VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ OPTIMALIZACE KONVOLUČNÍCH NEURONOVÝCH SÍTÍ

EVOLUTIONARY OPTIMIZATION OF CONVOLUTIONAL NEURAL NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PAVEL ROREČEK
AUTHOR

VEDOUCÍ PRÁCE — Prof. Ing. LUKÁŠ SEKANINA, Ph.D.
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Roreček Pavel, Bc.**
Obor: Inteligentní systémy
Téma: **Evoluční optimalizace konvolučních neuronových sítí**
Evolutionary Optimization of Convolutional Neural Networks
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s problematikou neuronových sítí, zaměřte se na konvoluční neuronové sítě.
2. Seznamte se s problematikou evoluční optimalizace v kontextu neuronových sítí.
3. Vyberte vhodnou knihovnu pro práci s konvolučními neuronovými sítěmi a seznamte se s jejím použitím.
4. Na zvolených testovacích úlohách ověřte funkčnost knihovny. Porovnejte dosažené výsledky s výsledky v literatuře.
5. S využitím evolučního algoritmu navrhněte vhodnou optimalizaci zvolené neuronové sítě. Cílem je vylepšit vybrané parametry neuronové sítě (např. složitost, dobu výpočtu apod.) i za cenu případného snížení kvality výstupu.
6. Navržené řešení implementujte a na zvolených úlohách ověřte jeho funkčnost.
7. Zhodnoťte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Sekanina Lukáš, prof. Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Tato diplomová práce se zabývá problematikou neuronových sítí se zaměřením na sítě konvoluční (CNN) a evoluční optimalizací v kontextu neuronových sítí. Z existujících knihoven pro modelování CNN byla po analýze vybrána jedna konkrétní, a to Keras. Její funkcionalita je demonstrována na úlohách klasifikace obrázků. S využitím kartézského genetického programování byla navržena a implementována optimalizace CNN za účelem snížení složitosti výpočtu konvolučních vrstev. Dopady navržené optimalizace na chování CNN byly otestovány a vyhodnoceny v rámci případové studie.

Abstract

This Master's Thesis is focused on the principles of neural networks, primarily convolutional neural networks (CNN). It introduces the evolutionary optimization in the context of neural networks. One of existing libraries devoted to the CNN design was chosen (Keras), analysed and used in image classification tasks. An optimization technique based on cartesian genetic programming that should reduce the complexity of CNN's computation was proposed and implemented. The impact of the proposed technique on CNN behaviour was evaluated in a case study.

Klíčová slova

Neuronové sítě, konvoluční neuronové sítě, strojové učení, evoluční algoritmy, kartézské genetické programování, optimalizace

Keywords

Neural networks, convolutional neural networks, machine learning, evolution algorithms, cartesian genetic programming, optimization

Citace

ROREČEK, Pavel. *Evoluční optimalizace konvolučních neuronových sítí*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Prof. Ing. Lukáš Sekanina, Ph.D.

Evoluční optimalizace konvolučních neuronových sítí

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Prof. Ing. Lukáše Sekaniny, Ph.D. a také že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Roreček
22. května 2018

Poděkování

Děkuji panu Prof. Ing. Lukáši Sekaninovi, Ph.D. za kvalitní vedení a podporu při řešení této práce.

Obsah

Kapitola 1

Úvod

V dnešní době se neuronové sítě využívají v mnoha oblastech pro řešení různých problémů. Díky jejich schopnosti učit se, jsou neuronové sítě schopny se ze vzorků trénovací sady naučit, jaké vlastnosti vzorků (tzv. příznaky) jsou pro správné vyřešení úlohy důležité a není tedy třeba popisovat řešení daného problému a výběr příznaků algoritmicky. Často se jedná o úlohy tak komplexní, že exaktní algoritmus pro jejich řešení prakticky vytvořit nelze. Této schopnosti se často využívá například při klasifikačních úlohách, shlukování či rozpoznávání řeči nebo znaků. Obecně se však ví, že tyto sítě jsou výpočetně velice náročné. Cílem této práce je s využitím evolučního algoritmu navrhnout optimalizaci vybraných neuronových sítí za účelem vylepšení některých jejich parametrů (složitost, doba výpočtu a podobně) i za cenu případného zhoršení kvality výstupu.

Předmětem této práce je nastudování teorie neuronových sítí, se zaměřením na konvoluční neuronové sítě, a teorie problematiky evoluční optimalizace v kontextu neuronových sítí. Dále je v práci popsána zvolená knihovna (Keras) pro práci s výše zmíněnými sítěmi a na zvolených úlohách je otestována její funkčnost. Kvalita naučených sítí je srovnána s výsledky v literatuře. V rámci tohoto projektu byla vytvořena knihovna pro kartézské genetické programování (CGP) s jehož využitím je navržena optimalizace konvolučních sítí. Optimalizace spočívá v nahrazení operace konvoluce, využité v konvolučních vrstvách neuronové sítě, aproximací získanou pomocí algoritmu CGP. Cílem této optimalizace je snížení složitosti výpočtu, konkrétně se jedná o eliminaci operace násobení, která se využívá při výpočtu konvoluce. Úspěšnost navržené optimalizace je ověřena na různých úlohách.

V kapitole 2 bude popsán přechod od biologického neuronu, přes umělý neuron až k neuronovým sítím. Kapitola 3 popisuje konvoluční neuronové sítě a vrstvy, které se v nich vyskytují. Na konci této kapitoly je popsána vybraná knihovna pro implementaci daných sítí. Kapitola 4 je zaměřena na popis genetických algoritmů a jejich variant. Kapitola 5 popisuje neuroevoluci, což je forma umělé inteligence využívající evoluční algoritmy v kombinaci s neuronovými sítěmi. V kapitole 6 je popsána úloha symbolické regrese, na které bude ověřena korektnost vytvořené knihovny pro CGP, zároveň je v této kapitole ověřena funkčnost knihovny pro konvoluční neuronové sítě. Kapitola 7 obsahuje popis navržené optimalizace založené na náhradě konvoluční vrstvy aproximační vrstvou získanou pomocí algoritmu CGP. Kapitola 8 obsahuje poznámky k implementaci programové části této práce. V kapitole 9 jsou popsány navržené experimenty společně s jejich vyhodnocením. Závěr práce tvoří kapitola 10.

Kapitola 2

Umělé neuronové sítě

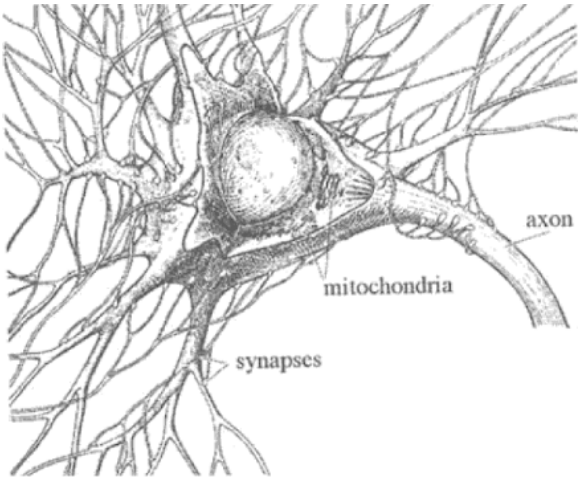
Umělé neuronové sítě se studují v mnoha oborech: v neurobiologii, neurofyziologii nebo také v psychologii. Hlavním důvodem pro zkoumání je jejich podobnost s biologickým nervovým systémem. V těchto oborech se neuronové sítě využívají jako výpočetní modely, které jsou využity k simulaci za účelem pochopení, jak fungují neurony a mozek. V informatice potom existuje snaha napodobovat určité kognitivní schopnosti lidí (zvláště schopnost učení se) s využitím uměle vytvořených elementů nervové soustavy. Praktické využití potom mají umělé neuronové sítě ve výpočetních systémech provádějících klasifikaci, predikci a rozpoznávání.

V této kapitole bude čtenář seznámen s hlavními složkami a principem fungování neuronových sítí. Nejprve je popsán biologický základ – neuron. Následuje popis umělého neuronu a neuronové sítě.

2.1 Biologický neuron

Nervový systém zvířat se skládá z mozku, smyslové soustavy, která sbírá informace z ostatních částí těla (vizuální, zvukové, hmatové, atd.) a motorického systému, který ovládá pohyb. Nervový systém je tvořen velmi komplexními strukturami, ale všechny se skládají z podobných stavebních bloků, nervových buněk nebo neuronů. Tyto bloky mohou vykonávat mnoho různých funkcí, což vede na velmi variabilní morfologii. V biologických neuronových sítích je informace uložena v místech kontaktu mezi jednotlivými neurony, tzv. *synapsemi*. Větší část zpracování informace probíhá v mozku, avšak značné množství předzpracování informací může probíhat mimo mozek, např. v sítnici oka. Dle odhadů je lidský mozek tvořen přibližně 100 biliony neuronů, přičemž větší část z nich pracuje paralelně. Průměrný neuron je spojen s 1000 až 10000 dalšími neurony [11].

Neurony přijímají signál a produkují odezvu. Typická struktura obecného neuronu je ilustrována na obrázku 2.1. Větve nalevo jsou přenosové kanály zvané *dendrity*. Dendrity přijímají signál v místech kontaktu s jinými buňkami, synapsemi. Organely v těle buňky produkují potřebné chemikálie, které zajišťují kontinuální fungování neuronu. Na mitochondrie buňky (viz obrázek 2.1) lze nahlížet jako na část zásobárny energie, jelikož produkují chemikálie, které jsou spotřebovávány jinými buněčnými strukturami. Výstupní signál je přenášen *axonem*, který má každá buňka nanejvýš jeden. Některé buňky axon neobsahují, protože jejich jediným účelem je propojit jiné buňky s dalšími buňkami (např. v sítnici) [19].

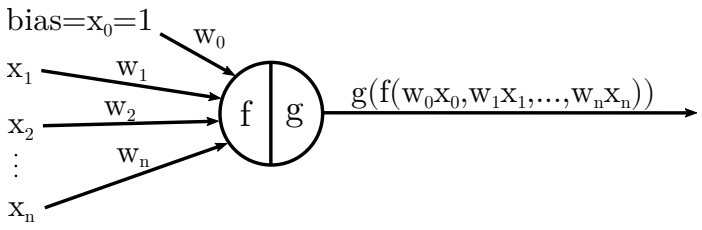


Obrázek 2.1: Typický motorický neuron. Převzato z [19].

.

2.2 Umělý neuron

Obrázek 2.2 zobrazuje strukturu umělého neuronu s n vstupy. Každý vstupní kanál i může vysílat hodnotu x_i, přičemž každému vstupu je přiřazena určitá váha w_i. Vstupní informace z i-tého kanálu je pak rovna $x_i w_i$. Součástí umělého neuronu je tzv. *bias*, což je vstup, jehož hodnota je vždy rovna 1 (bližší popis významu je v podkapitole 2.3.1). Informace přicházející do neuronu je tedy rovna $x_i w_i$. Vstupy vynásobené příslušnou vahou jsou v neuronu sjednoceny pomocí *integrační funkce* f, která převede n hodnot na jednu hodnotu, která je vstupem *funkce aktivační* g, jejíž výstup je roven výstupu umělého neuronu. Funkce f je obvykle sumační funkce a její výstup se označuje jako *potenciál*. Vlastnosti aktivační funkce g a příklady některých funkcí, které se využívají, se vyskytují v podkapitole 2.2.1.



Obrázek 2.2: Ilustrace umělého neuronu.

2.2.1 Aktivační funkce

Jak již bylo zmíněno, aktivační funkce je funkce, která z potenciálu neuronu počítá výstupní hodnotu neuronu. Většinou se jedná o spojité, nelineární, rostoucí a derivovatelné funkce. Následuje příklad některých aktivačních funkcí:

$$jednotkový\_skok(x) = \begin{cases} 0 & \text{pro } x < 0 \\ 1 & \text{pro } x \geq 0 \end{cases} \tag{2.1}$$

$$sigmoida(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

$$hyperbolický\_tangens(x) = tanh(x) = \frac{2}{1 + e^{-2x} - 1} \tag{2.3}$$

$$ReLU(x) = \begin{cases} 0 & \text{pro } x < 0 \\ x & \text{pro } x \geq 0 \end{cases} \tag{2.4}$$

2.3 Konstrukce umělých neuronových sítí

Pokud chápeme každý uzel neuronové sítě jako primitivní funkci schopnou transformovat její vstupy na přesně definovaný výstup, pak je lze umělou neuronovou síť chápat jako *síť primitivních funkcí*. Různé modely neuronových sítí se liší v použitých primitivních funkcí, způsobu propojení jednotlivých neuronů a časování přenosu informací [19].

2.3.1 Perceptron

Perceptron je model neuronové sítě obsahující pouze jeden neuron, přičemž jeho výstupní funkce f je definována takto:

$$f(x) = \begin{cases} 1 & \text{pro } \sum_{i=1}^{n} w_i x_i \geq \Theta \\ 0 & \text{pro } \sum_{i=1}^{n} w_i x_i < \Theta \end{cases} \tag{2.5}$$

kde Θ značí tzv. *práh*.

Pokud zvolíme vstup x_0 roven jedné a váhu jeho spojení s tělem neuronu rovnu Θ, můžeme pak ve výše zmíněném vzorci uvažovat sumu od $i = 0$ a porovnávat výstup neuronu s hodnotou 0 namísto Θ.

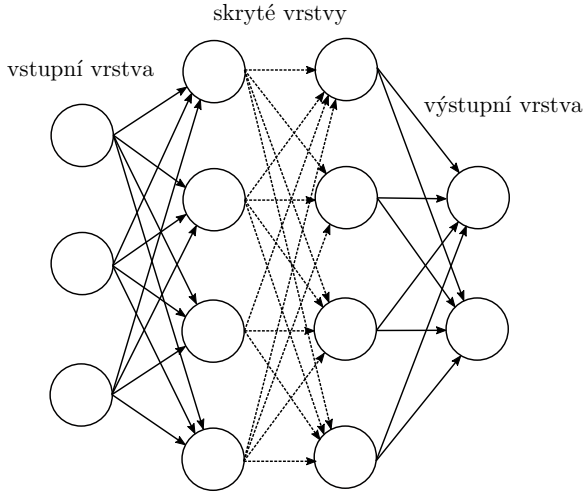
Perceptron lze učit speciálním pravidlem. Jeho váhy jsou měněny v závislosti na rozdílu (chybě) mezi známým řešením a výstupem neuronu. Chyba je funkce všech vah a obecně tvoří nepravidelnou multidimenzionální nadrovinu s mnoha vrcholy, sedly a minimy. Pomocí technik prohledávání prostoru se učící proces snaží najít množinu vah, která odpovídá globálnímu minimu. Tento proces objeví optimální vektor vah v konečném počtu iterací bez ohledu na nastavení počátečních vah. Toto pravidlo se však vztahuje pouze na situaci, kdy je možné vstupní vektory rozdělit do lineárně separovatelných tříd.

Pro zpracování nelineárně separabilních problémů je potřeba přidat další vrstvu/vrstvy neuronů mezi vstupní vrstvu (obsahující vstupní uzly) a výstupní neurony. Tato struktura je označována jako *vícevrstvý perceptron*. Vnitřní vrstvy neinteragují se svým okolím, a proto se jim říká *skryté vrstvy*. Učení takovéto sítě je složitější než učení samotného perceptronu [7].

2.3.2 Vícevrstvé sítě

Dopředná síť (ilustrována na obrázku 2.3) je jedna z nejvíce používaných neuronových sítí. Jedná se o vícevrstvý perceptron skládající se ze:

- vstupní vrstvy s uzly reprezentujícími vstupní proměnné,
- výstupní vrstvy s uzly reprezentujícími závislé proměnné (to, co je modelováno),
- jedné nebo více skrytých vrstev s uzly, které pomáhají řešit nelinearitu dat.



Obrázek 2.3: Ilustrace sítě, která obsahuje vstupní vrstvu se třemi neurony, 2 skryté vrstvy se čtyřmi neurony a výstupní vrstvu se dvěma neurony.

V této síti neexistují zpětné vazby mezi vrstvami a neexistují vazby mezi neurony téže vrstvy, což se označuje jako *dopředná neuronová síť*. Neurony mohou být propojeny řídce (pouze s některými neurony předchozí vrstvy) nebo hustě (se všemi neurony předchozí vrstvy). Sítě se zpětným šířením chyby se využívají pro modelování dat, klasifikaci, předpovídání, ovládání, kompresi dat a obrázků, a rozpoznávání vzorů [7].

2.3.3 Algoritmy učení pro neuronové sítě

Algoritmus učení je adaptivní metoda, díky které je síť výpočetních jednotek schopna upravit své váhy tak, aby dosáhla požadovaného chování. Algoritmus učení síti předkládá trénovací příklady mapování vstupů na výstupy. Úpravy sítě jsou prováděny iterativně, dokud se síť nenaučí produkovat požadované výstupy.

V jednoduchých případech lze váhy pro výpočetní jednotky najít pomocí sekvenčního testování náhodně generovaných numerických kombinací. Avšak takový algoritmus, který se snaží slepě najít řešení, se nepovažuje za algoritmus učení. Algoritmus učení musí adaptovat parametry sítě vzhledem k předchozím zkušenostem, dokud není nalezeno řešení (pokud řešení existuje).

Algoritmy učení se dají dělit na metody *učení s učitelem*, *učení bez učitele* a *posilované učení*. Učení s učitelem značí metody, které síti předloží vstupy a poté porovnají výstup sítě s očekávaným výstupem. Váhy jsou upraveny vzhledem k velikosti chyby, které se síť dopustila. Učení bez učitele a posilované učení je použito, pokud není znám přesný výstup, který má síť poskytnout [19].

Algoritmus zpětného šíření chyby

Algoritmus zpětného šíření chyby je algoritmus učení s učitelem, pomocí kterého se neuronové sítě mohou naučit mapování z jednoho prostoru dat do prostoru jiného s použitím zadaných příkladů. Zpětné šíření chyby označuje způsob, jakým je vypočítaná chyba výstupu sítě propagována zpět od výstupní vrstvy zpět k vrstvě vstupní (viz pseudoalgoritmus 1).

Nastav váhy v síti na náhodně vygenerovaná čísla (obvykle nízké hodnoty);
repeat
  Předej vstupní vrstvě trénovací vzorek;
  Vypočítej výstupy neuronů ve skrytých vrstvách;
  Vypočítej vystup neuronů ve výstupní vrstvě;
  Získej výsledek chybové funkce, do které vstupuje získaný výsledek a požadovaný výsledek;
  S využitím výsledku chybové funkce a gradientního sestupu (zde se využívá tzv. *koeficient učení*, který udává krok gradientního sestupu) uprav váhy spojení neuronů skryté vrstvy s neurony vrstvy výstupní, váhy neuronů skrytých vrstev a váhy neuronů vstupní vrstvy s neurony následující skryté vrstvy, tak aby došlo ke snížení chyby klasifikace;
until *Všechny trénovací vzorky nejsou správně klasifikovány nebo nenastala jiná ukončovací podmínka*;

Algorithm 1: Jednoduchý pseudoalgoritmus zpětného šíření chyby.

Kapitola 3

Konvoluční neuronové sítě

Schopnost vícevrstvé sítě se zpětným šířením chyby naučit se komplexní, vícedimenzionální, nelineární mapování z velkého počtu příkladů z ní dělá jistého kandidáta na řešení problému rozpoznávání obrázků nebo zpracování řeči. Obvykle se pomocí ručně navrženého extraktoru příznaků získají ze vstupu relevantní informace a irelevantní se ignorují. Pomocí těchto příznaků lze natrénovat klasifikátor, který kategorizuje vektory vstupů do tříd. Obecně však chceme ruční navrhování extraktoru příznaků odstranit, resp. do neuronové sítě zakomponovat vrstvy, které se stanou automatickými extraktory příznaků. Dalším problémem klasických neuronových sítí je, že neumožňují zpracovat strukturovaná data. Obecně však platí, že data jsou strukturována a v určitých částech korelována. Výše zmíněné problémy řeší konvoluční neuronové sítě (dále jen CNN), které jsou schopny extrahovat lokální příznaky ze vstupních dat [12].

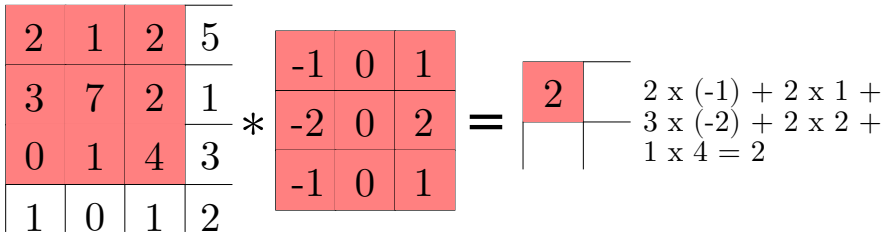
Následuje popis konvoluční vrstvy a vrstev, které se velmi často současně s konvolučními vrstvami využívají. V závěru kapitoly je popsána zvolená knihovna pro práci s CNN.

3.1 Konvoluční vrstva

Konvoluční vrstva je tvořena několika konvolučními filtry. Výstup každého filtru je roven výsledku operace matematické konvoluce, do které vstupují váhy filtru a vstup vrstvy. Často ke každému filtru přísluší tzv. *bias* hodnota, která se přičte k výstupu filtru. Vzorec pro dvourozměrnou konvoluci K ($*$ značí operaci konvoluce):

$$K(x,y) = M * f(x,y) = \sum_{i=0}^{n}\sum_{j=0}^{m} M_{n,m} f(x - i + \frac{n-1}{2}, y - j + \frac{m-1}{2}), \tag{3.1}$$

kde f značí funkci navracející hodnotu prvku na pozici (x,y) ve vstupních datech a M je konvoluční filtr. Konvoluce je ilustrována na obrázku 3.1.



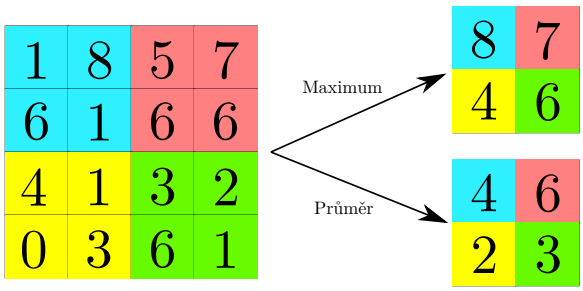
Obrázek 3.1: Demonstrace konvoluce. Na vstupní obraz je aplikován filtr s maskou 3×3 pixely.

.

Jak již bylo zmíněno, konvoluční vrstva neuronové sítě obsahuje obecně více filtrů, přičemž filtry vrstev na nižších úrovních slouží jako prosté detektory hran a filtry na úrovních vyšších mohou sloužit pro rozpoznávání komplexnějších struktur (např. oči, obočí a podobně). Výstupem konvoluční vrstvy jsou mapy příznaků, které jsou tvořeny výstupy jednotlivých konvolučních filtrů. Společně s konvoluční vrstvou se často využívají i jiné vrstvy. Vybrané z nich budou popsány níže.

3.2 Podvzorkovací vrstva

Tato vrstva slouží k redukci velikosti dat, které obdrží na vstupu. Redukce se provádí pomocí okénka určité velikosti, pomocí kterého se vstup rozdělí na nepřekrývající se čtvercové oblasti a v každé oblasti se nad prvky provede určitá matematické operace. Hodnota, která reprezentuje zpracovanou čtvercovou oblast, bývá vetšinou průměr nebo maximální hodnota z hodnot v oblasti. Ilustrace činnosti je zobrazena na obrázku 3.2.



Obrázek 3.2: Ilustrace činnosti podvzorkovací vrstvy.

.

3.3 Zplošťovací vrstva

Zplošťovací vrstva (anglicky flatten layer) slouží k pouhému zploštění vstupu, resp. transformaci obecně n-dimenzionálního pole hodnot do jednodimenzionálního vektoru. Často se nachází před plně propojenou vrstvou.

3.4 Vynechávající vrstva

Vynechávající vrstva (anglicky dropout layer) se využívá při trénovací fázi sítě. Její hlavní funkcí je zamezovat *přeučení*, tzv. overfitting, tím, že v obecně n-dimenzionálním poli vstupních hodnot některé hodnoty vynuluje (pravděpodobnost bývá zadána manuálně) a toto upravené pole bude výstupem vrstvy.

3.5 Plně propojená vrstva

Jedná se o vrstvu, která se obvykle vyskytuje na konci konvoluční neuronové sítě. Jedná se o vrstvu obsahující n neuronů, přičemž každý neuron této vrstvy je připojen na každý neuron vrstvy předchozí. Při klasifikačních úlohách je n rovno počtu kategorií, přičemž jeden neuron přísluší jedné kategorii. Cílem vrstvy je nalézt korelace vektoru příznaků z předchozí vrstvy s jednotlivými kategoriemi. Jako aktivační funkce se často využívá funkce softmax:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \text{ pro } i = 1, ..., N \tag{3.2}$$

kde platí $softmax : \mathbb{R}^N \rightarrow [0, 1]^N$. Funkce softmax transformuje vektor x hodnot velikosti N na vektor hodnot stejné velikosti, ve kterém je součet všech prvků roven 1.

3.6 Učení konvoluční neuronové sítě

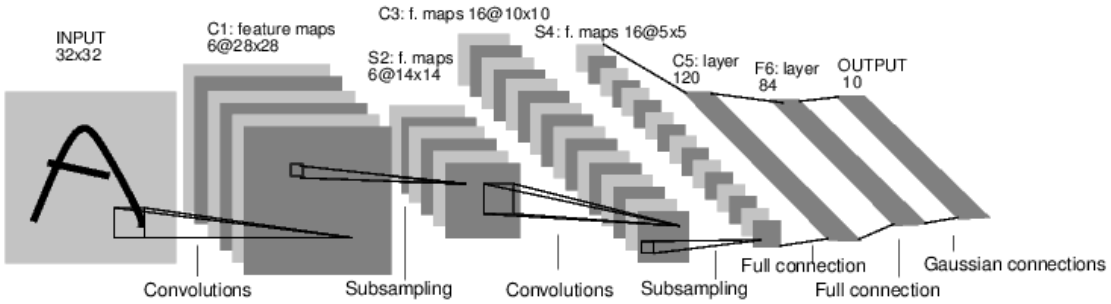
Pro učení konvolučních neuronových sítí je, stejně jako u klasických neuronových sítí, využit algoritmus zpětného šíření chyby. Algoritmus zpětného šíření chyby je však rozšířen o schopnost upravovat váhy konvolučních filtrů konvolučních vrstev, tzn. z filtrů se stanou automatické extraktory příznaků.

3.7 Příklady CNN

Jedny z nejznámějších architektur hlubokých neuronových sítí jsou sítě z rodiny LeNet. Jednou z těchto sítí je např. LeNet5 (ilustrována na obrázku 3.3) která je složena z konvoluční vrstvy obsahující 6 konvolučních filtrů s velikostí okénka 5×5, podvzorkovací vrstvy, další konvoluční vrstvy obsahující 16 konvolučních filtrů velikosti 5×5, podvzorkovací vrstvy, konvoluční vrstvy tvořené 120 filtry velikosti 1×1 a plně propojené vrstvy s 84 neurony a plně propojené vrstvy s 10 neurony. Architektura je ilustrována na obrázku 3.3. Architektury LeNet jsou poměrně jednoduché a nejsou ideální pro řešení složitějších problémů. V dnešní době se využívají mnohem složitější sítě viz tabulka 3.1.

3.8 Knihovna pro modelování konvolučních neuronových sítí

Pro potřeby této práce je velmi důležitá vhodná volba použité knihovny pro modelování hlubokých neuronových sítí. Po experimentování s různými knihovnami (např. TinyDNN, Caffee2 a TensorFlow), byla zvolena knihovna Keras, která je nástavbou nad knihovnou TensorFlow. Hlavním důvodem pro zvolení této knihovny je možnost snadného nahrazení vrstvy v již natrénovaném modelu hluboké neuronové sítě (ostatní knihovny tuto vlastnost postrádaly nebo byla úprava struktury modelu velmi složitá). Tato vlastnost tedy umožňuje



Obrázek 3.3: Ilustrace architektury LeNet5. Převzato z [13].

Metriky	LeNet5	AlexNet	VGG-16	GoogleLeNet (v1)	ResNet-50
Rozměr vstupu sítě	28x28	277x277	244x244	244x244	244x244
Počet konvolučních vrstev	2	5	16	21	49
Počet plně propojených vrstev	2	3	3	1	1
Počet vah v sít	60k	61M	138M	7M	25.5M
Počet MAC	341k	724M	15.5G	1.43G	3.9G

Tabulka 3.1: Příklady architektur hlubokých neuronových sítí a jejich parametrů. MAC značí operaci, která spočítá výsledek násobení a přičte jej do akumulátoru. Data převzata z [5].

velmi snadné nahrazení konvoluční vrstvy jinou vrstvou, v případě této práce vrstvou, která aproximuje původní konvoluční vrstvu, viz kapitola 7.

3.8.1 TensorFlow

TensorFlow je open source knihovna pro numerické výpočty s využitím grafů toku dat. Původně byla vyvíjena týmem Google brain ve výzkumné organizaci Google Machine Intelligence, která se zabývá strojovým učením a výzkumem hlubokých neuronových sítí.

Knihovna je multiplatformní a také umí využívat jak GPU tak CPU, a to i u mobilních zařízení a vestavěných platforem. Jádro knihovny je implementováno v jazyce C++ a knihovnu je možné používat jak v jazyce C++, tak i v jazyce Python [6].

Níže budou popsány základní koncepty této knihovny. Materiály jsou převzaty z [3].

Tenzor

Centrální jednotkou v knihovně TensorFlow je *Tenzor*. Tenzor v matematice značí zobecnění vektoru. Vektor označuje veličinu, která má směr a velikost. Složky vektoru lze indexovat jedním indexem, zatímco prvky tenzoru indexujeme pomocí n indexů.

Tenzor se skládá z množiny primitivních hodnot uspořádaných do pole libovolné dimenze. *Řád* tenzoru značí počet jeho dimenzí. Příklady tenzorů:

```
3 # řád tenzoru = 0; skalární hodnota s tvarem []
[1., 2., 3.] # řád tenzoru = 1; vektor s tvarem [3]
[[1., 2., 3.], [4., 5., 6.]] # řád tenzoru = 2; matice tvaru [2, 3]
[[[1., 2., 3.]], [[7., 8., 9.]]] # řád tenzoru = 3; tvar [2, 1, 3]
```

Výpočetní graf

Výpočetní graf je série TensorFlow operací uspořádaných do grafu. Následuje ukázka vytvoření jednoduchého výpočetního grafu:

```
import tensorflow as tf

node1 = tf.constant(4.0, dtype=tf.float32)
node2 = tf.constant(2.0) # tf.float32 je implicitní
node3 = tf.add(node1, node2)

sess = tf.Session()

print(sess.run(node3))
```

Pro použití knihovny TensorFlow je jako první nutno naimportovat danou knihovnu. Příkazy `tf.constant` vytvoří tensor s konstantní hodnotou. Příkaz `tf.add` vytvoří tensor, který sčítá hodnoty dvou tensorů. Posledním krokem je potřeba vytvořit prostředí, ve kterém lze spustit výpočetní graf a tím získat výsledek, který je vypsán.

3.8.2 Keras

Keras je vysokoúrovňové API, napsané v jazyce Python a je schopné využívat knihovny TensorFlow, CNTK nebo Theano. Zaměřuje se zejména na umožnění rychlého experimentování. Údaje o knihovně jsou čerpány z [4].

> „Možnost dopracovat se od myšlenky k výsledku s co nejmenším zpožděním je klíčové k provádění kvalitního výzkumu.“ - Keras

Knihovna Keras umožňuje jednoduché a rychlé prototypování. Poskytuje jak konvoluční tak i rekurentní sítě (sítě obsahující orientované grafy). Lze spustit bez problémů jak na CPU tak GPU.

Následuje jednoduchý příklad použití knihovny keras pro vytvoření velmi jednoduché neuronové sítě:

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()

model.add(Dense(units=64, activation=relu, input_dim=100))
model.add(Dense(units=10, activation=softmax))

model.compile(loss=categorical_crossentropy,
              optimizer=sgd,
              metrics=[accuracy])

model.fit(x_train, y_train, epochs=5, batch_size=32)

loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Jako první je potřeba naimportovat potřebné struktury. `Sequential` značí typ použitého modelu. Jedná se o velmi jednoduchý model, ve kterém jsou úrovně skládány na sebe. `Dense` značí hustě propojenou neuronovou síť. Dvě vrstvy tohoto typu jsou poskládány za sebe a oběma je nastavena aktivační funkce. U první vrstvy musí být vždy specifikován tvar vstupu, v tomto případě 100 hodnot. Dalším krokem je konfigurace učícího procesu příkazem `compile` – je nutno zadat chybovou funkci, druh učícího algoritmu a seznam metrik, které chceme sledovat. Příkazem `fit` je spuštěno iterování nad trénovacími daty (v dávkách velikosti 32 a celkově se provede 5 epoch). Příkazem `evaluate` dojde k evaluaci kvality natrénované sítě.

Kapitola 4

Evoluční a genetické algoritmy

Snahy o vytvoření umělé inteligence a umělého života lze zařadit na úplný počátek éry počítačů. První počítačoví vědci - Alan Turing, John von Neumann, Norbert Wiener a další byli z velké části motivováni vizí obohacení počítačového programu inteligencí, schopností rozmnožování a adaptivními schopnostmi – učení a interakce s prostředím. Tito prvotní průkopníci se také zajímali o oblast biologie, psychologie, elektroniku. Na přírodní systémy se dívali jako na inspiraci, jak dosáhnout svých vizí. Počítače jsou používány nejen k vypočítávání trajektorie raket a rozšifrovávání vojenských šifer, ale také k modelování mozku, napodobování lidského učení a simulování biologické evoluce. S postupem času se modelování mozku přetransformovalo na oblast neuronových sítí, napodobování učení na strojové učení a simulování biologické evoluce se dnes říká „evoluční počítání“, ze kterého jsou nejvíce známé *genetické algoritmy* [16].

Problémy, které se v informatice řeší, se postupně stávají složitějšími a proto existuje snaha, automatického hledání řešení těchto problémů – resp. algoritmů. Požadavkem je, aby tyto algoritmy byly aplikovatelné na celou řadu problémů, aby nevyžadovaly velký zásah člověka a aby nalezly kvalitní řešení (nemusí být optimální) v rozumném časovém intervalu. Na řadě problémů evoluční algoritmy ukázaly, že všechny tyto požadavky splňují. Další motivací k využívání evolučních procesů je zjistit, jak funguje biologická evoluce. Na rozdíl od klasické evoluce v biologii lze evoluční algoritmy (procesy) simulovat v počítači, je tedy možno simulovat miliony generací v krátkém časovém intervalu. Avšak je potřeba pamatovat, že není jisté, zda modely použité při výpočtu dostatečně odpovídají modelům biologickým, a proto je nutné obezřetně interpretovat dosažené výsledky. Evoluční algoritmy se používají pro řešení mnoha úloh, například v modelování – z mnoha dat bez zjevné struktury chceme zjistit určité závislosti, v optimalizaci – hledání co nejlepšího řešení optimalizačního problému, nebo ve studování biologických procesů [8].

4.1 Prohledávání stavového prostoru

Při řešení určitého problému se obecně snažíme nalézt takové řešení, které bude lepší než všechny ostatní řešení. Prostor všech možných řešení se nazývá *stavový prostor*. Každý bod v tomto prostoru označuje jedno konkrétní řešení. Každé takové řešení lze označit nějakou hodnotou – *fitness* hodnotou. Při hledání řešení se snažíme nalézt řešení s nejlepší fitness hodnotou. V ideálním případě se snažíme nalézt nejlepší řešení v celém stavovém prostoru, ale tento prostor je obecně velmi rozsáhlý a nelze testovat všechna možná řešení. Proto se využívají algoritmy sloužící pro jeho prohledávání, např. simulované žíhání, horolezecký

algoritmus nebo varianty evolučních algoritmů. Pokud řešíme složitý reálný problém, tak řešení získané pomocí těchto algoritmů nebývá obecně globálním optimem. Důvodem je, že získání dobrého suboptimálního řešení je obvykle postačující a navíc skutečné globální optimum většinou neznáme.

4.2 Evoluční algoritmus

Existuje mnoho variant evolučních algoritmů, avšak jejich hlavní myšlenka je vždy stejná. Prvním krokem je vygenerování počáteční množiny kandidátních řešení (tzv. *populace*) umístěné v určitém prostředí. Dále je každý jedinec ohodnocen evaluační/fitness funkcí a dle výsledku tohoto ohodnocení je určena šance, se kterou bude jedinec použit pro tvorbu jedinců následující generace. Po vybrání jednoho či více rodičů je pomocí daného operátoru křížení vytvořen jedinec, na který je aplikován operátor mutace a poté je zařazen do nové generace. Tímto způsobem dojde k vytvoření úplně nové generace a celý proces se opakuje (bez inicializace počáteční populace) do té doby, než je dosaženo ukončovací podmínky. Cílem je, aby s rostoucím časem (počtem generací) docházelo k postupnému zlepšování jedinců – resp. zlepšování fitness hodnot těchto jedinců. Obecný evoluční algoritmus je popsán pseudoalgoritmem 2. Kandidátní řešení se označuje jako *chromozom*, jeho složky se označují jako *geny* a konkrétní hodnoty těchto genů jsou *alely*.

```
Inicializuj počáteční populaci;
Ohodnoť všechny jedince v populaci;
while neplatí ukončující podmínka do
    Vyber rodiče z populace;
    Vytvoř nové potomky aplikací operátoru křížení na rodiče;
    Aplikuj operátor mutace na vzniklé potomky;
    Ohodnoť potomky;
    Vyber potomky pro další generaci;
end
Jedinec s nejlepší hodnotou fitness je výstupem algoritmu;
```

Algorithm 2: Pseudokód evolučního algoritmu.

Jako ukončovací podmínka se většinou uvažuje stav, ve kterém se počet vygenerovaných generací rovná zadanému maximálnímu počtu generací, nebo stav, ve kterém nedochází ke konvergenci k řešení, resp. za určitý počet generací nedošlo ke zlepšení fitness hodnoty nejlepšího jedince z populace.

4.2.1 Výběr rodičů

Pro výběr rodičů použitých pro vytvoření potomků nové generace existuje více možných algoritmů. Dále budou popsány vybrané metody: ruleta, turnaj a uniformní výběr.

Ruleta

Nejčastější metoda pro výběr rodičů je metoda *Rulety*. Aby tuto metodu bylo možno použít, je potřeba znát fitness hodnotu všech jedinců populace. Pravděpodobnost výběru i-tého jedince $p(i)$ jako rodiče je pak dána takto:

$$p(i) = \frac{f_i}{\sum_{j=1}^{N} f_j}, \tag{4.1}$$

kde f značí fitness hodnotu a N značí počet všech jedinců v populaci.

Nevýhodou této metody je, že se může vyskytnout tzv. *shlukování*. Shlukování označuje situaci, kdy fitness hodnota určitého jedince je v porovnání se všemi ostatními jedinci příliš vysoká, což má za výsledek to, že se tento jedinec prakticky vždy vybere jako rodič. Výsledkem je snížení diverzity populace. Řešením tohoto problému může být zvolení jiné metody pro výběr rodičů, například ruleta založená na pořadí (při výběru se bere v potaz pořadí jedince dle jeho fitness) nebo turnaj.

Turnaj

Výše zmíněné metody výběru rodičů z populace vyžadují znalosti o každém jedinci v populaci (jeho fitness hodnota). Ale v některých situacích je populace příliš velká nebo nějakým způsobem distribuovaná (např. v paralelních systémech), takže získání znalosti o každém jedinci je buď časově velmi náročné nebo v nejhorším případě nemožné.

Metoda turnaje vyžaduje pouze schopnost porovnat jedince (dle jejich fitness hodnoty), takže je konceptuálně jednoduchá a rychlá (jak v implementaci, tak použití), nemusí se zjišťovat fitness hodnota všech jedinců.

Algoritmus funguje tak, že se z celé populace vybere n jedinců, každý z nich je ohodnocen (zjistí se jeho fitness hodnota) a nejlepší z nich je vybrán jako rodič. Toto se opakuje, dokud není dosaženo požadovaného počtu rodičů [8].

Náhodný výběr

V některých oblastech evolučních algoritmů je obvyklé, že každý jedinec z populace má stejnou šanci stát se rodičem, tzn. pokud populace obsahuje μ jedinců, pak je šance na výběr i-tého jedince rovna $p(i) = \frac{1}{\mu}$. Nebere se tedy ohled na jeho kvalitu – fitness hodnotu. Což se může zdát neintuitivní (jako rodiče mohou být vybraní jedinci, kteří jsou mnohem horší než zbytek populace), ale většinou je náhodný výběr spjat s výběrem potomků pro další populaci, ve kterém se již fitness jedinců zohledňuje.

4.2.2 Operátory křížení

Operátory křížení se používají, pokud chceme ze dvou rodičů (vybraných chromozomů) vytvořit nového potomka nebo i více potomků. Existují různé varianty, přičemž každá má totožný cíl a to urychlení nalezení co nejlepšího řešení daného problému. Dále budou popsány, a na binární reprezentaci chromozomu ilustrovány, 3 různé operátory křížení. Informace jsou čerpány z [23].

Jednobodové

Jako první se zvolí náhodné číslo p_i v rozsahu od 0 do n, kde n je počet genů v chromozomu. Hodnoty genů, které následují za bodem křížení p_i, jsou mezi rodiči prohozeny, čímž vzniknou noví potomci (viz obrázek 4.1).

1. rodič	1	1	0	1	0	1	0	1
2. rodič	0	1	0	0	1	0	1	1
1. potomek	1	1	0	1	0	0	1	1
2. potomek	0	1	0	0	1	1	0	1

Obrázek 4.1: Ilustrace jednobodového křížení s bodem křížení mezi 5. a 6. genem.

K-bodové

K-bodové křížení je velmi podobné jednobodovému s tím rozdílem, že je bodů křížení vygenerováno více. Hodnoty genů, které jsou před prvním bodem křížení p_1 se neprohazují, body za bodem p_1 se prohazují, než je dosaženo dalšího bodu křížení, atp. pro další body (viz obrázek 4.2).

1. rodič	1	1	0	1	0	1	0	1
2. rodič	0	1	0	0	1	0	1	1
1. potomek	1	1	0	0	1	1	0	1
2. potomek	0	1	0	1	0	0	1	1

Obrázek 4.2: Ilustrace 3-bodového křížení s body křížení mezi 2. a 3. genem a 5. a 6. genem.

Uniformní

Při uniformním křížení platí, že se pro každý gen vygeneruje náhodné číslo $i \in< 0, 1)$ a pokud platí, že $i < 0.5$, pak bude 1. potomek obsahovat hodnotu tohoto genu (alelu) od 1. rodiče a 2. potomek obdrží alelu od 2. rodiče, pokud je $i >= 0.5$, pak bude 1. potomek obsahovat tuto alelu od 2. rodiče a 2. potomek obdrží alelu od 1. rodiče (viz obrázek 4.3).

1. rodič	1	1	0	1	0	1	0	1
2. rodič	0	1	0	0	1	0	1	1
1. potomek	1	1	0	0	1	1	1	1
2. potomek	0	1	0	1	0	0	0	1

Obrázek 4.3: Číslo i bylo větší nebo rovno 0.5 pro 2., 4., 5. a 7. gen.

4.2.3 Operátor mutace

Mutace je proces, při kterém se změní hodnota genu. Jedná se o procházení celého chromozomu jedince, přičemž existuje malá šance, že se změní hodnota genu na jinou hodnotu. Význam mutace je velmi velký, umožňuje totiž vznik nových vlastností, které by bez mutace nemohly vzniknout.

4.2.4 Elitismus

Elitismus je obvykle používán společně s jinými metodami a zajišťuje zachování určitého počtu nejlepších jedinců v další generaci. Tito jedinci by mohli být ztraceni, pokud by nebyli vybráni k reprodukci nebo by byli zničení křížením či mutací [16].

4.3 Varianty evolučních algoritmů

Níže budou popsány základní varianty evolučních algoritmů. Jedná se o genetické algoritmy, evoluční strategie a genetické programování.

4.3.1 Genetické algoritmy

Genetický algoritmus je nejznámější typ evolučních algoritmů. Jedinec je reprezentován jako řetězec bitů, pro výběr rodičů se využívá metoda rulety (popř. metoda rulety založená na pořadí), pravděpodobnost mutace je poměrně malá a je použit jednobodový nebo vícebodový operátor křížení. Genetický algoritmus je popsán pseudoalgoritmem 3.

Mějme populaci o μ jedincích;
Ohodnoť všechny jedince v populaci;
while *neplatí ukončující podmínka* **do**
  Metodou rulety vyber μ jedinců a umísti je do dočasné populace (duplikáty jsou povoleny);
  Náhodně promíchej jedince v dočasné populaci a vytvoř z nich páry;
  S pravděpodobností p_c na každý pár aplikuj operátor křížení a vzniklými potomky nahraď rodiče (pokud došlo ke křížení);
  Na všechny jedince z dočasné populace aplikuj operátor křížení s pravděpodobností p_m;
  Ohodnoť potomky;
  Nahraď současnou generaci dočasnou generací;
end
Jedinec s nejlepší hodnotou fitness je výstupem algoritmu;

Algorithm 3: Pseudokód evolučního algoritmu.

Doporučená míra mutace je mezi $1/l$, kde l je délka řetězce bitů, kterým je reprezentován jedinec. Pravděpodobnost aplikace operátoru křížení je mezi $0.6 - 0.8$ a velikost populace je mezi $50 - 100$ jedinci [8].

4.3.2 Evoluční strategie

V evoluční strategii je jedinec reprezentován jako vektor reálných čísel, výběr rodičů zajišťuje uniformní výběr a výběr jedinců pro další generace využívá elitismus. Jako operátor mutace je využita gaussovská perturbace, tzn. k hodnotě genu je přičtena náhodná hodnota z Gaussova rozložení se střední hodnotou 0 a směrodatnou odchylkou σ, které se říká *krok mutace*. Součástí reprezentace jedince bývají i tzv. *parametry strategie*, které se vyvíjí společně s jedincem (často se jedná o parametr σ). Křížení je zajištěno pomocí uniformního křížení s tím, že výsledná alela pochází buď od jednoho z rodičů, nebo je výsledná alela rovna průměrů alel rodičů.

Algoritmy se značí jako $(\mu+\lambda)$ a (μ, λ), kde μ udává počet jedinců v populaci a λ udává počet potomků generovaných v jedné generaci. Rozdíl v těchto dvou algoritmech je, že se v první variantě pro výběr jedinců, kteří budou tvořit další populaci, uvažuje jak λ potomků, tak i μ rodičů. V druhé variantě se nová generace vytváří pouze z λ potomků. Obvykle se volí poměr $\mu = 1$ a $\lambda = 7$ [8].

4.3.3 Genetické programování

Genetické programování využívá pro popis chromozomu stromové struktury, křížení je zajištěno výměnou podstromů, mutace je realizována jako náhodná změna ve stromové struktuře a pro výběr rodičů je využita metoda rulety.

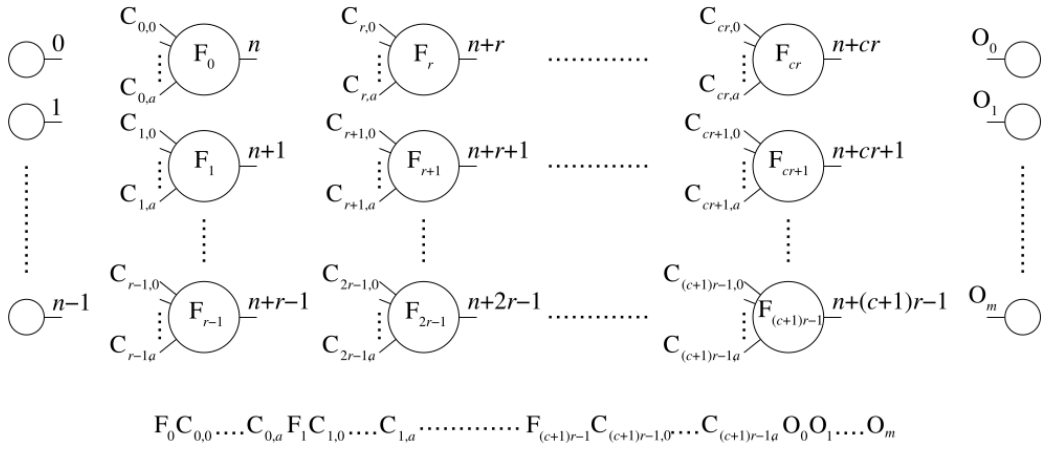
Stromové struktury používané v generickém programování popisují výrazy v dané formální syntaxi. Dle problému se může jednat o syntaxi aritmetických výrazů, formulí predikátové logiky nebo programovacího jazyka. Na stromy lze nahlížet jako na spustitelné kód – programy, díky tomu se genetické programování označuje jako prostředek pro automatickou evoluci počítačových programů [8].

4.3.4 Kartézské genetické programování

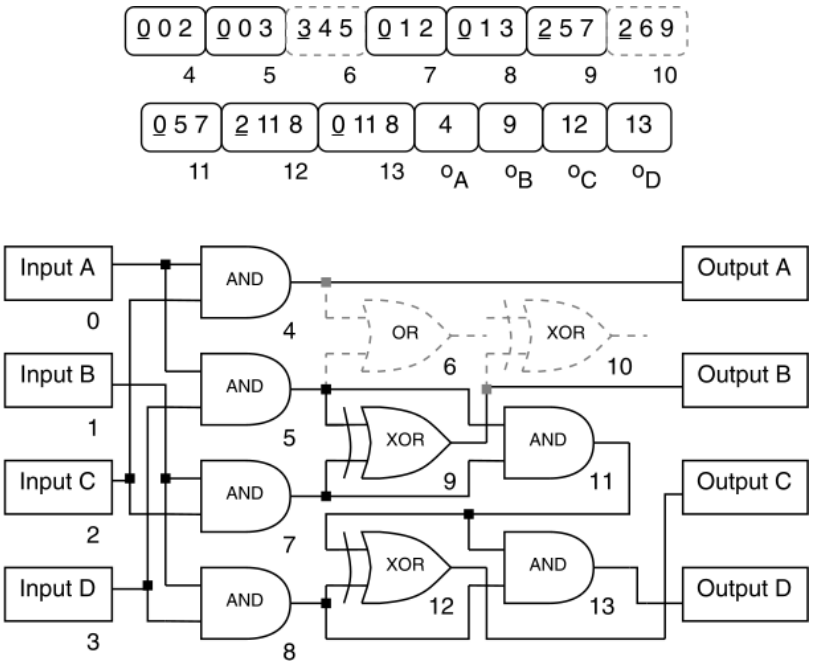
Kartézské genetické programování (dále jen CGP) je druh genetického programování, ve kterém jsou programy reprezentovány ve formě acyklických orientovaných grafů. Tyto grafy jsou reprezentovány jako dvourozměrná mřížka výpočetních uzlů. Graf je ilustrován na obrázku 4.4. Geny tvořící genotyp v CGP jsou celá čísla, která určují zdroj dat a typ operace, která se nad těmito daty má vykonat. Pokud se výpočetní uzel nepodílí na výpočtu výstupní hodnoty je označován jako „nekódující“. Genotyp má fixní délku, avšak velikost fenotypu může být rovna nule, pokud neobsahuje žádný výpočetní uzel (výstup je připojen přímo na vstup). Maximálně obsahuje N uzlů, což je celkové množství uzlů definované v genotypu (každý uzel se podílí na výpočtu výstupní hodnoty). Převod chromozomu na fenotyp je ilustrován na obrázku 4.5. CGP může reprezentovat celou řadu struktur, chromozom může kódovat například digitální obvody nebo může reprezentovat matematické funkce.

Typy operací, které může výpočetní uzel provádět, jsou definovány uživatelem ve vyhledávací tabulce funkcí. Uzel je reprezentován několika geny. Jeden gen určuje funkci ve vyhledávací tabulce a zbylé geny určují zdroj dat (uzly předchozích vrstev).

CGP má tři parametry zadávané uživatelem. Jedná se o počet sloupců - c, počet řádků r a parametr l. Maximální počet výpočetních uzlů je vypočítán jako $N = cr$. Parametr l (tzv. l-back) určuje konektivitu grafu, resp. z kolika předchozích vrstev může uzel získat vstupní hodnotu [17].



Obrázek 4.4: Obecná forma CGP. Jedná se o mřížku uzlů, jejichž funkce jsou zvoleny z množiny primitivních funkcí. Mřížka má c sloupců (v obrázku je potřeba za c dosadit počet sloupců snížený o 1) a r řádků. Počet vstupů je n a počet výstupů je m. U každého uzlu se předpokládá, že je počet jeho vstupů roven maximální aritě použitých funkcí (ve funkci samotné se však mohou uvažovat pouze některé vstupy). Každý datový vstup a výstup uzlu je označen po sobě jdoucími celými čísly (začínajícími od 0), což určuje unikátní index, díky kterému lze přistupovat ke vstupním datům a výstupům uzlů. Převzato z [17].



Obrázek 4.5: CGP genotyp (chromozom) a korespondující fenotyp pro dvoubitovou násobičku. Podtržené geny v genotypu kódují funkci každého uzlu. Vyhledávací tabulka funkcí obsahuje AND (0), AND s jedním invertovaným vstupem (1), XOR (2) a OR (3). Pod každým vstupem a uzlem (v genotypu i fenotypu) je poznačena jeho adresa. Nekódující (neaktivní) oblasti genotypu a fenotypu jsou vyznačeny šedou čerchovanou čarou. Převzato z [17].

Operátor mutace

Operátor mutace používaný v CGP je bodový mutační operátor. To znamená, že se náhodně vybere pozice v chromozomu a hodnota genu na této pozici se změní na jinou validní hodnotu. Pokud je zvolen gen reprezentující funkci, pak se jeho hodnota změní na jakoukoliv hodnotu reprezentující funkci z dostupných funkcí. Pokud zvolený gen reprezentuje vstup uzlu, pak je nastavena hodnota genu na index náhodně zvoleného uzlu, který tomuto uzlu předchází, nebo na index vstupu. Obvykle se počet genů, které se zmutují udává relativně k délce chromozomu [17].

Prohledávací algoritmus

Standardní varianta CGP používá pouze mutaci a prohledávací strategii $(\mu + \lambda)$, kde $\mu = 1$ a $\lambda = 4$.

Kapitola 5

Neuroevoluce

Neuroevoluce je forma umělé inteligence, která využívá neuronové sítě v kombinaci s evolučními algoritmy. Evoluční algoritmy se v kontextu neuronových sítí využívají převážně pro trénování sítě, návrh struktury a optimalizaci parametrů sítě.

5.1 Trénování sítě

Pro trénování neuronových sítí bylo vynalezeno mnoho algoritmů, přičemž většina je založena na gradientu. Díky tomu může dojít k uváznutí v lokálním minimu. Řešením tohoto problému je například přidání šumu do vstupních dat, rušení vah nebo jejich znehodnocování.

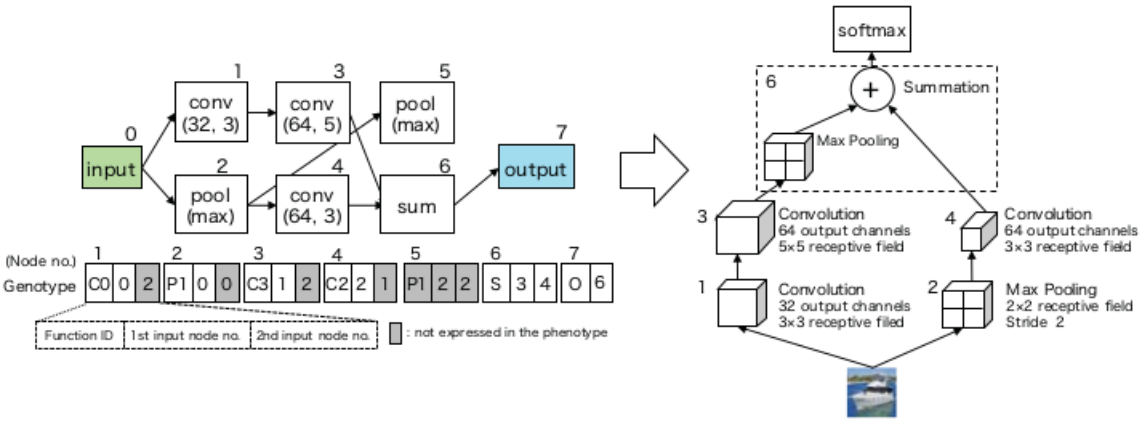
Na rozdíl od klasických metod učení sítě evoluční algoritmy nespoléhají na znalosti derivací, které jsou potřeba při klasickém učení, takže by u nich měla být menší pravděpodobnost uváznutí v lokálním minimu [15]. Obvykle se využívá evoluční strategie, přičemž geny chromozomu reprezentují jednotlivé váhy. Pro ohodnocení jedince je potřeba sestrojit neuronovou síť určité architektury, dle chromozomu nastavit příslušné váhy a síť otestovat. Po proběhnutí algoritmu je výstupem jedinec, jehož chromozom značí kombinaci vah dávající nejlepší výsledky.

5.2 Návrh architektury

Navrhování architektur hlubokých sítí je velmi obtížná úloha, protože existuje mnoho parametrů, které je nutno brát v potaz (například počet vrstev, typ a parametry každé vrstvy a konektivita vrstev). Současné sítě podávající nejlepší výkony jsou poměrně rozsáhlé a komplexní, což znamená, že počet parametrů, které je nutno optimalizovat, aby síť podávala co nejlepší výsledky, je velmi veliký. Takže se při návrhu architektur využívá metoda pokus-omyl nebo znalosti expertů. Z tohoto důvodu by byl značně výhodný automatický návrh architektur za využití evolučních algoritmů.

5.2.1 Kartézské genetické programování pro návrh architektury hluboké neuronové sítě

Následuje popis řešení tohoto problému dle odborné publikace [22]. Uvažujme, že je k dispozici jeden vstup reprezentující např. klasifikovaný obrázek, výstupní hodnota reprezentující např. příslušnost vstupu do určité kategorie a že jednotlivé uzly reprezentují jednu z dostupných vrstev využívaných v hlubokých neuronových sítí. Pak si lze představit, že cesta



Obrázek 5.1: Genotyp (nalevo) popisující fenotyp – architekturu (vpravo). Převzato z [22].

grafem od vstupu k výstupu reprezentuje danou architekturu neuronové sítě. Ilustrováno na obrázku 5.1.

Pro určení fitness jedince je potřeba architekturu sítě natrénovat, což je velmi výpočetně náročné. Fitness hodnota jedince je pak rovna úspěšnosti, se kterou je vytvořená síť schopna klasifikovat trénovací vzorky. Tento přístup lze v současnosti použít jen pro relativně malé CNN.

5.3 Optimalizace parametrů sítě

Obecně platí, že velké sítě obsahují mnoho uzlů a tedy i mnoho vah, zvláště pokud jsou neurony vrstev hustě propojeny. Síť může být pro určitou úlohu příliš složitá a nebo může obsahovat váhy, které nejsou při výpočtech potřeba. Obvykle je vstupem optimalizačního algoritmu natrénovaná síť a výstupem je nějakým způsobem optimalizovaná síť, která má sice určité parametry optimalizovány, avšak může poskytovat horší výsledky.

Eliminace nepotřebných vah může být provedena tak, v chromozomu tvořeném reálnými čísly každý gen přísluší k jedné konkrétní váze. Pokud je hodnota genu menší než nula, v síti se příslušná váha neuvažuje – je rovna nule, v opačném případě se v síti uvažuje váha původní. V podstatě je ke každému spojení neuronu přiřazen „vypínač“. Po proběhnutí evolučního algoritmu je výstupem chromozom, jehož hodnoty genů značí, které váhy v síti nehrají velkou roli [14].

Genetické algoritmy lze využít nejen pro optimalizaci samotné sítě, ale také pro optimalizaci učícího algoritmu. Lze pomocí nich nalézt výhodnější volbu parametrů daného algoritmu (např. míru učení) nebo určit, jak moc se mezi jednotlivými trénovacími epochami má míra učení snižovat. Do chromozomu je také možno zakomponovat gen, který určuje, v jakém rozsahu musí být náhodně vygenerované váhy při vytvoření sítě.

Dalším využitím je objevení samotného učícího algoritmu pomocí genetických algoritmů. Chromozom tedy kóduje algoritmus, dle kterého se upravují váhy [20].

Výše uvedené techniky byly zavedeny a zkoumány před nástupem hlubokých neuronových sítí. V současnosti je jejich využití pro složité neuronové sítě výpočetně těžko zvládnutelné.

Kapitola 6

Ověření funkčnosti knihovny na zvolených úlohách

Na řešení problému symbolické regrese bude v podkapitole 9.1 demonstrována funkčnost knihovny pro kartézské genetické programování. Zbylé úlohy slouží k ověření funkčnosti vybrané knihovny pro práci s konvolučními neuronovými sítěmi. Kapitola dále obsahuje porovnání výsledků konkrétních natrénovaných architektur sítí s výsledky popsanými v literatuře.

6.1 Symbolická regrese

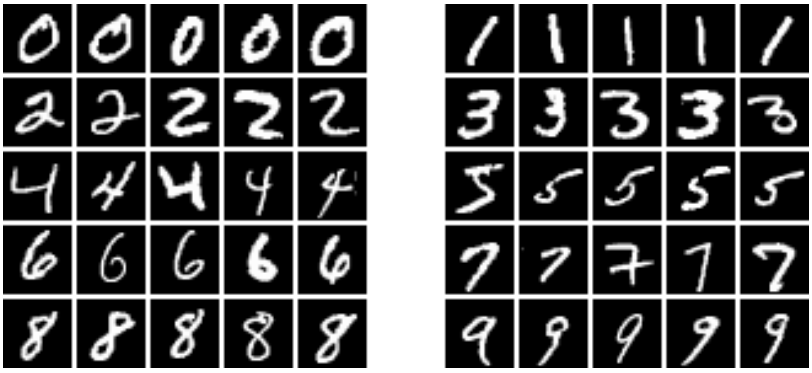
Jedním z problémů, které lze řešit pomocí genetického programování, je symbolická regrese. Hlavním cílem je najít model $\hat{f}(x)$, který dobře aproximuje hodnoty cílové proměnné y při daném počtu vstupních proměnných x. Hodnoty y jsou produkovány neznámou funkcí $f(x)$. Vstupem pro symbolickou regresi je množina hodnot každé proměnné zkoumaného systému a výsledkem je funkce $\hat{f}(x)$, získaná pomocí genetického programování, která je zakódovaná jako strom symbolických výrazů (sestavený například z $+$, $-$, $*$, $/$). Jedna z možných a často používaných fitness funkcí je střední čtvercová chyba (dále jen MSE) hodnot vypočtených pomocí modelu $\hat{f}(x)$ a původních zadaných hodnot y [10]. Cílem genetického programování je nalézt výraz, který minimalizuje

$$MSE = (\hat{f}, y) = \frac{1}{n} \sum_{i=1}^{n} (\hat{f}(x)_i - y_i)^2. \tag{6.1}$$

6.2 Klasifikace obrázků – MNIST

MNIST je databáze složená ze dvou databází SD-1 a SD-3. Původní NIST databáze byla navržena tak, že používala SD-3 jako trénovací sadu a SD-1 jako sadu testovací. Avšak SD-3 je mnohem kvalitnější a je v ní snazší rozlišování než v SD-1. To vyplývá z faktu, že vzorky v SD-3 byly nasbírány od zaměstnanců Amerického úřadu pro sčítání lidu a SD-1 je nasbírána mezi žáky středních škol. Pro získání rozumných výsledků z experimentů s databází je potřeba, aby byl výsledek nezávislý na volbě trénovací a testovací sady.

Trénovací sada MNIST je tvořena polovinou vzorků z SD-1 (zhruba 30000), která je doplněna vzorky z SD-3 tak, aby celkový počet vzorků byl 60000. Podobně je vytvořena testovací sada, která obsahuje vzorky z druhé poloviny SD-1 a je doplněna vzorky z SD-3



Obrázek 6.1: Ukázka obrázků z datové sady MNIST.

(jinými než při tvorbě sady trénovací) tak, aby také obsahovala 60000 vzorků. Výsledná databáze MNIST je však tvořena pouze 10000 vzorky z testovací sady, ale obsahuje celou sadu trénovací.

Původní černobílé obrázky obsahující číslice od 0 do 9 byly normalizovány na velikost 20x20 pixelů se zachováním poměru stran. Výsledné obrázky z důvodů vyhlazení (interpolace) obsahují odstíny šedi. Obrázky jsou poté umístěny vycentrovány v obrázku 28x28 pixelů s využitím těžiště pixelů [13]. Ilustrace vzorků z datové sady je na obrázku 6.1.

6.2.1 Použité architektury sítí

Pro účely této práce byla navržena poměrně jednoduchá konvoluční neuronová síť, která sama o sobě dosahuje horších výsledků, než jsou výsledky jiných sítí řešící tento klasifikační problém, ale první konvoluční vrstva by měla být poměrně snadno aproximovatelná. V textu budeme tuto architekturu označovat jako *jednoduchá* síť. Vrstvy využité v této síti:

- Konvoluční vrstva s 32 filtry a velikostí okénka 3×3.
- Relu aktivační funkce.
- Podvzorkovací vrstva s velikostí okénka 2×2.
- Vynechávající vrstva s pravděpodobností $\frac{1}{5}$.
- Zplošťovací vrstva.
- Plně propojená vrstva se 128 neurony.
- Relu aktivační funkce.
- Plně propojená vrstva s 10 neurony (počet tříd).
- Softmax aktivační funkce.

Další architekturou je tzv. LeNet. LeNet označuje celou rodinu architektur konvolučních neuronových sítí, které se používají nejen ke klasifikaci vzorků z MNIST databáze. Ve srovnání s jinými sítěmi se jedná o sítě poměrně malé, ale i přes to dosahují kvalitních výsledku. Níže popsané architektura je inspirována LeNet5 architekturou, která je ilustrována na obrázku 3.3. Na tuto architekturu se dále budeme odkazovat jako na *LeNet* síť. Vrstvy využité v této síti:

- Konvoluční vrstva se 6 filtry a velikostí okénka 5×5.
- Aktivační funkce hyperbolický tangens.
- Podvzorkovací vrstva s velikostí okénka 2×2.
- Konvoluční vrstva s 16 filtry a velikostí okénka 5×5.
- Aktivační funkce hyperbolický tangens.
- Podvzorkovací vrstva s velikostí okénka 2×2.
- Vynechávající vrstva s pravděpodobností $\frac{1}{4}$.
- Konvoluční vrstva se 120 filtry a velikostí okénka 1×1.
- Aktivační funkce hyperbolický tangens.
- Zplošťovací vrstva.
- Plně propojená vrstva s 84 neurony.
- Aktivační funkce hyperbolický tangens.
- Vynechávající vrstva s pravděpodobností $\frac{1}{2}$.
- Plně propojená vrstva s 10 neurony (počet tříd).
- Softmax aktivační funkce.

Tabulky 6.1 a 6.2 zobrazují úspěšnost naučených sítí při klasifikování 10000 testovacích vzorků. Učení každé architektury bylo provedeno 5-krát, pokaždé pro 12 epoch. Výsledky jsou srovnatelné s výsledky podobných sítí uvedených v literatuře [1].

Běh	Úspěšnost
1	98.71 %
2	98.78 %
3	98.76 %
4	98.83 %
5	98.74 %

Tabulka 6.1: Vyhodnocení úspěšnosti jednoduché sítě. Průměrná úspěšnost je 98.76 %.

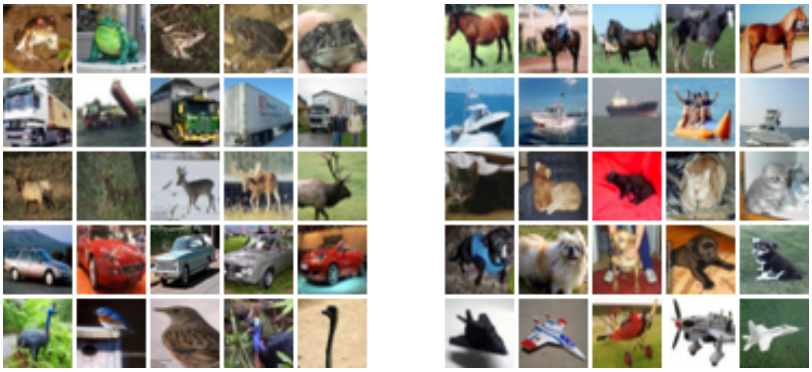
Běh	Úspěšnost
1	98.64 %
2	98.76 %
3	98.66 %
4	98.76 %
5	98.87 %

Tabulka 6.2: Vyhodnocení úspěšnosti LeNet sítě. Průměrná úspěšnost je 98.73 %.

6.3 Klasifikace obrázků – CIFAR-10

CIFAR-10 a CIFAR-100 jsou podmnožiny databáze obrázku zvané *80 million tiny images dataset*. CIFAR-10 se skládá z 10 tříd, přičemž každá obsahuje 6000 obrázků (60000 obrázků celkem). Trénovací množina obsahuje 50000 a testovací 10000 obrázků.

Třída u obrázku značí, zda se na obrázku vyskytuje letadlo, automobil, pták, kočka, jelen, pes, žába, kůň, loď nebo kamión. Dále existuje varianta CIFAR-100, která obsahuje 100 tříd a pro každou třídu 600 obrázků [9]. Ilustrace vzorků z datové sady je na obrázku 6.2.



Obrázek 6.2: Ukázka obrázků z datové sady CIFAR-10.

6.3.1 Použité architektury sítí

Architektura pro klasifikaci vzorků z databáze CIFAR-10 pochází z github repozitáře knihovny Keras [1]. Popis architektury:

- Konvoluční vrstva s 32 filtry a velikostí okénka 3×3.
- Relu aktivační funkce.
- Konvoluční vrstva s 32 filtry a velikostí okénka 3×3.
- Relu aktivační funkce.
- Podvzorkovací vrstva s velikostí okénka 2×2.
- Vynechávající vrstva s pravděpodobností $\frac{1}{4}$.
- Konvoluční vrstva s 64 filtry a velikostí okénka 3×3.
- Relu aktivační funkce.
- Konvoluční vrstva s 64 filtry a velikostí okénka 3×3.
- Relu aktivační funkce.
- Podvzorkovací vrstva s velikostí okénka 2×2.
- Vynechávající vrstva s pravděpodobností $\frac{1}{4}$.

[1] https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py

- Zplošťovací vrstva.
- Plně propojená vrstva se 512 neurony.
- Relu aktivační funkce.
- Vynechávající vrstva s pravděpodobností $\frac{1}{2}$.
- Plně propojená vrstva s 10 neurony (počet tříd).
- Softmax aktivační funkce.

Architektura byla 5-krát natrénována a úspěšnost výsledných modelů je zachycena v tabulce 6.3. Průměrná, experimentálně zjištěná, úspěšnost sítě při klasifikaci je rovna 77.71 %, což v porovnání s jinými sítěmi, které dosahují úspěšnosti až 96.53 % [2] nesrovnatelné, ale tyto sítě jsou mnohem rozsáhlejší a složitější než výše popsaná, která pro potřeby této práce postačí. Dále označujeme architekturu jako *C10* síť.

Běh	Úspěšnost
1	78.15 %
2	78.35 %
4	76.59 %
3	78.26 %
5	77.23 %

Tabulka 6.3: Vyhodnocení úspěšnosti C10 sítě. Průměrná úspěšnost je 77.71 %.

Kapitola 7

Návrh optimalizace CNN

Cílem této práce je navrhnout optimalizaci, která bude zlepšovat určité parametry sítě. Konkrétně se budeme snažit nahradit operaci konvoluce aproximací, jejíž výpočet bude jednodušší při obvodové realizaci. Konkrétně se jedná o eliminaci operace násobení, která je výpočetně poměrně náročná. Toto zjednodušení by mělo vést k redukci příkonu CNN, což je důležité pro realizaci CNN v nízkopříkonových vestavěných systémech.

Navržený přístup je inspirován výsledky v oblasti evolučního návrhu obrazových filtrů [21], kde bylo ukázáno, že řešení založené na konvoluci lze nahradit výpočtem založeným na sčítání a logických operacích, tj. bez využití operace násobení.

Abychom mohli určitým způsobem srovnat složitost (počet tranzistorů) původní konvoluce konvoluce se složitostí sítě získané pomocí algoritmu CGP (aproximace filtru počítající na 8 bitech), budeme uvažovat, že původní konvoluční vrstva využívala 8 bitů, tzn. vstupy do sítě byly na 8 bitech, stejně tak jako váhy jednotlivých filtrů a výstupy vrstvy.

V následujících podkapitolách budou znázorněna schémata logických operací na 8 bitech a pro každou operaci bude určeno, kolik tranzistorů je potřeba pro jejich implementaci. Odhad efektivity optimalizace bude realizován srovnáním počtu tranzistorů potřebných pro výpočet konvoluce na 8 bitech s počtem tranzistorů využitých v aproximaci konvolučního filtru pomocí CGP.

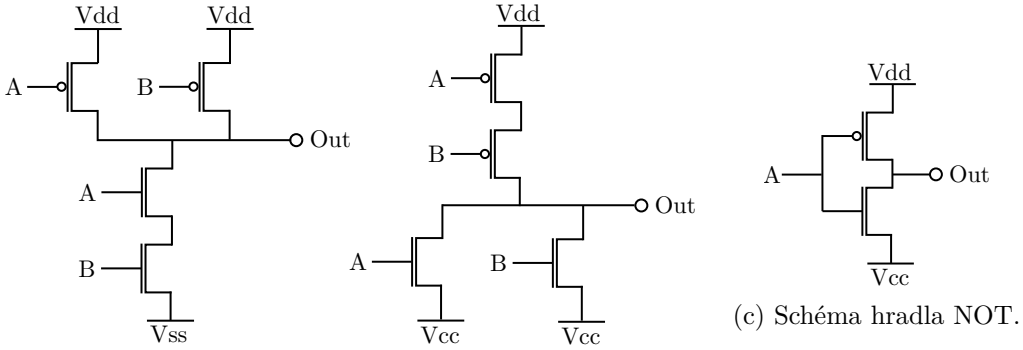
Zdůrazňujeme, že se jedná o odhad počtu tranzistorů. Přesnější výpočet by vyžadoval uvážit další technologií ovlivněné parametry, což je mimo záměr této diplomové práce. Pro odhad příkonu budeme předpokládat, že podíl původního počtu tranzistorů a počtu tranzistorů aproximace získané pomocí CGP zhruba odpovídá podílu příkonů původní operace konvoluce a aproximace konvoluce.

7.1 Odhad počtu tranzistorů logických obvodů

V této kapitole bude pro hradla NAND, NOR, NOT a XOR určen počet tranzistorů, které jsou potřeba pro jejich obvodovou realizaci. S využitím těchto hradel bude určen počet tranzistorů, ze kterých se skládají složitější funkce.

7.1.1 Základní hradla

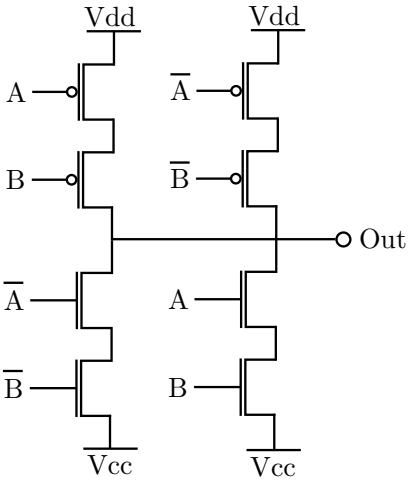
Popis všech logických funkcí je založen na tzv. CMOS, což značí způsob návrhu, ve kterém se využívají tranzistory MOSFET typu n a p. Jak lze vidět ze schémat (viz 7.1), hradlo NAND a NOR se skládá ze 4 tranzistorů.



(a) Schéma hradla NAND. (b) Schéma hradla NOR. (c) Schéma hradla NOT.

Obrázek 7.1: Schémata hradel NAND, NOR a NOT.

Hradla AND a OR jsou implementována pouhým přidáním hradla NOT – jsou složena ze 6 tranzistorů. Schéma hradla XOR je zobrazeno na obrázku 7.2 a je složeno z 12 tranzistorů – $\overline{A}$ a $\overline{B}$ značí negované hodnoty.



Obrázek 7.2: Schéma hradla XOR.

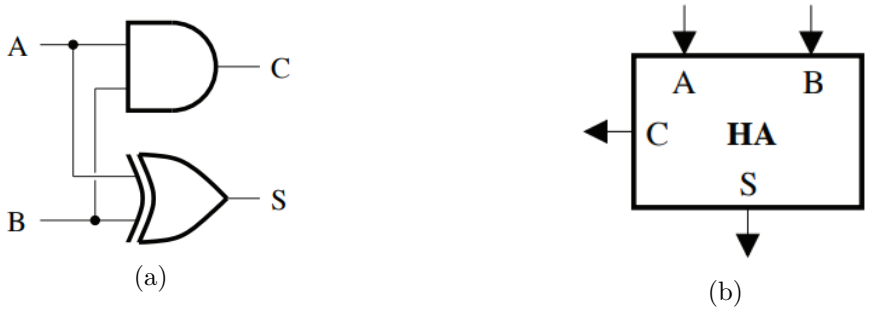
.

7.1.2 Sčítačky

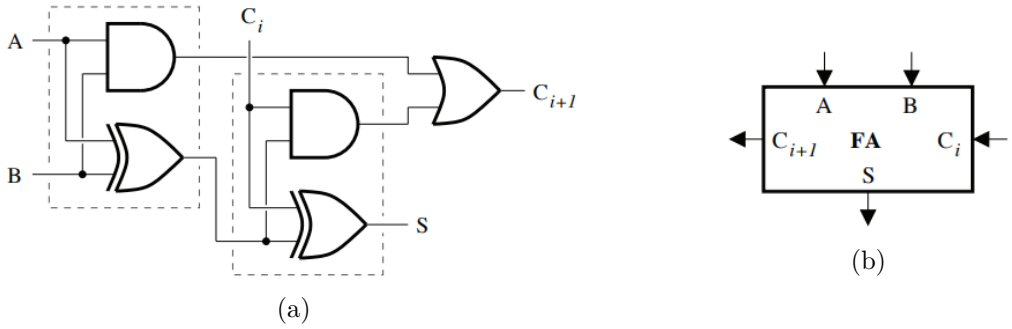
Na obrázku 7.3 je zobrazena poloviční sčítačka (HA) složena z 18 tranzistorů – 6 pro AND hradlo a 12 pro hradlo XOR. Úplná sčítačka (FA) na obrázku 7.4 obsahuje 42 tranzistorů – 24 pro 2 XOR hradla a 18 pro hradla AND, AND a OR.

7.1.3 Osmibitová sčítačka s postupným přenosem

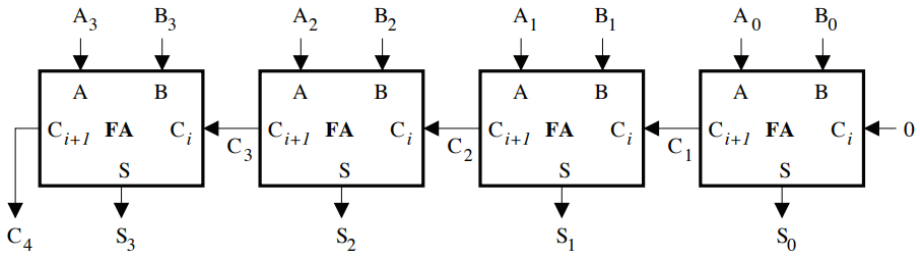
Čtyřbitová sčítačka s postupným přenosem je zobrazena na obrázku 7.5. Osmibitová varianta by byla sestavena totožně jako čtyřbitová, obsahovala by však 8 úplných sčítaček, tzn. 336 tranzistorů.



Obrázek 7.3: Schéma a symbol poloviční sčítačky. Převzato z [18].



Obrázek 7.4: Schéma a symbol úplné sčítačky. Převzato z [18].

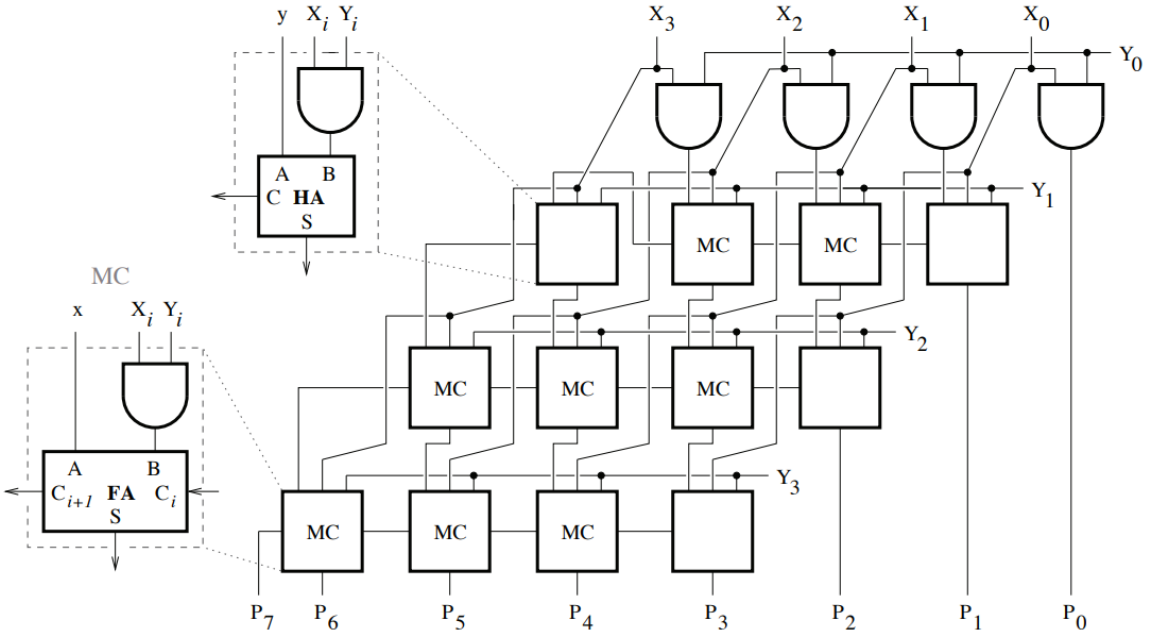


Obrázek 7.5: Čtyřbitová sčítačka s přenosem.Převzato z [18].

.

7.1.4 Osmibitová násobička

Čtyřbitová násobička složená z polovičních a úplných sčítaček je zobrazena na obrázku 7.6. Při násobení dvou n-bitových je potřeba n^2 AND hradel, n polovičních sčítaček a $n^2 - 2n$ úplných sčítaček [18]. Což je pro dvě 8 bitová čísla rovno 64 AND hradel, 8 polovičních sčítaček a 48 úplných sčítaček. Osmibitová násobička je tedy složena z 2544 tranzistorů.



Obrázek 7.6: Čtyřbitová násobička.Převzato z [18].

.

7.2 Operace použité v CGP

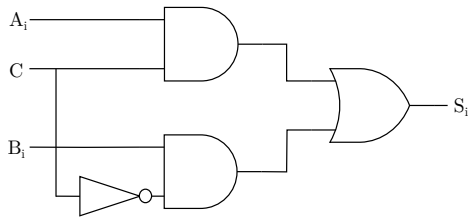
Tato podkapitola popisuje operace realizované bloky sítě CGP. Zároveň je u každé operace určen, případně odhadnut, počet tranzistorů potřebných pro jejich implementaci v hardware. Následuje tabulka s operacemi, kde x značí první vstup a y druhý vstup:

Operace	# tranzistorů	Poznámky
$255 - x$	16	8x NOT pro bity z x
$\overline{x} \vee y$	64	8x NOT pro x, 8x OR
$x \wedge y$	48	8x AND
$\overline{x \wedge y}$	32	8x NAND
$x \oplus y$	96	8x XOR
$x \gg 1$	0	Výsledný bit i je napojen na $i+1$ bit z x
$x \gg 2$	0	Výsledný bit i je napojen na $i+2$ bit z x
$x + y$	336	8 bitová sčítačka
$max(x, y)$	512	8x NOT pro bity z y, 8 bitová sčítačka, 8x obvod ze 7.7
$min(x, y)$	512	Podobně jako max(x,y), ale je prohozeno A_i za B_i
$x +^s y$	496	viz 7.2.2
$(x + y) \gg 1$	336	Výsledný bit i je napojen na $i+1$ bit výsledku 8 bitové sčítačky

Tabulka 7.1: Funkce využívané bloky v CGP a počet elementárních hradel potřebných pro jejich implementaci.

7.2.1 Maximum ze dvou čísel

Platnost nerovnosti $A > B$ je zjištěna tak, že do 8 bitové sčítačky vstupují bity čísla A a negované bity čísla B a podle výstupního carry bitu C sčítačky jsme schopni určit, zda nerovnost platí. Pokud platí $C = 1$, pak je $A > B$, jinak je $A < B$. Ilustrace schémata, které bude mít na výstupu jeden bit z čísla, které je větší je zobrazeno na obrázku 7.7. Pro 8 bitová čísla je potřeba 8 těchto logických obvodů, přičemž jeden je složen z 20 tranzistorů.



Obrázek 7.7: Schéma obvodu, který má na výstupu bit většího čísla.

.

7.2.2 Saturovaný součet

Pokud je výstupní carry bit C sčítačky roven 1, pak je každý bit výstupu roven 1, jinak je roven bitu výstupu sčítačky. Schéma pro určení výstupního bitu je podobné jako 7.7, avšak A_i je rovno bitu výstupu sčítačky a B_i je rovno logické 1. Jedná se tedy o jednu osmibitovou sčítačku a 8x zmíněné schéma - tedy 496 tranzistorů.

7.3 Nahrazení konvoluční vrstvy

Získání sítě, ve které bude zadaná konvoluční vrstva nahrazeny aproximační vrstvou, lze zapsat následujícím pseudo algoritmem:

Data: Původní konvoluční síť
Result: Síť, ve které je konvoluční vrstva nahrazena aproximací
Načti model ze souboru obsahující natrénovanou neuronovou síť;
Vytvoř množinu vstupních dat V pro neuronovou síť;
Vytvoř prázdnou síť N_p – značí částečnou síť;
Vytvoř prázdnou síť N_r – značí výslednou síť;
foreach *vrstva* $\in$ *načtený model* **do**
    Přidej vrstvu do N_p;
    if *vrstva má být nahrazena* **then**
        if *vrstva má být načtena ze souboru* **then**
            Načti vrstvu ze souboru a přidej ji do N_r
        else
            Získej výstupy sítě N_p pro vstupy V;
            Nastav parametry pro CGP;
            S využitím vstupů V a získaných výstupů vypočítej pomocí CGP aproximace všech konvolučních filtrů vrstvy a vytvoř z nich vrstvu V_{cgp};
            Přidej V_{cgp} do N_r;
            Ulož vrstvu tvořenou aproximacemi do souboru společně s daty popisující detaily konverze;
        end
    end
end
Vrať síť N_r;

Algorithm 4: Pseudokód náhrady konvolučních vrstev.

Kapitola 8

Poznámky k implementaci

V této kapitole budou popsány poznámky k implementaci knihovny pro kartézské genetické programování a získání aproximace konvoluční vrstvy.

8.1 Knihovna pro CGP

Pro potřeby této práce byla kompletně naimplementována knihovna pro kartézské genetické programování napsaná v jazyce c++. Knihovna je umístěna ve složce src/CGPEvolver a přeložena spuštěním příkazu `make`.

Při spuštění algoritmu kartézského genetického programování je vygenerována počáteční populace – pole chromozomů, přičemž každý chromozom reprezentuje graf, který odpovídá implementaci konkrétního programu řešícího danou úlohu. Každý chromozom je tvořen geny, které mají určité hodnoty reprezentující funkci, kterou bude každý blok sítě vykonávat a vstupy, na které je blok připojen.

Dalším krokem je vypočtení hodnoty fitness každého chromozomu. Pro každý vstup do CGP (viz kapitola 8.1.2) je pro každou síť, reprezentovanou určitým chromozomem, získán výsledek, který je porovnán s výsledkem, který je poskytnut konvolučním filtrem, který se snažíme nahradit. Fitness hodnota tohoto chromozomu $f(c)$ je určena jako:

$$f(c) = \sum_{i=1}^{n} (x_i - y_i)^2, \tag{8.1}$$

kde x značí pole originálních výsledků velikosti n a y značí pole velikosti n výsledků získaných aproximací. U každého chromozomu je uchována informace o tom, zda jeho fitness hodnota již byla vypočtena společně s fitness hodnotou samotnou. Pokud je do nové populace vybrán jedinec jehož hodnota již byla vypočtena, není ji třeba znovu přepočítávat. Fitness hodnota se nepočítá ani v případě, že máme 2 jedince, přičemž jeden jedinec má vypočtenou fitness hodnotu a druhý ne, avšak sítě reprezentované oběma jedinci využívají stejné bloky (i se vstupy) pro získání výsledku. Fitness hodnota jedince s nevypočítanou fitness je rovna fitness hodnotě jedince s vypočtenou fitness.

Následně je z celé populace vybrán jedinec s nejlepší fitness hodnotou, který se stane rodičem potomků v nové generaci. Vytvoření jedinců je řešeno tak, že se do nové generace velikosti n zařadí rodič a $n - 1$ potomků získaných aplikací operátoru mutace na rodiče. Pokud při výběru rodiče pro novou generaci nastane situace, že jsou fitness hodnoty rodiče dané generace a potomka, z tohoto rodiče vytvořeného shodné, vybere se jako rodič následující generace potomek.

Po vytvoření a vyhodnocení určitého počtu generaci je výsledkem jedinec, jehož síť nejlépe aproximuje převod vstupů na výstupy originální funkce – v našem případě konvoluce. Algoritmus CGP je spuštěn paralelně na n procesorech, takže je výsledkem n nejlepších aproximací, ze kterých je vybrána ta celkově nejlepší.

Ukázka použití knihovny (pro přehlednost proveden pouze jeden běh CGP):

```
std::vector<std::vector<double>> inputs;
std::vector<std::vector<double>> outputs;
Settings settings;
// ... naplň inputs a outputs trénovacími hodnotami; nastav parametry cgp

Approx<double> approximation = runCgp(settings, inputs, outputs, 0);

std::vector<std::vector<double>> testInputs;
// ... naplň testInputs testovacími hodnotami

auto result = approximation.getOutput(testInputs);
// ... result obsahuje výsledek sítě získané pomocí cgp
```

8.1.1 Parametry CGP

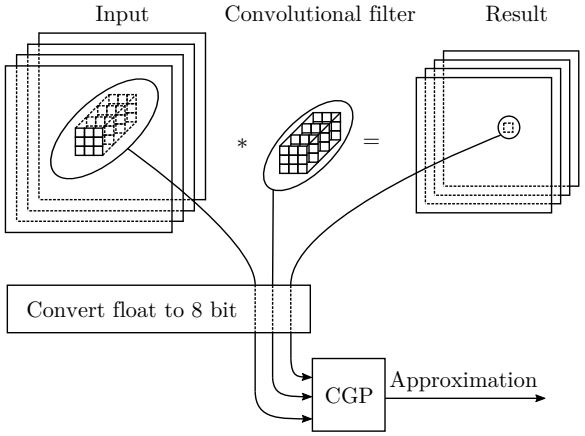
Jeden ze vstupů algoritmu CGP jsou jeho parametry (nastavení). Parametry udávají:

- Počet jedinců v populaci
- Počet generací
- Počet sloupců a řádků sítě bloků
- Pravděpodobnost, se kterou dojde k mutaci genu
- Parametr l-back, který udává, z kolika předchozích sloupců může blok získávat vstup
- Počáteční hodnota pro generátor náhodných čísel
- Zda konvoluční vrstva používá bias
- Počet běhů algoritmu CGP
- Počet vstupů
- Počet výstupů

8.1.2 Vytvoření vstupů a výstupů pro CGP

Algoritmus získání aproximace původní konvoluční sítě (vrstvy obsahující aproximované konvoluční filtry) je spuštěn po zavolání funkce `createUInt8Layer` s parametry pro CGP algoritmus, referenčními vstupy a odpovídajícími výstupy (viz algoritmus 4), váhami a bias hodnotami filtrů původní vrstvy.

Vzhledem k tomu, že funkce implementované uzly v CGP počítají s 8 bitovými čísly je potřeba vstupy původní vrstvy, váhy a referenční výstupy převést převést na 8 bitová čísla – provedeno pro každý filtr v konvoluční vrstvě zvlášť. Převod je vyřešen tak, že se



Obrázek 8.1: Ilustrace vytvoření vstupních dat pro CGP. Vstupy CGP jsou tvořeny: vstupy, které vstupují do operace konvoluce, váhy konvolučního filtru, bias hodnota daného konvolučního filtru a nastavení CGP. Číselné vstupy do CGP musí být upraveny do rozsahu 0-255. Výstupem CGP je aproximace původního konvolučního filtru.

najde nejmenší číslo min ve vstupní množině, tomuto číslu bude odpovídat hodnota 0, a největšímu číslu v množině bude odpovídat hodnota max rovna 255. Ostatní čísla jsou převedena do rozsahu $0 - 255$ dle vzorce 8.2.

$$f(x) = \frac{255(x - min)}{max - min} \tag{8.2}$$

.

Každý filtr z konvoluční vrstvy je potřeba evolvovat pomocí CGP sám o sobě, přičemž vstupy jsou pro všechny filtry stejné. Vstupy pro CGP jsou vygenerovány obdobně jako hodnoty, které vstupují do matematické operace konvoluce v trojrozměrném prostoru. Posuvem okénka o velikosti (x, y) nad vstupem získáme obecně tenzor o velikosti (x, y, z), kde z je hloubka vstupu. S tímto tenzorem a konvolučním filtrem provedeme operaci konvoluce a získáme výsledek, od kterého odečteme bias hodnotu konkrétního filtru. Jako vstupní data do CGP se uvažují právě tyto tenzory (avšak transformované na rozsah 0-255) a příslušný výsledek konvoluce, (taktéž upravený na rozsah 0-255). Ke všem takto vygenerovaným vstupům pro CGP se dále přidají váhy konkrétního konvolučního filtru (v rozsahu 0-255), jenž se snažíme aproximovat. Vytváření vstupů a výstupů je ilustrováno na obrázku 8.1. Po získání výsledku pomocí aproximace původního konvolučního filtru je potřeba nad tento výsledek převést z rozsahu 0-255 do původního rozsahu a přičíst k němu bias hodnotu daného konvolučního filtru.

Kapitola 9

Experimentální vyhodnocení metody

V této kapitole bude ověřena korektnost implementace kartézského genetického programování na úloze symbolické regrese. Dále je vyhodnoceno, jaký vliv má nahrazení konvolučních vrstev aproximacemi získanými pomocí CGP na cenu implementace a přesnost klasifikace sítě. Vyhodnocení je realizováno jako odhad, výsledná cena implementace operací závisí na mnoha faktorech, které zde neuvažujeme.

Experimenty byly vyhodnocovány na výpočetním zařízení obsahující 4 GB paměti RAM, čtyřjádrový procesor Intel(R) Core(TM) i5-2430M běžící na frekvenci 2.40GHz a operační systém Ubuntu. Jelikož je pro každý filtr spuštěn algoritmus CGP několikrát (na více jádrech procesoru), dochází k tomu, že počáteční hodnota pro generátor náhodných čísel hodnota nehraje velkou roli, protože se využívá funkce `random`, která není tzv. *thread safe*. Nelze tedy přesně reprodukovat popsané experimenty, avšak po jejich spuštění by mělo být dosaženo podobných výsledků.

9.1 Symbolická regrese

Pokud je knihovna pro kartézské genetické programování korektně naimplementována, pak by měla být schopna poskytnout síť bloků, která bude aproximovat původní funkci. V ideálním případě bude síť přesně modelovat původní funkci.

Jako funkce, která má být aproximována pro ověření implementace CGP byla zvolena tato polynomiální funkce (viz obrázek 9.1):

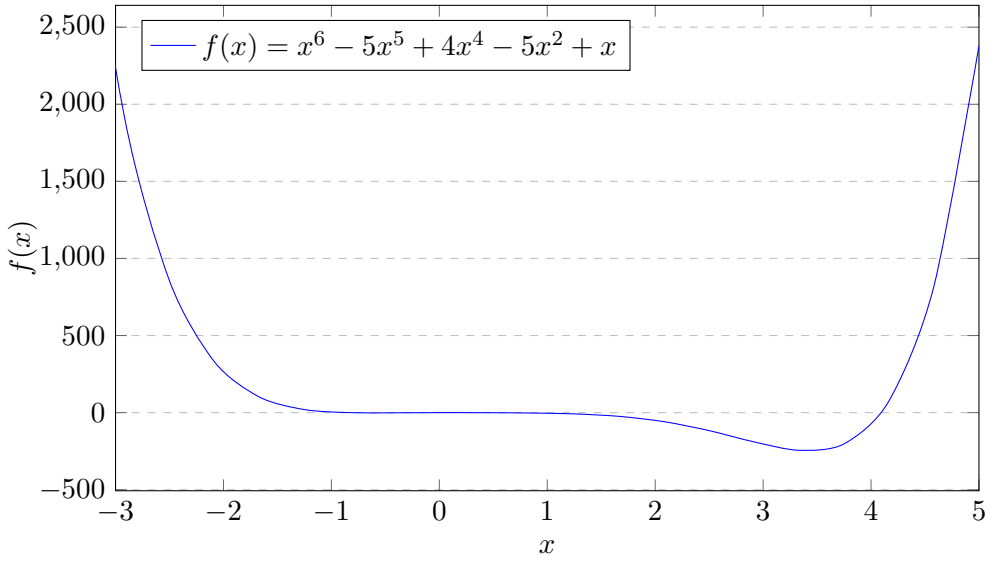
$$f(x) = x^6 - 5x^5 + 4x^4 - 5x^2 + x \tag{9.1}$$

Jako vstupy CGP bylo vybráno 28 hodnot $x \in < -1.5, 4 >$ a příslušné $f(x)$. Fitness hodnota každého kandidátního řešení je vypočtena dle vzorce:

$$fitness = \sum_{x} (\hat{f}(x) - f(x))^2, \tag{9.2}$$

kde $\hat{f}(x)$ značí výsledek, který poskytne síť reprezentovaná kandidátním řešením.

Po vygenerování množiny trénovacích vstupů x a trénovacích výstupů $f(x)$ byl spuštěn algoritmus CGP s parametry uvedenými v tabulce 9.1.

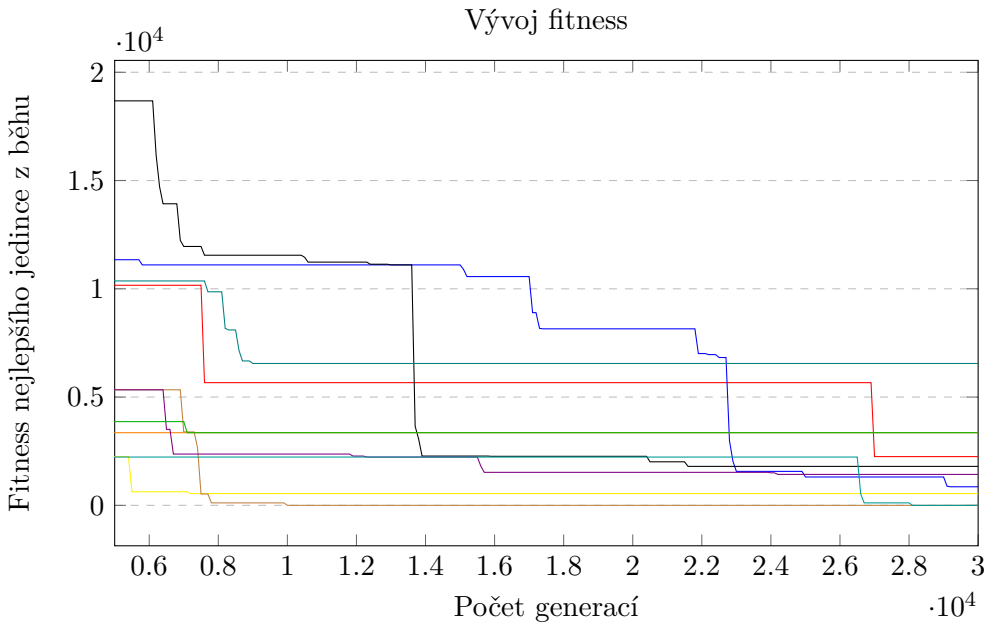


Obrázek 9.1: Funkce použitá pro úlohu symbolické regrese.

Parametr	Hodnota
Velikost populace	5
Počet generací	30000
Počet sloupců	10
Počet řádků	4
Pravděpodobnost mutace	0.01
Parametr lback	0
Počet běhů CGP na filtr	10

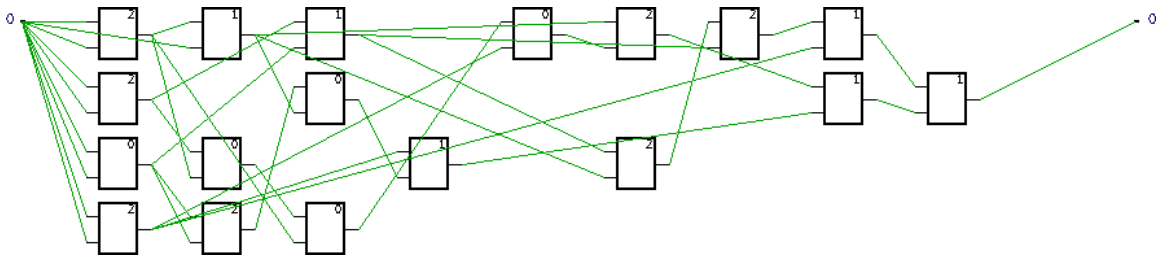
Tabulka 9.1: Parametry pro CGP řešící úlohu symbolické regrese.

Graf na obrázku 9.2 zobrazuje postupné vylepšování fitness hodnoty nejlepších jedinců z jednotlivých běhů CGP. Fitness hodnoty jedinců počátečních generací jsou v tomto případě poměrně vysoké, vykresluje se tedy až od 5000. generace.



Obrázek 9.2: Vývoj fitness při řešení symbolické regrese pomocí CGP.

Ze všech běhů algoritmu je vybrán jedinec s nejlepší fitness hodnotou – v tomto případě jedinec s nulovou fitness, což značí nulovou odchylku referenčních výstupů $f(x)$ a výstupů nejlepšího kandidátného řešení $\hat{f}(x)$. Na obrázku 9.3 je zobrazeno schéma sítě reprezentované nejlepším jedincem.



Obrázek 9.3: Schéma sítě reprezentované nejlepším jedincem ze všech běhů CGP. Je využita množina funkcí: sčítání (0), odčítání (1), násobení (2).

9.2 Náhrada konvolučních vrstev – MNIST

V následující podkapitole budou provedeny experimenty s nahrazováním vrstev sítí určených pro klasifikaci datasetu MNIST. Jedná se o náhradu první vrstvy jednoduché sítě a náhradu druhé vrstvy sítě LeNet.

9.2.1 Náhrada první vrstvy v jednoduché síti

V tomto experimentu ověřujeme, zda je možno nahradit poměrně jednoduchou konvoluční vrstvu aproximací získanou pomocí CGP s minimálním dopadem na přesnost sítě při kla-

sifikaci. Vrstva, kterou budeme nahrazovat, má vstupy o rozměrech $28 \times 28 \times 1$ a 32 filtrů o velikosti $3 \times 3 \times 1$. Počet vstupů sítě bloků je tedy 9 vah a 9 vstupních hodnot. Algoritmus CGP byl spuštěn s parametry uvedenými v tabulce 9.2. Ze 3 trénovacích vzorků datové sady MNIST bylo získáno 2028 trénovacích vzorků pro algoritmus CGP.

Parametr	Hodnota
Počet vstupů	18
Počet výstupů	1
Velikost populace	5
Počet generací	30000
Počet sloupců	4
Počet řádků	4
Pravděpodobnost mutace	0.1
Parametr lback	0
Počet běhů CGP na filtr	8

Tabulka 9.2: Parametry pro CGP řešící aproximaci první vrstvy v jednoduché síti.

Náhrada vrstvy a zjištění úspěšnosti při klasifikaci byla provedena celkem 5x. Tabulka 9.3 zobrazuje přesnost sítě s využitím získaných aproximačních vrstev na datové sadě MNIST.

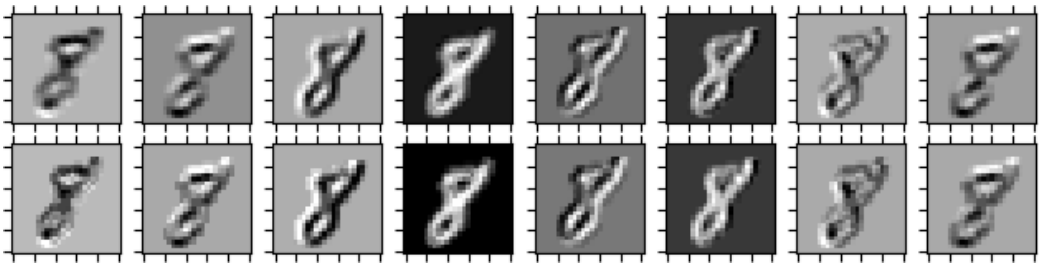
Typ vrstvy/sítě	Přesnost klasifikace v %	Doba výpočtu CGP v minutách
Originální síť	98.71	- - -
1. běh aproximace	98.57	82.6
2. běh aproximace	98.47	84.6
3. běh aproximace	98.37	79.4
4. běh aproximace	98.28	73.2
5. běh aproximace	98.43	72.8

Tabulka 9.3: Přesnost klasifikace jednoduché sítě obsahující původní konvoluční vrstvu a přesnost sítí s aproximovanou vrstvou při klasifikační úloze MNIST.

Pro získání výsledku násobení původního konvolučního filtru se vstupy je potřeba 288 násobení (filtr $3 \times 3 \times 1$ a 32 filtrů) a 256 sčítání. Tabulka 9.4 zachycuje počet funkcí, které jsou využity v sítích aproximujících jednotlivé filtry konvoluční vrstvy. Zároveň je určeno, kolik tranzistorů je využito při jejich výpočtu.

Počet operací	Originální síť	1. běh	2. běh	3. běh	4. běh	5. běh
8-bitová sčítačka	256	0	0	0	0	0
8-bitová násobička	288	0	0	0	0	0
$255 - x$	0	18	19	15	14	18
$\overline{x} \vee y$	0	20	16	14	11	13
$x \wedge y$	0	6	6	8	3	7
$\overline{x \wedge y}$	0	13	12	16	16	14
$x \oplus y$	0	6	13	14	16	14
$x \gg 1$	0	6	9	6	4	12
$x \gg 2$	0	7	6	6	9	7
$x + y$	0	6	8	6	13	8
$max(x, y)$	0	9	11	7	11	8
$min(x, y)$	0	13	9	3	5	8
$x +^s y$	0	8	9	14	12	8
$(x + y) \gg 1$	0	103	105	110	104	107
Počet využitých tranzistorů	818688	54704	55920	54416	56576	54048
Odhadované snížení příkonu v %	0	93.32	93.17	93.35	93.09	93.40

Tabulka 9.4: Počet využitých funkcí a tranzistorů v první konvoluční vrstvě jednoduché sítě a v aproximacích této vrstvy. Zároveň je odhadnuto snížení příkonu.



Obrázek 9.4: Ilustrace aproximace (dole) 8 konvolučních filtrů (nahoře).

Vyhodnocení

V tomto experimentu byla nahrazena první vrstva jednoduché sítě. Došlo k mírnému poklesu přesnosti klasifikace (zhruba o 0.3 %), za to byla dramaticky snížena složitost implementace, což odpovídá odhadovanému snížení příkonu o cca 93.27 %.

Na obrázku 9.4 je ilustrován výsledek operace konvoluce 8 konvolučních filtrů aplikovaných na jeden vzorek datové sady MNIST a výsledek příslušných aproximací aplikovaných na téže vzorek.

9.2.2 Nahrazení druhé konvoluční vrstvy v sítí LeNet

V tomto experimentu se budeme snažit nahradit druhou konvoluční vrstvu sítě LeNet. Jedná se o vrstvu s větším konvolučním filtrem než byl filtr v předchozím experimentu, tudíž se poměrně hodně zvětšil prohledávací stavový prostor. Vrstva má vstupy o rozměrech $12 \times 12 \times 6$ a filtr velikosti $5 \times 5 \times 6$. Počet vstupů sítě bloků je tedy 150 vah a 150 hodnot.

Algoritmus CGP byl spuštěn s uvedenými v tabulce 9.5. Z 20 trénovacích vzorků datové sady MNIST bylo získáno 1280 trénovacích vzorků pro algoritmus CGP.

Parametr	Hodnota
Počet vstupů	300
Počet výstupů	1
Velikost populace	5
Počet generací	60000
Počet sloupců	9
Počet řádků	4
Pravděpodobnost mutace	0.1
Parametr lback	4
Počet běhů CGP na filtr	12

Tabulka 9.5: Parametry pro CGP řešící aproximaci druhé vrstvy v síti LeNet.

Náhrada vrstvy a zjištění úspěšnosti při klasifikaci byla provedena celkem 5x. Tabulka 9.6 zobrazuje přesnost sítě s využitím získaných aproximačních vrstev na datové sadě MNIST.

Typ vrstvy/sítě	Přesnost klasifikace v %	Doba výpočtu CGP v minutách
Originální síť	98.64	- - -
1. běh aproximace	95.89	139.3
2. běh aproximace	95.78	144.5
3. běh aproximace	95.06	137.2
4. běh aproximace	96.55	139.9
5. běh aproximace	95.15	140.7

Tabulka 9.6: Přesnost klasifikace sítě LeNet obsahující původní konvoluční vrstvu a přesnost sítí s aproximovanou vrstvou při klasifikační úloze MNIST.

Pro získání výsledku násobení původního konvolučního filtru se vstupy je potřeba 2400 násobení (filtr $5 \times 5 \times 6$ a 16 filtrů) a 2384 sčítání. Tabulka 9.7 zachycuje počet funkcí, které jsou využity v sítích aproximujících jednotlivé filtry konvoluční vrstvy. Zároveň je určeno, kolik tranzistorů je využito při jejich výpočtu.

Počet operací	Originální síť	1. běh	2. běh	3. běh	4. běh	5. běh
8-bitová sčítačka	2384	0	0	0	0	0
8-bitová násobička	2400	0	0	0	0	0
$255 - x$	0	7	7	16	10	14
$\overline{x} \vee y$	0	8	10	3	14	5
$x \wedge y$	0	16	16	11	8	14
$\overline{x \wedge y}$	0	15	13	11	15	12
$x \oplus y$	0	7	6	5	5	7
$x \gg 1$	0	15	14	7	7	7
$x \gg 2$	0	10	10	17	11	12
$x + y$	0	23	16	29	15	22
$max(x, y)$	0	22	29	29	24	38
$min(x, y)$	0	27	29	28	23	27
$x +^s y$	0	37	35	35	43	30
$(x + y) \gg 1$	0	64	67	64	62	58
Počet využitých tranzistorů	6906624	75216	77456	79600	73664	77312
Odhadované snížení příkonu v %	0	98.91	98.88	98.85	98.93	98.88

Tabulka 9.7: Počet využitých funkcí a tranzistorů ve druhé konvoluční vrstvě sítě LeNet a v aproximacích této vrstvy. Zároveň je odhadnuto snížení příkonu.

Vyhodnocení

V tomto experimentu byla nahrazena druhá vrstva sítě LeNet. Došlo k poklesu přesnosti klasifikace zhruba o 3 % a snížení příkonu o cca 98.89 %. Snížení přesnosti je zde poměrně velké, ale vzhledem k velikosti CGP sítě a prohledávacího prostoru se jedná o zajímavý výsledek.

9.3 Náhrada druhé konvoluční vrstvy v sítí C10

V tomto experimentu se budeme snažit nahradit druhou konvoluční vrstvu sítě C10. Jedná se o vrstvu s mnohem větším konvolučním filtrem než byl filtr v předchozím experimentu. Vrstva, kterou budeme nahrazovat má vstupy o rozměrech $30 \times 30 \times 32$ a filtr velikosti $3 \times 3 \times 32$. Počet vstupů sítě bloků je tedy 288 vah a 288 hodnot. Algoritmus CGP byl spuštěn s parametry uvedenými v tabulce 9.8. Z 5 trénovacích vzorků datové sady Cifar-10 bylo získáno 4500 trénovacích vzorků pro algoritmus CGP.

Parametr	Hodnota
Počet vstupů	576
Počet výstupů	1
Velikost populace	5
Počet generací	50000
Počet sloupců	10
Počet řádků	4
Pravděpodobnost mutace	0.05
Parametr lback	5
Počet běhů CGP na filtr	8

Tabulka 9.8: Parametry pro CGP řešící aproximaci druhé vrstvy v C10 síti.

Náhrada vrstvy a zjištění úspěšnosti při klasifikaci byla provedena celkem 5x. Tabulka 9.9 zobrazuje přesnost sítě s využitím získaných aproximačních vrstev na datové sadě Cifar-10.

Typ vrstvy/sítě	Přesnost klasifikace v %	Doba výpočtu CGP v minutách
Originální síť	78.35	- - -
1. běh aproximace	65.46	558.2
2. běh aproximace	64.97	587.9
3. běh aproximace	65.46	752.1
4. běh aproximace	62.88	744.1
5. běh aproximace	65.11	584.5

Tabulka 9.9: Přesnost klasifikace sítě C10 obsahující původní konvoluční vrstvu a přesnost sítí s aproximovanou vrstvou při klasifikační úloze Cifar-10.

Pro získání výsledku násobení původního konvolučního filtru se vstupy je potřeba 9216 násobení (filtr 3x3x32 a 32 filtrů) a 9184 sčítání. Tabulka 9.10 zachycuje počet funkcí, které jsou využity v sítích aproximujících jednotlivé filtry konvoluční vrstvy. Zároveň je určeno, kolik tranzistorů je využito při jejich výpočtu.

Počet operací	Originální síť	1. běh	2. běh	3. běh	4. běh	5. běh
8-bitová sčítačka	9184	0	0	0	0	0
8-bitová násobička	9216	0	0	0	0	0
$255 - x$	0	49	37	34	35	42
$\overline{x} \vee y$	0	55	52	56	54	60
$x \wedge y$	0	24	11	12	10	10
$\overline{x \wedge y}$	0	42	28	32	46	31
$x \oplus y$	0	17	20	18	20	13
$x \gg 1$	0	15	23	11	10	15
$x \gg 2$	0	14	15	12	11	14
$x + y$	0	154	134	139	130	142
$max(x, y)$	0	47	37	40	32	28
$min(x, y)$	0	71	67	59	65	73
$x +^s y$	0	94	82	78	69	74
$(x + y) \gg 1$	0	98	66	81	90	69
Počet využitých tranzistorů	26531328	200144	168384	170752	165696	166544
odhadované snížení příkonu v %	0	99.25	99.37	99.36	99.38	99.37

Tabulka 9.10: Počet využitých funkcí a tranzistorů ve druhé konvoluční vrstvě sítě C10 a v aproximacích této vrstvy. Zároveň je odhadnuto snížení příkonu.

Vyhodnocení

V tomto experimentu byla nahrazena druhá vrstva sítě C10. Došlo k poklesu přesnosti klasifikace zhruba o 13.5 % a snížení příkonu o cca 99.34 %. Snížení přesnosti je zde velké, což souvisí s velikostí prohledávacího prostoru a pro tuto úlohu nevyhovujícím nastavením algoritmu CGP. Pro získání lepších výsledků je zapotřebí větší síť, větší počet generací a/nebo běhů na filtr. Což vede k velké výpočetní náročnosti.

Kapitola 10

Závěr

V diplomové práci je popsán teoretický úvod do problematiky konvolučních neuronových sítí, který začíná popisem biologického neuronu a dostává se přes jednoduchý perceptron až ke složitějším celkům, jako jsou konvoluční neuronové sítě. Dále je popsán princip fungování kartézského genetického programování a evolučních algoritmů obecně. S využitím knihovny pro modelování konvolučních neuronových sítí byly na zvolených úlohách natrénovány tři architektury a jejich úspěšnost byla porovnána s literaturou. Dále byla navržena optimalizace založená na aproximaci konvolučních vrstev konvoluční sítě pomocí sítě bloků získaných pomocí algoritmu CGP počítající na 8 bitech (s využitím logických operací a sčítání), na rozdíl od původní operace konvoluce počítající v pohyblivé řádové čárce s využitím operací násobení a sčítání.

Pro potřeby této práce byla vytvořena knihovna pro kartézské genetické programování a její funkčnost byla ověřena na řešení úlohy symbolické regrese.

Navržená optimalizace konvolučních neuronových sítí byla implementována a na několika experimentech vyhodnocena. Pro složitější vrstvy – vrstvy obsahující větší konvoluční filtry – došlo k poměrně velkému snížení přesnosti, převážně díky nevhodnému nastavení parametru algoritmu CGP pro dané experimenty. Pro získání lepších výsledků je zapotřebí zvýšit počet řádků, počet sloupců, počet jedinců v populaci, počet generací a/nebo počet běhů CGP na jeden filtr. Avšak experiment, ve kterém byla nahrazena první vrstvy jednoduché konvoluční sítě sloužící pro klasifikaci datové sadě MNIST, ukázal, že při snížení přesnosti klasifikace sítě, náhradou vrstvy, o 0.3 % lze snížit složitost implementace o 93 % (hrubý odhad). Tento experiment ukazuje, že navržený přístup optimalizace má určité uplatnění a měl by být dále detailněji prozkoumán.

Literatura

[1] *Classification datasets results.*
URL `http://yann.lecun.com/exdb/mnist/`

[2] *Classification datasets results.*
URL `http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`

[3] *Getting Started With TensorFlow.*
URL `https://www.tensorflow.org/get_started/get_started`

[4] *Keras: The Python Deep Learning library.*
URL `https://keras.io/`

[5] *Survey of DNN Development Resources.*
URL `http://www.rle.mit.edu/eems/wp-content/uploads/2017/06/Tutorial-on-DNN-2-of-9-Survey-of-DNN-Development-Resources.pdf`

[6] *What is the TensorFlow machine intelligence platform?*
URL `https://opensource.com/article/17/11/intro-tensorflow`

[7] Basheer, I.; Hajmeer, M.: Artificial neural networks: fundamentals, computing, design, and application. *Journal of Microbiological Methods*, ročník 43, č. 1, 2000: s. 3 – 31, ISSN 0167-7012, neural Computting in Micrbiology.

[8] Eiben, A.; Smith, J. E.: *Introduction to Evolutionary Computing*. Springer-Verlag Berlin Heidelberg, 2015, ISBN 978-3-662-44873-1.

[9] Krizhevsky, A.: Learning multiple layers of features from tiny images. 2009.

[10] Kronberger, G. K.: *Symbolic Regression for Knowledge Discovery - Bloat, Overfitting, and Variable Interaction Networks*. Dizertační práce, Johannes Kepler University, Linz, Austria, 2010.

[11] Kruse, R.; Borgelt, C.; Braune, C.; aj.: *Computational intelligence: a methodological introduction*. Springer, 2016.

[12] LeCun, Y.; Bengio, Y.; aj.: Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, ročník 3361, č. 10, 1995: str. 1995.

[13] Lecun, Y.; Bottou, L.; Bengio, Y.; aj.: Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, ročník 86, č. 11, Nov 1998: s. 2278–2324, ISSN 0018-9219, doi:10.1109/5.726791.

[14] Leung, F. H.-F.; Lam, H.-K.; Ling, S.-H.; aj.: Tuning of the structure and parameters of a neural network using an improved genetic algorithm. *IEEE Transactions on Neural networks*, ročník 14, č. 1, 2003: s. 79–88.

[15] Mandischer, M.: A comparison of evolution strategies and backpropagation for neural network training. *Neurocomputing*, ročník 42, č. 1, 2002: s. 87 – 117, ISSN 0925-2312, evolutionary neural systems.

[16] Melanie, M.: *An Introduction to Genetic Algorithms*. A Bradford Book The MIT Press, 1999, ISBN 0-262-13316-4 (HB), 0-262-63185-7 (PB).

[17] Miller, J. F.: *Cartesian Genetic Programming*. Springer-Verlag Berlin Heidelberg, 2011, ISBN 978-3-642-17309-7.

[18] Ndjountche, T.: *Digital Electronics 2: Sequential and Arithmetic Logic Circuits*. číslo sv. 2 in Digital Electronics, Wiley, 2016, ISBN 9781848219854.

[19] Rojas, R.: *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[20] Schaffer, J. D.; Whitley, D.; Eshelman, L. J.: Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, IEEE, 1992, s. 1–37.

[21] Sekanina, L.; Harding, S. L.; Banzhaf, W.; aj.: Image Processing and CGP.

[22] Suganuma, M.; Shirakawa, S.; Nagao, T.: A Genetic Programming Approach to Designing Convolutional Neural Network Architectures. *arXiv preprint arXiv:1704.00764*, 2017.

[23] Umbarkar, A.; Sheth, P.: Crossover Operators in Genetic Algorithms: a review. *ICTACT journal on soft computing*, ročník 6, č. 1, 2015.

Příloha A

Obsah přiloženého paměťového média

- **doc** – zdrojové soubory technické zprávy
- **src** – zdrojové soubory programu
  - **CGPevolver** – knihovna pro CGP
  - **models** – naučené CNN architektury
  - **logs_doc** – log soubory experimentů, uložené aproximace vrstev získané pomocí CGP
  - **Utils** – podpůrné programy
    - **cgpViewer** – nástroj sloužící pro zobrazení chromozomu ve formátu chr, (Prof. Ing. Lukáš Sekanina, Ph.D., Doc. Ing. Zdeněk Vašíček, Ph.D.)
    - **jsonToChr** – převod všech filtrů z aproximované vrstvy uložené v json formátu do chr formátu
- **results.ods** – výsledky experimentů

Příloha B

Návod

B.1 Kompilace

Hlavní část programu je napsaná v jazyce Python, avšak knihovna pro CGP je napsána v jazyce C++. Před spuštěním je tedy potřeba spustit příkaz `make lib` ve složce src/CGPevolver. Příkazem `make` v téže složce dojde k přeložení programu řešícího úlohu symbolické regrese.

B.2 Spuštění hlavní části

Příkazem `python3 src/main.py` dojde ke spuštění hlavního programu. V souboru je velmi snadné určit, jaký experiment (viz kapitola 8) chceme spustit pouhou změnou hodnoty proměnné `experiment`. Pokud chceme spustit experiment vlastní, je potřeba specifikovat název modelu, úlohu, kterou řeší (MNIST nebo Cifar10), nastavení CGP a vrstvu, kterou chceme nahradit.