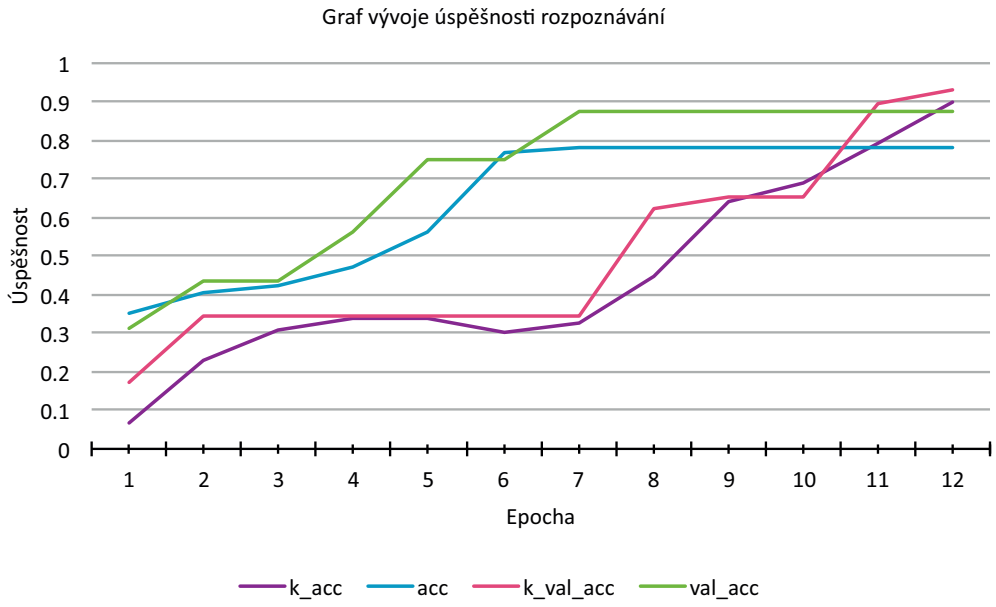
Ukázka kódu 7.7: Definice architektury a trénování sítě pomocí implementované knihovny pro datový set Iris

Kód se stejnou funkcí při použití knihovny *Keras*:

```
epoch_count = 12
curr_batch_size = 16

model = models.Sequential()
model.add(layers.Dense(32, activation='relu', input_shape
    =(4,)))
```

```
model.add(layers.Dense(3, activation='softmax'))
model.compile(optimizer='adam', loss='
    categorical_crossentropy', metrics=['accuracy'])

history = model.fit(train_images, train_labels, epochs=
    epoch_count, batch_size=curr_batch_size, validation_data=(
    test_images, test_labels))
```

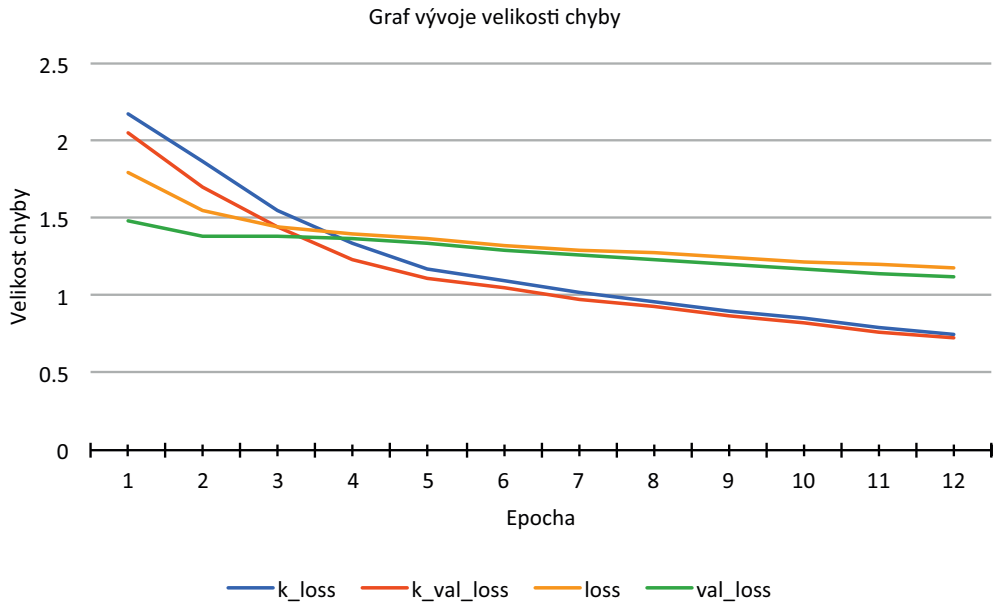
Ukázka kódu 7.8: Definice architektury a trénování sítě pomocí knihovny Keras pro datový set Iris



Obrázek 7.9: Graf vývoje úspěšnosti na datovém souboru Iris

V grafu 7.9 vidíme, že úspěšnost v obou případech měření pozvolna narůstá. Implementovaná knihovna dosahuje lepších výsledků v prvních epochách a v posledních šesti je konstantní. Úspěšnost architektury realizované pomocí knihovny *Keras* stoupá pomaleji, ale na testovací množině je v závěru trénování téměř o 10 % vyšší.

Vývoj chyby se pro obě implementace příliš neliší, pouze pro implementaci v *Kerasu* se chyba začíná po několika prvních iteracích snižovat rychleji, což je patrné z grafu 7.10.



Obrázek 7.10: Graf vývoje velikosti chyby na datovém souboru Iris

7.4 Jednotkové testy

Většina výpočetních operací je soustředěna do třídy `MatOp`. V případě chyby v některé z metod může dojít k tomu, že učicí proces nebude fungovat a sítě nebudeme schopni naučit. V takové chvíli je knihovna nepoužitelná. Abychom se takového stavu vyvarovali, byly v rámci této práce vytvořeny jednotkové testy. Microsoft Visual Studio podporuje vývoj těchto testů v odděleném projektu díky jmennému prostoru `Microsoft.VisualStudio.TestTool.UnitTesting`.

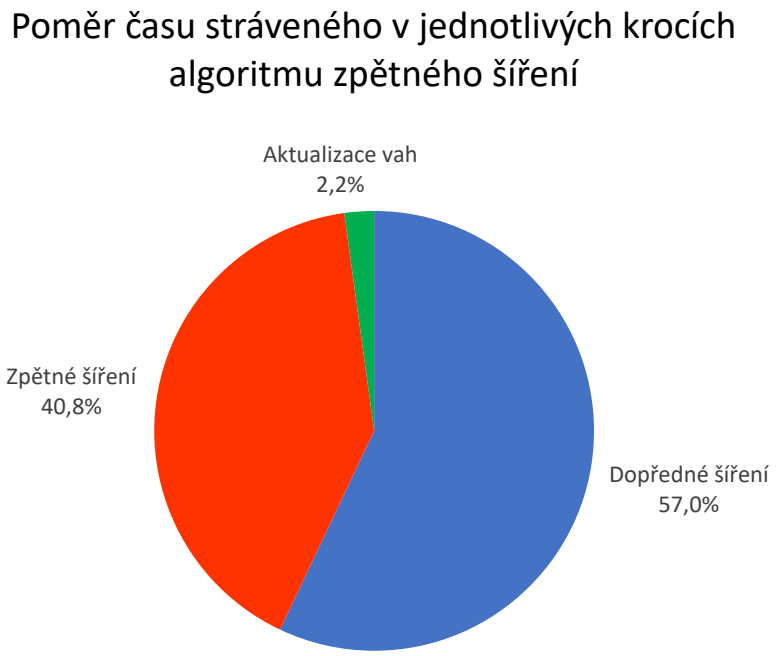
Všechny metody jsou tak otestovány, zda vrací správný výsledek na definovaných datech, a také ověřují, zda při předání nevalidních dat bude vyvolána výjimka.

7.5 Čas běhu pro konkrétní vrstvy

Pro budoucí optimalizace knihovny je užitečné zjistit, kolik času v průběhu učicího procesu strávíme v konkrétních vrstvách. Na základě toho poté můžeme optimalizovat kritická místa. Pro potřeby měření času byla použita stejná architektura sítě jako v experimentu s datovým souborem Dogs vs

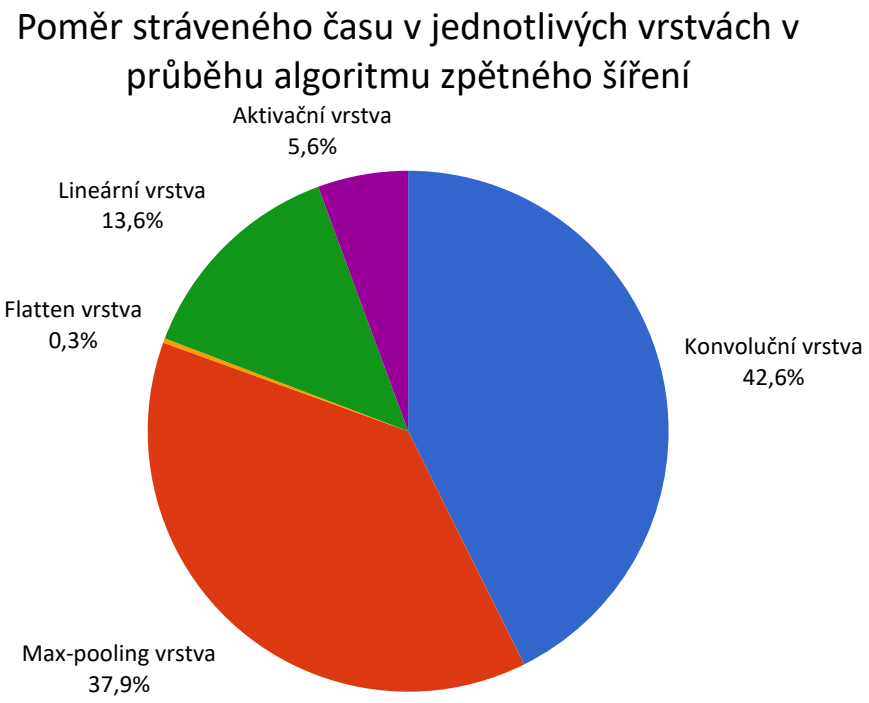
Cats v kapitole 7.2 a stejně tak byl shodný počet epoch. Na základě prací zabývajících se touto problematikou [51] můžeme předpokládat, že nejvíce stráveného času běhu na CPU by mělo náležet konvoluční vrstvě (v zmíněném článku to bylo téměř 50 % pro implementaci v knihovně Numpy [42] a 25 % zpětné šíření napříč konvoluční vrstvou).

V grafu 7.11 vidíme čas strávený v konkrétních krocích algoritmu zpětného šíření. Nejvíce času program vykonává kód v rámci dopředného šíření, očekávaným úzkým hrdlem je konvoluce v konvolučních vrstvách. Naopak zanedbatelně malou dobu program tráví v části pro aktualizaci vah a ve s ní spojeném prováděním optimalizačního algoritmu.



Obrázek 7.11: Graf času stráveného během vykonávání algoritmu zpětného šíření

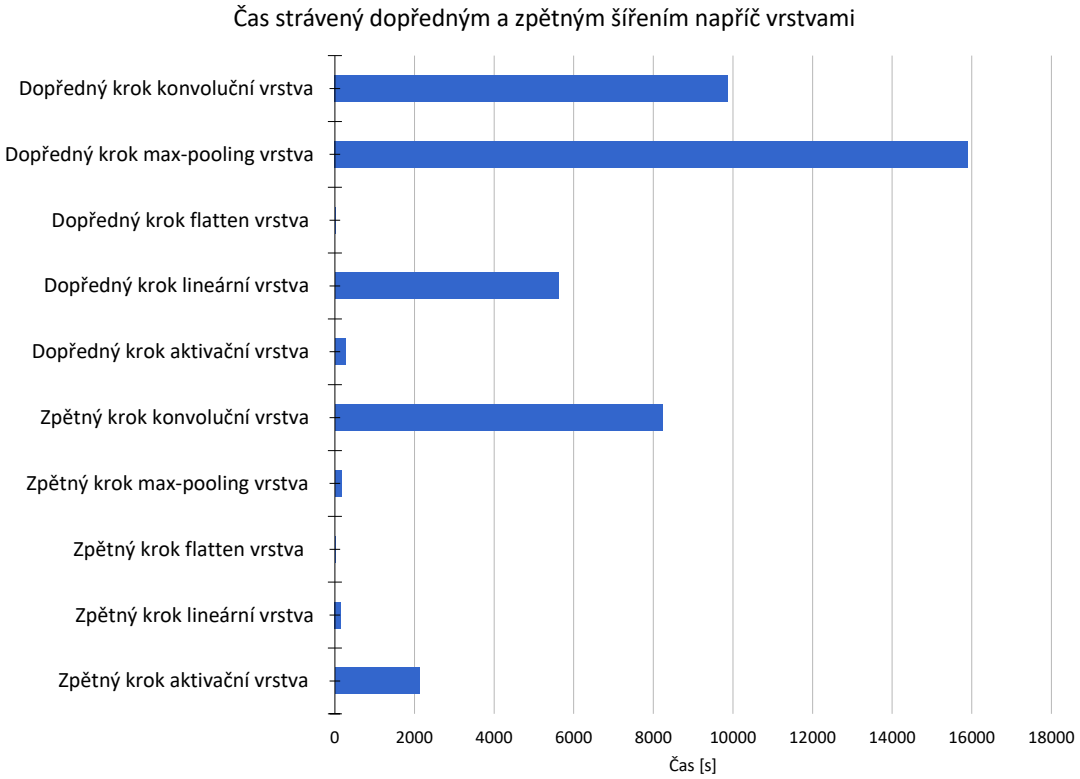
Detailnější pohled na problém stráveného času v průběhu učení sítě nám poskytuje graf 7.13, kde vidíme, že nejvíce času (42, 6 %) strávíme v konvoluční vrstvě, tak jak jsme předpokládali. Překvapivě v max-pooling vrstvě strávíme jen o 5 % méně. Ostatní výsledky korespondují s odkazovaným článkem v úvodu kapitoly.



Obrázek 7.12: Graf času stráveného během dopředného a zpětného šíření

Graf 7.13 prezentuje dobu strávenou v jednotlivých vrstvách rozdělenou na dopředné a zpětné šíření během sedmi epoch. Vidíme, že časově nejnáročnější jsou úlohy, které nejsou implementovány pomocí maticových operací (konvoluční vrstva, max-pooling operace). Zajímavým aspektem je příspěvek max-pooling vrstvy při dopředném šíření; v budoucnu by bylo vhodné se na to zaměřit a zkusit algoritmus zoptimalizovat.

Díky těmto vizualizacím můžeme potvrdit, že převod operací v konvoluční a max-pooling vrstvě na maticové operace a jejich následné zrychlení by mělo pozitivní dopad na rychlost konvolučních neuronových sítí realizovaných implementovanou knihovnou. Tyto vrstvy tvoří úzké hrdlo práce. Vzhledem k tomu, že se v těchto vrstvách stráví nejvíce času, urychlení by zde bylo nejvhodnější. Zároveň musíme zmínit, že z referenčního článku jsme čerpali jen část související s dobou vykonávanou na CPU. Pokud se bude algoritmus zpětného šíření provádět na GPU, očekávané časy budou jiné.



Obrázek 7.13: Graf času stráveného během dopředného a zpětného šíření

8 Návrhy na budoucí rozšíření

V této kapitole budou blíže popsány a prioritizovány možné kroky budoucího rozšíření knihovny. Návrhy na rozšíření jsou založeny především na zkušenostech s ostatními knihovnami a také na požadavku na rychlost, bez které se žádná podobná knihovna v praxi neobejde.

Výčet vhodných rozšíření dle priorit.

1. Zrychlení procesu učení – knihovna má většinu výpočetní logiky separovanou do třídy `MatOp`. Tuto logiku by bylo dobré v budoucnu optimalizovat. Vzhledem k aktuální realizaci načítání dat (načítáme menší bloky na základě velikosti batche) se nabízí optimalizace pomocí vektorových instrukcí. V budoucnu by bylo vhodné navrhnout takový způsob načítání, který by umožnil naplno využít grafickou kartu počítače, na kterém knihovna běží. Zároveň by bylo nutné přepsat šíření napříč konvoluční vrstvou na maticové operace pomocí metody *im2col* [6].

2. Přidání grafového modelu – implementace možnosti vytvářet grafové modely v rámci knihovny je nutnou podmínkou pro případné masové rozšíření. Dnešní nejlepší frameworky totiž mají převážně grafovou architekturu.

3. Přidání dalších typů vrstev – v nejnovějších architekturách CNN se běžně používá normalizační vrstva (angl. *batch normalization*), která normalizuje hodnoty z vrstvy předchozí. Dále není v knihovně implementována vrstva, která by podporovala 1D average-pooling nebo spojovací vrstva (angl. *merge layer*). Tyto a další typy vrstev by v budoucnu mohly být implementovány pro podpoření konceptu grafových modelů, kde se hojně užívají.

4. Implementace zpětného volání – funkcionalita, která by umožnila dynamicky reagovat na průběh trénovacího procesu. Například uložit nejlépe natrénovaný model a vyhnout se tak přetrénování, nebo reagovat na proces trénování dynamickou změnou učicí konstanty.

5. Obalovací třídy pro známé modely – velmi často používaným nástrojem knihoven pro konvoluční neuronové sítě je možnost načíst architekturu ověřené sítě a vyhnout se tak jejímu definování. S tím úzce souvisí možnost využití metody *transfer learningu*. Jedná se o funkci, kdy načteme již natrénovanou síť a můžeme s ní dále manipulovat, buď ji

ihned využít anebo přetrénovat pro naše účely. Tato metoda obvykle sníží dobu trénování neuronové sítě.

6. Zavedení monitorovacích prostředků – pokud uživatel potřebuje natrénovat hlubokou neuronovou síť, trénování může trvat i několik dní. V těchto případech se velmi hodí prostředky, které nám umožní monitorovat průběh učicího procesu tak, abychom mohli učení zastavit, pokud se chyba dlouhodobě nesnižuje anebo dochází k přetrénování či jiným problémům. Inspirací zde může být například nástroj *TensorBoard*, jejž nabízí knihovna *TensorFlow*.

7. Implementace vizualizací CNN – lze realizovat vizualizaci aktivací neuronů v jednotlivých vrstvách, hodnoty filtrů anebo míst, které jsou pro CNN nejpodstatnější při rozpoznávání.

8. Implementace dalších cenových funkcí, inicializačních metod vah a druhů regularizace.

9. Přidání dalších různorodých experimentů – čím více provedených experimentů s knihovnou, tím snazší bude pro nové uživatele začít s knihovnou pracovat.

9 Závěr

Práce seznamuje čtenáře s jednotlivými komponentami konvoluční neuronové sítě, popisuje proces učení a vysvětluje funkci komponent. Pochopení problematiky učicího procesu je klíčové, jelikož implementace knihovny z této myšlenky vychází.

V rámci práce vznikla knihovna, která umožní sestavení, natrénování a klasifikaci vstupních dat pomocí konvoluční neuronové sítě. Knihovna implementuje sekvenční model pro definici architektury sítí. V rámci tohoto modelu můžeme vybírat mezi různými typy vrstev jako je konvoluční, max-pooling, average-pooling, lineární nebo aktivační vrstva. Dále máme na výběr ze tří chybových funkcí a ze dvou metod pro nastavení počátečních vah. Proces učení probíhá pomocí optimalizačního algoritmu Adam anebo mini-batch stochastického gradientního sestupu. Zdrojový kód je volně k dispozici na platformě GitHub, kde se nachází také detailní dokumentace v anglickém jazyce.

Knihovna je doplněná o kód jednoduchých experimentů zvolených na základě ověření a demonstrace funkčnosti knihovny. Každý experiment popsaný v práci vždy obsahuje stručný popis použitého datového souboru, realizaci řešení v námi implementované knihovně a v knihovně *Keras* s porovnáním výsledků. Definici architektury sítě se podařilo navrhnout tak, aby se maximálně přibližovala knihovně *Keras*, která sloužila jako inspirace pro její uživatelskou přívětivost. Výsledky ukazují, že *Keras* dosahuje vyšší stability a v některých experimentech i lepších výsledků. Nejvýraznějším rozdílem je nízká rychlost implementované knihovny v porovnání s knihovnou *Keras*, jak ale práce popisuje, nejednalo se o prioritu vzhledem k rozsáhlé problematice optimalizací konvolučních neuronových sítí.

Práce obsahuje návrh budoucího rozšíření pro lepší konkurenceschopnost knihovny a její praktické využití na komplexnějších problémech. Kvůli předpokladu budoucího vývoje vznikla také sada jednotkových testů pro udržování funkčnosti knihovny.

Knihovna byla uvolněna ve formě NuGet balíčku doplněného o odkaz na webové stránky projektu na GitHubu, kde se nachází návod k použití, dokumentace a informace o možnosti zapojení se do vývoje knihovny. Za prvních 14 dní existence knihovny balíček použilo 55 uživatelů.

Literatura

[1] Convolutional Neural Network. 2016. Dostupné z: https://www.mathworks.com/discovery/convolutional-neural-network.html.

[2] New Navy Device Learns By Doing. 1958. Dostupné z: https://www.nytimes.com/1958/07/08/archives/new-navy-device-learns-by-doing-psychologist-shows-embryo-of.html.

[3] Argawal, A. Loss Functions and Optimization Algorithms. Demystified. 2017. Dostupné z: https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c.

[4] Bansal, R. What is XOR problem in neural networks? 2017. Dostupné z: https://www.quora.com/What-is-XOR-problem-in-neural-networks.

[5] Behnke, S. *Hierarchical Neural Networks for Image Interpretation.* Springer, 2003. ISBN 978-3-540-40722-5.

[6] Chellapilla, K. – Puri, S. – Simard, P. High Performance Convolutional Neural Networks for Document Processing. 2006. Dostupné z: https://hal.inria.fr/inria-00112631/document.

[7] Cholett, F. *Deep learning in Python.* Manning Publications, 2017. ISBN 978-1617294433.

[8] Chollet, F. Xception: Deep Learning with Depthwise Separable Convolutions. 2016. Dostupné z: https://arxiv.org/pdf/1610.02357.pdf.

[9] Deshpande, A. A Beginner's Guide To Understanding Convolutional Neural Networks. 2016. Dostupné z: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/.

[10] Dheeru, D. – Karra, E. UCI Machine Learning Repository. 2017. Dostupné z: http://archive.ics.uci.edu/ml.

[11] Dozat, T. Incorporating Nesterov Momentum into Adam. 2015. Dostupné z: http://cs229.stanford.edu/proj2015/054_report.pdf.

[12] Duchi, J. – Hazan, E. – Singer, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. 2011. Dostupné z: http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[13] Elson, J. et al. Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categorization. 2007. Dostupné z: https://www.kaggle.com/c/dogs-vs-cats/data.

[14] Erhan, D. et al. Why Does Unsupervised Pre-training Help Deep Learning? 2010. Dostupné z: http://jmlr.org/papers/volume11/erhan10a/erhan10a.pdf.

[15] Fairhead, H. The McCulloch-Pitts Neuron. 2014. Dostupné z: http://www.i-programmer.info/babbages-bag/325-mcculloch-pitts-neural-networks.html.

[16] Fukushima, K. Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. 1980. Dostupné z: http://www.cs.princeton.edu/courses/archive/spr08/cos598B/Readings/Fukushima1980.pdf.

[17] Gao, T. – Stark, M. – Kller, D. What Makes a Good Detector? – Structured Priors for Learning From Few Examples. 2011. Dostupné z: http://ai.stanford.edu/~tianshig/papers/transfer-ECCV2012.pdf.

[18] Glorot, X. – Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. 2010. Dostupné z: http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

[19] He, K. et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015. Dostupné z: https://arxiv.org/pdf/1502.01852.pdf.

[20] Hebb, D. *The Organization of Behavior*. JOHN WILEY AND SONS. INC., 1949. Dostupné z: http://pubman.mpdl.mpg.de/pubman/item/escidoc:2346268/component/escidoc:2346267/Hebb_1949_The_Organization_of_Behavior.pdf.

[21] Heesch, D. *Doxygen* [online]. Dostupné z: https://github.com/doxygen/doxygen.

[22] Hinton, G. *Geoffrey Hinton talk "What is wrong with convolutional neural nets ?"* [online]. Dostupné z: https://www.youtube.com/watch?v=rTawFwUvnLE.

[23] Hinton, G. – Rumelhart, D. E. – Williams, R. J. Learning representations by back-propagating errors. 1985. Dostupné z: http://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf.

[24] Hinton, G. – Rumelhart, D. E. – Williams, R. J. Learning internal representations by error propagation. 1986. Dostupné z: http://www.dtic.mil/dtic/tr/fulltext/u2/a164453.pdf.

[25] Hinton, G. – Srivastava, N. – Swersky, K. Overview of mini-batch gradient descent. 2012. Dostupné z: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[26] Hinton, G. – LeCun, Y. – Bengio, Y. Deep learning. 2015. Dostupné z: https://www.researchgate.net/publication/277411157_Deep_Learning.

[27] Hinton, G. – Sabour, S. – Frosst, N. Dynamic routing between capsules. 2017. Dostupné z: https://arxiv.org/pdf/1710.09829.pdf.

[28] Howard, A. et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. 2017. Dostupné z: https://arxiv.org/pdf/1704.04861.pdf.

[29] Huang, G. et al. Densely Connected Convolutional Networks. 2016. Dostupné z: https://arxiv.org/pdf/1608.06993.pdf.

[30] Hubel, D. – Wiesel, T. *Hubel and Wiesel - Cortical Neuron - V1* [online]. [cit. 2018/04/05]. Dostupné z: https://www.youtube.com/watch?v=8VdFf3egwfg.

[31] Joshi, P. How To Train A Neural Network In Python – Part II. 2016. Dostupné z: https://prateekvjoshi.com/2016/01/19/how-to-train-a-neural-network-in-python-part-ii/.

[32] Karpathy, A. Convolutional Neural Networks (CNNs / ConvNets). 2016. Dostupné z: http://cs231n.github.io/convolutional-networks/.

[33] Kingma, D. – Ba, J. Adam: A method for Stochastic Optimization. 2015. Dostupné z: https://arxiv.org/pdf/1412.6980v8.pdf.

[34] Krizhevsky, A. – Sutskever, I. – Hinton, G. ImageNet Classification with Deep Convolutional Neural Networks. 2012. Dostupné z: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[35] LeCun, Y. et al. Backpropagation Applied to Handwritten Zip Code Recognition. 1989. Dostupné z: http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf.

[36] LeCun, Y. et al. Gradient-based learning applied to document recognition. 1998. Dostupné z: http://yann.lecun.com/exdb/mnist/.

[37] Lim, J. – Salakhutdinov, R. – Torralba, A. ransfer Learning by Borrowing Examples for Multiclass Object Detection. 2011. Dostupné z: http://people.csail.mit.edu/lim/lst_nips2011/index.htm.

[38] McCulloch, W. S. – Pitts, W. A logical calculus of the ideas immanent in nervous activity. 1943. Dostupné z: http://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf.

[39] Medek, M. *ConvSharp* [online]. Dostupné z: https://mmedek.github.io/convsharp/index.html.

[40] Million, E. The Hadamard Product. 2007. Dostupné z: http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf.

[41] Minsky, M. – S., P. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969. Dostupné z: https://mitpress.mit.edu/books/perceptrons.

[42] Oliphant, T. *A guide to NumPy* [online]. [cit. 2018/04/05]. Dostupné z: http://www.numpy.org.

[43] Pettey, C. – Meulen, R. Gartner Says Global Artificial Intelligence Business Value to Reach 1.2 Trillion dolars in 2018. 2018. Dostupné z: https://www.gartner.com/newsroom/id/3872933.

[44] Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. 1957. Dostupné z: http://psycnet.apa.org/record/1959-09865-001.

[45] Ruder, S. An overview of gradient descent optimization algorithms. 2018. Dostupné z: https://arxiv.org/pdf/1609.04747.pdf.

[46] Simonyan, K. – Zisserman, A. Very deep convolutional networks for large-scale image recognition. 2015. Dostupné z: https://arxiv.org/pdf/1409.1556.pdf.

[47] Srivastava, N. et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. 2014. Dostupné z: https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf.

[48] Sutskever, I. Training Recurrent neural Networks. 2013. Dostupné z: http://www.cs.utoronto.ca/~ilya/pubs/ilya_sutskever_phd_thesis.pdf.

[49] Szegedy, C. et al. Going deeper with convolutions. 2014. Dostupné z: https://arxiv.org/pdf/1409.4842.pdf.

[50] Szegedy, C. et al. Rethinking the Inception Architecture for Computer Vision. 2015. Dostupné z: https://arxiv.org/pdf/1512.00567.pdf.

[51] Vauhkonen, M. – Vohra, Q. – Madaan, S. Deep Learning Benchmarks. 2013. Dostupné z: http://cs229.stanford.edu/proj2013/MadaanVohraVauhkonen-ProjectDeepLearningBenchMarks.pdf.

[52] Werbos, P. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. 1974. Dostupné z: https://www.researchgate.net/publication/35657389_Beyond_regression_new_tools_for_prediction_and_analysis_in_the_behavioral_sciences.

[53] Widrow, B. Adaptive "Adaline" neuron using chemical "memistors". 1960. Dostupné z: http://www-isl.stanford.edu/~widrow/papers/t1960anadaptive.pdf.

[54] Xu, B. et al. Empirical Evaluation of Rectified Activations in Convolution Network. 2015. Dostupné z: https://arxiv.org/pdf/1505.00853.pdf.

[55] Yung, J. Explaining TensorFlow code for a Multilayer Perceptron. 2016. Dostupné z: http://www.jessicayung.com/explaining-tensorflow-code-for-a-multilayer-perceptron/.

[56] Zeiler, M. ADADELTA: AN ADAPTIVE LEARNING RATE METHOD. 2012. Dostupné z: https://arxiv.org/pdf/1212.5701.pdf.

[57] Zeiler, M. – Ferguson, R. Visualizing and Understanding Convolutional Networks. 2013. Dostupné z: https://arxiv.org/pdf/1311.2901.pdf.

Příloha: Obsah přiloženého CD

Součástí práce jsou níže uvedené přílohy, které jsou umístěny na přiloženém CD:

- elektronická verze tohoto dokumentu a jeho zdrojové soubory (adresář **doc**),
- elektronická verze dokumentace k implementované knihovně (adresář **lib_doc**),
- zdrojové soubory implementované knihovny s příklady použití a jednotkovými testy (adresář **src**),
- poster (adresář **poster**),
- README (soubor **README**).