VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

GENETICKÉ PROGRAMOVÁNÍ S PAMĚTÍ V ÚLOZE SYMBOLICKÉ REGRESE

GENETIC PROGRAMMING WITH MEMORY FOR SYMBOLIC REGRESSION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. TADEÁŠ JŮZA**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

154228

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Jůza Tadeáš, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Strojové učení
Název:	**Genetické programování s pamětí v úloze symbolické regrese**
Kategorie:	Umělá inteligence
Akademický rok:	2023/24

Zadání:

1. Seznamte se s využitím genetického programování v úloze symbolické regrese. Zaměřte se na kartézské genetické programování. Zpracujte přehled používaných benchmarkových úloh.
2. Navrhněte metodu využití genetického programování v úloze symbolické regrese, kdy genetické programování buduje kromě modelu i malou paměť, která umožní podchytit odlehlé či jinak důležité hodnoty v datech.
3. Navrhněte vhodné benchmarkové úlohy pro ověření funkčnosti vytvořené implementace. Dále navrhněte způsob využití metody v oblasti optimalizace konvolučních neuronových sítí.
4. Navrženou metodu implementujte.
5. Proveďte důkladné statistické ověření navržené metody na benchmarkových úlohách z bodu 3.
6. Zhodnoťte dosažené výsledky a diskutujte další možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Sekanina Lukáš, prof. Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	30.10.2023

Abstrakt

Cílem práce je ověřit možnosti rozšíření genetického programování o paměť pro řešení problémů symbolické regrese. Dále pak vytvoření sady úloh pro testování kvality takovýchto řešení. V práci je navržen způsob praktického využití takovéhoto rozšíření, a to pro potencionální snížení energetické náročnosti načítání vah konvolučních neuronových sítí. Zde místo načítání všech vah sítě z paměti je načítáno pouze malé procento vah a zbylé jsou vygenerovány za pomocí evolučně nalezené funkce. Tento způsob byl převážně testován na vahách konvolučních vrstev malé konvoluční neuronové sítě řešící úlohu klasifikace obrazu z testovací sady MNIST. Dále byla také ověřena možnost generování vah na dalších konvolučních neuronových sítích řešících složitější problémy. Podařilo se nalézt různé kompromisy mezi přesností klasifikace a velikostí paměti vah.

Abstract

The purpose of this thesis is to evaluate the possibility of extending genetic programming with memory for solving symbolic regression problems. Furthermore, a set of problems for testing the quality of such solutions is developed. The thesis proposes a practical application of such an extension to reduce the energy consumption of loading weights of convolutional neural networks. Instead of retrieving all the weights of the network from external memory, only a small percentage of the weights is retrieved and the remaining ones are generated using the evolved expression. This method was primarily evaluated on reducing the set of weights of convolutional layers of a small convolutional neural network classifying the MNIST dataset. Furthermore, the possibility of generating weights was also tested on other convolutional neural networks solving more complex classification problems. The proposed method has delivered interesting tradeoffs between the classification accuracy and weight memory size.

Klíčová slova

kartézské genetické programování, genetické programování, neuronové sítě, konvoluční neuronové sítě, symbolická regrese, strojové učení, komprese vah

Keywords

cartesian genetic programming, genetic programming, neural network, convolutional neural network, symbolic regression, machine learning, weight compression

Citace

JŮZA, Tadeáš. *Genetické programování s pamětí v úloze symbolické regrese*. Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Lukáš Sekanina, Ph.D.

Genetické programování s pamětí v úloze symbolické regrese

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Lukáš Sekanina Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Tadeáš Jůza
16. května 2024

Poděkování

Rád bych poděkoval prof. Ing. Lukášovi Sekaninovi, Ph.D za trpělivé vedení, podporu a cenné odborné rady v průběhu vypracování této diplomové práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

V dnešní době probíhá bouřlivý rozvoj umělé inteligence založené na hlubokých neuronových sítích. Využívá se například v možnosti uživatelů diktovat text a příkazy pro zařízení, nebo umožňuje zlepšování fotografií a videí, případně na základě mnoha senzorů dokáže řídit auto. Všechny tyto systémy mají společné úskalí a to je jejich výpočetní náročnost. Jednou z energeticky nejnáročnějších operací prováděných při běhu těchto aplikací je načítání hodnot vah neuronových sítí z paměti [33]. U modelů umělé inteligence, které vzniknou v budoucnu, lze předpokládat tuto náročnost pro celý model čím dál větší kvůli tomu, že se počet těchto vah v modelech zvětšuje vzhledem ke zvětšující se složitosti úkolu, který má umělá inteligence řešit.

Stejně jako jsou neuronové sítě součástí strojového učení, tak je jeho součástí i genetické programování, které umožňuje, pomocí různých metod inspirovaných evolucí v přírodě, vytvářet programy, respektive řešení různých problémů. Je žádoucí, aby řešení nalezená pomocí genetického programování dokázala generalizovat. Například, pokud je hledána funkce, která má procházet zadanými body, nalezené řešení nemusí procházet všemi body, pokud jsou například některé výrazně odlehlé. V této práci je popsáno genetické programování s pamětí, které se snaží tento problém odlehlých hodnot odstranit pomocí malé paměti pro uložení takovýchto hodnot. Toto řešení je následně ověřené na testovacích úlohách.

Tato práce dále popisuje a následně navazuje a rozšiřuje práci [13], ve které byl popsán nový způsob načítání vah neuronové sítě z paměti. V tomto novém způsobu se z paměti načítá pouze malé procento původních vah a zbylé váhy jsou získávány pomocí jednoduché funkce, která tyto váhy aproximuje. Přičemž tato práce ověřuje, zda načtení této funkce a několika původních vah z paměti a následný výpočet zbývajících dat, je méně náročné, než načtení všech vah z paměti. Hlavní zaměření práce je kladeno na získání vah, které je třeba uchovat nezměněné v paměti a na získání funkce aproximující váhy zbylé. Oba tyto problémy jsou řešeny zároveň pomocí evolučního programování, konkrétně o paměť rozšířené implementace kartézského genetického programování.

Tato práce je rozdělena do sedmi kapitol. V kapitole 2 je nejprve představena problematika symbolické regrese a problematika evolučních algoritmů vhodných pro řešení symbolické regrese. Dále je v kapitole 3 popsán základ neuronových sítí potřebný pro následující experimenty. Kapitola také obsahuje často používané metody pro kompresi neuronových sítí. V kapitole 4 je popsán koncept a návrh systému, kterým se tato práce zabývá, a také je zde navržena základní testovací sada úloh pro testování různých variant systému. Kapitola 5 se zabývá samotnou implementací systému. V kapitole 6 jsou popsány experimenty s implementovaným systémem a jeho variantami nad dříve navrženou sadou problémů. Kapitola 7 rekapituluje nejlepší dosažené výsledky a prezentuje nejvhodnější variantu systému.

Kapitola 2

Symbolická regrese a genetické programování

V této kapitole je v sekci 2.1 popsána metoda symbolická regrese. Dále je v sekci 2.2 uveden současný stav genetického programování řešícího symbolickou regresi. Popis algoritmů se převážně zaměřuje na kartézské genetické programování, ale jsou zde také popsány technologie, které reprezentují jednotlivé funkce ve stromové struktuře.

2.1 Symbolická Regrese

Jedním ze speciálních případů regrese je symbolická regrese. Jejím cílem je najít funkci $f(x)$ v prostoru matematických funkcí, která co nejlépe aproximuje zadanou sadu bodů [17]. Míra shody je definována vhodnou chybovou metrikou. Důležité je, že metoda neklade žádné požadavky (např. spojitost, diferencovatelnost, apod.) na výslednou funkci. K tomuto hledání se typicky využívá sada dvojic hodnot v podobě $\{(x_i, y_i\}_{i=1}^{n}$, kde x představuje body ve vstupním prostoru hledané funkce, y reprezentuje odpovídající výstupní hodnoty, které se $f(x)$ snaží co nejlépe aproximovat, a n je počet těchto rozdílných bodů. V těchto bodech jsou pak během hledání funkce $f(x)$ testována kandidátní řešení. Ve výkladu se omezíme na funkce jedné proměnné, zobecněné na více proměnných je přirozené.

Při testovacích úlohách pro systémy řešící symbolickou regresi jsou body typicky generovány pomocí vybrané funkce $F(x)$. Sada dvojic používaná v hledání má pak podobu $\{(x_i, F(x_i)\}_{i=1}^{n}$, kde oproti dříve popsané sadě dvojic je jediný rozdíl v tom, že výstupní hodnoty jsou získávány pomocí známé funkce $F(x)$.

Cílem úlohy ale typicky není pouze nalézt řešení v podobě funkce $f(x)$, která odpovídá funkci $F(x)$ pouze v těchto bodech. Hledané řešení by mělo data dobře generalizovat a tedy funkce $f(x)$ by měla odpovídat hledané funkci $F(x)$ i pro body, které se v původní množině nenachází. Typicky se pro otestování této vlastnosti vytváří speciální testovací množina bodů. Tyto body se většinou volí ze stejného intervalu, jako byly body trénovací.

Pro testování jednotlivých přístupů k symbolické regresi jsou často využívány matematické funkce různé složitosti. Jako často používané funkce je možné uvést například Koza-1 (vztah (2.1)) a Vladislavleva-5 (vztah (2.2)). Podobných funkcí existuje velké množství a sjednocení z různých sad takovýchto funkcí je možné nalézt například v článku [25].

$$\text{Koza-1 [17]: } f(x) = x^4 + x^3 + x^2 + x \tag{2.1}$$

$$\text{Vladislavleva-5 [35]: } f(x_0, x_1, x_2) = 30\frac{(x_0 - 1)(x_2 - 1)}{x_1^2(x_0 - 10)} \tag{2.2}$$

Jako testovací případy, které vychází z reálného světa, jsou pak často využívány různé datové sady. Mezi známé a často využívané patří například datasety:

- Energetická efektivita [34] – datová sada obsahující simulace energetické efektivity na 768 tvarů budov,
- Predikce kvality vody [37] – sada ve které je cílem z 11 měřených hodnot predikovat hladinu pH vody pro další den.

Míru shody dat z datasetu a dat získaných pomocí kandidátní funkce je třeba změřit pomocí vhodné chybové metriky. Mezi často používané metriky patří průměrná absolutní chyba (MAE vztah (2.3)) a průměrná kvadratická chyba (MSE vztah (2.4)). Výsledná hodnota těchto chybových funkcí pro konkrétní kandidátní řešení f se často označuje jako fitness tohoto řešení.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|f(x_i) - F(x_i)| \tag{2.3}$$

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - F(x_i))^2 \tag{2.4}$$

2.2 Genetické programování

Mezi nejvhodnější algoritmy pro řešení symbolické regrese popsané výše se řadí genetické programování [17]. Tato metoda spadá do kategorie strojové učení a staví na evolučních algoritmech. Inspirace pro genetické programování vychází tedy z principů přírodní evoluce. Genetické programování se od evolučních algoritmů liší zejména v reprezentaci kandidátních řešení a ve způsobu jejich evoluce. Genetické programování pracuje s programy, které jsou vhodně zakódovány. Vykonáním programů je získána jejich fitness hodnota. Genetické programování má mnoho oblastí využití, v této práci je ale kladen důraz pouze na řešení oblasti problému symbolické regrese.

Jednotlivé přístupy řešení symbolické regrese vychází ze základního algoritmu genetického programování. Ten je pomocí pseudo kódu popsán v Algoritmu 1.

Dále jsou popsány jednotlivé důležité body Algoritmu 1:

- **řádek 1** – Vytvoření populace probíhá většinou náhodně. Počáteční populaci tvoří náhodně generované programy respektující předem zadané omezení (např. max. počet instrukcí, max. hloubka stromu). Mezi hodnoty, které je nutné předem nastavit, patří i velikost samotné populace, tedy počet kandidátních řešení, které obsahuje.
- **řádek 2** – Základem genetického programování je cyklus, ve kterém dochází k evoluci. K ukončení tohoto cyklu dochází při splnění některé z ukončovacích podmínek. Mezi takové podmínky se nejčastěji řadí:
  - dosažení předem zvoleného počtu generací evoluce, které algoritmus provede,
  - časový limit pro dobu běhu programu, nebo

```
Algoritmus GenetickéProgramovaní()
1   Populace ← VytvořitPopulaci()
2   while PokračovatVEvoluci do
3       Ohodnocení(Populace)
4       NováPopulace ← Populace.nejlepší()
5       while Velikost(NováPopulace) < Velikost(Populace) do
6           jedinec1 ← Výběr(Populace)
7           jedinec2 ← Výběr(Populace)
8           Křížení(jedinec1, jedinec2)
9           if PravděpodobnostMutace() then
10              Mutace(jedinec1)
            end
11          if PravděpodobnostMutace() then
12              Mutace(jedinec2)
            end
13          NováPopulace.přidat(jedinec1, jedinec2)
        end
14      Populace ← NováPopulace
    end
15  return Populace.nejlepší()
```

Algoritmus 1: Abstraktní kód genetického programování dle [17].

– dosažení dostatečně dobrého řešení.

- **řádek 3** – Ohodnocení populace. Z populace je pro každé zakódované kandidátní řešení toto kandidátní řešení vytvořeno a je otestována jeho kvalita při řešeném problému. Podle toho, jak je kandidátní řešení schopné řešit tento problém, je mu přiřazeno hodnocení, většinou označované jako fitness jedince.

- **řádek 4** – Vytvoření nové populace. Při vytváření nové populace je možné tuto populaci vytvořit prázdnou, případně do ní hned od začátku přidat nejlepšího jedince z předchozí populace, případně i více těchto jedinců. Tomuto procesu říkáme elitizmus a zajistí nám, že o nejlepší řešení z předchozí generace nepřijdeme, protože je ochráněno před změnami křížením nebo mutací popsanými dále. Díky tomu není nutné uchovávat doposud nejlepší nalezené řešení mimo populaci, protože nejlepší řešení za celý běh je vždy nejlepší řešení v aktuální populaci.

- **řádek 5** – Dokud nová populace obsahuje menší počet kandidátních řešení než populace stará, jsou do ní nová kandidátní řešení přidávána.

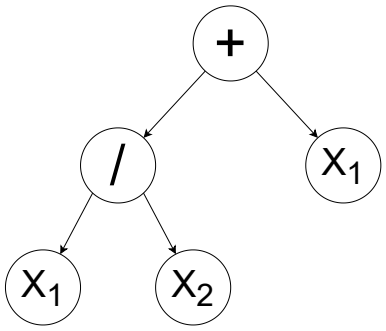
- **řádek 6 a 7** – Výběr jedinců z populace pro křížení a mutaci. Cílem výběru je z populace náhodně vybrat jedince tak, aby jedinci s lepší fitness měli větší pravděpodobnost výběru. Jednou z nejpoužívanější metod výběru je výběr pomocí turnaje, kdy je z populace náhodně vybráno k jedinců, ze kterých je následně vybrán ten s největším fitness. Hodnota k je předem nastavena a je menší než počet jedinců v populaci.

- **řádek 8** – Křížení kandidátních řešení. Cílem křížení je aby si vybraná kandidátní řešení vyměnila nějakou svoji část. Jaká část je mezi jedinci vyměněna je voleno náhodně. Je tedy možné, že k výměně s určitou pravděpodobností vůbec nedojde.

- **řádek 9 a 11** – Každý z jedinců má po křížení nějakou, typicky malou, pravděpodobnost, že dojde k jeho mutaci.
- **řádek 10 a 12** – Mutace jedince. Cílem mutace je do jedince vložit novou část. Standardně je tedy náhodně vybrána část mutovaného jedince, která je nahrazena novou náhodně vygenerovanou částí. Takovéto mutace mohou mít na fitness kandidátního řešení jeden ze tří vlivu:
  - zlepšení fitness, tedy změna má pozitivní dopad na kandidátní řešení,
  - zhoršení fitness, změna má negativní dopad na kandidátní řešení,
  - fitness zůstává stejná, tedy mutace se nijak neprojevily na hodnotě fitness. Tyto mutace se označují jako neutrální mutace a jejich vliv se může projevit až při další úpravě kandidátního řešení.
- **řádek 13** – Upravení jedinci jsou následně přidáni do nové populace.
- **řádek 14** – Poté co je nová populace stejně velká jako populace původní, je původní populace nahrazena novou.
- **řádek 15** – Po ukončení genetického programování některou z ukončovacích podmínek je jako výsledek vracen nejlepší jedinec aktuální populace. S využitím elitizmu se tedy jedná o nejlepšího nalezeného jedince za celý běh populace.

2.2.1 Stromová reprezentace

Pro reprezentaci matematických funkcí se v genetickém programování nejčastěji využívá stromová struktura. Algoritmus 1, popsaný dříve, s touto reprezentací kandidátních řešení dokáže pracovat bez jakýkoliv úprav. Na obrázku 2.1 je znázorněna grafická reprezentace stromové struktury pro funkci $f(x_1, x_2) = \frac{x_1}{x_2} + x_1$.

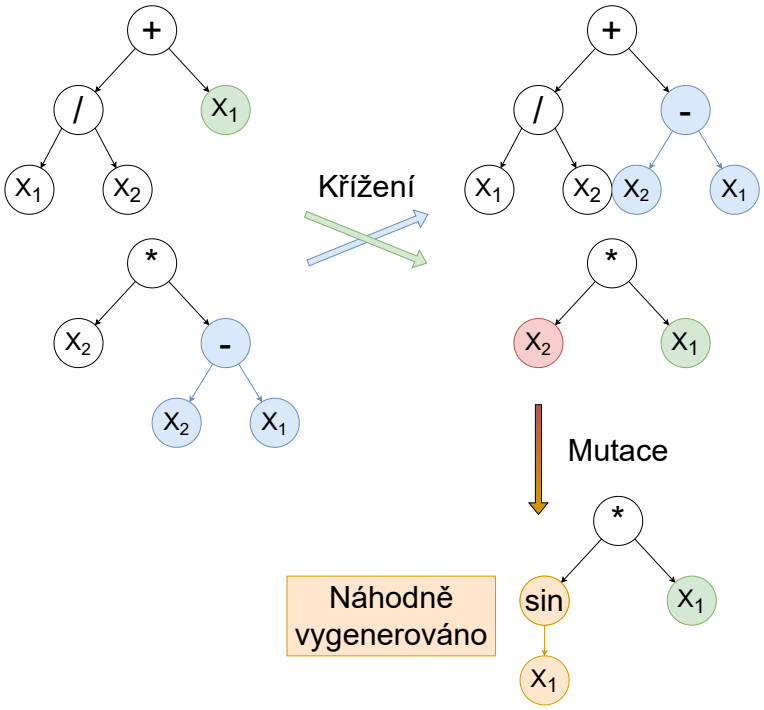


Obrázek 2.1: Ukázka stromové reprezentace funkce $f(x_1, x_2) = \frac{x_1}{x_2} + x_1$

Pro tuto reprezentaci matematických funkcí je třeba definovat následující komponenty:

- množina operací[1], které mohou uzly stromu reprezentovat,
- sada vstupů, ze které je voleno při tvorbě listových uzlů,

[1]Množina operací musí být uzavřená, což znamená, že výstup operace může být vstupem libovolné jiné operace.



Obrázek 2.2: Ukázka křížení dvou funkcí ve stromové reprezentaci a následná mutace jednoho z nově vzniklých řešení.

- maximální výška stromu, aby nedocházelo ke generování zbytečně velkých kandidátních řešení a prohledávaný prostor byl omezen a
- minimální výška stromu, aby nebyla generována ani příliš jednoduchá řešení, další omezení velikosti prohledávaného prostoru.

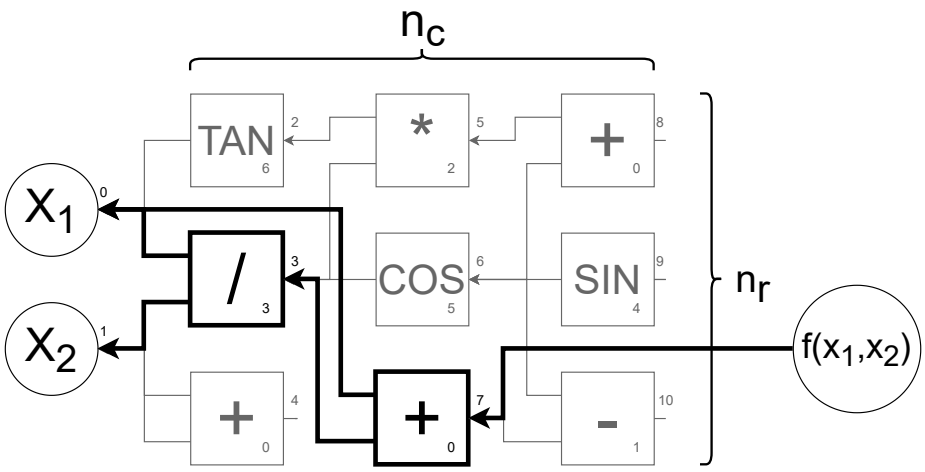
Jednou z možností. jak probíhá křížení dvou stromů, je náhodné zvolení uzlu, tedy podstromu v každém z rodičů, a následným prohozením těchto podstromů mezi křiženými jedinci. Graficky znázorněné křížení i mutaci lze vidět na obrázku 2.2. Při mutaci jedince je nejčastěji náhodně zvolen uzel, a tak i podstrom, který je následně nahrazen nově náhodně vytvořeným stromem. Jak při křížení tak při mutaci je třeba respektovat zvolená omezení uvedena výše.

2.2.2 Kartézské genetické programování

Další možností, jak reprezentovat kandidátní řešení pro symbolickou regresi, je v dvojrozměrné mřížce elementů. Tato reprezentace byla detailně popsána v roce 2011 v knize *Cartesian genetic programming* [27], poprvé byl tento koncept zmíněn v roce 1999 [26]. Jednotlivé elementy v mřížce představují matematické operace a propojení mezi nimi reprezentuje vstupy těchto operací. Na obrázku 2.3 je graficky znázorněna grafická reprezentace mřížky elementů pro funkci $f(x_1, x_2) = \frac{x_1}{x_2} + x_1$, tedy pro stejnou funkci jako je ve stromové reprezentaci na obrázku 2.1.

Mezi typické parametry reprezentace v kartézském genetickém programování patří:

- velikost dvojrozměrné mřížky elementů, tedy kolik má mřížka řádků (n_r) a sloupců (n_c), zobrazené na obrázku 2.3,



Obrázek 2.3: Ukázka možné reprezentace funkce $f(x_1, x_2) = \frac{x_1}{x_2} + x_1$ v mřížce elementů používané kartézským genetickým programováním. Chromozom zobrazeného řešení: (0,0,6) (0,1,3)(1,1,0)(2,3,2)(3,3,5)(0,3,0)(5,6,0)(6,6,4)(6,7,1)(7).

- L-back vzdálenost, na kterou se mohou jednotlivé elementy propojovat, například L-back = 1 znamená, že se vstupy elementů mohou připojit pouze na výstupy elementů z předchozí vrstvy, tedy pokud je L-back větší nebo roven rozměru $x-1$, může se vstup elementu připojit na libovolný výstup předcházejícího elementu,
- množina vstupů, které jsou využívány jako jedna z variant možných vstupů jednotlivých elementů,
- počet výstupů, na rozdíl od předchozí stromové reprezentace je zde možnost generovat z jedné reprezentace více matematických funkcí a
- množina operací, ze které se volí operace jednotlivých elementů při jejich tvorbě, respektive mutaci.

Pro zakódování kartézského genetického programování se typicky využívá chromozom, který se skládá z N trojic ($N = n_r \times n_c$), tedy jedné trojice pro každý element v mřížce. V těchto trojicích jsou první dvě hodnoty vyhrazeny pro vstupy elementu, a poslední hodnota je vyhrazena pro určení operace, jakou element vykonává. Na konci chromozomu je I-tice, kde počet hodnot závisí na počtu výstupů kartézského genetického programování. Ukázka tohoto zakódovaní je pro řešení vykreslené v obrázku 2.3 uvedena v jeho popisku.

Kartézské genetické programování vychází z genetického programování, nicméně se jeho typický algoritmus od algoritmu 1, který je popsaný výše, mírně liší. Typický prohledávací algoritmus kartézského genetického programování je popsán v Algoritmu 2.

Hlavní změny mezi těmito algoritmy jsou v cyklu, který v obou algoritmech začíná na řádku 5. Obsah cyklu u kartézského genetického programování je pak následující:

- **řádek 4** – výběr nejlepšího jedince z populace. Pokud je více nejlepších jedinců, vybírá se náhodně z těch jedinců, kteří v předchozí generaci prošli mutací, je tedy preferována neutrální mutace před žádnou mutací, čímž se docílí postupných změn v hledaném řešení.

```
   Algoritmus KartézskéGenetickéProgramovaní()
1  |  Populace ← VytvořitPopulaci()
2  |  while UkončeníGenetickéhoProgramovaní do
3  |  |  Ohodnocení(Populace)
4  |  |  NováPopulace ← Populace.nejlepší()
5  |  |  while Velikost(NováPopulace) < Velikost(Populace) do
6  |  |  |  jedinec ← Populace.nejlepší()
7  |  |  |  for i ← 1 to PočetMutací do
8  |  |  |  |  Mutace(jedinec)
   |  |  |  end
9  |  |  |  NováPopulace.přidat(jedinec)
   |  |  end
10 |  |  Populace ← NováPopulace
   |  end
11 |  return Populace.nejlepší()
```

Algoritmus 2: Abstraktní kód kartézského genetického programování dle [27].

- **řádek 6** – jako jedinec pro následující mutace je brán nejlepší jedinec z předchozí generace, stojí za povšimnutí, že tento jedinec je již součástí nové populace díky elitizmu na řádku 4.
- **řádek 7** – jedinec po výběru zpravidla nepodstupuje křížení s jinými jedinci, ale pouze předem nastavený počet mutací.
- **řádek 8** – mutace jedince, reprezentovaného mřížkou elementů, spočívá ve změně operace některého z elementů nebo ve změně propojení elementů, tedy ke změně vstupu operace, kterou má mutovaný element aktuálně přiřazenou.

Kartézské genetické programování tedy typicky neobsahuje křížení a jsou prováděny pouze mutace. Mutace je realizována tak, jak je popsáno na *řádku 8*, přičemž mutovaný element může, ale také nemusí, být součástí funkce, kterou mřížka aktuálně reprezentuje. Pokud mutovaný uzel není zapojen, tak se jedná o mutaci neutrální. Pokud jsou všechny mutace neutrální, není nutné, aby byla po provedení mutací znovu vypočtena fitness této reprezentace, protože nedojde k její změně.

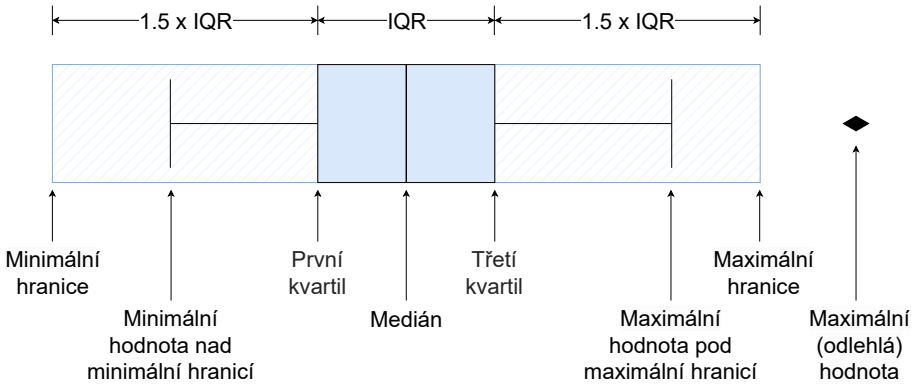
2.3 Statistické vyhodnocení experimentů

Tato sekce je věnována vyhodnocování experimentů prováděných pomocí genetického programování. Genetické programování obsahuje značný prvek náhodnosti a proto není při jednom spuštění jisté, zda se nalezne optimální nebo špatné řešení problému. Proto je nutné experimenty opakovat vícekrát. Při porovnávání různých variant genetického programování je pak vhodné využít statistických metod, aby případné rozdíly mezi těmito variantami byly odhaleny jako statisticky významné.

2.3.1 Grafické metody

Mezi jedny z nejjednodušších metod pro porovnávání výsledků a rychlé předání základních statisticky významných hodnot uživateli patří grafické metody. Existuje několik možností, jaké grafy se dají využít.

Mezi jedny ze základních metod pro vizualizaci a analýzu výsledků patří krabicový graf (boxplot), který je například popsán v normě ISO [11]. Tyto grafy poskytují stručný souhrn klíčových statistických ukazatelů, včetně mediánu, kvartilů a rozsahu distribuce dat.



Obrázek 2.4: Schéma popisující základní hodnoty vynášené krabicovým grafem (boxplot). Šrafovaná oblast, označující maximální/minimální hranice, se nevykresluje.

Na obrázku 2.4 je znázorněn a popsán ukázkový krabicový graf. Na něm jde vidět jednotlivé důležité hodnoty jako:

- **Medián** – neboli druhý kvartil, tedy střední hodnota dat, při sudém počtu dat se jedná o průměr dvojice středních dat a vykresluje se jako hranice v boxu.
- **První/Třetí kvartil** – střední hodnoty polovin při rozdělení dat pomocí mediánu, interval mezi těmito hodnotami tedy obsahuje 50 % řešení a vykresluje se jako box, velikost tohoto intervalu se označuje jako IQR.
- **Minimální/Maximální hranice** – jedná se o hranici, která se nevykresluje, typicky se umisťuje na vzdálenost 1,5×IQR od hranice boxu, ale často se používá i varianta 3×IQR od hranice boxu, případně existují i další varianty. Tato hranice určuje, zda se pro konkrétní hodnoty jedná o odlehlé hodnoty.
- **Minimální/Maximální hodnoty v hranicích** – minimální a maximální hodnota která se nachází v hranicích.
- **Odlehlé hodnoty** – odlehlé hodnoty jsou takové hodnoty, které se nenachází mezi boxem a minimální, respektive maximání hranicí, tyto hodnoty jsou následně vykreslovány samostatně jako body.

Krabicový graf tedy umožňuje získat rychlý přehled o tom, v jakém intervalu se nachází největší počet nalezených řešení a jaká jsou nejlepší nalezená řešení.

2.3.2 Výpočetní metody

Dříve popsané grafické metody můžou být nejednoznačné, pokud je patrné že mezi porovnávanými skupinami výsledků je rozdíl, ale není jasné, jestli tento rozdíl je dostatečně statisticky významný, aby bylo možné tvrdit, že jsou výsledky stejné nebo rozdílné. V tomto případě je dobré použít některou ze statistických metod a daný rozdíl na určité hodnotě významnosti podpořit nebo zamítnout.

K tomuto nám u těchto případů může pomoci jednosměrná ANOVA [30]. Jedná se o široce používanou statistickou metodu, která slouží k porovnání a k analýze výsledků posouzením variability a významnosti rozdílů mezi více skupinami hodnot. Jednosměrná ANOVA usnadňuje testování hypotéz podobnosti tím, že vyhodnocuje, zda existují statisticky významné rozdíly v průměrech více skupin hodnot. Jako nulová hypotéza se používá *H0: Rozdíly v průměrech všech skupin jsou nulové*. Přičemž se tato hypotéza testuje s určitou hladinou spolehlivosti, typicky 5 %.

Pro možnost provedení je nutné, aby všechny porovnávané hodnoty byly nezávislé, s normálním rozdělením se stejným rozptylem, což lze ověřit například pomocí využití Shapiro-Wilkova testu, který na určité statistické významnosti určí, zda jsou hodnoty z normálního rozdělení.

Pokud tedy všechna porovnávaná data splňují všechny podmínky pro použití jednosměrné ANOVY, je možné ANOVU použít a zjistit, zda jsou všechny porovnávané skupiny hodnot stejné či nikoliv. Pokud je jedna skupina rozdílná, výsledek ANOVY neurčuje, o kterou skupinu se jedná a je tedy nutné použít některý jiný test, který určí, která skupina je skutečně odlehlá. Mezi takovéto testy patří například Tukeyho test, případně lze tyto rozdíly určit z dříve popsaných grafických metod.

2.3.3 Nastavování parametrů

Genetické programování má řadu parametrů, které je pro jeho spuštění třeba nastavit na konkrétní hodnoty. Každý z těchto parametrů může nějakým způsobem ovlivnit podobu výsledného nalezeného řešení, přičemž ideální nastavení těchto parametrů se může lišit v závislosti na tom, jaká je datová sada, nad kterou je genetické programování spuštěno a jaké jsou požadavky na výsledné řešení. Tato podsekce se věnuje různým metodám, které usnadní vhodné nastavení těchto parametrů.

Existuje několik variant, jakým způsobem lze vhodně zvolit velikost parametrů. Mezi nejzákladnější často používané patří:

- **Empirická analýza** – zahrnuje provádění systematických experimentů pro hodnocení výsledků genetického programování s různými konfiguracemi parametrů. Přičemž, aby byl vliv parametrů ověřen, je potřeba aby byly experimenty se stejným nastavením spuštěny vícekrát. Následně je nutné rozdíl statisticky vyhodnotit pro výsledky experimentů s různým nastavením. Prostřednictvím experimentování a analýzy tak lze získat náhled na účinky jednotlivých parametrů a jejich interakcí, což vede k nalezení nastavení parametrů genetického programování.
- **Vyhledávání v mřížce** (grid-search) [20] – je přímočará optimalizační technika, při které je vyhodnocena každá varianta parametrů na předdefinované mřížce hodnot. Je tedy třeba ručně definovat pro každý parametr, které jeho hodnoty budou testovány. Genetické programování se následně provádí s každou možnou kombinací hodnot parametrů v rámci mřížky a zaznamenávají se dosažené kvality řešení. Opět je nutné pro statistickou významnost nalezených výsledků provádět experimenty vícekrát. Vyhledávání v mřížce tedy usnadňuje důkladné prozkoumání prostoru parametrů, ale bývá kvůli velkému množství testovaných kombinací výpočetně náročné.

Mezi další varianty, jak hledat vhodné parametry, patří například náhodné vyhledávání, nebo optimalizace roje částic (PSO) [15]. Pro oba tyto algoritmy je nutné zadat prohledávaný prostor, ve kterém se hledá vhodné nastavení. Náhodné vyhledávání v takovém prostoru jednoduše testuje náhodné kombinace hodnot parametrů, zatímco algoritmus PSO do

tohoto prostoru umístí několik částic, které reprezentují nastavení, které bude testováno. Tyto částice následně mají různé rychlosti a na základě svých rychlostí a již otestovaných kombinací parametrů se pohybují prostorem a testují nové kombinace nastavení.

Algoritmů, které se dají pro hledání parametrů použít, existuje celá řada. Mezi nejčastější ale patří zde popsaná empirická analýza, následovaná vyhledáváním na mřížce, kdy empirická analýza slouží pro zjištění jaké hodnoty je vhodné prohledávat pomocí vyhledávání na mřížce.

Kapitola 3

Neuronové sítě a jejich komprese

V této kapitole je uvedena problematika neuronových sítí a kompresních metod určených pro snížení výpočetní a paměťové náročnosti. Tato problematika je popsána s ohledem na to, že dále navrhovaný systém bude využíván pro optimalizaci při načítání vah neuronové sítě z paměti. Nejprve jsou v sekci 3.1 stručně popsány základy neuronových sítí a jejich základních vrstev. V další sekci 3.2 jsou popsány aktuální metody optimalizace, se kterými bude vhodné dále navrhovaný systém porovnávat. Tato kapitola je zpracována dle [1].

3.1 Neuronové sítě

Neuronové sítě jsou stejně jako geentické programování, popsané dříve v kapitole 2.2, součástí strojového učení. Strojové učení sjednocuje algoritmy, které se dokáží učit, a díky tomu adaptivním způsobem řešit konkrétní problém. Není tedy nutné, aby programátor přesně definoval algoritmus pro řešení problému, protože algoritmy strojového učení se dokáží naučit, jak problém řešit. Příkladem je klasifikace ručně psaných číslic v datové sadě MNIST [22] je sada obrázků ručně napsaných číslic, kde je cílem klasifikovat, která číslice se nachází na daném obrázku. Jedním z dnes nejčastěji používaných algoritmů strojového učení jsou neuronové sítě, pro které byla podobně jako pro evoluční algoritmy inspirací příroda, konkrétně funkce mozku. Umělé neurony byly poprvé představeny v roce 1943 v časopisu *Bulletin of Mathematical Biophysics* [24]. K většímu používání neuronových sítí ale došlo až při nárůstu výpočetní kapacity dostupných výpočetních zařízení pro trénování těchto modelů.

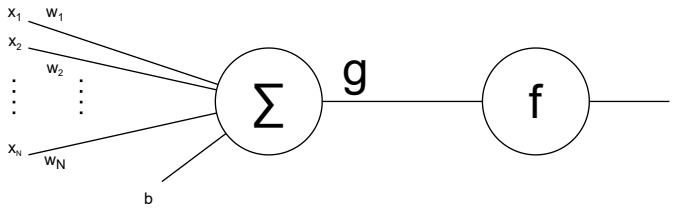
3.1.1 Umělý neuron

Dnes nejčastěji používaná reprezentace umělého neuronu je graficky zobrazená na obrázku 3.1. Matematický zápis funkce g neuronu je v rovnici 3.1:

$$g(\vec{\boldsymbol{x}}) = \sum_{i=1}^{N} w_i x_i + b \tag{3.1}$$

Následně jsou uvedeny možné funkce f v rovnicích 3.2, 3.3 a 3.4. Takto vypadající umělé neurony mezi sebou můžou být různě propojeny.

Ve funkci 3.1 a obdobně i na obrázku 3.1 představuje $\vec{\boldsymbol{x}}$ vstupy konkretního neuronu, w váhy neuronu, b bias neuronu a N počet vstupů. Funkce f se označuje jako aktivační funkce neuronu a mezi nejčastější patří:



Obrázek 3.1: Grafická reprezentace umělého neuronu.

$$\text{sigmoida: } f(g) = \frac{1}{1+e^{-g}} \tag{3.2}$$

$$\text{RELu: } f(g) = \max(0,g) = \begin{cases} 0 & \text{pro } g \leq 0 \\ g & \text{pro } g > 0 \end{cases} \tag{3.3}$$

3.1.2 Plně propojené vrstvy

Představené neurony lze uspořádat do různých architektur. Mezi standardní architektury propojení u neuronových sítí patří:

- plně propojená síť – všechny neurony v neuronové síti jsou připojeny na všechny ostatní neurony,
- plně propojená symetrická síť – všechny neurony v síti jsou připojeny na všechny ostatní a váhy mezi dvěma neurony jsou v obou směrech stejné,
- vrstvová síť – neurony jsou zapojeny ve vrstvách a jsou propojeny pouze s neurony v dřívějších vrstvách, než se samy nachází, a s neurony ve stejné vrstvě,
- acyklická síť – stejné jako vrstvová síť, ale bez propojení na neurony ve stejné vrstvě,
- dopředná síť – neurony jsou organizovány do vrstev a jejich vstupy jsou pouze neurony z předcházející vrstvy.

V této práci budou využívány pouze sítě vykazující architekturu dopředné sítě.
Funkce softmax (vztah 3.4)

$$f(\vec{g})_i = \frac{e^{g_i}}{\sum_{j=1}^{K} e^{g_j}} \tag{3.4}$$

je speciálním případem aktivační funkce neuronu, kde je pro výpočet nutné znát všechny výstupy z vrstvy neuronů. Alternativní názvy pro tuto funkci jsou softargmax, nebo normalizovaná exponenciální funkce. Funkce z vektoru hodnot x vypočítá znormalizované rozložení pravděpodobnosti, přičemž využívá součtu K, tedy všech hodnot. Tato funkce se tedy nejčastěji využívá u poslední vrstvy neuronové sítě, díky čemuž je výstupem celé sítě normované rozložení pravděpodobnosti.

3.1.3 Učení sítě

Pro správné nastavení vah se nejčastěji využívá algoritmus zpětného šíření chyby [29] (anglicky backpropagation), který využívá gradientního sestupu při optimalizaci chybové funkce. Algoritmus, jak již název napovídá, postupuje od posledních neuronových vrstev k předním vrstvám. Vždy na základě aktuálního výstupu sítě a očekávaného výstupu vypočte chybu. Na základě této chyby jsou následně váhy upravovány tak, aby se tato chyba minimalizovala při stejných vstupních datech. Opakováním takovýchto úprav vah dochází k postupnému zvýšení podobnosti výstupu sítě a očekávaného výstupu. Při vytvoření nové neuronové sítě je tedy nutné vahám nastavit nějaké hodnoty, které tento algoritmus následně upravuje. Typicky toto nastavení probíhá přidělením náhodných hodnot jednotlivým vahám.

Trénování sítě nejčastěji probíhá na datasetu, který se rozdělí na dvě množiny: trénovací a testovací. Poměr rozdělení množin záleží na množství dat a náročnosti problému, často se využívá například poměr 70/30. Trénovací množina je využívána pro zjišťovaní chyby neuronové sítě a následnou úpravu jejich vah. Testovací sada slouží pro zjištění chyby neuronové sítě na datech, na kterých neuronová síť nebyla trénována. Chyba neuronové sítě na testovací sadě by se měla během trénování snižovat. Pokud dojde k tomu, že se chyba začne zvětšovat, dochází pravděpodobně k přetrénování sítě a je tedy dobré trénování ukončit a využít řešení, kde chyba na testovací sadě byla nejmenší. Takovouto neuronovou síť využíváme pro inferenci, tedy pro generování odpovídajících výstupů nad daty, které síť neviděla během trénování a nemusí se nacházet ani v datové sadě.

3.1.4 Konvoluční neuronové sítě

Speciálním případem neuronových sítí jsou konvoluční neuronové sítě. První konvoluční neuronová síť byla představena v roce 1998 a jmenovala se LeNet-5 [21]. Více se však začaly používat až po představení sítě AlexNet [19] v roce 2012. Tyto sítě kromě plně propojených vrstev obsahují i konvoluční a pooling vrstvy. Důvodem využívání těchto vrstev je efektivnější práce s rozsáhlými vstupy, jako jsou obrázky a signály. Lepší efektivnost se projeví jak při trénování sítě, tak při samotném použití. Dochází k výrazným snížením počtu parametrů v síti. Plně propojená vrstva by při vstupu ve formě full HD obrázku, tedy obrázku o rozlišení 1 920×1 080 pixelů a 3 barvách, potřebovala pro každý neuron 6 220 800 vah. Při použití 5 konvolučních vrstev o velikosti jádra 5×5 pixelů, z nichž každá je následována pooling vrstvou o velikosti jádra 2×2, bude počet vah snížen na 5 130 pro jeden výstupní kanál poslední vrstvy. Takovýchto kanálů bude na výstupu několik, ale například při 128 kanálech by síť obsahovala 656 640 vah, což je snížení téměř o 90 % proti řešení bez konvolučních vrstev. Samozřejmě konvoluční vrstvy mají také váhy, ale oproti neuronům výrazně menší počet.

Konvoluční vrsva

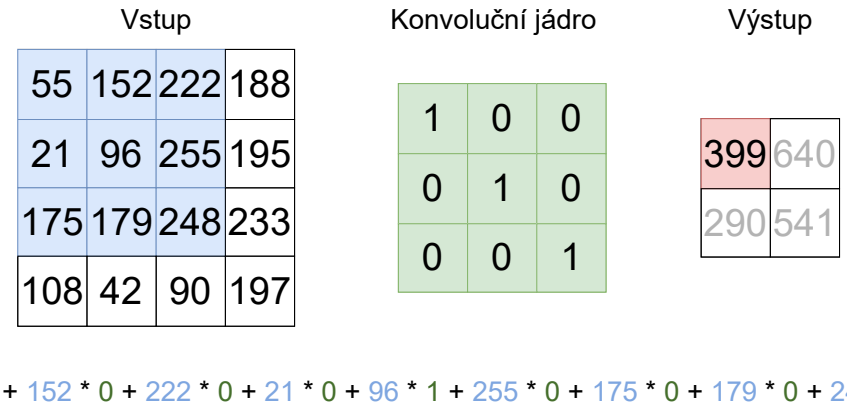
Konvoluční vrstva vykonává operaci konvoluce mezi jádrem s parametry vrstvy a vstupem vrstvy. Matematickou rovnici pro výpočet nové hodnoty pomocí 2D konvoluce můžeme vidět v rovnici 3.5:

$$y_{i,j} = \sum_{m=-M}^{M} \sum_{m=-N}^{N} w_{m,n} * x_{i+m,j+n} \tag{3.5}$$

$y_{i,j}$ označuje nově vypočtenou hodnotu na souřadnicích i, j, hodnoty M a N vychází z velikosti konvolučního jádra. Skutečnou velikost jádra je možné vypočítat jako $2 * M + 1$,

respektive $2*N+1$. Samotné váhy v konvolučním jádře jsou pak označeny jako w a hodnoty na vstupu konvoluční vrstvy jako x.

Princip fungování konvoluční vrstvy je také možné vidět na obrázku 3.2. Z grafické reprezentace funkčnosti je patrné, že dochází ke zmenšení výstupu oproti vstupu a to konkrétně o hodnoty $2 * M$, respektive $2 * N$. Toto zmenšení je způsobeno tím, že pro krajní hodnoty neobsahuje vstup jejich sousední hodnoty, které jsou nutné pro výpočet hodnoty nové. Tento problém lze řešit pomocí *paddingu*, tedy rozšíření vstupního vektoru o okrajové hodnoty. Kolik hodnot je přidáno se volí právě parametrem *padding*, který je dobré volit podle většího z parametrů M a N, aby nedocházelo ke zmenšování výstupu. Doplněné krajní hodnoty mají typicky všechny hodnotu 0.



Obrázek 3.2: Grafická ukázka principu fungování konvolučních vrstev.

Dalším parametrem, který lze u konvolučních sítí nastavit, je parametr *stride*, který udává, o kolik polí se konvoluční jádro posouvá mezi jednotlivými výpočty. Na obrázku 3.2 je použit $stride = 1$.

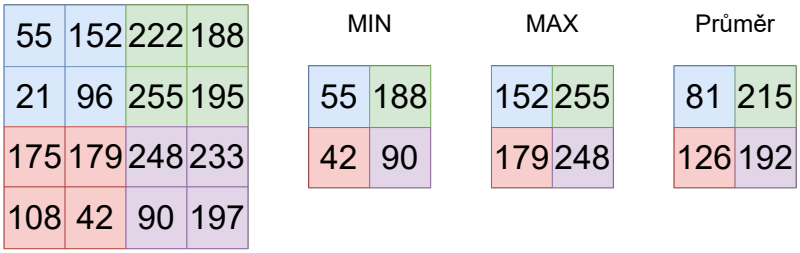
Podobně jako při nastavování vah neuronů i pro váhy konvolučních jader se nejčastěji používá algoritmus zpětného šíření chyby [29].

Pooling vrstvy

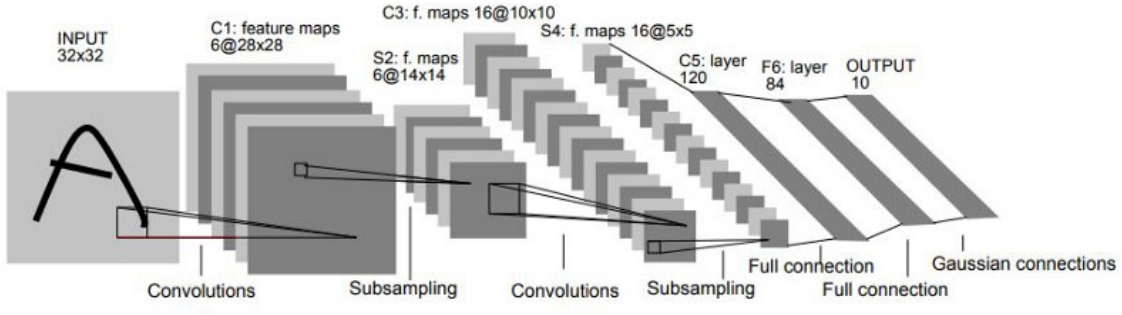
Další důležité vrstvy pro redukci velikosti vstupních dat jsou pooling vrstvy. Tyto vrstvy pracují na podobném principu jako vrstvy konvoluční, ale nemají žádné váhy. Využívají jádra o předem dané velikosti, ze kterého získají maximální hodnoty, respektive hodnotu průměrnou či minimální, která se stává novou hodnotou. Dále se u těchto vrstev volí jejich krok, o který se jádro posouvá vůči vstupu mezi výpočty. Typicky se tento krok volí stejně velký jako velikost jádra. Podle velikosti tohoto kroku v jednotlivých dimenzích vstupních dat dokážeme tyto data právě tolikrát zmenšit. Tedy při full HD obrázku o rozlišení $1\,920 \times 1\,080$ pixelů a kroku 2 v obou dimenzích, při velikosti jádra 2×2, získáme obrázek o rozměrech 960×540 pixelů.

Na obrázku 3.3 můžeme vidět, jak vypadá vstup a výstupy pro různé typy pooling vrstev, pokud je velikost jádra nastavena na velikost 2×2 a krok v obou dimenzích také na hodnotu 2.

Obrázek 3.4 zobrazuje schéma první konvoluční neuronové sítě LeNet-5 [21]. Síť využívá dvě konvoluční vrstvy, vždy následované poolingovou vrstvou, a konec sítě tvoří tři vrstvy s klasickými neurony popsanými dříve. První konvoluční vrstva vytváří ze vstupu 32×32



Obrázek 3.3: Grafická ukázka principu fungování pooling vrstev.



Obrázek 3.4: Schéma konvoluční neuronové sítě LeNet-5 [21].

výstup 28×28 o 6 kanálech, tedy samostatných výstupech, pro jejichž vytvoření byla využita rozdílná konvoluční jádra.

3.2 Komprese neuronových sítí

V této sekci jsou popsány způsoby, kterými je možné zmenšit počet bitů váhy neuronových sítí, respektive snížit jejich počet, případně jiným způsobem zmenšit celkovou velikost sítě. Konvoluční neuronové sítě obsahují velké množství vah, jejichž hodnota musí být uložena, například ResNet-152 má přes 60 milionů vah. Zavedení komprese tedy umožňuje snížit výpočetní náročnost trénování i interference.

3.2.1 Sdíleni vah

Princip metody sdíleni vah [6] (anglicky Weight Sharing) je rozdělení vah neuronové sítě pomocí shlukovací metody do několika podmnožin. V těchto podmnožinách se následně nalezne hodnota, která tuto podmnožinu nejlépe reprezentuje. Z těchto reprezentujících hodnot je následně vytvořena tabulka. Na místo vah je pak následně ukládán pouze index do této tabulky, podle toho, do jaké množiny patřila původní váha a na jakém řádku je reprezentující prvek této množiny. Tabulka s reprezentujícími prvky musí být co nejmenší, aby indexy řádků v ní byly také co nejmenší a komprese tedy dosahovala dobrých hodnot. Na druhou stranu při příliš malé tabulce není rozmanitost vah dostatečná a celá neuronová síť může dosahovat výrazně horších výsledků.

3.2.2 Prořezávání neuronových sítí

Metoda prořezávání neuronových sítí [2] má za cíl z vah neuronové sítě odstranit ty váhy, které nepřispívají ke správnému výsledku sítě při žádném vstupu. Staví na myšlence, že sítě jsou navrhovány větší (s větším počtem parametrů) než je nutné pro získání správného výsledku. Dokáží proto některé parametry odstraňovat a tím snížit jak celkovou velikost sítě, tak i množství operací, které je nutné provést při vyhodnocování odezvy sítě.

3.2.3 Kvantizace

Kvantizace [12] je metoda, která umožňuje snížit počet bitů, které jsou potřeba pro zakódovaní každé váhy. Pro zakódování vah do binární podoby se standardně používá norma IEEE 754 [4], která udává několik formátů kódování. Váhy bez kompresí se nejčastěji kódují jako 32 bitové v pohyblivé řádové čárce. Normovaný formát pro zakódování využívá 1 bit pro znaménko čísla, 8 bitů pro exponent čísla a 23 bitů pro mantisu uložené hodnoty. Pomocí kvantizace však lze tento formát hodnot vah neuronové sítě převést na jiný jednodušší typ kódování, který sice dokáže reprezentovat pouze menší množinu hodnot, ale při správně zvolené velikosti je tato množina stále dostatečně obsáhlá, aby neuronová síť podávala stále kvalitní výsledky. Mezi nejčastější bitové velikosti, které se volí, patří velikost 8 bitů, kdy je využito reprezentace s pevnou řádovou čárkou. Tímto dochází tedy ke zmenšení celkové velikosti paměti vah 4×. Váhy mohou být také kódovány jako čísla v pevné nebo pohyblivé řádové čárce, nebo pomocí speciálního kódování, například množiny {-1, 0, 1} [33].

Kapitola 4

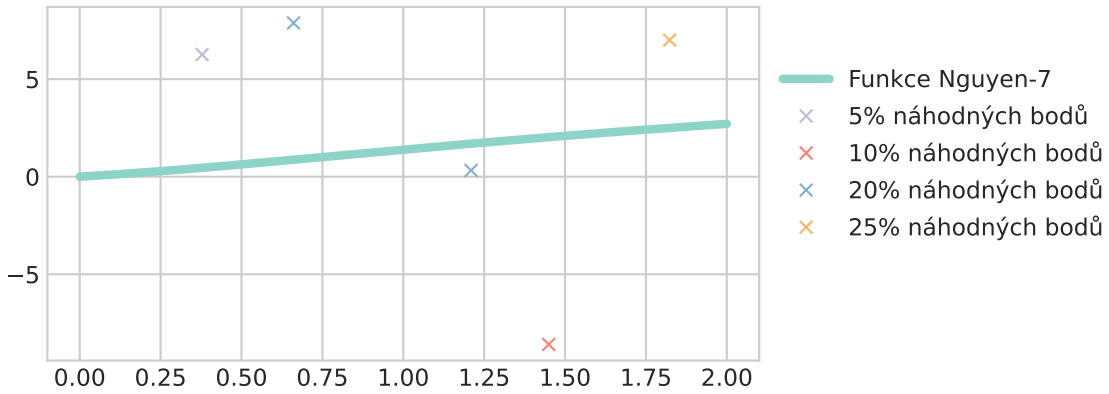
Návrh systému genetického programování s pamětí

V této kapitole je nejdříve představen v sekci 4.1 koncept systému, který umožní genetickému programování pracovat s malou externí pamětí. Následně je v sekci 4.2 popsán konkrétní návrh tohoto systému. Jako poslední je v sekci 4.3 navržena testovací sada pro testovaní, jakým způsobem funguje genetické programování s pamětí při teoretických a praktických aplikacích.

4.1 Koncept

Mnoho datových sad z reálného života obsahuje nějaké množství odlehlých či zcela náhodných hodnot. Cílem většiny přístupů pro zjednodušení takovýchto dat je jejich generalizace, při které je snaha vliv odlehlých, respektive náhodných hodnot potlačit, a výsledný systém je již většinou není schopen správně modelovat. Je však možné najít aplikace, při kterých je právě vliv těchto hodnot nezanedbatelný.

Obrázek 4.1 zobrazuje testovací funkci s několika náhodnými body; z pohledu testování dále popsanou v sekci 4.3.1. Úlohou je najít co nejlepší matematickou reprezentaci takto zadané datové sady, která je složena z bodů generovaných zvolenou funkcí a náhodně



Obrázek 4.1: Ukázka testovací funkce Nguyen-7, s postupně přidávanými náhodnými body, tedy pro 25 % náhodných bodů je potřeba brát v potaz všechny zobrazené body.

generovaných bodů. Pokud by byla tato úloha řešena jako symbolická regrese klasickým genetickým programováním, tak v závislosti na zvolené chybové funkci a počtu hodnot generovaných z funkce, tedy procentuálního zastoupení náhodných bodů, pak by výsledná funkce typicky:

- náhodnými body procházela a tedy by špatně modelovala hledanou funkci *Nguyen-7*,
- přibližovala by se k náhodným bodům a špatně modelované by byly jak body tak i funkce, nebo
- náhodné body zcela ignorovala a ty by tedy byly špatně modelované.

Pokud by však genetické programování obsahovalo malou paměť pro uložení některých bodů, bylo by schopné vytvořit řešení, které obsahuje funkci dobře modelující většinu bodů, a ty, které jsou modelovány špatně, jsou obsaženy v paměti daného řešení.

Jednou z praktických možností využití jsou například váhy neuronových sítí, kdy váhy, jejichž hodnota je odlehlá od vah ostatních, mají na funkčnost neuronové sítě vetší vliv, než sousední k této hodnotě [7]. Důvod, proč generalizovat datovou sadu skládající se z vah neuronové sítě, je následná možnost síť efektivněji spustit na zařízení, které nemá dostatečnou paměť pro práci s váhami neuronové sítě. Pro každé vyhodnocení výstupu sítě totiž může docházet k milionům přístupům do externí paměti vah (DRAM), dle velikosti sítě. Tyto přístupy do paměti mohou být až o dva řády energeticky náročnější, než náročnost jednoho násobení [33].

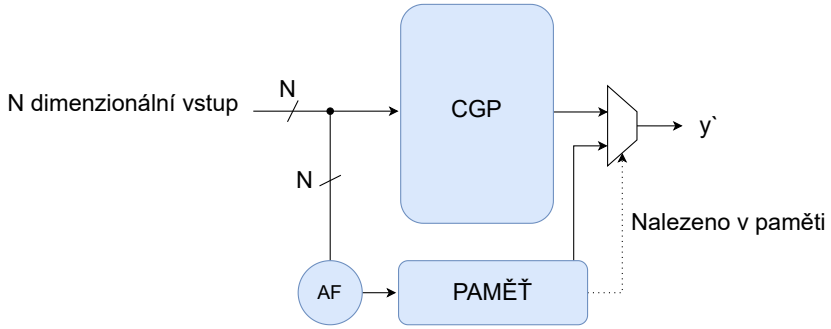
Pokud by tedy bylo možné pomocí genetického programování vytvořit funkci, jejíž výpočet je energeticky méně náročný než tento přístup do paměti a která dokáže generovat většinu vah, z paměti by bylo nutné číst pouze menší množství vah neuronové sítě. Tímto způsobem by tedy bylo možné znatelně snížit energetickou náročnost načítání vah z paměti.

4.2 Návrh systému

Genetické programování s pamětí bude tedy využívat klasické kartézské genetické programovaní a asociativní paměť pro ukládání odlehlých, respektive náhodných hodnot v datech. Očekávané chování systému pak lze popsat rovnicí 4.1, kde $y'(\vec{x})$ je výstup systému pro vstup $\vec{x}$, $AF(\vec{x})$ je agregační funkce nad vstupem $\vec{x}$, $AP(AF(\vec{x}))$ označuje hodnotu v asociativní paměti uloženou pod klíčem získaným z $AF(\vec{x})$ a G značí funkci nalezenou pomocí kartézského genetického programování. Jedná se o rozšíření konceptu symbolické regrese.

$$y'(\vec{x}) = \begin{cases} AP(AF(\vec{x})) & \text{pokud } AF(\vec{x}) \text{ je v } AP \\ G(\vec{x}) & \text{jinak.} \end{cases} \tag{4.1}$$

Systém je kromě reprezentace pomocí funkce 4.1 možné také znázornit graficky, viz obrázek 4.2. Agregační funkce (AF) pak může nabývat různých podob. Její vstup bude vždy vektor všech vstupních hodnot a výstup v podobě hodnoty použitelné v asociativní paměti jako klíč, případně v běžné paměti jako adresa některé z hodnot. V případě použití běžné paměti je nutné určit, jakým způsobem bude označen příznak, že byla hodnota v paměti nalezena. Agregační funkce tedy může být prostá konkatenace všech vstupních hodnot, jejich XOR (který ale již může způsobit stejný výstup pro různé vstupní vektory), případně zobecnění vstupních hodnot do nějakého intervalu, čímž se docílí toho, že systémem uložené hodnoty přísluší intervalu vstupních hodnot.



Obrázek 4.2: Schéma univerzálního návrhu systému využívajícího kartézské genetické programování (CGP) rozšířené o paměť. Pro upravení vstupu systému pro asociativní paměť se využívá agregační funkce (AF).

Schéma na obrázku 4.2 zobrazuje kartézské genetické programování rozšířené o paměť. Chromozom takovýchto řešení je tedy rozšířen o sadu dvojic, kdy každá dvojce reprezentuje jednu hodnotu uloženou v paměti a skládá se z klíče a hodnoty.

Při vytvoření nového řešení je kartézské genetické programování inicializováno klasickými způsobem, tedy náhodně. Obsah paměti je při vytvoření zvolen také náhodně a to jako náhodné hodnoty ze sady, na které bude probíhat běh. Při následném běhu a s tím souvisejících mutacích je pro systém zaveden nový nastavitelný parametr udávající pravděpodobnost mutace v paměti. Na základě tohoto parametru je tedy při každé mutaci rozhodnuto, zda proběhne v genetickém programování, které hledá funkci, nebo v paměti. Pokud probíhá mutace, v paměti tak je náhodná hodnota z paměti nahrazena za náhodnou hodnotu z trénovací sady.

Pro výpočet fitness jednotlivých řešení produkovaných genetickým programováním s pamětí byla zvolena funkce 4.2, kde M je počet hodnot v datové sadě, $y'(\vec{\boldsymbol{x_i}})$ hodnoty získané funkcí G či případně získané z paměti systému AP a o jsou body ze zadaného datasetu.

$$\text{Fitness} = \sum_{i=1}^{M} (y'(\vec{\boldsymbol{x_i}}) - o_i)^2 \tag{4.2}$$

Kartézské genetické programování umožňuje, na rozdíl například od stromové reprezentace popsané v 2.2.1, vytvářet více funkcí během jednoho běhu. Pokud by byl zde představený systém spuštěn tak, aby obsahoval více výstupů, byla by pro každý tento výstup vytvořena samostatná paměť a fitness funkce by se skládala ze součtů dříve uvedených funkcí 4.2 pro každý výstup.

4.3 Testovací sada úloh

Tato sekce se zabývá návrhem sady testovacích funkcí pro otestování navržené metody kartézského genetického programování s pamětí. Bohužel není možné převzít některou z již vytvořených umělých testovacích sad. Tyto sady neobsahují často žádná odlehlá data. Také jsou často rozdělené na testovací a trénovací části. Toto rozdělení pro testovaný systém není vhodné, protože sytém se snaží odhalit odlehlou sadu dat v sadě, na které je trénován. Tedy testování a všeobecně spouštění na sadě jiných vstupů neotestuje kvalitu vytvořené paměti. Samozřejmě se nabízí některé možnosti jak vytvořit testovací sadu tak, aby paměť

měla smysl. Jedna možnost vytvoření testovací sady je popsána a testována sekci 6.8, kromě tohoto experimentu jsou však výsledky prezentovány na sadách trénovacích.

Návrh testovacích úlohy je rozdělen do dvou kategorií:

1. Umělé (matematické) funkce rozšířené o náhodné body a
2. Neuronové sítě, tedy generování hodnoty vah.

V následujících podsekcích jsou tyto kategorie jednotlivě popsány.

4.3.1 Testovací sady: Matematické funkce

V této podkapitole je blíže popsána tvorba testovacích problémů na základě matematických funkcí, výběr těchto funkcí, výběr testovaného intervalu na těchto funkcích, a jejich rozšíření o náhodné body.

I když klasické testovací funkce nejsou vhodné pro testování dříve navrženého systému, tak bylo z těchto funkcí vycházeno. Z článku [25], který se zabývá sumarizací různých testovacích sad pro genetické programování, byla zvolena sada 10 funkcí a to tak, aby byly zastoupeny všechny původní sady zmíněné v tomto článku. Přehled vybraných funkcí je možné vidět v tabulce 4.1.

Tabulka 4.1: Vybrané matematické funkce do testovací sady.

Jméno	Funkce	Velikost	Interval	Krok	Původní sada
Koza-1 [17]	$f(x) = x^4 + x^3 + x^2 + x$	40	$[-1, 1]$	0.05	Koza
Nguyen-7 [23]	$f(x) = \ln(x + 1) + \ln(x^2 + 1)$	20	$[0, 2]$	náhodně	Koza
Nguyen-10 [23]	$f(x_0, x_1) = 2 * \sin(x_0) * \cos(x_1)$	100	$[-1, 1]$	náhodně	Koza
Korns-1 [16]	$f(x_0, x_1, x_2, x_3, x_4) = 1.57 + (24.3 * x_3)$	10 000	$[-50, 50]$	náhodně	Korns
Korns-4 [16]	$f(x_0, x_1, x_2, x_3, x_4) = -2.3 + 0.13 * \sin(x_2)$	10 000	$[-50, 50]$	náhodně	Korns
Keijzer-1 [14]	$f(x) = 0.3 * x * \sin(2\pi x)$	20	$[-1, 1]$	0.1	Keijzer
Keijzer-8 [14]	$f(x) = \sqrt{x}$	100	$[0, 100]$	1	Keijzer
Vladislavleva-5 [35]	$f(x_0, x_1, x_2) = 30\frac{(x_0-1)(x_2-1)}{x_1^2(x_0-10)}$	300	$x_0, x_2 : [0.05, 2]$ $x_1 : [1, 2]$	náhodně	Vladislavleva-A
Vladislavleva-1 [35]	$f(x_0, x_1) = \frac{-(x_0-1)^2}{1.2+(x_1-2.5)^2}$	100	$[0.3, 4]$	náhodně	Vladislavleva-B
Vladislavleva-2 [35]	$f(x) = ^{-x}\ x^3(\cos(x)\sin(x))(\cos(x)\sin^2(x) - 1)$	100	$[0.05, 10]$	0.1	Vladislavleva-C

Sloupec *Velikost* v tabulce 4.1 popisuje počet hodnot v datové sadě. Určité procento z těchto hodnot je vždy nahrazeno náhodnými hodnotami. Každá funkce má přidělenu sadu náhodných hodnot, ze které je určitý počet těchto hodnot vybrán a následně nahradí hodnoty generované funkcí. Například pro funkci Nguyen-7 můžeme vidět na obrázku 4.1 její průběh a sadu náhodných bodů. Body jsou přidávány akumulativně, tj. sada s 10 % náhodných bodů obsahuje všechny body sady s 5 % náhodných bodů plus další. Zbylé body do datové sady, se kterou se algoritmus spustí, je generován z rovnice funkce uvedené v tabulce 4.1. Ze stejné tabulky také můžeme zjistit ze sloupečku *Krok*, jakým způsobem jsou zbylá data generována. Pokud je krok náhodný, pak jsou vstupní data pro funkci generována náhodně z intervalu ze sloupce *Interval* (pokud není konkrétně uvedeno jinak, je interval stejný pro všechny vstupní hodnoty). Pokud má krok uvedenou konkrétní hodnotu, jsou data generována v pravidelných intervalech podle daného kroku ve stejném intervalu, a náhodné hodnoty nahradí některé z těchto hodnot. Samotná náhodná hodnota je následně generována jako náhodné číslo z intervalu [-10, 10].

Dále se pro jednotlivé funkce liší i samotné nastavení systému. Hlavním rozdílem je množina možných operací používaných při hledání funkce pomocí kartézského genetického programování. Rozdílné množiny jsou použity podle toho, z jaké původní množiny funkce pochází (sloupec *Původní sada* v tabulce 4.1). Operace v jednotlivých těchto sadách jsou

Tabulka 4.2: Množina operací, kterou používá kartézské genetické programování pro symbolickou regresi pro danou funkci.

Název sady	Operace
Koza	$+$, $-$, $*$, %, sin, cos, e^x, $log(\|x\|)$
Korns	$+$, $-$, $*$, %, sin, cos, e^x, $log(\|x\|)$, x^2, x^3, $sqrt$, tan, $tanh$
Keijzer	$+$, $*$, $\frac{1}{x}$, $-x$, $sqrt$
Vladislavleva-A	$+$, $-$, $*$, %, x^2
Vladislavleva-B	$+$, $-$, $*$, %, x^2, e^x, e^{-x}
Vladislavleva-C	$+$, $-$, $*$, %, x^2, e^x, e^{-x}, sin, cos

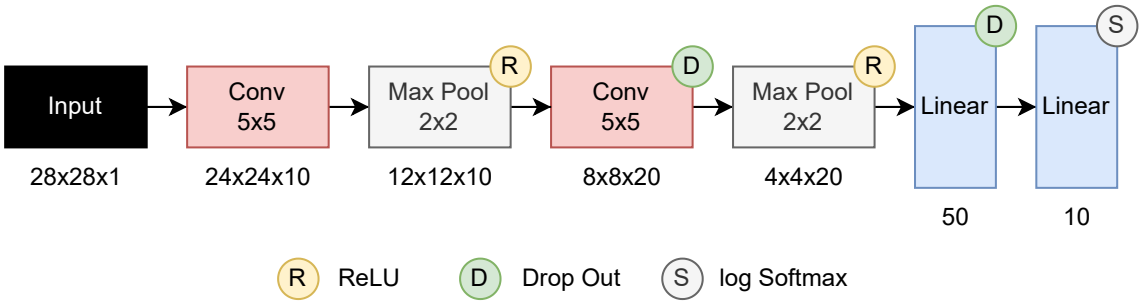
pak uvedeny v tabulce 4.2. Množiny operací respektují původní množiny a nebyly nijak pozměněny.

4.3.2 Testovací sady: Posloupnosti vah

Jak již bylo uvedeno výše, hlavní úlohou systému by mělo být hledání takového řešení (tj. dvojci G, AP), které zvládne generovat rozumné váhy neuronových sítí tak, aby síť fungovala s podobnou přesností, jako síť s původními váhami. Proto je vhodné do testovací sady zařadit i sady vah vhodně zvolené neuronové sítě. Výsledky experimentů pak bude možné měřit ve dvou metrikách:

M1: blízkost generovaných vah k vahám skutečným a

M2: přesnost neuronové sítě s generovanými vahami na testovací sadě problému.



Obrázek 4.3: Schéma neuronové sítě, řešící problém MNIST [22], jejíž váhy jsou součástí testovací sady.

Kvůli výpočetní náročnosti bude během hledání vhodného řešení kartézské genetické programování s pamětí využívat jako fitness funkci metriku M1. M2 bude využívána až pouze pro vyhodnocení kvality výsledného řešení.

Množina vah, kterou se bude kartézské genetické programování s pamětí snažit generovat, bude posloupnost vah konvoluční neuronové sítě z obrázku 4.4, která pracuje jako klasifikátor číslic z datové sady MNIST [22]. Mezi důvody zvolení tohoto problému patří to, že je známý, ne příliš náročný, a tedy otestování velkého množství experimentů nebude příliš výpočetně náročné.

Pro řešení problému MNIST byla vytvořena a natrénována síť, jejíž architekturu je možné vidět na obrázku 4.4. Tato neuronová síť obsahuje dvě konvoluční vrstvy a dvě plně propojené vrstvy. Síť je natrénována pomocí optimalizátoru SGD (s nastavenou setrvačností na 0.5 a rychlostí učení na 0.1) na datové sadě MNIST s využitím *PyTorch* [28]. Během trénování proběhlo 10 epoch na trénovací sadě o velikosti 60 000 obrázků a trénování probíhalo po 64 obrázcích. Bylo dosaženo výsledné přesnosti sítě 97.36 % na testovací sadě MNIST, která čítá 10 000 testovacích vstupů.

Z takto natrénované sítě byly následně vytvořeny dvě oddělené testovací sady vah:

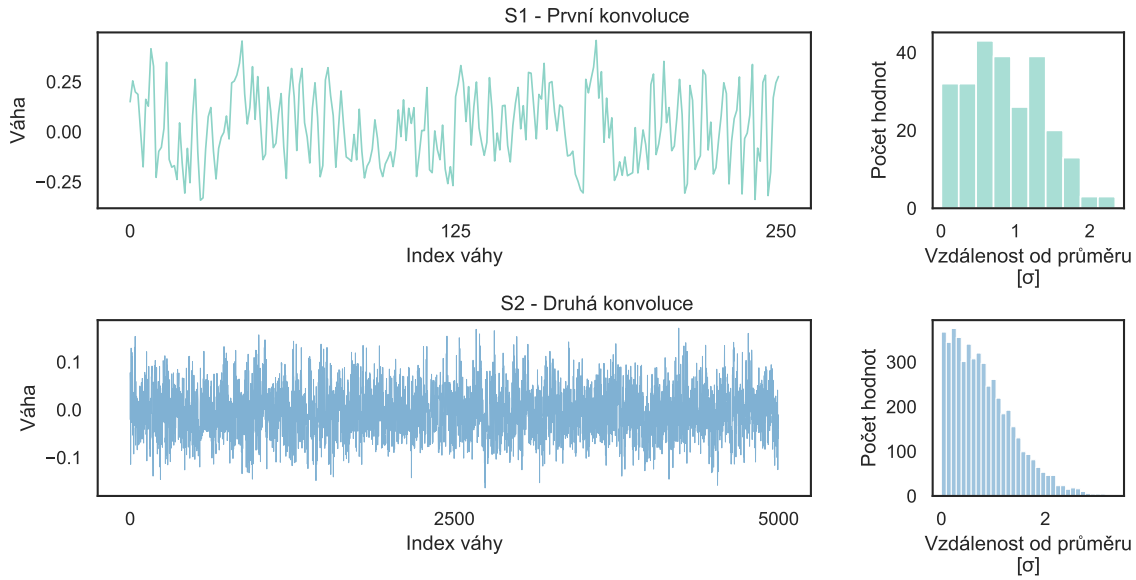
S1: sada obsahuje 250 vah z první konvoluční vrstvy a

S2: sada obsahuje 5000 vah z druhé konvoluční vrstvy.

Váhy v těchto sadách neobsahují bias a jejich počet je tedy možné vypočítat pomocí vzorce 4.3, kde V je velikost sady, x a y představují velikost jádra konvoluce a k počet vstupních, respektive výstupních kanálů dané vrstvy:

$$V = x * y * k_{in} * k_{out} \tag{4.3}$$

Kromě analýzy dopadu zavedení generátoru vah na přesnost klasifikace bude kvantifikována i úspora v počtu bitů nutných pro uložení sítě.



Obrázek 4.4: Hodnoty vah podle jejich indexu v datových sadách S1 a S2. Vpravo pak histogram vah dle jejich vzdálenosti od průměru, vzdálenost je vynesena ve směrodatné odchylce.

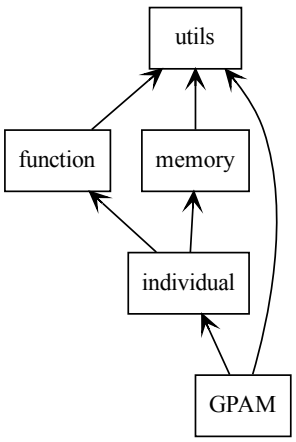
Kapitola 5

Implementace a testování

Tato kapitola se zabývá provedením konkrétní implementace genetického programování s pamětí představeného v minulé kapitole. Systém byl implementován v programovacím jazyce `Python` (verze 3.10). Tento programovací jazyk byl zvolen díky implementační jednoduchosti, přehlednosti kódu a dobré návaznosti na práci s konvolučními neuronovými sítěmi (*PyTorch*). Tyto parametry se řadí mezi podstatné kvůli experimentálnímu rázu práce, kdy jsou v následující kapitole popsány jak experimenty lišící se ve vstupu systému, tak experimenty s rozdílnou implementací některých jeho částí. Implementace je v této kapitole popsána stručně, pro detailní popis je možné využít generované dokumentace implementace na přiložením paměťovém mediu.

5.1 Implementace

V této sekci je stručně popsána implementace dříve navrženého kartézského genetického programování s pamětí. Podrobnější popis implementace a skriptů pro zpracování dat, tvorbu grafů případně práci s neuronovými sítěmi je obsažen na paměťovém médiu.



Obrázek 5.1: Diagram závislostí mezi jednotlivými soubory.

Implementace je rozdělena do několika souborů. Diagram závislosti mezi hlavními soubory je možné vidět na obrázku 5.1. Jednotlivé soubory pak následně obsahují:

- **`GPAM`** – soubor se základní implementací genetického programování,

- **individual** – implementace jedinců skládajících se z funkce a paměti,
- **function** – implementace genetického programování pro hledání funkce, konkrétně implementace kartézského genetického programování,
- **memory** – implementace jednotlivých tříd pro experimenty uvedené v kapitole 6 (diagram těchto tříd je dále uveden na obrázku 5.2) a
- **utils** – pomocné funkce pro implementaci.

Žádný ze zde uvedených souborů není samostatně spustitelný a pro samotné spuštění genetického programování s pamětí slouží soubor `main.py`, který využívá implementace v těchto souborech pro spouštění jednotlivých experimentů. Implementace v souboru `main.py` využívá pro testování různých variant nastavení parametrů vyhledávání v mřížce (grid-search). Přičemž kolikrát je otestováno stejné nastavení parametru závisí na parametru, který je popsán dále v textu. Pro běh experimentů se stejným nastavením je využíváno paralelního spouštění.

Jak již bylo zmíněno, implementace je provedena v jazyce `Python`, který pro výpočty s hodnotami v základních strukturách, není například oproti jazyku `C` efektivní. Aby byla snížena náročnost jedné z typicky nejnáročnějších částí genetických algoritmů, tedy výpočtu fitness hodnoty pro jednotlivé kandidátní řešení, je výpočet fitness dle funkce 4.2 (z předchozí kapitoly 4.2) akcelerován pomocí knihovny `numpy` [8]. Akcelerován je samozřejmě i výpočet výstupních hodnot pro funkci, kterou aktuálně generované řešení reprezentuje. Akcelerace těchto dvou výpočtů musí být oddělené kvůli využívání paměti v systému. Protože by bylo náročné akcelerovat výpočet jenom vybraných hodnot, jsou vždy funkcí generovány výstupní hodnoty pro všechny vstupní hodnoty. Následně jsou dle hodnot v paměti některé hodnoty zaměněny za hodnoty z této paměti. Nad takto vzniklým polem, které tedy obsahuje hodnoty jak generované funkcí, tak hodnoty z paměti, je vypočtena fitness.

Jazyk `Python` umožňuje objektově orientované programování, kterého bylo během implementace využito. Pro přehlednost zde ale není uveden celý implementovaný diagram tříd, ale pouze část implementovaná v souboru `memory`, která je zobrazena na obrázku 5.2. Na diagramu je vidět, že je implementována třída `Memory`, ze které následně vychází implementace různých pamětí, které se liší v implementaci agregační funkce AF, kterou využívají a jejíž vliv je následně testován a popsán v kapitole 6.

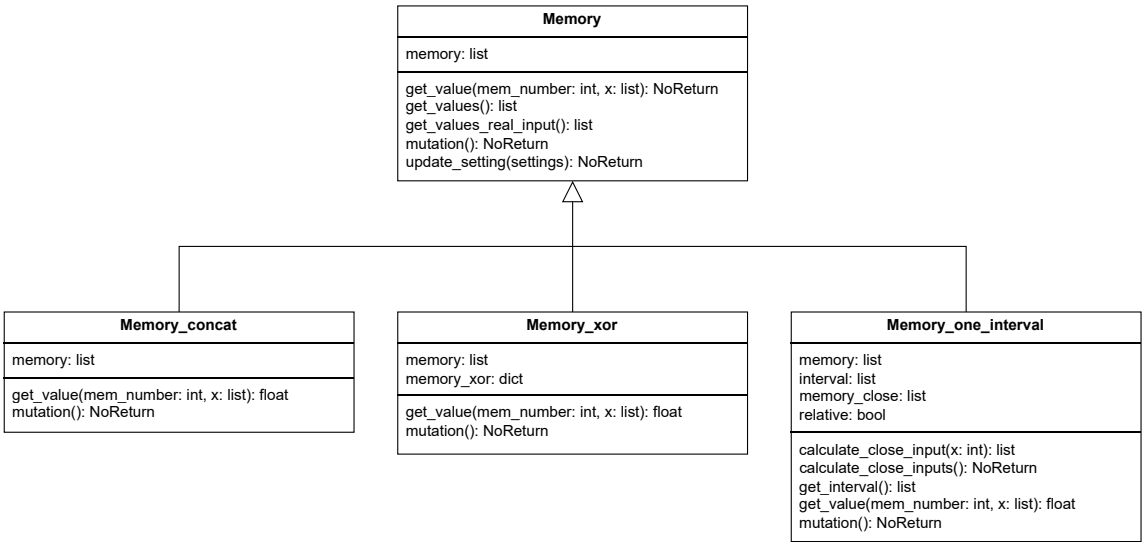
5.2 Spuštění a rozhraní systému

Jak již bylo zmíněno, je implementace provedena v jazyce `Python`. Pro její spuštění je tedy nutné mít interpret tohoto jazyka ve verzi 3.10, případně ve verzi, která je k této verzi kompatibilní. Dále je nutné doinstalovat moduly nutné pro běh implementace, které jsou uvedeny v souboru `requirements.txt` a pro jejich instalaci je možné využít příkaz

```
pip install -r requirements.txt
```

Následně je možné implementaci, ve složce obsahující soubor `main.py`, spustit pomocí příkazu:

```
python main.py
```



Obrázek 5.2: Diagram tříd pro třídy pracující s pamětí, které jsou obsažené v modulu (souboru) `memory`.

Implementace `main.py` umožnuje zadat několik parametrů, konkrétně:

- --help (-h) – pro výpis nápovědy a následné ukončení,
- --config_file CONFIG_FILE (-c CONFIG_FILE) – pro zadání cesty a jména souboru s konfigurací pro prováděné experimenty, výchozí hodnota je `./config.yaml`,
- --experiments EXPERIMENTS (-e EXPERIMENTS) – slouží pro zadání kolikrát se má opakovat experiment se stejným nastavením, výchozí hodnota je nastavena na 30 opakování experimentů a
- --cpu CPU – pro nastavení maximálního počtu jader procesoru, která budou pro běh použita. Výchozí hodnota je nastavena na menší z dvojice (počet jader, počet experimentů).

Pro konkrétní nastavení hodnot nutných pro běh genetického programování s pamětí slouží konfigurační soubor, který je blíže popsán v příloze B. Zde jsou popsány parametry, které byly používány pro spuštění jednotlivých experimentů popsaných dále v kapitole 6.

5.3 Testování

Každá implementace potřebuje testy. O to více, když je rozšiřována mezi různými experimenty o novou funkcionalitu a je potřeba, aby během rozšiřování možností implementace nedošlo k neúmyslné změně již implementované funkcionality. Dobrou variantou řešící tento problém jsou automatické dynamické testy.

Pro vytvoření jednotlivých testů pro implementaci v *pythonu* byl použit modul *unittest*[1]. Jedná se tedy o metodu se znalostí testovaného kódu a jeho struktury (White box). Pro sledování pokrytí kódu testy byla využita metoda kombinující pokrytí řádků a větví, kdy

[1] unittest: https://docs.python.org/3/library/unittest.html

Tabulka 5.1: Tabulka zobrazující pokrytí jednotlivých souborů testy.

Soubor	Počet řádků	Neotestovaných řádků	Větví	Částečně pokrytých větví	Pokrytí [%]
`main.py`	150	58	97	6	59
`src/GPAM.py`	188	67	90	5	63
`src/individual.py`	321	49	179	2	82
`src/memory.py`	145	6	76	1	96
`src/function.py`	356	4	196	4	98
`src/utils.py`	23	0	6	0	100
Celkem	1183	185	644	19	83

je procentuální pokrytí počítáno pomocí vztahu 5.1. Toto pokrytí bylo zvoleno, protože ho lze jednoduše automaticky měřit pomocí `Coverage.py`[2].

Celkem bylo po parametrizaci vytvořeno 428 testů, které základní implementační soubory pokrývají tak, jak je uvedeno v tabulce 5.1. Přičemž důraz byl hlavně kladen na základní části implementace, zatímco například v souboru `main.py` nebylo testováno, jakým způsobem se zpracovávají parametry programu při spuštění.

$$\text{Pokrytí} = \frac{\text{Otestovaných řádků} + \text{Otestovaných větví}}{\text{Řádků} + \text{Větví}} \tag{5.1}$$

V tabulce 5.1 lze vidět, že bylo dosaženo dobrého pokrytí, přičemž celkové pokrytí na všech souborech je 83 %. To že byly implementovány testy samozřejmě nezaručuje, že implementace neobsahuje žádné chyby. Dosažené pokrytí však výrazně snižuje možnost nezamýšlené změny ovlivňující stávající implementaci při přidávaní nových funkcionalit do implementace.

[2]Coverage.py: https://coverage.readthedocs.io/en/7.5.1/

Kapitola 6

Experimentální ověření navržené metody

V této kapitole je věnován prostor různým experimentům s dříve představeným konceptem. Každý experiment obsahuje část uvedení do problematiky, kterou se bude zabývat, a položením výzkumných otázek. Následně je popsáno s jakým konkrétním nastavením byl systém v experimentu spouštěn. Dále jsou popsány a graficky reprezentovány výsledky jednotlivých experimentů. Pokud není uvedeno jinak, jsou krabicové grafy konstruovány z 30 nezávislých běhů. Aby byly grafy lépe porovnatelné a protože fitness funkce 4.2 nebere v potaz velikost datové sady, jsou fitness hodnoty v jednotlivých grafech normalizovány počtem hodnot v datové sadě. Každý experiment je pak zakončen zodpovězením dříve položených výzkumných otázek.

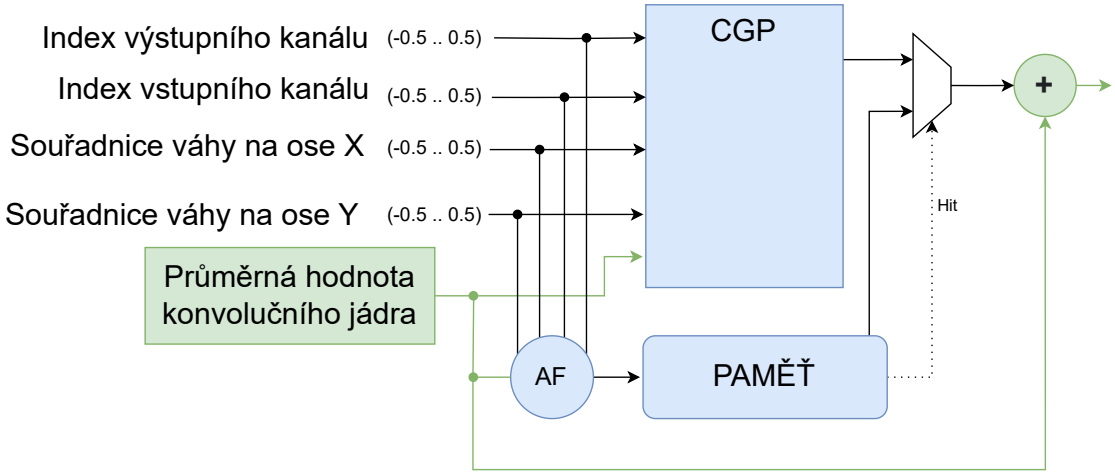
6.1 Základní funkcionalita

První experiment ze zabývá zjištěním, jak se implementovaný systém genetického programování s pamětí chová na sadě testovacích experimentů navržených v sekci 4.3. Nejdříve byla otestována funkčnost systému na sadě matematických funkcí s různou velikostí paměti. Následně je systém otestován i na sadách S1 a S2 (vahách neuronové sítě). Zde je nutné také zvolit vhodný vstup systému tak, aby byla dostatečně předána informace, která váha se aktuálně generuje.

Hlavní částí experimentu bylo otestovat, jaký vliv na výslednou kvalitu řešení má velikost paměti, o kterou je obohaceno standardní kartézské genetické programování. Proto byla většina parametrů kartézského genetického programování nastavena na statické hodnoty, které je možné vidět v tabulce 6.1. Tyto parametry byly na uvedené konkrétní hodnoty nastaveny po provedení několika experimentů a i když se nemusí jednat o nejlepší možné nastavení, tak se s tímto nastavením dosahuje dostatečně dobrých výsledků, na kterých je možné dobře pozorovat vliv různého nastavení parametrů, na které se práce zaměřuje. Množina operací, které mohou být použity v hledaných funkcích, je pak nastavena podle aktuálně testované funkce, viz tabulka 4.2 v podsekci 4.3.1, zabývající se návrhem testovací sady matematických funkcí. Pro testování generování vah neuronové sítě pak byla vybrána množina operací používaná sadou *Korns*. Tato množina byla vybrána kvůli tomu, že obsahuje nejvíce operací.

Tabulka 6.1: Základní nastavení kartézského genetického programování s pamětí.

Parametr	Hodnota
velikost populace	$1 + \lambda$, $\lambda = 4$
počet generací	5 000
velikost CGP (sloupce × řádky)	20 × 2 (20 × 10 při testovací sadě S1 a S2)
L-back	maximální
počet mutací na chromozomu	2
pravděpodobnost mutace paměti	20 %
pravděpodobnost náhodné konstanty	10 %



Obrázek 6.1: Schéma generování vah konvolučních neuronových sítí pomocí kartézského genetického programování s pamětí. Schéma obsahuje dvě varianty: s a bez zelené části.

Kartézské genetické programování umožňuje kromě funkcí pracovat i s konstantami. V tabulce 6.1 je uvedena pravděpodobnost vytvoření konstanty, která je po volání mutace generována náhodně jako desetinné číslo z intervalu $[-1\,000, 1\,000]$.

Pro testovaní na datasetech S1 a S2 je také nutné určit vhodné vstupy a architekturu zapojení systému, který se snaží generovat váhy pomocí kartézského genetického programování s pamětí. Na obrázku 6.1 můžeme vidět dvě možná zapojení, která se liší v zelené části schématu. Tedy tím, jestli je na vstup připojena i průměrná hodnota konvolučního jádra, do kterého se aktuální hodnota generuje. Další rozdíl je v přičítání této hodnoty k hodnotě výstupní. Tímto je do značné míry zjednodušena hledaná funkce. Také ze schématu můžeme vidět zvolených 5 vstupů. Všechny vstupy až na vstup s průměrnou hodnotou konvolučního jádra jsou dále normalizovány do intervalu od -0.5 do 0.5, protože většina generovaných hodnot se nachází právě v tomto intervalu. Poznamenejme, že genetické programování s pamětí potřebuje nějaké vstupní hodnoty, na jejichž základě bude aproximovat zadanou množinu dat. Není zřejmé jak tyto vstupy generovat. Navržená možnost není jistě optimální, ale nějakou bylo potřeba zvolit.

6.1.1 Výzkumné otázky

Cílem experimentů v této sekci je zodpovědět následující otázky:

1. Zvládá kartézské genetické programování s pamětí dobře aproximovat běžné testovací matematické funkce a jaký vliv má velikost paměti na chybu výsledného řešení?

2. Jaká architektura zapojení systému je vhodná pro generování vah konvolučních vrstev?

3. Zlepšuje se přesnost neuronové sítě přidáním paměti při generovaní vah?

6.1.2 Výsledky experimentů

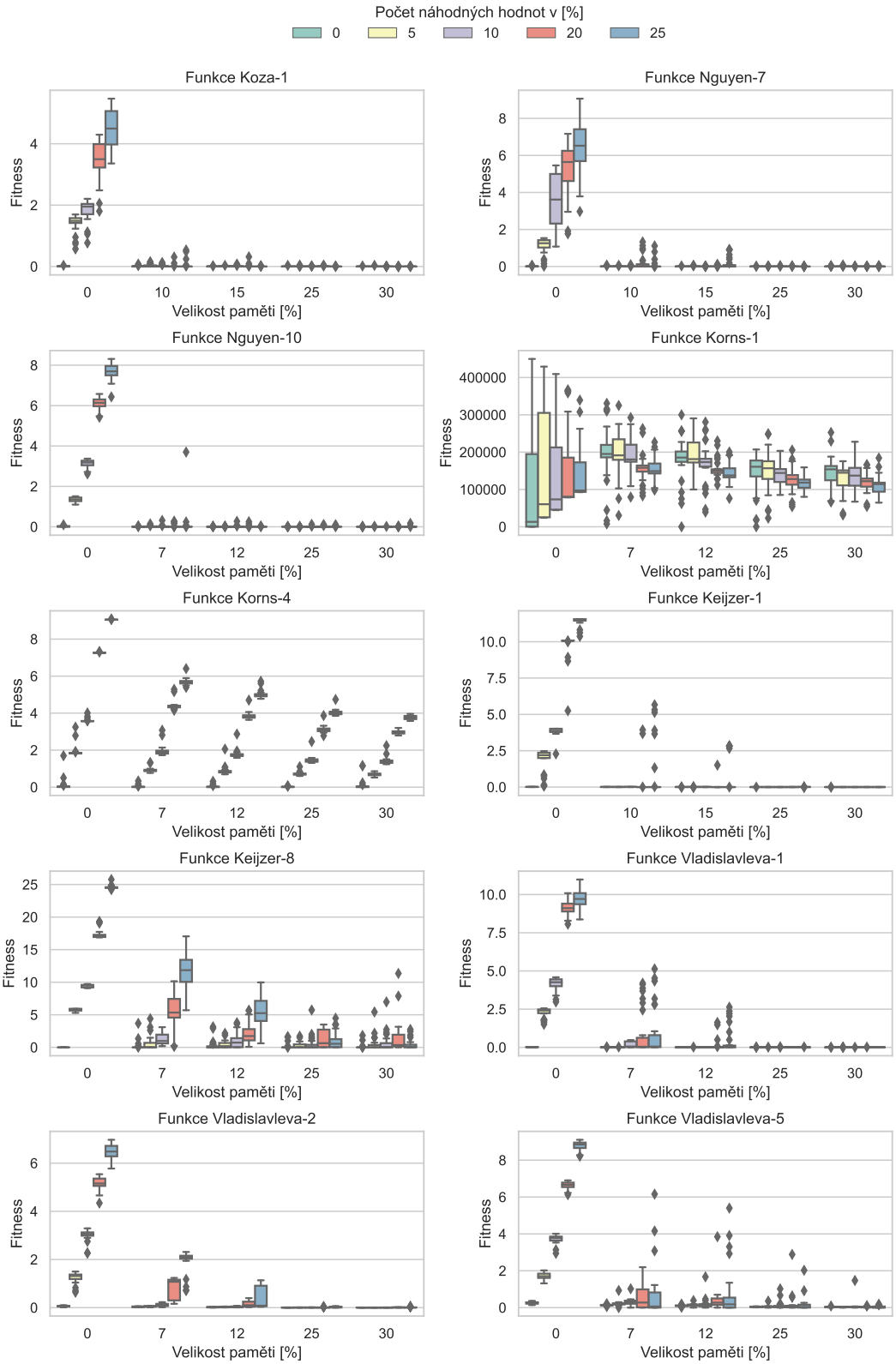
Tato podsekce je věnována jednotlivým grafům zobrazujícím výsledky z běhů nad dříve navrženými testovacími sadami.

Nejprve byl systém otestován na základních matematických funkcích. Na obrázku 6.2 můžeme vidět grafy pro všech 10 vybraných matematických funkcí a to tak, že na ose x je vynesena dostupná velikost paměti systému. Kvůli rozdílné velikosti datových sad není velikost paměti u všech experimentů stejná. Velikosti paměti byly voleny tak, aby pro každé množství náhodných hodnot existovala mírně větší velikost paměti, například pro 5 % náhodných hodnot velikost paměti o 7 %. Na ose y je pak vynesena fitness, se kterou algoritmus skončil (cílem je minimalizovat vztah 4.2). Kvůli rozdílné velikosti sad je tato hodnota normalizována, aby jednotlivé funkce byly porovnatelné. Barevně jsou pak označeny výsledky pro různé zastoupení náhodných hodnot v datové sadě, barva je bohužel často špatně zjistitelná, ale bloky mají vždy stejné pořadí.

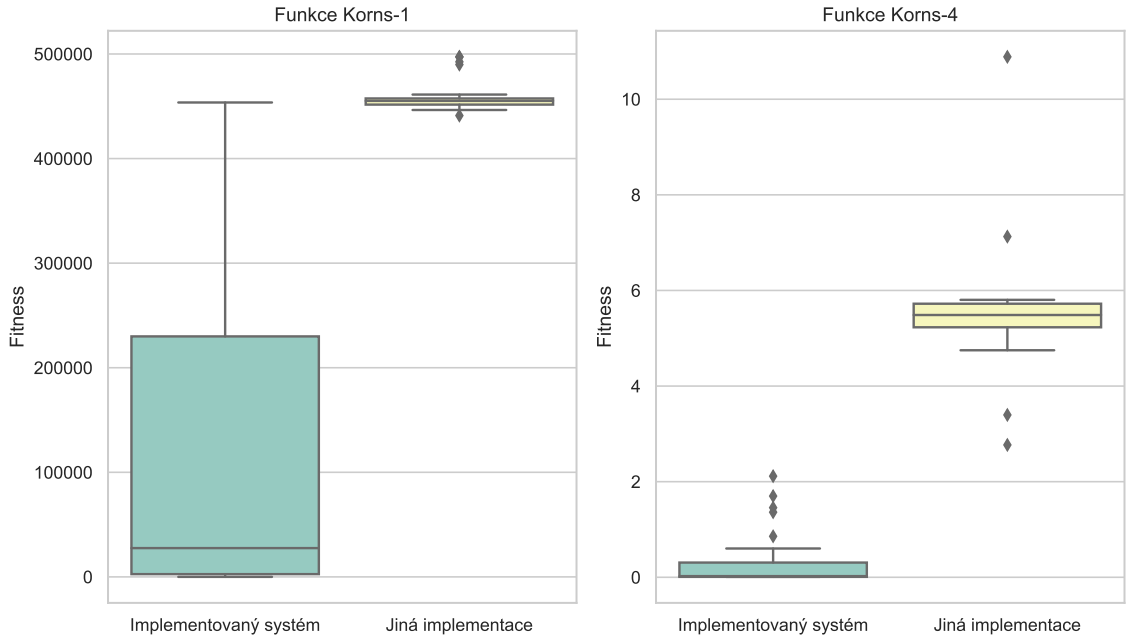
Můžeme vidět, že u jednodušších funkcí jako je *Koza-1*, *Nguyen-7* a *Nguyen-10* je dostatečná i malá paměť, která zachytí nejvíce odlehlé hodnoty a zbylé dokáže modelovat funkcí. Proto výsledky i při velikosti paměti 10 % dosahují dobrých hodnot u všech zastoupení náhodných hodnot. U dalších funkcí je již pokles fitness se zvětšující se pamětí postupnější.

Na obrázku 6.2 si také můžeme všimnout, že graf funkce *Korns-1* zobrazuje vysoké hodnoty fitness, které se od ostatních výrazně liší. To může být způsobeno tím, že funkce je špatně řešitelná pomocí kartézského genetického programování a nebo že implementace systému obsahuje chybu. Aby se vyloučila druhá varianta, tedy že v implementaci je chyba, byla řešitelnost funkce otestována pomocí jiné implementace kartézského genetického programování, konkrétně byla použita implementace [32]. Protože tato implementace nedisponuje potřebnou sadou operací, byla o chybějící operace rozšířena. Nastavení experimentu tedy mohlo být zvoleno stejné, jako nastavení testovaného systému. Protože se výsledky z této implementace mohou lišit od výsledků systému, například výpočtem fitness, byla otestována i funkce *Korns-4*, u které má testovaný systém výsledky dobré. Obě funkce byly testovány v podobě bez náhodných bodů. Na obrázku 6.3 je možné vidět grafy porovnávající tuto jinou implementaci oproti implementovanému systému bez použití paměti. Můžeme vidět že implementovaný systém dosahuje v obou případech lepších výsledků než jiná implementace. Také můžeme vidět že problém s řešitelností *Korns-1* přetrvává i u této implementace, zatímco funkce *Korns-4* je oběma implementacemi řešena výrazně lépe.

Kromě výsledných fitness jednotlivých řešení na obrázku 6.2 může být také vhodné se podívat jak se jednotlivé řešení graficky liší. Na obrázku 6.4 můžeme vidět dvě nalezená řešení pro funkci *Nguyen-7*. Je zde funkce nalezená bez paměti (fialová) jejíž matematický zápis je dán vztahem 6.1. Také je zde vykreslena funkce nalezená s pamětí (modrá) s matematickým zápisem 6.2 a body (zelené) které byly u řešení s touto funkcí uloženy do paměti. Jak z matematických zápisů tak z grafické reprezentace můžeme vidět, že řešení které nevyužívá paměť, je výrazně složitější funkce. Na grafu pak jde vidět, že funkce s pamětí obstojně aproximuje všechny body z původní datové sady a to buď funkcí nebo obsahem



Obrázek 6.2: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi, při změnách počtu náhodných hodnot v sadě trénovacích dat a velikosti paměti testovaného systému.



Obrázek 6.3: Porovnání výsledků dvou implementací kartézského genetického programování u dvou funkcí. Datasety neobsahují náhodné hodnoty a implementace nevyužívají paměť. Jiná implementace je [32].

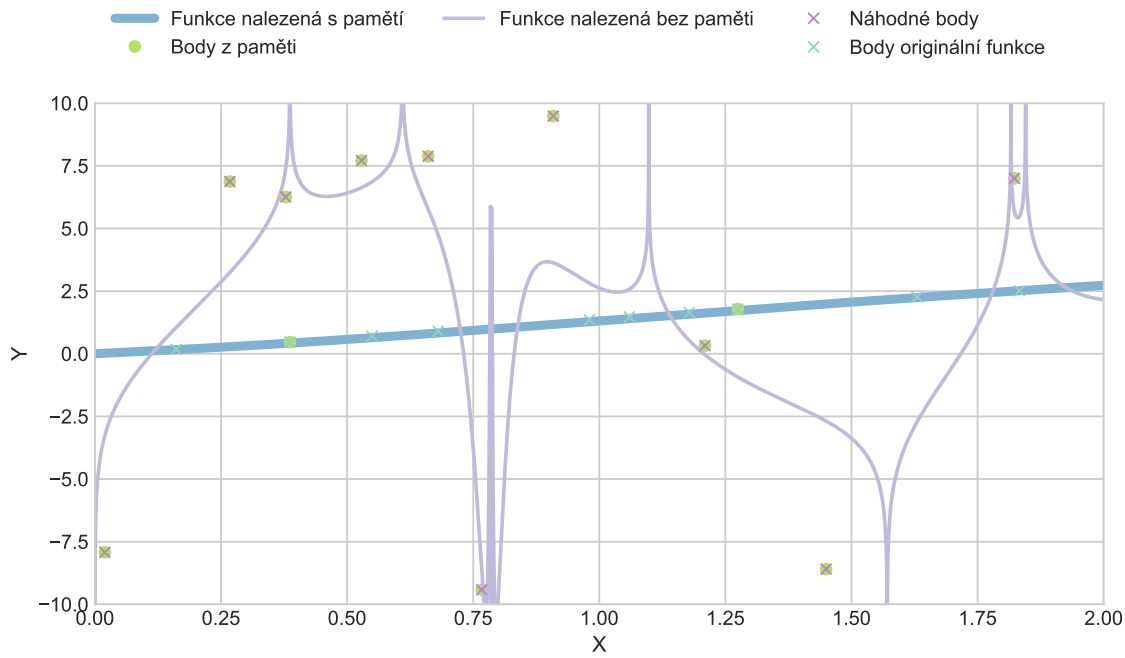
paměti. Na druhou stranu funkce nalezená bez použití paměti se k velkému počtu bodů pouze přibližuje, ale rozhodně jimi neprochází.

$$f(x) = \frac{1}{x}(x(-2x\cos(x) - \ln\left(\frac{|\frac{(2x\cos(x)-\cos(2x))\ln(2|x\cos(x)|)}{x\cos(x)}|}{2}\right) + \ln(|\cos(2x)|) - \sin\left(\frac{\ln(|\cos(2x)|)}{x}\right) + \cos(2x)) - (e^{\cos(x)} * x - \ln(|\cos(2x)|))\sin(\ln(|\cos(2x)|)x)) \quad (6.1)$$

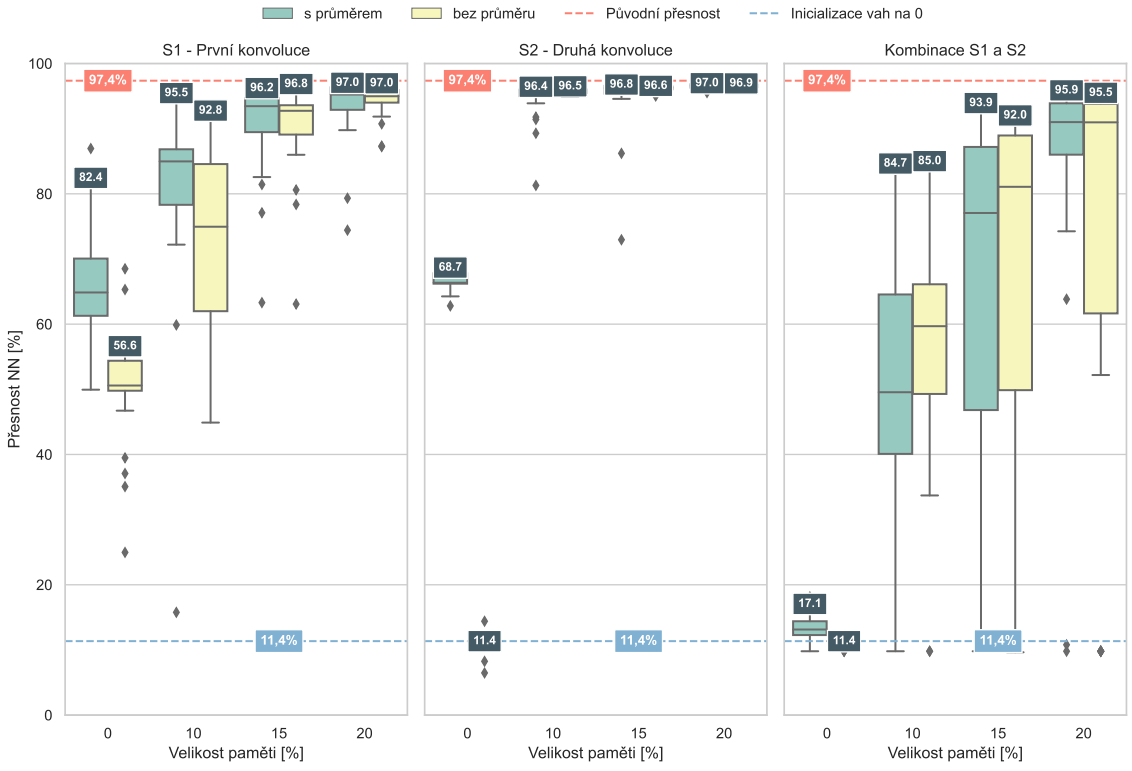
$$f(x) = ln(|e^x + x^3|) \quad (6.2)$$

Obrázek 6.5 je věnován výsledkům kartézského genetického programování s pamětí v úloze generování vah natrénované konvoluční neuronové sítě z obrázku 4.4. Referenční hodnoty vah jsou sady S1 a S2. Na ose x je velikost paměti a na ose y přesnost neuronové sítě na testovacích datech datasetu MNIST [22]. Trojice grafů pak prezentuje výsledky tak, že:

1. Graf obsahuje výsledky, když jsou generovány váhy pouze první konvoluční vrstvy (S1).
2. Graf prezentuje výsledky pokud jsou generovány váhy pouze druhé konvoluční vrstvy (S2).
3. Graf kombinuje výsledky obou vrstev a to tak, že při stejné velikosti paměti bere výsledek se stejným indexem, při seřazení podle fitness hodnoty a z těchto párů řešení vytvoří síť, ve které jsou generovány váhy obou konvolučních vrstev v síti.



Obrázek 6.4: Porovnání dvou generovaných řešení pro funkci Nguyen-7 s 50 % náhodných bodů, z nichž jedno je s pamětí o velikosti 60 % a druhé bez paměti.



Obrázek 6.5: Dosažené přesnosti neuronové sítě na testovacích datech datasetu MNIST [22]. Váhy sítě byly generovány pomocí testovaného systému s různou velikostí paměti a různou architekturou.

V grafech je také vynesena původní přesnost sítě, která používá váhy získané pomocí *PyTorch*. Dále je zde i vynesena přesnost sítě při nahrazení všech generovaných vah za váhy s hodnotou 0.

Z grafů je patrné, že řešení, která neobsahují paměť, mají výrazně horší výsledky než řešení ostatní. Dále co se týče výsledků, které využívají, respektive nevyužívají, průměrnou hodnotu jádra konvoluce (PJK), můžeme vidět, že u experimentů s pamětí dosahují často podobných výsledků. U experimentů s první konvolucí platí, že medián řešení bez PJK je často horší než to s PJK, ale nejlepší řešení jsou často srovnatelná. U druhé vrstvy po přidání paměti dosahují dobrých výsledků obě řešení. Při sloučení obou vrstev do jednoho řešení jsou výsledky také značně podobné, ale varianta bez PJK dosahuje lepšího mediánu při velikosti paměti 15 a 20 % a lepších nejlepších výsledků při velikosti paměti 10 a 15 %.

6.1.3 Závěr – Odpovědi na výzkumné otázky

V této podsekci jsou zodpovězené dříve položené otázky.

1. **Zvládá kartézské genetické programování s pamětí dobře aproximovat běžné testovací matematické funkce a jaký vliv má velikost paměti na chybu výsledného řešení?**
   Systém zvládá dobře řešit většinu funkcí, jedinou výjimkou je funkce *Korns-1*, která dává špatné výsledky. Tato funkce ale byla otestována i na jiné implementaci kartézského genetického programování, která měla také problém s nalezením dobré aproximace této funkce. Tedy nejedná se o problém, který by byl specifický pro testovanou implementaci. U ostatních funkcích z testovací sady můžeme vidět, že velikost paměti pomáhá řešit problémy nad datasety, které obsahují náhodné hodnoty.

2. **Jaká architektura zapojení systému je vhodná pro generování vah konvolučních vrstev?**
   Obě navržené architektury dávají dobré výsledky. Varianta, která neobsahuje PJK, je lepší v paměťové náročnosti celého řešení. Varianta obsahující PJK dává v případech testování pouze na jedné vrstvě stabilně lepší výsledky převážně u první vrstvy. Pro testování systémů, u kterých je požadavek na velikost výsledného řešení, bude tedy lepší varianta bez průměru. U experimentů, ve kterých nezáleží na velikosti, je vhodnější použití varianty s průměrem.

3. **Zlepšuje se přesnost neuronové sítě přidáním paměti při generovaní vah?**
   Ano, i malé velikosti paměti pomáhají zajistit výrazně lepší výsledky sítě s generovanými váhami. Řešení, které neobsahovalo přičtený průměr pro první vrstvu bez paměti, nalezlo většinu řešení stejně kvalitních, jako inicializace těchto vah na hodnotu 0. Ale po přidání paměti o velikosti pouze 10 % bylo řešení již pouze o 1.1 % horší než původní síť.

6.2 Využití hodnot v paměti

V tomto experimentu je cílem otestovat, zda by bylo možné využít hodnoty uložené v paměti jako konstanty ve funkci hledané kartézským genetickým programováním.

V předchozím experimentu bylo systému umožněno do funkce při mutaci vložit konstantu s pravděpodobností 10 %. Kdy při vkládání konstanty byla její hodnota náhodně

generována z intervalu $[-1\,000, 1\,000]$. Jako lepší zdroj těchto hodnot by mohla sloužit paměť systému. Pro otestování, jak se varianty liší, byly otestovány 4 nastavení, která můžeme vidět v tabulce 6.2. Nastavení *Float konstanty* je tedy stejné jako v předchozím experimentu. Kromě varianty *Bez konstant* mají všechny varianty dohromady vždy 10 % pravděpodobnost na vytvoření konstanty. Experimenty které využívají hodnoty z paměti není možné testovat na řešeních která nemají paměť, nicméně i přesto byly experimenty bez paměti spuštěny pro porovnání variant *Bez konstant* a s *Float konstanty*.

Tabulka 6.2: Tabulka s nastavením pravděpodobnosti konstant z intervalu nebo z paměti.

Varianta	Pravděpodobnost náhodné hodnoty [%]	Pravděpodobnost hodnoty z paměti [%]
Bez konstant	0	0
Float konstanty	10	0
Konstanty z paměti	0	10
Oba typy konstant	5	5

Další nastavení parametrů systému zůstává stejné jako v předchozím experimenty. Vychází tedy z tabulky 6.1 se sadami operací dle právě testované funkce.

Pro testování s datovými sadami S1 a S2 bylo stejně jako pro matematické funkce využito nastavení z předchozího experimentu pouze s rozdílem zdroje konstant ve stejné podobě, jako je popsáno v tabulce 6.2. Co se týče hodnot na vstupu, byla z dříve otestovaných variant využita ta obsahující i průměrné hodnoty jednotlivých konvolučních jader.

6.2.1 Výzkumné otázky

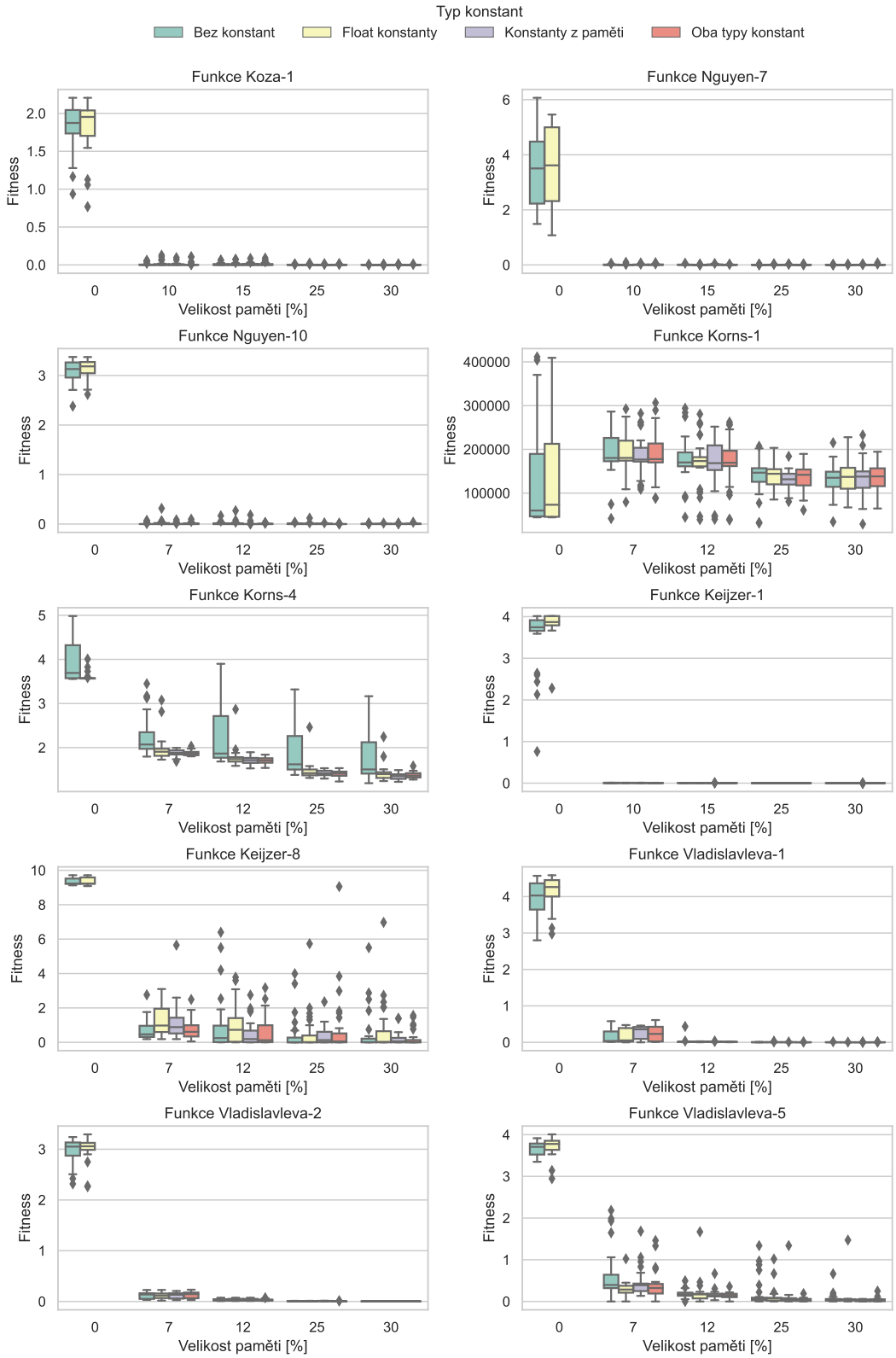
Cílem experimentů v této sekci je zodpovědět následující otázky:

1. Který typ generování konstant vede na minimální chybu symbolické regrese?
2. Liší se vliv při generování konstant z paměti v závislosti na velikosti paměti?
3. Mají varianty různý vliv na fitness během hledání řešení?
4. Které konstanty jsou používanější u varianty *Oba typy konstant*?

6.2.2 Výsledky experimentů

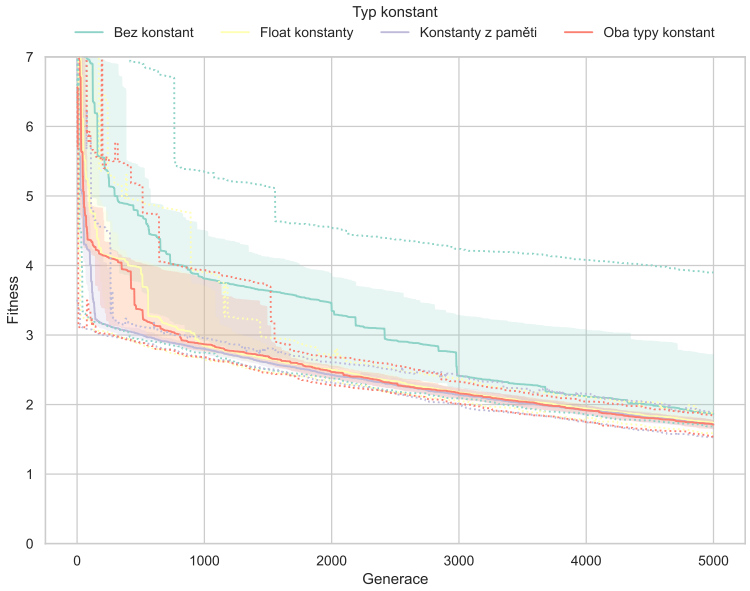
Pro otestování různých variant zdrojů konstant byl systém nejdříve otestován na matematických funkcích. Na obrázku 6.6 kde jsou vyneseny výsledky běhů s různým nastavením zdroje konstant pro všech 10 testovaných funkcí. Na těchto grafech všechny datové sady obsahují 10 % náhodných hodnot, protože ale počet náhodných hodnot by mohl mít vliv na výsledky experimentů, jsou grafy pro jiné množství náhodných hodnot v příloze C. Rozdíly mezi výsledky na obrázku 6.6 a v příloze C jsou však způsobeny počtem náhodných hodnot a z pohledu zdroje konstant pro funkci jsou rozdíly nevýznamné.

Co se týče samotných rozdílů jednotlivých variant je na obrázku 6.6 vidět, že u většiny funkcí jsou rozdíly velmi malé. Největší rozdíl je u funkce *Korns-4*, kde můžeme vidět, že varianta *Bez konstant* není tak stabilní jako ostatní varianty v nacházení dobrých výsledků. Přesto je schopna kvalitní výsledky vytvořit. Také můžeme vidět, že ve všech variantách je funkce *Korns-1* stále řešena velmi špatně, ale u těchto experimentů nebylo očekávané, že by se výsledky zlepšily.



Obrázek 6.6: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi s 10 % náhodných hodnot, při využití různých druhů zdroje konstant ve funkci.

Protože jsou rozdíly výsledků velmi malé, byla pro statistické ověření nevýznamnosti použita jednofaktorová ANOVA, která byla použita na výsledky vynesené na obrázku 6.6, konkrétně pak pro řešení s velikostí paměti 30 %. Během ověřování, zda výsledné fitness patří do normálního rozložení, provedených pomocí Shapiro-Wilkova testu, bylo zjištěno, že pro funkci Korns-1 nemá ani jedna z variant výsledky z tohoto rozložení. Nenáležitost do normálního rozložení byla nalezena i u funkcí *Korns-4* a *Vladislavleva-1*, nikdy ale ne pro všechny typy konstant. Co se týče výsledků samotné ANOVY tak s $\alpha = 0.05$ byla hypotéza, že výsledky jsou pro všechny varianty stejné zamítnuta pouze u funkce *Korns-4*, což je způsobeno větším rozptylem varianty bez konstant popsané výše.

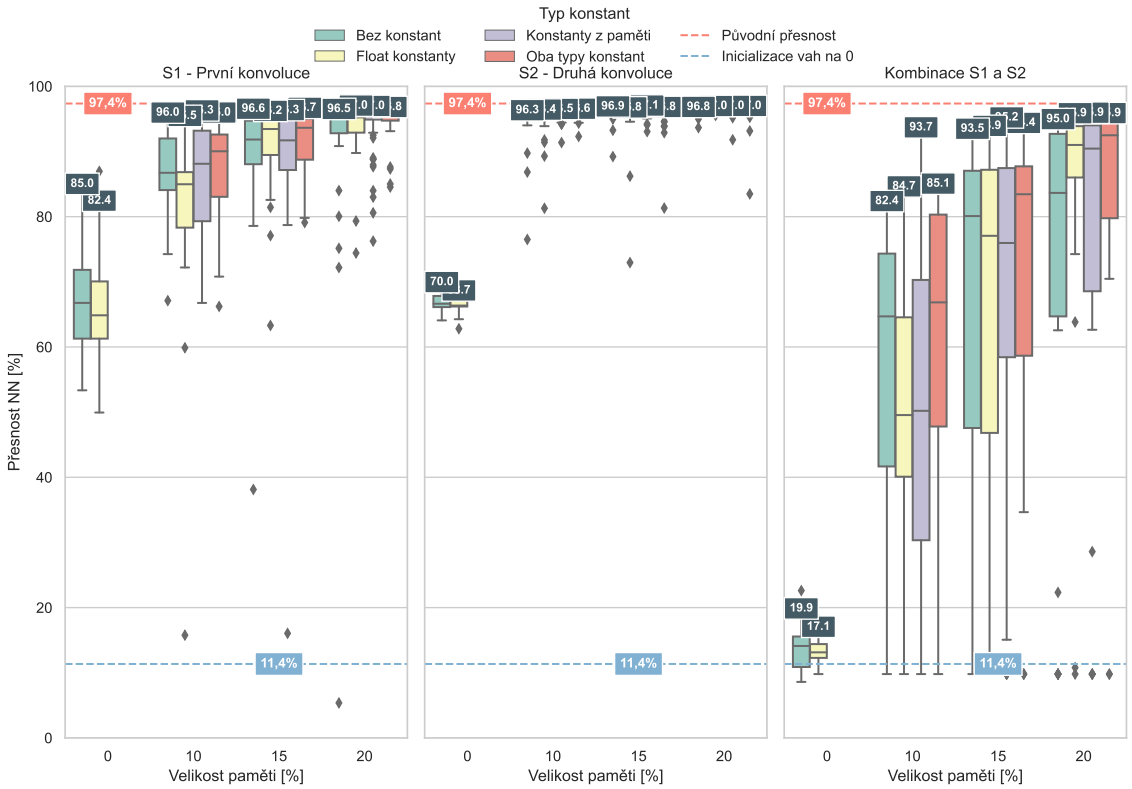


Obrázek 6.7: Průběhy fitness, medián a rozsah mezi 25 a 75 kvantilem, přerušovaně je pak vynesena maximální/minimální hodnota do vzdálenosti $1,5 \times IQR$ od 75/25 kvantilu, během generací s využitím různých zdrojů konstant pro funkci *Korns-4* při 10 % náhodných hodnot v datové sadě a systému s velikostí paměti 12 %.

Kromě dosažené fitness v poslední generaci by se druh generování konstant mohl projevit na průběhu fitness během jednotlivých generací. Na grafu 6.7 můžeme vidět tento průběh pro funkci *Korns-4* při 10 % náhodných hodnot v datové sadě a s pamětí o velikosti 12 %. Tedy běhům, při kterých není varianta *Bez konstant* tak stabilní v nalezení dobrých výsledků, jako varianty ostatní. Z průběhu fitness lze vidět, že varianta *Bez konstant* je horší během skoro celého průběhu. Další varianty jsou většinu generací velice srovnatelné vyjma varianty *Konstanty z paměti* která dosahuje lepších výsledků rychleji a je pak dorovnána zbývajícími dvěma variantami.

I u výsledků nad datovými sadami z vah neuronových sítí jsou výsledky dosti podobné jako u předchozích výsledků. Na obrázku 6.8 je zobrazena přesnost neuronových sítí. Stejně jako v předchozím experimentu jsou na třech grafech rozděleny výsledky tak, že na prvních dvou jsou výsledky pouze při generování vah pro jednu konvoluční vrstvu a na posledním, třetím, jsou kombinace výsledků.

Varianta bez konstant se v mnohých experimentech vyrovná variantám ostatním, nabízí se tedy otázka, zda se konstanty vůbec používají, případně v jakém množství. Na obrázku 6.9 můžeme vidět, že konstanty jsou více používané při řešení datových sad vy-



Obrázek 6.8: Dosažené přesnosti neuronové sítě na testovacích datech datasetu MNIST [22]. Během generování vah byly využity funkce, které mohly obsahovat konstanty z různých zdrojů.

cházejících z matematických funkcí, než u sad vytvořených z vah neuronových sítí. Také můžeme vidět, že při variantě *Oba typy konstant* je častěji použita konstanta z paměti.

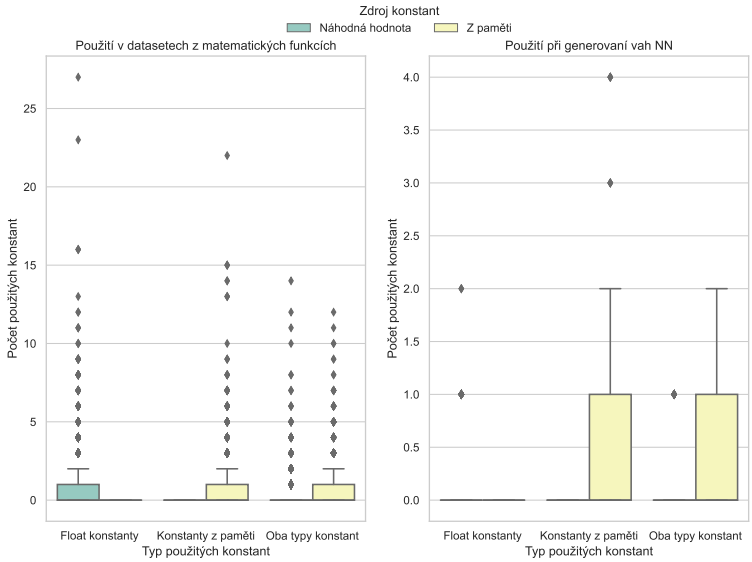
6.2.3 Závěr – Odpovědi na výzkumné otázky

1. **Který typ generování konstant vede na minimální chybu symbolické regrese?**

   V žádném experimentu se nedosáhlo výsledků, ve kterých by jedna z variant dosahovala statisticky lepších výsledků než ostatní tři varianty, ověření bylo u datových sad z matematických funkcí provedeno i pomocí ANOVY. Nicméně varianta využívající konstanty z paměti může vest k rychlejšímu dosažení lepších fitness a zároveň v řešeních využívající *Oba typy konstant* jsou konstanty z paměti více využívané. Protože řádná varianta nedodává lepší výsledky a zároveň jsou hodnoty z paměti využívané, tak se jako dobrá varianta nabízí kombinace, tedy varianta *Oba typy konstant*.

2. **Liší se vliv při generování konstant z paměti v závislosti na velikosti paměti?**

   Vliv zdroje konstant při jejich generování se nijak znatelně nemění v závislosti na velikosti paměti, ani při změně počtu náhodných hodnot v datových sadách u sad vytvořených z matematických funkcí.



Obrázek 6.9: Počty konstant z různých zdrojů pro matematické funkce (vlevo) a pro sady datové sady S1 a S2 (vpravo).

3. **Mají varianty různý vliv na fitness během hledání řešení?**

   Ve většině generací je vliv variant malý, ale v prvních generacích může být varianta *Konstanty z paměti* lepší. Důvodem pro tento jev bude pravděpodobně to, že pokud datová sada obsahuje vhodné konstanty pro použití ve funkci, tak je tímto zmenšen prohledávaný prostor a je tedy snazší nalézt zajímavé řešení.

4. **Které konstanty jsou používanější u varianty *Oba typy konstant*?**

   U této varianty byly jak v případě testovacích sad z matematických funkcí, tak u sad z vah neuronových sítí, používanější konstanty z paměti, ale konstanty z intervalu zde byly také použity.

6.3 Zpětná vazba

Cílem tohoto experimentu je otestovat možnost využití zpětné vazby v systému, kdy na vstupu kartézského genetického programování bude kromě vstupních hodnot i hodnota výsledná z předchozí iterace.

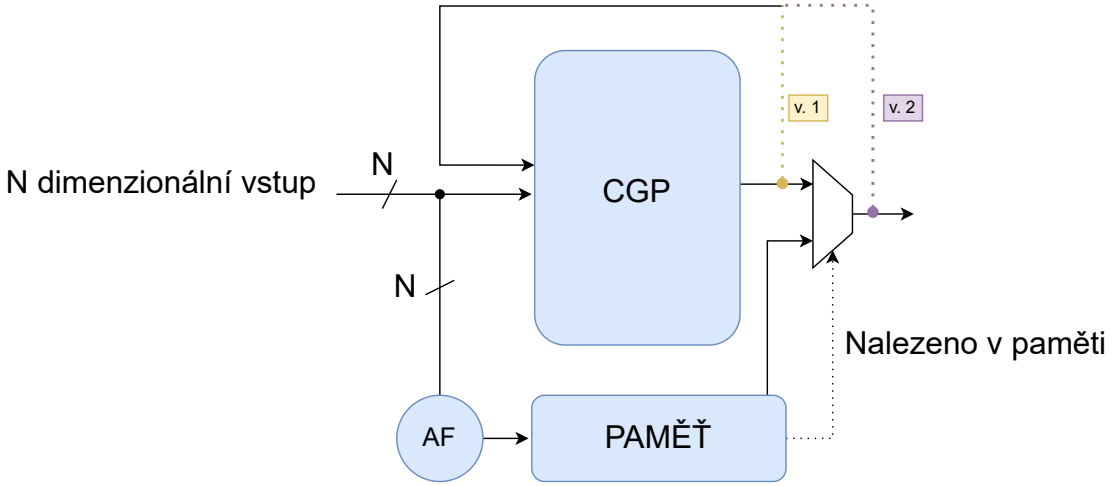
Nabízí se dvě varianty zapojení zpětné vazby do systému, kdy je na vstup přivedena předchozí hodnota:

v. 1: generovaná funkcí, nebo

v. 2: celého systému, tedy generovaná funkcí pokud není v paměti, jinak hodnota z paměti.

Schéma zapojení těchto variant lze vidět na obrázku 6.10, kde jsou obě varianty naznačeny různými barvami. Na schématu lze také vidět, že zpětná vazba je přivedena pouze jako vstup hledané funkce a ne jako hodnota určující klíčovou hodnotu v paměti, tedy není přivedena na vstup agregační funkce.

Je dobré připomenout, že ne všechny matematické funkce použité pro testování mají statickou velikost kroku při generování datové sady pro testování. Tímto je způsobeno, že



Obrázek 6.10: Schéma zobrazující dva způsoby rozšíření systému o zpětnou vazbu.

funkce s náhodným krokem mají náhodné pořadí bodů, což znemožňuje získání některých, ale ne všech, výhod zpětné vazby, jako je například postupné zvyšování hodnot.

Co se týče datových sad S1 a S2, tak zde jsou data vždy generována ve stejném pořadí a to tak, že váhy které jsou ze stejného konvolučního jádra jsou vždy generovány po sobě. Co se týče hodnot na vstupu pro tyto sady, byla z dříve otestovaných variant využita ta obsahující i průměrné hodnoty jednotlivých konvolučních jader.

Nastavení systému se stejně jako v předchozím experimentu odvíjí od nastavení v experimentu prvním, tedy tabulky 6.1. Jediným rozdílem je nastavení zdroje konstant, které bylo řešeno v předchozím experimentu. V tomto experimentu je využito nastavení *Oba typy konstant*, tedy nastavení při kterém je pravděpodobnost konstanty z paměti nebo z intervalu $[-1\,000, 1\,000]$ vždy 5 %, tedy dohromady je pravděpodobnost konstant 10 %. Protože zpětná vazba bude testována i při nastavení velikosti paměti na 0, tedy bez paměti, bude v těchto případech využito nastavení zdroje konstant z intervalu na 10 %. Pravděpodobnost některé konstanty je tak ve všech bězích nastavena na 10 %.

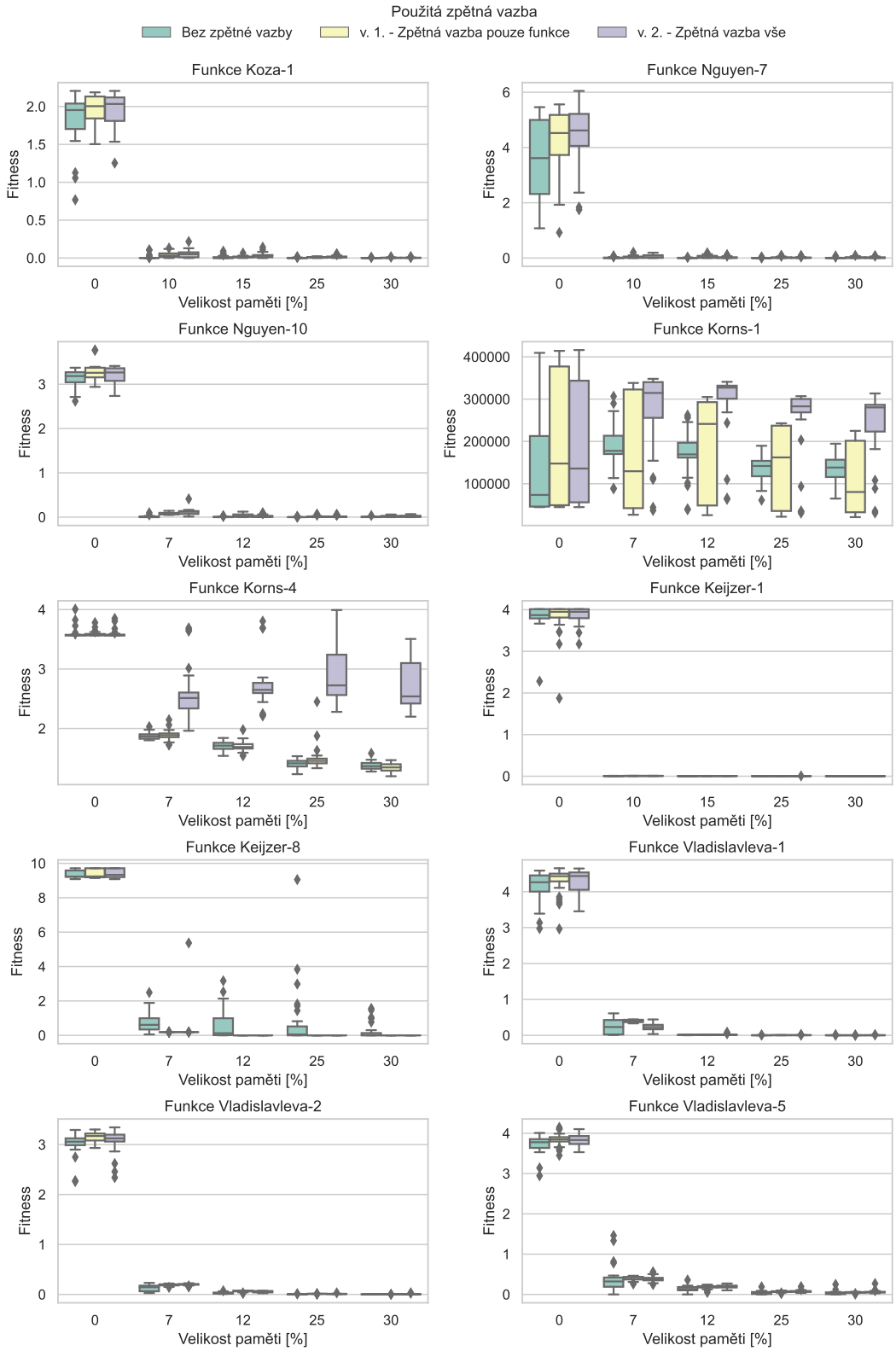
6.3.1 Výzkumné otázky

Cílem experimentů v této sekci je zodpovědět následující otázky:

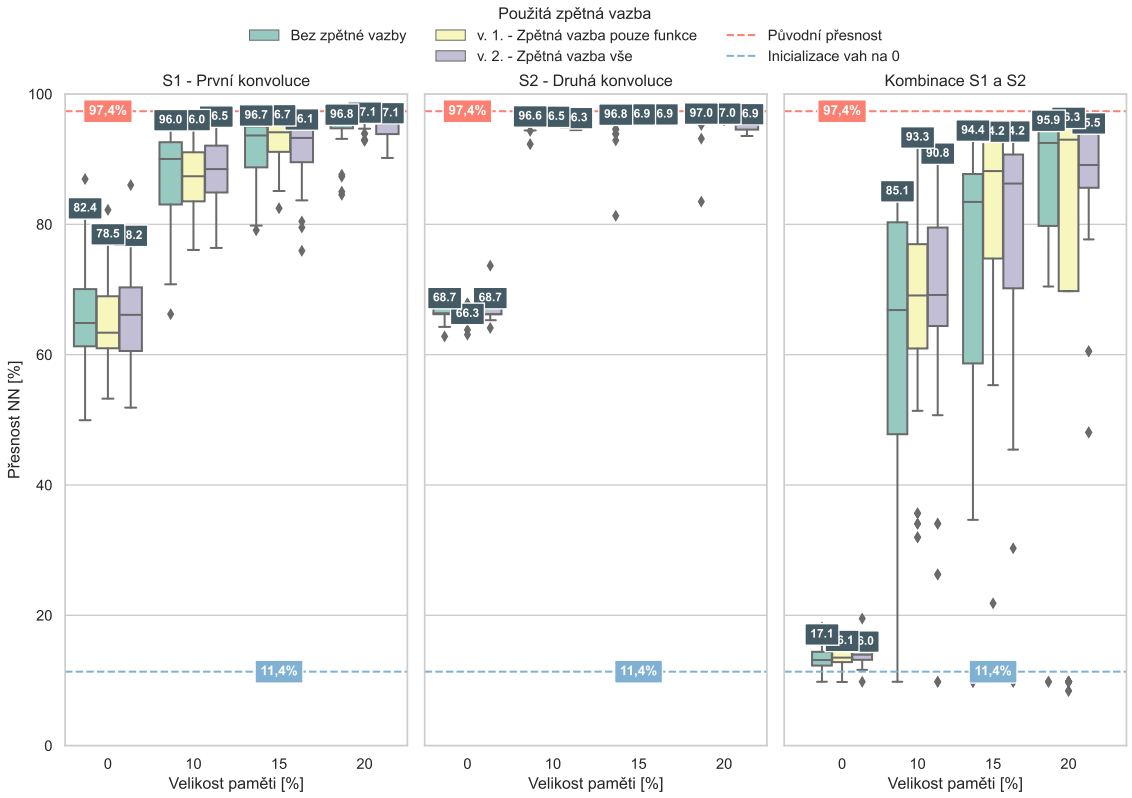
1. Pomáhá možnost zpětné vazby k nalezení lepších řešení úloh v testovací sadě?
2. Je vstup zpětné vazby evolucí využíván?
3. Která varianta zpětné vazby dosahuje lepších výsledků?
4. Která varianta je častěji používána?

6.3.2 Výsledky experimentů

Grafy na obrázku 6.11 zobrazují výsledky pro experimenty nad matematickými funkcemi, přičemž ve většině případů jsou výsledky velice podobné jako výsledky *Bez zpětné vazby*. Výsledek, který nabývá znatelně jiných hodnot můžeme vidět u funkce *Korns-4*, kde 2. varianta zpětné vazby dosahuje výrazně horších výsledků, než varianta 1. a varianta *Bez zpětné*



Obrázek 6.11: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi s 10% náhodných hodnot, při využití různých zapojení zpětné vazby v systému.



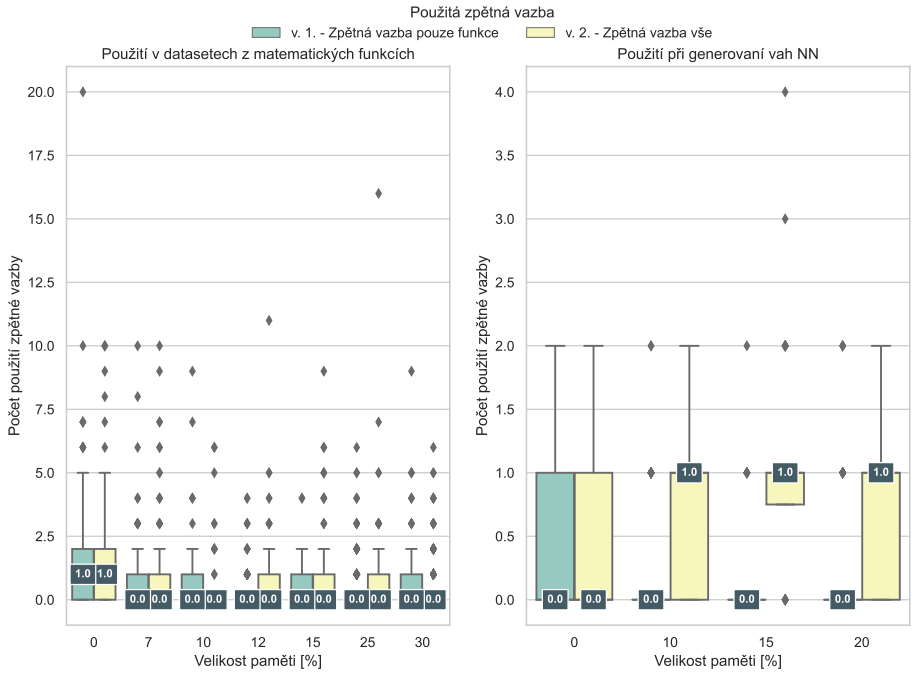
Obrázek 6.12: Dosažené přesnosti neuronové sítě na testovacích datech datasetu MNIST [22] při využití různých zapojení zpětné vazby v systému.

vazby. Dále pak u funkcí *Nguyen* bez paměti můžeme vidět, že varianta *Bez zpětné vazby* dosahuje mírně lepších výsledků než varianty s některou ze zpětných vazeb. Pro ověření, že tento rozdíl je statisticky významný byla použita ANOVA s $\alpha = 0.05$, která hypotézu, že jsou řešení stejná, zamítla.

U funkce *Korns-1* můžeme vidět, že výsledky jsou u variant se zpětnou vazbou méně stabilní, a stále nedochází k zlepšení řešení. Zlepšení vlivem zpětné vazby by ale bylo u této funkce zvláštní i když ne nemožné, protože krok této funkce je náhodný.

Na grafech v obrázku 6.12 je vynesena přesnost neuronové sítě pro testovací sadu datasetu MNIST, když má genetické programování s pamětí možnost využít zpětnou vazbu. Výsledky jsou stejně jako u datových sad z matematických funkcí velice podobné výsledkům *Bez zpětné vazby*.

Protože jsou výsledky skoro ve všech testovacích případech velice podobné výsledkům, ve kterých není možnost využít žádné zpětné vazby, nabízí se otázka, zda je zpětná vazba využívána v nalezených funkcích. Na grafech v obrázku 6.13 jsou vyneseny počty použití pro zpětné vazby rozdělené podle datových sad z matematických funkcí (vlevo) a sad S1, S2 vytvořených z vah neuronových sítí. Číselně vynesené jsou pak mediány počtu použití. Nejčastější medián počtu použití je 0, tedy zpětná vazba není příliš často používána. Také si zde lze všimnout, že při generování vah neuronových sítí je používanější zpětná vazba 2. varianty.



Obrázek 6.13: Počet použití různých variant zpětných vazeb, pro testovací sady.

6.3.3 Závěr – Odpovědi na výzkumné otázky

1. **Pomáhá možnost zpětné vazby k nalezení lepších řešení úloh v testovací sadě?**

   V testovaných případech byla dosažená fitness při umožnění použití zpětné vazby podobná jako bez této možnosti. V určitých případech, například u funkce *Korns-4*, vedla možnost přidaní zpětné vazby ve variantě 2. ke zhoršení dosahované fitness.

2. **Je vstup zpětné vazby evolucí využíván?**

   Vstup zpětné vazby využíván je, ale ne příliš často. Nejčastější mediální počet použití, při různých nastaveních velikosti paměti, je nula, při některých experimentech se dosáhlo jednoho mediálního počtu použití.

3. **Která varianta zpětné vazby dosahuje lepších výsledků?**

   Ve většině případů dosahují výsledky podobných hodnot fitness. U funkcí *Korns* dosahuje však 2. varianta horších výsledků než varianta 1.

4. **Která varianta je častěji používána?**

   U datových sad vycházejících z matematických funkcí jsou varianty zpětné vazby používány podobně často. Rozdíl v počtu použití je však patrný u variant pro generování vah neuronových sítí, kde je častěji používána 2. varianta.

6.4 Testování agregační funkce XOR

Tento experiment se zabývá využitím operace XOR v agregační funkci genetického programování s pamětí. Hlavní motivací pro využití operace XOR v agregační funkci pro paměť je zmenšení generovaného klíče a tím zmenšení celkové velikosti asociativní paměti.

Jako cílová velikost byla zvolena velikost jedné vstupní hodnoty. Tedy pro funkce na jejichž vstupu je vícero vstupních hodnot byly tyto hodnoty sjednoceny operací XOR do jedné hodnoty. Nabízí se pomocí operace XOR zmenšit bitovou šířku hodnoty ještě více, ale tento způsob nebyl zvolen, protože povede na větší množství kolizí (různých sad vstupů při kterých má agregační funkce stejný výstup), jak metoda zvolené která vstup zmenšuje méně. Tyto, z pohledu velikosti klíče lepší, varianty mohou být otestovány dále, pokud bude testovaná varianta dosahovat dobrých výsledků.

Protože ne všechny funkce v navržené sadě funkcí mají na svém vstupu vícero hodnot, byly funkce pouze s jedním vstupem pro tento experiment vyřazeny.

Co se týče hodnot na vstupu pro datové sady S1 a S2, byla využita ta varianta obsahující i průměrné hodnoty jednotlivých konvolučních jader. Ke změně této varianty může dojít, pokud se prokáže vhodnost tohoto způsobu minimalizace bitové velikosti klíče v paměti.

Nastavení systému je stejné jako v předchozím experimentu a není zde využita žádná zpětná vazba. Protože není důvod tyto experimenty provádět v případech, kdy není agregační funkce využívána, tedy v případech, kdy je velikost paměti nastavena na 0, tak je ve všech případech využíváno typu konstant *Oba typy konstant*, představené v experimentu v sekci 6.2.

6.4.1 Výzkumné otázky

Cílem experimentů v této sekci je zodpovědět následující otázky:

1. Jaká má vliv nahrazení agregační funkce na fitness u datových sad generovaných z matematických funkcí?

2. Jaký je tento vliv na přesnost neuronové sítě?

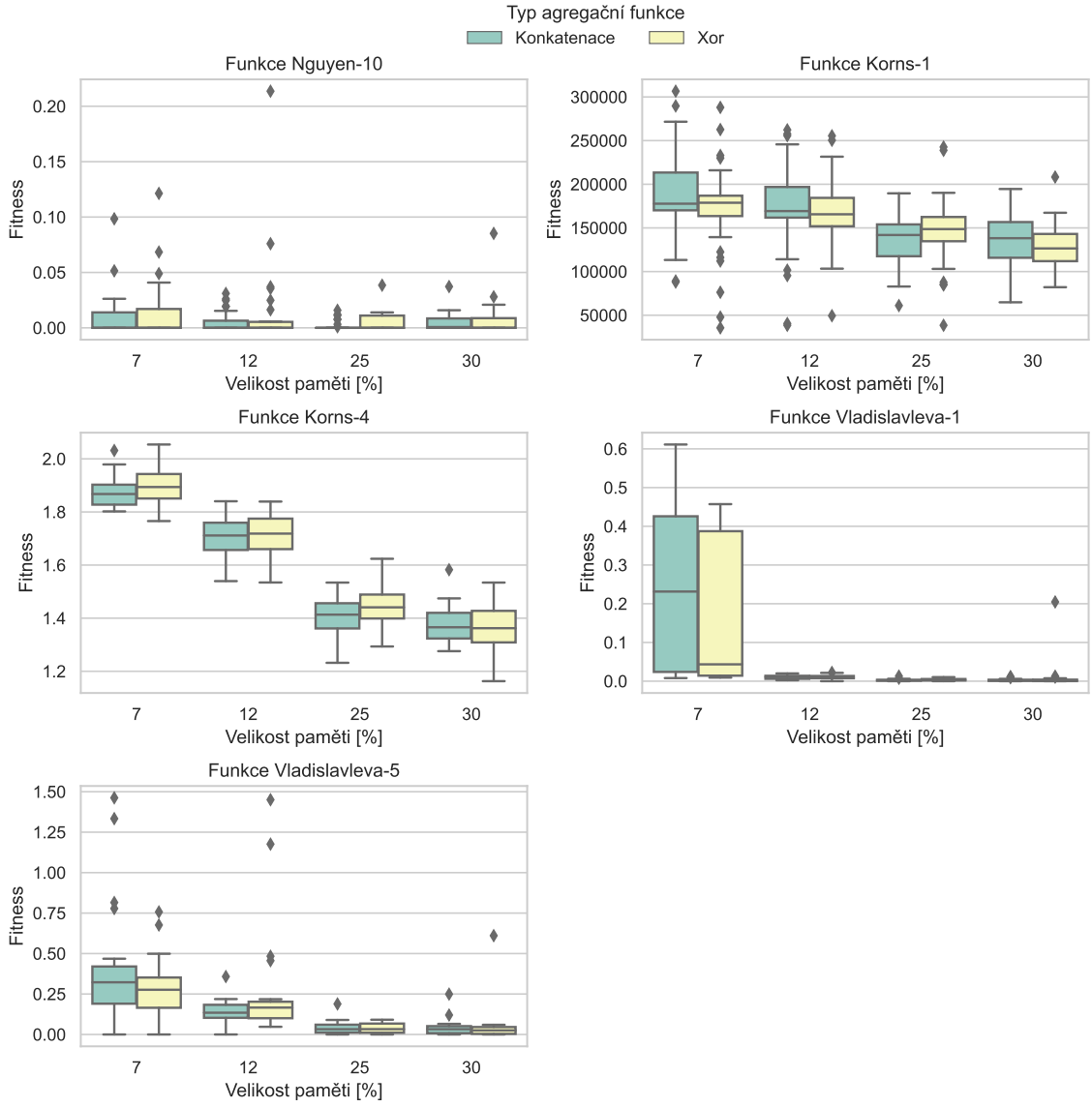
6.4.2 Výsledky experimentů

Na obrázku 6.14 jsou vyneseny grafy s výslednou fitness pro funkce, které mají na svém vstupu dvě nebo více hodnot, přičemž je varianta s agregační funkcí XOR porovnávána s výsledky při použití varianty s konkatenací. Lze vidět, že rozdíly mezi těmito variantami jsou malé.

Při ověřování, zda jsou tyto malé rozdíly statisticky významné, bylo při $\alpha = 0.05$ zjištěno, že ne všechny skupiny výsledků mají vlastnosti normálního rozložení, přičemž nejčastěji se toto rozložení nepotvrdilo u funkcí *Korns*. Při použití ANOVY při stejné hodnotě α byl pouze jeden případ, kdy jsou výsledky rozdílné, a to u funkce *Nguyen-10* při velikosti paměti 25 %, kde varianta s agregační funkcí XOR dosahuje horších výsledků.

Na dalším obrázku 6.15 můžeme vidět porovnání variant s agregačními funkcemi XOR a konkatenace pro datové sady S1 a S2. Lze vidět, že rozdíly mezi variantami jsou zde větší než u datových sad generovaných z matematických funkcích. Přesto zde bylo provedeno statistické vyhodnocení pomocí ANOVY při $\alpha = 0.05$, kdy bylo zjištěno že varianty jsou statisticky rozdílné u obou sad (S1 a S2) a při všech nastaveních velikosti paměti, přičemž z grafů je patrné, že varianta s agregační funkcí XOR je vždy horší. Také bylo zjištěno, že u výsledků při datové sadě S2 a velikosti paměti 20 % nemají hodnoty přesnosti sítě

charakter normálního rozložení, při využití stejné hodnoty α jako u ANOVY, který ostatní výsledky mají.

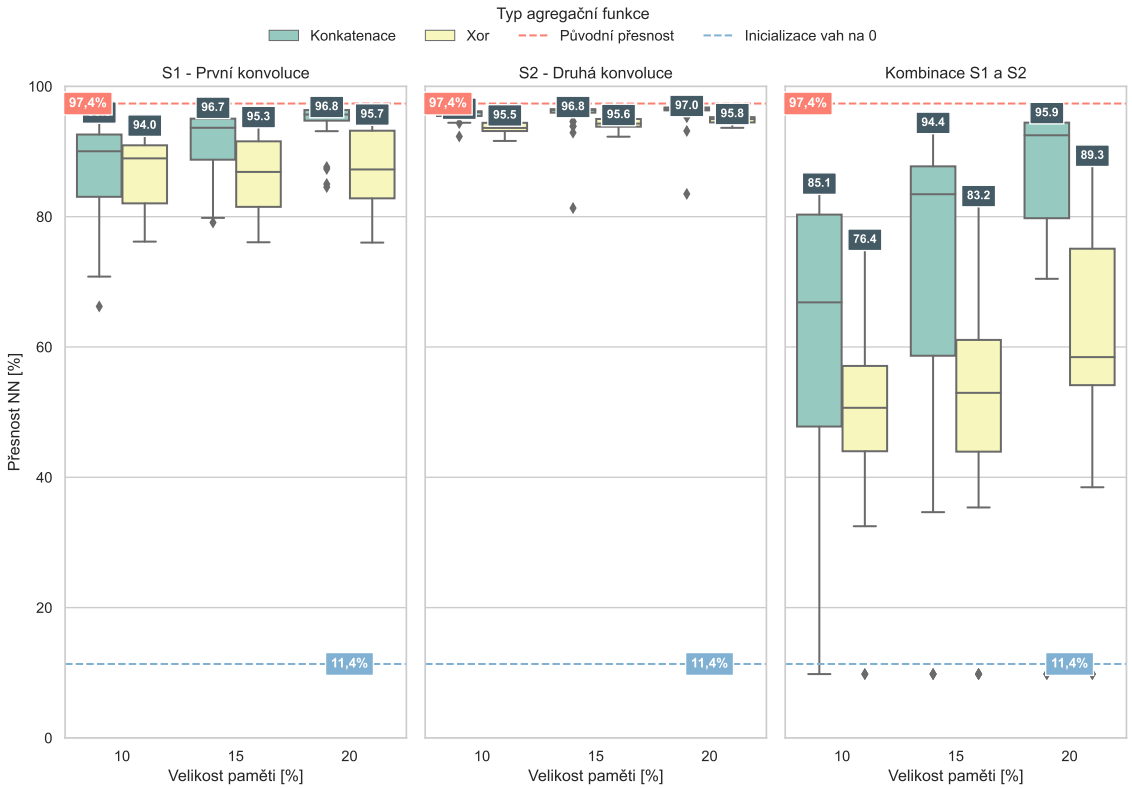


Obrázek 6.14: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi s 10 % náhodných hodnot, při využití různých druhů agregační funkce.

6.4.3 Závěr – Odpovědi na výzkumné otázky

1. **Jaká má vliv nahrazení agregační funkce na fitness u datových sad z matematických funkcí?**

   Vliv změny agregační funkce z konkatenace na operaci XOR pro agregaci vícero stejně velkých vstupů do velikosti jednoho z nich, nemá statisticky významný vliv na fitness na těchto testovacích sadách. Toto bylo ověřeno pomocí ANOVY při $\alpha = 0.05$.



Obrázek 6.15: Dosažené přesnosti neuronové sítě na testovacích datech datasetu MNIST [22], při využití agregační funkce XOR a konkatenace.

2. **Jaký je tento vliv na přesnost neuronové sítě?**

   Při testování agregační funkce XOR bylo na testovacích sadách S1 a S2 zjištěno zhoršení přesnosti neuronové sítě, pro kterou byly váhy generovány ve všech provedených případech. Statistická významnost tohoto zhoršení byla opět otestována pomocí ANOVY při $\alpha = 0.05$.

   Cílem tohoto experimentu bylo nalézt možnost zmenšení klíčů hodnot ukládaných v paměti. Zmenšení této hodnoty, pokud možnost bez efektu na výsledné řešení, je důležité zvláště u datových sad generujících váhy pro neuronové sítě a protože při využití agregační funkce XOR došlo ve všech případech ke zhoršení výsledků sítě, není tato varianta vhodná.

6.5 Testování agregační funkce typu interval

Podobně jako předchozí experiment se i tento zabývá funkcionalitou agregační funkce generující klíč do paměti. V tomto případě je hlavní motivací předpoklad, že v některých případech (datových sadách) může docházet k shlukovaní podobných hodnot, které je vhodné uložit do paměti.

Agregační funkce v tomto experimentu pracuje se souřadnicemi hodnot uložených v paměti a je jejím cílem tyto hodnoty použít i pro vstupy, které se nachází v určité vzdálenosti

od této hodnoty, tedy nachází se v určitém intervalu. Tato vzdálenost může být definována různými způsoby:

- vzdálenost může být absolutní nebo relativní v závislosti na hodnotě uložené v paměti,
- při datových sadách s více vstupními hodnotami (dimenzemi) může být pro každou tuto dimenzi jiný interval a
- velikost vzdálenosti může být zadána pevně nebo bude hledána pomocí evoluce z určitého intervalu.

Při variantě, kdy bude vzdálenost zadána pevně, je třeba určit na jaké konkrétní hodnotě, a u variant, ve kterých ji může mutace během běhu ovlivňovat, je třeba určit s jakou pravděpodobností takováto mutace nastane, a v jakém intervalu bude tato vzdálenost hledaná.

Při generování vah dle datových sad S1 a S2, byla využita varianta obsahující i průměrné hodnoty jednotlivých konvolučních jader na vstupu genetického programování s pamětí.

Pro funkce, které mají v datové sadě náhodný krok, byly hodnoty velikosti intervalu vypočteny pomocí vztahu 6.3:

$$Velikost\_intervalu = \frac{h\frac{i}{m}}{2} \tag{6.3}$$

kde hodnota i je velikost celkového intervalu, ve kterém se generují body z matematické funkce a m označuje maximální počet prvků, který se uloží do paměti při jejím největším nastavení, které je v experimentech používáno. Hodnota h je pak závislá na aktuálně počítané hodnotě a je uvedena dále. Pro funkce, které mají hodnoty v pravidelném intervalu, byl tento interval vynásoben konstantou k. Hodnoty h a k jsou pro jednotlivé sloupce v tabulkách následující:

- Počáteční velikost
  - $k = 1,2$
  - $h = 0,3$
- Spodní hranice
  - $k = 0,2$
  - $h = 0,003$
- Horní hranice
  - $k = 2,2$
  - $h = 0,6$

V tabulce 6.3 jsou uvedeny vypočtené hodnoty velikostí intervalu. Jsou zde také hodnoty při relativním výpočtu vzdálenosti podle toho, jaká je hodnota uložená v paměti. Tyto hodnoty byly získány tak, že odpovídající hodnoty pro absolutní velikost byly vynásobeny průměrnou hodnotou z absolutních hodnot, kterých můžou hodnoty uložené v paměti nabývat.

Protože se v tomto experimentu testuje také pouze vliv agregační funkce jako v experimentu popsaném v sekci 6.4, bylo využito stejné nastavení systému. Jediné, co se mění, je agregační funkce. Dále je přidáno nastavení nutné pro funkci této nové agregační funkce zda je interval hledaný evolucí nebo zůstává na počátečních hodnotách a jaká je pravděpodobnost jeho modifikace během mutace paměti, pokud je hledaný. Tato pravděpodobnost mutace byla ve všech experimentech nastavena na 5 % ze všech mutací které se provedou v paměti.

Tabulka 6.3: Maximální vzdálenosti, při kterých je hodnota ještě brána z paměti, případně interval, ve kterém je tato vzdálenost hledána. Tyto hodnoty se dále liší podle toho, zda je vzdálenost absolutní nebo relativní.

Jméno funkce	Počáteční velikost		Spodní hranice		Horní hranice	
	Absolutní	Relativní	Absolutní	Relativní	Absolutní	Relativní
Koza-1	0,06	0,12	0,01	0,02	0,11	0,22
Nguyen-7	0,05	0,05	0,000 5	0,000 5	0,1	0,1
Nguyen-10	0,01	0,02	0,000 1	0,000 2	0,02	0,04
Korns-1	0,005	0,000 2	0,000 05	0,000 002	0,01	0,000 4
Korns-4	0,005	0,000 2	0,000 05	0,000 002	0,01	0,000 4
Keijzer-1	0,12	0,24	0,02	0,04	0,22	0,44
Keijzer-8	1,2	0,024	0,2	0,004	2,2	0,044
Vladislavleva-1	0,019	0,008 6	0,000 19	0,000 086	0,037	0,017
Vladislavleva-2	0,12	0,024	0,02	0,004	0,22	0,044
Vladislavleva-5	0,002 5	0,002 1	0,000 017	0,000 011	0,006 5	0,006 3

6.5.1 Výzkumné otázky

Cílem experimentů v této sekci je zodpovědět následující otázky:

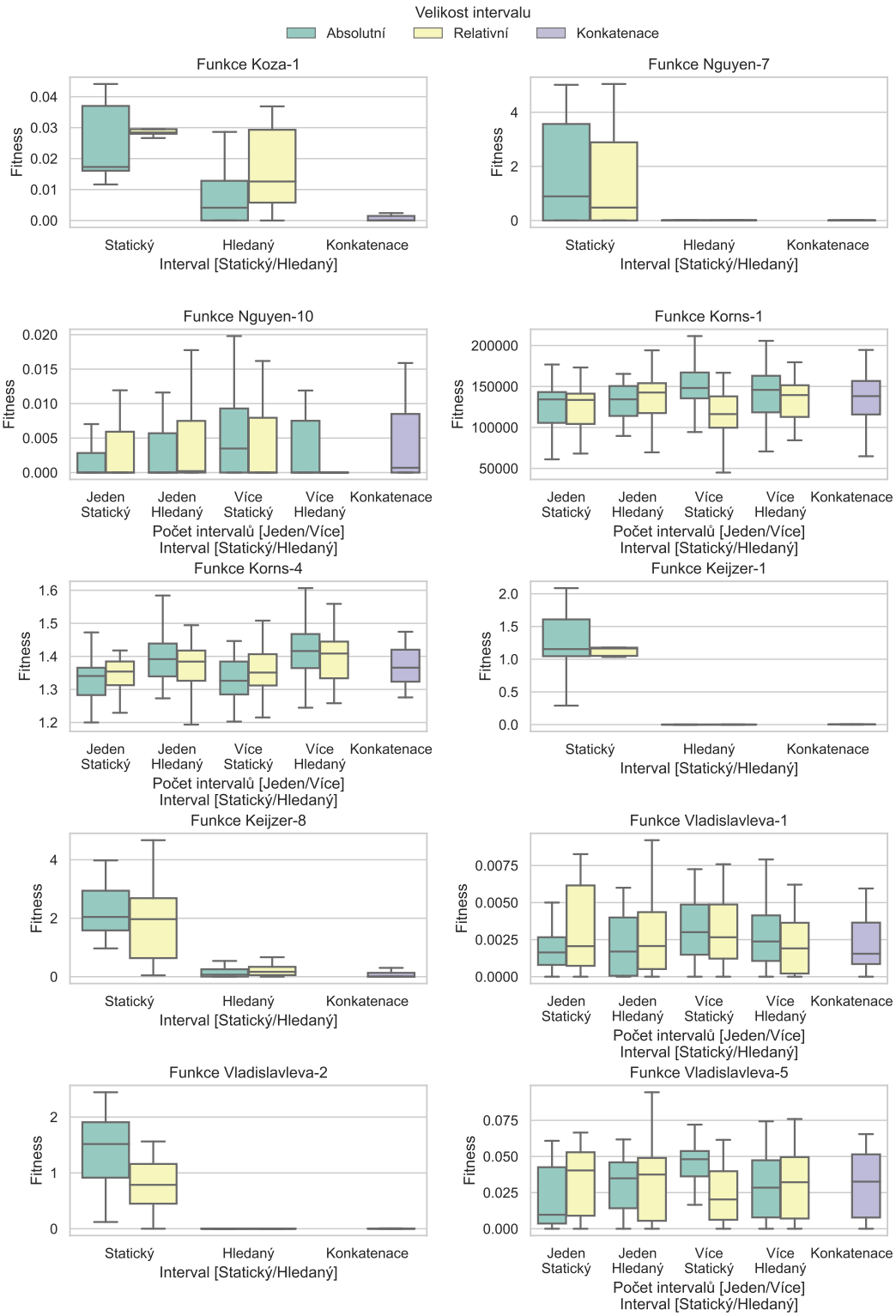
1. Jak se výsledné hodnoty fitness mění oproti agregační funkci využívající konkatenaci?
2. Má smysl maximální možnou vzdálenost hledat nebo používat pouze statické hodnoty?
3. Je lepší absolutní nebo relativní velikost, která varianta dosahuje menší konečné fitness hodnoty?
4. Přináší varianta obsahující jinou maximální vzdálenost pro každou vstupní dimenzi pozitivní efekt?

6.5.2 Výsledky experimentů

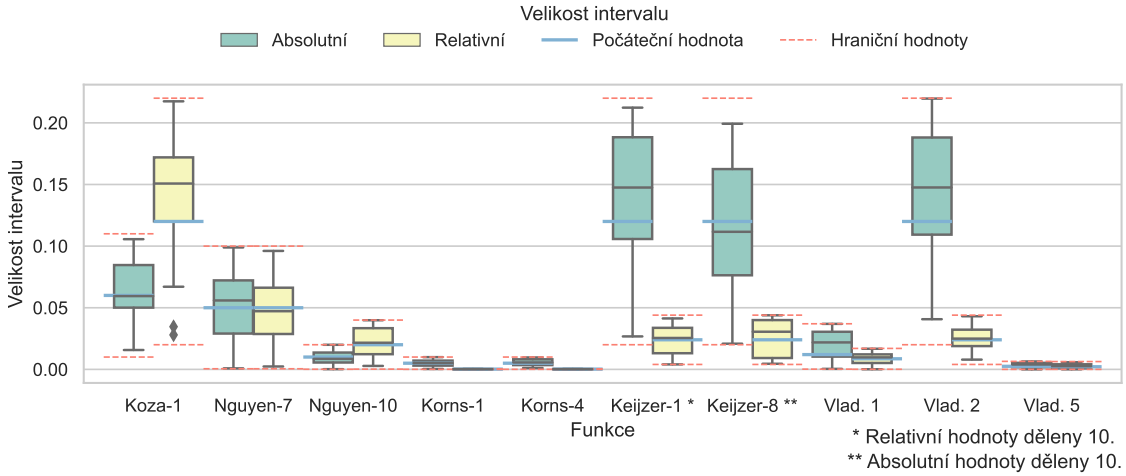
Na obrázku 6.16 jsou vyneseny finální fitness hodnoty pro různé varianty intervalů. Dále jsou zde pro porovnání vyneseny výsledky pro agregační funkci typu konkatenace. Na ose x jsou u funkcí, které mají jenom jednu vstupní dimenzi, výsledky rozděleny podle toho, zda je velikost intervalu statická, nebo je jeho velikost hledaná. U funkcí, které obsahují na vstupu více dimenzí, je toto rozdělení rozšířeno ještě o další parametr, kterým je počet intervalů, tedy zda je velikost intervalů jediná nebo je pro každou vstupní dimenzi hledaná jiná velikost.

Hodnoty v těchto grafech dosahují podobných hodnot ve všech variantách s ojedinělými případy, ve kterých je varianta s hledanou velikostí intervalu lepší, než varianty se statickým intervalem. Nejlepší varianty využívající agregační funkci interval většinou dosahují stejné nebo horší výsledné fitness jako varianta s konkatenací. To je samozřejmě způsobeno tím, že v těchto sadách jsou ojedinělé hodnoty náhodné a tedy výskyt skupin odlehlých hodnot je čistě náhodný. Zajímavější tedy budou výsledky na datových sadách S1 a S2.

Nejdříve jsou ale na obrázku 6.17 vyneseny velikosti intervalů u řešení, kde byla umožněna mutace této velikosti. U funkcí jsou kromě velikostí také vyneseny počáteční a hraniční hodnoty, které byly dříve vyneseny v tabulce 6.3. Pro experimenty, které hledaly velikost



Obrázek 6.16: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi s 10 % náhodných hodnot, při využití agregační funkce konkatenace a různých druhů intervalu.



Obrázek 6.17: Velikosti intervalů ve výsledných řešeních, kdy byla umožněna mutace velikosti těchto intervalů.

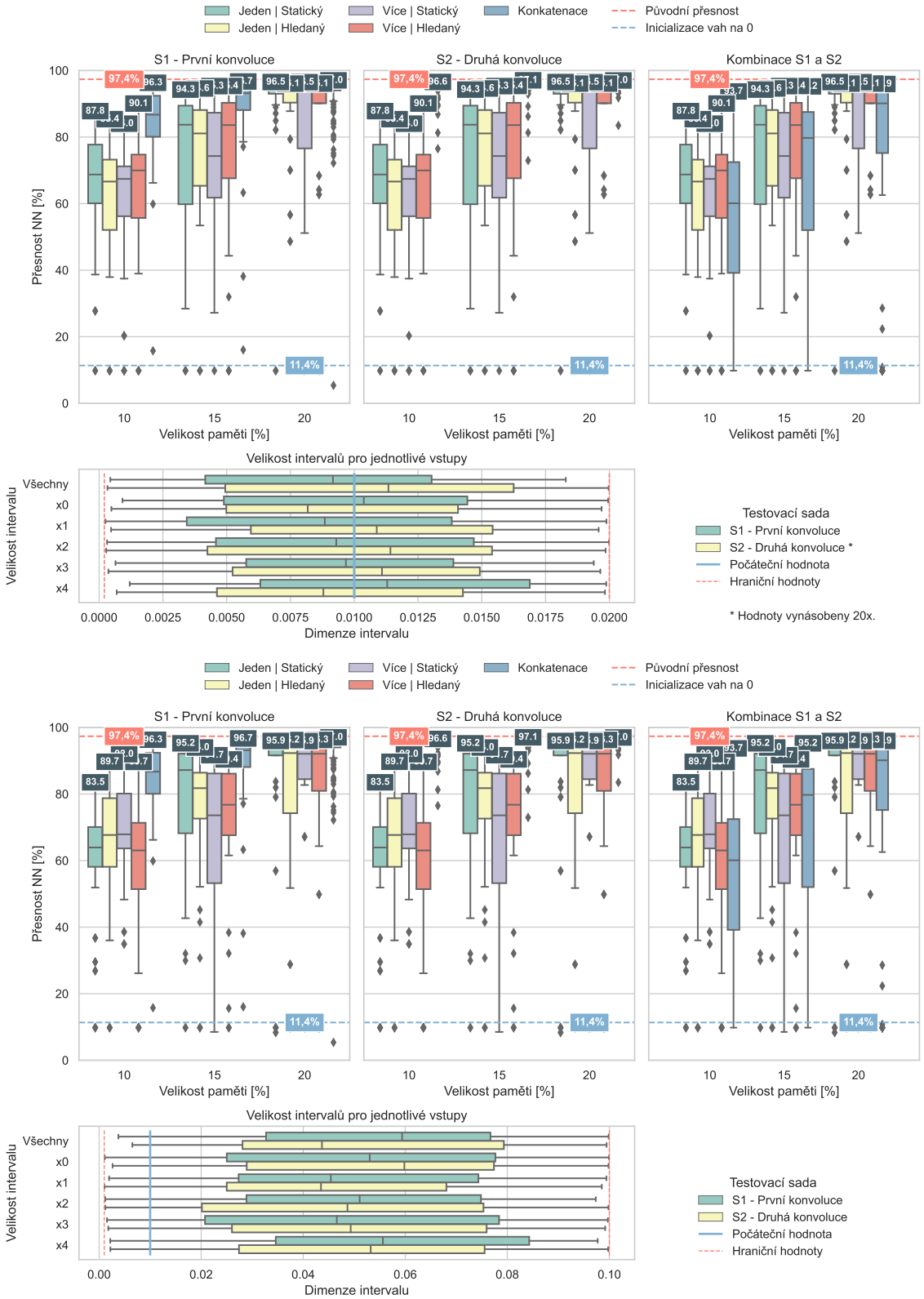
intervalu pro každou vstupní dimenzi odděleně, byly do tohoto grafu zesumarizovány do jednoho boxu, protože zde nedocházelo k významným rozdílům. Lze vidět, že u žádného nastavení nedocházelo k častému využívání krajních hodnot z prohledávaného intervalu. Proto jsou pravděpodobně hodnoty v tabulce 6.3 zvoleny vhodně.

Výsledky pro sady S1 a S2 byly kvůli přehlednosti rozděleny na dvě skupiny grafů na obrázku 6.18, kdy v horní polovině jsou výsledky pro datovou sadu využívající absolutní velikost intervalu a v dolní polovině pak výsledky popisující nastavení s relativní velikostí. Přesnost sítě u řešení, která využívají interval jako agregační funkci, však nikdy nedosahují takové přesnosti výsledné sítě jako řešení využívající konkatenaci.

Na obrázku 6.18 jsou také věnovány dva grafy pro velikost intervalů z běhů, ve kterých tato velikost mohla mutovat. Na ose y je uvedeno, pro kterou vstupní dimenzi je daný interval použit, kde jednotlivá označení znamenají:

- **Všechny** – pro nastavení, kdy je interval pro všechny dimenze stejný,
- **x0** – index vstupního kanálu,
- **x1** – index výstupního kanálu,
- **x2** – souřadnice osa x,
- **x3** – souřadnice osa y, a
- **x4** – průměrná hodnota konvolučního jádra.

Jsou zde také uvedeny hranice, mezi kterými se hodnota mohla pohybovat, a počáteční hodnota. U absolutní velikosti je možné vidět, že nedochází k žádným výrazným odchylkám. Mediány velikostí se nachází v okolí počáteční hodnoty. U relativních velikostí ale dochází ke zvětšení mediálních velikostí intervalu, kdy medián je 5× až 6× větší než počáteční hodnota.



Obrázek 6.18: Dosažené přesnosti pro datové sady S1 a S2 při využití agregační funkce interval. Velikost intervalu je pro grafy v horní polovině absolutní a v dolní polovině je velikost relativní

6.5.3 Závěr – Odpovědi na výzkumné otázky

1. **Jak se výsledné hodnoty fitness mění oproti agregační funkci využívající konkatenaci?**

   Hodnoty využívající agregační funkci interval dosahují v nejlepších případech stejných výsledných hodnot jako hodnoty využívající konkatenaci.

   U datových sad S1 a S2 je výsledek vždy lepší u varianty s agregační funkcí konkatenace. Nejvíce jsou výsledky srovnatelné u přesnosti sítě využívající kombinaci výsledků.

2. **Má smysl maximální možnou vzdálenost hledat nebo používat pouze statické hodnoty?**

   Má smysl umožnit mutaci měnit velikost intervalu. Ve většině případů, kdy byla varianta využívající interval horší než varianta s konkatenací, dosahovaly varianty s hledanou velikostí intervalu lepších výsledků než varianta se statickou velikostí.

3. **Je lepší absolutní nebo relativní velikost, která varianta dosahuje menší konečné fitness hodnoty?**

   Tyto varianty u většiny datových sad dosahovaly podobných výsledků. U žádné datové sady nevznikl výrazný rozdíl mezi těmito dvěma variantami.

4. **Přináší varianta obsahující jinou maximální vzdálenost pro každou vstupní dimenzi pozitivní efekt?**

   Stejně jako varianty absolutní/relativní velikost intervalu ani tato varianta nepřinesla pozorovatelný vliv na kvalitu výsledného řešení.

6.6 Zmenšení velikosti výsledného řešení

V předchozích experimentech bylo otestováno několik možných řešení aplikace genetického programování s pamětí, které byly testovány na testovacích sadách vycházejících z matematických funkcí a na sadách S1 a S2, které obsahovaly váhy konvolučních vrstev neuronové sítě. Tato sekce je zaměřená na zmenšení velikosti paměti nutné pro uložení řešení pro generování vah neuronových sítí. Také je kladen důraz na náročnost generování těchto vah po načtení uloženého řešení z paměti.

Tato sekce je rozdělena do dvou podsekcí. V první je řešena optimalizace výrazu G používaného pro generování většiny vah snížením bitové šířky operací a ve druhé pak optimalizace přístupu do paměti.

6.6.1 Výzkumné otázky

Cílem experimentů v této sekci je zodpovědět následující otázky:

1. Optimalizace funkce:

   1.1. Zhorší se výsledky na testovacích sadách při změně operací které CGP využívá?

1.2. Jaká je velikost paměti potřebné pro uložení výrazu vytvořeného CGP?

2. Optimalizace paměti:

   2.1. Jaké metody jsou vhodné pro zmenšení velikosti paměti?

   2.2. Kolik bitů je potřeba pro uložení paměti, která je součástí řešení s dobrými výsledky?

3. Kolik bitů je potřeba pro uložení celkového řešení, které dodává dobré výsledky?

4. Jak si navržená metoda vede oproti alternativním kompresním metodám?

6.6.2 Optimalizace výrazu

V tomto experimentu je testováno, jaký vliv má změna množiny operací, kterou používá kartézské genetické programování během hledání výrazu G. Bude využita množina 8bitových operací, protože je cílem snížit hardwarovou náročnost výsledného systému. Následně je také navrženo několik možností pro uložení výrazu do paměti.

Tabulka 6.4: Základní nastavení kartézského genetického programování s pamětí.

Parametr	Hodnota
velikost populace	$1 + \lambda$, $\lambda = 4$
počet generací	5000
velikost CGP (sloupce × řádky)	20×10
L-back	maximální
počet mutací	2
pravděpodobnost mutace paměti	20 %
pravděpodobnost náhodné konstanty	
při 32bitové sadě operací	5 % (10 % při velikosti paměti 0)
při 8bitové sadě operací	0
pravděpodobnost konstanty z paměti	
při 32bitové sadě operací	5 % (0 % při velikosti paměti 0)
při 8bitové sadě operací	0

Nastavení genetického programování s pamětí zůstane pro tento experiment podobné jako pro předchozí experimenty. To je zrekapitulováno v tabulce 6.4 kvůli změnám, které byly provedeny na základě výsledků dříve provedených experimentů. Jako agregační funkce je použita konkatenace, která byla zvolena z důvodů popsaných dále v podsekci 6.6.4 při optimalizaci paměti. Jako sada možných operací v hledaném výrazu pak pro 32bitovou variantu stále slouží sada funkce *Korns* z tabulky 4.2. Nově je definována 8bitová varianta, která obsahuje následující operace:

- x (identita)
- bitová inverze
- max hodnota
- min hodnota
- x>>1
- x>>2
- x<<1
- x<<2
- x OR y
- x AND y
- x XOR y
- x+y
- x-y
- x*y

Tyto operace pracují s 8 bitovými hodnotami, kde jsou zakódovány hodnoty z intervalu $[-0,5; 0,5]$, ve kterém se také nachází hodnoty ze sad S1 a S2. Výsledky všech operací se také nachází v uvedeném formátu. Výsledky operací $+$, $-$ a $*$ jsou saturovány na hranice intervalu, tedy $0,3 + 0,4 = 0,5$.

Poslední změna v nastavení genetického programování s pamětí souvisí s použitými konstantami v hledaném výrazu. Jediné konstanty jsou max/min hodnoty, které jsou součástí sady operací. Tímto se odstraní problémy s ukládáním různých konstant a indikací, kde se v nalezeném výrazu mají konstanty používat.

Co se týče hodnot na vstupu kartézského genetického programování, pro datové sady S1 a S2 byla využita varianta neobsahující průměrné hodnoty jednotlivých konvolučních jader.

Dále je nutné nalezené výrazy G efektivně uložit do paměti, aby bylo možné je co nejrychleji načíst a začít generovat váhy pro neuronovou síť. Pro samotné uložení evolučně navrženého výrazu G do paměti pak může být použito různých variant:

v. 1. G je reprezentován podobně jako v chromozom v CGP. Pro každý uzel je zakódován identifikátor operace a dva identifikátory vstupů, přičemž zde se počet možností (a s tím spojená bitová velikost) mění na základě polohy v mřížce elementů.

v. 2. V tomto kódování výrazu G má variabilní vstupy pouze vstupní sloupec elementů, který má na svém vstupu některý ze vstupů CGP. Pro všechny elementy je pak uložen pouze identifikátor operace, přičemž pro správnou propagaci operací je nutné aby sada operací obsahovala identitu pro oba vstupy.

6.6.3 Optimalizace výrazu: výsledky experimentů

Grafy na obrázku 6.19 zobrazují přesnosti sítě pokud jsou váhy generovány pomocí 32bitových operací a váhy také mají 32bitů, dále pak přesnosti, kdy jsou tyto hodnoty po vygenerování následně převedeny na 8bitové a neuronová síť tak využívá 8bitové váhy. Poslední varianta je, že neuronová síť využívá 8bitové váhy, které jsou generovány pomocí výrazu nalezeného za použití 8 bitové sady operací.

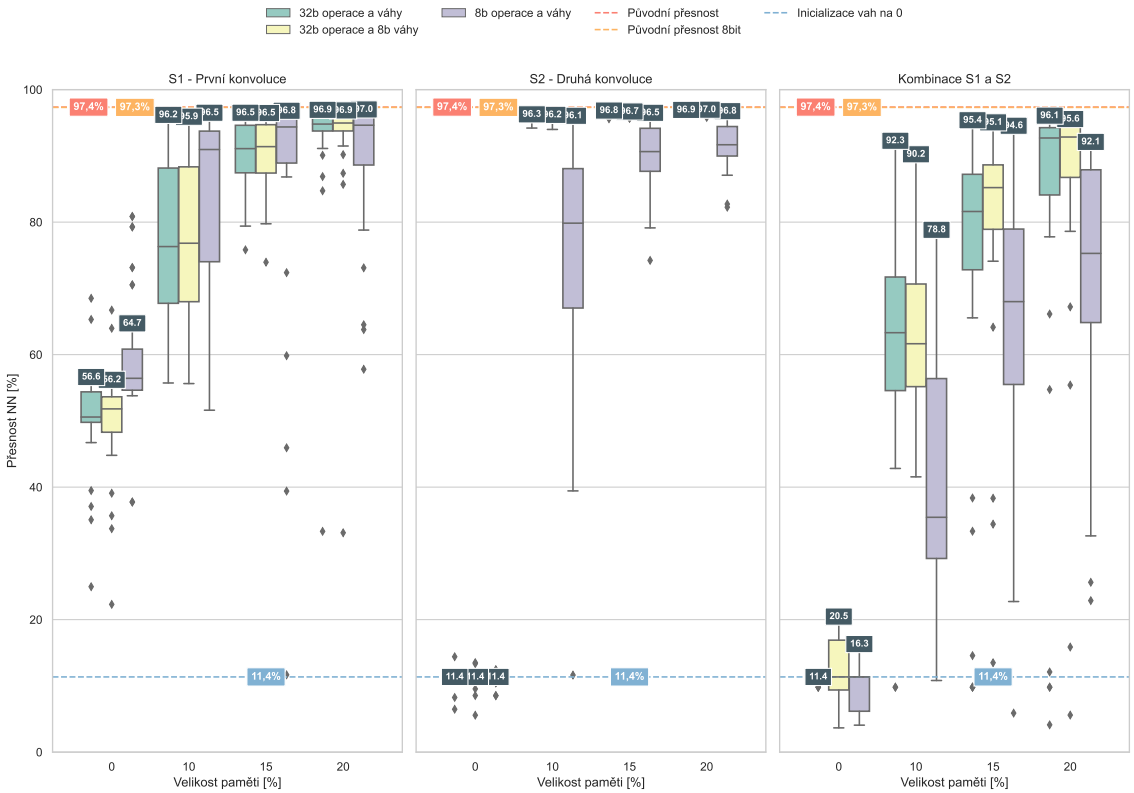
Lze vidět, že převodem vah vygenerovaných 32bitovými operacemi na 8bitové se přesnost sítě příliš nezmění. Při změně operací na 8bitové u sady S1 dochází k dosažení lepších výsledků při malých velikostí paměti. U sady S2 mají výsledky výrazně vetší rozptyl pro sadu 32bitových operací, ale nejlepší nalezená řešení jsou si podobná a liší se maximálně o 0,3 %.

$$G_{S1}(x_0, x_1, x_2, x_3) = ((max \gg 2)*(((((x1 \text{ XOR } x_2)-min) \text{ OR } ((x_1 \text{ XOR } x_2)-min)) \text{ OR } (x_1 \text{ XOR } x_2)) \text{ AND } ((\sim (x_2\&(x_1 \text{ XOR } x_2))) \text{ OR } (min \ll 2)))) \quad (6.4)$$

$$G_{S2}(x_0, x_1, x_2, x_3) = \sim (x_2 \text{ XOR } x_2) = -0.00390625 \quad (6.5)$$

Ve vztazích 6.4 a 6.5 můžeme vidět nejlepší nalezená řešení pro S1 a S2 s využitím 8bitové sady operací. Hodnoty *min* a *max* ve výrazu G_{S1} označují dříve popsané jediné možné konstanty při sadě 8bitových operací. Nalezený výraz G_{S2} je konstantní a jeho výstup je vždy -0,00390625.

V tabulce 6.5 jsou hodnoty kolik operací, respektive jak vysoký strom, mají evolučně nalezené výrazy G. Tyto hodnoty jsou potřeba pro výpočet možných velikostí pro uložení těchto výrazů. Při výpočtu paměťové velikosti je také potřeba znát počet bitů pro



Obrázek 6.19: Dosažené přesnosti neuronové sítě na testovacích datech datasetu MNIST [22] při změně vah a operací na 8bitové alternativy.

zakódování identifikátoru operace $ID_{operace}$, který je 4, protože sada obsahuje 14 operací. U varianty 2 kódovaní funkce je také potřeba do sady přidat identitu pro druhý vstup sady, to ale počet nutných bitů neovlivní.

Výsledné velikosti funkcí, pro různé varianty výpočtu, jsou následně vypsány v tabulce 6.6. Pro výpočet velikosti funkce podle varianty 1. byl použit počet operací z tabulky 6.5. Výsledná velikost je vypočtena jako součet velikostí všech elementů, kdy velikost elementu N je zjištěna jako $ID_{operace} + 2 \times bit(N - 1 + P)$, kde P je počet vstupů celého genetického programování s pamětí a funkce $bit()$ vrací na kolik bitů je možné zakódovat daný počet možností .

Druhá varianta používá pro výpočet výšku stromu, kdy je velikost vypočtena jako $N_{celkem} \times ID_{operace} + 2 \times N_{list} \times bit(P)$, kde $bit(P)$ je počet bitů pro zakódování identifikátoru pro libovolný vstup celého genetického programování s pamětí, N_{celkem} je počet uzlů celého stromu a N_{list} je počet listových uzlů stromu.

Můžeme vidět, že nejmenších velikostí pro sadu S2 se dosáhlo druhou variantou při ořezání velkých funkcí. Pokud bychom chtěli kódovat všechny řešení, tak je lepší použít variantu první. Sada S1 je ve všech testovaných případech lépe zakódována pomocí 1. varianty kódování.

Tabulka 6.5: Velikostmi nalezených výrazů při použití 8bitové sady operací, podle počtu operací a hloubky stromu funkce. Velikosti jsou uvedeny pro všechny, 75 % a 50 % nejmenších funkcí s tím, že pokud se tímto odstraní nejlepší řešení, je vynesena ztráta přesnosti generované sítě mezi celkovým nejlepším řešením a nejlepším řešením obsaženým v 75/50 % nejmenších funkcí.

Velikost funkcí	S1				S2			
	Operací	Ztráta	Výška stromu	Ztráta	Operací	Ztráta	Výška stromu	Ztráta
Všechny	24	–	9	–	24	–	8	–
75 % nejmenších	8	-0.25 %	5	-0.25 %	8	–	5	–
50 % nejmenších	–	–	4	-0.36 %	–	–	3	–

Tabulka 6.6: Počet bitů pro zakódování nalezených výrazů (z tabulkdy 6.5) určených pomocí v. 1. a v. 2.

Velikost funkcí	S1		S2	
	v.1.	v.2.	v.1.	v.2.
Všechny	298	3068	298	1532
75 % nejmenších	84	188	84	188
50 % nejmenších	–	92	–	44

6.6.4 Optimalizace paměti

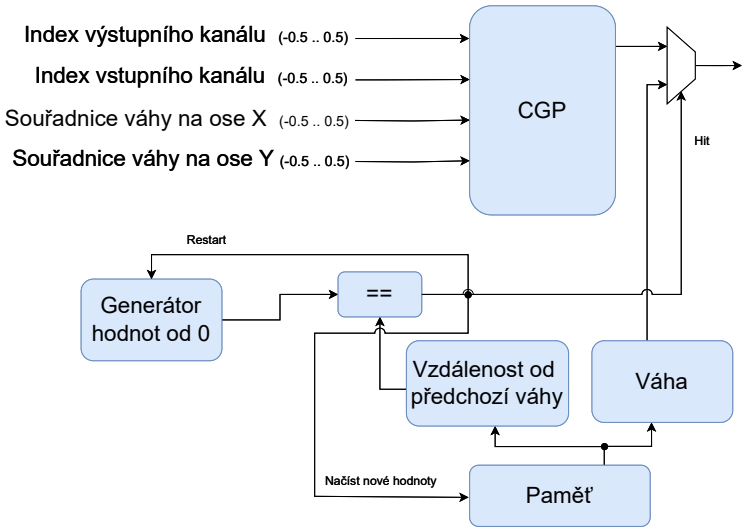
V této části práce je navržen nový způsob pro práci s pamětí, který si klade za cíl zmenšit její velikost, a to bez snížení kvality nalezeného řešení. Cílem je nahradit drahou asociativní paměť pomocí paměti s běžným adresováním. Je však nutné vyřešit problém tvorby adresy.

Pro účely zjednodušení paměti bylo navrženo nové zapojení paměti do systému, které je znázorněno na obrázku 6.20. Předpokládáme, že existuje neměnné pořadí vah, ve kterém musí být generovány. Dále je známo pořadí těch vah, které musí být uloženy v paměti. Tudíž můžeme pro každou váhu předem určit vzdálenost od předchozí váhy. Tato vzdálenost je následně uložena do paměti.

Na schématu je tedy vidět paměť, ze které se načítá dvojice vzdálenost od předchozí váhy a váha (DW, W). Hodnota DW je následně porovnávána s hodnotou generovanou generátorem. Pokud jsou hodnoty rozdílné, je hodnota generována výrazem G a hodnota vygenerovaná generátorem je navýšena o 1. Pokud jsou hodnoty stejné ($Hit = 1$), tak se provedou následující tři úkony:

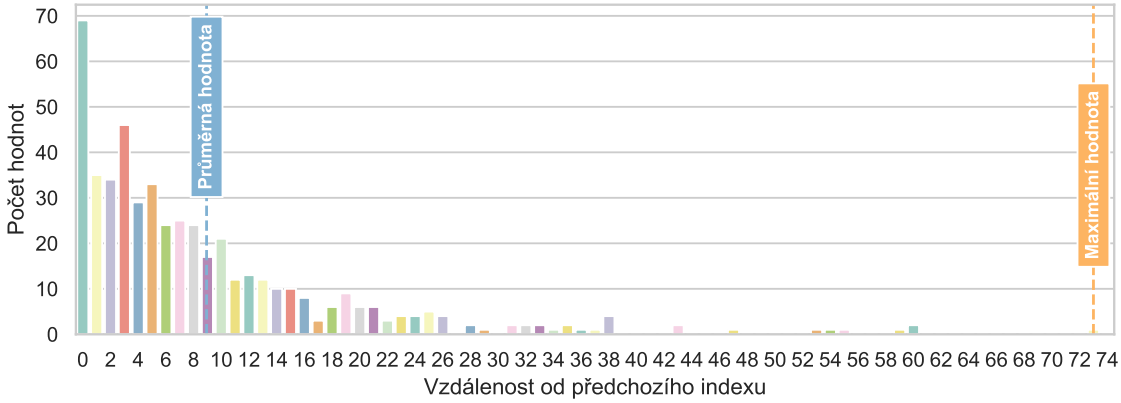
1. Váha (W) z paměti je přivedena na výstup systému.
2. Generátor je nastaven na generování hodnoty 0.
3. Z paměti je načtena další dvojice hodnot (DW, W) do registrů.

Takto se cyklus opakuje, dokud nejsou vygenerovány všechny váhy z datové sady. Aby bylo možné použít toto schéma pro generování, je nutné, aby byla každá hodnota v paměti použita pouze jednou. Tuto podmínku pro sady S1 a S2 splňují nalezená řešení, která využívají agregační funkci konkatenace. Řešení s jinými druhy testovaných agregačních funkcí by bylo možné upravit pomocí duplikování hodnot do paměti tolikrát, kolikrát se objeví na výstupu, to ale vede k výraznému zvětšování této paměti.



Obrázek 6.20: Schéma zobrazující možnost zapojení pro generování vah s efektivním využitím paměti.

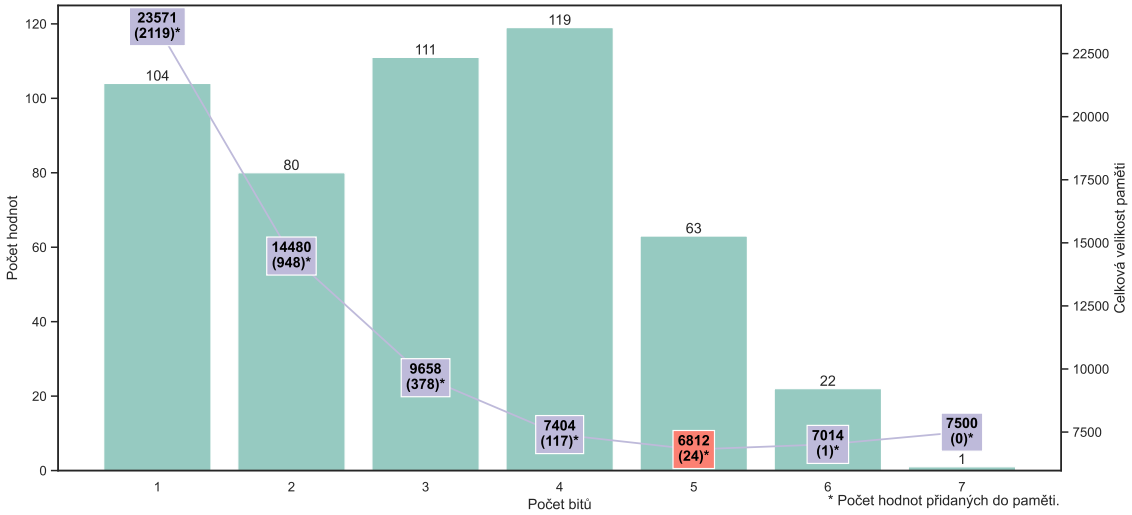
Protože výsledky při použití konkatenace dosahují nejlepších hodnot, byly pro experimenty použity tyto výsledky. To je také důvod, proč byla konkatenace zvolena jako agregační funkce u experimentů, které využívají operace pracující s 8bitovými váhami.



Obrázek 6.21: Histogram zastoupení vzdáleností od předchozí váhy (DW) pro nejlepší nalezené řešení pro sadu S2 při velikosti paměti 10 % a využití 8bitových operací.

Nabízí se otázka, kolik bitů je potřeba pro uložení hodnoty DW. Na obrázku 6.21 je vidět jakých hodnot a v jakém počtu hodnota DW dosahuje pro S2. Z grafu lze vyčíst, že v 69 případech byly do paměti uloženy hodnoty, které spolu sousedily a průměrná vzdálenost mezi dvěma hodnotami v paměti je 8,94. Maximální hodnota, která určuje, kolik bitů je nutných pro uložení těchto hodnot je 73. Je tedy nutných 7 bitů pro uložení hodnoty DW.

Protože je však zastoupení vysokých hodnot DW nízké, můžeme před hodnotu s vysokým DW vložit hodnotu pomocnou, díky čemuž snížíme hodnotu DW. Pokud bude počet hodnot, pro které musíme vložit pomocnou hodnotu dostatečně malý, aby celková velikost pomocných dvojic nepřesáhla velikost, která se ušetří snížením počtu bitů potřebných pro všechny hodnoty DW, tak se tato operace vyplatí. Jako praktickou ukázku takovéhoto vklá-



Obrázek 6.22: Počet hodnot DW, které se zakódují právě na daný počet bitů pro sadu S2. V obdelníku je výsledná velikost paměti, pokud je pomocí přidání hodnot snížen počet bitů potřebný pro zakódování DW.

dání pomocných hodnot berme v potaz hodnoty vynesené v obrázku 6.21, tedy pro paměť s 500 hodnotami (10 % sady S2). Velikost DW je tedy 7b kvůli hodnotě 73, která jediná potřebuje 7b. Ostatních 499 hodnot je možné zakódovat na 6b. Velikost W je 8b. Provedením výpočtu $500(7+8)$ zjistíme, že celková velikost je 7 500b. Vhodným vložení jedné pomocné hodnoty před hodnotu s DW = 73 zvýšíme počet hodnot v paměti o 1 a snížíme počet bitů nutných pro zakódování DW o 1. Nová velikosti paměti je $501(6+8) = 7014$ bitů. Přidáním jedné pomocné hodnoty jsme ušetřili 489 bitů.

Na obrázku 6.22 je na grafu ve sloupcích možné vidět, kolik hodnot DW potřebuje pro své zakódování právě konkretní počet bitů vynesený na ose x. Dále je zde vynesena křivka popisující celkovou velikost paměti, pokud je bitová velikost zmenšována přidáváním pomocných hodnot. Pro bitovou velikosti DW = 7 a 6 je zde vynesena dříve vypočítaná velikost 7 500 a 7 014 bitů. Nejmenší velikost paměti je dosažena při využití zakódování hodnot DW na 5bitů, kdy je do paměti potřeba přidat 24 pomocných hodnot. Z toho 22 pomocných hodnot je potřeba přidat pro hodnoty, které by bylo jinak nutné zakódovat na 6 bitů, a 2 pomocné hodnoty je potřeba přidat pro hodnotu 73, která se kóduje na 7 bitů.

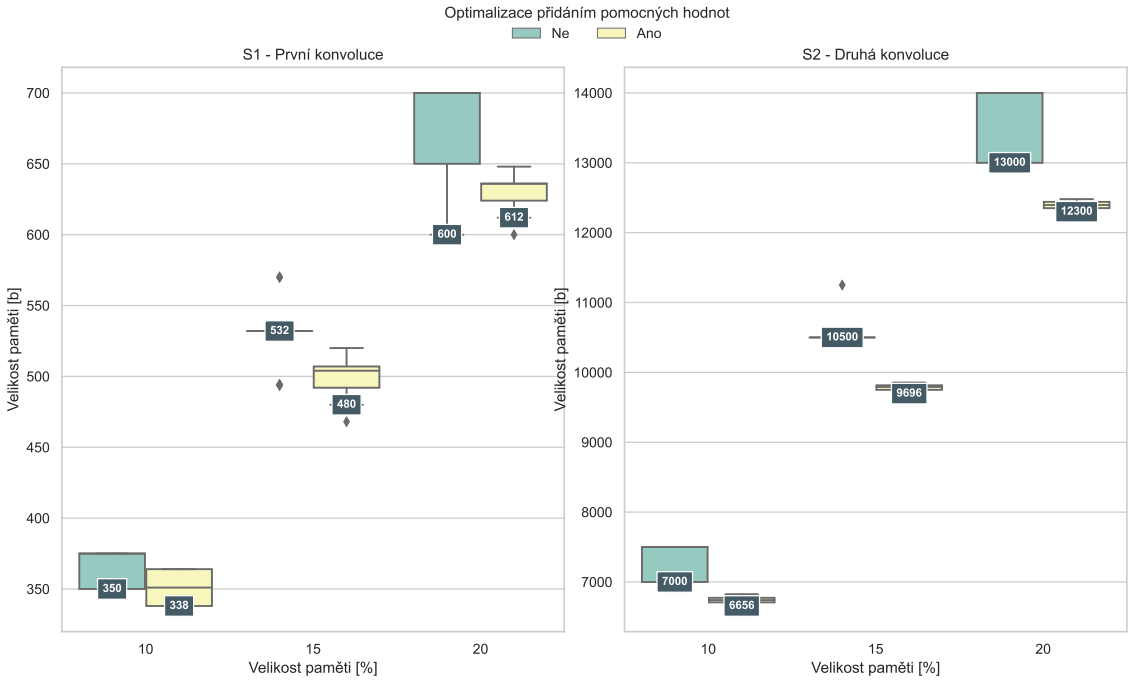
Tyto výsledky jsou samozřejmě pouze pro jedno konkrétní řešení nalezené pro sadu S2 při velikosti paměti 10 % a využití 8bitových operací. Na obrázku 6.23 je tedy vynesena velikost paměti pro všechna nalezená řešení využívající sadu 8bitových operací. Všechna tato řešení předpokládají uložení dvojic DW a W, ale je zde rozlišena varianta komprese řešení pomocí přidávání pomocných hodnot.

6.6.5 Alternativní kompresní metody

Dále bude vhodné otestovat, jakým způsobem dokáží provést kompresi datové sady různé jiné kompresní algoritmy. Pro porovnání budou využity tři univerzální algoritmy, pro bezztrátovou kompresi:

- Aritmetické kódování[1]

[1] Aritmetické kódování: https://github.com/ahmedfgad/ArithmeticEncodingPython



Obrázek 6.23: Výsledné velikosti pamětí, s/bez optimalizace přidáním pomocných hodnot pro všechny nalezená řešení využívající sadu s 8bitovými operacemi.

- Huffmanovo kódování[2]
- ZIP – komprese souboru obsahující váhy pomocí `zip -9`

Dále byla také spočítána teoreticky optimální velikost s využitím entropie. Komprese byla vždy počítána na 8bitových variantách vah neuronové sítě.

Výsledné počty bitů jsou uvedeny v tabulce 6.7. Je patrné, že většina algoritmů nedokáže zmenšit váhy první vrstvy na velikost menší než mají samotné váhy. U druhé vrstvy toto už neplatí a většina algoritmů umožní snížit počet bitů. Výjimkou je Huffmanovo kódování, které má, ve variantě bez tabulky, vždy velikost relativně těsně nad teoreticky optimální velikostí dané sady. V tabulce jsou pro každou sadu také uvedena dvě nejlepší nalezená řešení. U každého řešení je uvedena ztráta přesnosti klasifikace při jeho použití oproti původní síti a velikost při použití optimalizace zde popsané. Také je zde uvedena velikost bez optimalizace za použití asociativní paměti kdy tato velikost vždy přesahuje originální velikost dat.

6.6.6 Závěr – Odpovědi na výzkumné otázky

1. **Optimalizace funkce:**

   1.1. **Zhorší se výsledky na testovacích sadách při změně operací které CGP využívá?**

   Dochází ke zvětšení rozptylu přesnosti nalezených řešení, ale nejlepší nalezená řešení jsou srovnatelná s řešeními dosaženými s 32b operacemi. Je tedy větší

[2]Huffmanovo kódování: https://www.javatpoint.com/huffman-coding-using-python

Tabulka 6.7: Počet bitů získaných různými komprimačními metodami pro 8bitové váhy vrstev neuronové sítě. Uvedené jsou i dvě nejlepší řešení nalezené navrženou metodou a ztráta přesnosti sítě při jejich použití. Velikosti nalezených řešení jsou uvedeny při využití optimalizace velikosti nebo asociativní paměti (tedy bez optimalizace).

Metoda	Váhy z první vrstvy (S1)	Váhy z druhé vrstvy (S2)
Originální velikost (váhy 8b)	2 000	40 000
Aritmetické kódování	2 624	29 312
Aritmetické kódování s tabulkou pravděpodobností	40 192	47 488
Huffmanovo kódování	1 737	28 183
Huffmanovo kódování s tabulkou	3 867	29 355
ZIP	3 344	39 464
Teoreticky optimální	1 732	28 067
1. nalezené řešení ztráta:	0,8 %	1,2 %
optimalizované řešení	636	6 765
s asociativní pamětí	3 698	68 044
2. nalezené řešení ztráta:	1 %	2,6 %
optimalizované řešení	448	6 739
s asociativní pamětí	3 484	68 044

pravděpodobnost že nalezené řešení bude špatné, ale stále je možné nalést dobrá řešení.

1.2. **Jaká je velikost paměti potřebné pro uložení výrazu vytvořeného CGP?**
Pro S1 datovou sadu lze 50 % nejmenších funkcí (podle velikosti stromu) uložit na 76b a pro sadu S2 je potřeba 36b při stejném ořezaní funkcí. U sady S1 se tímto ořezáním odstraní 2 nejlepší řešení (třetí řešení na první ztrácí přesností -0,36 %).

2. **Optimalizace paměti:**

2.1. **Jaké metody jsou vhodné pro zmenšení velikosti paměti?**
Byla popsána nová metoda umožňující zredukovat paměťovou náročnost. Jako identifikátor, zda se má hodnota použít, je použita vzdálenost od poslední hodnoty. Tato metoda lze následně minimalizovat přidáváním pomocných hodnot tak, aby byl počet bitů nutný pro tyto hodnoty co nejmenší.

2.2. **Kolik bitů je potřeba pro uložení paměti, která je součástí řešení s dobrými výsledky?**
Paměť u řešení, které při 8bitových operacích dodává nejlepší výsledky při velikosti datové sady 10 %, má při použití pomocných hodnot velikost 338b pro S1 a pro sadu S2 6 721b.

3. **Kolik bitů je potřeba pro uložení celkového řešení, které dodává dobré výsledky?**

U sady S2 je možné zakódovat nejlepší řešení při velikosti paměti 10 % na 6 765bitů (funkce: 44b, paměť: 6 721). Komprese oproti 32bitovým vahám je 23,65×. Pro sadu S1 jsou nalezené funkce větší, proto bude dobré uvést velikosti pro více řešení:

- 1. nejlepší řešení: celková velikost 636bitů (funkce: 298b, paměť: 338b), komprese tedy je 12,58×,
- 2. nejlepší řešení: celková velikost 448bitů (funkce: 84b, paměť: 364b), komprese tedy je 17,86×,
  ztráta přesnosti oproti 1. je -0,25 %.

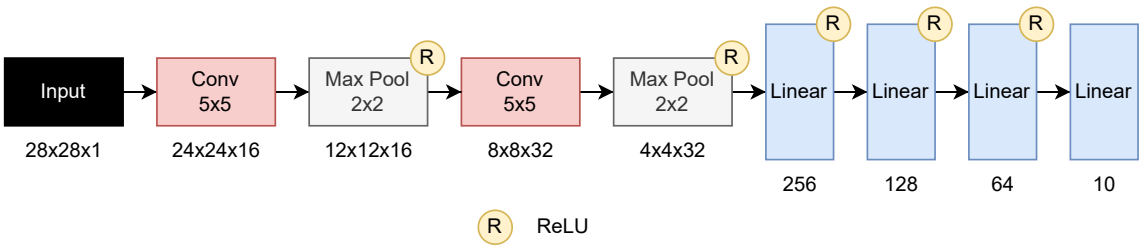
4. **Jak si navržená metoda vede oproti alternativním kompresním metodám?** Žádný z testovaných kompresních algoritmů nedosahuje lepšího nebo i podobného kompresního poměru, jako metoda genetického programování s pamětí. Dokonce bývají horší nebo na podobné velikosti jako váhy samotné bez použití jakéhokoliv kompresního algoritmu, což je způsobeno rozmanitostí hodnot jednotlivých vah. Je ovšem třeba brát v potaz, že testované algoritmy používaly bezztrátovou kompresi, zatímco genetické programování s pamětí je z tohoto pohledu komprese ztrátová.

6.7 Další sady z konvolučních vrstev

V předchozích experimentech bylo dosaženo dobrých výsledků při generování vah konvoluční neuronové sítě řešící klasifikaci na datové sadě MNIST. Tato klasifikační úloha se však dnes řadí mezi jednoduché. Je proto vhodné navržený způsob generování vah otestovat i na jiných neuronových sítích, které řeší jiné problémy. V následujících podsekcích jsou vždy navržené nové datové sady, na kterých je následně systém testován.

6.7.1 Fashion-MNIST

Jako první jiný úkol než klasifikace na datové sadě MNIST byla zvolena klasifikace na datové sadě Fashion-MNIST (FMNIST) [36]. Tato sada je, co se týče parametrů vstupů a velikostí trénovací a testovací sady, velice podobná datové sadě MNIST. Rozdílná je pouze v tom, že místo klasifikace ručně psaných číslic je cílem klasifikovat, do které třídy patří vyobrazené oblečení.



Obrázek 6.24: Schéma neuronové sítě pro klasifikaci na datová sadě FMNIST [36], z jejichž vah jsou vytvořeny další sady pro otestování systému.

Pro klasifikaci na této datové sadě byla vytvořena konvoluční neuronová síť, jejíž schéma je na obrázku 6.24. Její architektura je podobná architektuře sítě se kterou byly prováděny veškeré předchozí experimenty. Pro natrénování sítě byl použit optimalizátor Adam s rychlostí učení (lr) 0,01, který byl během běhu trénování (20 epoch) postupně měněn podle rovnice 6.6, kde lr_{new} je nový lr pro aktuální epochu, která je ve vzorci značena e, E_{max} je pak maximální počet epoch (20) a $lr_{original}$ je původní lr, tedy 0,01.

$$lr_{new} = \frac{lr_{original}}{2} * (\cos\left(\pi * (\frac{E_{max} - e + 1}{E_{max} + 1})\right) + 1) \tag{6.6}$$

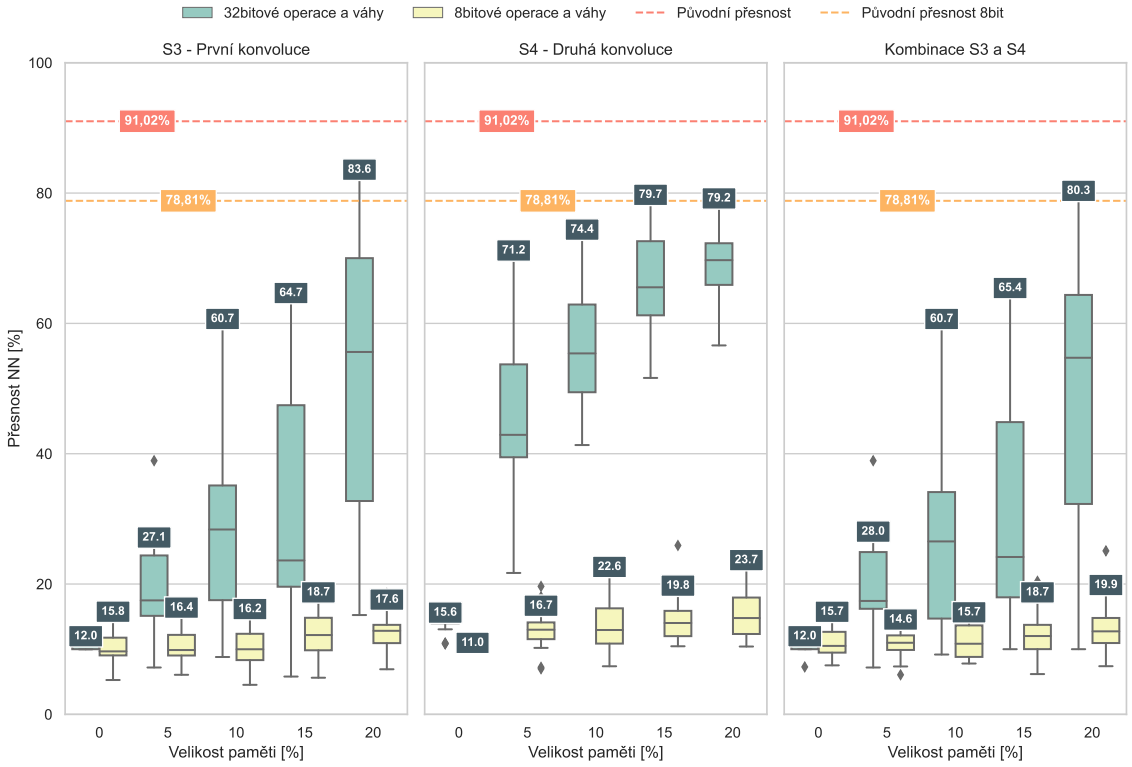
Stejně tak jako u sítě z předchozích experimentů, jsou i u této sítě vytvořeny datové sady ze všech konvolučních vrstev, které síť obsahuje. Vytvořené datové sady tedy jsou:

S3: sada obsahuje 400 vah z první konvoluční vrstvy.

S4: sada obsahuje 12 800 vah z druhé konvoluční vrstvy.

Tyto datové sady jsou tedy o něco větší, protože sady S1, S2 obsahují 250 a 5000 vah.

Co se týče nastavení genetického programování s pamětí, tak protože síť je relativně podobná jako dříve testovaná síť, bylo zvoleno stejné nastavení (viz tabulka 6.4). Experimenty jsou provedeny s oběma sadami operací, tedy jak se sadou obsahující operace pro 32bitové hodnoty, tak pro 8bitové.



Obrázek 6.25: Dosažené přesnosti kalsifikace neuronové sítě na testovacích datech datasetu FMNIST [36].

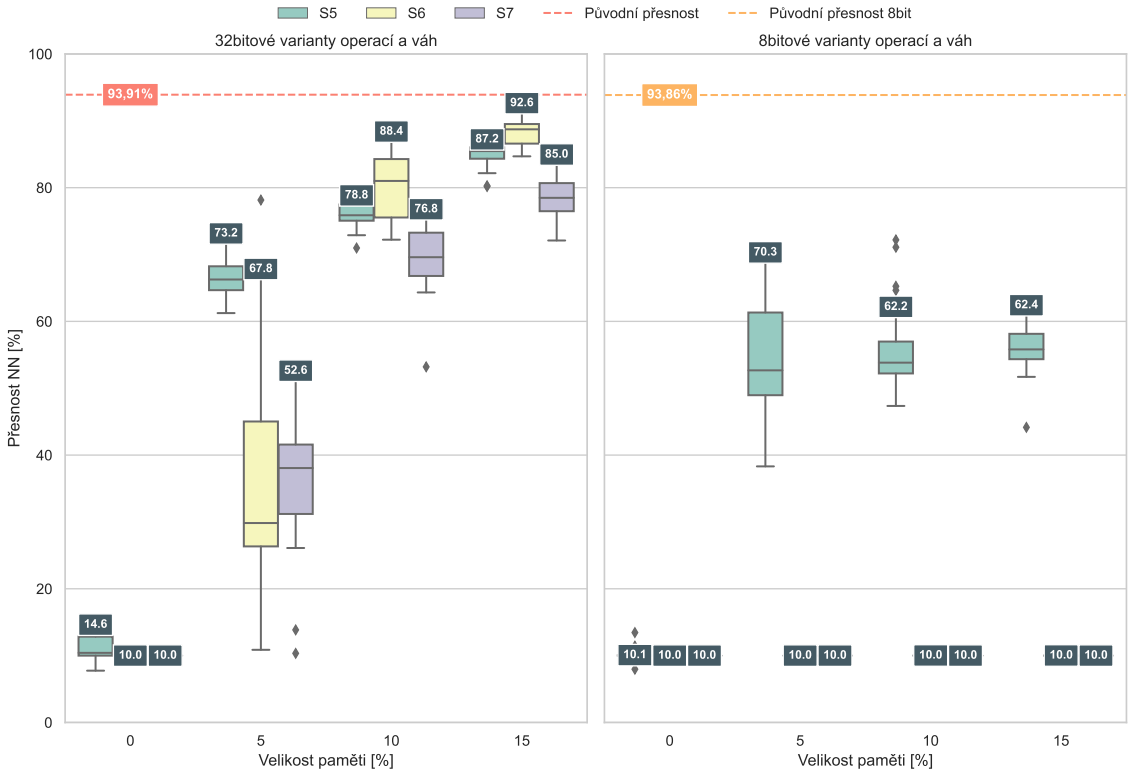
Přesnost sítě po natrénování je 91,02 %, ale přesnost sítě po převedení vah na velikost 8b je pouhých 78,81 %. Dochází tedy ke ztrátě 12,21 %, je tedy dost pravděpodobné, že síť je velice náchylná na přesnost svých vah.

Na grafu 6.25 můžeme vidět dvě skupiny výsledků rozdělených podle sady operací používaných při hledání funkce, kde jsou používány operace buď 32bitové nebo 8bitové. Bohužel se ukazuje, že sada 8bitových operací dosahuje slabých výsledků. Na druhou stranu výsledky při použití 32bitových operací a pamětí pro 20 % vah, respektive i 15 % pro sadu S4, dosahují u nejlepších řešení lepší přesnosti než zmenšení velikosti vah na velikost 8b. Stále

jsou ale výsledky dosti vzdálené od původní přesnosti natrénované sítě a nejmenší ztráta je 7,42 %. Všechny tyto horší výsledky můžou být způsobeny dříve zmíněnou pravděpodobnou náchylností na přesnost vah, která se může odvíjet od toho, že použitá síť je dosti podobná síti používané pro sady S1 a S2, ale klasifikace tříd na sadě FMNIST je náročnější než klasifikace tříd v sadě MNIST.

6.7.2 MobileNetV2

Jako další zdroj pro datové sady byla zvolena síť MobileNetV2 [31] v úpravě pro klasifikaci na datové sadě CIFAR-10 [18]. Protože tato síť je v základní konfiguraci určena na složitější datové sady než je CIFAR-10, byla použita její upravená varianta[3], natrénovaná pro klasifikaci na této datové sadě.



Obrázek 6.26: Dosažené přesnosti klasifikace MobileNetV2 [31] na testovacích datech datasetu CIFAR-10 [18]. Při generování vah pro různé vrstvy s využitím 32/8 bitových operací a vah.

Síť MobileNetV2 je optimalizovaná, aby její běh byl efektivní a proto její vrstvy obsahují relativně malý počet vah. Pro otestování navržené metody byly vytvořeny následující tři datové sady:

S5: sada obsahuje 864 vah z první konvoluční vrstvy.

S6: sada obsahuje 288 vah z depthwise konvoluční vrstvy s jádrem 3×3 z 1. IR bloku.

S7: sada obsahuje 864 vah z depthwise konvoluční vrstvy s jádrem 3×3 z 2. IR bloku.

[3]Úprava a natrénování MobileNetV2 pro CIFAR-10: https://github.com/huyvnphan/PyTorch_CIFAR10

Můžeme si všimnout, že datové sady obsahují relativně málo hodnot, což je způsobeno tím, že první konvoluce nepracuje s velkým množstvím kanálů a další dvě, ze kterých jsou sady S6 a S7, jsou depthwise konvoluční vrstvy a tedy nemíchají informace z různých kanálů.

Nastavení genetického programování je s ohledem na počet hodnot v sadách S5, S6 a S7 voleno stejné jako v předchozím experimentu a tedy stejné jako v tabulce 6.4 a v experimentech s neuronovými sítěmi řešícími klasifikaci u datových sad (F)MNIST.

U této sítě již nebyla testována kombinace řešení a grafy jsou tedy rozděleny bodle typu použitých operací a velikosti vah sítě. První (levý) graf na obrázku 6.26 obsahuje přesnosti neuronové sítě při velikosti vah 32bitů a použití operací pracujících s 32bitovými hodnotami. Můžeme vidět, že při použité paměti pro uložení 15 % hodnot se pro sadu S6 dosahuje výsledků které jsou pouze o 1,3 % horší než je přesnost sítě po natrénování. Na druhém grafu (pravý) můžeme vidět přesnosti při využití 8bitových vah a operací pracujících s 8bitovými hodnotami. Můžeme vidět, že výsledky jsou zde nedostateční a pouze u sady S5 dochází k nějakému zlepšení přidáním paměti.

6.7.3 ResNet-34

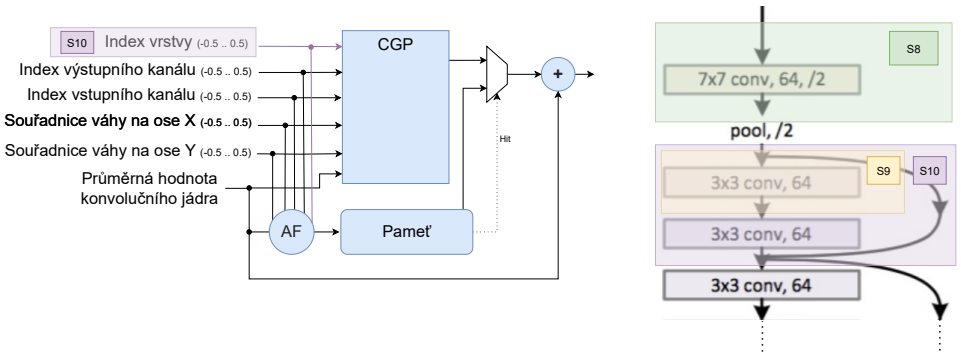
Jako nejsložitější konvoluční neuronová síť, na jejíž vahách bylo generování pomocí genetického programování s pamětí otestováno, byla zvolena síť ResNet-34 [9]. Tato síť byla natrénována na datové sadě ImageNet [10], ale testování probíhá na datové sadě Imagenette [5], což je klasifikace do 10 jednoduchých tříd z ImageNet.

Ze sítě ResNet-34 byly vytvořeny následující tři sady vah:

S8: sada obsahuje 9 408 vah z první konvoluční vrstvy.

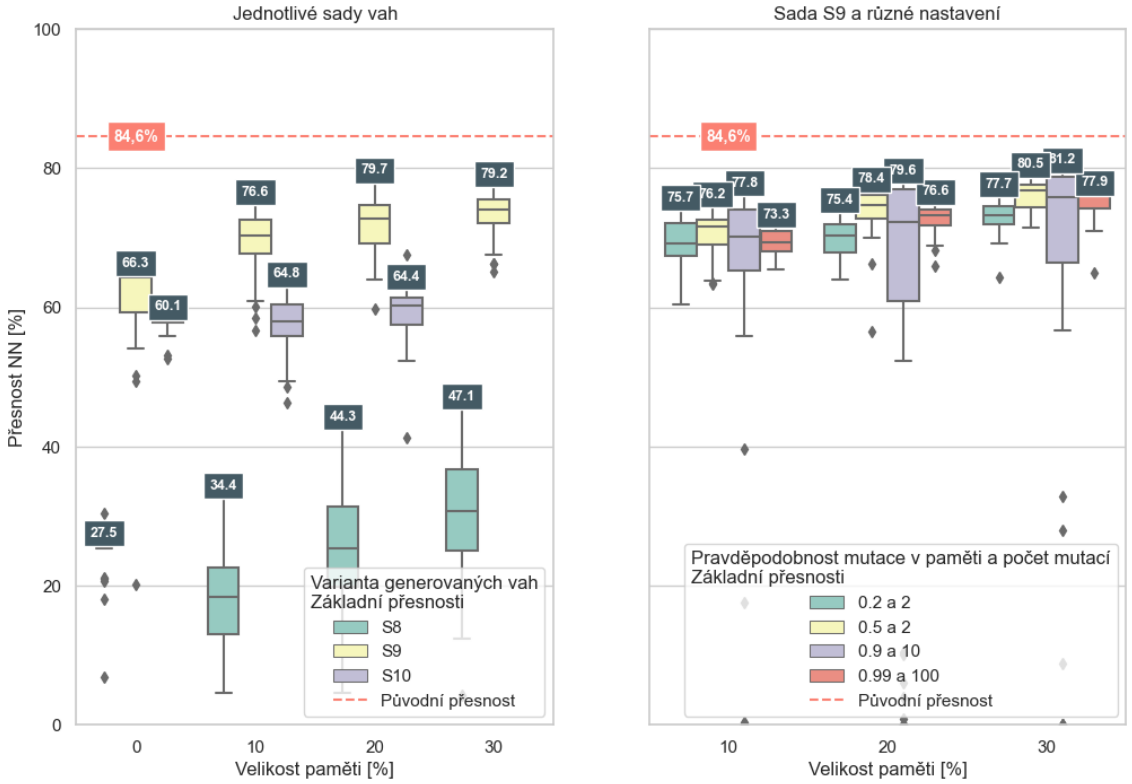
S9: sada obsahuje 36 864 vah z druhé konvoluční vrstvy.

S10: sada obsahuje 73 728 vah z druhé a třetí konvoluční vrstvy, jedná se tedy o celý 1. residual blok sítě.



Obrázek 6.27: Schéma zapojení systému pro sady S8 až S10 a sch0ma části ResNet-34 [9], ze které jsou tyto sady vytvořeny.

Sada S10 obsahuje váhy ze dvou konvolučních vrstev sítě, což nebylo doposud testováno a je třeba pro tento případ zvolit rozšíření systému. A to tak, aby měl informaci o tom, pro kterou vrstvu sítě jsou určené právě generované váhy. Proto byl pro tuto sadu přiveden na vstup genetického programování s pamětí i index vrstvy. Také u těchto sad využíváme i průměrů hodnot konvolučních jader na vstupu CGP a AF. Konkrétní zapojení systému a grafické znázornění zdroje konstant na části schématu ResNet-34 je na obrázku 6.27.



Obrázek 6.28: Dosažené přesnosti sítě ResNet-34 [9] na testovacích datech datasetu Imagenette [5] při generování vah pro různé vrstvy, respektive s různým nastavením.

Nastavení genetického programování je s ohledem na vyšší počet hodnot v datových sadách voleno pro začátek podobné jako v předchozích experimentech, tedy nastavení popsané v tabulce 6.4, s následnými úpravami popsanými dále. U této sítě již nedocházelo k testování experimentů se sadou operací pracující s 8bitovými hodnotami a tím spojené testování neuronové sítě při snížení přesnosti vah na 8bitové hodnoty. Další změnou je, že byla změněna agregační funkce na interval tak, jak je popsáno v experimentu 6.5, kdy byl zvolen interval o absolutní velikosti, který mohl být pro každou vstupní dimenzi jiný. Maximální vzdálenost od hodnoty v paměti byla stanovena na 0.000 05, přičemž se tato hodnota mohla mutací měnit v intervalu od 0.000 005 do 0.000 53, pravděpodobnost této mutace byla 5 %.

Na obrázku 6.28 jsou na grafu vlevo vyneseny výsledky pro jednotlivé datové sady s dříve popsaným nastavením. Lze vidět, že výsledky jsou nejlepší pro datovou sadu S9, kde se dosahuje nejmenší ztráty 4.9 % při nastavení velikosti paměti na 20 %. Výsledky pro sadu S8 se postupným přidáváním paměti zlepšují, ale všechny jsou nedostatečné s nejmenší dosaženou ztrátou 37,5 %. To je způsobeno pravděpodobně tím, že sada S8 obsahuje váhy první vrstvy sítě a nepřesnosti v této vrstvě dokáží výrazně ovlivnit výstup celé sítě. U sady S10 nedochází se zvětšující se pamětí k výraznému zlepšení. To může být způsobeno nedostatečným množstvím mutací v paměti vzhledem k velikosti datové sady a tedy i velké velikosti paměti, případně příliš velkou náročností problému, kdy jsou v této sadě obsaženy váhy ze dvou konvolučních vrstev sítě. Sada S10 v grafu také postrádá výsledky při nasta-

vení velikosti paměti na 30 %. To je kvůli tomu, že experimenty s touto sadu jsou časově náročné a velikost paměti 30 % již není příliš zajímavá.

Na pravém grafu na obrázku 6.28 jsou vyneseny výsledky pro sadu S9, tedy nejlepší z testovaných sad pro neuronovou síť ResNet-34, s rozdílným nastavením. Změna nastavení spočívá v záměně agregační funkce interval na funkci konkatenace. Další změna pak spočívá v testování různých pravděpodobností mutace v paměti a počtu provedených mutací. Motivací pro tyto změny je výrazně větší datová sada, než v kterémkoli z předchozích experimentů. Byly otestovány 4 varianty, jejichž konkrétní hodnoty jsou uvedené v legendě grafu, včetně varianty používané ve všech předchozích experimentech (0,2 a 2). Nejlepších výsledků bylo dosaženo při nastavení pravděpodobnosti mutace v paměti na 0,9 a počtu mutací 10, při velikosti paměti systému 30 %, kde byla ztráta oproti původní přesnosti sítě 3,4 %.

6.7.4 Závěr

V tomto experimentu bylo popsáno dalších 8 datových sad vytvořených z vah neuronových sítí. Dalších 6 takovýchto sad je popsáno v příloze D. Navržená metoda nedosahuje pro tyto sady v příloze zajímavých výsledků. Celkem tedy bylo otestováno 16 (i s S1 a S2) sad pro generování vah neuronových sítí.

Nejlepších výsledků bylo dosahováno u datových sad, které obsahují váhy vrstev, které jsou v dané síti na druhém místě. Pro síť MobileNetV2 bylo u takovéto sady dosaženo ztráty přesnosti 1,3 % při ukládání 15 % vah do paměti a pro síť ResNet-34 bylo při velikosti paměti 30 % dosaženo ztráty přesnosti sítě 3,4 %. Těchto výsledků bylo dosaženo při použití sady operací pracujících s 32bitovými hodnotami. Při využití operací pro práci s hodnotami na 8bitech nebylo dosahováno dobrých výsledků.

Tabulka 6.8: Zajímavé výsledky na sadách S1 až S10.

Id	Neuronová síť	Datová sada	Vrstva	Počet vah	Velikost vah	Ztráta přesnosti [%]	Velikost paměti [%]
S1	Jednoduchá CNN 1	MNIST	1.	250	8	0.8	10
S1	Jednoduchá CNN 1	MNIST	1.	250	8	0.3	20
S2	Jednoduchá CNN 1	MNIST	2.	5 000	8	1.2	10
S2	Jednoduchá CNN 1	MNIST	2.	5 000	32	0.5	20
S3	Jednoduchá CNN 2	FMNIST	1.	400	32	7.4	20
S4	Jednoduchá CNN 2	FMNIST	2.	12 800	32	11.3	15
S5	MobileNetV2	CIFAR-10	1.	864	32	6.7	15
S6	MobileNetV2	CIFAR-10	3.	288	32	1.3	15
S7	MobileNetV2	CIFAR-10	5.	864	32	8.9	15
S8	ResNet-34	Imagenette	1.	9 408	32	37.5	30
S9	ResNet-34	Imagenette	2.	36 864	32	3.4	30
S10	ResNet-34	Imagenette	2. & 3.	73 728	32	19.8	10

Tabulka 6.8 shrnuje nejzajímavější výsledky pro datové sady testované v textu práce. V tabulce 6.8 lze kromě jiného vidět kolik sada obsahuje vah a kolik procent z těchto vah je potřeba uložit do paměti, aby generované řešení bylo o uvedenou ztrátu přesnosti horší, než řešení s původními váhami.

6.8 Otestování na testovacích sadách

V předchozích kapitolách byla pro ověřování funkce systému genetického programování s pamětí použita pouze trénovací data získána z 10 běžných benchmarkových funkcí. V této části textu jsou popsány výsledky, kdy jsou pro validaci použity jiné hodnoty než pro trénování. Náhodně generované hodnoty zůstávají v trénovací i testovací sadě zachovány stejné. Zbylé hodnoty, tak aby trénovací i testovací sady byly stejně velké, jsou generovány z matematických rovnic a to tak, že pro všechny funkce jsou generovány náhodné hodnoty z intervalu, ze kterého byly generovány hodnoty trénovací.

Testovací datové sady tedy obsahují stejné náhodné hodnoty a jiné hodnoty z matematické funkce, než trénovací datové sady.

Co se týče datových sad vzniklých z vah neuronových sítí, tak zde nedává smysl vytvořit testovací sady k sadám trénovacím, protože vždy chceme aby systém generoval stejné hodnoty.

6.8.1 Výsledky experimentů

Jako varianta výsledků vhodná pro toto otestování byla z dříve provedených testů zvolena varianta, která jako agregační funkci využívá konkatenaci. Při velikosti paměti 0 využívá nastavení *Float konstanty* a při dalších velikostech paměti pak *Oba typy konstanty* (tyto nastavení jsou popsány v experimentu 6.2)

Po porovnání výsledků na trénovacích a testovacích sadách můžeme na obrázku 6.29 vidět rozdíly fitness na těchto sadách pro různé zastoupení náhodných hodnot a velikosti pamětí pro jednotlivé funkce. Kvůli tomu, že se při některých funkcích objevily rozdíly, jako výrazně odlehlé hodnoty, které svým vynesením do grafu výrazně zhoršovaly jeho čitelnost, byly všechny odlehlé hodnoty z grafů odstraněny.

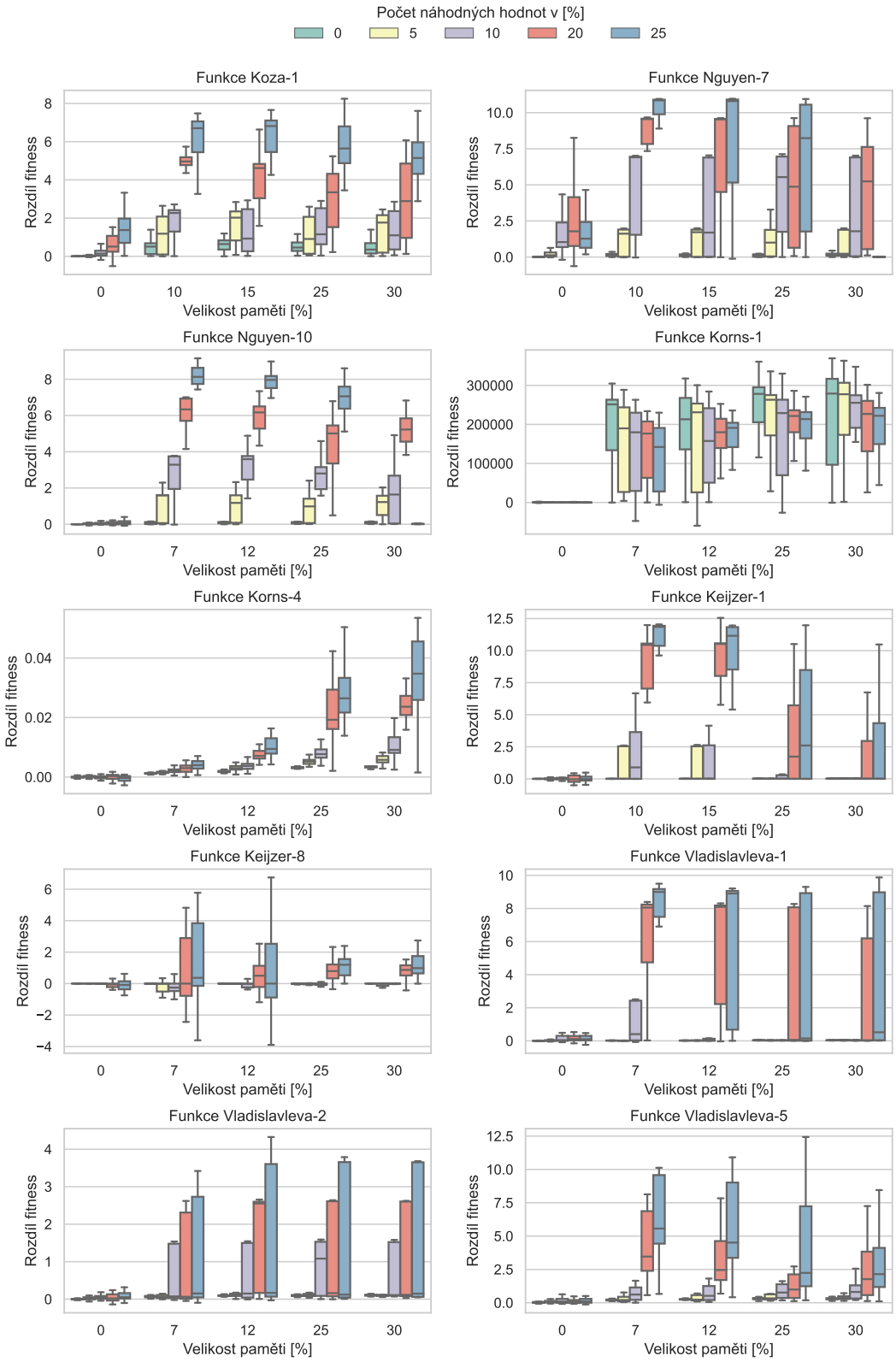
Je patrné, že řešení bez paměti fungují podobně na všech datových sadách bez ohledu na počet náhodných hodnot v těchto sadách, pouze u funkcí Koza-1 a Nguyen-7 jsou patrné rozdíly. Při použití nenulové velikosti paměti je následně vidět, že se chyby často zvětšují při datových sadách s větším zastoupením náhodných hodnot. To je způsobeno tím, že u těchto řešení jsou reprezentovány náhodné hodnoty pomocí vygenerovaného výrazu, zatímco hodnoty generováné z originální funkce jsou uloženy do paměti. Následně tedy při záměně hodnot generovaných z originální funkce jsou hodnoty převážně generovány výrazem, paměť není využívána. Protože výraz byl hledán tak, aby docházelo ke generování odlehlých hodnot, jsou výsledky na takovéto sadě horší.

Co se týče vlivu velikosti paměti na rozdíl výsledků, tak například u funkce *Korns-4* dochází ke zhoršení výsledků se zvětšující se pamětí. Na druhou stranu například u funkcí *Vladislavleva-5* a *Nguyen-10* je tento jev opačný, tedy se zvětšující se pamětí je rozdíl fitness menší.

V některých případech dochází i k dosažení lepší fitness na sadě testovací než na sadě trénovací, ale tyto výsledky (na grafu záporné hodnoty) jsou spíše ojedinělé.

6.8.2 Závěr

Kvalita řešení na datových sadách, které obsahují jiné hodnoty než hodnoty ze sady trénovací, se často zhoršuje podle toho, kolik náhodných hodnot sady obsahují. Toto je pravděpodobně způsobeno tím, že při velkém množství náhodných hodnot je v některých řešeních výhodnější proložit pomocí vygenerovaných výrazů některé hodnoty, které jsou zde



Obrázek 6.29: Rozdíl fitness na testovací a trénovací sadě.

náhodné, a naopak hodnoty, které náhodné nejsou, se ukládají do paměti. Tímto se při následné záměně nenáhodných hodnot dosáhne horších výsledků.

6.9 Další možnosti vylepšení metody

V této sekci jsou navrženy další možnosti vylepšení genetického programování s pamětí, které v této práci nebyly otestovány.

6.9.1 Jiná fitness funkce pro váhy neuronových sítí

V celé práci se používá fitness funkce popsána rovnicí 4.2, která je vhodná pro testovací sady tvořené z matematických funkcí. Nicméně u sad generovaných z vah neuronových sítí tato fitness funkce získává hodnotu reprezentující rozdíl generovaných vah a vah původních. Tento rozdíl ale nemusí být vhodný, protože cílem generovaných vah není co největší podobnost na váhy původní, ale co nejlepší přesnost neuronové sítě, pro kterou jsou váhy generovány.

Jako alternativní varianty funkce fitness pro řešení, která si kladou za cíl generovat váhy neuronových sítí, se nabízí například tyto dvě varianty:

- Pro výpočet fitness by mohla být použita přesnost sítě. Tedy při každém výpočtu fitness by byla sestavena odpovídající neuronová síť s vahami generovanými testovaným řešením. Takováto síť by byla následně testována na určité datové sadě a jako fitness by byla použita její přesnost na této sadě. Cílem by tedy bylo maximalizovat přesnost této sítě.

  Výhoda tohoto řešení je, že fitness reprezentuje lépe důležité vlastnosti výsledného řešení.

  Nevýhoda je velká výpočetní náročnost.

- Upravení aktuální fitness funkce 4.2 na variantu 6.7, která se liší pouze v přidání hodnot w_i, tedy hodnot, které by reprezentovaly důležitost i-té generované váhy neuronové sítě na její přesnost. Tyto hodnoty by bylo nutné zjistit na základě experimentů s neuronovou sítí, kdy by byla vždy jedna váha nahrazena za jinou, jí nepodobnou hodnotu, a otestována na testovací sadě. Váhy by se následně určily na základě poklesů přesnosti sítě během těchto experimentů.

$$\text{Fitness} = \sum_{i=1}^{M} w_i (y'(\vec{\boldsymbol{x_i}}) - o_i)^2 \tag{6.7}$$

  Výhoda tohoto řešení je podobná výpočetní náročnost jako u prováděných experimentů během výpočtu fitness.

  Nevýhoda je nutnost zjistit, jak je která váha důležitá pro přesnost sítě. Tato varianta také nehodnotí dobře varianty, které by vytvořily neuronovou síť s dobrou přesností, ale liší se od původních vah sítě.

6.9.2 Jiná varianta genetického programování

Pro experimenty s navrženým genetickým programováním s pamětí bylo v této práci využito kartézské genetické programování. To by však mohlo být nahrazeno genetickým programováním, které pro reprezentaci funkce využívá stromovou reprezentaci.

Tato změna by také mohla přinést některé zajímavé aspekty do hledání vhodných bodů do paměti. Například přidáním křížení dvou řešení včetně paměti by mohlo vést k rychlejšímu nalezení zajímavého obsahu paměti v prvních generacích běhu. Také by mohly být testovány různé varianty křížení pamětí.

Obecně by tato změna mohla vést k jiným výsledkům a i když nelze předpokládat obecné zlepšení, tak některé problémy by mohly být řešeny lépe.

6.9.3 Otestování jiných aplikací genetického programování s pamětí

V této práci byl systém rozšíření genetického programování o paměť otestován na uměle vytvořených testovacích úlohách a praktických úlohách testujících generování vah pro konvoluční vrstvy neuronových sítí. Aplikace tohoto navrženého systému může být ale větší.

I přesto, že přidání paměti je hlavně výhodné při aplikacích, kdy je systém stabilně používán na stejných datech, tak některé varianty, například ty využívající interval, které jsou popsané v sekci 6.5, může být zajímavé využít v klasických případech pro symbolickou regresi. Tedy například na datových sadách z reálného světa zmiňovaných na začátku práce.

Dále by se navržený systém dal testovat například pro hledání obrazového filtru, a jistě by se daly nalézt další typické aplikace genetického programování, ve kterých by přidání paměti mohlo přinést zajímavé výsledky a řešení problémů.

Kapitola 7

Závěr

Byla nastudována problematika řešení symbolické regrese pomocí genetického programování se zaměřením na kartézské genetické programování. Dále byly prostudovány typické testovací přístupy pro metody řešící symbolickou regresi. Aby bylo možné využít symbolickou regresi v úloze minimalizace místa nutného pro uložení vah konvolučních vrstev neuronových sítí, byla nastudována i problematika těchto sítí a jejich optimalizace.

Dále byl popsán koncept a návrh základního principu nového systému, který rozšiřuje genetické programování o paměť. Byla vytvořena nová sada testovacích funkcí tak, aby byla vhodná pro testování genetického programování s pamětí. Systém byl implementován a následně byly provedeny experimenty.

Byla provedena řada experimentů, které se zaměřovaly na vliv rozšíření genetického programování o paměť, kdy se některé experimenty zabývaly různými možnostmi tvorby adres pro hodnoty uložené v paměti. Jiné experimenty také testovaly možnost využití hodnot uložených v paměti pro generování konstant do hledaných funkcí, případně přivedení hodnot jako možný vstup systému při využívání zpětných vazeb. Z provedených experimentů bylo pro datové sady vytvořené z matematických funkcí zjištěno, že paměť pomáhá, pokud sada obsahuje náhodné body.

Části experimentů a některé celé experimenty byly také věnovány možnosti využití genetického programování s pamětí pro generování vah neuronových sítí. U neuronové sítě řešící klasifikaci na datové sadě MNIST byla nalezena varianta schopná generovat váhy pro jednu konvoluční vrstvu sítě, jejíž paměťová náročnost je 23,7× menší, než při ukládání nijak neupravovaných vah vrstvy, přičemž je ztráta přesnosti sítě 1,3 % oproti síti s původními váhami. U tohoto řešení je 10 % vah sítě uloženo v paměti v 8bitové podobě a zbylých 90 % vah je generováno funkcí, která využívá pouze operace pracující s 8bitovými hodnotami.

Dále bylo testováno použití pro generování vah pro sítě řešící komplikovanější problémy, než je klasifikace na datové sadě MNIST. Mezi nalezené zajímavé výsledky patří řešení generující váhy pro síť MobileNetV2, řešící klasifikaci na datové sadě CIFAR–10, kde při uložení 15 % vah do paměti dojde ke snížení přesnosti o 1,3 % oproti původnímu řešení. Mezi další nalezená zajímavá řešení patří řešení pro síť ResNet-34 využívající uložení 30 % vah do paměti, kde je pokles přesnosti oproti původní síti 3,4 %, kdy přesnost byla měřena pro klasifikaci na datové sadě Imagenette.

V poslední části práce byly diskutovány možné další rozšíření a oblasti použití genetického programování s pamětí, které nebyly otestovány v této práci, a jejichž otestování by mohlo přinést zajímavé výsledky.

Literatura

[1] Aggarwal, C. C. *Neural Networks and Deep Learning - A Textbook*. Springer, 2018. ISBN 978-3-319-94462-3. Dostupné z: https://doi.org/10.1007/978-3-319-94463-0.

[2] Blalock, D. W.; Ortiz, J. J. G.; Frankle, J. a Guttag, J. V. What is the State of Neural Network Pruning? *CoRR*, 2020, abs/2003.03033. Dostupné z: https://arxiv.org/abs/2003.03033.

[3] Clanuwat, T.; Bober-Irizar, M.; Kitamoto, A.; Lamb, A.; Yamamoto, K. et al. Deep Learning for Classical Japanese Literature. *CoRR*, 2018, abs/1812.01718. Dostupné z: http://arxiv.org/abs/1812.01718.

[4] Cowlishaw, M., ed. *IEEE Standard for Floating-Point Arithmetic*. Standard IEEE Std 754-2008. New York, NY, USA: IEEE Computer Society, srpen 2008. Dostupné z: https://web.archive.org/web/20160806053349/http://www.csee.umbc.edu/~tsimo1/CMSC455/IEEE-754-2008.pdf.

[5] Deng, J.; Dong, W.; Socher, R.; Li, L.; Li, K. et al. ImageNet: A large-scale hierarchical image database. In: *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 2009, s. 248–255. Dostupné z: https://doi.org/10.1109/CVPR.2009.5206848.

[6] Dupuis, E.; Novo, D.; O'Connor, I. a Bosio, A. A Heuristic Exploration of Retraining-free Weight-Sharing for CNN Compression. In: *27th Asia and South Pacific Design Automation Conference, ASP-DAC 2022, Taipei, Taiwan, January 17-20, 2022*. IEEE, 2022, s. 134–139. Dostupné z: https://doi.org/10.1109/ASP-DAC52403.2022.9712487.

[7] Guo, C.; Tang, J.; Hu, W.; Leng, J.; Zhang, C. et al. OliVe: Accelerating Large Language Models via Hardware-friendly Outlier-Victim Pair Quantization. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ACM, červen 2023. ISCA '23. Dostupné z: http://dx.doi.org/10.1145/3579371.3589038.

[8] Harris, C. R.; Millman, K. J.; Walt, S. J. van der; Gommers, R.; Virtanen, P. et al. Array programming with NumPy. *Nature*. Springer Science and Business Media LLC, září 2020, sv. 585, č. 7825, s. 357–362. Dostupné z: https://doi.org/10.1038/s41586-020-2649-2.

[9] He, K.; Zhang, X.; Ren, S. a Sun, J. *Deep Residual Learning for Image Recognition*. 2015.

[10] Howard, J. *Imagewang*. Dostupné z: https://github.com/fastai/imagenette/.

[11] ISO Central Secretary. *Statistical interpretation of data — Part 4: Detection and treatment of outliers*. Standard ISO 16269-4:2010. Geneva, CH: International Organization for Standardization, 2010. Dostupné z: https://www.iso.org/standard/44396.html.

[12] Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M. et al. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR*, 2017, abs/1712.05877. Dostupné z: http://arxiv.org/abs/1712.05877.

[13] Juza, T. a Sekanina, L. GPAM: Genetic Programming with Associative Memory. In: Pappa, G.; Giacobini, M. a Vasicek, Z., ed. *Genetic Programming*. Cham: Springer Nature Switzerland, 2023, s. 68–83. ISBN 978-3-031-29573-7.

[14] Keijzer, M. Improving Symbolic Regression with Interval Arithmetic and Linear Scaling. In: Ryan, C.; Soule, T.; Keijzer, M.; Tsang, E. P. K.; Poli, R. et al., ed. *Genetic Programming, 6th European Conference, EuroGP 2003, Essex, UK, April 14-16, 2003. Proceedings*. Springer, 2003, sv. 2610, s. 70–82. Lecture Notes in Computer Science. Dostupné z: https://doi.org/10.1007/3-540-36599-0_7.

[15] Kennedy, J. a Eberhart, R. C. Particle swarm optimization. In: *Proceedings of the IEEE International Conference on Neural Networks*. 1995, s. 1942–1948.

[16] Korns, M. Accuracy in Symbolic Regression. In: *Genetic Programming Theory and Practice IX. Springer*. New York. Kaufmann Publishers, Listopad 2011, s. 129–151. ISBN 978-1-4614-1769-9.

[17] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge: Bradford Book, London : MIT Press, 1992.

[18] Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images, 2009, s. 32–33. Dostupné z: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[19] Krizhevsky, A.; Sutskever, I. a Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F.; Burges, C.; Bottou, L. a Weinberger, K., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012, sv. 25. Dostupné z: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[20] LaValle, S. M.; Branicky, M. S. a Lindemann, S. R. On the relationship between classical grid search and probabilistic roadmaps. *The International Journal of Robotics Research*. SAGE Publications, 2004, sv. 23, 7-8, s. 673–692.

[21] LeCun, Y.; Bottou, L.; Bengio, Y. a Haffner, P. Gradient-Based Learning Applied to Document Recognition. In: *Proceedings of the IEEE*. 1998, sv. 86, č. 11, s. 2278–2324. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.7665.

[22] LeCun, Y. a Cortes, C. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010. Dostupné z: http://yann.lecun.com/exdb/mnist/.

[23] McConaghy, T. FFX: Fast, Scalable, Deterministic Symbolic Regression Technology. *Genetic Programming Theory and Practice IX*, Leden 2011, s. 235–260.

[24] Mcculloch, W. a Pitts, W. A Logical Calculus of Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 1943, sv. 5, s. 127–147.

[25] McDermott, J.; White, D. R.; Luke, S.; Manzoni, L.; Castelli, M. et al. Genetic programming needs better benchmarks. In: Soule, T. a Moore, J. H., ed. *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012*. ACM, 2012, s. 791–798. Dostupné z: https://doi.org/10.1145/2330163.2330273.

[26] Miller, J. F. An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, s. 1135–1142. GECCO'99. ISBN 1558606114.

[27] Miller, J. F. *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. 17–34 s. Dostupné z: https://doi.org/10.1007/978-3-642-17310-3_2.

[28] Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J. et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *CoRR*, 2019, abs/1912.01703. Dostupné z: http://arxiv.org/abs/1912.01703.

[29] Rojas, R. *Neural Networks - A Systematic Introduction*. Berlin: Springer-Verlag, 1996. Dostupné z: http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/pmwiki/pmwiki.php?n=Books.NeuralNetworksBook.

[30] Ross, A. a Willson, V. L. *Basic and Advanced Statistical Tests: Writing Results Sections and Creating Tables and Figures*. Rotterdam: SensePublishers, 2017. ISBN 978-94-6351-086-8. Dostupné z: https://link.springer.com/book/10.1007/978-94-6351-086-8.

[31] Sandler, M.; Howard, A. G.; Zhu, M.; Zhmoginov, A. a Chen, L. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 2018, s. 4510–4520. Dostupné z: http://openaccess.thecvf.com/content_cvpr_2018/html/Sandler_MobileNetV2_Inverted_Residuals_CVPR_2018_paper.html.

[32] Schmidt, M. a Jordan, J. hal-cgp: Cartesian genetic programming in pure Python. Zenodo, červen 2020. Dostupné z: https://doi.org/10.5281/zenodo.3889163.

[33] Sze, V.; Chen, Y.; Yang, T. a Emer, J. S. *Efficient Processing of Deep Neural Networks*. Morgan & Claypool Publishers, 2020. Synthesis Lectures on Computer Architecture.

[34] Tsanas, A. a Xifara, A. *Energy efficiency* UCI Machine Learning Repository. 2012. DOI: https://doi.org/10.24432/C51307.

[35] Vladislavleva, E. K.; Smits, G. a Hertog, D. den. Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming. *Evolutionary Computation, IEEE Transactions on*, Květen 2009, sv. 13, s. 333 – 349.

[36] Xiao, H.; Rasul, K. a Vollgraf, R. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR*, 2017, abs/1708.07747. Dostupné z: http://arxiv.org/abs/1708.07747.

[37] Zhao, L. *Water Quality Prediction* UCI Machine Learning Repository. 2022. DOI: https://doi.org/10.1145/3339823.

Příloha A

Obsah přiloženého paměťového média

V této příloze je naznačena struktura základních souborů a složek na přiloženém paměťovém médiu.

Paměťové médium

- `Text/` – Text diplomové práce.
  - `src/` – Složka se zdrojovými soubory textu práce v LaTeX.
  - `DP_xjuzat00.pdf` – Vytvořené PDF ze zdrojových souborů.
- `GPAM/` – Složka s implementací.
  - `configs/` – Složka s konfiguračními soubory pro experimenty.
  - `data/` – Složka s daty.
    - `datasets/` – Složka pro datové sady.
    - `graphs/` – Složka pro ukládání vytvořených grafů.
    - `models/` – Složka pro modely natrénovaných sítí.
    - `results/` – Složka s výsledky genetického programování.
  - `doc/` – Vygenerovaná dokumentace kódu.
  - `src/` – Zdrojové kódy implementace genetického programování s pamětí.
  - `test/` – Implementací testů pro soubory v `src/`.
  - `utils/` – Pomocné implementace. (např. trénování neuronových sítí)
  - `main.py` – Skript pro spuštění genetického programování s pamětí.
  - `README.md` – Podrobnější popis souborů, implementace a její spustitelnosti.
  - `requirements.txt` – Seznam použitých modulů a jejich verzích.

Příloha B

Konfigurační soubor

V této příloze jsou popsány jednotlivé parametry, které se vyskytují v konfiguračním souboru a byly používány pro experimenty popsané v kapitole 6. Konfigurační soubor však může, v některých případech dokonce musí, obsahovat další parametry, které jsou popsány v README na paměťovém médiu. Tyto parametry slouží k nastavení vlastností, jejichž vliv byl testován v experimentech, které nebyly uvedené v kapitole 6, protože jejich vliv, případně důvod použití, nebyl dostatečně zajímavý.

Konfigurační soubor tedy obsahuje následující parametry:

P1. `generation` – Maximální počet generací, které se provedou.

P2. `end_evolution` – Pokud fitness dosáhne této nebo lepší hodnoty, výpočet se ukončí.

P3. `probability_operation_mutation`[1] – Pravděpodobnost mutace operace, komplementární pravděpodobnost je symetricky rozdělena na mutace vstupů.

P4. `probability_of_constant`[1] – Pravděpodobnost že při mutaci vstupu elementu vznikne konstanta (desetiné číslo) vygenerována z intervalu zadaným parametrem P5.

P5. `constant_interval` – Interval (list) s dvěma hodnotami, kdy první je spodní a druhá je horní hranice intervalu ze kterého se generují konstanty. Nejčastěji používané nastavení [-1 000, 1 000].

P6. `cgp` – Následují hodnoty vztahující se ke kartézskému genetickému programování.

- P6.1. `population`[1] – Počet jedinců v populaci.
- P6.2. `operations` – Seznam možných operací pro elementy kartézskému genetickému programování.
- P6.3. `n_word_for_8bit` – Počet bitů pro reprezentaci. U 8bitové reprezentace 8.
- P6.4. `n_frac_for_8bit` – Počet bitů pro reprezentaci desetinné části. Nesmí být větší než P6.3.
- P6.5. `signed_for_8bit` – T/F jestli má být reprezentace se znaménkem.
- P6.6. `x`[1] – Počet sloupců elementů.
- P6.7. `y`[1] – Počet řádků elementů.
- P6.8. `L-back`[1] – L-back.

[1] Tento parametr může být hodnota nebo seznam hodnot nad kterými je následně spuštěn grid-search.

P6.9. `num_mutation`[1] – Počet mutací, které se mají provést.

P7. `memory_size`[1] – Počet hodnot, které se mohou uložit do paměti.

P8. `probability_memory_mutation`[1] – Pravděpodobnost provedení mutace v paměti. Komplementární pravděpodobnost je pravděpodobnost mutace v genetickém programování hledající funkci.

P9. `probability_of_constant_from_memory`[1] – Pravděpodobnost, že při mutaci vstupu elementu vznikne konstanta vybráním náhodné hodnoty z paměti.

P10. `maximal_probability_of_constant`[2] – Maximální hodnota při součtu hodnot parametrů P4 a P9.

P11. `minimal_probability_of_constant`[2] – Minimální hodnota při součtu hodnot parametrů P4 a P9. Mimo hodnotu součtu 0 která je vždy povolena.

P12. `minimal_probability_of_constant_from_memory`[2] – Minimální hodnota parametru P9 při velikosti paměti rozdílné od 0.

P13. `memory_type` – Typ agregační funkce použité u paměti. Možné varianty:

- `concat`
- `xor`
- `one_interval`
- `one_interval_relative`

P14. `one_interval` – Následují hodnoty nastavení jsou použity při hodnotách `one_interval` nebo `one_interval_relative` parametru P13.

P14.1. `same_for_all_dimensions`[1] – T/F, zda má být interval stejný pro všechny dimenze.

P14.2. `static`[1] – T/F, zda má být velikost intervalu hledána.

P14.3. `starting_interval`[1] – Počáteční maximální vzdálenost.

P14.4. `probability_of_interval_mutation`[1] – Pravděpodobnost změny velikosti intervalu při mutaci paměti a parametru P14.2 = T.

P14.5. `interval_of_interval_size` – Interval (list) s dvěma hodnotami, kdy první je spodní a druhá je horní hranice intervalu ze kterého se při P14.2 = T generuje při mutaci velikosti intervalu nová velikost.

P15. `input_formula` – Kód, který při spuštění vytvoří do proměnné `f` seznam jednotlivých vstupů. Kde jeden vstup je seznam s jednotlivými vstupními dimenzemi.

P16. `static_formula` – T/F informace, zda je vstup vždy stejný.

P17. `output_f` – Kód který při spuštění vytvoří do proměnné `f` seznam jednotlivých výstupů. Kde jeden výstup je seznam s jednotlivými výstupními dimenzemi. Přičemž má k dispozici seznam vytvořený parametrem P15 v proměnné X.

[1] Tento parametr může být hodnota nebo seznam hodnot nad kterými je následně spuštěn grid-search.
[2] Hodnota pro omezení velikosti grid-search.

P18. `f_name` – Jméno pro vhodné uložení výsledků a jejich identifikaci.

P19. `number_of_random_values`[1] – Počet hodnot, které se z parametru P20 použijí.

P20. `random_values` – Náhodné hodnoty, jejichž formát záleží na parametru P16 pro nastavení:

- T – `X: Y`, kde X je hodnota, kolikátou hodnotu nahradí hodnota Y, a Y je seznam hodnot se všemi výstupními dimenzemi, nebo
- F – `[X, Y]`, kde X/Y je seznam hodnot pro všechny vstupní/výstupní dimenze.

[1] Tento parametr může být hodnota nebo seznam hodnot nad kterými je následně spuštěn grid-search.

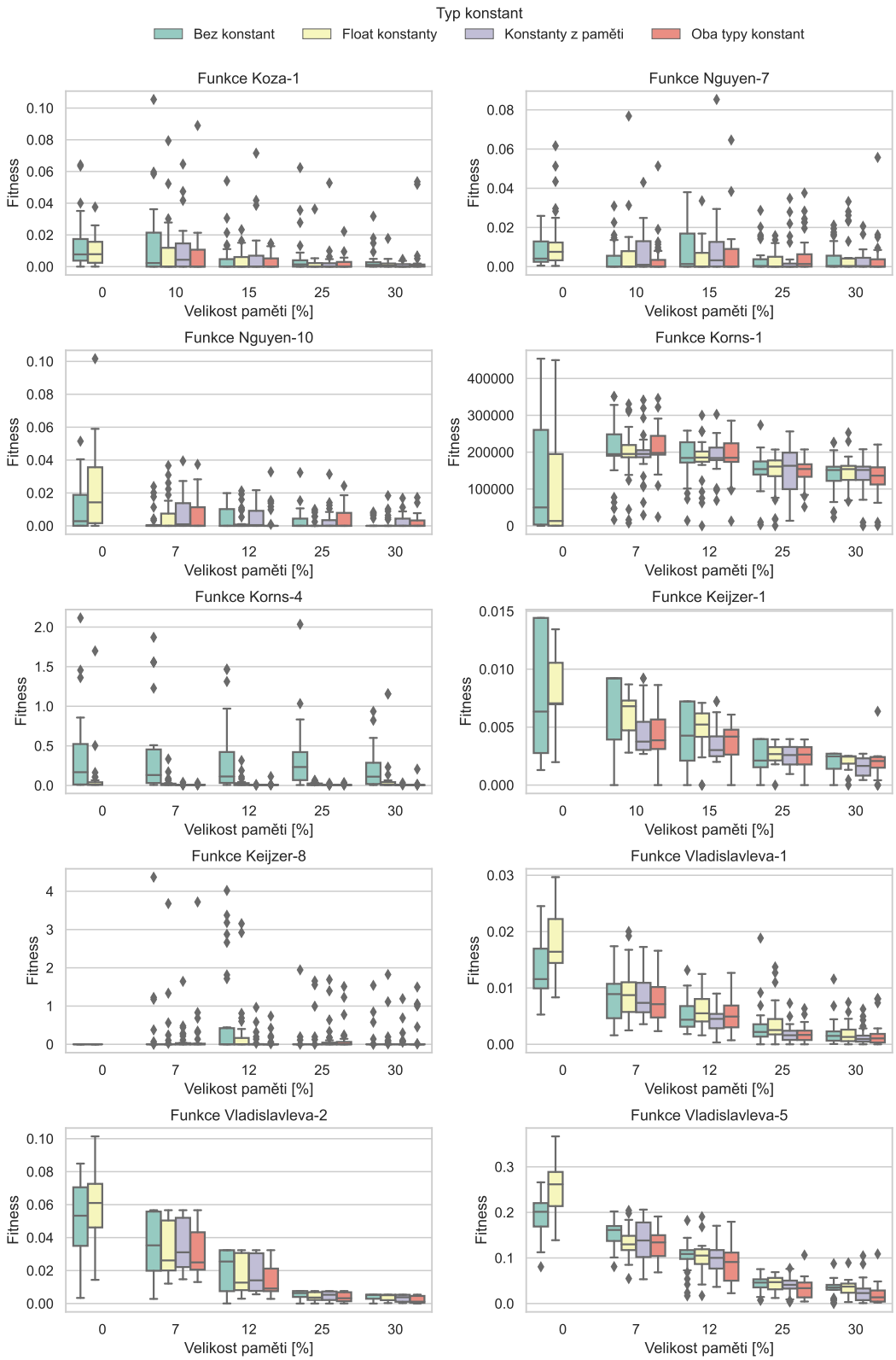
Příloha C

Rozšíření – Využití hodnot

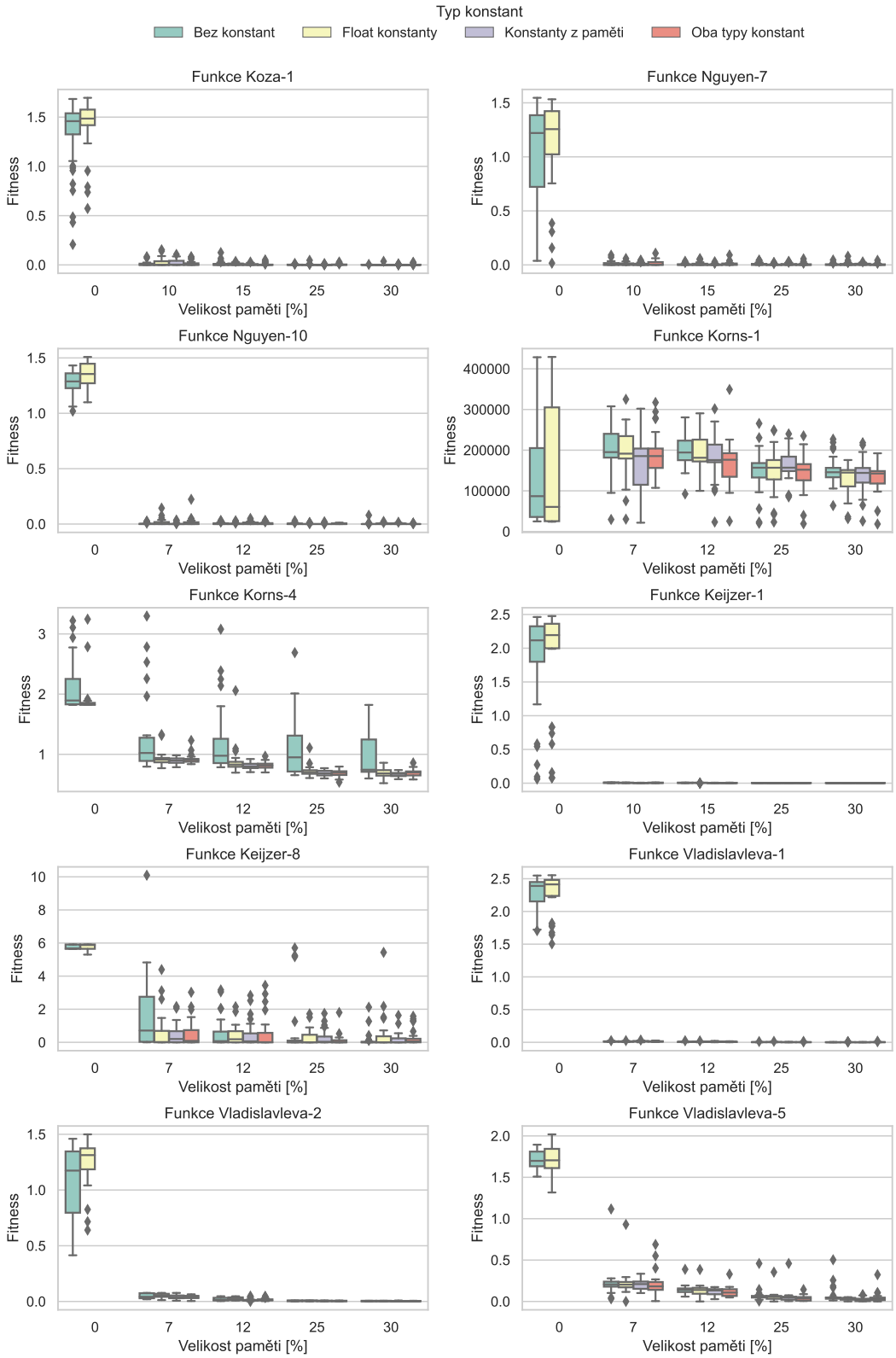
Tato příloha obsahuje další grafy vynášející výsledky experimentů popsaných v sekce 6.2. Experimenty prováděné v této sekci byly provedeny na testovacích sadách vycházejících z matematických funkcí s různým počtem přidaných náhodných hodnot. V sekci 6.2 je uveden graf kdy 10 % z hodnot v sadách bylo náhodných. V této příloze jsou pak grafy pro:

- 0 % náhodných hodnot, na obrázku C.1,
- 5 % náhodných hodnot, na obrázku C.2,
- 20 % náhodných hodnot, na obrázku C.3 a
- 25 % náhodných hodnot, na obrázku C.4.

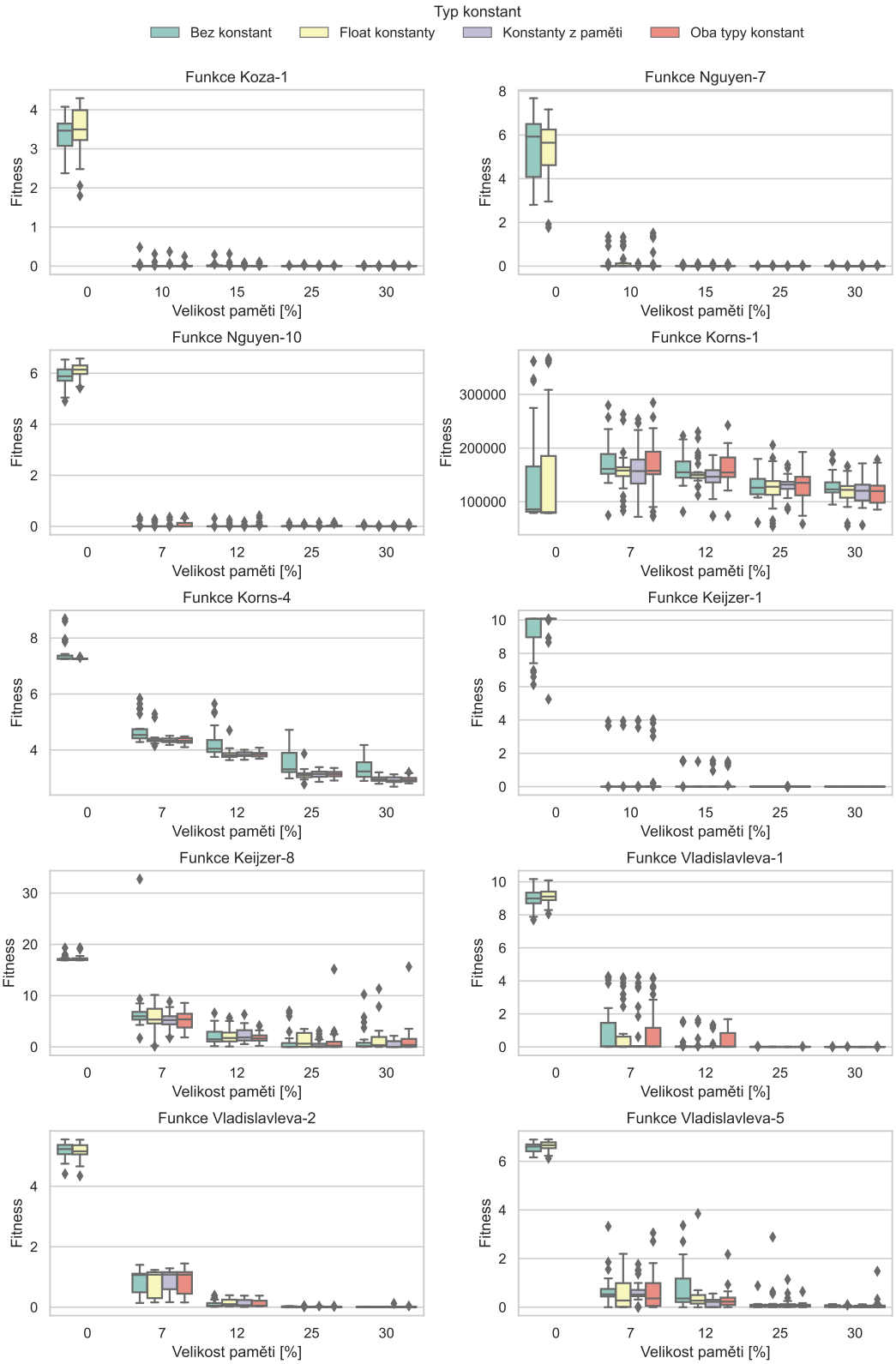
Popis grafů a výsledky experimentů jsou podrobně popsány v sekci 6.2.



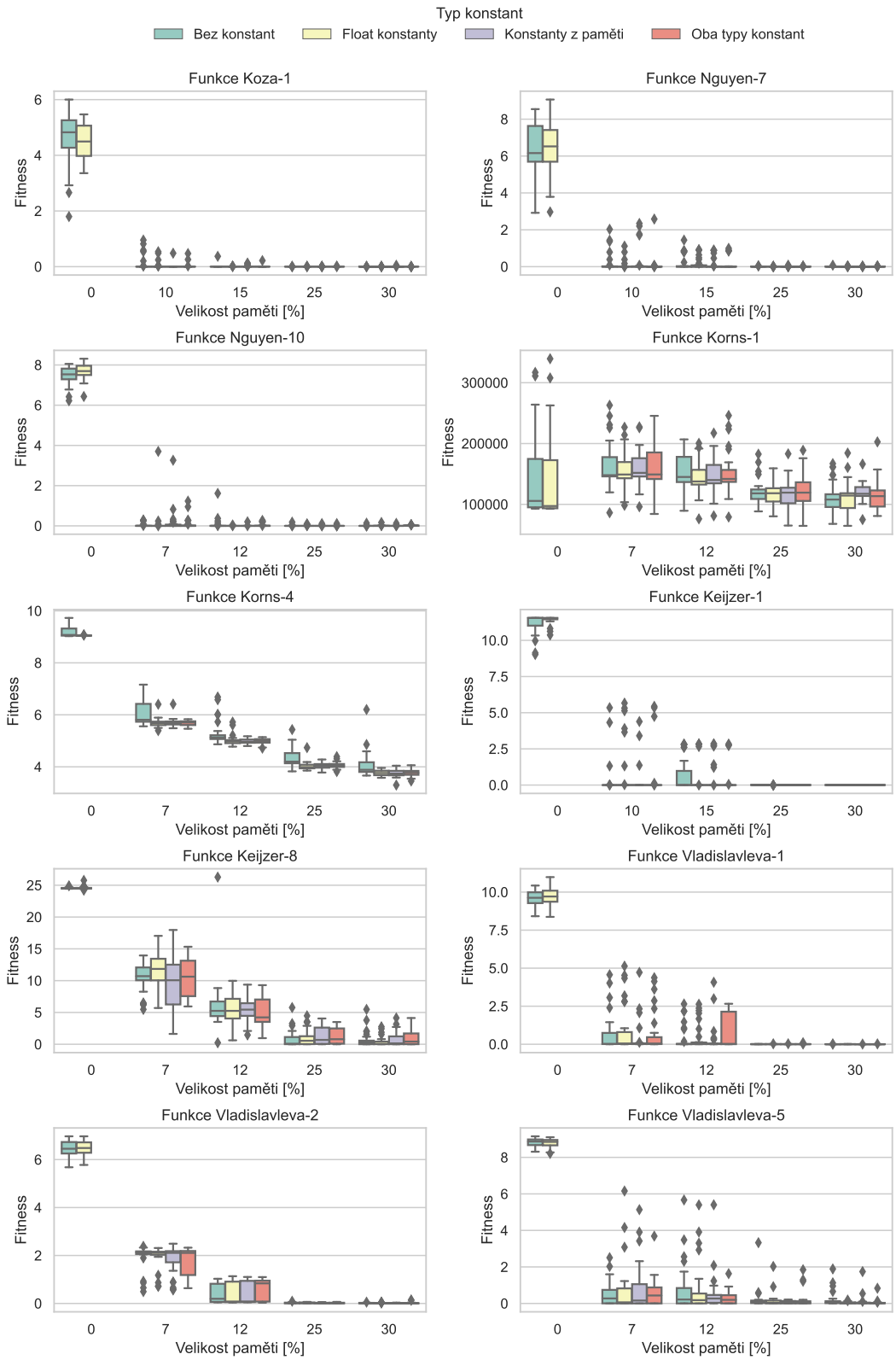
Obrázek C.1: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi s 0 % náhodných hodnot, při využití různých druhů zdroje konstant ve funkci.



Obrázek C.2: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi s 5 % náhodných hodnot, při využití různých druhů zdroje konstant ve funkci.



Obrázek C.3: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi s 20 % náhodných hodnot, při využití různých druhů zdroje konstant ve funkci.



Obrázek C.4: Dosažené fitness (s cílem minimalizovat) nad jednotlivými testovacími funkcemi s 25 % náhodných hodnot, při využití různých druhů zdroje konstant ve funkci.

Příloha D

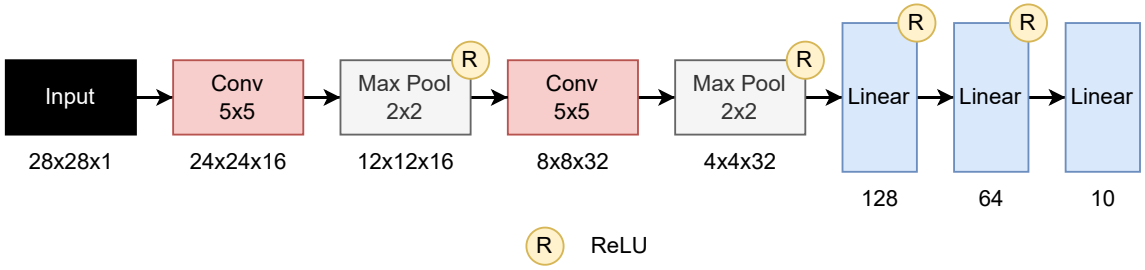
Další sady z konvolučních vrstev

Kromě datových sad generovaných z konvolučních vrstev uvedených v sekci 6.7 byly testovány další datové sady, na kterých se bohužel nepodařilo dosáhnout zajímavých výsledků a proto jsou pouze zde v příloze.

D.1 Kuzushiji-MNIST

Obdobně jako byly vytvořeny datové sady S3 a S4 ze sítě, která řeší klasifikaci na podobném datasetu jako je MNIST [22] (pro sady S3 a S4 se jednalo konkrétně o Fashion-MNIST [36]), jsou v této příloze popsány dvě neuronové sítě řešící klasifikaci na datasetu Kuzushiji-MNIST (KMNIST) [3] který je opět, jak již název napovídá, podobný datové sadě MNIST. KMNIST dataset obsahuje stejné množství obrázků jako MNIST o stejných paramentrech. Hlavním rozdílem je pouze to, že tyto obrázky nezobrazují jednotlivé číslice, ale jeden charakter pro každý z 10 řádků Hiragana[1], opět se tedy jedná o klasifikaci do 10 tříd.

Klasifikace na této datové sadě je řešena pomocí neuronové sítě, kterou lze vidět na obrázku D.1. Trénování této sítě proběhlo s využitím optimalizátoru Adam s $lr_{oroginal} = 0.01$, který byl následně měněn podle rovnice 6.6 pro 20 epoch běhu trénování.



Obrázek D.1: Schéma první neuronové sítě, řešící klasifikaci na datová sadě KMNIST [3], z jejichž vah jsou vytvořeny další sady pro otestování genetického programování s pamětí.

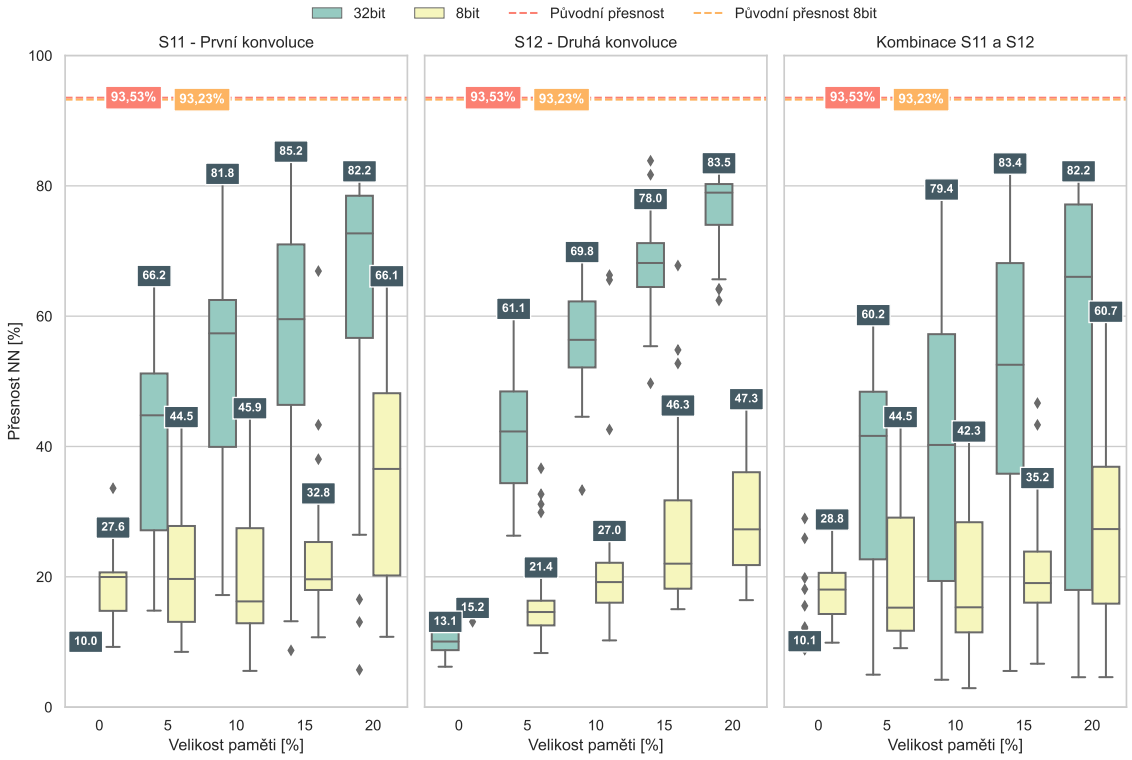
Z této sítě byly následně vytvořeny následující datové sady:

S11: sada obsahuje 400 vah z první konvoluční vrstvy.

S12: sada obsahuje 12 800 vah z druhé konvoluční vrstvy.

[1] Hiragana je jedním z japonských slabičných písem, které obsahuje 46 symbolů.

Protože je síť i velikostí podobná jako dříve testované, je opět použito nastavení genetického programování s pamětí z tabulky 6.4.



Obrázek D.2: Dosažené přesnosti první neuronové sítě na testovacích datech datasetu KMNIST [3].

Na obrázku D.2 můžeme vidět výsledky této sítě, nejlepší výsledek je u sady S11 při paměti pro 15 % vah, 32bitové sady operací a 32bitových vahách sítě, kdy je ztráta přesnosti 8,3 %. Ztráta při zmenšení vah na 8bitů a změně sady operací pak v nejlepším případě činí 27,13 % znovu u datové sady S11, ale tentokrát při velikosti paměti 20 %.

Přesnosti sítí u nichž byly genetickým programováním s pamětí vygenerovány váhy jsou nedostatečné. To může být způsobeno několika aspekty, například složitostí řešené klasifikace, nevhodností architektury sítě atd.

D.2 Kuzushiji-MNIST 2

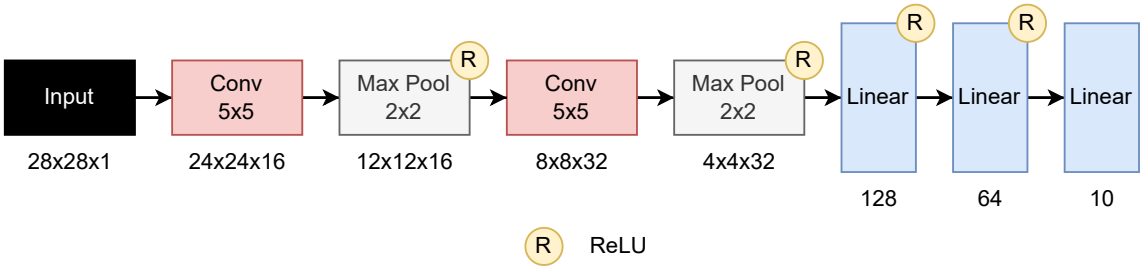
Protože výsledky předchozí sítě nejsou příliš dobré, byla vytvořena ještě další varianta sítě s architekturou na obrázku D.3, která je ještě více podobná síti pro klasifikaci na datové sadě MNIST, na které generování vah funguje.

Tato síť byla trénována identickým způsobem jako síť předchozí.

Z této sítě byly následně vytvořeny následující sady:

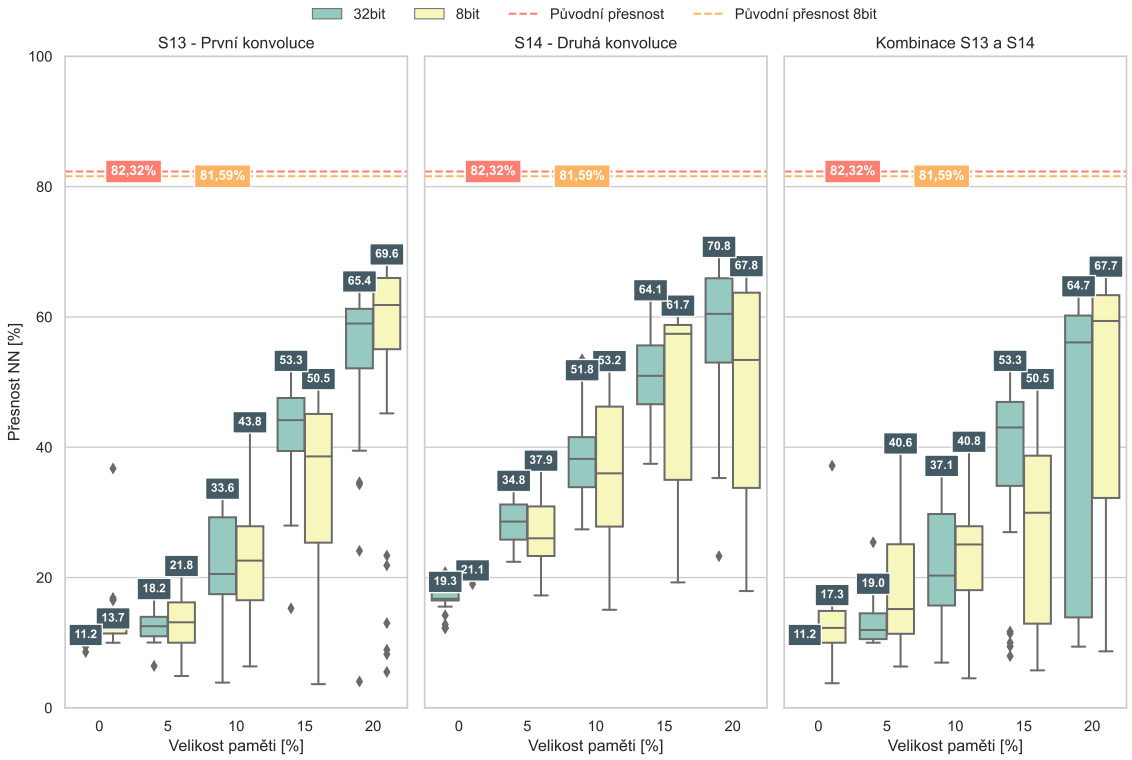
S13: sada obsahuje 250 vah z první konvoluční vrstvy.

S14: sada obsahuje 5 000 vah z druhé konvoluční vrstvy.



Obrázek D.3: Schéma druhé neuronové sítě, řešící klasifikaci na datová sadě KMNIST [3], z jejichž vah jsou vytvořeny další sady pro testování.

Protože je neuronová síť stejná, a tedy i velikost datových sad je stejná jako dříve testovaná síť, je opět použito nastavení genetického programování s pamětí z tabulky 6.4.



Obrázek D.4: Dosažené přesnosti druhé neuronové sítě na testovacích datech datasetu KMNIST [3].

Na grafech na obrázku D.4 můžeme vidět, že oproti předchozí síti řešící klasifikaci na KMNIST došlo ke zhoršení původní přesnosti natrénované sítě a to o 11,21 %, což je způsobeno zjednodušením sítě. Bohužel přesnosti sítí s vygenerovanými váhami dosahují stále nedostatečných výsledků, nejlepšího výsledku bylo dosaženo při použití sady 32bitových operací a generování 32bitových vah pro sadu S14, kdy při velikosti paměti 20 % bylo dosaženo ztráty 11,5 %. Jedná se o větší ztrátu, než u nejlepšího řešení v minulém experimentu.

Pozitivní na těchto experimentech je podobnost dosažených výsledků při použití sad operací pracující s 32/8bitovými hodnotami a tomu odpovídající velikosti vah neuronových

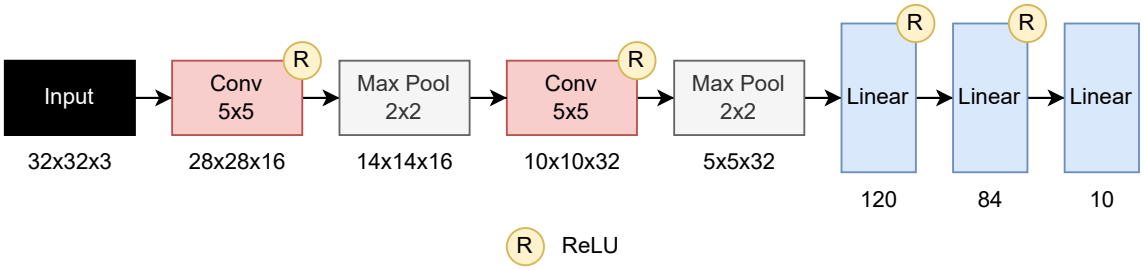
sítí. Kdy nejlepší výsledek pro 8bitovou sadu operací je u sady S13 při velikosti paměti 20 %, kde ztráta přesnosti činí 11,99 %. Toto je více jak dvakrát lepší ztráta, než u nejlepšího řešení pro 8bitovou sadu operací u předchozího experimentu.

Celkové výsledky jsou nedostatečné a i samotná síť dosahuje pouze malé přesnosti na datové sadě, přičemž při generování vah jsou výsledky ještě výrazně horší.

D.3 CIFAR-10

Obdobně jako byly vytvořeny datové sady ze sítě, která řeší klasifikaci na datasetu MNIST [22], a následně ze sítí řešících podobné klasifikační problémy, jako o něco výrazněji složitější problém pro řešení byla zvolena klasifikace pro datovou sadu CIFAR-10 [18]. Na této sadě již byla testována možnost generovat váhy pro síť MobileNetV2 popsaná v 6.7.2. Zde byla otestavána výrazně jednodušší síť, schéma architektury této sítě lze vidět na obrázku D.1.

Trénování této sítě proběhlo s využitím optimalizátoru SGD s rychlostí učení 0,01 a setrvačností 0,9 při 10 epochách běhu trénování. Vzhledem k přesnosti výsledné sítě 69,3 % je dobré uvést, že byly testovány i jiné nastavení trénování sítě, ale žádné z testovaných nedosahovalo lepších výsledků.



Obrázek D.5: Schéma neuronové sítě, řešící klasifikaci na datové sadě CIFAR-10 [18], z jejichž vah jsou vytvořeny další sady pro testování.

Z této sítě byly následně vytvořeny následující sady:

S15: sada obsahuje 1 200 vah z první konvoluční vrstvy.

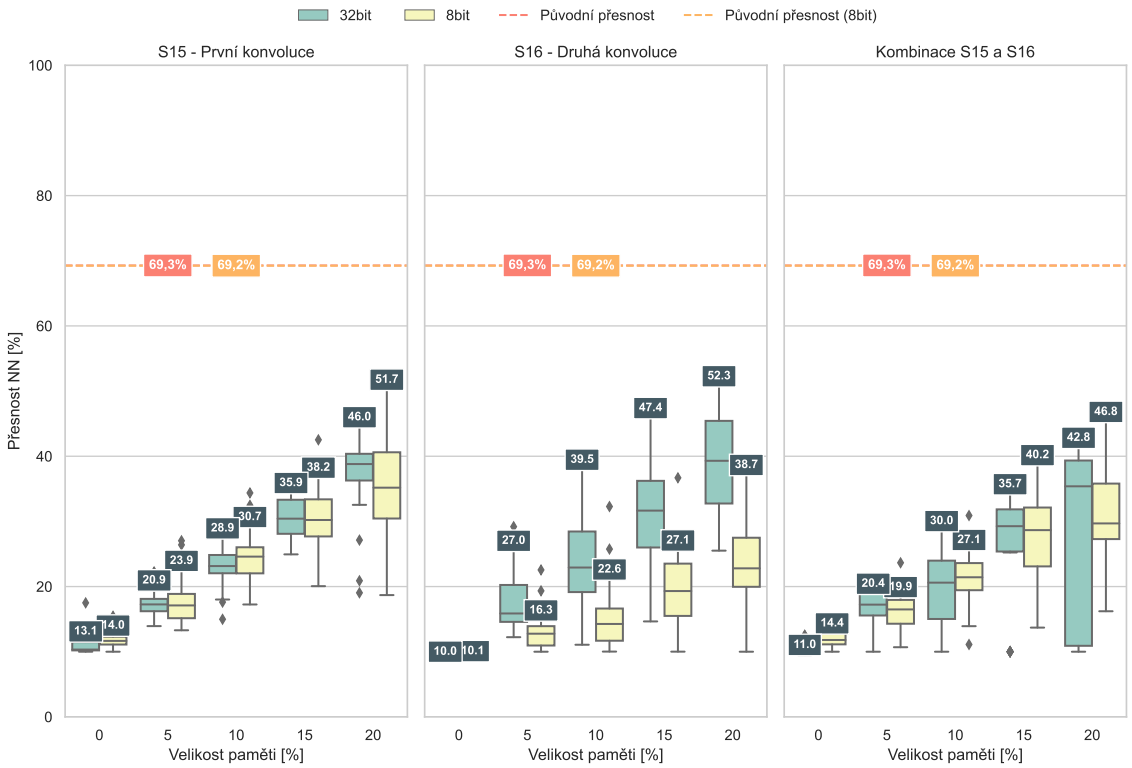
S16: sada obsahuje 12 800 vah z druhé konvoluční vrstvy.

I zde je použito nastavení genetického programování s pamětí z tabulky 6.4.

Na obrázku D.6 jsou uvedeny grafy zobrazující přesnosti neuronových sítí. I původní přesnost sítě je nedostatečná a o proti dříve na tomto datasetu testované síti MobileNetV2 zde dochází ke ztrátě 24,61 %, tato ztráta je převážně způsobena výrazně jednodušší sítí.

Výsledky při generovaných vahách jsou nedostatečné a i přes nízkou původní přesnost sítě dochází k velké ztrátě přesnosti, která je při použití 32bitových vah sítě a operací, které pracují s takto velkými hodnotami, nejmenší u datové sady S16 a to konkrétně 17 % při velikosti paměti 20 %. Při sadě operací pro 8bitové hodnoty a korespondující velikosti vah sítě je nejmenší ztráty dosaženo u S15 při velikosti paměti 20 %, kde je ztráta 17,5 %.

Celkové výsledky jsou nedostatečné a i samotná síť dosahuje pouze malé přesnosti na datové sadě, přičemž při generování vah jsou výsledky ještě výrazně horší.



Obrázek D.6: Dosažené přesnosti jednoduché neuronové sítě na testovacích datech datasetu CIFAR-10 [18].

D.4 Závěr

V této příloze byly popsány další tři neuronové sítě, ze kterých bylo dohromady vytvořeno dalších 6 testovacích sad. Výsledky na těchto sadách nejsou nijak výrazně zajímavé a dochází zde spíše ke nedostatečným výsledkům, to je také důvod proč jsou tyto experimenty uvedeny v příloze a ne v hlavním textu práce.